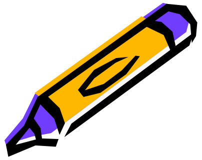
Combinational Logic Modules

Source:

- Slides partial from Digital System Designs and Practices Using Verilog HDL and FPGAs, Ming-Bo Lin
- Slides partial from Digital IC Design, By Pei-Yin Chen

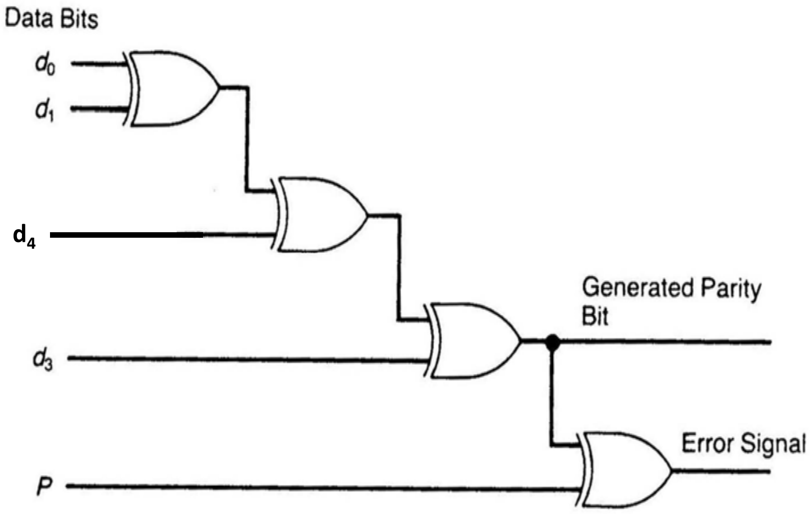


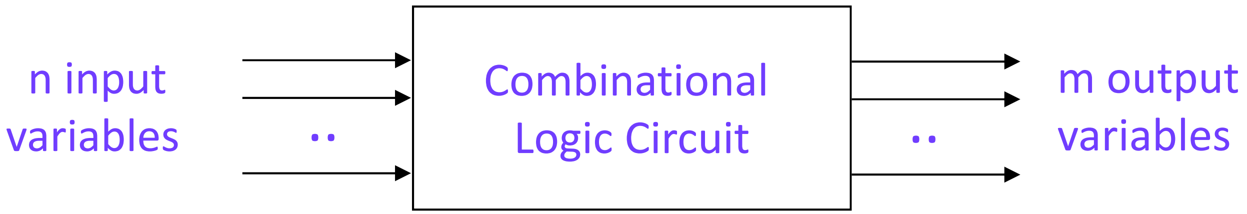
Outline

- Combinational Circuits
  - Decoders, and it’s implementation
  - Encoders, and it’s implementations
  - Priority encoders, and it’s implementation
  - Multiplexers, and it’s implementation
  - Demultiplexers, and it’s implementation
  - Comparators and Magnitude comparators, and it’s implementation
  - Describe how to design a parameterized module

Combinational Circuit

- A combinational circuit: Outputs at any time are determined directly from the present combination of inputs without regard to previous inputs.





n input variables

..

Combinational Logic Circuit

..

m output variables

Example – Alarm (1/2)

Assume that four persons might come. Alarm is activated when (1) more than three persons come or (2) the fourth person come together with other persons

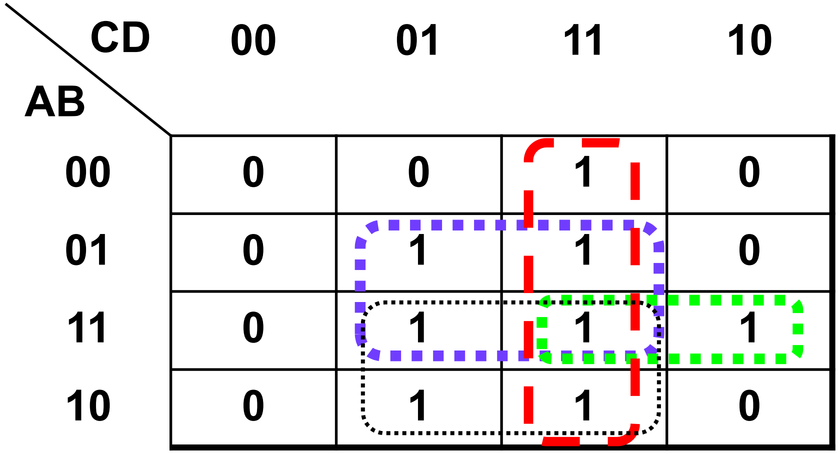
```
module four(A , B , C , D , Out);
input A , B , C , D;
output Out;
reg Out , temp;
always @(A or B or C or D)
begin
   case({A , B , C , D})
    4’b0000: Out = 0;
    4’b0001: Out = 0;
    4’b0010: Out = 0;
    4’b0011: Out = 1;
    4’b0100: Out = 0;
    4’b0101: Out = 1;
    4’b0110: Out = 0;
    4’b0111: Out = 1;
    4’b1000: Out = 0;
    4’b1001: Out = 1;
    4’b1010: Out = 0;
    4’b1011: Out = 1;
    4’b1100: Out = 0;
    4’b1101: Out = 1;
    4’b1110: Out = 1;
    default: Out = 1;
    endcase
    end
    endmodule
```

Optimization is done by tools

A	B	C	D	Out
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	1
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

Example – Alarm (2/2)

```
module four(A , B , C , D);
input A , B , C , D;
output Out;

wire t1 , t2 , t3 , t4;

and a1(t1 , A , D);
and a2(t2 , B , D);
and a3(t3 , C , D);
and a4(t4 , A , B , C);
or o1(Out , t1 , t2 , t3 , t4);
endmodule
```



AB \ CD	00	01	11	10
00	0	0	1	0
01	0	1	1	0
11	0	1	1	1
10	0	1	1	0

Out = AD + BD + CD + ABC

Traditional design method

(optimization is done by hand)

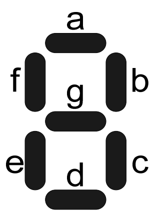
⇨ not suitable for HDL design

Example - Seven Segment Display

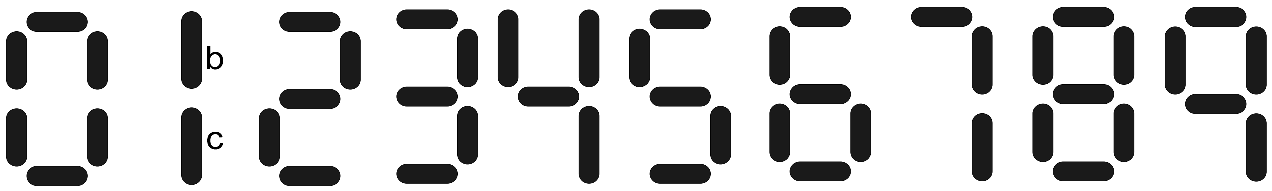
A BCD (Binary-Coded Decimal)-to-seven-segment decoder is a combinational circuit that accepts a decimal digit in BCD and generates the appropriate output for selection of segments in a display indicator used for displaying the decimal digit.

The seven outputs of the decoder (a, b, c, d, e, f, g) select the corresponding segments in the display as shown in Fig. (a). The numeric designation chosen to represent the decimal digit is shown in Fig. (b).

Design the BCD-to-seven-segment decoder circuit.



(a) Segment designation



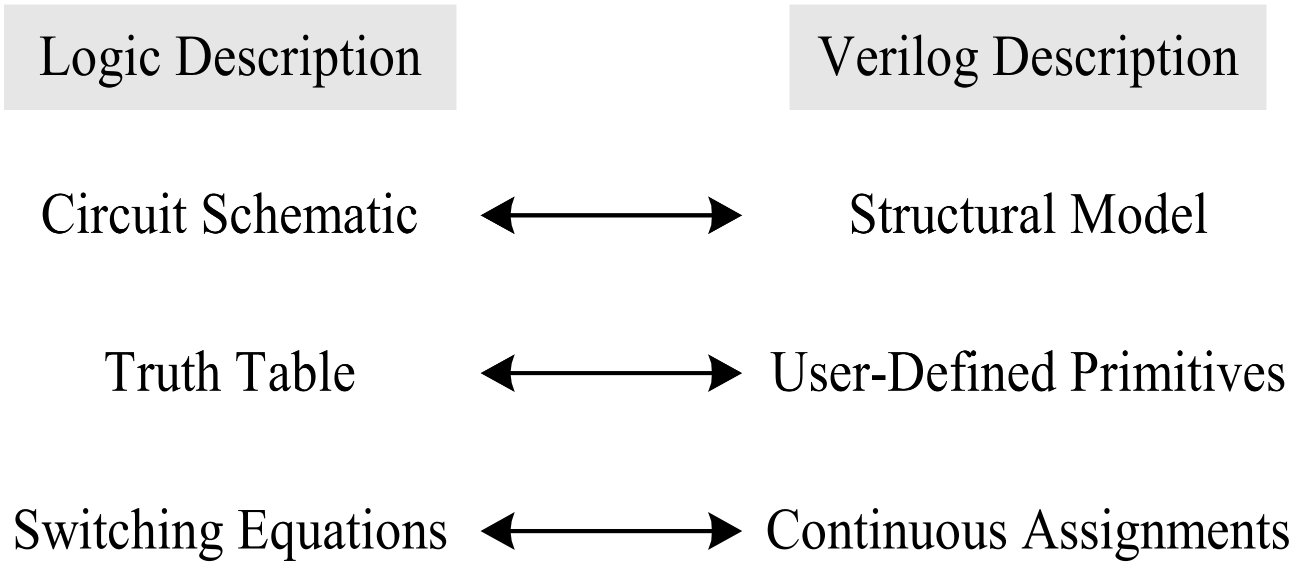
(b)Numerical designation for display

A	B	C	D	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
	.	.	.			.	.	.		

Options for Modeling Combinational Logic

- Options for modeling combinational logic:
  - Verilog HDL primitives
    - E.g.: and (a, b, c)
  - Continuous assignment
    - E.g.: assign out = i1& i2;
  - Behavioral statement
    - E.g.: initial, always
  - Interconnected combinational logic modules
  - Combinational UDP (User-defined primitives)

Three Descriptions of Combination Logic



Logic Description — Verilog Description

Circuit Schematic ↔ Structural Model

Truth Table ↔ User-Defined Primitives

Switching Equations ↔ Continuous Assignments

UDP example

Not Synthesizable

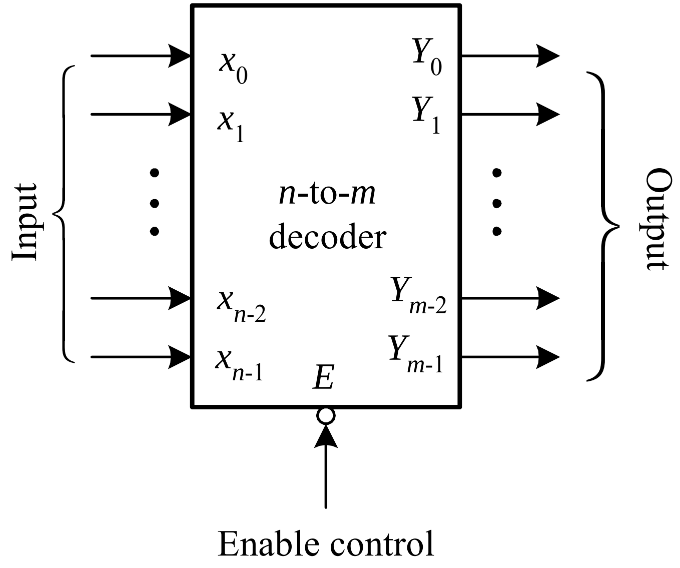
```
//To be used with the file fulladd.v
//Primitive name and terminal list
primitive udp_and(out, a, b);

//Declarations
output out; //must not be declared as reg for combinational UDP
input a, b; //declarations for inputs.

//State table definition; starts with keyword table
//The following comment is for readability only
//Input entries of the state table must be in the
//same order as the input terminal list.
table
 // a   b   :   out;
     0   0   :   0;
     0   1   :   0;
     1   0   :   0;
     1   1   :   1;

endtable //end state table definition
endprimitive //end of udp_and definition
```

