

Digital VLSI System Design
Dr. S. Ramachandran
Department of Electrical Engineering
Indian Institute of Technology, Madras

Lecture – 39

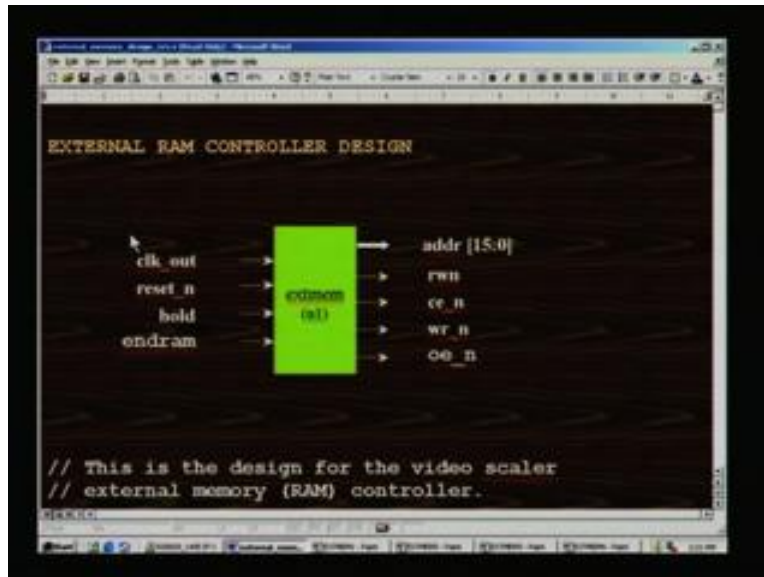
Design of External RAM

(Refer Slide Time: 01:37)



We are looking into the design of RAM. We have seen how to design a dual RAM which was on chip. That is, it is internal to the chip we are designing, be it FPGA or ASIC.

(Refer Slide Time: 02:24)



We will now consider how to interface with external RAMs? This is important to locate the RAM externally in some of the applications. For example, for video scaling you require huge memory storage. For example for the picture size of 1600 by 1200 pixels size color motion picture, it would require per frame about 2MB for one color; for three colors, three times of that, so 6MB that will be the size of the RAM that we will be requiring, if you have to store just 1 frame and if you require two frames storage, double that capacity will be required. Needless to say, we need this for such applications RAM which are external to the FPGA or even ASIC. What we have here is a design for such a controller which will connect to the external RAM. Looking at this figure here, we need this controls in order to interface with external RAM on to the FPGA, we will be deciding. This design will be housed in the FPGA or ASIC which we design. Clock out is the system clock for the memory being a synchronous system. There will be a reset as usual and there can be a hold, in order the hold the process. In addition to this, we need enable for the external RAM. What is shown here is only controller for one RAM. In cases like video scaling or data compression we need dual redundant RAM. In which case we need to select this given enable signal as well for that. In addition to this we need two sets like this for two RAMs. These signals are... ce stands for the chip enable; (Refer Slide Time: 04:38) if you have two RAMs, naturally ce_1 , ce_2 you can have. Similarly, this is the write signal for the external RAM; this is the read signal for the external RAM; y is output enable. Normally, the nomenclature used in most of the memory vendors is precisely this. As usual we have one read.

There is a change here we will make read low, I mean read active high and write active low. So, rwn is the one which will select between two RAMs in case we have. In addition to that, it also solves the purpose of telling us whether we are writing into the RAM or reading, we can use in the test bench appropriately here. There is also an address bus for 16 bits. That means it can address an external RAM of 64 kilobytes. For higher memory requirements you need to jack up this here. This is the design for the video scalar external memory, RAM controller, as I mentioned before. This is the maximum number of address location that you will encounter in a 64K, this in decimal here.

(Refer Slide Time: 06:07)

A screenshot of a code editor window showing Verilog code. The code defines a constant 'max_drpxaddr' as 65535 and declares a module named 'extmem'. The module has several input and output signals: 'clk_out', 'reset_n', 'hold', 'addr', 'endram', 'rwn', 'ce_n', 'wr_n', and 'oe_n'. The code is as follows:

```
define max_drpxaddr 65535

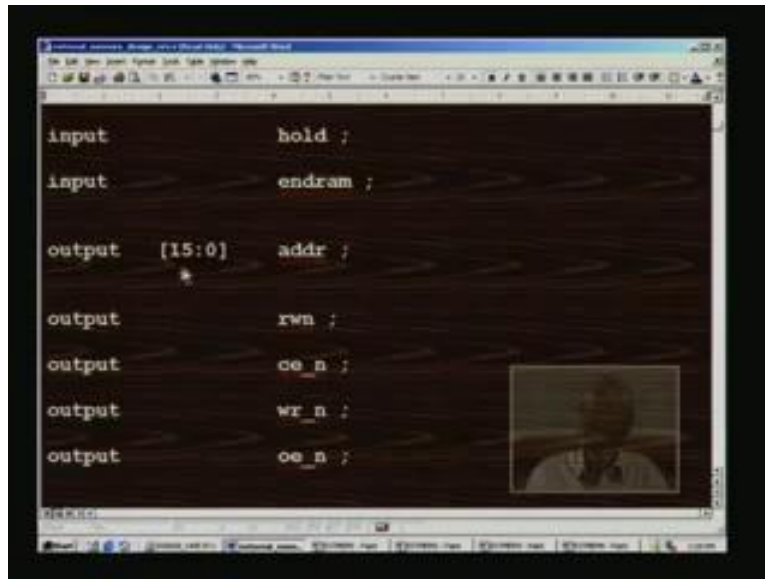
module extmem(

    clk_out,
    reset_n,
    hold,
    addr,
    endram,
    rwn,
    ce_n,
    wr_n,
    oe_n

);
```

We first declare the module for the external memory. This is the control design we are looking into. All I/Os that we have seen on the diagram are listed here, followed by the declaration of I/Os.

(Refer Slide Time: 06:17)



```
input          hold ;
input          endram ;

output [15:0]  addr ;
output        rwn ;
output        ce_n ;
output        wr_n ;
output        oe_n ;
```

This 15 through 0 is for address and all other signals are here. In addition to that we will be requiring, we have assign statements using some of these outputs. Therefore they need to be declared as either wire or reg. Here it is as a register, because its falls as always block with positive edge of the clock.

(Refer Slide Time: 06:46)



```
reg           oe_n ;

reg [15:0]    drvaddr ;
reg [15:0]    drraddr ;

wire [15:0]   drvaddr_next ;
wire [15:0]   drraddr_next ;

wire         envaddr ;
wire         enwr_cnt ;
```

In addition to this we need read address and write address separately for the video scalar. Therefore it has been bifurcated in this fashion. Once again the width is same as the address what we have considered. This is next value; whereas, these are all the registered values.

(Refer Slide Time: 07:10)



```
wire [15:0] drvaddr_next ;
wire [15:0] drraddr_next ;

wire enwaddr ;
wire enwr_cnt ;
wire enraddr ;
wire wr_cnt_next ;
wire res_addr ;
wire res_waddr ;
```

Similarly, there will be one for address write; there will be one enable signal internal use. Since we are using external RAM, we will not be in a position to achieve that high speed that we can, with the one chip RAM. For example, you may have to scale it down by a factor of 2. For instance, if you are operating the system clock at 100 megahertz, we can access external memory only at 50 megahertz rate. In order to do this one, we have a counter here the counters are separate for read as well as write. In order to enable that counter we need another signal and they are all assign statements. That is what and in addition to this for write address we have a read address. Similarly, write count, next also is required in the course of before registering.

(Refer Slide Time: 08:02)

A screenshot of a video lecture showing a code editor window with Verilog code. The code declares several wires and registers. A small inset video of a speaker is visible in the bottom right corner of the code editor.

```
wire    enwr_cnt ;
wire    enraddr ;
wire    wr_cnt_next ;
wire    res_addr ;
wire    res_waddr ;

reg     wr_cnt ;

wire    enrd_cnt ;
wire    rd_cnt_next ;
```

We also have some resetting the address and resetting write address similarly for other signals.

(Refer Slide Time: 08:12)

A screenshot of a video lecture showing a code editor window with Verilog code. The code declares wires and registers, and includes an assign statement for res_addr. A small inset video of a speaker is visible in the bottom right corner of the code editor.

```
wire    res_waddr ;

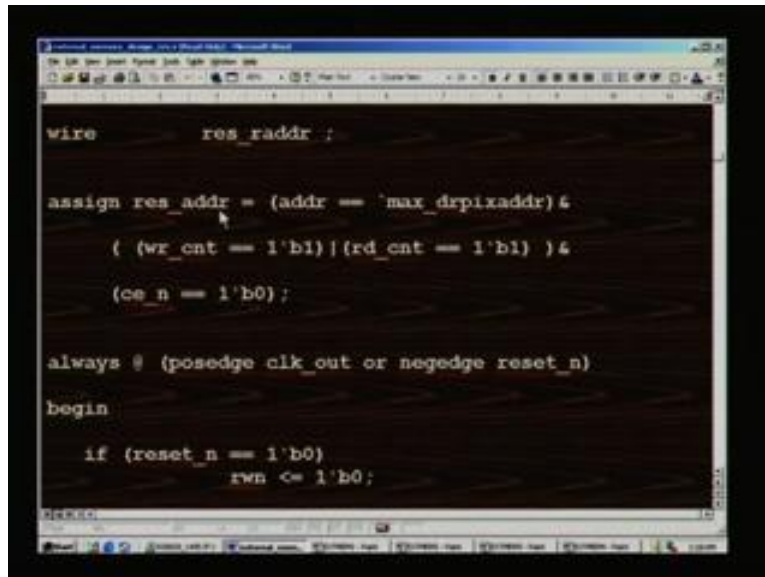
reg     wr_cnt ;

wire    enrd_cnt ;
wire    rd_cnt_next ;
reg     rd_cnt ;
wire    res_raddr ;

assign res_addr = (addr == `max_drpixaddr) &
```

Such as write counter is registered; enabled read counter; its next value; read count itself here. This is enabled. This is the actual counter (Refer Slide Time: 08:25). It is only a single bit counter, it just counts zero and one; it is nearly a toggling flip flop. We have further read address reset as well.

(Refer Slide Time: 08:45)



```
wire      res_raddr ;

assign res_addr = (addr == `max_drpixaddr) &
    ( (wr_cnt == 1'b1) | (rd_cnt == 1'b1) ) &
    (ce_n == 1'b0);

always @ (posedge clk_out or negedge reset_n)
begin
    if (reset_n == 1'b0)
        rwn <= 1'b0;
```

We will have a look into the actual Boolean expressions that we need to evaluate for achieving this. First is the reset address, whenever you want to reset the actual address, we have to look for is whether we have already processed all the addresses write up to the maximum. This is the 65535 in this case and when it matches with the current running address. Then if the write counter or the read counter is 1. As I said before the write counter is to scale down the speed of the access memory access by 2. That is the reason why we have one bit counter for separate counter for write as well as read. Whether this counter is 1 or this counter is 1. It will go from 0 to 1. It starts with 0, so we take every alternate clock only we need to process. Together with this we also need a chip enable and when it is enabled only then we reset the address. The address resetting means, when it touch the running address will be from 0 1 2 etc., 65535. When it touches 65535, then this signal is created.

(Refer Slide Time: 10:01)

```
always @(posedge clk_out or negedge reset_n)
begin
    if (reset_n == 1'b0)
        rwn <= 1'b0;

    else if (hold == 1'b1)
        rwn <= rwn ;

    else if (res_addr == 1'b1)
        rwn <= ~rwn ;

    else
        rwn <= rwn ;
end
```

We have one register here, for creating this rwn, which we have seen earlier and this is meant for whether I mean selection of read or write. For example, if it is 1 it will be a read; otherwise it will be write. As usual we have taken for reset hold condition and what we should do? This is initialised to 0 here at the power on reset. When reset address is encountered what we need to do is, just change this rwn and invert it. Because we can access the next external RAM in the case that we have dual RAM.

