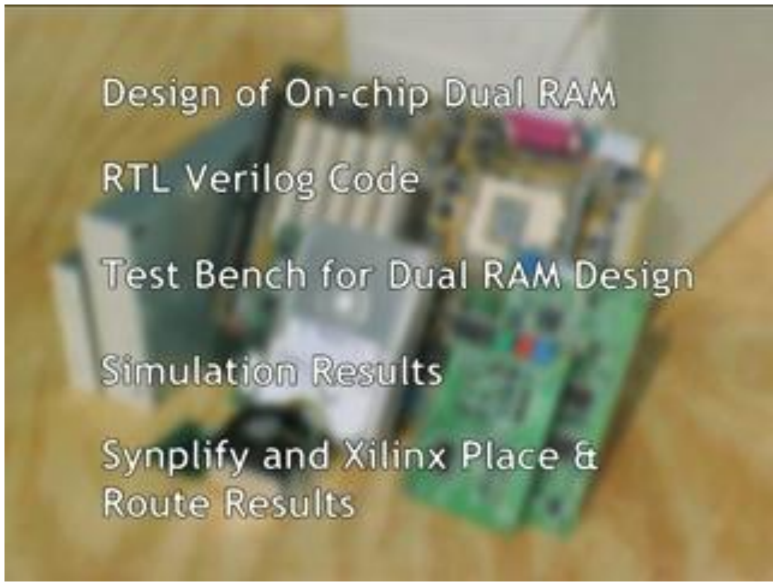
Digital VLSI System Design

Prof. Dr. S. Ramachandran
Department of Electrical Engineering
Indian Institute of Technology, Madras

Lecture - 38

Design of Memories - RAM

(Refer Slide Time: 01:49)


Design of On-chip Dual RAM

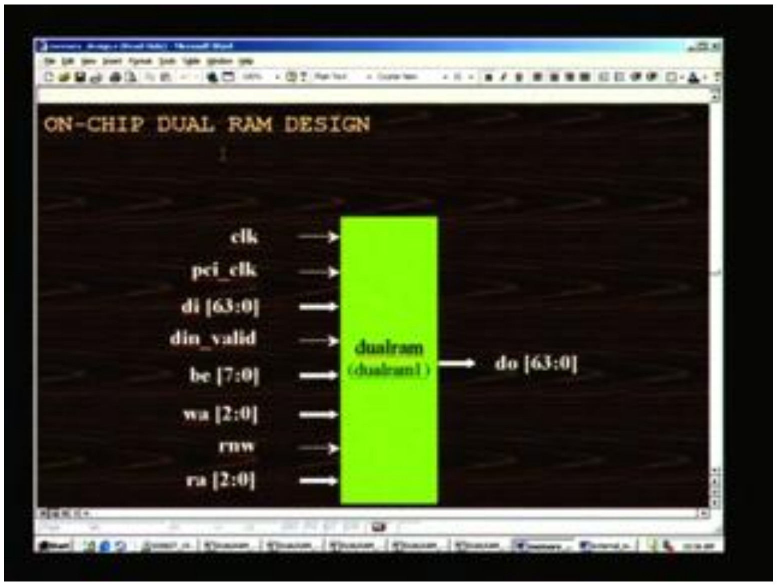
RTL Verilog Code

Test Bench for Dual RAM Design

Simulation Results

Synplify and Xilinx Place & Route Results

(Refer Slide Time: 02:21)



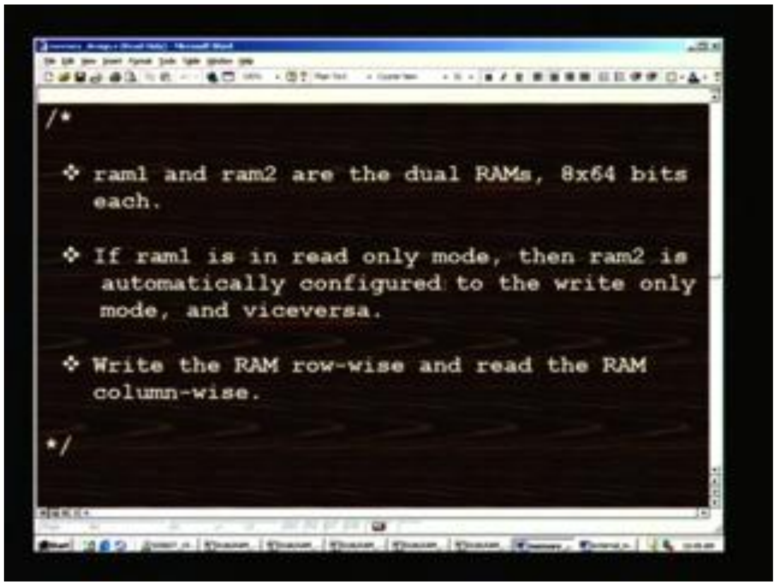
We were looking into the design of ROMs earlier, we will now have a look at the design of RAM and this falls deep into the VLSI category in the sense, that we may encounter much, about 50,000 transistors. Once again here, the requirement arises for a particular memory organization from the design application. Actually, this application is also for DCTQ which we have seen earlier and it requires dual RAM; it is not a dual port RAM, mind you, it is a dual RAM in the sense, we have two RAMs inside this black box to start with and each of these RAMs will be storing the image information and this information will be written through PCI bus. Prior to going to describing these signals, we will see what it really does. As I mentioned before, it has two RAMs when you start with, we will be writing into one of the RAMs and the other RAM will not be accessed. Once the first RAM is filled the same image will be filled, I mean different block of image will be filled into RAM two and when this happens, concurrently we will be reading from RAM one which was filled earlier and then process DCTQ from that block of image.

That is, first block image will be processed while second block of image is being concurrently loaded into this RAM that is how the requirement arose here. Another special feature in this dual RAM is, it is not really as I mean it is special in the sense, it is arising on account of the requirement for this DCTQ evaluation. The feature is, when we write into the RAM we will be writing 8 bytes at a time that is 64-bits. We have also said that it is through PCI interface bus assuming 64-bit data bus here; this is for the PCI signal. We will be writing only 8 bytes at a time, because what we need is only a block of image data, which is 8 by 8 pixels. Each of these pixels will be just one byte, if it is a monochrome, if it is colour it will be a 3 bytes. Right now, it has been implemented for monochrome. You can always duplicate this dual RAMs, I mean triplicate rather and in order to achieve colour motion picture as well. Here, as I mentioned before, we will be writing into one of the RAMs here and in order to write what we need is a separate input here; it should be 8 bytes in width that is 64-bits. This is the PCI compatible input bus through which we will be writing the actual data image data. In bus arbitration logic earlier we have precisely considered this particular application. We have seen that a video grabber is used and it goes over to the codec thereafter and within the codec, this is the design that has been housed. When this is valid, it is indicated, this also we can derive it from one of the PCI signals and there are many more PCI signals. We are not considering all at this point of time; you can bunch all the other requirements into what is called a data invalid.

When this data is valid is being indicated here and once again there are 8 bytes and each of these bytes when it is valid will be indicated by be signal stands for byte enable. This is also a PCI signal and there are 8-bits totally. Whichever is made 0 or 1, we can always invert the logic if you find any mistake. I think I have assumed 1 here, may be 0 is the right thing. It all calls for just inversion in the logic later on. To start with, suppose you make a particular be-bit go high, then only that byte will be written into and in real practice being a 64-bit all the 8 will be active simultaneously. That is the implication there. In order to write here, we need only a 3-bit address, for write address and this corresponds to eight such locations and each location is 64-bit in width. Here I am at liberty to do that, so I have made what you had expressed and what you have written you can also read and prior to reading, we should know whether a particular RAM is in write mode or read mode.

There are two RAMs; say RAM1 and RAM2 Out of these two, only one RAM will be in write mode, the other will be in read mode. So, that is how we accomplish continuous stream of image being fed in. Processing is happening concurrently and in order to distinguish this one, you have another control called rnw: n for negative that is low, read low and write is high. This is the nomenclature adapted here. These are all the signals pertaining to the design whereas DI then PCI clock is also there. All these four pertain to the PCI bus and there is also a system clock for this, especially for evaluating DCTQ, so that clock is this here (Refer Slide Time: 09:00). There are two clocks here: one is PCI clock; other is the system clock and once you have written into a RAM so, what we will be doing is we will change this value, so that the two RAMs interchange their roles. So, what was written into will now be read and you need a read address and that is once again it is eight locations read, that is what we are seeing here. As I mentioned there is a special access in this RAM which is quite different from the convention. In fact the whole design is totally unconventional and that special feature is as you write you will be writing along the row. The organization is 8 into 64 so, all along the row you will be writing into but while reading you will have to read it vertically that means to say, let us say there are locations location loc 0 through 7. So, we will be reading loc 0 higher order bit, I mean, byte 63 through 56 and location 1, next same number of bits and so on. So, what you write horizontally or row-wise you will be reading vertically or that is the requirement arising from the DCTQ algorithm the speaker has developed earlier for the DCTQ and which we will be considering later on. With this background I think we will be in a position to go through the design.

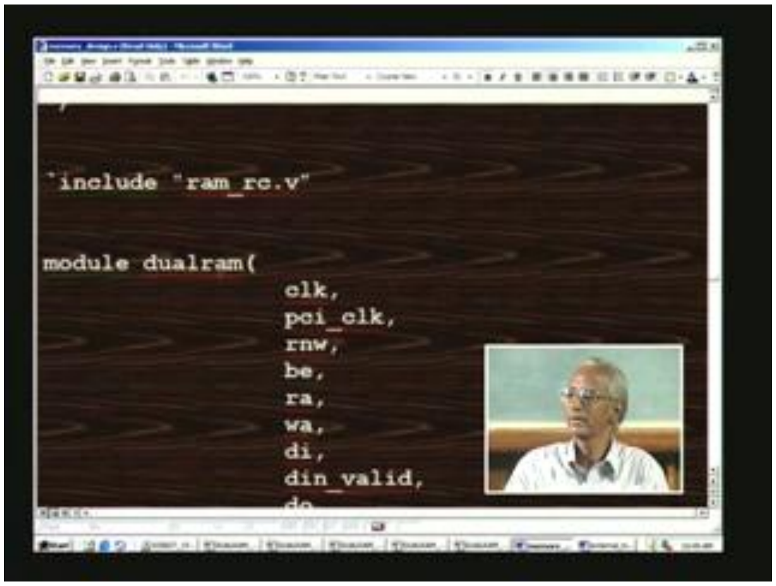
(Refer Slide Time: 10:32)



```
/*

 ❖ ram1 and ram2 are the dual RAMs, 8x64 bits
   each.

 ❖ If ram1 is in read only mode, then ram2 is
    automatically configured to the write only
    mode, and viceversa.

 ❖ Write the RAM row-wise and read the RAM
   column-wise.

*/
```

So before this, let us look at the comments here. RAM1 and RAM2 are the dual RAMs, 8 into 64-bits each. If RAM1 is in read only mode then RAM2 is automatically configured to the write only mode, and vice versa. This is what we have already seen. Write the RAM row-wise and read the RAM column-wise.

