HARDWARE DESCRIPTION LANGUAGE FOR DIGITAL DESIGN

數位設計硬體描述語言

Combinational and Sequential Logic Blocks

Materials partly adapted from "Digital System Designs and Practices Using Verilog HDL and FPGAs," M.B. Lin.

NCKU EE
LPHP Lab

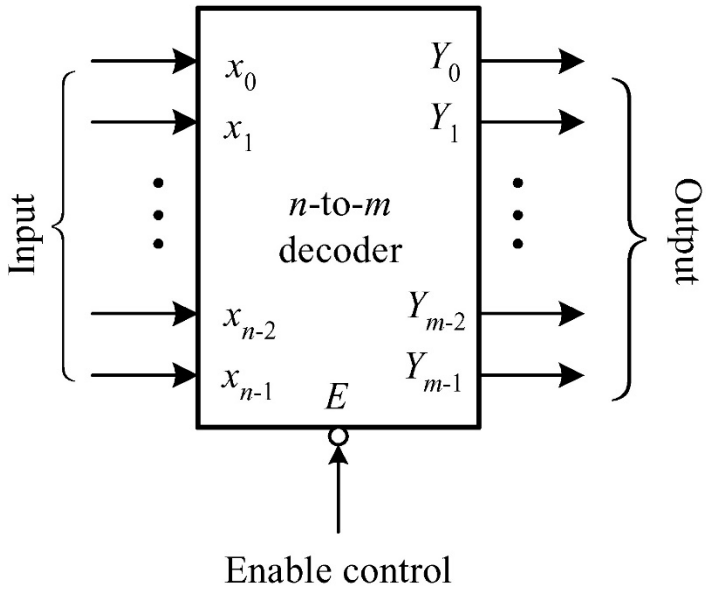
OUTLINE

- Combinational Logic Blocks
- Sequential Logic Blocks

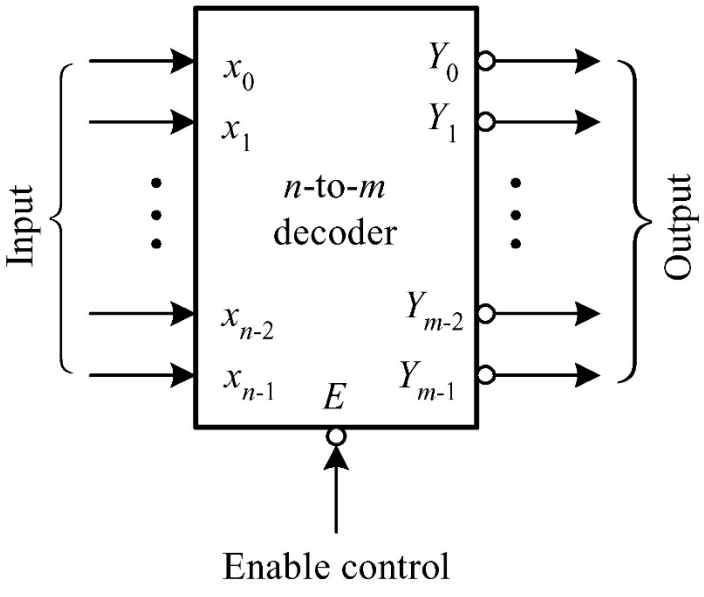
COMBINATIONAL LOGIC BLOCKS

DECODER BLOCK DIAGRAMS

- *n*-to-*m* decoders

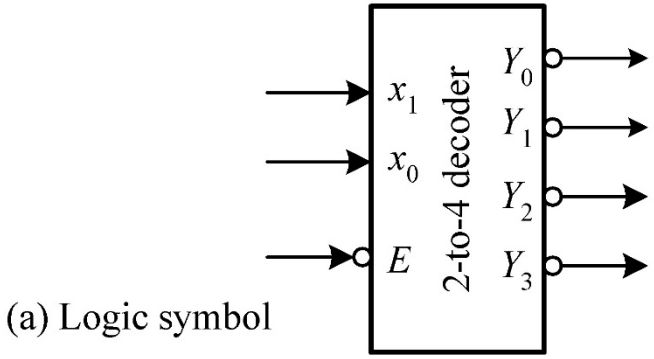


(a) Noninverted output



(b) Inverted output

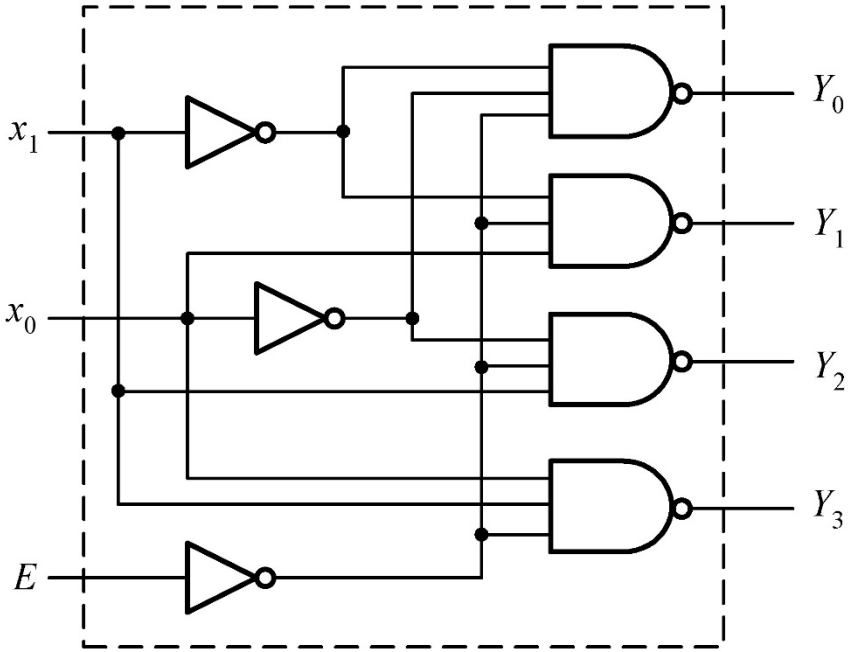
A 2-TO-4 DECODER EXAMPLE



(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

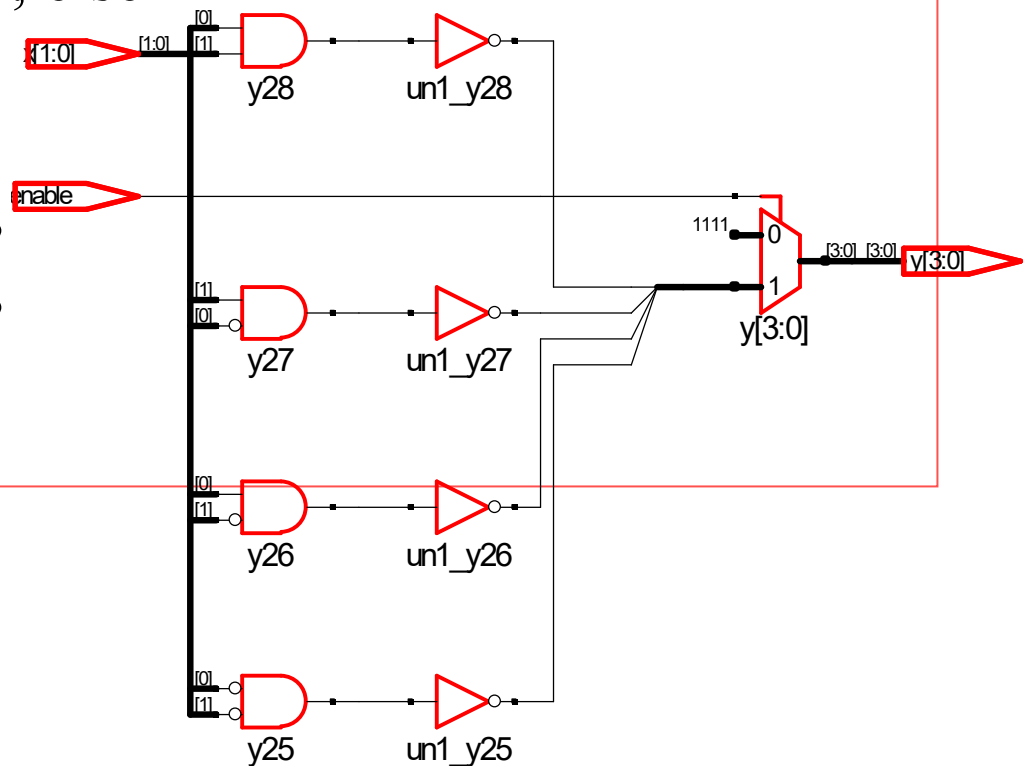
(b) Function table



(c) Logic circuit

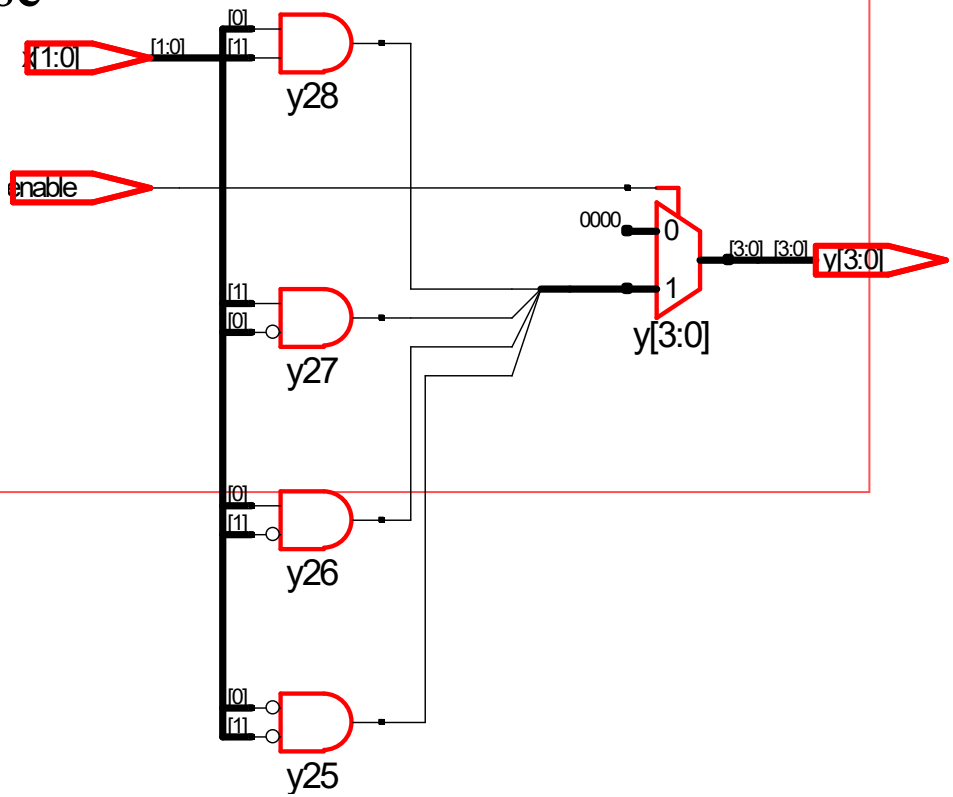
A 2-TO-4 DECODER EXAMPLE

```
// 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
```



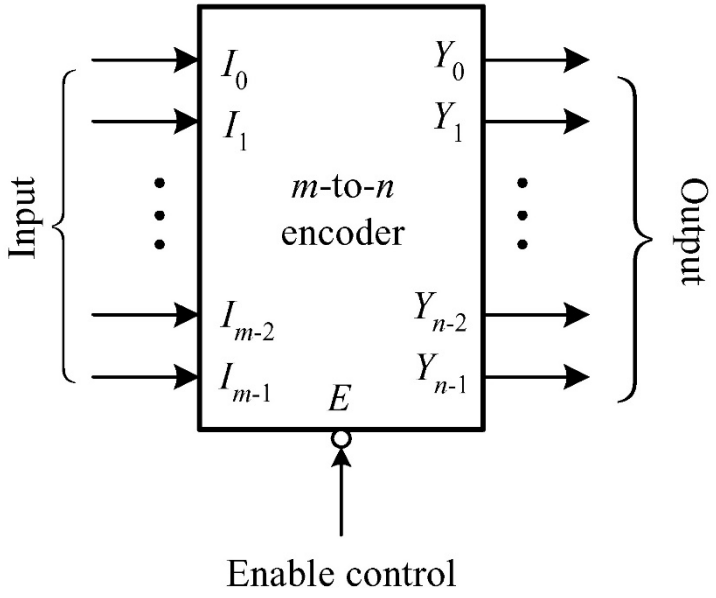
A 2-TO-4 DECODER WITH ENABLE CONTROL

```
// 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
```

