

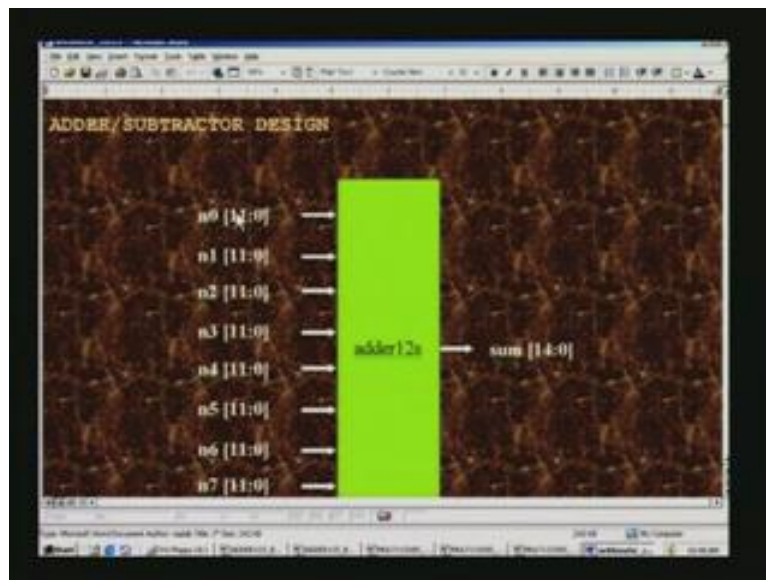
**Digital VLSI System Design**  
**Prof Dr. S. Ramachandran**  
**Department of Electrical Engineering**  
**Indian Institute of Technology, Madras**

**Lecture - 41**

**Design of Arithmetic Circuits (continued)**

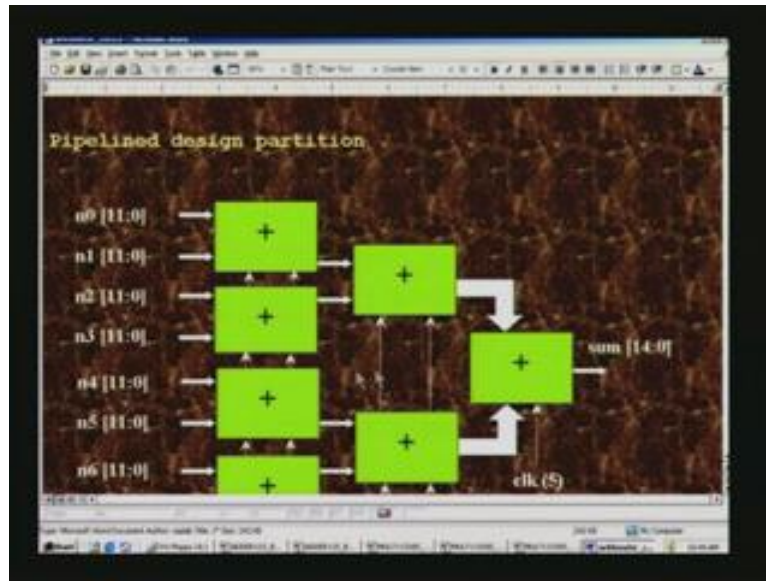
We are looking at the design for adder or subtractor design in which we had added eight numbers,  $n_0$  through  $n_7$ .

(Refer Slide Time: 02:40)



We got the sum which is three bits more than each of these bits. The last bit is a sign bit as is also the case with the sum.

(Refer Slide Time: 02:53)



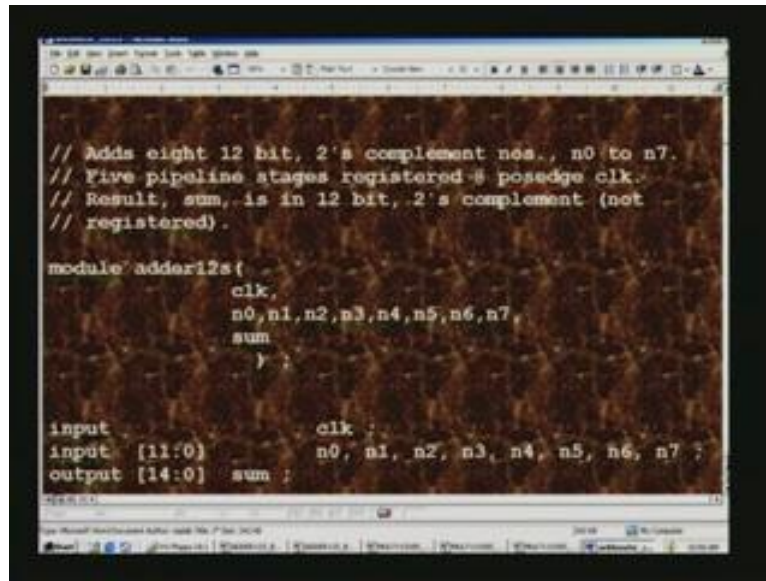
We had seen that the whole thing was a pipeline and a partition for data width as well as the functionality. In this example, the functionality happens to be same add or subtract. When we take up bigger applications such as DCTQ, we will appreciate the functionality difference coming in. We had three stages of pipelining here with the five internal registers used as pipelined registers. Note that the last sum is not registered.

(Refer Slide Time: 03:30)



We had seen how to take a two's complement of signed number and these were the examples we had considered earlier.

(Refer Slide Time: 03:42)



```
// Adds eight 12 bit, 2's complement nos., n0 to n7.
// Five pipeline stages registered @ posedge clk.
// Result, sum, is in 12 bit, 2's complement (not
// registered).

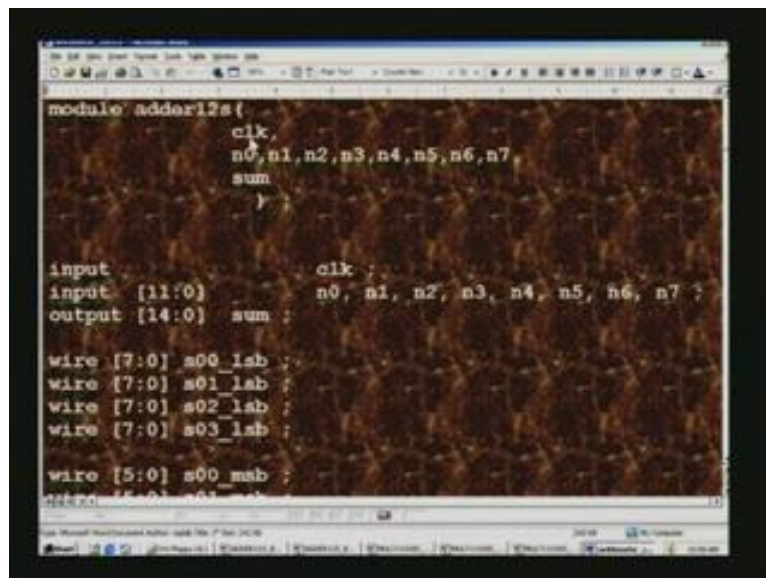
module adder12s(
    clk,
    n0,n1,n2,n3,n4,n5,n6,n7,
    sum
);

input          clk;
input [11:0]   n0, n1, n2, n3, n4, n5, n6, n7;
output [14:0]  sum;

endmodule
```

Now, let us have a look at the verilog code for this adder subtractor. I will add eight 12 bit, two's complement numbers, n0 to n7. There are five pipeline stages registered at positive edge clock. The resultant sum is in 12 bit, two's complement not registered. There is a mistake here. The sum must be fifteen bits.

(Refer Slide Time: 04:12)



```
module adder12s(
    clk,
    n0,n1,n2,n3,n4,n5,n6,n7,
    sum
);

input          clk;
input [11:0]   n0, n1, n2, n3, n4, n5, n6, n7;
output [14:0]  sum;

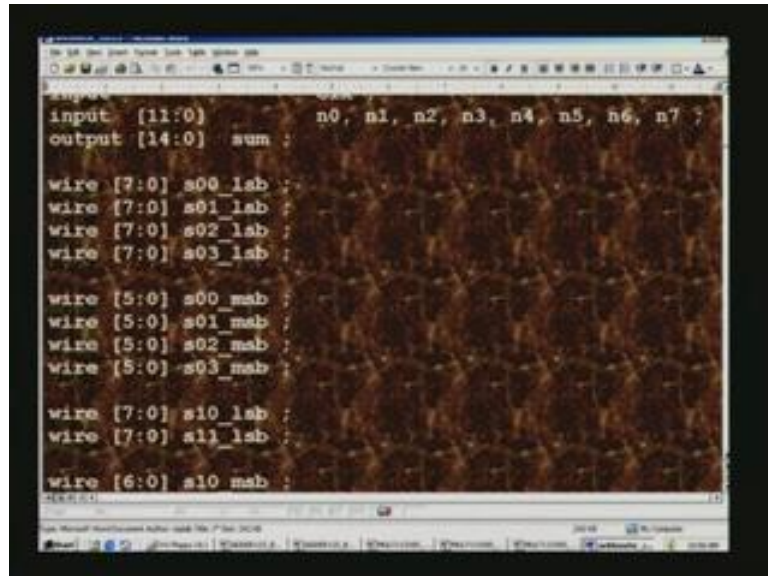
wire [7:0]  s00_lsb;
wire [7:0]  s01_lsb;
wire [7:0]  s02_lsb;
wire [7:0]  s03_lsb;

wire [5:0]  s00_msb;
wire [5:0]  s01_msb;
```

We will first have to declare the module. This is the module name we are giving. If you wish you can designate as 8 into 12s as there are eight inputs; clock is one of the inputs and n0 through n7 are the eight numbers. So what we have here is that each of them is twelve bits

and that is why we declare this and clock is also declared. So also is the case with output in which the sum is of width fifteen bits. That is the output listed there.

(Refer Slide Time: 04:53)



```
input [11:0] n0, n1, n2, n3, n4, n5, n6, n7 ;
output [14:0] sum ;

wire [7:0] s00_lsb ;
wire [7:0] s01_lsb ;
wire [7:0] s02_lsb ;
wire [7:0] s03_lsb ;

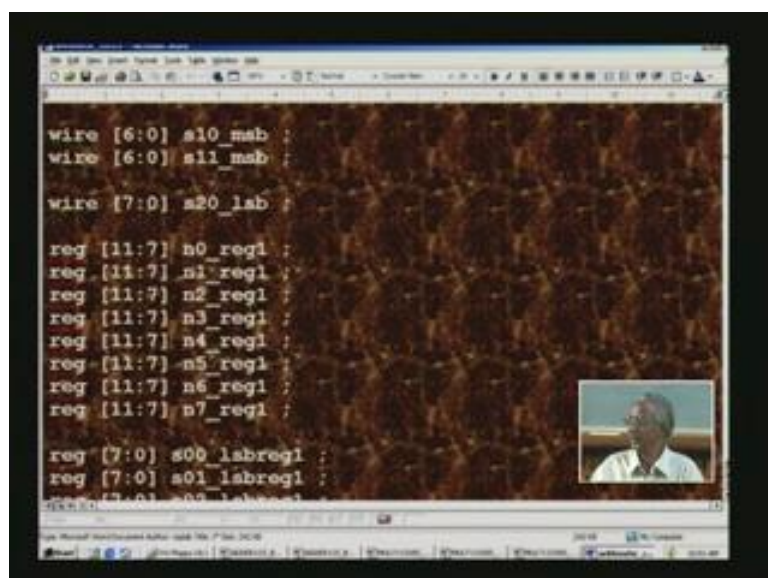
wire [5:0] s00_msb ;
wire [5:0] s01_msb ;
wire [5:0] s02_msb ;
wire [5:0] s03_msb ;

wire [7:0] s10_lsb ;
wire [7:0] s11_lsb ;

wire [6:0] s10_msb ;
```

During the course of the other statements we will actually work out the arithmetic portion. We will encounter many intermediate signals. Some of them may be with assigned statements and therefore, they are declared as wire. The width of each is also declared here. For example, we will be adding lsb as we had mentioned in the pipelined partition earlier. We will also be adding msb with the carry of the lsb addition and so on. They all have to be declared here.

