Combinational Circuit Design

Combinational Circuit Design

Commonly used combinational logic modules:

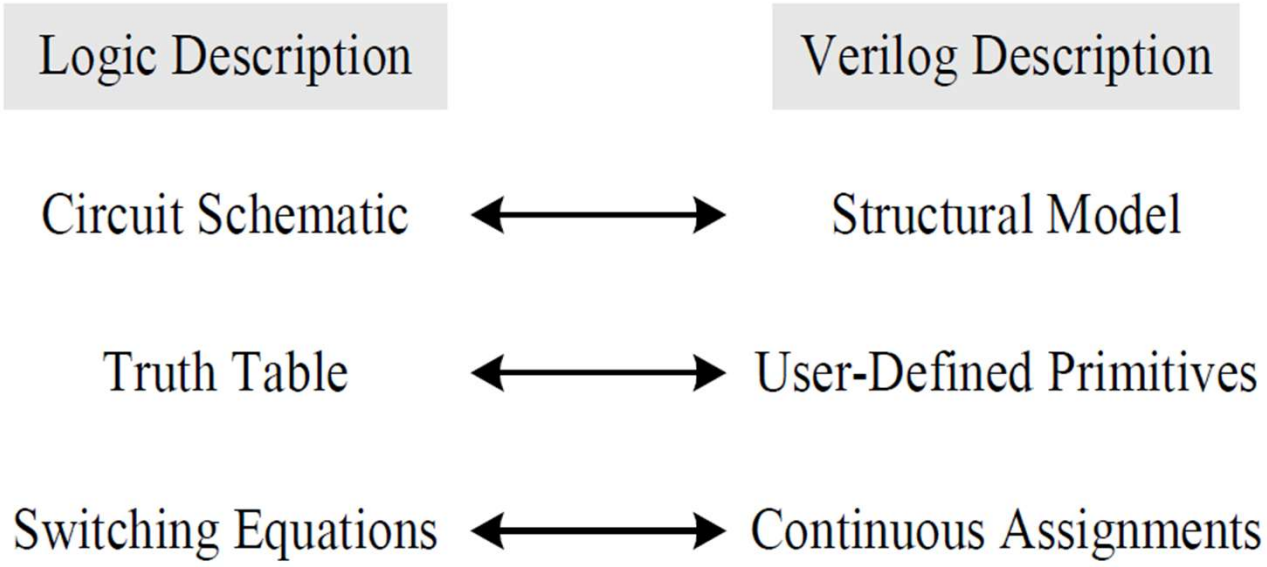
- ❖ Decoder
- ❖ Encoder
- ❖ Multiplexer
- ❖ Demultiplexer
- ❖ Comparator
- ❖ Adder (CLA)
- ❖ Subtracter (subtractor)
- ❖ Multiplier
- ❖ PLA
- ❖ Parity Generator

Combinational Circuit Design

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

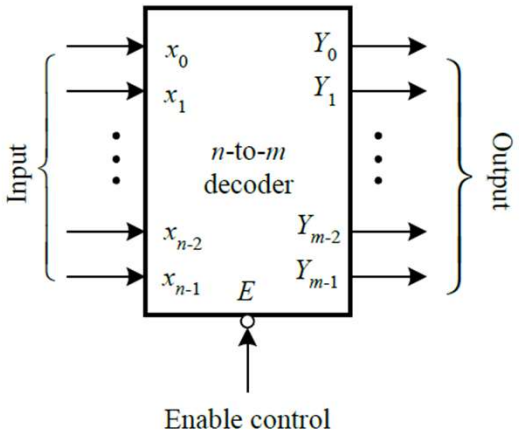
Combinational Circuit Design

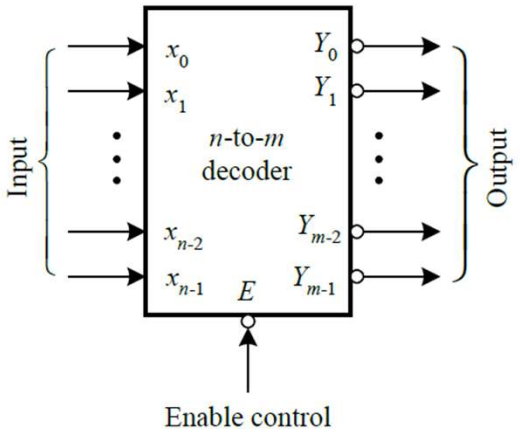


Decoder

Decoder Block Diagrams

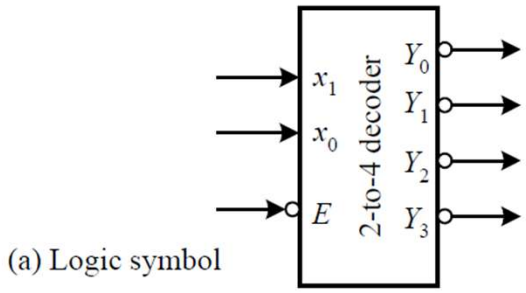
- An $n \times m$ decoder has n input lines and m output lines. Each output line Y_i corresponds to the ith minterm of input (line) variables.
  - Total decoding: when $m = 2^n$.
  - Partial decoding: when $m < 2^n$.





Decoder

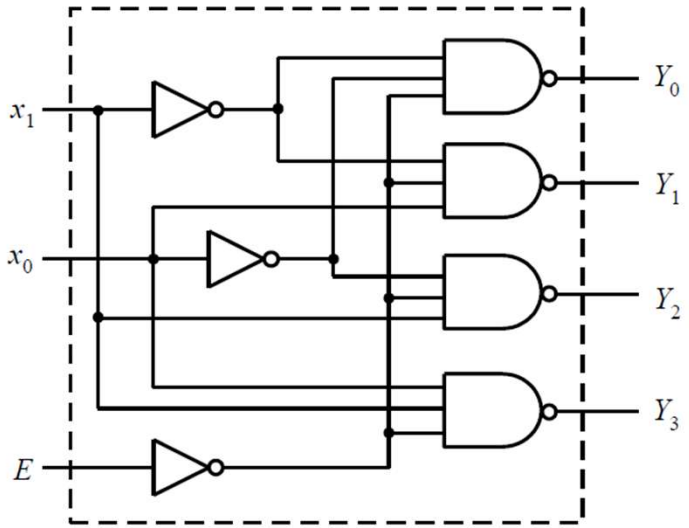
A 2-to-4 Decoder Example



(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

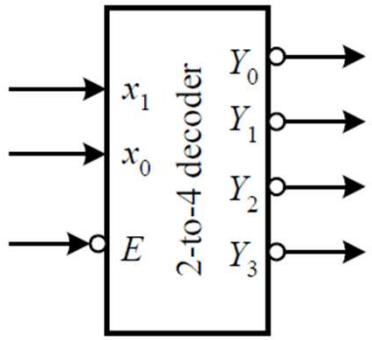


(c) Logic circuit

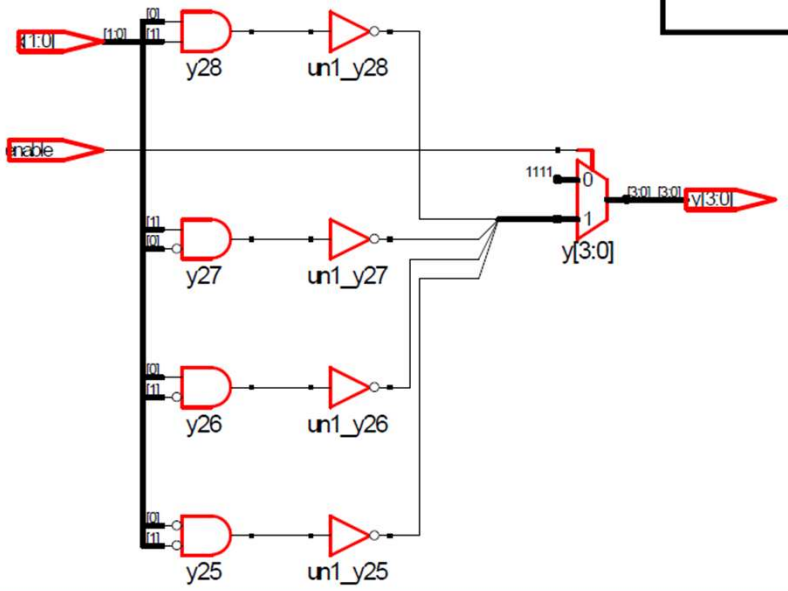
ϕ : don't care

Decoder

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
```



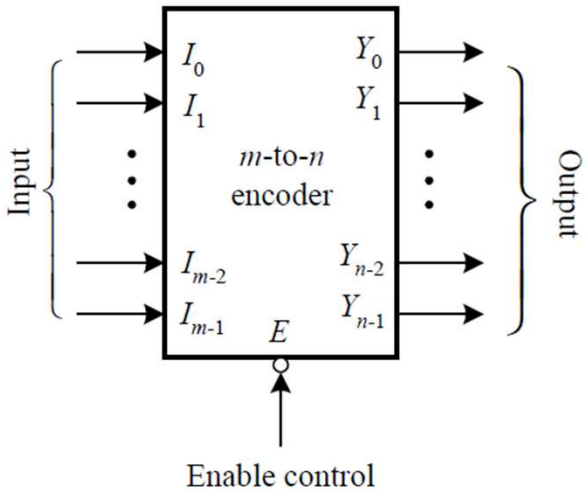
Decoder

An m-to-n decoder with active-high output

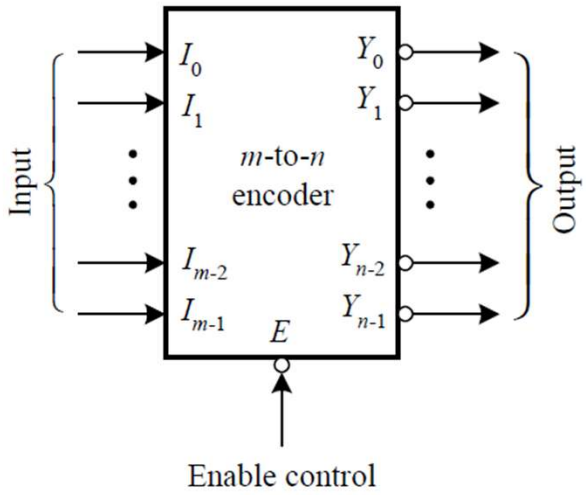
```
module decoder_m2n_high(x, enable, y);
parameter m = 3; // define the number of input lines
parameter n = 8; // define the number of output lines
input [m-1:0] x;
input enable;
output reg [n-1:0] y;// The body of the m-to-n decoder
always @(x or enable)
if (!enable) y = {n{1'b0}};
else y = {{n-1{1'b0}},1'b1}}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

Encoder

- 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



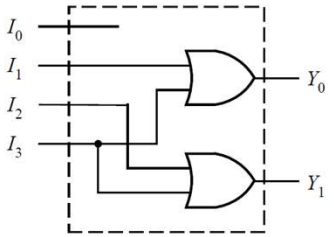
(b) Inverted output

Encoder

A 4-to-2 Encoder Example

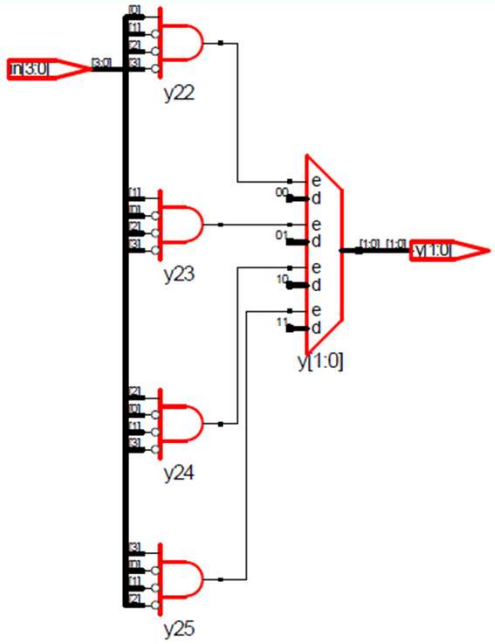
I_3	I_2	I_1	I_0	Y_1	Y_0
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

(a) Function table



(b) Logic circuit

```
// a 4-to-2 encoder 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
```

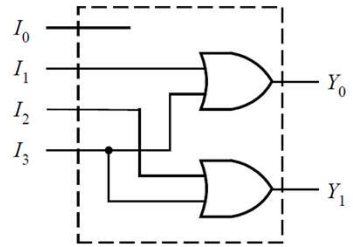


Encoder

Another 4-to-2 Encoder Example

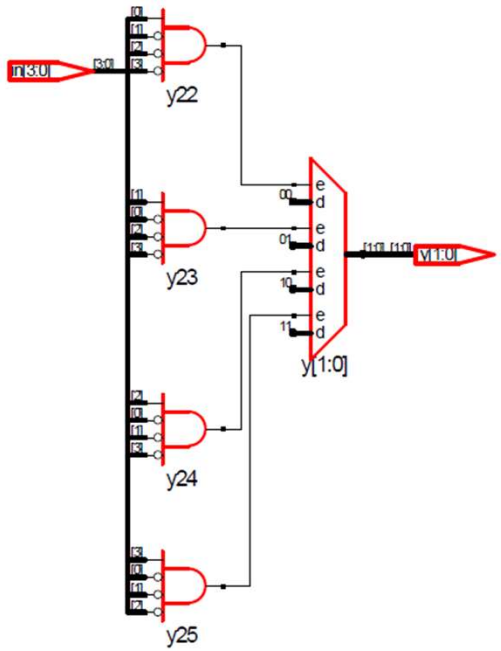
I_3	I_2	I_1	I_0	Y_1	Y_0
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

(a) Function table



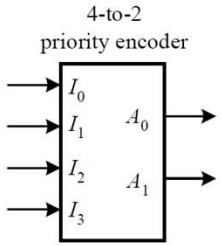
(b) Logic circuit

```
// a 4-to-2 encoder 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
```



Encoder

A 4-to-2 Priority Encoder

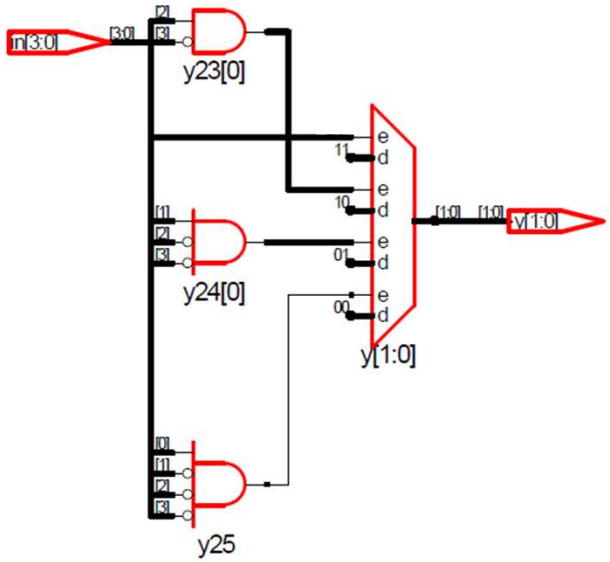


(a) Block diagram

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

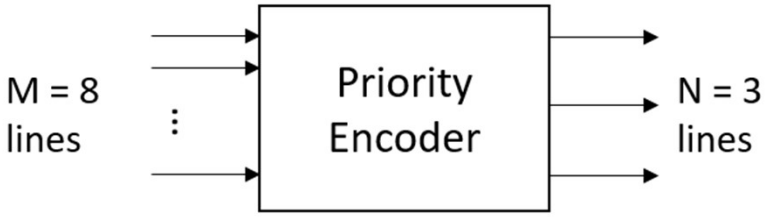
```
// a 4-to-2 priority encoder using case structure
module priority_encoder_4to2_case(in,          y);
input  [3:0] in;
output reg [1:0] y;

// the body of the 4-to-2 priority encoder

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
```



