Verilog Codes for Basic Digital Components

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
```

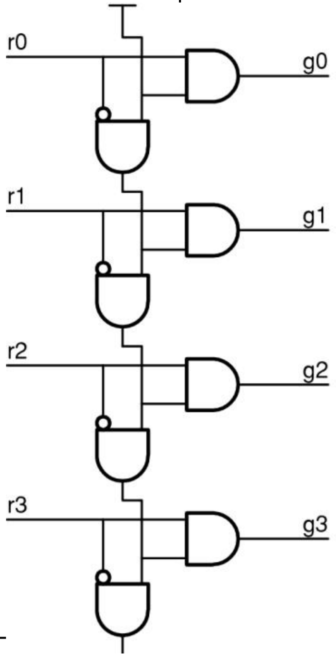
demultiplexer

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

always @(s, in0, in1, in2, in3)
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
```

arbiter

```
// arbiter 4-bit wide
//  LSB is highest priority
module Arb_4b_LSB (r, g);
output [3:0] r;
output [3:0] g;
reg [3:0] g;

always @ (*) begin
  casex (r)
    4'b0000: g = 4'b0000;
    4'bxxx1: g = 4'b0001;
    4'bxx10: g = 4'b0010;
    4'bx100: g = 4'b0100;
    4'b1000: g = 4'b1000;
    default: g = 4'bx;
  endcase
end
endmodule
```

```
// arbiter of arbitrary width:  LSB is highest priority
module Arb_LSB (r, g);
parameter n=4;
input [n-1:0] r;
output [n-1:0] g;
wire [n-1:0]
c = { (~r[n-2:0] & c[n-2:0]), 1'b1 };

assign g = r & c;
endmodule
```



```
// arbiter of arbitrary width:  MSB is highest priority
module Arb_MSB (r, g);
parameter n=4;
input [n-1:0] r;
output [n-1:0] g;
wire [n-1:0] c = { 1'b1, (~r[n-2:0] & c[n-2:0]) };

assign g = r & c;
endmodule
```

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
```

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;
paramete log2N = 3;
input [log2N-1: 0] in1;
output [N-1:0] out1;

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

endmodule
```

```
module decoder_loop (in1, out1);
parameter N=8;
paramete 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
```

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
```

leading one detector

```
`define TRUE 1'b1;
`define FALSE 1'b0;

module LOD (
input [15:0] flag;
output integer i) ;

reg continue;

always
begin
i = 0;  continue = `TRUE;
while ( i<16 && continue )
   begin
      if (flag[15-i]) continue = `FALSE;
      i=i+1;
   end
end
endmodule
```

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

assign b = g;
always @ (*) begin
   casex®
      8'b1xxxxxxx: g = 0;
      8'b01xxxxxx: g = 1;
      8'b001xxxxx: g = 2;
      8'b0001xxxx: g = 3;
      8'b00001xxx: g = 4;
      8'b000001xx: g = 5;
      8'b0000001x: g = 6;
      8'b00000001: g = 7;
      default: g = x;
   endcase
end
endmodule
```

ripple carry adder

```
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
```

left/right shifter

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

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

endmodule
```

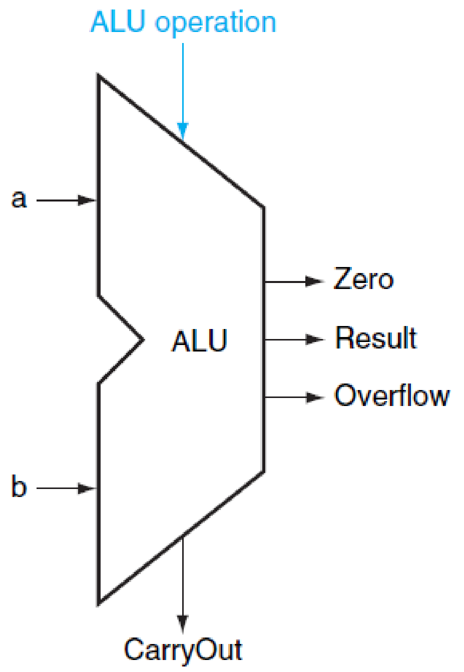
wrap-around barrel shifter

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

paremeter k=8;
parameter lk=3;
input [lk-1:0] n;
output [k-1:0] b;

wire [2*k-2:0] x = a << n;

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

endmodule
```

