Digital VLSI System Design

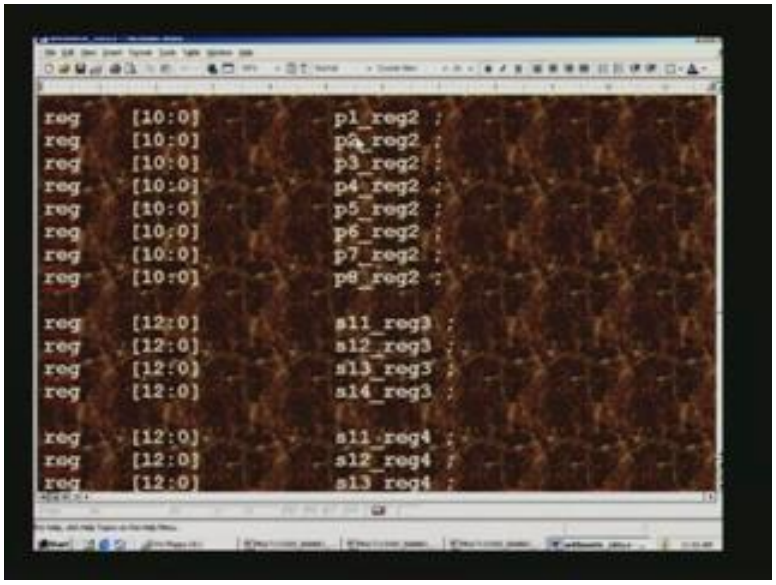
Prof. Dr. S. Ramachandran
Department of Electrical Engineering
Indian Institute of Technology, Madras

Lecture – 42

Design of Arithmetic Circuits (continued)

We are looking into the verilog coding of multiplier algorithm; we are multiplying two numbers of 11 bits and eight bits, say n1 and n2. We have also seen the definition of the various I/Os of the module multi 11s by 8s. We have also seen the wire listing. Right up to this we were looking into different reg wire and reg declaration.

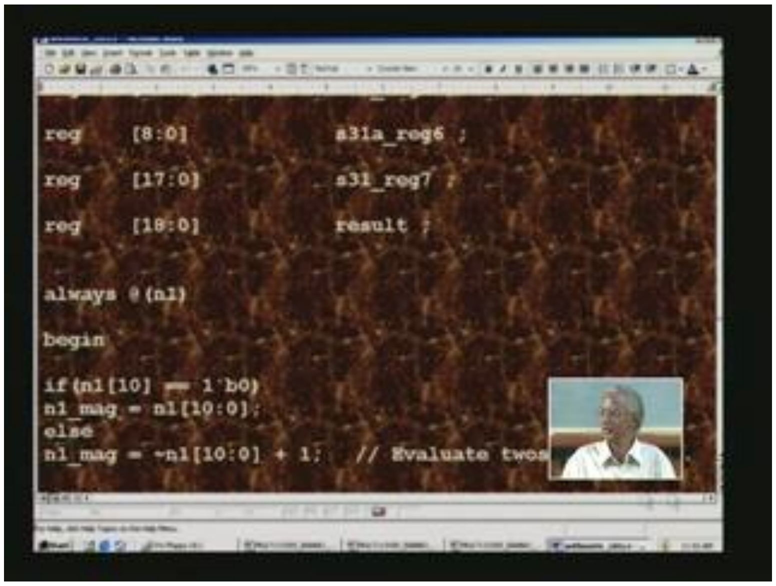
(Refer Slide Time: 3:06)



```
reg     [10:0]          p1_reg2 ;
reg     [10:0]          p2_reg2 ;
reg     [10:0]          p3_reg2 ;
reg     [10:0]          p4_reg2 ;
reg     [10:0]          p5_reg2 ;
reg     [10:0]          p6_reg2 ;
reg     [10:0]          p7_reg2 ;
reg     [10:0]          p8_reg2 ;

reg     [12:0]          s11_reg3 ;
reg     [12:0]          s12_reg3 ;
reg     [12:0]          s13_reg3 ;
reg     [12:0]          s14_reg3 ;

reg     [12:0]          s11_reg4 ;
reg     [12:0]          s12_reg4 ;
reg     [12:0]          s13_reg4 ;
```

I think we have gone this far last time. These are all for subsequent stages say, reg 2 then reg 3 are all clock 2 and 3. All of which naturally the signals would be a registers because we use in an always block.

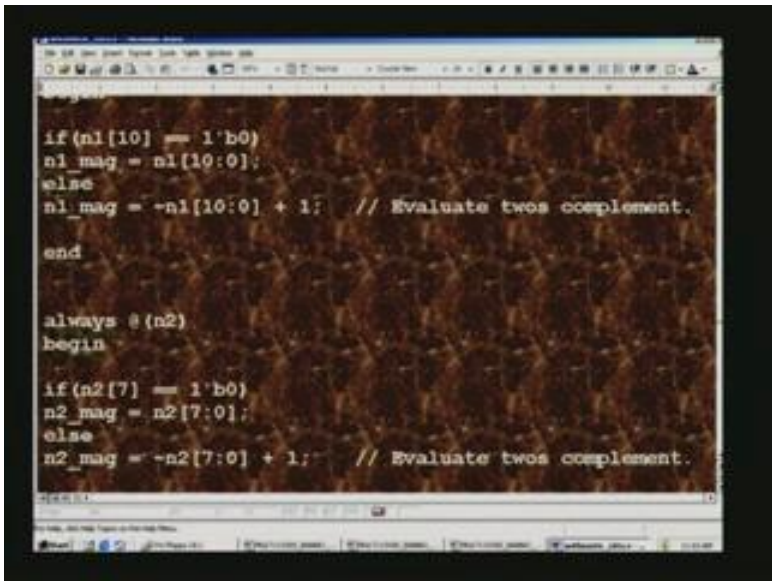
(Refer Slide Time: 3:27)



```
reg     [8:0]             a31a_reg6 ;

reg     [17:0]            s31_reg7 ;

reg     [18:0]            result ;


always @(n1)

begin

if(n1[10] == 1'b0)
n1_mag = n1[10:0];
else
n1_mag = ~n1[10:0] + 1;   // Evaluate twos
```

To start with, the final result will be in 19 bits and last bits will be signed. Here, we have a combinational circuit to start with.

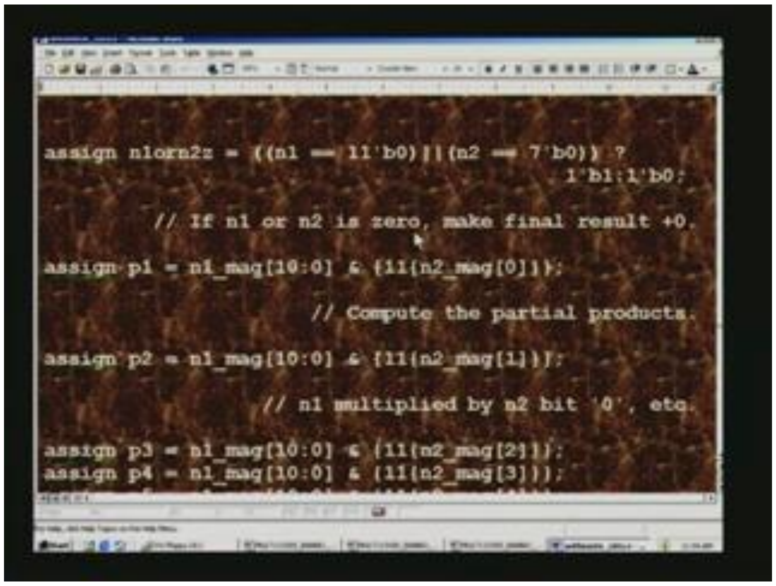
(Refer Slide Time: 3:38)



```
if(n1[10] == 1'b0)
n1_mag = n1[10:0];
else
n1_mag = ~n1[10:0] + 1;   // Evaluate twos complement.

end


always @(n2)
begin

if(n2[7] == 1'b0)
n2_mag = n2[7:0];
else
n2_mag = ~n2[7:0] + 1;    // Evaluate twos complement.
```

What we are going to do is, as mentioned before, we will remove only the magnitude of the two numbers n1 and n2 in these two blocks. So we take this value only whenever n1 changes from one value to another. Now, if n1 10, which happens to be the sign bits is 0 that implies it is a positive number. So just take the n1 as such and call it as n1 magnitude. This means n1 number is in magnitude form. So is the case for n2 magnitude, which is basically the same here. If MSB, that is, the sign bit is 1 and else will be applied. In that case, what we will do is take the two's complement. This is nothing but two's complement. We have already seen that we can get a magnitude merely by taking two's complement and that is what is here. That is how we extract the magnitude of the two numbers and then apply the algorithm here afterwards. These are all combinational circuits.

(Refer Slide Time: 04:53)



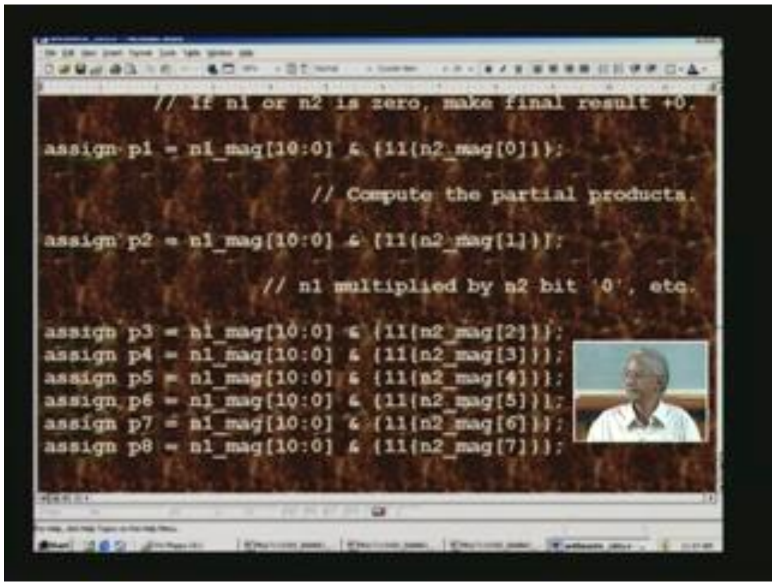
```
assign n1orn2z = ((n1 == 11'b0)||(n2 == 7'b0)) ?
                                          1'b1:1'b0;

            // If n1 or n2 is zero, make final result +0.

assign p1 = n1_mag[10:0] & {11{n2_mag[0]}};

                            // Compute the partial products.

assign p2 = n1_mag[10:0] & {11{n2_mag[1]}};

                        // n1 multiplied by n2 bit '0', etc.

assign p3 = n1_mag[10:0] & {11{n2_mag[2]}};
assign p4 = n1_mag[10:0] & {11{n2_mag[3]}};
```

This is also another combinational statement wherein we inspect whether n1 and n2 are 0s. Even if one is a 0 the final result will be 0. If you do not take this one you may get a problem later on with a finite fixed point arithmetic which we are dealing with. Note that all our arithmetic circuits that we have used so far including this multiplier and adder will be fixed point arithmetic and not floating point arithmetic because floating point arithmetic is more complex and would take more chip area. This is enough for the present and hence, we use only fixed point arithmetic. This particular statement evaluates whether it is a 0 or not. If the result is 0, this will be a 1. That is what this statement means. See, if n1 is 0 or if n2 is 0, then 1 is put into this signal and this is a wire there.

Similarly, we evaluate p1. We mentioned before, we had already taken in our algorithm p1 through p8.

But how do you get this? So we have to look into this. What we saw in the earlier example is we inspected the n2 number and let us say MSB happens to be 0 bits width. We inspect the particular bits, if it is a 0, the partial product will all be 0 that is all the bits will be 0 pertaining to p1, which is nothing other than n1. If it is a 1, the final partial product will be same as n1. Whatever is the n1 number you will get exactly the same thing. So in one case all of them will be 0s and in the other case it will be the duplication of this number. So either this number is got or 0 is got. This can be easily evaluated by merely ANDing each of these bits which are concatenated. For example, we have taken the n2 bits LSB alone. We are replicating, rather, we are repeating 11 times totally. Each of these bits is under with each of the bits of n1 number. Naturally, if you AND, even if there is one 0 at any point that particular bits will be forced to 0. That is how you get 0 here. If one bits happens to be 0 all 11 bits will be 0 because it is concatenation of all the same bits. So, if the first bit of n2 is 0, naturally the final result will be 0 for all the bits. That is what it is here. If it is 1, all will be 1 and therefore, you will get exactly n1 repeated and assigned to the p1.

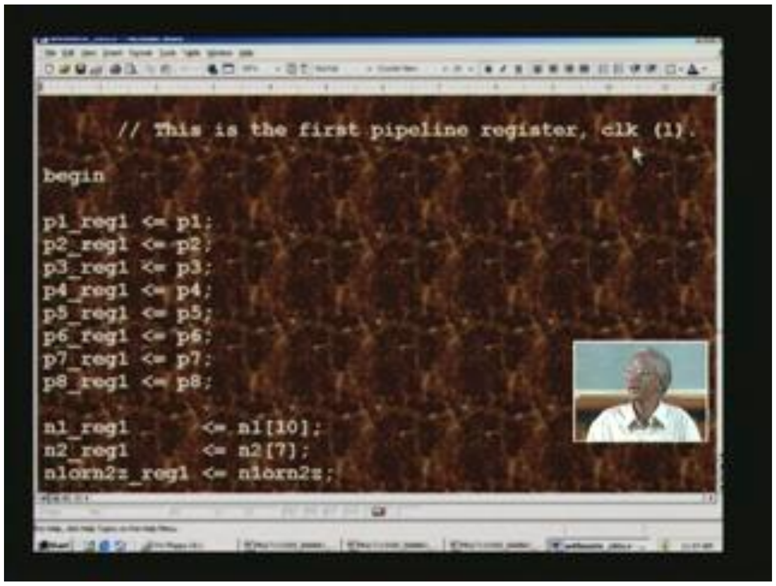
(Refer Slide Time: 07:56)



```
            // If n1 or n2 is zero, make final result +0.

assign p1 = n1_mag[10:0] & {11{n2_mag[0]}};

                        // Compute the partial products.

assign p2 = n1_mag[10:0] & {11{n2_mag[1]}};

                    // n1 multiplied by n2 bit '0', etc.

assign p3 = n1_mag[10:0] & {11{n2_mag[2]}};
assign p4 = n1_mag[10:0] & {11{n2_mag[3]}};
assign p5 = n1_mag[10:0] & {11{n2_mag[4]}};
assign p6 = n1_mag[10:0] & {11{n2_mag[5]}};
assign p7 = n1_mag[10:0] & {11{n2_mag[6]}};
assign p8 = n1_mag[10:0] & {11{n2_mag[7]}};
```

So is the case for all the partial product p1 through p8 here. That is what is shown there.