Encoder

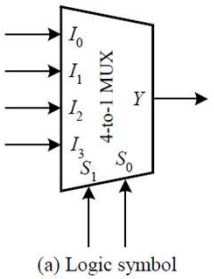
// an m-to-n priority encoder

```
module priencoder_m2n(x, y);
parameter m = 8; // number of inputs
parameter n = 3; // number of outputs
input [m-1:0] x;
output reg [n-1:0] y;
integer i;
// the body of the m-to-n priority encoder
always @(*)
begin //check_for_1
for (i = m -1 ; i > 0 ; i = i -1)
if (x[i] == 1)
y = i;
else y = 0;
end
endmodule
```




```
always @(*)
begin
i = m -1 ;
while(x[i] == 0 && i >= 0 )
i = i -1;
y = i;
end
```

Multiplexer



(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

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

```
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
```

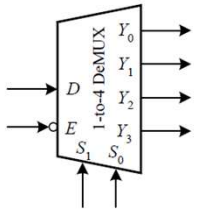
```
always @(select or in0 or in1 or in2 or in3)
y = select[1] ?(select[0] ? in3 : in2) :(select[0] ? in1 : in0) ;
```

```
integer i;
always @(*)
    for (i = 0; i < M; i = i + 1)
        if (select == i) y = in[i];
endmodule
```

```
always @(*)
    case (select)
        2'b11: y = in3 ;
        2'b10: y = in2 ;
        2'b01: y = in1 ;
        2'b00: y = in0 ;
    endcase
```

Demultiplexer

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



(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

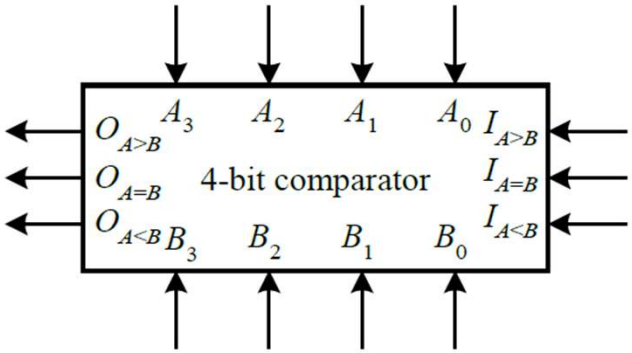
(b) Function table

```
module demux(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
```

```
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
```

Comparator

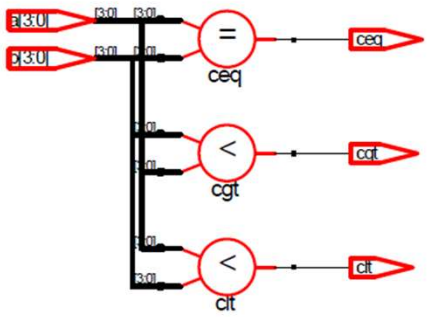
- A magnitude comparator is a device that determines the relative magnitude of two numbers being applied to it.
- Two types of magnitude comparator circuits:
  - Comparator
  - Cascadable comparator



A 4-bit cascadable comparator block diagram

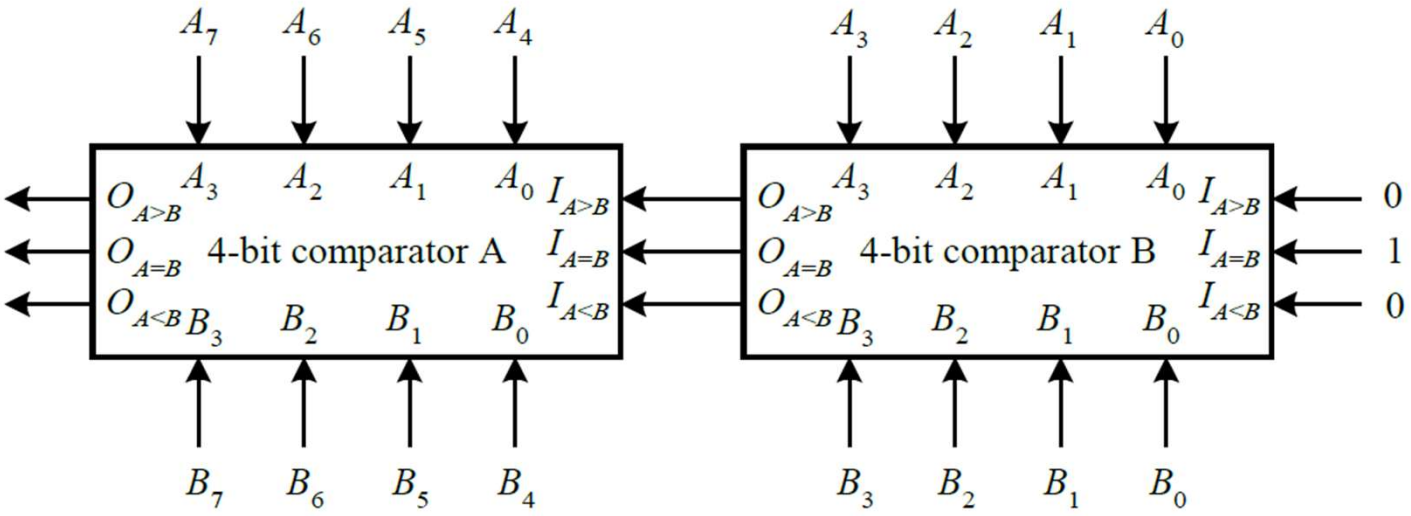
Comparator

```
// 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
```



Comparator

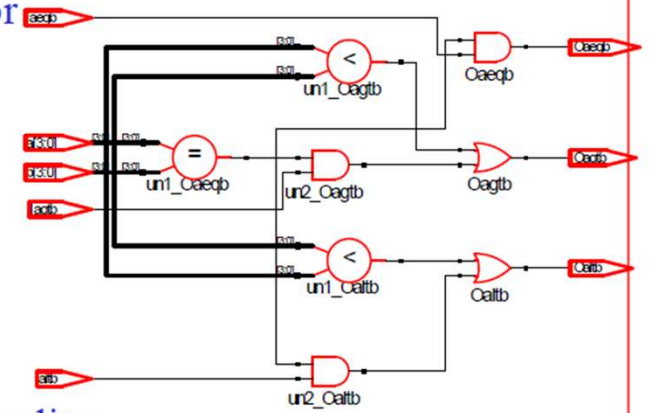
- Cascading two 4-bit comparators to form an 8-bit comparator.



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

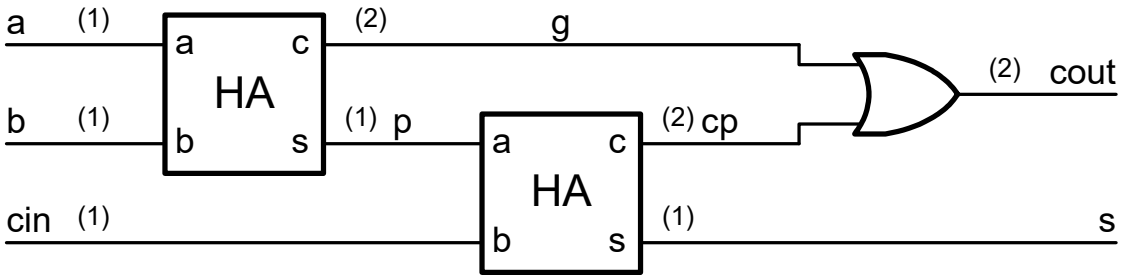
Comparator

```
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
```



Combinational Circuit Design

Adder




```
// full adder - from half adders
module FullAdder1(a,b,cin,cout,s) ;
  input a,b,cin ;
  output cout,s ;  // carry and sum
  wire g,p ;     // generate and propagate
  wire cp ;
  HalfAdder ha1(a,b,g,p) ;
  HalfAdder ha2(cin,p,cp,s) ;
  or o1(cout,g,cp) ;
endmodule
```

```
# 000 00
# 001 01
# 010 01
# 011 10
# 100 01
# 101 10
# 110 10
# 111 11
```

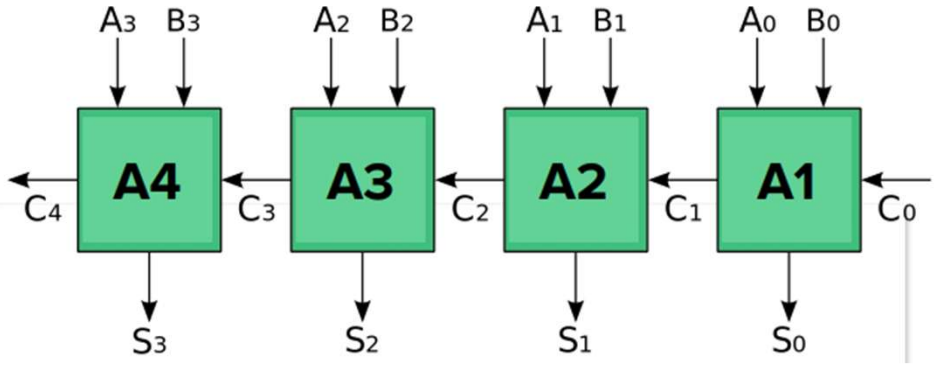
Adder

Implement a n-Bit Adder

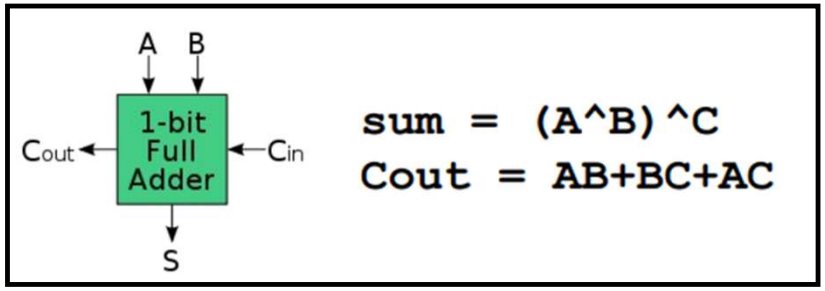
```
// multi-bit adder - behavioral
module Adder1(a,b,cin,cout,s) ;
  parameter n = 8 ;
  input [n-1:0] a, b ;
  input cin ;
  output [n-1:0] s ;
  output cout ;
  assign {cout, s} = a + b + cin ;
endmodule
```

Adder

Implement a 4-Bit Ripple Carry Adder



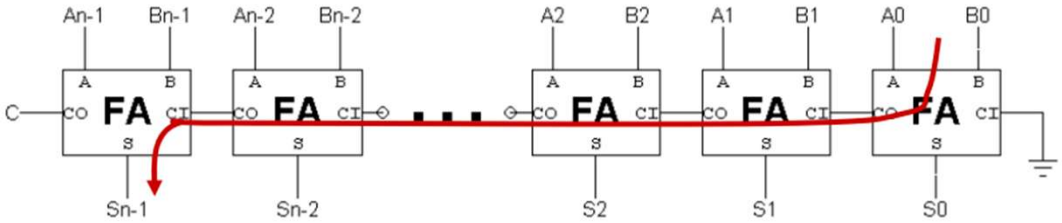
A	B	C	S	CO
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



Adder

Speed: t_{PD} of Ripple-carry Adder

$$C_O = AB + AC_I + BC_I$$



Worst-case path: carry propagation from LSB to MSB, e.g., when adding 11…111 to 00…001.

$$t_{PD} = (N-1)*(t_{PD,OR} + t_{PD,AND}) + t_{PD,XOR} \approx \Theta(N)$$

$t_{PD,OR} + t_{PD,AND}$: CI to CO

$t_{PD,XOR}$: CI_{N-1} to S_{N-1}

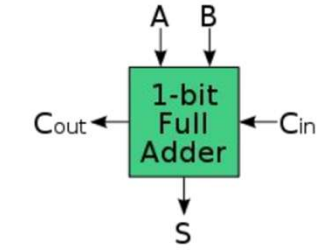
$$t_{adder} = (N-1)t_{carry} + t_{sum}$$

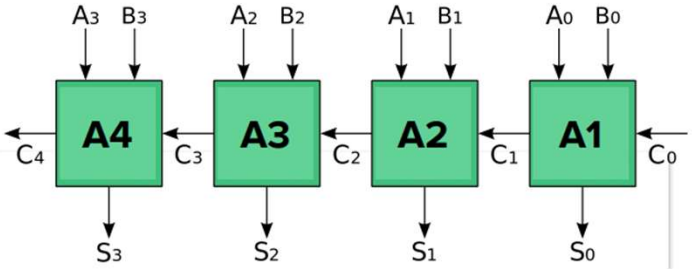
Θ(N) is read "order N" : means that the latency of our adder grows at worst in proportion to the number of bits in the operands.

