Chapter 8: Combinational Logic Modules

Prof. Soo-Ik Chae

Objectives

After completing this chapter, you will be able to:

- ❖ Understand the features of decoders
- ❖ Understand the features of encoders
- ❖ Understand the features of priority encoders
- ❖ Understand the features of multiplexers
- ❖ Understand the features of demultiplexers
- ❖ Describe how to design comparators and magnitude comparators
- ❖ Describe how to design a parameterized module

Basic Combinational Logic Modules

- Commonly used combinational logic modules:
  - Decoder
  - Encoder
  - Multiplexer
  - Demultiplexer
  - Comparator
  - Adder (CLA)
  - Subtracter (subtractor)
  - Multiplier
  - PLA
  - Parity Generator

-

Options for Modeling Combinational Logic

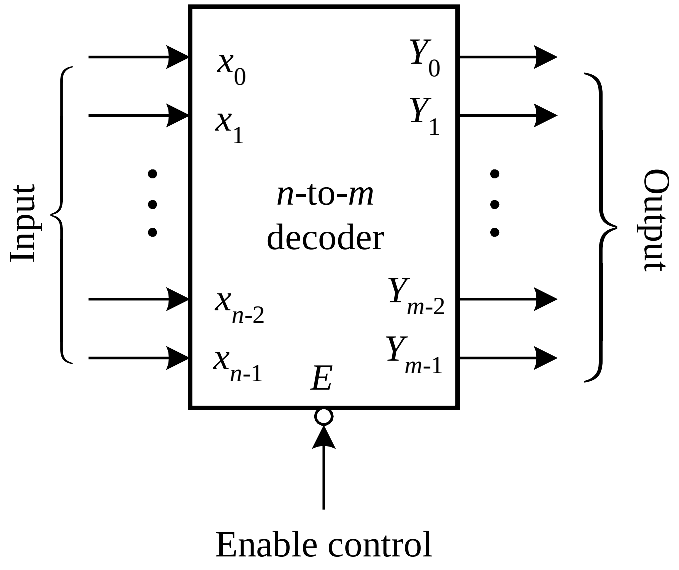
- Options for modeling combinational logic:
  - Verilog HDL primitives
  - Continuous assignment
  - Behavioral statement
  - Function
  - Task without delay or event control
  - Combinational UDP
  - Interconnected combinational logic modules

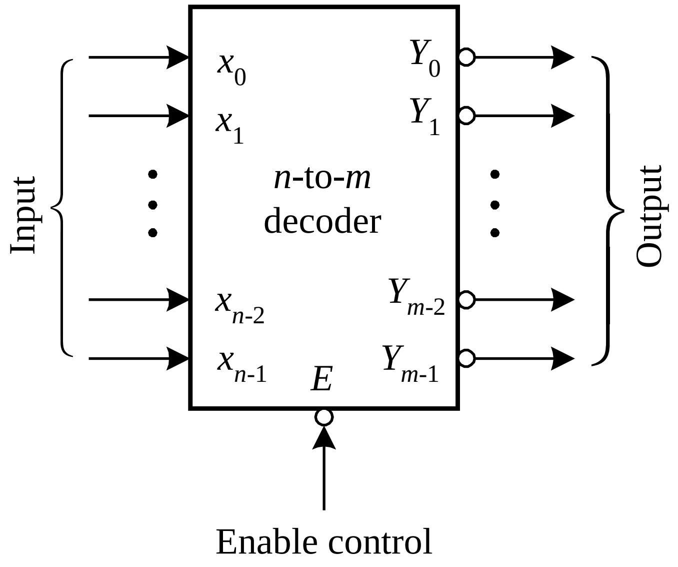
Three Descriptions of Combination Logic

Logic Description		Verilog Description
Circuit Schematic	$\longleftrightarrow$	Structural Model
Truth Table	$\longleftrightarrow$	User-Defined Primitives
Switching Equations	$\longleftrightarrow$	Continuous Assignments

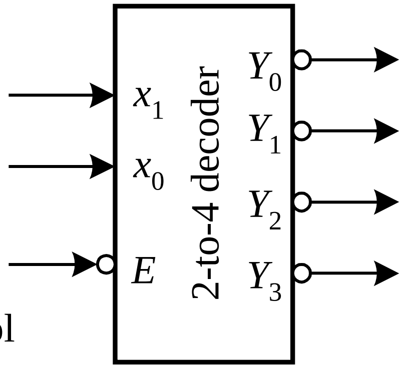
Decoder Block Diagrams

- An $n \times m$ decoder has n input lines and m output lines. Each output line Y_i corresponds to the ith minterm of input (line) variables.
  - Total decoding: when $m = 2^n$.
  - Partial decoding: when $m < 2^n$.





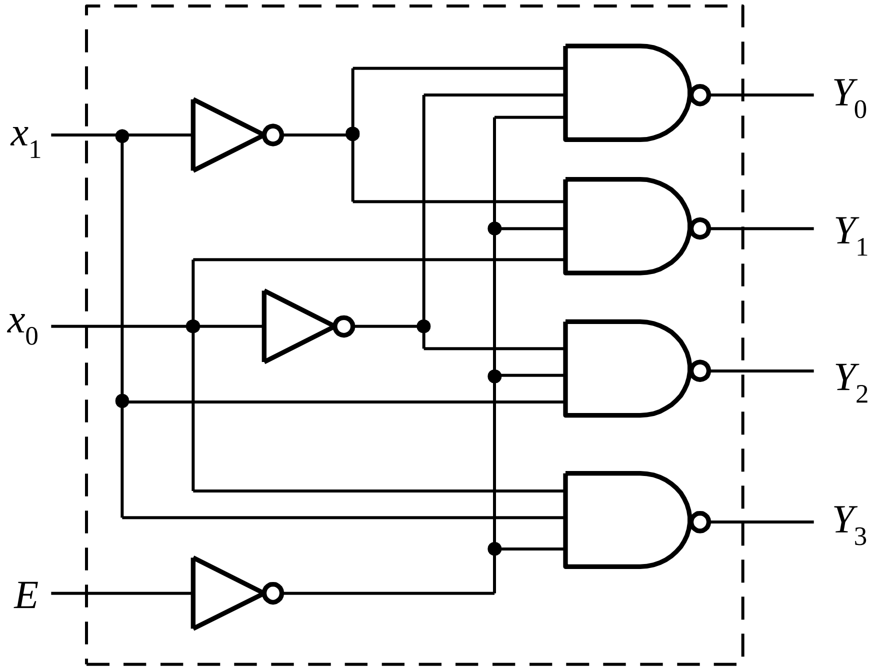
A 2-to-4 Decoder Example



(a) Logic symbol

E	x_1	x_0	Y_3	Y_2	Y_1	Y_0
1	ϕ	ϕ	1	1	1	1
0	0	0	1	1	1	0
0	0	1	1	1	0	1
0	1	0	1	0	1	1
0	1	1	0	1	1	1