(Refer Slide Time: 10:50)

```
        rwn <= rwn ;
end

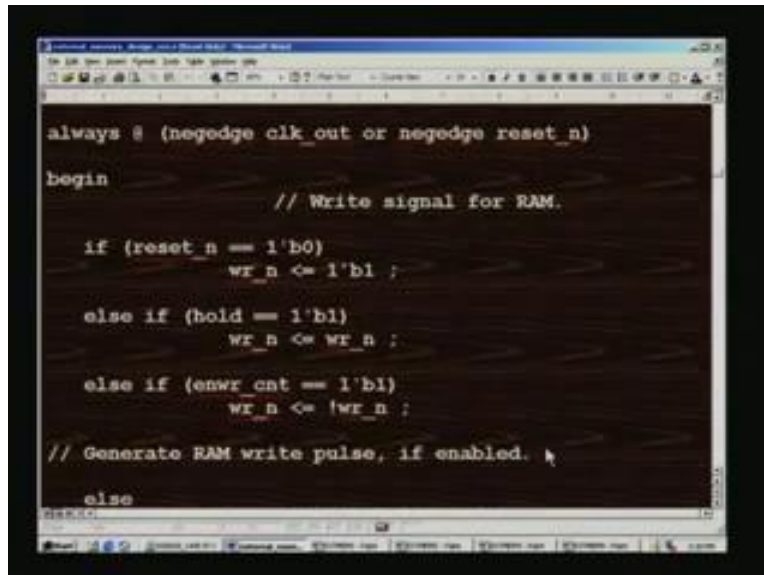
assign gnwr_cnt = (endram == 1'b1) &
                  (rwn == 1'b0) & (ce_n == 1'b0);

always @(negedge clk_out or negedge reset_n)
begin
    // Write signal for RAM.

    if (reset_n == 1'b0)
        wr_n <= 1'b1 ;
end
```


We need an enable for write counter and that is derived from the enable RAM being high and rwn being low that corresponds to write here. That is why we have a write counter here, also ce must be activated, that is ship enable must be activated.

(Refer Slide Time: 11:11)



```
always @ (negedge clk_out or negedge reset_n)
begin
    // Write signal for RAM.

    if (reset_n == 1'b0)
        wr_n <= 1'b1 ;

    else if (hold == 1'b1)
        wr_n <= wr_n ;

    else if (enwr_cnt == 1'b1)
        wr_n <= !wr_n ;

    // Generate RAM write pulse, if enabled.
    else
        wr_n <= 1'b0 ;
end
```

In this always block, note that, we are creating at negative edge; all along we have seen at positive edge clock. We also use the negative edge clock here, in order to create address at the positive edge of the clock and following negative edge this write pulse is issued. When a write pulse is issued the address is always stable that is the reason why we have design in this fashion. That is the reason why we need a negative edge of the clock here. This creates the write signal for RAM. Once again these are all the usual reset and hold and so long as it is enable as we have seen before. What we need to do is, invert the write. If it is in... the write pulse is not there that is, active high it is not active high it is a high position. We need to clock it I mean make it low. That is possible by inverting here. If it is already low with the arrival of the next clock pulse at the negative edge this will go high again. Thereby you create at every two clock cycles one read write pulse which is active low.

(Refer Slide Time: 12:29)

```
else if (wr_cnt == 1'd1)
    wr_n <= !wr_n ;

// Generate RAM write pulse, if enabled.

else
    wr_n <= 1'b1 ;

// Otherwise, disable write pulse.

end

assign wr_cnt_next = wr_cnt + 1 ;

always @ (posedge clk_out or negedge reset_n)
```

You just take it to the safe state of 1, being active low here.

(Refer Slide Time: 12:36)

```
// Otherwise, disable write pulse.

end

assign wr_cnt_next = wr_cnt + 1 ;

always @ (posedge clk_out or negedge reset_n)
begin
    if (reset_n == 1'b0)
        wr_cnt <= 1'b0;

// Counter to slow down RAM write by a factor // of
two
```

Then what we need to do is, increment the counter and this is advance increment. In this always block at positive edge again here. What we do is we reset the counter to start with.....

(Refer Slide Time: 12:53)

```
// Counter to slow down RAM write by a factor // of
two.

else if (hold == 1'b1)
    wr_cnt <= wr_cnt ;

else if (wr_cnt == 1'b1)
    wr_cnt <= 1'b0;

else if (enwr_cnt == 1'b1)
    wr_cnt <= wr_cnt_next;

else
    wr_cnt <= wr_cnt ;

end
```

When write count is 1. That is counter is running between 0 and 1. Whenever it is 1, that is, the identification of the second clock, only then write count will be force to 0. That is why it is going from 0 to 1. So long as that enable is activated it will keep counting. That means, to say it counts from 0 and 1. Similarly we will have a read counter it is exact counterpart. I do not have to explain it length, that one.

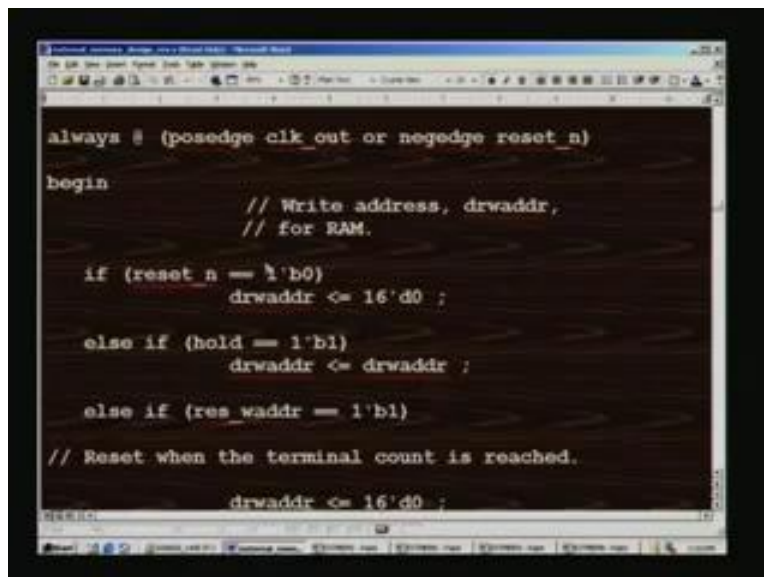
(Refer Slide Time: 13:30)

```
assign drwaddr_next = drwaddr + 1 ;
assign enwaddr = (endram == 1'b1) &
                (rwn == 1'b0) & (wr_cnt == 1'b1) &
                (ce_n == 1'b0);
assign res_waddr = (drwaddr == max_drpixaddr) &
                  (wr_cnt == 1'b1) & (ce_n == 1'b0);

always @ (posedge clk_out or negedge reset_n)
begin
```

Before that, we have write address generated here; this is advance increment here and we increment only with this enable signal here. That is enabled only if that enable RAM is activated and read write is in write mode, so wn here. Also write count is 1 that is the second clock only we process. Further you need chip enable active here and there is also a reset write address and which again is depend upon the maximum value of the address. That is encountered for write address. Once again that write counter 1 and c is also taken into account here.

(Refer Slide Time: 14:21)



```
always @ (posedge clk_out or negedge reset_n)
begin
    // Write address, drwaddr,
    // for RAM.

    if (reset_n == 1'b0)
        drwaddr <= 16'd0 ;

    else if (hold == 1'b1)
        drwaddr <= drwaddr ;

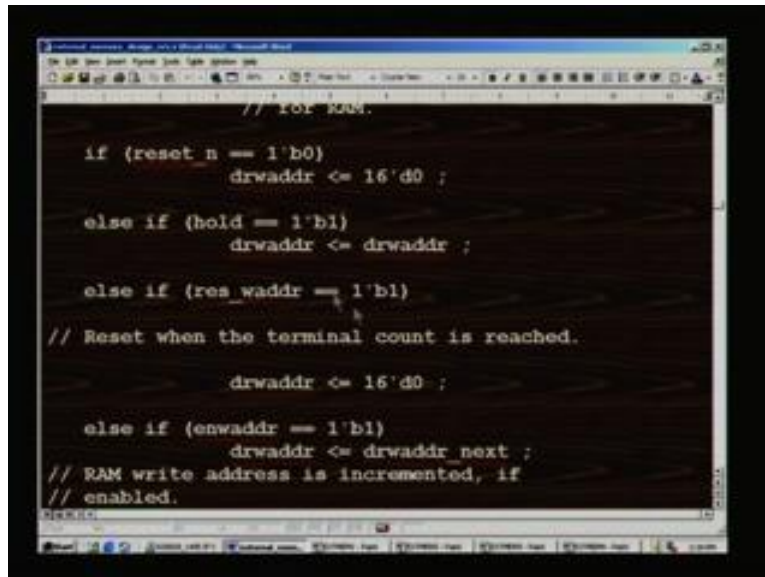
    else if (res_waddr == 1'b1)

// Reset when the terminal count is reached.

        drwaddr <= 16'd0 ;
```

In the next block, we process the actual that is only an assign statement earlier. What we do is? We really process only here, based upon the reset write address and we have seen there.

(Refer Slide Time: 14:37)



```
// for RAM.
if (reset_n == 1'b0)
    drwaddr <= 16'd0 ;

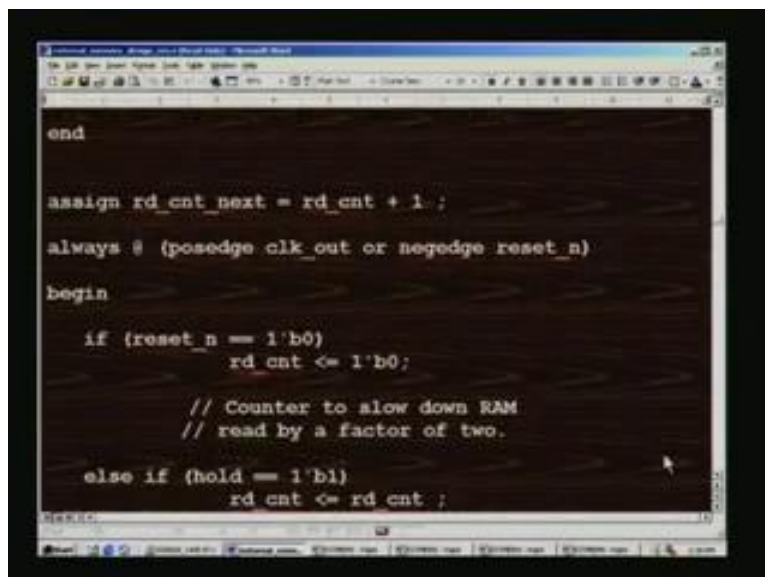
else if (hold == 1'b1)
    drwaddr <= drwaddr ;

else if (res_waddr == 1'b1)
// Reset when the terminal count is reached.
    drwaddr <= 16'd0 ;

else if (enwaddr == 1'b1)
    drwaddr <= drwaddr next ;
// RAM write address is incremented, if
// enabled.
```

We reset the address, because it has touch 65535. We need to reset that one. It will be resettled at this point of time. As long as the enable is on, which we have seen the condition earlier. Write address is incremented this also we have seen in assign statement earlier.

(Refer Slide Time: 15:00)



```
end

assign rd_cnt_next = rd_cnt + 1 ;

always @ (posedge clk_out or negedge reset_n)
begin
    if (reset_n == 1'b0)
        rd_cnt <= 1'b0 ;

        // Counter to slow down RAM
        // read by a factor of two.

    else if (hold == 1'b1)
        rd_cnt <= rd_cnt ;
```

Similarly, we handle the read counter as we have done before for the write counter. It is precisely the same except that write is replaced by read and this is advance increment.

(Refer Slide Time: 15:16)

```
begin
  if (reset_n == 1'b0)
    rd_cnt <= 1'b0;

    // Counter to slow down RAM
    // read by a factor of two.

  else if (hold == 1'b1)
    rd_cnt <= rd_cnt ;

  else if (rd_cnt == 1'b1)
    rd_cnt <= 1'b0;

  else if (enrd_cnt == 1'b1)
    rd_cnt <= rd_cnt_next;

  else
    rd_cnt <= rd_cnt ;
end
```

Actual read counter is processed here. As long as read counter is 1 only then it is made as 0. That is, 0 is made 1; 1 is made 0 and so on, keeps on toggling by these two expressions here. It is exactly the same as write counter.