Adder

```
module Four_Bit_RCA(
    input [3:0] A,
    input [3:0] B,
    input Cin,
    output [3:0] S,
    output Cout
    );
    wire [2:0] Carries;
    Full_Adder A1(.A(A[0]),.B(B[0]),.Cin(Cin),.S(S[0]),.Cout(Carries[0]));
    Full_Adder A2(.A(A[1]),.B(B[1]),.Cin(Carries[0]),.S(S[1]),.Cout(Carries[1]));
    Full_Adder A3(.A(A[2]),.B(B[2]),.Cin(Carries[1]),.S(S[2]),.Cout(Carries[2]));
    Full_Adder A4(.A(A[3]),.B(B[3]),.Cin(Carries[2]),.S(S[3]),.Cout(Cout));
endmodule
```

```
module Full_Adder(
    input A,
    input B,
    input Cin,
    output S,
    output Cout
    );
    assign S = (A^B)^Cin;
    assign Cout = (A&B)|(B&Cin)|(Cin&A);
endmodule
```





Adder

Some Ideas to Implement:

- ❖ Carry Look-Ahead Adders
- ❖ BCD Adder

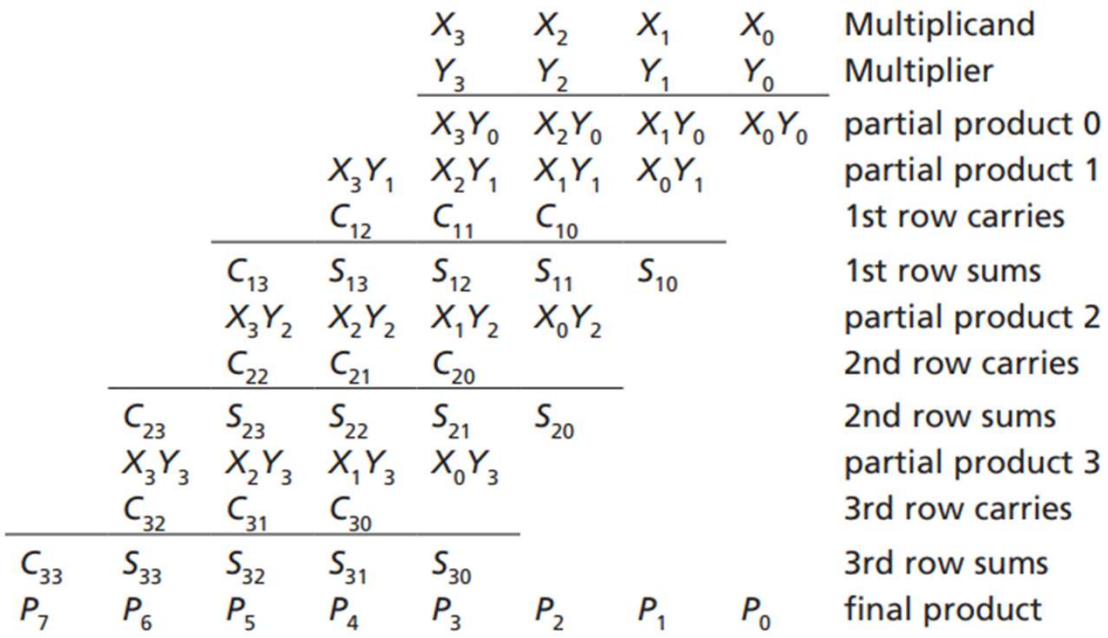
Adder

BCD Adder

```
module bcd_adder(a,b,carry_in,sum,carry);
    input [3:0] a,b;
    input carry_in;
    output [3:0] sum;
    output carry;
    //Internal variables
    reg [3:0] sum;
    reg carry;
    reg [4:0] sum_temp;
    always @(a,b,carry_in)
    begin
        sum_temp = a+b+carry_in; //add all the inputs
        if(sum_temp > 9)    begin
            sum_temp = sum_temp+6; //add 6, if result is more than 9.
            carry = 1;  //set the carry output
            sum = sum_temp[3:0];    end
        else    begin
            carry = 0;
            sum = sum_temp[3:0];
        end
    end
endmodule
```

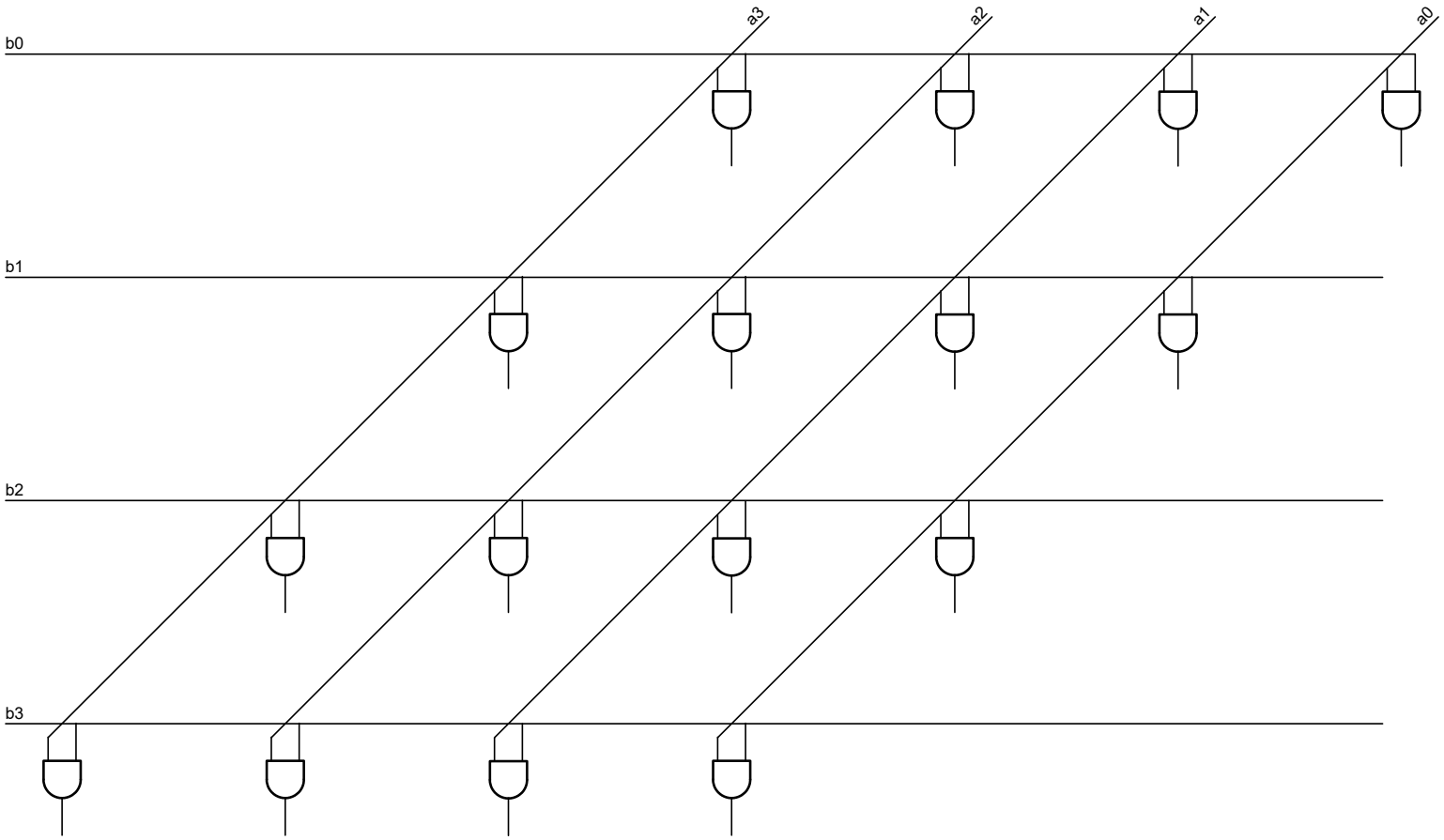
Multiplier

Array Multiplier



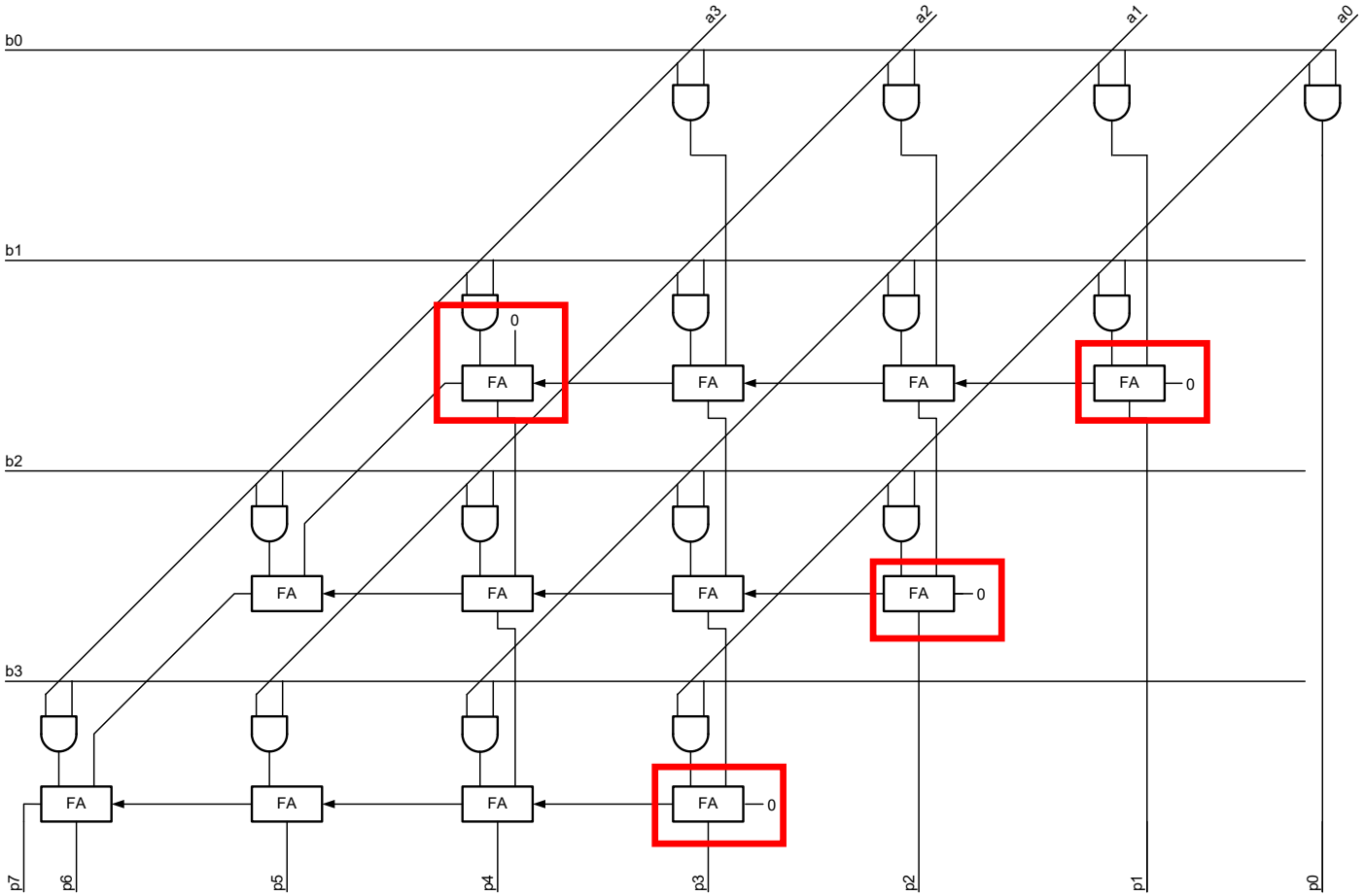
Multiplier

Array Multiplier



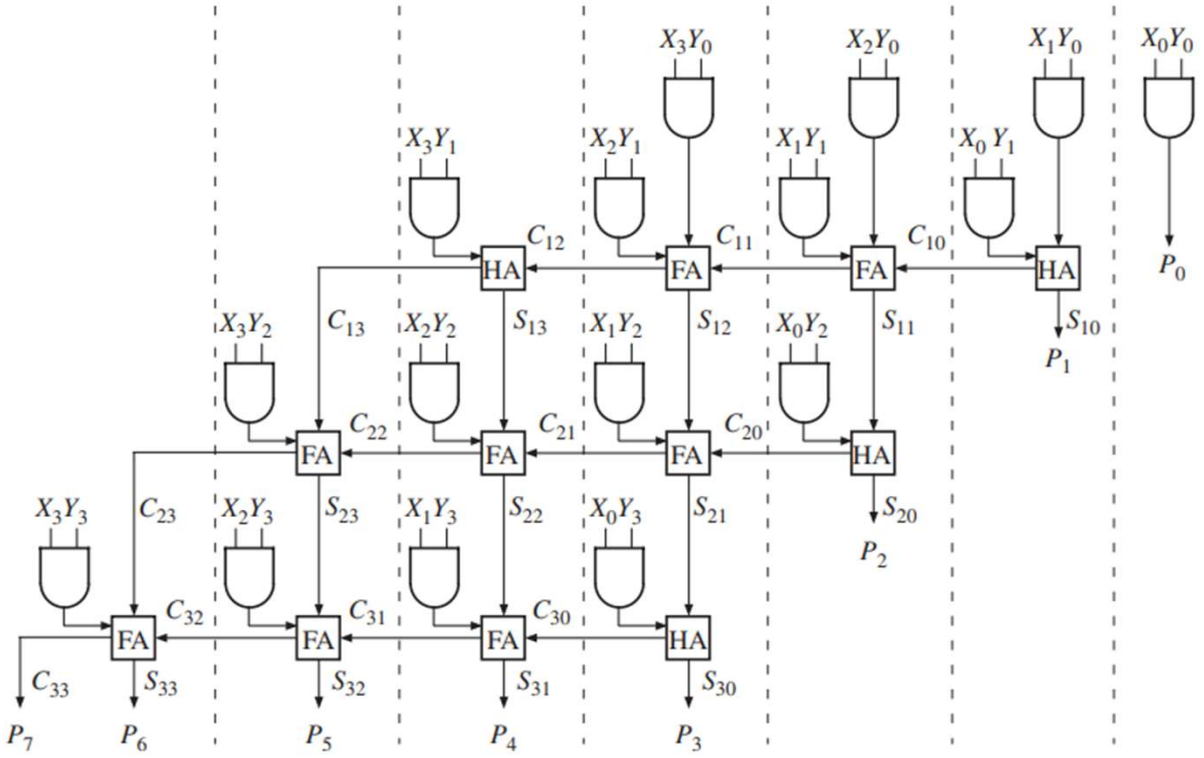
Multiplier

Array Multiplier



Multiplier

Array Multiplier



This multiplier requires 16 AND gates, 8 full adders, and 4 half-adders.

