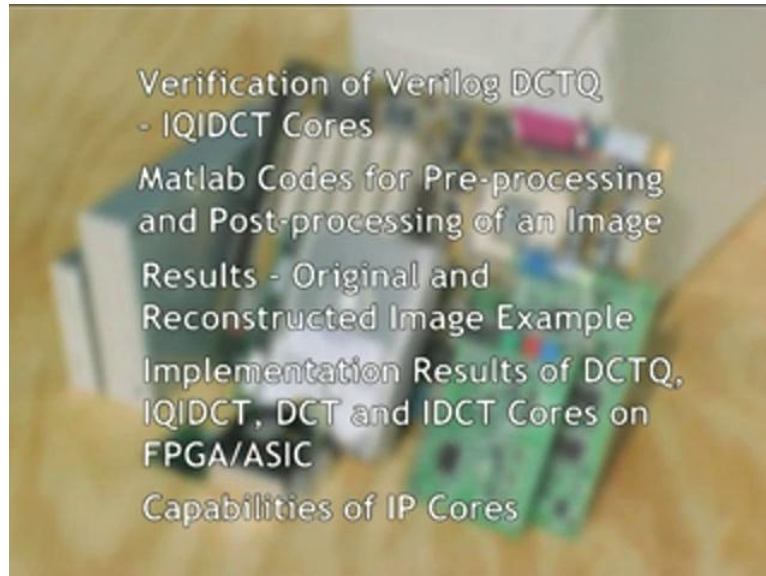


Digital VLSI System Design
Dr. S. Ramachandran
Department of Electrical Engineering
Indian Institute of Technology, Madras

Lecture No. 47
System Design Examples (Continued)

(Refer Slide Time: 01:27)





We will continue with the Verilog DCTQ controller design.

(Refer Slide Time: 02:33)

```
end

assign swon_ready = ((start_reg1 == 1'b1)&&(cnt1_reg ==
                    6'd01)) ? 1'b1 : 1'b0;

always @ (posedge clk or negedge reset_n)
begin
  if (reset_n == 1'b0)
    ready <= 1'b1;
  else if (hold == 1'b1)
    ready <= ready ;
  else if (swon_ready)
    ready <= 1'b1;
  else
    ready <= !start_reg1 ;
end
```

We stopped last time at the ready signal. The ready signal is turned on only if this condition is met. This condition is given here – as long as the start is asserted and we have the counter as 01 here. This implies that we have already processed DCTQ for the previous block. We have already started the current block just at the beginning. At this stage, we can allow this host to write data into the other RAM whose value has been already processed. That is what we are doing at this end only when this condition is satisfied. If this is not satisfied, ready will get this inversion of start_reg, for example, if it is 1, it will not make it ready. This means that there is no start signal.

On the other hand, if start is asserted and count value is something else, then also this will not be satisfied. Therefore, 1 will be inverted and it will continue to be low, for other [04:00] as well.

(Refer Slide Time: 04:03)

```
always @ (posedge clk or negedge reset_n)
begin
  if (reset_n == 1'b0)
  begin
    dctq_valid_prev <= 1'b0 ;
    dctq_valid <= 1'b0 ;
  end

  else if (hold == 1'b1)
  begin
    dctq_valid <= 1'b0 ;
  end

  else if (cnt1_reg == 6'd44) // dctq is valid from
```

Here, we produce the dctq_valid signal. In order to have this, we need one more signal and we will see why. During reset, we reset both these signals.

(Refer Slide Time: 04:20)

```
begin
  dctq_valid_prev <= 1'b0 ;
  dctq_valid <= 1'b0 ;
end

else if (hold == 1'b1)
begin
  dctq_valid <= 1'b0 ;
end

else if (cnt1_reg == 6'd44) // dctq is val
// cnt1_reg =
begin
  dctq_valid <= 1'b1 ;
  dctq_valid_prev <= 1'b1 ;
```

If a hold is encountered, dctq_valid must be switched off because the DCTQ processing is held now and so, the valid signal must also be withdrawn. That is why

we do this. You remember that pipeline was 45 before DCTQ starts out. dctq_valid goes high only at this stage – because of this, it is going high and that is for cnt1 being equal to 44. At this stage, we also make this signal 1 for this purpose. For example, after this, it keeps on processing. At that point of time, let us say, the hold comes again. So, what will happen? DCTQ is forced to 0.

(Refer Slide Time: 05:11)

```

else if (hold == 1'b1)
begin
dctq_valid <= 1'b0;
end

else if (cnt1_reg == 6'd44) // dctq is valid from
// cnt1_reg = 44 dec. onwards.
begin
dctq_valid <= 1'b1;
dctq_valid_prev <= 1'b1;
end
else if (hold == 1'b0)
begin

```

```

dctq_valid <= dctq_valid_prev;
end

else
dctq_valid <= dctq_valid;
end

assign start_next1 = (start == 1'b1) && (cnt1_reg == 0);

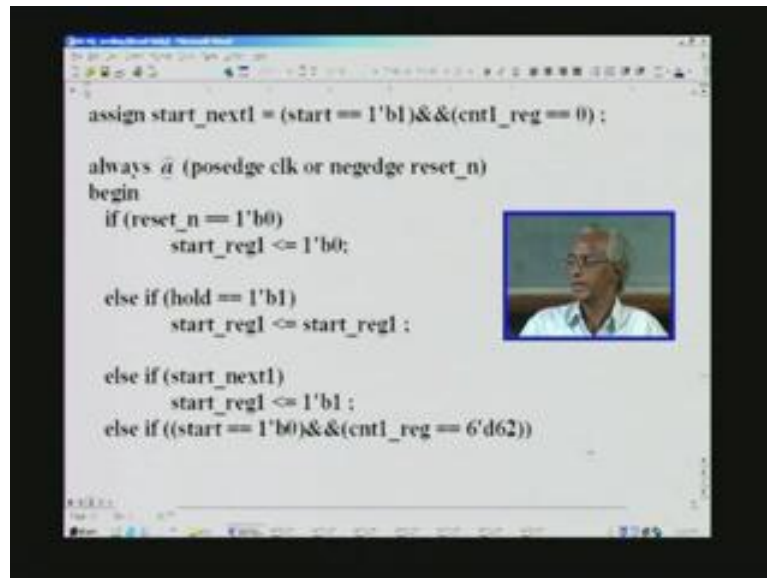
always @ (posedge clk or negedge reset_n)
begin
if (reset_n == 1'h0)

```

Once it is withdrawn, it will come to this stage. What will happen here? Earlier, dctq was 1, but because the hold has come, it has been forced to 0. How to retrieve this when hold is removed and continue from where we left? That is possible only if you make this once again 1 from 0. That is possible by copying these contents into

dctq_valid, which we do at this step. If nothing else, then we just preserve the contents.

(Refer Slide Time: 05:47)



```
assign start_next1 = (start == 1'b1)&&(cnt1_reg == 0);  
  
always @ (posedge clk or negedge reset_n)  
begin  
    if (reset_n == 1'b0)  
        start_reg1 <= 1'b0;  
  
    else if (hold == 1'b1)  
        start_reg1 <= start_reg1;  
  
    else if (start_next1)  
        start_reg1 <= 1'b1;  
    else if ((start == 1'b0)&&(cnt1_reg == 6'd62))
```

The next signal is **start reg**, which we have been using before. For this, the condition is $start = 1$. Mind you, this is not start register or start next, but the actual start input itself. When the start input is applied, it will be sensed only when **cnt1** is 0. Then only, it will assign 1 for this start_reg. That is this condition here. This will take place one clock cycle later. In the next case, suppose start is 0 and then counter is 62, implying that the DCTQ processing is nearing the end, what happens? At that point of time if start goes to 0, then what should happen?

(Refer Slide Time: 06:39)

```
else if (start_next1)
    start_reg1 <= 1'b1 ;
else if ((start == 1'b0)&&(cnt1_reg == 6'd62))

-----

    start_reg1 <= 1'b0 ;
else
    start_reg1 <= start_reg1 ;
end
```

It must reset the start here. That is what it does.

(Refer Slide Time: 06:47)

```
// This is the test bench to test the DCTQ Design.
// Input image frame is lena.txt - change it for processing a
// different image frame.
// dctq.txt is the dctq output of the image frame, lena.txt.
// dctq\_test.v file.

-----

`define clkperiodbv2 5 // Both clocks clk & rci clk
```

This completes the controller module. We have to do the test bench. This is the DCTQ test file. We take a lena.txt as the input file name and final output will be dctq.txt from the same.

(Refer Slide Time: 07:05)

```
'define clkperiodby2 5 // Both clocks clk & pci_clk
                        // operate at 100 MHz.
'define pci_clkperiodby2 5
'define NUM_BLKs 1024 // Defines no. of blocks in a
                      // frame. 256x256 pixel contains
                      //1024 blocks. Change this for a different image size.

'include "dctq.v" //Design module.
```

As usual for the test bench, we run at 100 Megahertz. Therefore, this variable is made 5 and another clock is also made 5. We define a variable called NUM_BLKs to indicate how many blocks there are in 256 by 256 pixels, each block being 64 pixels. From this, you can get this. This is the design module, which we include here.

(Refer Slide Time: 07:30)

```
'include "dctq.v" //Design module.

module dctq_test ;

reg pci_clk ;
reg clk ;
reg reset_n ;
reg start ;

reg [63:0] di ;

reg din_valid ;
```

Then, start the DCTQ test module here. reg denotes all the inputs that we need to declare here.

(Refer Slide Time: 07:40)

```
reg [2:0] wa ;
reg [7:0] be ;


reg    hold;

wire   ready ;

wire [8:0]  dctq : //dctq output

wire      dctq_valid ;

wire [5:0]  addr ;
wire [10:0] eobcnt_next ;
```



```
reg [7:0] be ;

reg    hold;


wire   ready ;

wire [8:0]  dctq : //dctq output

wire      dctq_valid ;

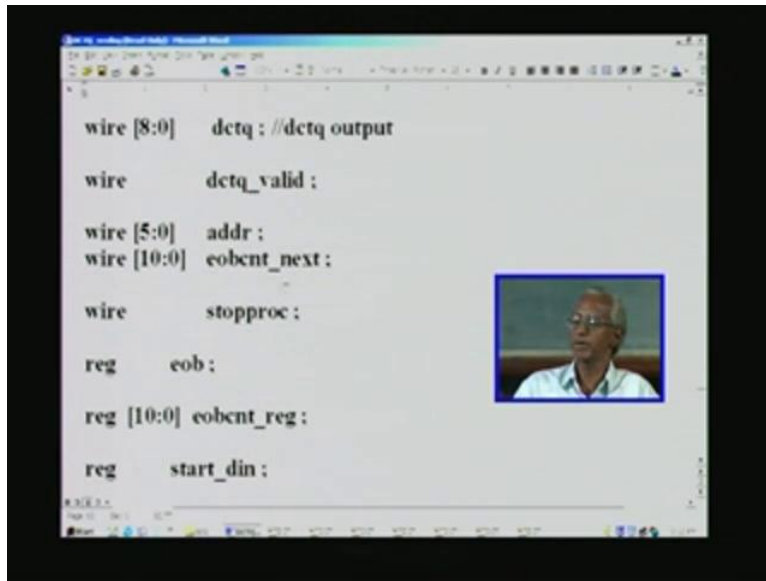
wire [5:0]  addr ;
wire [10:0] eobcnt_next ;

wire      stopproc ;
```



Wire denotes the outputs. Notice that DCTQ, its validity, and address are all wire.

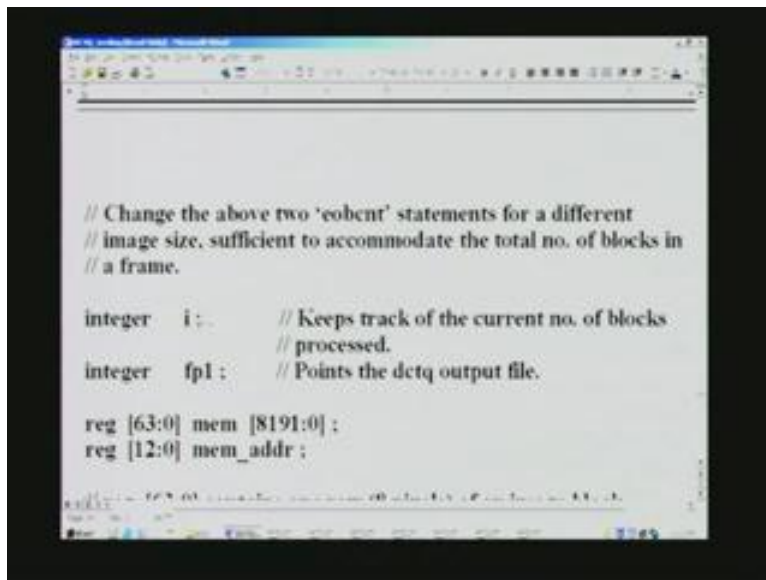
(Refer Slide Time: 07:52)



```
wire [8:0]   dctq ; //dctq output
wire       dctq_valid ;
wire [5:0]  addr ;
wire [10:0] eobcnt_next ;
wire       stopproc ;
reg        eob ;
reg [10:0] eobcnt_reg ;
reg        start_din ;
```

We also need a counter for end of block and also a signal called **stopproc** to stop after **1024 blocks are processed** and so on – all these are used here. When a particular block has ended, that also should be known. When the test bench must start the whole DCTQ processing is governed by this signal.

(Refer Slide Time: 08:14)



```
// Change the above two 'eobcnt' statements for a different
// image size, sufficient to accommodate the total no. of blocks in
// a frame.
integer   i ; // Keeps track of the current no. of blocks
           // processed.
integer   fp1 ; // Points the dctq output file.
reg [63:0] mem [8191:0] ;
reg [12:0] mem_addr ;
```

If you want a different picture, we have to change the blocks. All the statements to be changed are listed here. In the test bench, we have a counter that keeps track of the number of blocks processed. DCTQ output file list is **contained in fp1**.

(Refer Slide Time: 08:33)

```
// image size, sufficient to accommodate the total no. of blocks in
// a frame.

integer  i;      // Keeps track of the current no. of blocks
              // processed.
integer  fp1;    // Points the dctq output file.

reg [63:0] mem [8191:0];
reg [12:0] mem_addr;

// reg [63:0] contains one row (8 pixels) of an image block -
// 8 such rows, one block.
// 1024 such blocks means 8192 rows.
// Change mem[8191:0] & reg [12:0] for a different image size.

dctq  dctq1(      // Invoke dctq design module to get
                // the dctq output.

                .pci_clk(pci_clk),
```

First, we read a disk file and copy it into a memory array. We have seen this in external memory design earlier. We have a total of 8192 here, each being 64 bits. This much memory is required in order to have 1024 blocks of 256 by 256. This is related in terms of number of bits – for address, you need 13 bits, because it is nothing but 8K.

(Refer Slide Time: 09:05)

```
reg [12:0] mem_addr;

// reg [63:0] contains one row (8 pixels) of an image block -
// 8 such rows, one block.
// 1024 such blocks means 8192 rows.
// Change mem[8191:0] & reg [12:0] for a different image size.


dctq  dctq1(      // Invoke dctq design module to get
                // the dctq output.

                .pci_clk(pci_clk),
```

That is what it says here. We now invoke the DCTQ design and list all the I/Os therein.

(Refer Slide Time: 09:15)

```
.clk(clk),
.reset_n(reset_n),
.start(start),
.di(di),
.din_valid(din_valid),
.wa(wa),
.be(be),
.hold(hold),
.ready(ready),
.dctq(dctq),
.dctq_valid(dctq_valid),
.addr(addr)
```



```
);

initial
begin

    Sreadmemb("lena.txt",mem); // mem receives the input image
                               // frame, lena.txt
                               //change the name for a different image frame.
```

Then, we start the initial block of the test bench. In the first step, we read lena.txt, which is the actual image file in a text form and put it in the memory that we have declared earlier as **reg mem**. That is what this command does – **\$readme hex format** is the instruction for that.


(Refer Slide Time: 09:44)

```
fp1 = Sfopen("dctq.txt");
//dctq.txt is the dctq output of the image frame, lena.txt.

pci_clk = 0 ;
clk = 0 ;
reset_n = 1 ;
start = 0 ;
di = 0 ;
din_valid = 0 ;
wa = 0 ;
be = 8'hff ;
hold = 0 ;
```

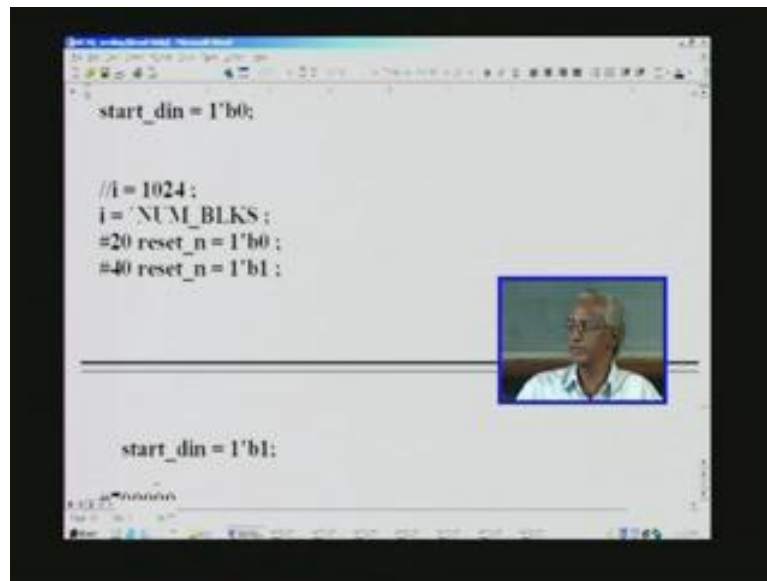
```
fp1 = Sfopen("dctq.txt");
//dctq.txt is the dctq output of the image frame, lena.txt.

pci_clk = 0 ;
clk = 0 ;
reset_n = 1 ;
start = 0 ;
di = 0 ;
din_valid = 0 ;
wa = 0 ;
be = 8'hff ;
hold = 0 ;
mem_addr = 0 ;
start_din = 1'b0;
```



Then, we need to identify the output file, which we do here and also ask it to open that. This is the command for that. **dctq.txt is the dctq output** and is derived from lena.txt, which is the input. Here, we initialize all the inputs. These are all the signals that you are already familiar with and so, we will not go into the details.