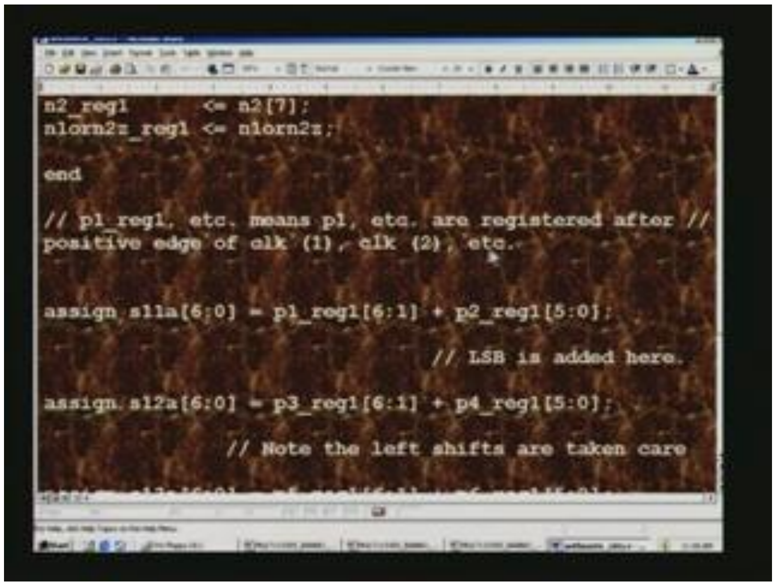
(Refer Slide Time: 08:05)



```
     // This is the first pipeline register, clk (1).
begin

p1_reg1  <= p1;
p2_reg1  <= p2;
p3_reg1  <= p3;
p4_reg1  <= p4;
p5_reg1  <= p5;
p6_reg1  <= p6;
p7_reg1  <= p7;
p8_reg1  <= p8;

n1_reg1         <= n1[10];
n2_reg1         <= n2[7];
n1orn2z_reg1  <= n1orn2z;
```

Now the first pipeline starts and when the positive edge of clock is encountered, this is first pipeline register clock one. We will assign to registers the partial product that we have got in using assign statements earlier. This is the pipeline register actually. So at every point of time, at every clock, you will see a register within always block at positive edge clock. Whatever comes within the positive edge clock always block and will all be registers. That is the reason why given a nomenclature like this. 1 stands for the first clock one so that you can readily find out to which stage a particular signal belongs. These are all the partial products that we had using assign statements. That we will transfer it to reg 1 when clock strikes; so also for the n1 and n2 sign bit. This is actually n1, this 10th bits is sign bits so we preserve that as n1 reg1 and n2 reg1 here for the sign bits of n2. We also preserve whether it is a 0 or the product is 0 or not. We will have to propagate this at every stage because you are will use this only at the fag end. If you forget this statement naturally the whole system will be a black out; it will not work right. So is the case with any statement even now. If a single statement is missing nothing will work.

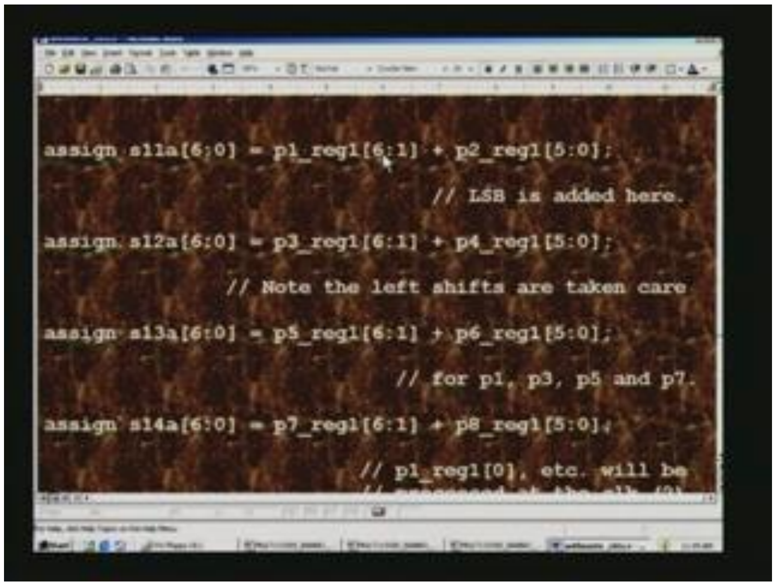
(Refer Slide Time: 09:40)



```
n2_reg1       <= n2[7];
n1orn2z_reg1 <= n1orn2z;

end

// p1_reg1, etc. means p1, etc. are registered after //
positive edge of clk (1), clk (2), etc.


assign s11a[6;0] = p1_reg1[6:1] + p2_reg1[5:0];

                                  // LSB is added here.

assign s12a[6:0] = p3_reg1[6:1] + p4_reg1[5:0];

                   // Note the left shifts are taken care
```

So p1 reg1 etc. means p1 etc., that is what we have seen here are registered after the positive edge of clock 1, clock 2 etc. So is the case for other clocks that we are going to see shortly.

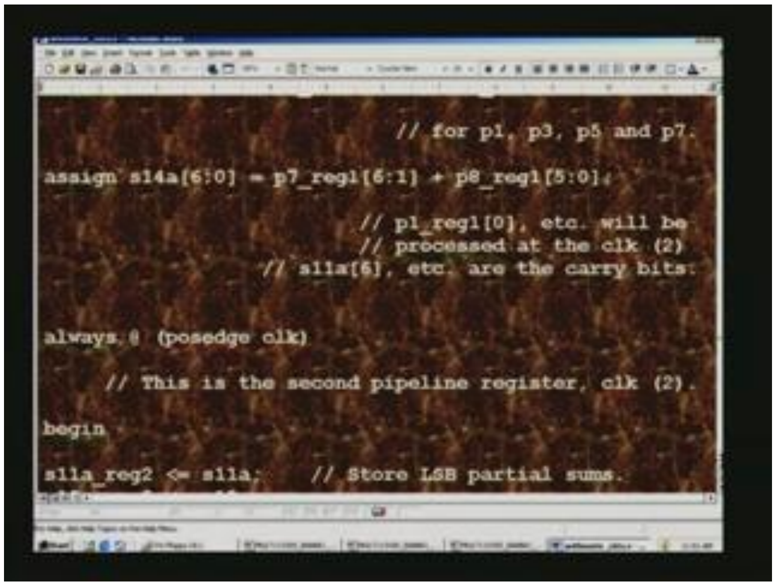
(Refer Slide Time: 09:55)



```
assign s11a[6;0] = p1_reg1[6:1] + p2_reg1[5:0];

                                  // LSB is added here.

assign s12a[6:0] = p3_reg1[6:1] + p4_reg1[5:0];

                   // Note the left shifts are taken care

assign s13a[6:0] = p5_reg1[6:1] + p6_reg1[5:0];

                              // for p1, p3, p5 and p7.

assign s14a[6:0] = p7_reg1[6:1] + p8_reg1[5:0];

                             // p1_reg1[0], etc. will be
```

In the first stage, what we will do is we take only LSB, half the number of total partial product. This is the very first line of the partial product we had. We also mentioned that 0 need not be added initially because you can straightaway put that particular result after doing all the addition. That is why 0 does not come here. So we start with 1 through 6. P2 is one bits shifted left and added only then. That is what we do here. We said that in verilog it is very easy; you do not have to do any shifting operation. We are doing in addition, right within addition we can do the shifting also without any clock penalty. That is the beauty of this verilog. Note that one we are taking the very LSB here 0 through 5; total will be 6 bits; here also 6 bits and it is equivalent to having p2 is the second one. So what we have is if you take this p1 number 6 through 1 then 0 bits also you assume and this particular 0 after left shifting should align itself with 1. That is the reason why we align 1 and 0 here. It is equivalent to left shifting by 1. This is the LSB addition and this 0 you should not forget to add at later stage. So you remember that p1 reg1 0 bits which has not been accounted here will have to be accounted later on. This LSB is added here and note the left shifts are taken care of. That is what we had described here. And we have s11 through s14 doing precisely the same set of operation but on different partial product p1, p2, p3, p4, p5, p6, p7, p8.

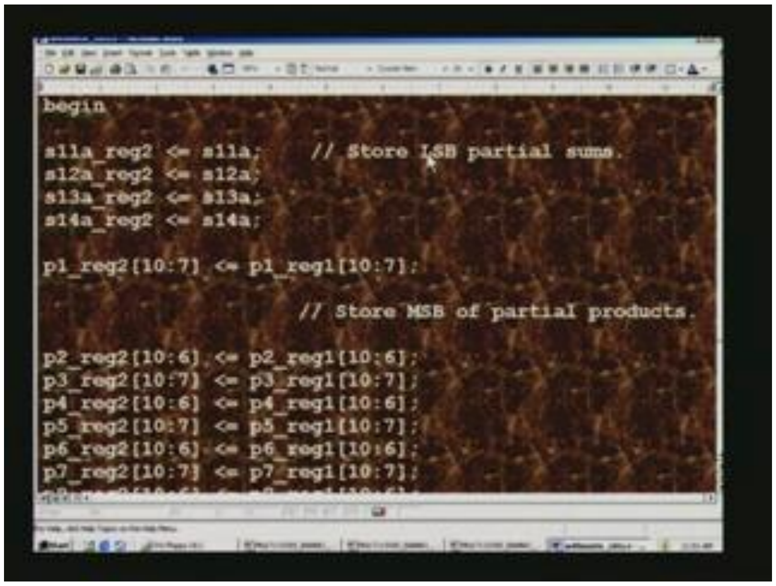
(Refer Slide Time: 11:48)



```
                                          // for p1, p3, p5 and p7.

assign s14a[6:0] = p7_reg1[6:1] + p8_reg1[5:0];

                                        // p1_reg1[0], etc. will be
                                        // processed at the clk (2)
                                   // s11a[6], etc. are the carry bits.


always @ (posedge clk)

     // This is the second pipeline register, clk (2).

begin

s11a_reg2 <= s11a;     // Store LSB partial sums.
```

P1 reg1 0 we have mentioned earlier etc., will be processed at the clock 2; s11a6 etc., are the carry bits. This will be the carry bits resulting from LSB addition. We also have to take care of this while adding the MSB. With the arrival of the next positive edge of clock this is the second pipeline register clock 2.

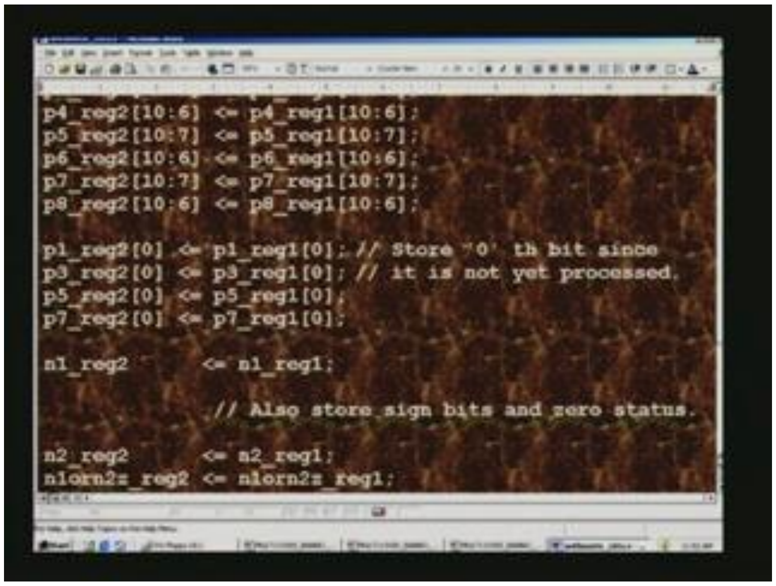
(Refer Slide Time: 12:12)



```
begin

s11a_reg2 <= s11a;     // Store LSB partial sums.
s12a_reg2 <= s12a;
s13a_reg2 <= s13a;
s14a_reg2 <= s14a;

p1_reg2[10:7] <= p1_reg1[10:7];

                      // Store MSB of partial products.

p2_reg2[10:6] <= p2_reg1[10:6];
p3_reg2[10:7] <= p3_reg1[10:7];
p4_reg2[10:6] <= p4_reg1[10:6];
p5_reg2[10:7] <= p5_reg1[10:7];
p6_reg2[10:6] <= p6_reg1[10:6];
p7_reg2[10:7] <= p7_reg1[10:7];
```

What we do here is store LSB partial sums. We have already evaluated the LSB partial sums and we need to store that here and put it in a pipeline register. This is how it should propagate because we have not yet used the result and will be using it only later on. It will only be used at subsequent clock pulse; till then you have to keep propagating. All this would mean extra chip area but the goal is to speed up the entire operation we have and this is the price we are paying for it. Similarly, store MSB of partial products. They are all, p1 through p8. This is the MSB here. Note that we have used 6 through 1 there and we have not yet used 0. We are now processing the MSB. What we do is merely registering this in order to process.

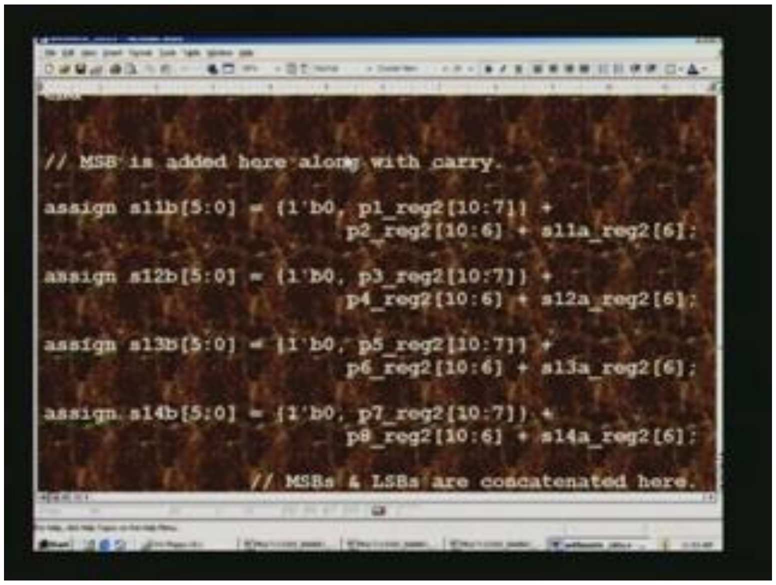
(Refer Slide Time: 13:07)



```
p4_reg2[10:6] <= p4_reg1[10:6];
p5_reg2[10:7] <= p5_reg1[10:7];
p6_reg2[10:6] <= p6_reg1[10:6];
p7_reg2[10:7] <= p7_reg1[10:7];
p8_reg2[10:6] <= p8_reg1[10:6];

p1_reg2[0] <= p1_reg1[0]; // Store '0' th bit since
p3_reg2[0] <= p3_reg1[0]; // it is not yet processed.
p5_reg2[0] <= p5_reg1[0];
p7_reg2[0] <= p7_reg1[0];

n1_reg2      <= n1_reg1;

               // Also store sign bits and zero status.

n2_reg2      <= n2_reg1;
n1orn2z_reg2 <= n1orn2z_reg1;
```

Here also you can store 0th bits since it is not yet processed. We have seen that we have not added 0th bits but we have to register that also. Otherwise, that will be lost and we will process later on. Similarly the sign also will have to be propagated. Store sign bits and zero status. That is what is here. n1 sign bit, n2 sign bit, whether n1 or n2 is 0 or the result is 0 that also is propagated.