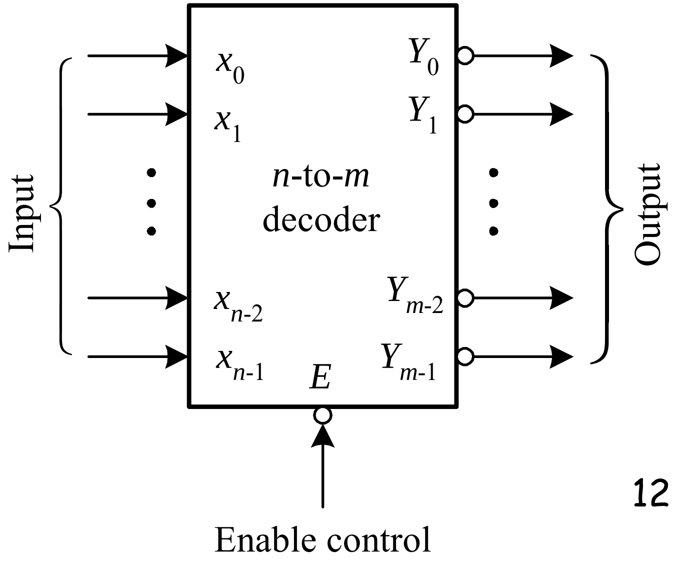
Basic Combinational Logic Modules

- Commonly used combinational logic modules:
  - Decoder
  - Encoder
  - Multiplexer
  - Demultiplexer
  - Comparator
  - Adder (Carry Lookahead Adder)
  - Subtracter
  - Multiplier
  - PLA (Programmable Logic Array)
  - Parity Generator

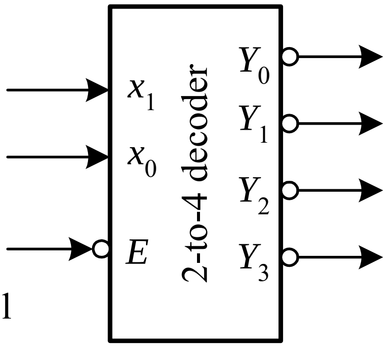
Decoder Block Diagrams

- An $n \times m$ decoder has n input lines and m output lines. Each output line Y_i corresponds to the *i*th minterm of input (line) variables.
  - Total decoding: when $m = 2^n$
    - *E.g.* (3, 8)
  - Partial decoding: when $m < 2^n$.





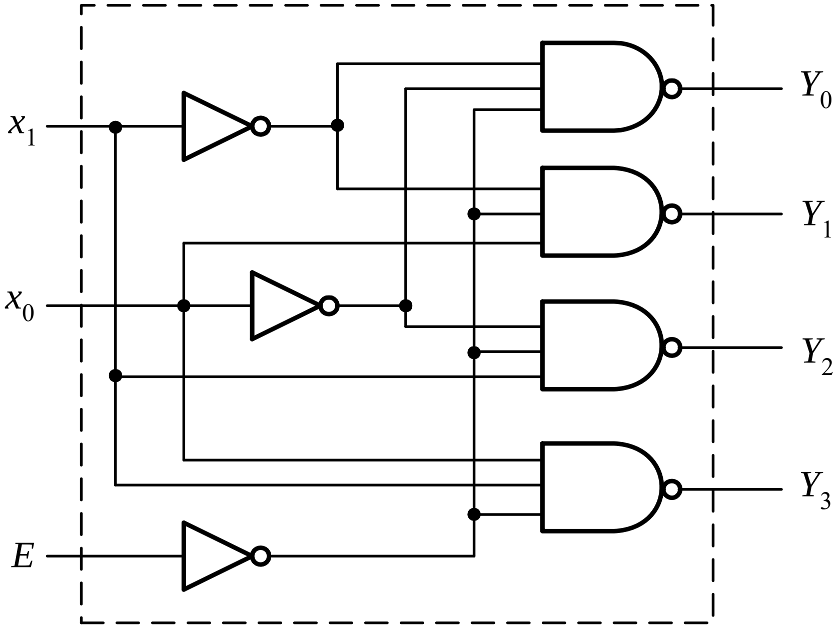
A 2-to-4 Decoder Example



(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

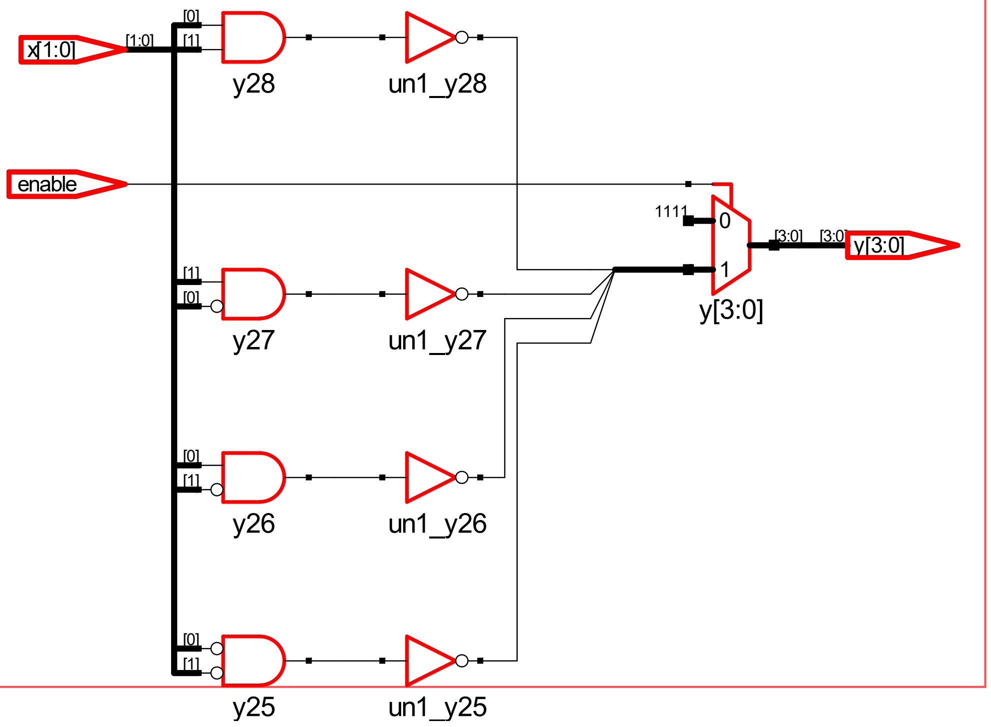
(b) Function table



(c) Logic circuit

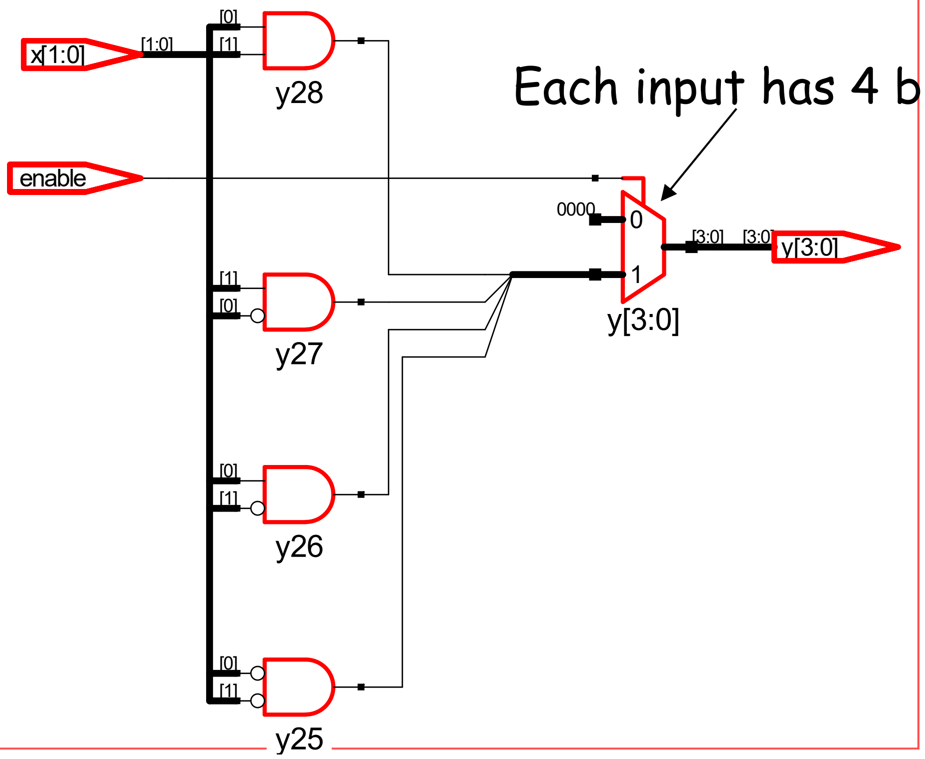
A 2-to-4 Decoder Example

```
// 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;
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;

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
```