(Refer Slide Time: 10:16)



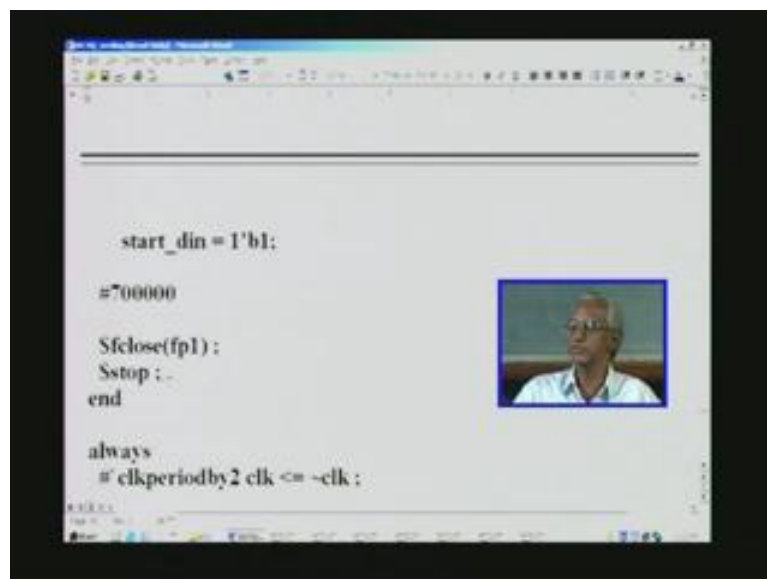
```
start_din = 1'b0;

//i = 1024;
i = 'NUM_BLKs;
#20 reset_n = 1'b0;
#40 reset_n = 1'b1;

start_din = 1'b1;
```

i is a counter that we have already set before. At this point of time, we apply the reset, so active low here for 20 nanoseconds, **then withdraw** and then give the start signal.

(Refer Slide Time: 10:29)



```
start_din = 1'b1;

#700000
Sfclose(fp1);
Sstop;
end

always
# clkperiodby2 clk <= ~clk;
```

```
#700000

Sfclose(fp1);
Sstop;
end

always
# clkperiodby2 clk <= ~clk;

always
# pci_clkperiodby2 pci_clk <= ~pci_clk;

always @ (start_din or i or clk or pci_clk or reset_n or wa or
```

Once you give the start signal, because we have invoked the DCT design already, it will start functioning. After 700,000 nanoseconds, the file will be closed and stopped, during which time you would have already processed one frame of image and got it in **dctq.txt** as output file. This is the statement for inverting the clock so that the clock can run. **This is for both the clocks.**

(Refer Slide Time: 10:56)


```
always
# pci_clkperiodby2 pci_clk <= ~pci_clk;

always @ (start_din or i or clk or pci_clk or reset_n or wa or
mem_addr)
begin
if(start_din == 1'b1)
begin
```

```
always a (start_din or i or clk or pci_clk or reset_n or wa or mem_addr)
begin
if(start_din == 1'b1)
begin


```

```
a (posedge pci_clk)
if(i != 0) // Image block counter.
begin
```



We now have an always block to take further decisions and process block after block. We will see how it is done. We need to take action only if start is 1, which we have already made 1 there.

(Refer Slide Time: 11:11)

```
a (posedge pci_clk)
if(i != 0) // Image block counter.
begin
a (posedge pci_clk);
=1;
din_valid = 1;
wa = 0;
di = mem[mem_addr]; // Inputs first row of an image
// block.
mem_addr = mem_addr + 1;
repeat(7)
```



```

begin
  @ (posedge pci_clk);
  #1;
  din_valid = 1;
  wa = 0;
  di = mem[mem_addr]; // Inputs first row of an image
                      // block.
  mem_addr = mem_addr + 1;
  repeat(7)

  begin

    @ (posedge pci_clk);
    #1;
    din_valid = 1;

```


```

  din_valid = 1;
  wa = 0;
  di = mem[mem_addr]; // Inputs first row of an image
                      // block.
  mem_addr = mem_addr + 1;
  repeat(7)

  begin

    @ (posedge pci_clk);
    #1;
    din_valid = 1;
    wa = wa + 1;
    di = mem[mem_addr]; // Inputs second to eight rows
                        // of the image block.

```



```

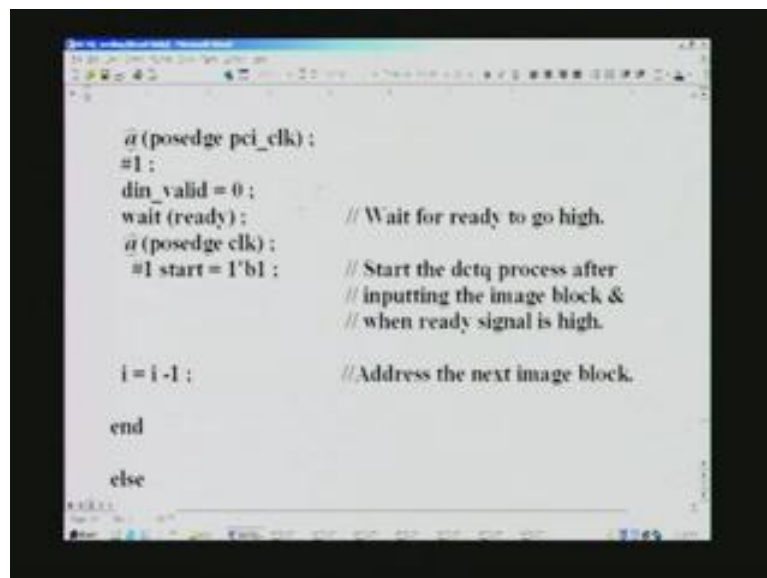
always

  @ (posedge pci_clk);
  #1;
  din_valid = 1;
  wa = wa + 1;
  di = mem[mem_addr]; // Inputs second to eight rows
                      // of the image block.
  mem_addr = mem_addr + 1;

```

At the positive edge of the PCI clock, we have to read the image block and input it into the actual design. Before we do that, we make sure that all the blocks are not already processed. That is why `i` checks for not equal to 0. Since `i` was initialized to 1024, this will be decremented block after block and checked. At the positive edge of the clock, we make data in valid, because we are going to input the very first block there, which can be got from the memory that we have already stored using `readmemh` – we have already stored the disk file into the actual memory. From that, we start reading right from the first location, which is 0 here, and input to the dual RAM. This is nothing but DCTQ design core. Every time we do this, we increment the address, so that we go to the next location and keep on writing row-wise in a block. You need seven more times in order to complete one block, because there are eight such rows. That is what we do by repeat. This is the block that does precisely the same. This is all exactly the same.

(Refer Slide Time: 12:37)




```
always @(posedge pci_clk)
#1 ;
din_valid = 0 ;
wait (ready) ; // Wait for ready to go high.
always @(posedge clk)
#1 start = 1'b1 ; // Start the dctq process after
// inputting the image block &
// when ready signal is high.

i = i - 1 ; //Address the next image block.
end
else
```

```
        // inputting the image block &
        // when ready signal is high.

        i = i - 1;      //Address the next image block.
    end
else
begin
    wait(eobcnt_reg == 'NUM_BLKs)
```



Next, we wait for ready to go high again, because the DCTQ core has started functioning and it is churning out the first DCTQ block. Once it is complete, it will assert ready. Till that time, we do not input. What we have done so far is we have merely input one block of image data into the dual RAM. Now, we have to start the DCTQ process. That can be done only by asserting the start signal here. We keep repeating this block after block. That is why we are decrementing from 1024, then it becomes 1023 and so on. Finally, it will become 1. When it becomes 0, it quits the always block, as we have seen before.

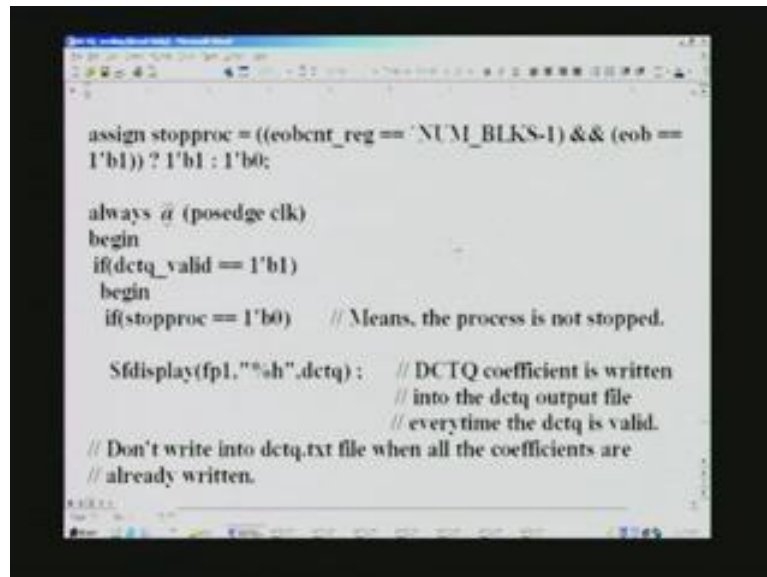
(Refer Slide Time: 13:30)

```
begin
    wait(eobcnt_reg == 'NUM_BLKs);

//Completion of all the image blocks.
Sfclose(fp1);
Sstop;
end
end
```

In addition to this, we need a counter for counting the number of blocks processed and then compare it with the number of blocks we have already given right at the beginning, which is 1024. This is the running counter. When **the two are equal**, then we stop and close the file.

(Refer Slide Time: 13:53)



```
assign stopproc = ((eobcnt_reg == 'NUM_BLK-1) && (eob == 1'b1)) ? 1'b1 : 1'b0;

always @ (posedge clk)
begin
if(dctq_valid == 1'b1)
begin
if(stopproc == 1'b0) // Means, the process is not stopped.

Sfdisplay(fp1,"%h",dctq); // DCTQ coefficient is written
// into the dctq output file
// everytime the dctq is valid.

// Don't write into dctq.txt file when all the coefficients are
// already written.
```

The next thing is that we need a signal called **stopproc** in order to stop the entire thing after one frame is already completed – **DCTQ value is computed** and stored in dctq.txt file. This expression does that precisely. The running counter is examined for the number of blocks and we see that end of block is also 1. Only then, we take this action. If that is not met, it means that DCTQ processing is still going on. Only then, we write into the output file, which is dctq.txt. If it is 1, we will not write here, which means that **stopproc** will go right at the end of all the blocks being processed.

(Refer Slide Time: 14:42)


```
end

always @ (posedge clk or negedge reset_n)
begin

    if(reset_n == 1'b0)
        eob <= 1'b0;
    else if(addr == 6'd63)
        eob <= 1'b1;
    else
        eob <= 1'b0;

end

assign eobcnt_next = eobcnt_reg + 1;
```




```
end

assign eobcnt_next = eobcnt_reg + 1;

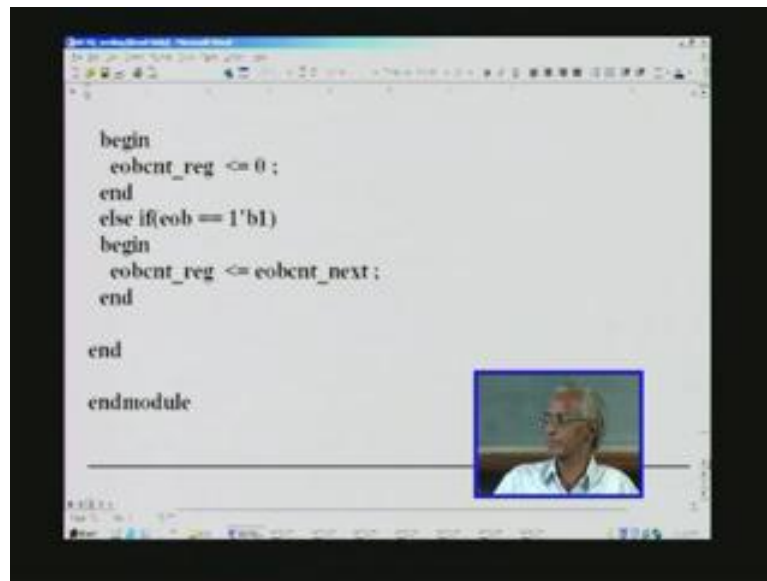
always @ (posedge clk or negedge reset_n)
begin

    if(reset_n == 1'b0)
```



In the next step, we have to use end of block. This is generated by just examining `addr = 63`. This is the running address of the DCT coefficient; 63 means that it has just now processed all the 64 coefficients and so we set it to 1, otherwise, we set it to 0. That is how we count this in the next using advance increment for the end of block counter, which we process here.

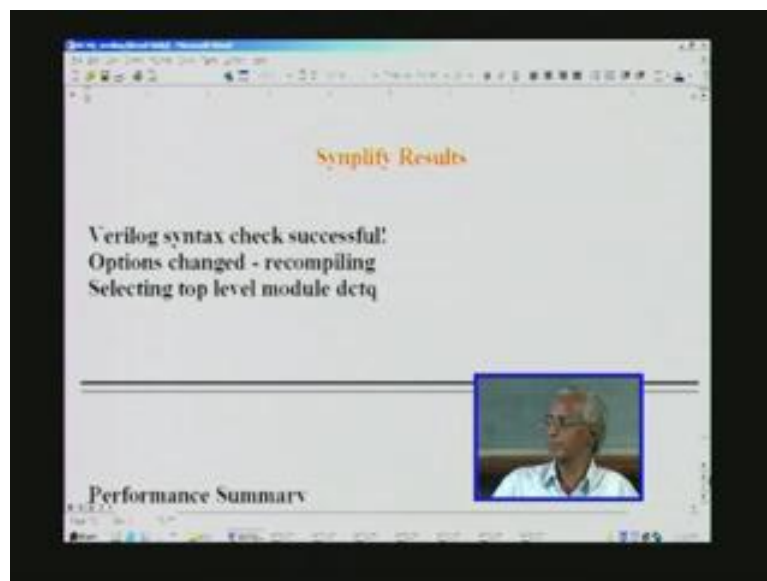
(Refer Slide Time: 15:12)



```
begin
  eobcnt_reg <= 0 ;
end
else if(eob == 1'b1)
begin
  eobcnt_reg <= eobcnt_next ;
end
end
endmodule
```

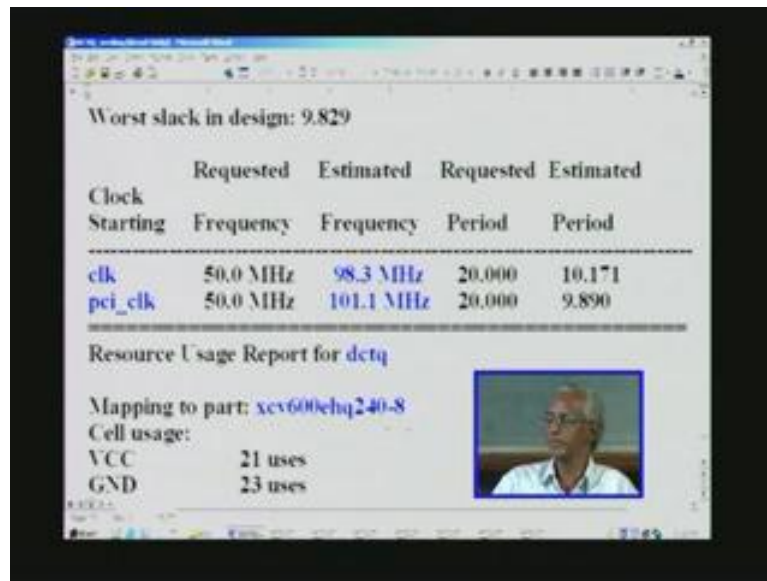
Whenever this end of block is 1, only then we increment – that is how it is taken care of. This completes the test module.

(Refer Slide Time: 15:25)



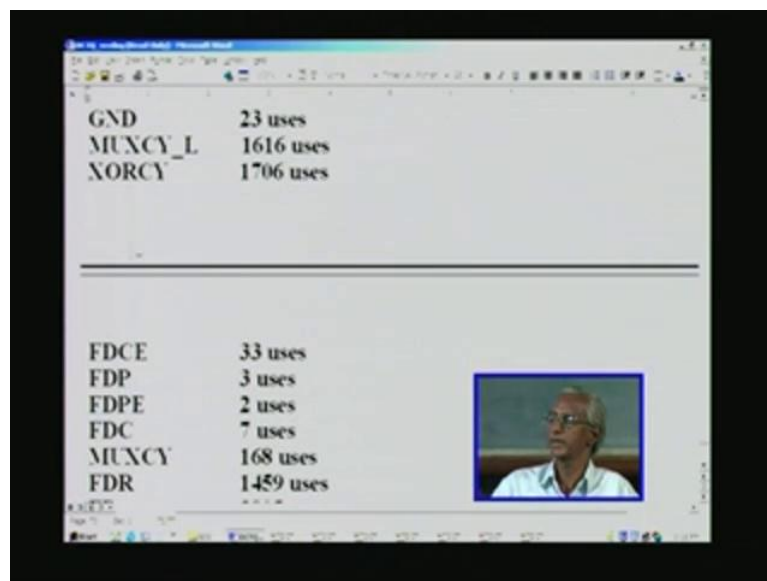
We have the Synplify results for the DCTQ processor here.