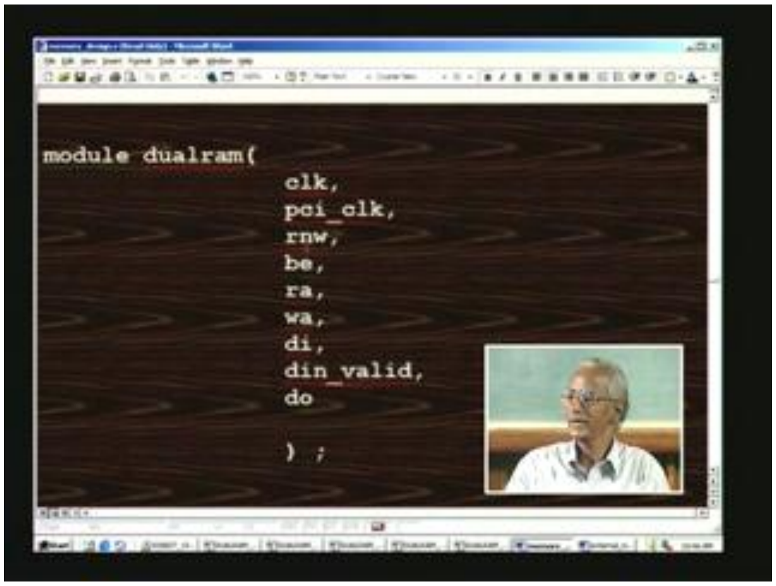
(Refer Slide Time: 10:57)



```
`include "ram_rc.v"

module dualram(
                    clk,
                    pci_clk,
                    rnw,
                    be,
                    ra,
                    wa,
                    di,
                    din_valid,
```

Here, we have to include another sub-module, we have seen earlier how the hierarchical approach by a flow chart that will be a top module. The module that you see here, which called dual RAM is given an apt name and this contains two RAMs you have to access RAM1 and RAM2. This RAM will be in a different file as such and that RAM must be included here and we have also said its speciality here is row right and column axis and that is why I have labelled like this. It is basically a RAM. This implies only a single RAM but this is invoked twice in this design and thereby, you make a dual RAM out of it. We have to declare module name, then give apt name, and list all the I/Os here and I/Os are precisely what we have seen in the diagram above.

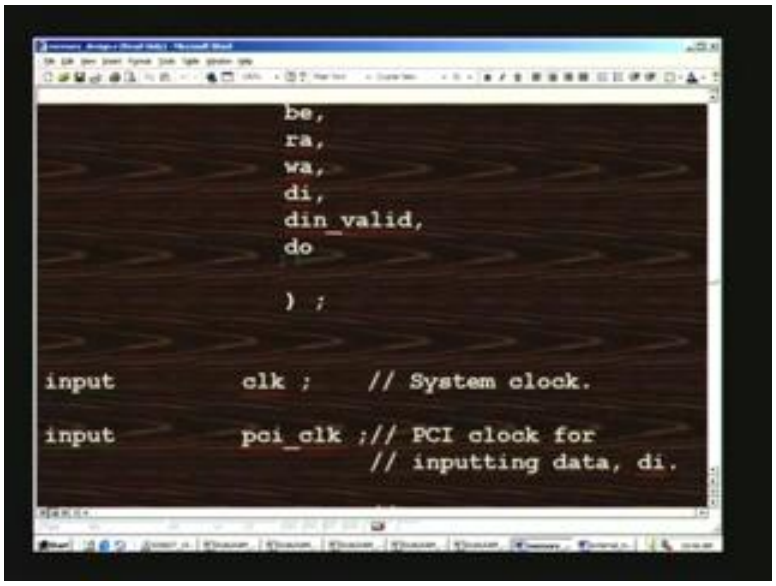
(Refer Slide Time: 11:57)



```
module dualram(
                clk,
                pci_clk,
                rnw,
                be,
                ra,
                wa,
                di,
                din_valid,
                do

                ) ;
```

It is a clock, PCI clock, rnw, be, read address, write address and this is the data in and its valid signal. This is the signal that out comes the column-wise read data. What we have listed here, we have just tabulated here.

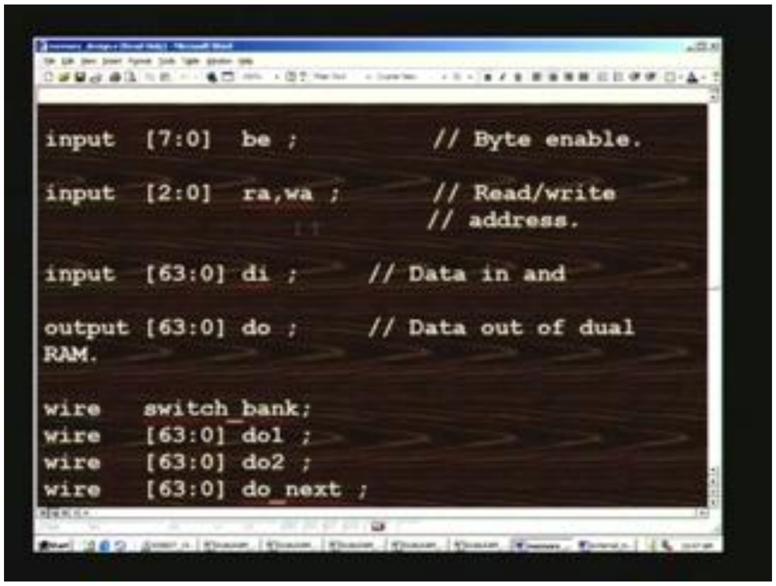
(Refer Slide Time: 12:15)



```
                be,
                ra,
                wa,
                di,
                din_valid,
                do

                ) ;


input          clk ;    // System clock.

input          pci_clk ;// PCI clock for
                        // inputting data, di.
```

We can just have a look quickly. So, all these are inputs and comment will make it self-explanatory. PCI clock, for inputting data, di. Sets one RAM in write only mode and the other RAM in read only mode. This is data valid and this byte enable.

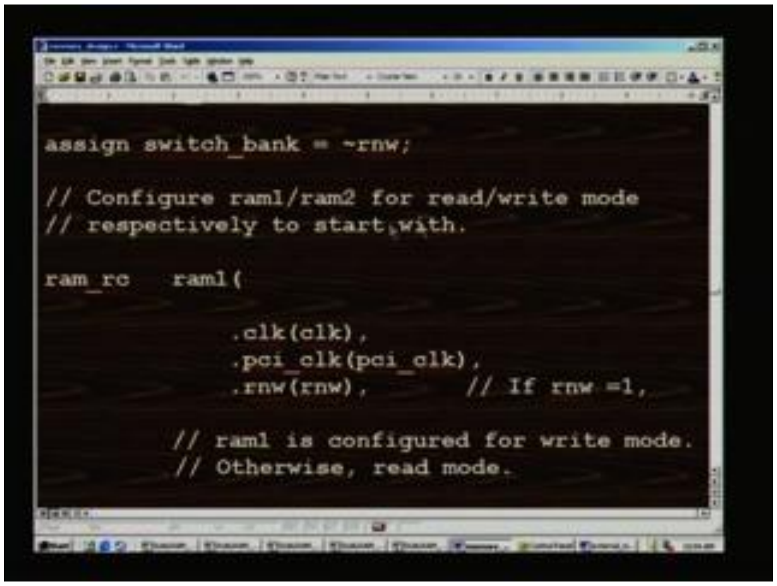
(Refer Slide Time: 12:45)



```
input  [7:0]  be ;          // Byte enable.

input  [2:0]  ra,wa ;       // Read/write
                            // address.

input  [63:0] di ;      // Data in and

output [63:0] do ;      // Data out of dual
RAM.

wire   switch_bank;
wire   [63:0] do1 ;
wire   [63:0] do2 ;
wire   [63:0] do_next ;
```

Then read write, you can even club like this, if you wish you can make it separate. Input for the data here and data out and width is also mentioned here. We have a signal called switch bank and this is nothing other than rnw inverted, as you can see in this statement "assign switch bank" is inverted signal. So why it is inverted I will explain a little later and we have two outputs corresponding to the two RAMs, that is why we have nomenclature it as do1 and do2 and do_next as we have used earlier is to indicate the next value of do which is the register value which will be at the next clock positive clock edge after this do next goes active. do is the output and we have to declare all these as wire because they are all assign statements and this is basically in always block with the positive edge clock and they are all basically registers.

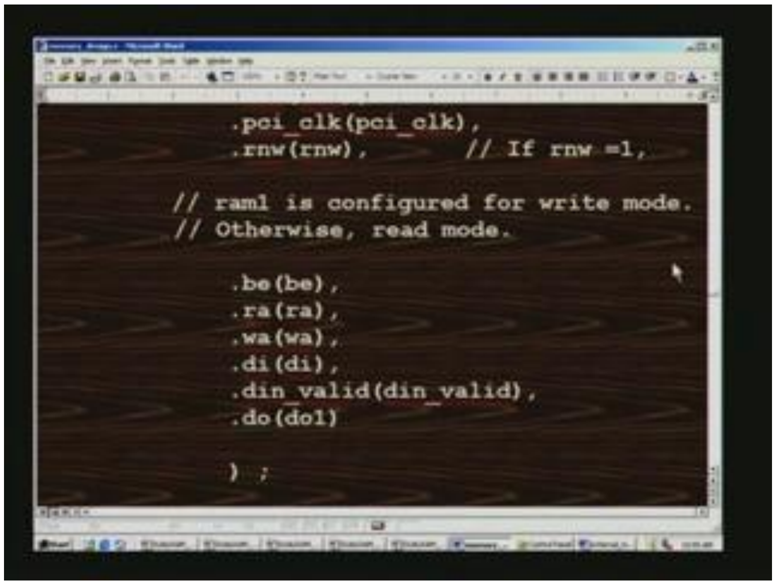
(Refer Slide Time: 14:14)



```
assign switch_bank = ~rnw;

// Configure raml/ram2 for read/write mode
// respectively to start with.

ram_rc   raml(

              .clk(clk),
              .pci_clk(pci_clk),
              .rnw(rnw),        // If rnw =1,

          // raml is configured for write mode.
          // Otherwise, read mode.
```

This particular control is to configure RAM1 and RAM2 for read or write mode respectively to start with. That is why we need here and this will be clearer when we really go into the actual module. Now we are in the dual RAM, this is the top design and we are going to call sub module such as I mean ram_rc. This is what we are going to see later and here we call this design twice. Once for invoking RAM1 and all the I/Os are listed here.

