

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

Lecture No. 46
System Design Examples (Continued)

(Refer Slide Time: 01:46)



We were looking into the design of DCTQ. We have covered a novel algorithm for implementation of the same with a view to speed up the throughput. We also considered the architecture earlier. Now, we are about to start the Verilog codes for

the same design. Before we start this design, let us have a look at the image, how it is going to be processed, what the resolutions are, what does color mean and so on.

(Refer Slide Time: 02:44)



What you see here is a very high resolution picture of a bird. You can see that the resolution is 1024 by 768 and it is true color. You may not be in a position to read here and I am reading out for your sake. It is written 24 here. This stands for true color. That means to say there are three color components in this. Depending upon the type of picture that you have, there are different formats. For example, bitmap pixel, which is what you see right now, with an extension of .bmp.

There are other files called raw. What we have seen earlier as a gamma-corrected RGB will be available in a raw file. There are other files like JPEG, which are compressed files, whereas bmp and raw files are not compressed as such. We also have one more type called TIF file, which we will be requiring for MATLAB and other **codes**, which we will be describing later on in order that we may use it for generating these pictures in a form that can be an input to the simulator. You see that it is a very high resolution here. Even if I increase the zoom, you still see a very good picture.

(Refer Slide Time: 04:07)



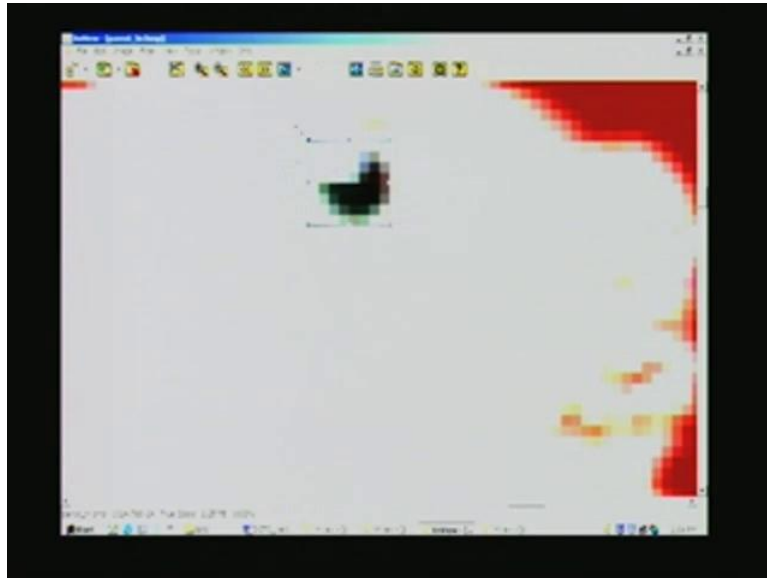
Similarly, we have a special software called XnView, the site of which is given towards the end of this lecture – you can download from there. Using this, you can have resize and **they are all** menu driven. You can resize and do cropping so that we can extract from the picture of 1024 by 768 pixels and we can get 256 by 256. That is what you will be seeing here.

(Refer Slide Time: 04:43)



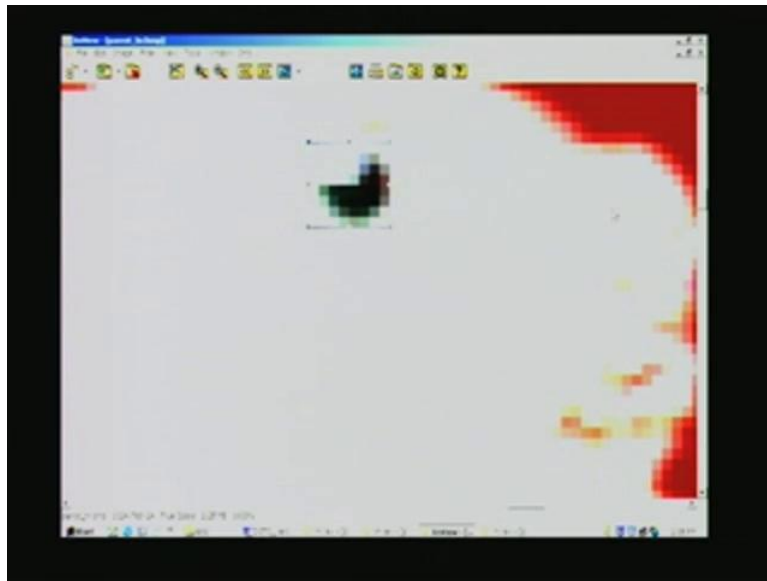
You can see that the picture is not that good and you can see some information being lost. This is because it is a 256 by 256 image, although it is continuing to be true color here.

(Refer Slide Time: 05:03)



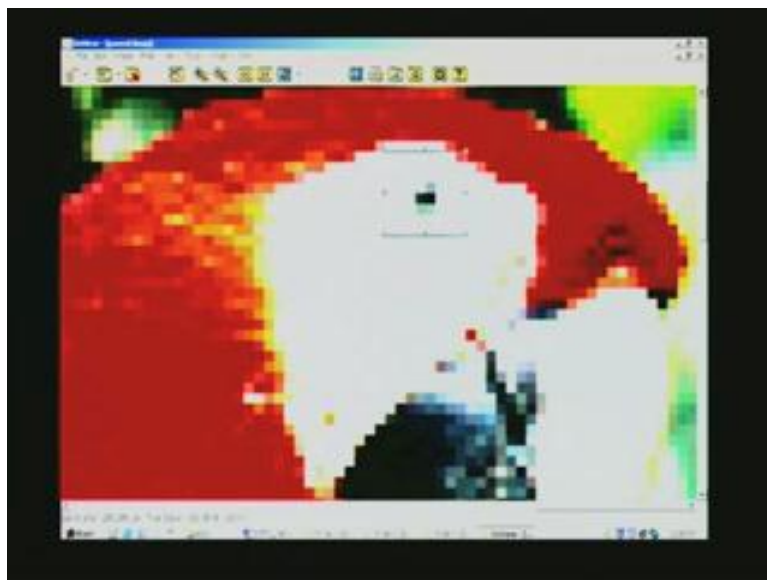
If you zoom the first high-resolution picture, you will get all squares. Each of these squares is known as a pixel. A collection of 64 pixels or 8 by 8 pixels we take in order to process DCTQ and that is referred to as a block. A block consists of 8 by 8 pixels. If you examine each pixel very closely, you will see that one particular square has only one common intensity all through. This is because this is a pixel and if you zoom further, you cannot go deeper into this – deeper than a pixel, you cannot see anything. If it is a true color, this will be a 24 bits representation. For each of the three color components, you need 8 bits. The software can also help you in converting this into a monochrome image should you desire one. This is for 1024 high resolution into 768 resolution.

(Refer Slide Time: 06:04)



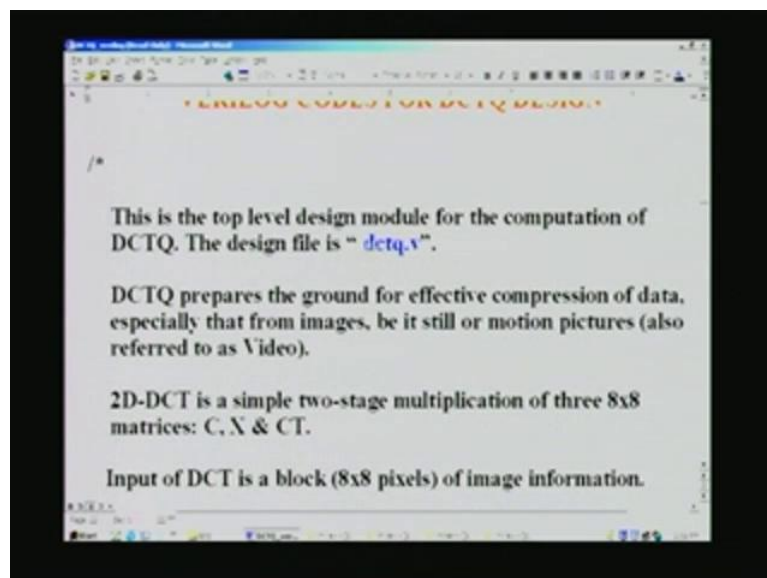
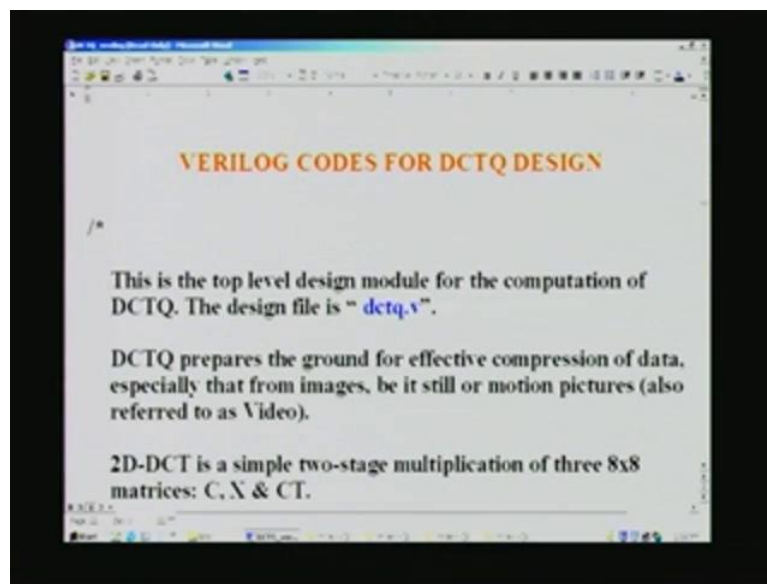
Note that you cannot make out anything here, except perhaps the eye of the parrot. If you see in the second one, which is also a zoomed version, you can see the parrot here to some extent. Once again, you can see the pixelations here and there is a mark for the 8 by 8 pixel block. You can see a much wider area covered.

(Refer Slide Time: 06:28)



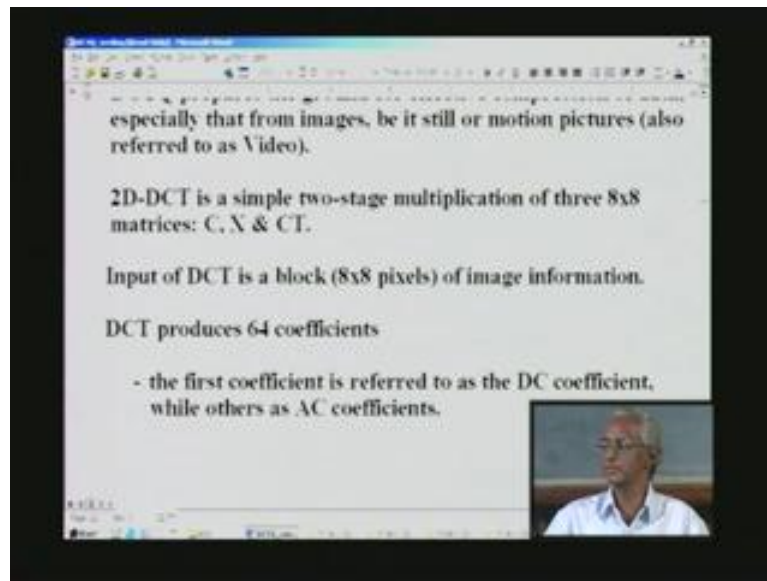
That is because it is for a lower resolution and the resolution is 256 by 256. Once again, it is true color. Now, we will continue with the Verilog design.

(Refer Slide Time: 06:38)



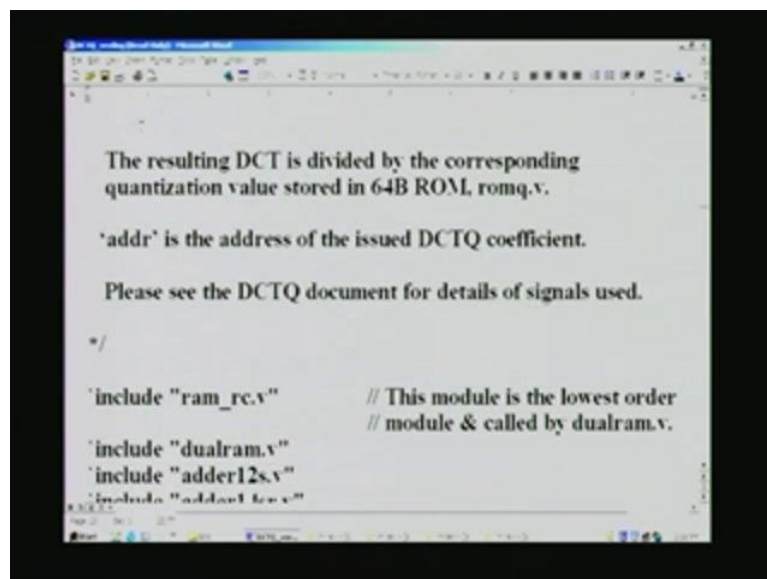
I will read this out. To start with, this is the top-level design module for the computation of DCTQ. The design file is dctq.v. DCTQ prepares the ground for effective compression of data, especially that from images, whether it is still or motion pictures. Motion pictures are also referred to as video sequence. 2D-DCT is a simple two-stage multiplication of three 8 by 8 matrices: C, X and CT, which you have already seen earlier. Input for this DCT is a block, which we have seen in the image just now. It is 8 by 8 pixels as mentioned before.