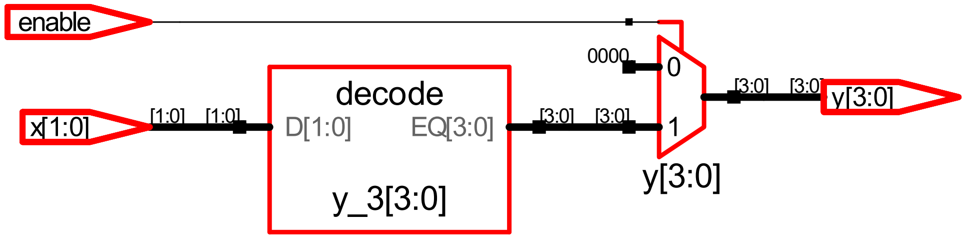
Each input has 4 bits

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
```

```
// a 2-to-4 decoder with active-
high output
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
```

000=> 00000001 shift 0 bit
001=> 00000001 shift 1 bit
010=> 00000001 shift 2 bits ….



Q: Design an m-to-n decoder with active-low output.

Expansion of Decoders

- Building a big decoder using small decoders

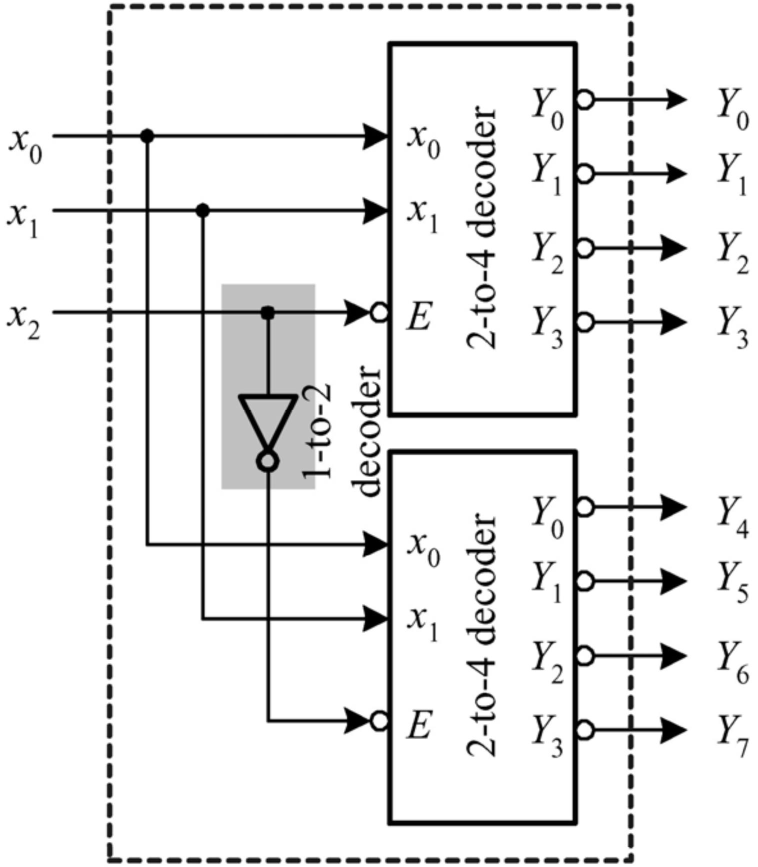
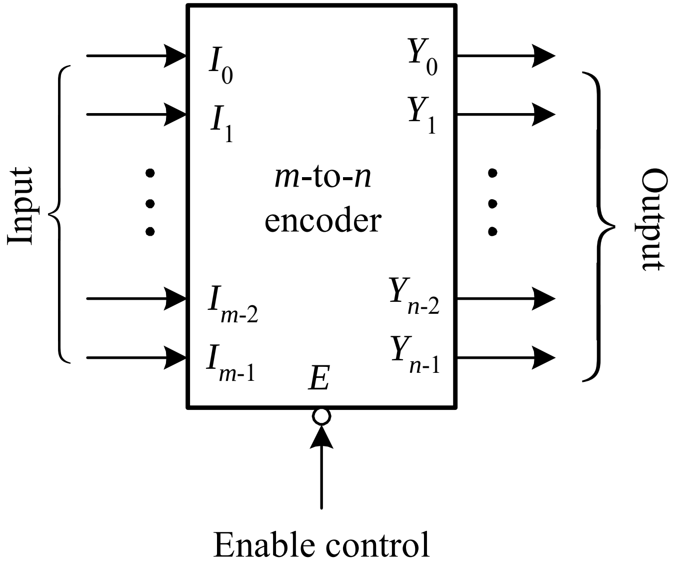


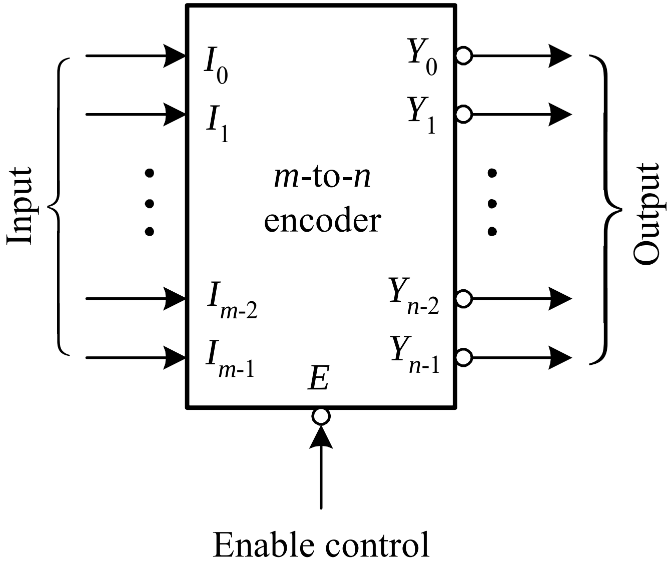
Figure 8.3: A 3-to-8 decoder constructed by cascading two 2-to-4 decoders.

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

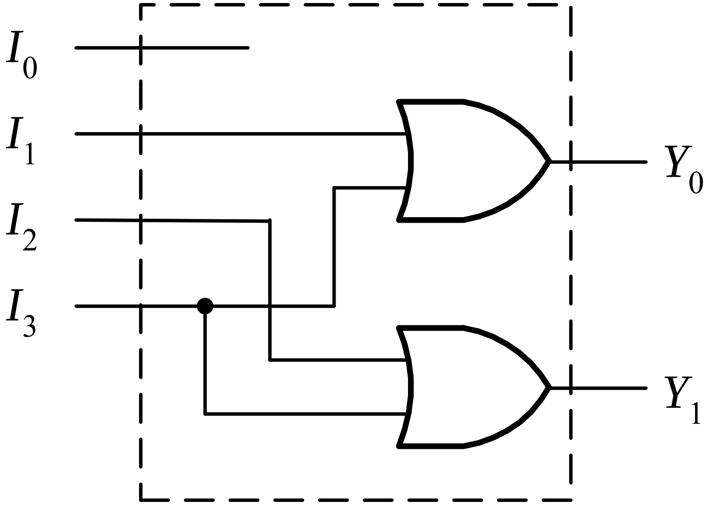


(b) Inverted output

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

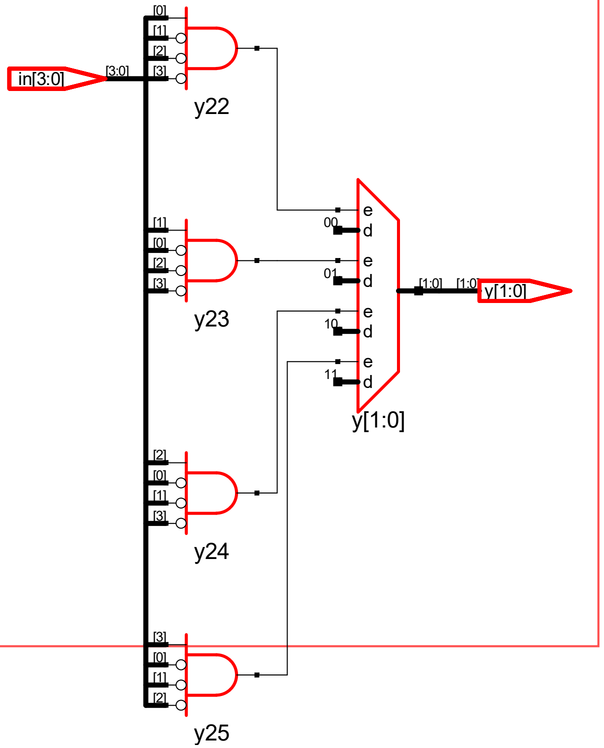


(b) Logic circuit

A 4-to-2 Encoder Example

using if ... else structure

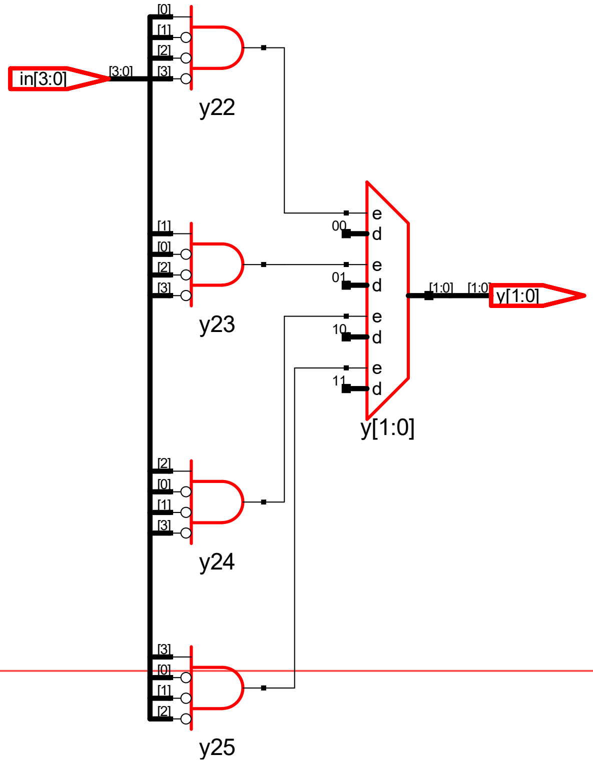
```
module encoder_4to2_ifelse(in, y);
input  [3:0] in;
output reg [1:0] y;
always @(in) begin
   if (in == 4'b0001) y = 0; else
   if (in == 4'b0010) y = 1; else
   if (in == 4'b0100) y = 2; else
   if (in == 4'b1000) y = 3; else
       y = 2'bx;
end
endmodule
```



Another 4-to-2 Encoder Example

using case structure

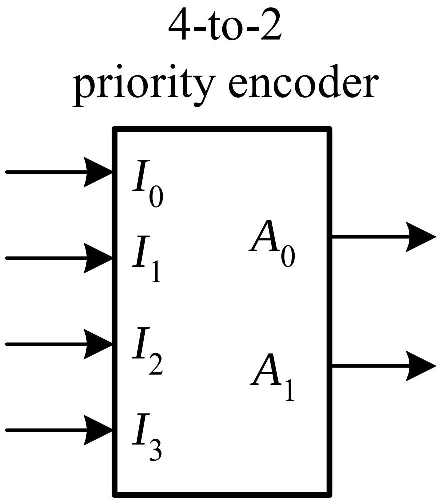
```
module encoder_4to2_case(in, y);
input   [3:0] in;
output reg [1:0] y;
always @(in)
   case (in)
      4'b0001 : y = 0;
      4'b0010 : y = 1;
      4'b0100 : y = 2;
      4'b1000 : y = 3;
      default : y = 2'bx;
   endcase
endmodule
```



A 4-to-2 Priority Encoder

A priority is associated with the index values of inputs:

Priority $I_3 > I_2 > I_1 > I_0$



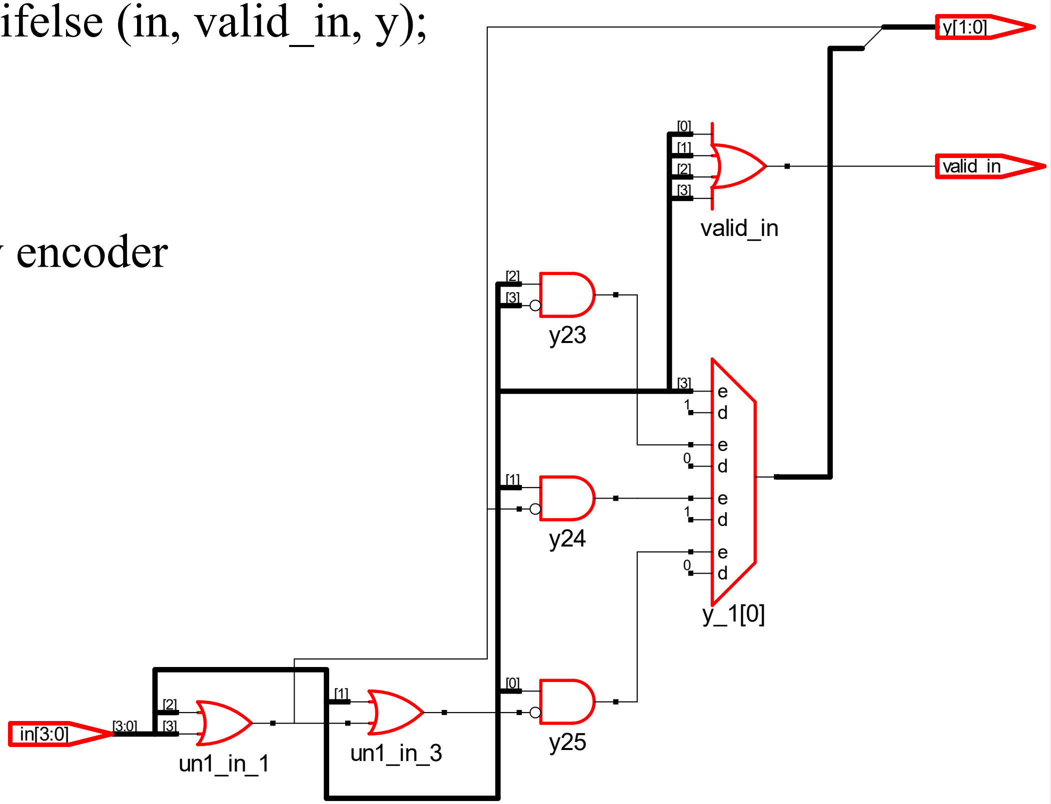
(a) Block diagram

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
   if (in[2]) y = 2; else
   if (in[1]) y = 1; else
   if (in[0]) y = 0; else
   y = 2'bx;
end
endmodule
```