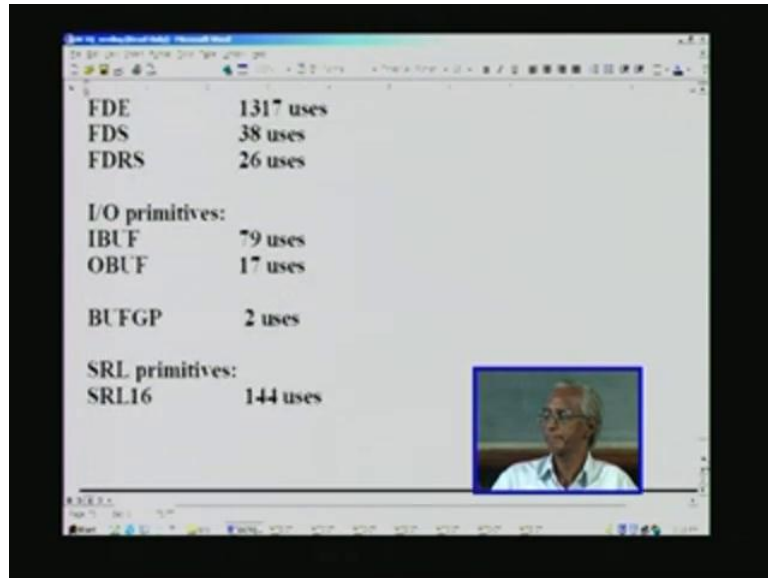
(Refer Slide Time: 15:29)



You will notice that it operates at about 100 Megahertz here. This is the very same device we have used for RAM, ROM, multipliers, adders, etc. earlier.

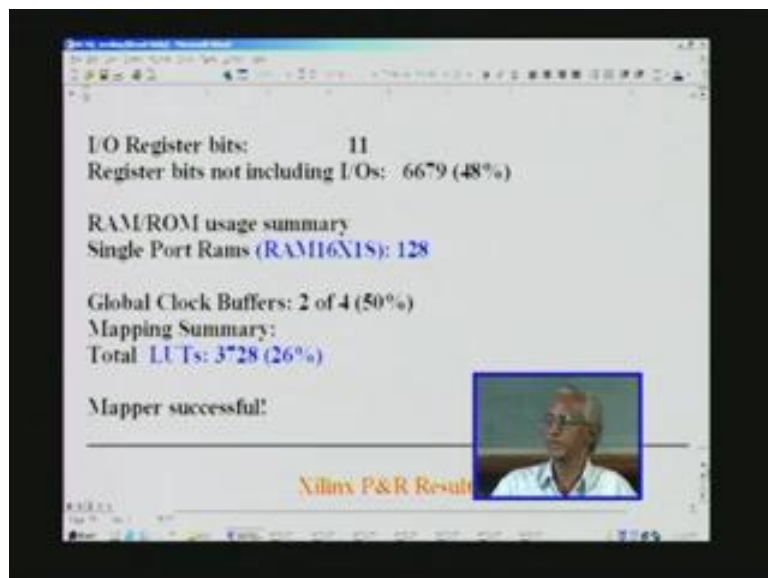
(Refer Slide Time: 15:39)





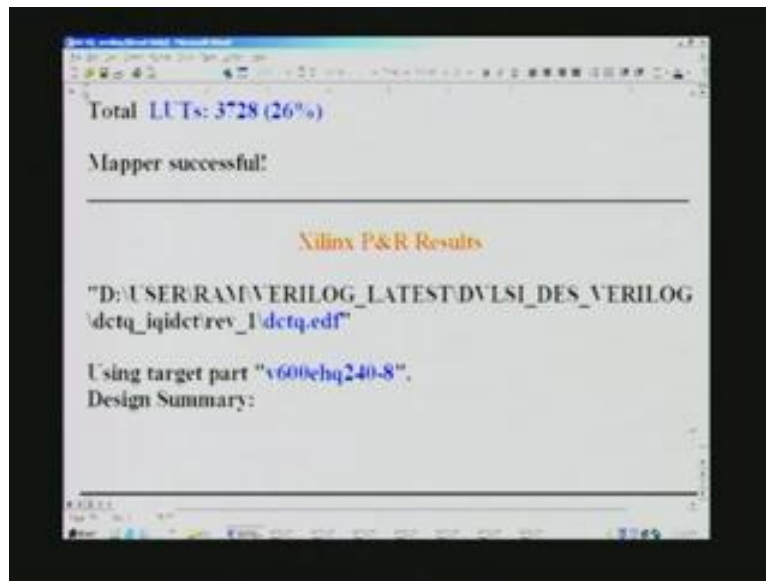
It lists all the primitives that it has consumed. Here too.

(Refer Slide Time: 15:48)



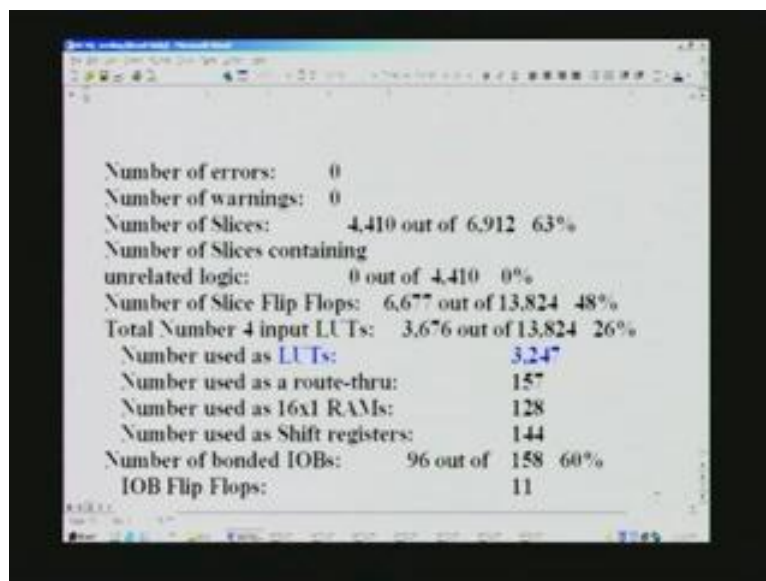
It has taken **16 by 1 RAMs**, 128 in number and it has taken 3728, which is roughly 25 percent of 600,000 gate capacity FPGA. In Xilinx, we will have a little more optimization here, maybe 3200. This may remain the same.

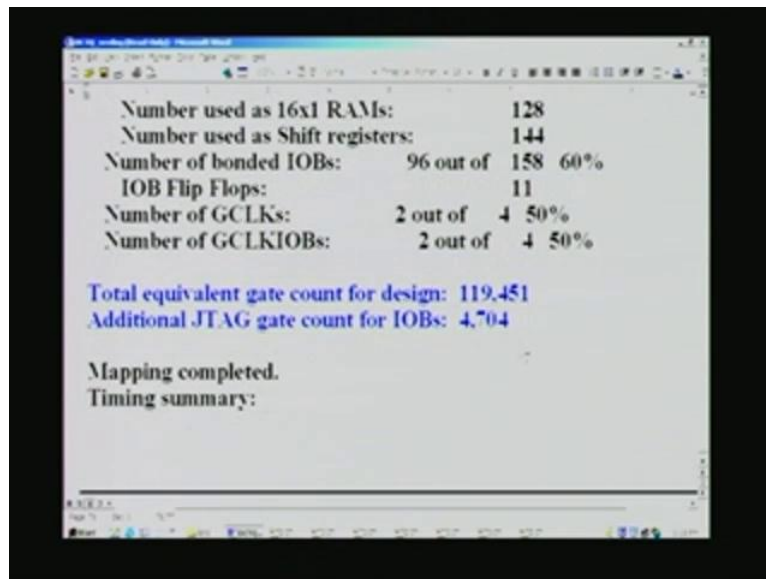
(Refer Slide Time: 16:12)



The input for Xilinx is dctq.edf, which we generated from Synplify. It has mapped the same device.

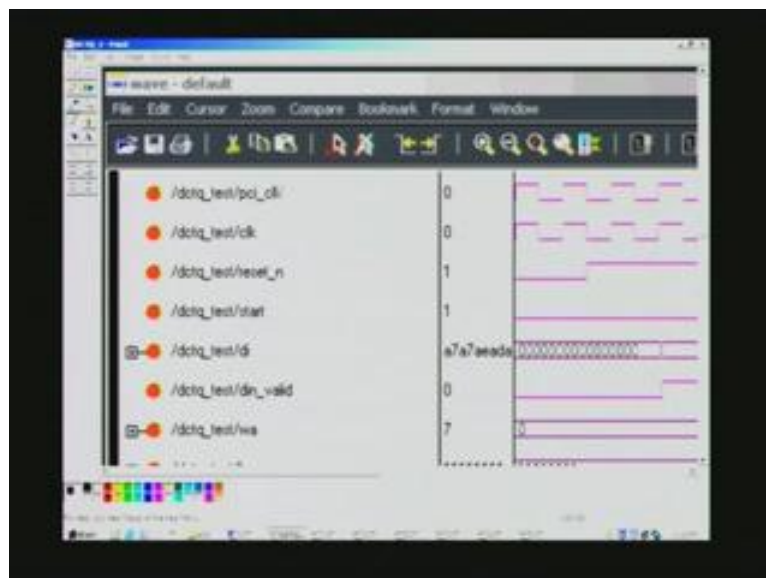
(Refer Slide Time: 16:25)

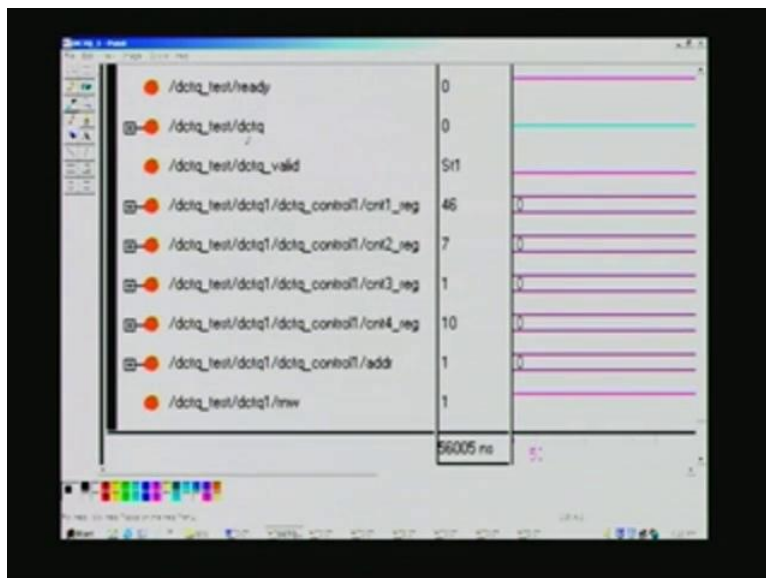
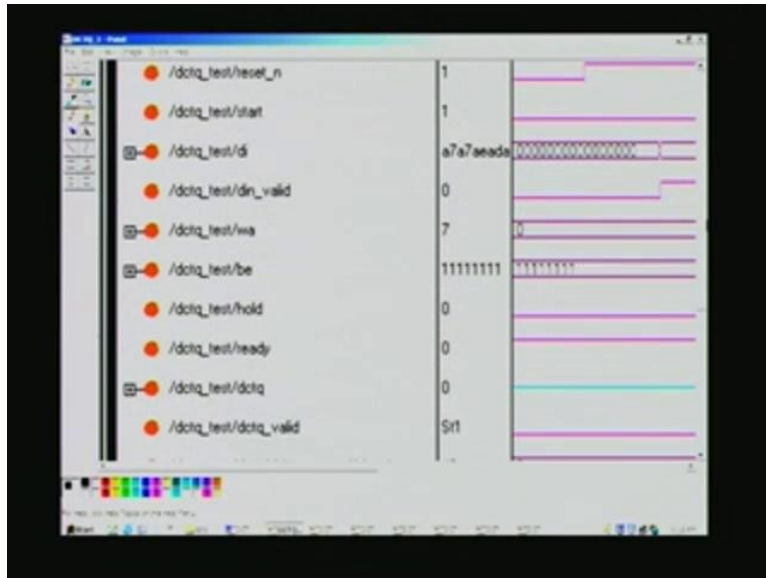




It lists the number of slices, etc. Notice that it has optimized the number of LUTs from 3700 to 3200. The number of gates **count** for this particular DCTQ design is around 124,000 gates. We will see the waveform so that we can verify whether it has functioned correctly or not.

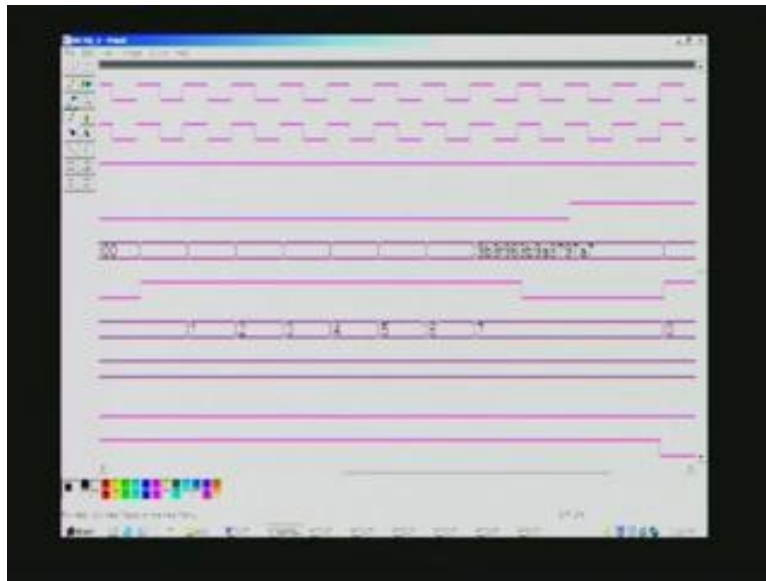
(Refer Slide Time: 16:47)





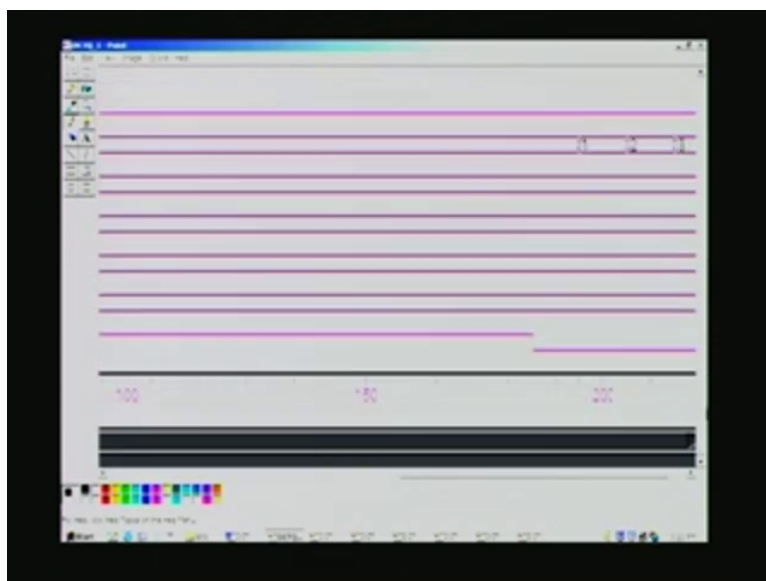
The different signals are listed here. There are over 100 signals. It is physically impossible for us to take all the signals. We will examine the vital signals here. For example, DCTQ is here and its validity is here. Notice that no DCTQ comes here. These are all different counters we have used. We will also examine one rnw signal. In the first thing, di gets the dual RAM input – gets the first block of data. This is `din_valid`.

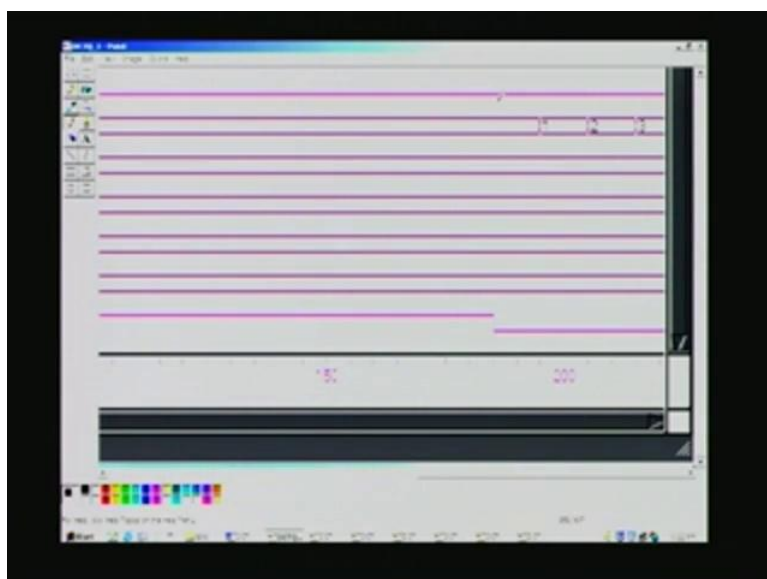
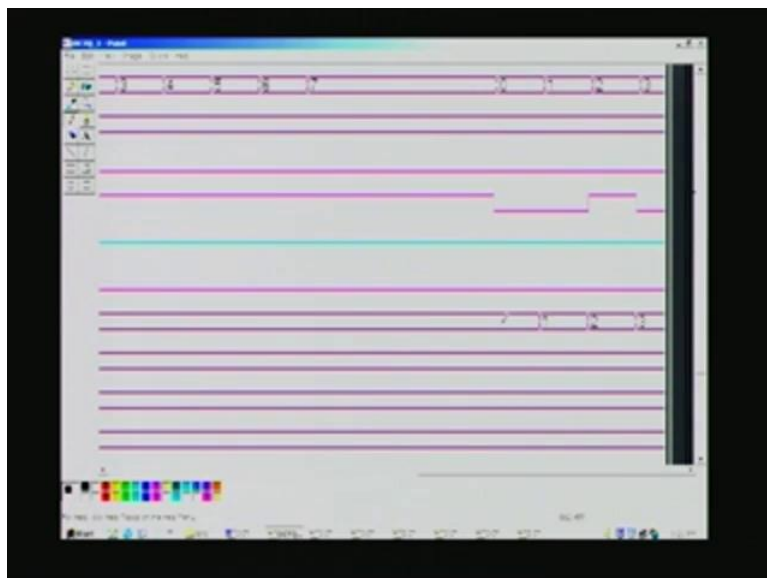
(Refer Slide Time: 17:20)



You can see this counter running here – 0, 1, 2, 3 and data is input precisely at that point. **din_valid** is also high, 0 through 7, which means 8 such **writes** are possible and each **write** will be a total of eight bytes; you can see 16 digits of hex decimal. Then, **din** is withdrawn and only after a while, it will be written here. This signal is the ready signal that you have at the bottom. It is normally ready and when the first data is written, it will temporarily be withdrawn and once again, asserted when the counter is 1, which we have already seen in the design.