(Refer Slide Time: 05:30)



```
wire [6:0] s10_msb ;
wire [6:0] s11_msb ;

wire [7:0] s20_lsb ;

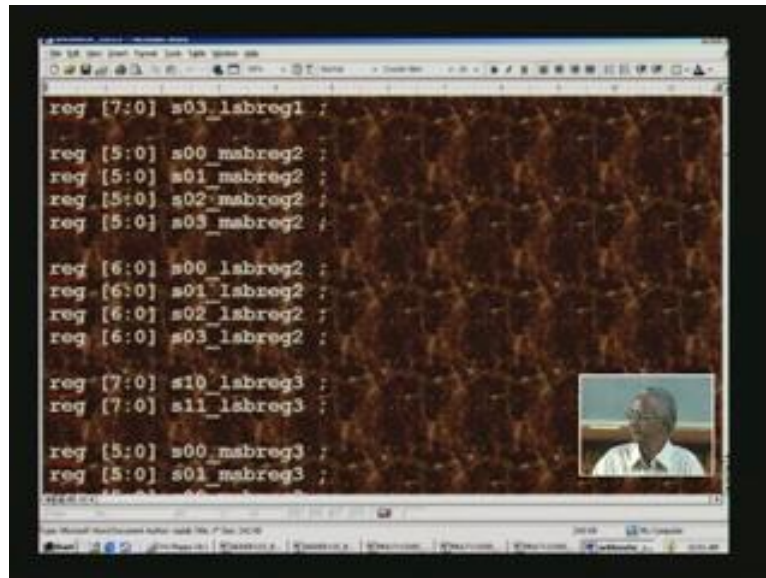
reg [11:7] n0_req1 ;
reg [11:7] n1_req1 ;
reg [11:7] n2_req1 ;
reg [11:7] n3_req1 ;
reg [11:7] n4_req1 ;
reg [11:7] n5_req1 ;
reg [11:7] n6_req1 ;
reg [11:7] n7_req1 ;

reg [7:0] s00_labreg1 ;
reg [7:0] s01_labreg1 ;
reg [7:0] s02_labreg1 ;
```



There are also numbers n0 through n7 that will have to be propagated which is not computed at a particular stage. For instance, msb has not been computed therefore they have to be registered and propagated through the pipe line in order to process when the time is right for them. There are other similar next stage msbs and lsbs listed here and they are declared as reg and so is the case here.

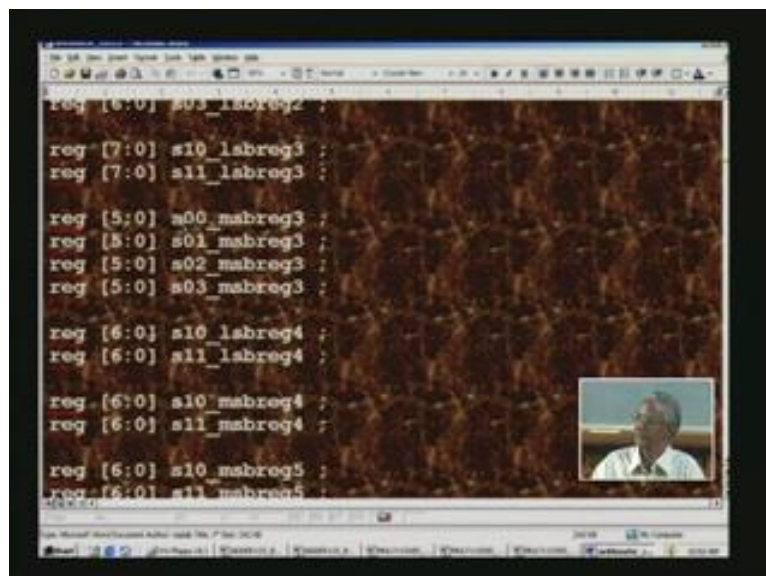
(Refer Slide Time: 05:58)



```
reg [7:0] s03_labreg1 ;  
reg [5:0] s00_mabreg2 ;  
reg [5:0] s01_mabreg2 ;  
reg [5:0] s02_mabreg2 ;  
reg [5:0] s03_mabreg2 ;  
  
reg [6:0] s00_labreg2 ;  
reg [6:0] s01_labreg2 ;  
reg [6:0] s02_labreg2 ;  
reg [6:0] s03_labreg2 ;  
  
reg [7:0] s10_labreg3 ;  
reg [7:0] s11_labreg3 ;  
  
reg [5:0] s00_mabreg3 ;  
reg [5:0] s01_mabreg3 ;
```

This is the second stage. 1 stands for the second stage and 0 for the first stage.

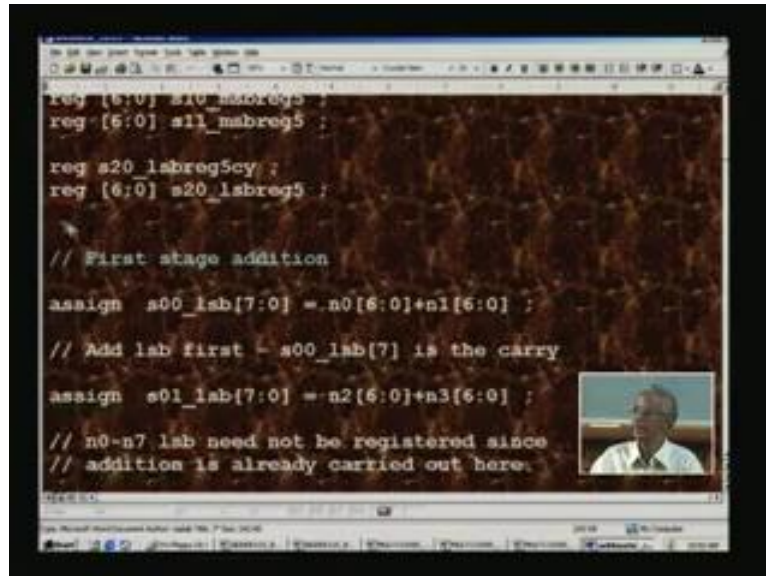
(Refer Slide Time: 06:05)



```
reg [6:0] s03_labreg2 ;  
  
reg [7:0] s10_labreg3 ;  
reg [7:0] s11_labreg3 ;  
  
reg [5:0] s00_mabreg3 ;  
reg [5:0] s01_mabreg3 ;  
reg [5:0] s02_mabreg3 ;  
reg [5:0] s03_mabreg3 ;  
  
reg [6:0] s10_labreg4 ;  
reg [6:0] s11_labreg4 ;  
  
reg [6:0] s10_mabreg4 ;  
reg [6:0] s11_mabreg4 ;  
  
reg [6:0] s10_mabreg5 ;  
reg [6:0] s11_mabreg5 ;
```

Once again you can see 1 0 is second stage of msb registers. This completes the reg wire declaration.

(Refer Slide Time: 06:20)



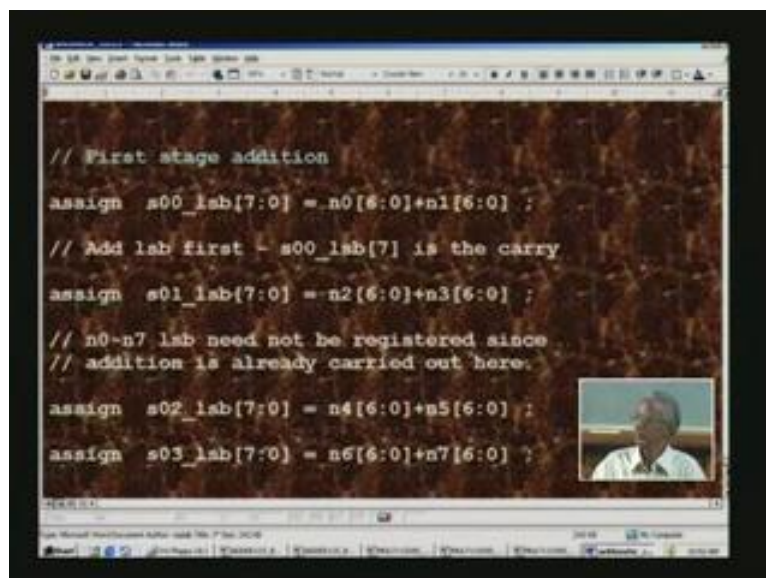
```
reg [6:0] s10_mabreg5 ;
reg [6:0] s11_mabreg5 ;

reg s20_lsbreg5cy ;
reg [6:0] s20_lsbreg5 ;

// First stage addition
assign s00_lsb[7:0] = n0[6:0]+n1[6:0] ;
// Add lab first - s00_lsb[7] is the carry
assign s01_lsb[7:0] = n2[6:0]+n3[6:0] ;
// n0-n7 lsb need not be registered since
// addition is already carried out here.
```

We also need some carries. That is for we are doing. The final result will be here. Actually, the ultimate result will be in assigned statements. Therefore, that is not listed here but it had appeared elsewhere.

(Refer Slide Time: 06:36)

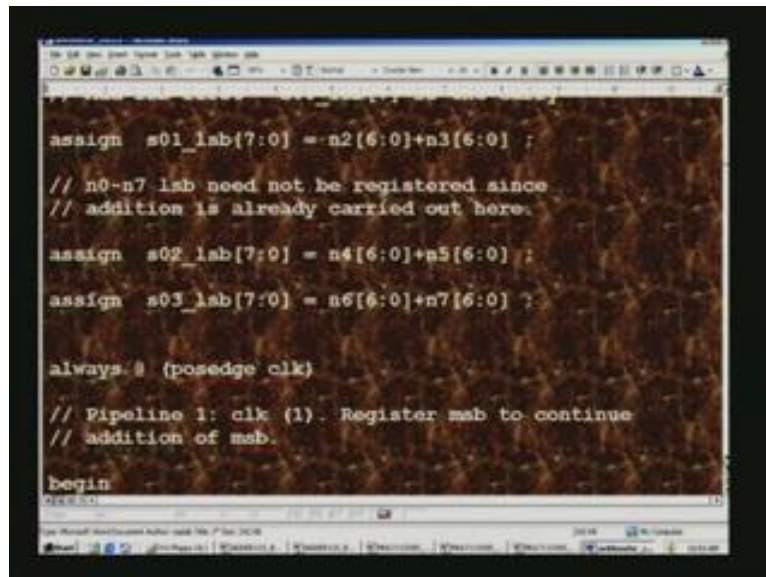


```
// First stage addition
assign s00_lsb[7:0] = n0[6:0]+n1[6:0] ;
// Add lab first - s00_lsb[7] is the carry
assign s01_lsb[7:0] = n2[6:0]+n3[6:0] ;
// n0-n7 lsb need not be registered since
// addition is already carried out here.
assign s02_lsb[7:0] = n4[6:0]+n5[6:0] ;
assign s03_lsb[7:0] = n6[6:0]+n7[6:0] ;
```

The first statement is as we mentioned before, in the first stage, we add two numbers at a time  $n_0, n_1$  and we add only the lsbs of this. For example, seven bits are only added and in parallel

that is concurrently, we add other numbers as well n2 and n3 and using this assign statement and the result is put in this, assign statement is declared as wire. So this is nothing other than a combinational output. Similarly, we have s01, s02, s03 arising from adding other numbers so n2, n3, here n4, n5, and n6, n7.