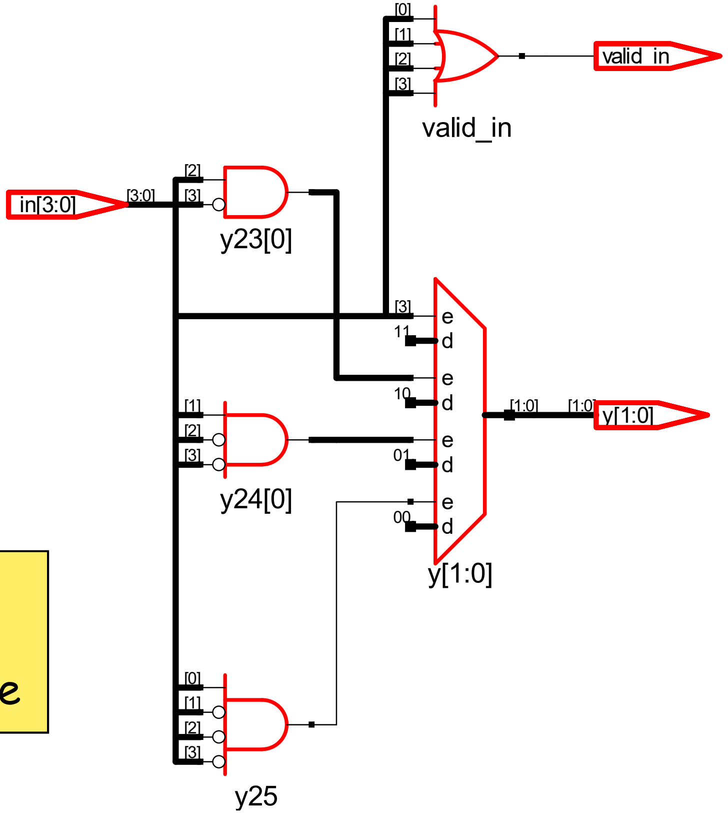


Position 3 has the highest priority

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
```

Avoid infering a latch because casex statement is incomplete



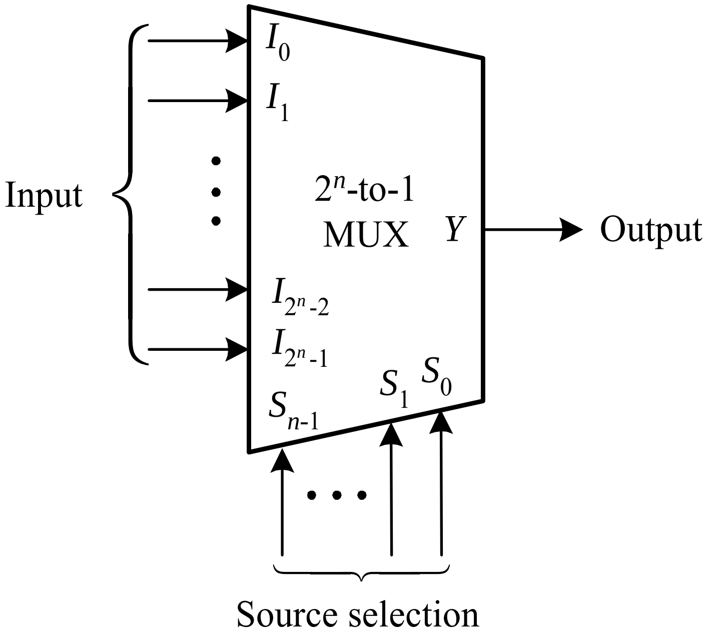
casez : Treats z as don't care.
casex : Treats x and z as don't care

An m-to-n Priority Encoder using a for Loop

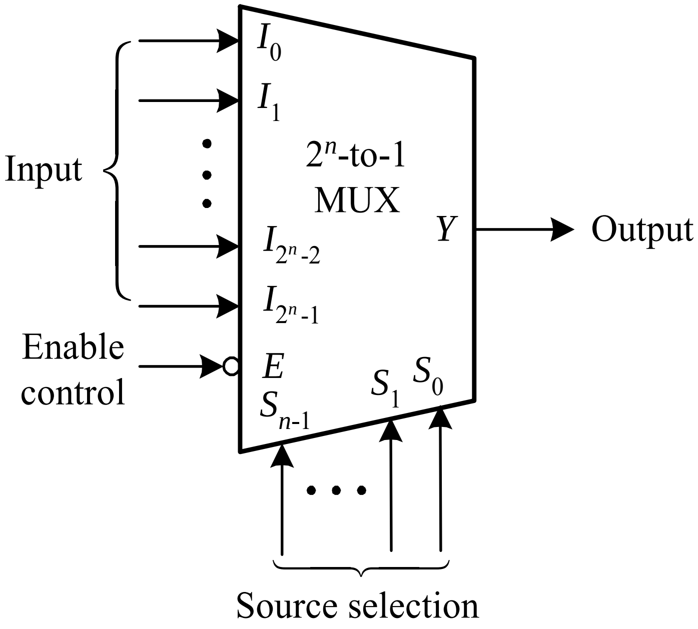
```
// a parameterized M-to-N priority encoder.
module priencoder_m2n(x, valid_in, y);
parameter M = 8;  // define the number of inputs
parameter N = 3;  // define the number of outputs
input  [M-1:0] x;
output valid_in;  // indicates the data input x is valid.
output reg [N-1:0] y;
integer i;
// the 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;
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*.



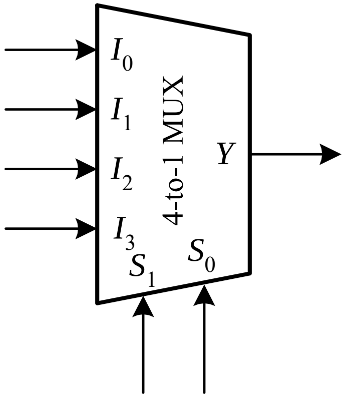
Without enable control



With enable control

A 4-to-1 Multiplexer Example

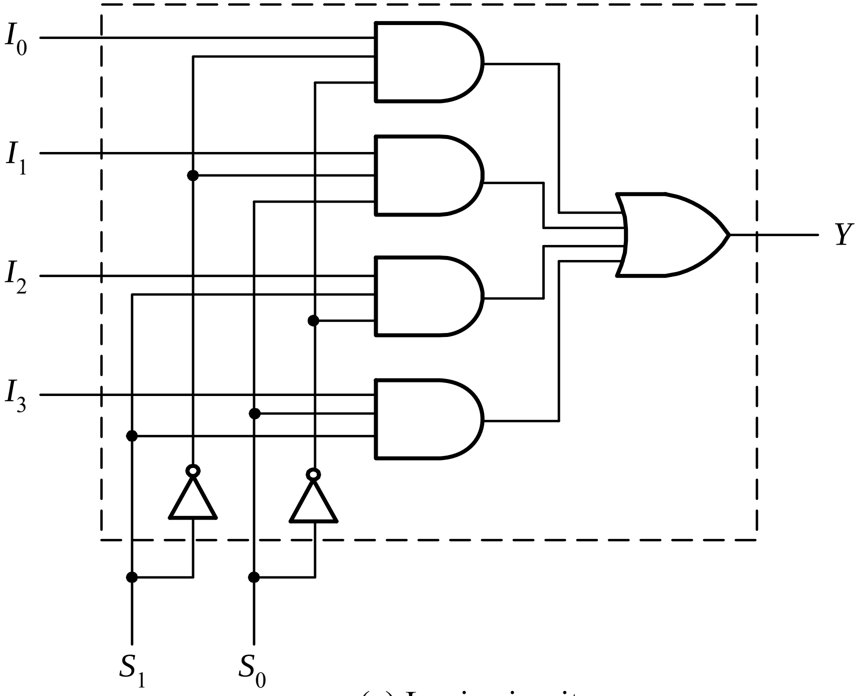
Gate-based 4-to-1 multiplexers



(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

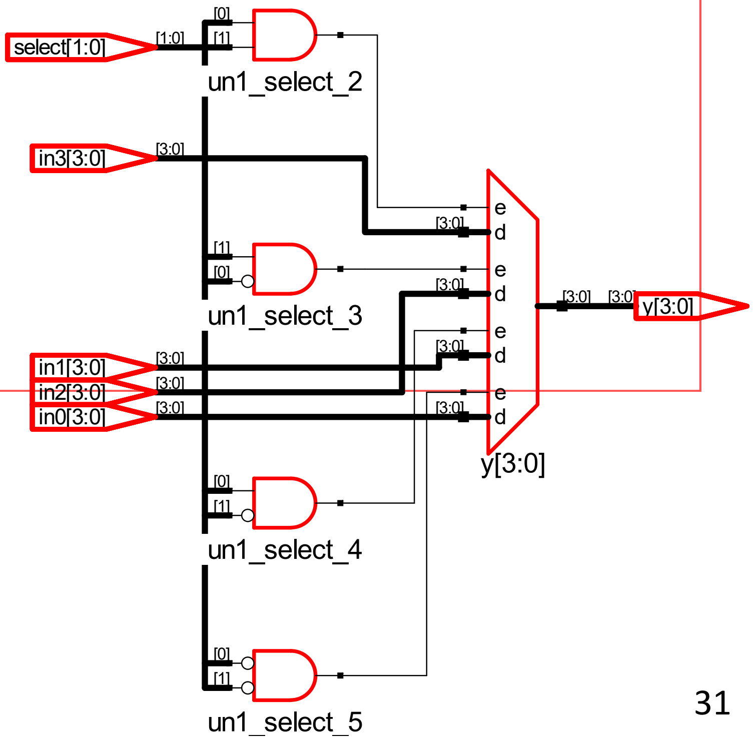


(c) Logic circuit

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
```



Data flow modeling

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

// an N-bit 4-to-1 multiplexer with enable control.

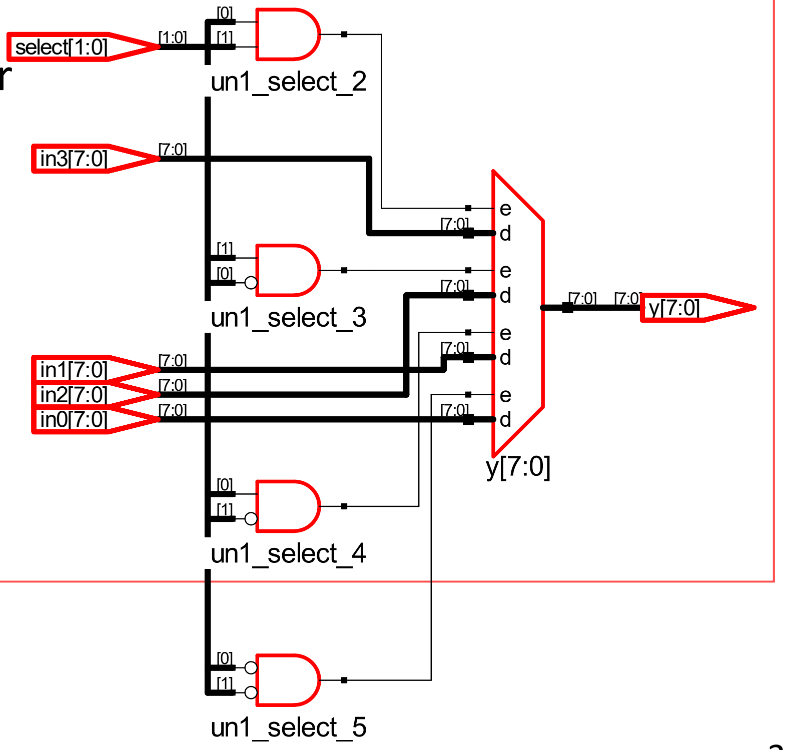
```
module mux_nbit_4to1_en (select, enable, in3, in2, in1, in0, y);
parameter N = 4; // define the width of 4-to-1 multiplexer
input [1:0] select;
input enable;
input [N-1:0] in3, in2, in1, in0;
output reg [N-1:0] y;
// the body of the N-bit 4-to-1 multiplexer
always @(select or enable or in0 or in1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

Behavioral modeling

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

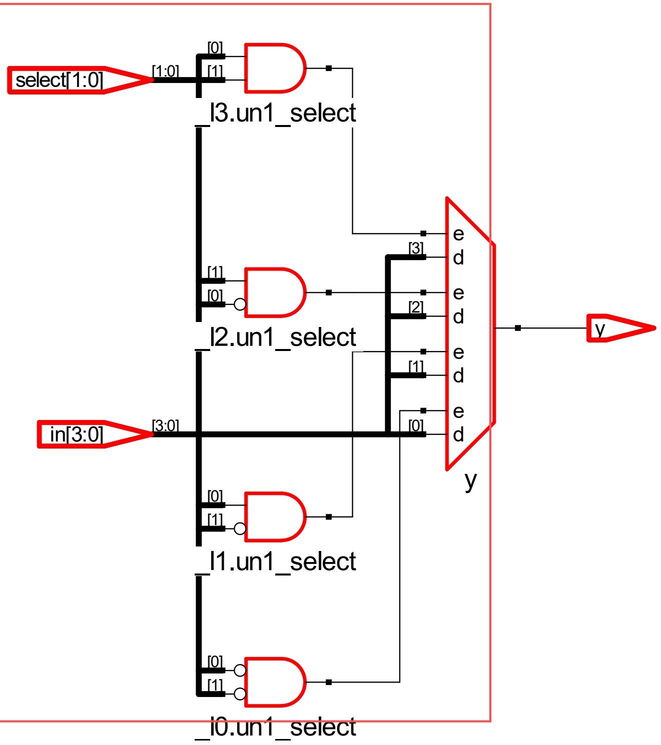
// an N-bit 4-to-1 multiplexer using case structure.

```
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 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
```



```
If(select==0) y=in[0]
If(select==1) y=in[1]
If(select==2) y=in[2]
If(select==3) y=in[3]
```

Q: Explain the philosophy behind this program. Why is it so simple? (*Hint*: Please go back to the basic definition of multiplexer.)