Multiplier

Array Multiplier

```
module Array_Mult (X, Y, P);

    input[3:0] X;
    input[3:0] Y;
    output[7:0] P;

    wire[3:0] C1;
    wire[3:0] C2;
    wire[3:0] C3;
    wire[3:0] S1;
    wire[3:0] S2;
    wire[3:0] S3;
    wire[3:0] XY0;
    wire[3:0] XY1;
    wire[3:0] XY2;
    wire[3:0] XY3;

    assign XY0[0] = X[0] & Y[0] ;
    assign XY1[0] = X[0] & Y[1] ;
    assign XY0[1] = X[1] & Y[0] ;
    assign XY1[1] = X[1] & Y[1] ;
    assign XY0[2] = X[2] & Y[0] ;
    assign XY1[2] = X[2] & Y[1] ;
    assign XY0[3] = X[3] & Y[0] ;
    assign XY1[3] = X[3] & Y[1] ;
    assign XY2[0] = X[0] & Y[2] ;
    assign XY3[0] = X[0] & Y[3] ;
    assign XY2[1] = X[1] & Y[2] ;
    assign XY3[1] = X[1] & Y[3] ;
    assign XY2[2] = X[2] & Y[2] ;
    assign XY3[2] = X[2] & Y[3] ;
    assign XY2[3] = X[3] & Y[2] ;
    assign XY3[3] = X[3] & Y[3] ;

    FullAdder FA1 (XY0[2], XY1[1], C1[0], C1[1], S1[1]);
    FullAdder FA2 (XY0[3], XY1[2], C1[1], C1[2], S1[2]);
    FullAdder FA3 (S1[2], XY2[1], C2[0], C2[1], S2[1]);
    FullAdder FA4 (S1[3], XY2[2], C2[1], C2[2], S2[2]);
    FullAdder FA5 (C1[3], XY2[3], C2[2], C2[3], S2[3]);
    FullAdder FA6 (S2[2], XY3[1], C3[0], C3[1], S3[1]);
    FullAdder FA7 (S2[3], XY3[2], C3[1], C3[2], S3[2]);
    FullAdder FA8 (C2[3], XY3[3], C3[2], C3[3], S3[3]);
    HalfAdder HA1 (XY0[1], XY1[0], C1[0], S1[0]);

    HalfAdder HA2 (XY1[3], C1[2], C1[3], S1[3]);
    HalfAdder HA3 (S1[1], XY2[0], C2[0], S2[0]);
    HalfAdder HA4 (S2[1], XY3[0], C3[0], S3[0]);

    assign P[0] = XY0[0] ;
    assign P[1] = S1[0] ;
    assign P[2] = S2[0] ;
    assign P[3] = S3[0] ;
    assign P[4] = S3[1] ;
    assign P[5] = S3[2] ;
    assign P[6] = S3[3] ;
    assign P[7] = C3[3] ;
endmodule
```

```
// Full Adder and half adder modules
// should be in the project

module FullAdder (X, Y, Cin, Cout, Sum);

    input X;
    input Y;
    input Cin;
    output Cout;
    output Sum;

    assign Sum = X ^ Y ^ Cin ;
    assign Cout = (X & Y) | (X & Cin) | (Y & Cin) ;
endmodule

module HalfAdder (X, Y, Cout, Sum);

    input X;
    input Y;
    output Cout;
    output Sum;

    assign Sum = X ^ Y ;
    assign Cout = X & Y ;
endmodule
```

Sequential Circuit Design

Sequential Circuit Design

Commonly used sequential logic modules:

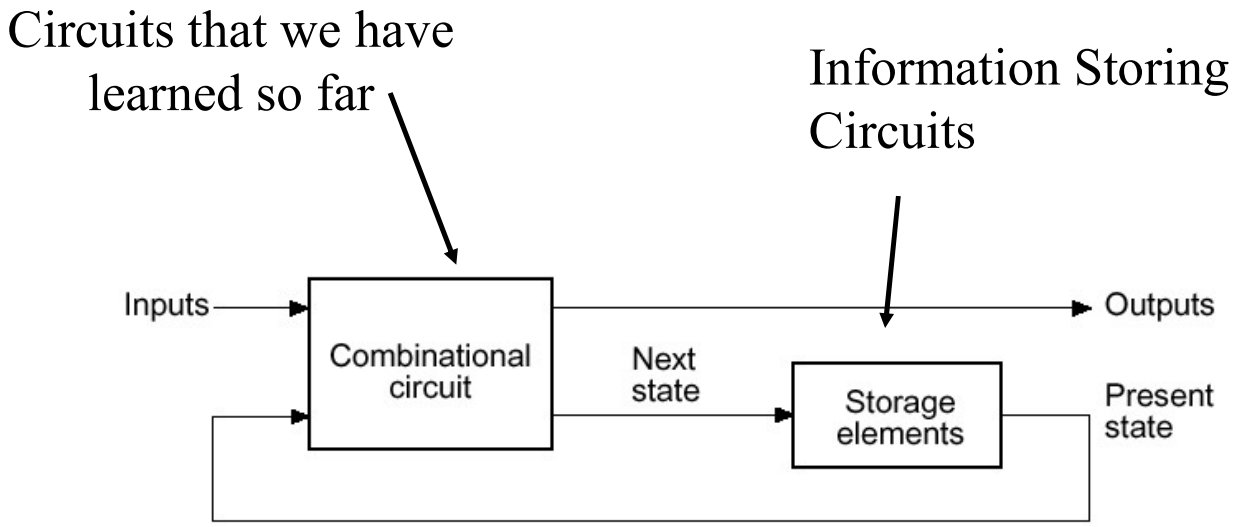
- ❖ Synchronizer
- ❖ Finite state machine
- ❖ Sequence detector
- ❖ Data register
- ❖ Shift register
- ❖ Register file
- ❖ Counters (binary, BCD, Johnson)
- ❖ Timing generator
- ❖ Clock generator
- ❖ Pulse generator

Sequential Logic

- Combinational Logic
  - Output depends only on current input
- Sequential Logic
  - Output depends not only on current input but also on past input values
  - Need some type of memory to remember the past input values

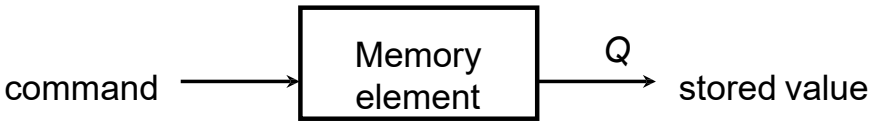
Sequential Circuit Design

A **sequential circuit** consists of a *feedback path*, and employs some *memory elements*.



Memory Elements

- **Memory element**: a device which can remember value indefinitely, or change value on command from its inputs.



- Characteristic table:

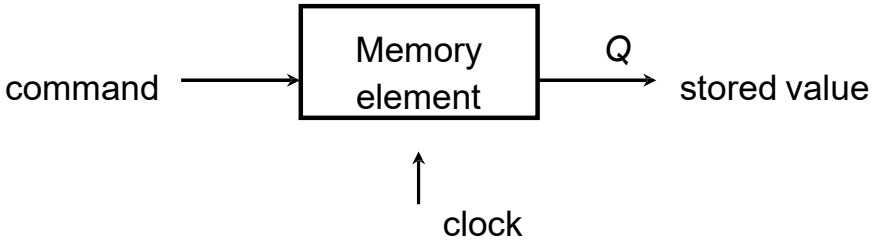
Command (at time *t*)	*Q(t)*	*Q(t+1)*
Set	X	1
Reset	X	0
Memorise / No Change	0	0
	1	1

Q(t): current state

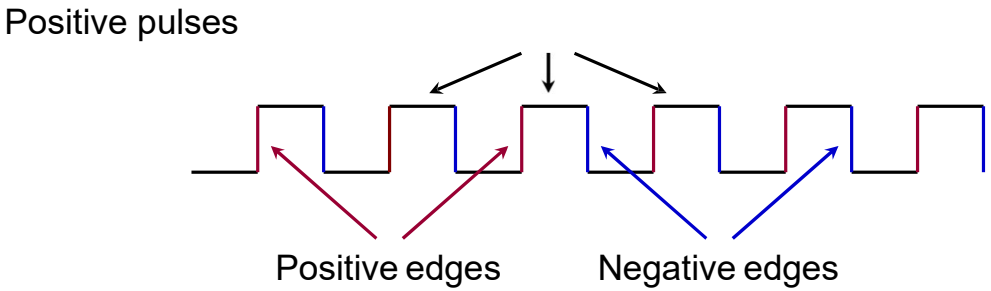
Q(t+1) or Q^+: next state

Triggering Mechanism

- **Memory element with clock**. Flip-flops are memory elements that change state on clock signals.

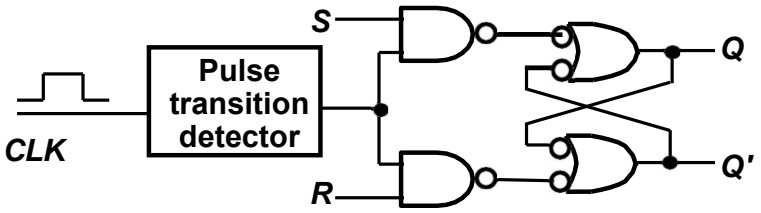


- Clock is usually a square wave.



S-R Flip-flop

- S-R flip-flop: on the triggering edge of the clock pulse,
  - *S*=HIGH (and *R*=LOW) a SET state
  - *R*=HIGH (and *S*=LOW) a RESET state
  - both inputs LOW a no change
  - both inputs HIGH a invalid
- Characteristic table of positive edge-triggered S-R flip- flop:



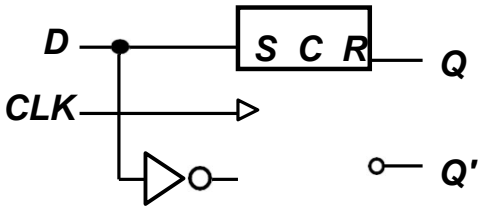
S	R	CLK	Q(t+1)	Comments
0	0	X	Q(t)	No change
0	1	↑	0	Reset
1	0	↑	1	Set
1	1	↑	?	Invalid

X = irrelevant ("don't care")

↑ = clock transition LOW to HIGH

D Flip-flop

- D flip-flop: single input *D* (data)
  - *D*=HIGH a SET state
  - *D*=LOW a RESET state
- *Q* follows *D* at the clock edge.
- Convert S-R flip-flop into a D flip-flop: add an inverter.



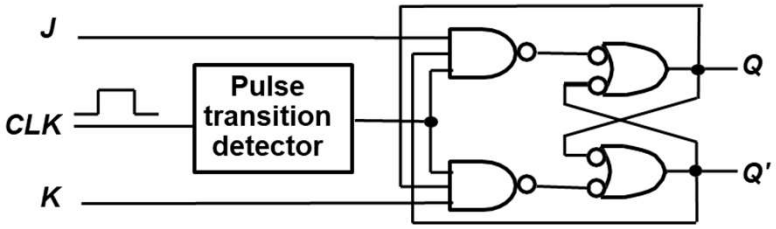
A positive edge-triggered D flip-flop formed with an S-R flip-flop.

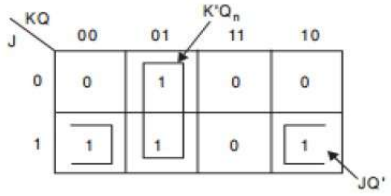
D	*CLK*	*Q(t+1)*	Comments
1	↑	1	Set
0	↑	0	Reset

↑ = clock transition LOW to HIGH

J-K Flip-flop

- No invalid state.
- Include a *toggle* state.
  - *J*=HIGH (and *K*=LOW) a SET state
  - *K*=HIGH (and *J*=LOW) a RESET state
  - both inputs LOW a no change
  - both inputs HIGH a toggle





$Q(t+1) = J.Q' + K'.Q$

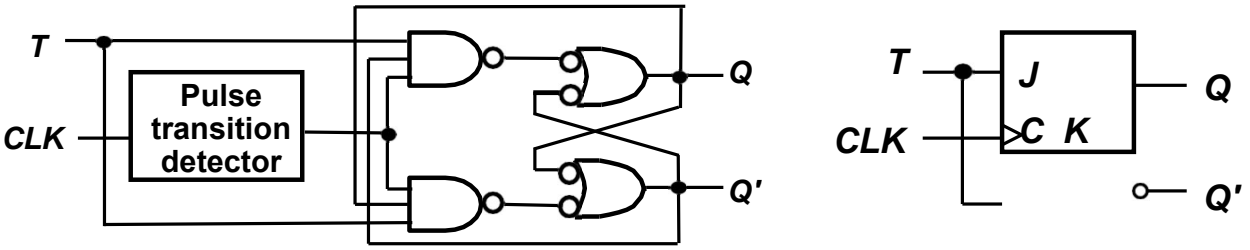
- **Characteristic table**

J	*K*	*CLK*	*Q(t+1)*	Comments
0	0	↑	*Q(t)*	No change
0	1	↑	0	Reset
1	0	↑	1	Set
1	1	↑	*Q(t)'*	Toggle

Q	*J*	*K*	*Q(t+1)*
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

T Flip-flop

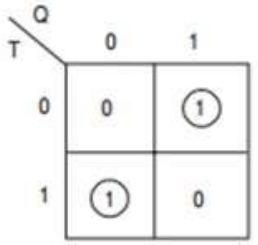
- T flip-flop: single-input version of the J-K flip flop, formed by connecting both inputs together.



- **Characteristic table.**

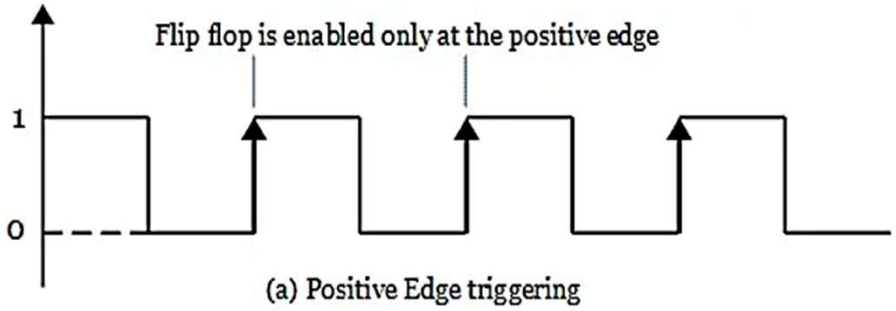
T	CLK	Q(t+1)	Comments
0	↑	Q(t)	No change
1	↑	Q(t)'	Toggle

Q	T	Q(t+1)
0	0	0
0	1	1
1	0	1
1	1	0



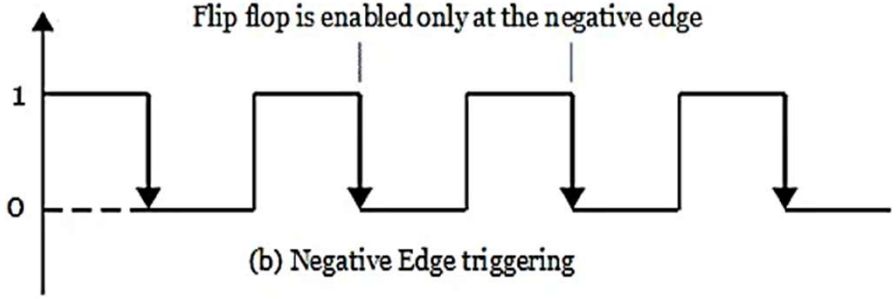
$Q(t+1) = T.Q' + T'.Q$

Triggering Mechanism



(a) Positive Edge triggering

```
always @ (posedge clk)
Begin
.
.
end
```



(b) Negative Edge triggering

```
always @ (negedge clk)
Begin
.
.
end
```

Triggering Mechanism

Synchronous Active Low Reset

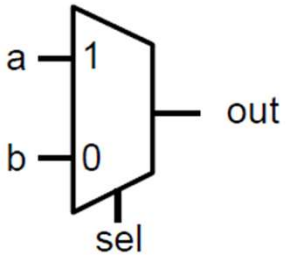

```
always @ (posedge clk)
Begin
if (!reset)
        q <= 0;
        else
        ....
        ...
        end
```

Asynchronous Active high Reset


```
always @ (posedge clk or posedge reset)
begin
   if(reset)
      q <= 0;
   else
...
...
      end
```

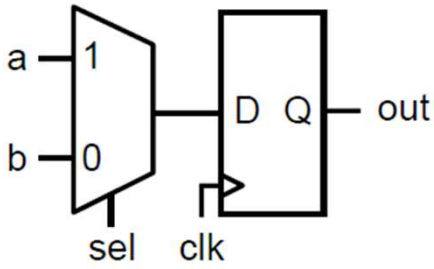
Combinational vs Sequential

Combinational




```
module combinational(a, b, sel,
                                out);
  input a, b;
  input sel;
  output out;
  reg out;
  always @ (a or b or sel)
  begin
    if (sel) out = a;
    else out = b;
  end
endmodule
```