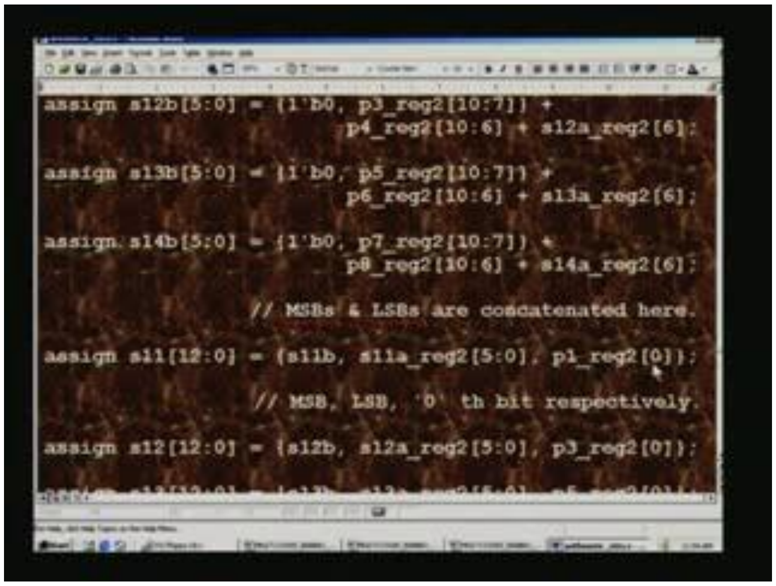
(Refer Slide Time: 13:42)



```
// MSB is added here along with carry.

assign s11b[5:0] = {1'b0, p1_reg2[10:7]} +
                          p2_reg2[10:6] + s11a_reg2[6];

assign s12b[5:0] = {1'b0, p3_reg2[10:7]} +
                          p4_reg2[10:6] + s12a_reg2[6];

assign s13b[5:0] = {1'b0, p5_reg2[10:7]} +
                          p6_reg2[10:6] + s13a_reg2[6];

assign s14b[5:0] = {1'b0, p7_reg2[10:7]} +
                          p8_reg2[10:6] + s14a_reg2[6];

                    // MSBs & LSBs are concatenated here.
```

There is a little scope for you to optimize. You have a look into that and see where you can marginally optimize. I have not taken much trouble in optimizing; probably, left it to the synthesis tool but you can do it and find out whether your manual optimization saves any chip area. It will be very nominal, that is why I will not go into depths. So MSB is added here along with carry. We have added LSB earlier and now we are adding MSB. But LSB had resulted in carry so that is what we are adding here. That is this here. Once again, these are the p1 and p2 numbers and we saw that in MSB we need to put one more bits here. This may not be really necessary; you experiment with it. I had just put so as to keep track of the number of bits. For example, it is 10 to 7 here; whereas, in p2 it is 10 to 6. We have seen why it is so. This is a one bits shift and so is the case for the other four results obtained by adding p1, p2 in this stage and then p3, p4, p5, p6 and the last two here.

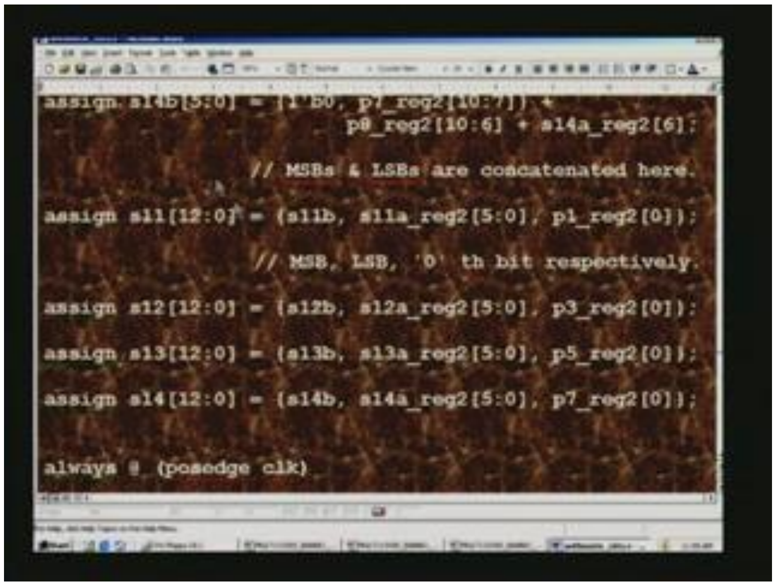
(Refer Slide Time: 15:18)



```
assign s12b[5:0] = {1'b0, p3_reg2[10:7]} +
                          p4_reg2[10:6] + s12a_reg2[6];

assign s13b[5:0] = {1'b0, p5_reg2[10:7]} +
                          p6_reg2[10:6] + s13a_reg2[6];

assign s14b[5:0] = {1'b0, p7_reg2[10:7]} +
                          p8_reg2[10:6] + s14a_reg2[6];

                    // MSBs & LSBs are concatenated here.

assign s11[12:0] = {s11b, s11a_reg2[5:0], p1_reg2[0]};

                    // MSB, LSB, '0' th bit respectively.

assign s12[12:0] = {s12b, s12a_reg2[5:0], p3_reg2[0]};
```

MSBs and LSBs are concatenated here so this is the partial product. We have not processed earlier as such said 0 bits. So in the partial product of p1 we started only with 1, therefore, you should remember to put that here. So, what we do here is nearly arrange all MSB and LSB and also the first bit which was not considered earlier. We have only the final result of the first stage. We take this MSB; this is the result of MSB s11b. They are all part of this. This is MSB we have seen. We need to put that result on the top here first followed by the LSB result that we had computed earlier. This MSB and this LSB we are concatenating. Concatenating is putting together as a single number. That is this one. We should also concatenate the last bits that we have not processed which happens to be the very same p1 LSB. That is how we get the result of the first edge.

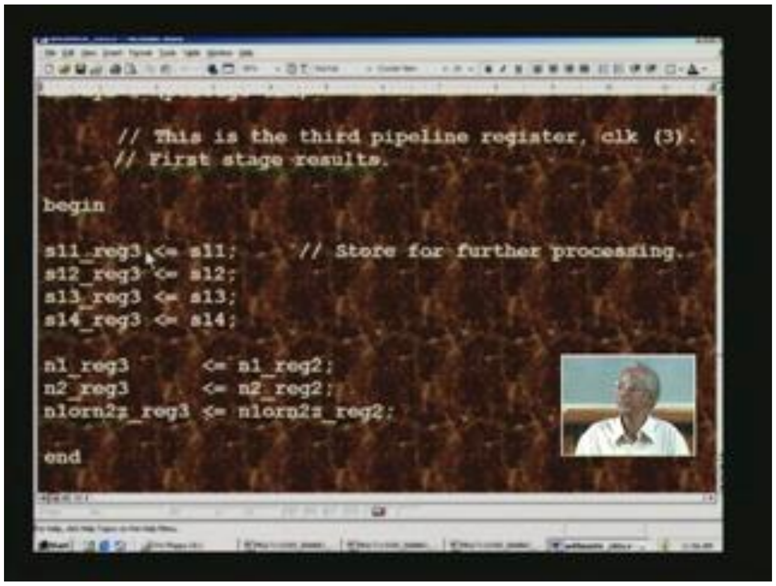
(Refer Slide Time: 16:36)



```
assign s14b[5:0] = {1'b0, p7_reg2[10:7]} +
                              p8_reg2[10:6] + s14a_reg2[6];

                 // MSBs & LSBs are concatenated here.

assign s11[12:0] = {s11b, s11a_reg2[5:0], p1_reg2[0]};

                 // MSB, LSB, '0' th bit respectively.

assign s12[12:0] = {s12b, s12a_reg2[5:0], p3_reg2[0]};

assign s13[12:0] = {s13b, s13a_reg2[5:0], p5_reg2[0]};

assign s14[12:0] = {s14b, s14a_reg2[5:0], p7_reg2[0]};


always @ (posedge clk)
```

For the first edge, you have four such outputs. This is the first stage of addition that we have seen in the diagram earlier for the algorithm. This is all precisely the same thing.

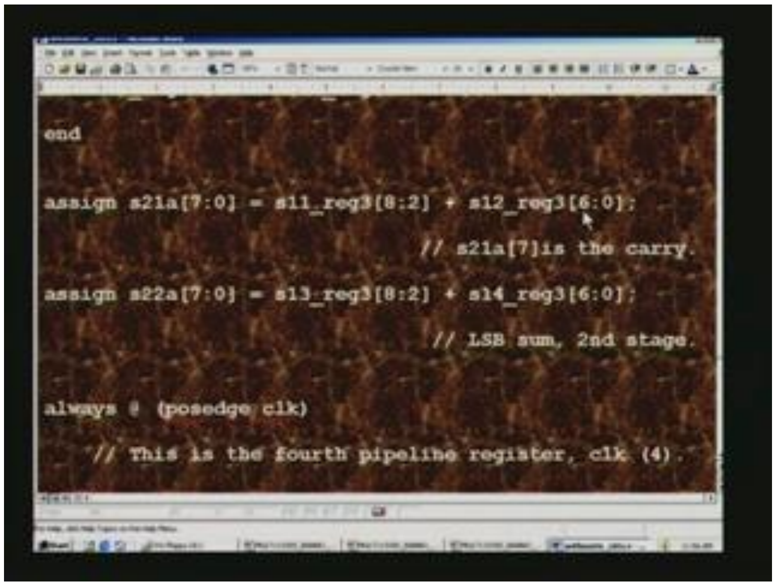
(Refer Slide Time: 16:46)



```
     // This is the third pipeline register, clk (3).
     // First stage results.

begin

s11_reg3 <= s11;       // Store for further processing.
s12_reg3 <= s12;
s13_reg3 <= s13;
s14_reg3 <= s14;

n1_reg3       <= n1_reg2;
n2_reg3       <= n2_reg2;
n1orn2z_reg3 <= n1orn2z_reg2;

end
```

Then the next clock, that is the clock 3; it is the third pipeline register. First stage results are stored for further processing. s11 through s14, these are all the first stage results and we store it in pipeline register and this being clock 3 we give this nomenclature. We should once again store the n1 and n2 sign as well as the 0 results.

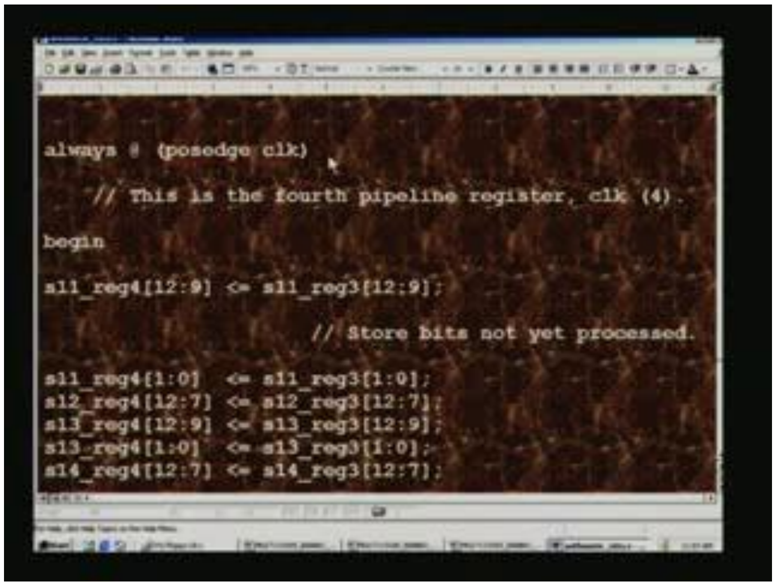
(Refer Slide Time: 17:17)



```
end

assign s21a[7:0] = s11_reg3[8:2] + s12_reg3[6:0];

                                    // s21a[7]is the carry.

assign s22a[7:0] = s13_reg3[8:2] + s14_reg3[6:0];

                                     // LSB sum, 2nd stage.


always @ (posedge clk)

    // This is the fourth pipeline register, clk (4).
```

This is the second stage. Before the clock strikes, we will have to do the evaluation to speed up the process. That is what we do here. Once again, we process only the LSB here. In the second stage, as I mentioned before, this is the first row of the second stage and 1 through 0 we will left shift the second number which is why we have 6 through 0. So, instead of 2 you have 0; corresponding thing is 2, which means that s12 is shifted by two bits. That is the second stage we have seen in the diagram earlier. As we have shifted two bits, naturally, for this 1 and 0, that is, two bits of LSB need not be added. We can merely duplicate that thing later on. We add only other than those two bits. We add seven bits here and this also naturally seven bits and outcomes eight bits result and this s21a7 will be the carry bits. So is the case for the next result. We have just two results in the second stage. In the first stage, there are four results and in the second stage only two results because two, two numbers are added, so four of them. LSB sum second stage is got here.

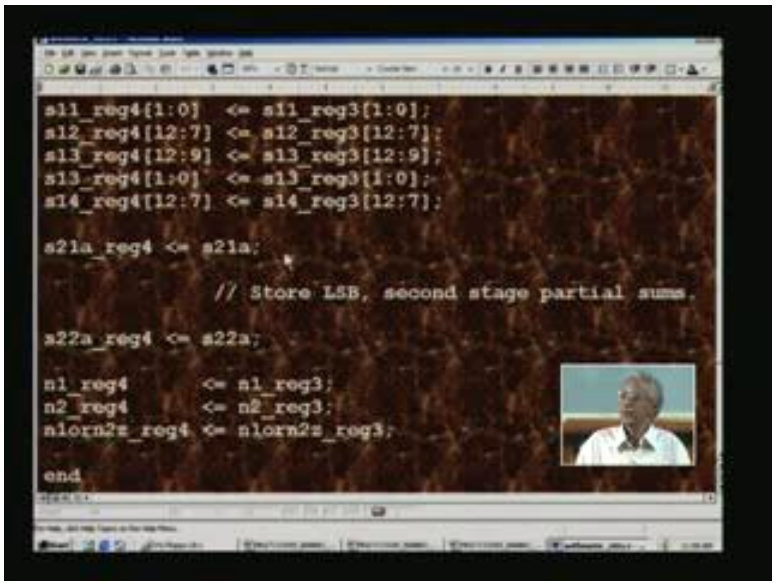
(Refer Slide Time: 18:45)



```
always @ (posedge clk)

    // This is the fourth pipeline register, clk (4).

begin

s11_reg4[12:9] <= s11_reg3[12:9];

                        // Store bits not yet processed.

s11_reg4[1:0]  <= s11_reg3[1:0];
s12_reg4[12:7] <= s12_reg3[12:7];
s13_reg4[12:9] <= s13_reg3[12:9];
s13_reg4[1:0]  <= s13_reg3[1:0];
s14_reg4[12:7] <= s14_reg3[12:7];
```

With the arrival of the positive edge clock this is the fourth pipeline register clock 4 and we store bits not yet processed. These are all the bits not yet processed especially, the MSB we have not processed; other bits, two bits, we have seen for s11 register we have not processed 1 0. It was here, see s13. You can see all this here, 1 0 not used here. That is what is here s13 1 0. So is the case for s11. And we have also not processed MSB. Therefore, there is no need for you to propagate these through pipeline registers.