(Refer Slide Time: 07:16)



DCT produces 64 coefficients. The first coefficient is known as DC coefficient, while the other 63 are known as AC coefficients. We have also seen this earlier.

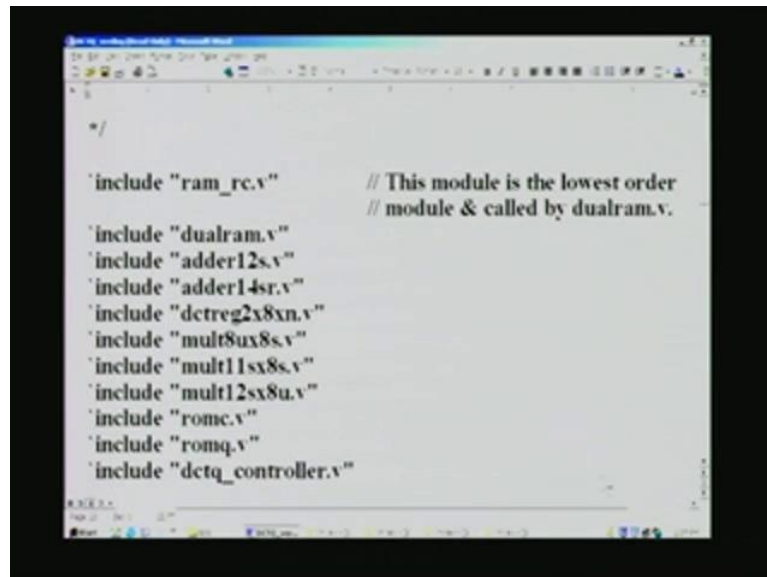
(Refer Slide Time: 07:29)



Ming you, what we are doing is we are looking into the actual DCTQ design file, which has a .v extension. dctq.v is the actual design file. The need for putting all these as comments in this is it is a ready reckoner for a designer or anyone else who is verifying the design, so that he need not go and look into the massive document that we have covered earlier. That is why it says here 'Please see the DCTQ document for details' should you require more details.

The resulting DCT is divided by the corresponding quantization value stored in the 64 byte ROM romq.v. addr serves as the address for the DCTQ coefficient. Address equal to 0 would correspond to DC coefficient, whereas all others are AC coefficients up to 63.

(Refer Slide Time: 08:23)



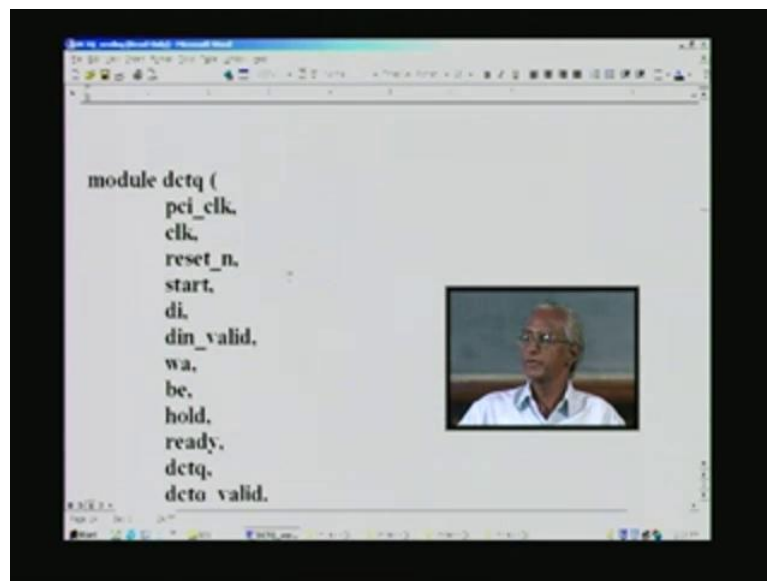
```
*/
include "ram_rc.v" // This module is the lowest order
// module & called by dualram.v.
include "dualram.v"
include "adder12s.v"
include "adder14sr.v"
include "dctreg2x8xn.v"
include "mult8ux8s.v"
include "mult11s8s.v"
include "mult12sx8u.v"
include "romc.v"
include "romq.v"
include "dctq_controller.v"
```

Earlier, if you remember, we have seen the bottom-up approach of the design. Here, we will see the reverse, that is, top-down approach of the design in the sense that in the earlier bottom-up approach, we started with the lowest module such as ram_rc and then, we went on to a higher level, which invoked this ram_rc and that higher level is dual RAM. Like that, we went on from the bottom to the top. Now, what we see is the top is the very design file. dctq.v is the top design file that we are seeing right now. This will merely invoke different modules or sub-modules down the hierarchy. What are included are precisely those modules.

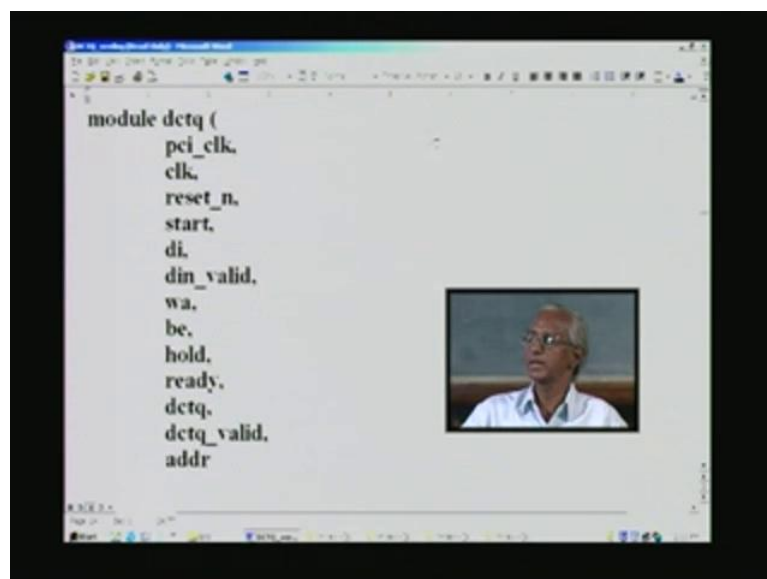
For example, ram_rc, dualram, the design of which we have covered already and we have covered also adder12s and 14sr was given as an assignment to you. What we are yet to see is this dctreg in order to store the partial products of a CX matrix computation, if you recollect our architecture shown before. We have also seen the design of the multiplier for 8u into 8s where u stands for unsigned, s for signed. As usual, we also gave assignments for two other multipliers. That is precisely what we shall use here.

We also covered the design of romc as well as CT. These are the C and CT matrices for the evaluation of DCT, which requires the multiplication of three matrices: C, X and CT. We also had a quantization table. It is actually the inverse of the quantization table and we have seen this earlier. We are yet to see the next one. This is the last module, which is called the controller. Any design is dependent upon how good a controller design is. It is always a good practice to have a separate module for the controller as we see here.

(Refer Slide Time: 10:39)



```
module dctq (  
    pci_clk,  
    clk,  
    reset_n,  
    start,  
    di,  
    din_valid,  
    wa,  
    be,  
    hold,  
    ready,  
    dctq,  
    dcto_valid.
```



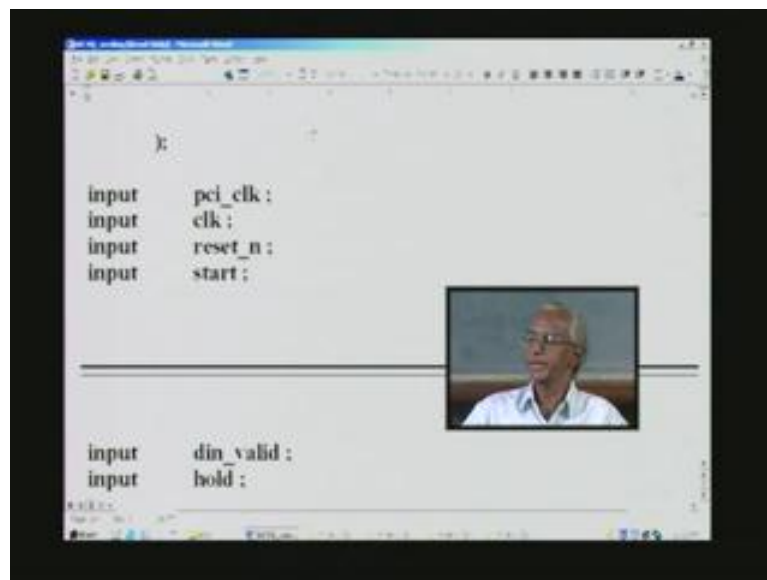
```
module dctq (  
    pci_clk,  
    clk,  
    reset_n,  
    start,  
    di,  
    din_valid,  
    wa,  
    be,  
    hold,  
    ready,  
    dctq,  
    dctq_valid,  
    addr
```

This is the top-level module for DCTQ. Before we go into the details, let us point out one fact. We know that FPGA and ASIC based designs are massively parallel and

highly pipelined, resulting in very fast implementations. This is in contrast to the conventional approach of using a Pentium or DSP, using either C language or assembly language, which are mainly sequential in nature. If you have a computation of the order of n^3 or even more, n is 8 in this case of DCTQ evaluation, a huge number of computations are involved. Naturally, these sequential machines such as Pentium and DSPs will take quite a lot of time in order to process even a single frame.

In order to get a real-time image, you may have to scale down the picture resolution may be to 352 by 288, which you write as a CD in MPEG-1 format. This course is probably going to be on such CDs. You will be seeing 352 by 288, which is called QCIF. Of 288, I am not sure – you can crosscheck in any text book. This means to say we can only process a very low resolution picture using Pentium and so on. If you really want a sophisticated system, you need to go for either FPGA or ASIC based designs. This is the top module we have declared here. DCTQ is the design and these are all the signals that you have seen before, in the form of a block diagram shown earlier. I do not have to go into all the details. The final output is going to be DCTQ. When it is valid is indicated by this signal and its address is also given by this signal.

(Refer Slide Time: 12:50)



What you have put there we declare as either inputs or outputs and that is precisely what you see in the next few instructions.


(Refer Slide Time: 12:57)

```
input    din_valid ;
input    hold ;

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

output    ready ;
output [8:0] dctq ;
output    dctq_valid ;
output [5:0] addr ;

wire     ready ;
wire [8:0] dctq ;
```




```
input [1:0] oc ;

output    ready ;
output [8:0] dctq ;
output    dctq_valid ;
output [5:0] addr ;

wire     ready ;
wire [8:0] dctq ;
wire    dctq_valid ;
wire [5:0] addr ;

wire    rnw ;
wire    encnt2 ;

wire [15:0] result1 ;
```



Along with the size of these signals you see there. For example, address is 6 bits, DCTQ is 9 bits, and DCT will be 12 bits and so on.

(Refer Slide Time: 13:05)

```
wire [8:0] dctq ;
wire      dctq_valid ;
wire [5:0] addr ;

wire      rnw ;
wire      enct2 ;

wire [15:0] result1 ;

wire [15:0] result2 ;
wire [14:0] result3 ;
```

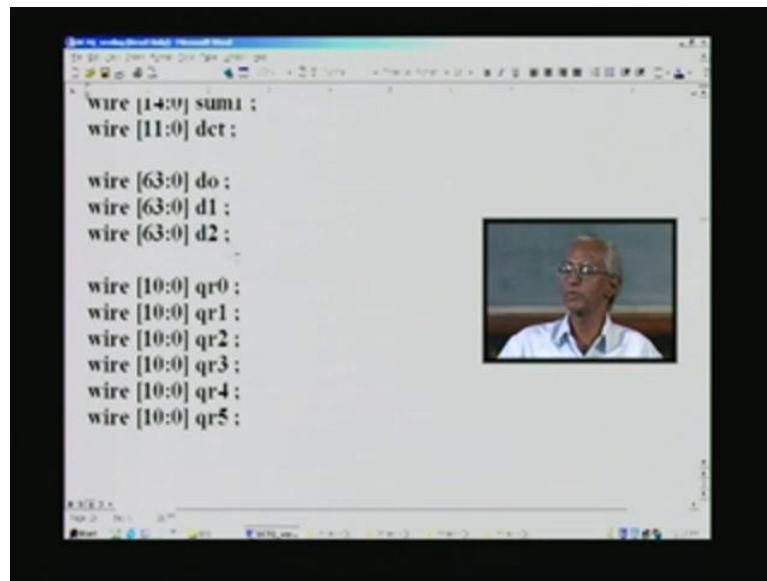
```
wire [15:0] result2 ;
wire [15:0] result3 ;
wire [15:0] result4 ;
wire [15:0] result5 ;
wire [15:0] result6 ;
wire [15:0] result7 ;
wire [15:0] result8 ;

wire [14:0] sum1 ;
wire [11:0] det ;

wire [63:0] det ;
```

If you remember, in the first CX multiplication, we used eight multipliers. They produced the output result1 and they are 16 bits in width. They are listed here as result1 to result8. Finally, we added all this in order to create **one [13:45]** and that is declared here. This is by means of an adder – 12s, if I recollect. There is also another DCT that we need, which is of size 12 bits. Sum is only 15 bits – although it is high precision, it is being truncated here. We will look into the details later.