(Refer Slide Time: 18:05)

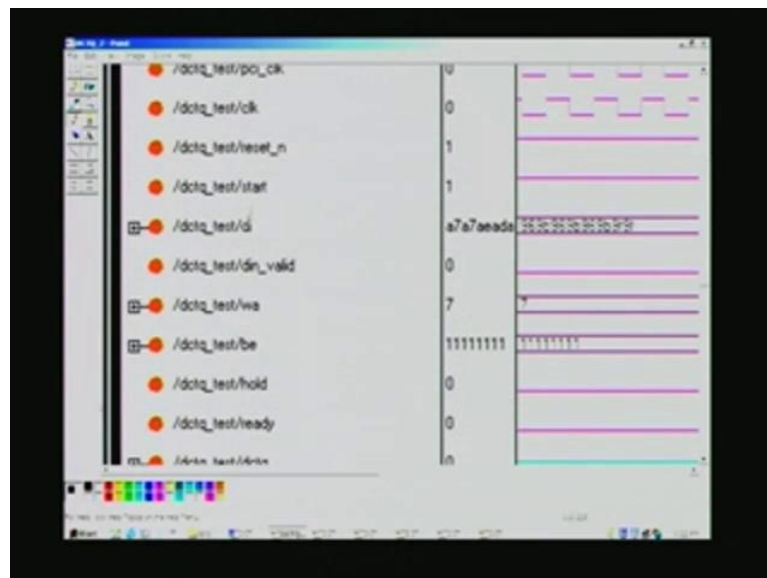




Just remember one aspect here. I think we are starting here, the start pulse is here. That is the third waveform and it is happening right at this point of time. Although it is happening at this point of time, it will be recognized only at the positive edge. It has missed the clock marginally and so, this edge will be noticed only here. Once it is noticed, this start is converted into another **start reg**, which we have seen in the design. That will **come only one more** clock cycle here. Therefore, only at this point of time where this is 0 here is the actual start happening. Corresponding to this 0, you can see this is also 0 here. This is nothing other than **cnt1**.

These are the different counters we had used for keeping track of the pipeline. Let us just remember when this 0 starts. That corresponds to say 185 nanoseconds here, right at this point. Simultaneously at the same point, *rnw* also switches control from 1 to 0, so that another RAM can be written here. *cnt1* is 0 and starting to count implies that the first block of DCTQ is going to be processed right from here. This is because the start pulse takes effect only here. This happens to be 185 nanoseconds, which we have noted down. Let us go onto the second one.

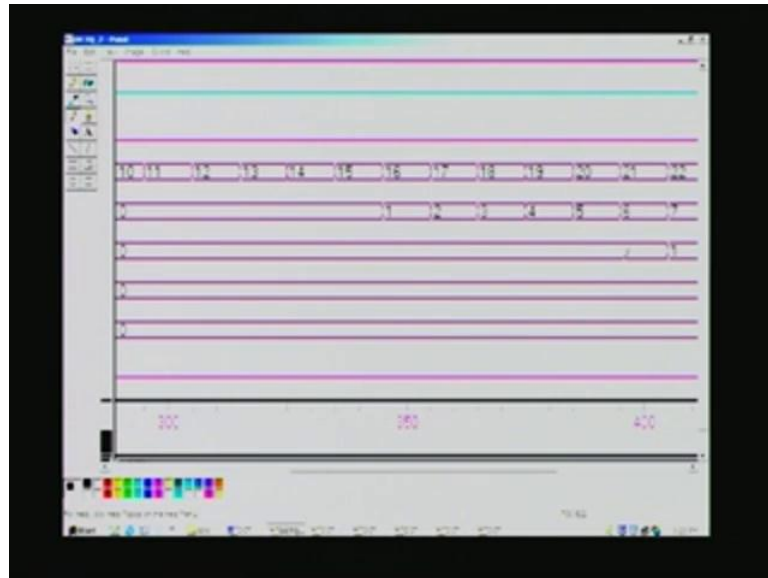
(Refer Slide Time: 19:44)



You can see that **din** is not applied right here, because it is not yet time. What is happening here is that **cnt1** is still running and you can see that it has just crossed 10, 11, etc. We have seen in the DCT controller that precisely when **counter is 15**, only then **cnt2** must stop. Likewise, **cnt3** will stop only for 20. Although it appears to be 21 here, **it is 20 and we have enabled the counter**. The actual counter itself will take place

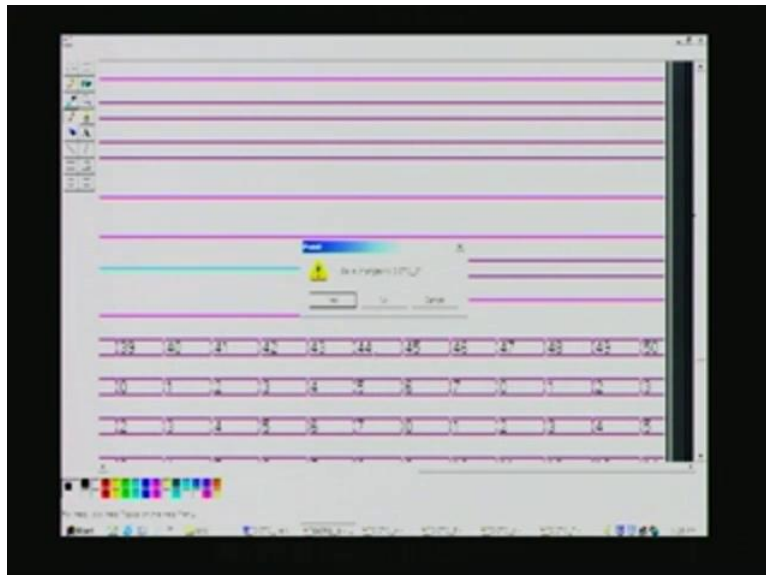
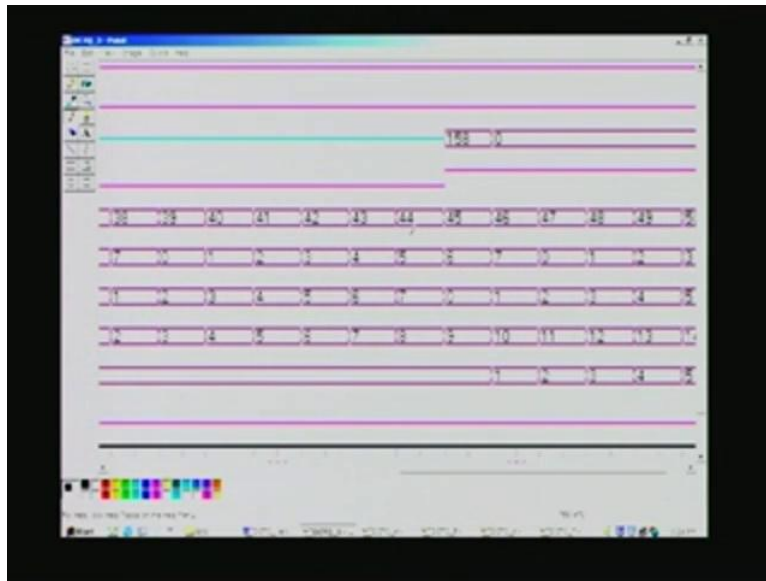
only one clock later on. That is why it is deferred here. In fact, you can see right here that the 0 must coincide with 20. That is the implication. We have precisely the same thing earlier for 15; you can see 15 and 15 start here. **I think I have to examine that. I do not clearly remember what value it is.**

(Refer Slide Time: 20:52)



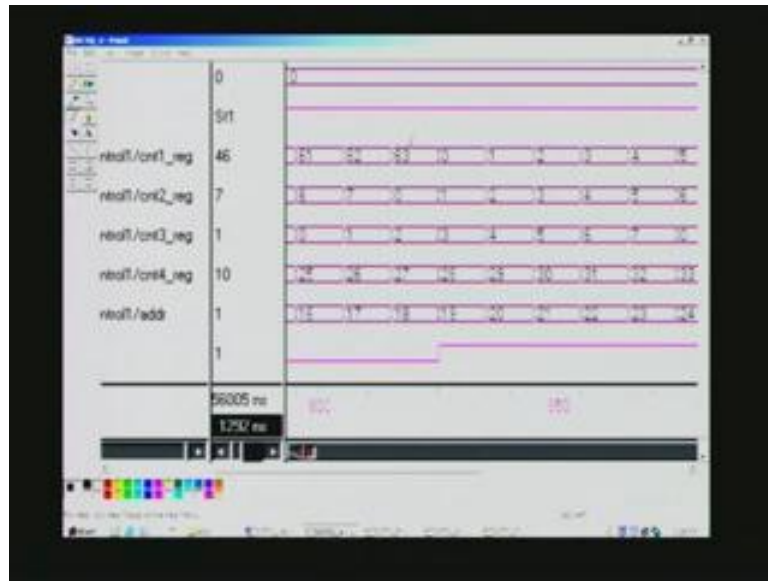
Similarly, we have seen that for the next counter **0 is 20**. These are all the pipelines. If you make a single mistake, even one clock cycle this way or that way, you will not get proper output and everything will be in total disarray. This is what the waveform says. We will straightaway go to DCTQ and the counter here; further counting has taken place here. Now, at this stage, the last counter has gone – it is cnt4 here and it is happening at 35. **You must be remembering 20, 35 was okay.** Then, it starts counting.

(Refer Slide Time: 21:41)



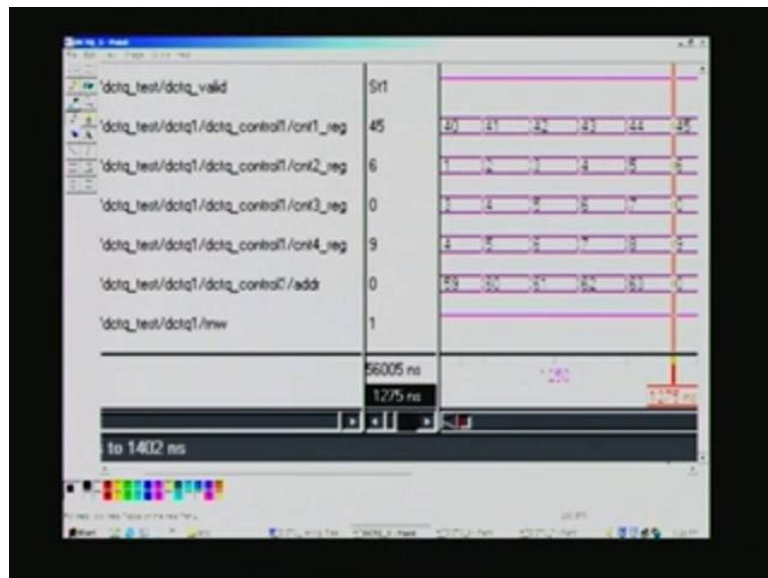
Similarly, for DCTQ, address will also be given by the last counter; that will take place only at 44 or so. You can see 44 right at this point of time. Now, you notice that this is the DCTQ. The very first DCTQ block has come only after the 45 clock cycles right from the start. At the start, it was 185 nanoseconds. Now, let us see how much time this is occupying here. This one is 635 nanoseconds. If you take the difference, you get precisely 450 nanoseconds, which means 45 clock cycles. This is because our time period of 100 Megahertz is 10 nanoseconds. Precisely after the latency of 45 clock cycles, you have got the first output. This is the DC coefficient and the others are all AC coefficients. They will be 64 in number. **If you go through that,** we will go into the other waveforms.

(Refer Slide Time: 22:38)



You can see that it has completed the very first block data writing. **cnt1** is for the **di input** for the dual RAM and it has completed. Since it has completed, **rnw** is once again automatically switching and so, not a single clock cycle is wasted. It is a continuous firing of the input image and so also the output. You can also see the output here. This is not yet complete. We will see **in the** next waveform.

(Refer Slide Time: 23:15)



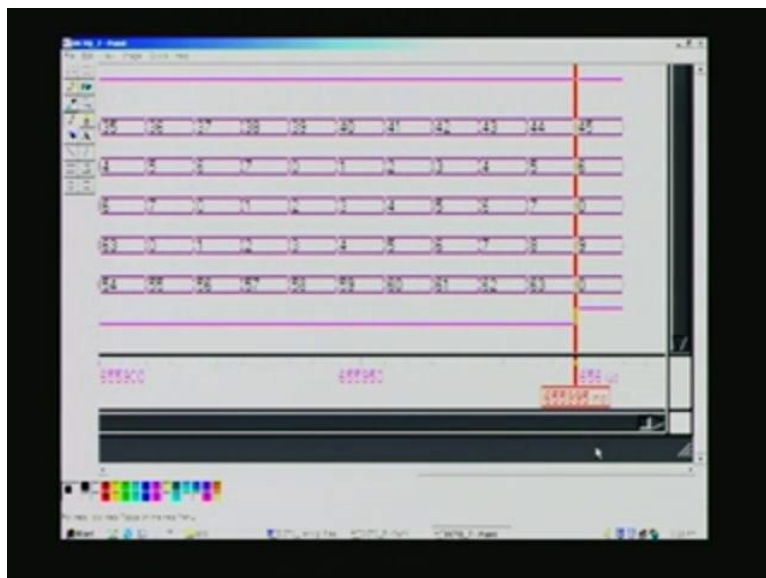
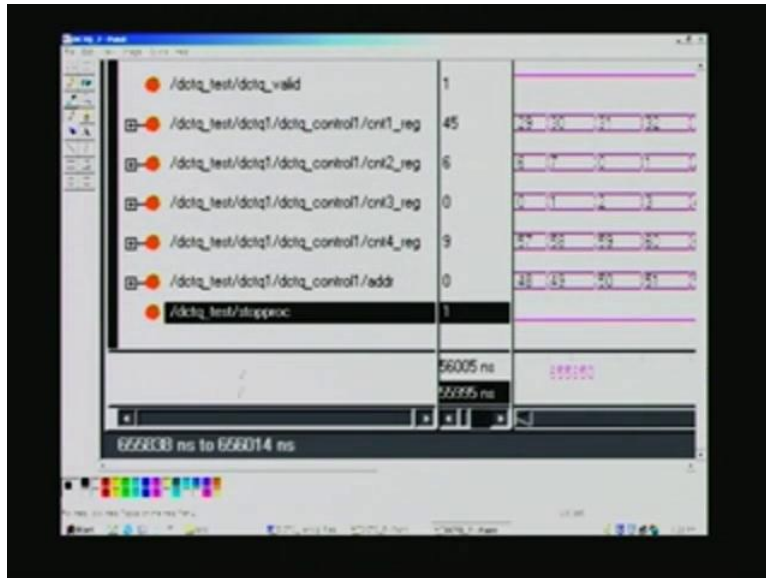
In the next waveform, you can see that this is the address for the coefficient. You can see that the coefficient number is 63, which means that the 64th coefficient is just processed. Immediately, without wasting any clock cycle, it starts processing the next

block. At this point of time, it is 1275 nanoseconds. The start of the block was at 635 nanoseconds. If you take the difference between these two, you will get precisely 640 nanoseconds and if you divide this by 10, you get 64 clock cycles. Our claim was that every coefficient is created at every clock cycle. This is proved and it is continuing here. That is what we want to see. It keeps going there. We have some more waveforms.

(Refer Slide Time: 24:15)

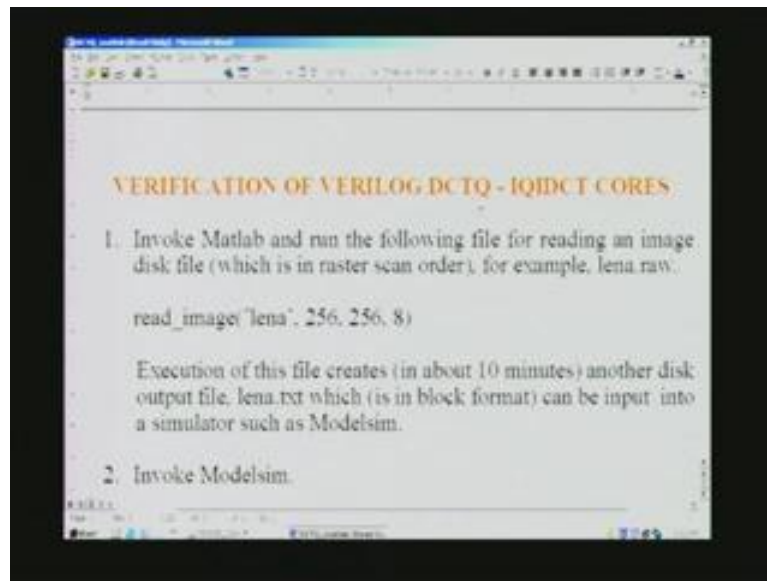


This is towards the end of all the 1024 blocks being processed. That is the start. This **04b** is now not shown in decimal, but in hex decimal just for a change, because the output file derives only in hex decimal. You can see that AC coefficient is here. The first AC coefficient is 8, the others are 0. As we mentioned before, DCTQ succeeds in making most of them zeros – there will be only a few non-zero coefficients. The first coefficient will always be a very high value. That is for DC coefficient. This is the last but one block that is processed here.



