Verilog HDL

Design Examples

1. Digital system designs and practices: Using Verilog HDL and FPGAs
2. Fundamentals of Digital Logic with Verilog Design 3e

Teacher: Prof. Shu-Yen Lin
Email: sylin@saturn.yzu.edu.tw
Date: 2020/11/11

Outline

Combinational Logic Modules

- Decoders
- Encoders
- Multiplexers
- Demultiplexers

Sequential Logic Modules

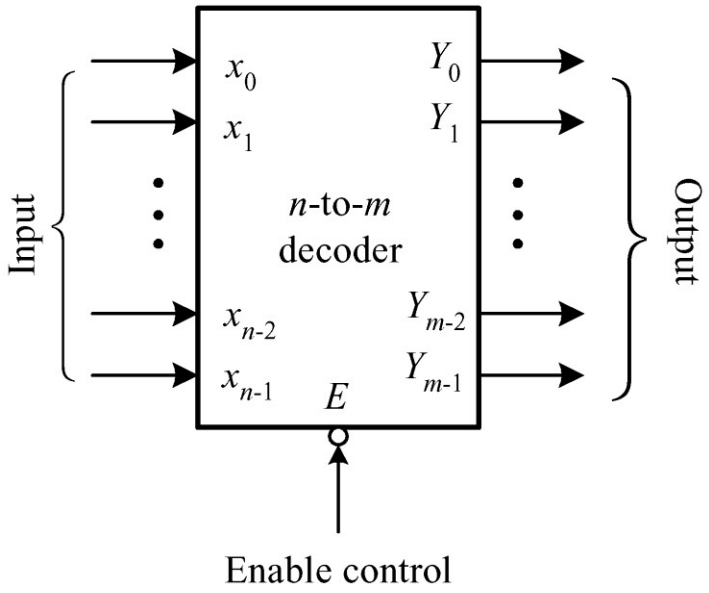
- Flip-flops
- Memory elements
- Data Registers
- Counters
- Finite State Machines (FSM)

Combinational Logic Modules

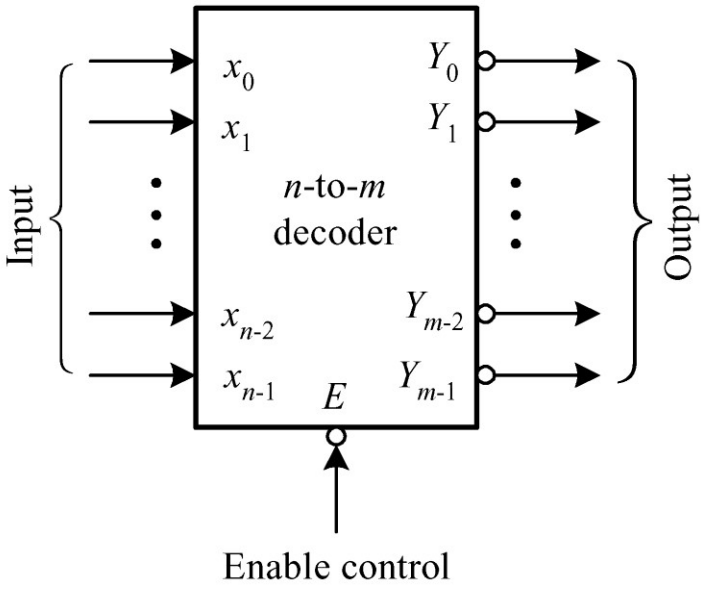
- Decoders
- Encoders
- Multiplexers
- Demultiplexers