(Refer Slide Time: 07:25)



```
assign s01_lab[7:0] = n2[6:0]+n3[6:0] ;

// n0-n7 lab need not be registered since
// addition is already carried out here.

assign s02_lab[7:0] = n4[6:0]+n5[6:0] ;

assign s03_lab[7:0] = n6[6:0]+n7[6:0] ;

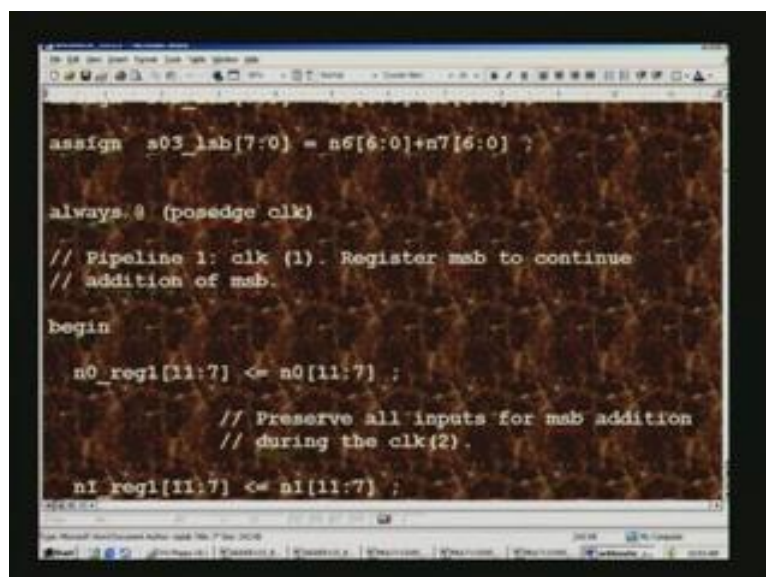
always @ (posedge clk)

// Pipeline 1: clk (1). Register mab to continue
// addition of mab.

begin
```

We add using 4 adders concurrently so that this is at the first stage. In this, we add just the lsb and we latch or rather register when the first clock arrives at the positive edge.

(Refer Slide Time: 07:31)



```
assign s03_lab[7:0] = n6[6:0]+n7[6:0] ;

always @ (posedge clk)

// Pipeline 1: clk (1). Register mab to continue
// addition of mab.

begin

    n0_reg1[11:7] <= n0[11:7] ;

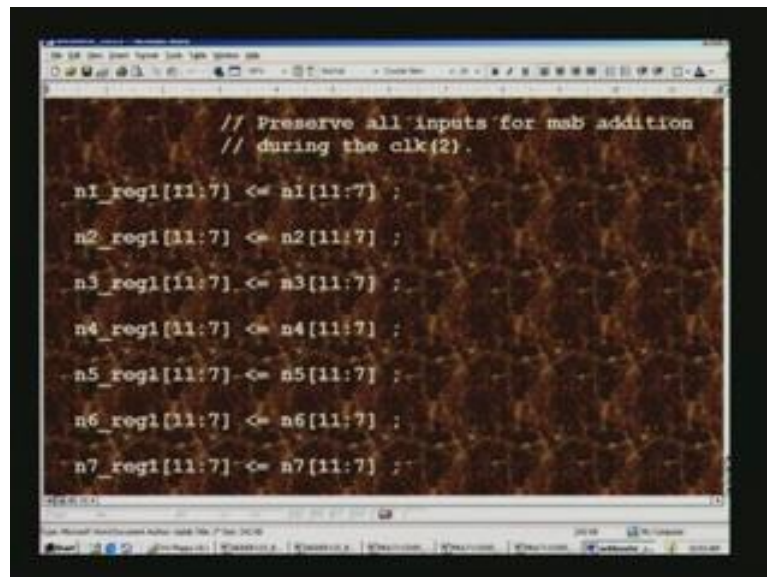
    // Preserve all inputs for mab addition
    // during the clk(2).

    n1_reg1[11:7] <= n1[11:7] ;
```



This is the pipeline 1 clock 1 register msb to continue addition of msb. Since we are not adding the msb, we have to register it separately. That is what we are doing here. This is the msb for all the numbers. Then propagate it for use in the next when the next clock arrives.

(Refer Slide Time: 07:51)

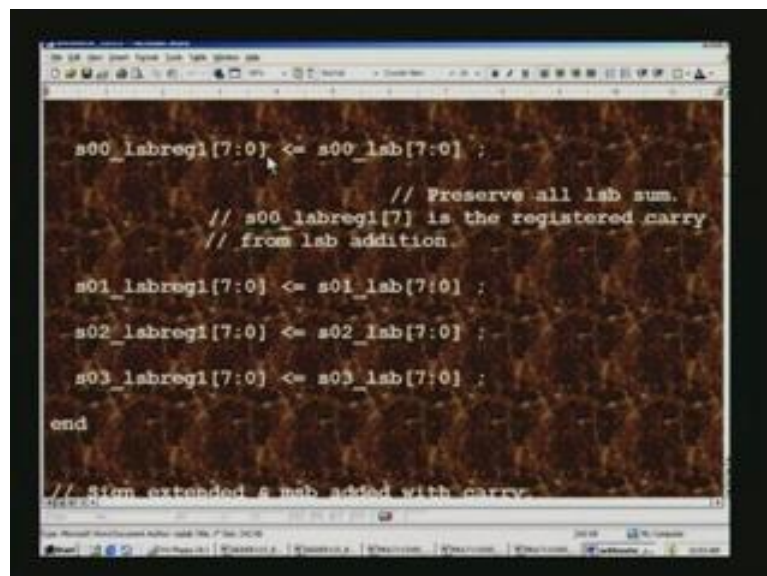


```
// Preserve all inputs for mab addition
// during the clk(2).

n1_reg1[11:7] <= n1[11:7] ;
n2_reg1[11:7] <= n2[11:7] ;
n3_reg1[11:7] <= n3[11:7] ;
n4_reg1[11:7] <= n4[11:7] ;
n5_reg1[11:7] <= n5[11:7] ;
n6_reg1[11:7] <= n6[11:7] ;
n7_reg1[11:7] <= n7[11:7] ;
```

We will do this before the next clock arrives.

(Refer Slide Time: 07:57)



```
s00_labreg1[7:0] <= s00_lab[7:0] ;

// Preserve all lsb sum.
// s00_labreg1[7] is the registered carry
// from lab addition.

s01_labreg1[7:0] <= s01_lab[7:0] ;
s02_labreg1[7:0] <= s02_lab[7:0] ;
s03_labreg1[7:0] <= s03_lab[7:0] ;

end

// Sign extended & mab added with carry
```

We also preserve the sum. This is the lsb sum and that has also been preserved here. So we had four results by adding 8 numbers, that is why 00 through s03 will be the lsb results.



(Refer Slide Time: 08:18)

```
end

// Sign extended & msb added with carry.

assign s00_mab[5:0] = {n0_reg1[11], n0_reg1[11:7]}+
                    {n1_reg1[11],
                     n1_reg1[11:7]}+s00_lsbreg1[7];

//s00_mab[6] is ignored

assign s01_mab[5:0] = {n2_reg1[11], n2_reg1[11:7]}+
                    {n3_reg1[11],
                     n3_reg1[11:7]}+s01_lsbreg1[7];

assign s02_mab[5:0] = {n4_reg1[11], n4_reg1[11:7]}+
                    {n5_reg1[11],
                     n5_reg1[11:7]}+s02_lsbreg1[7];
```

Before we add the msb, we should not forget to extend this sign before we add the msb. This extension of sign is nearly a duplication of msb. This is the msb 11 through 7 and this is the sign bit. We just have to duplicate here, and the whole thing is concatenation, that is, to say first we write the duplicated sign bit here of this number called n0 reg 1 that is registered at clock 1 and the whole thing put together is a sign extension of the msb. We add the second number to this and follow this same standard style with that too. For example, this is the signed extended bit and this is the actual msb bit of n1 and we have to add to this, we should not forget to add the carry resulting from the lsb addition which is available in this. This was the last bit and final result of lsb addition and that should not be forgotten here, that is, to make total addition; same is the case for other pairs of numbers.

(Refer Slide Time: 9:25)

```
        n3_reg1[11],
        n3_reg1[11:7])+s01_labreg1[7];
assign s02_mab[5:0] = {n4_reg1[11], n4_reg1[11:7]}+
                    {n5_reg1[11],
                    n5_reg1[11:7]}+s02_labreg1[7];
assign s03_mab[5:0] = {n6_reg1[11], n6_reg1[11:7]}+
                    {n7_reg1[11],
                    n7_reg1[11:7]}+s03_labreg1[7];

always @ (posedge clk)
// Pipeline 2: clk (2). Register mab to continue
// addition of mab.
begin
```

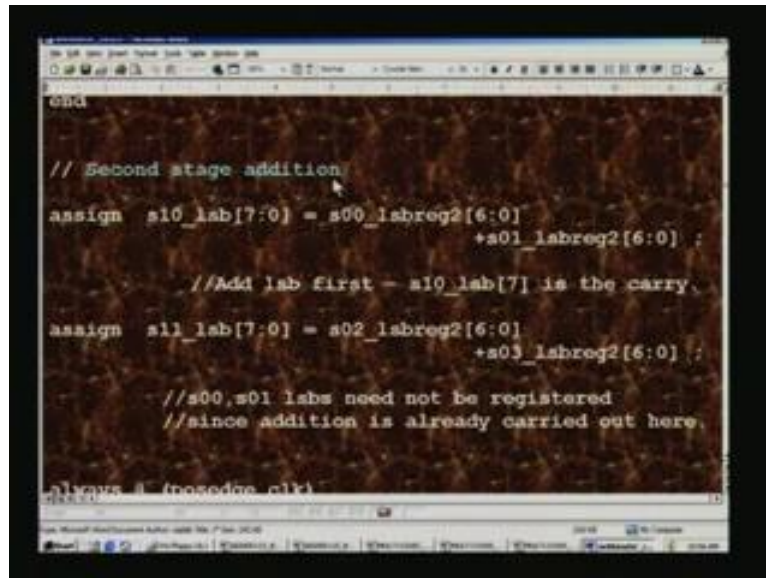
For example, we have n2, n3, then n4, n5, then n6 and n7. So, in this case, s00, s01 through s03 are all assign statements. This is pre-computed because being a time consuming thing, you want to speed up the system. Therefore, we pre-compute and only with the following clock edge, we assign these values to another set of registers. We preserve the msb sum that we have computed earlier and we use this same nomenclature in another register. This is clock 2 so we just add reg 2 to discriminate from this one. This will have to be used in the subsequent stages.

(Refer Slide Time: 10:15)

```
s01_mabreg2[5:0] <= s01_mab[5:0] ;
s02_mabreg2[5:0] <= s02_mab[5:0] ;
s03_mabreg2[5:0] <= s03_mab[5:0] ;
s00_labreg2[6:0] <= s00_labreg1[6:0] ;
// Preserve all lab sum.
s01_labreg2[6:0] <= s01_labreg1[6:0] ;
s02_labreg2[6:0] <= s02_labreg1[6:0] ;
s03_labreg2[6:0] <= s03_labreg1[6:0] ;
end
```

Here, we should also preserve the lsb sum because we have not yet made the total result of the first stage. This completes the very first stage of addition that we have seen in the partitioned pipelining approach in the diagram we have seen earlier.

(Refer Slide Time: 10:33)



```
end

// Second stage addition
assign a10_lab[7:0] = s00_lsbreg2[6:0]
                    +s01_lsbreg2[6:0] ;
                    //Add lab first - a10_lab[7] is the carry
assign a11_lab[7:0] = s02_lsbreg2[6:0]
                    +s03_lsbreg2[6:0] ;
                    //s00,s01 labs need not be registered
                    //since addition is already carried out here

always # (posedge clk)
```

In the second stage, we have four results that have come from the first stage s00 through s03, and that we will add here. Once again we adopt the same strategy that we have done earlier. We take the whole number and bifurcate into roughly equal numbers and call them lsb and msb and first add just the lsb. This is what we do here. The lsb is added or the first stage s00 to the s01 that was the second output of the first stage. So is the case with s02 and s03, which are the other two outputs of the first stage.