(Refer Slide Time: 15:02)



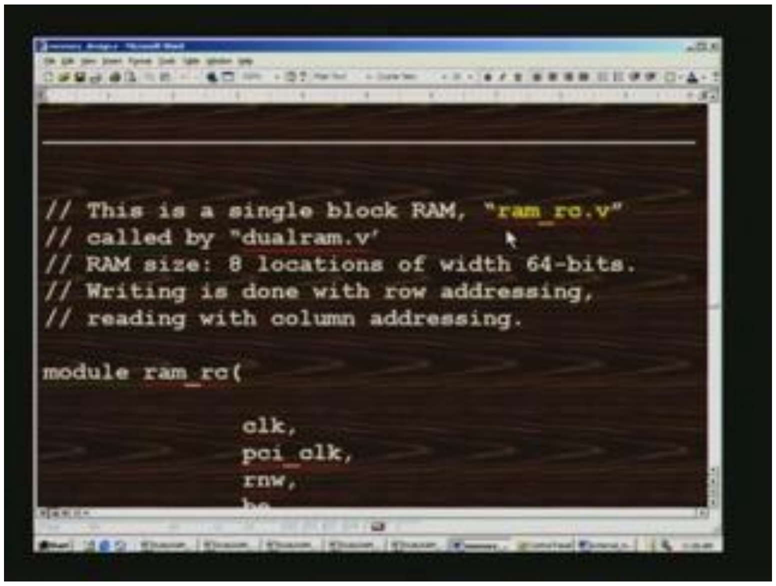
```
              .pci_clk(pci_clk),
              .rnw(rnw),        // If rnw =1,

          // raml is configured for write mode.
          // Otherwise, read mode.

              .be(be),
              .ra(ra),
              .wa(wa),
              .di(di),
              .din_valid(din_valid),
              .do(do1)

              ) ;
```

This is precisely the same I/Os that we have on dual RAM; namely clk, pci_clk, rnw, be, read address and write address. Everything is precisely same and di and din_valid and final output, except for the fact that we have two such RAMs invoked in this design. Otherwise this signal listing is precisely the same, so is the case for the RAM2. We invoke this RAM twice and thereby get two RAMs in this particular design and how we blend this, we will have a look. Here, as I mentioned switch bank was inverted rnw and here for RAM2 it is different here, this rnw control is the one which will configure it as read only or write. Let us have a look as to what has been configured. In this case, what it says is if rnw is equal to 1 that is without any inversion we are straight away taking rnw here for RAM1. If rnw is 1, one stands for the write portion and that means to say this RAM1 is configured for write mode and otherwise it will be read mode.

For the time being, let us stick on to one particular state so as to know what the role of RAM1 and RAM2 are. Clearly, RAM1 is in write mode if rnw is equal to 1. Let us see the state of affairs for RAM2 for rnw equal to 1. In this case, it is an inversion of rnw, because switch bank is an inverted rnw and if rnw is 1 automatically here, the local rnw is actually 0 so that means, it goes to read mode. When the RAM1 is in write mode RAM2 is automatically configured for the read mode and vice versa and that is how you can continuously fire the image into this dual RAM. Automatically this swapping will take place, this rnw will be automatically taking care in a controller which will be considering when we consider DCTQ design that is that control is not part of this dual RAM.

Once again, all the other signals are precisely same except for the fact that, do is the nomenclature now, this do2 refers to this top design that is the dual RAM and similarly for the RAM1, the nomenclature as do1. That is why; this has been declared as wire. I mentioned in one of the test benches way back so, how we call from top design a lower sub modules. This is the first thing I think we have encountered calling the sub module and this is a separate file which we will be looking into shortly. We see that, if rnw equal to 1 RAM2 is configured for read mode otherwise write mode, it is exact reverse of the RAM1 condition and here what we do is we want to have pipeline stage, we want to register basically. We do only at always positive edge clock and here the final output is do and the do is derived from basically a do next, this in turn is derived from either RAM1 data output or RAM2 data output and that depends upon this rnw control. The reason for one delay is being invoked here is to delay the actual rnw by 1 clock pulse just to keep pace with this delay because we are outputting at after clock edge. So, in order to keep step with that we are delaying that, otherwise the whole thing goes topsy-turvy, just one error you make, the whole system will become totally dead. In fact, I landed up in such a trouble, only after a few days I could discover that. So be on the lookout for such pit falls. That is how you get d not next at the final output and this is what you want either RAM content from RAM1 or RAM2 content that is for reading. Only while reading, this is valid here and while writing it from the PCI bus it will be straight away written. When we go into the RAM details the next file will become clear. This top design is very simple design here it merely invokes two RAMs. That is the end of the module there and we will go into the RAM rc.

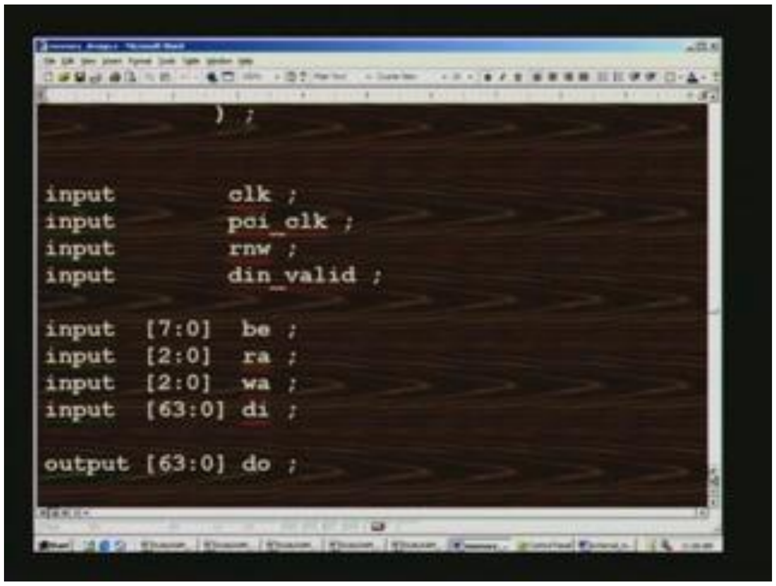
(Refer Slide Time: 20:09)



```
// This is a single block RAM, "ram_rc.v"
// called by "dualram.v'
// RAM size: 8 locations of width 64-bits.
// Writing is done with row addressing,
// reading with column addressing.

module ram_rc(

                clk,
                pci_clk,
                rnw,
```

This is another file and this file is being called by dual RAM that is what is indicated here. RAM size is as you know eight locations of width 64-bits and writing is done with row addressing reading with column addressing. This also we have covered for dual RAM, same thing applies for RAM_rc and once again the I/O details are exactly the same as we have seen before.

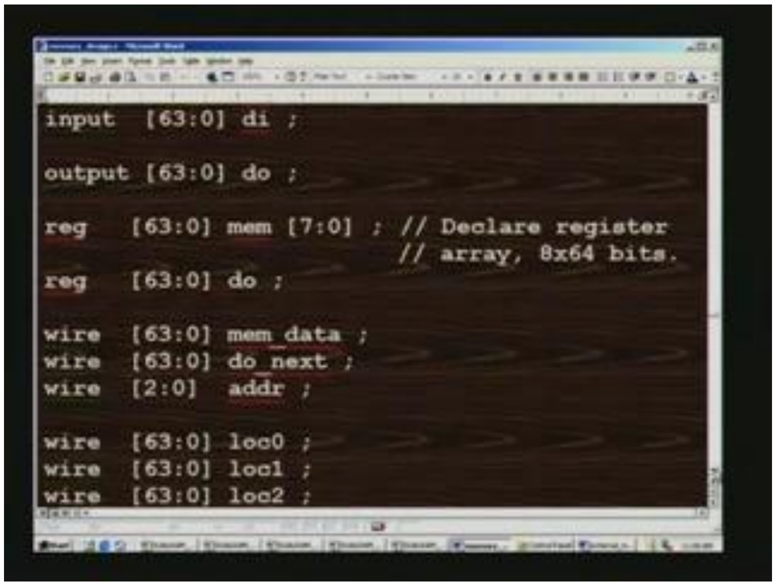
(Refer Slide Time: 20:34)



```
            ) ;


input        clk ;
input        pci_clk ;
input        rnw ;
input        din_valid ;

input  [7:0]  be ;
input  [2:0]  ra ;
input  [2:0]  wa ;
input  [63:0] di ;

output [63:0] do ;
```

The listing is the same.

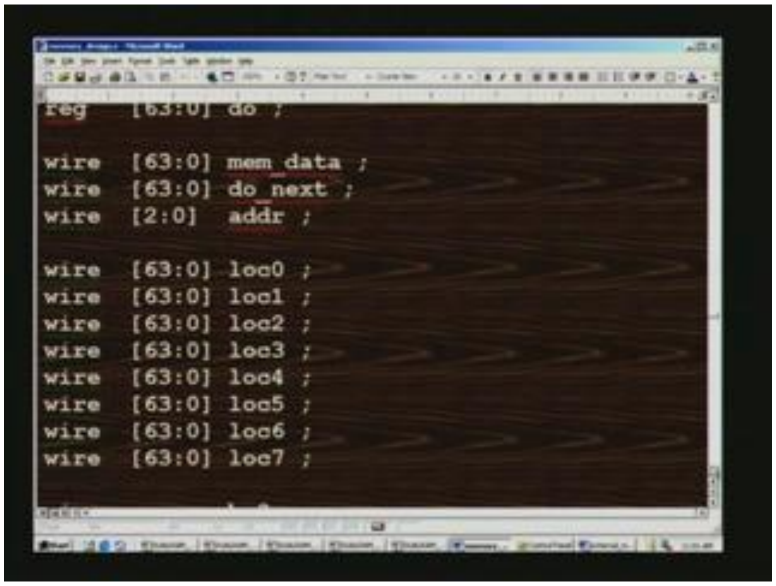
(Refer Slide Time: 20:41)



```
input  [63:0] di ;

output [63:0] do ;

reg   [63:0] mem [7:0] ; // Declare register
                         // array, 8x64 bits.
reg   [63:0] do ;

wire  [63:0] mem_data ;
wire  [63:0] do_next ;
wire  [2:0]  addr ;

wire  [63:0] loc0 ;
wire  [63:0] loc1 ;
wire  [63:0] loc2 ;
```

Now we have this special command called reg mem. We have seen before that we can have a RAM of size 64-bit and location I mean 0 through 7 this being the highest order: location 0, location 1, location 7 and so on. We have seen earlier in the ROM design. The same thing can also be used in RAM design and here the difference is it was read only there in ROM and here it will be both read as well as right and do is the final output for the RAM. It is basically a register array of 8 into 64-bits; that is the meaning of this statement.