Decoder Block Diagrams

- *n*-to-*m* decoders

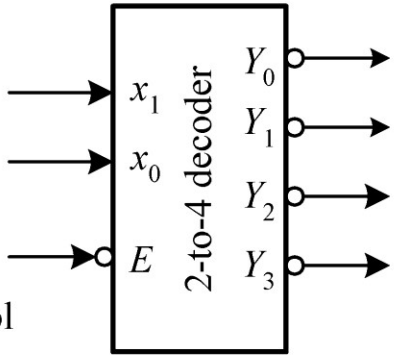


(a) Noninverted output



(b) Inverted output

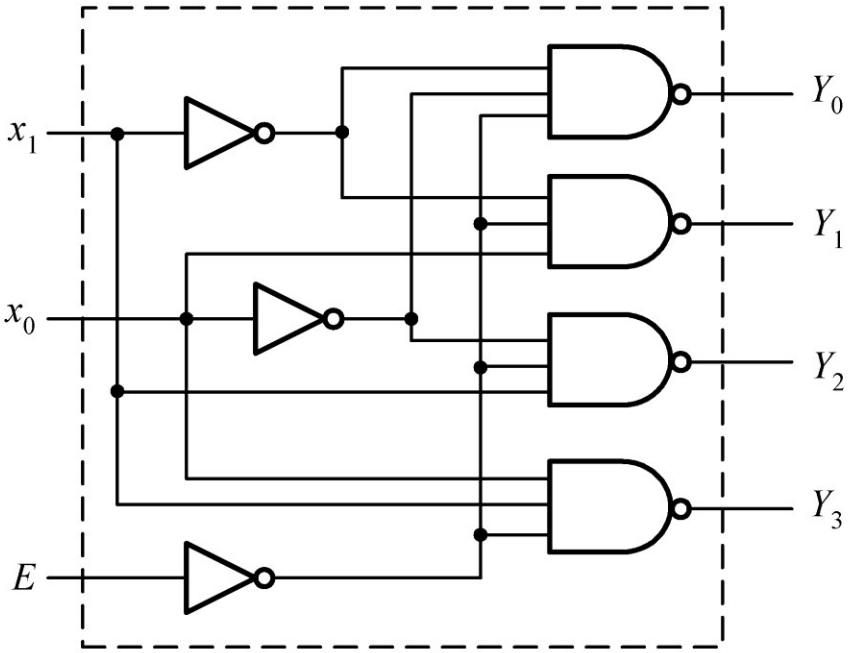
A 2-to-4 Decoder Example



(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

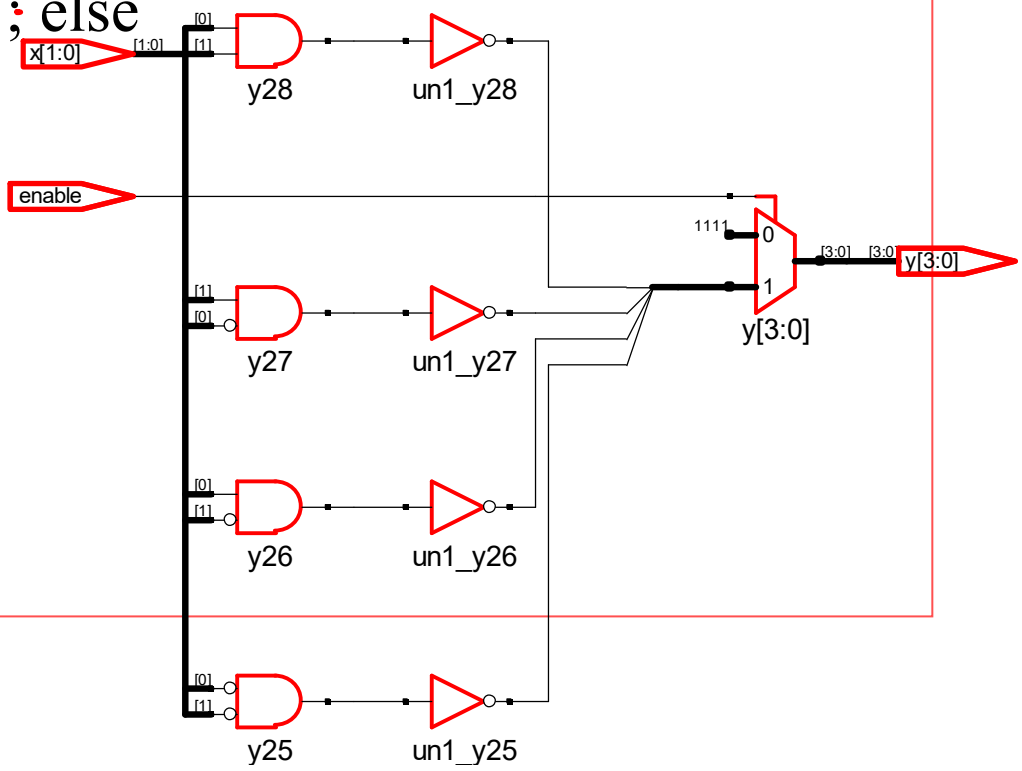
(b) Function table



(c) Logic circuit

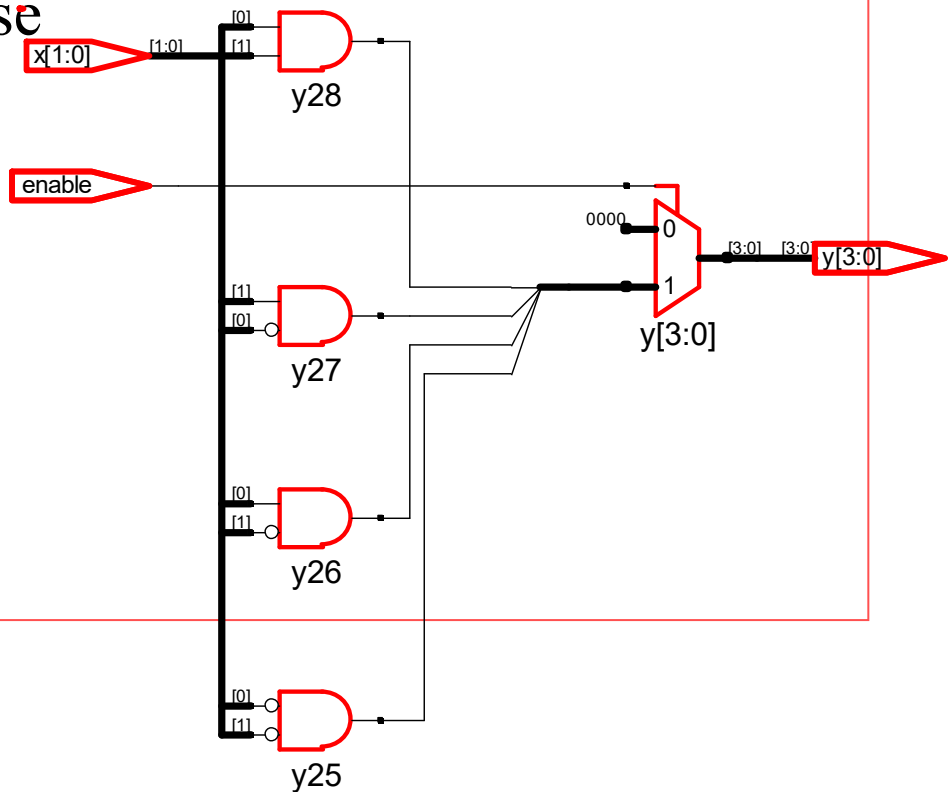
A 2-to-4 Decoder Example

```
// a 2-to-4 decoder with active low output
always @(x or enable_n)
    if (enable_n) 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;
endcase
```



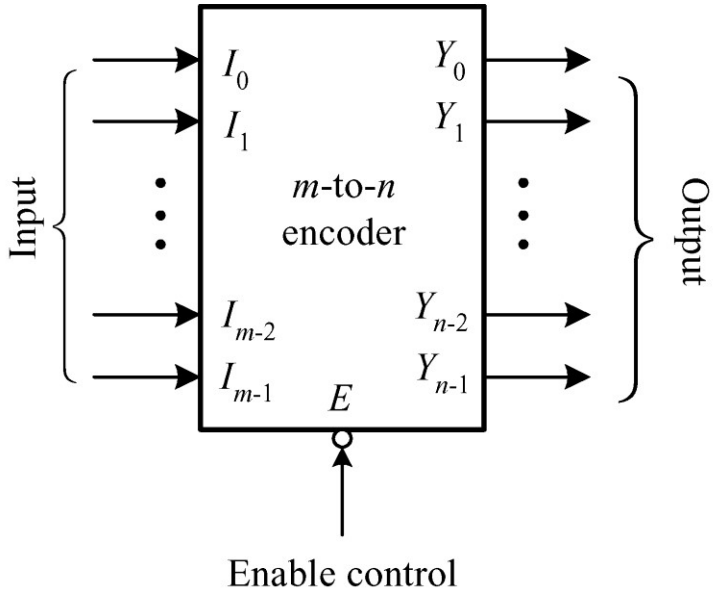
A 2-to-4 Decoder with Enable Control

```
// 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;
endcase
```