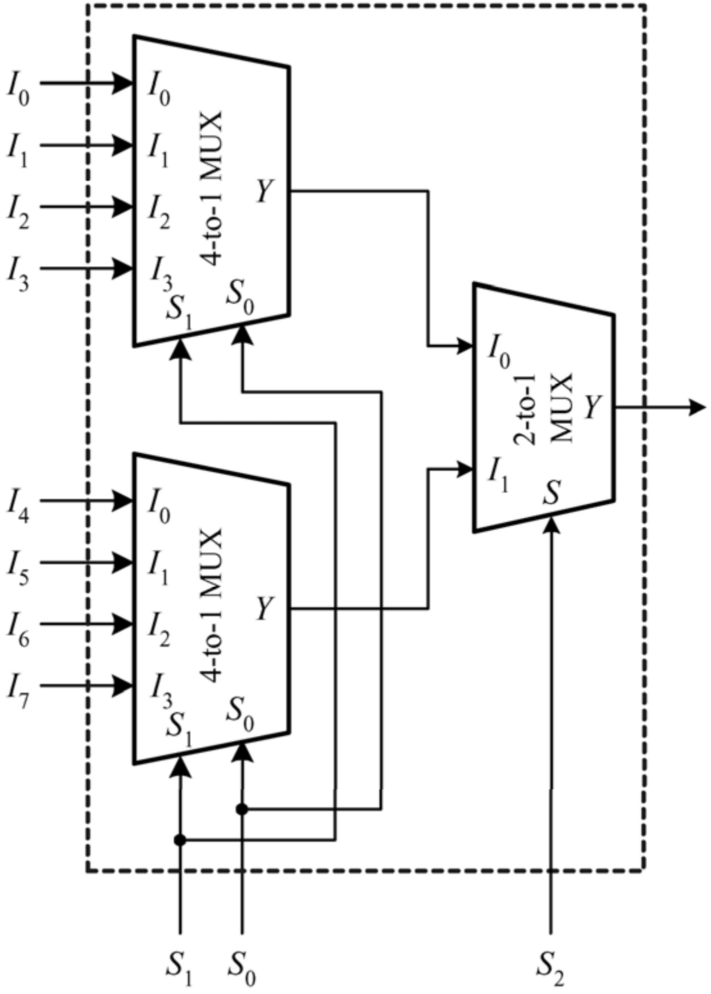
Expansion of Multiplexers

- Two ways to build big mux

1. Straightforward approach like what we have done

2. Cascading small multiplexer modules to construct a big mux

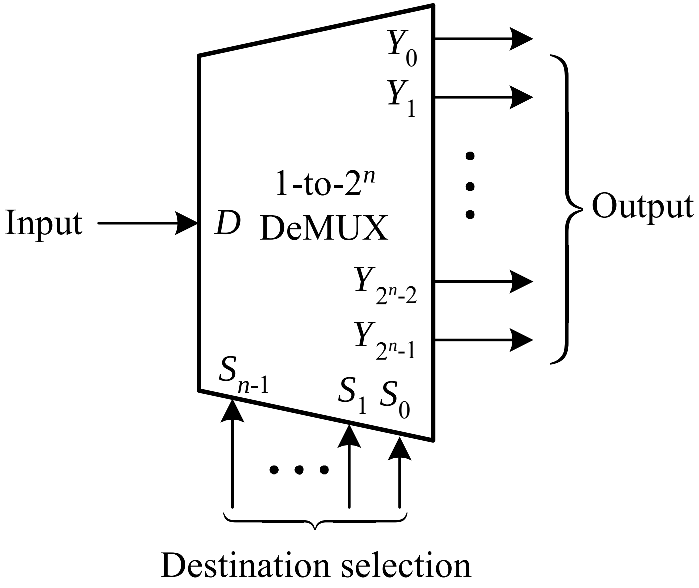
- Multiplexer Tree



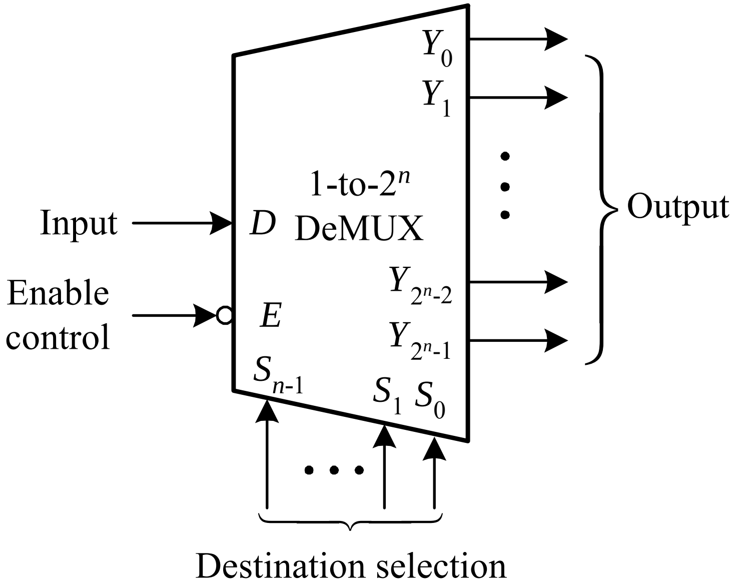
8-to-1 mux constructed by cascading two 4-to-1 and one 2-to-1 muxes

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.



Without enable control

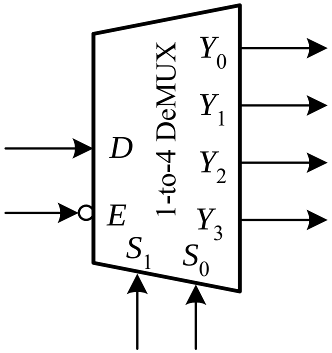


With enable control

A 1-to-4 DeMultiplexer Example

Gate-based 1-to-4 demultiplexers

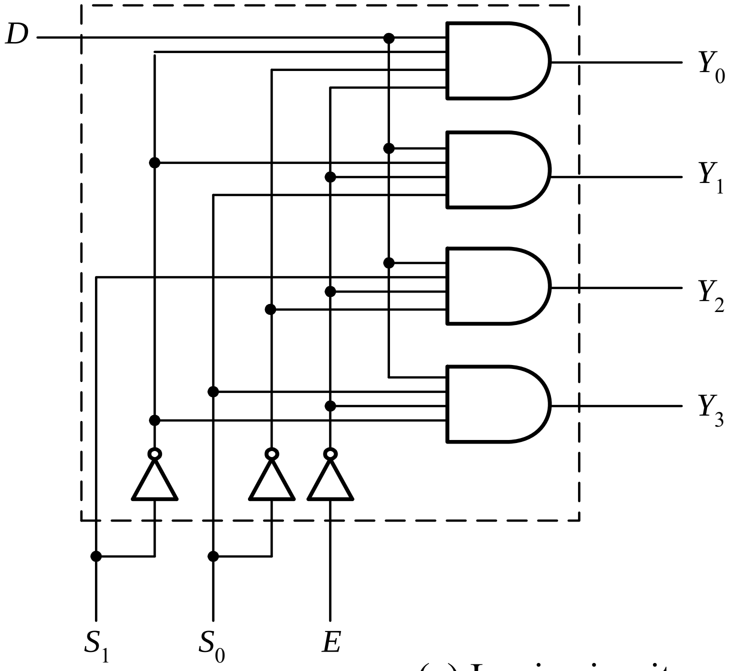
Input D is directed to the output Y



(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

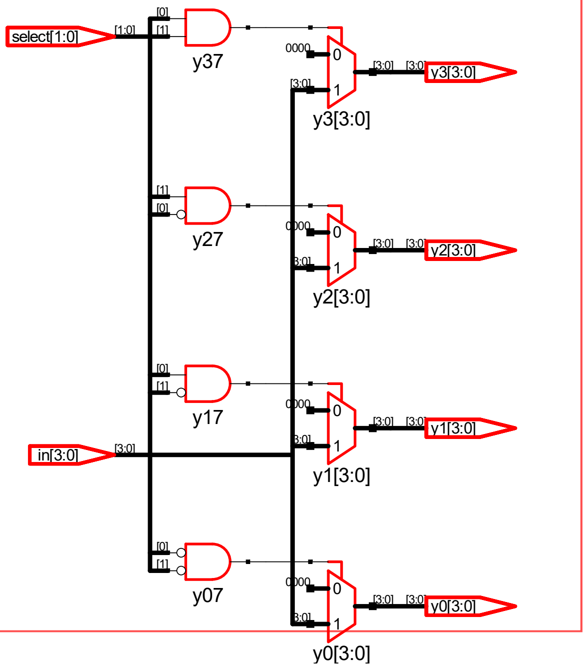


(c) Logic circuit

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 with Enable control

An N-bit 1-to-4 demultiplexer using if … else structure with Enable control

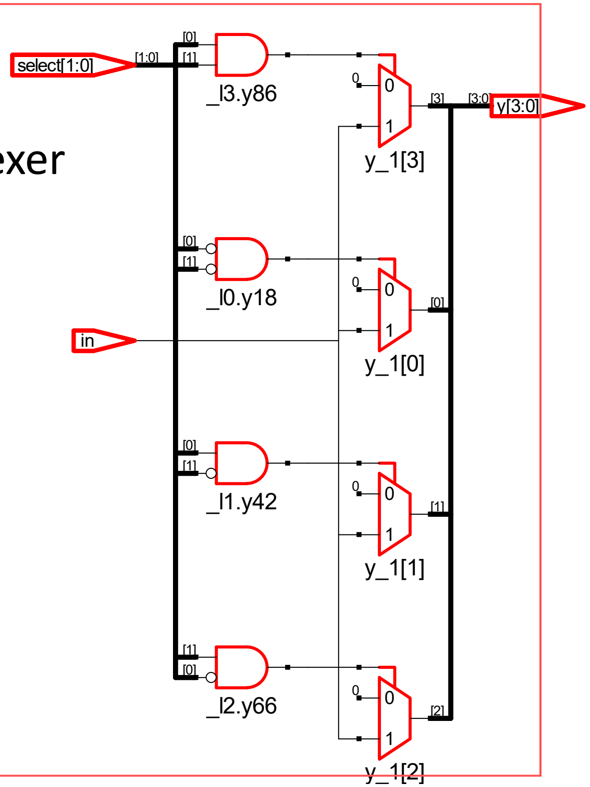
```
module demux_1to4_ifelse_en(select, enable, in, y3, y2, y1, y0);
parameter N = 4;      // Define the width of the demultiplexer
input      [1:0] select;
input      enable;
input      [N-1:0] in;
output reg [N-1:0] y3, y2, y1, y0;
// the body of the N-bit 1-to-4 demultiplexer
always @(select or in 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
```

n-bit 1-to-4 DeMultiplexer using case struct

```
module demux_1to4_case (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
  y3 = {N{1'b0}}; y2 = {N{1'b0}};
  y1 = {N{1'b0}}; y0 = {N{1'b0}};
  case (select)
    2'b11: y3 = in;
    2'b10: y2 = in;
    2'b01: y1 = in;
    2'b00: y0 = in;
  endcase
end
endmodule
```