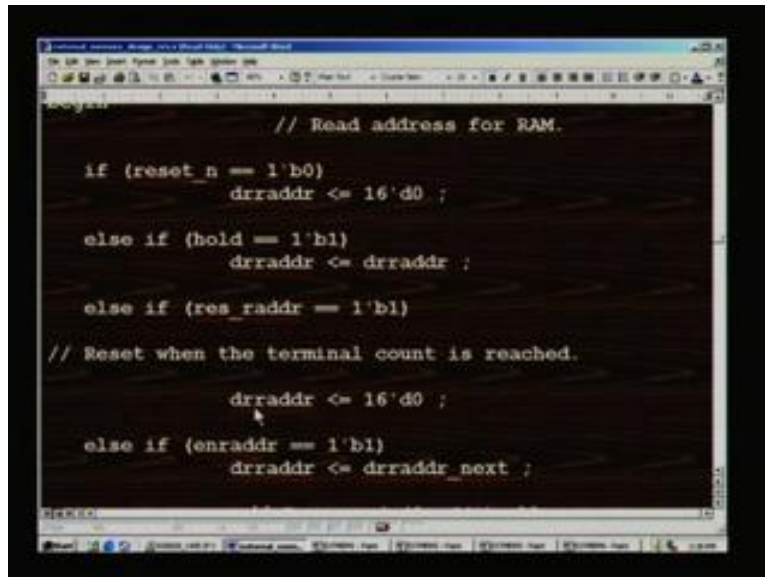
(Refer Slide Time: 15:32)

```
else
  rd_cnt <= rd_cnt ;
end

assign drraddr_next = drraddr + 1;
assign enraddr = (endram == 1'b1) &
  (rwn == 1'b1) & (rd_cnt == 1'b1) &
  (ce_n == 1'b0);
assign res_raddr = (drwaddr == `max_drpixaddr) &
  (rd_cnt == 1'b1) & (ce_n == 1'b0);
```


Once again assign statements are there for read address and it is exactly the same as write address, we have seen before. Exactly matching condition except for the fact that read is 1. That is in read mode. Earlier it was 0 for the write address.

(Refer Slide Time: 16:10)



```
// Read address for RAM.
if (reset_n == 1'b0)
    drraddr <= 16'd0 ;
else if (hold == 1'b1)
    drraddr <= drraddr ;
else if (res_raddr == 1'b1)
// Reset when the terminal count is reached.
    drraddr <= 16'd0 ;
else if (enraddr == 1'b1)
    drraddr <= drraddr_next ;
```

In this block, we actually take that assign statements, intermediate results and then process accordingly. For example, when the terminal count is reached it is reset here and otherwise it is incremented here (Refer Slide Time: 16:14), advance increment was shown earlier. If none of these conditions are satisfied we simply write it back to the same thing that will not disturb the contents.

(Refer Slide Time: 16:29)

```
end

assign enrd_cnt = (endram == 1'b1) &
                 (rwn == 1'b1) & (ce_n == 1'b0);

always @ (negedge clk_out or negedge reset_n)
begin
    // Read signal for RAM.
    if (reset_n == 1'b0)
        ce_n <= 1'b1 ;

    else if (hold == 1'b1)
        ce_n <= ce_n ;
end
```

We have an enable read counter and that is based on enable RAM being high and read write is in read mode and chip is enabled. For this condition, what it does is, read signal for RAM.

(Refer Slide Time: 16:45)

```
always @ (negedge clk_out or negedge reset_n)
begin
    // Read signal for RAM.
    if (reset_n == 1'b0)
        ce_n <= 1'b1 ;

    else if (hold == 1'b1)
        ce_n <= ce_n ;

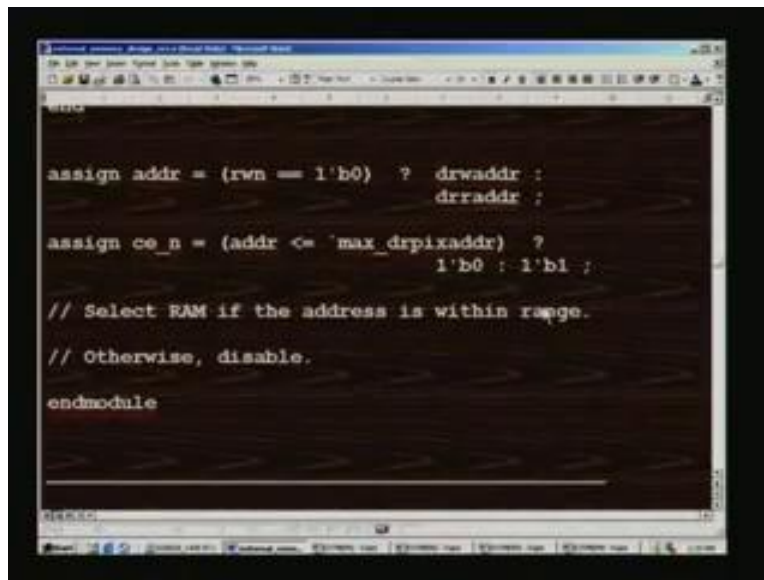
    else if (enrd_cnt == 1'b1)
        ce_n <= ~ce_n ;

    // RAM read signal.
    else
        ce_n <= 1'b1 ;
end
```

That is, this is the read signal we generate for the external RAM access. If it is reset, this is active low. Therefore it is taken to a safe state here and for hold it is preserved; whereas, for enable read count, we invert that y, this is on similar lines as the write. This is precisely similar meaning

here. Otherwise we take read to the inactive state. This means, we can have a true random access in the sense that we can change address and change from read to write and vice versa.

(Refer Slide Time: 17:30)

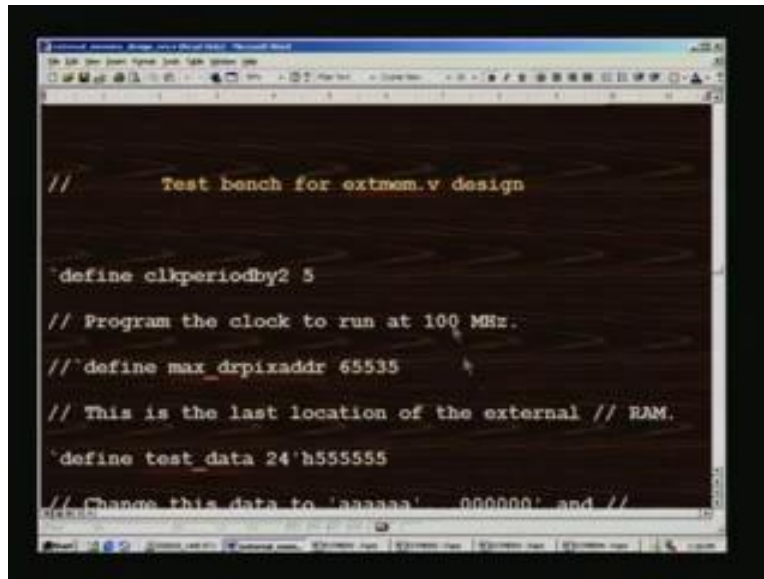


```
assign addr = (rwn == 1'b0) ? drwaddr :  
                drraddr ;  
  
assign ce_n = (addr <= `max_drpixaddr) ?  
                1'b0 : 1'b1 ;  
  
// Select RAM if the address is within range.  
// Otherwise, disable.  
  
endmodule
```

At this step here, what we need is single address to go to the actual RAM and that address may be either read address or write address and that depends upon the read wn control. If it is 1, it would mean read that is here and if it is 0 it would mean write here. When it is 0 it is write therefore this condition being satisfied it will return 1. For 1, this is the condition. This is the MUX that you have already familiar with. So address gets either write address or read address depending upon rwn.

We have also had a ce generated here. Although this rather a dummy statement here in this example, because we use only one single RAM. In dual case RAM we can give a meaningful address and then select 1 of the 2 RAMs and this also a MUX statement. That is the end of the module here and this completes the design for the external memory.

(Refer Slide Time: 18:35)



```
//      Test bench for extmem.v design

`define clkperiodby2 5

// Program the clock to run at 100 MHz.

//`define max_drpxaddr 65535

// This is the last location of the external // RAM.

`define test_data 24'h555555

// Change this data to 'aaaaaa' '000000' and //
```

We will look into the test bench for the external memory. As usual we want 100 megahertz clock. So, we initiate this value to 5 and maximum address we want to 65535 and this is the last location of the external RAM. What we want to do is? First, we want to write a known data pattern. For example, all 5 means this RAM is 24 bit in width. This is because we need it for video scaling and we need to process three colors for the motion picture. That would demand three bytes that means two for red and another two for green another two for blue. If you want to test for some other data pattern you are free to do so, by nearly changing this. For example, you can change it to all a's or all 0s or all ff's. I suggest that you take in turn and check this.

(Refer Slide Time: 19:41)

```

// Change this data to 'aaaaaa', ,000000' and //
'EEEEEE' in turn and run the test again.

`include "extmem.v"

module extmem_test (    addr,
                        ce_n,
                        wr_n,
                        oe_n
                      );

output [15:0] addr ;

output          ce_n ;

```

Then we need to include the design which is external memory. We had to call the module; this present module is the test module, that is, the external memory underscore test. What we need is address, then chip enable write and read pulse. Address is 15 bit in width and all other bits are single bits all other signals are single bits.

(Refer Slide Time: 20:08)

```

output          ce_n ;

output          wr_n ;

output          oe_n ;

reg            clk_out ;

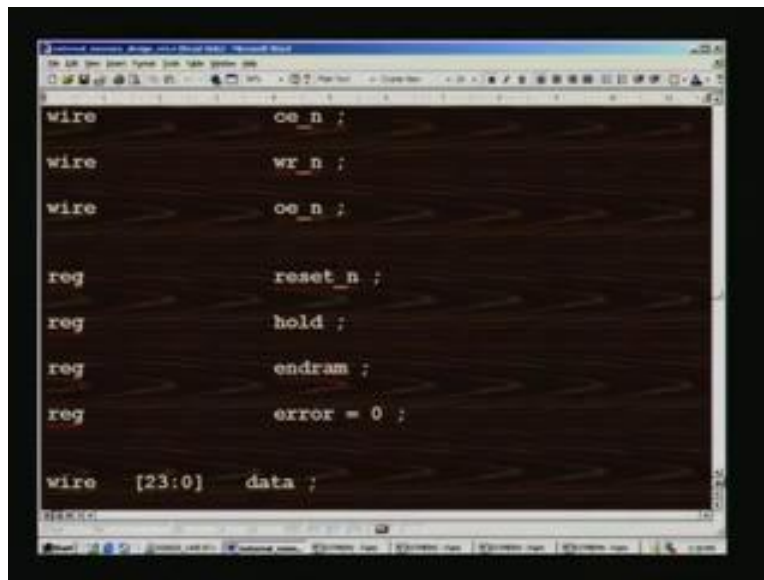
wire [15:0]    addr ;

wire          ce_n ;

```

We need to declare clock out as reg, because this input here. All the outputs are declared as wire here, being a test bench.

(Refer Slide Time: 20:19)



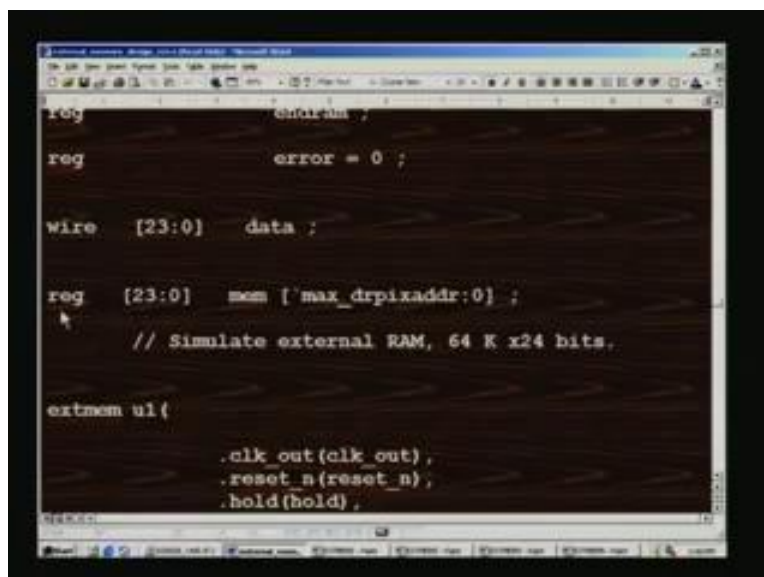
```
wire      oe_n ;
wire      wr_n ;
wire      oe_n ;

reg       reset_n ;
reg       hold ;
reg       endram ;
reg       error = 0 ;

wire [23:0] data ;
```

Inputs must be reg that is what we have here. In addition to this we also have 1 data this is the test data that we have input in the test bench and in this case it was all 5s.