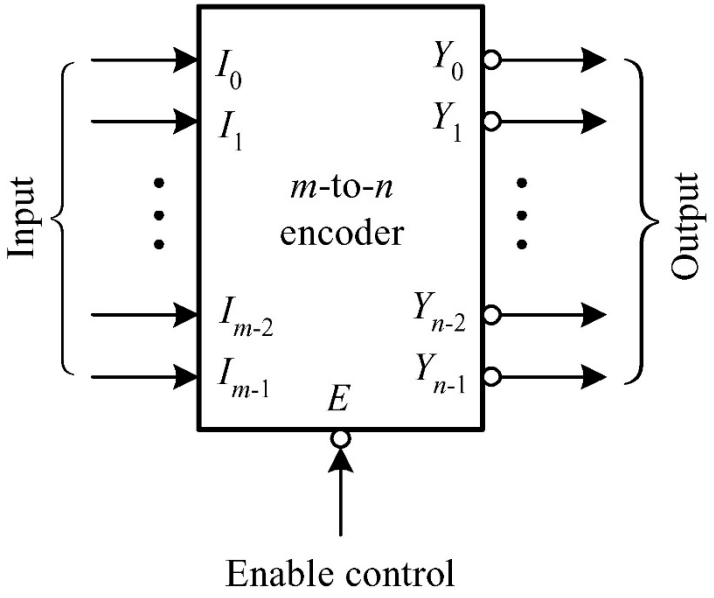


ENCODER BLOCK DIAGRAMS

- *m*-to-*n* encoders



(a) Noninverted output

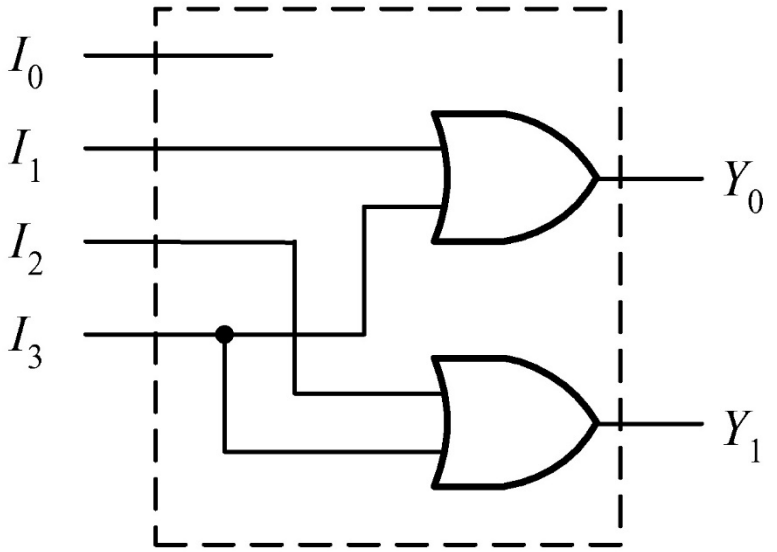


(b) Inverted output

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

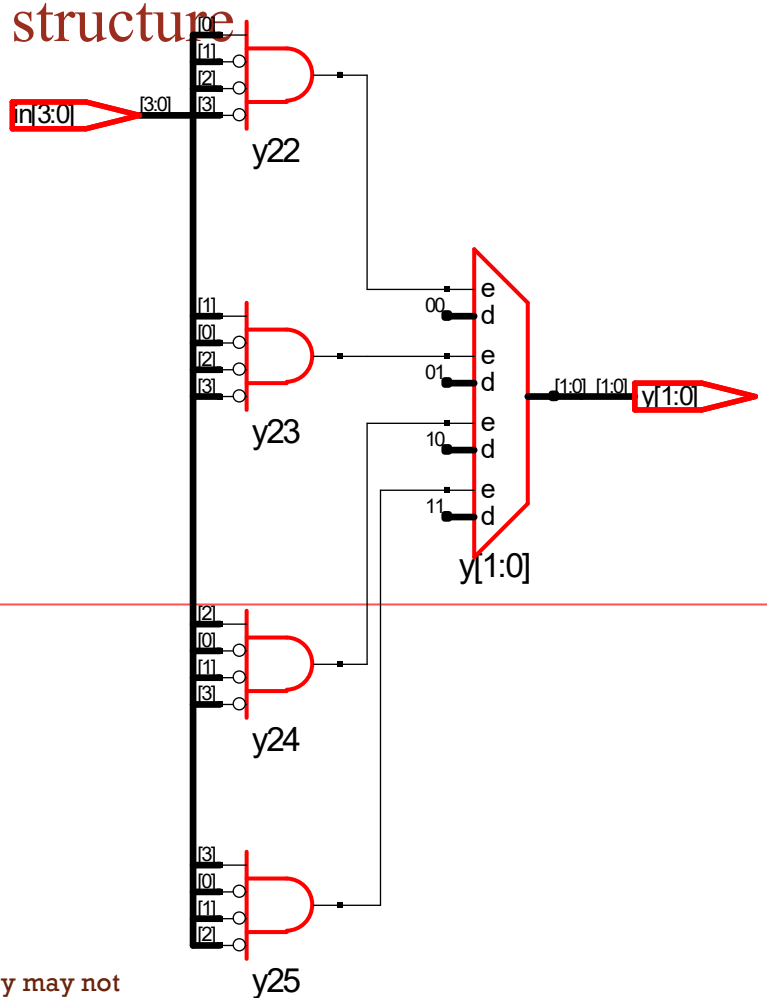


(b) Logic circuit

Q: What is the problem of this encoder?

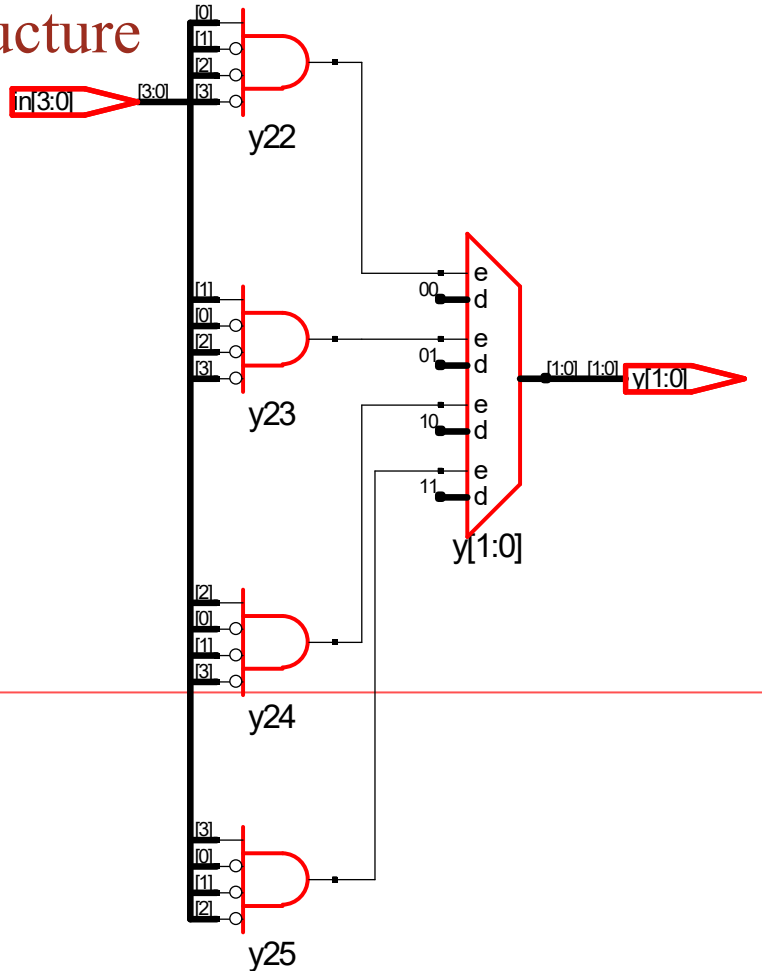
A 4-TO-2 ENCODER EXAMPLE

```
// 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
```