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



ROM

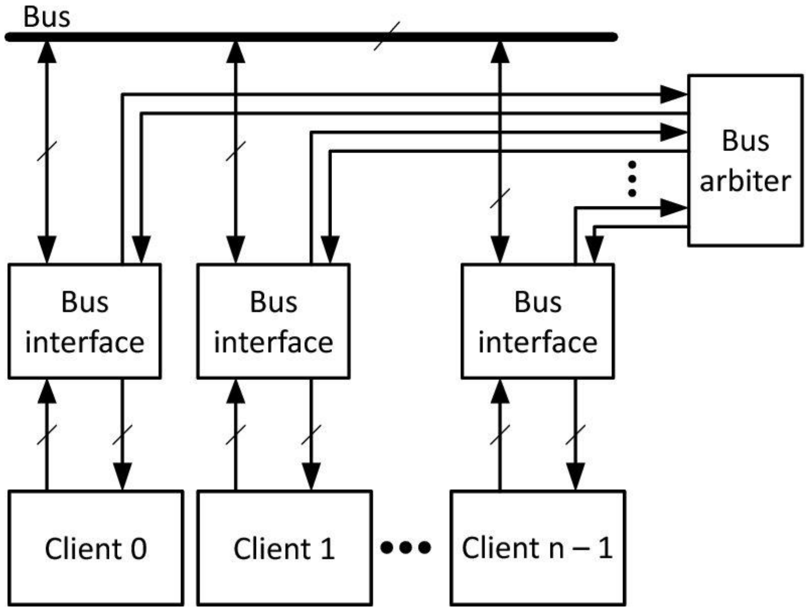
```
module rom_case (a, d) ;
  input [3:0] a;
  output [7:0] d;
  reg [7:0]  d;
  always@(*) begin
    case(a)
      4'h0: d=8'h00;
      4'h1: d=8'h11;
      4'h2: d=8'h22;
      4'h3: d=8'h33;
      4'h4: d=8'h44;
      4'h5: d=8'h12;
      4'h6: d=8'h34;
      4'h7: d=8'h56;
      4'h8: d=8'h78;
      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 = 32;
  parameter w = 4;
  parameter fileName = "dataFile";
  input [w-1:0] a;
  output [b-1:0] d;
  reg [b-1:0]    rom [2**w-1:0] ;
  initial begin
    $readmemb(fileName, rom);
  end
  assign d = rom[a];
endmodule
```

Sequential Logic

bus

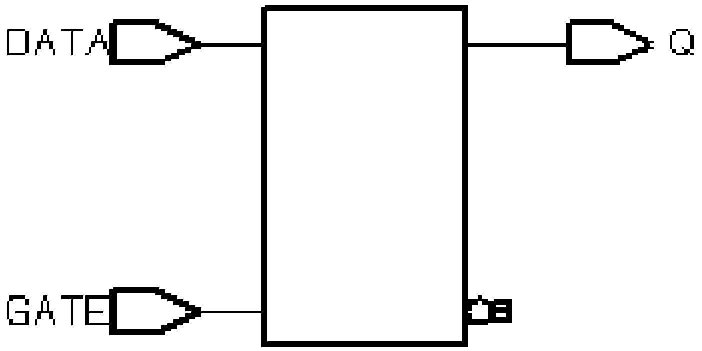
```
module BusInt(cr_valid, cr_ready, cr_addr, cr_data, // bus rx - to the bus
           ct_valid, ct_data,                // bus tx - from the bus
           br_addr, br_data, br_valid,         // to the bus
           bt_addr, bt_data, bt_valid,         // from the bus
           arb_req, arb_grant,                // the arbiter
           my_addr) ;                        // address of this interface
  parameter aw = 2 ; // address width
  parameter dw = 4 ; // data width
  input cr_valid, arb_grant, bt_valid ;
  output cr_ready, ct_valid, arb_req, br_valid ;
  input [aw-1:0] cr_addr, bt_addr, my_addr ;
  output [aw-1:0] br_addr ;
  input [dw-1:0] cr_data , bt_data ;
  output [dw-1:0] br_data, ct_data ;

  // arbitration
  assign arb_req = cr_valid ;
  assign cr_ready = arb_grant ;

  // bus drive
  assign br_valid = arb_grant ;
  assign br_addr = arb_grant ? cr_addr : 0 ;
  assign br_data = arb_grant ? cr_data : 0 ;

  // bus receive
  assign ct_valid = bt_valid & (bt_addr == my_addr) ;
  assign ct_data = bt_data ;
endmodule
```