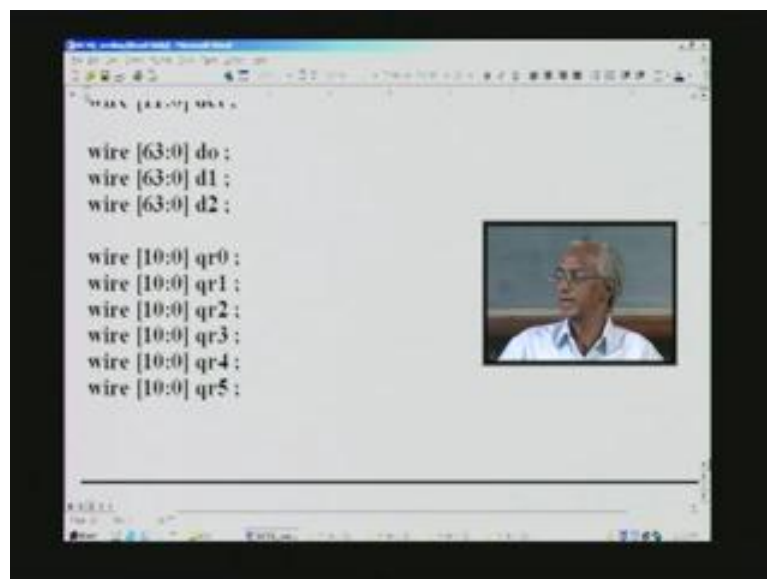
(Refer Slide Time: 13:58)



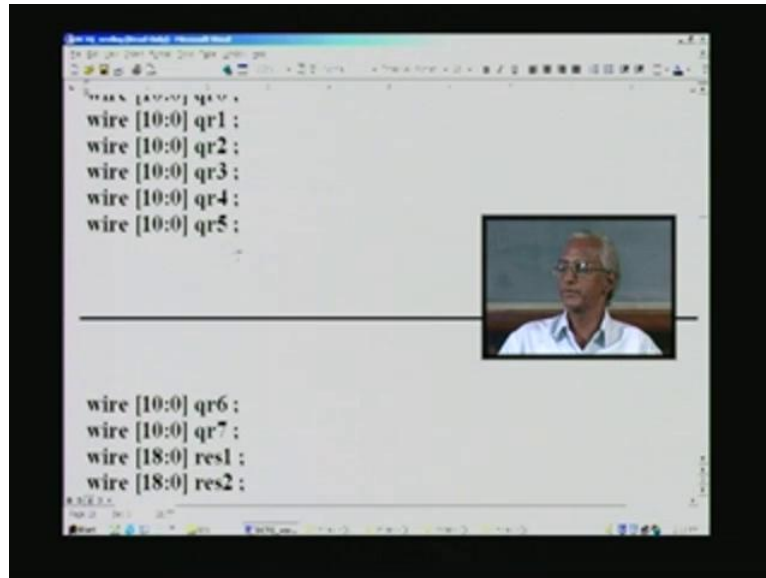
```
wire [14:0] sum1 ;  
wire [11:0] d1 ;  
  
wire [63:0] do ;  
wire [63:0] d1 ;  
wire [63:0] d2 ;  
  
wire [10:0] qr0 ;  
wire [10:0] qr1 ;  
wire [10:0] qr2 ;  
wire [10:0] qr3 ;  
wire [10:0] qr4 ;  
wire [10:0] qr5 ;
```

We also used a dual RAM and there are two outputs, namely d1 and d2 and they have got to be declared as a wire. We will come to why it is wire a little later on.

(Refer Slide Time: 14:02)

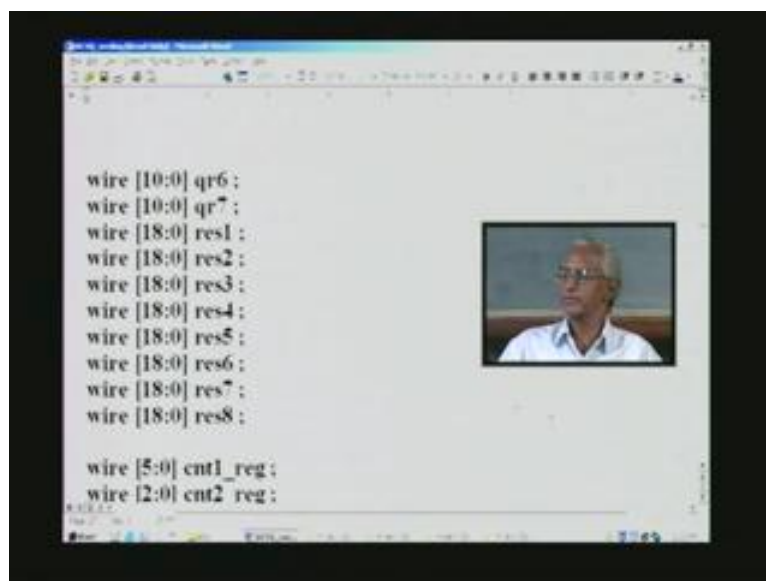


```
wire [63:0] do ;  
wire [63:0] d1 ;  
wire [63:0] d2 ;  
  
wire [10:0] qr0 ;  
wire [10:0] qr1 ;  
wire [10:0] qr2 ;  
wire [10:0] qr3 ;  
wire [10:0] qr4 ;  
wire [10:0] qr5 ;
```



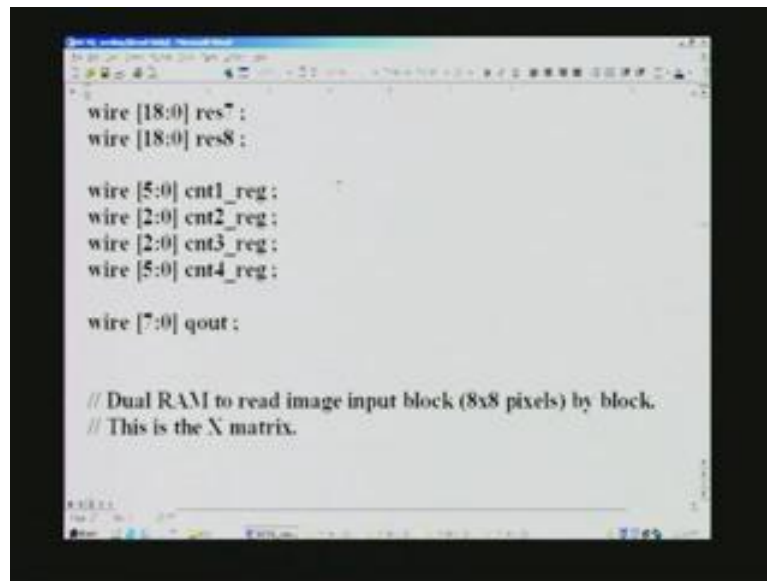
We also need to have partial products stored. For these partial products, we had earlier used **P0 [14:16] etc.** You may regard it as qr0, qr1 and so on. Row-wise, it will be different and size is 11 bits.

(Refer Slide Time: 14:31)



At the second stage, we take these partial products and multiply with CT to produce independent multiplied results. They are all indicated as res1 through res8. Now, notice that the bit precision has improved – that is, increased quite a bit.

(Refer Slide Time: 14:52)



```
wire [18:0] res7 ;
wire [18:0] res8 ;

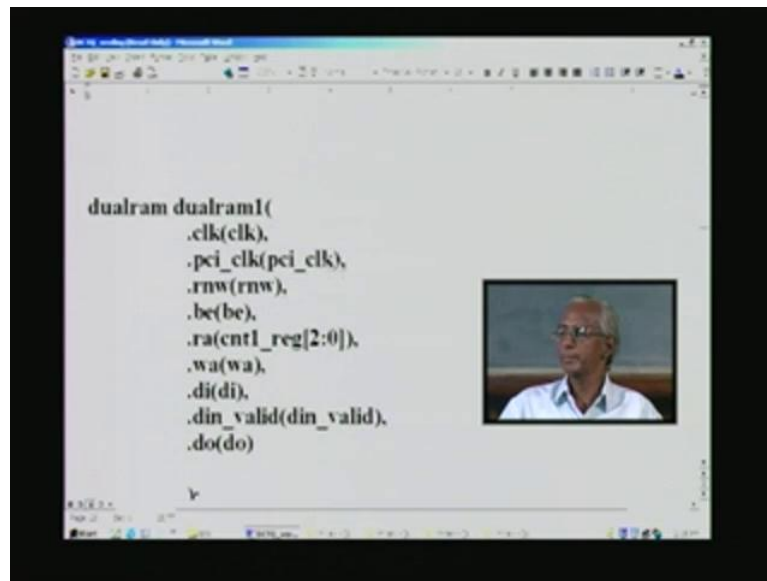
wire [5:0] cnt1_reg ;
wire [2:0] cnt2_reg ;
wire [2:0] cnt3_reg ;
wire [5:0] cnt4_reg ;

wire [7:0] qout ;

// Dual RAM to read image input block (8x8 pixels) by block.
// This is the X matrix.
```

Then, we will be needing some counters in order to realize address generation as well as address for the ROMs, RAMs, etc., which we use in our architecture. These are all generated in the controller. For a quantization table, we have a ROM whose output is qout. An important thing: a good design practice is that you should not have any logic on the main design, that is, the top design. What you should have is only calling all submodules here. For example, dual RAM is being called here and that is what is indicated here – dual RAM to read the image input block (8 by 8 pixels); it is read block by block. This is the X matrix that we have. One row of a particular block is being read at one time. **Every clock, for that matter, we keep reading every row in a block.**

(Refer Slide Time: 15:57)



```

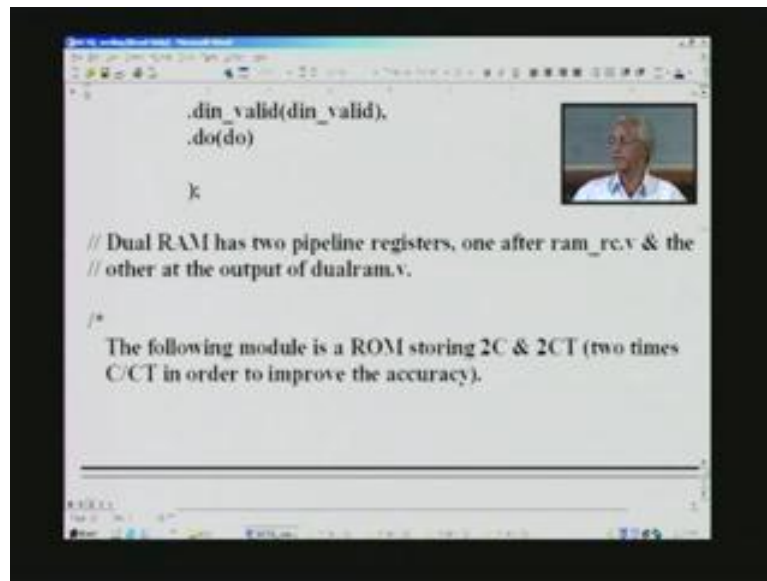
dualram dualram1(
    .clk(clk),
    .pci_clk(pci_clk),
    .rnw(rnw),
    .be(be),
    .ra(cnt1_reg[2:0]),
    .wa(wa),
    .di(di),
    .din_valid(din_valid),
    .do(do)
)

```

In order to **read that** into the RAM, we have to invoke a dual RAM here. This is the instantiation and these are all the signals that you are already familiar with. The only difference is that in dual RAM, we have basically two modes: we can be in either read-only mode or write-only mode. To start with, it will be in write-only mode and this rnw signal is what is being used for that. While in read mode, we execute the DCTQ. This read address will be delivered from the controller by using a counter. The same counter is also used for ROM later on – ROM for C and **CT matrices**.

Note that only three-bit LSB is used for the X matrix. This is because in the first computation of CX, we take a row of C, then take a column of X matrix and repeat the same for other columns of the X matrix. That means we advance to the next row of C only after processing eight columns of X. That is the reason why this is put as LSB. The MSB for the counter, that is, 3, 4 and 5 will be for the C matrix, which we will be seeing later on. The final output **is d0**. This is applied to the second stage, which we will cover shortly.

(Refer Slide Time: 17:28)



```
.din_valid(din_valid),
.do(do)

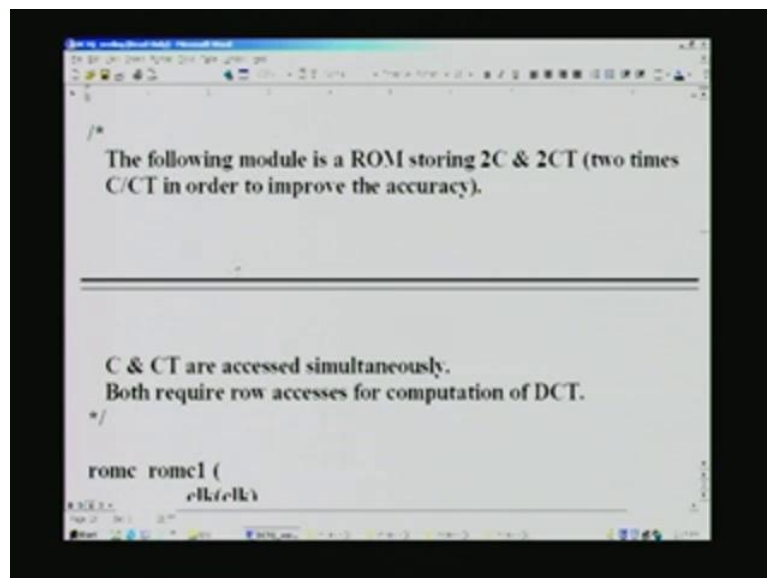
};

// Dual RAM has two pipeline registers, one after ram_rc.v & the
// other at the output of dualram.v.

/*
The following module is a ROM storing 2C & 2CT (two times
C/CT in order to improve the accuracy).
*/
```

Dual RAM has two pipeline registers. You remember these as one after RAM, which is **being called by dual RAM** and also another at the top level of the dual RAM. Two pipeline registers are inside the dual RAM.