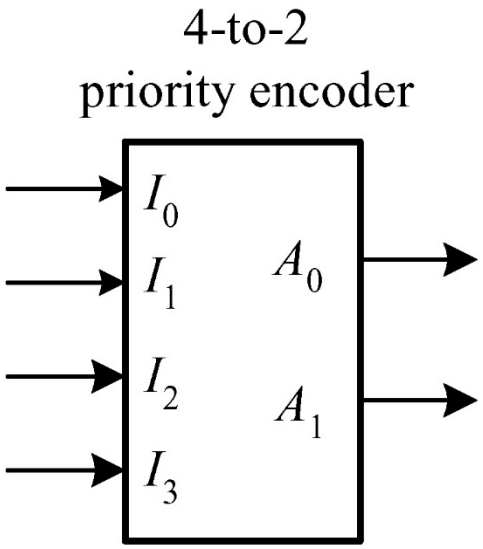


ANOTHER 4-TO-2 ENCODER EXAMPLE

```
// 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



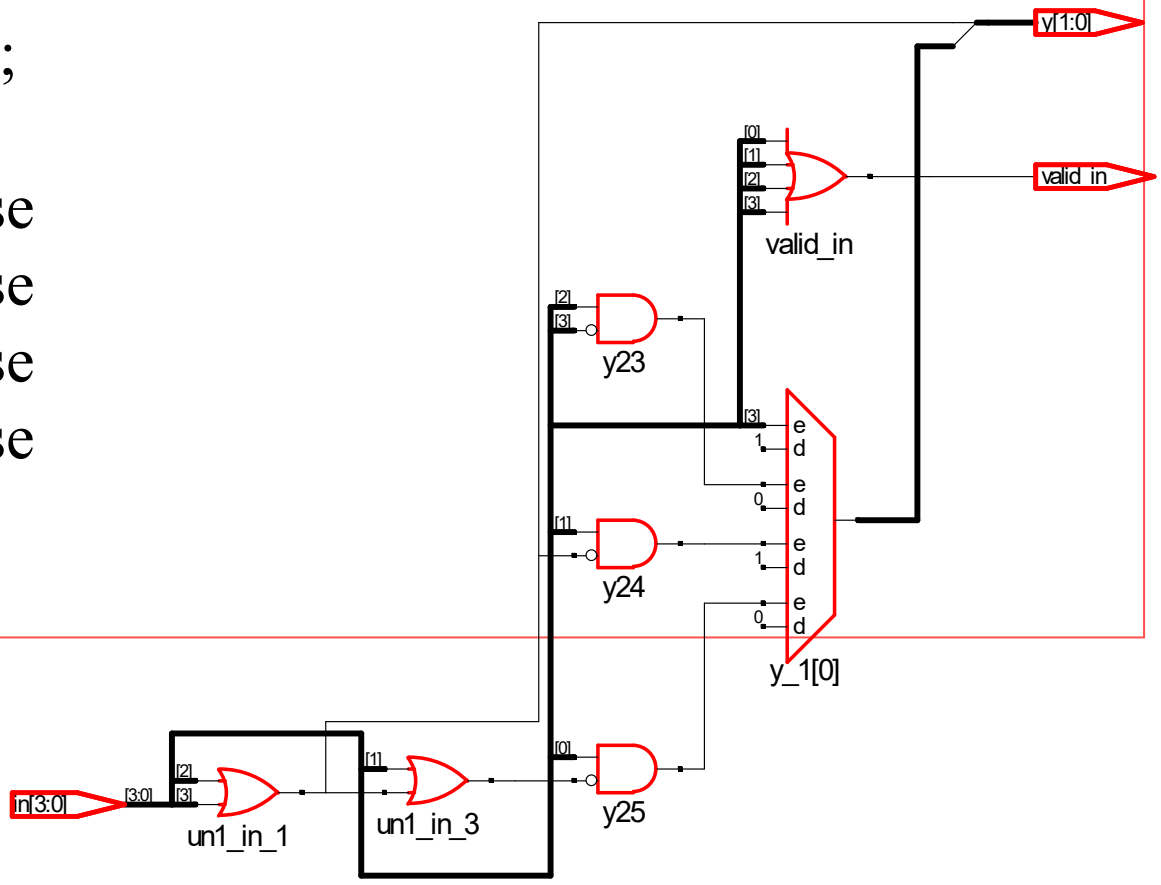
(a) Block diagram

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

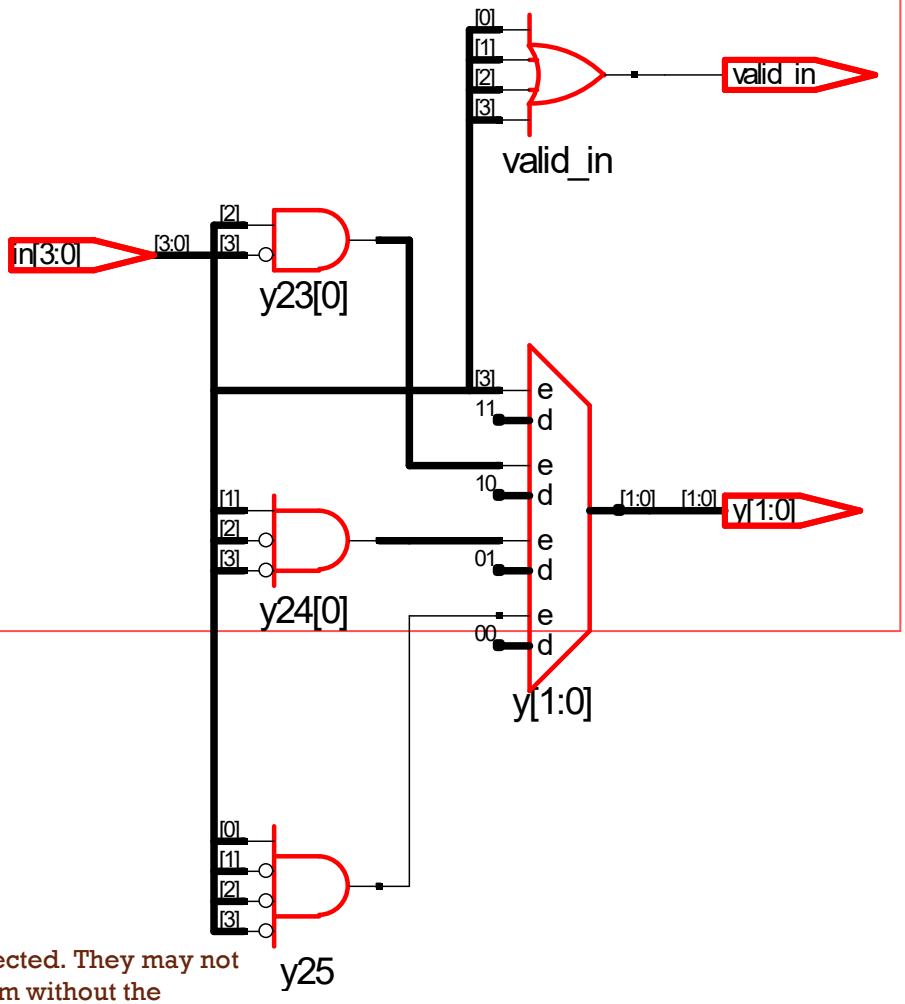
A 4-TO-2 PRIORITY ENCODER EXAMPLE

```
// 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
```



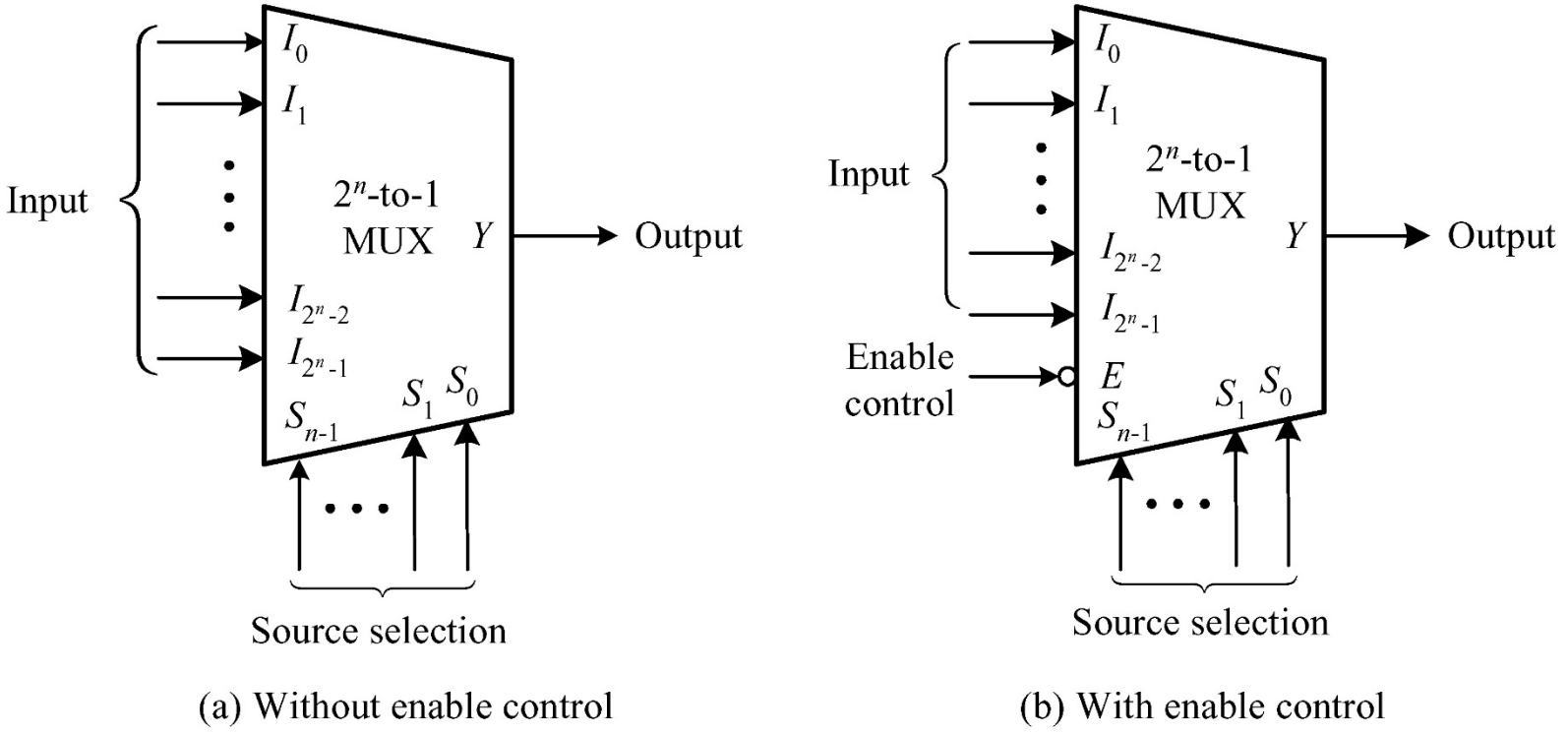
ANOTHER 4-TO-2 PRIORITY ENCODER EXAMPLE

```
// 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
```



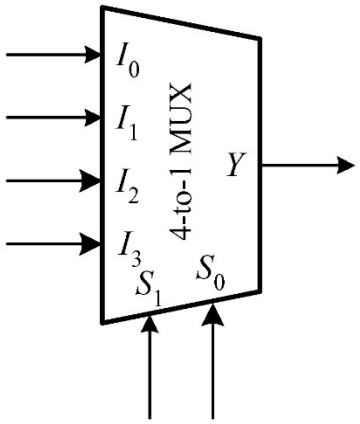
MULTIPLEXER BLOCK DIAGRAMS

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



A 4-TO-1 MULTIPLEXER EXAMPLE

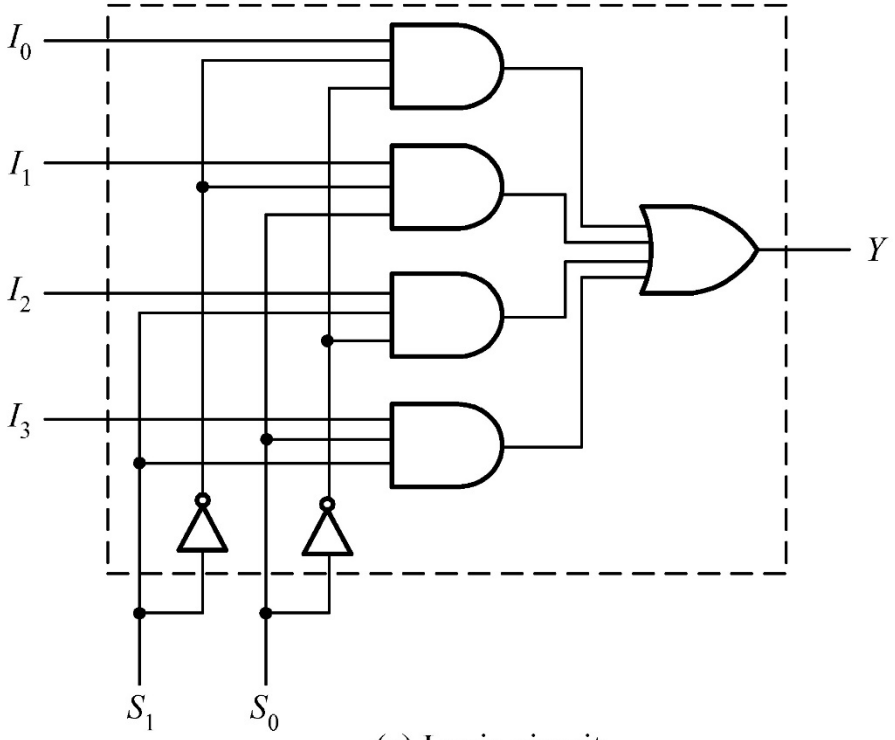
- Gate-based 4-to-1 multiplexers



(a) Logic symbol

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

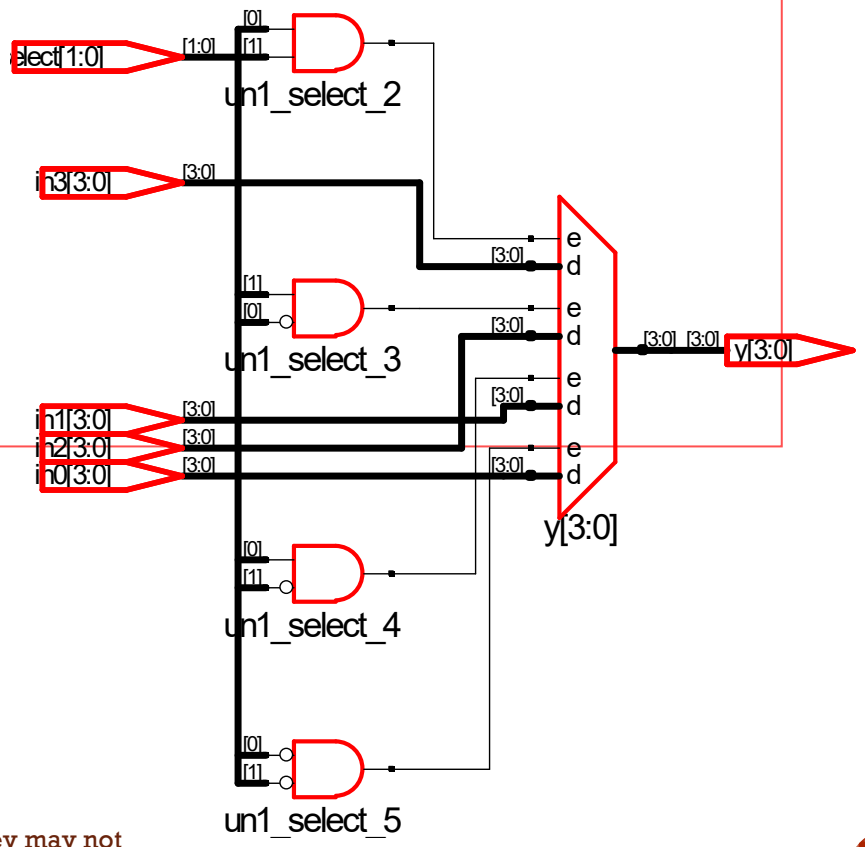
(b) Function table



(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) ;
```