Sequential




```
module sequential(a, b, sel,
                        clk, out);
  input a, b;
  input sel, clk;
  output out;
  reg out;
  always @ (posedge clk)
  begin
    if (sel) out <= a;
    else out <= b;
  end
endmodule
```

Blocking and Nonblocking Statement

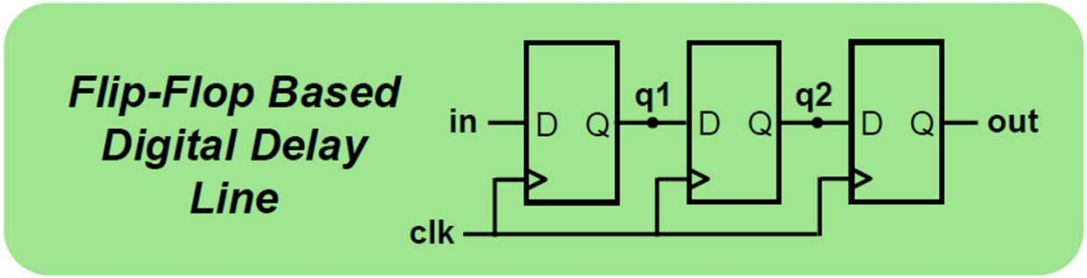
- ***Blocking assignment:*** **evaluation and assignment are immediate**

```
always @ (a or b or c)
begin
  x = a | b;          1. Evaluate a | b, assign result to x
  y = a ^ b ^ c;      2. Evaluate a^b^c, assign result to y
  z = b & ~c;         3. Evaluate b&(~c), assign result to z
end
```

- ***Nonblocking assignment:*** **all assignments deferred until all right-hand sides have been evaluated (end of simulation timestep)**

```
always @ (a or b or c)
begin
  x <= a | b;         1. Evaluate a | b but defer assignment of x
  y <= a ^ b ^ c;     2. Evaluate a^b^c  but defer assignment of y
  z <= b & ~c;        3. Evaluate b&(~c) but defer assignment of z
end                   4. Assign x, y, and z with their new values
```

Blocking and Nonblocking Statement



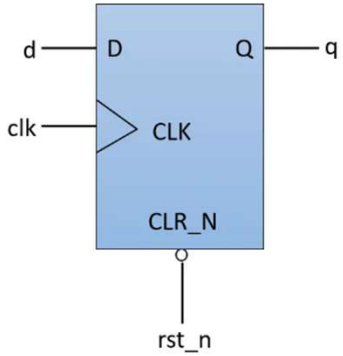
- Will nonblocking and blocking assignments both produce the desired result?

```
module nonblocking(in, clk, out);
  input in, clk;
  output out;
  reg q1, q2, out;

  always @ (posedge clk)
  begin
    q1 <= in;
    q2 <= q1;
    out <= q2;
  end

endmodule
```

```
module blocking(in, clk, out);
  input in, clk;
  output out;
  reg q1, q2, out;

  always @ (posedge clk)
  begin
    q1 = in;
    q2 = q1;
    out = q2;
  end

endmodule
```

Sequential Circuit Design

Design a positive edge-triggered D FF with synchronous active low reset.



Asynchronous active low reset D flip flop

```
module flipflop (D, Clock, Resetn, Q);
        input D, Clock, Resetn;
        output Q;
        reg Q;
        always @(posedge Clock)
        if (!Resetn)
        Q <= 0;
        else
        Q <= D;
endmodule
```

```
` `timescale 1ns/1ns
`include "flipflop.v"    //Name of the Verilog file

module flipflop_tb1();
reg D, Clock, Resetn;    //Input
wire Q;                  //Output

flipflop f1(D, Clock, Resetn, Q);//Instantiation of the module
initial
begin

        $dumpfile("flipflop_tb1.vcd");
        $dumpvars(0, flipflop_tb1);

 Clock=0;
   forever #20 Clock = ~Clock;
end
initial begin

        D=1;  Resetn=1;
        #20;
        D=1; Resetn=1;
        #20;
        D=1;  Resetn=1;
        #20;
        D=0; Resetn=1;
        #20;
        $display("Test complete");
end
endmodule
```

Sequential Circuit Design

Design a **positive edge-triggered** D FF with **synchronous active high reset**.

D	Clk	Reset	Q	Qbar
0	↑	0	0	1
1	↑	0	1	0
0	↑	0	0	1
1	↑	0	1	0
0	↑	1	0	1
1	↑	1	0	1
0	↑	1	0	1
1	↑	1	0	1

```
always @(posedge Clk)
        if (Reset)
                Q <= 0;
        else
                Q <= D;
endmodule
```

```
module d_ff (clk, reset, d, q);
    input clk;    // Clock input
    input reset;  // Synchronous active-high reset
    input d;      // Data input
    output reg q;       // Output
always @(posedge clk) begin
    if (reset)        // Check if reset is active
        q <= 1'b0;     // Set output to 0 if reset is high
    else
        q <= d;        // Otherwise, set output to input data
end
endmodule
```

Sequential Circuit Design

Design a **positive edge-triggered** D FF with **Asynchronous active high reset.**

```
module d_ff (clk, reset, d, q);
  input clk;    // Clock input
  input reset;  // Synchronous active-high reset
  input d;      // Data input
  output reg q;      // Output
always @(posedge clk)
begin
  if (reset)       // Check if reset is active
    q <= 1'b0;     // Set output to 0 if reset is high
  else
    q <= d;        // Otherwise, set output to input data
end
endmodule
```

always @(posedge clk or posedge reset

Sequential Circuit Design

Design a positive edge-triggered JK FF with asynchronous active high reset

```
always @ (posedge clk or posedge reset)
begin
  if(reset)       q <= 0;
  else
begin
            case({j,k})
             2'b00: q <= q;
            2'b01: q <= 0;
             2'b10: q <= 1;
            2'b11: q <= ~q;
       default:   q <= q;
     endcase
  end
end
endmodule
```

Clk	Reset	J	K	Output		Comments
				Q_n	$Q_n$bar	
↑	0	0	0	0	1	No change
↑	0	0	1	0	1	Reset
↑	0	1	0	1	0	Set
↑	0	1	1	$Q_n$bar	Q_n	Toggles
↑	1	x	x	0	1	Clear

Registers

Registers

- A register is a group of flip-flops suitable for storing binary information.

1-bit each stage

- A register capable of shifting binary information either to the right or to the left is called a *shift register*.
- In a shift register, the flip-flops are connected in such a way that the bits of a binary number are entered into the shift register, shifted from one position to another and finally shifted out.

  - **Serial Shifting**
  - **Parallel Shifting**

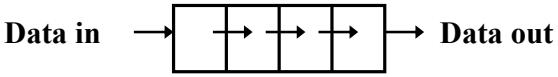
Types of shift registers

❖Shift registers are of four types :

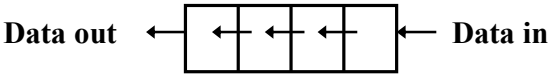
1. Serial-in Serial-out (SISO)
2. Serial-in Parallel-out (SIPO)
3. Parallel-in Serial-out (PISO)
4. Parallel-in Parallel-out (PIPO)

❖An *n*-bit shift register consists of *n* flip-flops and the required gates to control the shift operation.