(Refer Slide Time: 11:20)

```
assign s10_lsb[7:0] = s00_lsbreg2[6:0]
                    +s01_lsbreg2[6:0] ;
//Add lsb first - s10_lsb[7] is the carry.
assign s11_lsb[7:0] = s02_lsbreg2[6:0]
                    +s03_lsbreg2[6:0] ;
//s00,s01 lsbs need not be registered
//since addition is already carried out here.

always # (posedge clk)

// Pipeline 3; clk (3). Register msb to continue
// addition of msb.

begin
```

We designate these as s10 and s11. This is the lsb sum and this will naturally be the carry resulting here to which we will add msb later on.

(Refer Slide Time: 11:33)

```
//s00,s01 lsbs need not be registered
//since addition is already carried out here.

always # (posedge clk)

// Pipeline 3; clk (3). Register msb to continue
// addition of msb.

begin

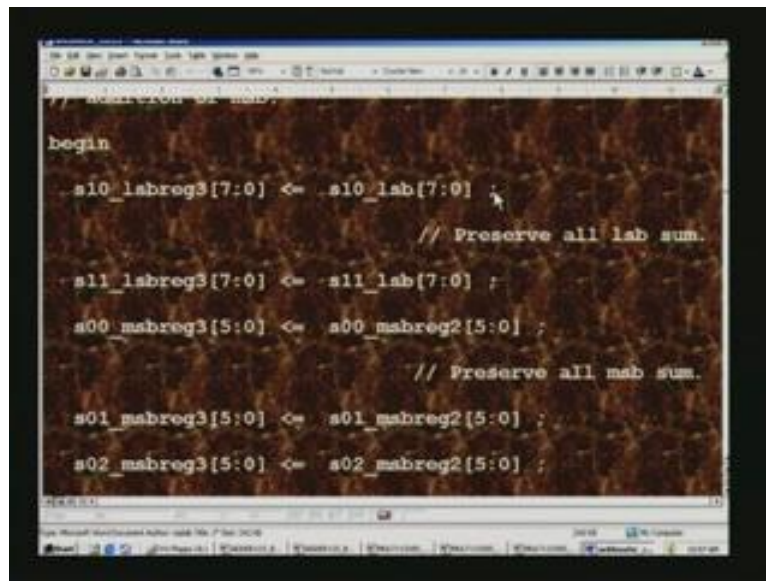
    s10_lsbreg3[7:0] <= s10_lsb[7:0] ;
// Preserve all lsb sum.

    s11_lsbreg3[7:0] <= s11_lsb[7:0] ;
```

Now what we have to do is when the positive edge of the clock is encountered, that is at clock 3; we will register msb to continue addition of msb later on.



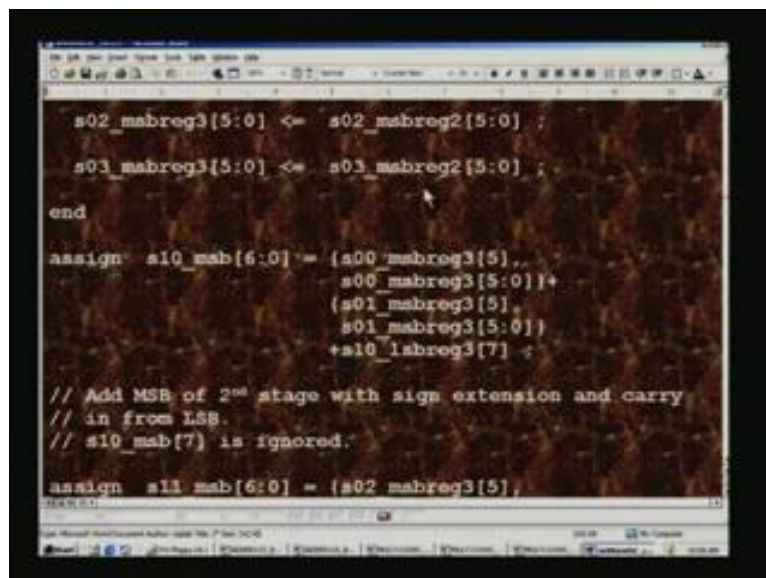
(Refer Slide Time: 11:46)



```
begin
  s10_lsbreg3[7:0] <= s10_lsb[7:0] ;
                                     // Preserve all lsb sum.
  s11_lsbreg3[7:0] <= s11_lsb[7:0] ;
  s00_mabreg3[5:0] <= s00_mabreg2[5:0] ;
                                     // Preserve all mab sum.
  s01_mabreg3[5:0] <= s01_mabreg2[5:0] ;
  s02_mabreg3[5:0] <= s02_mabreg2[5:0] ;
```

We have also to preserve this lsb sum because we need it at the end. We have here s10 and s11. This is the second stage so there will be only two outputs here. This is msb so we need to preserve the msb. That is lsb and this is msb. We continue the addition.

(Refer Slide Time: 12:19)



```
s02_mabreg3[5:0] <= s02_mabreg2[5:0] ;
s03_mabreg3[5:0] <= s03_mabreg2[5:0] ;

end

assign s10_mab[6:0] = {s00_mabreg3[5],
                    s00_mabreg3[5:0]]+
                    {s01_mabreg3[5],
                    s01_mabreg3[5:0]]
                    +s10_lsbreg3[7] ;

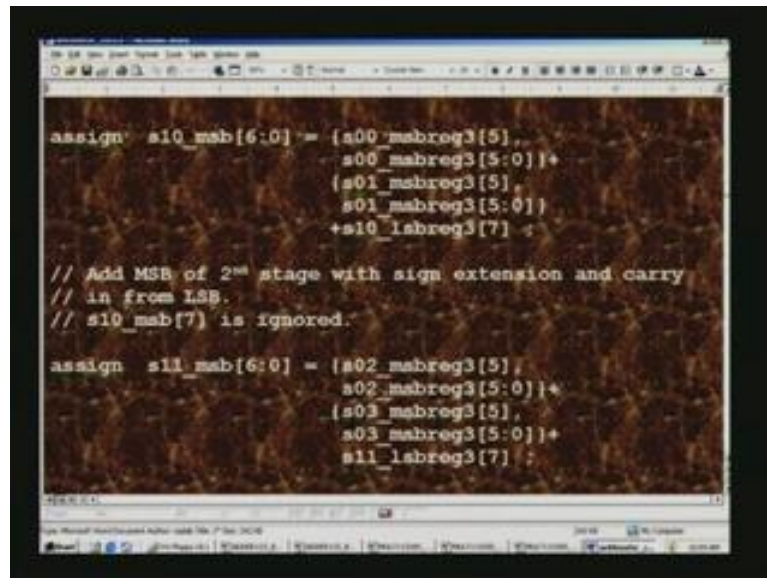
// Add MSB of 2nd stage with sign extension and carry
// in from LSB.
// s10_mab[7] is ignored.

assign s11_mab[6:0] = {s02_mabreg3[5],
```

As far as the msb is concerned we will add the msb in this fashion. For example, as this is the second stage, we have added only the lsb earlier. We must continue adding the msb along with carry that was generated in the lsb addition of the first stage output, which is this one here. As usual we concatenate the sign extended bit as well as the actual number of msbs that is. So is the case for the first output or the first stage and this is the second output so when

you add these together out comes this result which is also in assign statement being a combinational output.

(Refer Slide Time: 13:04)



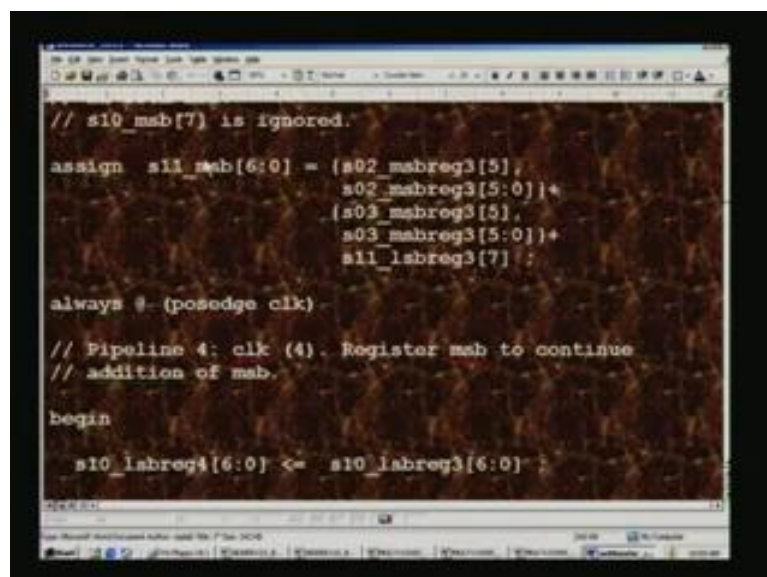
```
assign s10_msb[6:0] = {s00_msbreg3[5],
                    s00_msbreg3[5:0]}+
                    {s01_msbreg3[5],
                    s01_msbreg3[5:0]}+
                    +s10_lsbreg3[7] ;

// Add MSB of 2nd stage with sign extension and carry
// in from LSB.
// s10_msb[7] is ignored.

assign s11_msb[6:0] = {s02_msbreg3[5],
                    s02_msbreg3[5:0]}+
                    {s03_msbreg3[5],
                    s03_msbreg3[5:0]}+
                    s11_lsbreg3[7] ;
```

And note that we have added the carry here. That is what the comment says here add msb of second stage with sign extension and carry in. This is carry in from lsb. The lsb carry out becomes the carry in for this msb. It also results in 1 more bit, which is nothing but duplication of the same sign bit. Therefore, we ignore that.

(Refer Slide Time: 13:26)



```
// s10_msb[7] is ignored.

assign s11_msb[6:0] = {s02_msbreg3[5],
                    s02_msbreg3[5:0]}+
                    {s03_msbreg3[5],
                    s03_msbreg3[5:0]}+
                    s11_lsbreg3[7] ;

always @(posedge clk)

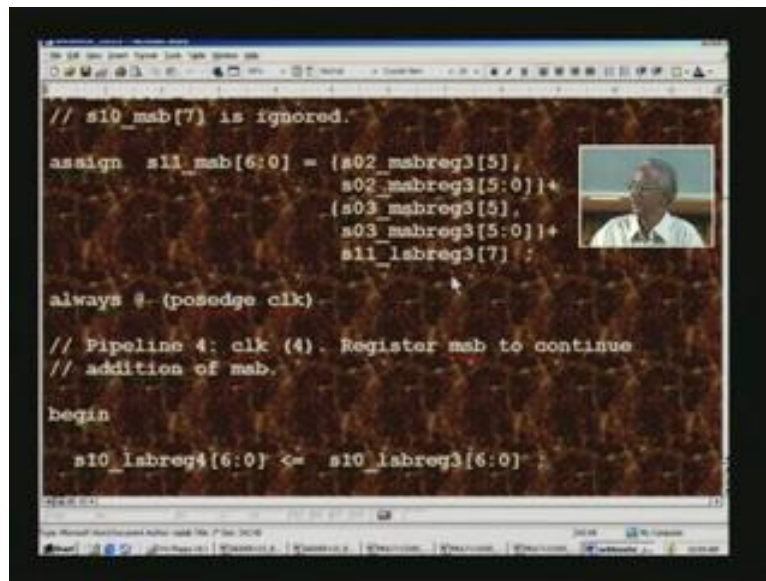
// Pipeline 4: clk (4). Register msb to continue
// addition of msb.

begin

    s10_lsbreg4[6:0] <= s10_lsbreg3[6:0] ;
```

We have msb, sign extended and this is also msb; once again sign is extended.