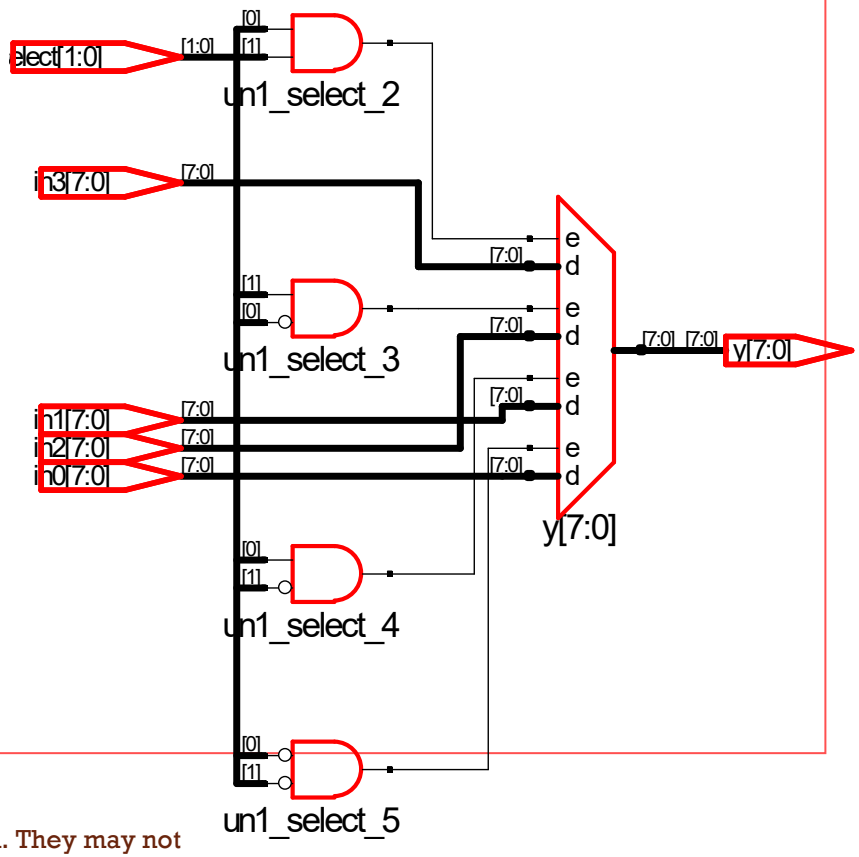


THE SECOND *N*-BIT 4-TO- 1 MULTIPLEXER EXAMPLE

```
// 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) ;
```

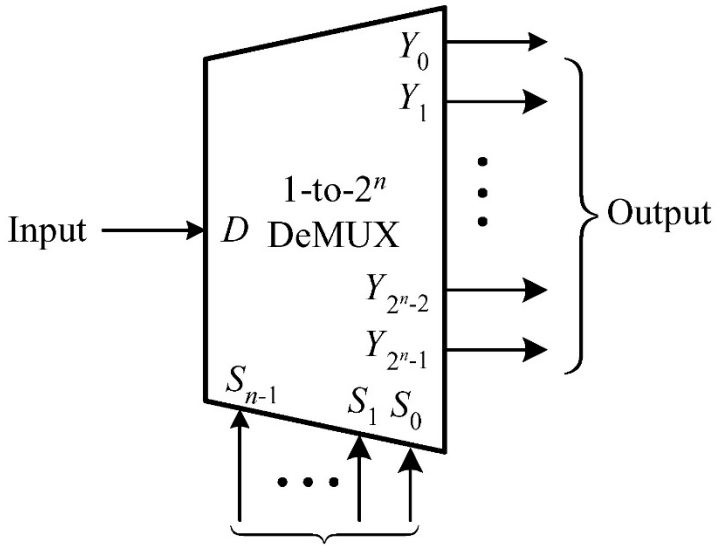
THE THIRD N-BIT 4-TO- 1 MULTIPLEXER EXAMPLE

```
// 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
```

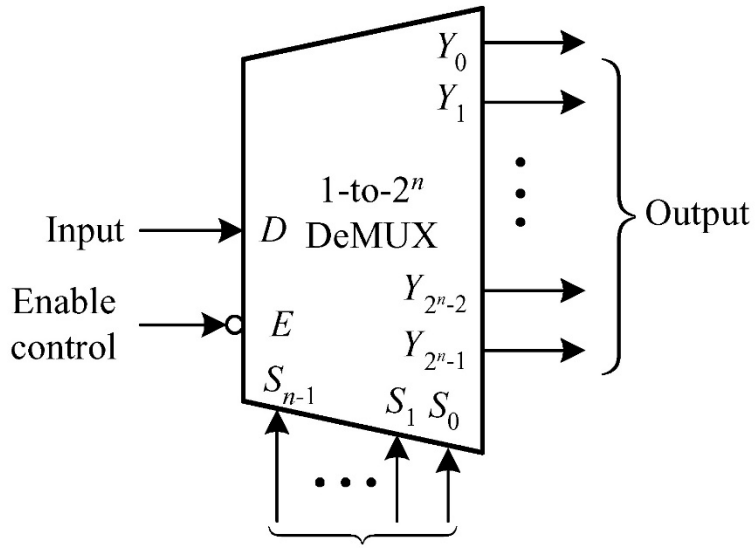


DEMULTIPLEXER BLOCK DIAGRAMS

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



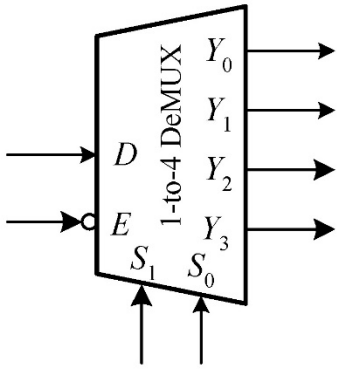
(a) Without enable control



(b) With enable control

A 1-TO-4 DEMULTIPLEXER EXAMPLE

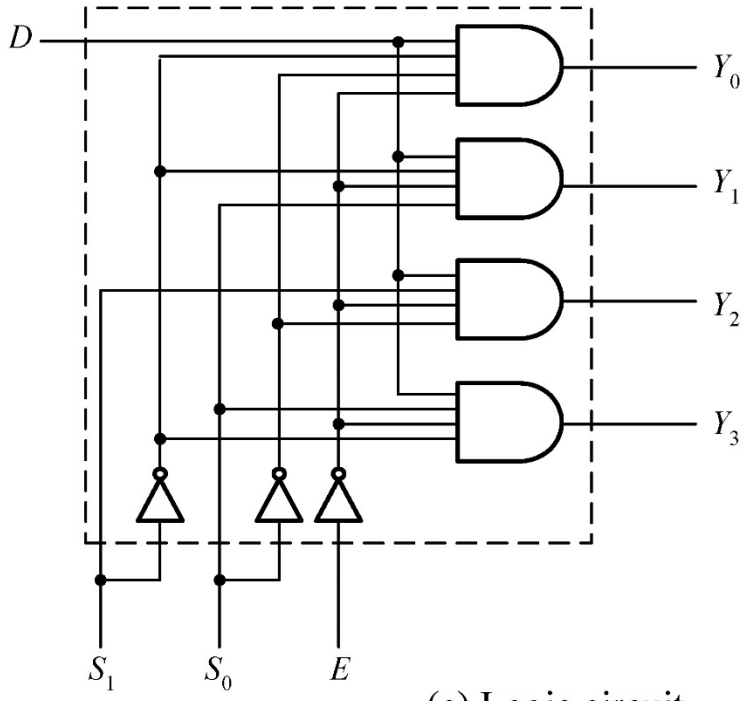
- Gate-based 1-to-4 demultiplexers



(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

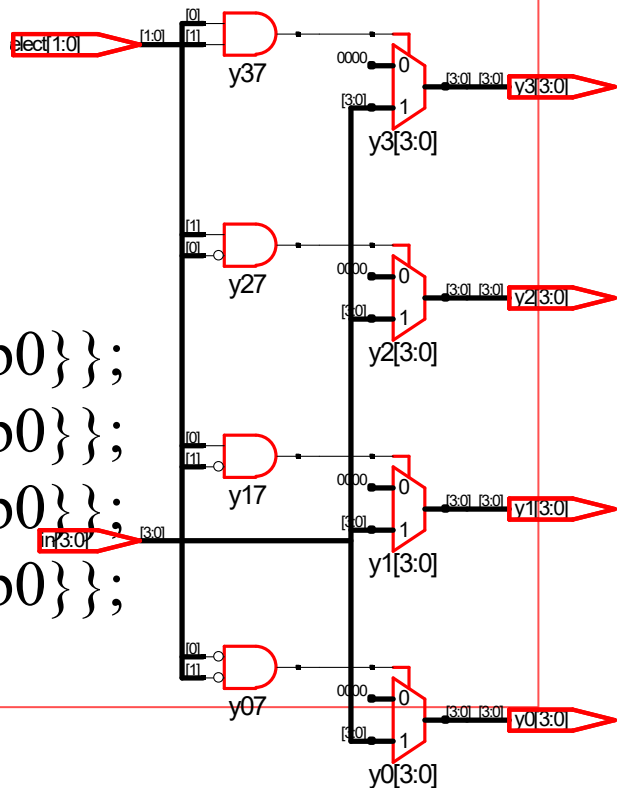
(b) Function table



(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
```

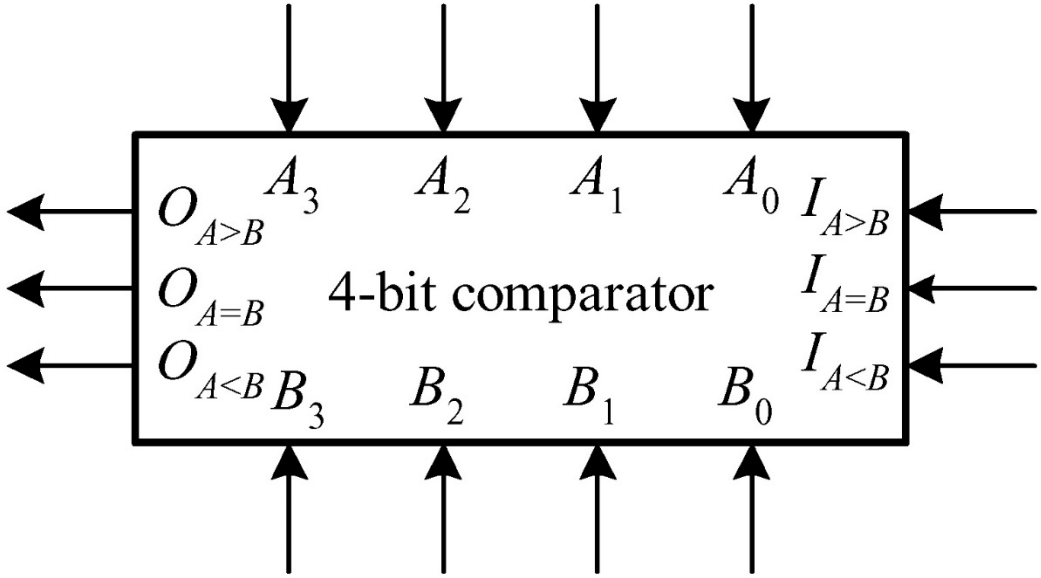


THE SECOND *N*-BIT 1-TO-4 DEMULTIPLEXER EXAMPLE