(Refer Slide Time: 17:46)



```
/*
The following module is a ROM storing 2C & 2CT (two times
C/CT in order to improve the accuracy).
*/

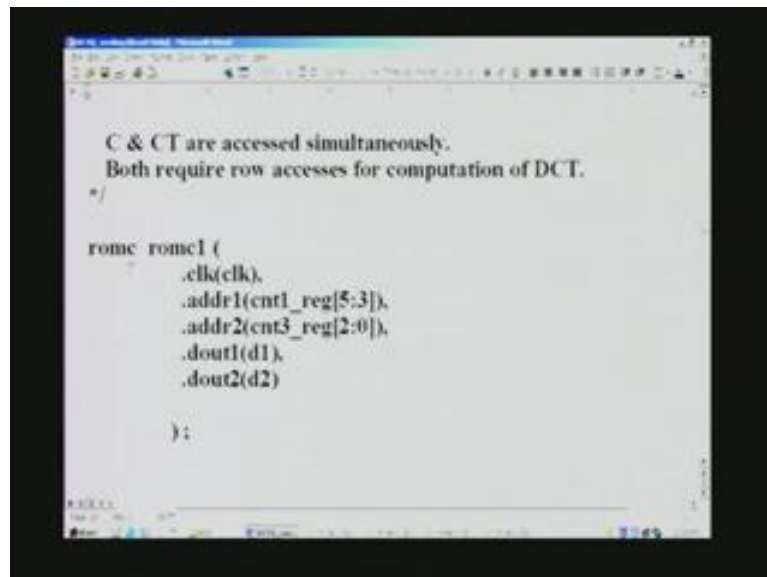
C & CT are accessed simultaneously.
Both require row accesses for computation of DCT.
*/

romc romc1 (
    eH(eHk)
```

Next, we have to use a ROM, which stores C matrix and CT. If you notice, C and CT are nothing but the same matrices. One is to be interpreted as a regular matrix, whereas the other one is a transpose of the same matrix. In this case, the contents of both are the same. That is the reason why we just use a single ROM. We have covered this design also earlier. We do not take just the C value alone – we take twice the C

value and divide by 2 later, so that the functionality is not changed. Yet, we increase the precision while processing. That is the reason why this trick is adopted. Both these require row accesses for computation of DCT.

(Refer Slide Time: 18:39)

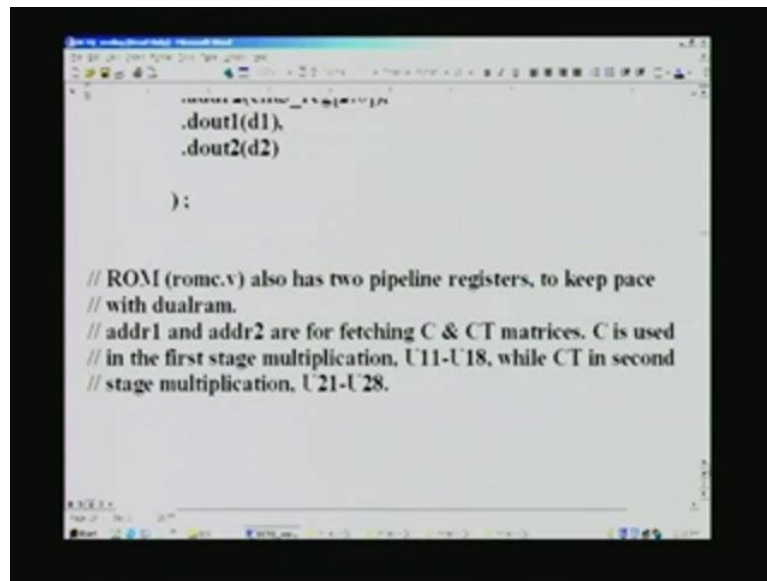


```
C & CT are accessed simultaneously.
Both require row accesses for computation of DCT.
*/

romc romc1 (
    .clk(clk),
    .addr1(cnt1_reg[5:3]),
    .addr2(cnt3_reg[2:0]),
    .dout1(d1),
    .dout2(d2)
);
```

The next step is to call the ROM C and CT module. You remember that we have two addresses: addr1 for C matrix retrieval and addr2 for CT matrix. As I mentioned before, MSB is used here for C matrix. Note that 5 to 3 is the MSB of **cnt1**, whereas LSB is for X matrix, which we have seen just now. In addition to this, we also need CT matrix. That is available by accessing through this address, which is given from **cnt3**. There will be only 3 bits for this counter and this is for CT matrix. Out comes two corresponding data, d1 and d2, which we will use subsequently.

(Refer Slide Time: 19:17)



```
.dout1(d1),  
.dout2(d2)  
  
);  
  
// ROM (romc.v) also has two pipeline registers, to keep pace  
// with dualram.  
// addr1 and addr2 are for fetching C & CT matrices. C is used  
// in the first stage multiplication, U11-U18, while CT in second  
// stage multiplication, U21-U28.
```

Here, romc.v, which is one of the submodules for realizing **C CT ROM**, also has two pipeline registers to keep pace with the **dual RAM**. We have to evaluate CX. Unless the ROM, which has the C content, and dual RAM, which has the X content, keep pace with each other, you will not get any meaningful result. Therefore, the number of pipeline registers has been increased to two, although only one is required here, just to keep in step with the dual RAM. addr1 and addr2 are for fetching C and CT matrices. C is used in the first stage multiplication. Having got C from the ROM and also X from the dual RAM, what we need to do is the multiplication. Remember that we had eight multipliers in the architecture. We will be invoking eight such multipliers, each of them instantiated as U11 through U18. Similarly, we need to **evaluate the result**, which will have to be added. The **adder result** will again be multiplied by another set of eight registers in order to have the last DCT produced, using the CT matrix.

(Refer Slide Time: 19:28)

```
// CX is computed using the following eight multipliers. do is the
// image input, X & d1, the C input. do is unsigned, while d1 is in
// twos complement. result is in twos complement.


mult8ux8s u11(
    .clk(clk),
    .n1(do[63:56]),
    .n2(d1[63:56]),
    .result(result1) //16-bit signed
);
```

CX is computed using the following eight multipliers: **d0** is the image input, X and d1 is the C input, which we have seen before, **d0** is unsigned, while d1 is in twos complement. The entire thing is generally in twos complement – all the computations everywhere. The result is in twos complement. This is the first of the set of multipliers, which numbers from u11 through u18 – eight multipliers. At each step, this one will be the X matrix and notice that only one byte is covered by one multiplier. The corresponding C value is available here. Once again, the corresponding byte is taken in order to process and it produces a result called result 1. This is 16 bit, because it is 8 unsigned into 8 signed – totally, it will be 16 bits.

(Refer Slide Time: 21:04)

```
        .n1(d0[05:20]),
        .n2(d1[63:56]),
        .result(result1) //16-bit signed
    );

    mult8ux8s u12(
        .clk(clk),
        .n1(d0[55:48]),
        .n2(d1[55:48]),
        .result(result2) //16-bit signed
    );
```




Likewise, all the eight multipliers are identical, except for the fact that different bytes are used. Next, eight bytes are used in this case, producing result2.

(Refer Slide Time: 21:41)

```
        .clk(clk),
        .n1(d0[47:40]),
        .n2(d1[47:40]),
        .result(result3) //16-bit signed
    );

    mult8ux8s u14(
        .clk(clk),
        .n1(d0[39:32]),
        .n2(d1[39:32]),
        .result(result4) //16-bit signed
    );
```



```
mult8x8s u12(  
    .clk(clk),  
    .n1(do[31:24]),  
    .n2(d1[31:24]),  
  
    .result(result5) //16-bit signed  
);  
  
mult8x8s u16(  
    .clk(clk),  
    .n1(do[23:16]),  
    .n2(d1[23:16]),  
    .result(result6) //16-bit signed  
);  
  
mult8x8s u17(  
    .clk(clk),  
    .n1(do[15:8]),  
    .n2(d1[15:8]),  
    .result(result7) //16-bit signed  
);
```

It goes on. See here u13, u14. Notice that the bytes are different, progressively less.


(Refer Slide Time: 22:02)

```
mult8x8s u13(  
    .clk(clk),  
    .n1(do[7:0]),  
    .n2(d1[7:0]),  
    .result(result3) //16-bit signed  
);  
  
mult8x8s u14(  
    .clk(clk),  
    .n1(do[15:8]),  
    .n2(d1[15:8]),  
    .result(result4) //16-bit signed  
);  
  
mult8x8s u15(  
    .clk(clk),  
    .n1(do[23:16]),  
    .n2(d1[23:16]),  
    .result(result5) //16-bit signed  
);  
  
mult8x8s u16(  
    .clk(clk),  
    .n1(do[31:24]),  
    .n2(d1[31:24]),  
    .result(result6) //16-bit signed  
);  
  
mult8x8s u17(  
    .clk(clk),  
    .n1(do[39:32]),  
    .n2(d1[39:32]),  
    .result(result7) //16-bit signed  
);
```

This is u16, u17.

(Refer Slide Time: 22:02)


```
mult8ux8s u18(  
    .clk(clk),  
    .n1(d0[7:0]),  
    .n2(d1[7:0]),  
    .result(result8) //16-bit signed  
);  
  
// Partial products of CX are added here.
```



Then, this is the last multiplier in the set. It is processing the very last X. If you take a row, it is very last pixel that is being taken. It could be either the last or the first – it is immaterial as long as you take C and X, the identical bytes. The only thing is you have to keep track of what you are doing.

(Refer Slide Time: 22:38)

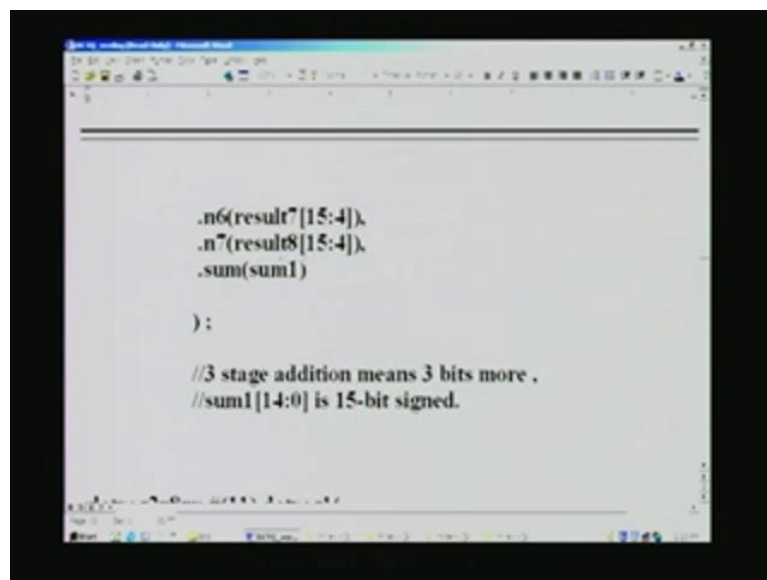
```
// Partial products of CX are added here.  
  
adder12s adder12s1(  
    .clk(clk),  
    .n0(result1[15:4]), // 12-bit signed Ex.: (156).0010  
    .n1(result2[15:4]), // Five pipeline stages -output  
                        // not registered  
    .n2(result3[15:4]), // since it is registered in the  
                        // following dcreg1.  
    .n3(result4[15:4]),  
    .n4(result5[15:4]),  
    .n5(result6[15:4]),  
);
```



At this step, we have multiplied and got eight results; result1 through result8. The next step is to add these eight results that we have got by using adder12s, which we have already designed before. n0 through n7 are none other than this result 1 through result 8, which we have got before. Note that we are dropping some bits here, because

that much precision is not really required. At this step, if you take the **modelsim** results and ponder over the waveform, you will notice that it will be of the order of 156 in decimal number. A typical example has been given for one of the results and other results will be hovering around this. I want to point out that there is a decimal point here and after this, 4 bits. So, it is a mix of a decimal number and binary number. After the decimal point, there will be four digits here. This adder, as we have seen before in the design, has only five pipeline stages and output is not registered. This is not registered because we are going to register the next module that we are going to look into. That is called **dctreg**.

(Refer Slide Time: 24:01)



```
.n6(result7[15:4]),
.n7(result8[15:4]),
.sum(sum1)

);

//3 stage addition means 3 bits more ,
//sum1[14:0] is 15-bit signed.
```

You have up to n7 here. Note that we have dropped 4 bits here because we do not need that high a precision. These precisions have been arrived after a few iterations and keeping in mind the quality of the image that we create. We have very high precision in the computation of the same DCTQ and reconstruction in MATLAB. That serves as the standard reference for the computation of the quality. After truncating these many bits, we always compute the PSNR or what is called the image quality. Then, we compare with the corresponding MATLAB output. If it is quite close, of the order of say 0.5 dB, maybe we can say that accuracy is 2 percent or less. Then, we truncate as per that. That is how we do the truncation here.