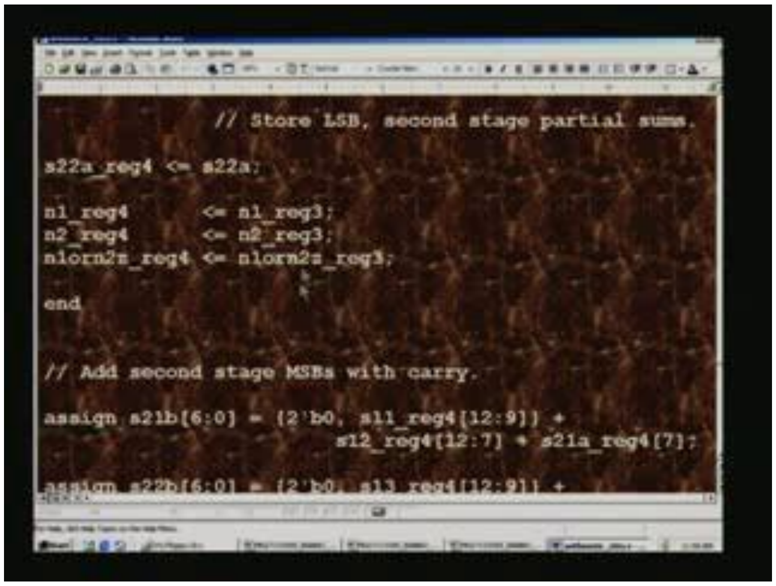
(Refer Slide Time: 19:28)



```
s11_reg4[1:0]   <= s11_reg3[1:0];
s12_reg4[12:7]  <= s12_reg3[12:7];
s13_reg4[12:9]  <= s13_reg3[12:9];
s13_reg4[1:0]   <= s13_reg3[1:0];
s14_reg4[12:7]  <= s14_reg3[12:7];

s21a_reg4 <= s21a;

               // Store LSB, second stage partial sums.

s22a_reg4 <= s22a;

n1_reg4      <= n1_reg3;
n2_reg4      <= n2_reg3;
n1orn2s_reg4 <= n1orn2s_reg3;

end
```

So also is the s21a, which is store of LSB second stage partial sums. You have already computed LSB and that is the second stage; that partial sum is also to be stored and we put it in another pipeline register. So is the case for s22a, which is the next output of the second stage.

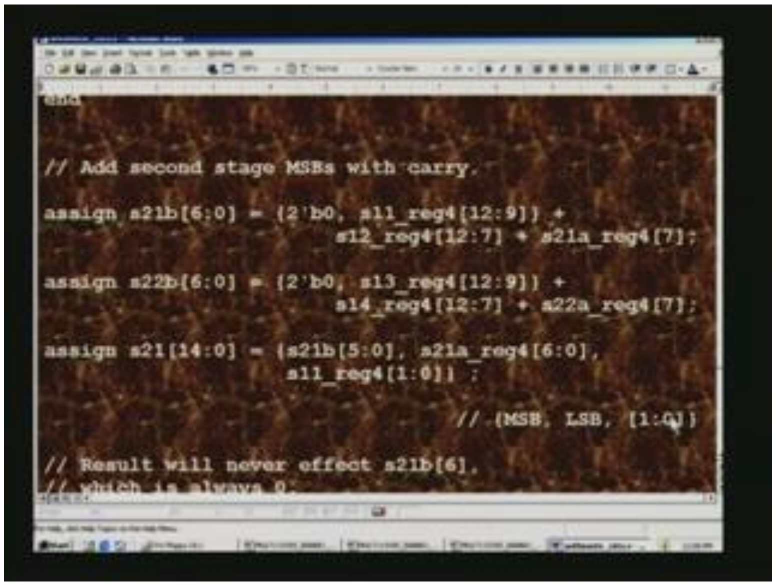
(Refer Slide Time: 19:51)



```
                // Store LSB, second stage partial sums.

s22a_reg4 <= s22a;

n1_reg4      <= n1_reg3;
n2_reg4      <= n2_reg3;
n1orn2z_reg4 <= n1orn2z_reg3;

end


// Add second stage MSBs with carry.

assign s21b[6:0] = {2'b0, s11_reg4[12:9]} +
                          s12_reg4[12:7] + s21a_reg4[7];

assign s22b[6:0] = {2'b0, s13_reg4[12:9]} +
```

We should also not forget these n1 and n2 sign bits as well as the final result is 0 or not. We now add second stage MSBs with carry.

(Refer Slide Time: 20:10)

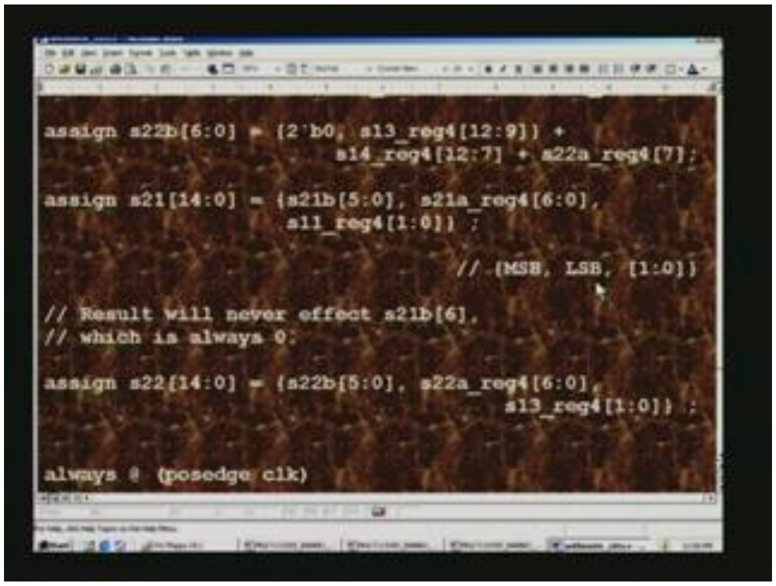


```
end


// Add second stage MSBs with carry.

assign s21b[6:0] = {2'b0, s11_reg4[12:9]} +
                          s12_reg4[12:7] + s21a_reg4[7];

assign s22b[6:0] = {2'b0, s13_reg4[12:9]} +
                          s14_reg4[12:7] + s22a_reg4[7];

assign s21[14:0] = {s21b[5:0], s21a_reg4[6:0],
                    s11_reg4[1:0]} ;

                                      // {MSB, LSB, [1:0]}

// Result will never effect s21b[6],
// which is always 0.
```

What we are doing here once again may or may not be required. You can just experiment here. It is there for the sake of completeness. Probably an optimising tool such as simplify will do it automatically and even if you put some extraneous thing do not really bother much about optimizing at your step.

This is MSB here and this is s11. The second stage outputs are s11 and s12 and once again we are doing a shift because what we are trying to do is only MSB addition. So the same two bits shift is applicable for MSB as well because this is still part of the stage. We add two numbers s11 and s12. Similarly, this is s11, s12 and s13, s14 are there in the second stage for MSB. So we need to add with carry of the LSB. That is what is here. Out comes this result and this will be a final sign bits. Let us have a look on being done here.

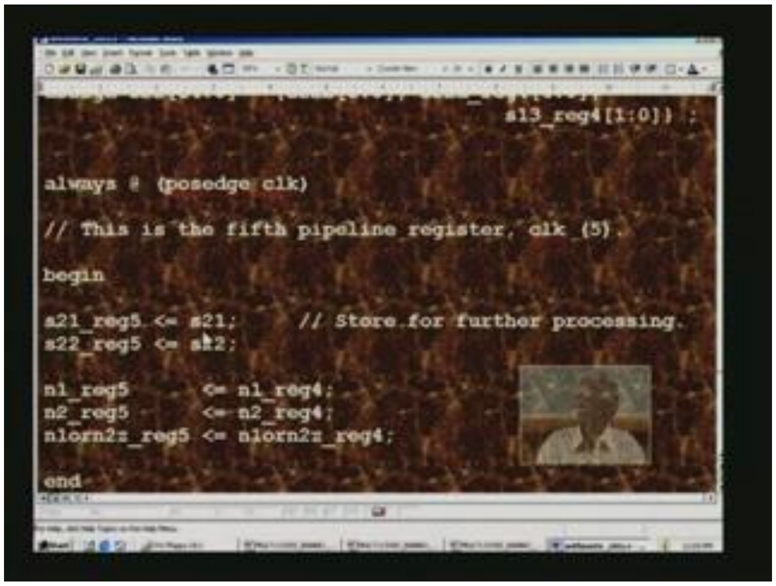
(Refer Slide Time: 21:30)



```
assign s22b[6:0] = {2'b0, s13_reg4[12:9]} +
                       s14_reg4[12:7] + s22a_reg4[7];

assign s21[14:0] = {s21b[5:0], s21a_reg4[6:0],
                    s11_reg4[1:0]} ;

                              // {MSB, LSB, [1:0]}

// Result will never effect s21b[6],
// which is always 0;

assign s22[14:0] = {s22b[5:0], s22a_reg4[6:0],
                                 s13_reg4[1:0]} ;


always @ (posedge clk)
```

So, this is MSB, then LSB and then 1 through 0. We have merely added the MSB; we have not put it together. So we need to put them together to form one result. That is what is here, and that will be 14 through 0 bits say, 15 bits. This is got from s21 6 bits, s21 is seven bits and then finally s11. These are all the numbers that we have not processed earlier. The LSB two bits so that is what is here. This was the LSB result earlier and this was the MSB result. Another point you will note is that we have used 6 through 0; whereas, here we have only used 5 through 0. The reason is that this has a duplicate sign bits or it is a duplicated bits; it will not affect the last bits; it always remains at 0. You will realize this when you really take an example and work it out. That is the reason why this one gets affected. In fact, we do not have to remove it physically; the synthesis tool will automatically take care of it. However, if you can spot it out, you can do it right in your code itself. That is what I have spotted out here with an example. Therefore I just limited to this. Otherwise, do not worry about that because the synthesis tool will automatically do the optimisation. So we put the MSB of the second stage addition and LSB of this, followed by concatenating and also not forgetting the last two bits that we have not processed earlier. This makes up the second stage complete result. The result will never effect s21 6 bits, which is always 0. This is what we have already seen so just as we had s21, we will have s22 here. This is precisely the same as what we have seen before for the other two outputs of the second stage.

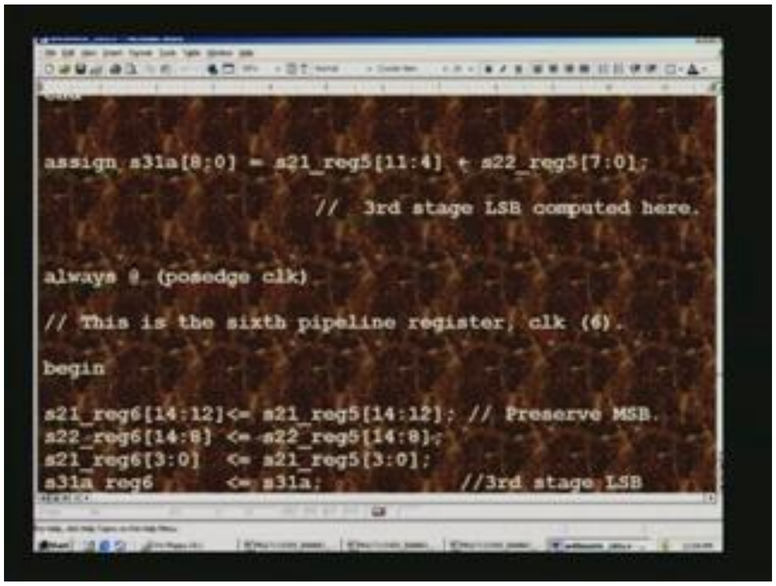
(Refer Slide Time: 23:54)



```
                                        s13_reg4[1:0]} ;

always @ (posedge clk)

// This is the fifth pipeline register, clk (5).

begin

s21_reg5 <= s21;      // Store for further processing.
s22_reg5 <= s22;

n1_reg5       <= n1_reg4;
n2_reg5       <= n2_reg4;
n1orn2z_reg5 <= n1orn2z_reg4;

end
```

So far we have seen clock 4 and what happens. This is the fifth pipeline and we need to once again store sum of these values. For example, s21 and s22 are what we have just now processed of the second stage partial sum. We also need to store the 0 value or the two sign bits and then process further.

(Refer Slide Time: 24:22)

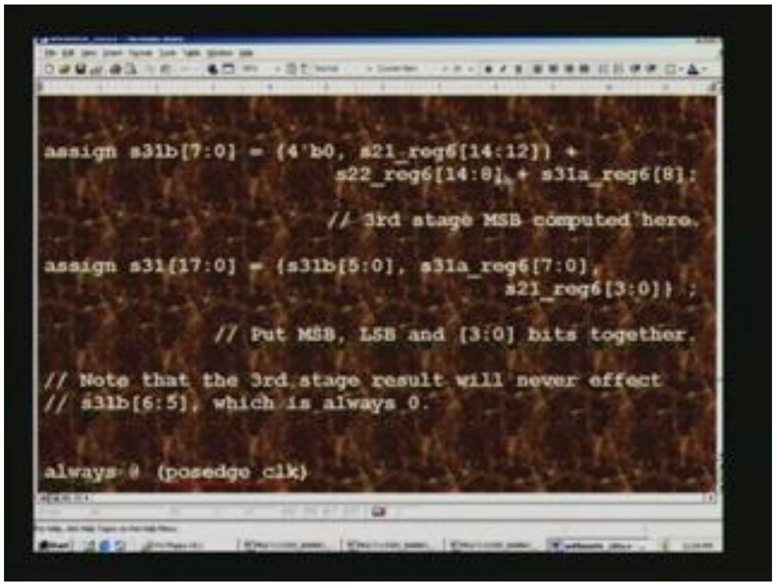


```
assign s31a[8:0] = s21_reg5[11:4] + s22_reg5[7:0];
                        //  3rd stage LSB computed here.

always @ (posedge clk)

// This is the sixth pipeline register, clk (6).

begin

s21_reg6[14:12]<= s21_reg5[14:12]; // Preserve MSB.
s22_reg6[14:8]  <= s22_reg5[14:8];
s21_reg6[3:0]   <= s21_reg5[3:0];
s31a_reg6       <= s31a;            //3rd stage LSB
```

This stage is the third stage; at this stage LSB is computed. In this third stage, the final output is going to be this whereas, the second stage output, which is this one is being taken. In this case, note that we will shift for the third stage by four bits. That is why 3 through 0 not being used and we need to once again take from 0. This 0 corresponds to 4 because we need to left shift 4 bits. This has to two reg, same as s22. When you do this one, outcomes LSB process which we will register when clock 6 arrive.

