Verilog Codes for Basic Digital Components

W. J. Dally and E. C. Harting, “Digital Design, a Systems Approach,” 2012
Chap. 8: Combinational Building Block

Outlines

- Combinational Logic
  - multiplexers, de-multiplexers
  - encoder, decoder, priority encoder
  - ripple carry adder
  - saturation adder
  - logical and arithmetic shifter
  - ALU
  - Dot Product
- Sequential Logic
  - latch, asynchronous reset latch
  - flip-flop, asynchronosu/synchronous reset flip-flop
  - register file
  - clock gating
  - finite state machine (FSM)
  - shift registers
  - FIR Filter
- Memory
  - ROM, RAM

Combinational Logic

multiplexer (1/2)

```
module mux4_1 (
output reg out,
input i0, i1, i2, i3,
s1, s0);

always @(*)
begin
   case ({s1, s0})
   2'd0 : out = i0;
   2'd1 : out = i1;
   2'd2 : out = i2;
   2'd3 : out = i3;
end

endmodule
```

```
module mux4_1 (
output out,
input i0, i1, i2, i3, s1, s0);


assign out = s1 ? (s0? i3 : i2) : (s0 ?
i1 : i0);


endmodule
```

multiplexer (2/2)

```
// MUX3 with one-hot select
module MUX3_onehot (a2, a1, a0, s, b);
parameter k = 32;
input [k-1:0] a0, a1, a2;
input [2:0] s; // one-hot select
output [k-1:0] b;
reg [k-1:0] b;

always @ (*) begin
   case (s)
      3'b001: b = a0;
      3'b010: b = a1;
      3'b100: b = a2;
      default: b = {k{1'bx}};
   endcase
end
endmodule
```

```
// MUX3 with binary  select
module MUX3_onehot (a2, a1, a0,
s, b);
parameter k = 32;
input [k-1:0] a0, a1, a2;
input [1:0] s; // binary select
output [k-1:0] b;
reg [k-1:0] b;

always @ (*) begin
   case (s)
      0: b = a0;
      1: b = a1;
      2: b = a2;
      default: b = {k{1'bx}};
   endcase
end
endmodule
```

demultiplexer

```
module demux1_4 (
output reg  out0, out1, out2, out3;
input in,
input [1:0] s );

always @(s, in)
case (s)
    2'b00: begin out0 = in;     out1=1'bz;  out2=1'bz;  out3=1'bz;   end
    2'b01: begin out0 = 1'bz;  out1=in;     out2=1'bz;  out3=1'bz;   end
    2'b10: begin out0 = 1'bz;  out1=1'bz;   out2=in;    out3=1'bz;   end
    2'd11: begin out0 = 1'bz;  out1=1'bz;   out2=1'bz;  out3=in;     end
    default: $display ("Invalid control signals");
endcase
```

encoder

```
module encoder (
input i3, i2, i1, i0,
output reg [1:0] out);

always @(i3, i2, i1, i0)
valid = 1;
case ({i3, i2, i1, i0})
    4'b1000: out = 2'b11;
    4'b0100: out = 2'b10;
    4'b0010: out = 2'b01;
    4'b0001: out = 2'b00;
endcase

endmodule
```

priority encoder

```
module priority_enc42
(input i3, i2, i1, i0,
output reg [1:0] out;)

always @(i3, i2, i1, i0)
casex ({i3, i2, i1, i0})
    4'b1xxx: out = 2'b11;
    4'b01xx: out = 2'b10;
    4'b001x: out = 2'b01;
    default  : out = 2'b00;
endcase
```

```
// 8:3 priority encoder
module priority_enc83 (r, b);
input [7:0] r;
output [2:0] b;
reg [2:0] g;

assign b = g;
always @ (*) begin
  casex®
    8'bxxxxxxx1: g = 0;
    8'bxxxxxx10: g = 1;
    8'bxxxxx100: g = 2;
    8'bxxxx1000: g = 3;
    8'bxxx10000: g = 4;
    8'bxx100000: g = 5;
    8'bx1000000: g = 6;
    8'b10000000: g = 7;
    default: g = x;
  endcase
end
endmodule
```

decoder (1/3)

```
module decoder (
input  [1:0] in;
output reg  out0, out1, out2, out3) ;

always @(in)
case (in)
    2'b00: begin out0=1;   out1=0;    out2=0;   out3=0;   end
    2'b01: begin out0=0;   out1=1;    out2=0;   out3=0;   end
    2'b10: begin out0=0;   out1=0;    out2=1;   out3=0;   end
    2'd11: begin out0=0;   out1=0;    out2=0;   out3=1;   end
endcase

endmodule
```

decoder (2/3)

```
module decoder_index
(in1, out1);
parameter N=8;
parameter log2N = 3;
input [log2N-1: 0] in1;
output reg [N-1:0] out1;


always @ (in1) begin
   out1 = 0;
   out1[in1] = 1'b1;
end

endmodule
```

```
module decoder_loop (in1, out1);
parameter N=8;
parameter log2N = 3;
input [log2N-1: 0] in1;
output reg [N-1:0] out1;

integer i;

always @ (in1)
   for (i=0; i<N; i=i+1)
      out1[i] = (in1 == i);

endmodule
```

decoder (3/3)

```
// n-to-m decoder
module Dec (a, b);
parameter n=3;
parameter m=8;
input [n-1: 0] a;
output [m-1:0] b;

assign b = 1 << a;

endmodule
```

Ripple Carry Adder (RCA) using always

```
always @ (*) begin
   c[0] = cin;
   for (i=0; i<=31; i++)
      {c[i+1], s[i]} = a[i] + b[i] + c[i];
   cout = c[32];
end
// the for loop will be unrolled as follows
// {c[1],s[0]}=a[0]+b[0]+c[0];
// {c[2],s[1]}=a[1]+b[1]+c[1];
…
// {c[32],s[31]}=a[31]+b[31]+c[31];
```

RCA (gate primitive vs. logical operator)

```
module ripple_adder(co, sum, a0, a1, ci);
parameter N=4;
output [N-1:0] sum;
output co;
input [N-1:0] a0, a1;
input ci;
wire [N-1:0] carry;

assign carry[0]=ci;

genvar i;
generate
      for (i=0; i<N; i=i+1)
      begin: r_loop
            wire t1, t2, t3;
            xor g1(t1, a0[i], a1[i]);
            xor g2(sum[i], t1, carry[i]);
            and g3(t2, a0[i], a1[i]);
            and g4(t3, t1, carry[i]);
            or g5(carry[i+1], t2, t3);
      end
endgenerate

assign co = carry[N]

endmodule
```

```
module ripple_adder(co, sum, a0, a1, ci);
parameter N=4;
output [N-1:0] sum;
output co;
input [N-1:0] a0, a1;
input ci;

wire [N-1:0] p = a0 ^ a1;
wire [N-1:0] g = a0 & a1;
wire [N:0] carry = {g | ( p & carry[N-1:0] ), ci};
// carry[0] = ci;
// carry[1] = g[0] | ( p[0] & carry[0] );
// carry[2] = g[1] | ( p[1] & carry[1] );
// carry[3] = g[2] | ( p[2] & carry[2] );
// carry[4] = g[3] | ( p[3] & carry[3] );

assign sum = p ^ carry[N-1:0];
assign co = carry [N];

endmodule+
```

RCA (always vs. logic perators)

```
module ripple_adder(co, sum, a0, a1, ci);
parameter N=4;
output [N-1:0] sum;
output co;
input [N-1:0] a0, a1;
input ci;
wire [N:0] carry;

assign carry[0]=ci;

always @ (a0 or a1)
begin
for (i=0; i<N; i=i+1)
      {carry[i+1], sum[i]} = a0[i] + a1[i] +
      carry[i] ;
end

assign co = carry[N]

endmodule
```

```
module ripple_adder(co, sum, a0, a1, ci);
parameter N=4;
output [N-1:0] sum;
output co;
input [N-1:0] a0, a1;
input ci;
wire [N:0] carry;

assign carry[0]=ci;

always @ (a0 or a1)
begin
for (i=0; i<N; i=i+1)
   begin
      carry[i+1] = a0[i]&a1[i] | a0[i]&carry[i] |
      a1[i]&carry[i];
      sum[i] = a0[i]^a1[i]^carry[i];
   end
end

assign co = carry[N]

endmodule
```

RCA with generate loop

- use **generate** loop to duplicate module instances
  - e.g. in FA-cascaded RCA
- or simple
  - **array of instances**

```
// structural level modeling
module RCA (sum, c_out, a, b, c_in);

output [3:0] sum;
output c_out;
input [3:0] a, b;
input c_in

wire c1, c2,c3;
// construct 4-bit adder fullladd4 from
// previously defined module fulladd

fulladd   fa0 (sum[0], c1, a[0], b[0], c_in);
fulladd   fa1 (sum[1], c2, a[1], b[1], c1);
fulladd   fa2 (sum[2], c3, a[2], b[2], c2);
fulladd   fa3 (sum[3], c_out, a[3], b[3], c3);

endmoudle
```

```
module RCA_generate (sum, c_out, a, b, c_in);
parameter N=4;
output [N-1:0] sum;
output c_out
input [N-1:0] a, b;
input c_in;
wire [N:0] c;

assign c[0]=c_in;

genvar i;
generate
  for (i=0; i<=N-1; i=i+1)
    begin: FA_loop  // block named "FA_loop"
      fulladd  fa (sum[i], c[i+1], a[i], b[i], c[i]);
      // fulladd FA_loop[0].fa ( sum[0], c[1], a[0], b[0], c[0] );
      // fulladd FA_loop[1].fa ( sum[1], c[2], a[1], b[1], c[1] );
      // fulladd FA_loop[2].fa ( sum[2], c[3], a[2], b[2], c[2] );
      // fulladd FA_loop[3].fa ( sum[3], c[4], a[3], b[3], c[3] );
    end
endgenerate

// fulladd FA_loop[0:3] (sum[0:3], c[1:4], a[0:3], b[0:3], c[0:3]);

assign  c_out = c[N];

endmodule
```

Verilog code for Saturation Adder

- clip the computation results to a fixed range

```
wire signed  [7:0] a, b, tmp, maxk min;
reg signed [7:0] sum;

assign max = 8'd127;   // max = 0111_1111
assign min = 8'd-128;   // min = 1000_0000
assign tmp = a + b;
always @ (*) begin
   if ( a[7] == b[7] && tmp[7] != a[7] )     // carry[ [8] != carry]7]
      sum = a[7] ? min : max;   // sum = { a[7], 7{~a[7]} };
   else
      sum = tmp;
end
```

left/right shifter

```
module shifter (
input left,
input [2:0] shift_amt,
input [7:0] data_in,
output [7:0] shifted_data);

assign shifted_data = (left) ? data_in << shift_amt : data_in >>
shift_amt;

endmodule
```

wrap-around barrel shifter

```
module barrel_shift (n, a, b);

parameter k=8;
parameter lk=3;
input [lk-1:0] n;   // how much to shift
Input [k-1:0] a;    // number to shift
output [k-1:0] b; // the output

wire [2*k-2:0] x = a << n;  // output before wrapping

assign b = x[k-1 : 0] | {1'b0, x[2k-2 : k] };

endmodule
```

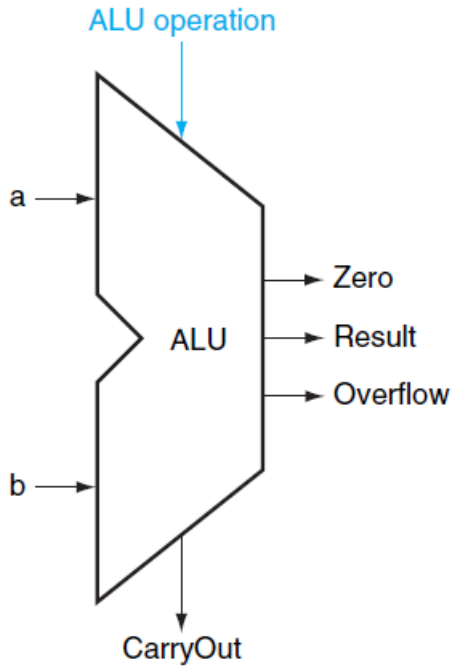
arithmetic shift (<<<, >>>) in Verilog-2001

```
// Verilog-2001

integer  data_value,
         data_value_1995,
         data_value_2001;
…
data_value = -9;                              // stored as 1111_…_1111_0111
…
data_value_1995 = data_value  >> 3;   // stored as  0001_..._1111_1110
…
data_value_2001 = data_value  >>> 3; // stored as  1111_...._1111_1110
                                              // repeat sign bit during arithmetic right shift
…
data_value_1995 = data_value << 3;    // stored as 1111_..._1011_1000
…
data_value_2001 = data_value <<< 3;  // stored as 1111_..._1011_1000
                                               // repeat sign bit during arithmetic left shift
// give an example where the logical left shift << 3 is different from arithmetic shift
// e.g., data_value2 = 32'b 1110_xxx ….
```

ALU

```
module MIPSALU (
input [31:0] a, b;
input [3:0] ALU_ctl;
output reg [31:0] ALU_out;
output Zero );

assign Zero = (ALU_out == 0);

always @ (ALU_ctl, a, b)
   case (ALU_ctl)
      4'd0 : ALU_out = a & b;
      4'd1 : ALU_out = a | b;
      4'd 2 : ALU_out = a + b;
      4'd6 : ALU_out = a – b;
      4'd 7: ALU_out = (a < b) ? 1 : 0;
      4'd12: ALU_out = ~(a | b);
      default: ALU_out = 32'b0;
   endcase

endmodule
```

ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set on less than
1100	NOR



Dot Product 4 (DP4)

- 4-D dot-product: a[0]*b[0]+ a[1]*b[1]+ a[2]*b[2]+ a[3]*b[3]

```
module DP4 (input[16*4-1:0]in_a, in_b, output [31:0]out);
wire [15:0] a[3:0],  b[3:0];

// convert two long bit-vectors in_a, in_b into two arrays a, b
assign {a[3],a[2],a[1],a[0]}=in_a;
assign {b[3],b[2],b[1],b[0]}=in_b;

integer i;
reg [31:0] tmp[0:3];
always @ (*) begin
   tmp[0] = a[0]*b[0];
   for (i=1; i<=3; i=i+1) tmp[i] = a[i]*b[i] + tmp[i-1];
end

assign out = tmp[3];

endmodule
```

DPn

- n-D dot-product

```
module #parameter n=4 DPn (input[16*n-1:0]in_a, in_b, output [31:0]out);
reg [15:0] a[n-1:0];
reg [15:0] b[n-1:0];
integer i;
always@ (*) begin   // convert two long bit-vectors into two arrays
            for (i=0; i<n; i=i+1) begin
                        a[i]= in_a[16*i +:16];
                        b[i]= in_b[16*i +:16];
            end
end

reg [31:0] tmp[0:n];
always @ (*) begin
   tmp[0] = 0;
   for (i=0; i<=n-1; i=i+1) tmp[i+1] = a[i]*b[i] + tmp[i];
end
assign out = tmp[n];
endmodule
```

Pipelined DP4

- see previous lectures for pure CL DP4
  - first convert I/O ports into arrays; then pipeline it

```
module DP4 (input clk, input[16*4-1:0]in_a, in_b, output reg [31:0]out);
wire [15:0] a[3:0];
wire [15:0] b[3:0];
assign {a[3],a[2],a[1],a[0]}=in_a;
assign {b[3],b[2],b[1],b[0]}=in_b;
integer i;
reg [31:0] pipe1_tmp[0:3], pipe2_tmp[1:4];
always @(posedge clk) begin
        for (i=0; i<=3; i=i+1)  pipe1_tmp[i] <= a[i]*b[i];
end
always @ (posedge clk) begin
        pipe2_tmp[1] = pipe1_tmp[0];
        for (i=1; i<=3; i=i+1) pipe2_tmp[i+1] <= pipe2_tmp[i] + pipe1_tmp[i];
        out<= pipe2_tmp[4];
end
endmodule
```

