Chapter 8

(8.1)

These specs will not synthesize to a shift register! See spec #15, page 64 for one that would.

(8.2)

only one extra FF needed
over those in declared regs

if (new) count <= -1;
else
case(Q[1:0])
2'b10 : PP <= P - M;
etc
PP <= PP >> 1;
PP <= PP >> 1;
count <= count >> 1;

(8.3) (a)

state diagram when machine is considered as a straight FSM:
At this point the problem solutions begin to become large, so only selected solutions only will be given.

The following is an example of a FIFO done in the Zorian94 style and using a Xilinx RAM (see page 235). The data input and output is specified in separate bits to facilitate incorporation into the BIST model and testing...

```verilog
module explicit_fifo(clk, flush, insert, remove, data_out)
parameter ws=4, as=3, nw=8; //
input clk, flush, insert, remove;
input [ws-1:0] data_in;
output [ws-1:0] data_out;
reg [ws-1:0] ram_data[nw-1:0] // RAM
reg [ws-1:0] data_out; // output buffer
wire [ws-1:0] data_in;
reg [as-1:0] rp, wp;
parameter S0=0, S1=1, S2=2, S3=3, S4=4, S5=5; // state assignment modelled on fig 8.6
reg [2:0] present_state, next_state;

always @(flush or insert or remove or present_state)
case(present_state)
S0: if (flush) next_state<=S5;
   elseif (insert) next_state<=S1;
   elseif (remove) next_state<=S3;
   else next_state<=S0;
S1: begin ram_data[wp]<=data_in; next_state<=S2; end
S2: begin wp<=wp+1; next_state<=S0; end
S3: begin data_out<=ram_data[rp]; next_state<=S4; end
S4: begin rp<=rp+1; next_state<=S0; end
S5: begin wp<=0; rp<0; next_state<=S0; end
endcase
always @(posedge clk) present_state <= next_state;
endmodule
```

At this point the problem solutions begin to become large, so only selected solutions only will be given.

(8.5)

The following is an example of a FIFO done in the Zorian94 style and using a Xilinx RAM (see page 235). The data input and output is specified in separate bits to facilitate incorporation into the BIST model and testing.
module zfifo(clk, flush, insert, remove, win_3, win_2, win_1, win_0, empty, full, wout_3, wout_2, wout_1, wout_0, wp, rp, WAck, RAck, WI, RI, SLO, RT)
parameter nw=8, as=3, ws=4;
input clk, flush, insert, remove, win_3, win_2, win_1, win_0, WI, RI, SLO, RT;
output empty, full, wout_3, wout_2, wout_1, wout_0, WAck, RAck;
output [nw-1,0] rp, wp;
reg empty, full, we, dummy, WAck, RAck;
reg [nw-1,0] rp, wp;
reg [as-1,0] read, write;
integer i;
    tpm M({1'b0,read},{1'b0,write}, win,we,wout,~clk);
always @(posedge clk);
begin
    case (1'b1)
    flush :
    begin
    full<=0; empty<=1; WAck<=0; RAck<=0; end
    SLO :
    begin
    full<=0; empty<=1; we<=0; rp <= -2; wp <= -2; WAck<=0; RAck<=0; end
    RT   :
    begin
    full<=0; empty<=0; we<=0; rp <= 1; WAck<=0; RAck<=0; end
    remove: begin
    if (insert==1 && ~full && ~empty)
    begin
    rp <= { rp[nw-2:0], rp[nw-1] };
    wp <= { wp[nw-2:0], wp[nw-1] };
    we <=0; WAck<=1; RAck<=1;
    end
    elseif (~empty || RI )
    begin
    if (~RI)
    begin
    rp <= { rp[nw-2:0], rp[nw-1] };
    if (wp == {rp[nw-2:0], rp[nw-1] } ) empty <=1;
    end
    we <= 0; full <= 0; WAck <= 1; RAck <= 1;
    end
    empty <= 0;
    end
    insert :
    begin
    if (~full || WI)
    begin
    if (~WI)
    begin
    wp <= { wp[nw-2:0], wp[nw-1] };
    if (rp == {wp[nw-2:0], wp[nw-1] } ) full <=1;
    end
    we <= 1; empty <= 0; WAck <= 1; RAck <= 0;
    end
    default : dummy <= 0;
    endcase
    for (i=0; i <= nw-1; i=i+1)
    begin
    if ( rp[i] == 1) read <= i;
    if (wp[i] == 1) write <= i;
    end
endendmodule