(b) Function table

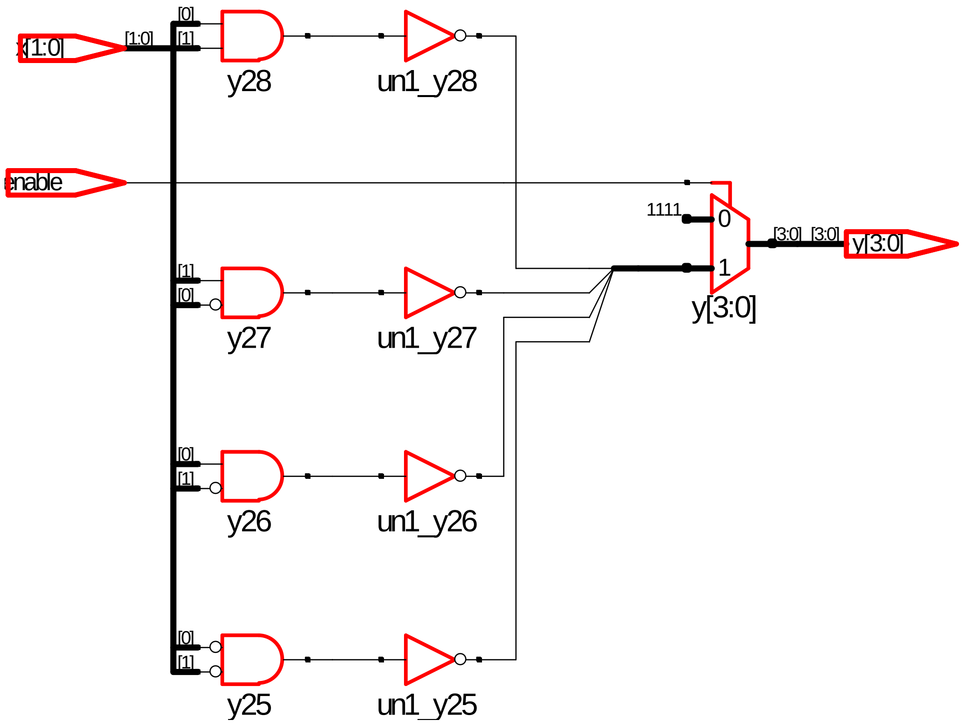


(c) Logic circuit

ϕ : don't care

A 2-to-4 Decoder Example

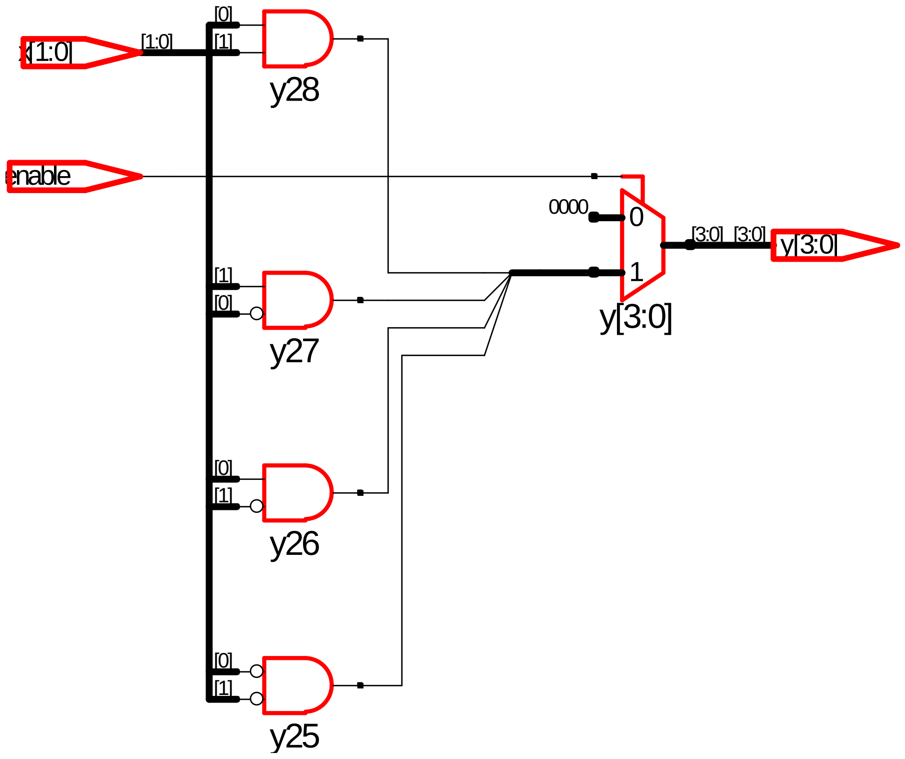
```
// a 2-to-4 decoder with active low output
module decoder_2to4_low(x,enable,y);
input  [1:0] x;
input  enable;
output reg [3:0] y;
// the body of the 2-to-4 decoder
always @(x or enable)
    if (!enable) y = 4'b1111; else
        case (x)
            2'b00 : y = 4'b1110;
            2'b01 : y = 4'b1101;
            2'b10 : y = 4'b1011;
            2'b11 : y = 4'b0111;
          default : y = 4'b1111;
        endcase
endmodule
```



A 2-to-4 Decoder with Enable Control

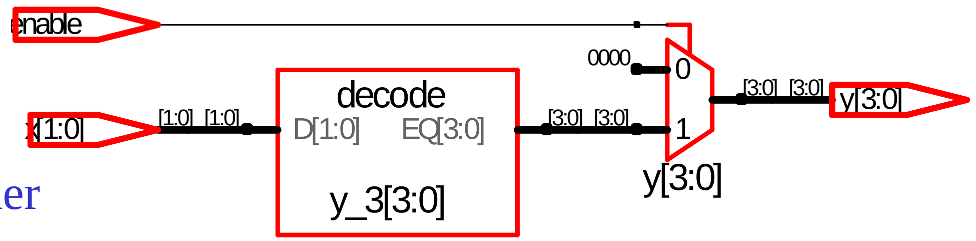
```
// a 2-to-4 decoder with active-high output
module decoder_2to4_high(x,enable,y);
input  [1:0] x;
input  enable;
output reg [3:0] y;

// the body of the 2-to-4 decoder
always @(x or enable)
   if (!enable) y = 4'b0000; else
       case (x)
            2'b00 : y = 4'b0001;
            2'b01 : y = 4'b0010;
            2'b10 : y = 4'b0100;
            2'b11 : y = 4'b1000;
          default : y = 4'b0000;
       endcase
endmodule
```



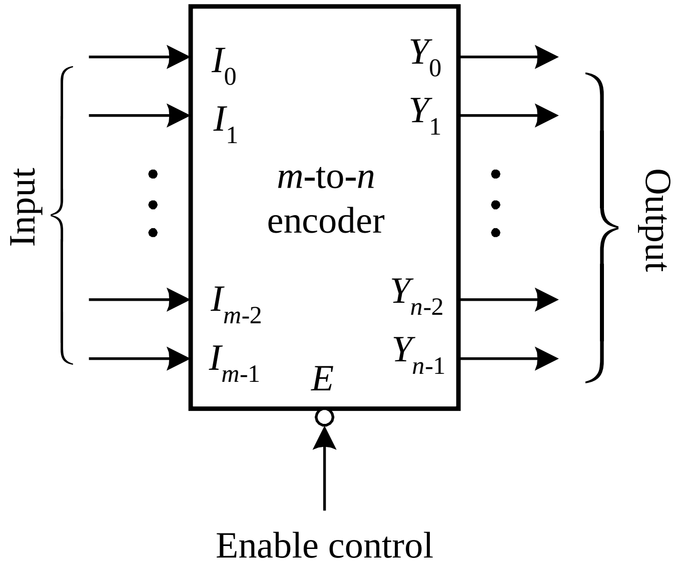
A Parameterized Decoder Module

```
// an m-to-n decoder with active-high output
module decoder_m2n_high(x,enable,y);
parameter  m = 3;  // define the number of input lines
parameter   n = 8;  // define the number of output lines
input  [m-1:0] x;
input  enable;
output reg [n-1:0] y;
// The body of the m-to-n decoder
always @(x or enable)
   if   (!enable) y = {n{1'b0}};
   else               y =  {{n-1{1'b0}},1'b1}} << x;
endmodule
```

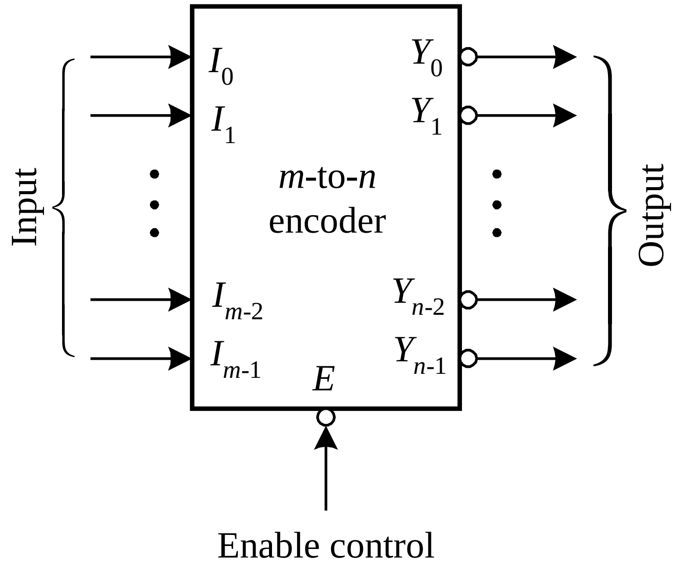


Encoder Block Diagrams

- An encoder has $m = 2^n$ (or fewer) input lines and n output lines. The output lines generate the binary code corresponding to the input value.