Pipelined DPn

- parametrized long bit-vectors as I/O ports

```
module  #parameter n=4 DPn (input clk, input[16*n-1:0]in_a, in_b, output reg [31:0]out);
reg [15:0] a[n-1:0];
reg [15:0] b[n-1:0];
integer i;
always@ (*) begin
        for (i=0; i<n; i=i+1) begin
                a[i]= in_a[16*i +:16];
                b[i]= in_b[16*i +:16];
        end
end
reg [31:0] pipe1_tmp[0:n-1], pipe2_tmp[1:n];
always @(posedge clk) begin
        for (i=0; i<=n-1; i=i+1) pipe1_tmp[i] <= a[i]*b[i];
end
always @ (*) begin
        pipe2_tmp[1] = pipe1_tmp[0];
        for (i=1; i<=n-1; i=i+1) pipe2_tmp[i+1] = pipe2_tmp[i] + pipe1_tmp[i];
end
always @ (posedge clk) begin
        out <= pipe2_tmp[n];
end
endmodule
```

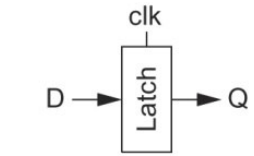
Sequential Logic

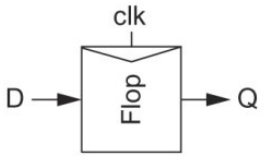
Latches and Flip-Flops

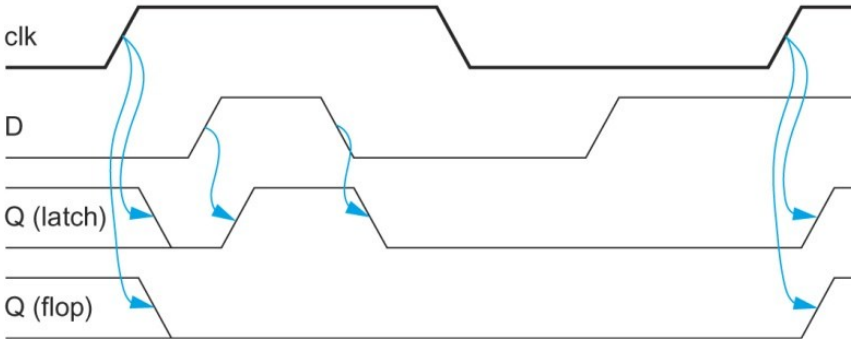
Sequencing Elements (Latch and Flip-Flop)

- **Latch**: *Level sensitive*
  - pass input to output at **time interval** when level of control is logic 1
  - a.k.a. transparent latch

```
// Verilog code for latch
always @ (CLK or D)
   if (CLK)  Q <= D;
```



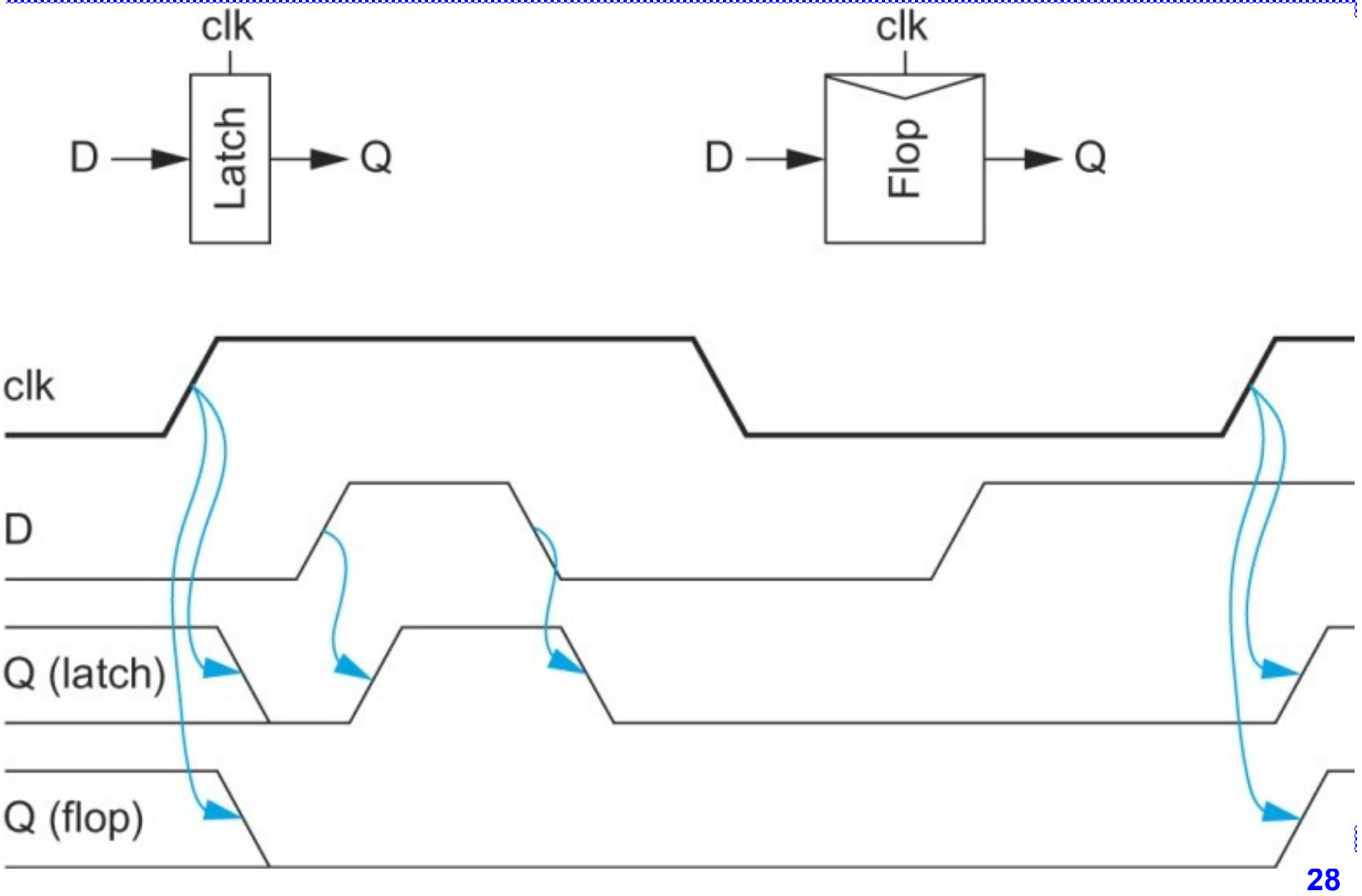




- **Flip-Flop**: *edge triggered*
  - pass input to output at the **moment** of clock rising or falling edge
  - a.k.a. opaque edge-triggered flip-flop

```
// Verilog code for DFF
always @ (posedge CLK)
   Q <= D;
```

latch vs. FF

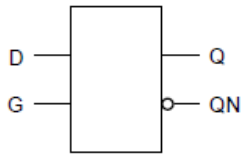


register inference

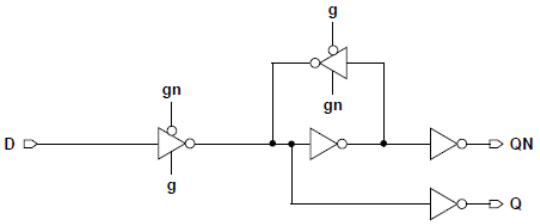
- D latch
  - with asynchronous set
  - with asynchronous reset
  - with asynchronous set/reset
- D flip flop
  - with asynchronous set
  - with asynchronous reset
  - with asynchronous set/reset
  - with synchronous set
  - with synchronous reset
  - with synchronous and asynchronous load

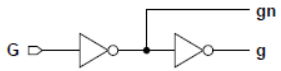
D Latches in TSMC 0.18um Cell Library

TLAT
active-high

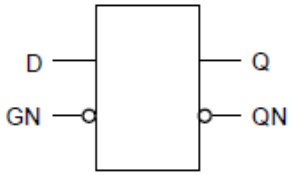


G	D	Q[n+1]	QN[n+1]
1	0	0	1
1	1	1	0
0	x	Q[n]	QN[n]

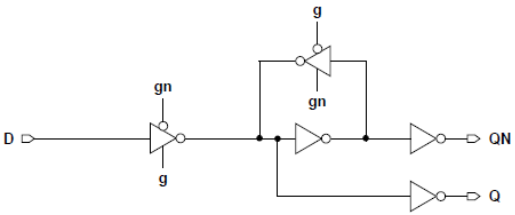


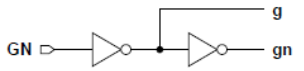


TLATN
active-low

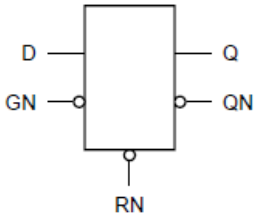


GN	D	Q[n+1]	QN[n+1]
0	0	0	1
0	1	1	0
1	x	Q[n]	QN[n]

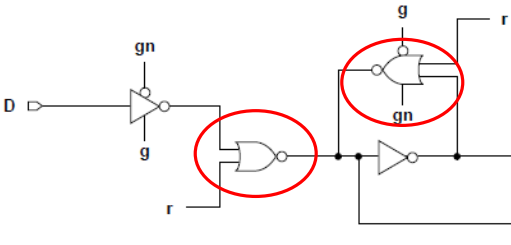


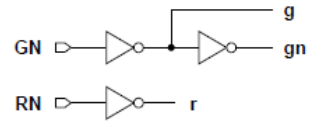


TLATNR
asynchronous
active-low
reset

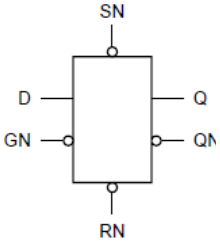


RN	GN	D	Q[n+1]	QN[n+1]
1	0	0	0	1
1	0	1	1	0
1	1	x	Q[n]	QN[n]
0	x	x	0	1

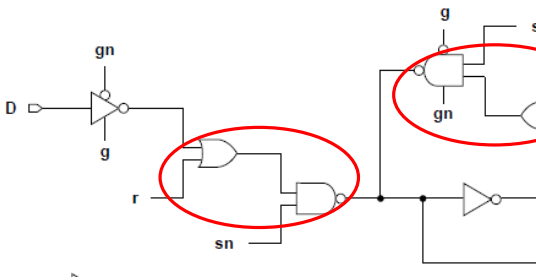


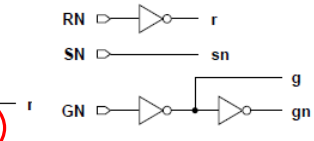


TLATNSR
asynchronous
active-low
set/reset



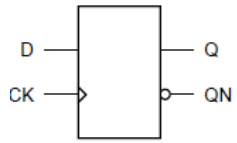
RN	SN	GN	D	Q[n+1]	QN[n+1]
1	1	0	0	0	1
1	1	0	1	1	0
1	1	1	x	Q[n]	QN[n]
0	1	x	x	0	1
1	0	x	x	1	0
0	0	x	x	1	0



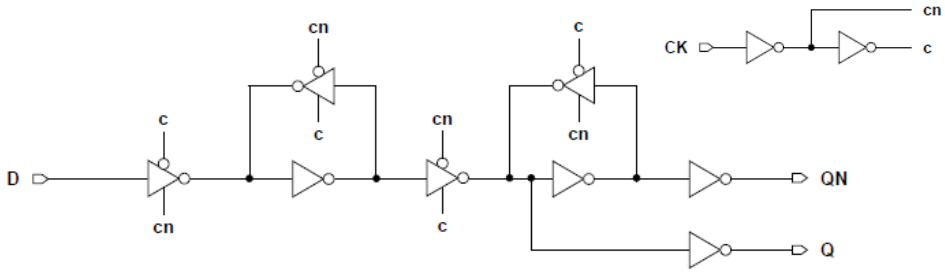


DFFs in TSMC 0.18um Cell Library

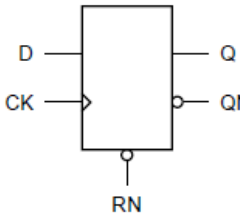
DFF



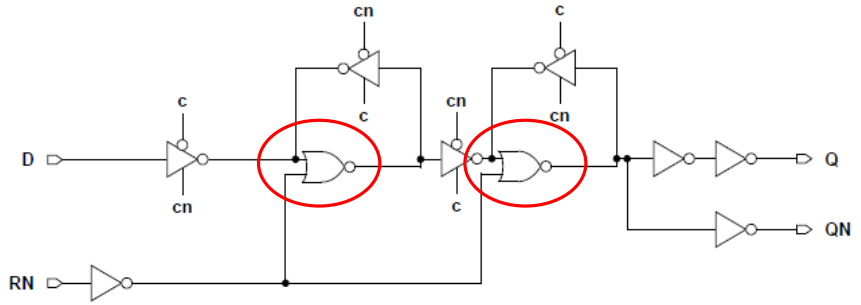
D	CK	Q[n+1]	QN[n+1]
0	↑	0	1
1	↑	1	0
x	↓	Q[n]	QN[n]



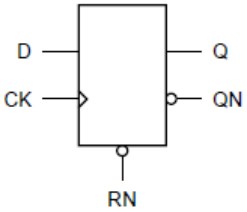
DFFR
asynchronous
active-low



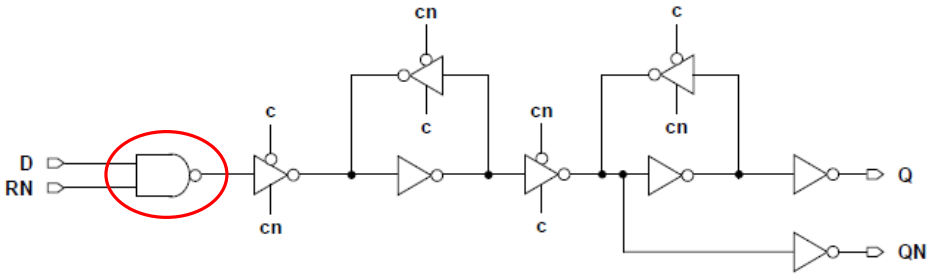
RN	D	CK	Q[n+1]	QN[n+1]
0	x	x	0	1
1	0	↑	0	1
1	1	↑	1	0
1	x	↓	Q[n]	QN[n]



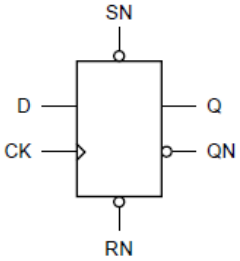
DFFTR
synchronous
active-low



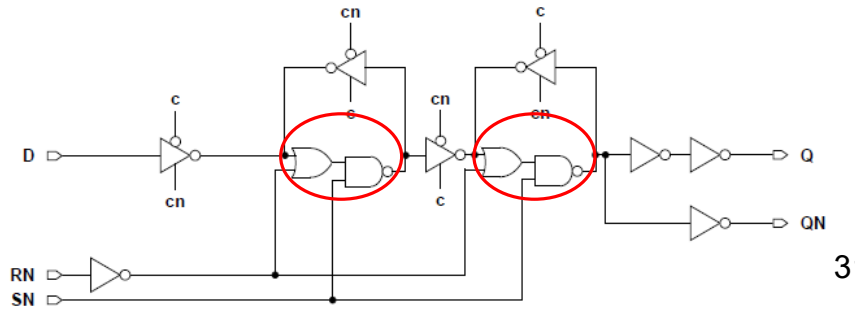
RN	D	CK	Q[n+1]	QN[n+1]
0	x	↑	0	1
x	x	↓	Q[n]	QN[n]
1	0	↑	0	1
1	1	↑	1	0