(Refer Slide Time: 20:37)



```
reg       endram ;
reg       error = 0 ;

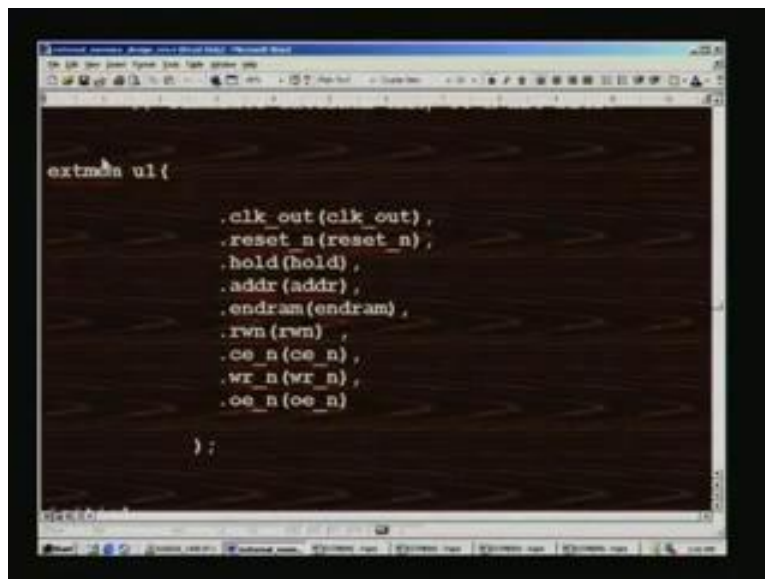
wire [23:0] data ;

reg [23:0] mem ['max_drpxaddr:0] ;
// Simulate external RAM, 64 K x24 bits.

extmem u1(
    .clk out(clk out),
    .reset_n(reset_n),
    .hold(hold),
```

This is the special instruction that you are already familiar. What we are doing in this test bench is? We are having an external RAM simulated write in this test bench. We do not have an external RAM. So what we do, we create an external RAM write within the test bench by using this reg mem, which is a special instruction for invoking arrays, using flip-flops. This is the maximum address that you can have in decimal say for example in this case it is 65535. The width of this memory is 24 bits as we have already explained there. That is what is mentioned here. Simulate external RAM 64K into 24 bits. Because we need three colors; we need three bytes or 24 bits.

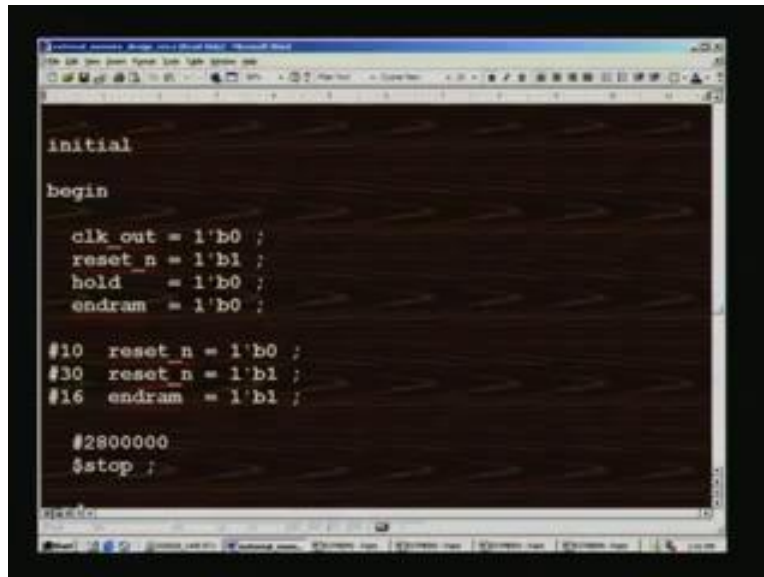
(Refer Slide Time: 21:30)



```
external_ram u1 (  
    .clk_out (clk_out),  
    .reset_n (reset_n),  
    .hold (hold),  
    .addr (addr),  
    .endram (endram),  
    .rwn (rwn),  
    .ce_n (ce_n),  
    .wr_n (wr_n),  
    .oe_n (oe_n)  
);
```

We instantiate the external memory design with all its I/Os here and we use ports by name.

(Refer Slide Time: 21:41)



```
initial
begin
    clk_out = 1'b0 ;
    reset_n = 1'b1 ;
    hold    = 1'b0 ;
    endram  = 1'b0 ;

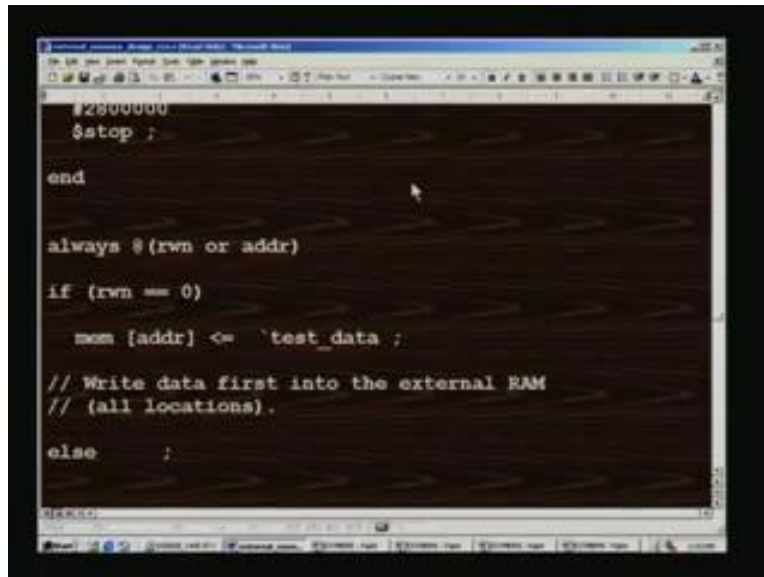
    #10 reset_n = 1'b0 ;
    #30 reset_n = 1'b1 ;
    #16 endram  = 1'b1 ;

    #2800000
    $stop ;
end
```

Once again we start the initializing here of clock out, reset hold, enable RAM to the respective safe states. For example, reset is inactive state here. We apply the reset pulse only at this point of time at 10 nanoseconds. Withdraw the reset at 30 nanoseconds. Just for 20 nanoseconds we apply a reset pulse here. So that the controller resets whatever internal registers that are being used here. The actual design has automatic change of address. In the test bench you do not have to, there is no need for you to change the address. It is all automatic, only covered in the actual design which we have seen before.

What all we have to do is? Just terminate the whole process after a delay. Note that, a huge delay is so many clock pulses are required for the delay because we are treating 64000 locations and each we have seen takes two clock cycles for either a read or write. We have **read as well as write as well as read** to be performed that means 2 plus 2 into 2 and 64000. If you multiply you will get this figure with 1 cyber less the extra cyber is arising because each clock cycle is 20 nanoseconds. You multiply that result by 20. That is 2 into 2 into 20 into 64K must come close to this. That is the reason why we have such a big value here. After this value the test bench will automatically stop the testing. Although we have written elaborate test bench like this in order to write particular data pattern say namely all 5s for all the 64000 at one time.

(Refer Slide Time: 23:43)



```
#2500000
$stop ;

end

always @(rwn or addr)

if (rwn == 0)

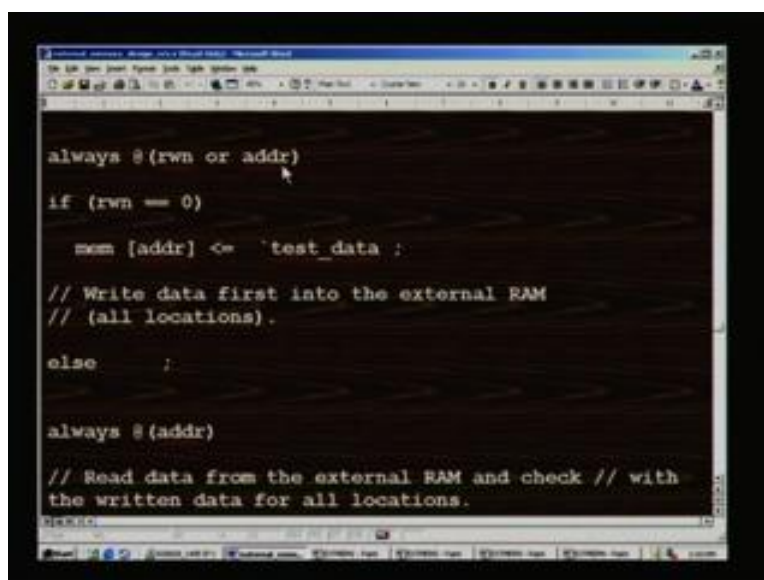
    mem [addr] <= `test_data ;

// Write data first into the external RAM
// (all locations).

else ;
```

In the next phase, we will change over to the read mode and check that all five 5s are intact. That is what the test is. The result of this test, we should be in a position to indicate as a Go-No Go test. That is to say that, whether the test has passed or failed only that must be reported. No unnecessary locations reported or address reported.

(Refer Slide Time: 24:20)



```
always @(rwn or addr)

if (rwn == 0)

    mem [addr] <= `test_data ;

// Write data first into the external RAM
// (all locations).

else ;

always @(addr)

// Read data from the external RAM and check // with
the written data for all locations.
```

Before we wind up the test bench so what we have here is one always block, which takes this following action here. For rwn or address changing and this one notice that it is read or write mode selection, here if it is 0, that is, if it is write, what should happen? Should write the test data; this data was initialized to all 5s here and that 5s will have to go into the memory. This reg mem we have seen to be within the test bench. To start with address will be 0 and that is what we have seen while looking into the address and initialize to 0 at power on reset. It starts with 0 and automatically the design increments the address and also applies read or write pulse only every once in two clock cycles. Automatically, the design takes care of writing into this RAM, although this external RAM is within the test bench here. To this particular location, address by the controller will be return into the data test pattern, namely 555 will be written into the RAM here, in the test bench.

(Refer Slide Time: 25:31)



```
if (rwn == 0)

    mem [addr] <= `test_data ;

    // Write data first into the external RAM
    // (all locations).
else ;

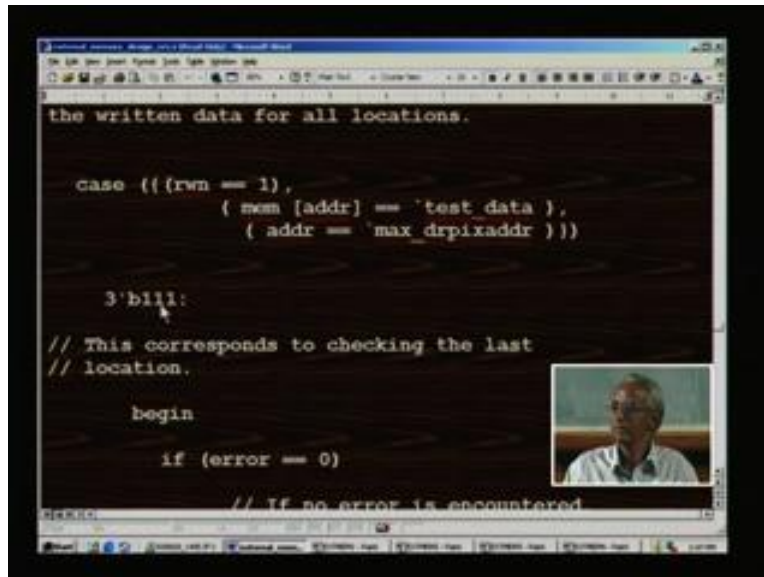
always @(addr)

// Read data from the external RAM and check // with
the written data for all locations.

    case ((rwn == 1),
          { mem [addr] == `test_data },
```

Write data first into the external RAM all locations. So what we do is we write all five 5s in every location. There are 65535 locations. All locations it will write the same data in the first phase. Then change this rwn and then start reading, in order to check whether the contents are still 555 that are what we do here. In order to do this one, we look for the change of address. Earlier always block is also a combinational as much as this block and read data from the external RAM and check with the written data for all locations. That is what we have said here. We use a case in order to do the read operation.