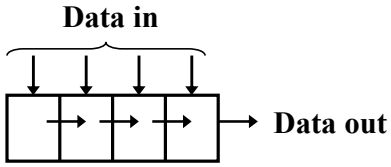
Types of shift registers



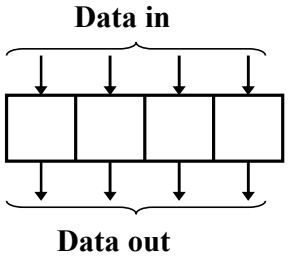
(a) Serial in/shift right/serial out



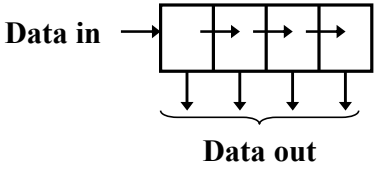
(b) Serial in/shift left/serial out



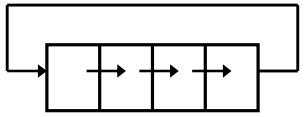
(c) Parallel in/serial out



(e) Parallel in/parallel out

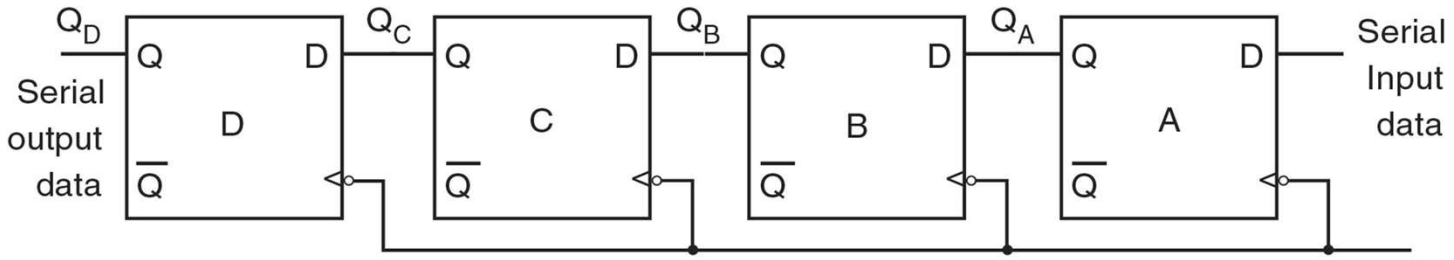


(d) Serial in/parallel out

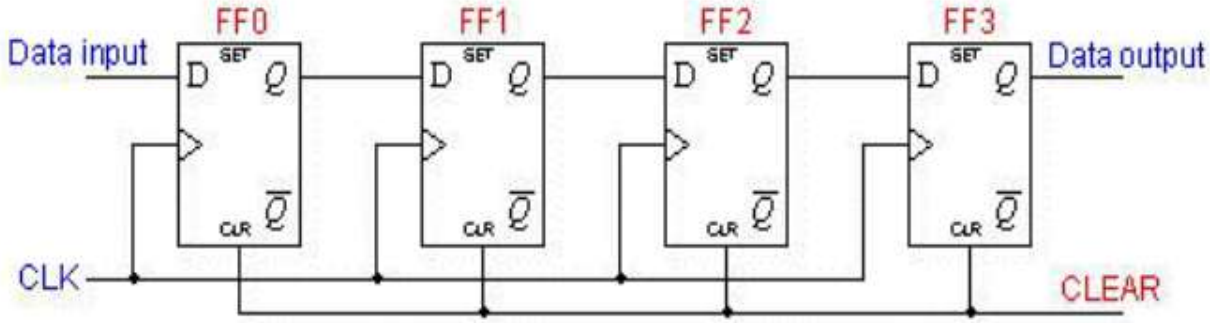


(f) Rotate right

SISO Shift Register

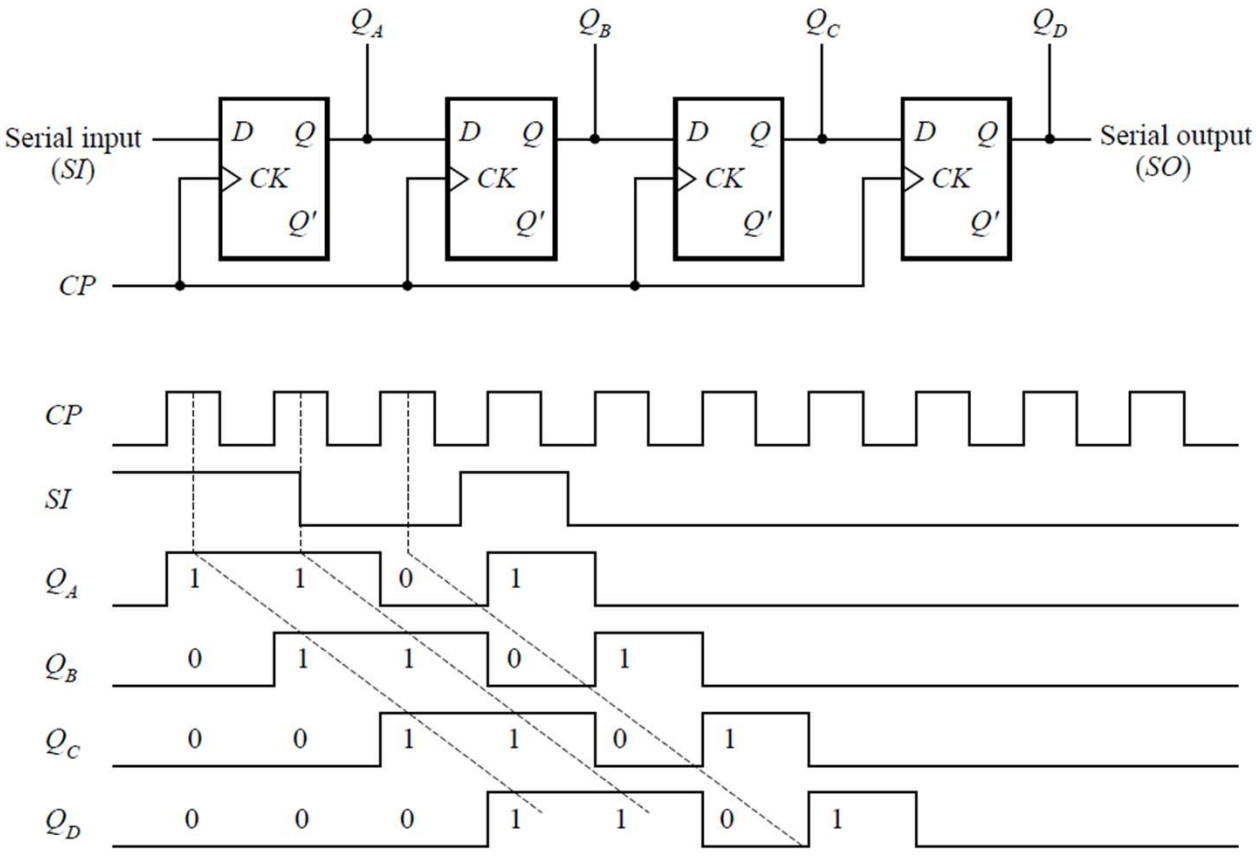


Shift Left Operation

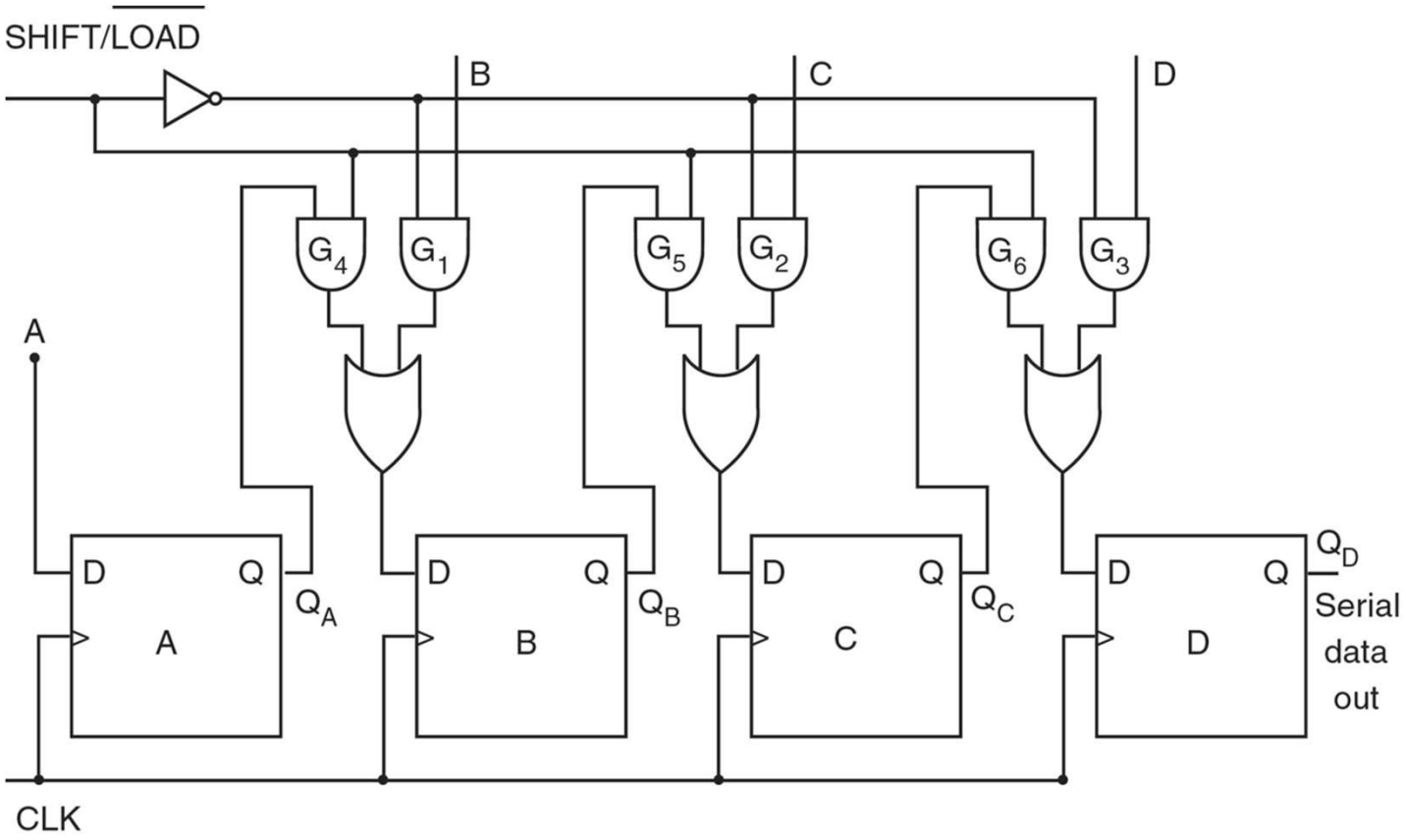


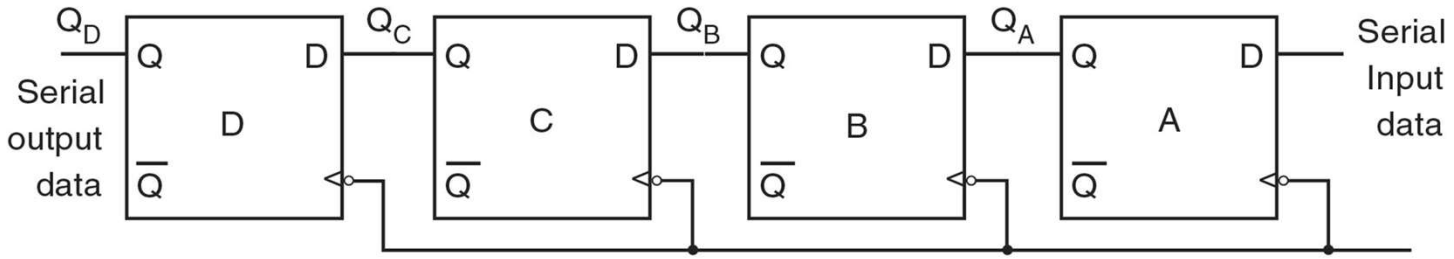
Shift Right Operation

Types of shift registers



PISO Shift Register






```
`timescale 1ns/1ps
module shift (input clk, input D, output reg [3:0]Q=0);
  always @(negedge clk)
        begin
        Q[D]<=Q[C];
        Q[C]<=Q[B];
        Q[A]<=Q[A];
        Q[A]<=D;
        end
endmodule
```

Sequential Circuit Design

Design a n-bit data register with asynchronous reset

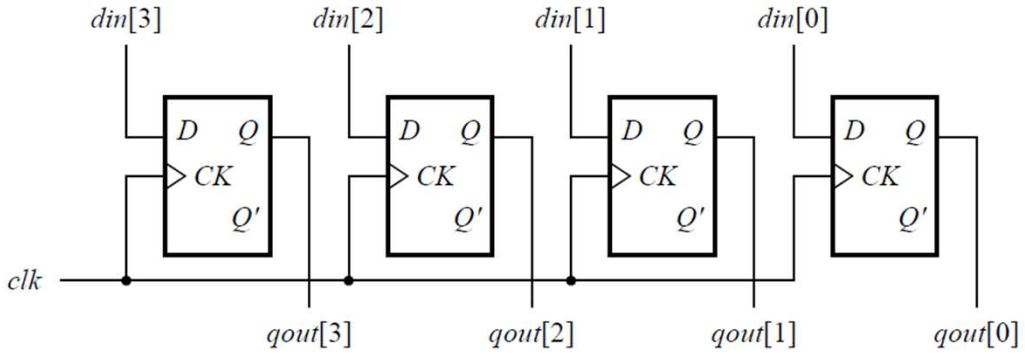
SISO

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

// the body of an N-bit shift register
always @(posedge clk or negedge reset_n)
  if (!reset_n) qout <= {N{1'b0}};
  else          qout <= {din, qout[N-1:1]};
endmodule
```

Sequential Circuit Design

PIPO can be utilized to design Memory

Data Registers




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

Sequential Circuit Design

Design a n-bit data register with asynchronous reset

PIPO

```
// an n-bit data register with asynchronous reset
module register_reset (clk, reset_n, din, qout);
parameter N = 4;   // number of bits
input   clk, reset_n;
input   [N-1:0] din;
output reg [N-1:0] qout;

// The body of an n-bit data register
always @(posedge clk or negedge reset_n)
  if (!reset_n) qout <= {N{1'b0}};
  else          qout <= din;
endmodule
```

Sequential Circuit Design

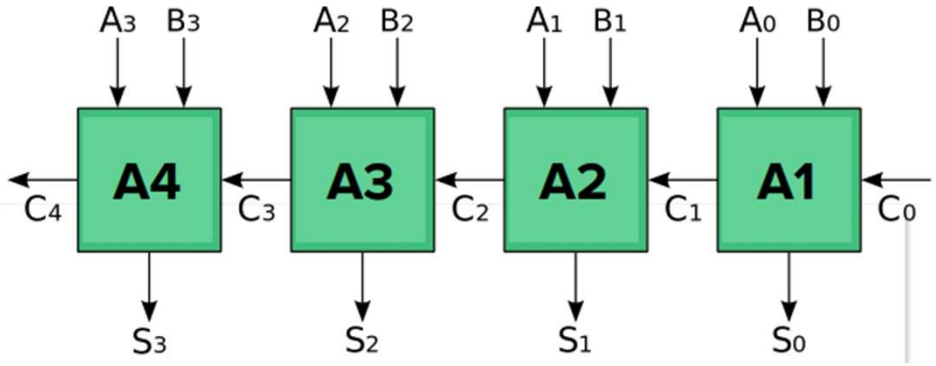
Design a n-bit data register with synchronous load and asynchronous reset

PIPO

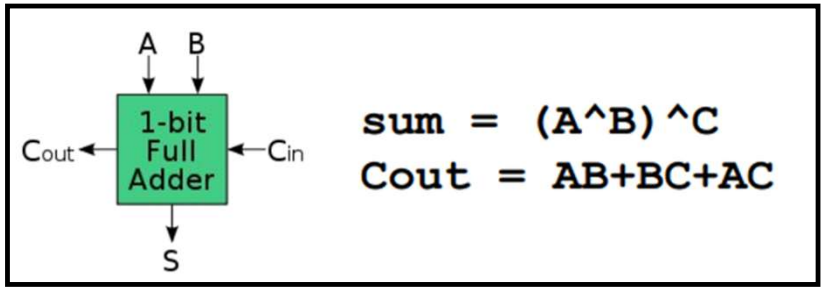
```
// an N-bit data register with synchronous load and
// asynchronous reset
module register_load_reset (clk, load, reset_n, din, qout);
parameter N = 4;   // number of bits
input  clk, load, reset_n;
input  [N-1:0] din;
output reg [N-1:0] qout;

// the body of an N-bit data register
always @(posedge clk or negedge reset_n)
  if (!reset_n)    qout <= {N{1'b0}};
  else  if (load)  qout <= din;
  else qout <= qout;                          // a redundant expression
endmodule
```

Adder

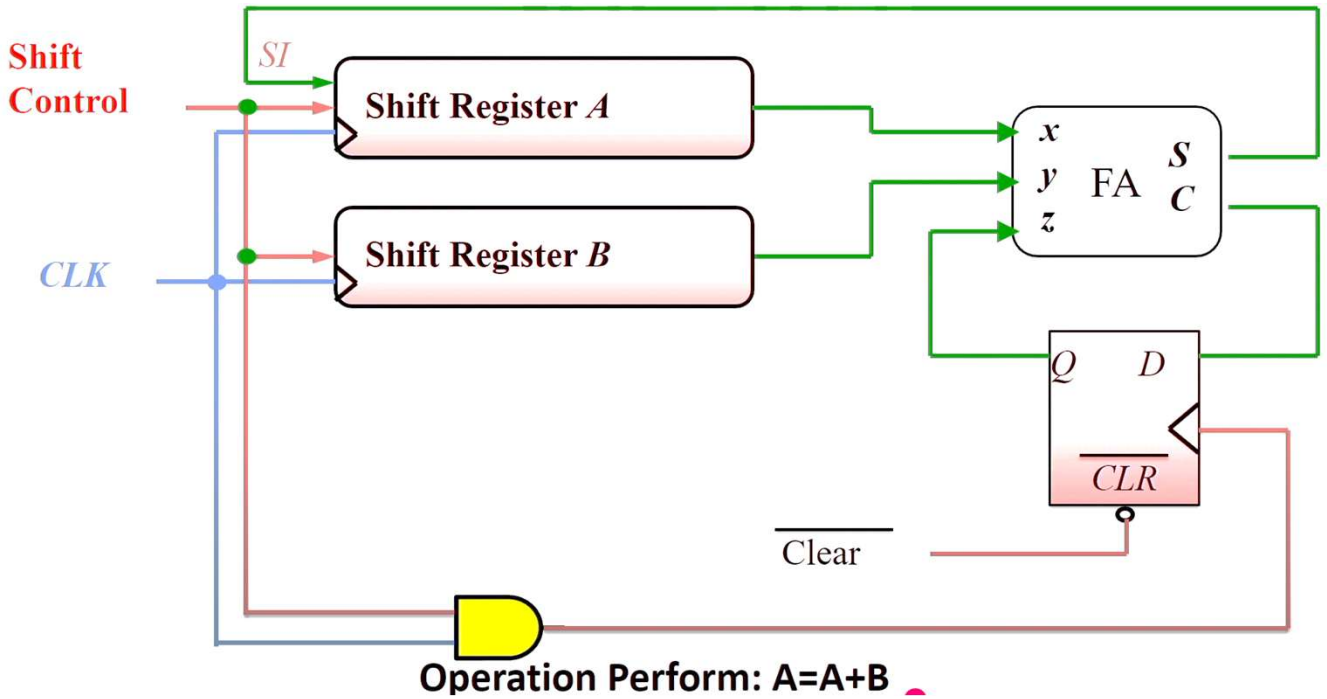
Implement a 4-Bit Ripple Carry Adder



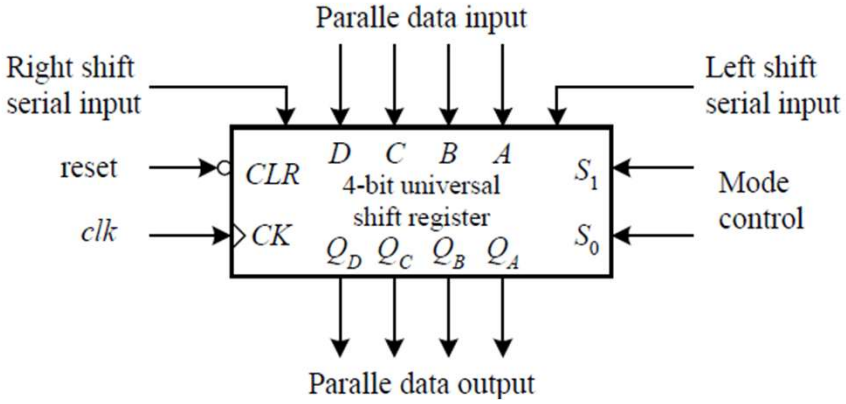
A	B	C	S	CO
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



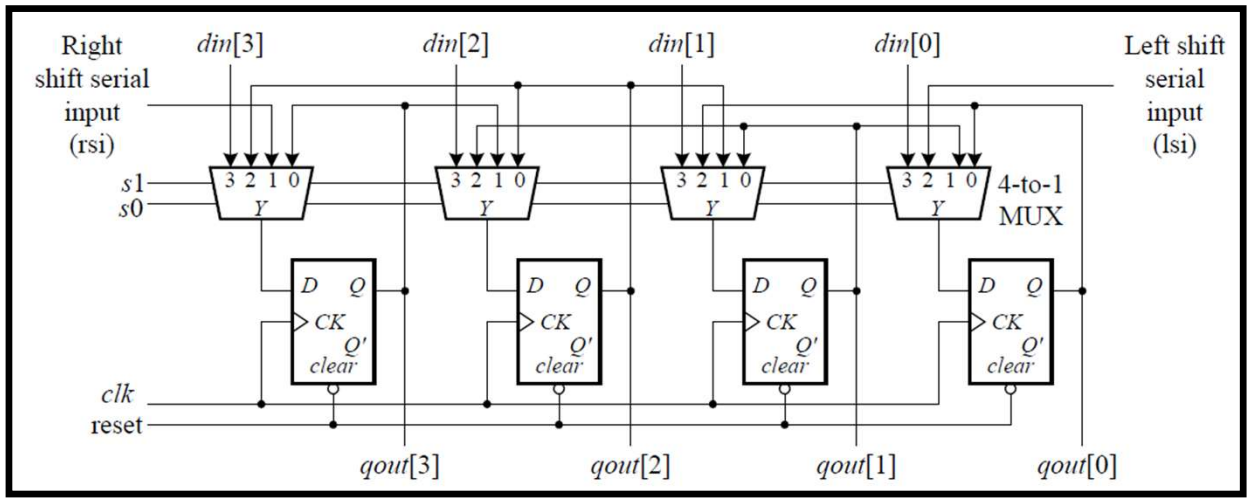
Sequential Circuit Design



Universal Shift Register



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



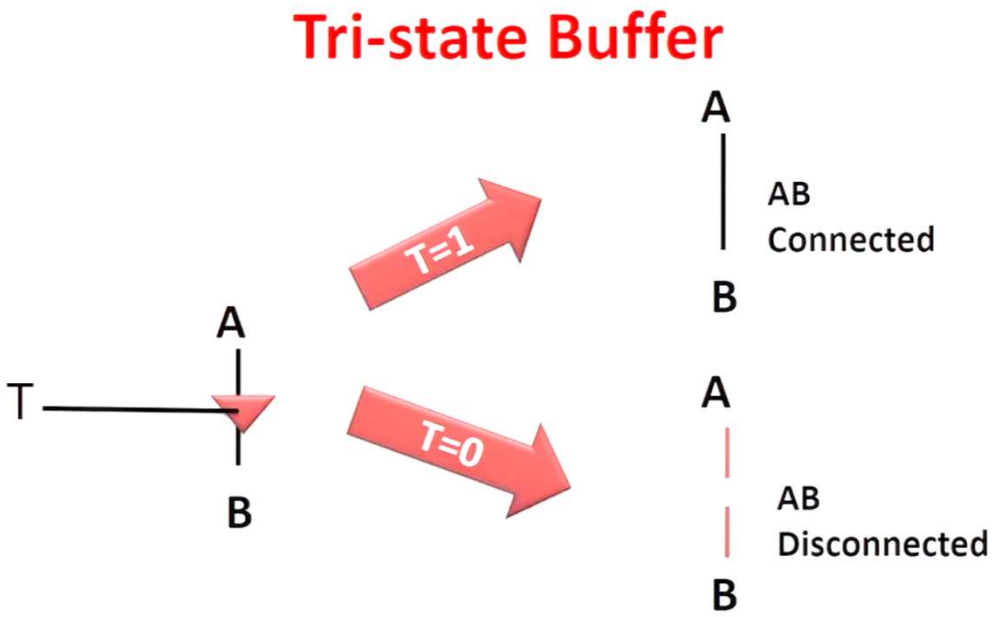
Universal Shift Register