When the clock strikes we should also preserve MSB from the previous stage because we have not yet processed and propagate through some other register here. That is what we are doing here. This is the third stage LSB also registered here. We have just now processed LSB there and we should once again take care of the two sign bits as well as the 0 results registered here.

(Refer Slide Time: 25:35)

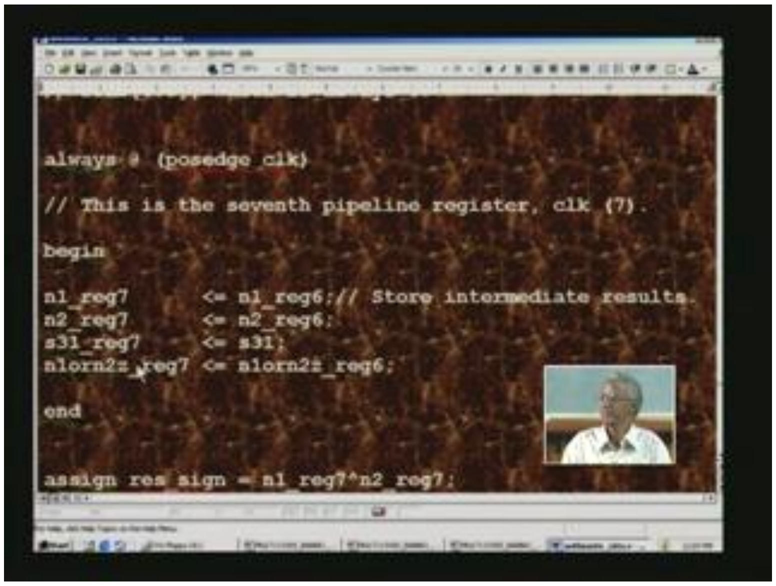


```
assign s31b[7:0] = {4'b0, s21_reg6[14:12]} +
                            s22_reg6[14:8] + s31a_reg6[8];

                            // 3rd stage MSB computed here.

assign s31[17:0] = {s31b[5:0], s31a_reg6[7:0],
                                       s21_reg6[3:0]} ;

                 // Put MSB, LSB and [3:0] bits together.

// Note that the 3rd stage result will never effect
// s31b[6:5], which is always 0.

always @ (posedge clk)
```

So far, we have processed the LSB; we need to process MSB. Once again, we use assign statements and process. As usual, we have put four 0s here although you need not do it. We take the MSB of the first number and MSB of the second number as well. This is taking care of the 4 bits shift. It is taking care of it and you can cross check. You can see from here 14 through 12 is 3 bits plus 4, it is seven bits. To this, we add the carry resulting from the LSB addition which we have done earlier. That is also to be done without fail. This will give you the third stage MSB computation.

You concatenate all the results put together. We have earlier done LSB computation and we are doing the MSB. We have to put them together by concatenation. That is what we are doing here. This one s31b is put here. Once again, these two bits 6 and 7 will be 0s, so we are not using that. Therefore, we optimize 5 to 0 by hand right here. Then, we put together the MSB results and the LSB results. What we have is concatenate of 3 through 0, 4 bits we have not used earlier for s21. That is what we are concatenating here. All this put together will be the final multiplier output which are not yet registered. This puts MSB, LSB and 3 to 0 bits together. That is what we have done here. We have to note that the third stage result will never affect 6 through 5. That is what we have seen here; this is always 0. With the arrival of the positive edge of the clock, this is the seventh pipeline register at clock 7.

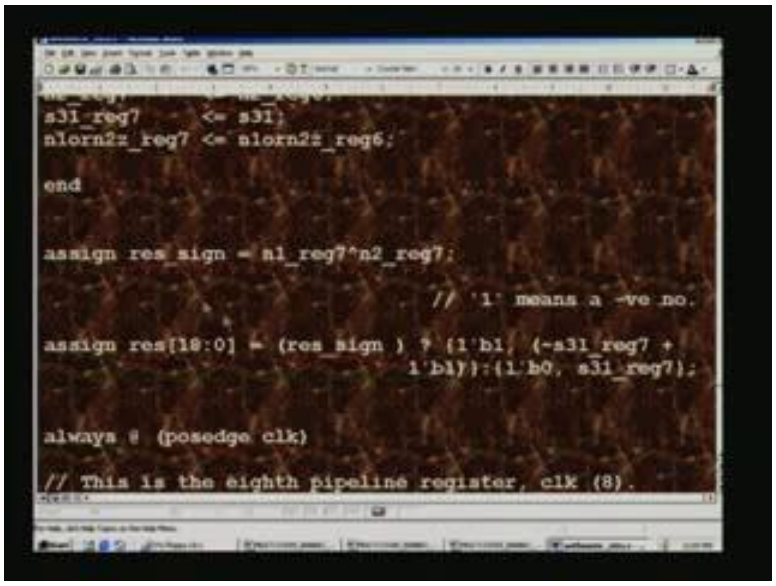
(Refer Slide Time: 27:57)



```
always @ (posedge clk)

// This is the seventh pipeline register, clk (7).

begin

n1_reg7       <= n1_reg6;// Store intermediate results.
n2_reg7       <= n2_reg6;
s31_reg7      <= s31;
n1orn2z_reg7 <= n1orn2z_reg6;

end

assign res_sign = n1_reg7^n2_reg7;
```

We are once again storing all intermediate results. For example, sign bits, that 0 and also the results. This is the result from the third stage that we have just computed. This is the 3, 1 that we have concatenated the MSB, LSB as well as the unused four bits earlier.

(Refer Slide Time: 28:45)



```
s31_reg7      <= s31;
n1orn2z_reg7 <= n1orn2z_reg6;

end

assign res_sign = n1_reg7^n2_reg7;

                                  // '1' means a -ve no.

assign res[18:0] = (res_sign ) ? {1'b1, (~s31_reg7 +
                                1'b1)}:{1'b0, s31_reg7};

always @ (posedge clk)

// This is the eighth pipeline register, clk (8).
```

So, s31 is the final result. That is what we need to register which we will process at the arrival of the next clock. Before that, we do the computation if there is any. Let us see what we are doing here. First, we said we have been propagating all along this n1 sign bits as well as n2- second number sign bits for all. If you take the exclusive OR you get the final result. For example, this is an exclusive airing which is simple in logic because you know the simple multiplication rules. Say, if you multiply plus into plus, you get plus; minus into minus is plus once again. As long as both the sign bits are the same whether it is plus or minus the result will be plus. If one is plus and the other is minus you get a minus; minus is represented as 1 in binary number, so 0 is the positive. So when exclusive OR two numbers, for example, 00 which stands for plus into plus, out comes exclusive airing is 0. Similarly, if you exclusive OR 1 and 1, it will be 10. You have to drop the carry 1 so what you get as the result is only 0. In this case also, minus into minus is plus; similarly, plus into minus, if you put 1 exclusive OR with 0, it will give you 1. So 1 is the final result which will be put here and that naturally is a minus sign because 1 stands for this sign and 0 stands for the positive number. That is what we are evaluating here. That is mere exclusive OR here of the sign bits of the two numbers n1 and n2 which we started multiplying; 1 means a negative number.

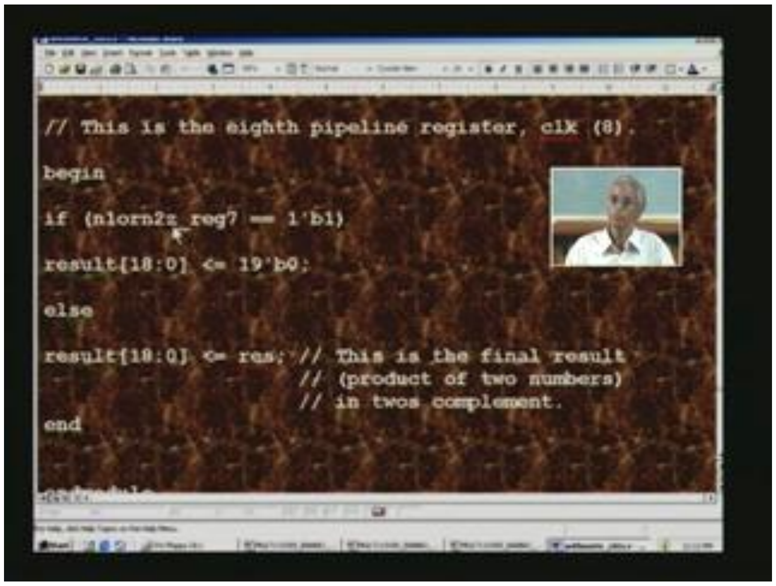
The final result is here. We said in the diagram before the partitioning and diagram. So 18 through 0, we have mentioned that is the final result, which is the product of two numbers n1 and n2. We have done the entire algorithm that we had computed using this verilog code. So far, this handles only the magnitude as mentioned before and we have just evaluated the sign of the number. Based upon this sign bits we will manipulate the final result by reflecting the two’s complement or the number straightaway. For example, if the sign bit is 1, if the final result is 1, which means a negative number, we take two’s complement which we have seen earlier. By taking two’s complement of a signed number you get a magnitude; if you take the two’s complement of the magnitude you will get the signed number, not exactly, we had to force the sign bits separately. That is what is being done here.

By concatenating the last bits to be the 18th bits, we have to force it 1 because you know at this stage that this number is going to be a negative number. If it is negative, this will be 1; if it is 1, this is the MUXs realization. So, the MSB will have to be 1. That is why we force here. So we need to take the two’s complement in order to get the magnitude.

Earlier, I said that incorrectly; the sign bits will have to be separately dealt with. Here, we just take the two's complement of s31. (Refer Slide Time: 31:57) s31 is actually the result in magnitude form. The entire result is 17th through 0.

That is what it is. So you have to take the two's complement and also append 1, that is, a signed bits; this is for the negative number as long as this is 1. If it is 0, that is, a positive number result is positive number. We have only to take that s31 which is the result and add one more bits because that was only 1eight bits and the result must be 19 bits. Therefore, we append 0 implying that this is the positive number. So this is clear here. That is one statement here and another here. This statement is executed if this is 1 otherwise this is assigned to this final result. Either this is assigned or this is assigned depending upon the sign bits here. So, here we have appended 1 because this is going to be a negative number and we have appended 0 which is positive number. Either this 1 or this 0 will reflect as reset 18, that is the MSB. This tells you how we take care of the sign bits as well. We are nearing completion of this design for the multiplier. We have seen that it follows the exact algorithm that we have developed earlier. Finally, we want to register this output for the DCTQ application. That is the reason why we have one more positive edge of clock which is the clock 8. We nearly assign this final value that we have just evaluated depending upon the sign bits is also taken care of. What we have not taken care of is whether n1 or n2 is 0.

(Refer Slide Time: 33:52)

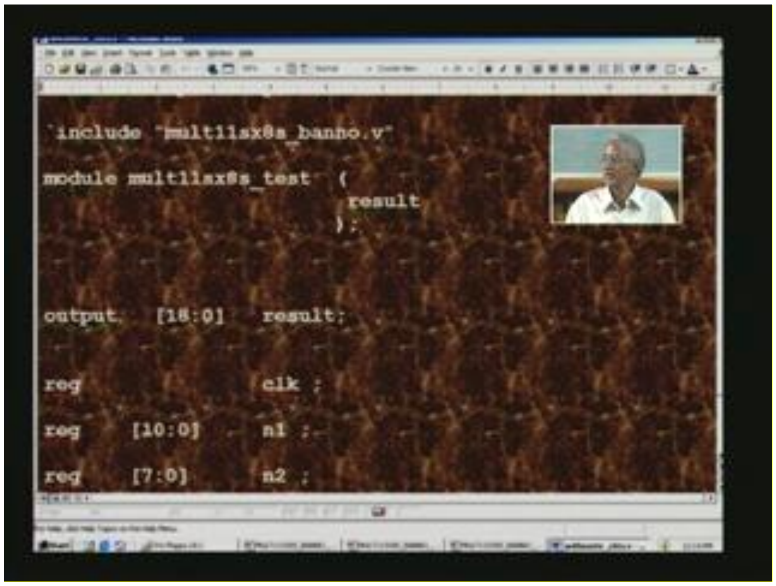


```
// This is the eighth pipeline register, clk (8).
begin
if (n1orn2z_reg7 == 1'b1)
result[18:0] <= 19'b0;
else
result[18:0] <= res; // This is the final result
                     // (product of two numbers)
                     // in twos complement.
end
endmodule
```

Earlier we had 0 flag propagating all through implying whether the final result is 0 or not. The final result is 0 if n1 or n2 is 0. So that has also propagated and based upon that one, if it is 1, the result is 0; so forth result is 0 by this statement. Otherwise, that is the else statement. Assign the same result that we have just computed using that assign statement res, this is what we have. The actual sign number is available; that is the usual normal number, whereas, if the result is 0 you can assign separately.

You can also try dispensing with this 0 flag. You can remove that everywhere right from the beginning and find out if there are any problems. If there is no problem, you can settle for that changed code. If you find any problem, you may have to stick to this. So I will leave it to you as an exercise. This is the final result, product of two numbers in two's complement. We started with n1 and n2 sign numbers, one was 11 bits and the other was eight bits, so out comes 11 plus 8, 19 bits. That is what it is here, eighteenth is a sign bits. This completes the design of the multiplier.

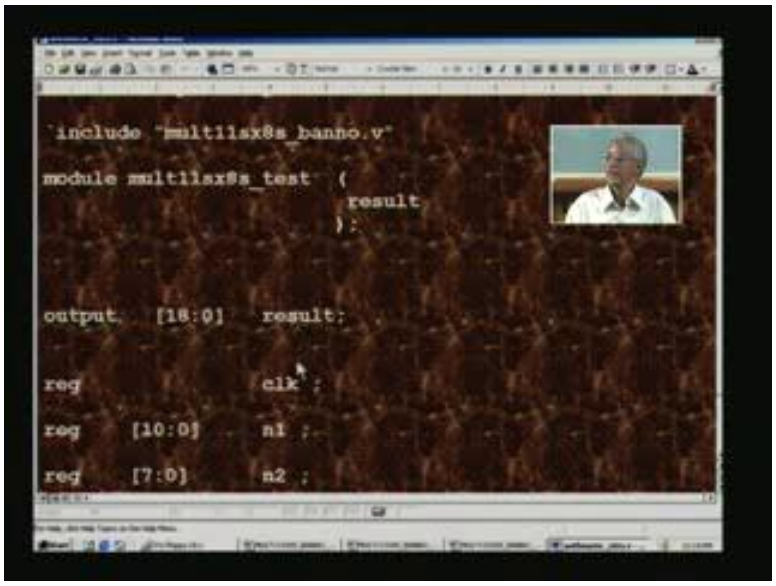
(Refer Slide Time: 35:29)



```
`include "mult11sx8s_banno.v"
module mult11sx8s_test  (
                         result
                        );


output   [18:0]   result;

reg               clk ;
reg     [10:0]    n1 ;
reg     [7:0]     n2 ;
```

Let us have a look at the test bench. Suppose it is 50 megahertz operation, we need a 10 here and we need to include the back annotated designed files which is mult11s into 8s. Of course, we do not add the back annotation for the normal design. This particular test bench is a module, the name of which is declared.