This is the last waveform. Let us see towards the very end of the block. We have what is called a stop processor processing after all the 1024 are done. Now you can see that the last coefficient is 63, this address. You can see that not a single cycle is wasted anywhere and you can verify it in a minute. You can see that **stopproc** goes high at this point of time, indicating that all the 1024 blocks are processed. We started the processing at 635 nanoseconds and now, we are ending the process at 655,995. If you subtract the two, you will be getting 655,360 nanoseconds. Knock off that 0 and you will get it in clock cycles – it has taken 65536 clock cycles for 1024 blocks. Each block is 64 bytes. Therefore, it is 65,536, which you know as 64K. You see that the entire image has been processed at exactly one pixel per clock cycle. This proves the first statement we made at the time of explaining the algorithm.

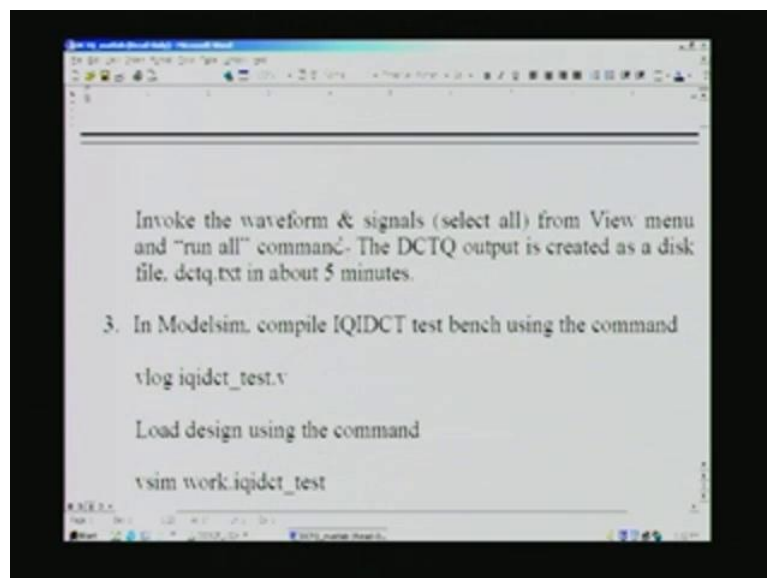
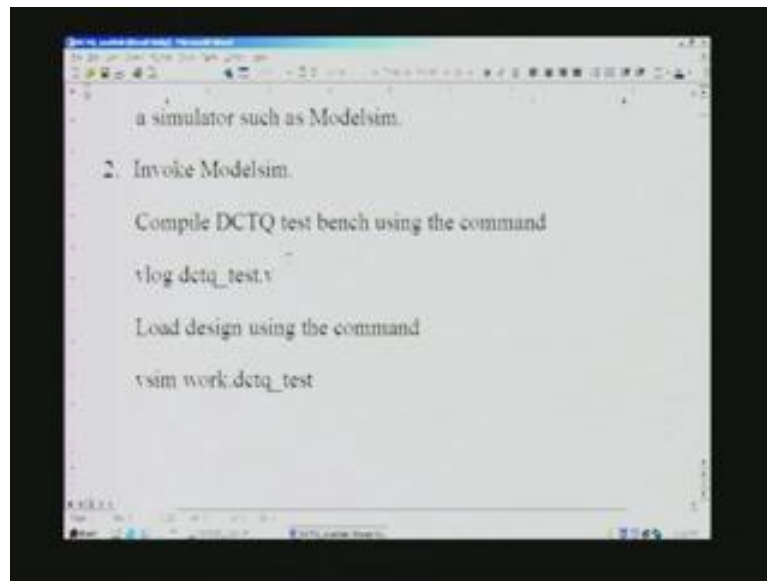
(Refer Slide Time: 26:15)



We have completed the design of DCTQ. In a similar fashion, you may have to do IQIDCT on your own, which is just the inverse of DCTQ. We need to verify whether these DCTQ – IQIDCT cores are functioning properly. In order to do this, we will have to resort to MATLAB coding, which we will cover. Before we start the MATLAB coding, let us see what we need to do for verification. I will just read out. Invoke MATLAB and run the following file for reading an image disk file, which is in raster scan order. The disk file is lena.raw and you need to run this file in MATLAB (after invoking MATLAB).

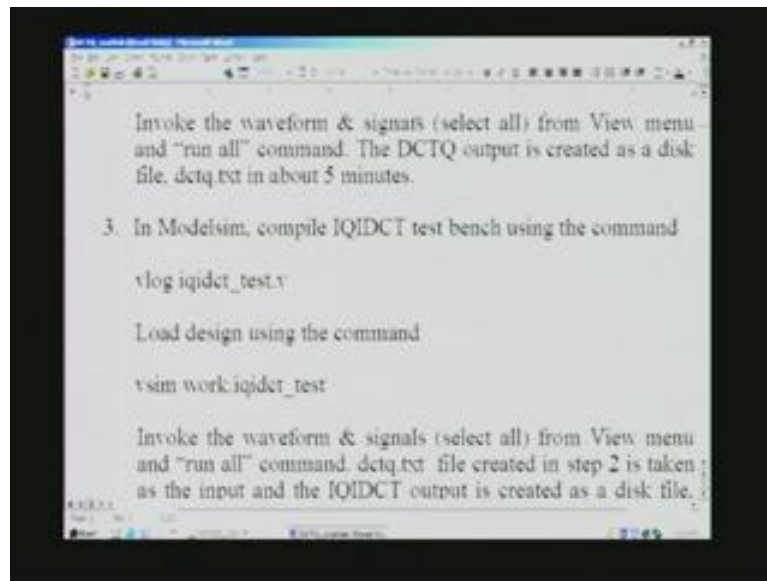
You have also to specify what the size is and that you want to process block by block; 8 indicates 8 by 8 block and lena is the file name. Execution of this file creates (it takes about 10 minutes, depending upon the computer that you use) another disk output file lena.txt, which is in block format. Now, it is converted into block format. We have to change the raster scan into block format, because we want to simulate. That is what we are doing here so that we can use it in ModelSim.

(Refer Slide Time: 27:27)



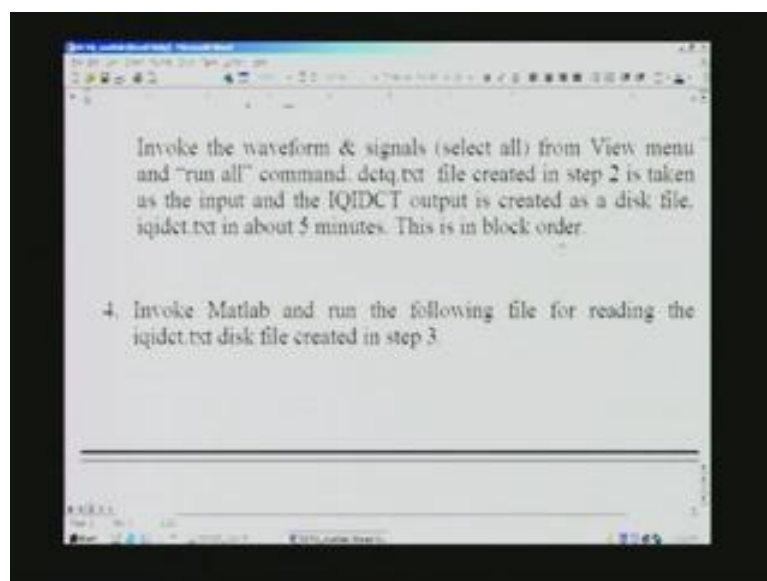
The next step is to invoke ModelSim, compile the DCTQ test bench and also load. You can use these commands instead of the menu, which we have seen before. Also, invoke the waveform and signals (select all) from menu and the "run all" command. DCTQ output is created as a disk file dctq.txt. It takes about five minutes. DCTQ has produced the result and that will be in dctq.txt. This will be precisely what you have seen as the waveform, say 158, etc., which you have seen – all that will be contained in this. That will be in hexadecimal or rather binary, because we are dealing with the hardware.

(Refer Slide Time: 28:11)



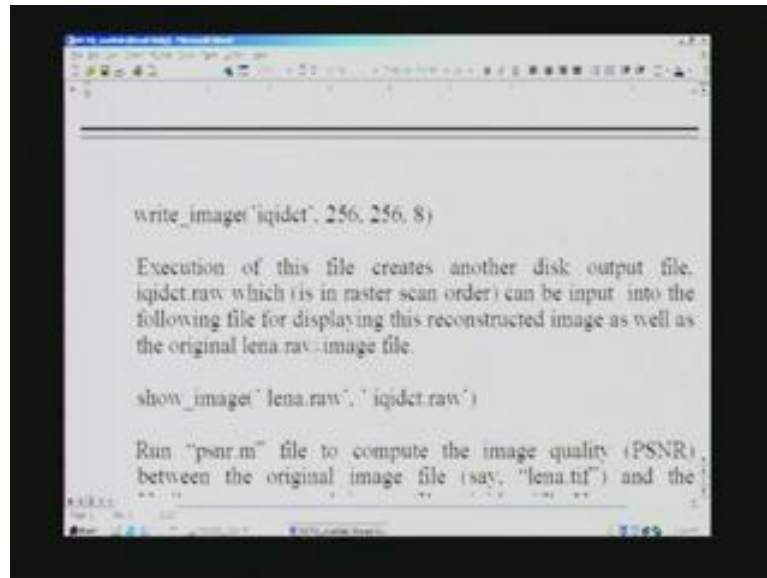
The next step is to invoke the IQIDCT. It is the inverse and it is left as an exercise for you. I hope that you complete it and then you can run this. Once again, you do the compilation as well as load the design and then invoke the signals and run the same. When you do that, it will take dctq.txt that got in the previous step and do the inverse operation and thus, reconstruct the picture. It will have iqidet.txt as the output. Once again, it takes so much time. This will be in the block order, because we have converted into block earlier.

(Refer Slide Time: 28:52)



In order to see the image in the right perspective, we need to once again convert this block order into raster scan order, which we do by invoking another MATLAB program. It takes this, which was created in step three.

(Refer Slide Time: 29:04)



```
write_image('iqidct', 256, 256, 8)

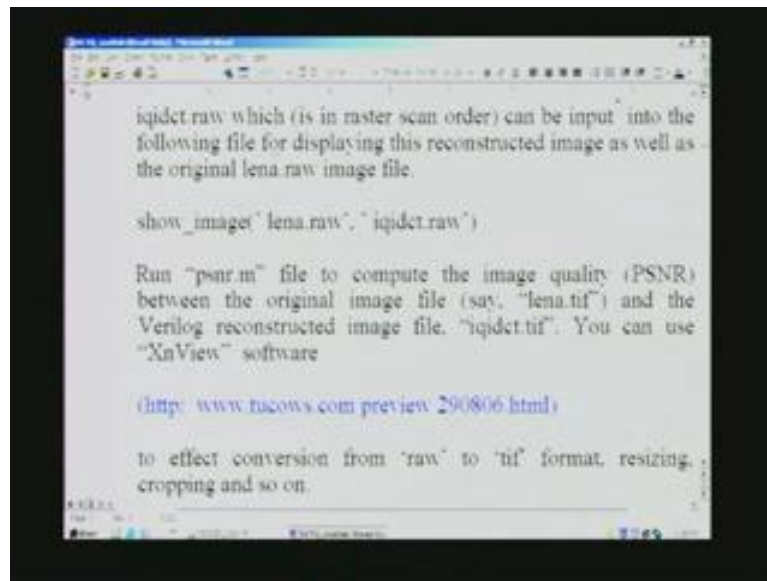
Execution of this file creates another disk output file,
iqidct.raw which (is in raster scan order) can be input into the
following file for displaying this reconstructed image as well as
the original lena.raw image file.

show_image('lena.raw', 'iqidct.raw')

Run "psnr.m" file to compute the image quality (PSNR)
between the original image file (say, "lena.tif") and the
```

This is the command you have to run. This is the file you have to run in MATLAB. iqidct is the file. Execution of this creates another file called iqidct.raw, which is in raster scan order. This will automatically be done in raster scan order. When we see the MATLAB code, we will understand this. The file for displaying this reconstructed image as well as the original lena.raw image. We have got the reconstructed output as a file and we have also succeeded in converting it into a raw file, which is raster scan order. We need to show this, so that we can visually see the picture and compare. For example, original picture is lena.raw and the reconstructed picture can be compared visually. Then, you can make out whether the picture quality is good or not.

(Refer Slide Time: 29:55)



```
iqidct.raw which (is in raster scan order) can be input into the
following file for displaying this reconstructed image as well as
the original lena.raw image file.

show_image('lena.raw', 'iqidct.raw')

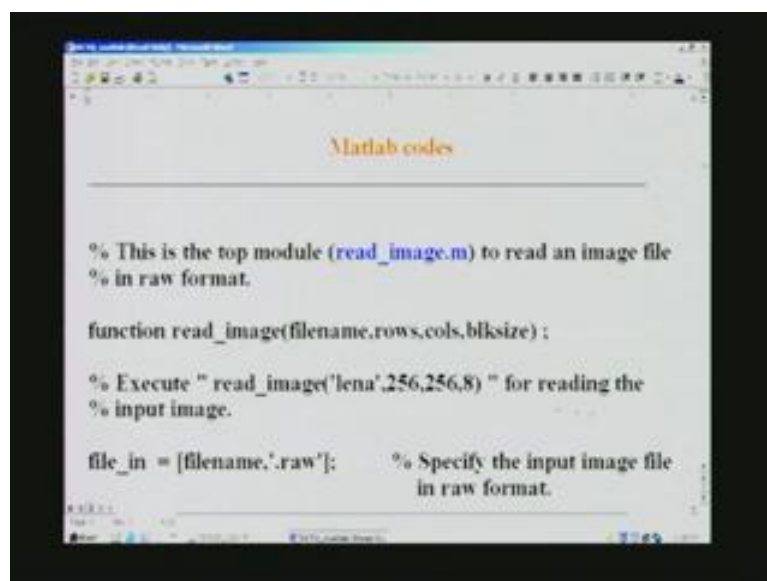
Run "psnr.m" file to compute the image quality (PSNR)
between the original image file (say, "lena.tif") and the
Verilog reconstructed image file, "iqidct.tif". You can use
"XnView" software

(http://www.tucows.com/preview/290806.html)

to effect conversion from 'raw' to 'tif' format, resizing,
cropping and so on.
```

In order to have some ready reckoner, you also have a computation that is called **psnr** – power signal noise ratio. It is conducted over the entire image from the reference image (the original image) and the reconstructed image. For this also, you need to run another MATLAB code called show_image for showing and psnr.m for computing the PSNR codes. We have said before that we need to download the XnView software. You can download it from this site. You can use XnView in order to convert from one format to another – raw to TIF format and so on, which we will need for running the MATLAB code.

(Refer Slide Time: 30:43)



```
Matlab codes

% This is the top module (read_image.m) to read an image file
% in raw format.

function read_image(filename,rows,cols,blksize) ;


% Execute " read_image('lena',256,256,8) " for reading the
% input image.

file_in = [filename,'.raw']; % Specify the input image file
in raw format.
```

```
% Execute " read_image('lena',256,256,8) " for reading the
% input image.

file_in = [filename,'.raw']; % Specify the input image file
                             % in raw format.
file_out = [filename,'.txt']; % and output image file in txt
                             % format.

fp1 = fopen(file_in,'r');
fp2 = fopen(file_out,'w');
```



Here are the MATLAB codes. This is the top module in order to read the image. Here, we declare that as a function. We give the file name, number of rows, block size, etc. An example is given here. We also have to declare the input file as a raw format and output file as text format, because we need to get lena.txt here, which we will input into ModelSim. That is what we are trying to do here. We have to declare it as read and write after opening the file.

(Refer Slide Time: 31:17)

```
image = fscanf(fp1,'%c'); % Read the input file which is in
                          % raster scan order.

blks = imageread(image,rows,cols,blksize); % 1-D vector.

% Call the image read function to organize it into blocks.
% Refer imageread.m file for details.

blkn = [];

n = 1;% No. of pixels counter.
```

It is here that we actually read the input file, which is in raster scan order. After reading, this image will be put into a 1-D vector.

(Refer Slide Time: 31:34)

```
blks = imageread(image.rows,cols,blksize); % 1-D vector.

% Call the image read function to organize it into blocks.
% Refer imageread.m file for details.

blkn = [];

n = 1;% No. of pixels counter.

for k = 1:1024,    % 1024 blocks in an image frame.

    for i = 1:8,    % Process one block along height next

        for j = 1:8,    % and along width first.
```

Then, call the image read function to organize it into blocks and see this for details. We initialize a variable here. This is a pixel counter. They are all C-like structures.

(Refer Slide Time: 31:47)

```
blkn = [];

n = 1;% No. of pixels counter.

for k = 1:1024,    % 1024 blocks in an image frame.

    for i = 1:8,    % Process one block along height next

        for j = 1:8,    % and along width first.

            onblk(i,j) = blks(n); % Rearrange it as one block at a time.
            n = n + 1;
```

You have a for loop here, **start end point here** for k, i and j. The innermost loop is this. We have a **raster scan order image** that we are trying to convert into block. That is what we are doing here, for which we need two dimensions, i and j. There are 8 by 8 pixels. Therefore, it is 1 to 8 and 1 to 8. Like this, we have 1024 blocks in an image. That is what this variable **k is for**. We also increment the **counter for the** number of pixels.