(Refer Slide Time: 13:43)

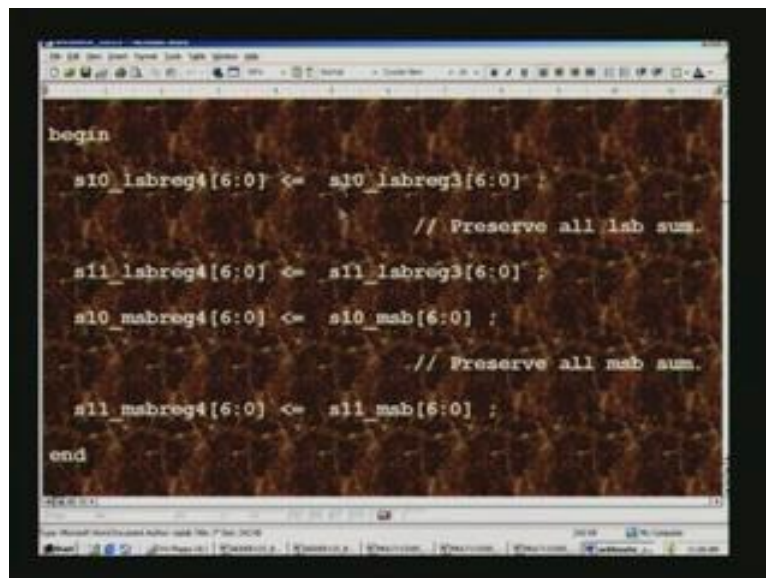


```
// s10_mab[7] is ignored.
assign s11_mab[6:0] = {s02_mabreg3[5],
                    s02_mabreg3[5:0]]+
                    (s03_mabreg3[5],
                    s03_mabreg3[5:0]]+
                    s11_lsbreg3[7] ;

always @(posedge clk)
// Pipeline 4: clk (4). Register mab to continue
// addition of mab.
begin
    s10_lsbreg4[6:0] <= s10_lsbreg3[6:0] ;
```

There is also one carry resulting from the previous one. What we have seen earlier is for first output, s10 and here we see for s11. It is exactly the same thing as that. Now we have added lsb as well as msb. At the next clock edge, we will register the msb to continue addition of msb. This is at the positive edge of the clock and this is clock 4.

(Refer Slide Time: 14:13)



```
begin
    s10_lsbreg4[6:0] <= s10_lsbreg3[6:0] ;
                                // Preserve all lsb sum.
    s11_lsbreg4[6:0] <= s11_lsbreg3[6:0] ;
    s10_mabreg4[6:0] <= s10_mab[6:0] ;
                                // Preserve all mab sum.
    s11_mabreg4[6:0] <= s11_mab[6:0] ;
end
```

Here, we also preserve the lsb that we have added earlier because we finally have to put them all together, msb and lsb. That is the reason why we preserve. All these are preserving lsb and this is preserving the msb. Note that in second stage we have only two outputs. Once again, the last bit will be the carry or even the sign bit.



(Refer Slide Time: 14:44)

```
// Third stage addition.
assign s20_lsb[7:0] = s10_lsbreg4[6:0] +
                    s11_lsbreg4[6:0];

//Add lsb first - s20_lsb[7] is the carry.

always @(posedge clk)
// Pipeline 5: clk (5). Register msb to continue
// addition of msb.
begin
//...
end
```

In the third stage addition, this is the final output that nomenclature has. What we got from the second stage was s10 and s11 and that was only lsb. Therefore, we add the two lsbs of the second stage. Add lsb first s20 lsb7 is the carry. Here, this is the carry.

(Refer Slide Time: 15:15)

```
//...
always @(posedge clk)
// Pipeline 5: clk (5). Register msb to continue
// addition of msb.
begin
s10_msbreg5[6:0] <= s10_msbreg4[6:0];
// Preserve all msb sum.
s11_msbreg5[6:0] <= s11_msbreg4[6:0];
s20_lsbreg5cy <= s20_lsb[7];
//...
end
```

At the arrival of next clock, that is, clock 5, register msb to continue addition of msb. We just transfer it to another register and note that only 5 is different from this. So we just preserve all msb here and for s00 as well as s11.



(Refer Slide Time: 15:45)

```
// addition of msb.
begin
  s10_mabreg5[6:0] <= s10_mabreg4[6:0];
                                     // Preserve all msb sum.
  s11_mabreg5[6:0] <= s11_mabreg4[6:0];
  s20_labreg5cy <= s20_lab[7];
                                     // Preserve all lab sum.
  s20_labreg5[6:0] <= s20_lab[6:0];
end
```

Addition of msb, preserve all msb and also the lsb sum. Finally, note that s20 is the output. This is the one. This is last, lsb and msb are here and this is only a carry.

(Refer Slide Time: 16:02)

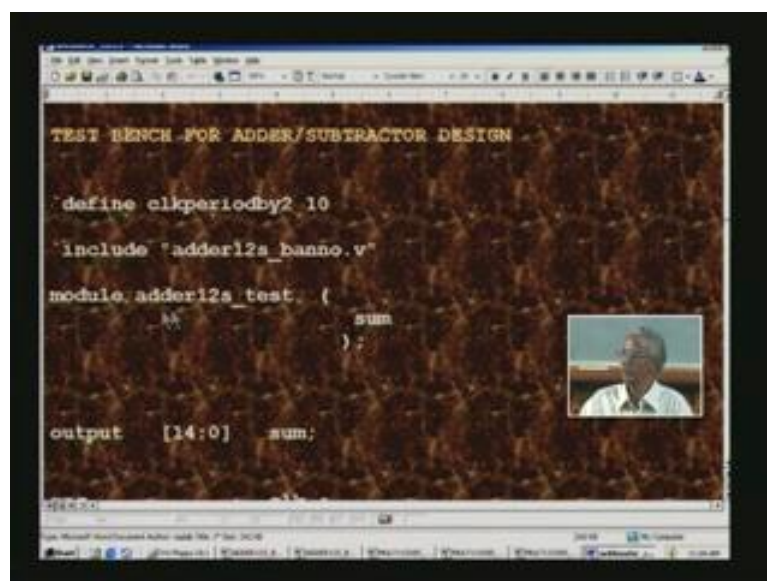
```
s20_labreg5[6:0] <= s20_lab[6:0];
end

// Add third stage MSB result and concatenate
// with LSB result to get the final result.
assign sum[14:0] = {((s10_mabreg5[6],
  s10_mabreg5[6:0]) +
  {s11_mabreg5[6],
  s11_mabreg5[6:0]) +
  s20_labreg5cy,
  s20_labreg5[6:0]};
endmodule
```

We add the third stage msb result and concatenate with lsb result to get the final result. What we have done so far is that we have added lsb of the second stage as well as the msb of the second stage along with the carry that is resultant from the addition of lsb. Now we have the total value available there in the form of two outputs namely, s10 as well as s20.

So the final result is addition of 2 numbers that results from the output of the second stage. There is basically s10 and s11 and we have to sign extend as we have done before. That is what we have done here. The very same thing is extended here. We have to add the msb to that because we have not done the addition as set; we have done addition only for lsb. Having done that one, that is, from this one we also have to concatenate the lsb because we have added that previous stages and that also lsb have to be concatenated. These are the flower brackets used for concatenation. Once again, concatenation here just it separate out, I mean to sign extend so is this case sign extension of this number. So in the second stage msb, there are two numbers, s10 and s11 are added with sign extension. To that we concatenate the lsb because we have added this earlier. So out comes the final sum, 14 through 0. You can very easily reason this out. For example, it is seven bits and concatenation eight bits; this is also eight bits and we have 7 plus 1 equals eight bits here. If you add all these, this will be eight bits, 8 plus 8 then plus 7 so you will get a total of fifteen bits. This completes the design.

(Refer Slide Time: 18:42)

A screenshot of a text editor window showing a Verilog test bench for an adder/subtractor design. The code includes a clock period definition, an include statement for the adder12a\_banno.v file, and a module declaration for adder12a\_test. The module has two 8-bit inputs, a 1-bit control signal, and a 15-bit output named sum. A small video inset of a person is visible in the bottom right corner of the editor window.

```
TEST BENCH FOR ADDER/SUBTRACTOR DESIGN

define clkperiodby2 10

include "adder12a_banno.v"

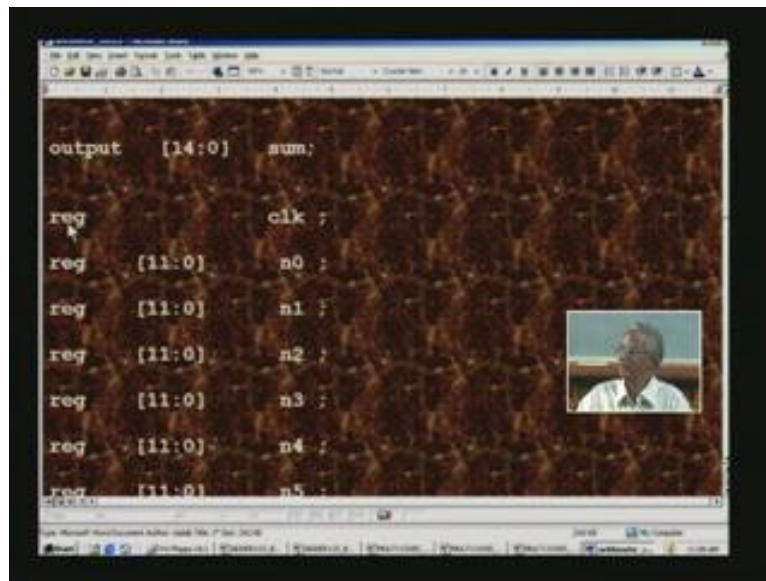
module adder12a_test (
    a, b, cin, sum
);

output [14:0] sum;
```

We will look into the test bench for adder subtractor design. As usual, we want to run at say, 100 megahertz. So we define clock period by 2 and we need to have back annotated file for adder. We declare the test module, which is this and we want to return only the final sum. Note that we had used only assign statement for this sum and therefore, it is not registered.

That is why it is in 5 clock cycle. If you want to register it, you need one more clock cycle. We do not want registration right now.

(Refer Slide Time: 19:18)

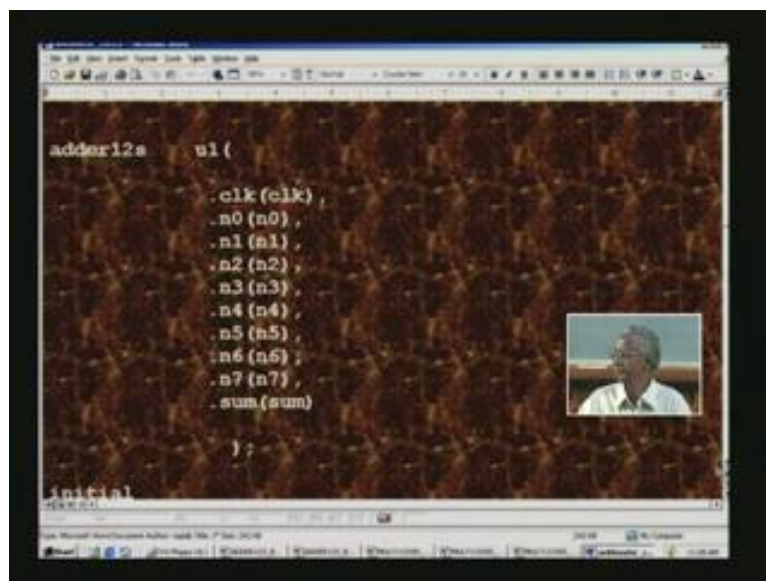


```
output [14:0] sum;

reg clk ;
reg [11:0] n0 ;
reg [11:0] n1 ;
reg [11:0] n2 ;
reg [11:0] n3 ;
reg [11:0] n4 ;
reg [11:0] n5 ;
```

Once again, the sum is declared and the entire inputs clock, n0 through n7 is registered in the test bench. Then, we invoke the actual design which is added 12s and instantiate and list I/Os.