module tpm(addr-r, addr-w, data_in, we, data_out, data_in, clk);  // Xilinx 2-port RAM (see page 235)
input clk, we;
input [3:0] addr_r, addr_w;
input [7:0] data_in;
output [7:0] data_out;
RAM16X1D u0(WCLK(clk), .SPO(data_in[0]), .WE(we), .DPO(data_out[0]),
    .A0(addr_w[0]), .A1(addr_w[1]), .A2(addr_w[2]), .A3(addr_w[3]),
    .DPRA0(addr_r[0]), .DPRA1(addr_r[1]), .DPRA2(addr_r[2]), .DPRA3(addr_r[3]) );
etc., for u1, u2, u3
(8.7) Fifo using unidirectional shift registers for the data and a bidirectional shift register for the fill indicator

```verilog
module fifo(clk, flush, insert, remove, data_out)
parameter ws=4, nw=8; // wordsize, number of words
input clk, flush, insert, remove;
input [ws-1:0] data_in;
output [ws-1:0] data_out;
wire [ws-1:0] data_out;
reg [nw,0] indicator;
assign full = indicator[nw], empty = indicator[0];
reg [nw-1:0] datas, data3, data2, data1; // can fully parameterize this only when
assign data_out = {datas[0],data3[0],data2[0],data1[0]}; // Verilog gets a generate statement!
always begin
  @(posedge clk);
  if (flush) indicator <= 1; @(posedge clk);
  else begin
    case ({remove,insert})
      2'b00 : ;
      2'b01 : if(~full) // insert only
        begin // insert bits of new word into each data shiftreg at position indicated
          datas <= datas& !indicator[n-1:0] | datas & indicator[n-1:0];
          data3 <= data3& !indicator[n-1:0] | data3 & indicator[n-1:0];
          data2 <= data2& !indicator[n-1:0] | data2 & indicator[n-1:0];
          data1 <= data1& !indicator[n-1:0] | data1 & indicator[n-1:0];
          @(posedge clk); // then shift indicator reg one bit position to the left
          indicator <= {indicator[n-1:0], 0};
        end
      2'b10 : if(~empty) // remove only if not empty
        begin // shift data right
          datas <= {0, datas[n-1:1]};
          data3 <= {0, data3[n-1:1]};
          data1 <= {0, data2[n-1:1]};
          data1 <= {0, data1[n-1:1]};
          @(posedge clk); // then shift indicator reg one bit position to the right
          indicator <= {0, indicator[n-1:1]};
        end
      2'b11 : if (~empty) // remove-insert if not empty
        begin // shift data right
          datas <= {0, datas[n-1:1]};
          data3 <= {0, data3[n-1:1]};
          data1 <= {0, data2[n-1:1]};
          data1 <= {0, data1[n-1:1]};
          @(posedge clk); then insert new hits without shifting indicator
          datas <= datas& !indicator[n-1:0] | datas & indicator[n-1:0];
          data3 <= data3& !indicator[n-1:0] | data3 & indicator[n-1:0];
          data2 <= data2& !indicator[n-1:0] | data2 & indicator[n-1:0];
          data1<= data1& !indicator[n-1:0] | data1 & indicator[n-1:0];
          end
    endcase
  end
endmodule
```
(8.8) keyboard style door lock:

```
module doorlock(clk, keyboard, door_reclose, red, open)
parameter N = 4, simult_time = 32; timeout =
  input clk, door_reclose;
  input [0:9] keybd;
  output red, opendoor;
  reg [0:9] combo [seqcount-1:0] // RAM storing combo settings
  reg red, opendoor; // flags, cleared at outset
  integer i, j, k;
  assign green = ~red;
always begin
  @(posedge clk); opendoor <=0;
  if (red ==0)
    begin : try
      red <=1;
      for (k=1; k <= N; k = k+1)
        begin
          while (keybd ==0) @posedge clk); // wait until new key is pressed
          for (i=0; i <= simult_time; i = i+1) @posedge(clk); // wait for simultaneity/debounce time
          if (keybd != combo[k])
            begin
              for (j=0; j <= timeout; j = j+1) @posedge(clk); // wait for new try time
              red <= 0; disable try; // failure - quit
            end
          while (keybd != 0) @(posedge clk); //wait until all keys released
        end // go back and look for next combo in sequence
    end // of try
  opendoor <=1; // if this point is reached, the total sequence of combinations was correct
  while (door_reclose == 0) @(posedge clk); // wait for door reclose before resetting lock
  opendoor <=0;
end
endmodule
```
In this case, it would not be more efficient to employ the one-hot state encoding. The minimal state encoding requires a 2-bit state vector, while one-hot state encoding requires 4 bits.