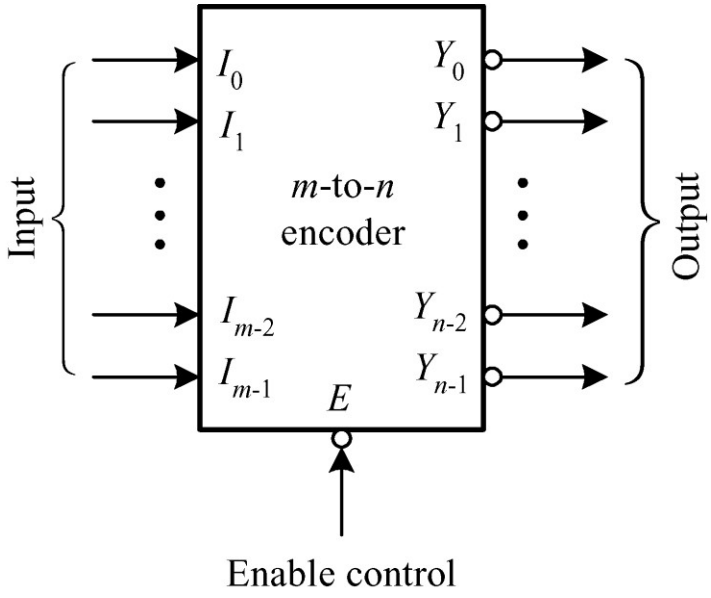


Encoder Block Diagrams

- *m*-to-*n* encoders



(a) Noninverted output

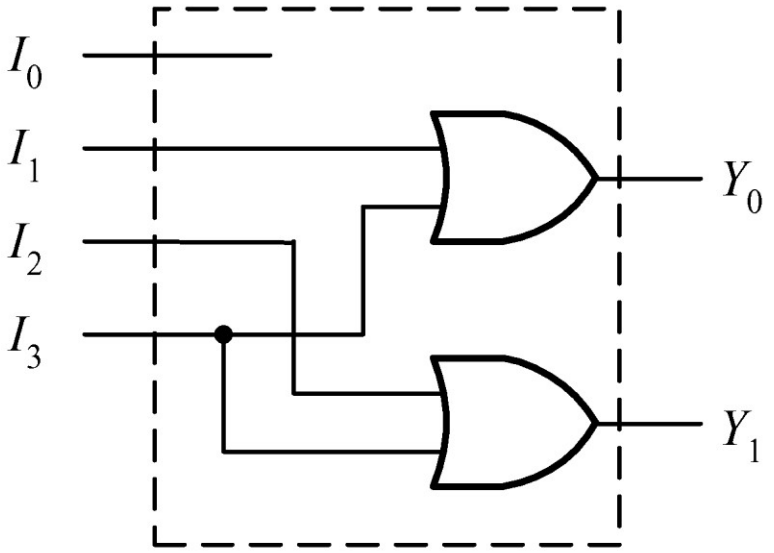


(b) Inverted output

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

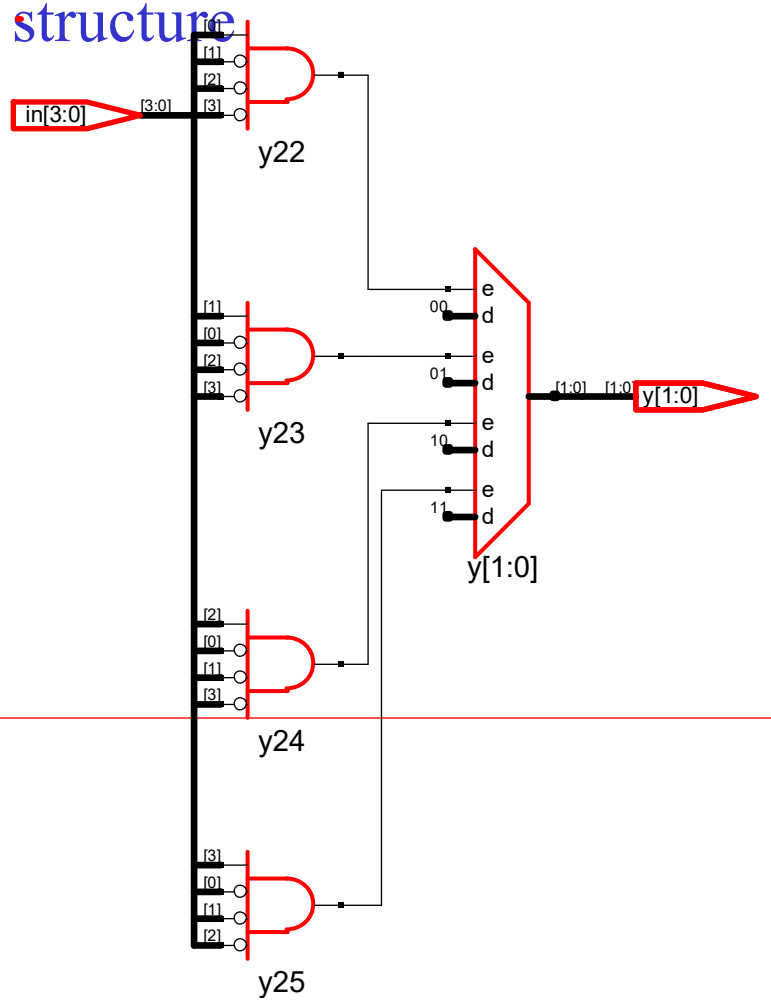
(a) Function table



(b) Logic circuit

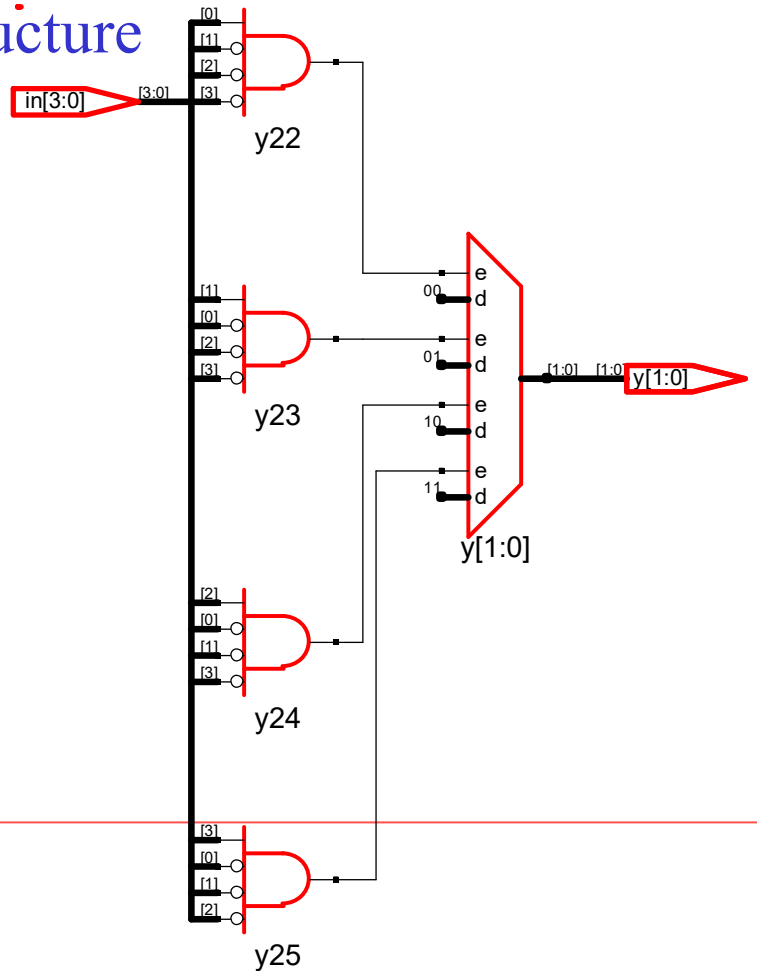
A 4-to-2 Encoder Example

```
// a 4-to-2 encoder using if ... else structure
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
```

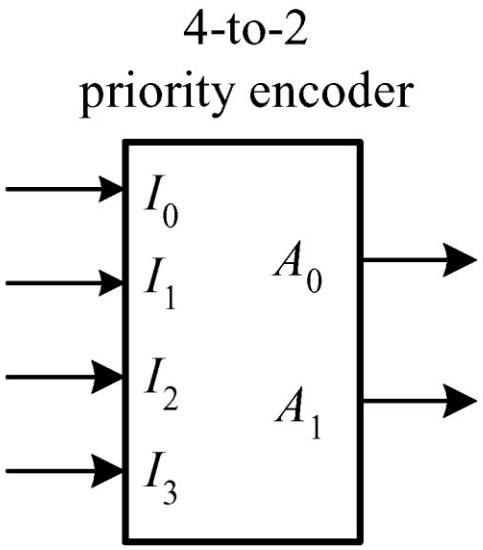


Another 4-to-2 Encoder Example

```
// a 4-to-2 encoder using case structure
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
```