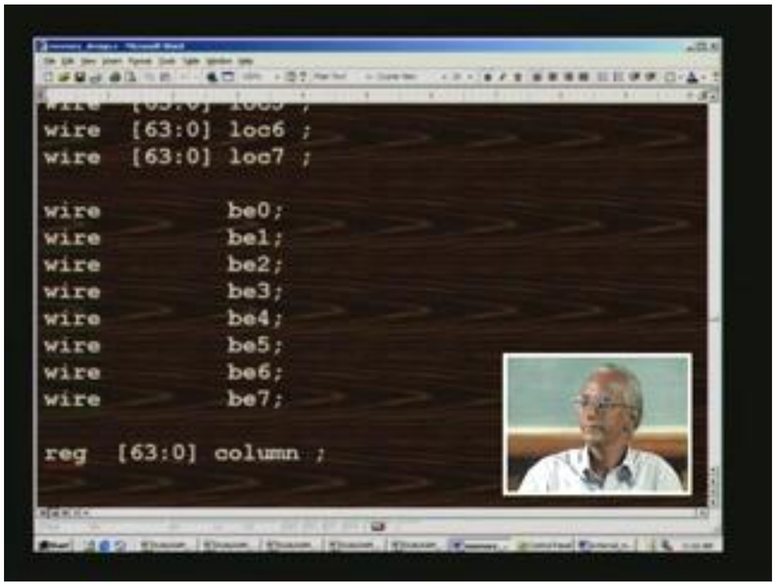
(Refer Slide Time: 21:29)



```
reg   [63:0] do ;

wire  [63:0] mem_data ;
wire  [63:0] do_next ;
wire  [2:0]  addr ;

wire  [63:0] loc0 ;
wire  [63:0] loc1 ;
wire  [63:0] loc2 ;
wire  [63:0] loc3 ;
wire  [63:0] loc4 ;
wire  [63:0] loc5 ;
wire  [63:0] loc6 ;
wire  [63:0] loc7 ;
```

As in ROM, we are using a lot of internal signals of different widths, that is what we have listed here and address is also either read address or write address. A common address is invoked here and width is 3-bits, because we need to address only eight locations and as in ROM design we will have location 0 through location 7 here and they are all combinational circuit. Therefore, they have been declaring as wire and width obviously 64-bits here.

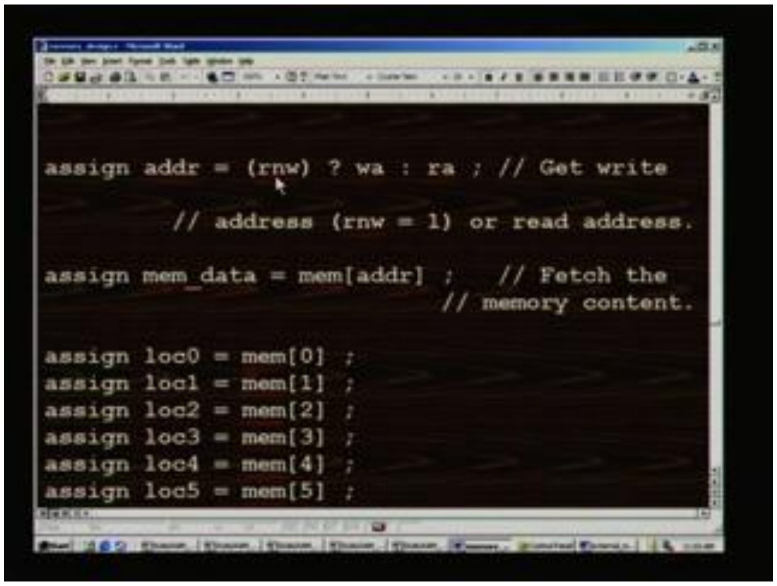
(Refer Slide Time: 22:07)



```
wire  [63:0] loc6 ;
wire  [63:0] loc7 ;

wire         be0;
wire         be1;
wire         be2;
wire         be3;
wire         be4;
wire         be5;
wire         be6;
wire         be7;

reg  [63:0] column ;
```

Then we also have signals derived from 7 through 0, we have seen earlier, that is different and from this one, this is local and there is more to this here. They are all using assign statements therefore, they are declared as wire and we also want to access column-wise, we want to read column. What we do is, we have to have another signal for column and that should be 64-bits and that is in an always block therefore, it is declared as reg. Now let us see how to get address. We have seen before, we have an address; this is nothing but a write address or read address of the RAM depending upon the rnw value. This is the MUX implementation here if rnw is 1 naturally write address is put into the address, otherwise read address and get right address rnw equal to 1 or read address that is precisely what the comment is saying. We have another signal called memory data and this is nothing other than reading that reg mem. That memory is being read here with addresses as addr and this address would depend upon rnw.

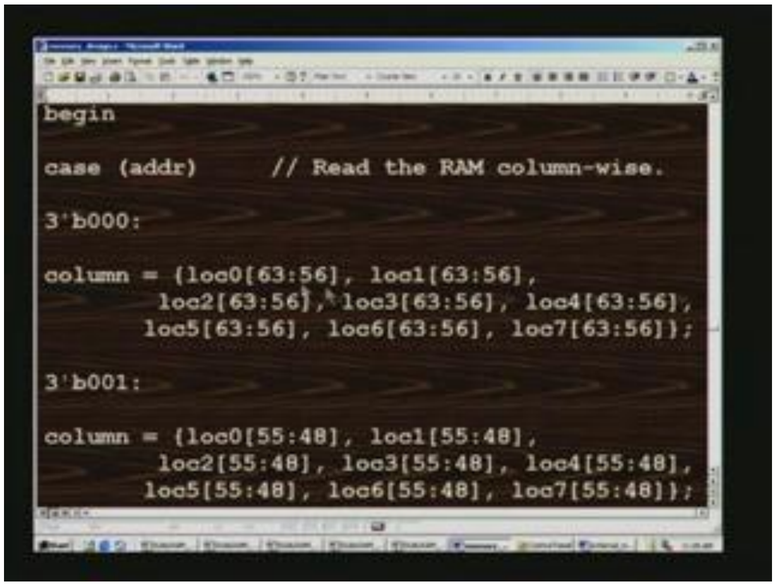
(Refer Slide Time: 23:06)



```
assign addr = (rnw) ? wa : ra ; // Get write
         // address (rnw = 1) or read address.

assign mem_data = mem[addr] ;   // Fetch the
                              // memory content.

assign loc0 = mem[0] ;
assign loc1 = mem[1] ;
assign loc2 = mem[2] ;
assign loc3 = mem[3] ;
assign loc4 = mem[4] ;
assign loc5 = mem[5] ;
```

It could be a write address or read address and fetch the memory content. That is what it says here. We fetch independent locations, this is basically a location 0; therefore we put it as location 0. Assign statements have been used. LOCs have been declared earlier as wire and you have eight locations 0 through 7 and they are nothing other than reading the actual content from memory straight away and this is the address here. This is how you invoke the address and we have an always block here and this for reading column-wise. In fact, I should have put the other way, I should have started with the write. But it does not really matter, in verilog all are concurrent statements, which way you put doesn't matter. Even now I can change it if I want to. We will consider the reading first so how a column is being read we will have a look. Here we see that address is the one which gives the read address or write address and you can use a case here in order to take actions depending upon what address is encountered. We are in the read only mode; because what we are going to do is we want to read the RAM column-wise. We should read only when there is a change of any of these parameters, that is why we have put an always loop and mind you this is still a combinational path only. Only when there is a positive edge of the clock it will become register sequential then.

(Refer Slide Time: 24:58)

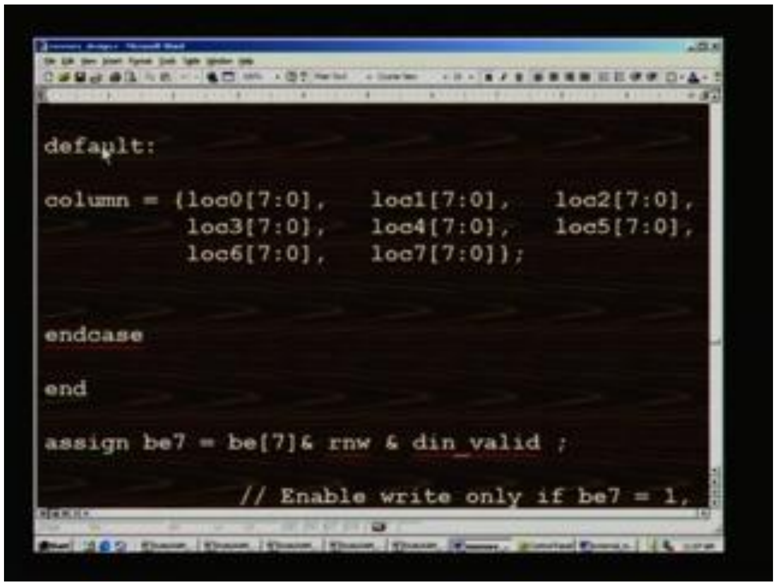


```
begin

case (addr)     // Read the RAM column-wise.

3'b000:

column = {loc0[63:56], loc1[63:56],
        loc2[63:56], loc3[63:56], loc4[63:56],
       loc5[63:56], loc6[63:56], loc7[63:56]};

3'b001:

column = {loc0[55:48], loc1[55:48],
        loc2[55:48], loc3[55:48], loc4[55:48],
       loc5[55:48], loc6[55:48], loc7[55:48]};
```

In the first case, what we do is we have seen location 0 is the first one with the MSB first and the LSB towards the end, because total is 8 bytes that is the 64-bits. We want to read column so what we should do is go to the first location 0, read the top most byte and go to the location 1 that is vertically now. Once again read the top most byte and keep repeating this and concatenate all this here. You have a total of 8 into 8, location 0 through 7 is 8 and each is 8-bits. 6 all the 64-bits are intact, that is now row-wise it is put, but actually it has been read column-wise from the RAM while writing it will be done row-wise, which we will be seeing later. This column gets this value for 0 0 address. If the address is 0 0 it has to get from location 0 onwards but other locations are also involved because it is column-wise reading.

That location 0 etc., have lost its meaning because of that and here for address 1 what you need to do is exactly say all the locations are involved, but next byte is involved. 56 is the last one so here, it will start from 55 to 48 next 8-bits and likewise for different addresses it will be different. And that is the reason why that was put row-wise like this we read column-wise. Shortly, we will come to that. For different addresses, it is precisely the same thing; each time only the bytes keep changing to lower and lower order. Finally, when you strike say 1 1 1 1 for address that the last one, you see that 7 through 0 which is the last, 63 to 0 was the one and 7 in last byte being accessed. This shows that, we are accessing vertically. Default is also covered, you can either put a dummy statement or put some relevant statement there and case ends here and there was an always block actually.