latch

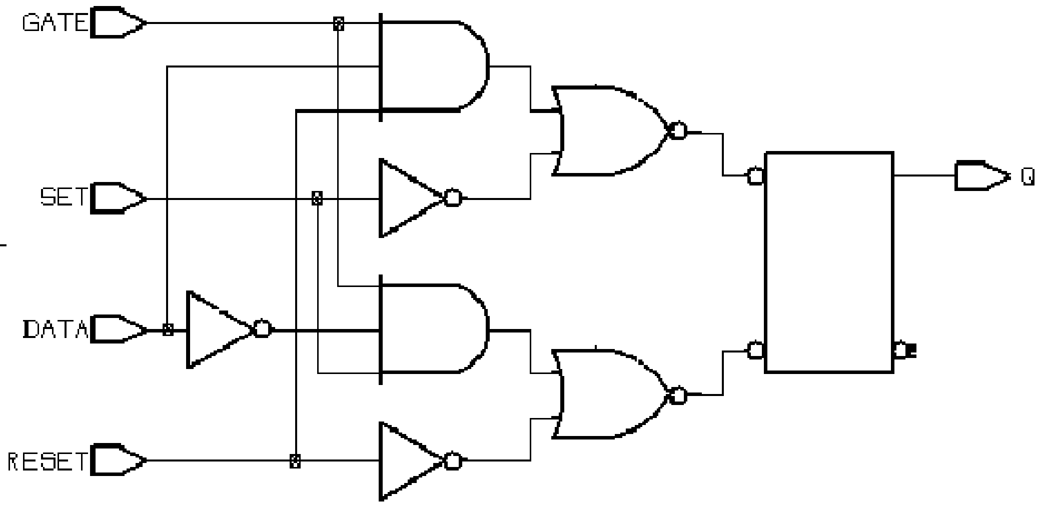
```
module d_latch (
input gate, data,
output reg q );

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
```



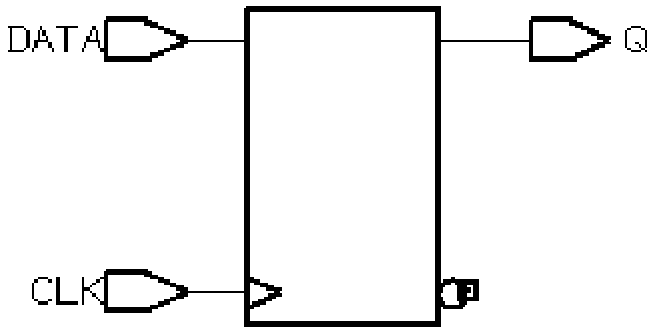
latch with asynchronous set and reset

```
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
```



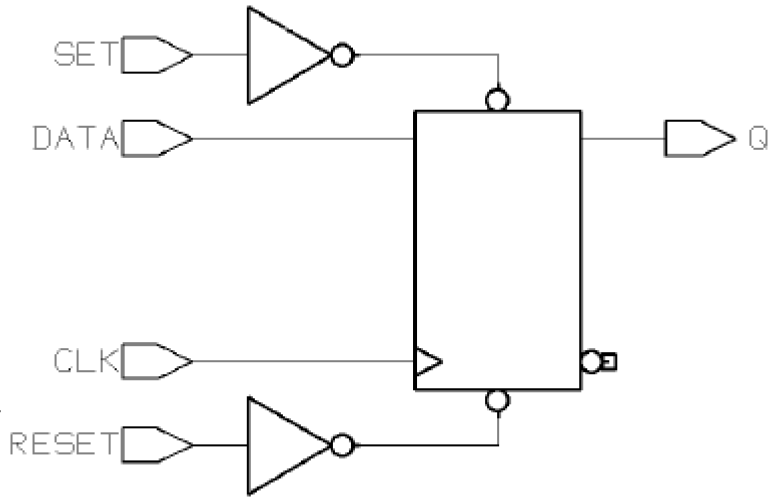
flip-flop

```
module dff (
input data, clk;
output reg q;

always @ (posedge)
   q <= data;

endmodule
```



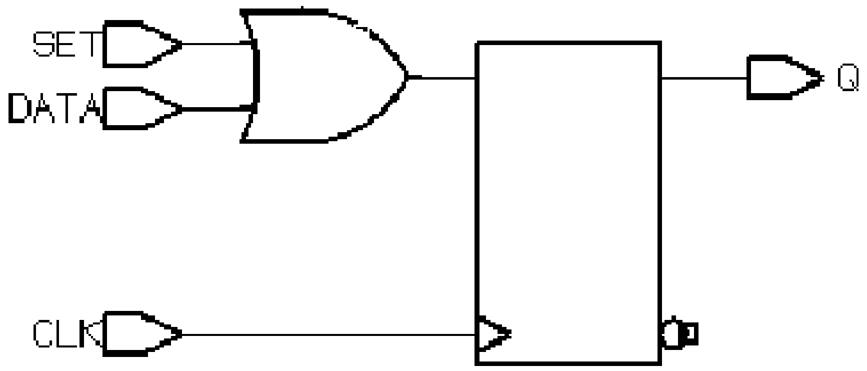
DFF with asynchronous set and reset

```
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
```