(Refer Slide Time: 19:29)



```
adder12s u1 (
    .clk (clk) ,
    .n0 (n0) ,
    .n1 (n1) ,
    .n2 (n2) ,
    .n3 (n3) ,
    .n4 (n4) ,
    .n5 (n5) ,
    .n6 (n6) ,
    .n7 (n7) ,
    .sum (sum)
);

initial
```

For example, clock and n through n7 are inputs and final sum is the output.

(Refer Slide Time: 19:33)



```
initial
begin
    clk = 1'b0 ;
    n0 = 12'h0 ;
    n1 = 12'h0 ;
    n2 = 12'h0 ;
    n3 = 12'h0 ;
    n4 = 12'h0 ;
    n5 = 12'h0 ;
    n6 = 12'h0 ;
    n7 = 12'h0 ;

    #17 n0 = 12'hfff ;
```

Then, we start the real testing process by initializing with begin and matching end. In between at a different of points, we keep changing the various inputs. For example, all the inputs are initialed to 0 here.

(Refer Slide Time: 19:50)



```
n4 = 12'h0 ;
n5 = 12'h0 ;
n6 = 12'h0 ;
n7 = 12'h0 ;

#17 n0 = 12'hfff ;
n1 = 12'hfff ;
n2 = 12'hfff ;
n3 = 12'hfff ;
n4 = 12'hfff ;
n5 = 12'hfff ;
n6 = 12'hfff ;
n7 = 12'hfff ;

#20 n0 = 12'h7ff ;
n1 = 12'h7ff ;
n2 = 12'h7ff ;
```

Let us say, we apply a different number at 17 nanoseconds. In this case, all fs mean all 1s. All 1s mean we have seen in two's complement representing it is minus 1. We shall add minus 1 here. We shall add all 0s here. After 20 nanoseconds we shall change again into a different value.



(Refer Slide Time: 20:18)



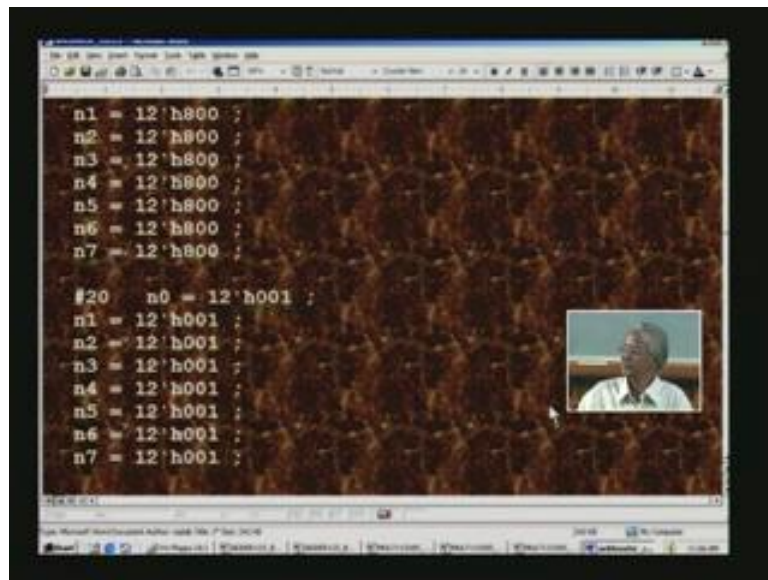
```
n3 = 12'hfff ;
n4 = 12'hfff ;
n5 = 12'hfff ;
n6 = 12'hfff ;
n7 = 12'hfff ;

#20 n0 = 12'h7ff ;
n1 = 12'h7ff ;
n2 = 12'h7ff ;
n3 = 12'h7ff ;
n4 = 12'h7ff ;
n5 = 12'h7ff ;
n6 = 12'h7ff ;
n7 = 12'h7ff ;

#20 n0 = 12'h800 ;
n1 = 12'h800 ;
```

It is 20, 47 or something which we will be clear when we look at the waveform. We have initially staggered a bit here, 17 nanoseconds, just to make sure the data is stable before the clock arrives.

(Refer Slide Time: 20:38)



```
n1 = 12'h800 ;
n2 = 12'h800 ;
n3 = 12'h800 ;
n4 = 12'h800 ;
n5 = 12'h800 ;
n6 = 12'h800 ;
n7 = 12'h800 ;

#20 n0 = 12'h001 ;
n1 = 12'h001 ;
n2 = 12'h001 ;
n3 = 12'h001 ;
n4 = 12'h001 ;
n5 = 12'h001 ;
n6 = 12'h001 ;
n7 = 12'h001 ;
```

We change here every 20 nanoseconds. I earlier mentioned 100 megahertz but it is actually 50 megahertz because the loaded value is only 10. That is just half the time period; therefore, it must be 20 nanoseconds time period. Hence, the clock will have to be 50 megahertz. We change another set of data to 800 then all 1s. We will have a look back when we see the waveform. Once again there is the difference 1 plus 1 minus 1 plus 1 minus 1. So you can

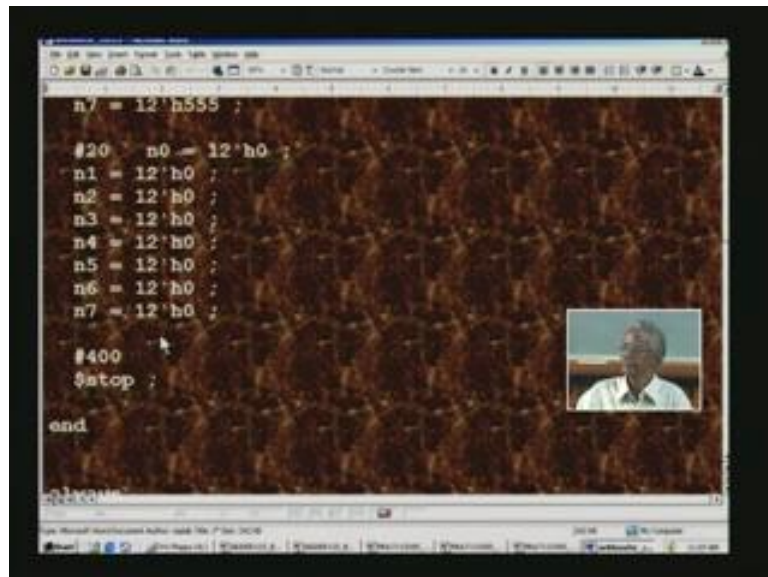
very easily compute the sum morally where possible. I have also jotted it down separately. We can compare with that.

(Refer Slide Time: 21:22)



We can, once again see one number here and another number and so is the case with all 1 0 1 0 pattern and 0 1 01 pattern here.

(Refer Slide Time: 21:38)



I think this is the final thing. Once again, we have all 0s and after giving some quotient of 400 nanoseconds we stop. This completes the test bench and we finally have, in the test

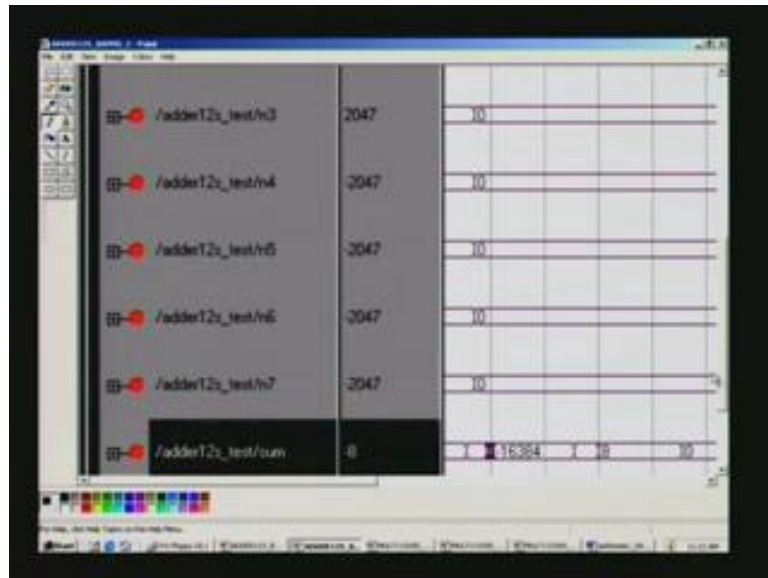






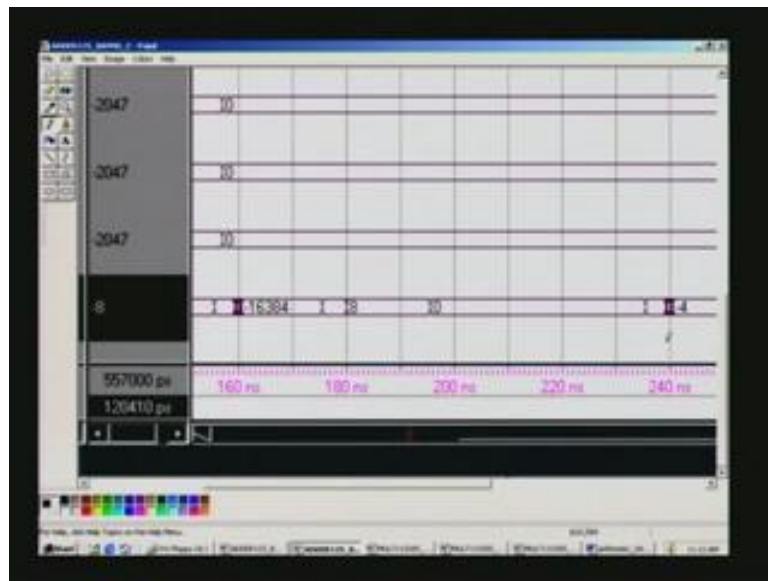
2047 so it should yield 16377. That is also correct. The next one is 8 into minus 2048 and the result must be minus 16384. That is correct. We have one more waveform. I will also zoom this which will continue from the previous one.

(Refer Slide Time: 26:02)



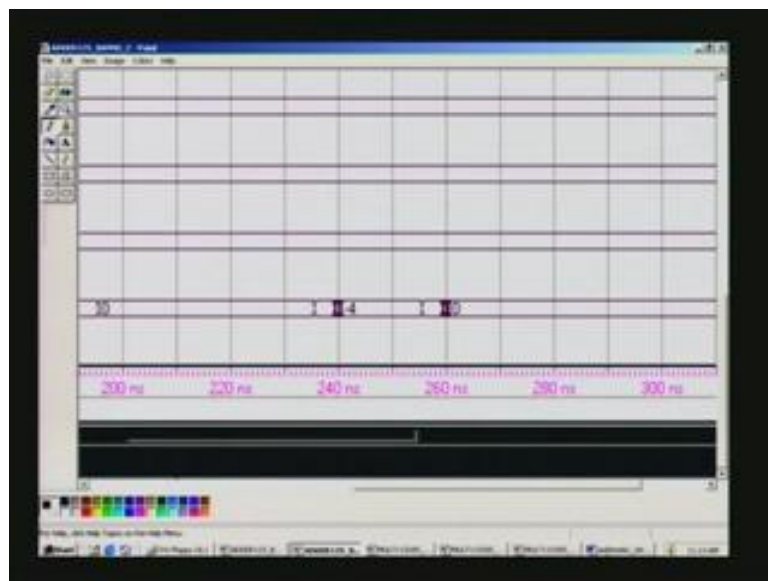
This is the last data. All of them are 0s and the first one is clock. The final sum is here. We have already seen minus 161384 just overlapped so that you will have continuity in visualizing this. The next one is 8 into 1 as per my list and it should give us a sum of 8. That is what it is here. The next one is 4 into 1 minus 1 into 4 that must produce 0 and that also is there. There is one more 0 here. Let us find out whether it is shown. The next value is 4 into 2047 minus 2047 and that naturally produces 0 that is what is here. The next value is 4 into minus 1366 plus 1365 and this is 4 into minus 1 that is minus 4. So that is what you have.