```
// a universal shift register module
module universal_shift_register (clk, reset_n, s1, s0, lsi, rsi, din, qout);
parameter N = 4;  // define the size of the universal shift register
input  s1, s0, lsi, rsi, clk, reset_n;
input  [N-1:0] din;
output reg [N-1:0] qout;
// the shift register body
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
endmodule
```

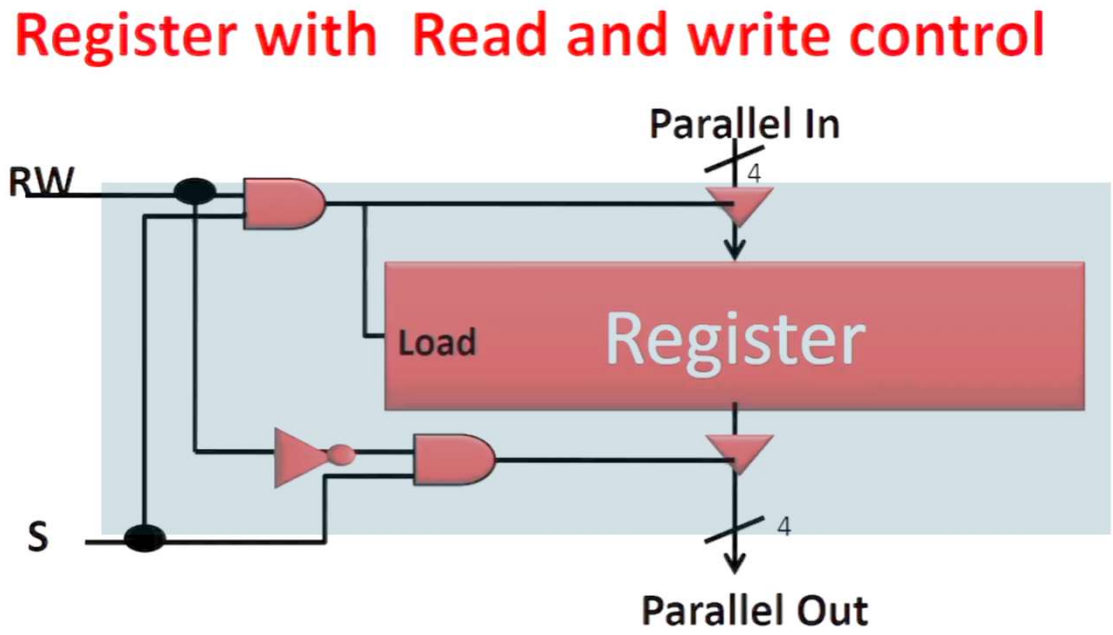
*s*1	*s*0	Function
0	0	No change
0	1	Right shift
1	0	Left shift
1	1	Load data

Shift Register

How to select a register from a group of registers??



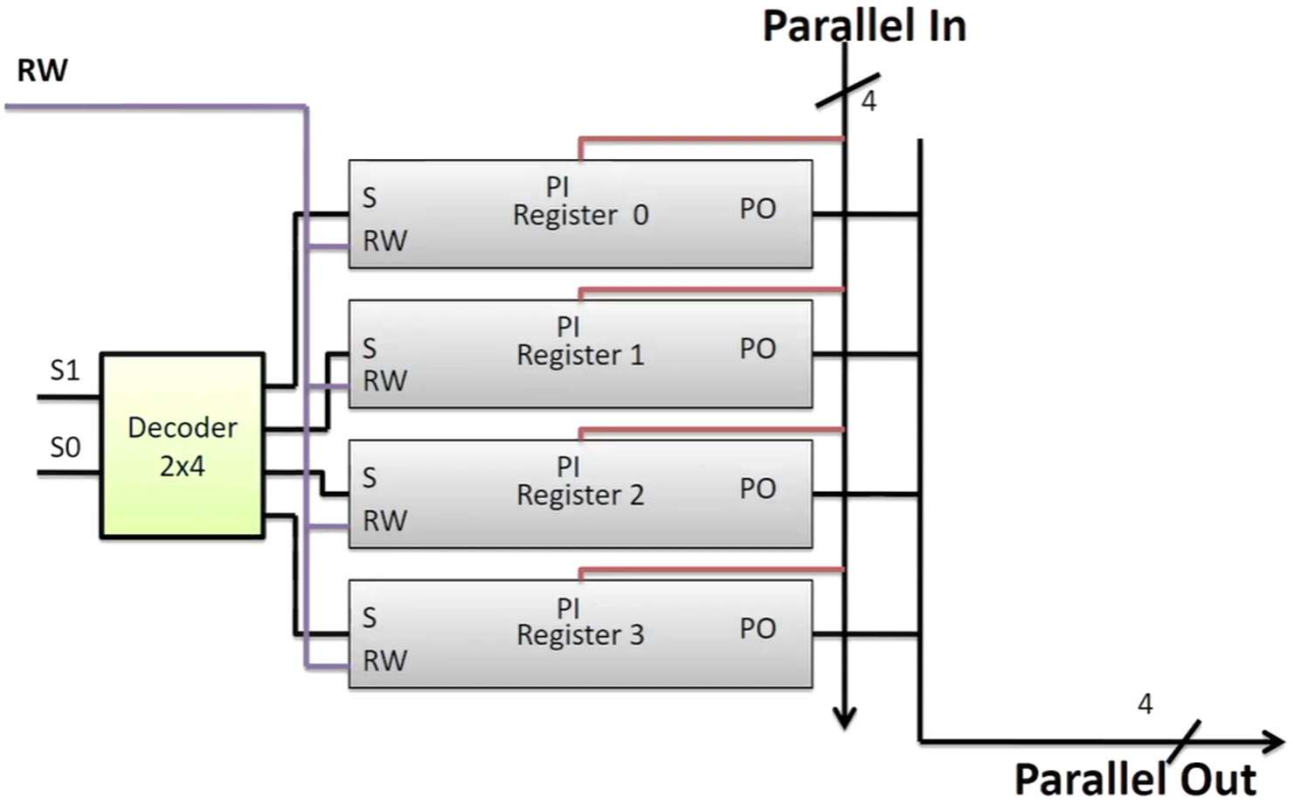
Shift Register - Memory



S = 0; Both TS1 and TS2 (Tristate Buffer) will be opened
S = 1; Depending on RW, read or write operation will be performed.
S = 1 and RW = 1: Write Operation
S = 1 and RW = 0; read Operation

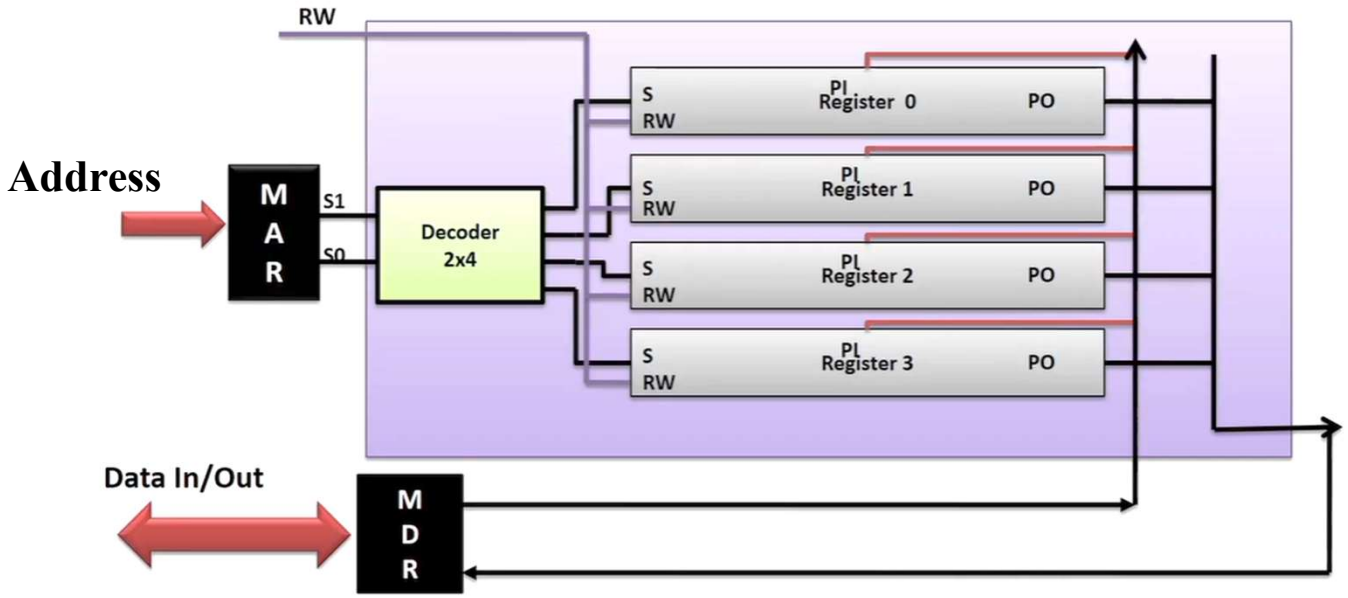
Shift Register - Memory

Many Register with RW control



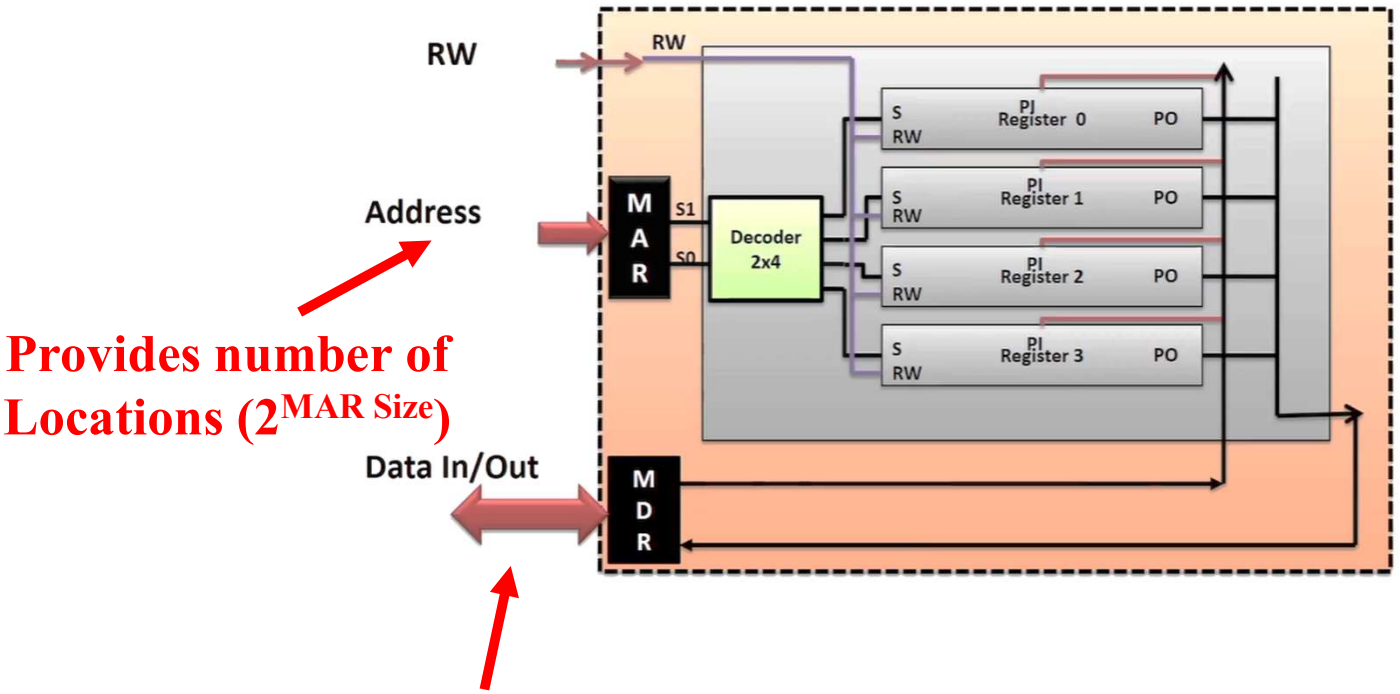
Memory - Cell

Memory with MAR and MDR



MAR: memory Address Register
MDR: Memory Data Register or Memory Buffer Register

Memory - Cell



Provides number of Locations ($2^{\text{MAR Size}}$)

The number of bits decides the word size

Synchronous RAM

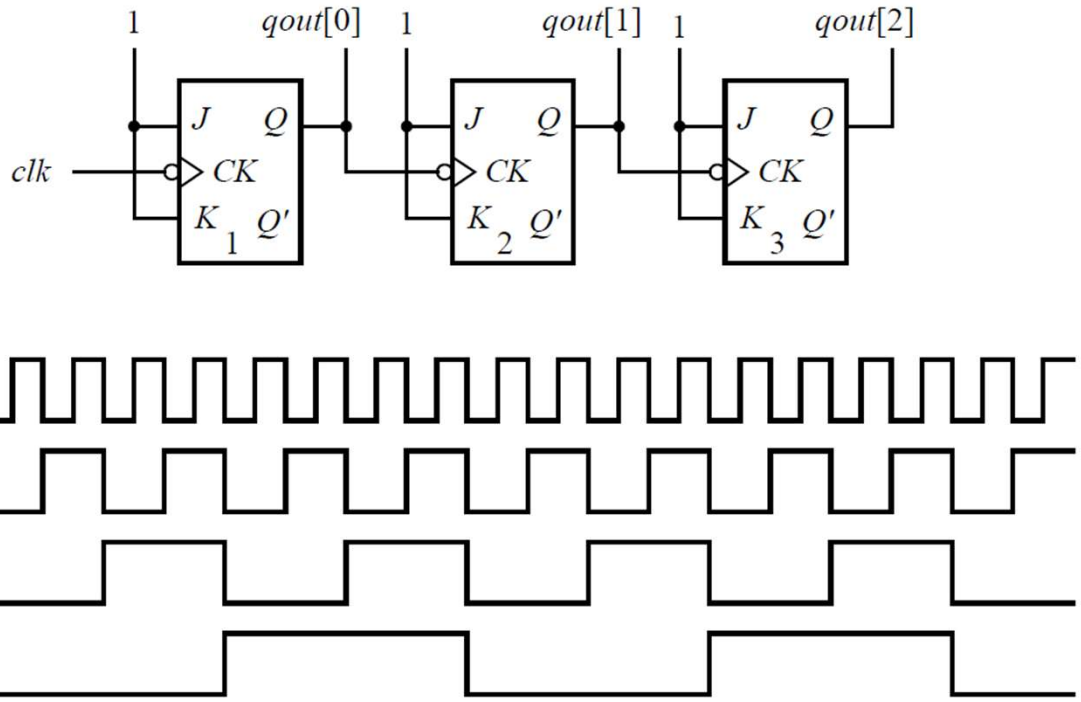
```
// a synchronous RAM module example
module syn_ram (addr, cs, din, clk, wr, dout);
parameter N = 16;   // number of words
parameter A = 4;    // number of address bits
parameter W = 4;    // number of wordsize in bits
input  [A-1:0]  addr;
input  [W-1:0] din;
input   cs, wr, clk;   // chip select, 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