DFFSR
asynchronous
active-low
set/reset



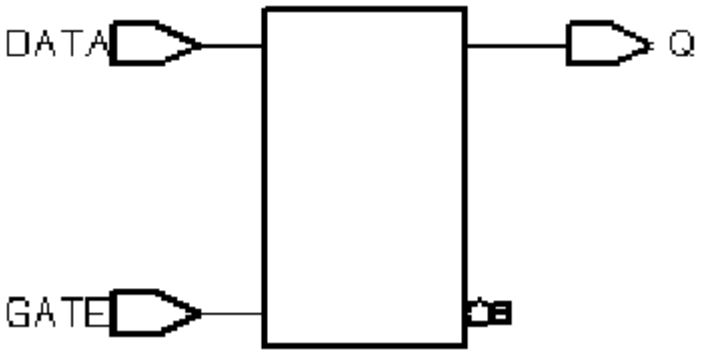
RN	SN	D	CK	Q[n+1]	QN[n+1]
0	1	x	x	0	1
1	0	x	x	1	0
0	0	x	x	1	0
1	1	0	↑	0	1
1	1	1	↑	1	0
1	1	x	↓	Q[n]	QN[n]



inferring simple D-latch

- always @ (clk or d) …
  - lclk evel-sensitive

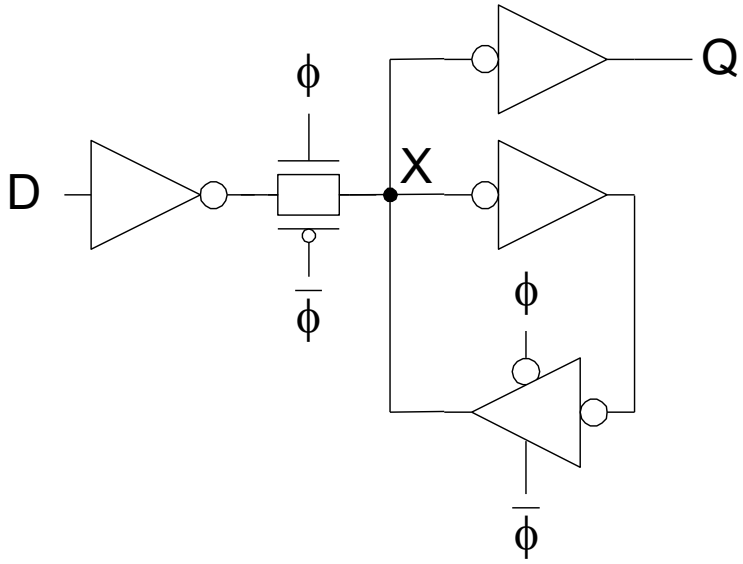
```
module d_latch (GATE, DATA, Q);
  input GATE, DATA;
  output Q;
  reg Q;

always @(GATE or DATA)
  if (GATE)
    Q = DATA;

endmodule
```

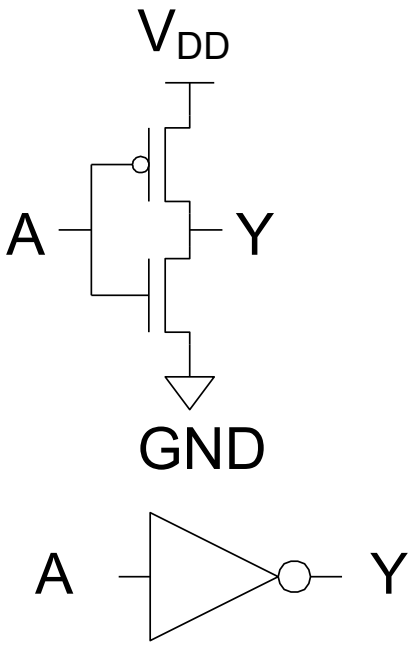



```
module d_CL (
input gate, data,
output reg q );

// infer a combinational logic,
// not a latch
always @ (gate or data)
  if (gate)
     q <= data;
  else
    q <= 'b0;

endmodule
```

D Latch Circuit using CMOS

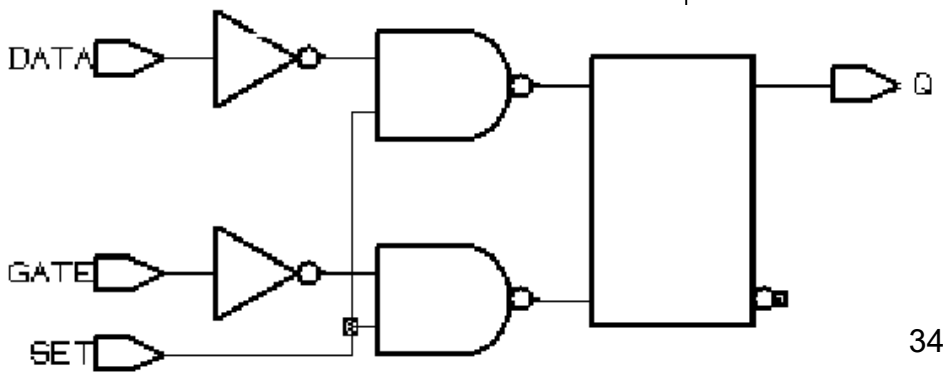
- constructed from back-to-back inverters





D-latch with asynchronous set (active-low)

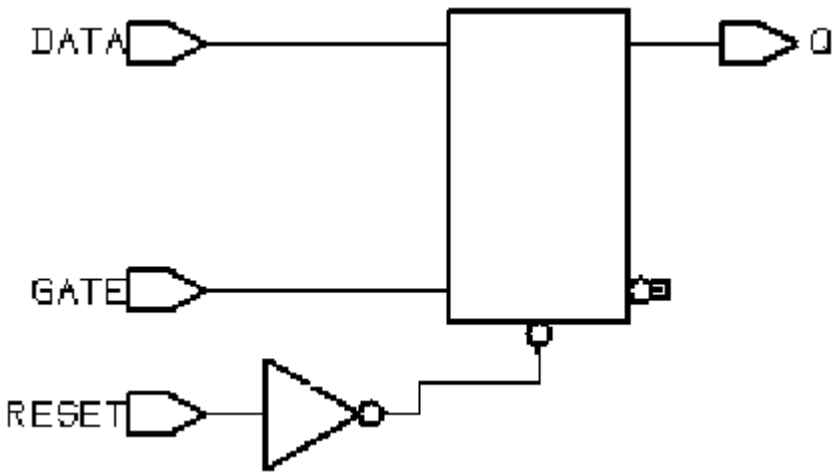
```
module d_latch_async_set (GATE, DATA, SET, Q);
  input GATE, DATA, SET;
  output Q;
  reg Q;

//synopsys async_set_reset "SET"
always @(GATE or DATA or SET)
  if (~SET)
    Q = 1'b1;
  else if (GATE)
    Q = DATA;
endmodule
```



D-latch with asynchronous reset (active-low)

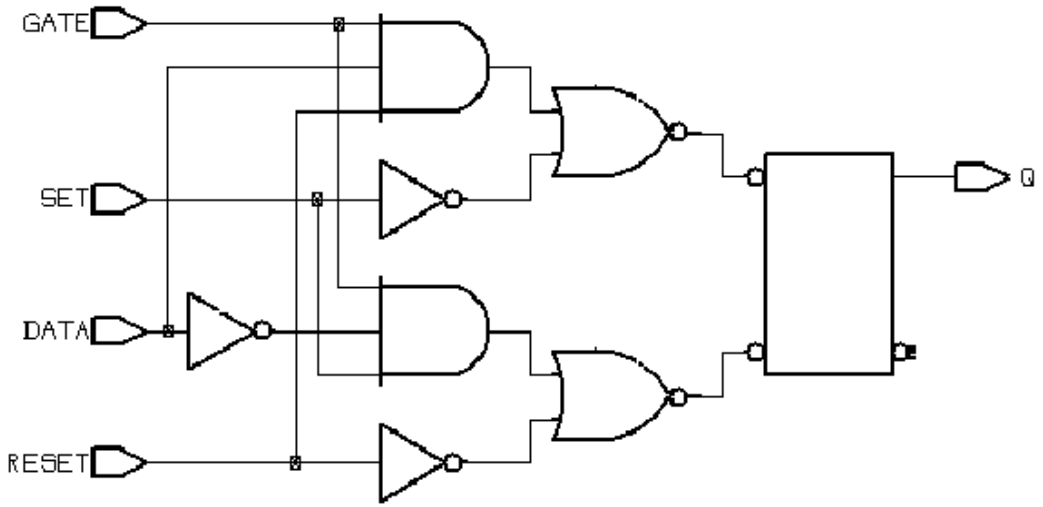
- sensitive list includes control signals and input data signals

```
module d_latch_async_reset (RESET, GATE, DATA, Q);
  input RESET, GATE, DATA;
  output Q;
  reg Q;

//synopsys async_set_reset "RESET"
always @ (RESET or GATE or DATA)
  if (~RESET)
    Q = 1'b0;
  else if (GATE)
    Q = DATA;
endmodule
```



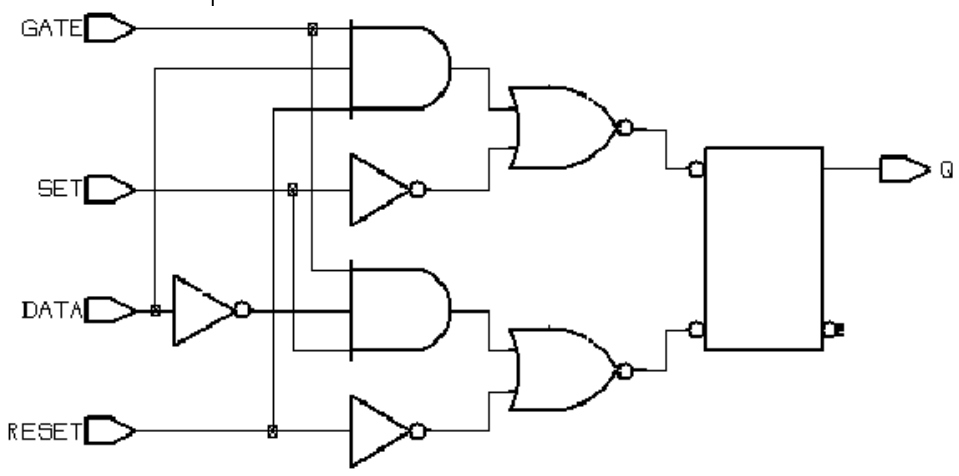
D-latch with asynchronous set and reset (active-low)

```
module d_latch_async (GATE, DATA, RESET, SET, Q);
  input GATE, DATA, RESET, SET;
  output Q;
  reg Q;

// synopsys async_set_reset_local infer "RESET, SET"
// synopsys one_cold "RESET, SET"
always @ (GATE or DATA or RESET or SET)
begin : infer
  if (!SET)
    Q = 1'b1;
  else if (!RESET)
   Q = 1'b0;
   else if (GATE)
    Q = DATA;
end

// synopsys translate_off
always @ (RESET or SET)
  if (RESET == 1'b0 & SET == 1'b0)
  $write ("ONE-COLD violation for RESET and SET.");
// synopsys translate_on
endmodule
```



latch with asynchronous set and reset

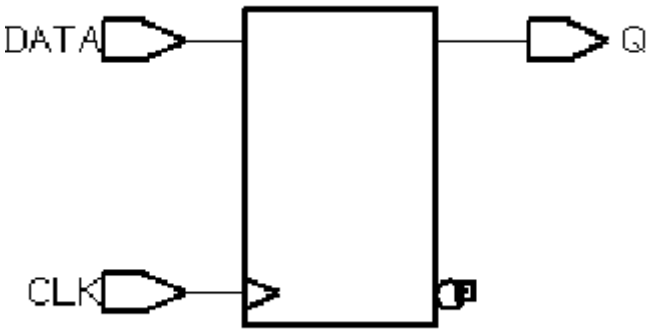
```
module d_latch_async (
input gate, data, reset_n, set_n;
output reg q;

always @ (gate, data, reset_n, set_n)
   if (!set_n) q <= 1'b1;
   else if (!reset_n) q <= 1'b0;
   else if (gate) q <= data;

endmodule
```



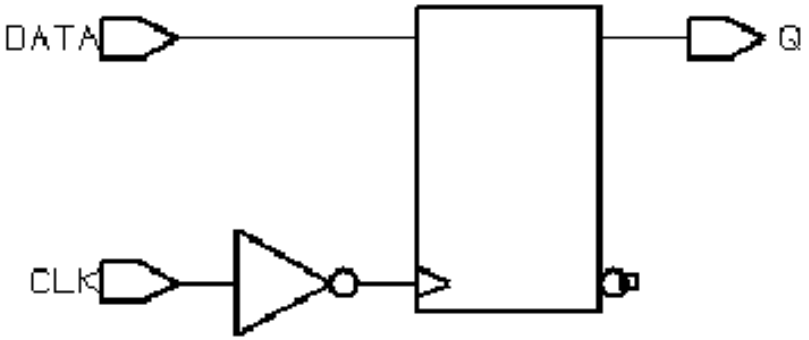
inferring D-flipflop

- always @ (posedge clk) …
  - clk edge-triggered

```
module dff_pos (DATA, CLK, Q);
  input DATA, CLK;
  output Q;
  reg Q;

always @(posedge CLK)
  Q <= DATA;
endmodule
```

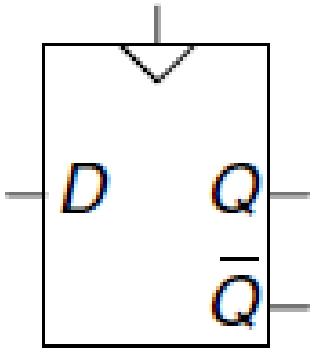


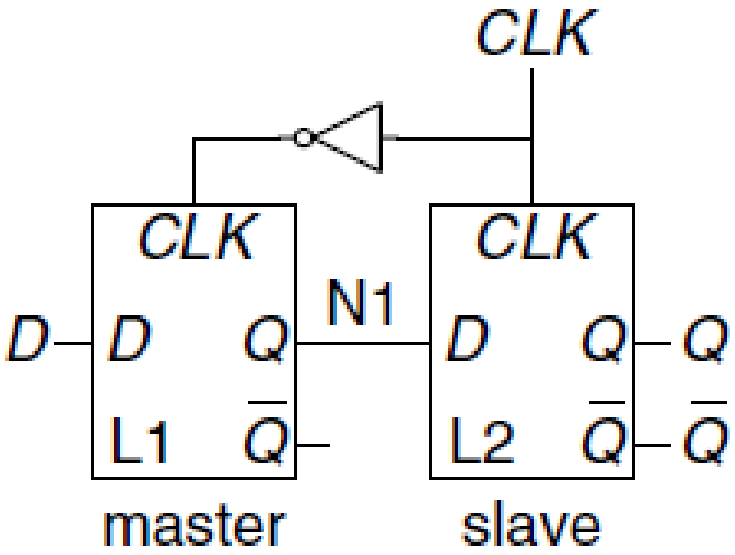

```
module dff_neg (DATA, CLK, Q);
  input DATA, CLK;
  output Q;
  reg Q;

always @(negedge CLK)
  Q <= DATA;
endmodule
```



Logic of D Flip-Flop

- composed of master and slave D-latches
- triggered at the time instance of clock edge
  - unlike latch which is active (transparent) when clock is at logic level of high

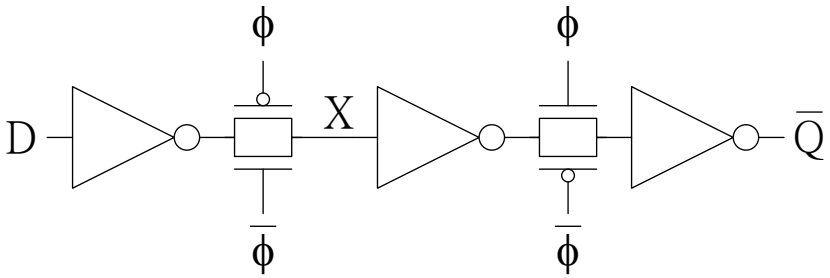




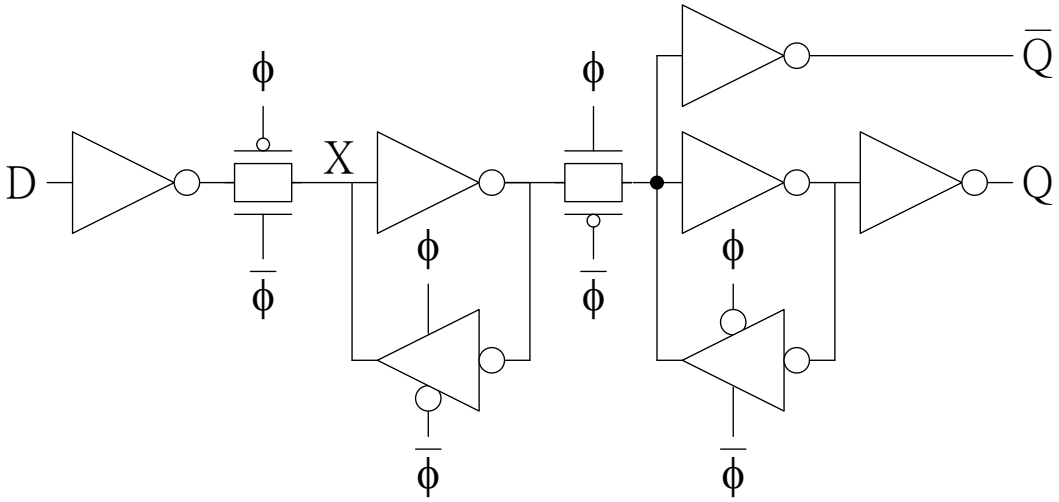
D Flip-Flop using CMOS

- constructed from two D latches (either dynamic or static)
  - cell library usually adopts static design