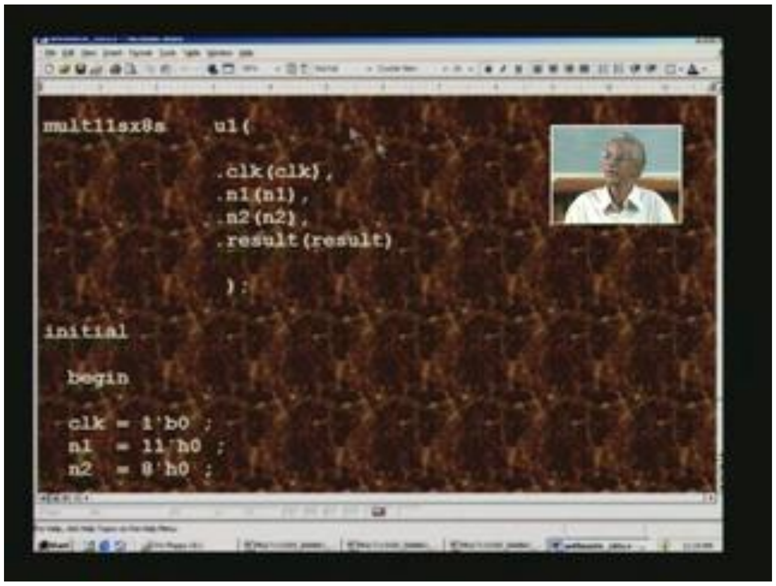
(Refer Slide Time: 35:53)



```
`include "mult11sx8s_banno.v"
module mult11sx8s_test  (
                         result
                        );


output   [18:0]   result;

reg               clk ;
reg     [10:0]    n1 ;
reg     [7:0]     n2 ;
```

You want only the final result, that is the product and that is what is listed here. We declare all the inputs which happen to be simple clock, n1 and n2. Although the algorithm is quite complex the result will be easy to analyse. We have seen that this is 11 bits, tenth bits being sign. So is the case with 7 which is also signed. This is eight bits for n2. We invoke the designed mult11 s into 8 s.

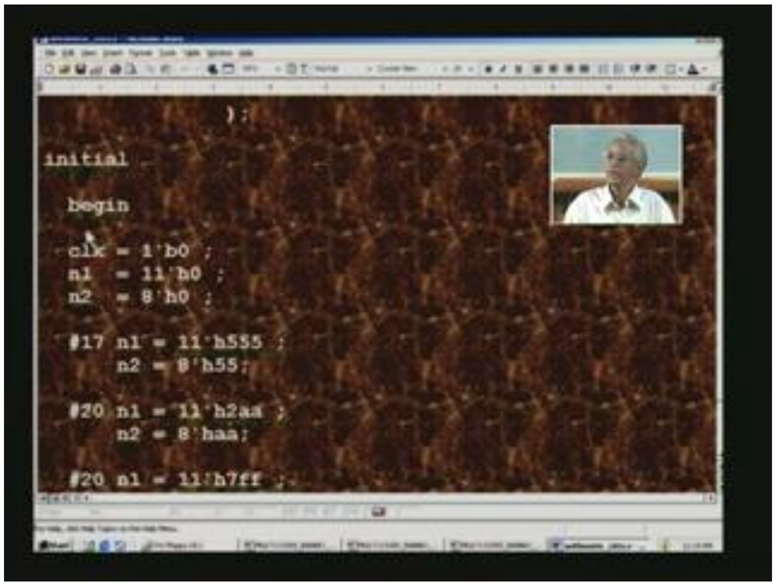
(Refer Slide Time: 36:22)



```
mult11sx8s    u1(

              .clk(clk),
              .n1(n1),
              .n2(n2),
              .result(result)

              );

initial

  begin

  clk = 1'b0 ;
  n1  = 11'h0 ;
  n2  = 8'h0 ;
```

This is the module name. It will be the same despite back annotation because we start only with this name with dot v extension for the design. We get a back annotated file from that as we have used here. We use the back annotation there because we want the final to get delays incorporated. We have listed all the ports by name.

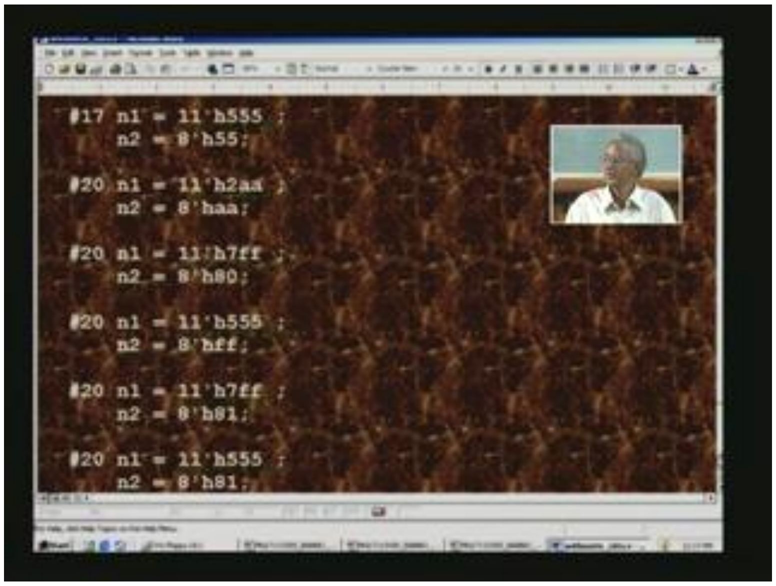
(Refer Slide Time: 37:00)



```
              );

initial

  begin

  clk = 1'b0 ;
  n1  = 11'h0 ;
  n2  = 8'h0 ;

  #17 n1 = 11'h555 ;
      n2 = 8'h55;

  #20 n1 = 11'h2aa ;
      n2 = 8'haa;

  #20 n1 = 11'h7ff ;
```

Once again, to process the actual test patterns, we need to apply different test patterns input. We started at 0 time. We have an initial, that is, begin and an end will be there. We stagger by a few nano seconds so that we apply the clock only after the data stabilizes which is why we need to put 17 instead of 20, so that the data is applied before the clock arrives at 20. That is the implication here. You would have noticed this all through the design. The very first inputs are clock; clock has to be initialized because we are going to run the clock and that also has to be initialized to 0. Later on, the clock will keep on toggling as a positive going pulse; n1 and n2 are the two numbers we have already seen, 11 and eight bits respectively. We are forcing them 0 to start with. Later on, every 20 nano seconds, with the exception of this, we change the data; in this case, triple 5, then 5, this corresponds to 0 1 0 1 and so on.

(Refer Slide Time: 37:58)

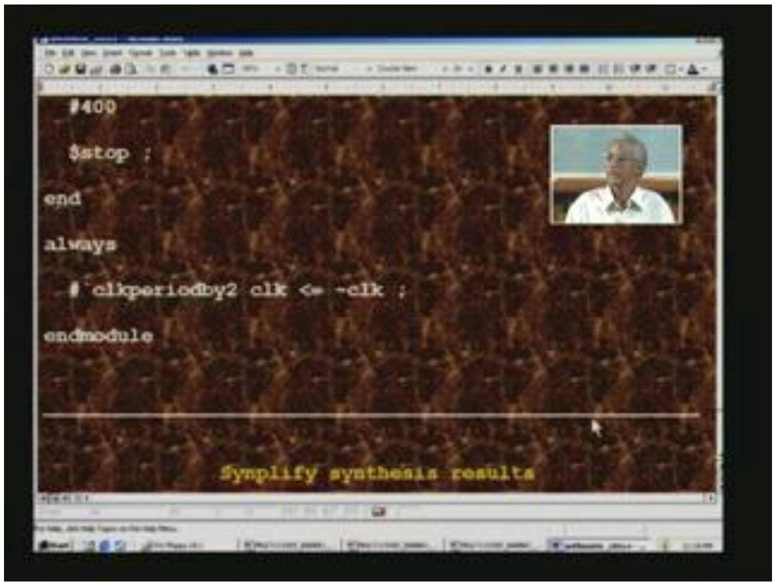


```
#17 n1 = 11'h555 ;
    n2 = 8'h55;

#20 n1 = 11'h2aa ;
    n2 = 8'haa;

#20 n1 = 11'h7ff ;
    n2 = 8'h80;

#20 n1 = 11'h555 ;
    n2 = 8'hff;

#20 n1 = 11'h7ff ;
    n2 = 8'h81;

#20 n1 = 11'h555 ;
    n2 = 8'h81;
```

Then we apply 2aa aa; there is no signtity for each of this number and they are all not an exhaustive test as we have seen in adder also. If you wish, you can do more exhaustive tests among the test benches. But what have been done is that all the n points and strategic values are checked by this combination. Similarly, there is another, all as here and then 7 f f 8 0 different combination; similarly all these

Once again I have noted all this down on paper. All these numbers are merely test patterns testing for different input values and last one is 0 and then 7f. There are hex decimal, I think the waveform gives both hex decimal and decimal, if I am right. Let us see what it has in store for us.

(Refer Slide Time: 38:45)

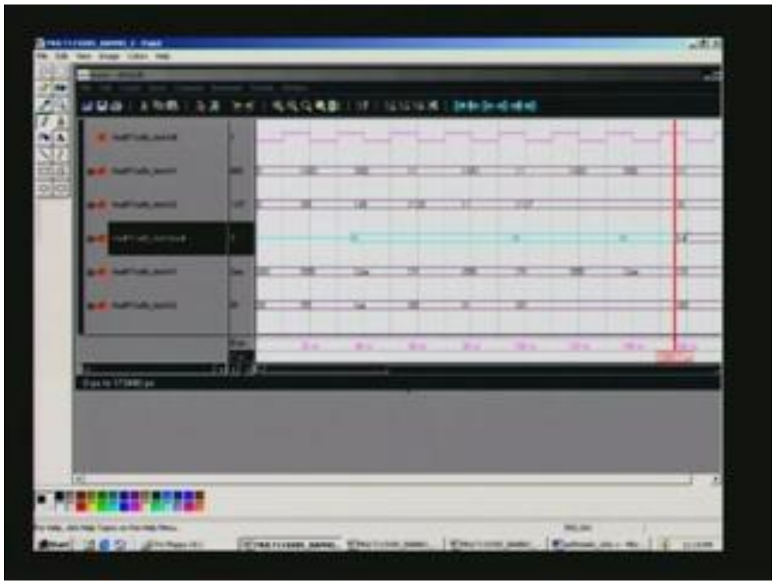


```
  #400
  $stop ;
end
always
  # `clkperiodby2 clk <= ~clk ;
endmodule
```

Symplify synthesis results

We give some more questioning for processing further say by 400nano seconds. This may not really be required. It can be 100 or even less. Finally stop. This completes the test bench with final statement of always; wherein as usual, you toggle the clock here. This is the end module; we have symplify results. We look at the waveform before we go to this result.

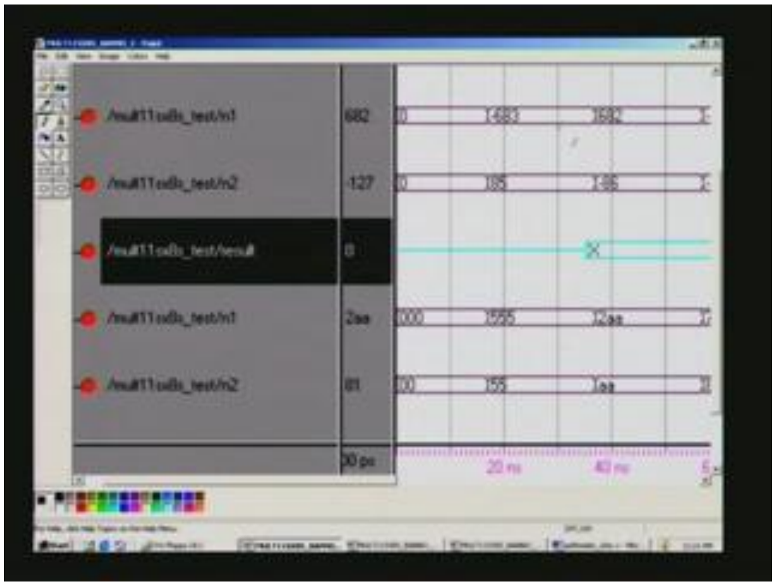
(Refer Slide Time: 39:33)



Before I zoom, I will just explain what this is. We have seen totally eight pipeline stages which are the clock 1 through 8. Naturally, the results will only manifest after the 8 clock pulses. Let us first have a look at whether it is shown. We have a waveform and 3, 6, 7, the eighth clock is here. Naturally, the final result must be delayed a bit because it is a back annotated file. In fact, there is a cursor mark for the first output here.

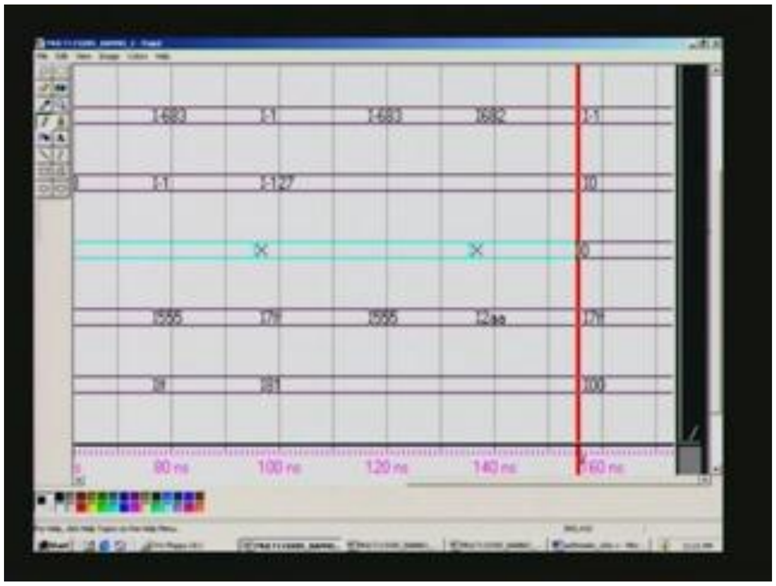
The very first output, as you see here, is 0. I will zoom in to it in a minute. What we have applied here is I will just read out different combinations of inputs. For example, as I mentioned before, I have n1 duplicated. In one case, it is in hex decimal and lower ones are hex decimal which you can easily compare with the test bench we have just seen. The corresponding decimal too can be had by using the format that we have seen earlier while looking into the Modalsim tool and using which we can have the very same waveform copied and then the format changed. That is why I have got it here. So this is nothing other than this is only in a different format. So it is very easy to change the format the way you like and it will be easy for us to deal with decimal numbers rather than hex decimal numbers. You can trust the simulation tool to do this conversion automatically. I have verified this to be correct. You too can verify independently if you wish. Let us see this. 0 into 0 must be the 0 and that is the result after the eighth pulse which we had already seen. So this result is correct. I have listed all this here. I will read this as well so that you can have a look. So, the second number is I will just read out from here and cross-check this paper. First is 0 into 0, next is minus 63 into 85, then 6682 into minus 86 and minus 1 into minus 12 8, then minus 683 into minus 1, then minus 1 into minus 127, reading from this paper you can cross check; then, minus 683 into minus 127, 682 into minus 127, then minus 1 into 0 and you will get some more in the subsequent waveform. The very first result is 0 and that is what 0 into 0 is which is correct.