A 4-to-2 Priority Encoder



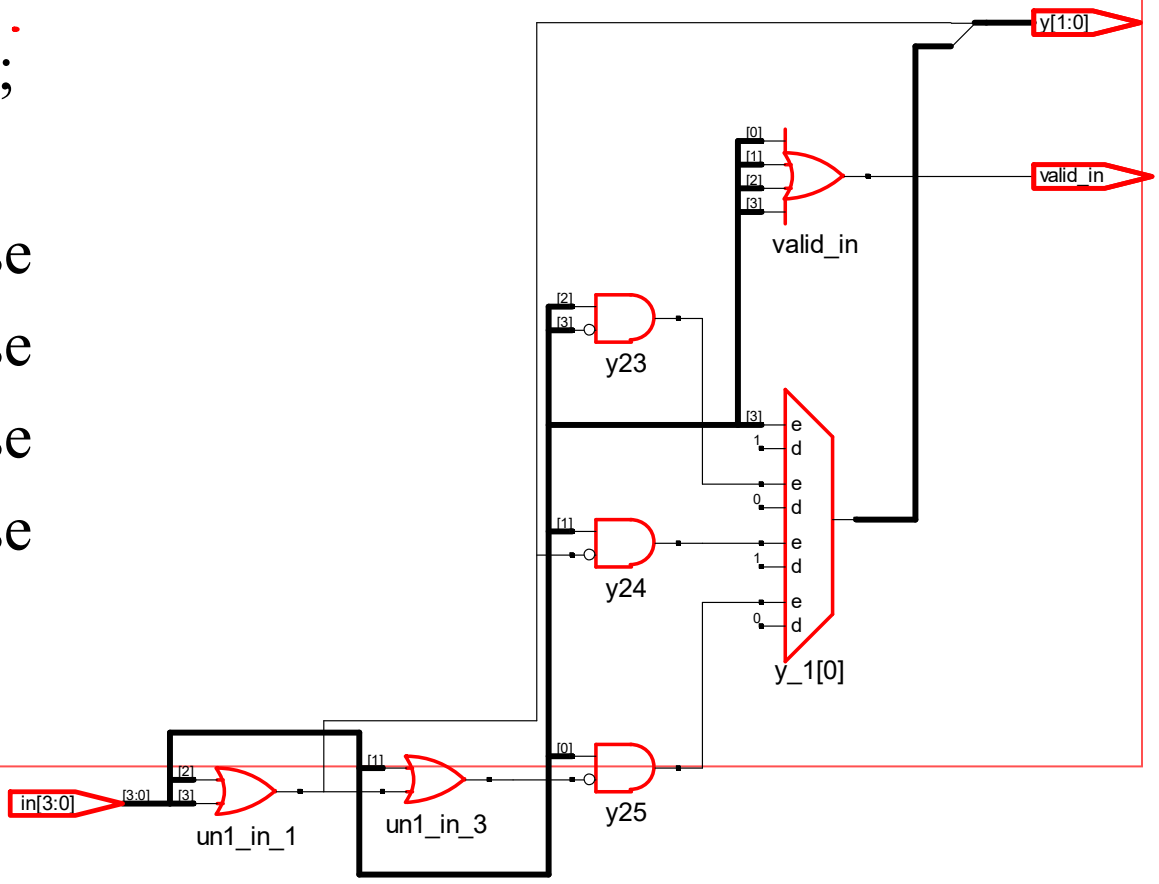
(a) Block diagram

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

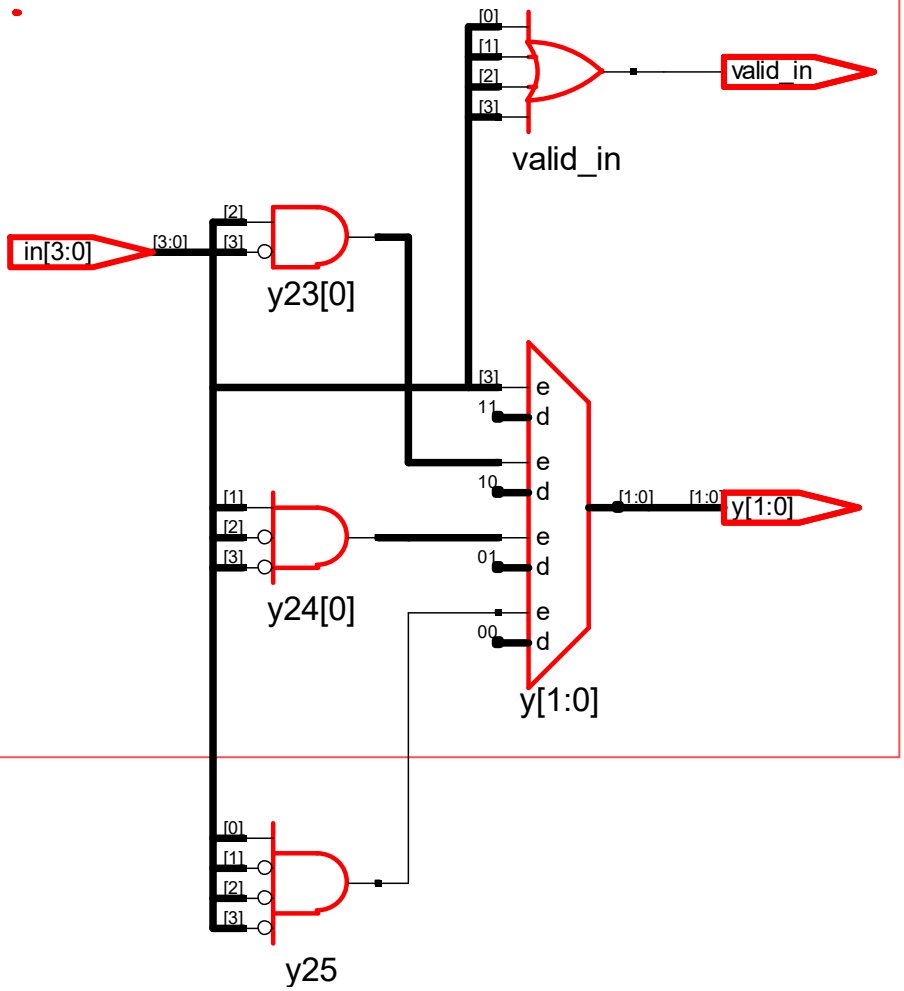
A 4-to-2 Priority Encoder Example

```
// using if ... else structure
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
```



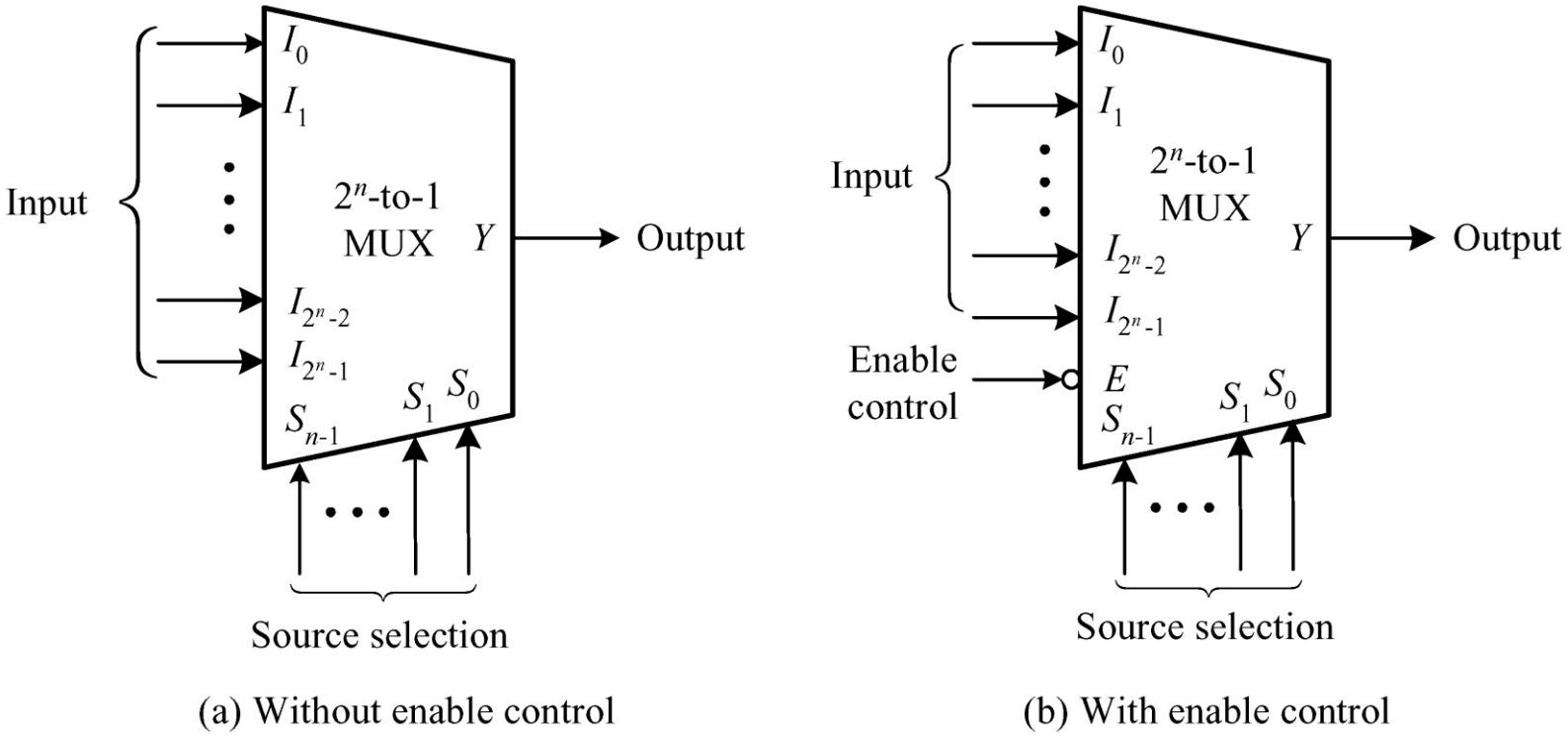
Another 4-to-2 Priority Encoder Example

```
// using casex structure
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
```