```
// 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
```

COMPARATORS

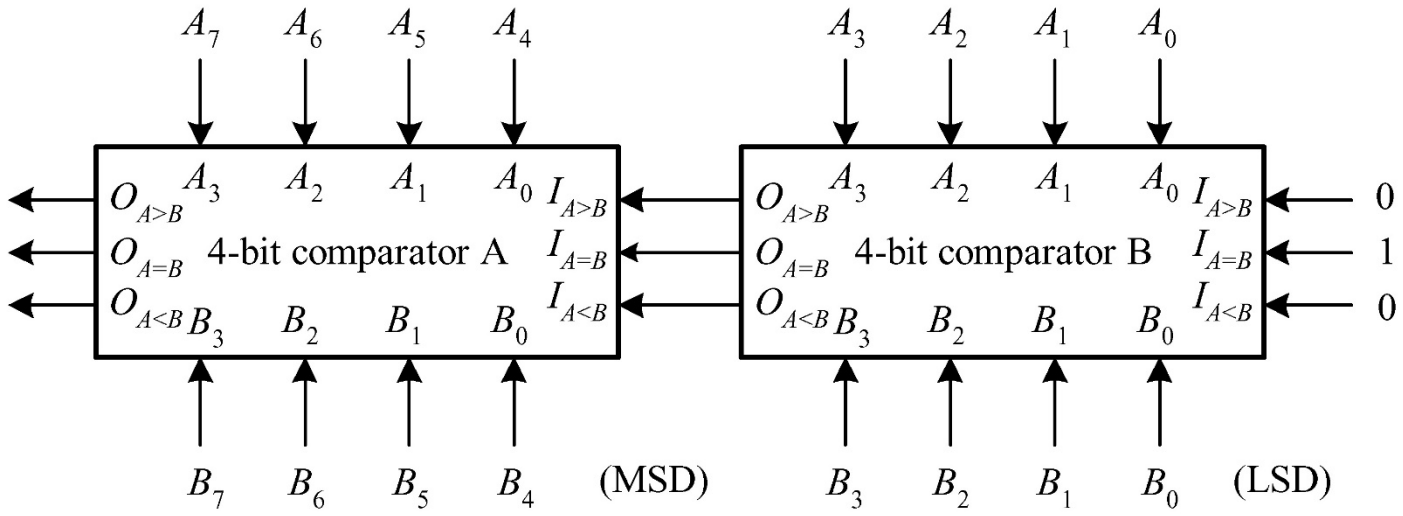
- A 4-bit comparator



A 4-bit cascadable comparator block diagram

COMPARATORS

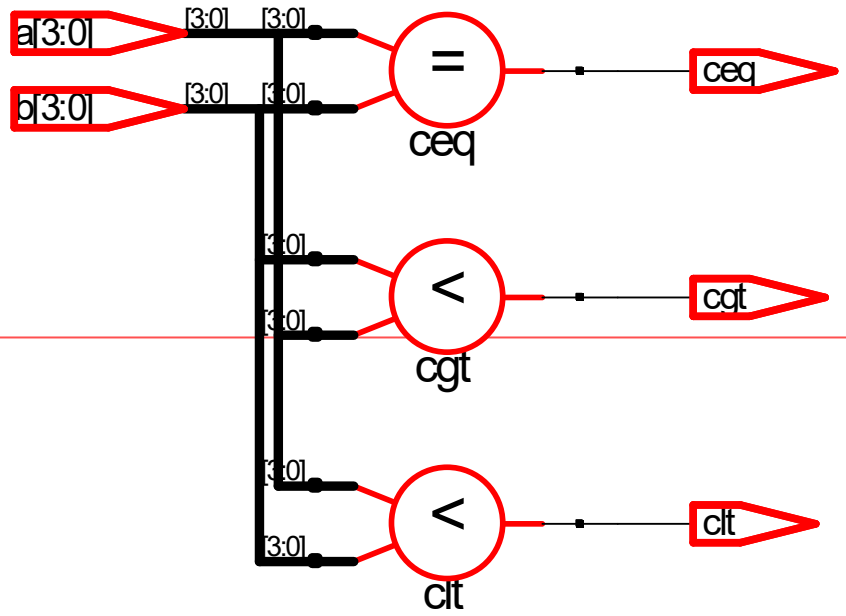
- An 8-bit comparator



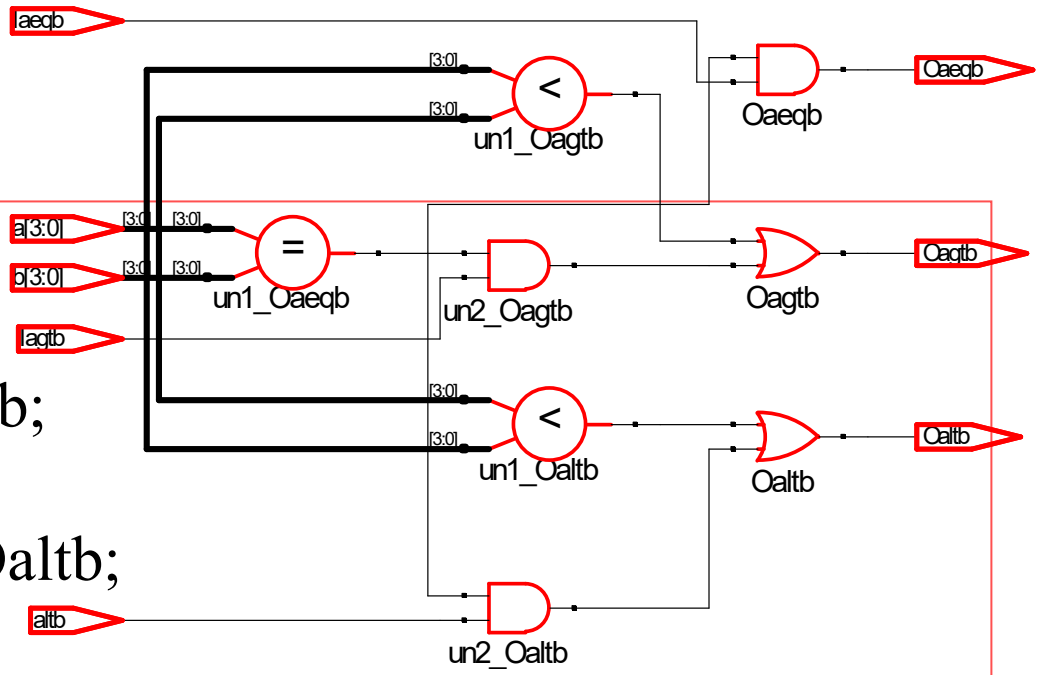
Q: What will happen if you set the input value (010) at the rightmost end to other values?

A SIMPLE COMPARATOR EXAMPLE

```
// an N-bit comparator module example
parameter N = 4;   // default size
input  [N-1:0] a, b;
output cgt, clt, ceq;

assign cgt  = (a > b);
assign clt  = (a < b);
assign ceq = (a == b);
```



A CASCADABLE COMPARATOR EXAMPLE




```
parameter N = 4;
// 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);  // =
assign Oagtb = (a > b) || ((a == b)&& (Iagtb == 1));  // >
assign Oaltb =  (a < b) ||  ((a == b)&& (Ialtb == 1));  // <
```

SEQUENTIAL LOGIC BLOCKS

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;
```

Q: How would you model a *D*-type latch?

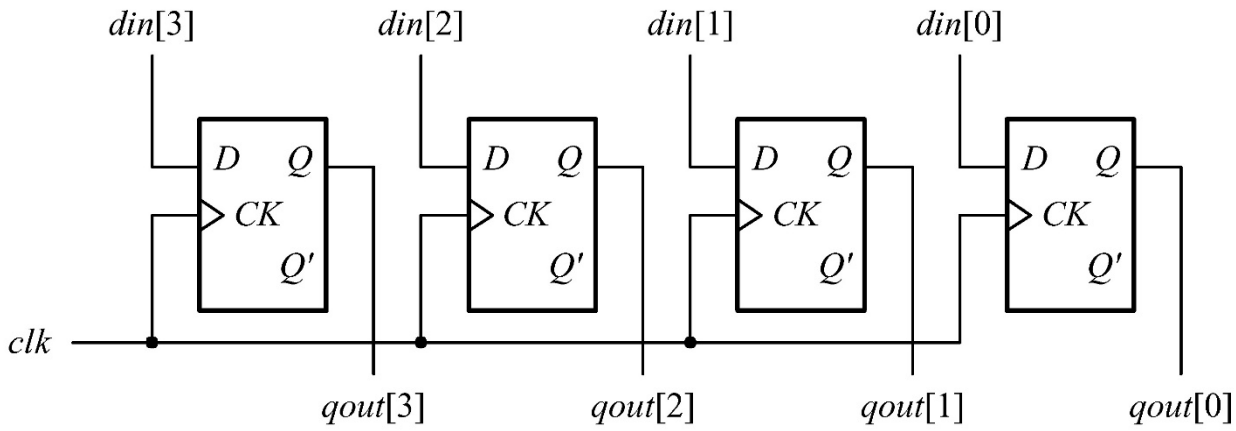
SYNCHRONOUS RESET D-TYPE FLIP-FLOPS

```
// 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;
```

REGISTERS

- Registers (or data registers)
- A flip-flop
  - Area: 10 to 20x of an SRAM cell
- In Xilinx FPGAs
  - Flip-flops
  - Distributed memory
  - Block memory

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;
```

A REGISTER FILE

```
// an N-word register file with one-write and two-read ports
parameter M = 4;   // number of address bits
parameter N = 16;  // number of words, N = 2**M
parameter W = 8;   // number of bits in a word
input  clk, wr_enable;
input   [W-1:0]   din;
output [W-1:0]   douta, doutb;
input   [M-1:0]   rd_addra, rd_addrb, wr_addr;
reg     [W-1:0]   reg_file [N-1:0];
…
assign douta = reg_file[rd_addra],
          doutb = reg_file[rd_addrb];
always @(posedge clk)
  if (wr_enable) reg_file[wr_addr] <= din;
```

Try to synthesize it and see what happens!!

AN SYNCHRONOUS RAM

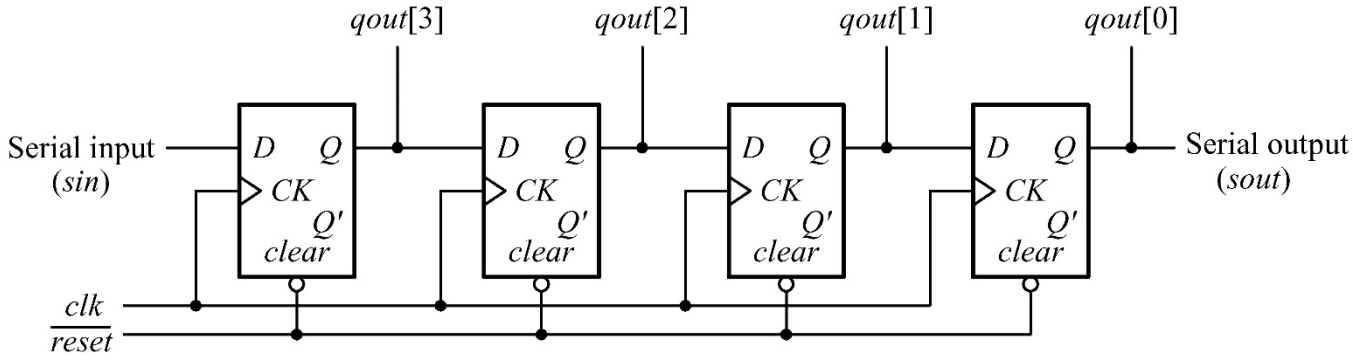
```
// a synchronous RAM module example
parameter N = 16;   // number of words
parameter A = 4;    // number of address bits
parameter W = 4;    // number of wordsize in bits
input  [A-1:0]  addr;
input  [W-1:0] din;
input   cs, wr, clk;   // chip select, read-write control, and clock signals
output reg [W-1:0] dout;
reg    [W-1:0] ram [N-1:0];  // declare an N * W memory array

always @(posedge clk)
    if (cs) if (wr) ram[addr] <= din;
            else            dout <= ram[addr];
```

Try to synthesize it and see what happens!!

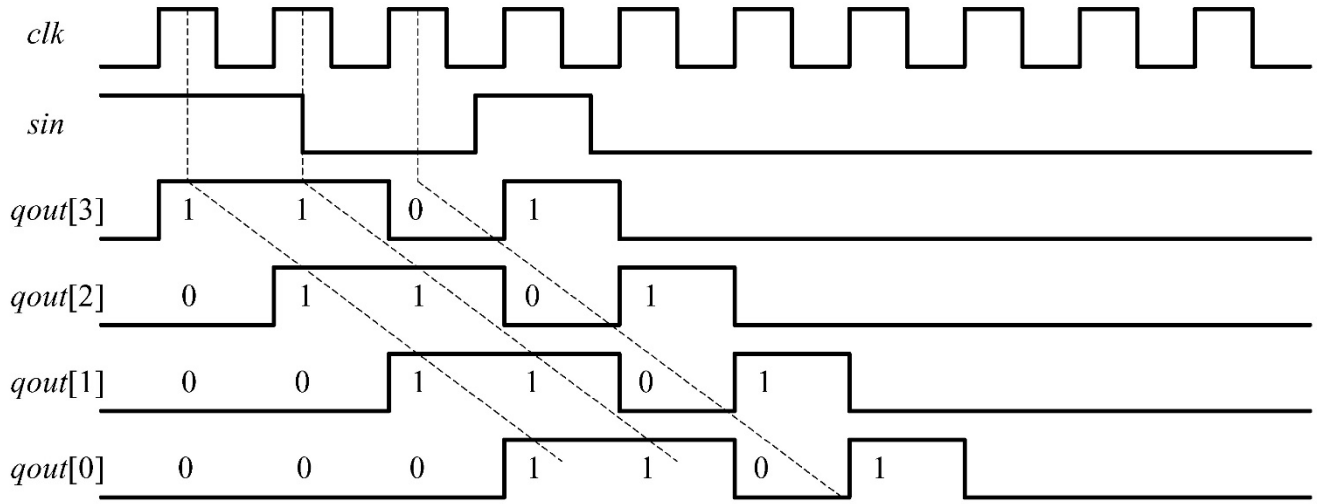
SHIFT REGISTERS

- Shift registers
- Parallel/serial format conversion
  - SISO (serial in serial out)
  - SIPO (serial in parallel out)
  - PISO (parallel in serial out)
  - PIPO (parallel in parallel out)

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]};
```