Could this machine be specified in implicit style? The question is omitted.
Chapter 9

Again, as the problems in these later chapters become more complex, selected example solutions only are given (9.5)

Content addressable memory function via the hashing algorithm outlined in the problem statement:

Top level module first, followed by components,

```vhdl
'include "hash1.v"
'include "hash2.v"
'include "camram.v"

module cam(phh2, phram, phcam, flush, search, enter, rept, key, we, addr, status);

parameter m=7, n=3, ws=7; // cam capacity, address length, and word size
input phh2, phram, phcam;
input flush, search, enter, we, rept;
input [ws-2:0] key;
output [n-1:0] addr;
output [2:0] status;
reg [n-1:0] addr;
reg [2:0] status;
wire [ws-2:0] dataout;
wire W_succ;
wire [2:0] camstatus;
wire [n-1:0] h1, h2;
integer i;

hash1    # (ws,n)     hash1(key, h1);
hash2    # (n)           hash2(phh2, h1, h2);
camram # (m,n,ws) camram(phram, flush, search, enter, rept, we, h2, key, camstatus, W_succ, dataout);

always @(posedge phcam)
begin
  case (camstatus)
  0:   status<=0;           // search success
  1:   status<=1;           // repeat, search continue
  2:   status<=2;           // write invalid
  3:   status<=3;           // write success
  4:   status<=4;           // found an empty slot, search fail, write enable
  5:   status<=5;           // flush, clean everything
  endcase
  status <= camstatus;
  addr  <= h2;
end
endmodule
```