(Refer Slide Time: 26:17)



```
the written data for all locations.

case ((rwn == 1),
      ( mem [addr] == `test_data ),
      ( addr == `max_drpixaddr ))


    3'bill:
// This corresponds to checking the last
// location.

    begin
        if (error == 0)
            // If no error is encountered
```

Read operation we do just note that, this is concordation here. This is actually 3 bits only. All this put together or only 3 bits as is depicted here and this first 1 refers to the returned value of rwn equal to 1. That is to say, if rwn is 1 then, this whole bracket will write just 1. If it is not 1, that is, if it is 0 it will turn 0, this particular bit pattern is test for the complete bracket, whatever is mentioned within brackets. Also for the entire thing here, in this case, what it says is check the read the memory whose address is given in addr. That is from the external RAM which is in the test bench and compared with the test data that we have already written. Initialize in the test bench. That is all 5s here. If it is equal to 5, the red content, then only it will this whole thing will return 1 and will correspond to the second bit here.

The third case is, when the address is the last address. For example, address is the current address which is running in the design and finally it will be incremented write from 0 then 1 2 3 and so on. It will go write up to 65535. When it touches that value that is what is here. This equality will be satisfied then logical 1 will be written and that is this here. In this case, what here means is we are in read mode that is what we want and memory address is also verified with the data already stored and that is why 1 is written here. Here this is the very last location that we had it, for example 65535 locations. That means, to say we have earlier we have already read all the 65000 locations.

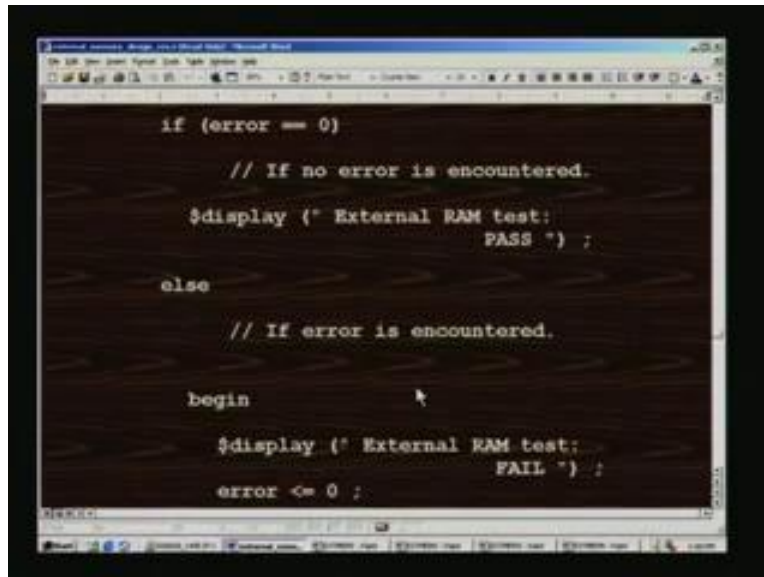
(Refer Slide Time: 28:25)

A screenshot of a code editor window with a dark background. The code is written in Verilog and is part of a module named '3'b111'. It contains a comment indicating that the code corresponds to checking the last location. The code uses an 'if' statement to check if the 'error' flag is 0. If it is, it displays the message 'External RAM test: PASS'.

```
3'b111:  
  
// This corresponds to checking the last  
// location.  
  
begin  
  
    if (error == 0)  
  
        // If no error is encountered.  
  
        $display (" External RAM test:  
                PASS ");  
  
    else
```

We are writing at the fag end of the RAM test and when this happens what we do is? We further have 1 more flag. This also a signal calling error and during the course of writing and checking back every location it may so happen that one of the locations is faulty which case in error flag is set. This will be seen later on. If that error is 0, this is it would assure as that there were no errors reading from any of the locations including this last location that is 65535 here. When there is no error and what we should do is just display this external RAM test PASS. As I mentioned after checking all the locations it will simply report that external RAM test is passed.

(Refer Slide Time: 29:13)

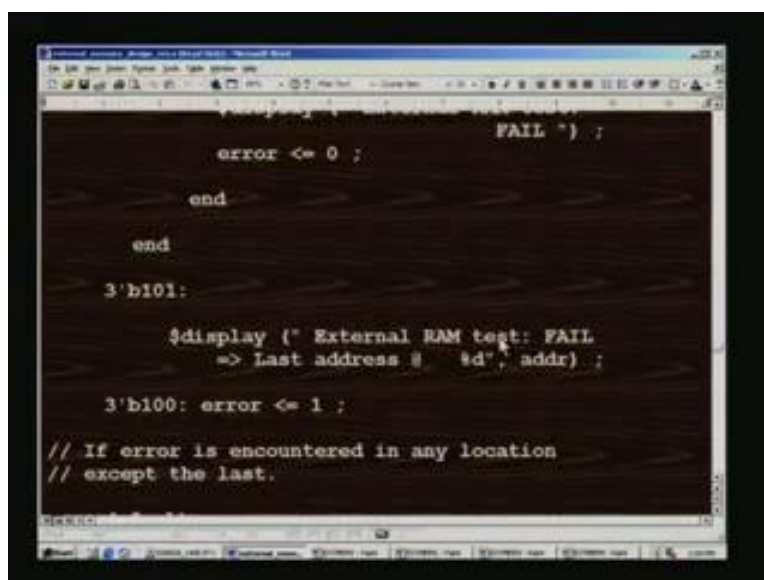


```
if (error == 0)
    // If no error is encountered.
    $display (" External RAM test:
                PASS ");
else
    // If error is encountered.

begin
    $display (" External RAM test:
                FAIL ");
    error <= 0 ;
```

If error is encountered on the other hand we have to report it is failure. After reporting this one, we should not forget error flag to be reset, because in this case error flag would have been set that is made 1. You should not forget this here. That is why beginning and end is given because more than two instructions are used, statements are used.

(Refer Slide Time: 29:38)



```
                FAIL ");
    error <= 0 ;

end

end

3'b101:
    $display (" External RAM test: FAIL
=> Last address # %d", addr) ;

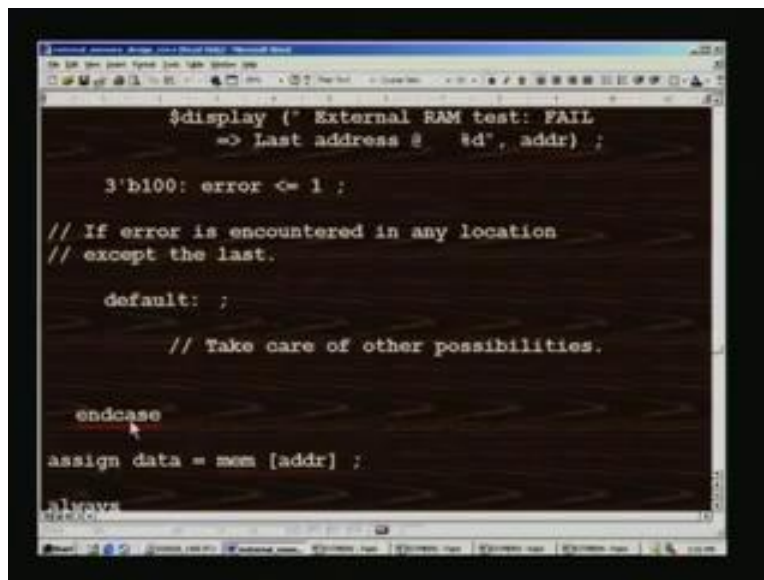
3'b100: error <= 1 ;

// If error is encountered in any location
// except the last.
```

The next combination in the case can be 101. In this case 0 corresponds to the error here and 1 corresponds to the very last memory being processed that means error has been encountered. This error has naturally come only for the very last location 65535. Therefore, we report this external RAM test fail and we also report the last address namely it is like a c comment here, precisely, c statement. That is why we use only in the test bench we should not use such statements in the actual design. It is precisely the same syntax or c, address is the variable which is the running address value and that is in decimal. Whatever is the actual address will be output here at 65535. It would have reported if there is an error at the last location.

Another thing is last combination here is 1 0 0 which implies, we are not in the last location. It may be any other address, other than the last right from 0 1 2 3 right up to the last but 1. When there is an error for any of the location this error flag is set. That is what I mentioned earlier, error flag is set later on and it is here, the error flag is set here. The error can occur at any location and it will be promptly recorded by this statement. We have covered all the conditions except for the 11 00 01 11 condition we have covered. That leaves 10. So, 110 we have not seen.

(Refer Slide Time: 31:31)



```
$display (" External RAM test: FAIL
=> Last address @ %d", addr) ;

3'b100: error <= 1 ;

// If error is encountered in any location
// except the last.

default: ;

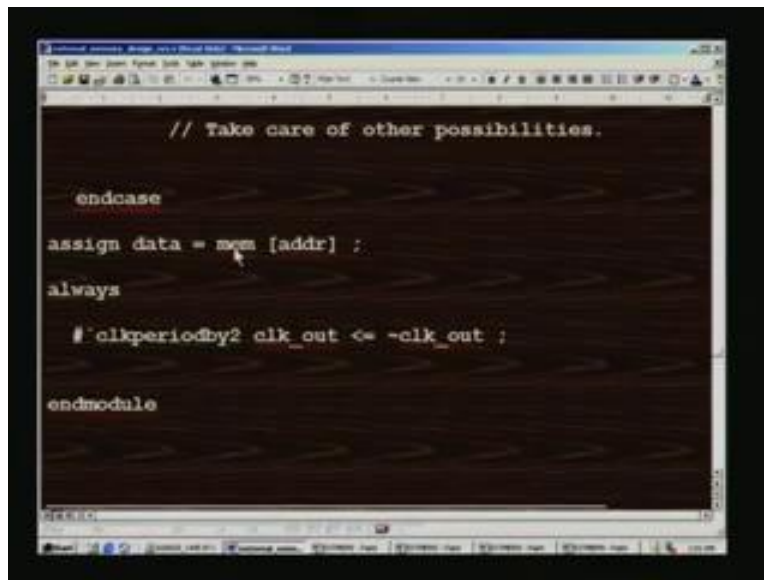
// Take care of other possibilities.

endcase
assign data = mem [addr] ;
```

Should we seen for that? It will be covered under default. If error is encountered in any locations except the last, that is what the statement comment for that. If there is a default take care of other possibilities. Other possibilities are 110. So 110 means, there is no error, but this is not the last

location. We do not have to do any work there. Therefore a dummy statement is written as a default and for other combinations it will be only 0 here. Four combinations will arise for 0 but 0 corresponds to write. Therefore we do not have to take any action. All those five states are automatically covered in the default and we started with a case therefore there is an end case.

(Refer Slide Time: 32:28)



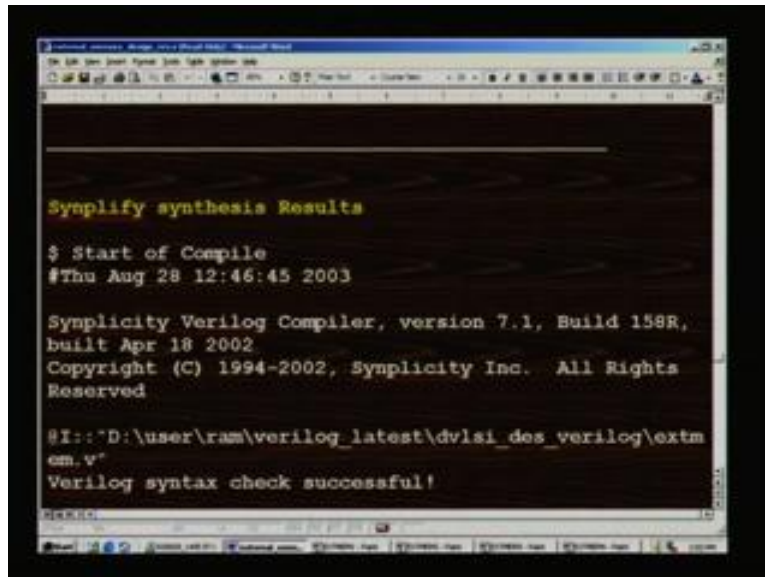
```
// Take care of other possibilities.

endcase
assign data = mem [addr] ;
always
    # clkperiod/2 clk_out <= ~clk_out ;

endmodule
```

What this instruction statement does is, we read memory particular address location memory and assign it to data, so that, we can view this also when we view the waveform. This is a usual statement we have been using for running the clock by inverting the clock and this is the end module for the test bench.