Dynamic DFF



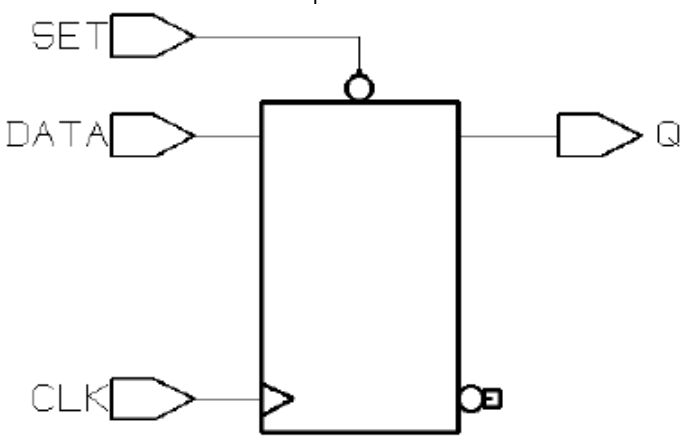
Static DFF



DFF with asynchronous set (active-low)

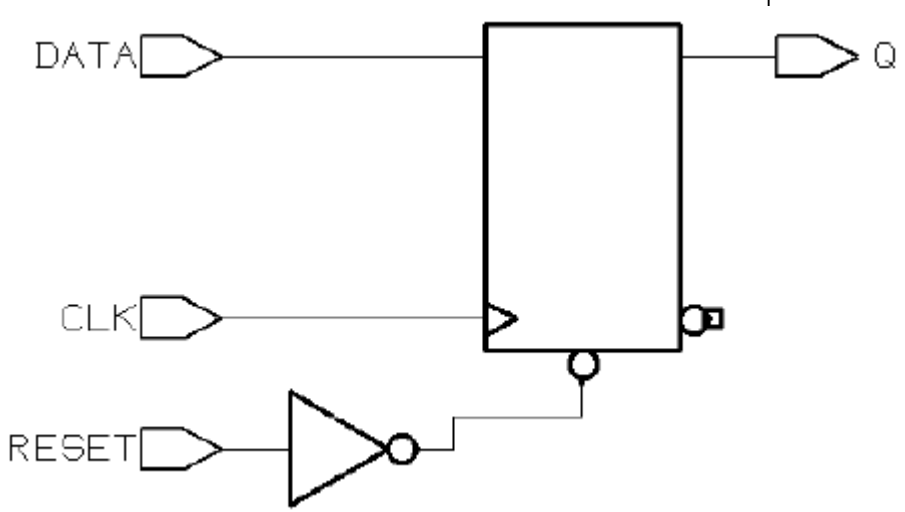
- sensitive list includes edges of both clock and control signals
  - synthesized into hardware different from normal DFF

```
module dff_async_set (DATA, CLK, SET, Q);
  input DATA, CLK, SET;
  output Q;
  reg Q;

always @(posedge CLK or negedge SET)
  if (~SET)
    Q <= 1'b1;
  else
    Q <= DATA;
endmodule
```



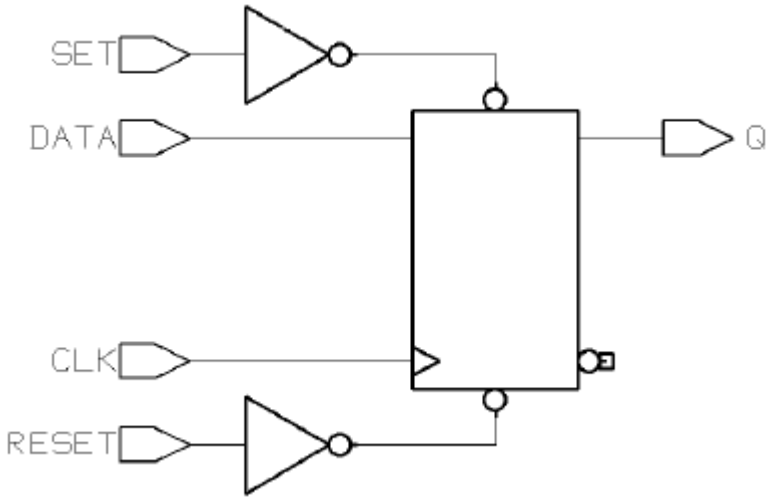
DFF with asynchronous reset (active-high)

```
module dff_async_reset (DATA, CLK, RESET, Q);
   input DATA, CLK, RESET;
   output Q;
   reg Q;

always @(posedge CLK or posedge RESET)
   if (RESET)
     Q <= 1'b0;
   else
     Q <= DATA;
endmodule
```



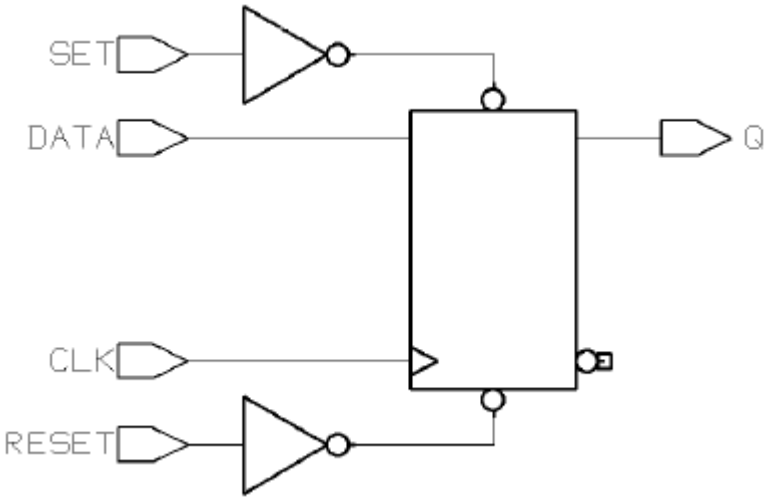
DFF with asynchronous set and reset (active-high)

```
module dff_async (RESET, SET, DATA, Q, CLK);
   input CLK;
   input RESET, SET, DATA;
   output Q;
   reg Q;

// synopsys one_hot "RESET, SET"
always @(posedge CLK or posedge RESET or
           posedge SET)
   if (RESET)
     Q <= 1'b0;
   else if (SET)
     Q <= 1'b1;
   else Q <= DATA;

// synopsys translate_off
always @ (RESET or SET)
   if (RESET + SET > 1)
   $write ("ONE-HOT violation for RESET and SET.");
// synopsys translate_on
endmodule
```



DFF with asynchronous set and reset

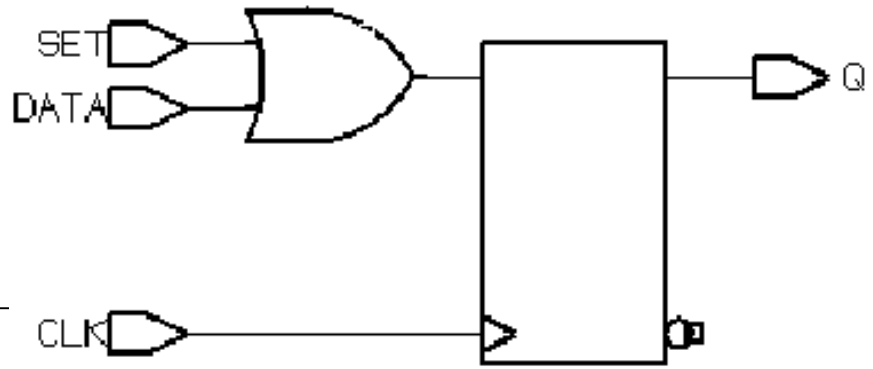
```
module dff_async (
input clk, reset, set, data,
output reg q);

always @ (posedge clk or posedge reset or posedge set)
   if (reset) q <= 1'b0;
   else if (set) q <= 1'b1;
   else q <= data;

endmodule
```



DFF with synchronous set (active-high)

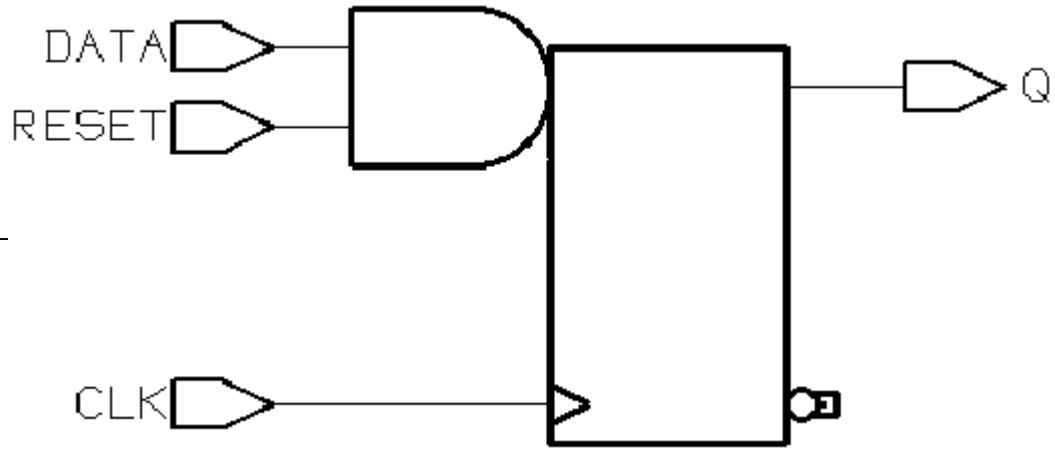
- sensitive list includes only clock signal
  - synthesized into normal DFF with extra combinational logic for the input data

```
module dff_sync_set (DATA, CLK, SET, Q);
  input DATA, CLK, SET;
  output Q;
  reg Q;

//synopsys sync_set_reset "SET"
always @(posedge CLK)
  if (SET)
    Q <= 1'b1;
  else
    Q <= DATA;
endmodule
```



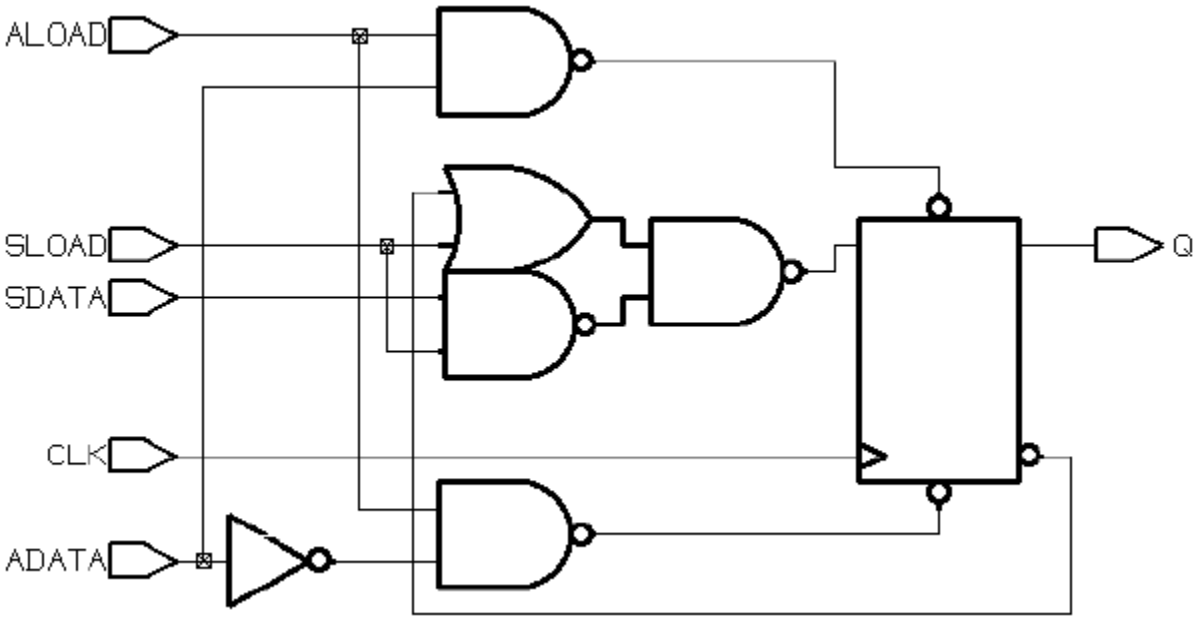
DFF with synchronous reset (active-low)

```
module dff_sync_reset (DATA, CLK, RESET, Q);
  input DATA, CLK, RESET;
  output Q;
  reg Q;

//synopsys sync_set_reset "RESET"
always @(posedge CLK)
  if (~RESET)
    Q <= 1'b0;
  else
    Q <= DATA;
endmodule
```



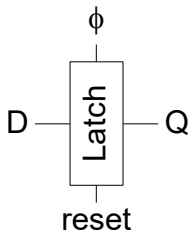
DFF with synchronous and asynchronous load

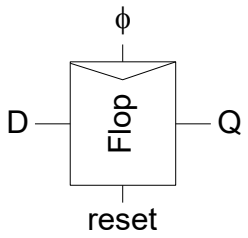
```
module dff_a_s_load (ALOAD, SLOAD, ADATA, SDATA, CLK, Q);
  input ALOAD, ADATA, SLOAD, SDATA, CLK;
  output Q;
  reg Q;

always @ (posedge CLK or posedge ALOAD)
  if (ALOAD)
    Q <= ADATA;
  else if (SLOAD)
    Q <= SDATA;
endmodule
```



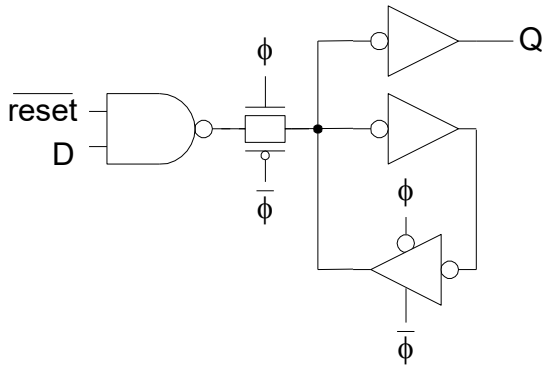
Resettable DFF Circuits using CMOS

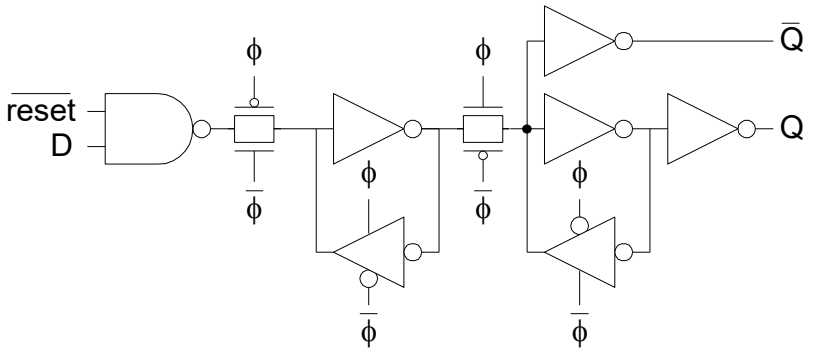
Symbol



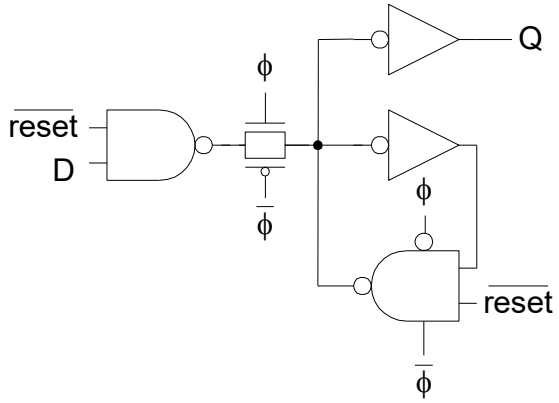


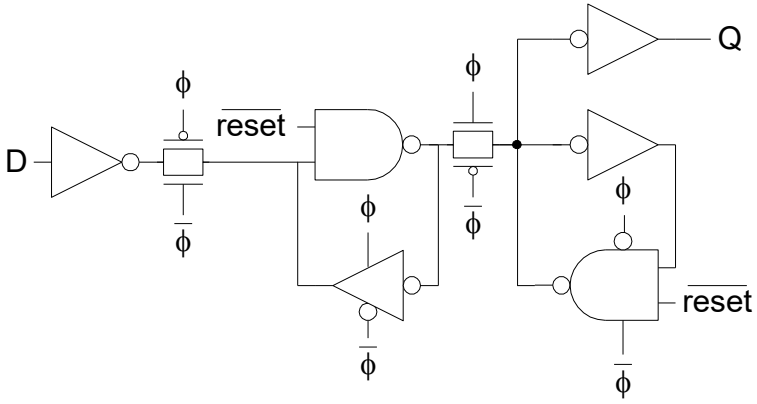
Synchronous Reset





Asynchronous Reset





register file

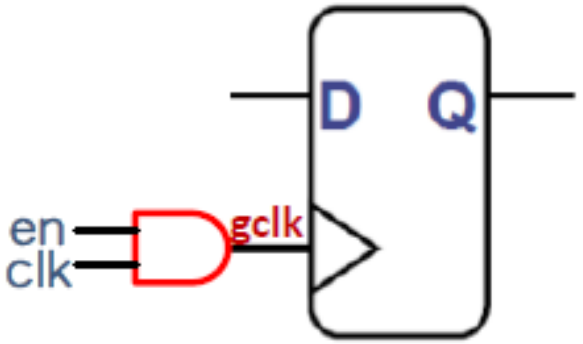
- register file is compose of registers
  - can access many registers at the same time
    - cp. single-port RAM which can access only one data

```
module RegisterFile (r_adr1, r_adr2, data1, data2, w_adr, w_data, w_cntl, clk)
input [5:0] r_adr1, r_adr2, w_adr;
input [31:0] w_data;
input w_cntl, clk;
output [31:0] data1, data2;
reg [31:0] RF [0:31];

// read from register f ile
assign data1 = RF[r_adr1];
assign data2 = RF[r_adr2];

// write to register file
always @ (posedge)
    if (w_cntl) RF[w_adr] = w_data;

endmodule
```