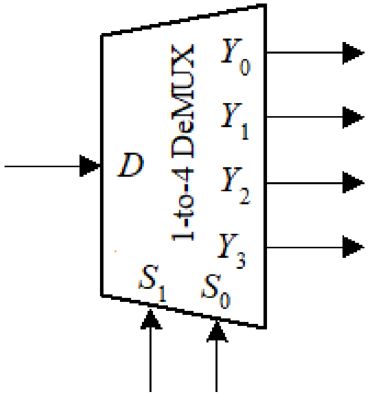
A Parameterized DeMultiplexer Example

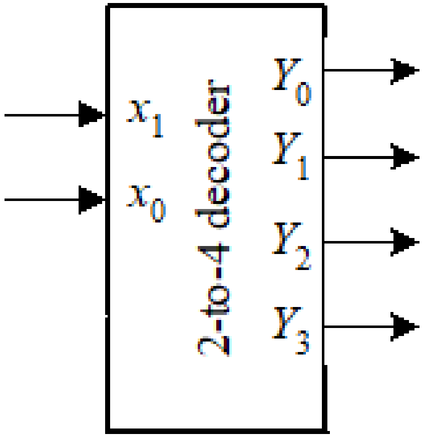
```
// an example of 1-to-M demultiplexer module
module demux_1_to_m(select, in, y);
parameter M = 4;  // define the size of 1-to-m demultiplexer
parameter K= 2;   // define the number of selection lines
input  [K-1:0] select;
input  in;
output reg [M-1:0] y;
integer i;
// the body of the 1-to-M demultiplexer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



DeMux vs. Decoder

- A 1-to-4 demux may function as a 2-to-4





S1	S0	Y3	y2	Y1	y0
0	0	0	0	0	D
0	1	0	0	D	0
1	0	0	D	0	0
1	1	D	0	0	0

x1	x0	Y3	y2	Y1	y0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

Set D to 1, become a decoder

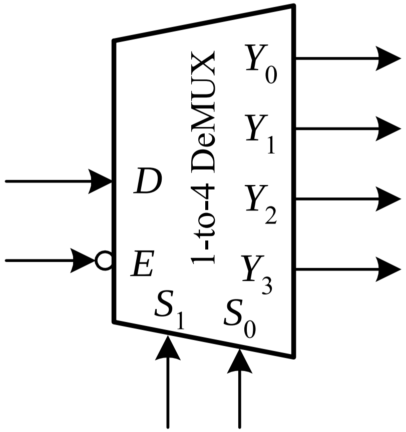
Question

- Convert a 1-to-8 demux may function as a 3-to-8 decoder

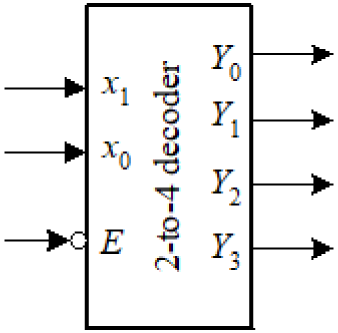
DeMux (with enable) vs. Decoder

- In 1-to-4 Demux become a 2-to-4 decoder

➔ If data input is set to 1 and has enable control



Set D to 1



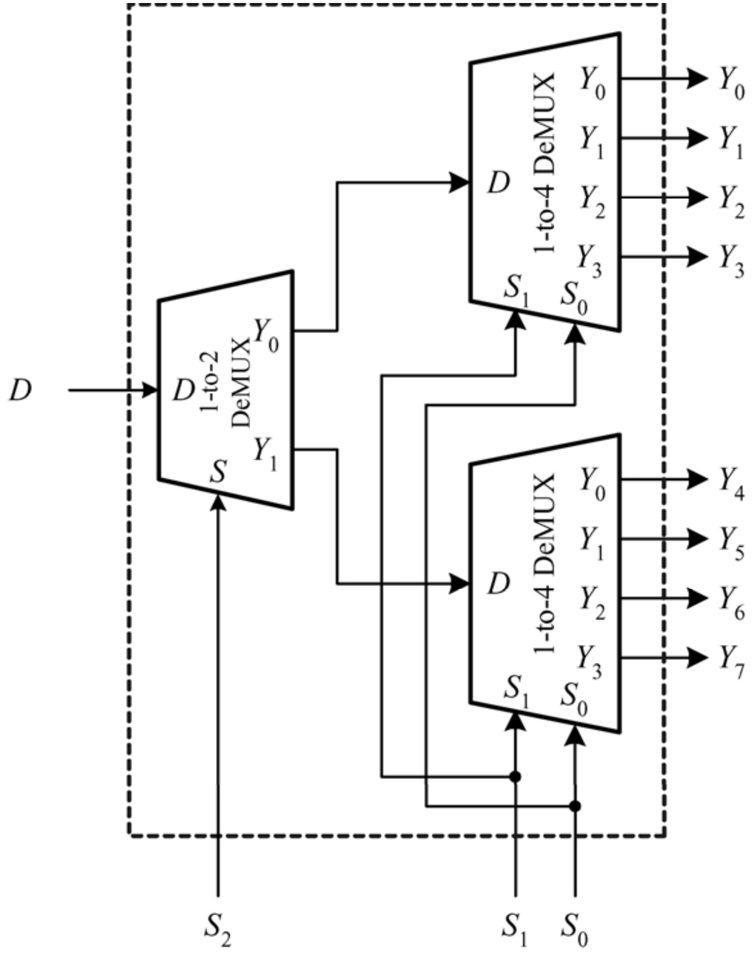
E	S_1	S_0	Y_3	Y_2	Y_1	Y_0
1	ϕ	ϕ	0	0	0	0
0	0	0	0	0	0	D
0	0	1	0	0	D	0
0	1	0	0	D	0	0
0	1	1	D	0	0	0

D=1

E	S1	S0	Y3	y2	Y1	y0
1	X	X	0	0	0	0
0	0	0	0	0	0	1
0	0	1	0	0	1	0
0	1	0	0	1	0	0
0	1	1	1	0	0	0

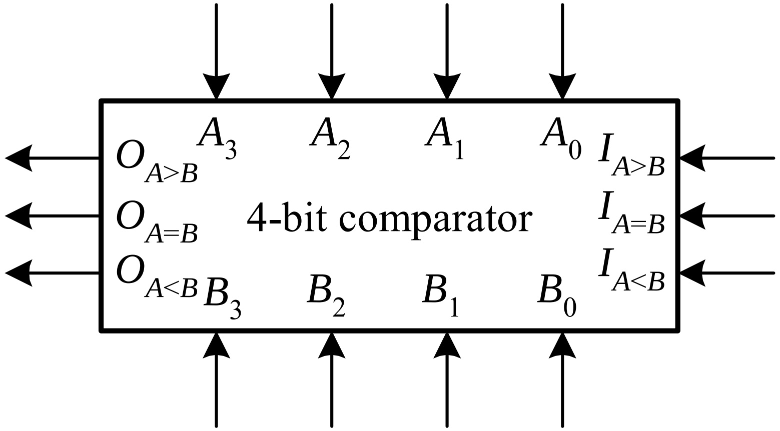
Expansion of DeMux

- A big DeMux, such as 1-to-128, may be needed
- A 1-to-8 DeMux constructed by cascading two 1-to-4 and one 1-to-2 DeMux
- Using case or if-else statement or demultiplexer tree



Comparators

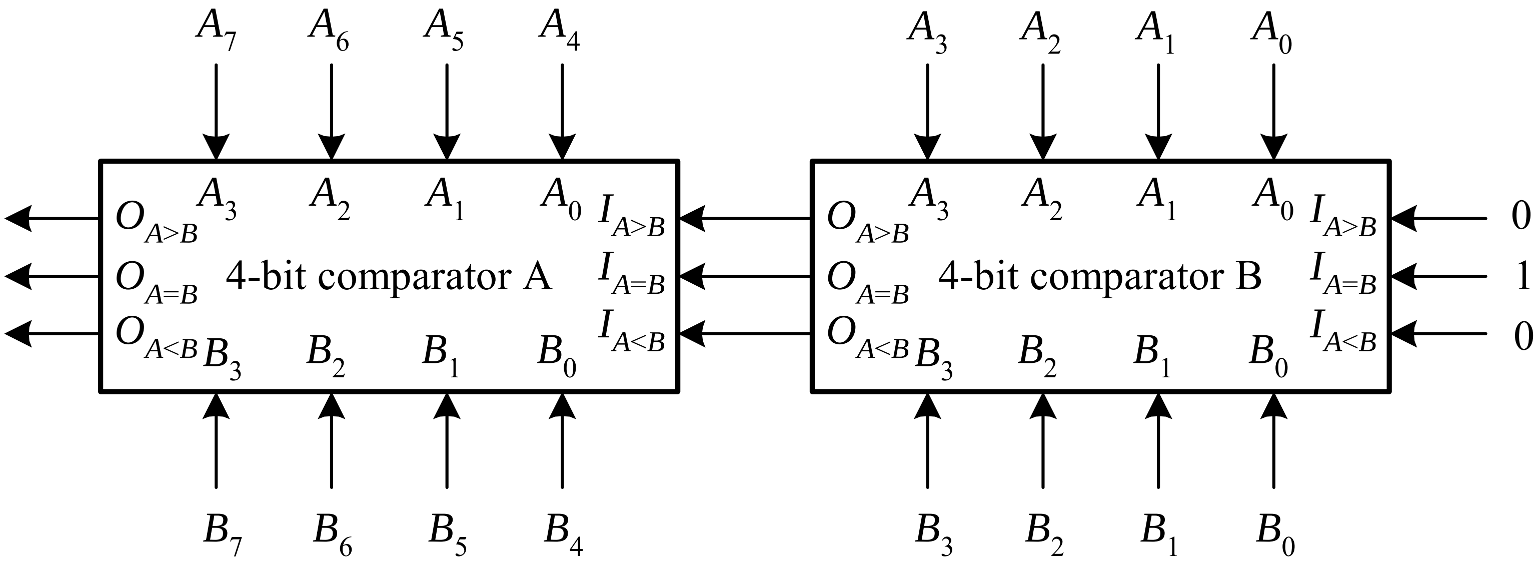
- Equality comparators:
  - Determine whether two numbers are equal
  - using n-bit XOR gate
- Magnitude comparator:
  - Determines the relative magnitude of two numbers
- Two types of magnitude comparator circuits:
  - Comparator
  - Cascadable comparator



A 4-bit cascadable comparator block diagram

Comparators

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

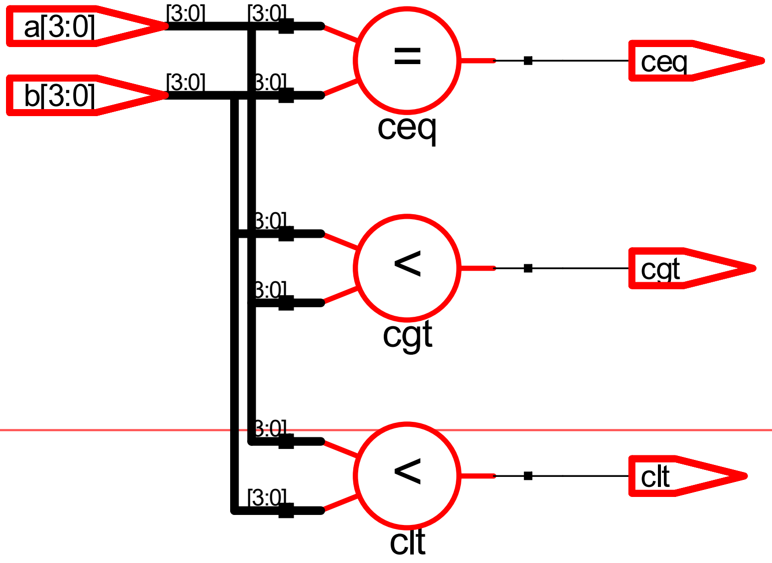


•What will happen if you set the input value (010) at the rightmost end to other values?

A Comparator Example

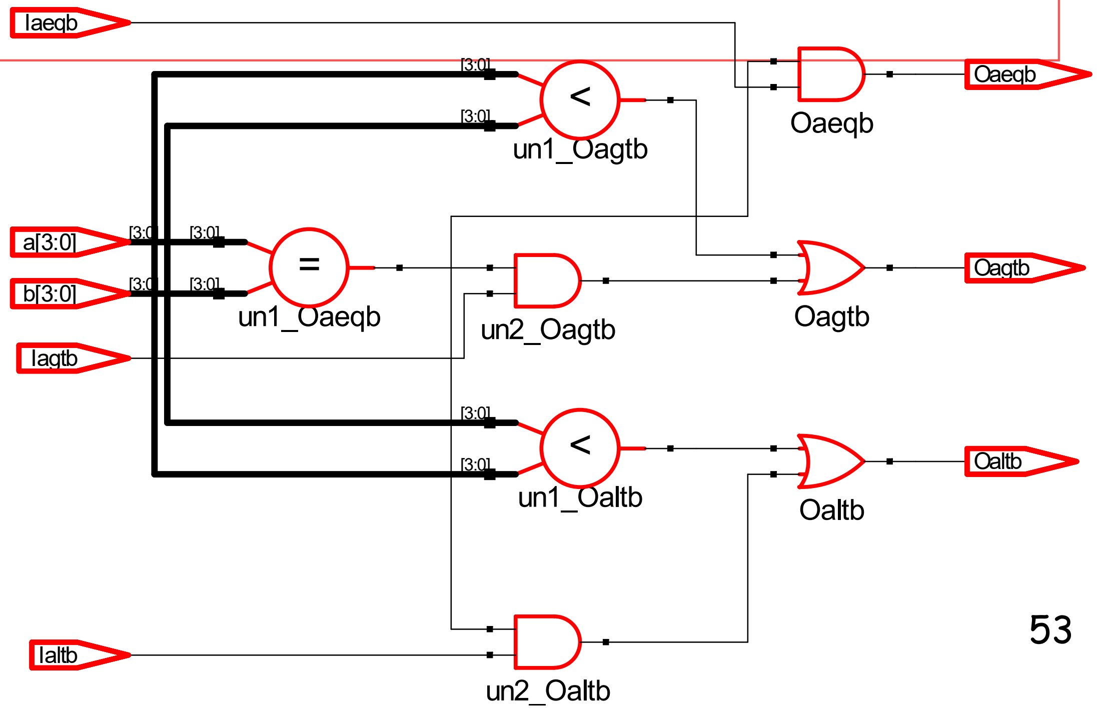
an N-bit comparator module example