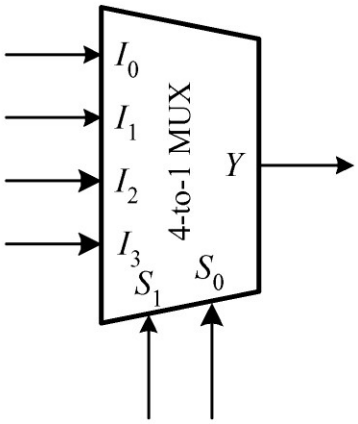
Multiplexer Block Diagrams

- *m*-to-1 ($m = 2^n$) multiplexers



A 4-to-1 Multiplexer Example

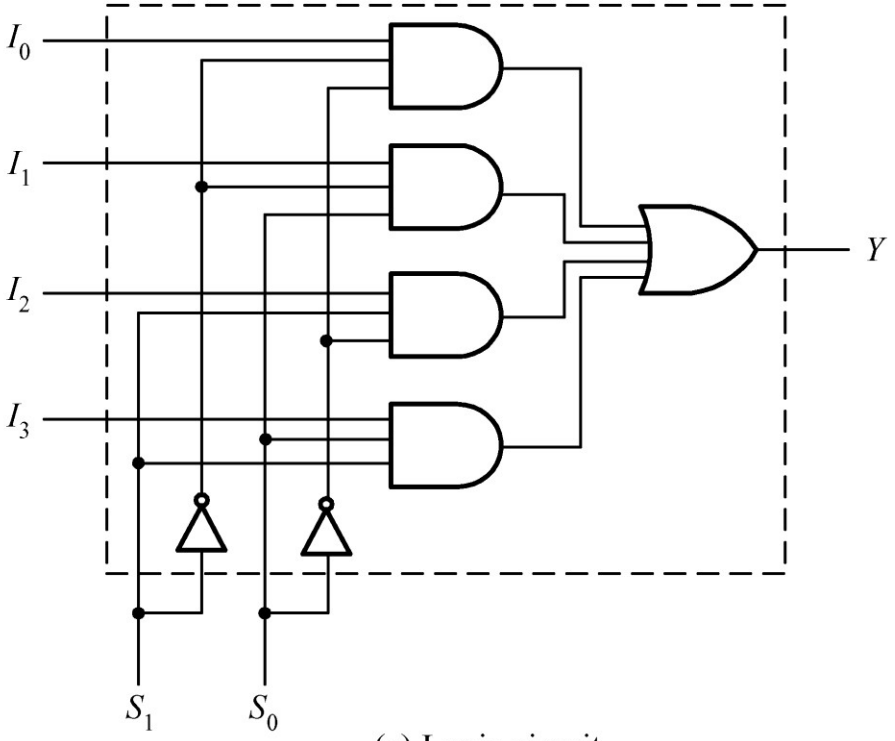
- Gate-based 4-to-1 multiplexers



(a) Logic symbol

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

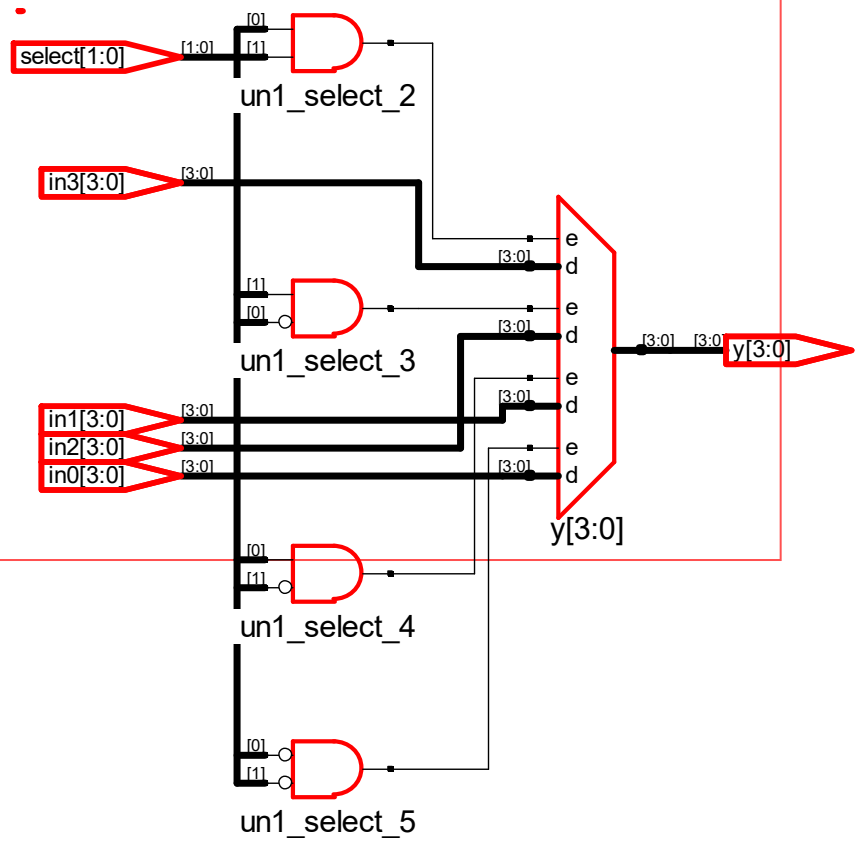
(b) Function table



(c) Logic circuit

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

```
// an N-bit 4-to-1 multiplexer using conditional operator
parameter N = 4; //
input [1:0] select;
input [N-1:0] in3, in2, in1, in0;
output [N-1:0] y;
assign y = select[1] ?
              (select[0] ? in3 : in2) :
              (select[0] ? in1 : in0) ;
```

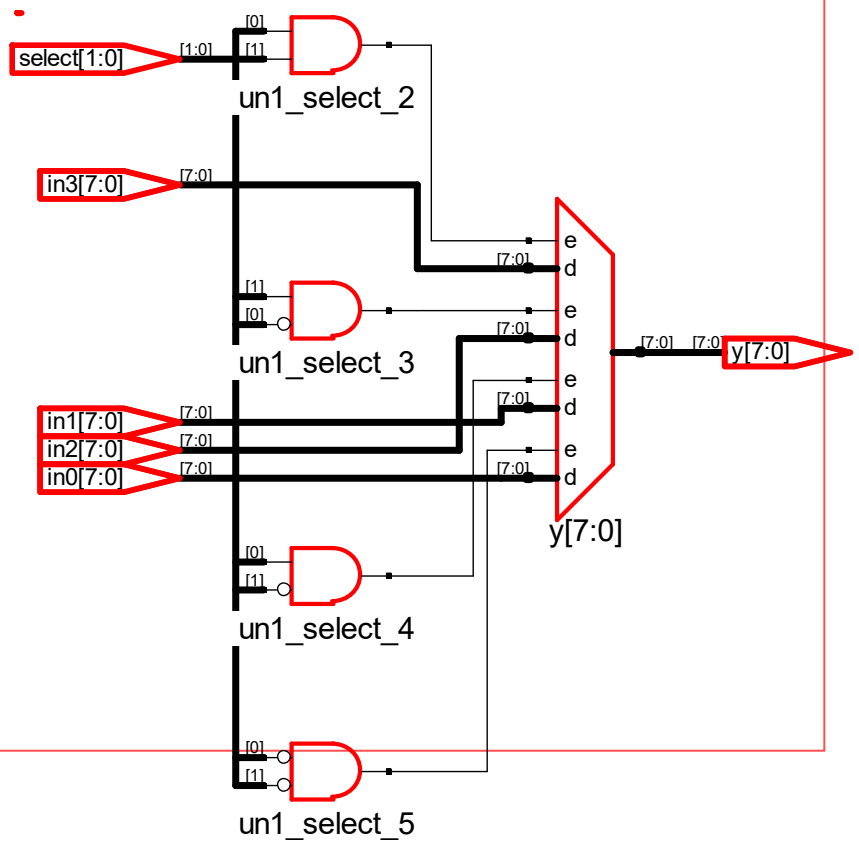


The Second *n*-bit 4-to- 1 Multiplexer Example

```
// an N-bit 4-to-1 multiplexer with enable control
parameter N = 4;
input [1:0] select;
input enable;
input [N-1:0] in3, in2, in1, in0;
output reg [N-1:0] y;

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) ;
```