Clock gating methods

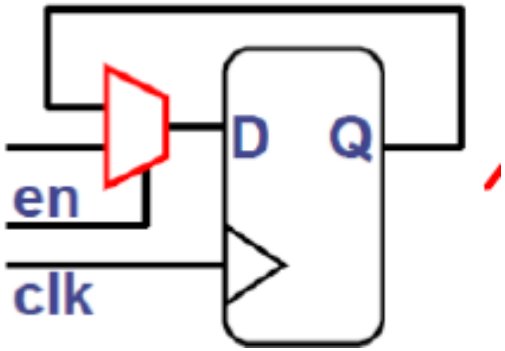
- Method 1 (unsafe gclk with possible glitches)

```
assign  gclk = clk && enable    ;

always@(posedge gclk or negedge nrst)
begin
    if(!nrst)
        Q   <=  8'd0;
    else
        Q   <=  D   ;
end
```



- Method2 (high switching activity of clk)

```
always@(posedge clk or negedge nrst)
begin
    if(!nrst)
        Q   <=  8'd0;
    else if(enable)
        Q   <=  D   ;
end
```



Finite State Machine (FSM)

Two types of Sequential Logic

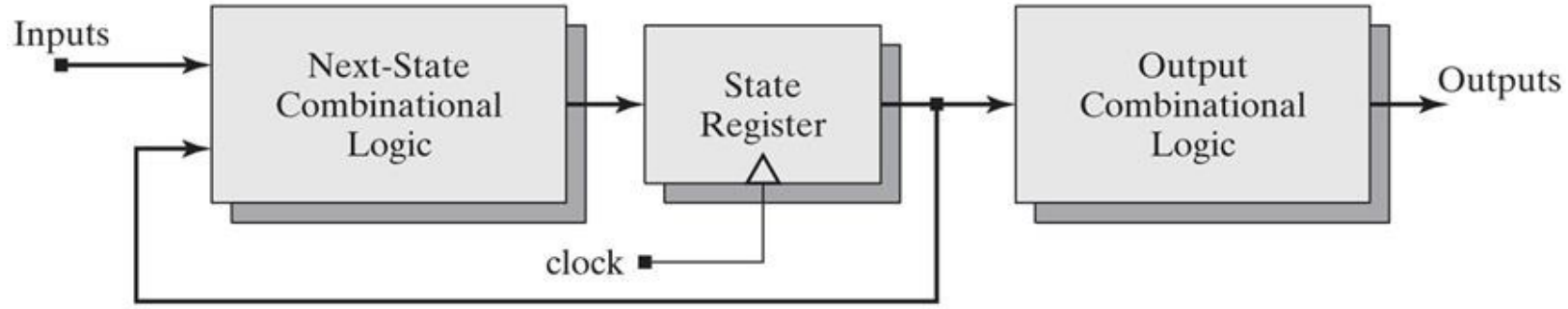
- Pipeline
  - break long datapath into several stages, each realized in one clock cycle
  - use pipelined registers to cut the combinational logic into several stages
  - better pipelining with balanced delay for each pipeline stage
- Finite State Machine (FSM)
  - generate control signals for datapath components
  - use registers to store the states
  - output depends on either the current states, and/or the current inputs

Pipeline

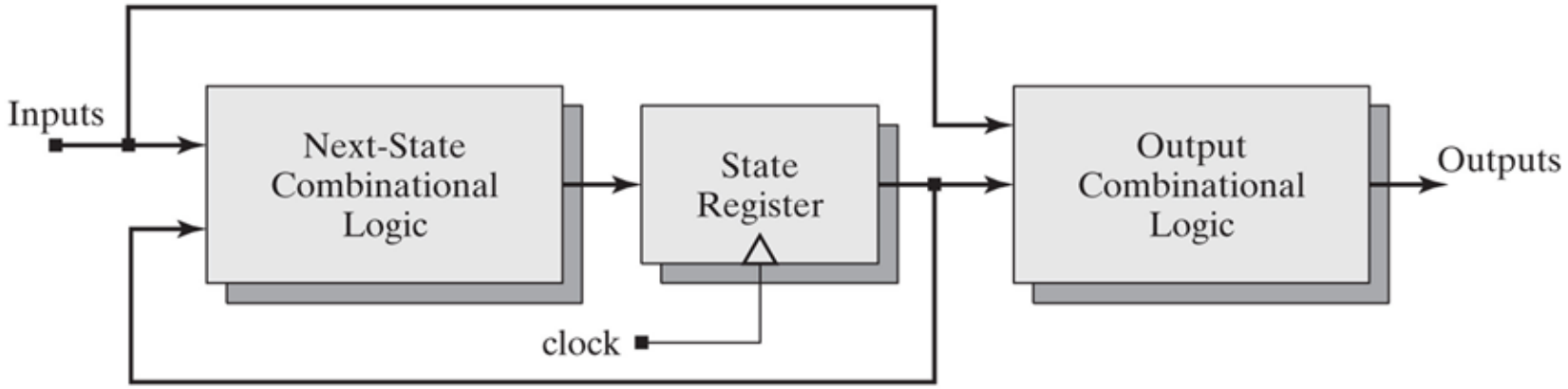
```
module pipelined (
input [31:0] A, B, C;
input clk;
output [63:0]  out);

reg [63:0] C_pipe1;
reg [63:0] out_pipe1, out_pipe2;

always @ (posedge clk) begin
   out_pipe1 <= A*B;
   C_pipe1 <= C;
end

always @(posedge clk)
   out_pipe2 <= out_pipe1 + C_pipe1;

assign out = out_pipe2;

endmodule
```

Moore vs. Mealy FSM

- Moore FSM
  - output is a function of only present state

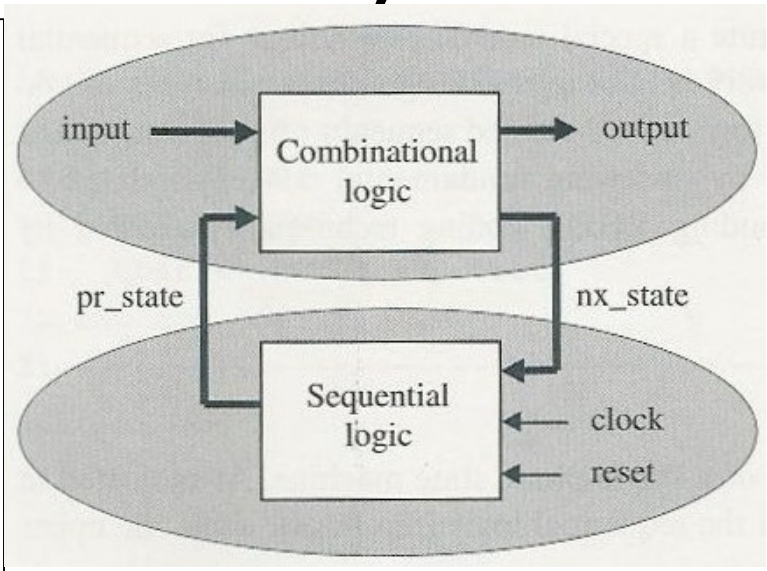


- Mealy FSM
  - output is a function both present state and input



FSM (Finite State Machine)

```
// lower section of FSM
always @ (posedge clk, posedge rst) begin
   if (rst == 1) pr_state <= … ;
   else          pr_state <= nx_state;
   // out <= temp_out;     // for stored output
end

// upper section of FSM
always @ (pr_state, input) begin
case pr_state
   state1: begin
             tmp_out = ...; // Moore machine
             if ( input == …) nx_state = …;
                          else nx_state = …;
             end
   state2: …
    …
endcase
end

// output section for not non-stored output
assign out = tmp_out;
```



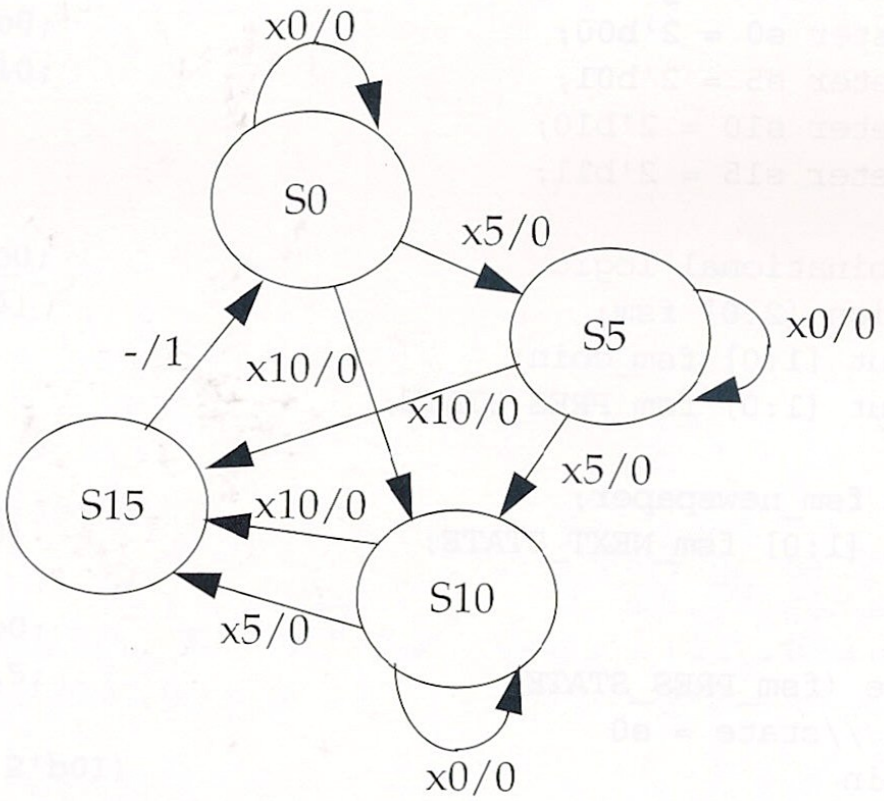
Mealy FSM:
output = f(input, pr_state)
nx_state = g(input, pr_state)

Example: Newspaper Vendor Machine

. newspaper costs 15 cents
. accept coins of nickels (5 cents)
  and dimes (10 cents) only
. do not return extra money

State	*Money*
S0	*0 cents*
S5	*5 cents*
S10	*10 cents*
S15	*15 cents*

Input	*coin[1:0]*
x0	*2'b00*
x5	*2'b01*
x10	*2'b10*
-	*don't care*

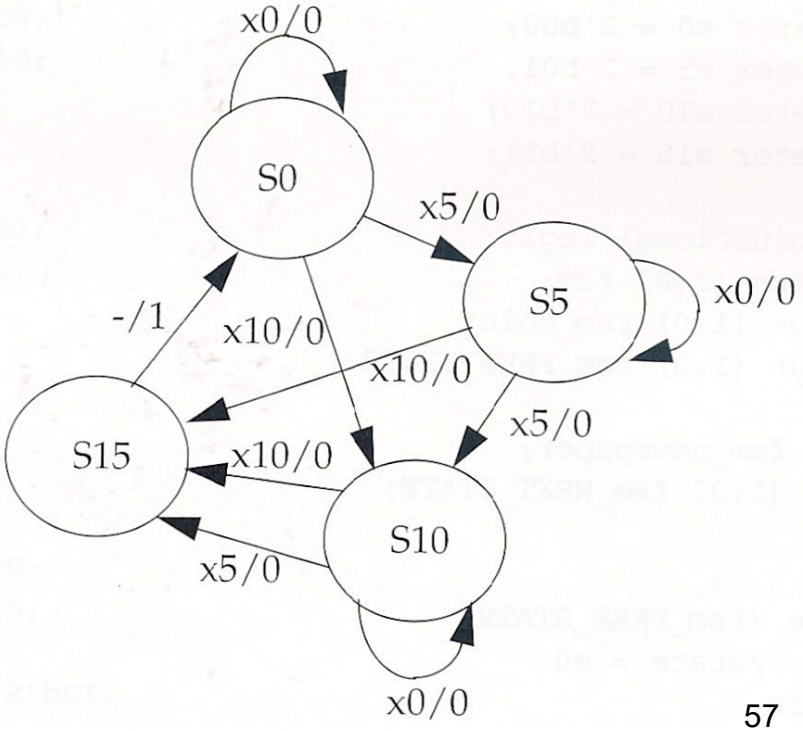


FSM Example: Vendor Machine (1/7) binary encoding of states

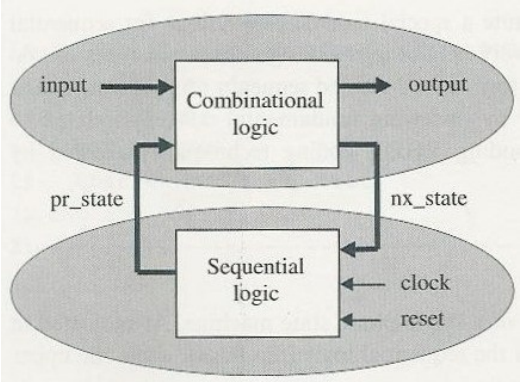
```
module vend(coin, clock, reset, newspaper);
input [1:0] coin;
input clock;
input reset;
output newspaper;
wire newspaper;
wire [1:0] NEXT_STATE;
reg [1:0] PRES_STSTE;

// state encodings (binary)
// cp. one-hot encoding
parameter s0 = 2'b00;
parameter s5 = 2'b01;
parameter s10 = 2'b10;
parameter s15 = 2'b11;
```