```
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

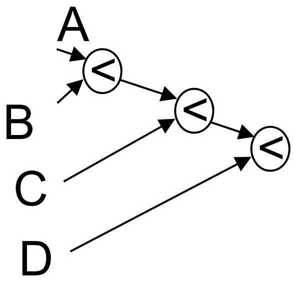
```
module comparator_cascadable (Iagtb, Iaeqb, Ialtb, a, b, Oagtb, Oaeqb, Oaltb);
parameter N = 4;  // define the size of comparator
// I/O port declarations
input     Iagtb, Iaeqb, Ialtb, input     [N-1:0] a, b;
output   Oagtb, Oaeqb, Oaltb;

// 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
```

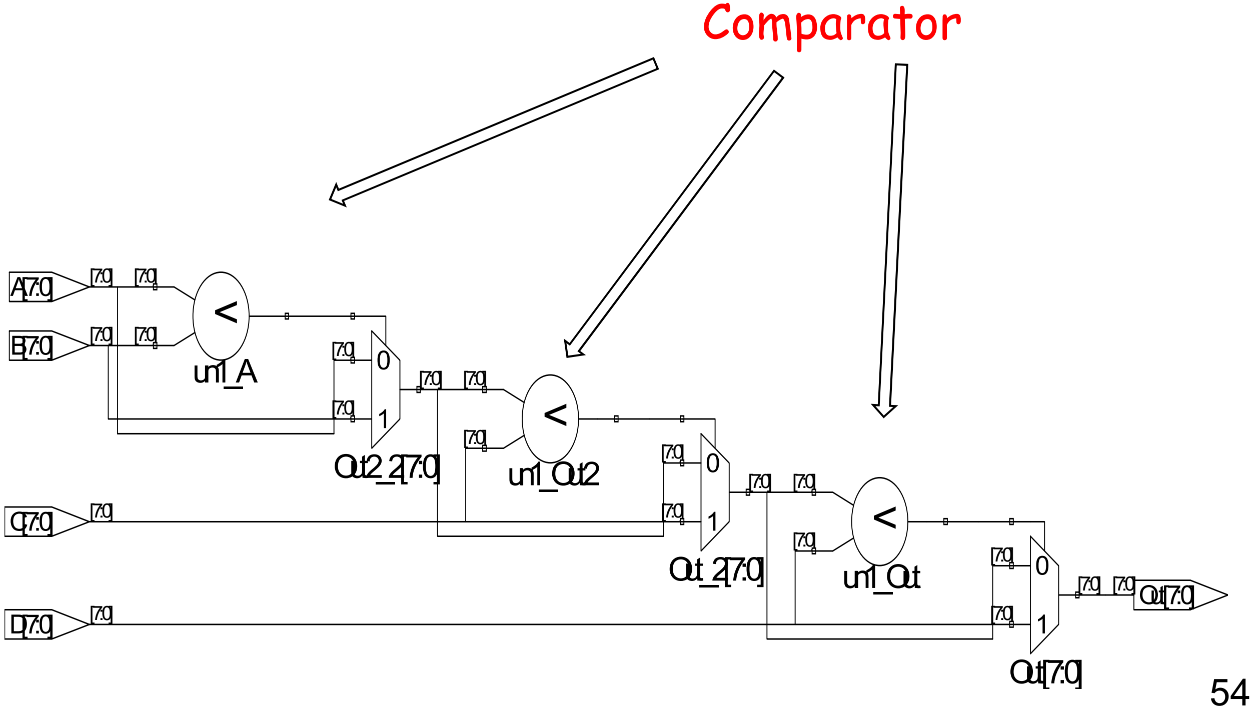


Delay of Comparator

Decide the biggest value among A, B, C, and D.

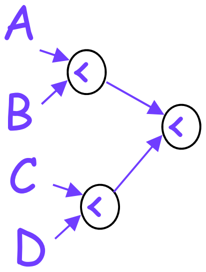
```
Method_1 ( (3 stages):
always @(A or B or C or D)
   begin
      if(A >= B)
         Out1=A;
      else
         Out1=B;
      if(Out1 >= C)
         Out2=Out1;
      else
         Out2=C;
      if(Out2 >= D)
         Out=Out2;
      else
         Out=D;
   end
```





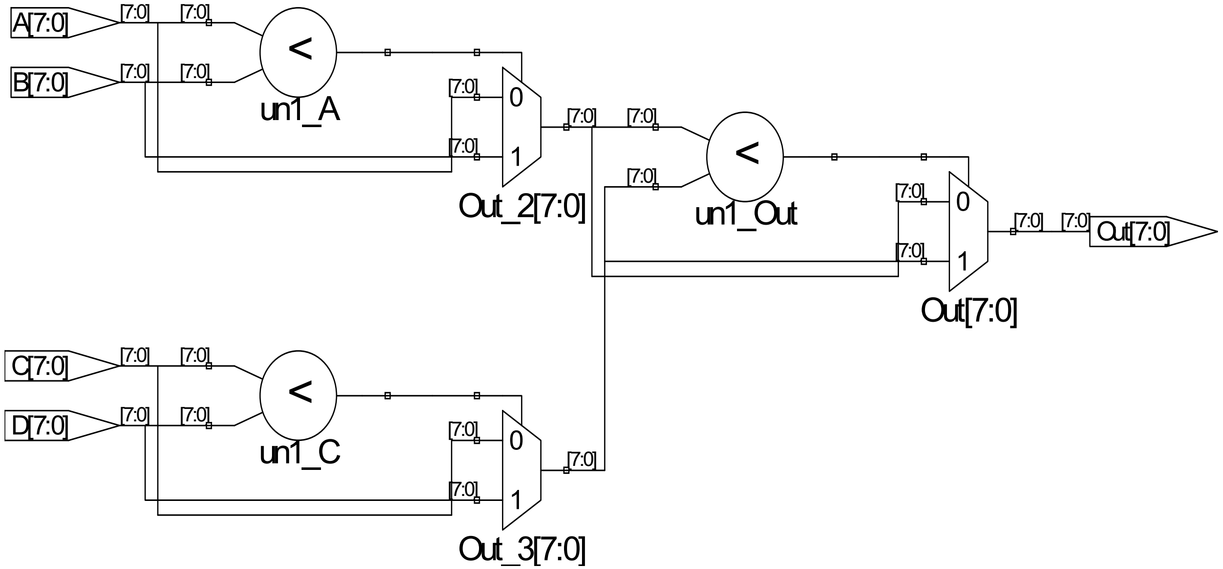
Delay of Comparator

Decide the biggest value among A, B, C, and D.

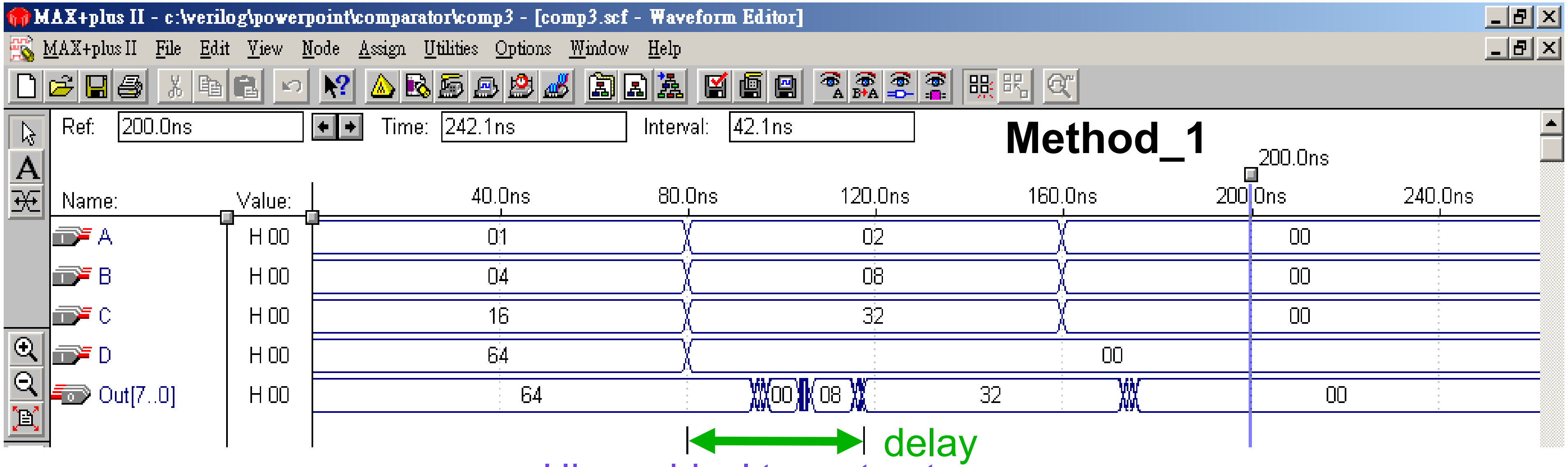


Method_2: (hierarchical tree structure, 2 stage)

```
always @(A or B or C or D)
  begin
    if(A >= B)
      Out1=A;
    else
      Out1=B;
    if(C >= D)
      Out2=C;
    else
      Out2=D;

   if(Out1 >= Out2)
      Out=Out1;
    else
      Out=Out2;
  end
```



Delay of Comparator



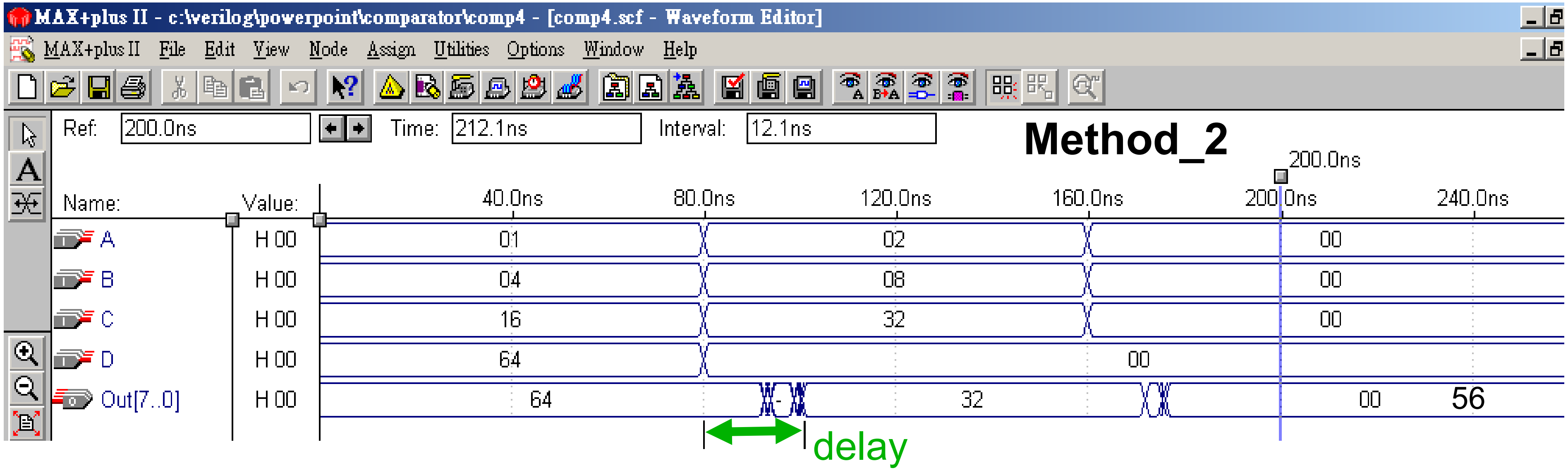
MAX+plus II - c:\verilog\powerpoint\comparator\comp3 - [comp3.scf - Waveform Editor]