This is a three-stage addition. This is the summation that we are doing. As a result, we started with 12 bits as the input for each of these numbers. The final result after

addition will be 3 bits more. This is because at every stage of addition, you have one extra bit if you add two numbers. That is how we have 15 bits – 14 through 0 – as the sum here. This will have to be stored as a partial product.

(Refer Slide Time: 24:01)

```
//3 stage addition means 3 bits more ,
//sum1[14:0] is 15-bit signed.

dctreg2x8xn #(11) dctreg1(
    .clk(clk),
    .din(sum1[14:4]), // 11-bit signed (integer)- dropping
                    // 3bits after dec. pt.
                    // C is actually taken as 2C in the
                    // ROM & hence 1 more bit is dropped.
                    // Similarly for CT.
```

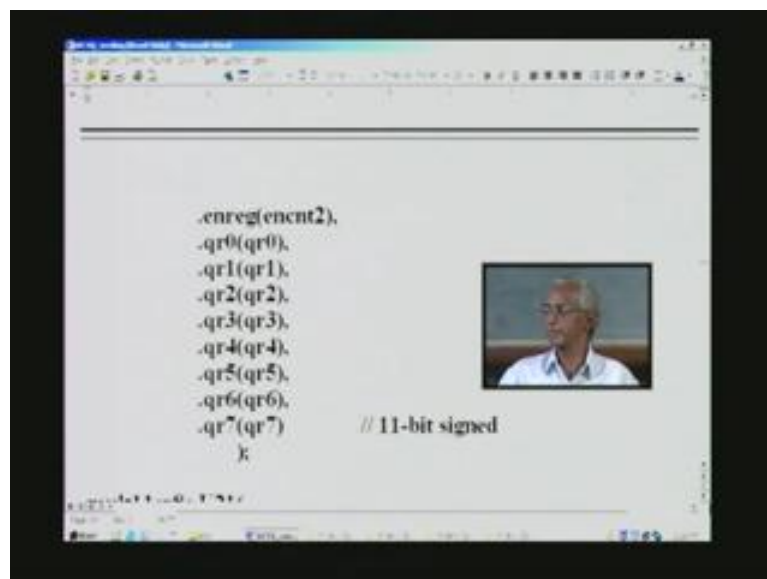
```
dctreg2x8xn #(11) dctreg1(
    .clk(clk),
    .din(sum1[14:4]), // 11-bit signed (integer)- dropping
                    // 3bits after dec. pt.
                    // C is actually taken as 2C in the
                    // ROM & hence 1 more bit is dropped.
                    // Similarly for CT.
    .wa(cnt2_reg[2:0]),
```

This is the actual partial product P, which we had seen in the algorithm before. That is what this module is going to be doing. It has one input. That is the same thing here. Once again, we see that we drop some more bits here. Otherwise, it will become too unwieldy **with the next stage**. Anyway, it is producing the desired quality. That is also a reason why we drop one more bit here. We will just have a look at the comments here.

This is actually 11 bits signed integer and there is no decimal point here. We drop 3 bits after the decimal point. In addition to this, we also drop one more bit, because 3 through 0 are dropped, that is, 4 bits are dropped and so, 3 bits have been accounted for here. We drop another bit, which is equivalent to dividing the result by 2. Any right shift, as you know, is equivalent to dividing by 2. That is because **we had taken earlier** in order to increase the precision **2C not C**, but we actually need C here in order to have the correct result. For that, we drop one more bit, which is equivalent to dividing the result that contains 2C into getting a result that only contains C. **ROM and hence** one more bit is dropped. That is what we have here. The same is true for CT as well.

Here, we have eight registers that will register all the partial products generated in the first stage of multiplication. You need to write it into the appropriate registers. For that, you need a three-bit address. As there are eight registers, you need 3 bits here in order to write into that.

(Refer Slide Time: 27:22)



This set of registers is called qr0 through qr7. This writing will be enabled only if the **enable counter is enabled – it is 1**. Each of these partial products will be 11 bits signed, as explained before.

(Refer Slide Time: 27:37)

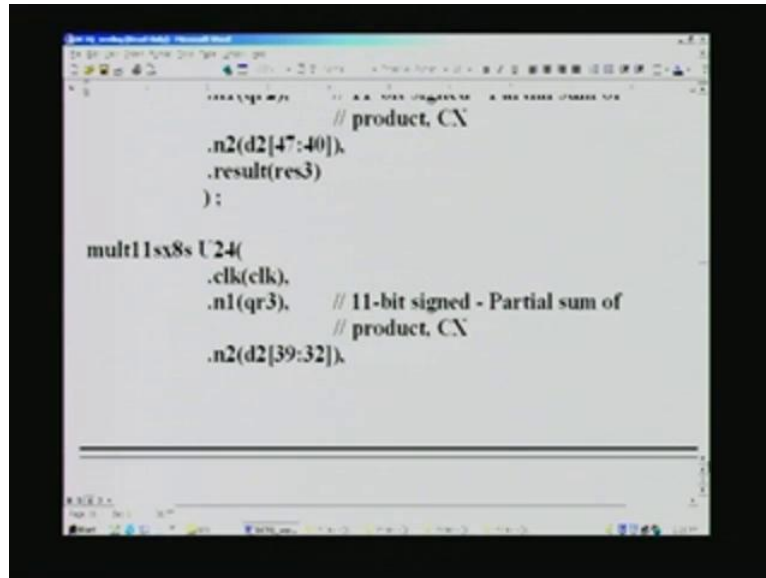
```
        .qro(qro),
        .qr7(qr7) // 11-bit signed
    );

    mult11sx8s U21(
        .clk(clk),
        .n1(qr0), // 11-bit signed - Partial sum of
                // product, CX
        .n2(d2[63:56]), // 8-bit signed - CT
        .result(res1) // 19-bit signed, [18:0]
    );
```

```
        .n1(qr1), // 11-bit signed - Partial sum of
                // product, CX
        .n2(d2[55:48]),
        .result(res2)
    );

    mult11sx8s U23(
        .clk(clk),
        .n1(qr2), // 11-bit signed - Partial sum of
                // product, CX
        .n2(d2[47:40]),
        .result(res3)
    );

    mult11sx8s U24(
```



```
        // product, CX
        .n2(d2[47:40]),
        .result(res3)
    );

    mult11x8s U24(
        .clk(clk),
        .n1(qr3), // 11-bit signed - Partial sum of
                 // product, CX
        .n2(d2[39:32]),
```

In the second stage, we have another eight multipliers. The set has eight multipliers here. In this case, we take these partial products, which are qr0 through qr7 and you can see **as one of the numbers**. The second number is the actual CT. This partial product is nothing other than CX. We need to multiply this with the CT matrix in order to get the final DCT. That is what we are doing here.

We need eight multipliers and all these multipliers will be working simultaneously or concurrently. That is the beauty of FPGA/ASIC design. You would have noticed that we have so many hardware. So far, we have seen eight multipliers and one adder so far. In addition to that, all the other hardware such as **other set of eight plus one** more multiplier, adder, and so on – all of them work simultaneously or concurrently. That is the beauty of this FPGA/ASIC design. To start with, we said it is massively parallel. That is how you have a massively parallel design.

This particular design has totally seventeen multipliers in order to have a DCTQ. We also had ROM, etc. All of these work concurrently for fresh data being input at every clock cycle. This is the CT input here. Once again, we apply only one byte at a time. This is 11 bits and this is 8 bits. Naturally, result will be 19 bit and we call it res1 to res8 – subsequent eight multipliers and two more multipliers here. You can notice that we are taking a different byte order each time.

(Refer Slide Time: 29:41)

```
        .n2(d2[15:8]),
        .result(res7)
    );

    mult11sx8s U28(
        .clk(clk),
        .n1(qr7), // 11-bit signed - Partial sum of
                // product, CX
        .n2(d2[7:0]), // 8-bit signed
        .result(res8) // 19-bit signed
    );

    adder14sr adder14sr1(
        .clk(clk),
```

Finally, the last multiplier **in the second stage**. Thus, we have seen sixteen multipliers being used, all of which work concurrently. res1 through res8 is the final result.

(Refer Slide Time: 29:53)

```
        .n2(d2[7:0]), // 8-bit signed
        .result(res8) // 19-bit signed
    );

    adder14sr adder14sr1(
        .clk(clk),
        .n0(res1[18:5]), // 14-bit signed (1065),0010
        .n1(res2[18:5]),
        .n2(res3[18:5]),
        .n3(res4[18:5]),
```

You add this in the next stage, which is adder with registered output. This was given as an assignment to you and I hope you have done it, in which case you can readily use the design. You notice that once again we are dropping some more bits for the same reason, so that the quality that we get is already satisfied even if you drop so many bits. Here, you are dropping nearly 5 bits and as an example for this value, the decimal point is shown here.

(Refer Slide Time: 30:34)

```
.n4(res5[18:5]),
.n5(res6[18:5]),
.n6(res7[18:5]),
.n7(res8[18:5]),
.dct(dct[11:0]) // 12-bit signed, [11:0] - This is the
// DCT output.

);

// This adder14sr.v has six pipeline stages - registered output.
```

Finally, we add **res1** through **res8** to get a DCT. When you add, you would notice that it is 18 through 0 **and we have** dropped so many bits here. **In spite of that, the number of bits is fourteen bits**. Actually, it will shoot up by 3 more bits, because the adder has three stages of addition inside the pipeline. Therefore, it will create totally 17 bits. This is 13, 14, and that will be 3 more bits. There will be 17 bits out of which we need to extract only a few bits.

This is only 12 bits and that is required. Once again, we are truncating here. This truncation is done right within the adder itself. You must also do the same thing when you do this design all by yourselves. This is the final DCT output. We have to get the DCTQ output from this. This is in twos complement. One more point is that this **DCT 12 bit** has not been put arbitrarily but is the requirement for JPEG through MPEG standards. That is the reason why we adopt 12 bits here.

(Refer Slide Time: 31:53)

```
// DCT output.

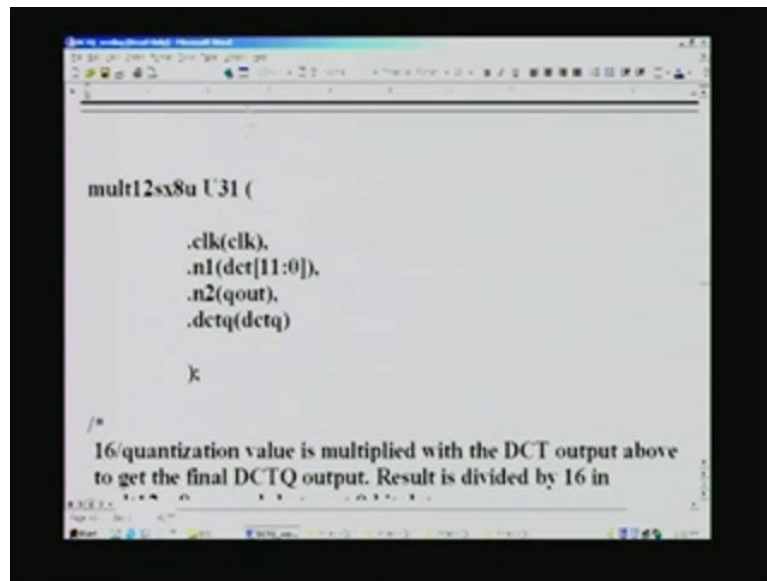
);

// This adder14sr.v has six pipeline stages - registered output.
// Quantization stage - 64B ROM, decimal point before msb.

romq romq1 (
    .clk(clk),
    .a(cnt4_reg),
    .d(qout)
);
//16 Quantization value is fetched from ROM.
```

This adder 14sr.v has six pipeline stages because one more stage is required when compared to 12s. This is because of registering the output. The next stage is to invoke the quantization matrix, which contains 16 divided by the quantization value. The inverse quantization value is being taken here and in order to increase the precision, as we have done before for C by multiplying it by 2 and then dividing by 2, here also, we adopt the same trick by multiplying it by 16 and later on we will divide by 16 so that the value is not changed as per the algorithm. In order to retrieve 16 by quantization value, what we need is address for the ROM so that you get the **actual output qout, 16 by quantization value.** cnt4 generated by the controller supplies this address here.

(Refer Slide Time: 32:55)



```
mult12sx8u U31 (  
    .clk(clk),  
    .n1(dct[11:0]),  
    .n2(qout),  
    .dctq(dctq)  
);  
  
/*  
16/quantization value is multiplied with the DCT output above  
to get the final DCTQ output. Result is divided by 16 in
```

After invoking that ROM, we need to multiply in order to get the DCTQ from DCT. One input for the multiplier will be the DCT, which we have already seen. `qout` is nothing other than the 16 by quantization value, which we have just seen in `romq` – that is being fed here. You will appreciate the feature of calling ports by name here, because now you can see all this. Right at the beginning, I mentioned that we have declared this as a wire. From this, it should be apparent. What we are doing here is we from one module we are connecting it to another module. It will have the very same name inside that module. Since we are making a physical connection like a wire or a net, it is called wire. You have a `qout` here, which is the ROM quantization table. That is this one. Finally, out comes the DCTQ and this also has eight pipeline stages. If you remember, we have seen the overall pipeline stages to be 45 in the architecture. We will understand this very clearly when we see the actual waveforms.