State	*Money*
S0	*0 cents*
S5	*5 cents*
S10	*10 cents*
S15	*15 cents*

Input	*coin[1:0]*
x0	*2'b00*
x5	*2'b01*
x10	*2'b10*
-	*don't care*



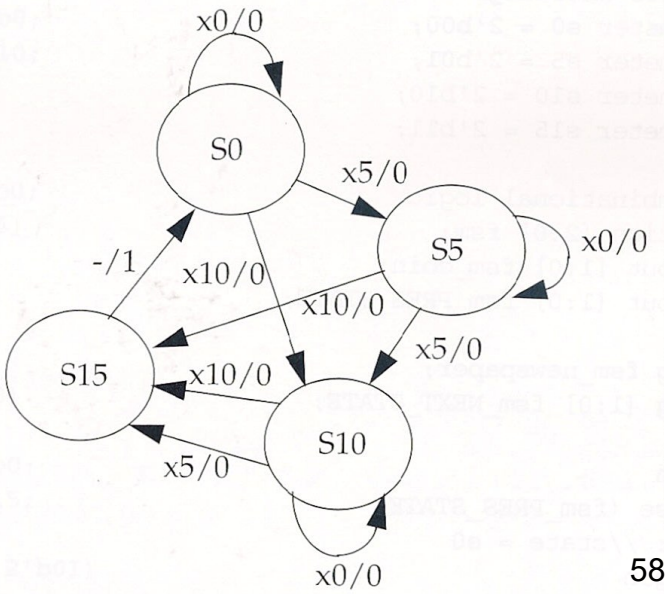
FSM Example: Vendor Machine (2/7) generate output/next states (combinational logic in upper FSM)



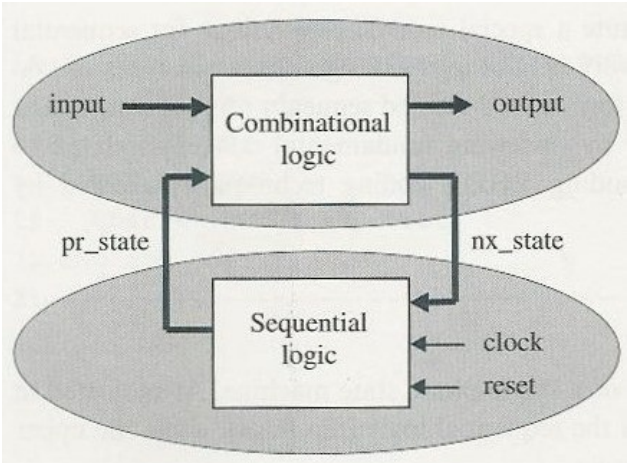

```
// combinational logic
function [2:0] fsm;
input [1:0] fsm_coin;
input [1:0] fsm_PRES_STATE;
reg fsm_newspaper;
reg [1:0] fsm_NEXT_STATE;
begin

case (fsm_PRES_STATE)
```

State	*Money*
S0	*0 cents*
S5	*5 cents*
S10	*10 cents*
S15	*15 cents*

Input	*coin[1:0]*
x0	*2'b00*
x5	*2'b01*
x10	*2'b10*
-	*don't care*



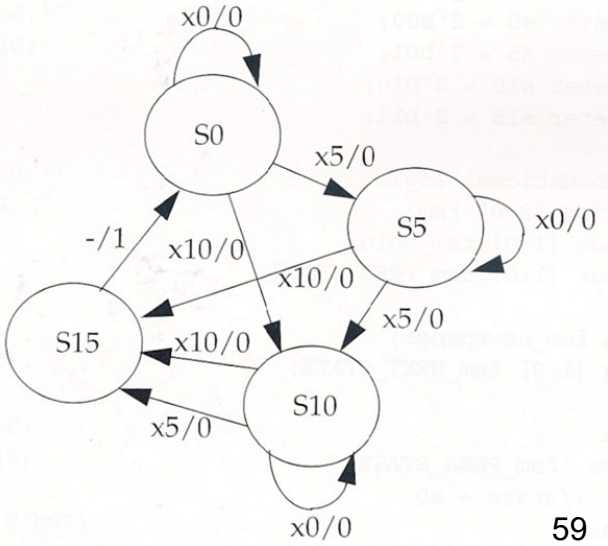
FSM Example: Vendor Machine (3/7) generate output and next state for each present state




```
s0: // state = s0
begin

if (fsm_coin == 2'b10) begin
        fsm_newspaper = 1'b0;
        fsm_NEXT_STATE = s10; end
else if (fsm_coint == 2'b01) begin
        fsm_newspaper = 1'b0;
        fsm_NEXT_STATE = s5; end
else begin
        fsm_newspaper = 1'b0;
        fsm_NEXT_STATE = s0; end

end
```

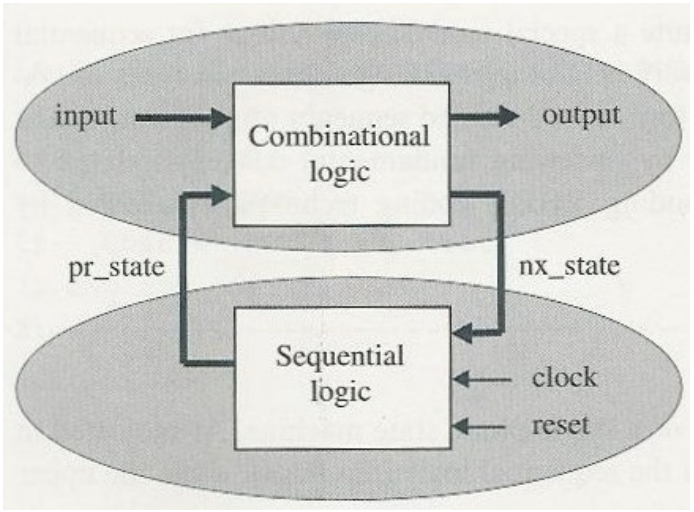
State	*Money*
S0	*0 cents*
S5	*5 cents*
S10	*10 cents*
S15	*15 cents*

Input	*coin[1:0]*
x0	*2'b00*
x5	*2'b01*
x10	*2'b10*
-	*don't care*



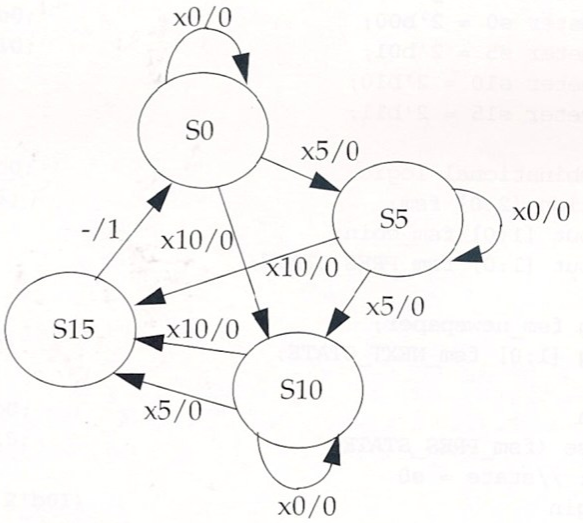
FSM Example: Vendor Machine (4/7) generate output and next state for each present state

```
s5: // state = s5
begin

if (fsm_coin == 2'b10) begin
        fsm_newspaper = 1'b0;
        fsm_NEXT_STATE = s15; end
else if (fsm_coint == 2'b01) begin
        fsm_newspaper = 1'b0;
        fsm_NEXT_STATE = s10; end
else begin
        fsm_newspaper = 1'b0;
        fsm_NEXT_STATE = s5; end

end
```

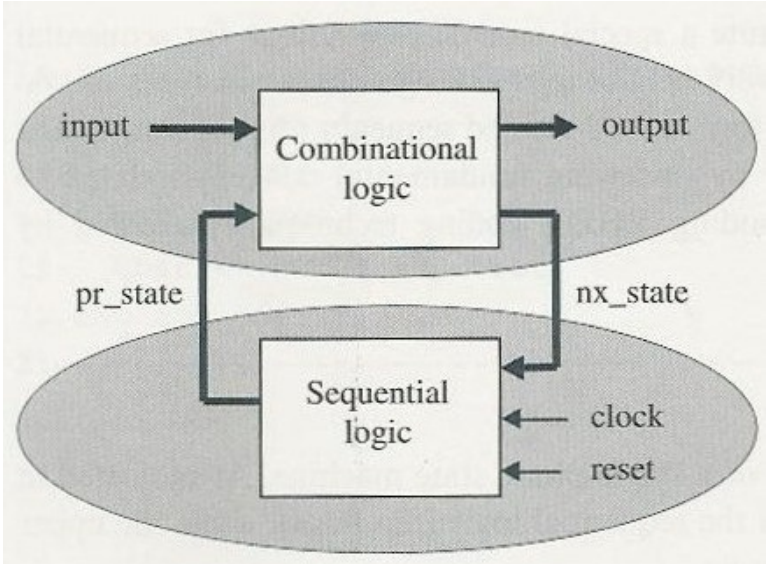


State	Money
S0	0 cents
S5	5 cents
S10	10 cents
S15	15 cents

Input	coin[1:0]
x0	2'b00
x5	2'b01
x10	2'b10
-	don't care



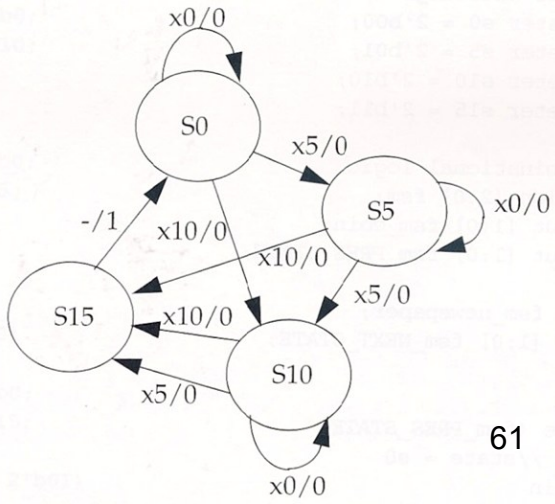
FSM Example: Vendor Machine (5/7) generate output and next state for each present state




```
s10: // state = s10
begin

if (fsm_coin == 2'b10) begin
        fsm_newspaper = 1'b0;
        fsm_NEXT_STATE = s15; end
else if (fsm_coint == 2'b01) begin
        fsm_newspaper = 1'b0;
        fsm_NEXT_STATE = s15; end
else begin
        fsm_newspaper = 1'b0;
        fsm_NEXT_STATE = s10; end

end
```

State	Money
S0	0 cents
S5	5 cents
S10	10 cents
S15	15 cents

Input	coin[1:0]
x0	2'b00
x5	2'b01
x10	2'b10
-	don't care

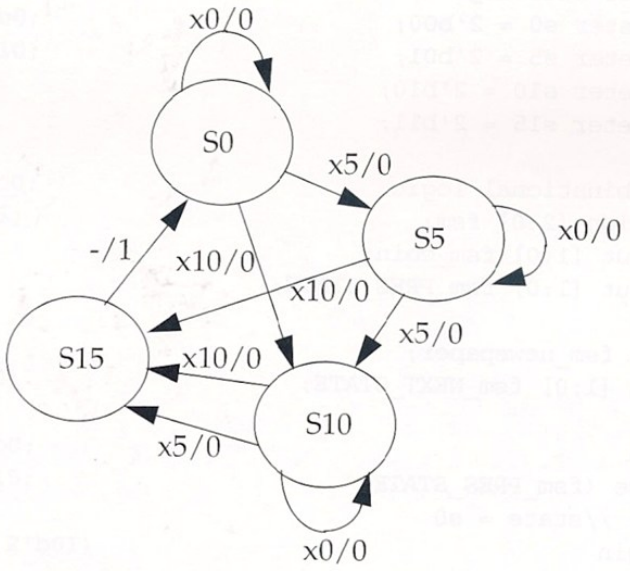


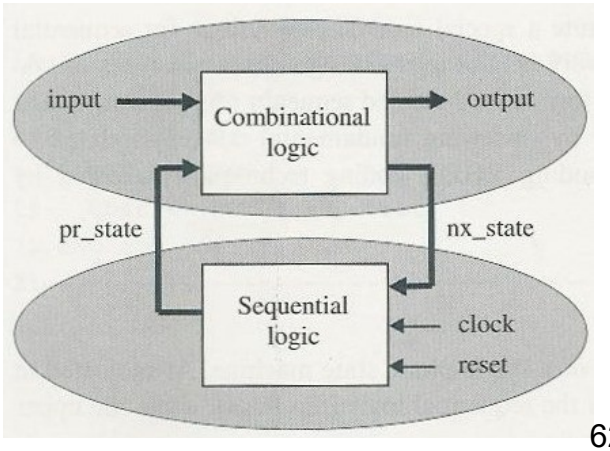
FSM Example: Vendor Machine (6/7) generate output and next state for each present state

```
s15: // state = s15
begin
        fsm_newspaper = 1'b1;
        fsm_NEXT_STATE = s0;
end

endcase



fsm = {fsm_newspaper, fsm_NEXT_STATE};

end


endfunction
```

State	Money
S0	0 cents
S5	5 cents
S10	10 cents
S15	15 cents

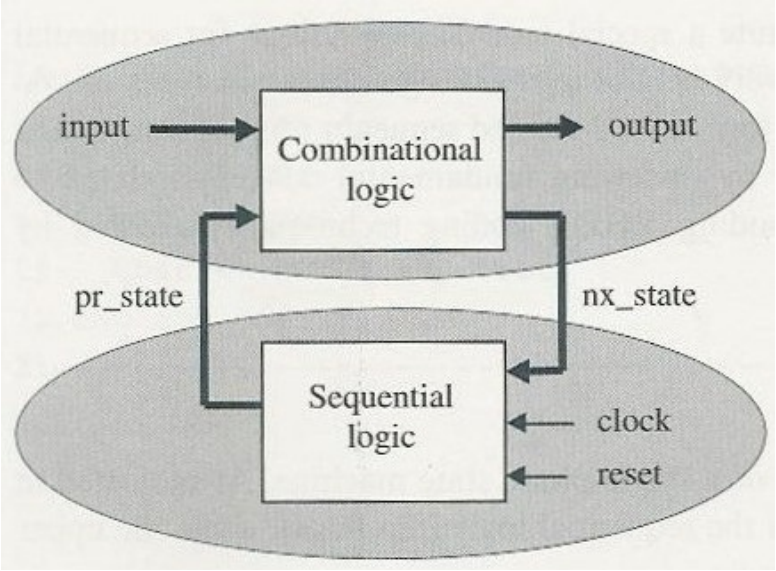
Input	coin[1:0]
x0	2'b00
x5	2'b01
x10	2'b10
-	don't care





FSM Example: Vendor Machine (7/7) store next state (sequential logic in lower FSM)

```
// reevaluate combinational logic each time a coin is put or
//  the present state changes
assign {newspaper, NEXT_STATE} = fsm(coin, PRES_STATE);

// clock the state FFs using synchronous reset
always @(posedge clock)
begin
    if (reset == 1'b)
        PRES_STSTE <= s0;
    else
        PRES_STATE <= NEXT_STSTE;
end

endmodule
```