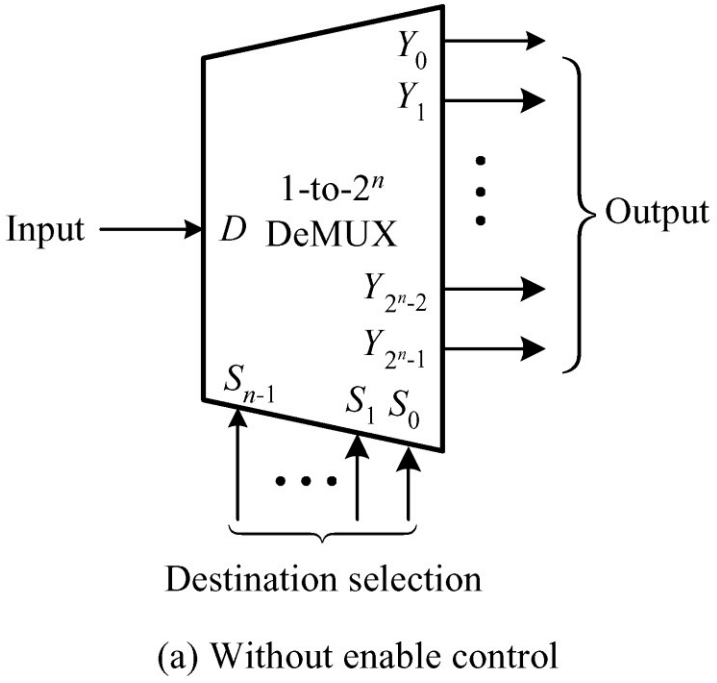
The Third *n*-bit 4-to- 1 Multiplexer Example

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

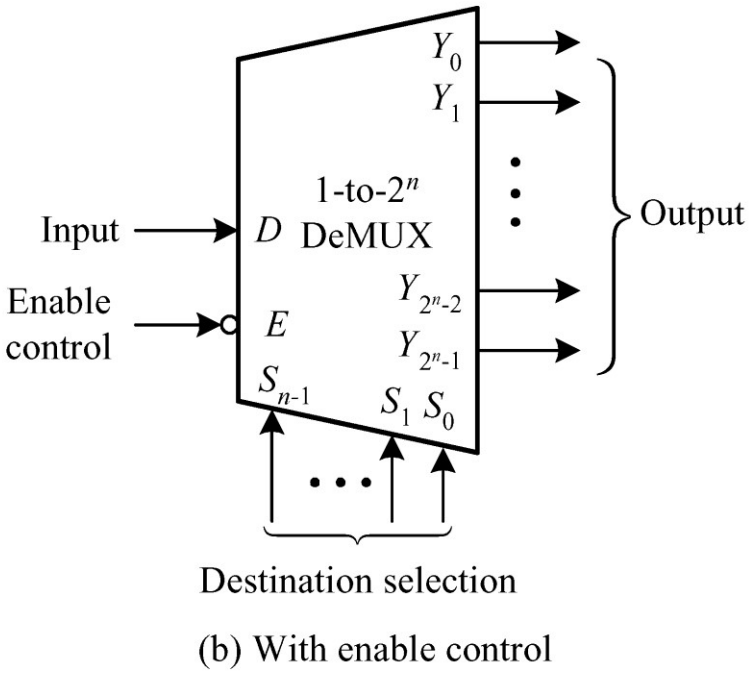


DeMultiplexer Block Diagrams

- 1-to-m ($m = 2^n$) demultiplexers



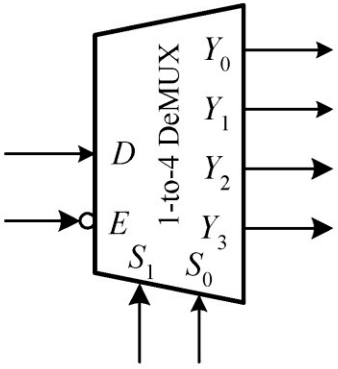
(a) Without enable control



(b) With enable control

A 1-to-4 DeMultiplexer Example

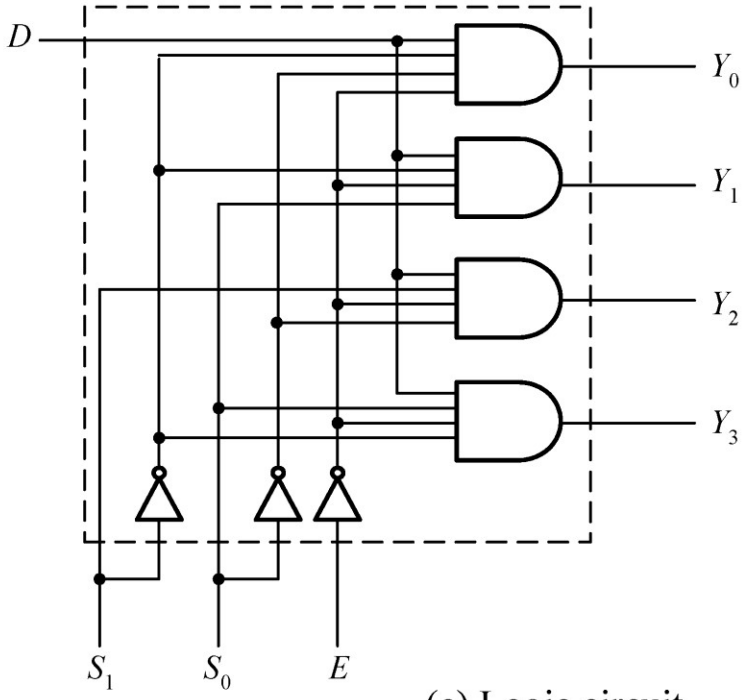
- Gate-based 1-to-4 demultiplexers



(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

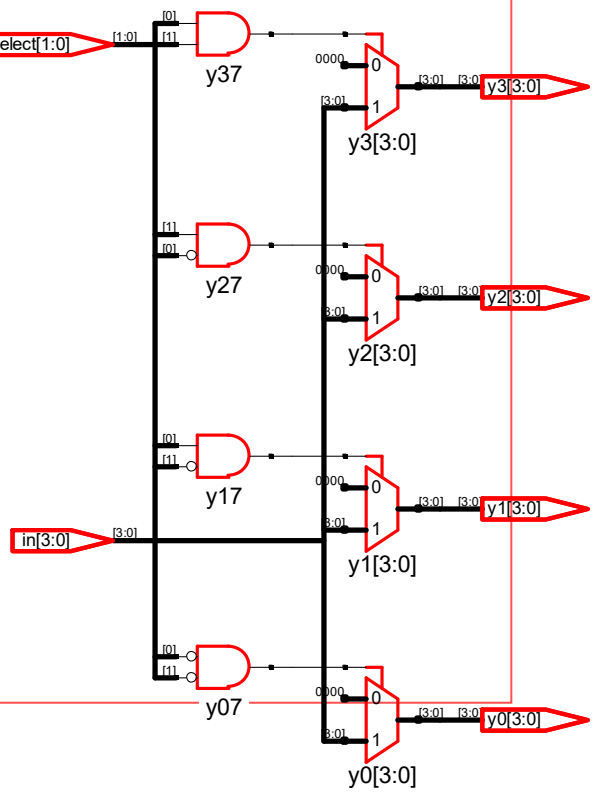
(b) Function table



(c) Logic circuit

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

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

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
```



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

```
// an N-bit 1-to-4 demultiplexer with enable control
parameter N = 4;      // Default width
…
output reg [N-1:0] y3, y2, y1, y0;
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
```

Sequential Logic Modules

- Flip-flops
- Data Registers
- Shift registers
- Counters

Asynchronous Reset *D*-Type Flip-Flops

```
// asynchronous reset D-type flip-flop
module DFF_async_reset (clk, reset_n, d, q);
…
output reg q;

always @(posedge clk or negedge reset_n)
    if (!reset_n) q <= 0;
    else          q <= d;
```

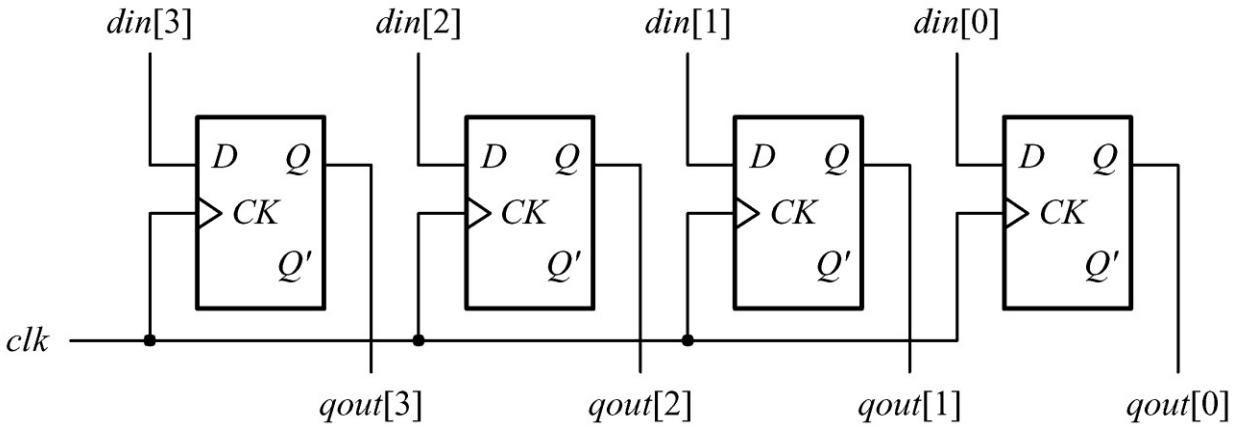
Synchronous Reset *D*-Type Flip-Flops