// the body of synchronous RAM
always @(posedge clk)
    if (cs) if (wr) ram[addr] <= din;
            else            dout <= ram[addr];
endmodule
```

Counter

- Counter is a device that counts the input events such as input pulses or clock pulses.
- 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)

Counter

Binary Ripple Counters



Counter

Binary Ripple Counters

```
module ripple_counter(clk, qout);
input  clk;
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];
endmodule
```

initial values of qout??

Counter

// a 3-bit ripple counter with enable Control

```
module ripple_counter_enable(clk, enable, reset_n, qout);
input clk, enable, reset_n;
output reg [2:0] qout;
wire c0, c1;
// the body of the 3-bit ripple counter
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];
endmodule
```

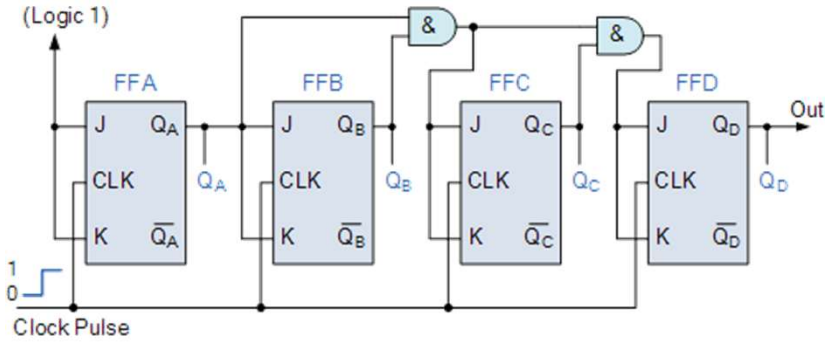
Can we make it more generalized?

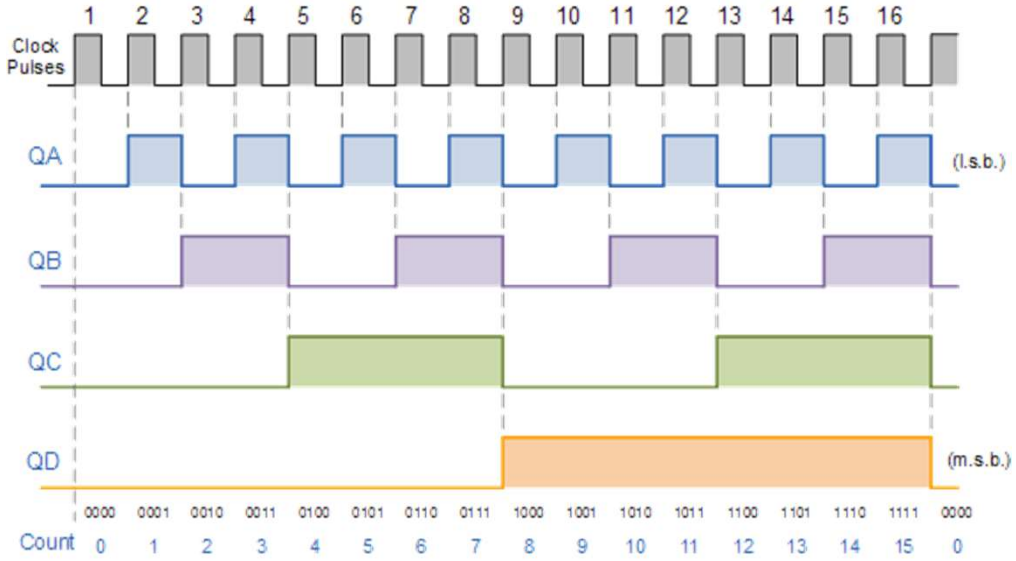
Counter

//An N-bit ripple counter using Loop

```
parameter N = 4; // define the size of counter
integer i;
for (i = 0; i < N; i = i + 1)
begin:
   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
endmodule
```

Synchronous Counter





Binary Counter

A n-bit binary counter with synchronous reset and enable control

```
module binary_counter(clk, enable, reset, qout, rco);
parameter N = 4;
input  clk, enable, reset;
output reg [N-1:0] qout;
output rco;
// the body of the N-bit binary counter
always @(posedge clk)
  if (reset)          qout <= {N{1'b0}};
  else if (enable)  qout <= qout + 1;
// generate carry output
assign #1 rco= &qout;          // why #1 is required?
endmodule
```

When rco is used as enable in a cascaded counter, possible hold time violation

Binary Counter

A n-bit binary counter with - Revised

```
module binary_counter(clk, enable, reset, qout, rco);
parameter N = 4;
input  clk, enable, reset;
output reg [N-1:0] qout;
output reg rco;
// the body of the N-bit binary counter
always @(posedge clk)
   if (reset)          qout <= {N{1'b0}};
   else if (enable)  qout <= qout + 1;
// generate carry output
always @(negedge clk)
   rco <= &qout;
endmodule
```

Binary Counter

// an N-bit binary up/down counter with synchronous reset and enable control

```
module binary_up_down_counter_reset(clk, enable, reset, upcnt, qout, cout, bout);
parameter N = 4;
input  clk, enable, reset, upcnt;
output reg [N-1:0] qout;
output cout, bout;  // carry and borrow outputs
// the body of N-bit up/down binary counter
always @(posedge clk)
  if (reset) qout <= {N{1'b0}};
  else if (enable) begin
            if (upcnt) qout <= qout + 1;
            else       qout <= qout - 1;  end
//      else  qout <= qout;  // a redundant expression
// Generate carry and borrow outputs
assign #1 cout = &qout;  // Why #1 is required ?
assign #1 bout = |qout;
endmodule
```

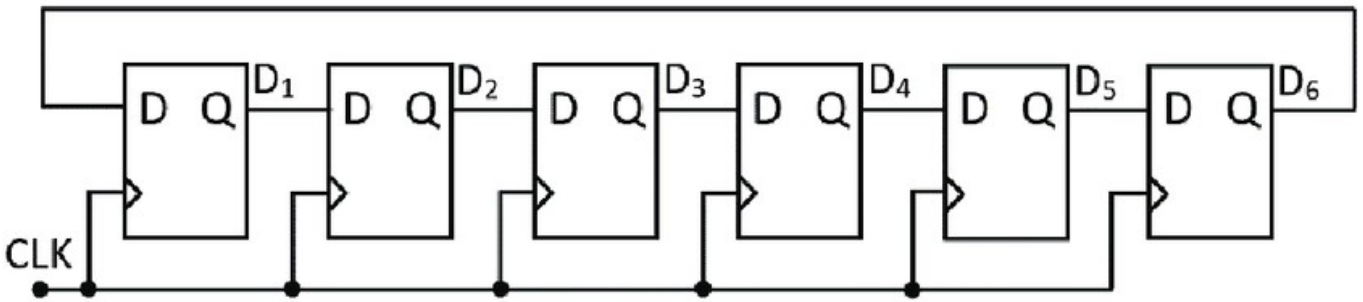
Binary Counter

// Design a BCD Counter or MOD-10 Counter or MOD-r Counter

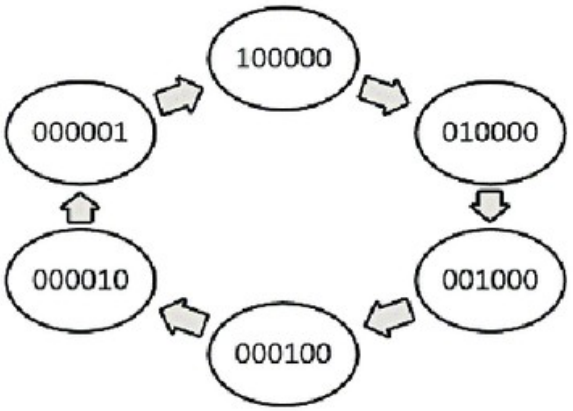
```
// the body of modulo r binary counter with synchronous reset and enable control.
module modulo_r_counter(clk, enable, reset, qout, rco);
parameter N = 4;
parameter R= 10;  // BCD counter
input  clk, enable, reset;
output reg [N-1:0] qout;
output rco;  // carry output
// the body of modulo r binary counter.
assign rco = (qout == R - 1);
always @(posedge clk)
    if (reset)  qout <= {N{1'b0}};
    else begin
        if (enable) if (rco) qout <= 0;
                    else     qout <= qout + 1;
    end
endmodule
```

Binary Counter

Ring Counters

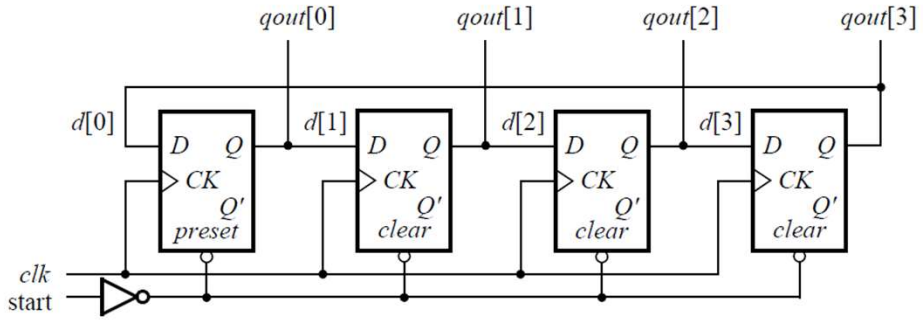


CLK	D_1	D_2	D_3	D_4	D_5	D_6
1	1	0	0	0	0	0
2	0	1	0	0	0	0
3	0	0	1	0	0	0
4	0	0	0	1	0	0
5	0	0	0	0	1	0
6	0	0	0	0	0	1



Binary Counter

Ring Counters

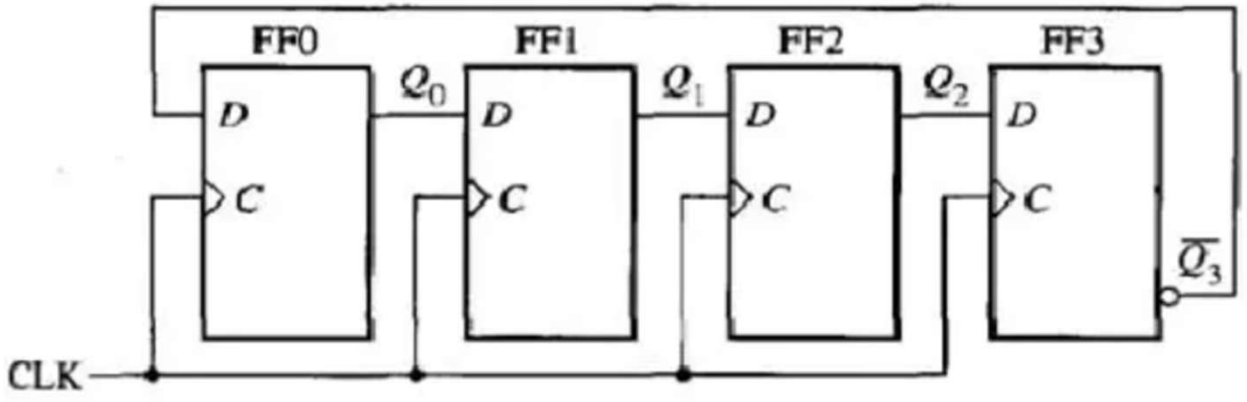



```
// a ring counter with initial value
module ring_counter(clk, start, qout);
parameter N = 4;
input  clk, start;
output reg [0:N-1] qout;
// the body of ring counter
always @(posedge clk or posedge start)
  if (start)  qout <= {1'b1,{N-1{1'b0}}};
  else        qout <= {qout[N-1], qout[0:N-2]};
endmodule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

Binary Counter

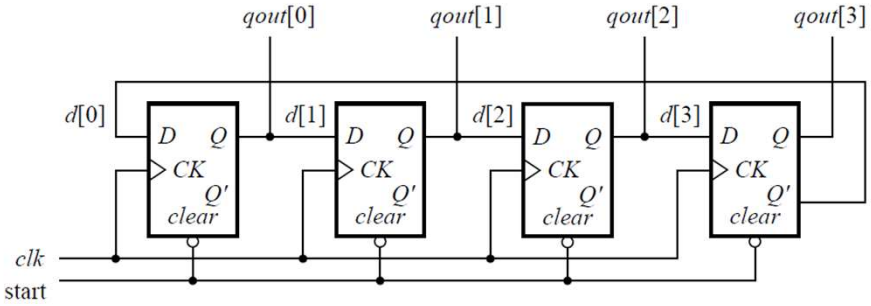
Johnson's Counters

Clock	Q_0	Q_1	Q_2	Q_3
0	0	0	0	0
1	1	0	0	0
2	1	1	0	0
3	1	1	1	0
4	1	1	1	1
5	0	1	1	1
6	0	0	1	1
7	0	0	0	1



Binary Counter

Johnson's Counters




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