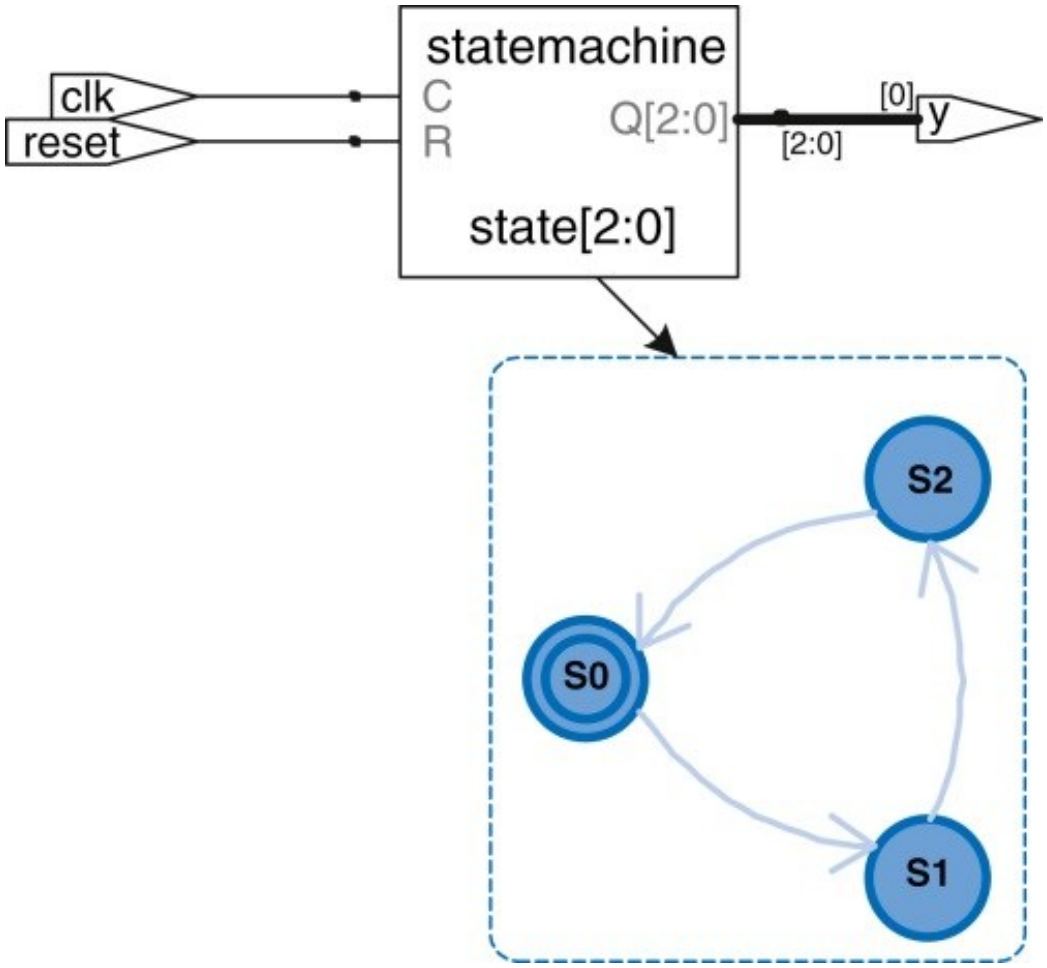


Stimulus for the FSM Vendor Machine

```
module stimulus;
reg clock, reset
reg [1:0] coin;
wire newspaper;
vend vendY(coin, clcok, reset,
    newspaper);
always
#20 clcok = ~clock;

initial
begin
clock = 0;
coin = 0;
reset = 1;
#50 reset = 0;
@(negedge clock);
// put 3 nickles
#80 coin=1; #40 coin=0;
#80 coin=1; #40 coin=0;
#80 coin=1; #40 coin=0;
// put 1 nickle and 1 dime
#180 coin=1; #40 coin=0;
#80 coint=2; #40 coin=0;
// put 2 dimes
#180 coin=2; #40 coin=0;
#80 coint=2; #40 coin=0;
// put 1 dime and 1 nickle
#180 coin=2; #40 coin=0;
#80 coint=1; #40 coint=0;

#80 $finish;
end
endmodule
```

Divide-by-3 counter using FSM

```
module divideby3FSM (input clk, input reset, output y);
reg [1:0] state, nextstate;
parameter S0 = 2'b00; //binary encoding
parameter S1 = 2'b01;
parameter S2 = 2'b10;

  // state register
  always @(posedge clk, posedge reset)
    if (reset) state <= S0;
    else       state <= nextstate;

  // next state logic
  always @ (*)
    case (state)
      S0:      nextstate = S1;
      S1:      nextstate = S2;
      S2:      nextstate = S0;
      default: nextstate = S0;
    endcase

  // output logic
  assign y = (state == S0); // duty cycle = 1/3 = 33%)

endmodule
```



divide-by-5 clock with ~1/2 duty cycle

- count the accumulated number of clk ecges
  - clk_ou=1 if count = 5/2 = 2,
  - reset count and clk_out if count = M = 5
- wo methods

```
// M=5
reg clk_out;
reg. [3:0] counter;
always @ (posedge clk) begin
counter <= counter+1;
if (counter == M/2)
     clk_out <= 1;
else if (counter == M) begin
          clk_out <= 0;
          counter <= 0; end
end
end
```

```
always @ (posedge clk)
   state <= next+state;

always @ (*) begin
case state
s0: begin next_state = s1; clk_out =0; end
s1: begin next_state = s2; clk_out =0; end
s2: begin next_state = s3; clk_out =1; end
s3: begin next_state = s4; clk_out =1; end
s4: begin next_state = s0; clk_out =1; end
endcase
end
```

Shift Registers

x[0] <= d

x[1] <= x[0]

x[2] <= x[1]

…

x[N-2] <= x[N-3]

out <= x[N-2]

```
// a total of N shift registers, each bw bits
reg [bw-1:0] x[0:N-2];
reg [bw-1:0] d, out;
always @ (posedge clk) begin
   x[0] <= d;
   for (i=1; i<=N-2; i=i+1)
      x[i] <= x[i-1];
   out <= x[N-2];
end
```

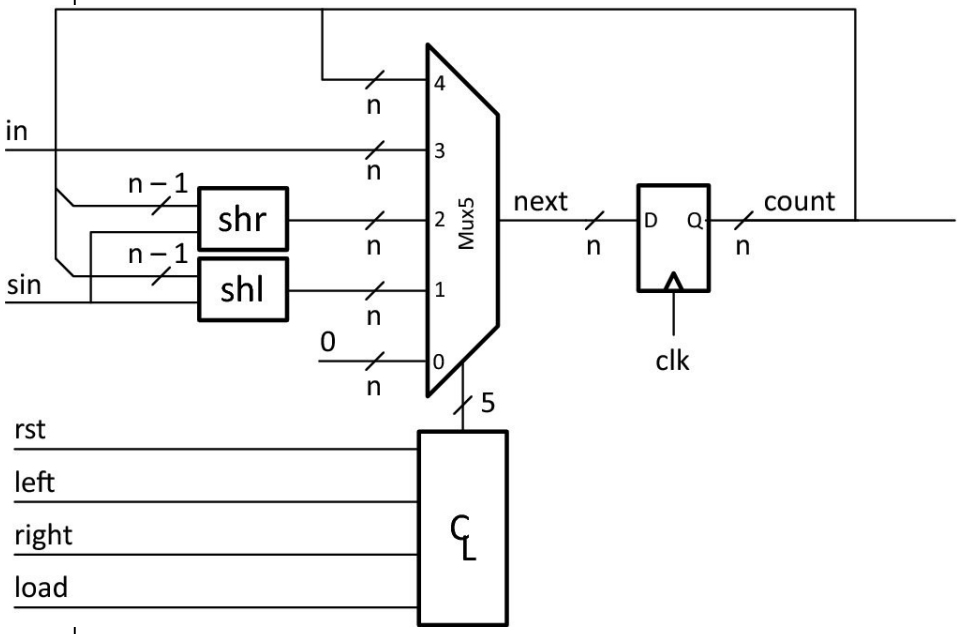
```
// a total of N shift registers
reg [bw-1:0] x[0:N-2];
reg [bw-1:0] d, out;
always @ (posedge clk) begin
   {x[0:N-2], out} <= {d, x[0:N-2]};
end
```

shift register with parallel load

```
module shiftreg #(parameter N=32)
(input clk, reset, load, si,
input [0:N-1] d,
output [0:N-1] q,
output so);

always @(posedge clk)

if (reset) q <= 0;
else if (load) q <= d;
else {q, so}  <= {si, q};

endmodule
```

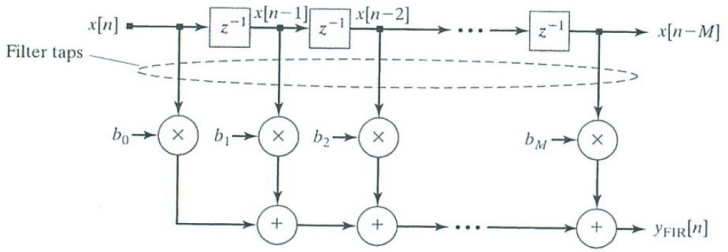
left/right/load shift register

```
module LRL_Shift_Register(clk, rst, left,
right, load, sin, in, out) ;
  parameter n = 4 ;
  input clk, rst, left, right, load, sin ;
  input [n-1:0] in ;
  output [n-1:0] out ;
  reg [n-1:0] next ;

  DFF #(n) cnt(clk, next, out) ;
// always @ (posedge clk)  out <= next;

  always @(*) begin
    casex({rst,left,right,load})
      4'b1xxx: next = 0 ;                // reset
      4'b01xx: next = {out[n-2:0],sin} ; // left
      4'b001x: next = {sin,out[n-1:1]} ; // right
      4'b0001: next = in ;               // load
      default: next = out ;              // hold
    endcase
  end
endmodule
```



FIR (Finite Impulse Response) Filter

- combinational logic of y=y+1 has the path from output to input, creating a feedback loop !




```
// wrong codes,
// Why?
…
reg [word_size_in-1:0] x[0:M];
reg [word_size_out-1:0] y;

y=0;
always @ (x)
   for (i=0; i<=M-1; i++)
      y = y + x[i]*b[i];
…
```

```
…
parameter bw=8, bw_out=16;
reg [bw-1:0] x[0:M-1]. b[0:M-1];
reg [bw_out-1:0] temp[0:M];
reg [bw_out-1:0] y;

always @ (posedge clk)
   x [0:M-1] <= {in, x[0:M-2]};

temp[0]=0;
always @ (x)
   for (i=0; i<=M-1; i++)
      temp[i+1] = temp[i] + x[i]*b[i];
assign  y = temp[M];
…
```

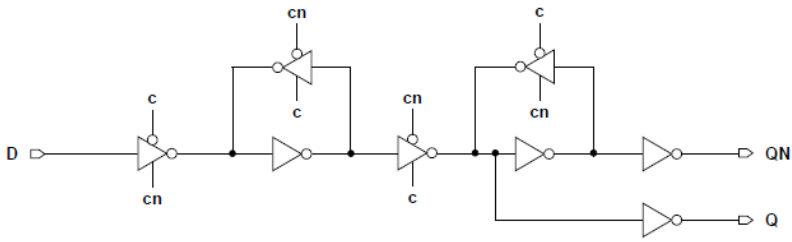
Quiz

- how about one-hot state encoding?
  - parameter S0 = 3'b001
  - parameter S1 = 3'b010
  - parameter S2 = 3'b100
  - cp. binary state encoding
- how to design divide-by-5 counter with duty cycle= 1/5=20% ?
- how to design divide-by-5 counter with duty cycle ~= 50% (e.g. 3/5) ?

Memory (ROM, RAM)

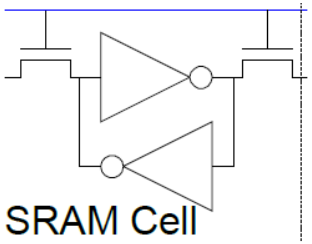
RAM vs. Registers

- Registers
  - use array to model registers
  - e.g., reg [31:0] R [0:15];
    - ✓ register file with 16 registers, each with 32 bits
    - ✓ realized using 16*32 FFs
  - many register locations can be accessed simultaneously
    - ✓ e.g., R[3] = R[0] + R]1] + R[2];
  - much more area than RAM
- RAM
  - need memory compiler tool to generate RAM
  - only the specified address can be accessed at a time
  - much smaller area than registers

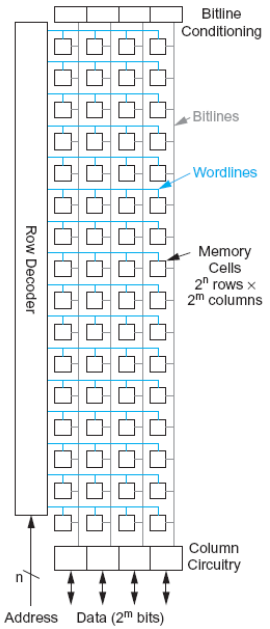


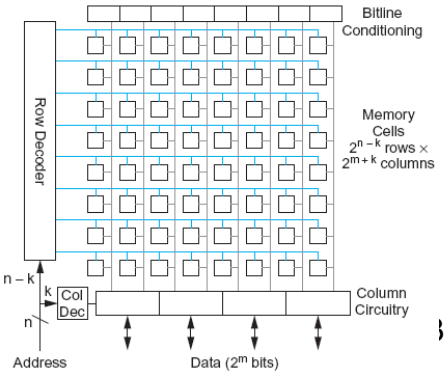
24 transistors in a flip-flop

6 transistors in a sram cell



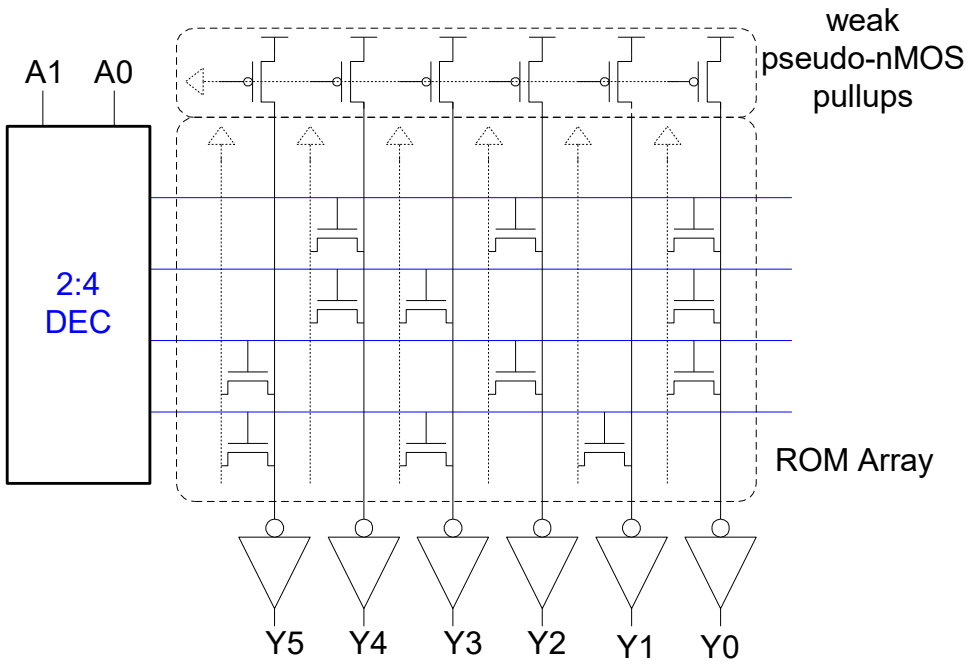
SRAM Cell



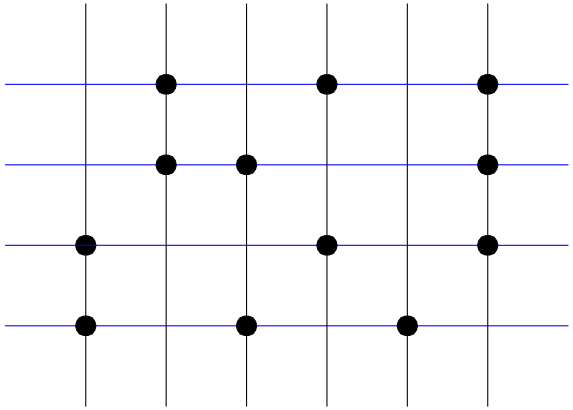


ROM vs. Random Logic

- Use **case** LookUp Table *LUT) to specify ROM
  - synthesized with random logic (such as Sum-of-Product Boolean equations)
- Use ROM compiler tool to implement ROM




```
Word 0: 010101
Word 1: 011001
Word 2: 100101
Word 3: 101010
```



Synthesis of ROM, RAM

- use synthesis pragma and attribute
  - (* synthesis, rom_block = “ROMCELLXYZ01” *)
  - (* synthesis, ram_block *)
  - without synthesis attributes, ROM is usually synthesized into combinational random logic, and synchronous RAM is usually synthesized into flip-flops
    - ✓ much larger area
- declare as 1-D array (e.g., bit-vector) for address and for memory cell
  - specify s ROM contents using case … endcase
- declare as 2-D array
  - specify ROM content using either of the following two ways
    - ✓ initial begin …. end
    - ✓ read from inputfiles