(a) Noninverted output

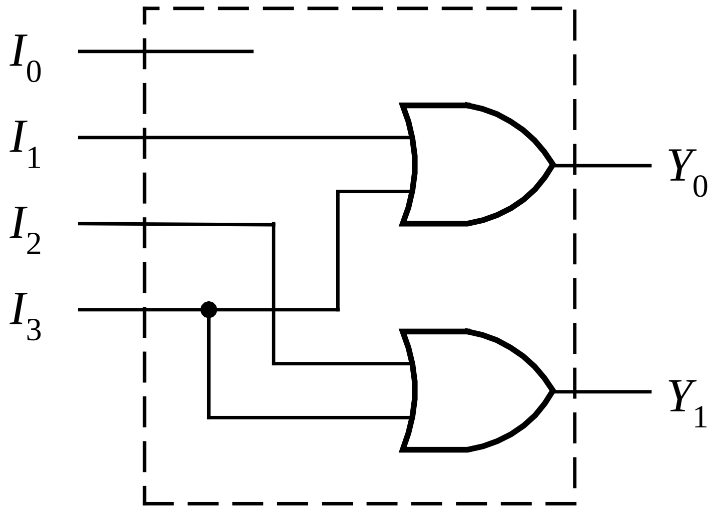


(b) Inverted output

A 4-to-2 Encoder Example

I_3	I_2	I_1	I_0	Y_1	Y_0
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

(a) Function table



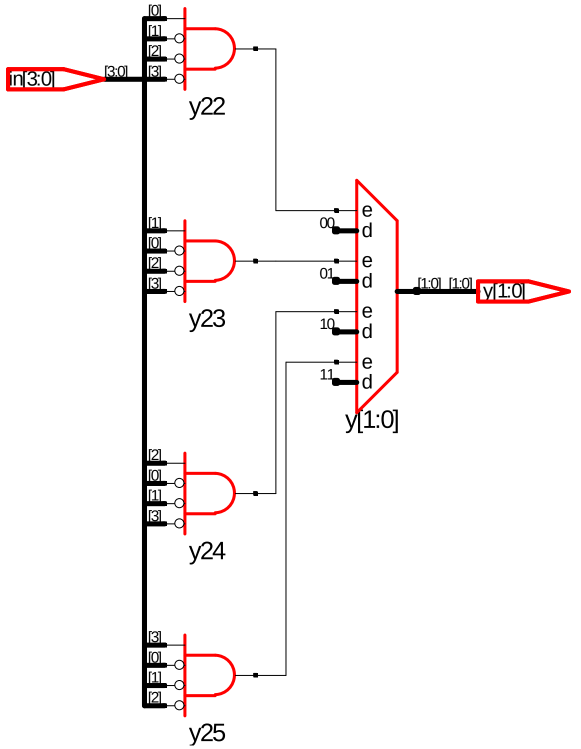
(b) Logic circuit

Q: What is the problem of this encoder?

There are many undefined input cases!

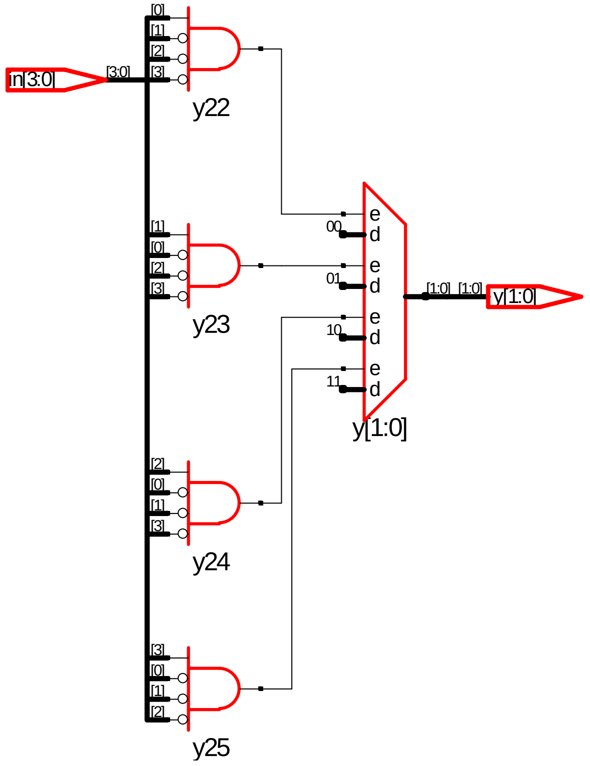
A 4-to-2 Encoder Example

```
// a 4-to-2 encoder using if ... else structure
module encoder_4to2_ifelse(in, y);
input  [3:0] in;
output reg [1:0] y;
// the body of the 4-to-2 enecoder
always @(in) begin
   if (in == 4'b0001) y = 0; else
   if (in == 4'b0010) y = 1; else
   if (in == 4'b0100) y = 2; else
   if (in == 4'b1000) y = 3; else
       y = 2'bx;
end
endmodule
```

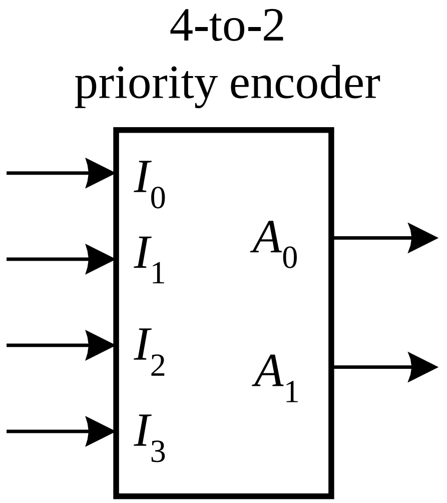


Another 4-to-2 Encoder Example

```
// a 4-to-2 encoder using case structure
module encoder_4to2_case(in, y);
input   [3:0] in;
output reg [1:0] y;
// the body of the 4-to-2 enecoder
always @(in)
   case (in)
      4'b0001 : y = 0;
      4'b0010 : y = 1;
      4'b0100 : y = 2;
      4'b1000 : y = 3;
      default : y = 2'bx;
   endcase
endmodule
```



A 4-to-2 Priority Encoder



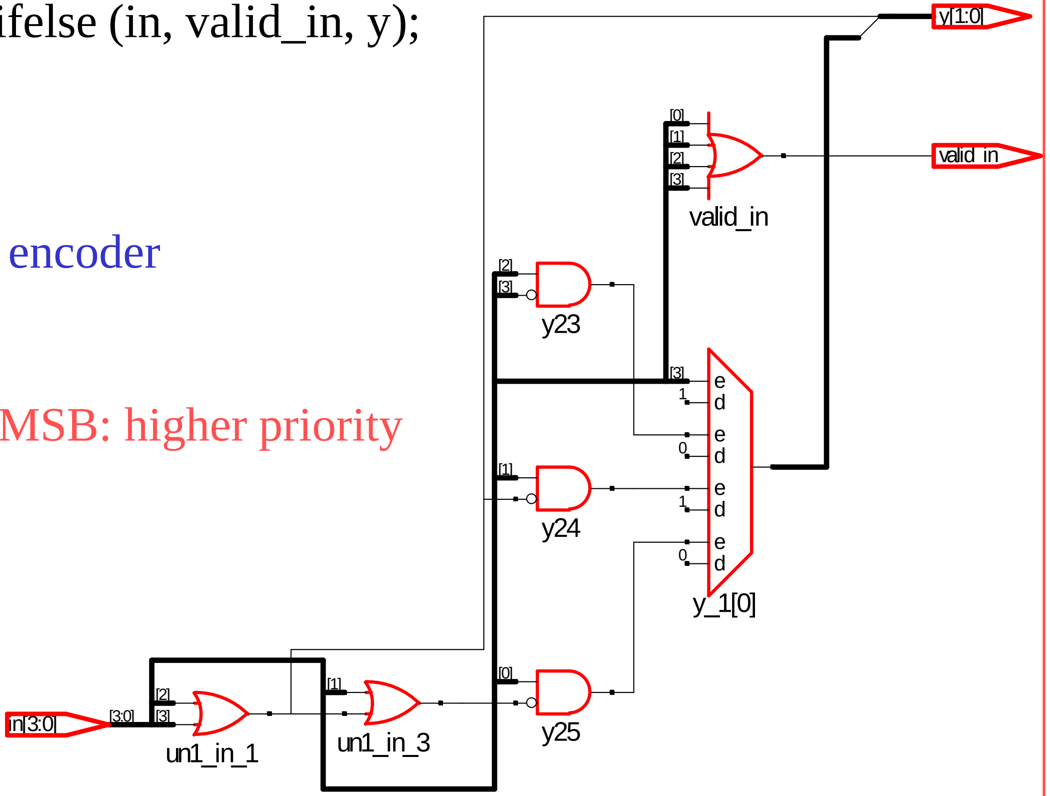
(a) Block diagram

Input				Output	
I_3	I_2	I_1	I_0	A_1	A_0
0	0	0	1	0	0
0	0	1	ϕ	0	1
0	1	ϕ	ϕ	1	0
1	ϕ	ϕ	ϕ	1	1