Ref: 200.0ns Time: 242.1ns Interval: 42.1ns

Method_1

MAX+plus II File Edit View Node Assign Utilities Options Window Help
200.0ns
Name: Value:
40.0ns 80.0ns 120.0ns 160.0ns 200.0ns 240.0ns
A H 00 01 02 00
B H 00 04 08 00
C H 00 16 32 00
D H 00 64 00
Out[7..0] H 00 64 00 08 32 00
delay

Hierarchical tree structure



MAX+plus II - c:\verilog\powerpoint\comparator\comp4 - [comp4.scf - Waveform Editor]

Ref: 200.0ns Time: 212.1ns Interval: 12.1ns

Method_2

MAX+plus II File Edit View Node Assign Utilities Options Window Help
200.0ns
Name: Value:
40.0ns 80.0ns 120.0ns 160.0ns 200.0ns 240.0ns
A H 00 01 02 00
B H 00 04 08 00
C H 00 16 32 00
D H 00 64 00
Out[7..0] H 00 64 32 00
delay

Arithmetic Logic Unit (1/2)

S4	S3	S2	S1	S0	Cin	Operation	Function	Implementation
0	0	0	0	0	0	Y <= A	Transfer A	Arithmetic Unit
0	0	0	0	0	1	Y <= A + 1	Increment A	Arithmetic Unit
0	0	0	0	1	0	Y <= A + B	Addition	Arithmetic Unit
0	0	0	0	1	1	Y <= A + B + 1	Add with carry	Arithmetic Unit
0	0	0	1	0	0	Y <= A + Bbar	A plus 1's complement of B	Arithmetic Unit
0	0	0	1	0	1	Y <= A + Bbar + 1	Subtraction	Arithmetic Unit
0	0	0	1	1	0	Y <= A - 1	Decrement A	Arithmetic Unit
0	0	0	1	1	1	Y <= A	Transfer A	Arithmetic Unit
0	0	1	0	0	0	Y <= A and B	AND	Logic Unit
0	0	1	0	1	0	Y <= A or B	OR	Logic Unit
0	0	1	1	0	0	Y <= A xor B	XOR	Logic Unit
0	0	1	1	1	0	Y <= Abar	Complement A	Logic Unit
0	0	0	0	0	0	Y <= A	Transfer A	Shifter Unit
0	1	0	0	0	0	Y <= shl A	Shift left A	Shifter Unit
1	0	0	0	0	0	Y <= shr A	Shift right A	Shifter Unit
1	1	0	0	0	0	Y <= 0	Transfer 0's	Shifter Unit

Arithmetic Logic Unit (2/2)

```
module alu_case2(Sel,CarryIn,A,B,Y);
input  [4:0] Sel;
input  CarryIn;
input  [7:0] A,B;
output [7:0] Y;
reg    [7:0] Y;
```

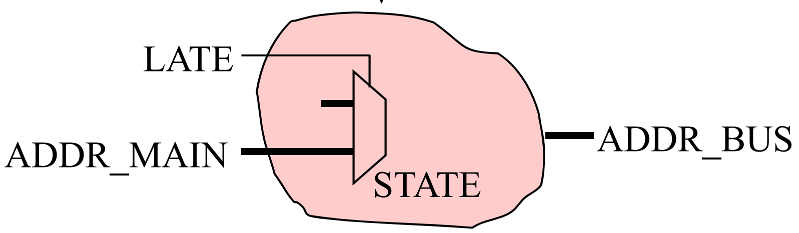
```
always@(Sel or A or B or CarryIn)
begin
    case({Sel[4:0],CarryIn})
        6'b000000 : Y = A;
        6'b000001 : Y = A + 1;
        6'b000010 : Y = A + B;
        6'b000011 : Y = A + B + 1;
        6'b000100 : Y = A + !B;
        6'b000101 : Y = A + !B + 1;
        6'b000110 : Y = A - 1;
        6'b000111 : Y = A;
        6'b001000 : Y = A & B;
        6'b001010 : Y = A | B;
        6'b001100 : Y = A ^ B;
        6'b001110 : Y = !A;
        6'b010000 : Y = A << 1;
        6'b100000 : Y = A >> 1;
        6'b110000 : Y = 0;
        default: Y = 8'bX;
    endcase
end
endmodule
```

Example for IF (1/4)

- Good style takes advantage of "if-else" priority to synthesize correct logic

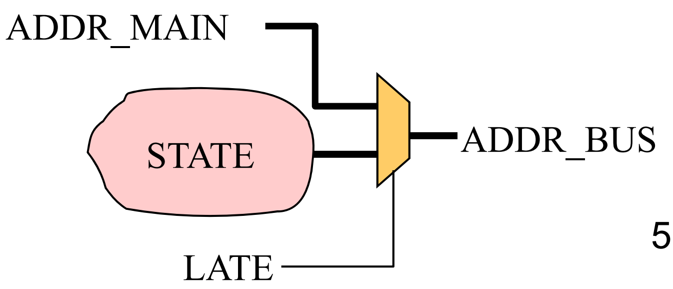
Bad

```
case (STATE)
   IDLE:
      if (LATE == 1'b1)
                 ADDR_BUS <= ADDR_MAIN;
      else
                 ADDR_BUS <= ADDR_CNTL;
   INTERRUPT:
      if (LATE == 1'b1)
                 ADDR_BUS <= ADDR_MAIN;
      else
                 ADDR_BUS <= ADDR_INT;
.....
```



Good

```
if (LATE == 1'b1)
   ADDR_BUS <= ADDR_MAIN;
else
   case (STATE)
      IDLE:
                 ADDR_BUS <= ADDR_CNTL;
      INTERRUPT:
                 ADDR_BUS <= ADDR_INT;
.....
```



Example for IF (2/4)

Bad Style

```
module style_bad(In_Data, State,
Out_Data, En);
input  En;
input  [1:0] State;
input  [2:0] In_Data;
output [3:0] Out_Data;
reg          [3:0] Out_Data;
parameter A1=0, A2=1, A3=2, A4=3;

always @(In_Data or State or En)
begin
    case(State)
     A1:
     begin
        if(En)
        Out_Data = In_Data;
        else
        Out_Data = In_Data - 1;
        end
 A2:
    begin
         if(En)
          Out_Data = In_Data;
         else
          Out_Data = In_Data + 1;
   end
A3:
    begin
         if(En)
          Out_Data = In_Data;
         else
          Out_Data = In_Data - 2;
   end
A4:
    begin
          if(En)
          Out_Data = In_Data;
         else
          Out_Data = In_Data + 2;
   end  endcase end  endmodule
```

Example for IF (3/4)

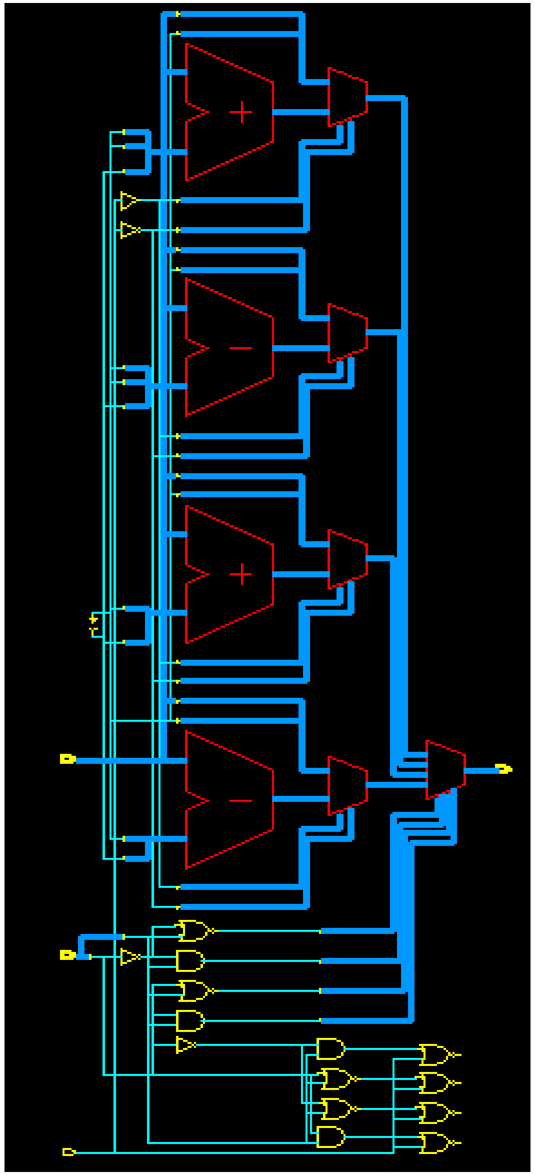
Good Style

```
module style_good(In_Data, State, Out_Data, En);
input  En;
input  [1:0] State;
input  [2:0] In_Data;
output [3:0] Out_Data;
reg   [3:0] Out_Data;

parameter A1=0, A2=1, A3=2, A4=3;

always @(In_Data or State or En)
begin
        if(En)
          Out_Data = In_Data;
        else
          begin
```

```
case(State)
        A1:  Out_Data = In_Data - 1;
        A2:  Out_Data = In_Data + 1;
        A3:  Out_Data = In_Data - 2;
        A4:  Out_Data = In_Data + 2;
                endcase
          end
end
endmodule
```

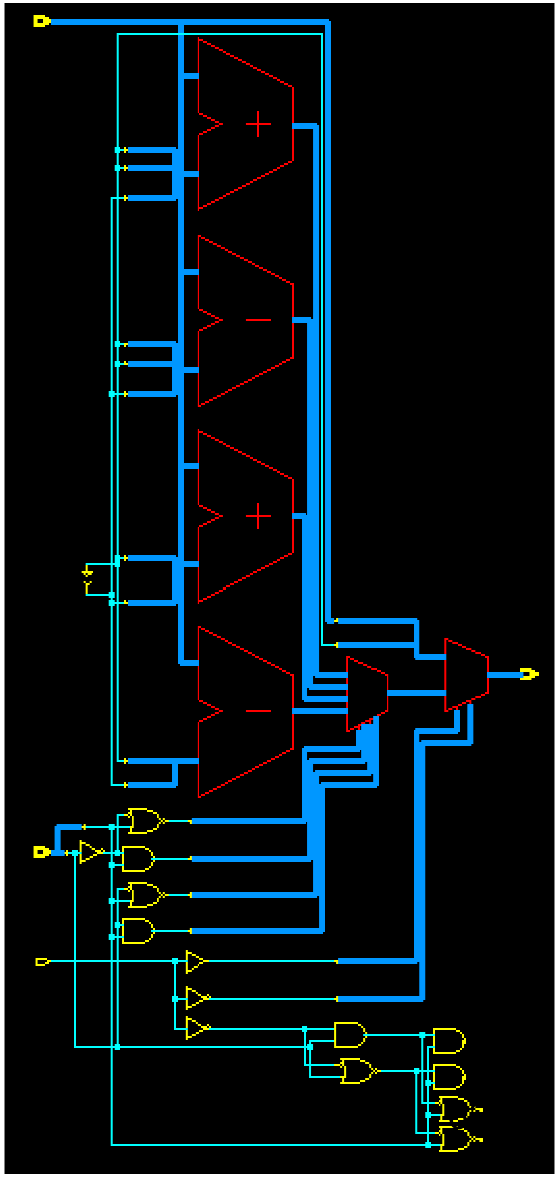
Example for IF (4/4)

Bad Style



Good Style

In_Data



Summary

- We studied the following circuits
  - Decoder
  - Encoder
  - Multiplexer
  - Demultiplexer
  - Comparator
  - ALU

Backup Slides