(Refer Slide Time: 32:23)

```
end
end

mem(oneblk,0,fp2); % Call the function to write the block
                  % into the output file.
% '0' for writing row-wise or '1' for column-wise writing.
% fp2 is the output image file in txt format.

end

fclose('all');
```

Then, we invoke another module, which will be explained later. **Output oneblk** that we have got into a disk file for use in the next stage. Then, close all the files.

(Refer Slide Time: 32:38)

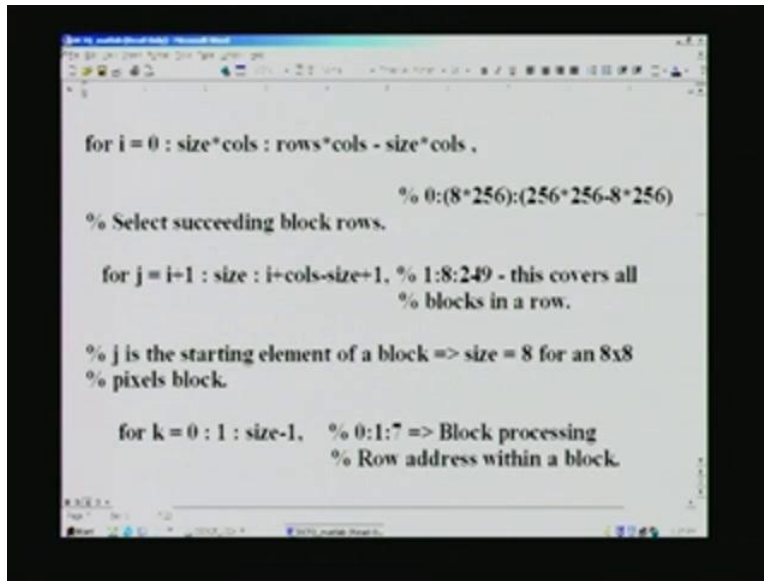
```
fclose('all');
```

```
% Function (imageread.m) to read the image frame block by
% block from top to bottom and, left to right.
% This function is called by "read_image.m" module.

function [x] = imageread(image,rows,cols,size)
x = [];
```

imageread is the function we had called earlier. That is what is declared here. Once again, that is available as x here. **We initialize here and convert all elements to 0.**

(Refer Slide Time: 32:53)

A screenshot of a MATLAB script window showing code for processing an image in blocks. The code uses nested for loops to iterate over rows and columns of blocks. Comments explain the logic, such as selecting succeeding block rows and processing each block of size 8x8 pixels.

```
for i = 0 : size*cols : rows*cols - size*cols ,  
    % 0:(8*256):(256*256-8*256)  
    % Select succeeding block rows.  
    for j = i+1 : size : i+cols*size+1, % 1:8:249 - this covers all  
        % blocks in a row.  
        % j is the starting element of a block => size = 8 for an 8x8  
        % pixels block.  
        for k = 0 : 1 : size-1, % 0:1:7 => Block processing  
            % Row address within a block.
```

Once again, we have for loops here. We already have one block of pixel. Like this, there may be several blocks along the horizontal dimension of the image. We will call that as a row block. It is a row for the block. It means that there are eight rows and so 8 into 256 pixels will be covered in one row block. You should understand that. How many such rows there are vertically in the picture is governed by this i here. You can start that from 0 and it advances as I mentioned in sets of 8 into 256. Then, the last one is 256 into 256 – all the pixels minus this particular step, because that is the last row that we need to process.

Then, within that row block, we need to identify which block we are processing. That is done by this j variable. j is another, which starts from 1. Each block is 8 pixels in width. So, it advances by 1, then 9, then 17 and so on. The very last block will be 249 and the end of that pixel will be 256 – 8 pixels there also.

(Refer Slide Time: 34:15)

```
for j = i+1 : size : i+cols-size+1, % 1:8:249 - this covers all
    % blocks in a row.

    % j is the starting element of a block => size = 8 for an 8x8
    % pixels block.

    for k = 0 : 1 : size-1, % 0:1:7 => Block processing
        % Row address within a block.

        for m = 0 : 1 : size-1, % 0:1:7 - column address within a
            % block.

            x = [x , image(j + k*cols + m) ]; % 1-D vector, appended.

            % (249-7x256-7=8x256), for the last pixel as an example.
```

```
% skip 8*256 pixels.

end
end
end
end

% Function (mem.m) to write 8x8 pixels image block as a hex
% ascii string.

% To be used by Modelsim for processing dctq.

function mem(blk8.col.fn) :
```

We have one more loop for k and m. This is to process a particular block. This will be done by 0 through 7 here. If you take one example, it will be clear. We write what we have processed into a matrix x, which is a one-dimensional vector. If you take one example, the very last in the row block, you remember that 249 was the last there – that is for j and 7 here is the last one and m is also 7 for the last block. **So column is 256. This is here.** It precisely turns out to be 8 into 256. This implies that this is the last pixel that we have processed and put here. This is in 1-D vector. That is what is meant here.

(Refer Slide Time: 35:12)

```
end

% Function (mem.m) to write 8x8 pixels image block as a hex
% ascii string.

% To be used by Modelsim for processing dctq.

function mem(blk8,col,fp) :

% col is '0' for writing row-wise or '1' for column-wise writing.

for i = 1:8,
    hexstr = [] ;
```

Next one is to write this block. What we have done before is we have got it arranged it into a block. Then, we have to write it into the output file. That is what we do by this file.

(Refer Slide Time: 35:23)

```
for j = 1:8,

    if(col == 1)
        num = blk8(j,i) ; % Select column-wise writing.

    else
        num = blk8(i,j) ; % Otherwise, select row-wise writing.

    end

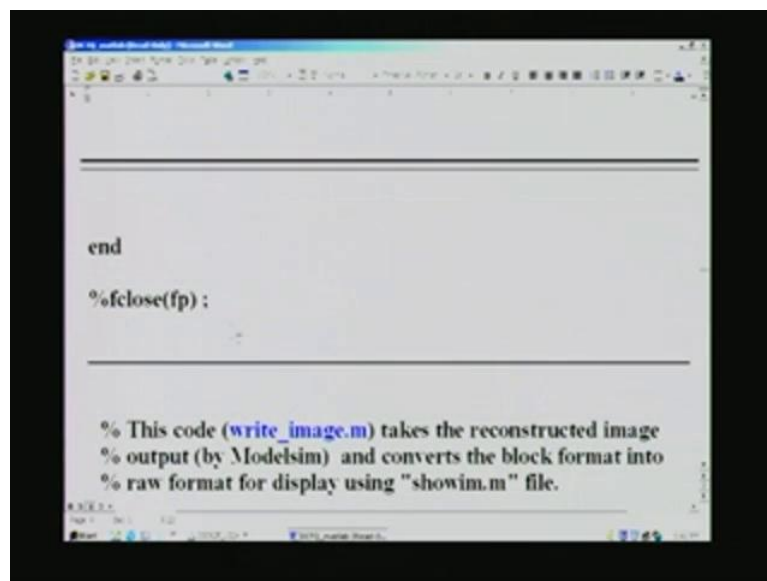
    temp = dec2hex(num,2) ; % Character string - 2 digits

    hexstr = [hexstr,temp] ; % Append into the final 1-D output.
```

In this case, just ignore this. We need to arrange the block in i, j form. This is the form in which we have already input. We need one more counter here, **j [35:37] 1, 8** and i counter also **and here** (Refer Slide Time: 35:43). We have to process one block. That is why it is given as 1 to 8 and **blk** is precisely the same thing. It is arranged as an 8 by 8 matrix and that is copied into a number here. That number will have to be converted

from decimal to hex, because what we get from the raw file is actually in decimal format and so, we have to do that here. Every time we process one pixel, we append it into another hex string 1-D vector here. Finally, **write the same variable into the output file**. This is that output file corresponding to that lena.txt or whatever name we have given. Every time you write, a new line must be taken into account. So, every next line you have to write. What you have here is everything arranged in a single file, pixel after pixel – arranged in that fashion. Write into the output file, each pixel in a new line.

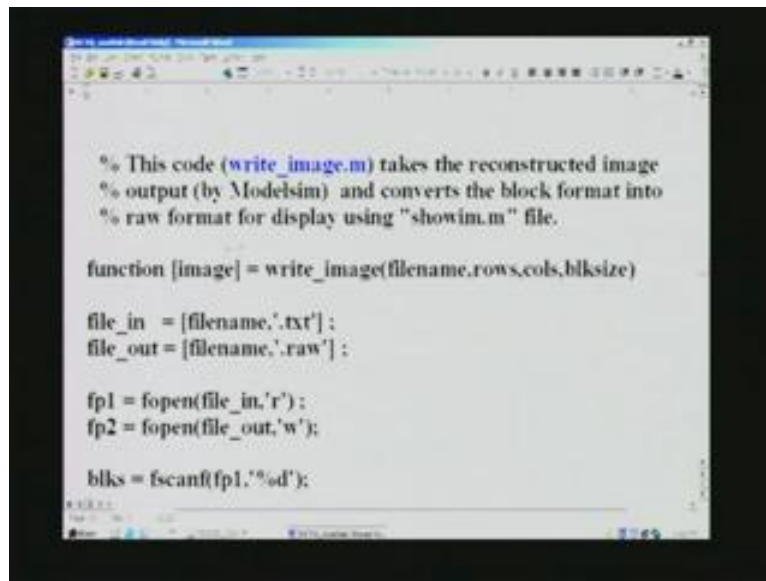
(Refer Slide Time: 36:59)

A screenshot of a MATLAB script editor window. The code is as follows:

```
end  
  
%fclose(fp);  
  
% This code (write_image.m) takes the reconstructed image  
% output (by Modelsim) and converts the block format into  
% raw format for display using "showim.m" file.
```

That completes the writing into the file, block by block. After we have seen verification **in that step**, this is the step for running the DCTQ. After that, you will run the DCTQ, and then its inverse IQIDCT. Finally, that will also give you a reconstructed picture. That will be in block format, which will have to be converted into raster scan format. For that, you need this MATLAB code called write_image file.

(Refer Slide Time: 37:27)

A screenshot of a MATLAB script editor window showing a function named 'write_image'. The code is as follows:

```
% This code (write_image.m) takes the reconstructed image
% output (by Modelsim) and converts the block format into
% raw format for display using "showim.m" file.

function [image] = write_image(filename,rows,cols,blksize)

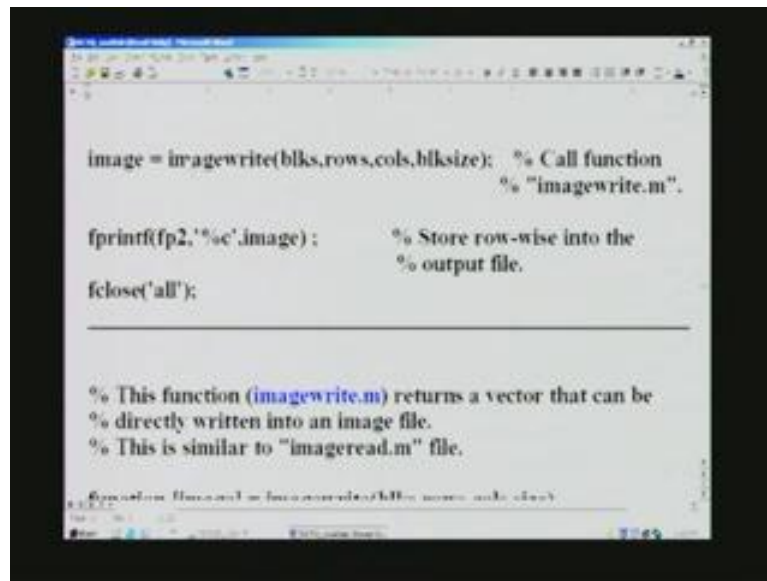
file_in = [filename,'.txt'];
file_out = [filename,'.raw'];

fp1 = fopen(file_in,'r');
fp2 = fopen(file_out,'w');

blks = fscanf(fp1,'%d');
```

It takes the reconstructed image and converts the block format into raw format for displaying using **showim**. This also we have seen before. We will first have a look at this. These are all the usual: filename, whether it is text format or raw format, whether it is read or write and **this is the one we read from that file**. fp1 is the first one here – file_in. If it is lena.txt, that will be read here and put in this variable here in decimal format. Note that it was in hex decimal – IQIDCT was still in binary or we can actually say hex decimal. It was converted earlier, if you remember, in **readimage**. Now, we have to do the conversion into decimal. This is a simple thing, a C-like instruction there.

(Refer Slide Time: 38:24)

A screenshot of a MATLAB script editor window. The code defines a function call and file handling. It includes comments explaining the function's purpose and how the output is written to a file.

```
image = imagewrite(blks,rows,cols,blksize); % Call function
                                         % "imagewrite.m".

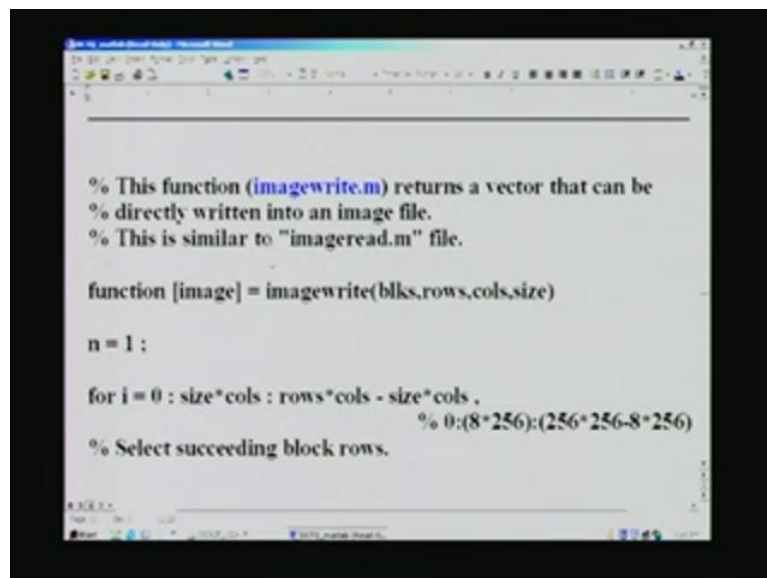
fprintf(fp2,'%c',image); % Store row-wise into the
                        % output file.

fclose('all');
```

```
% This function (imagewrite.m) returns a vector that can be
% directly written into an image file.
% This is similar to "imageread.m" file.
```

Here, we call this imagewrite once again, as we did in imageread earlier and finally, output that result into the final output. You can treat it as a character string. This will be **row-wise into** the raster scan order.

(Refer Slide Time: 38:41)

A screenshot of a MATLAB script editor window showing the definition of the 'imagewrite' function. It includes comments and the function signature.

```
% This function (imagewrite.m) returns a vector that can be
% directly written into an image file.
% This is similar to "imageread.m" file.
```

```
function [image] = imagewrite(blks,rows,cols,size)

n = 1 ;

for i = 0 : size*cols : rows*cols - size*cols ,
    % 0:(8*256):(256*256-8*256)
    % Select succeeding block rows.
```

This is the function that converts that block into raster scan order. This is a function we are going to call. That was a higher level program, which we have just now seen, which calls this one. Here, you will have this raster scan. It is the exact counterpart of imageread, which we have seen before. We can quickly go through this. This is a declaration. You remember this, 0, 8 into 256, the entire row block as I was saying.

We have now to write a block that is confined to... say it starts here, and then goes to the second line, and so on. It will be a block like that. Now, what we have to do is write the first line of the block row first, then go to the second row of the block, and write alongside the first and repeat in the same fashion till you have exhausted all 256. That is what we mean by raster scan. That is what we are doing here. That is the pointer there for that.

(Refer Slide Time: 39:43)

```

for j = i+1 : size : i+cols-size+1 % j is the starting element of
                                % a block.
                                % 1:8:249 - this covers all blocks in a
                                % row.

for k = 0 : 1 : size-1 % 0:1:7 -> Block processing
                    % Row address within a block.

for m = 0 : 1 : size-1 % 0:1:7 - column address within a
                    % block

    image(j - k*cols + m) = blks(n); % 1-D vector, appended.

```

```

                                % row.

for k = 0 : 1 : size-1 % 0:1:7 -> Block processing
                    % Row address within a block.

for m = 0 : 1 : size-1 % 0:1:7 - column address within a
                    % block

    image(j - k*cols + m) = blks(n); % 1-D vector, appended.

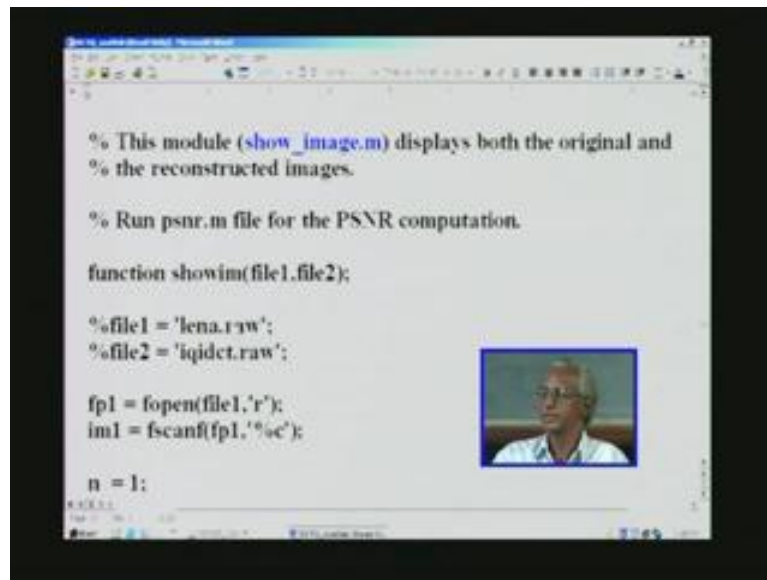
    n = n - 1;
end
end
end
end

```

As usual, we have some more variables, **j** and **0** for block processing. This is the block within the row block processing. The only difference is that this was on the right hand side at that time. Now, what we have is **block-arranged image details**. This is pixel

value. There will be as many data in 1-D vector as there are pixels, but arranged in a block fashion and that is got here. Now, it is arranging, as we have seen before, into a raster scan order. It goes right up to the very last 8 into 256 pixels. That shows that it has converted correctly. Every pixel we advance and this is the pixel counter. These are the corresponding end for the for loops.