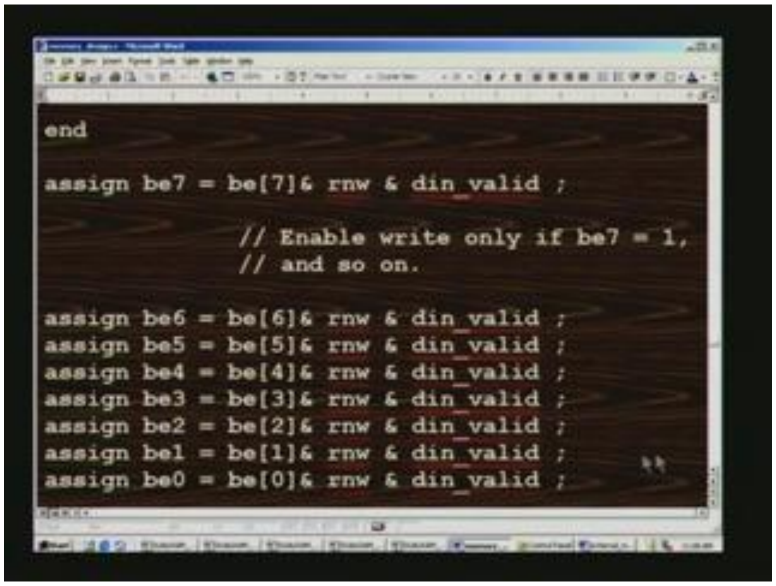
(Refer Slide Time: 27:37)



```
default:

column = {loc0[7:0],   loc1[7:0],   loc2[7:0],
          loc3[7:0],   loc4[7:0],   loc5[7:0],
          loc6[7:0],   loc7[7:0]};


endcase

end

assign be7 = be[7]& rnw & din_valid ;

              // Enable write only if be7 = 1,
```

Begin and end here and now we are going to see how to write. Prior to writing, this writing is going to happen through the PCI bus.

(Refer Slide Time: 27:45)

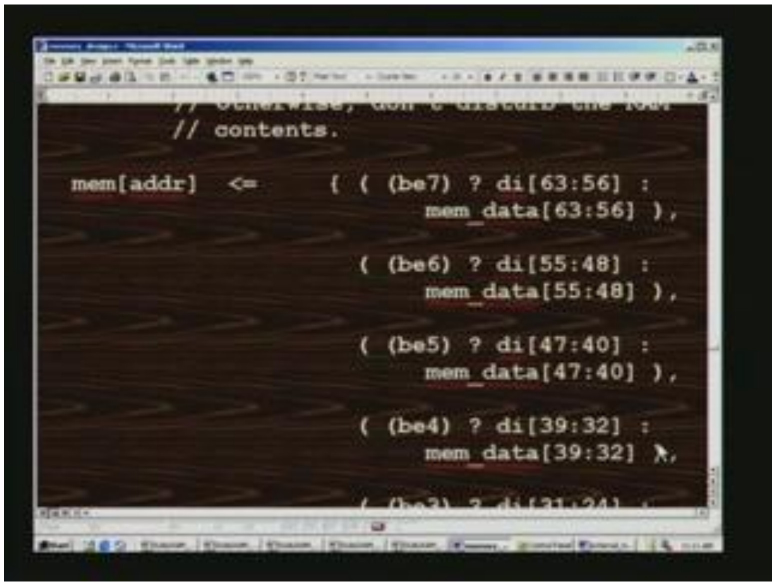


```
end

assign be7 = be[7]& rnw & din_valid ;

              // Enable write only if be7 = 1,
              // and so on.

assign be6 = be[6]& rnw & din_valid ;
assign be5 = be[5]& rnw & din_valid ;
assign be4 = be[4]& rnw & din_valid ;
assign be3 = be[3]& rnw & din_valid ;
assign be2 = be[2]& rnw & din_valid ;
assign be1 = be[1]& rnw & din_valid ;
assign be0 = be[0]& rnw & din_valid ;
```

Another PCI master such as a video grabber we have seen earlier. I will be writing into this core straight away and this will be housed in the video codec that we have seen in the earlier example. In this what we are going to do is in we have also seen in the block diagram at the start that be 7 through 0 being used in the as part of the PCI signal, this is the PCI signal straight away. Now we are going to generate from this, almost similar looking name but the difference is, it implies that particular bit and this is derived from and in rnw as well as din_valid, rnw means it is equal to 1 which means it is in write mode. Whenever we are in right mode only then this should be computed and all of them are assign statements therefore they have been declared as wire earlier and data is the one, we input the data and corresponding valid signal is this one. Unless the signal is valid and unless it is in right mode we should not write into that and this is just a preparation merely a byte enable only done in this fashion.

In case you do not have higher order address from the PCI bus not decoded, you can decode that and include in this din_valid. Enable write only if be 7 equal to 1 and so on, that is the implication there. If it is reverse, we may have to reverse the logic here, so we have only to put 1 inverter at the appropriate place. Any last minute changes can be very readily incorporated in verilog that way and same is the case for other bes here and except that different byte enables are selected for that. In order to write we need to look into the positive edge of the PCI clock and this PCI clock as I mentioned if it is PCI x bus it will be hundred megahertz. Other PCI bus will be operating at 66 megahertz or thirty three megahertz and it can be 32-bit or 64-bit and PCI x comes I think in 64-bits and we have taken 64-bits. We have seen bus arbiter giving over 250 megahertz operation and these designs may not give that. Ultimately what counts is the overall speed when we come to the total design and here in this particular writing what we do is write into RAM only if be 7 equal to 1 and so on otherwise do not disturb the RAM contents. Now data is valid here and it is in write mode and now what we need is the address.

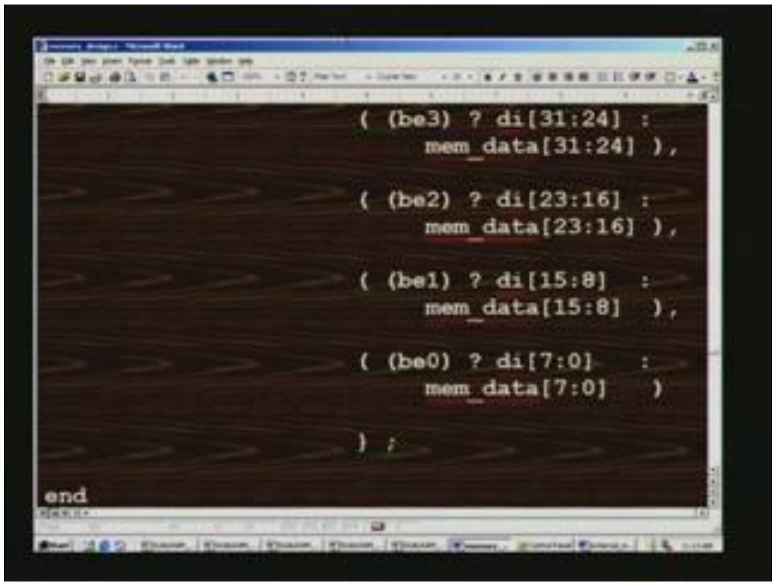
(Refer Slide Time: 30:47)



```
        // Otherwise, don't disturb the last
        // contents.

mem[addr]  <=     { ( (be7) ? di[63:56] :
                          mem_data[63:56] ),

                      ( (be6) ? di[55:48] :
                          mem_data[55:48] ),

                      ( (be5) ? di[47:40] :
                          mem_data[47:40] ),

                      ( (be4) ? di[39:32] :
                          mem_data[39:32] ),

                      ( (be3) ? di[31:24] :
```

Address is write address because rnw selects be 7 only when it is in write mode. Evidently this address is a write address, wa. Memory write address is going to be affected by these contents. What are the contents? We are going to write in to this memory; we are trying to do this. PCI write, and in 8 clock pulses you would have written 64 bytes whereas in DCTQ it will take 64 clock pulses to process. It means writing time is very high and this may have to be done from the video grabber and that means you have lots of time for the video grabber and because processing is going to take quite some time. What all we have here is, we had to read the di inpu,t this is the PCI data bus and we had to read the last higher order byte and provided that higher order be 7 is enable. If it is 1 then only this will be taken and assigned to memory whose address is pointed by this address. If this is not satisfied, especially for read operation this won't be satisfied, then also this will be executed. We have to take enough care for that, so what we do is simply write itself back to it, so this and this are precisely the same. You write the very same data from this and write it back to the same byte, so that means you do not disturb the contents because it is in read mode, so it is taken care of. You have to be extremely careful while designing systems and if you forget this one, naturally functioning will get affected and so is the case for all the byte enables till the variant the last byte here.

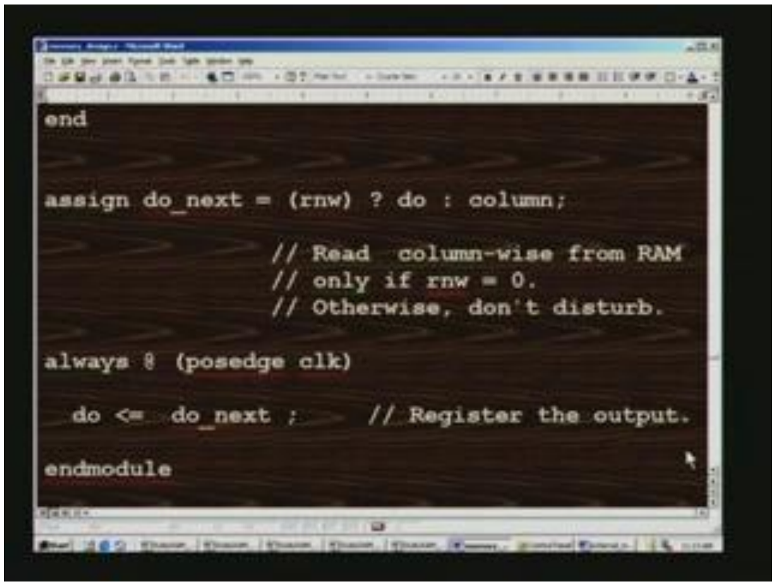
(Refer Slide Time: 32:42)



```
                    ( (be3) ? di[31:24] :
                        mem_data[31:24] ),

                    ( (be2) ? di[23:16] :
                        mem_data[23:16] ),

                    ( (be1) ? di[15:8]  :
                        mem_data[15:8]  ),

                    ( (be0) ? di[7:0]   :
                        mem_data[7:0]    )

                    } ;

end
```

The whole thing has been concatenated and separated by commas, so the whole thing what you have here is 64-bits in totality.