Other components of the CAM:
module camram (clk, flush, search, enter, rept, we, addr, key_in, camstatus, W_succ, dataout);
parameter m=7, n=3, ws=7;       // cam capacity, address length, word size
input flush, search, enter, we, rept,clk;
input [n-1:0] addr;
input [ws-2:0] key_in;
output [ws-2:0] dataout;
output W_succ;
output [2:0] camstatus;
reg [ws-2:0] dataout;                      // set Write status.
reg [ws-2:0] ram[m-1:1];   // Memory.
reg tabl[m-1:1];                // Using table to indicate occupancy.
reg [2:0] camstatus;
integer i;
always @(posedge clk)
begin
if (flush)
begin for (i=1; i<m; i=i+1)
begin
tabl[i] <= 0;           // set tabl[i]=0 when” flush”
camstatus <= 5;    // flush success, clean everything
end
end
else if (enter&&!we)
begin
if (tabl[addr])
begin
dataout[ws-2] <= {1'b0}; // invalid enter. slot not empty
W_succ <= 0;                // set out=6’b000000, write failed
camstatus <= 2;            // W_succ<=0; write invalid
end
else begin
tabl[addr] <=1;                           // mark this slot as occupied.
ram[addr] <= key_in[ws-2:0];   // enter the key in this slot.
dataout[ws-2] <= {1'b1};        // set out=6’b1xxxxx, enter success
camstatus <= 3;                          // write success, set W_succ<=1, reset we=0 and enter=0;
W_succ <= 1;
end
end
else if (search || rept)
begin
case (tabl[addr])
1’b1: begin
dataout[ws-2] <= ram[addr]; // if search success: camstatus=1; else rept=1; camstatus=0;
camstatus <= (key_in==ram[addr]);
end
1’b0: begin
dataout[ws-2:0] <= {ws-1[1’b0]};    // flush the ram data in this slot. set the ram slot =6’b000000
camstatus <= 4;                          // found an empty slot, set we=1;
end
endcase
end
endmodule
Finally, the test module: first the declarations part, then the execution part

```
//timescale 10 ns / 100 ps
module testcam;
parameter dt=50, dt1=100, dt2=200; // time clocks
parameter m=7, ws=7, n=3; // cam capacity, key size, address length
parameter key_num=20; // total numbers of input keys in this testbench
parameter Flush_cam=2'b01, Search_key=2'b10, Enter_key=2'b11, Stop_test=2'b00;
reg pha; // master phase clock
reg flush, search, enter, rept;
reg phcam, phh2, phram; // phase clock for hash2, cam, and ctrl
reg [ws-2:0] key; // input key size
wire [n-1:0] addr; // return address when search and enter.
wire [2:0] status; // return status from controller: cam.v
reg ovfl;
reg [n-1:0] ct;
reg [ws-2:0] keys [key_num-1:0]; // input keys for testbench
reg [1:0] ops [key_num-1:0]; // operations for testbench
reg S_succ, W_succ; // search success, and overflow status
reg we; // write enable
integer i;
wire addr2, addr1, addr0, status2, status1, status0;
assign addr = {addr2, addr1, addr0};
assign status = {status2, status1, status0};
// reg key5, key4, key3, key2, key1, key0;

cam cam(phh2,phram,phcam,flush,search,enter,rept,we,addr,status);
// cam return two status: addr and status
```
always begin #20 pha=1; #80 pha=0; end
always @(posedge pha) begin #27 phh2=1; #20 phh2=0; end // phh2
always @(posedge pha) begin #65 phram=1; #20 phram=0; end // phram
always @(posedge pha) begin #70 phcam=1; #20 phcam=0; end // phcam
always @(negedge pha)                  //check the return status
    case (status)
        0: begin flush<=0; search<=1; rept<=1; S_succ<=0; W_succ<=0; we<=0; end // repeat, search continue
        1: begin flush<=0; search<=0; rept<=0; S_succ<=1; W_succ<=0; we<=0; end // search success
        2: begin flush<=0; search<=0; rept<=0; S_succ<=0; W_succ<=0; we<=0; end // write invalid
        3: begin flush<=0; search<=0; rept<=0; S_succ<=0; W_succ<=1; we<=0; end // write success
        4: begin flush<=0; search<=0; rept<=0; S_succ<=0; W_succ<=0; we<=1; end // found empty slot, search fail
        5: begin flush<=0; search<=0; rept<=0; S_succ<=0; W_succ<=0; we<=0; end // flush, clean everything
    endcase
initial begin
    $display ("time F S E R   key  addr status S_succ W_succ ovfl count we");
    $monitor ("%4g,%b,%b,%b,%b,%b,%b, %b, %b, %b, %g, %b", 
                        "Stime, flush, search, enter, rept, key, addr, status, S_succ, W_succ, ovfl, ct, we);
    pha=0; flush=0; search=0; enter=0; rept=0; ct=0; ovfl<=0; 
    key = 0; // (key5, key4, key3, key2, key1, key0) = 6'b000000;
    S_succ=0; W_succ=0;   // addr, status, ovfl: returned by module cam 
    // setup the testbench
ops[0]=Flush_cam; keys[0]=3;
ops[1]=Enter_key; keys[1]=4;
ops[8]=Search_key; keys[8]=3;
ops[9]=Enter_key; keys[9]=1;
ops[10]=Enter_key; keys[10]=22;
ops[12]=Stop_test;
for (i=0;i<=key_num;i=i+1)
begin: testben
    while (rept) #dt2; // if rept=1 or writing in process, any new operation prohibited
    #110 case (ops[i])
    Flush_cam : begin flush<=1; search<=0; enter<=0; rept<=0; end
    Search_key: begin flush<=0; search<=1; enter<=0; rept<=0; end
    Enter_key : begin flush<=0; search<=1; enter<=1; rept<=0; end
    Stop_test : #1 $finish;
    endcase
    key=keys[i]; // {key5, key4, key3, key2, key1, key0} = key;
    #90 if (enter)
        begin if (ct==[3'b111]) begin ovfl<=1; #1 $finish; end
            else ct<=ct+1;
    end
#dt $finish;
end
endmodule
Chapter 10

(10.1) omitted

(10.2) omitted

(10.3) The exercise is changed to the following question:

Design hardware that will produce the following sequence, \( F = \sum_{i=1}^{N} i \),

(e.g. If \( N=3 \), \( F=1+2+3=6 \)). Design and implement a module that whenever its input \( N \) changes, it produces \( F \), at latest \( N \) clock cycles later. \( N \) will be any 4-bit number (meaning that \( F \) has to be 7 bits long). \( N \) will not change while a new \( F \) is being calculated. \( F \) must be held after the series has been calculated until \( N \) changes again.

a. Sketch your design. Clearly identify your datapath and control sections. Clearly identify your control strategy, explicitly labeling control lines and status lines. This design does NOT need a reset or any kind of ’start’ signal. Include a timing diagram showing the operation of important signals. \( N \) will not change during the calculation of the series.