(b) Function table

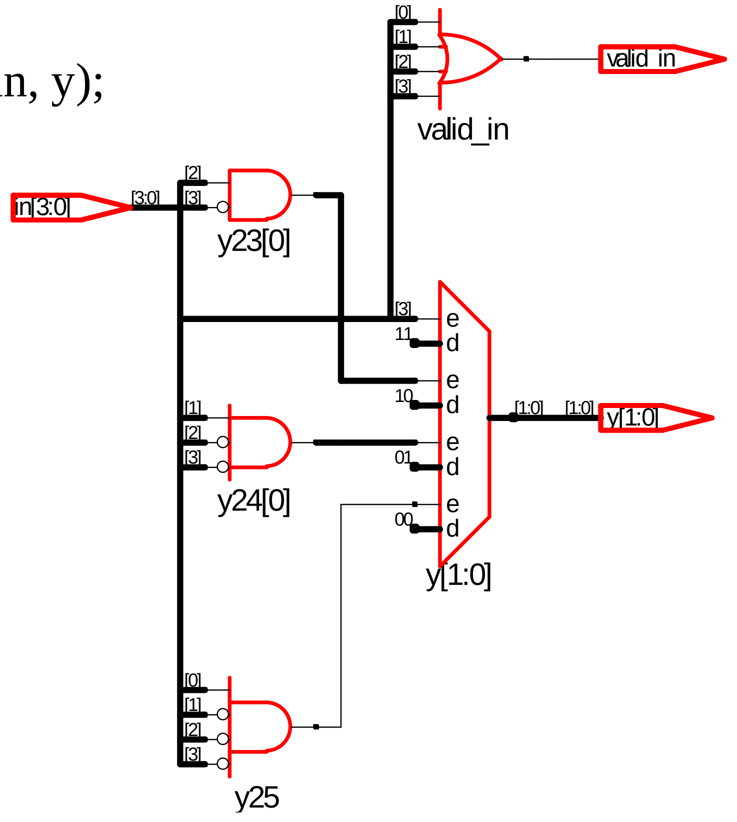
A 4-to-2 Priority Encoder Example

```
// a 4-to-2 priority encoder using if ... else structure
module priority_encoder_4to2_ifelse (in, valid_in, y);
input  [3:0] in;
output reg [1:0] y;
output valid_in;
// the body of the 4-to-2 priority encoder
assign valid_in = |in;
always @(in) begin
   if (in[3]) y = 3; else          // MSB: higher priority
   if (in[2]) y = 2; else
   if (in[1]) y = 1; else
   if (in[0]) y = 0; else
   y = 2'bx;
end
endmodule
```



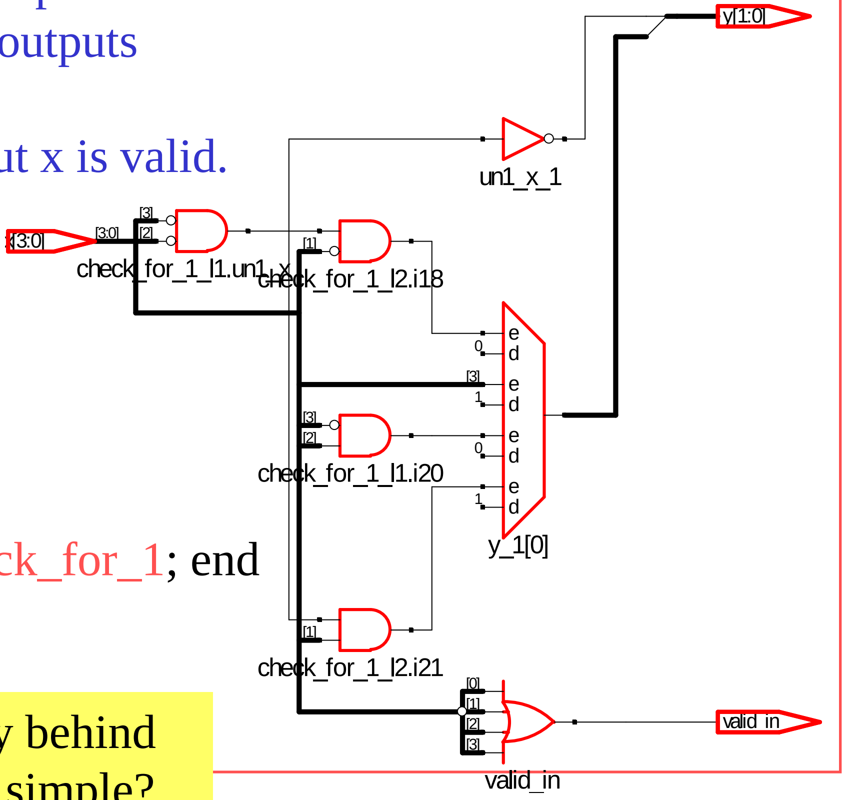
Another 4-to-2 Priority Encoder Example

```
// a 4-to-2 priority encoder using case structure
module priority_encoder_4to2_case(in, valid_in, y);
input  [3:0] in;
output reg [1:0] y;
output valid_in;
// the body of the 4-to-2 priority encoder
assign valid_in = |in;
always @(in) casex (in)
   4'b1xxx: y = 3;
   4'b01xx: y = 2;
   4'b001x: y = 1;
   4'b0001: y = 0;
   default:     y = 2'bx;
endcase
endmodule
```



A Parameterized Priority Encoder Example

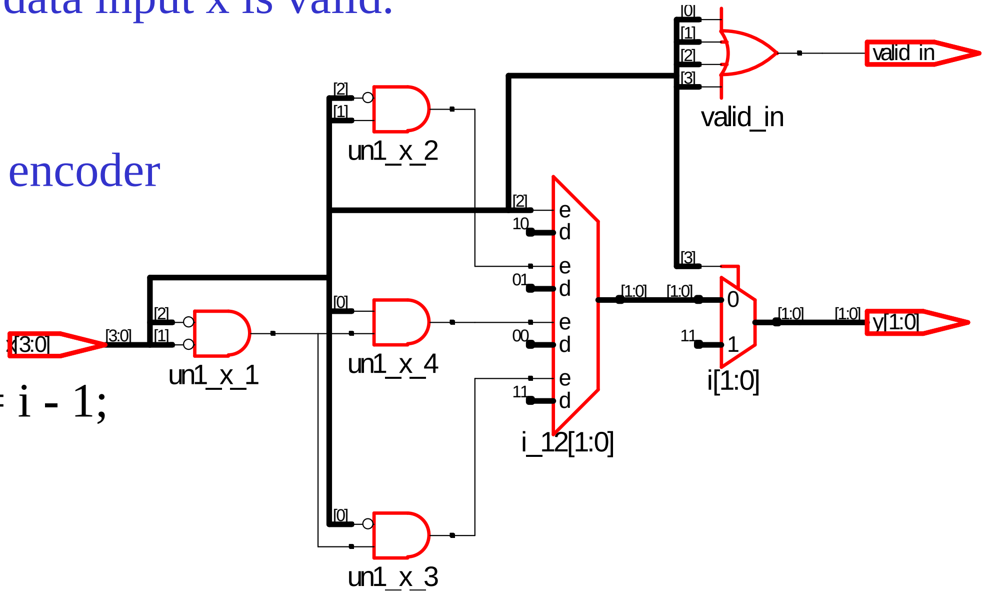
```
// an m-to-n priority encoder
module priencoder_m2n(x, valid_in, y);
parameter m = 8;  // define the number of inputs
parameter n = 3;   // define the number of outputs
input  [m-1:0] x;
output valid_in;    // indicates the data input x is valid.
output reg [n-1:0] y;
integer i;
// the body of the m-to-n priority encoder
assign valid_in = |x;
always @(*) begin: check_for_1
   for (i = m - 1 ; i > 0 ; i = i - 1)
       if  (x[i] == 1) begin y = i; disable check_for_1; end
       else y = 0;  // Why need else …?
end
endmodule
```



Q: Explain the philosophy behind this program. Why is it so simple?