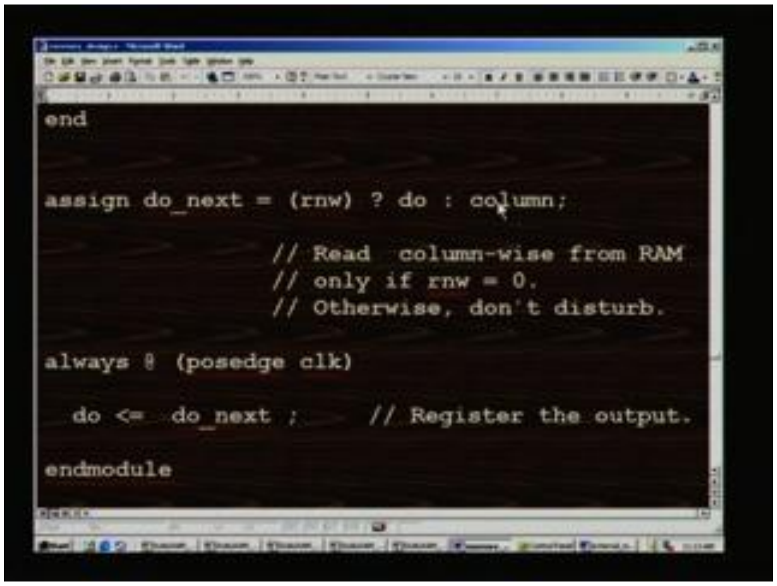
(Refer Slide Time: 32:57)



```
end

assign do_next = (rnw) ? do : column;

                // Read  column-wise from RAM
                // only if rnw = 0.
                // Otherwise, don't disturb.

always @ (posedge clk)

  do <=  do_next ;     // Register the output.

endmodule
```

Once again here we need to output this we are talking of the sub module RAM underscore RC and the corresponding output is a still a combinational output only, that is assigned from this column. We have seen this column earlier, we are reading column-wise, how to read we have already seen. What we have read and put in column earlier now we are going to output here to d not next and that is provided rnw is low so low corresponds to this one.

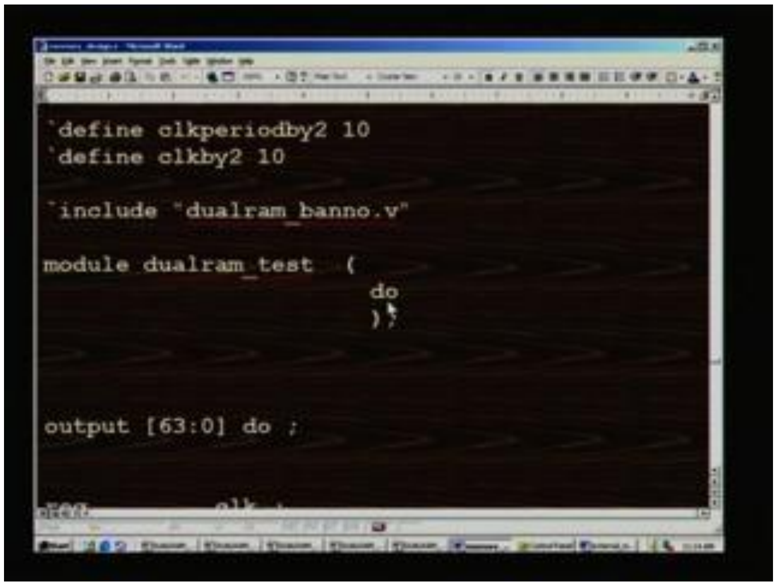
(Refer Slide Time: 32:57)



```
end

assign do_next = (rnw) ? do : column;

                 // Read  column-wise from RAM
                 // only if rnw = 0.
                 // Otherwise, don't disturb.

always @ (posedge clk)

  do <=  do_next ;     // Register the output.

endmodule
```

If it is in write mode naturally do will be taken as do_next so both its catering the statement is catering to both read as well as right. Read column-wise from RAM only if rnw is equal to 0 that is what we have said, otherwise do not disturb. Finally the underscore RC at positive edge of clock we need to just push d not next into the actual output d not, this is the output that goes straight away as we have seen in the block diagram which we started. This is the test bench for dual RAM design and as usual we will operate at 50 megahertz and there is one more clock here. One of these is PCI clock and the other one is for the system clock, forget which is what we will have a look at it a little later.

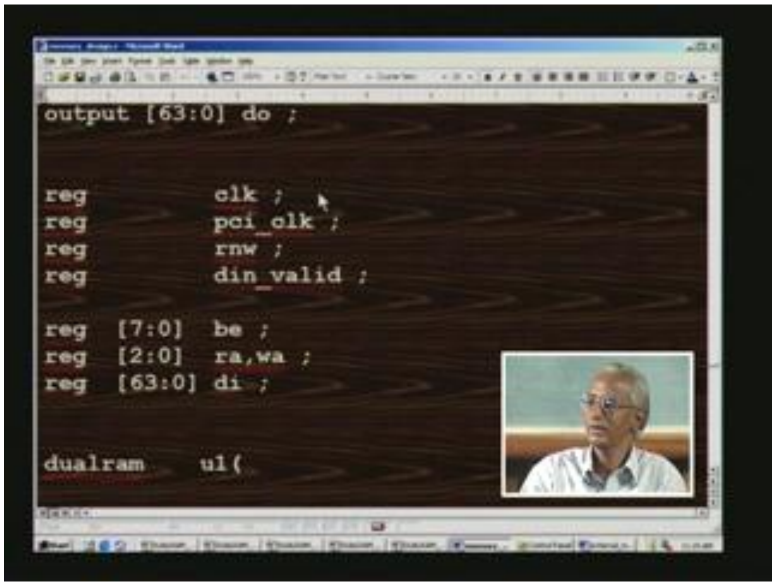
(Refer Slide Time: 34:22)



```
`define clkperiodby2 10
`define clkby2 10

`include "dualram_banno.v"

module dualram_test  (
                        do
                        );



output [63:0] do ;
```

We need to include back annotated file and that is why we have put dual RAM underscore banno dot v. This is the test bench and we are naming the module as such and we need to examine only the output here. In fact, in waveform we can have a look at all other signals and as for as test bench is concerned this is adequate and width we are declaring.

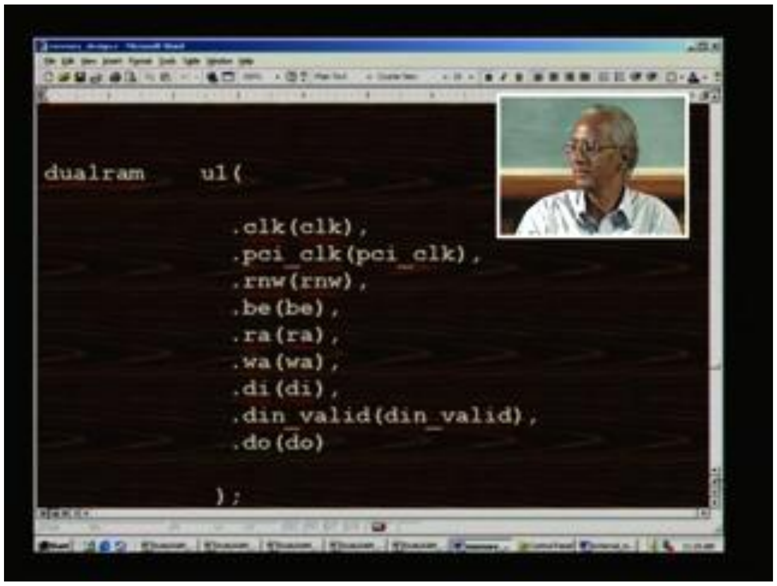
(Refer Slide Time: 34:48)



```
output [63:0] do ;


reg          clk ;
reg          pci_clk ;
reg          rnw ;
reg          din_valid ;

reg   [7:0]  be ;
reg   [2:0]  ra,wa ;
reg   [63:0] di ;


dualram    u1(
```

We are listing all the I/O s still need to be identified as reg for inputs clock PCI clock rnw they are all inputs. This be etcetera read write are all generated from this design, so you need to declare it as, I am sorry these are all inputs still. They are all inputs to this design right so all of them are inputs so you need to declare as reg and only one output is d not that has been declared there.

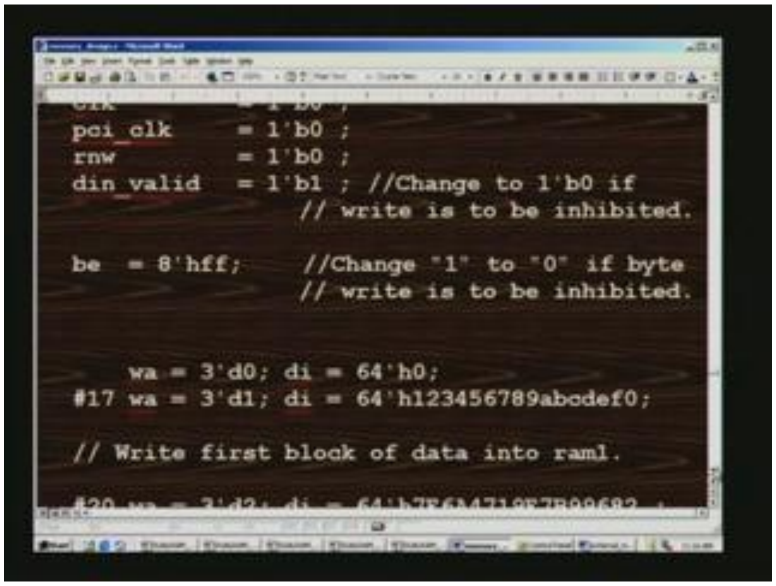
(Refer Slide Time: 35:29)



```
dualram    u1(

                .clk(clk),
                .pci_clk(pci_clk),
                .rnw(rnw),
                .be(be),
                .ra(ra),
                .wa(wa),
                .di(di),
                .din_valid(din_valid),
                .do(do)

               );
```

We invoke the actual design and instantiate it here and only one instantiation is used and these are all the listings of all I/Os that we have already seen. As usual we start initializing with clock PCI clock to zeroes and rnw in 0, so implying that what does it mean which RAM is being accessed you have to keep track. If it is 0 it means read mode here and why should it be read mode and data invalid is. How can this be read mode? It appears to be contradictory but let me remind you, it is not a dual port RAM but it is dual RAM. This is different, so one RAM is in write mode, the other is in read mode, so which RAM we can make a guess from this. This is a control to put one RAM in write mode and to put the other RAM in read mode that is what it means. This clearly implies it is in read mode and what will be affected you have to see.

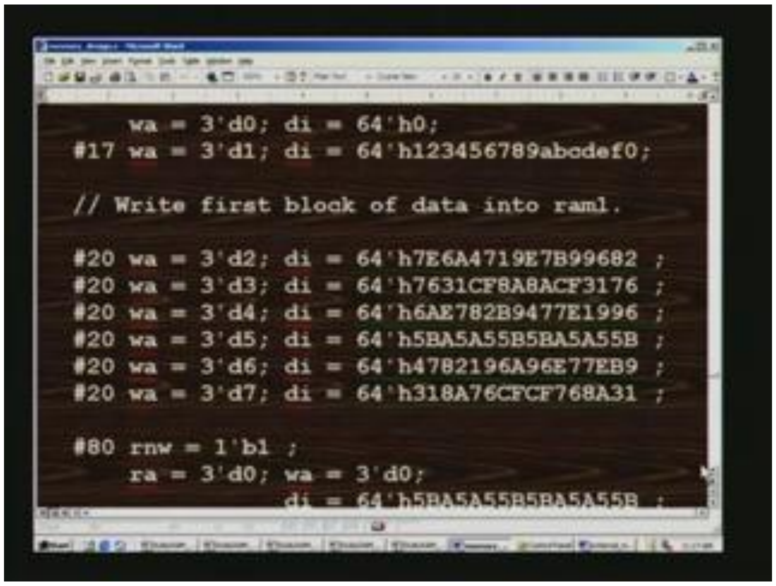
(Refer Slide Time: 36:42)



```
pci_clk     = 1'b0 ;
rnw         = 1'b0 ;
din_valid   = 1'b1 ; //Change to 1'b0 if
                     // write is to be inhibited.

be  = 8'hff;         //Change "1" to "0" if byte
                     // write is to be inhibited.


    wa = 3'd0; di = 64'h0;
#17 wa = 3'd1; di = 64'h123456789abcdef0;

// Write first block of data into ram1.
```

Suppose you want to inhibit the write, what all you have to do is make it 0, so want to test out other possibilities you can do all these. Once again, you can change bit by bit change 1 to 0 if byte write is to be inhibited. I have seen all these and all of the functions are working and you will have to run every time you make a change and now the data comes here.