(Refer Slide Time: 32:49)



```
Synplify synthesis Results

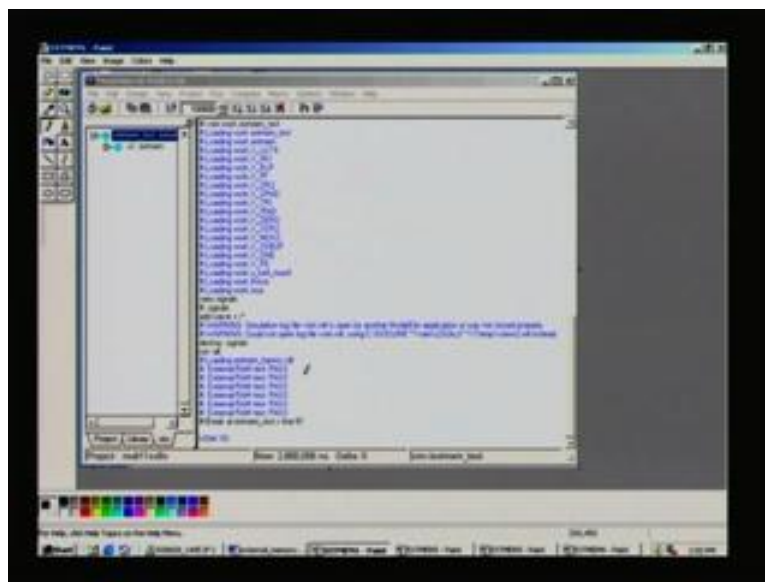
$ Start of Compile
#Thu Aug 28 12:46:45 2003

Synplify Verilog Compiler, version 7.1, Build 158R,
built Apr 18 2002
Copyright (C) 1994-2002, Synplify Inc. All Rights
Reserved

@I:: "D:\user\ram\verilog_latest\dvlsi_des_verilog\extm
em.v"
Verilog syntax check successful!
```

Before we see the synthesis results, we will have look at the waveforms.

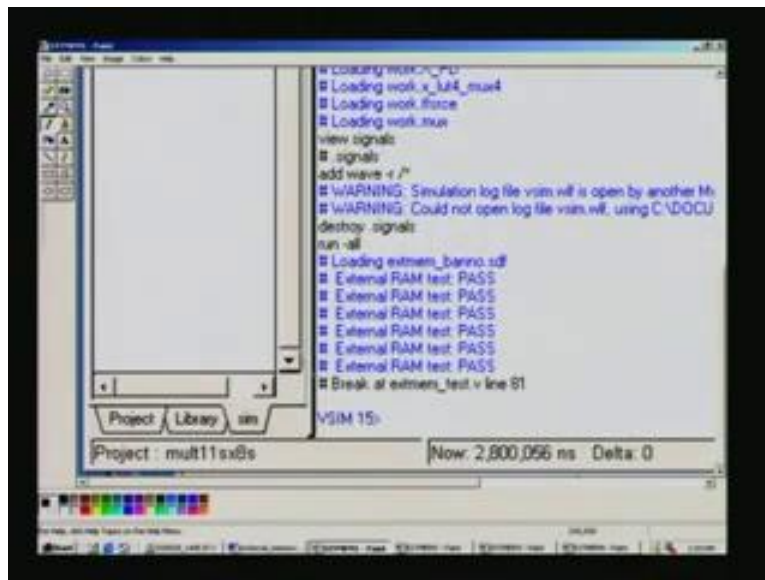
(Refer Slide Time: 32:59)



The first thing what we had was this is the model SIM result. You can see that external RAM test is passed. It is a Go-No Go test, this how it reports here. Have there been any error it would has mentioned fail and corresponding location or something it would have reported. You can just

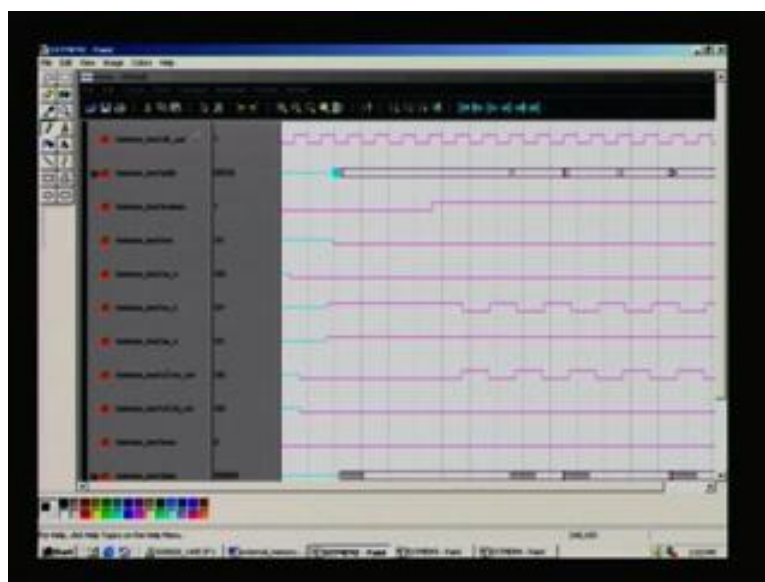
zoom here. This we have taken back annotated file and that is why loading shows all the primitives there.

(Refer Slide Time: 33:32)



Finally, we have run the actual process. It had run that external memory back annotated file and reported it that the test is passed. That means all the 65535 locations has been successfully written into and read back. This is with only data 555. You can experiment with other data.

(Refer Slide Time: 34:04)



Going to the waveforms, here, I will first read out before zooming, we have a clock out here and this is the clock signal operating at 100 megahertz. This is the address that we have it may be a write address or read address here depending upon this rwn control. If it is 0 it means write and it is right now in the right mode, because the start we want to write some data and let us see what data we want and we have. We have same all five 5s. You can see 2, 4, 6 5s are there. The same data is being rewritten, that is the reason why it is changing here at every write point. Address counter is also changing here. 0 here and then 1 here and so on and it keep going. The actual RAM is enabled only at this point of time. Therefore starts functioning only from this point of time. In fact from subsequent positive edge only it is actively processing and at every note that this counter is I mean address is changing only for once in two clock cycles. Next clock cycle it does not change and this is the reason why we have. We will have a look at the write pulse here. You need to select the chip, so we make the chips low the test bench we had made it here.

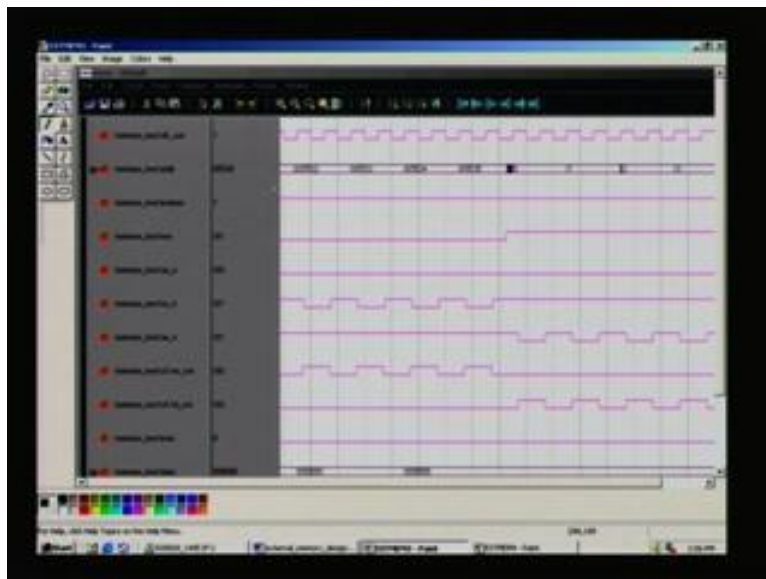
RW1 is in right mode as we have seen its low and chip is enabled all through and write pulse is applied for this is corresponding to the 0 address here. Write pulse is applied after the positive edge of the clock here. So that address is always applied at the positive edge, you can see here. Always at the positive edge address is changed; whereas, the write pulse is applied only after the address stabilizes. So, we give and register at the negative edge here. That is what we have seen in the design. This is created at the negative edge and because it is back annotated there is a delay you can see. Here we have a write counter running and this toggle, because it is only a single bit 01. This is what is responsible for making this write pulse or read pulse when we see, read counter will be activated later on. If you see here what we have seen is just few samples here up to some five locations. We will see other waveforms too. Before this we will just zoom so that you get a grasp.

(Refer Slide Time: 36:57)



So, clock out etc., address changing here, this is the clock and (Refer Slide Time: 36:59) this is the data is changing here and write counter read counter. Write counter is this and oe is this here that is what we have seen, so we will look at another waveform.

(Refer Slide Time: 37:25)



This is the fag end of the test 65532 and then 65535. Then it rolls over to 0 here 1 2 3 and so on this is the address here. Now let us watch what is happening to this rwn control. It is all along

low that is what we started with we started writing for every location up to 65535. The data that is 555 which is at the bottom and now you can see that write pulse has stopped here, because write is over for with 65535. Soon after that, this address changes to 0 and also that rwn control changes to 1 here. We can start reading or I mean this being 1 it is read only mode, now also notice that this OE bar is the read signal here. OE bar has started become active here and write has stopped. So, the safe state is high state that is, inactive state for the write. Both are being active low, so designed active low that is why it is like that. You can clearly see that write pulse stops here (Refer Slide Time: 38:37) and with the switch over of rwn and reads starts here.

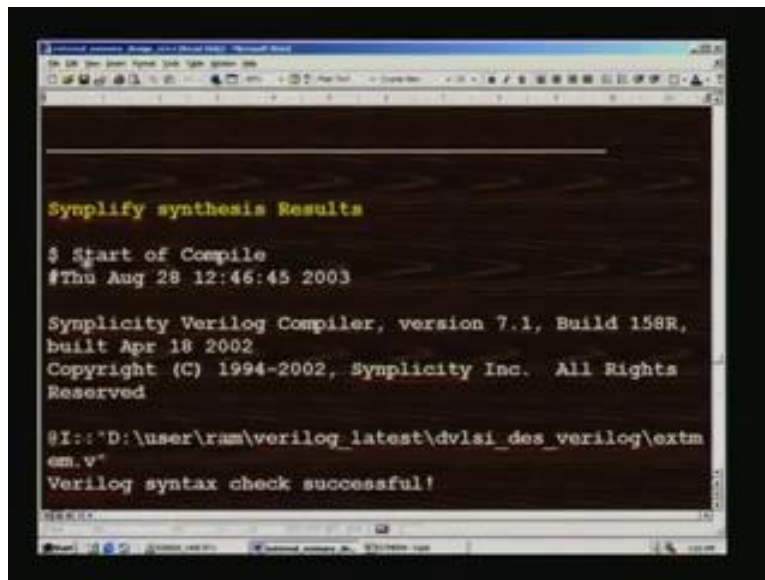
Once again you can see that write counter has been running in support of this write counter. Now this also stops here. Instead read counter is activated here which aids in the generating the OE bar signal here. Now there is no change of data, because it is continuously reading the same data. You can see huge time has elapsed and for 65535 locations we have not only written 555 data but also cross check and you can verify whether there is an error. You can see here this is the error here and it is 0 this lower than 0. There has been no error and we have seen in the model SIM window its reports test is passed. Coupled with this should prove that the whole thing is working and the zoomed version is here. You can see the clock out and address here and switching over to 65535 to 0, and you can see that rwn is changing to read here it was write here.

There is rwn, and then chip enable is always low. At the bottom of this so if it is high it would be on the top of this so you can see here. On the top write signals are activated here and OE is deactivated, because there should be no read pulse. Write counter is activated and there is no read counter there and that is what you see all through.