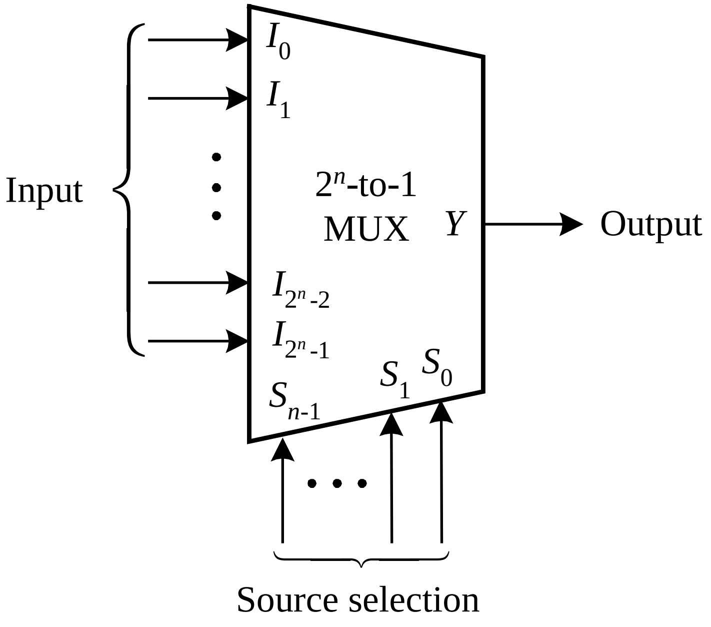
A Better Parameterized Priority Encoder Example

```
// an m-to-n priority encoder
module priencoder_m2n_while(x, valid_in, y);
parameter m = 4; // define the number of inputs
parameter n = 2;  // define the number of outputs
input  [m-1:0] x;
output valid_in;   // indicates the data input x is valid.
output reg [n-1:0] y;
integer i;
// the body of the m-to-n priority encoder
assign valid_in = |x;
always @(*) begin
   i = m - 1 ;
   while(x[i] == 0 && i >= 0 ) i = i - 1;
   y = i;
end
endmodule
```

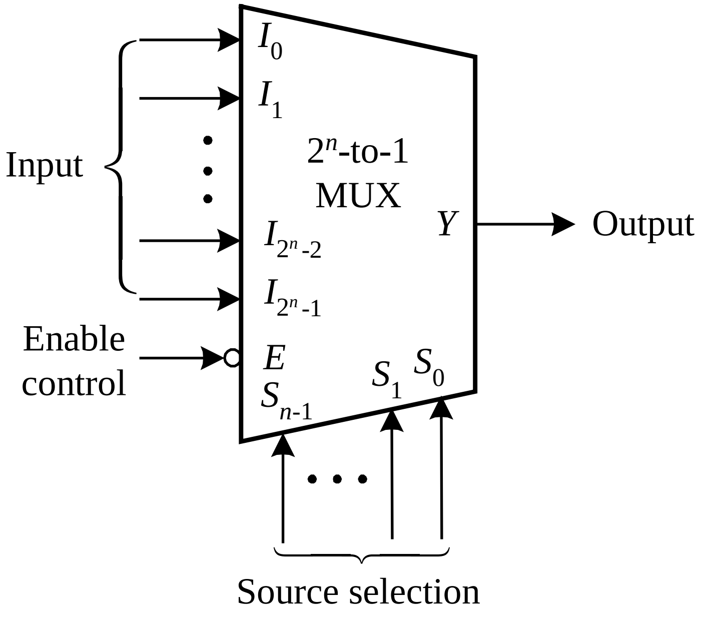


Multiplexer Block Diagrams

- An m-to-1 ($m = 2^n$) multiplexer has m input lines,1 output line, and n selection lines. The input line I_i selected by the binary combination of n source selection lines is directed to the output line, Y.



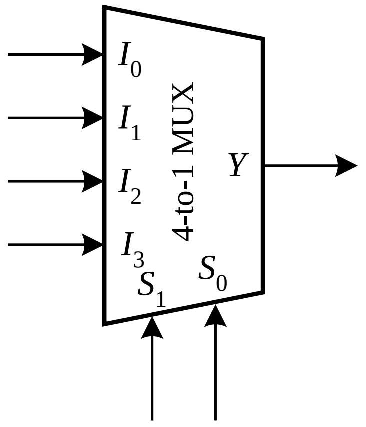
(a) Without enable control



(b) With enable control

A 4-to-1 Multiplexer Example

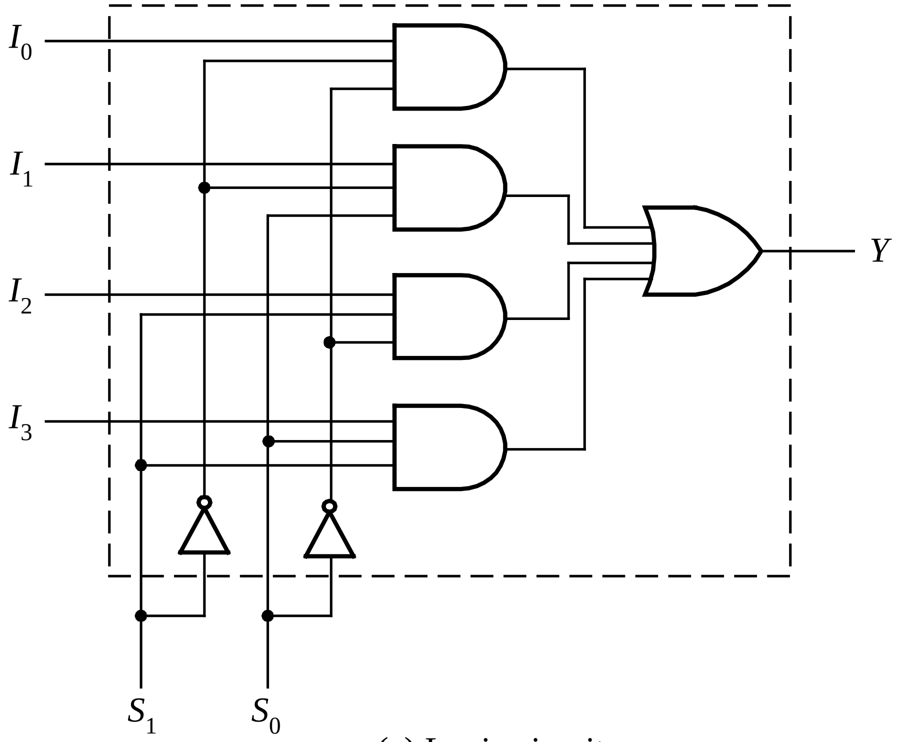
Gate-based 4-to-1 multiplexers



(a) Logic symbol

S_1	S_0	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

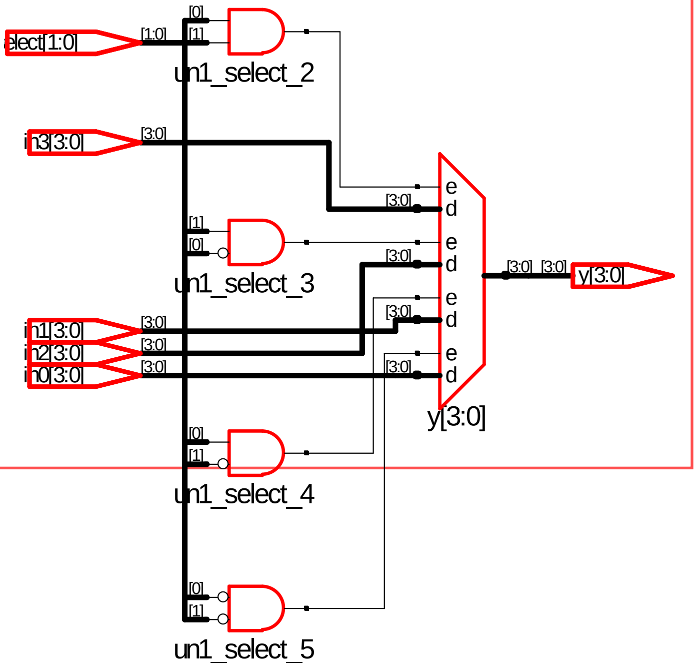
(b) Function table



(c) Logic circuit

An *n*-bit 4-to-1 Multiplexer Example

```
// an N-bit 4-to-1 multiplexer using conditional operator.
module mux_nbit_4to1(select, in3, in2, in1, in0, y);
parameter N = 4; // define the width of 4-to-1 multiplexer
input [1:0] select;
input [N-1:0] in3, in2, in1, in0;
output [N-1:0] y;
// the body of the N-bit 4-to-1 multiplexer
assign y = select[1] ?
              (select[0] ? in3 : in2) :
              (select[0] ? in1 : in0) ;
endmodule
```