(Refer Slide Time: 37:09)

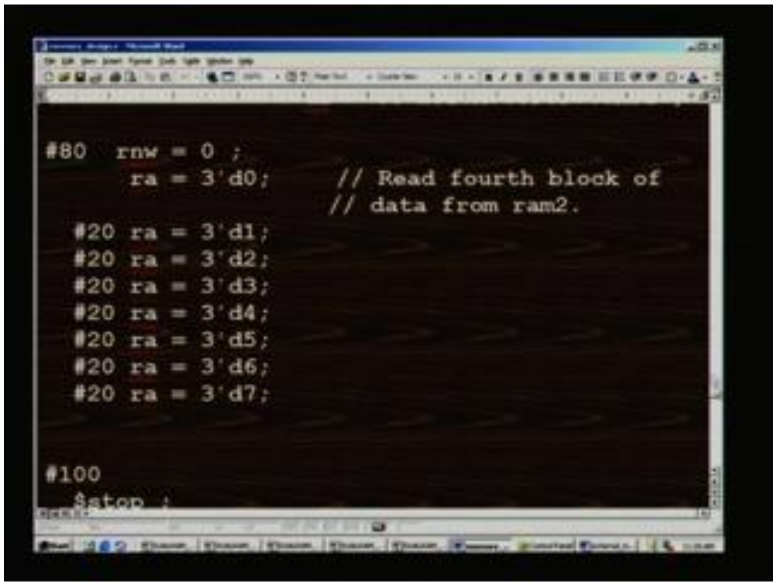


```
    wa = 3'd0; di = 64'h0;
#17 wa = 3'd1; di = 64'h123456789abcdef0;

// Write first block of data into ram1.

#20 wa = 3'd2; di = 64'h7E6A4719E7B99682 ;
#20 wa = 3'd3; di = 64'h7631CF8A8ACF3176 ;
#20 wa = 3'd4; di = 64'h6AE782B9477E1996 ;
#20 wa = 3'd5; di = 64'h5BA5A55B5BA5A55B ;
#20 wa = 3'd6; di = 64'h4782196A96E77EB9 ;
#20 wa = 3'd7; di = 64'h318A76CFCF768A31 ;

#80 rnw = 1'b1 ;
    ra = 3'd0; wa = 3'd0;
```

Follow this carefully. We will also have a look at the waveforms and what we are going to do is first we take one of the RAMs let us say RAM1, RAM1 is selected and we are going to write into this RAM1 . That means rnw is equal to 0 has pointed to RAM2, that means that is in read mode. But very first block of image that you are going to write, there is nothing available in the other RAM. There is no pointing in reading there. We do not have to read concurrently, that is why we are merely writing the data there. Just remember this, 0 implies all the 64-bits here; instead of putting so many zeroes I just confined to one. Just remember there are zeroes here and next it is easy for you to remember this data 1 2 3 4 all in progression. For each time there are 0 through 7 locations and every time, because it is 50 megahertz operation, we have made twenty nanosecond change.

Staggering is done just to offset, as we have seen before and next time what we do is we write the next block. Now, we will write it into RAM2 and next block of different block of data is being written into and note the difference here. While you are writing, this is right for the RAM2, we are also reading simultaneously from RAM1, that is the implication there. rnw is also changed here, earlier it was 0 now after having written into the first block, we are going to write into second block therefore we should not forget this rnw control. Actually in DCTQ there will be a controller which will do this automatically so here we have to force it in the test bench. You have second group of data here, so from this point of time onwards you have both write as well as read and then you have one more block here, probably there is one more block

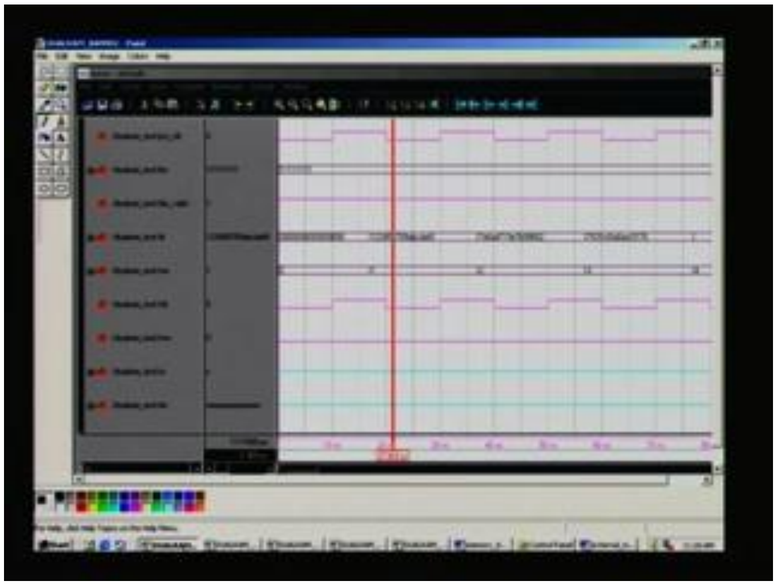
(Refer Slide Time: 39:42)



```
#80  rnw = 0 ;
      ra = 3'd0;      // Read fourth block of
                     // data from ram2.
  #20 ra = 3'd1;
  #20 ra = 3'd2;
  #20 ra = 3'd3;
  #20 ra = 3'd4;
  #20 ra = 3'd5;
  #20 ra = 3'd6;
  #20 ra = 3'd7;


#100
  $stop ;
```

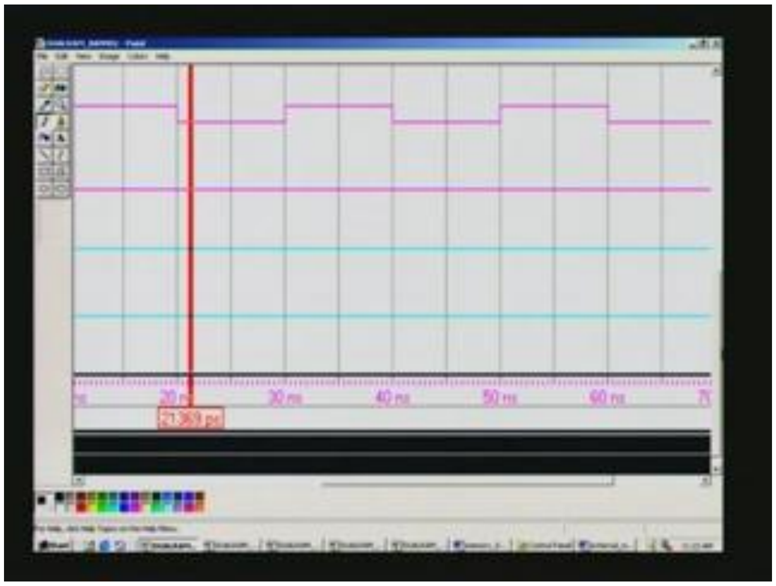
In order to see all the blocks, it will be difficult, so we will see some representative. Although I have all the waveforms, we will see some representative things. First let us go to the very first block we have written, 0 0. Remember while writing we are writing in this fashion row-wise that is horizontally while reading we should read vertically. For example, this is the first byte 0 0 then 1 2 7 e and so on and the next must be 0 0 3 4 and so on. That is how we have to read, let us see whether the waveform conforms to that.

(Refer Slide Time: 40:23)



First waveform is: We have a PCI clock here and byte enable all of them are forced to ff there, that means it is all ones here. You have a d din_valid asserted so 1 here and data is changing at every twenty nanosecond and here. To start with it was 0 there, so that is precisely what has appeared here and then we had written 1 2 3 4 up to f 0, that is the second one. Third was 7 e 6 a and so on and we have some more here, that is a right address, it is showing 0 here 1 2 3 4, that is what I have forced in the test bench and no read address here because we are writing the first block and there is a system clock here. DCTQ operates or dual RAM operates on this clock especially for the read mode and we have a rnw control here it is precisely 0 here that is what we have done in the test bench and d not output will come only later on, while writing the next block of data, then we will simultaneously read so after 8 clock pulses are so plus pipelining may be 10 clock cycles only you may start getting it and I will zoom this. We have PCI clock be and so on and this is the data in, you can see that 0 0 all these data and times axis here.

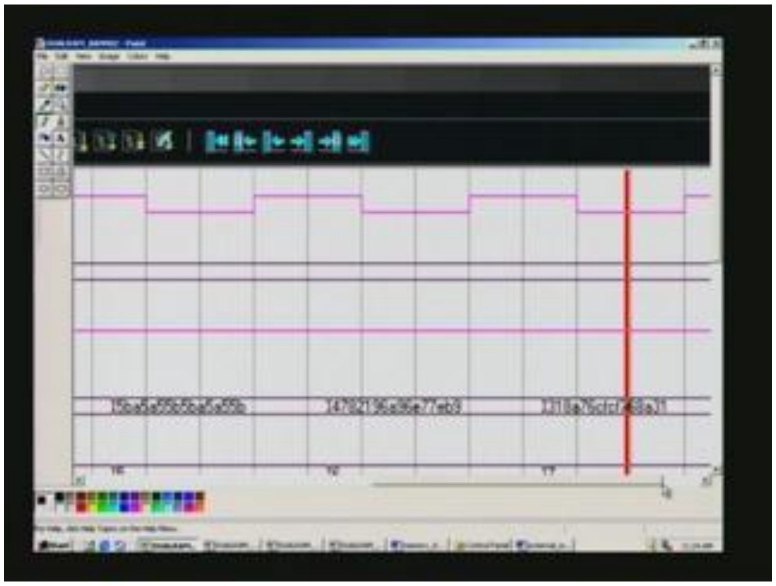
(Refer Slide Time: 42:12)