b. Write Verilog capturing this design.

```verilog
module F (clock, N, F);
input clock;
input [3:0] N;
output [6:0] F;
reg [6:0] F;
reg [3:0] Nprev;
reg [3:0] count;
reg state, next_state;

// control lines
reg [1:0] Fmux, Cmux;

//status line
wire zero;

parameter waitNp = 1'b0,
waitZ = 1'b1;
module F (clock, N, F);
```
// Datapath
always@(posedge clock)
case(Fmux)
  2'h0 : F <= N;
  2'h1 : F <= F + count;
endcase

always@(posedge clock)
case (Cmux)
  2'h0 : count <= N-1;
  2'h1 : count <= count - 1;
endcase

assign zero = (count == 0);

// Controller
always@(posedge clock)
Nprev <= N;
always@(posedge clock)
state <= next_state;

// Mealy machine is smaller and there are no synthesis issues in such a tiny FSM
always@(state or N or Nprev or zero)
case (state) // synopsys full_case parallel_case
  waitNp :
    if (N == Nprev)
      begin
        Fmux = 2;
        Cmux = 2;
        next_state = waitNp;
      end
    else
      begin
        Fmux = 0;
        Cmux = 0;
        next_state = waitZ;
      end
  waitZ :
    if (!zero)
      begin
        Fmux = 1;
        Cmux = 1;
        next_state = waitZ;
      end
    else
      begin
        Fmux = 2;
        Cmux = 2;
        next_state = waitNp;
      end
endcase
endmodule
Note: typo: the C should be (i=0; i<=31; i++) and the same for j

```verilog
module match(clock, address1, address2, data1, data2, start, found, not_found);

input  clock;
output [4:0] address1, address2;
input [7:0] data1, data2;
input  start;
output  found, not_found;
reg [4:0]  address1;
reg [4:0]  address2;
reg    found, not_found;

// note need for <= and shared logic (found || not_found)
always @(posedge clock)
begin
  if (!start)
    begin
      address1 <= 5'd0;
      address2 <= 5'd0;
    end
  else if (!(found || not_found))
    begin
      address1 <= address1 + 1'b1;
      if ((address1 == 5'd31)) address2 <= address2 + 1'b1;
    end
end

// note priority logic required
always @(posedge clock)
if ((data1 == data2) && start) found <= 1'b1;
else if ((address1 == 5'd31) && (address2 == 5'd31) && start) not_found <= 1'b1;
else
  begin
    found <= 1'b0;
    not_found <= 1'b0;
  end
endmodule
```
Chapter 11

(11.1)
We know that \( t_{\text{clock}} = 20\text{ns}, t_{\text{clock\_high\_max}} = t_{\text{clock\_high\_min}} = 10\text{ns}, t_{\text{skew}} = 1\text{ns}, \)
\( t_{\text{setup}} = 3\text{ns}, t_{\text{Q\_max}} = 5\text{ns}, \sum t_{\text{logic\_max}} = 9\text{ns}, \)
\( t_{\text{hold}} = 0, t_{\text{Q\_min}} = 3\text{ns}, \sum t_{\text{logic\_min}} = 2\text{ns}, \)
then to prevent a race, the following equation should be satisfied:
\[
t_{\text{hold}} + t_{\text{skew}} + t_{\text{clock\_high\_max}} + t_{\text{clock\_Q\_min}} + t_{\text{Q\_min}} \
\leq \sum t_{\text{logic\_min}}
\]
but now, the left side = 0 + 1 + 10 = 11ns, the right side = 3 + 2 = 5ns. We need to insert an extra delay \( t_{\text{extra}} \):
\[
11 \leq 5 + t_{\text{extra}}, \text{we get } t_{\text{extra}} \geq 6\text{ns}
\]
At the same time, we need to prevent setup violation, then the following equation should be satisfied:
\[
t_{\text{clock}} + t_{\text{clock\_high\_min}} + t_{\text{Q\_max}} + \sum t_{\text{logic\_max}} + t_{\text{extra}} + t_{\text{setup}} + t_{\text{skew}}
\]
\[20 + 10 \geq 5 + 9 + t_{\text{extra}} + 3 + 1, \text{we get } t_{\text{extra}} \leq 12\text{ns}\]
Finally, we get that \( 6\text{ns} \leq t_{\text{extra}} \leq 12\text{ns} \)

For an inverter, \( 2\text{ns} \leq t_{\text{delay}} \leq 4\text{ns}, \)
We need three inverters to be inserted in order to prevent a race, and use \( Q' \) instead of \( Q \) to ensure logic consistence..