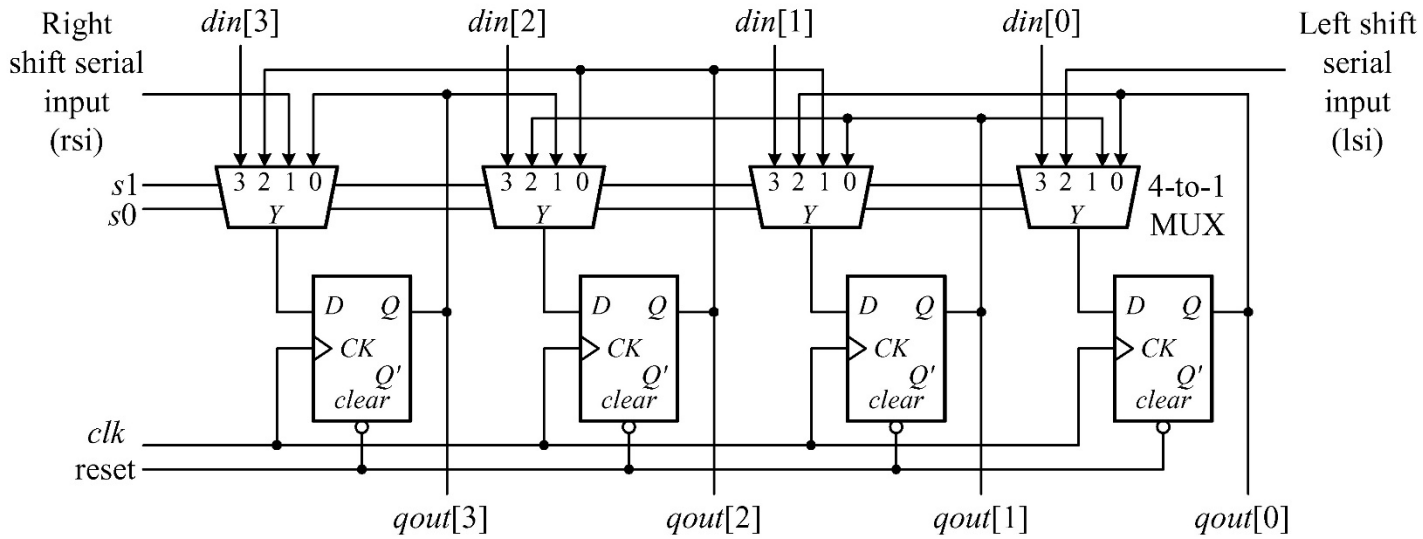
A SHIFT REGISTER WITH PARALLEL LOAD

```
// a shift register with parallel load module example
module shift_register_parallel_load
                                (clk, load, reset_n, din, sin, qout);
parameter N = 8;  // 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 if (load) qout <= din;
   else           qout <= {sin, qout[N-1:1]};
```

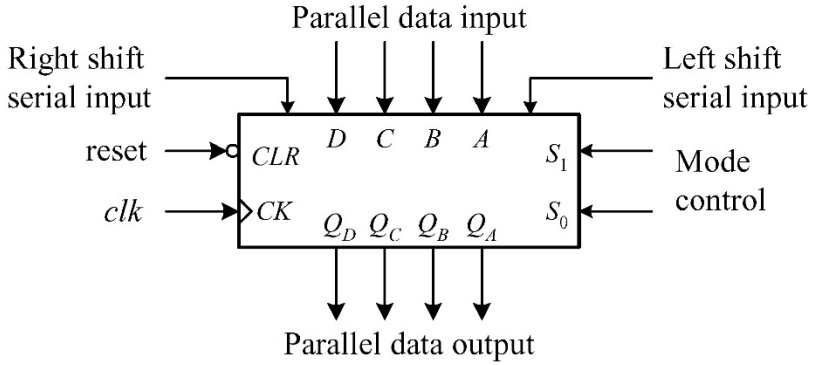
UNIVERSAL SHIFT REGISTERS

- A universal shift register can carry out
  - SISO
  - SIPO
  - PISO
  - PIPO
- The register must have the following capabilities
  - Parallel load
  - Serial in and serial out
  - Shift left and shift right

UNIVERSAL SHIFT REGISTERS



(a) Logic diagram



(b) Logic symbol

$s1$	$s0$	Function
0	0	No change
0	1	Right shift
1	0	Left shift
1	1	Load data

(c) Function table

UNIVERSAL SHIFT REGISTERS

```
// a universal shift register module
module universal_shift_register (clk, reset_n, s1, s0, …);
parameter N = 4;  // define the default size
…
always @(posedge clk or negedge reset_n)
   if (!reset_n) qout <= {N{1'b0}};
   else case ({s1,s0})
        2'b00: ; // qout <= qout;                  // No change
        2'b01: qout <= {lsi, qout[N-1:1]};   // Shift right
        2'b10: qout <= {qout[N-2:0], rsi};   // Shift left
        2'b11: qout <= din;                        // Parallel load
   endcase
```

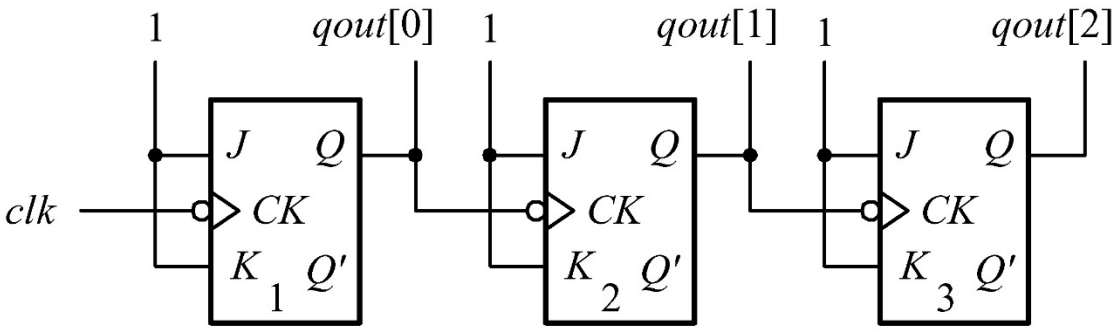
COUNTERS

- Counter
- Types
  - Counters
  - Timers

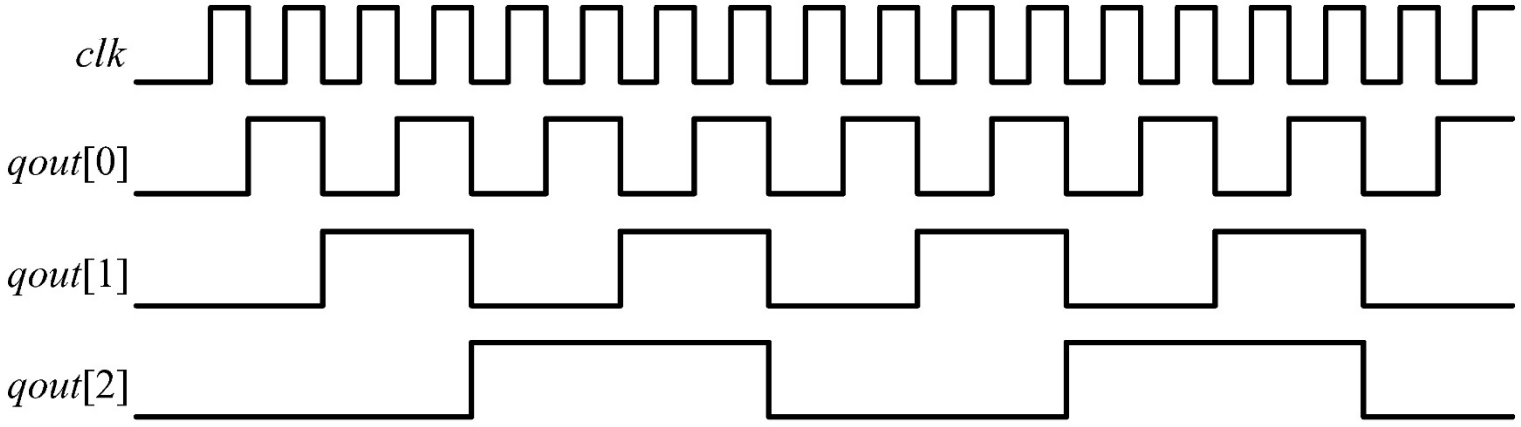
TYPES OF COUNTERS

- Types of counters
  - Asynchronous
  - Synchronous
- Asynchronous (ripple) counters
  - Binary counter (up/down counters)
- Synchronous counters
  - Binary counter (up/down counters)
  - BCD counter (up/down counters)
  - Gray counters (up/down counters)

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];
```

- Try to synthesize it and see what happens!!
- The output cannot be observed from simulators due to lacking initial values of qout.

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];
```

A BINARY RIPPLE COUNTER

```
// an N-bit ripple counter using generate blocks
parameter N = 4; // define the size of counter
…
output reg [N-1:0] qout;
genvar  i;
generate for (i = 0; i < N; i = i + 1) begin: ripple_counter
   if (i == 0) // specify LSB
      always @(negedge clk or negedge reset_n)
         if (!reset_n) qout[0] <= 1'b0;  else qout[0] <= ~qout[0];
   else          // specify the rest bits
      always @(negedge qout[i-1]or negedge reset_n)
         if (!reset_n) qout[i] <= 1'b0;  else qout[i] <= ~qout[i];
end endgenerate
```

A BINARY COUNTER EXAMPLE

```
module binary_counter(clk, enable, reset, qout, cout);
parameter N = 4;
…
output reg [N-1:0] qout;
output cout;       // carry output

always @(posedge clk)
   if (reset)           qout <= {N{1'b0}};
   else if (enable)  qout <= qout + 1;
// generate carry output
assign #2 cout = &qout;  // Why #2 is required ?
```

BINARY UP/DOWN COUNTERS --- VERSION 1

```
module binary_up_down_counter_reset
                        (clk, enable, reset, upcnt, qout, cout, bout);
parameter N = 4;
…
output reg [N-1:0] qout;
output cout, bout;  // carry and borrow outputs
always @(posedge clk)
   if (reset) qout <= {N{1'b0}};
   else if (enable) begin
            if (upcnt) qout <= qout + 1;
            else       qout <= qout - 1;  end
assign #2 cout = &qout;  // Why #2 is required ?
assign #2 bout = |qout;
```