(Refer Slide Time: 42:22)



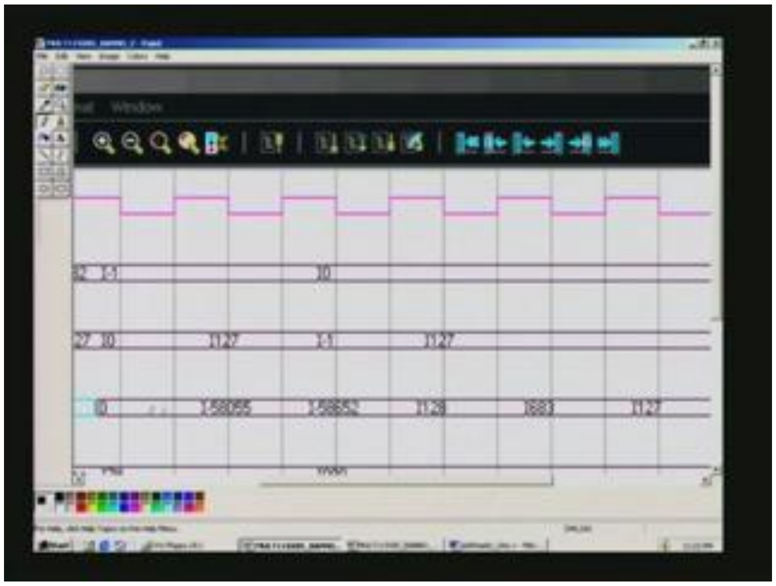
I will zoom this for you to see it clearly. So that is what we have seen here, 0 into 0 minus 683 into 285, 682 into minus 86 and so on. This is in hex decimal and this is in decimal.

(Refer Slide Time: 42:42)



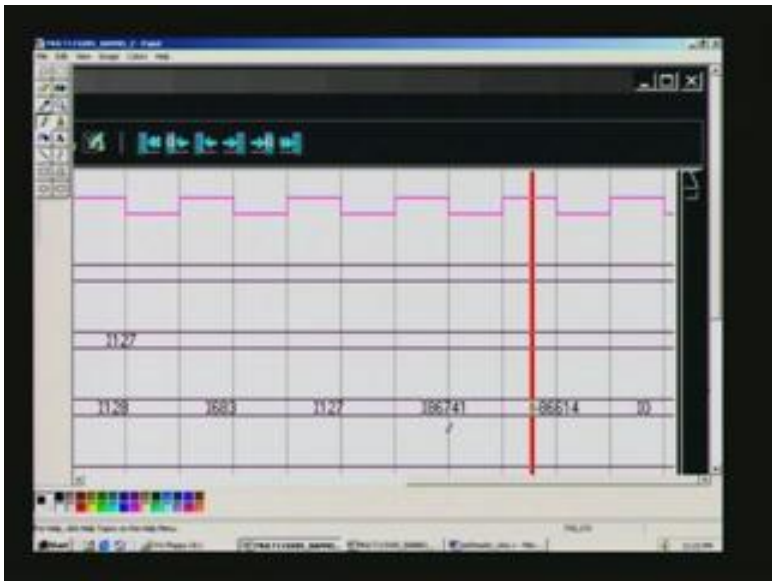
The final result is here and that is available only at eighth clock pulse. Each clock pulse is 50 megahertz and 20nano seconds so you get the result at around 160nano seconds. That is what the result is 0 here. If you go into the second waveform, I will zoom out straightaway. As far as the inputs are concerned what we have seen is minus 1 into 0 earlier that is what is here and to continue that minus 1 into 1 gives plus 127, then 0 into minus 127. I hope they are there. Yes it is there. The final results start with 0 that you are seeing and different results are here. We will cross check with the hand computed thing. First, I will zoom in to this one so that you can see. This is decimal result here. We will just zoom this.

(Refer Slide Time: 43:43)



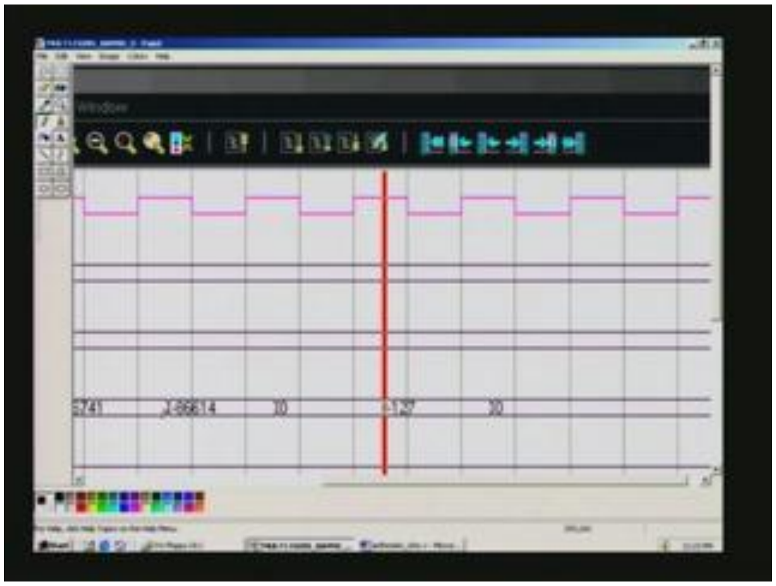
You can see here. First is 0result, then minus 58055, then minus 58652, then 128, 1683, 127, then 86741, minus 86614, then 0, then minus 127. It is probably not there in this.

(Refer Slide Time: 44:07)



We will see the next waveform. I will zoom out straight away. This is basically the same, 0, minus 27 you have seen.

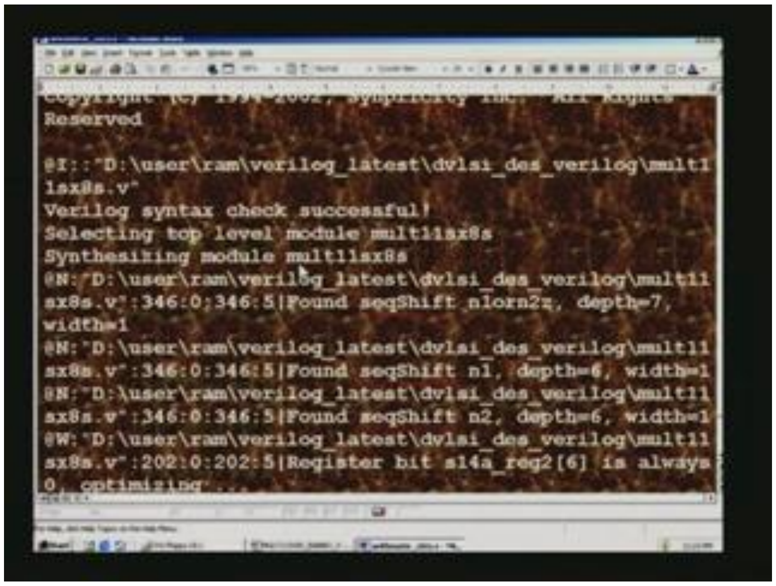
(Refer Slide Time: 44:20)



The final results for 0 into minus 1 must be 0. That is what is here. The next one is 0 into 127, which merges with the same 0 so it continues to be 0. This is the final result we had and the result was delayed by eight clock pulses because of eight levels of pipelining.

This establishes the working of the algorithm. We will just have a look at the symplify results, what it has to say for this multiplier.

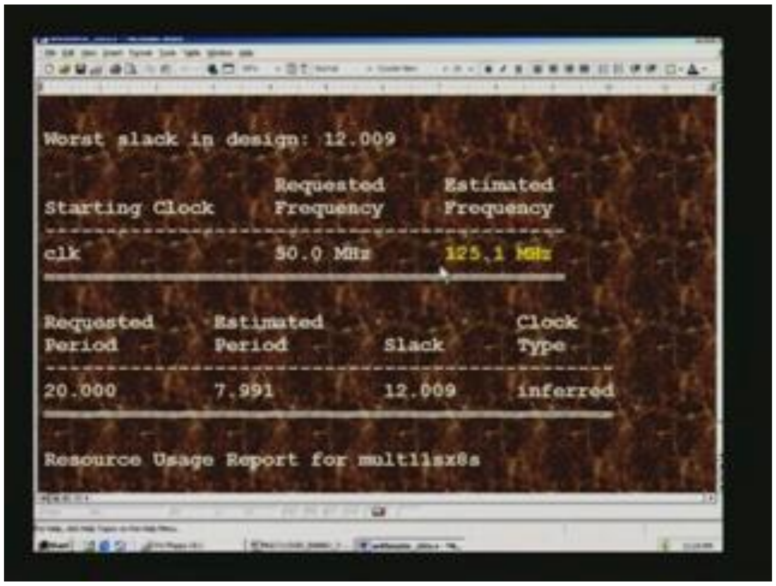
(Refer Slide Time: 44:50)



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

@I::"D:\user\ram\verilog_latest\dvlsi_des_verilog\mult1
1sx8s.v"
Verilog syntax check successful!
Selecting top level module mult11sx8s
Synthesizing module mult11sx8s
@N:"D:\user\ram\verilog_latest\dvlsi_des_verilog\mult11
sx8s.v":346:0:346:5|Found seqShift nlorn2z, depth=7,
width=1
@N:"D:\user\ram\verilog_latest\dvlsi_des_verilog\mult11
sx8s.v":346:0:346:5|Found seqShift n1, depth=6, width=1
@N:"D:\user\ram\verilog_latest\dvlsi_des_verilog\mult11
sx8s.v":346:0:346:5|Found seqShift n2, depth=6, width=1
@W:"D:\user\ram\verilog_latest\dvlsi_des_verilog\mult11
sx8s.v":202:0:202:5|Register bit s14a_reg2[6] is always
0, optimizing ...
```

This design is mult11s into 8s.

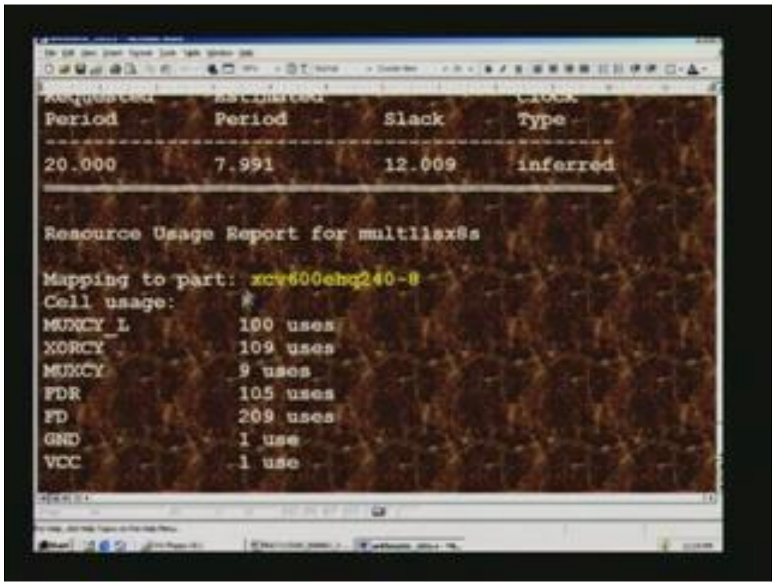
(Refer Slide Time: 45:00)



```
Worst slack in design: 12.009

                   Requested     Estimated
Starting Clock     Frequency     Frequency
-------------------------------------------
clk                50.0 MHz      125.1 MHz
===========================================

Requested    Estimated                 Clock
Period       Period         Slack      Type
---------------------------------------------
20.000       7.991          12.009     inferred
=============================================

Resource Usage Report for mult11sx8s
```

It works at 125 megahertz here and this is the symplify result.

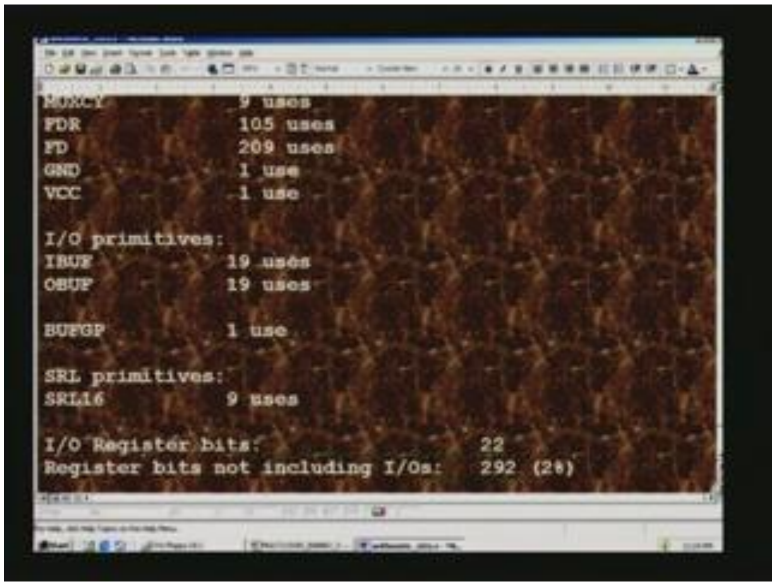
(Refer Slide Time: 45:03)



```
Requested       Estimated       Clock
Period          Period          Slack       Type
----------------------------------------------------
20.000          7.991           12.009      inferred
====================================================

Resource Usage Report for mult11sx8s

Mapping to part: xcv600ehq240-8
Cell usage:
MUXCY_L           100 uses
XORCY             109 uses
MUXCY             9 uses
FDR               105 uses
FD                209 uses
GND               1 use
VCC               1 use
```

We have selected the same device as what we are going to use for DCTQ later on.

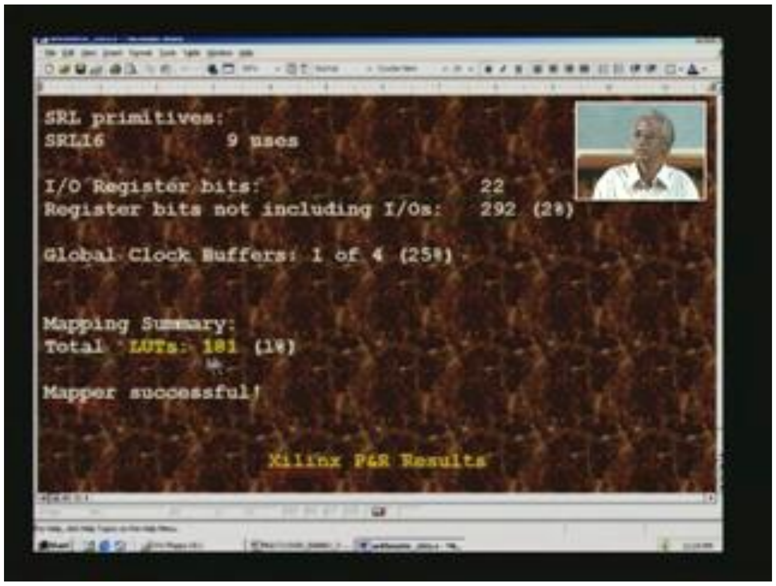
(Refer Slide Time: 45:07)