(Refer Slide Time: 34:07)

```
.clk(clk),
.n1(dct[11:0]),
.n2(qout),
.dctq(dctq)
);

/*
16/quantization value is multiplied with the DCT output above
to get the final DCTQ output. Result is divided by 16 in
mult12sx8u.v module to get 9 bit dctq.

n1 is DCT, signed 12 bit, integer.

n2 is unsigned, 8 bit - decimal point before msb.
```

```
);

/*
16/quantization value is multiplied with the DCT output above
to get the final DCTQ output. Result is divided by 16 in
mult12sx8u.v module to get 9 bit dctq.

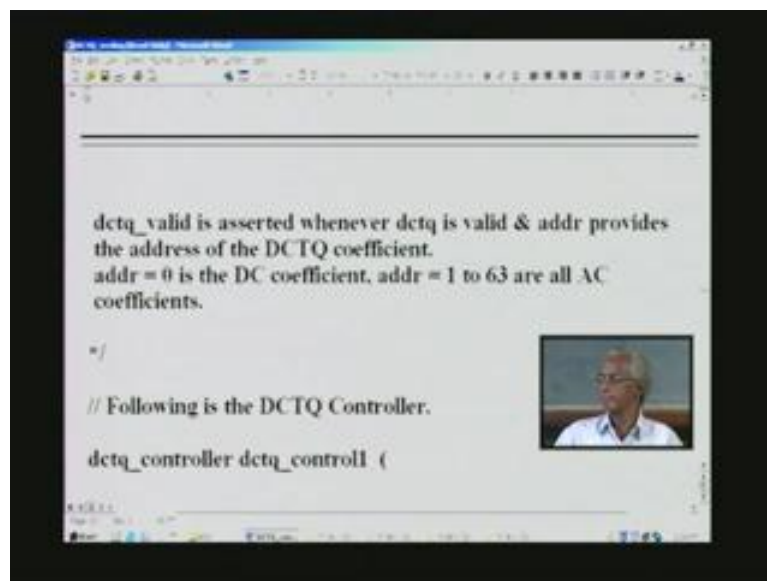
n1 is DCT, signed 12 bit, integer.

n2 is unsigned, 8 bit - decimal point before msb.

dctq[8:0] conforms to JPEG/MPEG-1/MPEG-2 standards, etc.
```

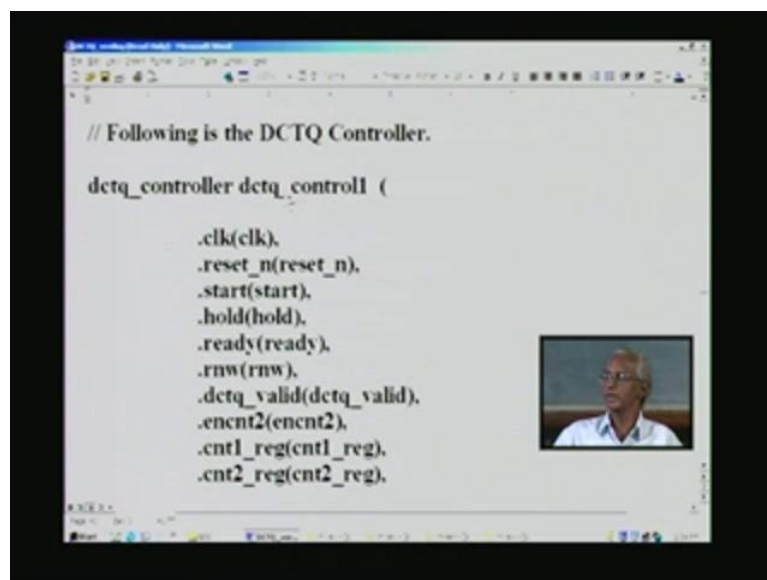
16 by quantization value is multiplied with the DCT output above to get the final DCTQ output. The result is divided by 16 in **mult12sx8u** module to get nine-bit DTCQ. Once again, these nine bits are dictated from the JPEG/MPEG standards. In order to conform to the standards, it has been taken as 9 bits. n1 is DCT, signed 12 bits and this is also as per JPEG. n2 is unsigned 8 bit with the decimal point before MSB and it is the 16 by quantization value. The actual value starts after the decimal point and that is what is mentioned here. This DCTQ 8 through 0, which is 9 bits, conforms to JPEG/MPEG-1/MPEG 2 standards and 'etc.' probably stands for H261, 263 and there may be other standards as well, maybe HDTV.

(Refer Slide Time: 35:02)



`dctq_valid` is asserted whenever `dctq` is valid and `addr` provides the address of the `dctq` coefficient. We have seen this before. `Addr = 0` is the DC coefficient and `addr = 1 to 63` are all AC coefficients. The higher the order, the higher is the frequency.

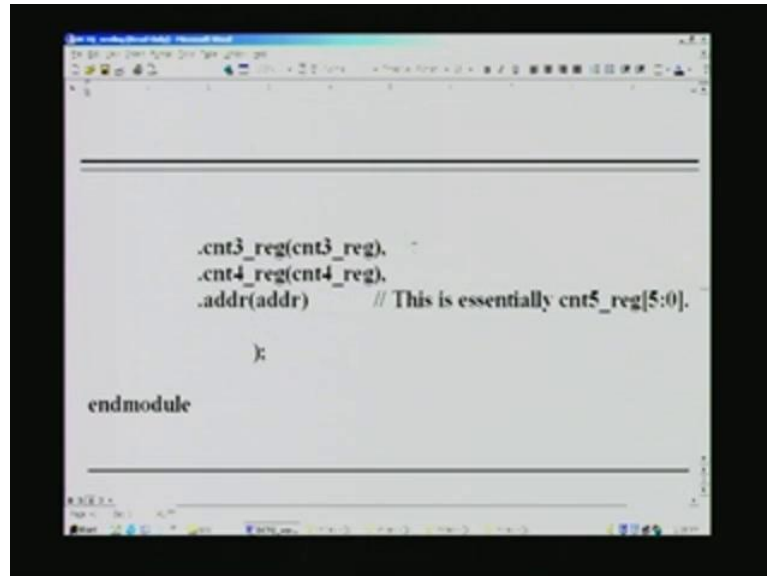
(Refer Slide Time: 35:20)



Before ending this design, we have to invoke a controller, which we have seen the need for. It has various signals. When we take the details of the controller, we will describe all this. Once again, we call by the port names. We have already seen start pins, `rnw`, etc. These are all the outputs generated by the controller. There are

different counters, **encnt2** as well as enable partial product register. Different registers are there.

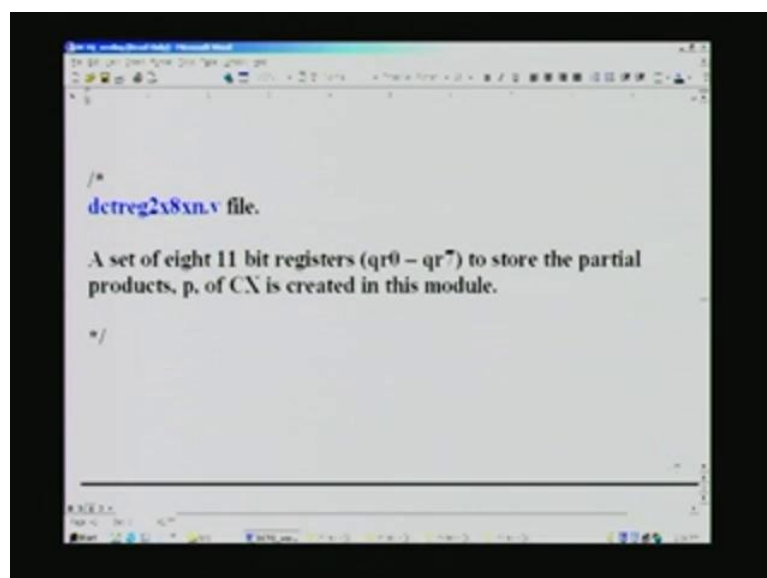
(Refer Slide Time: 36:01)



```
.cnt3_reg(cnt3_reg),  
.cnt4_reg(cnt4_reg),  
.addr(addr) // This is essentially cnt5_reg[5:0].  
  
);  
  
endmodule
```

This is counter5 or you can rather take it as an address register in providing the DCTQ coefficient address. This is the end of the top design module. You would have noticed an interesting feature that I explained right in the beginning – there is no logic in the top design module. It is merely calling the different modules that you have seen here, just calling the modules – adder or multiplier, as the case may be. I reinforce the statement that a good design practice is to avoid any logic on this.

(Refer Slide Time: 36:42)



```
/*  
dctreg2x8xn.v file.  
  
A set of eight 11 bit registers (qr0 – qr7) to store the partial  
products, p, of CX is created in this module.  
  
*/
```

Next, we will consider some of the submodules that we have not seen before. One is for the partial product. This is called the `dctreg2x8`, which means that sixteen registers are really required. We can also program the width here. That is why `n` is put here. If you do not like this name, you can change it to `partial product pp` or something like `that`. A set of eight 11-bit registers, that is, `qr0` through `qr7`, to store the partial products `p` of `CX` is created in this module.

(Refer Slide Time: 37:20)

```

module dctreg2x8xn (clk, wa, din, enreg, qr0, qr1, qr2, qr3, qr4,
qr5, qr6, qr7);
parameter WIDTH = 11 ;
output [(WIDTH-1):0] qr0, qr1, qr2, qr3, qr4, qr5, qr6, qr7;
input [(WIDTH-1):0] din;
input [2:0] wa;
input enreg, clk;

```

```

module dctreg2x8xn (clk, wa, din, enreg, qr0, qr1, qr2, qr3, qr4,
qr5, qr6, qr7);
parameter WIDTH = 11 ;
output [(WIDTH-1):0] qr0, qr1, qr2, qr3, qr4, qr5, qr6, qr7;
input [(WIDTH-1):0] din;
input [2:0] wa;
input enreg, clk;
reg [(WIDTH-1):0] qr0, qr1, qr2, qr3, qr4, qr5, qr6, qr7;
reg [(WIDTH-1):0] q0, q1, q2, q3, q4, q5, q6;

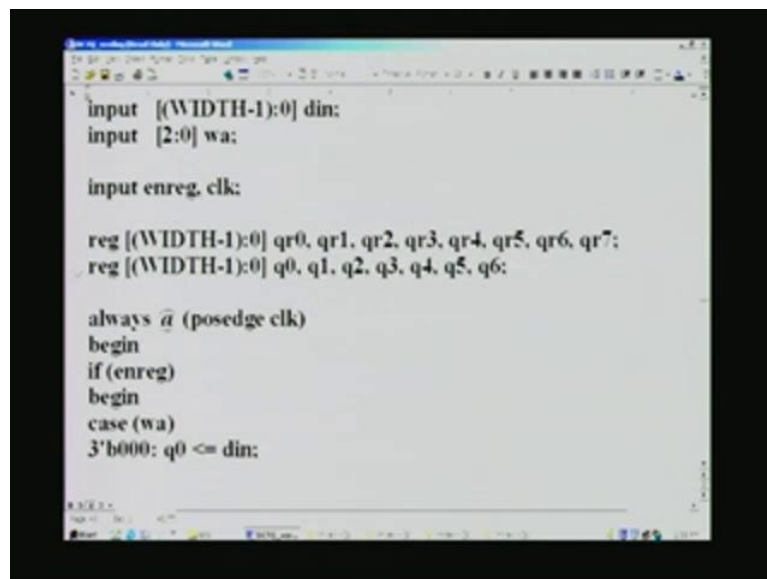
```

This is the module declaration. It lists `qr0` through `qr7` outputs here and these are all the inputs. `din` is what you have after the first adder, the 12s adder that we have seen. That is the partial product that we need to store in. At every clock, a new data will be coming in `din`, which we will have to route to one of these registers. For example,

with the arrival of the first clock, you will route it to qr0 and then divert the next data at the next clock to qr1 and so on. At the eighth clock, you will input into qr7.

As I mentioned, n is programmable and so you can do this by having a special **C-like instruction** here. We can declare any variable here and assign some known value. For example, width has been assigned **as 11 decimal** here and this is by declaring as a parameter. Once you declare this, instead of writing 11 at every point of time, we need to only write width. That is what we see. This is the **output width** and that is for register size. If you take 11 here, 11 minus 1 is 10, so 10 through 0 **are the 11 bits for** the qr registers. Similarly, **din** must also have the same width and that is why this is put here. These 0, 1, 2, 3 are addresses for the register – to point to the respective register. That is done by this signal, which must be 3 bits in order to have total 0 through 7.