(Refer Slide Time: 26:50)



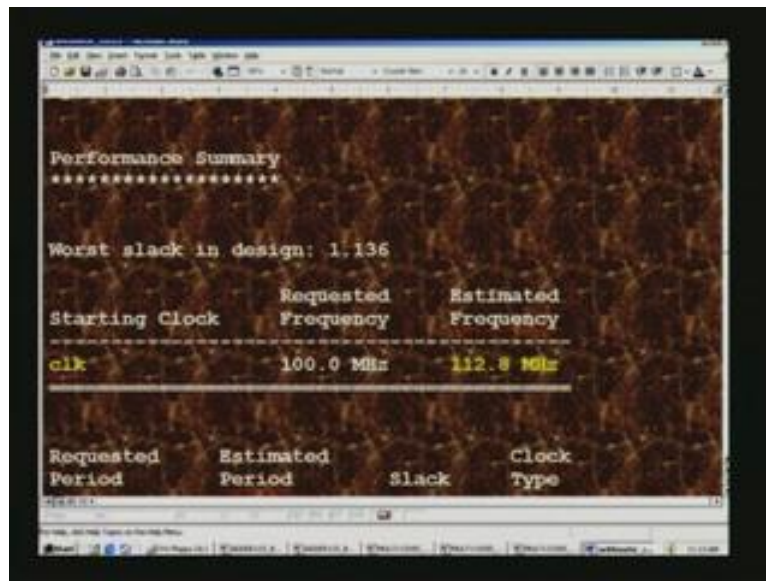
Finally, 8 into 0 must produce a result of 0. That is what you have here.

(Refer Slide Time: 27:08)



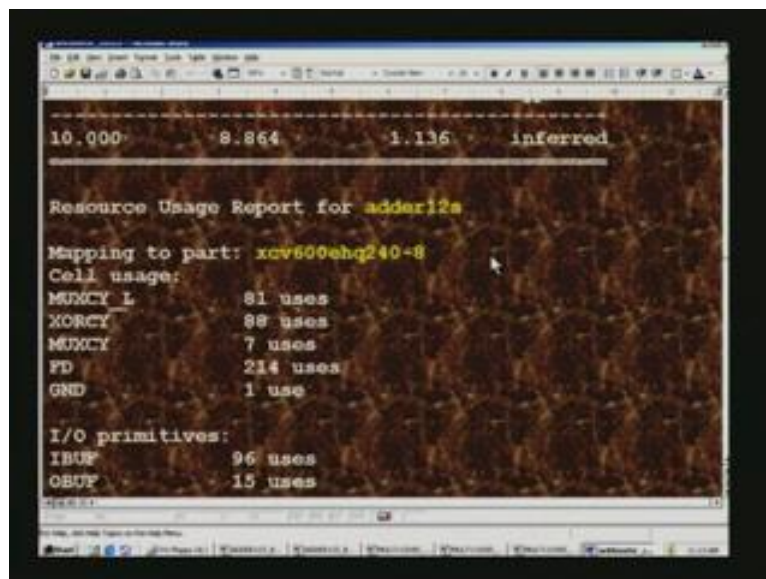
This proves that the addition is working and all of them are signed numbers. We will have a look at the simplicity results for this.

(Refer Slide Time: 27:22)



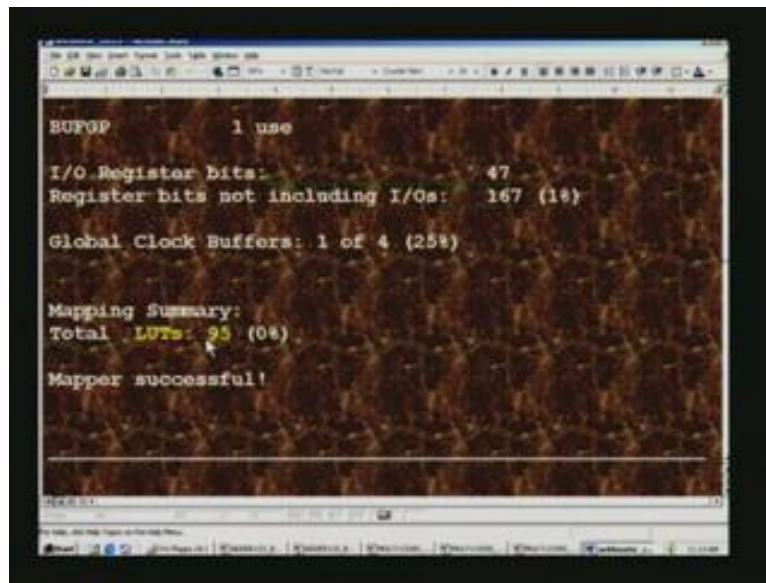
It works at clock of 112 113 megahertz. This is the simplify synthesis result.

(Refer Slide Time: 27:24)



We have mapped as usual on the standard device for the reason mentioned earlier and this is the design. Various internal primitives are all listed and finally a number of LUTs are listed here.

(Refer Slide Time: 27:41)




```
BUFGP1      1 use
I/O Register bits:      47
Register bits not including I/Os: 167 (18)
Global Clock Buffers: 1 of 4 (25%)

Mapping Summary:
Total LUTs: 95 (0%)
Mapper successful!
```

It takes only 95. Although it says 0 percent, it cannot really be 0 there and it is a fraction less than one percent. So that is the synplify result.

(Refer Slide Time: 27:53)



```
Xilinx FCR Results

Design Summary:
Number of errors:      0
Number of warnings:    0
Number of Slices:      97 out of 6,912  1%
Number of Slices containing
unrelated logic:      0 out of 97  0%
Number of Slice Flip Flops: 167 out of 13,824  1%
Number of 4 input LUTs: 95 out of 13,824  1%
Number of bonded IOBs: 111 out of 150  70%
IOB Flip Flops:      47
Number of GCLKs:      1 out of 4  25%
Number of GCLKIOBs:  1 out of 4  25%
```

Here come the Xilinx place and route results and you have all the slices, flip flops, LUTs listed.



(Refer Slide Time: 28:00)

```
Number of slice flip flops: 161 out of 13,824 1%
Number of 4 input LUTs: 95 out of 13,824 1%
Number of bonded IOBs: 111 out of 150 70%
IOB Flip Flops: 47
Number of GCLKs: 1 out of 4 25%
Number of GCLKIOBs: 1 out of 4 25%

Total equivalent gate count for design: 2,810
Additional JTAG gate count for IOBs: 5,376

Mapping Completed.

Timing summary:
-----

Timing errors: 4 Score: 3037

Constraints cover 1015 paths, 0 nets, and 468
```

What is our interest is the number of gates stands by; it says two input NAND gates and it comes to around 8000 gates for a single adder which adds eight numbers of 12 bit signed numbers.

(Refer Slide Time: 28:16)

```
Mapping completed.

Timing summary:
-----

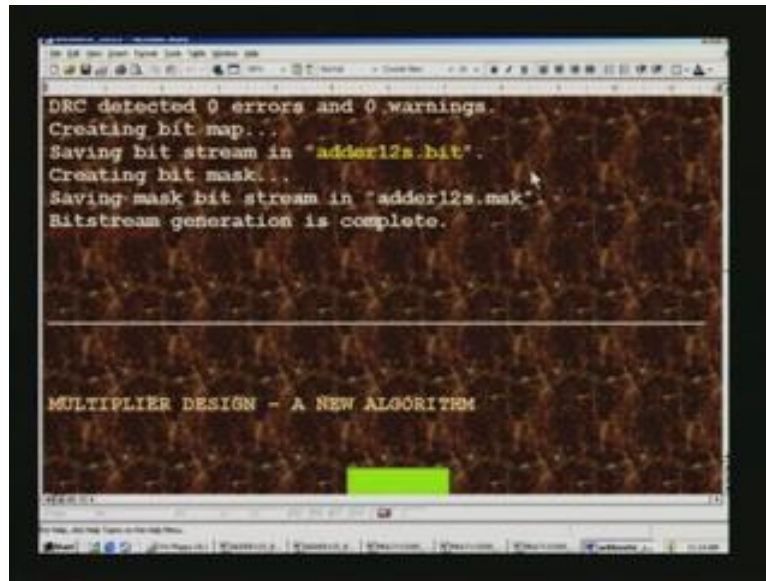
Timing errors: 4 Score: 3037

Constraints cover 1015 paths, 0 nets, and 468
connections (100.0% coverage)

Design statistics:
  Minimum period: 6.563ns (Maximum frequency:
  152.369MHz)
  Minimum input arrival time before clock: 4.259ns
  Minimum output required time after clock: 11.083ns
```

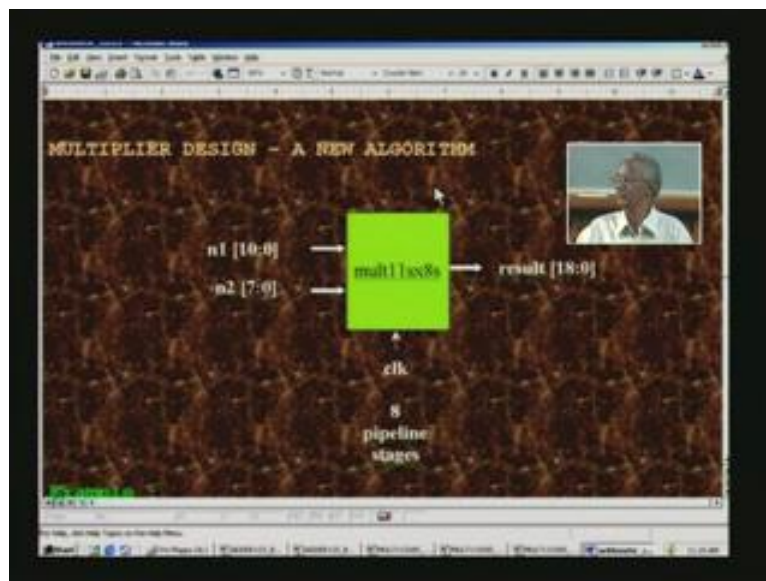
Now, surprisingly Xilinx report much have higher frequency say, 152 megahertz.

(Refer Slide Time: 28:30)



It also produces the dot bit output bit stream output.

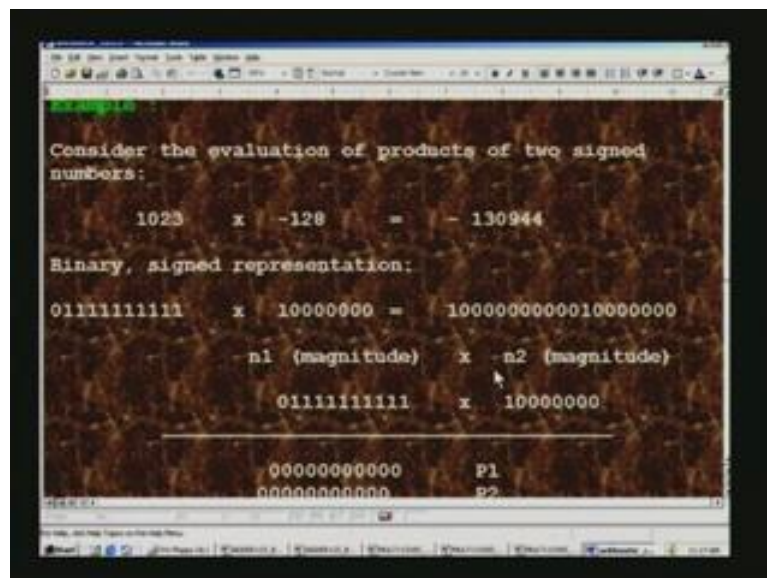
(Refer Slide Time: 28:35)



Next, we will consider the arithmetic circuit and will consider multiplier design. This is a new algorithm developed by the speaker for the sake of implementing on FPGA as well as ASIC yielding as high a through put as possible. This is primarily because I was designing a DCTQ and all the compression algorithms which demand a very high through put. We will have to process some very high resolution picture colors such as 1024 by 768 resolution. All this necessitates heavy pipelining and also a massively parallel circuit.