Finally, you have seen error being 0 here all through. It is 0 and this is what we have applied. It was changing because write pulse was active and there is no change during read phase. We have one more waveform towards the end, which is when it is complete the next address will switch over. For example, earlier it was high and it this is the second block has such, when it touches the highest value here and rwn is again made low. Meaning hereby it is taken to write here. With this we have done not only a write but also followed by a read. This completes the test and beyond that it is dummy running here. We are not really interested beyond this because with this it has

already reported error to be 0 and in model SIM window we have seen that it has reported correctly.

(Refer Slide Time: 41:54)



```
Synplify synthesis Results

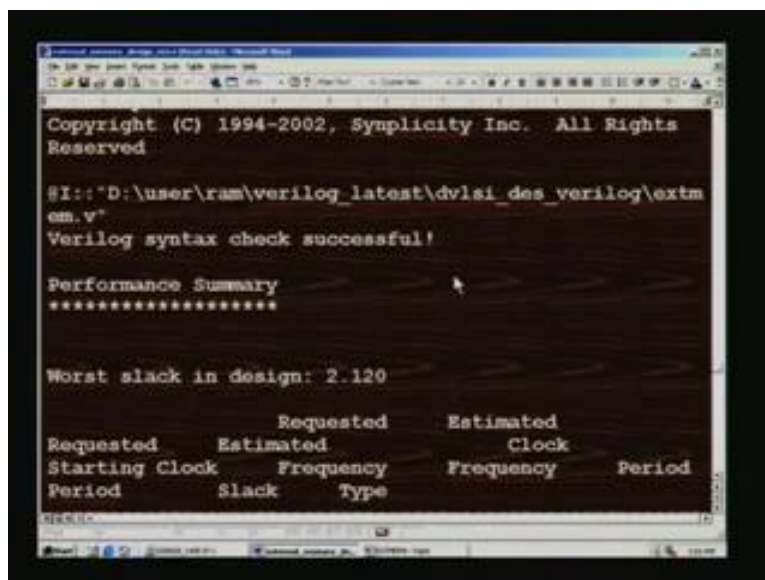
$ Start of Compile
#Thu Aug 28 12:46:45 2003

Synplicity Verilog Compiler, version 7.1, Build 158R,
built Apr 18 2002
Copyright (C) 1994-2002, Synplicity Inc. All Rights
Reserved

@I::"D:\user\ram\verilog_latest\dvlsi_des_verilog\extm
em.v"
Verilog syntax check successful!
```

Go back to the synthesis results and for this external memory.

(Refer Slide Time: 42:01)



```
Copyright (C) 1994-2002, Synplicity Inc. All Rights
Reserved

@I::"D:\user\ram\verilog_latest\dvlsi_des_verilog\extm
em.v"
Verilog syntax check successful!

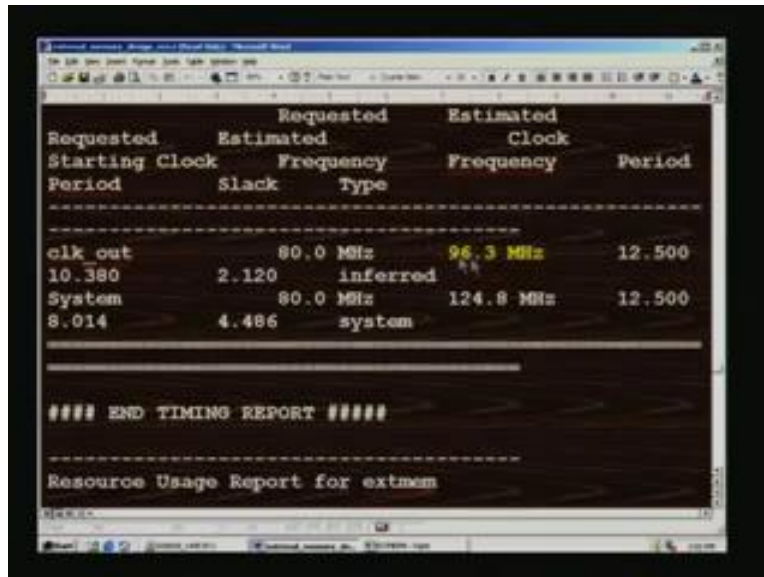
Performance Summary
*****

Worst slack in design: 2.120

Requested      Requested      Estimated
Requested      Estimated      Clock
Starting Clock  Frequency      Frequency      Period
Period          Slack          Type
```

Mind you, the design does not contain the actual memory; it is only a controller. You will see that gate counter etc., will be less.

(Refer Slide Time: 42:08)



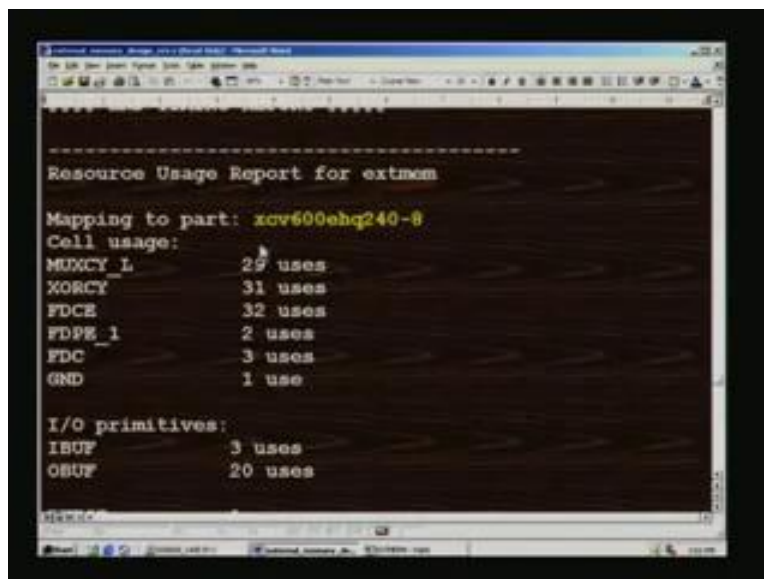
```
Requested      Requested      Estimated
Starting Clock Estimated      Clock
Period         Slack         Frequency      Frequency      Period
-----
clk_out         80.0 MHz      96.3 MHz      12.500
10.380         2.120         inferred
System         80.0 MHz      124.8 MHz     12.500
8.014         4.486         system

#### END TIMING REPORT ####

Resource Usage Report for extmem
```

The operating frequency reported by synthesis 96 megahertz.

(Refer Slide Time: 42:16)



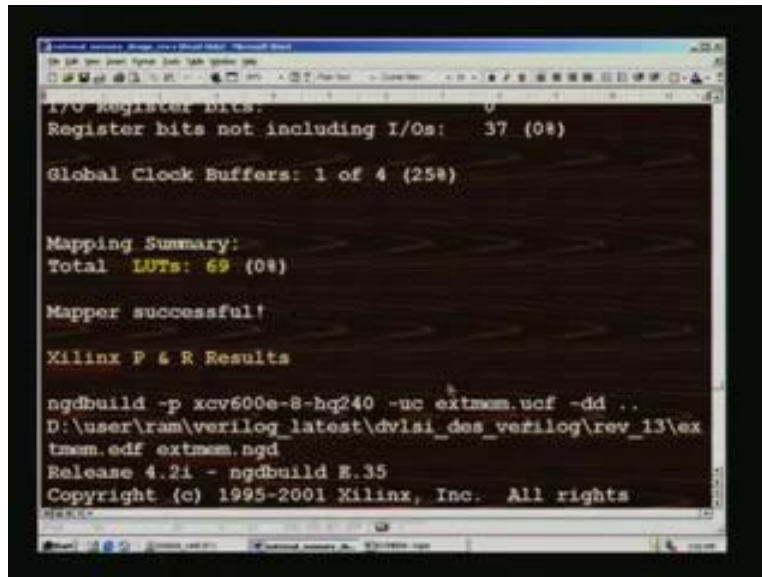
```
Resource Usage Report for extmem

Mapping to part: xcv600ehq240-8
Cell usage:
MUNCY_L      29 uses
XORCY       31 uses
FDCE        32 uses
FDPE_1       2 uses
FDC          3 uses
GND          1 use

I/O primitives:
IBUF         3 uses
OBUF        20 uses
```

With the same device that we have used before.

(Refer Slide Time: 42:22)



```
I/O Register Bits: 0
Register bits not including I/Os: 37 (0%)

Global Clock Buffers: 1 of 4 (25%)

Mapping Summary:
Total LUTs: 69 (0%)


Mapper successful!

Xilinx P & R Results

ngdbuild -p xcv600e-8-bq240 -uc extmem.ucf -dd ..
D:\user\ram\verilog_latest\dvlsi_des_verilog\rev_13\ex
tmem.edf extmem.ngd
Release 4.2i - ngdbuild E.35
Copyright (c) 1995-2001 Xilinx, Inc. All rights
reserved.
```

It has taken only 69 LUTs.

(Refer Slide Time: 42:29)



```
Number used as a route-thru: 8
Number of bonded IOBs: 23 out of 158
14%
Number of GCLKs: 1 out of 4
25%
Number of GCLKIOBs: 1 out of 4
25%
Total equivalent gate count for design: 794
Additional JTAG gate count for IOBs: 1,152

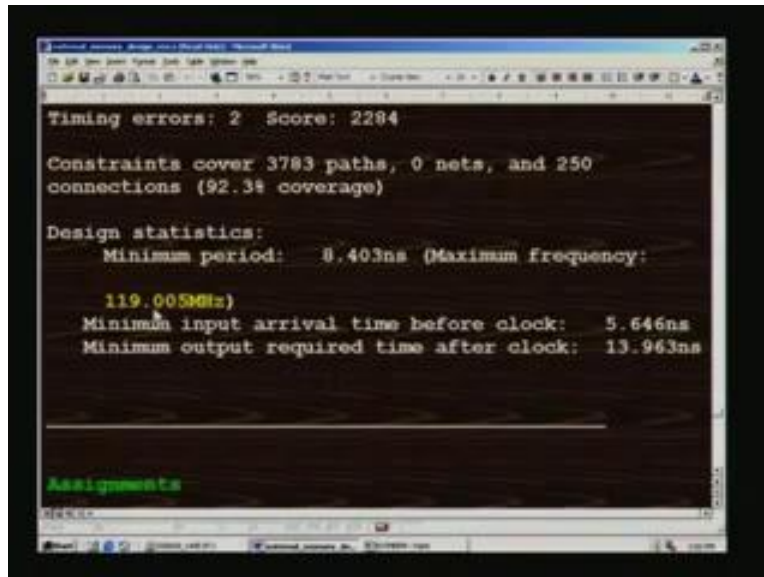
Timing summary:
-----

Timing errors: 2 Score: 2284

Constraints cover 3783 paths, 0 nets, and 250
connections (92.3% coverage)
```

xilinx place and route results are here for the same thing and it takes some 1800 gates only.

(Refer Slide Time: 42:35)



```
Timing errors: 2 Score: 2284

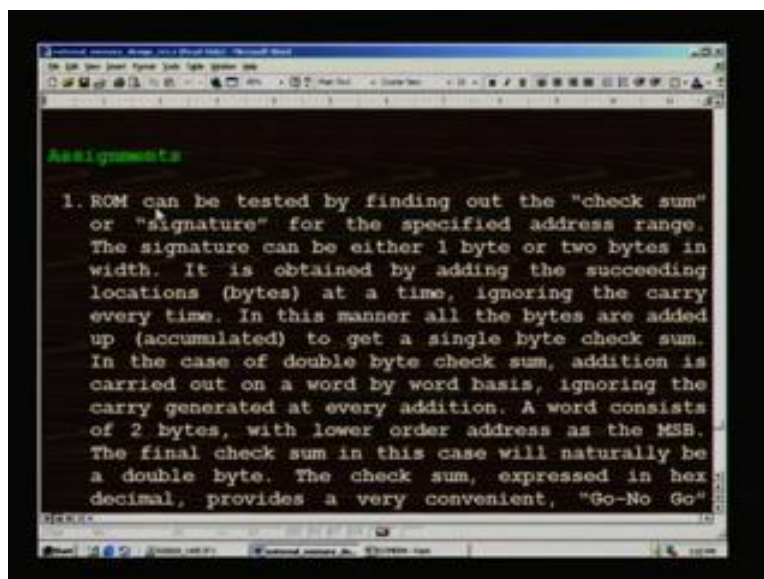
Constraints cover 3783 paths, 0 nets, and 250
connections (92.3% coverage)

Design statistics:
  Minimum period: 8.403ns (Maximum frequency:
  119.005MHz)
  Minimum input arrival time before clock: 5.646ns
  Minimum output required time after clock: 13.963ns

Assignments
```

Speed probably has improved to 119 megahertz here. We are justified in running at 100 megahertz. The clock rate is 100 megahertz where as the access is 50 megahertz. With 50 megahertz, we can have actual RAM. This has been designed with vendor's device, as such taken into account. That means to say that is stranded RAM available, so that we may use later on in the actual project that is 64k into 24 bits.