```
// synchronous reset D-type flip-flop
module DFF_sync_reset (clk, reset, d, q);
…
output reg q;

always @(posedge clk)
    if (reset)  q <= 0;
    else        q <= d;
```

D-type latch

```
module latch (D, clk, Q);
  input D, clk;
  output reg Q;

  always @(D, clk)
    if (clk)
      Q = D;

endmodule
```

Data Registers




```
// an n-bit data register
module register(clk, din, qout);
parameter N = 4;   // number of bits
…
input   [N-1:0] din;
output reg [N-1:0] qout;
    always @(posedge clk) qout <= din;
```

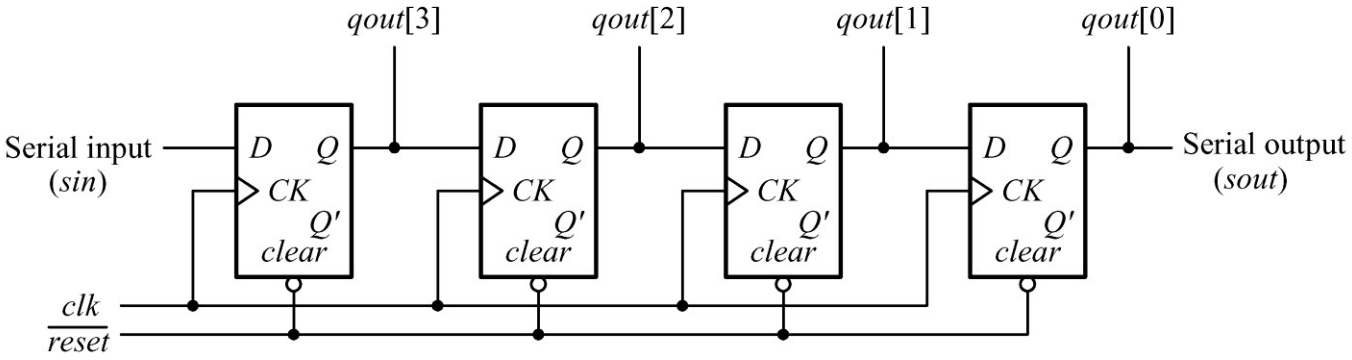
Data Registers

```
// an n-bit data register with asynchronous reset
module register_reset (clk, reset_n, din, qout);
parameter N = 4;   // number of bits
…
input   [N-1:0] din;
output reg [N-1:0] qout;
always @(posedge clk or negedge reset_n)
   if (!reset_n) qout <= {N{1'b0}};
   else          qout <= din;
```

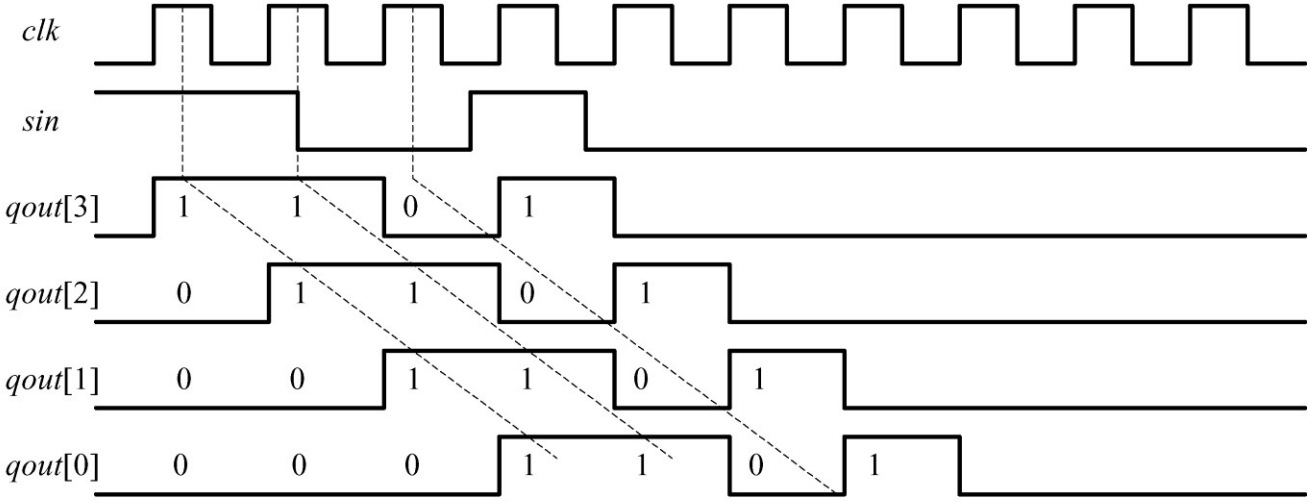
Data Registers

```
// an N-bit data register with synchronous load and
// asynchronous reset
parameter N = 4;   // number of bits
input  clk, load, reset_n;
input  [N-1:0] din;
output reg [N-1:0] qout;

always @(posedge clk or negedge reset_n)
   if (!reset_n)    qout <= {N{1'b0}};
   else  if (load)  qout <= din;
```

Shift Registers



(a) Logic circuit

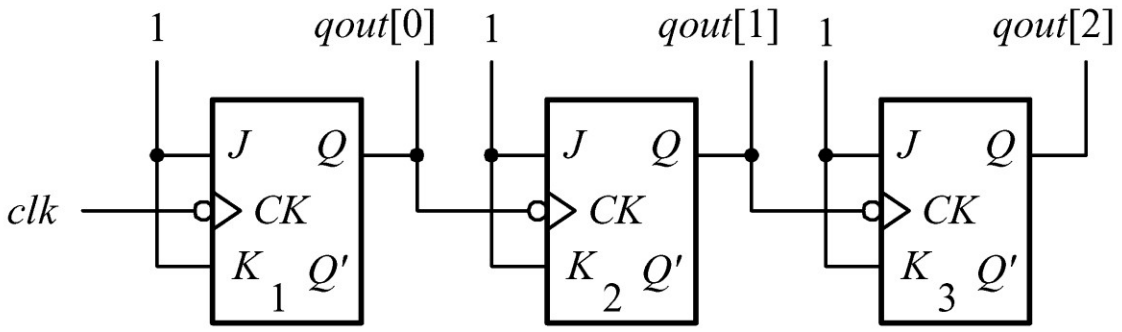


(b) Timing

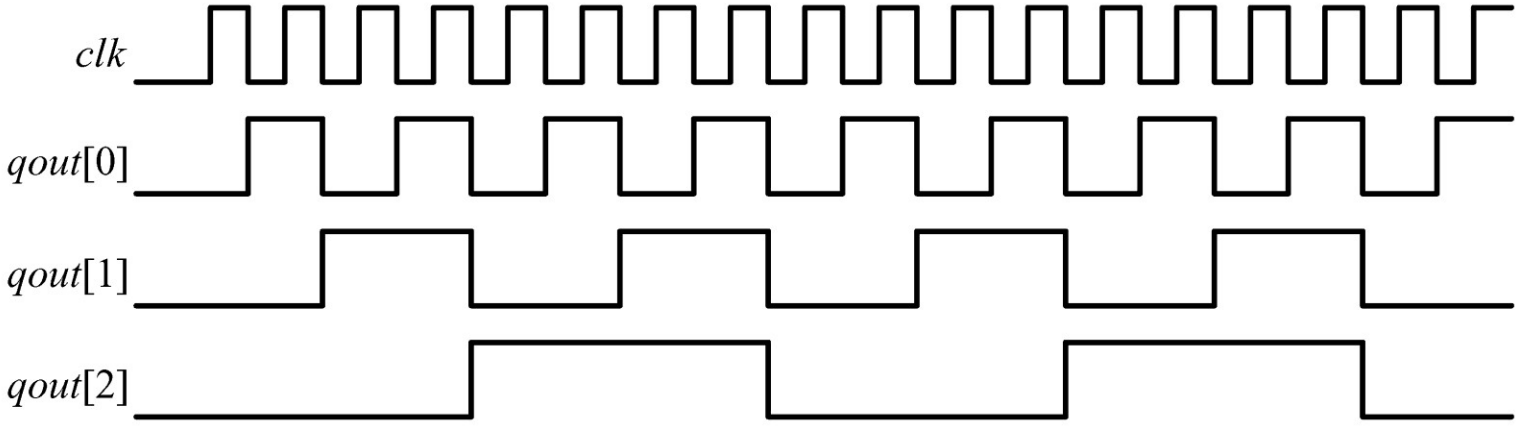
Shift Registers