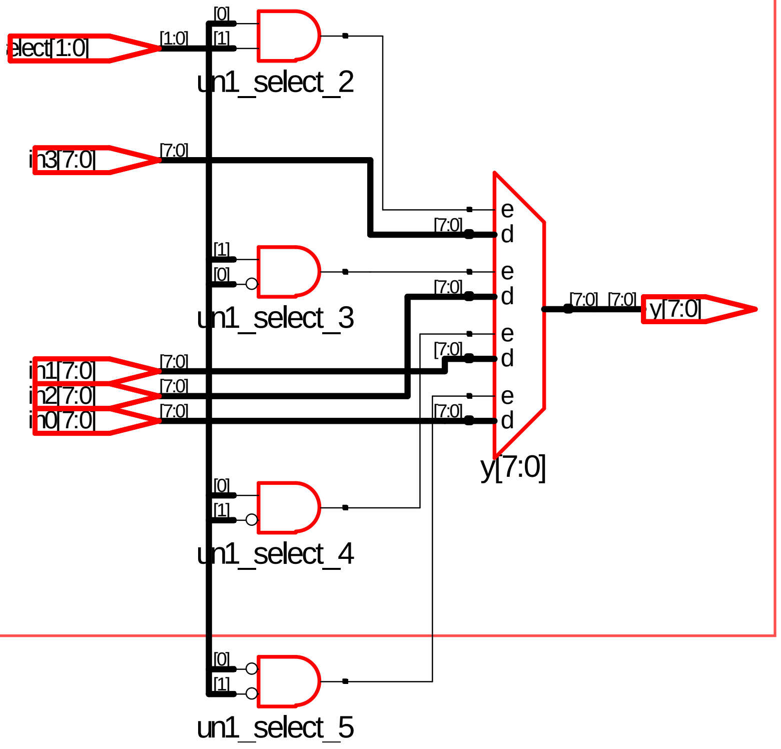


The Second *n*-bit 4-to- 1 Multiplexer Example

```
// an N-bit 4-to-1 multiplexer with enable control.
module mux_nbit_4to1_en (select, enable, in3, in2, in1, in0, y);
parameter N = 4; // define the width of 4-to-1 multiplexer
input [1:0] select;
input enable;
input [N-1:0] in3, in2, in1, in0;
output reg [N-1:0] y;
// the body of the N-bit 4-to-1 multiplexer
always @(select or enable or in0 or in1 or in2 or in3)
    if (!enable) y = {N{1'b0}};
    else y = select[1] ?
                (select[0] ? in3 : in2) :
                (select[0] ? in1 : in0) ;
endmodule
```

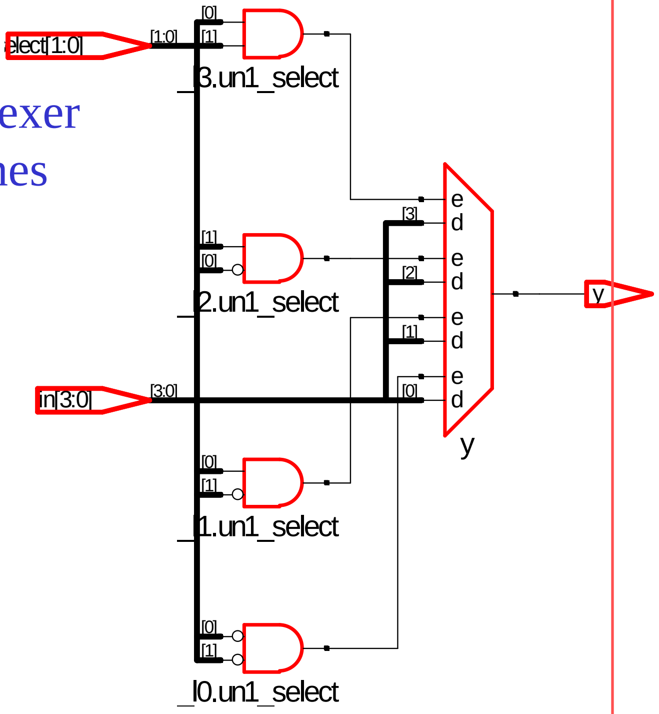
The Third *n*-bit 4-to- 1 Multiplexer Example

```
// an N-bit 4-to-1 multiplexer using case structure.
module mux_nbit_4to1_case(select, in3, in2, in1, in0, y);
parameter N = 8; // define the width of 4-to-1 multiplexer
input [1:0] select;
input [N-1:0] in3, in2, in1, in0;
output reg [N-1:0] y;
// the body of the N-bit 4-to-1 multiplexer
always @(*)
      case (select)
            2'b11: y = in3 ;
            2'b10: y = in2 ;
            2'b01: y = in1 ;
            2'b00: y = in0 ;
      endcase
endmodule
```



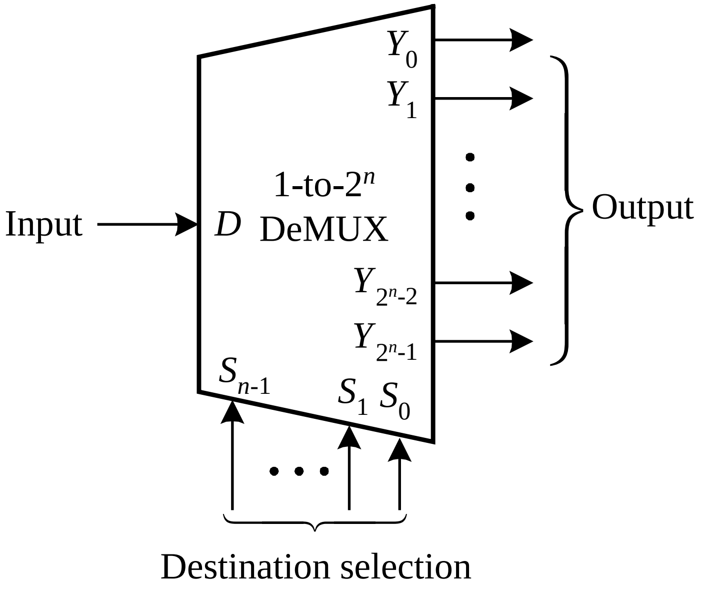
A Parameterized Multiplexer Example

```
// an example of parameterized M-to-1 multiplexer.
module mux_m_to_1(select, in, y);
parameter M = 4; // define the size of M-to-1 multiplexer
parameter K = 2; // define the number of selection lines
input [K-1:0] select;
input [M-1:0] in;
output reg y;
// the body of the M-to-1 multiplexer
integer i;
always @(*)
      for (i = 0; i < M; i = i + 1)
            if (select == i) y = in[i];
endmodule
```