(Refer Slide Time: 43:19)

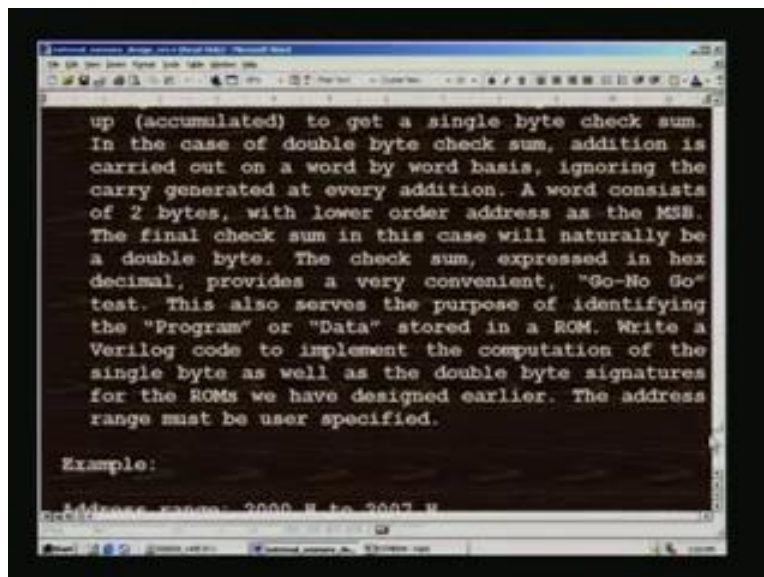


```
Assignments

1. ROM can be tested by finding out the "check sum"
or "signature" for the specified address range.
The signature can be either 1 byte or two bytes in
width. It is obtained by adding the succeeding
locations (bytes) at a time, ignoring the carry
every time. In this manner all the bytes are added
up (accumulated) to get a single byte check sum.
In the case of double byte check sum, addition is
carried out on a word by word basis, ignoring the
carry generated at every addition. A word consists
of 2 bytes, with lower order address as the MSB.
The final check sum in this case will naturally be
a double byte. The check sum, expressed in hex
decimal, provides a very convenient, "Go-No Go"
```

Before we wind up we have some assignments for you. I will quickly read and you will have to solve this problem. First, this covers the entire memories and we will look into the assignment for RAM. RAM can be tested by finding out the check sum or signature for the specified address range. The signature can be either 1 byte or 2 bytes in width. It is obtained by adding the succeeding locations bytes at a time ignoring the carry every time. In this manner all the bytes are added up, that is accumulated, to get a single byte check sum. In the case of double byte check sum, addition is carried out on a word by word basis, ignoring the carry generated at every addition.

(Refer Slide Time: 43:57)

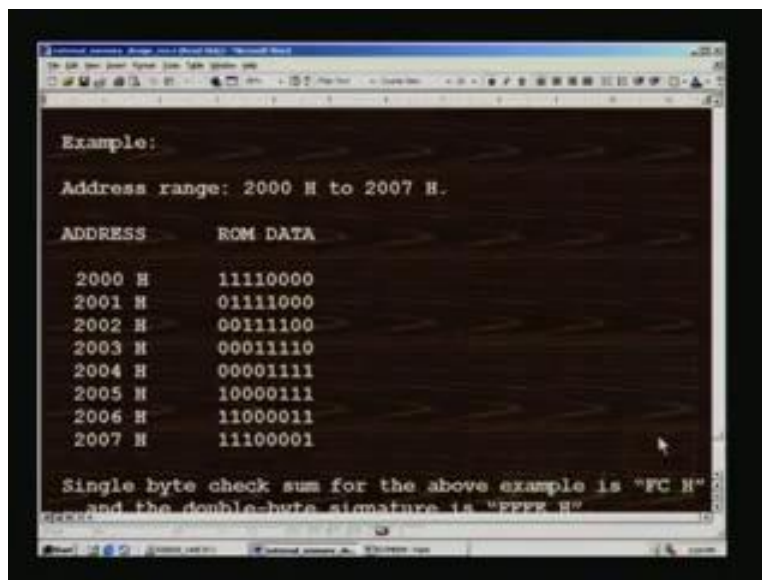


A word consists of 2 bytes with lower order address as the MSB. The final check sum in this case will naturally be a double byte. The check sum, expressed in hex decimal, provides a very convenient Go-No Go test. This also serves the purpose of identifying the program or data stored in a ROM. Write a verilog code to implement the computation of the single byte as well as the double byte signatures for the ROMs we have designed earlier. The address range must be user specified.

So the user will you have to write a test bench in order to compute the check sum it may be a single byte or double byte which I will take an example and explain. You have to write a test

bench for that. You should assume that user supplies the actual address range for example, this is the address range user will supply (Refer Slide Time: 44:43)

(Refer Slide Time: 44:47)



```
Example:
Address range: 2000 H to 2007 H.

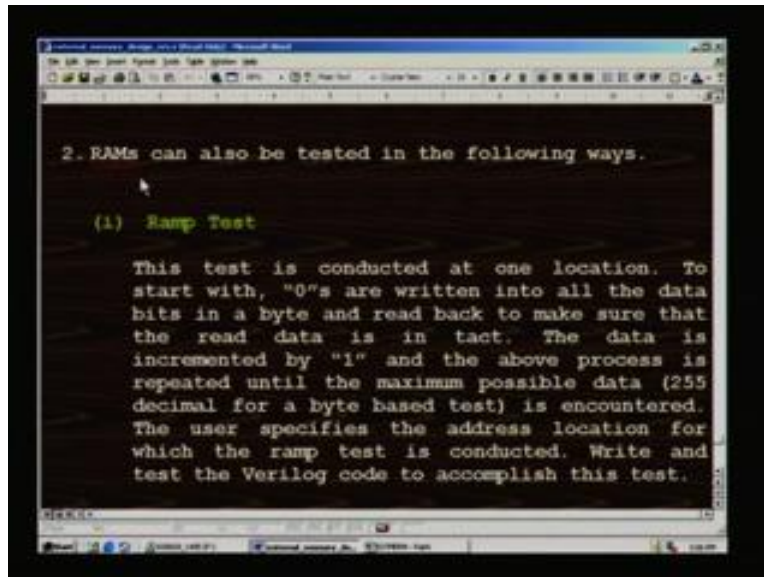
ADDRESS      ROM DATA
2000 H       11110000
2001 H       01111000
2002 H       00111100
2003 H       00011110
2004 H       00001111
2005 H       10000111
2006 H       11000011
2007 H       11100001

Single byte check sum for the above example is "FC H"
and the double-byte signature is "FFFE H"
```

You will have to take input this information and then compute on this check sum what we are going to explain now and then report finally, whether it is a pass or otherwise not pass or fail but the actual signature. For example, if it is a single byte signature check sum what we do is? We add these two bytes and ignore the carry from this. Keep the result and add repeated repetitively for the subsequent bytes. For example, that result you add with this one, once again ignore the carry and keep repeating till you have exhausted all the data. Finally, you will get 1 byte answer and expressed in hex decimal is the check sum. It is also known as the signature and he can also have a double byte signature. In this example single byte signature if you compute manually, you will see that it is FC in hex decimal, and if it is double byte check sum it is FFFE.

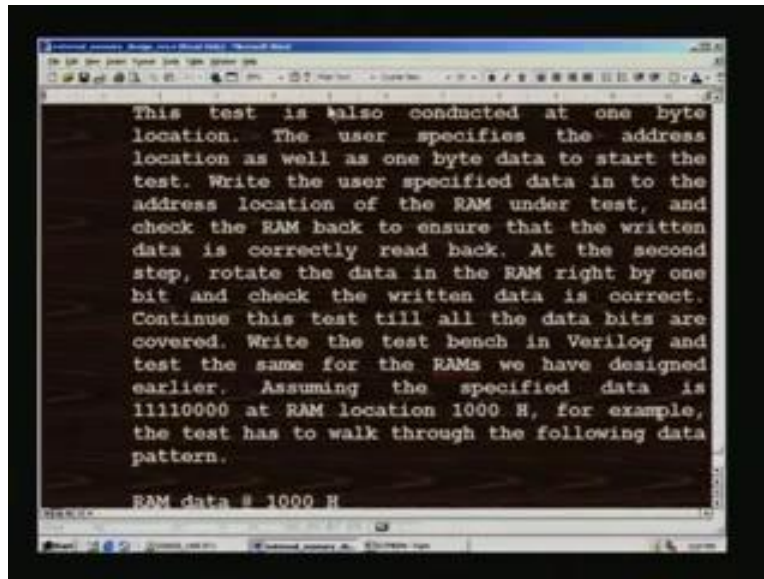
How to get this double byte signature is? You put this second byte here and treat the whole thing as a 16 bit number. Similarly, put this one here and treat as second number, so you have two 16 bit numbers; add the two as if it is whole 16 bit each and ignore the carry as we have done for the byte, check sum. Result will be again 16 bits and repeat the same thing for this group and this group. Finally, you will get a 16 bit check sum and that is also known as a signature. For this example it is FFFE.

(Refer Slide Time: 46:36)



We have assignments for RAM. There are two assignments. I will read it once again. RAMs can also be tested in the following ways: First is what is called ramp test. Normally, these tests are very popular in microprocessor based design. It is called the ramp test and walk test. In this, test is conducted at 1 location, earlier we have conducted at range of locations; whereas, this is conducted only at a particular location. This holds good only for RAM where you can read as well as write. To start with, 0s or written into all the data bits in a byte and read back to make sure that the read data is intact. What you have written into you should make sure by reading that same data is still intact. The data is incremented by 1 if it is 0 we increment it to 1, then 2 and so on right up to 255 if it is single byte. If it is high (47:30) 32 bit data you will have to go right up to the corresponding decimal value. This, needless to say this will be a very time consuming test and is a and the above process is repeated until the maximum possible data 255 decimal for a byte based test is encountered. The user specifies the address location for which the ramp test is conducted. Write and test the verilog code to accomplish this test. So write accordingly.

(Refer Slide Time: 48:01)



There is also a walk test. This test is conducted at one byte location, as in ramp test. The user specifies the address location as well as 1 byte data to start the test. Write the user specified data in to the address location of the RAM under test and check the RAM back to ensure that the written data is correctly read back. At the second step rotate the data in the RAM right by one bit and check the written data is correct. Continue this test till all the data bits are covered. Write the test bench in verilog and test the same for the RAMs we have designed earlier. Assuming the specified data is 11110000 at RAM location 1000 H, for example, the test has to walk through the following data pattern.

(Refer Slide Time: 48:45)



```
caller. Assuming the specified data is
11110000 at RAM location 1000 H, for example,
the test has to walk through the following data
pattern.

RAM data # 1000 H

11110000
01111000
00111100
00011110
00001111
10000111
11000011
11100001
```

We are testing this illustration with an example. What we need to do is, we have to have, I mean user will have to specify, what data to be written at what location. This location address as well as data will be furnished by the user; you have to take that as an input for this test; conduct this test and then report whether it is a pass or fail.

What we do here is, suppose the user has given this as the data. What we do is? We write this data into this RAM location and check back to make sure that it is the same data. Then, at the next step what you do is? Rotate this one, you may have some sort of accumulator where in you rotate and then write back the rotated value here. Rotate in sense, this 0 will rotate back to this position. Every time rotate by 1 and then write it as well as check the data written is correct and keep repeating every time shifting by 1 or rather rotating by 1. When you come to the last one when you rotate this, this 1 will go to 1 which will be the same pattern as we started before. Therefore we do not test that condition again, so you just exhaust all possibilities here being an 8 bit, you need only 8 such types of write. After this writing, when all are successfully return and checked back, you report pass. This ends the memory design. Thank You.

Summary of Lecture 39

(Refer Slide Time: 50:25)