Synthesis of ROM/RAM

- If synthesis tool supports *pragmas* to control the structure of the synthesized netlist or to give direction to the synthesis tool, ***synthesis attributes*** shall be used
  - The first attribute within the attribute instance shall be *synthesis* followed by a comma separated list of synthesis-related attributes.

```
(* synthesis, <attribute=value_or_optional_value>
             { , <attribute=value_or_optional_value> } *)

(* synthesis, async_set_reset [="signal_name1, signal_name2, ..."] *)
(* synthesis, black_box              [ =<optional_value> ] *)
(* synthesis, combinational          [ =<optional_value> ] *)
(* synthesis, fsm_state              [ =<encoding_scheme> ] *)
(* synthesis, full_case              [ = <optional_value> ] *)
(* synthesis, implementation         = "<value>" *)
(* synthesis, keep                   [ =<optional_value> ] *)
(* synthesis, label                  = "name" *)
(* synthesis, logic_block            [ = <optional_value> ] *)
(* synthesis, op_sharing             [ = <optional_value> ] *)
(* synthesis, parallel_case          [ = <optional_value> ] *)
(* synthesis, ram_block              [ = <optional_value> ] *)
(* synthesis, rom_block              [ = <optional_value> ] *)
(* synthesis, sync_set_reset  [="signal_name1, signal_name2, ..."] *)
(* synthesis, probe_port             [ = <optional_value> ] *)
```

Ref.: IEEE Std. 1364.1 Verilog Register Transfer Level Synthesis

Two Implementations of ROM
(using case, or using input text file)

```
module rom_case (a, d) ;
  input [3:0] a;
  output [7:0] d;
  reg [7:0]  d;
  always@(*)
  begin
    case(a)
      4'h0: d=8'h00;
      4'h1: d=8'h11;
      4'h2: d=8'h22;
      4'h3: d=8'h33;
      4'h4: d=8'h44;
      4'h5: d=8'h12;
      4'h6: d=8'h34;
      4'h7: d=8'h56;
      4'h8: d=8'h78;
      4'h9: d=8'h9a;
      4'ha: d=8'hbc;
      4'hb: d=8'hde;
      4'hc: d=8'hf0;
      4'hd: d=8'h12;
      4'he: d=8'h34;
      4'hf: d=8'h56;
      default: d=8'h0;
    endcase
  end
endmodule
```

```
module rom_reg (a, d) ;
  parameter b =8;
  parameter w = 4;
  parameter fileName = "datafile";
  input [w-1:0] a;
  output [b-1:0] d;
  reg [b-1:0]    memory [2**w-1:0] ;
  initial
  begin
    $readmemb (fileName, memory);
  end
  assign d = memory[a];
endmodule
```

```
// content of file “datafile”
// address in hexadecimal
@002
11111111  01010101
00000000  10101010
@006
1111zzzz  00001111
…
```

```
memory[0]=xxxxxxxx
memory[1]=xxxxxxxx
memory[2]=11111111
memory[3]=01010101
memory[4]=00000000
memory[5]=10101010
memory[6]=1111zzzz
memory[7]=00001111
…
```

ROM

- Three types of ROM models
  - one-dimensional array with data in case statement
  - two-dimensional array with data in initial statement
  - two-dimensional array with data in text file

Ref.: IEEE Std. 1364.1 Verilog Register Transfer Level Synthesis

ROM: 1D array using case

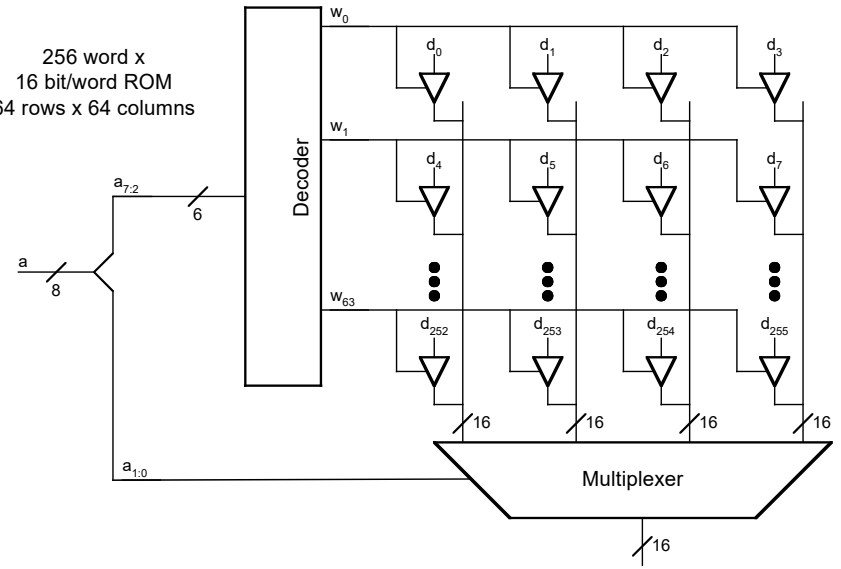
- Synthesis attribute ***rom_block*** model ROM
- Attribute ***logic_block*** implies combinational logic

```
module rom_case(
      (* synthesis, rom_block = "ROM_CELLXYZ01" *)
      output reg [3:0] z,
      input wire [2:0] a); // Address - 8 deep memory.

   always @*      // @(a)
      case (a)
         3'b000: z = 4'b1011;
         3'b001: z = 4'b0001;
         3'b100: z = 4'b0011;
         3'b110: z = 4'b0010;
         3'b111: z = 4'b1110;
         default: z = 4'b0000;
      endcase
endmodule // rom_case
// z is the ROM, and its address size is determined by a.
```

ROM: 2D array using initial

- initial statement shall be supported when synthesis attributes ***logic_block*** or ***rom_block*** is used
- without specifying attributes, a synthesis tool may opt to implement either as random combintional logic or as a ROM

```
module rom_2dimarray_initial (
      output wire [3:0] z,
      input wire [2:0] a); // address- 8 deep memory
// Declare a memory rom of 8 4-bit registers. The indices are 0 to 7:
   (* synthesis, rom_block = "ROM_CELL XYZ01" *) reg [3:0] rom[0:7];
   // (* synthesis, logic_block *) reg [3:0] rom [0:7];

   initial begin
      rom[0] = 4'b1011;
      rom[1] = 4'b0001;
      rom[2] = 4'b0011;
      rom[3] = 4'b0010;
      rom[4] = 4'b1110;
      rom[5] = 4'b0111;
      rom[6] = 4'b0101;
      rom[7] = 4'b0100;
   end

   assign Z = rom[a];
endmodule
```



ROM: 2D array with data in text file

- use **$readmemb** to read ROM data from a text file

```
module rom_2dimarray_initial_readmem (
      output wire [3:0] z,
      input wire [2:0] a);
      // Declare a memory rom of 8 4-bit registers.
      // The indices are 0 to 7:
   (* synthesis, rom_block = "ROM_CELL XYZ01" *) reg [3:0] rom[0:7];

   initial $readmemb("rom.data", rom);

   assign z = rom[a];
endmodule

// Example of content “rom.data” file:
   // file: /user/name/project/design/rom/rom.data
   // date : Jan 08, 02
   1011     // addr=0
   1000     // addr=1
   0000     // addr=2
   1000     // addr=3
   0010     // addr=4
   0101     // addr=5
   1111     // addr=6
   1001     // addr=7
```

RAM

Edge-sensitive RAM

- RAM shall be modeled with synthesis attribute ***ram_block***
- If latch or register logic is desired instead of a RAM, use the attribute ***logic_block*** instead of ***ram_block***
- RAM could be ***edge***- or ***level***-sensitive
- e.g. *we* (write-enable) control signal is synchronized with clock signal

```
// A RAM element is an edge-sensitive storage element:
   module ram_test(
      output wire [7:0] q,
      input wire [7:0] d,
      input wire [6:0] a,
      input wire clk, we);
   (* synthesis, ram_block *) reg [7:0] mem [127:0];

   always @(posedge clk) if (we) mem[a] <= d;

   assign q = mem[a];
endmodule
```

level-sensitive RAM

- we (write-enable) is the level-sensitive control signal
  - no clock signal is used

```
// A RAM element is a level-sensitive storage element:
module ramlatch (
      output wire [7:0] q, // output
      input  wire [7:0] d, // data input
      input  wire [6:0] a, // address
      input  wire       we); // clock and write enable
   // Memory 128 deep, 8 wide:
   (* synthesis, ram_block *) reg [7:0] mem [127:0];

   always @* if (we) mem[a] <= d;

   assign q = mem[a];
endmodule
```

dual-port (one-read, one-write) RAM

- check definitions of two-port or dual-port in document
  - 1R/1W, 2R, or 2W, or
  - 1R/1W only

```
module RAM(ra, wa, write, din, dout) ;
   parameter b = 32;
   parameter w = 4;

   input [w-1:0] ra, wa;
   input         write;
   input [b-1:0] din;
   output [b-1:0] dout;

   reg [b-1:0]      ram [2**w-1:0];

   assign dout = ram[ra];  // one read port

   always@(*) begin
      if(write == 1)
        ram[wa] = din;  // one write port
   end
endmodule
```

RAM with bidirectional data bus

```
module ram #(parameter N=6, M=32)
(input clk, we,
input [N-1:0] adr,
inout [M-1:0] data);

reg [M-1:0] mem [2**N-1:0];

always @ (posedge clk)
   if (we) mem[adr] <= data;

assign data = we ? 'z : mem[adr];

endmodule
```

RAM: 2D array using always

- in general, standard cell library vendors or FPGA synthesis tools usually provide automatic generator (compiler) of memory, such as RAM, ROM, register file which are usually custom designs

```
// A RAM element is an edge-sensitive storage element:
   module ram_test(
      output wire [7:0] q,
      input wire [7:0] d,
      input wire [6:0] a,
      input wire clk, we);
   (* synthesis, ram_block *) reg [7:0] mem [127:0];

   always @(posedge clk) if (we) mem[a] <= d;

   assign q = mem[a];
endmodule
```

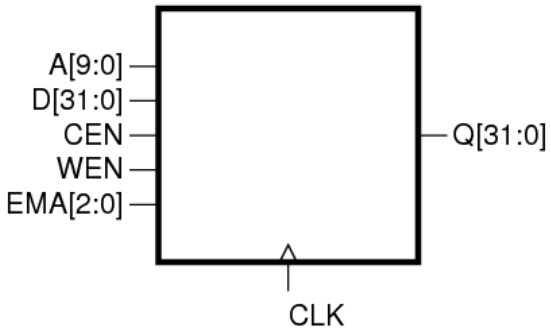
ARM TSMC Memory Compiler

TSMC 90nm Memory Compiler

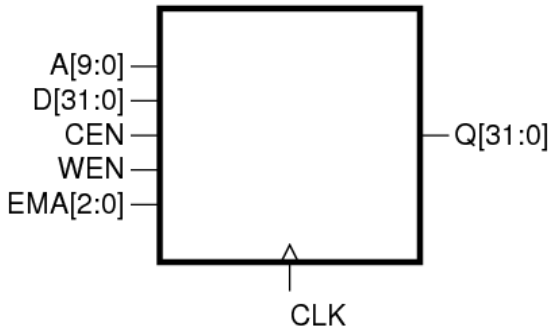
- 90nm

TSMC_90nm	Register file	SRAM
Single-port	RF_SP_ADV	SRAM_SP_ADV
Two-port	RF_2P_ADV	-
Dual-port	--	SRAM_DP_ADV

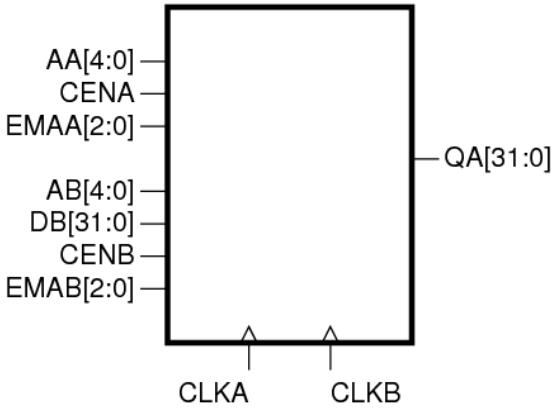
single-port,



two-port ,

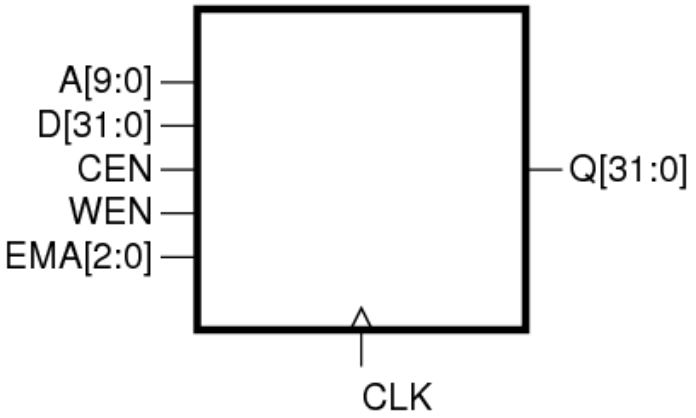


dual-port



TSMC_40nm	Register file	SRAM
Single-port	RF_SP_HDE (rvt_hvt_rvt) RF_SP_HSD (rvt_rvt_hvt)	SRAM_SP_HDE (rvt_hvt_rvt) SRAM_SP_HSC (rvt_hvt_rvt)
Two-port	RF_2P_HSE (rvt_hvt_rvt)	-
Dual-port	--	SRAM_DP_HDE (rvt_hvt_rvt)

TSMC 90nm Single-Port Pins



Pin	Description
A[9:0]	Addresses (A[0] = LSB)
D[31:0]	Data Inputs (D[0] = LSB)
CLK	Clock
CEN	Chip Enable (active low)
WEN	Write Enable (active low)
Q[31:0]	Data Outputs (Q[0] = LSB)
EMA[2:0]	Extra Margin Adjustment (EMA[0] = LSB)

TSMC 40nm Single-Port Pins

- more complicate pins

Pin	Descrption
CEN	Chipe Enable (active low)
WEN	Write Enable (active low)
A	Addresses(A[0]=LSB)
D	Data Inputs (D[0]=LSB)
Q	Data Outputs (Q[0]=LSB)
CLK	Clock
WENY	Multiplexor out (WEN CEN A D)
CENY	
AY DY	

Pin	Descrption
EMA	Extra Margin Adjustment
EMAW	
EMAS	
BEN	Bypass mode,active low
TEN (enable)	TEST MODE,active low (CEN WEN A D Q 同前面意思)
TCEN	
TWEN	
TA TD TQ	
RET1N	Retention mode, acitve low
STOV	Synchronous clock enable, acitve low

TSMC 90nm Memory: Register-File

- register-file-based memory

single-port

Single-Port 90nm Register File rf_sp_adv		
Parameter	Ranges	
Numbers of words	Mux=1	8 to 128
	Mux=2	16 to 256
	Mux=4	32 to 512
Numbers of bits	Mux=1	8 to 128
	Mux=2	4 to 128
	Mux=4	2 to 64
Total memory bits	Mux=1	64 to 16,384 bits
	Mux=2, 4	64 to 32,768 bits

two-port

Two-Port 90nm Register File rf_2p_adv		
Parameter	Ranges	
Numbers of words	Mux=1	8 to 128
	Mux=2	16 to 256
	Mux=4	32 to 512
Numbers of bits	Mux=1	2 to 128
	Mux=2	2 to 64
	Mux=4	2 to 32
Total memory bits	16 to 16,384 bits	

TSMC 90nm Memory: SRAM

- sram-based memory

single-port

Single-Port 90nm SRAM sram_sp_adv		
Parameter	Ranges	
Numbers of words	Mux=8	256 to 4096
	Mux=16	512 to 8192
	Mux=32	1024 to 16384
Numbers of bits	Mux=8	2 to 128
	Mux=16	2 to 64
	Mux=32	2 to 32
Total memory bits	512 to 524,288 bits	

dual-port

Dual-Port 90nm SRAM sram_dp_adv		
Parameter	Ranges	
Numbers of words	Mux=4	128 to 2048
	Mux=8	256 to 4096
	Mux=16	512 to 8192
Numbers of bits	Mux=4	2 to 128
	Mux=8	2 to 64
	Mux=16	2 to 32
Total memory bits	256 to 262,144 bits	

Memory Synthesis

- use memory compiler to generate .lib and .v files
  - select register-file or sram
  - select single-port, two-port, or dual-port
  - specify # of words, # of bits, mux_width
  - give module name (e.g., sram_sp_1024x32)
- add the memory modules into your design
  - first convert .lib into .db file for Synopsys DC synthesis
  - use module instantiation to add memory modules
  - combine with your other RTL codes, e.g., in top.v
- use Synopsys DC to synthesize the entire ckt.
  - synthesized ckt., e.g., in top_gate.v
- simulate with all .v files