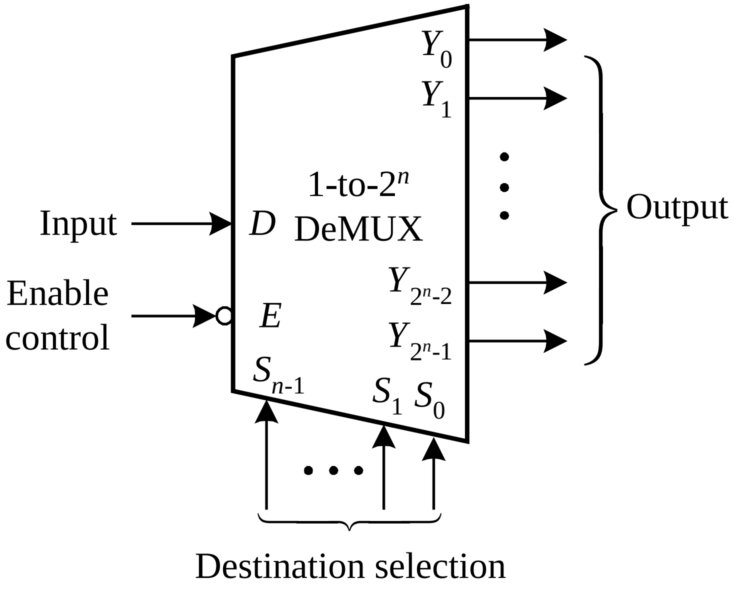


DeMultiplexer Block Diagrams

- A 1-to-m ($m = 2^n$) demultiplexer has 1 input line, m output lines, and n destination selection lines. The input line D is directed to the output line Y_i selected by the binary combination of n destination selection lines.



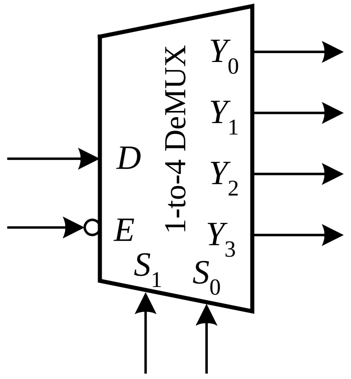
(a) Without enable control



(b) With enable control

A 1-to-4 DeMultiplexer Example

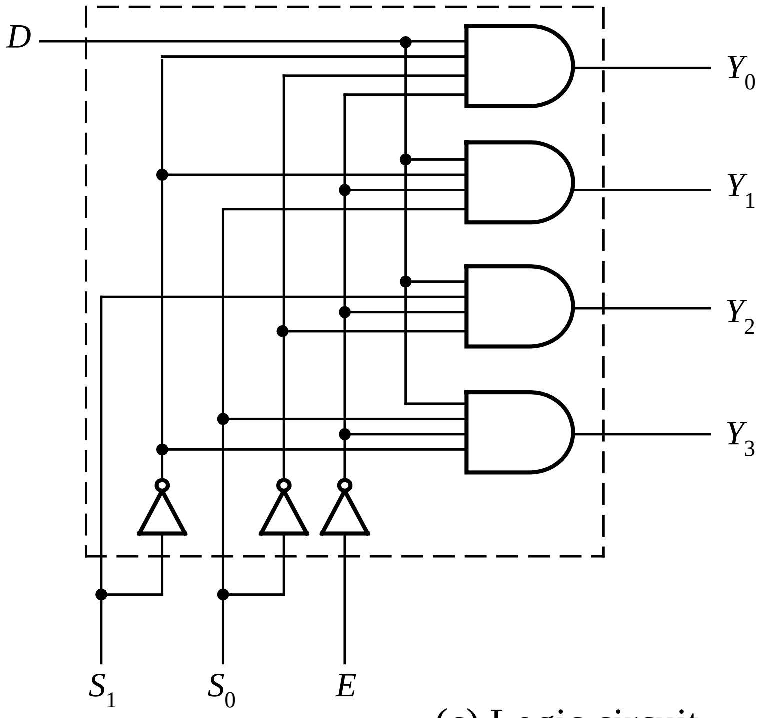
Gate-based 1-to-4 demultiplexers



(a) Logic symbol

E	S_1	S_0	Y_3	Y_2	Y_1	Y_0
1	ϕ	ϕ	0	0	0	0
0	0	0	0	0	0	D
0	0	1	0	0	D	0
0	1	0	0	D	0	0
0	1	1	D	0	0	0

(b) Function table

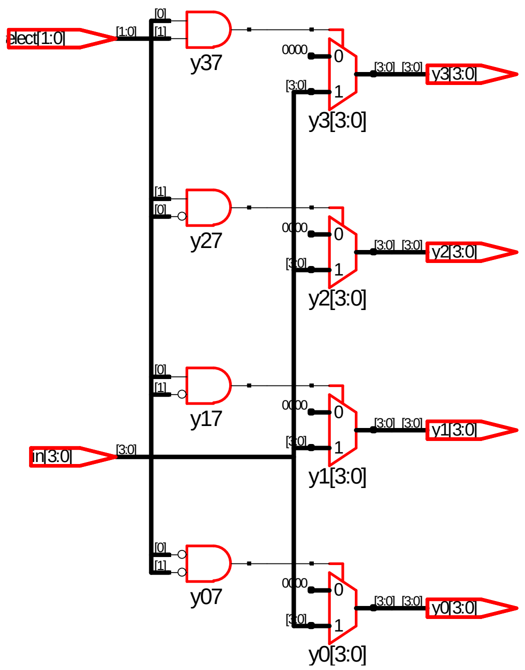


(c) Logic circuit

An *n*-bit 1-to-4 DeMultiplexer Example

```
// an N-bit 1-to-4 demultiplexer using if ... else structure
module demux_1to4_ifelse (select, in, y3, y2, y1, y0);
parameter N = 4;  // define the width of the demultiplexer
input      [1:0] select;
input      [N-1:0] in;
output reg [N-1:0] y3, y2, y1, y0;

// the body of the N-bit 1-to-4 demultiplexer
always @(select or in) begin
   if (select == 3) y3 = in; else y3 = {N{1'b0}};
   if (select == 2) y2 = in; else y2 = {N{1'b0}};
   if (select == 1) y1 = in; else y1 = {N{1'b0}};
   if (select == 0) y0 = in; else y0 = {N{1'b0}};
end
endmodule
```

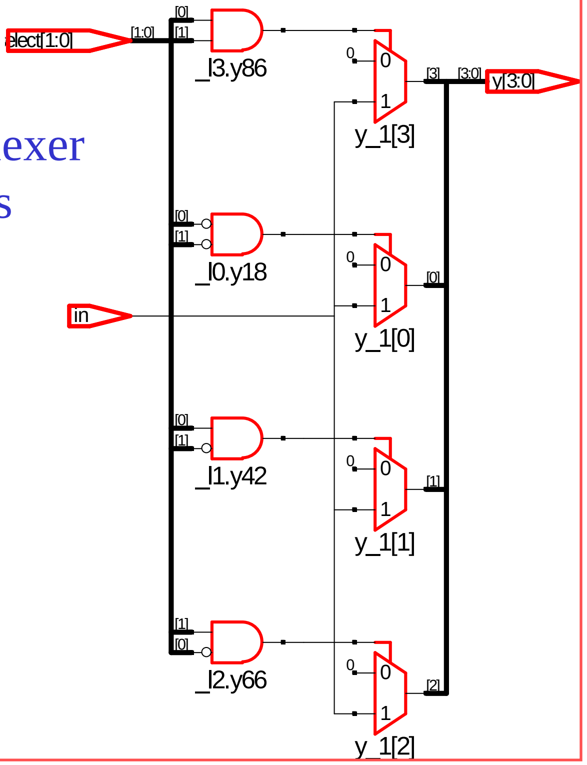


The Second *n*-bit 1-to-4 DeMultiplexer Example

```
// an N-bit 1-to-4 demultiplexer with enable control
module demux_1to4_ifelse_en(select, enable, in, y3, y2, y1, y0);
parameter N = 4;      // Define the width of the demultiplexer
input      [1:0] select;
input      enable;
input      [N-1:0] in;
output reg [N-1:0] y3, y2, y1, y0;
// the body of the N-bit 1-to-4 demultiplexer
always @(select or in or enable) begin
  if (enable)begin
    if (select == 3) y3 = in; else y3 = {N{1'b0}};
    if (select == 2) y2 = in; else y2 = {N{1'b0}};
    if (select == 1) y1 = in; else y1 = {N{1'b0}};
    if (select == 0) y0 = in; else y0 = {N{1'b0}};
  end else begin
    y3 = {N{1'b0}}; y2 = {N{1'b0}}; y1 = {N{1'b0}}; y0 = {N{1'b0}};
  end
end
endmodule
```