(Refer Slide Time: 40:38)



```
% This module (show_image.m) displays both the original and
% the reconstructed images.

% Run psnr.m file for the PSNR computation.

function showim(file1,file2);

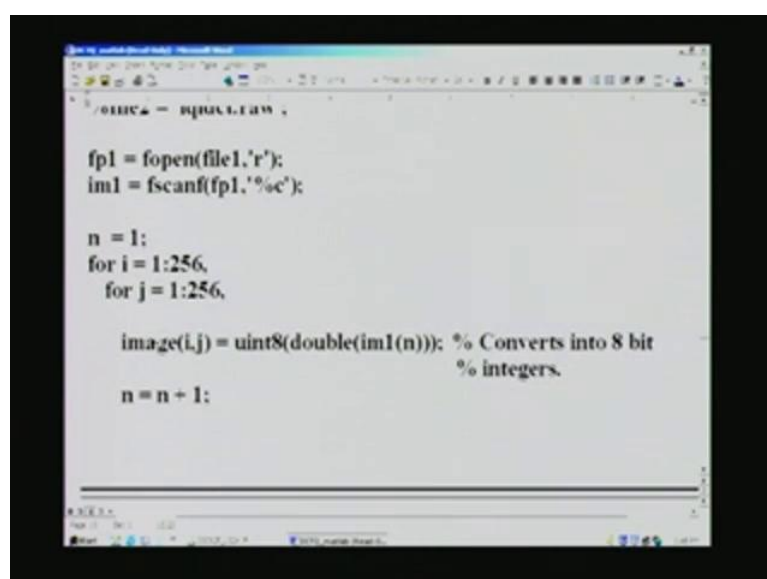
%file1 = 'lena.raw';
%file2 = 'iqidct.raw';

fp1 = fopen(file1,'r');
im1 = fscanf(fp1,'%c');

n = 1;
```

This is the module for showing the image and **that is the image**. We have two files: one is the raw format lena, which is the original and we now have iqidct created by **writeimage** here as a raw file.

(Refer Slide Time: 40:57)



```
image = iqidct.raw ;

fp1 = fopen(file1,'r');
im1 = fscanf(fp1,'%c');

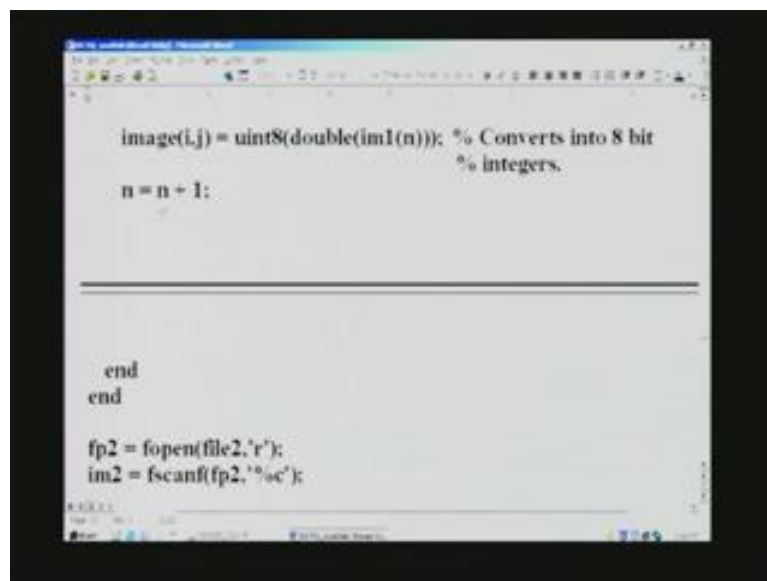
n = 1;
for i = 1:256,
    for j = 1:256,

        image(i,j) = uint8(double(im1(n))); % Converts into 8 bit
        % integers.

        n = n + 1;
```

Here, we have two loops for covering all the pixels. This corresponds to the horizontal width of the image and this corresponds to the height of the image. All the image pixels are covered here and we arrange it in two dimension here. **im1** is **the actual image read from that file**. We have to convert it into double format first and then make it into eight integers. This is the requirement for using the MATLAB command, which we will cover later. So you have to manipulate the conversion.

(Refer Slide Time: 41:37)



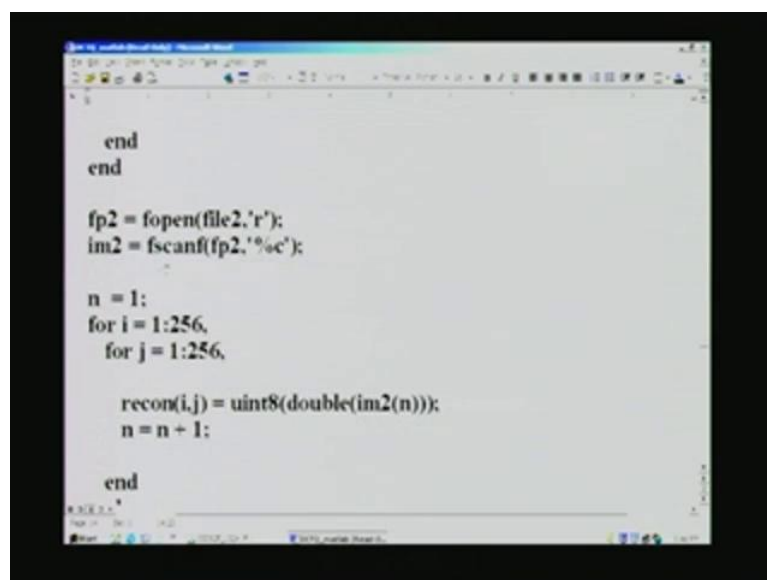
```
image(L,j) = uint8(double(im1(n))); % Converts into 8 bit
                                % integers.
n = n + 1;

end
end

fp2 = fopen(file2,'r');
im2 = fscanf(fp2,'%c');
```

Every pixel you process, you just increment that.

(Refer Slide Time: 41:43)



```
end
end

fp2 = fopen(file2,'r');
im2 = fscanf(fp2,'%c');

n = 1;
for i = 1:256,
    for j = 1:256,

        recon(i,j) = uint8(double(im2(n)));
        n = n + 1;

    end
end
```


Finally, you open an output file here and im2 is that output file. This is for the second one, reconstructed. What we are trying to see is **show image**. So far, we have processed the original file. Now, we are going to process the reconstructed file. That is exactly the same here. Once again, we take the double, then **uint** and then, put it as a reconstruct matrix i, j. i, j varies from 1 to 256.

(Refer Slide Time: 42:15)

```
fp2 = fopen('recon_1.p');
im2 = fscanf(fp2,'%c');

n = 1;
for i = 1:256,
    for j = 1:256,

        recon(i,j) = uint8(double(im2(n)));
        n = n + 1;

    end
end

figure(1);

title('Original Image');
```

```
end
end

figure(1);

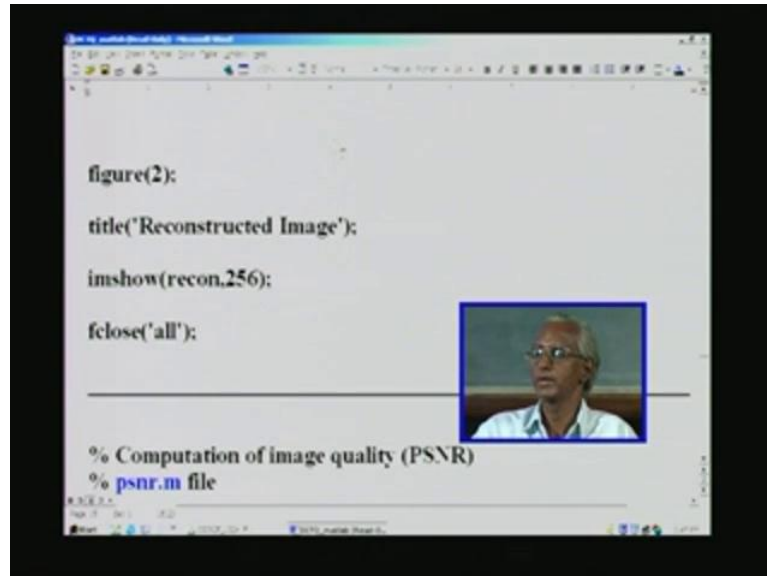
title('Original Image');

imshow(image,256);
```

n plus 1 is to increment the pixel counter. With this, we have processed and got in **original** and reconstructed matrices the actual original image and reconstructed image. What we have to do is just display. For that, there is one command called imshow. This is the format you have to use. This demands that **uint double precision** we have

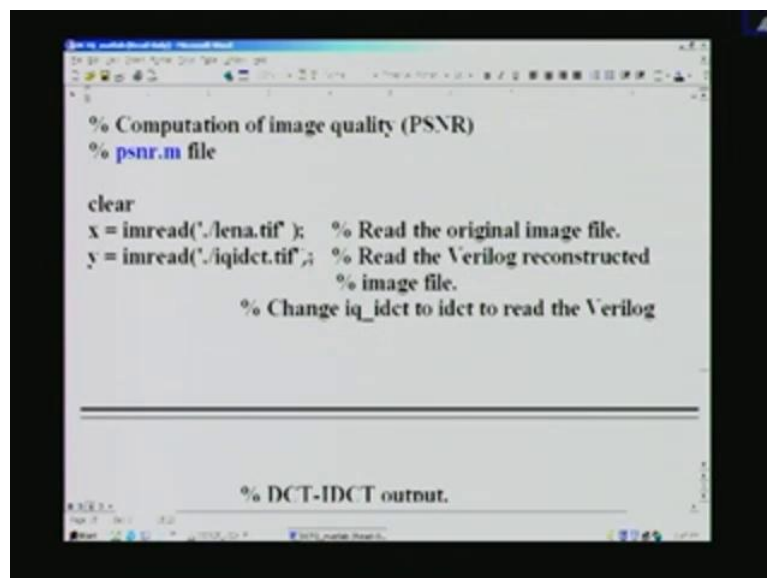
used earlier. We have done that because of this requirement. It shows straightaway on the screen, when you execute.

(Refer Slide Time: 42:45)



You can also give a title if you wish, here. That was for the original file. **figure(2)** is for the reconstructed file here. You can compute the PSNR value, which is very simple.

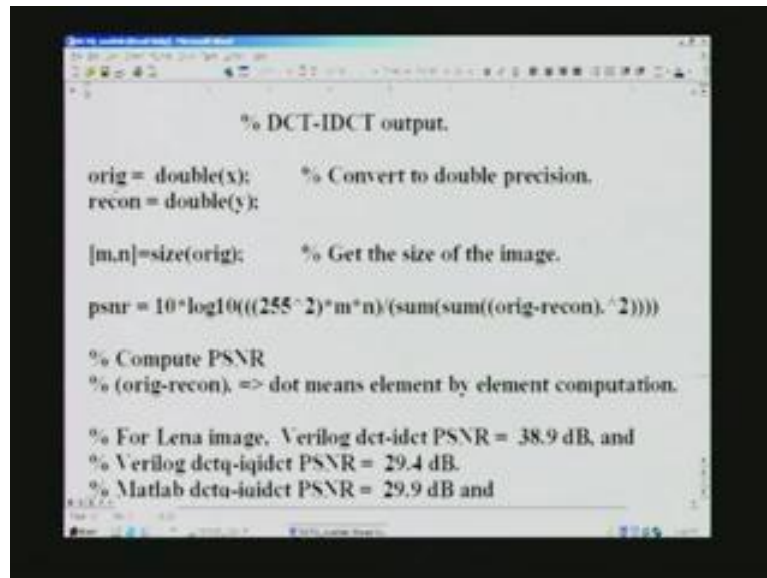
(Refer Slide Time: 42:58)



You need to use XnView and convert that raw file, which we got just now, into a TIF file. Only then, you can execute this. There are two files here, **for which we need to**

compute. This is the reconstructed file, this is the original file, and we need to compute the PSNR.

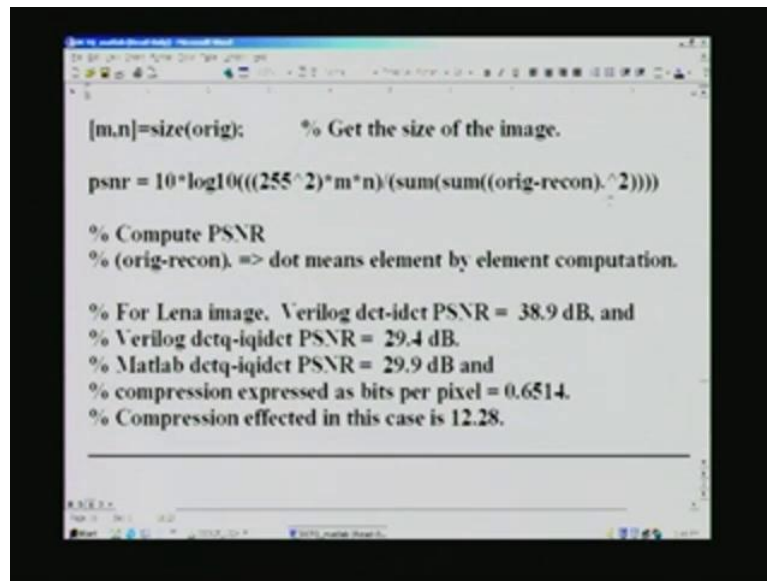
(Refer Slide Time: 43:19)

A screenshot of a MATLAB script window showing code for calculating PSNR. The code includes comments and mathematical expressions for converting images to double precision, getting their sizes, and computing the PSNR using a logarithmic formula. It also provides example PSNR values for the Lena image using Verilog and Matlab.

```
% DCT-IDCT output.  
orig = double(x);    % Convert to double precision.  
recon = double(y);  
  
[m,n]=size(orig);    % Get the size of the image.  
  
psnr = 10*log10(((255^2)*m*n)/(sum(sum((orig-recon).^2))))  
  
% Compute PSNR  
% (orig-recon). => dot means element by element computation.  
  
% For Lena image, Verilog dct-idct PSNR = 38.9 dB, and  
% Verilog dctq-iquidct PSNR = 29.4 dB.  
% Matlab dcta-iuidct PSNR = 29.9 dB and
```

We have two variables **orig** and **recon** and we use double precision here. **m, n** is the size of the actual image. PSNR computation is governed by this expression, which means $10 \log_{10}$ and then, on the numerator, you have 255 square multiplied by m into n , which is nothing but the picture size (in this case, 256 by 256 for lena) and divided by the denominator here, which is nothing other than pixel-wise intensity between the original and reconstructed figure. For every pixel difference, you get rid of the sign by squaring it and finally summing for all the pixels – that is the denominator. If you do this, you get the PSNR value in dB (decibels).

(Refer Slide Time: 44:10)



```
[m,n]=size(orig);    % Get the size of the image.

psnr = 10*log10(((255^2)*m*n)/(sum(sum((orig-recon).^2))))

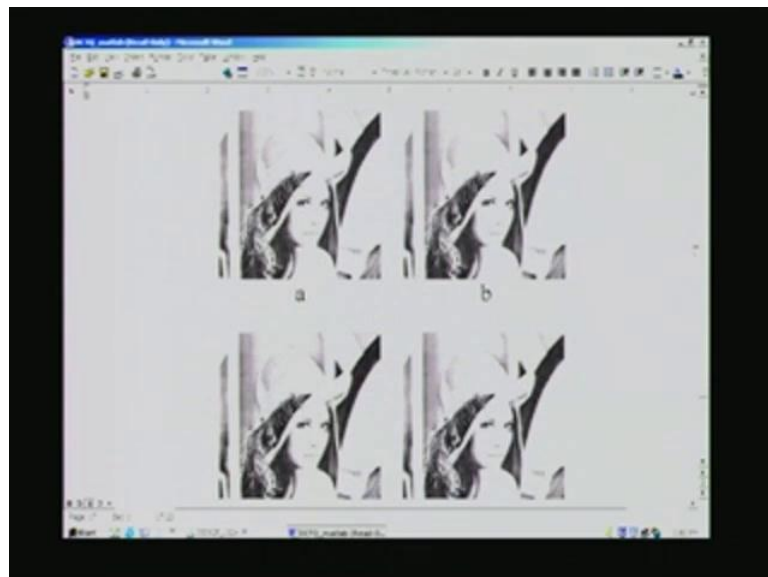
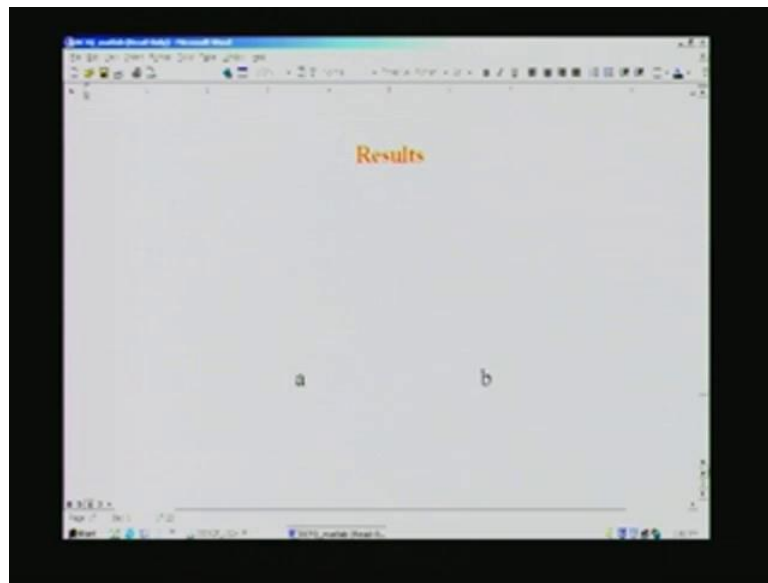
% Compute PSNR
% (orig-recon). => dot means element by element computation.

% For Lena image. Verilog dct-idct PSNR = 38.9 dB, and
% Verilog dctq-iqidct PSNR = 29.4 dB.
% Matlab dctq-iqidct PSNR = 29.9 dB and
% compression expressed as bits per pixel = 0.6514.
% Compression effected in this case is 12.28.
```

Dot means element to element. It means that this is not the entire file. For example, if you take a **lena image**, I have also done DCT as well as IDCT cores, which needs change from **DCTQ-IQIDCT**, in which case you get a very high PSNR value. It is 38.9 dB. If you apply DCTQ-IQIDCT in Verilog for this lena image, I get a PSNR of 29.4, which is very close to MATLAB DCTQ-IQIDCT and serves as the reference for checking this. This is not explained in the present course – it is left as an exercise for you to evaluate DCTQ. It is merely manipulation of matrices. It is very easy for you to do that.