```
// a shift register module example
module shift_register(clk, reset_n, din, qout);
Parameter N = 4;   // number of bits
….
output reg [N-1:0] qout;

always @(posedge clk or negedge reset_n)
   if (!reset_n) qout <= {N{1'b0}};
   else          qout <= {din, qout[N-1:1]};
```

Binary Ripple Counters



(a) Logic diagram



(b) Timing

Binary Ripple Counters

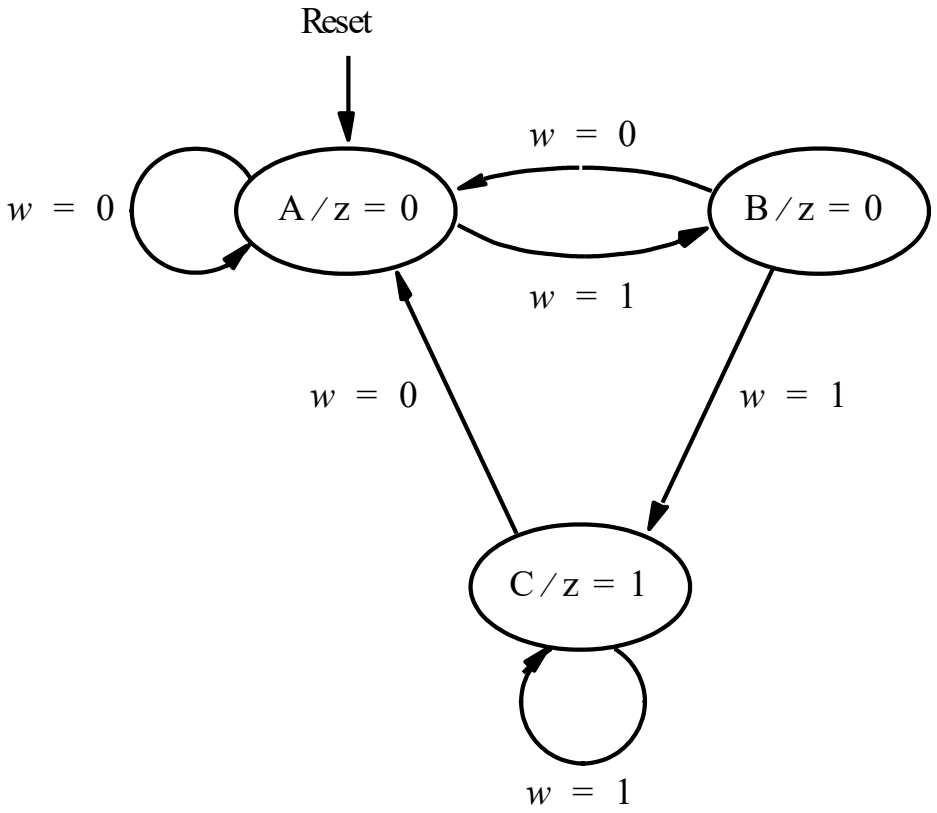
```
// a 3-bit ripple counter module example
module ripple_counter(clk, qout);
…
output reg [2:0] qout;
wire   c0, c1;
// the body of the 3-bit ripple counter
assign c0 = qout[0], c1 = qout[1];
always @(negedge clk)
    qout[0] <= ~qout[0];
always @(negedge c0)
    qout[1] <= ~qout[1];
always @(negedge c1)
    qout[2] <= ~qout[2];
```

Binary Ripple Counters

```
// a 3-bit ripple counter with enable control
module ripple_counter_enable(clk, enable, reset_n, qout);
…
output reg [2:0] qout;
wire   c0, c1;
assign c0 = qout[0], c1 = qout[1];
always @(posedge clk or negedge reset_n)
    if (!reset_n) qout[0] <= 1'b0;
    else if (enable)   qout[0] <= ~qout[0];
always @(posedge c0 or negedge reset_n)
    if (!reset_n) qout[1] <= 1'b0;
    else if (enable)   qout[1] <= ~qout[1];
always @(posedge c1 or negedge reset_n)
    if (!reset_n) qout[2] <= 1'b0;
    else if (enable)   qout[2] <= ~qout[2];
```

Finite State Machines (1)

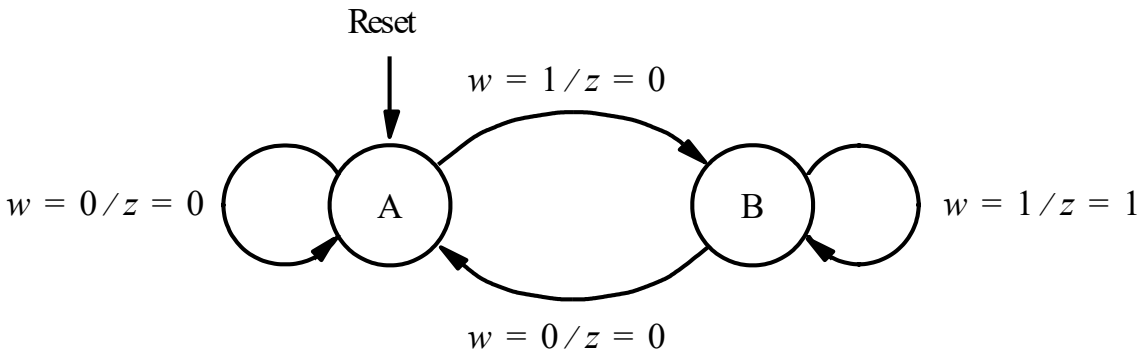
❖**Moore machine:** a finite-state machine whose output values are determined solely by its **current state**.




```
module moore (Clock, w, Resetn, z);
 input Clock, w, Resetn;
 output z;
 reg [1:0] y, Y;
 parameter A = 2'b00, B = 2'b01, C = 2'b10;

 always @(w, y)
 begin
  case (y)
   A: if (w == 0) Y = A;
      else Y = B;
   B: if (w == 0) Y = A;
      else Y = C;
   C: if (w == 0) Y = A;
      else Y = C;
   default: Y = 2'bxx;
  endcase
 end

 always @(posedge Clock, negedge Resetn)
 begin
  if (Resetn == 0)
     y <= A;
  else
     y <= Y;
 end

 assign z = (y == C);

endmodule
```

Finite State Machines (2)

❖**Mealy machine:** A finite-state machine whose output values are determined both by its **current state** and **the current inputs**.




```
module mealy (Clock, w, Resetn, z);
 input Clock, w, Resetn ;
 output reg z ;
 reg y, Y;
 parameter A = 1'b0, B = 1'b1;

 always @(w, y)
  case (y)
   A:  if (w = = 0)
     begin
      Y = A;
      z = 0;
     end
     else
     begin
      Y = B;
      z = 0;
     end
   B:  if (w == 0)
     begin
      Y = A;
      z = 0;
     end
     else
     begin
      Y = B;
      z = 1;
     end
  endcase

 always @(posedge Clock , negedge Resetn)
  if (Resetn = = 0)
     y <= A;
  else
     y <= Y;

endmodule
```