```
MUXCY             9 uses
FDR               105 uses
FD                209 uses
GND               1 use
VCC               1 use

I/O primitives:
IBUF             19 uses
OBUF             19 uses

BUFGP            1 use

SRL primitives:
SRL16            9 uses

I/O Register bits:                  22
Register bits not including I/Os:   292 (2%)
```

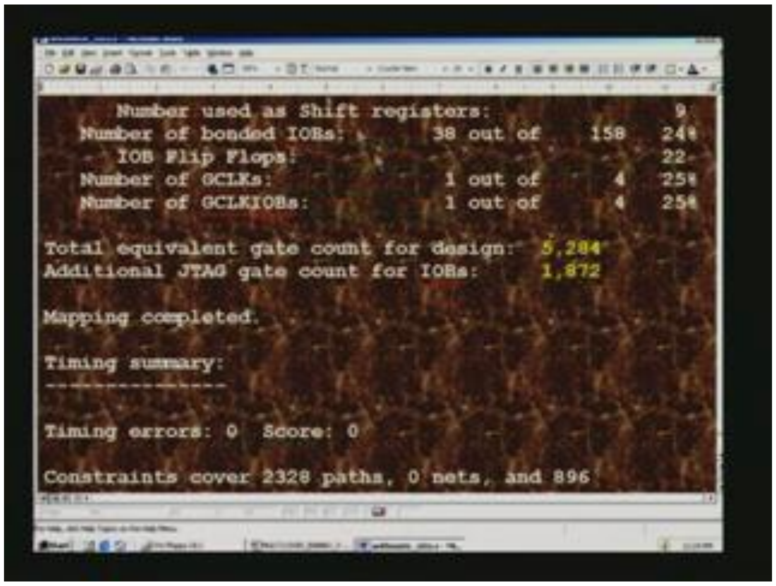
This gives all the primitives cell usage.

(Refer Slide Time: 45:11)


```
SRL primitives:
SRL16          9 uses

I/O Register bits:                     22
Register bits not including I/Os:     292  (2%)

Global Clock Buffers: 1 of 4  (25%)


Mapping Summary:
Total  LUTs: 181  (1%)

Mapper successful!

              Xilinx P&R Results
```

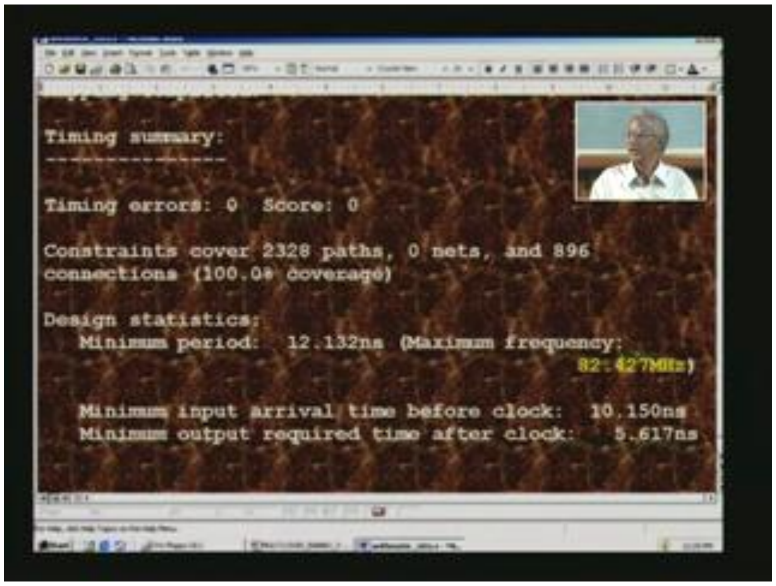
Finally, it gives number of LUTs for this design rather, high for a multiplier and it is 1 percent. It is more than the full adder that we had designed earlier. Of course, the designs are totally different. You cannot make a comparison.

(Refer Slide Time: 45:31)


```
      Number used as Shift registers:                 9
   Number of bonded IOBs:        38 out of    158    24%
      IOB Flip Flops:                                22
   Number of GCLKs:               1 out of      4    25%
   Number of GCLKIOBs:            1 out of      4    25%

Total equivalent gate count for design:  5,284
Additional JTAG gate count for IOBs:     1,872

Mapping completed.

Timing summary:
---------------

Timing errors: 0  Score: 0

Constraints cover 2328 paths, 0 nets, and 896
```

Xilinx place and route results are here. You see that slice etc., reported there. What is of interest is the number of gates. In the full adder, we had some 8000 gates and we have just 7000 here, although LUTs are different; the gate count is what determines the chip area.

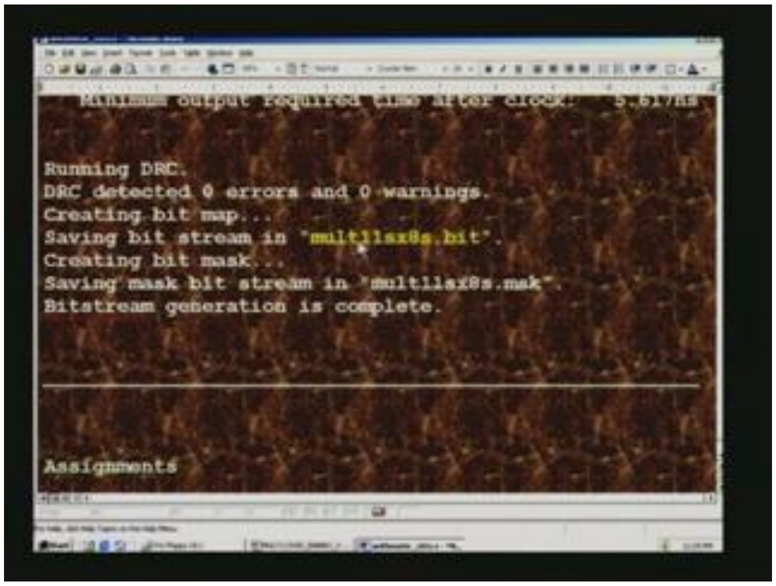
(Refer Slide Time: 45:54)



```
Timing summary:
---------------

Timing errors: 0  Score: 0

Constraints cover 2328 paths, 0 nets, and 896
connections (100.0% coverage)

Design statistics:
   Minimum period:  12.132ns (Maximum frequency:
                                        82.427MHz)

   Minimum input arrival time before clock:  10.150ns
   Minimum output required time after clock:   5.617ns
```

You can go through the same exercise with other algorithms that you may be learning such as Waugh algorithm, then Booths algorithm, there are so many algorithms. You can take some of those algorithms and make a comparison with this new algorithm that has been done by the present designer.

(Refer Slide Time: 46:16)

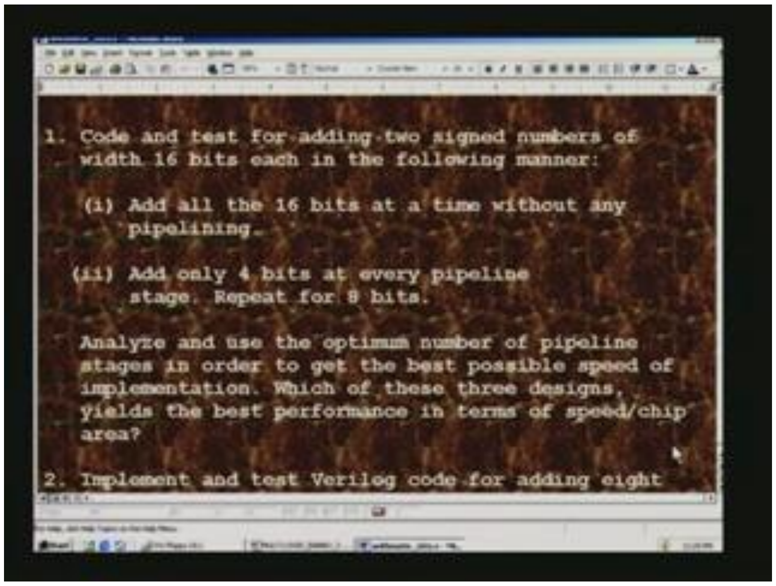


```
   Minimum output required time after clock:    5.617ns

Running DRC.
DRC detected 0 errors and 0 warnings.
Creating bit map...
Saving bit stream in "mult11sx8s.bit".
Creating bit mask...
Saving mask bit stream in "mult11sx8s.mak".
Bitstream generation is complete.
```

Assignments

Unfortunately, the frequency falls after the Xilinx place and route but it is still quite high, 82. It also creates a bit stream as output for you to load into the actual fpga in order to test on the hardware or cell as an ipcore if you happen to be ipcore designer.

(Refer Slide Time: 46:36)



1. Code and test for adding two signed numbers of width 16 bits each in the following manner:

   (i) Add all the 16 bits at a time without any pipelining.

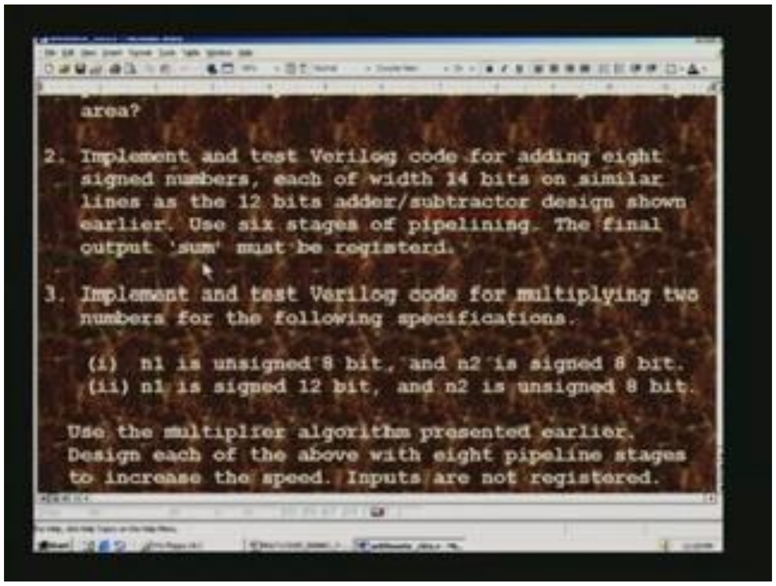
   (ii) Add only 4 bits at every pipeline stage. Repeat for 8 bits.

   Analyze and use the optimum number of pipeline stages in order to get the best possible speed of implementation. Which of these three designs, yields the best performance in terms of speed/chip area?

2. Implement and test Verilog code for adding eight

Before we wind up this arithmetic circuit there is an assignment for you. I will just read it out and explain. Let us see what the assignment says. Code and test for adding two signed numbers of width sixteen bits each. We have earlier seen twelve bits. Now I want it worked out for sixteen bits and not eight numbers but just two numbers this time in the following manner. You have to do it in this fashion: Add all the sixteen bits at a time without any pipelining. Earlier we have done pipelining as far as the data bus width is concerned by taking the LSB and then pipelining functionality as well. I want it done without any pipelining; just add straightaway and merely register it if you wish. Find out how much delay it can give. This is only for two numbers. If you wish, you can do this same thing for eight numbers and compare this result with the adder that we have already done. This is sixteen bits at a time. Do this same thing for two numbers addition using 4 bits at a time as well as eight bits at a time. In fact there are three different problems and this is another problem. This is another problem. You have to solve them independently. Add pipeline stage. So before doing that one, first analyze and use the optimum number of pipeline stages in order to get the best possible speed of implementation. So from the speed point of view, we are doing this partitioning of data bus, say, four bits or eight bits or sixteen bits. You repeat all this so that you will know where you stand as far as the speed performance is concerned as well as the chip area is concerned. Which of the three designs yields the best performance in terms of speed and chip area? That is what I have explained just now.

(Refer Slide Time: 48:37)



area?

2. Implement and test Verilog code for adding eight signed numbers, each of width 14 bits on similar lines as the 12 bits adder/subtractor design shown earlier. Use six stages of pipelining. The final output 'sum' must be registerd.

3. Implement and test Verilog code for multiplying two numbers for the following specifications.

   (i) n1 is unsigned 8 bit, and n2 is signed 8 bit.
   (ii) n1 is signed 12 bit, and n2 is unsigned 8 bit.

Use the multiplier algorithm presented earlier. Design each of the above with eight pipeline stages to increase the speed. Inputs are not registered.

We have a second assignment. In fact, the first assignment is actually three for you. The second one is to implement test verilog code for adding eight signed numbers and we have done this before, each of width fourteen bits instead of twelve bits that we had added earlier. Evaluate this on similar lines because we will use this particular exercise in our DCTQ algorithm which I will not explain. I am depending on you to work out for fourteen bits. This is exactly the same thing as what we have designed earlier for twelve bits. That is what it implies. Each of width fourteen bits on similar lines as the twelve bits adder subtractor design shown earlier; use six stages of pipelining. In this case, the only difference is that we earlier had in the twelve bits addition only five stages of pipelining. The final result, the sum, came only as an assign statement and we did not register. In this case, the final sum will have to be registered. This means that after fifth one you have to add one more clock after the assign statement for the sum. So you add one more register for that pipelining. Add that as a sixth register and that completes this sum. So this is the one that we will use.

Similarly, we will have a multiplier and that is this exercise, which once again is mandatory because we will use these multipliers in DCTQ register which I will not explain. So the problem is exactly like the multiplier that we have considered in this lecture. Implement and test verilog code for multiplying two numbers for the following specifications: n1 is unsigned eight bits and n2 is signed eight bits, and n1 is signed twelve bits and n2 is unsigned eight bits. The difference is that this is a separate multiplier from this so you need to do two independent multipliers: one is eight bits unsigned and the other number is signed eight bits. This is an eight into eight multipliers; one is signed and other is unsigned and which is signed is also mentioned here. Similarly, we need twelve bits into eight bits. What we have seen in this lecture is 11 bits into eight bits. Here it is different and this is also different. We need both of these for our video compression algorithm. That is a DCTQ algorithm which we will see later on. Be prepared when we come for that by solving all these assignments, especially this and the 14 bits 8 numbers adder; those two are the ones that we will use. Use the multiplier algorithm presented earlier; design each of the above with eight pipeline stages to increase the speed; inputs are not registered as usual as in the case we have seen. You see that there are once again 8 pipelines register and take care of this. This is exactly the same as the multiplier presented in this lecture. We have completed arithmetic. We will see design applications later on. Thank you.