(11.2)
\[
P = 100 \times 10^6 \times 1 \times 10^{-1} \times 10 \times 10^{-15} = 1 \times 10^{-7} \text{W}
\]

(11.3)

a. == logic:
\[
P = 0.5 \times 32 \times (100 \times 10^{-15}) \times 5^2 \times (100 \times 10^6) = 4 \text{mW}
\]
Flip-flop
\[
P = 0.6 \times (100 \times 10^{-15}) \times 5^2 \times (100 \times 10^6) = 0.15 \text{mW}
\]
TOTAL = 8.15 mW
b.

```verilog
reg [31:0] bus1, bus2;
reg [31:0] snoop;
reg snoop_on;
reg flag;

// reduces power in theory but CHECK synthesized results
always@(posedge clock)
  if (snoopon)
    if ((snoop == bus1) || (snoop == bus2))
      flag = 1'b1;
    else flag = 1'b0;
```

OR

Use (snoopon = 0) to high-Z input from buses.

OR

Create a new enabled clock (least preferred solution).

(11.4)

```verilog
reg En1;
reg [31:0] A, B, DataOut, MultOut, AddOut;
wire [31:0] A, B;

always@(posedge clock)
  if (En1 == 1) DataOut <= MultOut;
  else DataOut <= AddOut;

always@(posedge clock)
  begin
    saveA <= A;
    saveB <= B;
  end

assign InA = En1 ? A : saveA;
assign InB = En1 ? B : saveB;

assign MultOut = InA * InB;
assign AddOut = A + B;

/* dropping the save registers and replacing MultOut inputs with
assign InA = En1 ? A : 32'h0;
assign InB = En1 ? B : 32'h0;
is viable too. */
```
• Sketch the logic:

• min delay = 1 ns, typical delay = 2.5 ns, max delay = 3 ns.

For the following two questions, considering MUX as one gate.

• Yes. max delay = $t_{\text{clock}Q_{\text{max}}} + \sum t_{\text{logic}_{\text{max}}} = 2 + 3 + 3 = 8 \text{ ns}$, but $t_{\text{clock}} - t_{\text{skew}} - t_{\text{setup}_{\text{max}}} = 7 \text{ ns}$, $8\text{ns} > 7\text{ns}$.

• Yes. min delay = $t_{\text{clock}Q_{\text{min}}} + \sum t_{\text{logic}_{\text{min}}} = 0.5 + 1 = 1.5 \text{ ns}$, but $t_{\text{skew}} + t_{\text{hold}_{\text{max}}} = 2 \text{ ns}$, $1.5\text{ns} < 2\text{ns}$.

Chapter 12

(12.1) (12.2) (12.3) (12.4) omitted
Chapter 13

(13.1)
create_clock -period 5 -waveform {0 2.5} clock
set_clock_skew -uncertainty 0.2 clock

(13.2)
Compile runs a set of optimization algorithms that start from scratch.
Compile -incremental assumes that a compile has been run earlier.

(13.3)

(13.4) omitted

Chapter 14
omitted
(1) Devising extra test circuitry to test the test circuitry is like chasing Zeno’s paradox. Here we add a small amount of (untested) extra circuitry just to test the major test modules, the dbg and the fsr. A small rom checks that the dbg is going through its successive vectors correctly, and the existing comparators check that the fsr initiates, and shifts correctly one place to the left and right. The modified organization is shown below:

**On chip BIST for RAM, augmented for self test**

In the modified specifications, the extra sections added are highlighted with asterisks. First the declaration part:

and now the execution part: In the modified scheme the reset input is used to place the RAM into the normal function state, and to re-initialize the test circuitry. The two states of the test input are used to first test the bist circuitry, and then to use it to test the RAM.
module rambist(clk, reset, test,
    adbg_0,adbg_1,adbg_2,adbg_3,adbg_4,adbg_5,adbg_6,adbg_7,
    addr_0,addr_1,addr_2, bad, complete, D, L, P);
* input clk;
* parameter as=3, ts= 2; // RAM address size, ROM address size(for dbg testvectors)
* parameter nw=1<<as, nt=1<<ts; // RAM capacity, ROM capacity
* parameter ws = 1<<(nt-1);
*input test;     // 0 => test bist circuitry,1 => use bist circuitry to test RAM
*input reset;    // 1 = go to normal RAM mode and reset between tests
wire [ws-1:0] Win;
wire [as-1:0] ext_addr;
wire [ws-1:0] Wout;
output addr_2,addr_1,addr_0; // these outputs for validation only
output adbg_7, adbg_6, adbg_5, adbg_4, adbg_3, adbg_2, adbg_1, adbg_0;
output D, L, P, complete, bad;
reg bad,     // 1 = bad
    complete:// 1 = test complete
    D,  // direction 0= right (forward), 1= left (backward),
    P,  // polarity, 1= invert
    L;   // loading
reg WE, DBG_init, DBG_next, F_s, F_r, oddeven;
reg [ts-1:0] romcount;
wire [ws-1:0] ramDin, DBG_dout, DBG_tout; // interconnections
wire [as-1:0] addr, f_addr;
wire [as-1:0] first = -1;
wire [as-1:0] last = -2;
wire tst_eq, ext_WE;
wire [ws-1:0] adbg = DBG_dout;
assign addr_2=addr[2],addr_1=addr[1],addr_0=addr[0];
assign adbg_7=adbg[7], adbg_6=adbg[6], adbg_5=adbg[5], adbg_4=adbg[4];
assign adbg_3=adbg[3], adbg_2=adbg[2], adbg_1=adbg[1], adbg_0=adbg[0];
// instantiate submodules to size and identify
ram #(nw,as,ws) RAM(~clk&&~oddeven, rWE, addr, ramDin, Wout);
dbg #(ws) DBG(~clk,DBG_init,DBG_next,P,DBG_dout,DBG_tout);
fsrb #(as,2) PTR(~clk, D, F_s, F_r, f_addr);
comparator #(ws) CMP(DBG_dout, Wout, tst_eq);
mux #(as) MUXA(test, ext_addr, f_addr, addr);
mux #(ws) MUXD(test, Win, DBG_tout, ramDin);
mux #(1) MUXWE(test, ext_WE, WE, rWE);
comparator #(as) CMP0(f_addr, first, Fi);
comparator #(as) CMP1(f_addr, last, La);
rom #(as,ws) ROM(~clk&&&oddeven, rWE, addr, testvector);

specification of rambist-1: declarations & instantiations as augmented for BIST selftest
always @ (negedge clk)
if (reset) oddeven <= 0; else oddeven <= ~oddeven;

always
begin @ (posedge clk)
if (reset) // to functional (non-test) mode
begin
    DBG_next<=0; DBG_init<=1; romcount<=0; // get ready to start test
    L<=1; D<=0; P<=0; complete<=0; F_r<=1; F_s<=0; WE<=0; bad<=0;
end
* elseif (~test) // test bist circuitry first
* if (oddeven) // start only on odd clock
* begin
*     DBG_init<=0; F_r<=0; F_s<=1; // turn off resets and enable shifting
*     if (~tst_eq) bad<=1; // dbg did not initialize/advance to next vector
*     if (t_addr!=first) bad <=1; // fsr did not initialize/shift right
*     D<=1; @(posedge clk); // now reverse shift direction & advance to even clock
*     if (t_addr!=last) bad <=1; // fsr did not shift left
*     romcount= romcount+1; DBG_next<=1; D<=0;
* end
* else // if (test) // test RAM using bist circuitry
if (oddeven) // start only on odd clock
begin
    if (~tst_eq & WE & ~L & ~Fi & ~La)  bad<=1;
    casez ({L, D, P, Fi, La, WE})
  6'b100011: begin
        WE<=1; F_s<=1; end // end of load march0
  6'b000011: begin
        WE<=0; F_r<=0; D<=1; F_s<=0; end // end of load march1
  6'b000011: begin
        WE<=0; P<=1; L<=0; F_s<=0; end // end of load march2
  6'b000011: begin
        WE<=0; P<=1; D<=1; F_s<=0; end // end of load march3
  6'b000011: begin
        WE<=0; D<=0; F_s<=1; end // end of load march4
if (DBG_tout[ws-1])
    begin $display ("error"); end
else begin L<=1; DBG_next<=1; end // go to next DB
endcase // now advance to even clock & next dbg vector, and loop
end // of test block
end // endmodule

specification of rambist - part 2, as augmented for BIST selftest