Read address d not output, write address and some of this data if you want you can cross check may be 1 or 2 here, so 7 e 6 a and so on, 7 6 3 1, 7 e 6 a 7 6 3 1. What you have written into it has gone into the RAM that is what you have seen here so we do not require this. Next set of data is 5 b a 5 and so on; let us have a look at that. Here is what we get 5 b a 5 then 4 7 8 2 5 b, 4 7 8 2 and so on. The next block is being written into now let us have a look at the vertical reading whether it is happening or not. This is the first block, you should read 0 0 1 to 7 e when we read;

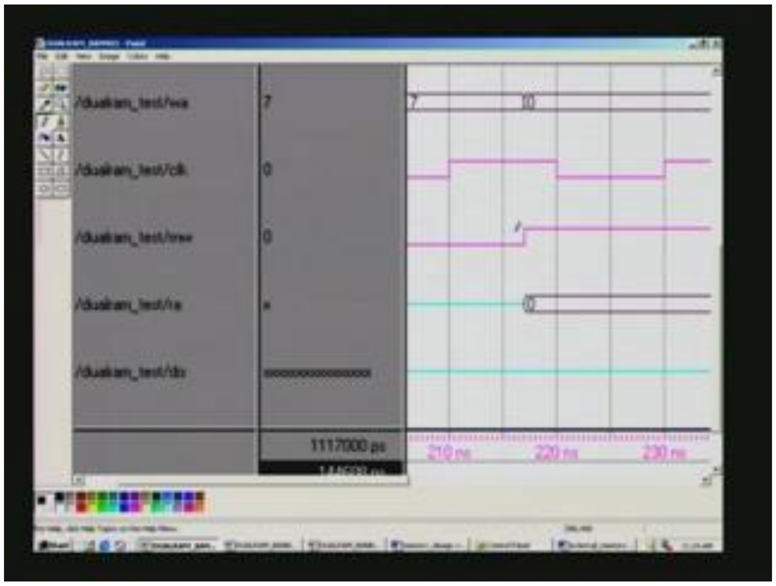
so let us have a look. I will zoom it later on. It is the second waveform also does not cover that, you can see the data there.

(Refer Slide Time: 43:46)



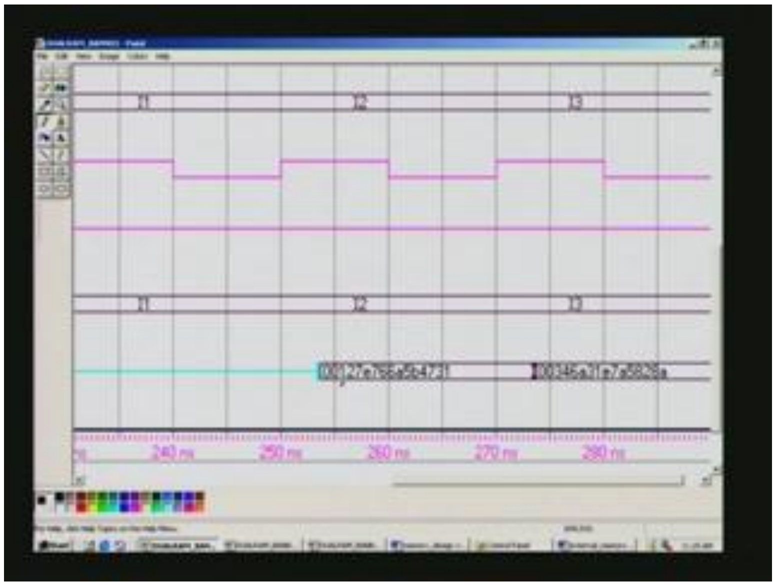
The third one is for the read, now comes the first read. While this is happening it is continuing to be written, both are concurrent as I mentioned. Now you see that read address has become active and rnw has switched over from 0 to 1 and this is the system clock and you can see the very first data here and I will zoom it, so that you can read. You can see here, so continue it to write here with a write address changing, so it is going to the next block writing and concurrently you can see that rnw switching from 0 to 1 to write into the second RAM.

(Refer Slide Time: 44:40)



Concurrently to read from the first RAM which has been filled prior to and d not output is this one read address and this. Let us have a look here so the output is here. Let us check whether this data is conforming or not.

(Refer Slide Time: 45:03)



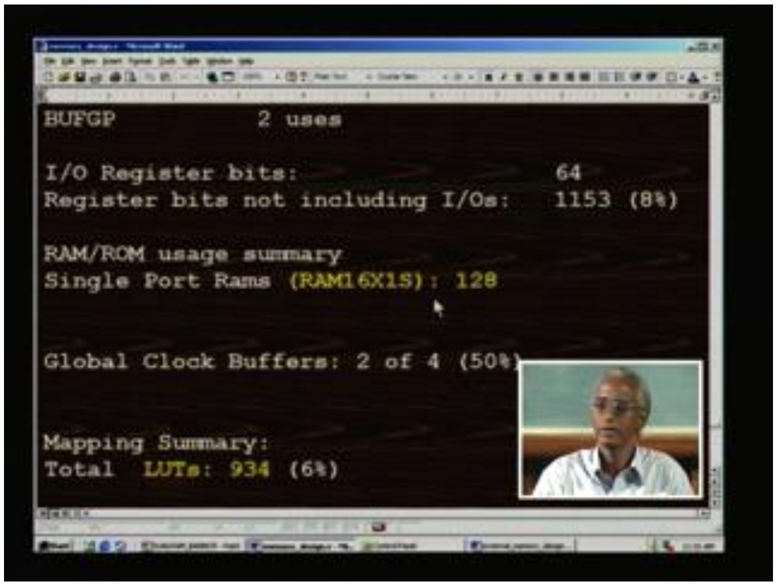
For example we have to read column-wise so just remember some of this 0 0 1 2 7 e 7 6. The last one is 4 7 3 1, so in this fashion you can see, next one will be 0 0 3 4 6 a, last one is 8 a. It is reading column-wise. We can see two more to inspect and it will be from different blocks. The only thing is we will have difficulty in identifying the blocks. What you have here is 0 0 f 0, you can see that, that is what we have, b 9 3 1 the last one. It is b 9 3 1 here, so you have seen 0 0 f 0 b 9 3 1, that is what it is and one more waveform it is precisely same. I think it is the very last block it may have a look, so let us just read the last one a 5 1 9 etcetera This is the one last data, we have to go to the fag end of the blocks, so that is somewhere here.

Finally, what we are doing is we are actually reading the data that we have written in the previous block. You can see this this one 0 0 f 0 8 2 and just remember that, this one f 0, 0 0 here there is one more there up that is 0 0 0 f 0 8 2 7 6, same. Let us go and see what is happening here. May be it has started writing the next block at some data. By enlarging you can see that it is there clearly. How is this dual RAM different from by getting to separate RAM chips and making use of it? Even there how will you access vertically? Suppose the organization is 8 into 64-bits? Suppose you have used a conventional memory of 8 into 64 bits that will be an only 1 address bus.

When you give address it will merely access row-wise and give you the thing all memories are designed in that fashion while reading; whereas, our requirement is while writing it should be horizontal while reading it should be row-wise. You can imagine it as a matrix so we are writing row-wise and I can just see. Say we are writing row-wise this order so that means 64-bits are written straight away there. While reading we need the conventional memory you can give a read you will get only this row-wise, whereas our requirement is not row-wise but vertically this is because of the algorithm that I have developed for DCTQ which demands this application. I cannot use the conventional memory and that is the reason why I went for this specialized design for this.

Let us have a look. We had two clocks, so they had to be taken care and that finishes the test bench. Let us have a look at the simplified results here. This is the dual RAM design so clock; this is the system clock it operates at 153 megahertz and PCI clock is 133. You can safely go to PCI hopefully. Let us not jump to a conclusion right now, because we need to see the back annotated results which will be got only in the XILINX place and route. The device has as usual been the same; namely, XCV 600, higher speed available there and this is the distribution of various buffers etc., and here in FPGA they have special RAMs available. 16 into 1, has been pressed into service and 128 such RAMs have been used in this design. We did not specifically mention this, but automatically the tool has done it.

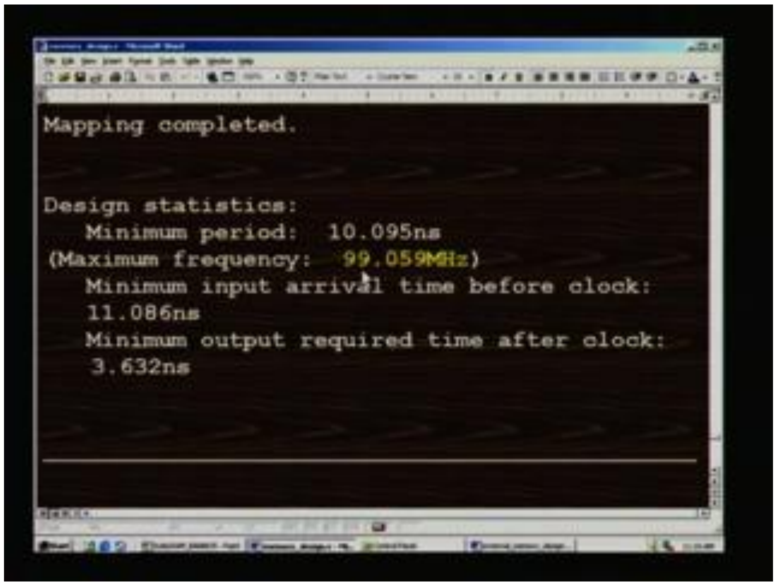
(Refer Slide Time: 51:23)



```
BUFGP           2 uses

I/O Register bits:                    64
Register bits not including I/Os:    1153 (8%)

RAM/ROM usage summary
Single Port Rams (RAM16X1S): 128


Global Clock Buffers: 2 of 4 (50%)


Mapping Summary:
Total  LUTs: 934  (6%)
```

In terms of LUTs, it takes 934 LUTs and it is only the 6 percent of this 600,000 gate. This RAM is over and above those LUTs and place and route results, once again number of slices etc., are given. Also here, it reports the same 128 RAMs and finally a total gate count is it is quite a high value. We have been so for considering only the 2000 - 3000 gates write up to the ROM design and now we have made a big leap, so because of the complexity involved the design. It takes around 40,000 gates. I am sure you would have gone through digital course, you may be aware of the package 74 LS 0 0 a NAND gate packed in a 14 pin dip. Each of this, if you look into the transistor of organization, you will see that 5 transistors have been used there. If you take 5 transistors per 2 input NAND gate and you take that as a standard, you will get enormous value there, 150,000 transistors, I mean 40 into 200000 transistors.

(Refer Slide Time: 52:48)



```
Mapping completed.

Design statistics:
   Minimum period:  10.095ns
 (Maximum frequency:  99.059MHz)
   Minimum input arrival time before clock:
   11.086ns
   Minimum output required time after clock:
    3.632ns
```

What crosses 50,000 is categorized as a VLSI and that is how it is and the maximum frequency operation is 99 megahertz. We are just falling from magical figure of 100 megahertz. Thank You.