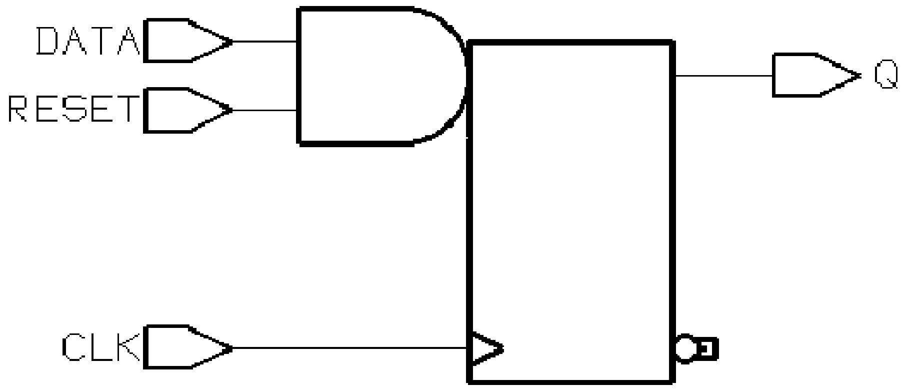
DFF with synchronous set

```
module dff_sync_set (
input data, clk, set;
output reg q;

always @ (posedge clk)
   if (set) q <= 1’b1;
   else q <= data;

endmodule
```



DFF with synchronous reset

```
module dff_sync_reset (
input data, clk, reset_n;
output reg q;

always @ (posedge clk)
   if (!reset_n) q<= 1’b0;
   else q <= data;

endmodule
```



register file

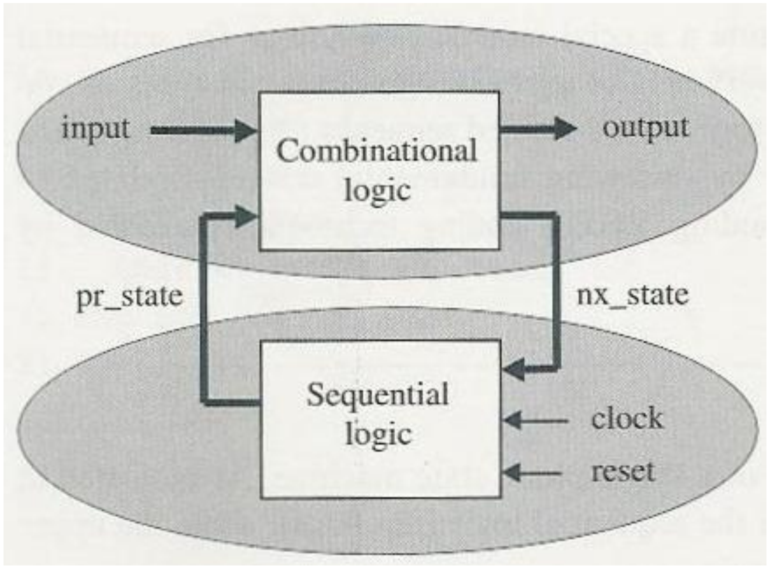
```
module RegisterFile (r_adr1, r_adr2, data1, data2, w_adr, w_data, w_cntl, clk)
input [5:0] r_adr1, r_adr2, w_adr;
input [31:0] w_data;
input w_cntl, clk;
output [31:0] data1, data2;
reg [31:0] RF [0:31];

assign data1 = RF[r_adr1];
assign data2 = RF[r_adr2];

always @ (posedge)
    if (w_cntl) RF[w_adr] = w_data;

endmodule
```

Pipeline

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

reg [31:0] 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
```

FSM

```
// 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 = ...;
          if ( input == …) nx_state = …; else nx_state = …;
          end
  state2: …
   …
endcase
end