(Refer Slide Time: 39:06)



```
input [(WIDTH-1):0] din;
input [2:0] wa;

input enreg, clk;

reg [(WIDTH-1):0] qr0, qr1, qr2, qr3, qr4, qr5, qr6, qr7;
reg [(WIDTH-1):0] q0, q1, q2, q3, q4, q5, q6;

always @ (posedge clk)
begin
if (enreg)
begin
case (wa)
3'b000: q0 <= din;
```


Once again, we have declaration of reg and width is given. We use all these registers in the **always positive edge clock** and this is the clock that we are referring to. In this block, what we are going to do is simply assign the **din value** to different registers, depending upon the right address. The right address is also one of the inputs. If the address is **000**, then we take whatever is the input at **din** and merely copy into the q0 register. Like this, q0 through q6 will be used for different addresses. We can do all this only as long as this chip is enabled – **enreg** is one of the inputs. This, in turn, is from the **enable counter2**. Only if it is enabled, this will take place. With the arrival of

the first clock, the right address will be 0, and with the next clock, it will be 1. Each time, it will go through one of these cases. Hence, the case statement is used here.


(Refer Slide Time: 40:12)

```
begin
case (wa)
3'b000: q0 <= din;
```


```
3'b001: q1 <= din;
3'b010: q2 <= din;
3'b011: q3 <= din;
3'b100: q4 <= din;
3'b101: q5 <= din;
3'b110: q6 <= din;
```




```
3'b001: q1 <= din;
3'b010: q2 <= din;
3'b011: q3 <= din;
3'b100: q4 <= din;
3'b101: q5 <= din;
3'b110: q6 <= din;
3'b111: begin
qr0 <= q0;
qr1 <= q1;
qr2 <= q2;
qr3 <= q3;
qr4 <= q4;
```



```
3> 0011: q3 <= din;
3'b100: q4 <= din;
3'b101: q5 <= din;
3'b110: q6 <= din;
3'b111: begin
    qr0 <= q0;
    qr1 <= q1;
    qr2 <= q2;
    qr3 <= q3;
    qr4 <= q4;
    qr5 <= q5;
    qr6 <= q6;
    qr7 <= din;
end
endcase
```




```
qr0 <= q0;
qr1 <= q1;
qr2 <= q2;
qr3 <= q3;
qr4 <= q4;
qr5 <= q5;
qr6 <= q6;
qr7 <= din;
end
endcase
end
end
```



```
endmodule

/*
dctq_controller.v file.

module dctq_controller (
```

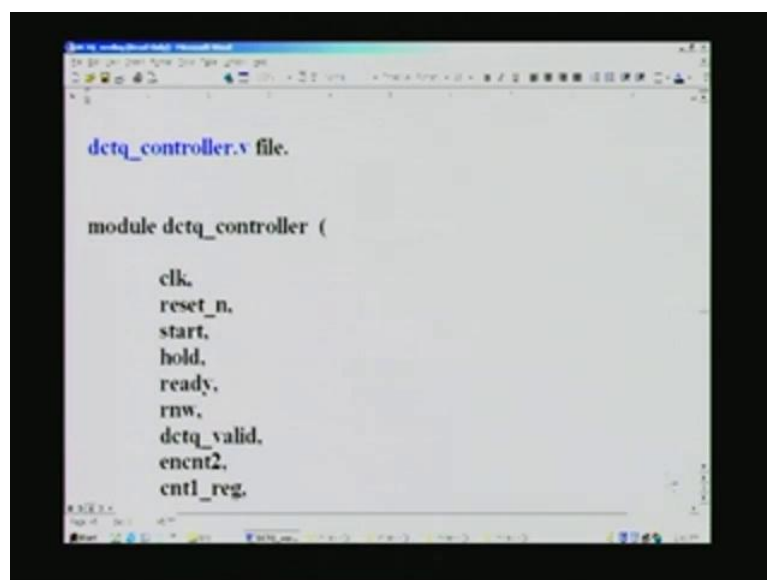


Like that, you have q0 through q6 arriving at different clock cycles one after another. Finally, you will have q7 also assigned but not here – towards the end. I will explain why this is being assigned here. Instead of putting **this for this statement with this alone. We have done some more thing**, which will take some more registers for this. The reason is as follows. **On the arrival** of the seventh clock cycle, this has been written and with the eighth clock cycle, the last partial product is written.

If this partial product that we have written continues to be in the same q0 register, with the arrival of the next clock, that is, ninth clock, this q0 will be overwritten and the partial product will get lost. We need to avoid this. Remember that we need the partial product for the computation of the next stage for which you need eight further clock cycles. This means that this partial product will have to be stable for the next eight clock cycles. That is possible if you **write it at this stage into** another register called qr0.

Right at the eighth clock cycle, we transfer all this q0 through q6 content into a safe register here as well as transfer the **last thing**, which we have not done before into this. With the arrival of the next clock, this q0 will be overwritten, but not qr0. qr0 will be overwritten only with the next cycle, after all the eight clock cycles. Thus, if you examine, this will be available for the next stage computation **to the tune of** eight clock cycles. This completes that DCT register.

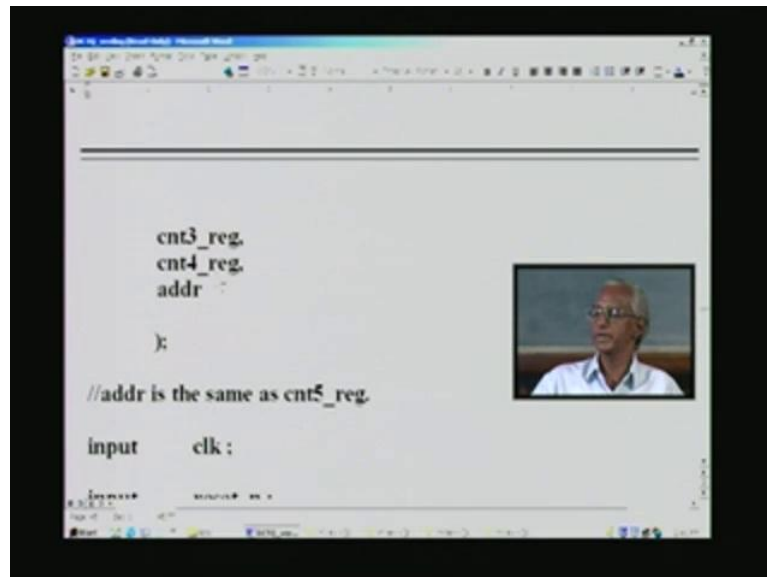
(Refer Slide Time: 42:02)



```
dctq_controller.v file.  
  
module dctq_controller (  
    clk,  
    reset_n,  
    start,  
    hold,  
    ready,  
    rnw,  
    dctq_valid,  
    enct2,  
    cnt1_reg.  
);
```


Next is the DCTQ controller. These signals are listed here. We have different inputs: reset, start, hold and ready and these are all the outputs. rrw is to switch from one RAM to another inside a dual RAM. When `dctq_valid` is to be given, all that is covered here on the counters.

(Refer Slide Time: 42:29)



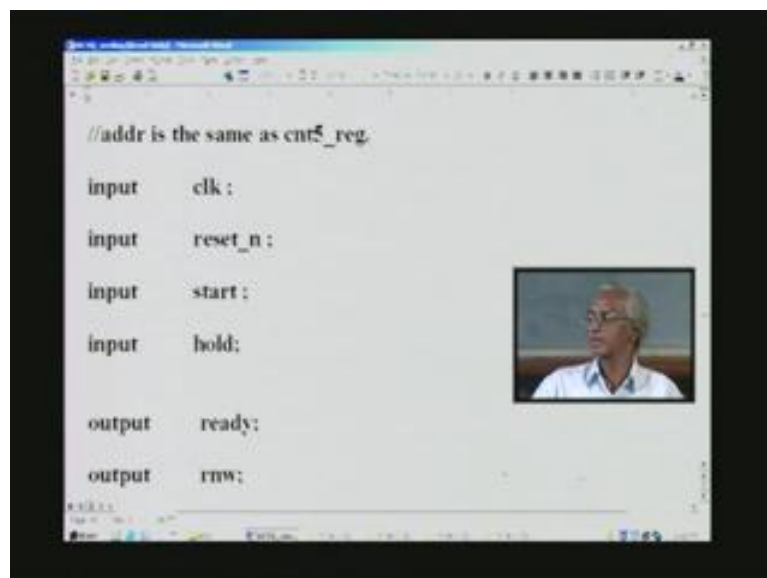
```
cnt3_reg,
cnt4_reg,
addr
);

//addr is the same as cnt5_reg.

input  clk;
```

So also the address for the DTCQ coefficient is given. Address is same as `cnt5`.

(Refer Slide Time: 42:37)



```
//addr is the same as cnt5_reg.

input  clk;

input  reset_n;

input  start;

input  hold;

output ready;

output rrw;
```

We declare here all the inputs as well as the outputs.

(Refer Slide Time: 42:42)

```
output    dctq_valid;
output    encnt2 ;

output [5:0]  cnt1_reg ;
output [2:0]  cnt2_reg ;
output [2:0]  cnt3_reg ;
output [5:0]  cnt4_reg ;
```

```
output [5:0]  cnt1_reg ;
output [2:0]  cnt2_reg ;
output [2:0]  cnt3_reg ;
output [5:0]  cnt4_reg ;
output [5:0]  addr ;

reg        ready;
```

So also the counters and their widths.

(Refer Slide Time: 42:49)

```
output [5:0] addr ;

reg    ready;
reg    rnw;
reg    dctq_valid;
```

The registers are declared here regarding the outputs.

(Refer Slide Time: 43:00)

```
reg [5:0] cnt4_reg;
reg [5:0] addr ;

reg    encnt1 ;
reg    encnt2 ;
reg    encnt3 ;

reg    encnt4 ;
reg    encnt5 ;
```

As mentioned before, we have five counters and they also need enabling. So, we declare them as reg here.

(Refer Slide Time: 43:09)

```
wire      start_next1 ;
wire      encnt1_next ;
wire      discnt1_next;

wire      swrnw1;
wire      swrnw2;
wire      swon_ready;

wire [5:0] cnt1_next ;
wire [2:0] cnt2_next ;
wire [2:0] cnt3_next ;
wire [5:0] cnt4_next ;
wire [5:0] cnt5_next ;
```

```
wire      swrnw1;
wire      swrnw2;
wire      swon_ready;

wire [5:0] cnt1_next ;
wire [2:0] cnt2_next ;
wire [2:0] cnt3_next ;
wire [5:0] cnt4_next ;
wire [5:0] cnt5_next ;

assign cnt1_next = cnt1_reg + 1 ;
assign cnt2_next = cnt2_reg + 1 ;
assign cnt3_next = cnt3_reg + 1 ;
assign cnt4_next = cnt4_reg + 1 ;
```

We also need intermediate results, which we will cover later. These are all the signals that we use for that. We also need pre-incremented signals using assign statements. They are declared as wire here. The actual **cnt1_reg** is this, which is pre-incremented here. That is what is done here and that is what is done here, including the DCTQ coefficient address.

(Refer Slide Time: 43:43)

```
assign cnt5_next = addr - 1 ;

assign encnt1_next = ((start_reg1 == 1'b1)&&(cnt1_reg == 0)) ?
                    1'b1 : 1'b0;
assign discnt1_next = ((start_reg1 == 1'b0)&&(cnt1_reg ==
                    6'd63)) ? 1'b1 : 1'b0;

always @ (posedge clk or negedge reset_n)
begin
    if (reset_n == 1'b0)
        encnt1 <= 1'b0 ;
    else if (hold == 1'b1)
        encnt1 <= encnt1 ;
```

We should enable the very first counter only if this condition is met. It must be disabled when this condition is met. **The first condition is we give the start**, signaling that DCTQ process must begin. After a delay, this **start_reg1**, which we will cover at the last, **always blocks in the controller**. If it is 1, this is equivalent to start being 1, with the clock cycle delay. If it is 1, as long as it is 1 and the **cnt1_reg** is 0, which signals that it is yet to start, **we are right at the beginning**. Only when these conditions are met, we will enable the first counter. The same counter is disabled if the **last of the** block is processed. For example, **start [44:33]** has become 0, in which case when the counter value is 63, only then we disable the counter. Otherwise, we do not. Note that it is for 0 as well as for 63 – not any other combination.

(Refer Slide Time: 44:50)

```
6'd63)) ? 1'b1 : 1'b0;

always @ (posedge clk or negedge reset_n)
begin
  if (reset_n == 1'b0)
    encnt1 <= 1'b0 ;
  else if (hold == 1'b1)
    encnt1 <= encnt1 ;
  else if (encnt1_next == 1'b1)
    encnt1 <= 1'b1 ;
  else if (discnt1_next == 1'b1)
    encnt1 <= 1'b0 ;

  else
    encnt1 <= encnt1 ;
```

We have a set of always blocks here and they are all identical. First, we initialize each of these **enable counters – 1 through 5**. If it is hold, we just do not disturb the contents. We enable this only when a particular condition is met. We have already seen this condition before. Only then, we enable the counter. Otherwise, if the disable condition (we have also seen that before) is met, then we will disable this. Putting a 0 into this is equivalent to disabling. We will see exactly the same thing. Otherwise, do not disturb **here for all the other counter enables**.