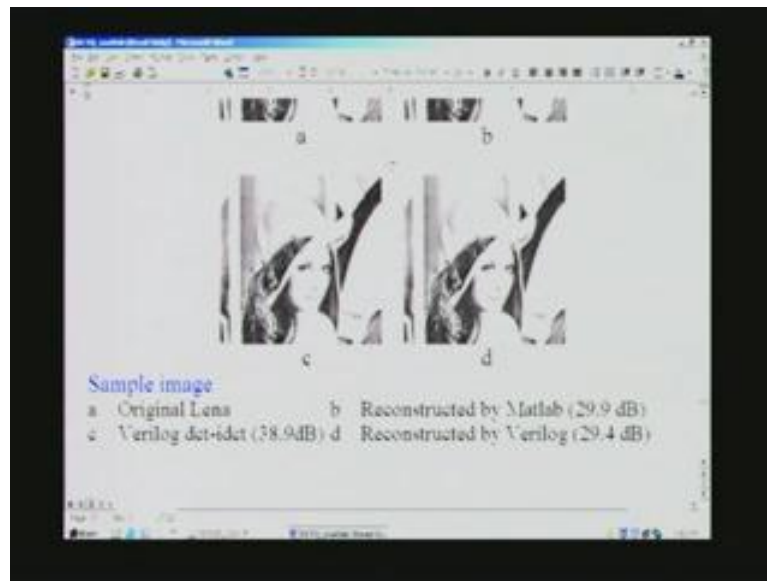
Compression effected will also have to be incorporated in that. Unless you do the VLC course, you cannot have that – that is covered in that course, **from which** you get this compression ratio to be 12.28 for this particular image. Now, let us have a look at the results for the image.

(Refer Slide Time: 45:25)



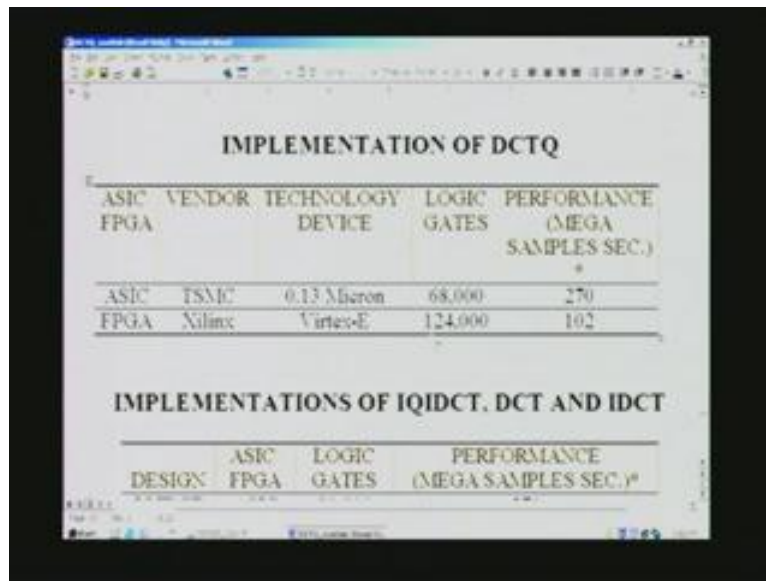
This is the original image. What you see here is the actual original image and from MATLAB, after DCTQ-IQIDCT, you get this image. It may appear to be close. I do not know how it appears on the TV monitor. Is there a difference between the two? I am not sure. You can probably see some quality suffering here. This is DCTQ-IQIDCT, which we report as 29.4 decibels.

(Refer Slide Time: 45:58)



This is Verilog-created DCT–IDCT. You see a very high decibel for this. It is as good as the original. A good quality picture is supposed to be 30 dB and above; a bad quality one is 25dB and below; you will get indistinguishable quality if it is 35 dB. That is the thumb rule for finding out how the picture is, apart from the visual findings. This is reconstructed by Verilog. This is actually DCTQ–IQIDCT applied here. You can see that this is very close to the reference from MATLAB. Some say this is better than that, but opinions are divided. The point is that you have a fairly good image and you are able to process at a very high rate. That is what you have here.

(Refer Slide Time: 47:01)

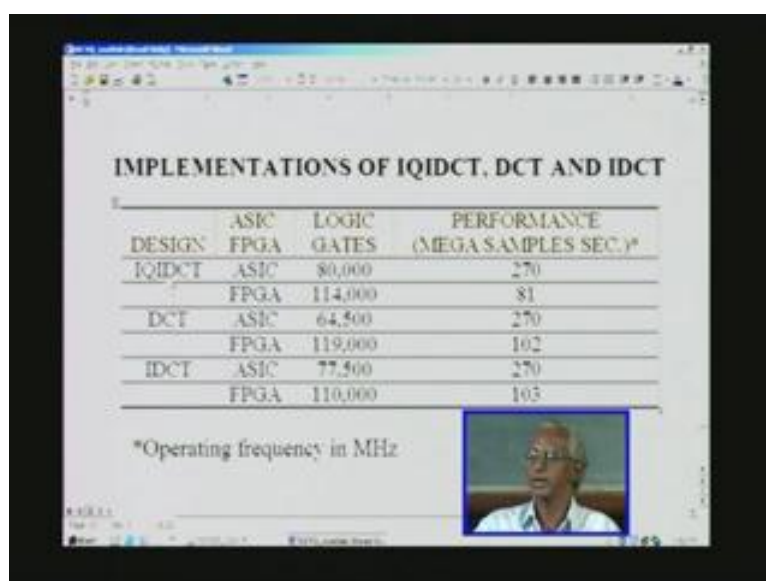


ASIC	VENDOR	TECHNOLOGY	LOGIC	PERFORMANCE
FPGA		DEVICE	GATES	(MEGA SAMPLES SEC.)
ASIC	TSMC	0.13 Micron	68,000	270
FPGA	Xilinx	Virtex-E	124,000	102

ASIC	LOGIC	PERFORMANCE	
DESIGN	FPGA	GATES	(MEGA SAMPLES SEC.)*
IQIDCT	ASIC	80,000	270
	FPGA	114,000	81
DCT	ASIC	64,500	270
	FPGA	119,000	102
IDCT	ASIC	77,500	270
	FPGA	110,000	103


Now, let us have a look at the implementations already done for DCTQ. ASIC has also been done as also the FPGA, which we have seen to be 124,000 for DCTQ gates count. If you extrapolate the area that is reported in ASIC design in 0.13 micron technology, you have only 68,000 gates for the same design. The very same design works on the ASIC platform on **Synopsys Design Compiler**. It works at 270 Megahertz, whereas Xilinx we have seen to be 102 Megahertz after Xilinx place and route.

(Refer Slide Time: 47:36)



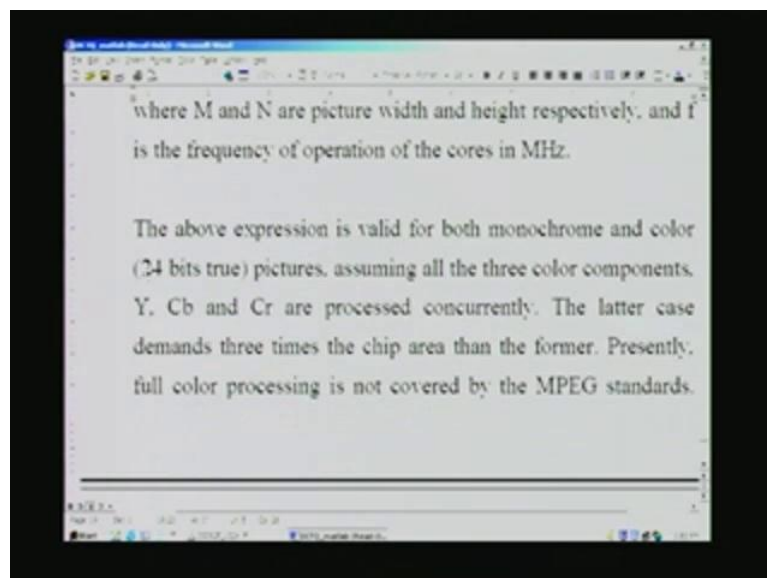
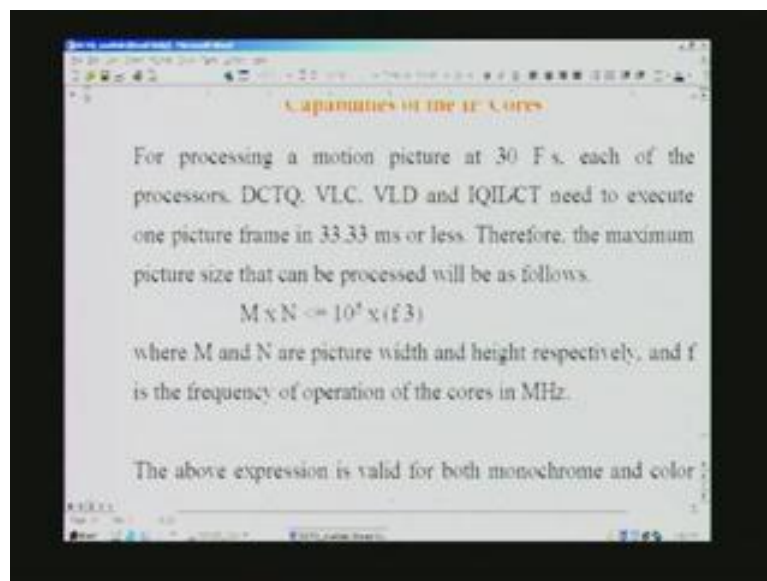
ASIC	LOGIC	PERFORMANCE	
DESIGN	FPGA	GATES	(MEGA SAMPLES SEC.)*
IQIDCT	ASIC	80,000	270
	FPGA	114,000	81
DCT	ASIC	64,500	270
	FPGA	119,000	102
IDCT	ASIC	77,500	270
	FPGA	110,000	103

*Operating frequency in MHz



The same thing was done for the other core, for example, IQIDCT, DCT and IDCT. The results are here. IQIDCT has 80,000, whereas FPGA has 114,000. The speed is hit here. This governs the speed of the overall system, because at the decoder end, we need IQIDCT. If it is motion picture processing, we need it even at the encoder stage. DCTQ is processed at 102 Megahertz, but this is the slower of the two – 81. Let us see for that. Similarly for DCT and IDCT.

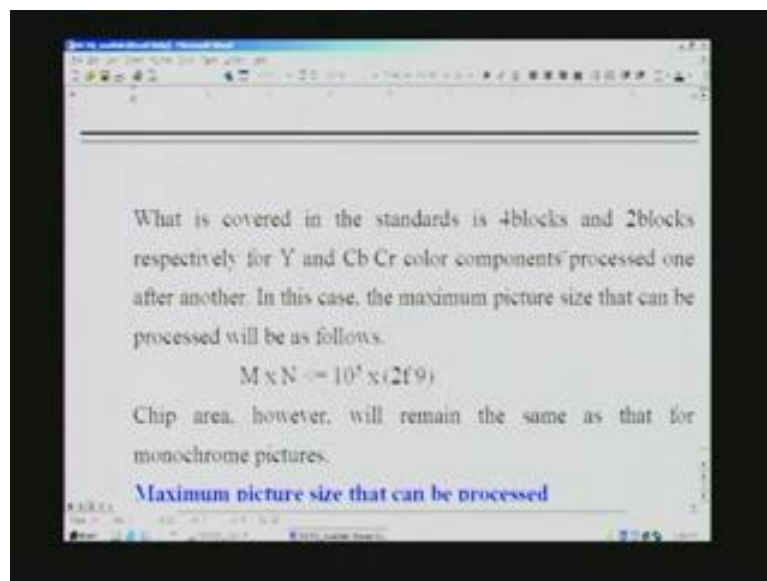
(Refer Slide Time: 48:11)



These are all the capabilities of IP cores that we have developed. For processing a motion picture at 30 frames per second, each of the processors, DCTQ, VLC, VLD IQIDCT need to execute one picture frame in 33.33 millisecond or less. Therefore, the

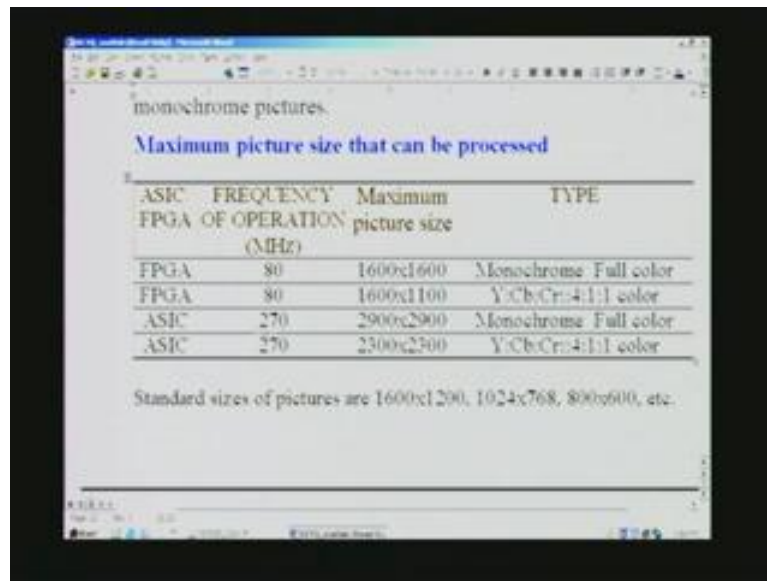
picture size is governed by this expression, which you can very easily derive. Here, f is the frequency of operation in Megahertz. M and N are the width and height of the picture, respectively. If it is true color processing, it is valid for both monochrome and color (24 bits true) pictures, assuming all the three color components Y , Cb and Cr are processed concurrently. If you process concurrently, you need three times the chip area but you will get a very high speed. The latter case demands three times the chip area than the former. Presently, full color processing is not covered by the MPEG standards.

(Refer Slide Time: 49:04)



In MPEG standard, what is covered is... We have seen this before – four blocks for Y luminance and two blocks for Cb and Cr . This is only six blocks instead of processing for full color, wherein we need 4 into 3 times – this is just half that requirement. You can process much higher. As a result, the picture that you can process in this scheme will be governed by instead of 3 there, it is 4.5 here. So 3 to 4.5 is the difference between monochrome and this, because here it is six blocks, there it is four blocks.

(Refer Slide Time: 49:49)



monochrome pictures.

Maximum picture size that can be processed

ASIC FPGA OF OPERATION	FREQUENCY (MHz)	Maximum picture size	TYPE
FPGA	80	1600x1600	Monochrome Full color
FPGA	80	1600x1100	Y:Cb:Cr:4:1:1 color
ASIC	270	2900x2900	Monochrome Full color
ASIC	270	2300x2300	Y:Cb:Cr:4:1:1 color


Standard sizes of pictures are 1600x1200, 1024x768, 800x600, etc.

The maximum picture that you can process is tabulated here. If you take FPGA implementation, I have taken 81 Megahertz as 80 Megahertz here for this case. ASIC is 270 Megahertz and the maximum picture size that you can process is 1600 by 1600. It is a very high resolution. Unfortunately, we do not have other hardware matching this speed, **in spite of low frequency for IQIDCT**. In fact, I got 100 Megahertz a couple of years back. I am not able to reason out why I got 80 – it actually went to **102**. Anyway, I will take the worst case here.

If you take ASIC, it will be a huge resolution that you can process, of the order mentioned here. If it is monochrome or full color, this is the case. **If it is as per the standard in this ratio 4, then one block, one block – six blocks instead of four blocks for monochrome.** Actually, the standard sizes come in this format: 1600 by 1200 and 1024 by 768, which is called XGA format and 800 by 600 is called **SVGA format**. QCIF is 352 by 288. This completes the DCTQ design. Thank you.

(Refer Slide Time: 51:08)





Verification of Verilog DCTQ
- IQIDCT Cores
Matlab Codes for Pre-processing
and Post-processing of an Image
Results - Original and
Reconstructed Image Example
Implementation Results of DCTQ,
IQIDCT, DCT and IDCT Cores on
FPGA/ASIC
Capabilities of IP Cores