BINARY UP/DOWN COUNTERS --- VERSION 2

```
module up_dn_bin_counter
                        (clk, reset, eup, edn, qout, cout, bout);
Parameter N = 4;
…
output reg [N-1:0] qout;
output cout, bout;

always @(posedge clk)
   if (reset)       qout <= {N{1'b0}};    // synchronous reset
   else if (eup)  qout <= qout + 1;
   else if (edn)  qout <= qout - 1;
assign #1 cout = (&qout)& eup;        // generate carry out
assign #1 bout = (~|qout)& edn;       // generate borrow out
```

BINARY UP/DOWN COUNTERS --- VERSION 2

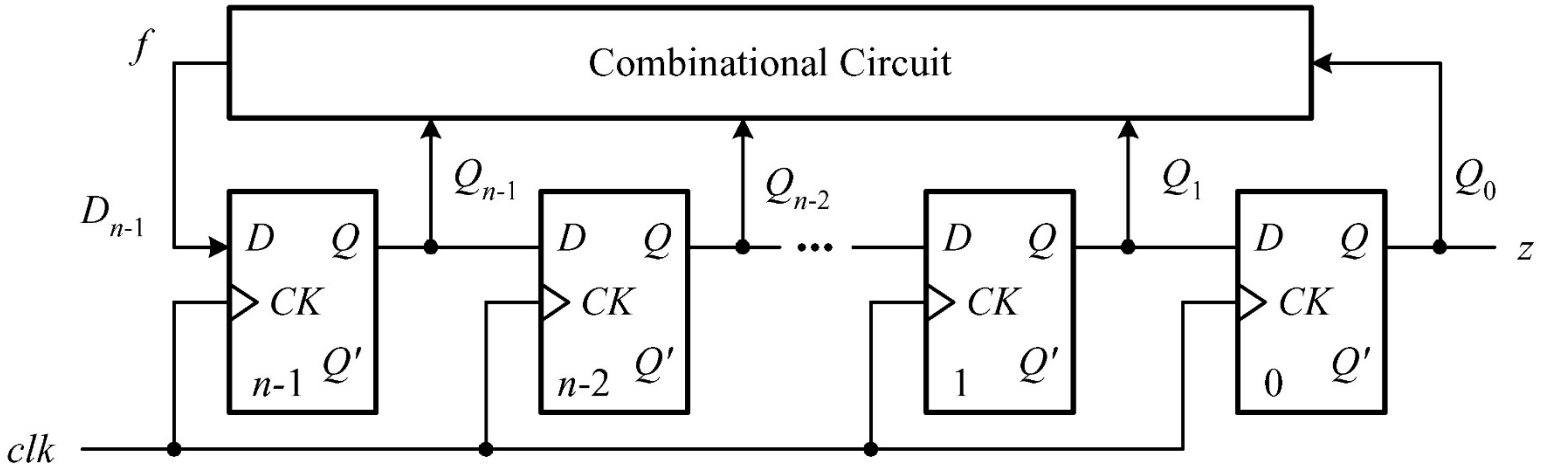
```
// the cascade of two up/down counters
module up_dn_bin_counter_cascaded(clk, reset,eup, …);
parameter N = 4;
…
output [2*N-1:0] qout;
output cout, bout;
wire   cout1, bout1;

    up_dn_bin_counter #(4) up_dn_cnt1 (clk, reset,eup, edn, …);
    up_dn_bin_counter #(4) up_dn_cnt2 (clk, reset, …);
```

A MODULO R BINARY COUNTER

```
module modulo_r_counter(clk, enable, reset, qout, cout);
parameter N = 4;
parameter R= 10;  // BCD counter
…
output reg [N-1:0] qout;
…
assign cout = (qout == R - 1);
always @(posedge clk)
    if (reset)  qout <= {N{1'b0}};
    else begin
        if (enable) if (cout) qout <= 0;
                    else      qout <= qout + 1; end
```

SEQUENCE GENERATORS

- We only focus on the following three circuits
  - PR (pseudo random)-sequence generator
  - Ring counter
  - Johnson counter

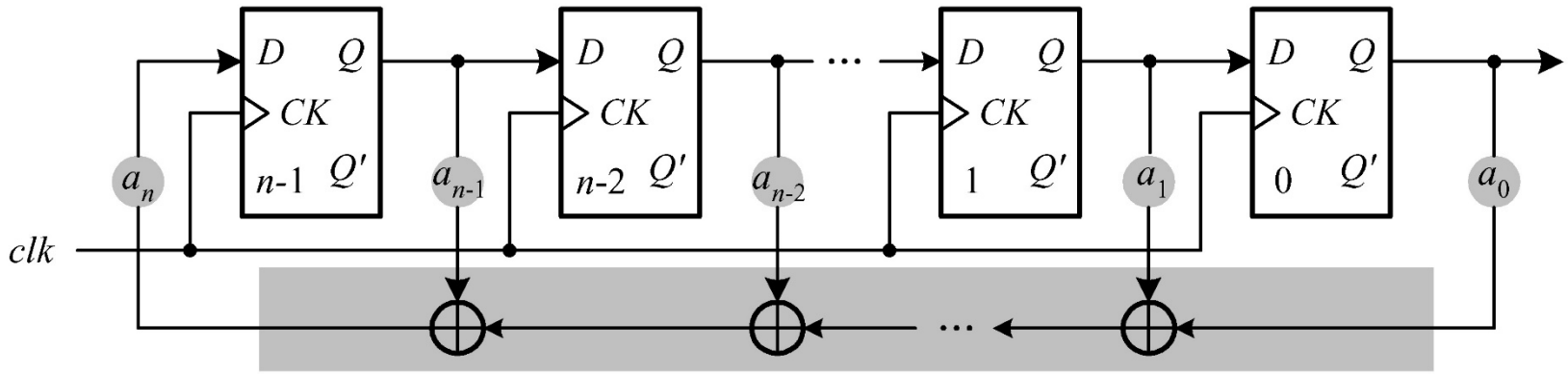


PRIMITIVE POLYNOMIALS

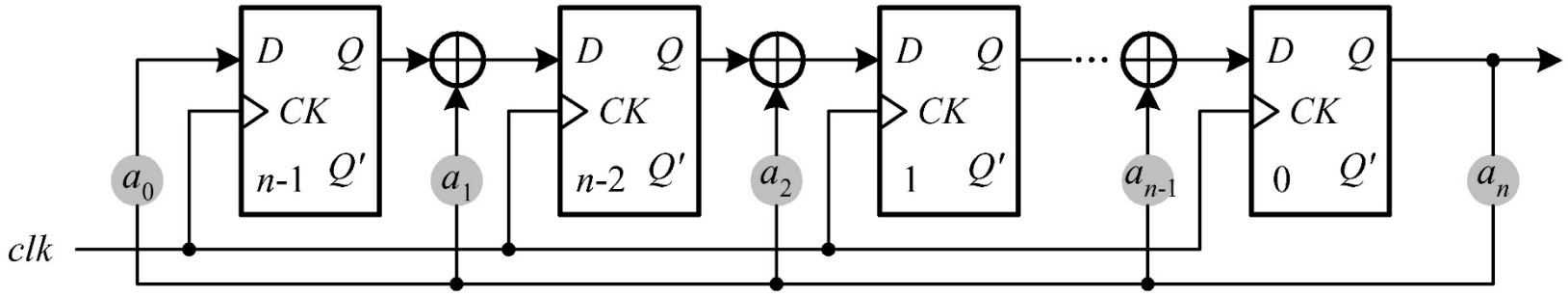
n	$f(x)$	n	$f(x)$	n	$f(x)$
1, 2, 3, 4, 6, 7, 15, 22, 60	$1+x+x^n$	24	$1+x+x^2+x^7+x^n$	43	$1+x+x^5+x^6+x^n$
		26	$1+x+x^2+x^6+x^n$	44, 50	$1+x+x^{26}+x^{27}+x^n$
5, 11, 21, 29	$1+x^2+x^n$	30	$1+x+x^2+x^{23}+x^n$	45	$1+x+x^3+x^4+x^n$
10, 17, 20, 25, 28, 31, 41, 52	$1+x^3+x^n$	32	$1+x+x^2+x^{22}+x^n$	46	$1+x+x^{20}+x^{21}+x^n$
		33	$1+x^{13}+x^n$	48	$1+x+x^{27}+x^{28}+x^n$
9	$1+x^4+x^n$	34	$1+x+x^{14}+x^{15}+x^n$	49	$1+x^9+x^n$
23, 47	$1+x^5+x^n$	35	$1+x^2+x^n$	51, 53	$1+x+x^{15}+x^{16}+x^n$
18	$1+x^7+x^n$	36	$1+x^{11}+x^n$	54	$1+x+x^{36}+x^{37}+x^n$
8	$1+x^2+x^3+x^4+x^n$	37	$1+x^2+x^{10}+x^{12}+x^n$	55	$1+x^{24}+x^n$
12	$1+x+x^4+x^6+x^n$	38	$1+x+x^5+x^6+x^n$	56,59	$1+x+x^{21}+x^{22}+x^n$
13	$1+x+x^3+x^4+x^n$	39	$1+x^4+x^n$	57	$1+x^7+x^n$
14, 16	$1+x^3+x^4+x^5+x^n$	40	$1+x^2+x^{19}+x^{21}+x^n$	58	$1+x^{19}+x^n$
19, 27	$1+x+x^2+x^5+x^n$	42	$1+x+x^{22}+x^{23}+x^n$		

$$f(x)=\sum_{i=0}^{n} a_i x^i = a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n$$