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

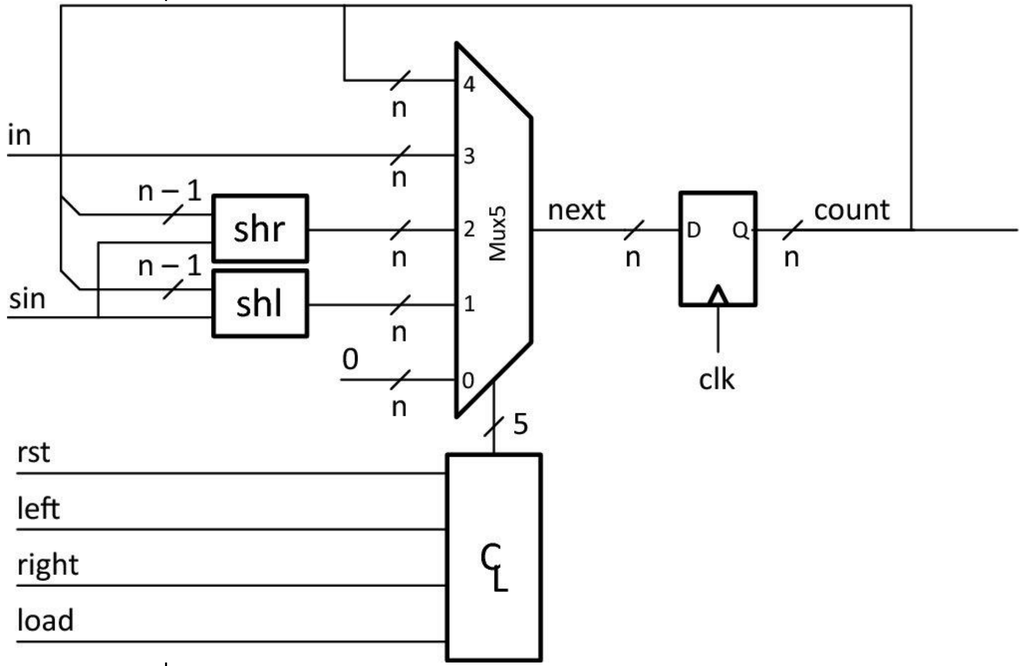


shift register with parallel load

```
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 @(*) 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
```



dual-port RAM

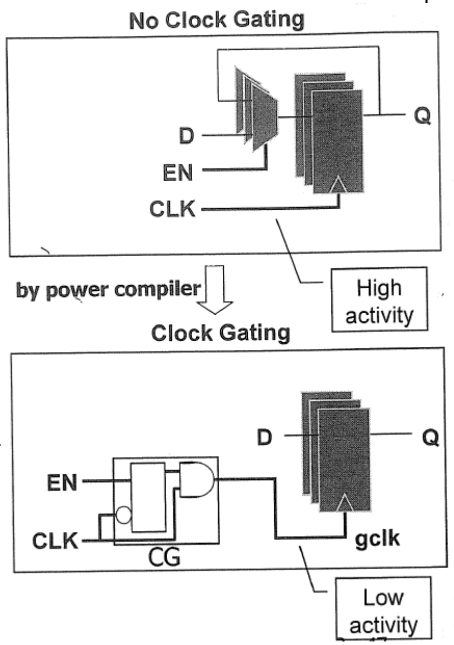
```
module RAM(ra, wa, write, din, dout) ;
  parameter b = 32;
  parameter w = 4;

  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];

  always@(*) begin
    if(write == 1)
      ram[wa] = din;
  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
```

clock gating

```
module dff(q, d, clk, set, rst);
input d, clk, set, rst;
output q;
reg q;
wire clk_en, en;

// clock input is from the output of AND
assign clk_en = clk & en;
always @(posedge clk_en)
if (rst)
q <= 1'b0;
else if (set)
q <= 1'b1;
else
q <= d;
endmodule
```



```
module dff(q, d, clk, set, rst);
input d, clk, set, rst;
output q;
reg q;
wire cen;

// data input from MUX controlled by en
always @(posedge clk)
if (en) begin
  if (rst)
    q <= 1'b0;
  else if (set)
    q <= 1'b1;
  else
    q <= d;
end
endmodule
```

ROM

- one-dimensional array with data in case statement
- two-dimensional array with data in initial statement
- two-dimensional array with data in text f ile

IEEE Std. 1364.1 Verilog Register Transfer Level Synthesis

ROM: 1D array using **case**

- Synthesis attribute *rom_block* model ROM

```
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 attributes *logic_block* or *rom_block* is used
- without specifying attributes, a synthesis tool may opt to implement either as random 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

```
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: 2D array using **always**

- in general, standard cell library vendors 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
```