(Refer Slide Time: 45:33)

```
always @ (posedge clk or negedge reset_n)
begin
  if (reset_n == 1'b0)
    encnt2 <= 1'b0 ;
  else if (hold == 1'b1)
    encnt2 <= encnt2 ;
  else if (discnt1_next == 1'b1)
    encnt2 <= 1'b0 ;
  else if (cnt1_reg == 6'd14) // cnt2 is enabled when
                             // cnt1_reg = 14 dec.
    encnt2 <= 1'b1 ;
  else
    encnt2 <= encnt2 ;
```

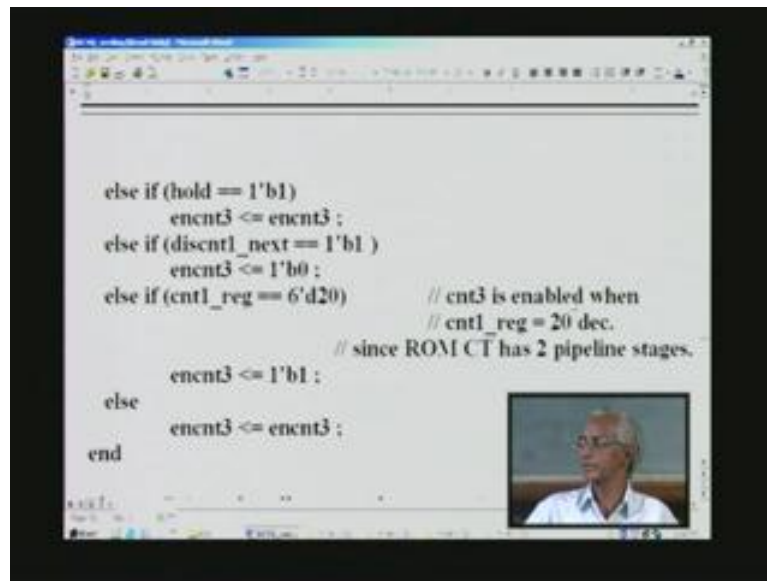
```
else if (discont1_next == 1'b1)
    encnt2 <= 1'b0 ;
else if (cnt1_reg == 6'd14)           // cnt2 is enabled when
                                     // cnt1_reg = 14 dec.
    encnt2 <= 1'b1 ;
else
    encnt2 <= encnt2 ;
end

always @ (posedge clk or negedge reset_n)
begin
    if (reset_n == 1'b0)
        encnt3 <= 1'b0 ;
```

Once again you see in the design that there is only one signal in one always block. This is a good practice and I hope you will follow the same. If you take hundreds of signals in one always block, you will get drowned. It is a good practice to have just one signal and very rarely violating that if the need arises, which we may see later on. Similarly, it is the same for reset, hold, etc. It has to be cleared if **cnt1** is to be disabled.

Remember that we need a partial product to start with and so, the **dctreg** we have used should be active. Only when the actual **cnt1**, which starts right from the beginning when the start is given, is 14, it will be the right time for enabling the **cnt2** as well as **dctreg**. That is what we are doing here and that is for **14 decimal**. Likewise, we will be doing for the different pipelines we have covered earlier. This is for the **dctreg**. We will understand this very clearly when we look at the waveform.

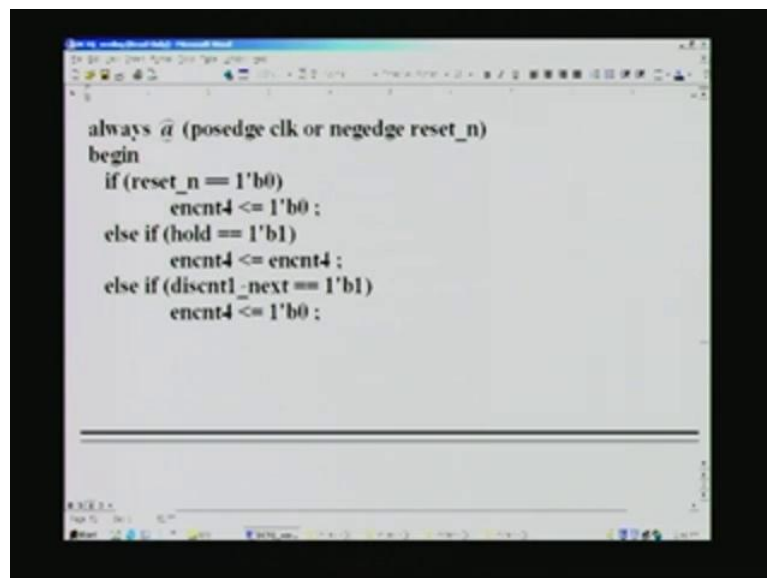
(Refer Slide Time: 46:53)

A screenshot of a video lecture showing a code editor window with Verilog code. The code is for controlling a counter 'cnt3'. It includes conditional assignments based on 'hold', 'discnt1_next', and 'cnt1_reg' values. Comments explain that 'cnt3' is enabled when 'cnt1_reg' is 20 decimal, which is because the ROM counter has two pipeline stages. A small inset video of a speaker is visible in the bottom right corner of the code editor.

```
else if (hold == 1'b1)
    encnt3 <= encnt3 ;
else if (discnt1_next == 1'b1 )
    encnt3 <= 1'b0 ;
else if (cnt1_reg == 6'd20)           // cnt3 is enabled when
                                     // cnt1_reg = 20 dec.
                                     // since ROM CT has 2 pipeline stages.
    encnt3 <= 1'b1 ;
else
    encnt3 <= encnt3 ;
end
```

This is to enable **cnt3**. Everything is the same, except that this is being done for 20. This is for **CT and ROM CT** access here. **cnt3** is enabled, when cnt1_reg is 20 – decimal, of course. ROM CT has two pipeline stages and therefore 20 is arising.

(Refer Slide Time: 47:17)

A screenshot of a video lecture showing a code editor window with Verilog code. The code is for controlling a counter 'cnt4' within an always block triggered by the clock or reset. It includes conditional assignments based on 'reset_n', 'hold', and 'discnt1_next' values.

```
always @ (posedge clk or negedge reset_n)
begin
    if (reset_n == 1'b0)
        encnt4 <= 1'b0 ;
    else if (hold == 1'b1)
        encnt4 <= encnt4 ;
    else if (discnt1_next == 1'b1)
        encnt4 <= 1'b0 ;
end
```



```
else if (cnt1_reg == 6'd35) // cnt4 is enabled when
                           // cnt1_reg = 35 dec.
    encnt4 <= 1'b1 ;
else
    encnt4 <= encnt4 ;
end

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

Similarly, for **cnt4**, it is exactly the same, except once again, it is a different value here, **for 35**. This is at the quantization ROM. When the quantization ROM must be enabled is governed by this 35 of **cnt1**.

(Refer Slide Time: 47:44)


```
always @ (posedge clk or negedge reset_n)
begin
    if (reset_n == 1'b0)
        encnt5 <= 1'b0 ;
    else if (hold == 1'b1)
        encnt5 <= encnt5 ;
    else if (discnt1_next == 1'b1)
        encnt5 <= 1'b0 ;
    else if (cnt1_reg == 6'd44) // cnt5 is enabled when
                               // cnt1_reg = 44 dec.
        encnt5 <= 1'b1 ;
    else
        encnt5 <= 1'b1 ;
end
```

Similarly, when DCTQ must be valid is determined at 44. It is exactly the same thing that we have seen before. You have seen in the architecture that we had totally 45 clock cycles delay. This counter starting from 0, 1, 2, 3 up to 44 is exactly 45 and this explains why there were 45 pipeline stages.

(Refer Slide Time: 48:06)

```
always @ (posedge clk or negedge reset_n)
begin
  if (reset_n == 1'b0)
    cnt1_reg <= 6'd00 ;
  else if (hold == 1'b1)
    cnt1_reg <= cnt1_reg ;
  else if (encnt1 == 1'b1)
    cnt1_reg <= cnt1_next ;
  else
    cnt1_reg <= cnt1_reg ;
end

always @ (posedge clk or negedge reset_n)
begin
  if (reset_n == 1'b0)
```



```
else if (encnt2 == 1'b1)
  cnt2_reg <= cnt2_next ;
else
  cnt2_reg <= cnt2_reg ;
end

always @ (posedge clk or negedge reset_n)
begin
  if (reset_n == 1'b0)
```

```
end

always @ (posedge clk or negedge reset_n)
begin
  if (reset_n == 1'b0)
    cnt3_reg <= 6'd00 ;
  else if (hold == 1'b1)
    cnt3_reg <= cnt3_reg ;
  else if (encnt3 == 1'b1)
    cnt3_reg <= cnt3_next ;
  else
    cnt3_reg <= cnt3_reg ;
end

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

This one is for the actual counter. The counter is also exactly similar to enable, except that when the counter is enabled, it will be incremented. So, cnt1_next was actually cnt1_reg plus 1 – pre-incremented, which we have seen before. We assign that only when cnt1 is enabled and similarly for all other counters. Once again, you can see reset, hold, and we increment only if **encnt2** is high. The same is the case for **cnt3** – increment only then.

(Refer Slide Time: 48:44)

```
cnt4_reg <= 6'd00 ;
else if (hold == 1'b1)
  cnt4_reg <= cnt4_reg ;
else if (encnt4 == 1'b1)
  cnt4_reg <= cnt4_next ;
else
  cnt4_reg <= cnt4_reg ;
end

always @ (posedge clk or negedge reset_n)
begin
  if (reset_n == 1'b0)
```

```

else if (encnt5 == 1'b1)
    addr <= cnt5_next;
else
    addr <= addr;
end

assign swrnw1 = ((start_reg1 == 1'b1)&&(cnt1_reg ==
0)&&(cnt_0 == 1'b1)) ? 1'b1 : 1'b0;

```

So also for **cnt4** as well as for **cnt5**. That is for **cnt5**.

(Refer Slide Time: 48:54)

```

assign swrnw1 = ((start_reg1 == 1'b1)&&(cnt1_reg ==
0)&&(cnt_0 == 1'b1)) ? 1'b1 : 1'b0;
assign swrnw2 = ((start_reg1 == 1'b1)&&(cnt1_reg == 63)) ?
1'b1 : 1'b0;

always @ (posedge clk or negedge reset_n)
begin
if (reset_n == 1'b0)
begin
cnt_0 <= 1'b1;
rnw <= 1'b1;
end
else if (hold == 1'b1)

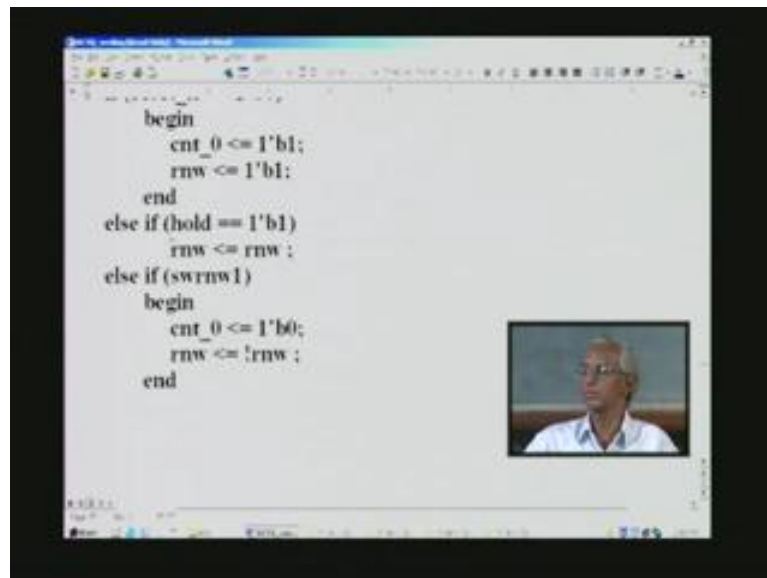
```

We need to differentiate between the very first block of processing and subsequent blocks of processing. That is possible only by having one signal here. It is just a single bit. Initially, it is initialized to 1 with power on here. This is meant for changing the rnw signal. Suppose you have written the very first block. After that, we have to **change this rnw signal only after** writing the very first block.

Having written it, we change it at that point of time and also switch over to a different rnw value, so that we can take in the next block of data and start processing the first

block of data. That is what we are doing here. This depends upon the start being 1 here. Similarly, this is for the ending of the block. When one block is completed, completed in the sense that the data has been consumed, DCTQ is not yet output. Only the data has been consumed as far as one block is concerned. If you read **cnt1** as 63, it means it has just consumed. Start must be continuing to be 1. Under this condition, what will happen here?

(Refer Slide Time: 50:21)



```
begin
  cnt_0 <= 1'b1;
  rnw <= 1'b1;
end
else if (hold == 1'b1)
  rnw <= rnw ;
else if (swrnw1)
  begin
    cnt_0 <= 1'b0;
    rnw <= !rnw ;
  end
```

This is the condition. What we have to do is we have to invert not only rnw but also force this cnt_0 to 0 because for subsequent blocks, this is forced to 0 and therefore, this will not come into this category. In the future, **it will not** come into this category.

(Refer Slide Time: 50:51)

```
else if (swrnw2)
    rnw <= !rnw ;
else
    rnw <= rnw ;
end

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

This was for the end of the block, which we have seen before. If it is the end of the block, we merely invert it. In the previous case, it was to differentiate between the very first block and subsequent blocks, corresponding to **cnt1** being 0 – that is why that trick was played there. This is done as a routine affair here.

(Refer Slide Time: 51:14)

```
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
```

Similarly, we create a ready signal. For this, the condition is start must still be 1. **cnt1** is 1 because after this, we want to take an action. We will continue with this in our next lecture. Thank you.

(Refer Slide Time: 51:31)