MAXIMAL LENGTH SEQUENCE GENERATORS



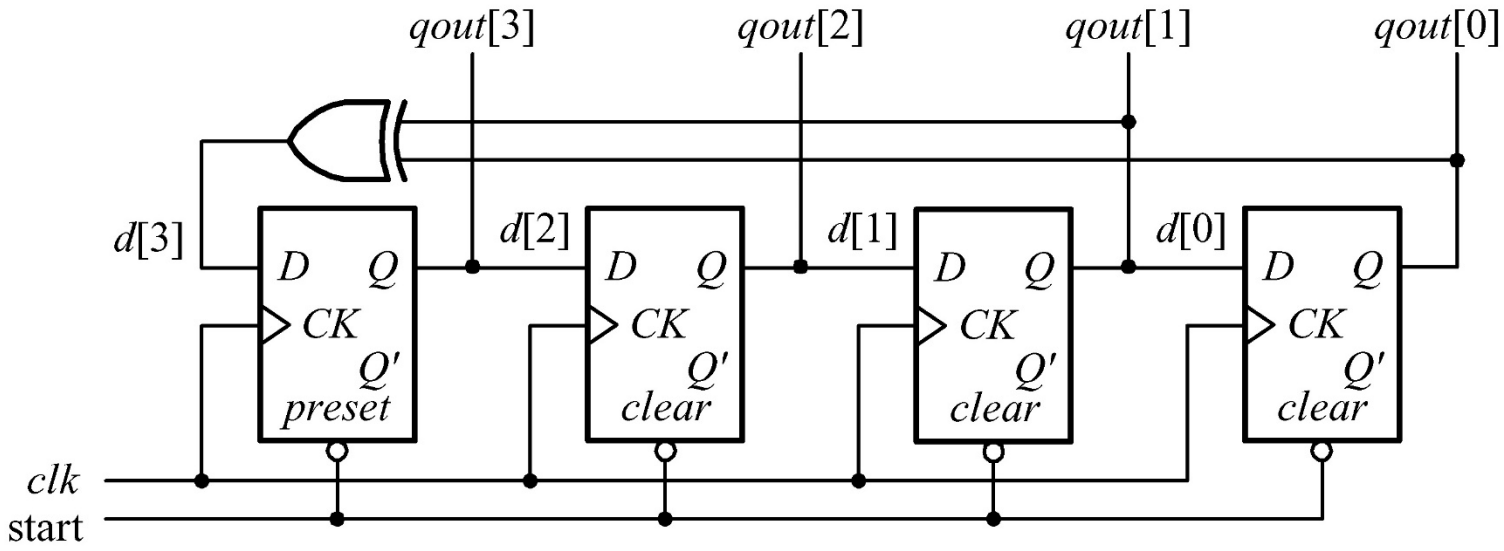
(a) Standard format



(b) Modular format

A PR-SEQUENCE GENERATOR EXAMPLE

- A 4-bit example
  - primitive polynomial: $1 + x + x^4$



A PR-SEQUENCE GENERATOR EXAMPLE

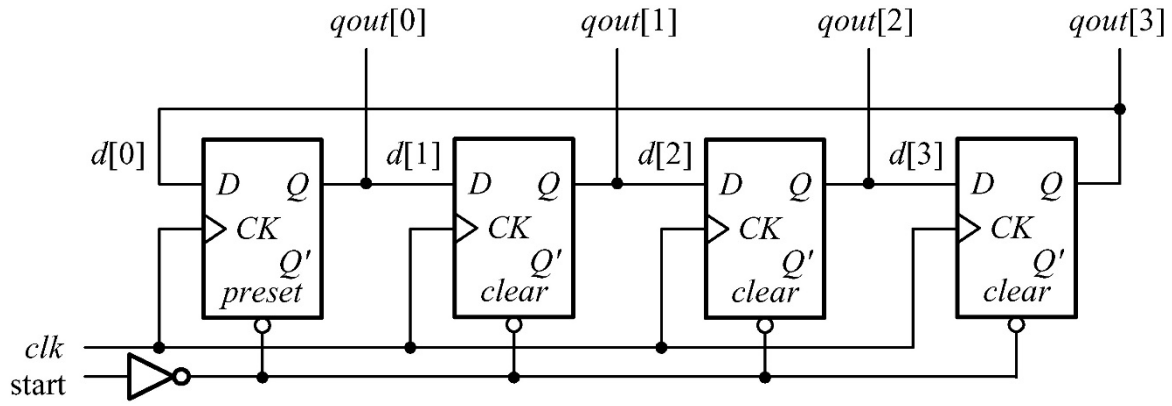
```
// an N-bit pr_sequence generator module --- in standard form
module pr_sequence_generate (clk, qout);
parameter N = 4;  // define the default size
parameter [N:0] tap = 5'b10011;
…
output reg [N-1:0] qout = 4'b0100;
wire   d;
…
assign d = ^(tap[N-1:0] & qout[N-1:0]);
always @(posedge clk)
      qout <= {d, qout[N-1:1]};
```

Q: Write an N-bit pr_sequence generator in modular form.

A PR-SEQUENCE GENERATOR EXAMPLE

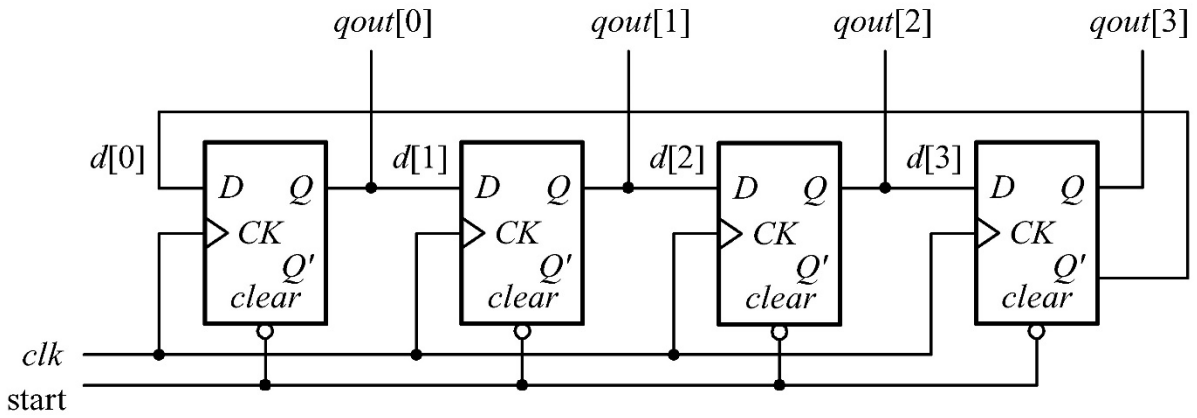
```
// an N-bit pr_sequence generator module --- in standard form
module pr_sequence_generate (clk, start, qout);
parameter N = 4;  // define the default size
parameter [N:0] tap = 5'b10011;
…
output reg [N-1:0] qout;
wire   d;
…
assign d = ^(tap[N-1:0] & qout[N-1:0]);
always @(posedge clk or posedge start)
   if (start)  qout <= {1'b1, {N-1{1'b0}}};
   else        qout <= {d, qout[N-1:1]};
```

RING COUNTERS




```
// a ring counter with initial value
module ring_counter(clk, start, qout);
parameter N = 4;
…
output reg [0:N-1] qout;
…
always @(posedge clk or posedge start)
  if (start)  qout <= {1'b1,{N-1{1'b0}}};
  else        qout <= {qout[N-1], qout[0:N-2]};
```

JOHNSON COUNTERS



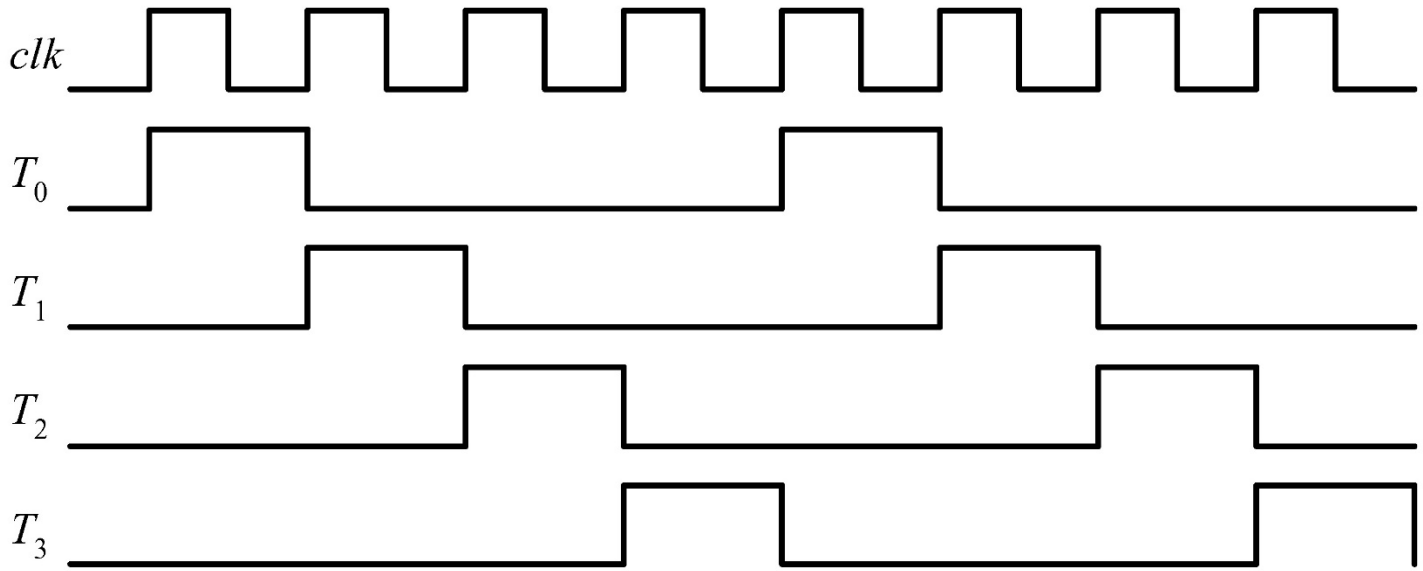

```
// Johnson counter with initial value
module ring_counter(clk, start, qout);
parameter N = 4; // define the default size
…
output reg [0:N-1] qout;
…
always @(posedge clk or posedge start)
    if (start)  qout <= {N{1'b0}};
    else        qout <= {~qout[N-1], qout[0:N-2]};
```

TIMING GENERATORS

- A timing generator
- Multiphase clock signals
  - Ring counter
  - Binary counter with decoder
- Digital monostable circuits
  - Retriggerable
  - Nonretriggerable

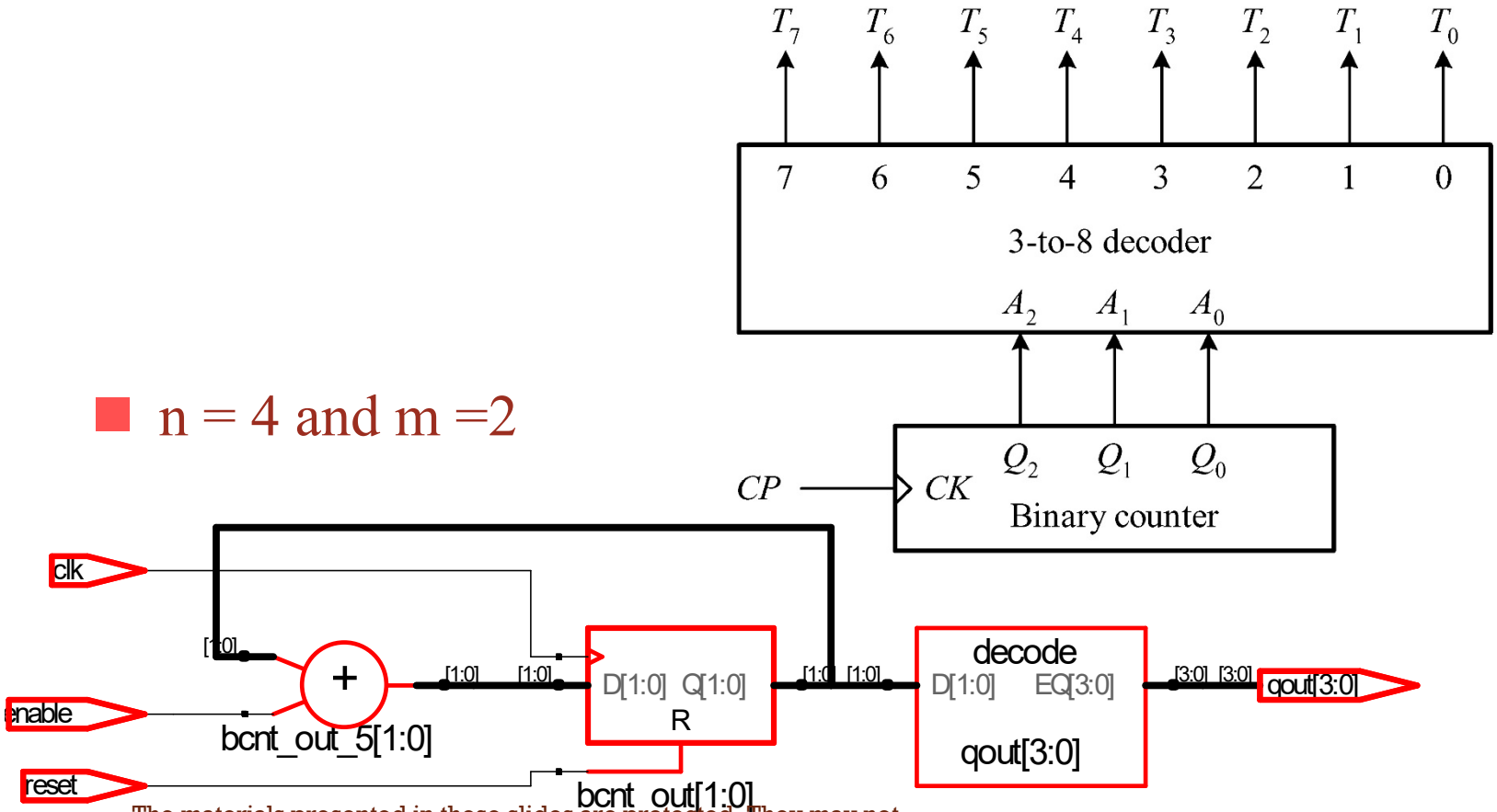
MULTIPHASE CLOCK SIGNALS

- Ways of generation
  - Ring counter approach
  - Binary counter with decoder approach



MULTIPHASE CLOCK GENERATORS

- Binary counter with decoder approach



- n = 4 and m =2

MULTIPHASE CLOCK GENERATORS

```
// a binary counter with decoder serve as a timing generator
module binary_counter_timing_generator(clk, reset, enable, qout);
parameter N = 8;   // the number of phases
parameter M = 3;  // the bit number of binary counter
…
output reg [N-1:0] qout;
reg    [M-1:0] bcnt_out;
…
always @(posedge clk or posedge reset)
  if (reset)  bcnt_out <= {M{1'b0}};
  else if (enable) bcnt_out <= bcnt_out + 1;
…
always @(bcnt_out)
   qout = {N-1{1'b0},1'b1} << bcnt_out;
```