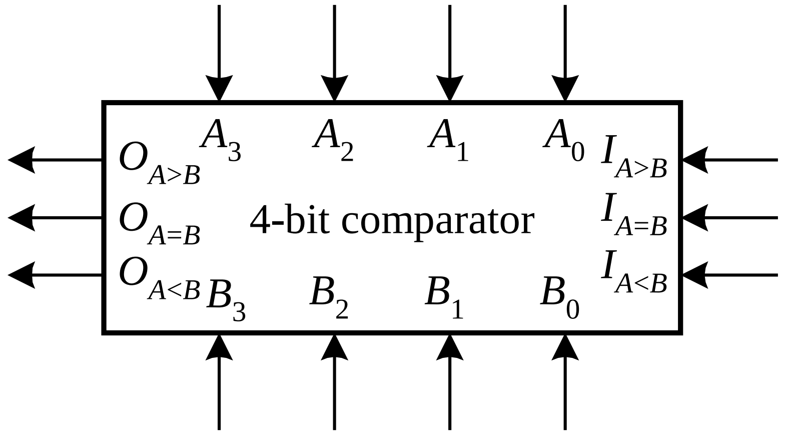
A Parameterized DeMultiplexer Example

```
// an example of 1-to-M demultiplexer module
module demux_1_to_m(select, in, y);
parameter M = 4;  // define the size of 1-to-m demultiplexer
parameter K= 2;   // define the number of selection lines
input  [K-1:0] select;
input  in;
output reg [M-1:0] y;
integer i;
// the body of the 1-to-M demultiplexer
always @(*)
   for (i = 0; i < M; i = i + 1) begin
       if (select == i) y[i] = in; else y[i] = 1'b0; end
endmodule
```



Comparators

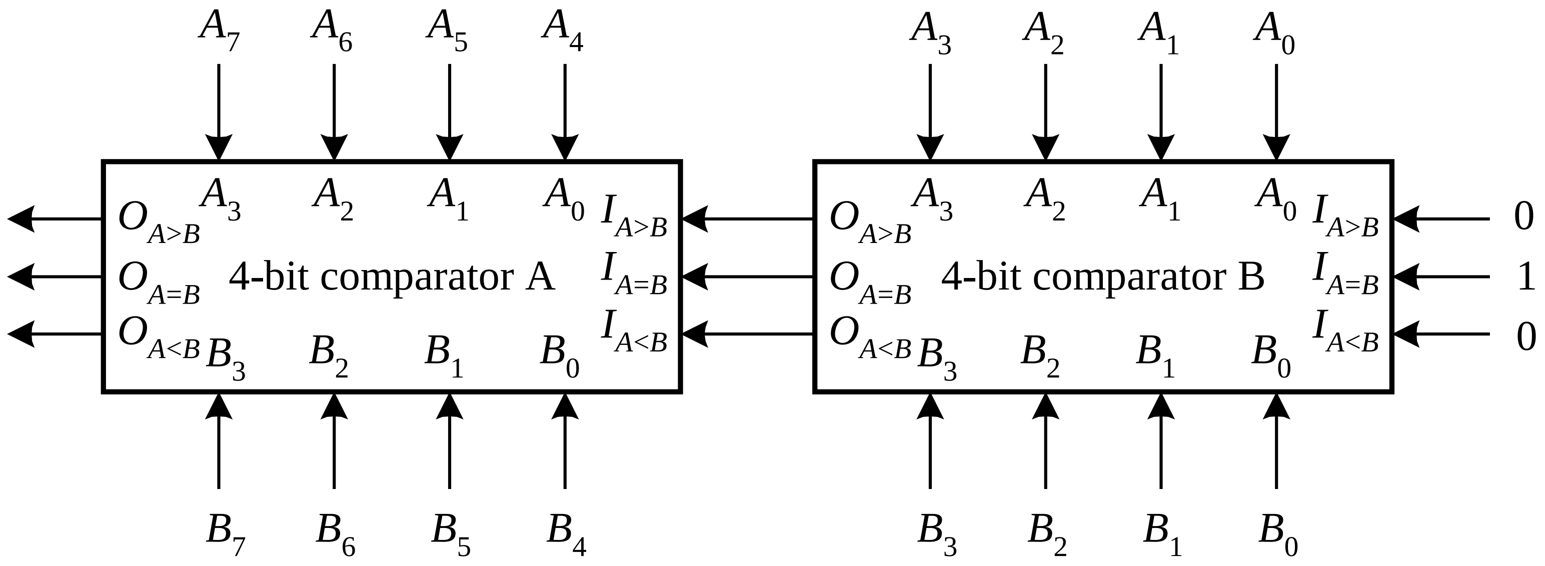
- A magnitude comparator is a device that determines the relative magnitude of two numbers being applied to it.
- Two types of magnitude comparator circuits:
  - Comparator
  - Cascadable comparator



A 4-bit cascadable comparator block diagram

Comparators

- Cascading two 4-bit comparators to form an 8-bit comparator.



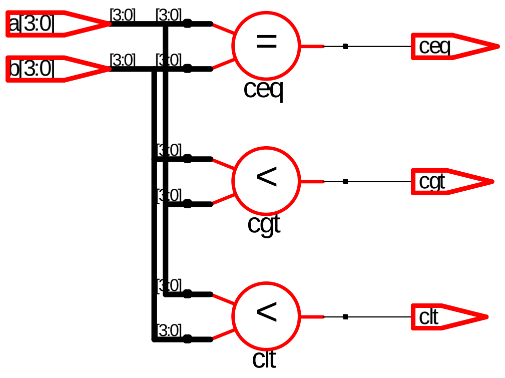
- What will happen if you set the input value (010) at the rightmost end to other values?

A Simple Comparator Example

```
// an N-bit comparator module example
module comparator_simple(a, b, cgt, clt, ceq);
parameter N = 4;   // define the size of comparator

// I/O port declarations
input  [N-1:0] a, b;
output cgt, clt, ceq;

// the body of the N-bit comparator
assign cgt  = (a > b);
assign clt   = (a < b);
assign ceq = (a == b);
endmodule
```

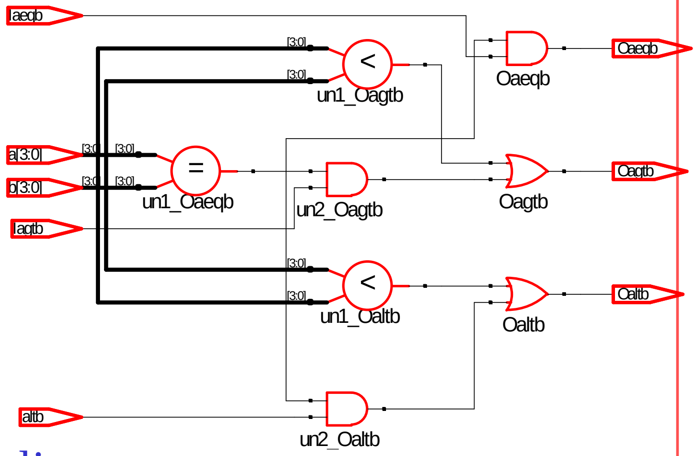


A Cascadable Comparator Example

```
module comparator_cascadable (Iagtb, Iaeqb, Ialtb, a, b, Oagtb, Oaeqb, Oaltb);
parameter N = 4;  // define the size of comparator

// I/O port declarations
input     Iagtb, Iaeqb, Ialtb;
input     [N-1:0] a, b;
output    Oagtb, Oaeqb, Oaltb;

// dataflow modeling using relation operators
assign Oaeqb = (a == b) && (Iaeqb == 1);  // equality
assign Oagtb = (a > b) || ((a == b)&& (Iagtb == 1));  // greater than
assign Oaltb =  (a < b) ||  ((a == b)&& (Ialtb == 1));  // less than
endmodule
```


Iaeqb
[3:0]
[3:0]
<
un1_Oagtb
Oaeqb
Oaeqb
a[3:0]
b[3:0]
[3:0]
[3:0]
[3:0]
[3:0]
=
un1_Oaeqb
un2_Oagtb
Oagtb
Oagtb
Iagtb
[3:0]
[3:0]
<
un1_Oaltb
Oaltb
Oaltb
Ialtb
un2_Oaltb