Mention may be made that FPGA, ASIC are primarily massively parallel and highly pipelined. These are all important characteristics of an FPGA, ASIC field when compared to the microprocessor and dsp processors, wherein, you do not have this much massive parallel sum or heavy pipelining. We get a very high throughput because of those two that is a parallel sum and pipelining on the high end. This particular multiplier is simply a sign multiplier for multiplying two numbers of this dimension for example, 11 bits. This arises from DCTQ application point of view. One number is here and another number is eight bits, 7 through 0 and we nomenclature it as multiplied 11s into 8s and the final output will naturally be the sum total of all the bits, that is, 18 through 0, which is 19 bits. You have 11 and 8 so it should produce 19 bits and that is what it is. Before we go into the details of the algorithm, let me make a mention that this multiplication is done primarily as a magnitude. Therefore we will first separate out the sign from the magnitude and apply the algorithm only for the magnitude. We can deal with the sign separately which turns out be a simple exclusively **or getting** and finally before we consider the algorithm, let us take an example.

(Refer Slide Time: 31:07)



We want to evaluate the product for two numbers, let us say 1023 into minus 128. This happens to be the last possible value in each of these two numbers; this corresponds to n1 and this corresponds to n2. The algorithm works primarily on magnitude and all those signs are automatically incorporated. Internal evaluations are all in magnitude. Let us take this example. If you write this in the binary fashion you will see all of them are 1 and 1 will be the

msb being a negative number. If you evaluate this you see that it is 128, 4 plus 3 is seven bits here

(Refer Slide Time: 32:12)



. The 8th bit naturally waits for 128 and final result will be this in two's complement fashion. This corresponds to minus 130944 and as mentioned before we will take only the sign of this number, which is this. This happens to be exactly the same and for 128, minus and plus happens to be the same. This being a positive number, automatically the same number is put for magnitude. So we can evaluate the magnitude nearly by taking two's complement of any number. For example, you can take a two's complement of this and then out comes the magnitude of this number. We can identify this number as negative number by inspecting the msb which happens to be 1. Now how do we compute? Multiply in the usual way.

We are going in to the algorithm now. When we take two numbers, we take the first bit and then multiply all this and then put here. It happens to be all 0 here because we are going to multiply with 0 and out comes 0. So is this case for all other 0 bits and every time we shift by 1, this is what you do while hand computing and the final one is 1. Therefore, the same gets duplicated and with, of course, one bit shift. You just add in the usual fashion, add bit by bit along with the carry and finally this will be the result as far as the magnitude is concerned. Let us nomenclature each of these partial product which is the multiplication of n1 with single bit of n2. The lsb bit will be first and all other bits will be in this fashion.

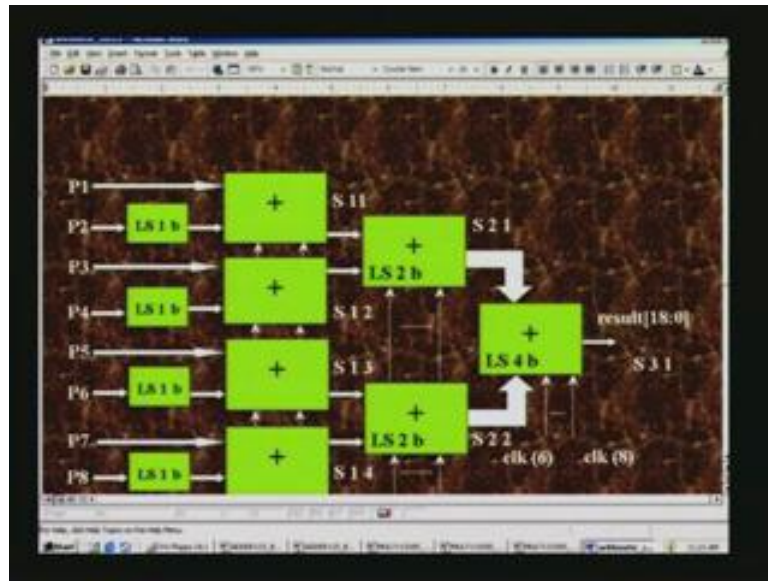


(Refer Slide Time: 33:44)



Out comes this result. This is in magnitude form whereas what we have put here as in two's complement which signifies a negative number. This and the last number are actually equivalent. So we can find it out from this. For example, if you take two's complement, as I mentioned earlier, you will get the magnitude straight away. We are not bothered about the negative number because what you are interested is only evaluating the magnitude. So we apply the two's complement. We just start inspecting till you encounter 1. Retain all the numbers as it is and then invert all the other bits. If you invert all the other bits, all this will become 1 and this will become 0 and then seven 1s will be generated. Apart from this, 1 and seven 0s and apart from that we have 9 bits. Is it same? Extra 1 is there maybe because of sign extension. Or is there a mistake on the top? You cross check this. Even if there is a 0 not the result will be same anyway. I may have put one extra 0 by mistake.

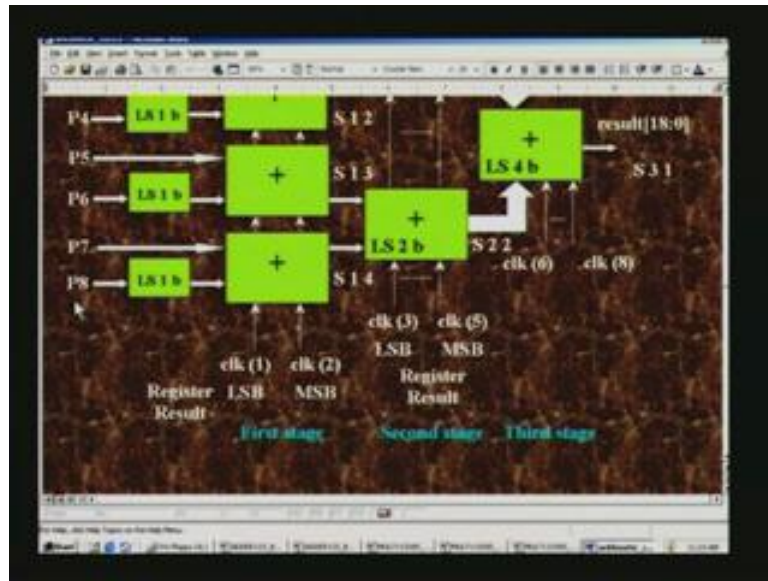
(Refer Slide Time: 35:21)



The algorithm goes like this. It is pictorially depicted. We have once again used bit partitioning as well as functional partitioning. It happens that the functionality will be basically add or subtract. This is in three stages as we have seen in the adder before. The only thing extra is that we need to do left shifting of 1 bit or 2 bits or 4 bits. I will explain the typical case of this new algorithm. Now what we had seen in this previous example is P1 through P8 has been partial product. In the verilog code that we are going to consider we will see how P1 through P8 are arrived at. For the time being it is sufficient if you just add the P1 and P2 straight away. When you do this one, we have four such adders that will add two numbers at one stroke so you will have all partial product results concurrently. That is what we said before that FPGA or ASIC design is characterized by a massively parallel circuit.

When you see DCTQ, you will be astounded by the parallel sum that will be built in. It is also highly pipelined, in the sense, that there are so many pipeline registers internally. For example, two registers are hidden inside and that is the reason why clock has been input there. In this case, two more here, again two more, and in fact, if you see a clock, we have the very first clock.

(Refer Slide Time: 37:04)



In fact, the very first clock will register this P1 through P8 and then n0 and n1 are the actual inputs that will be registered first and followed by this registry. Only with the arrival of the next clock pulse do we reckon it as first clock in order to do the internal processing. We need 1 through 8 clock pulses. There are pipelined registers internally. There are three stages of addition and there is also a shift here. That is happening inside here and shifting is very easy in verilog. We have only to take the relevant bits and therefore, the whole multiplier is reduced to mere addition and nothing more, primarily addition because two's complement, sign is taken care of by mere addition. So, the algorithm turns out to be very straightforward and simple.

(Refer Slide Time: 38:15)

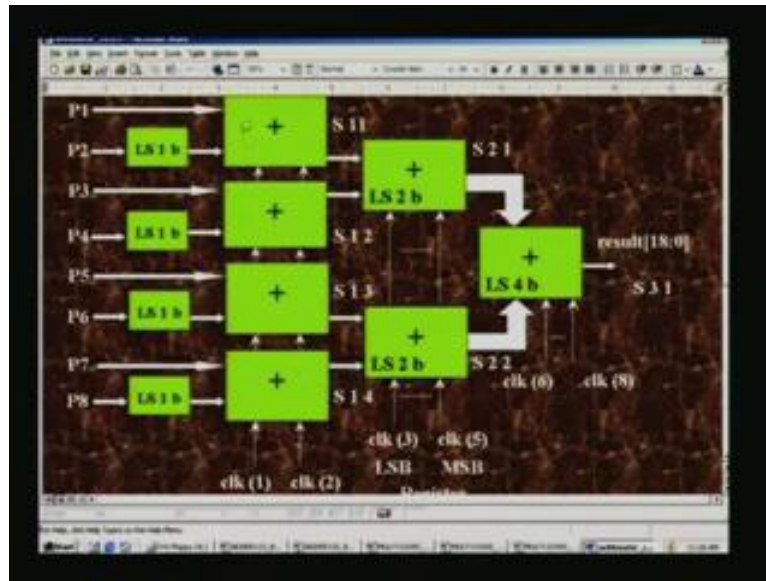


We have earlier seen that the partial products are P1 through P8. We add these two and concurrently we add these two, these two and these two, by the 4 adders of the first stage that we have seen. Out comes the result. While adding this one, note that we do not have to add this first bit because you know that there is no matching number here. It is equivalent to having a 0 here. When you add 0 and 0 you will get 0. It happens to be 0 in this example. But do not jump to the conclusion that all of them can be done in this fashion; it is applicable only for the first bit. There may be other numbers where they will all be non-zero, including this. Even if it is a non-zero, that is, 1, if you add 1 or 0 to 0 you will get the same number. Therefore, we do not add this. What we do is take the entire things bifurcates by 2 and call it lsb and msb. Add just the lsb at every clock and then add the msb at the following clock as we have done before in the adder case. We will follow the very same pattern and at every stage it will be the same. The subsequent addition will result from this. That result will be added in the second stage. Naturally for eight numbers you will have only four outputs because there were only four adders. Those four outputs will be taken two at a time.

Once again, in the second stage of addition, add two numbers at a time and out comes one result. There will be two such outputs. We will add those two in the third stage resulting in one final sum. All these would be registered at different points of time. In total, we have eight clocks and, therefore, eight pipeline stages inside. Hence, eight pipelines register as well.



(Refer Slide Time: 40:12)



Let us say, s11 through s14 are the intermediate results arising from the first stage of the addition. We did not say what this lsb 1 is, so P1 and P2. (Refer Slide Time: 40:34) If you take the example of P1 and P2, if you inspect these two partial products, what we see is this: P2 is staggered a bit so as good as left shift by 1 bit. That is precisely what we are doing for P2 and not only for P2 but also for P4, then P6, as well as P8. So you will see left shift by 1 bit as far as the first stage of addition is concerned; P2 is left shifted by 1 and only then added to P1. This is clock 1, corresponding to which there will internally be a register which will register the sum of lsb alone. What we have seen is the total number is bifurcated into two say, roughly half, one we call lsb and the other we call msb. So, we add only the lsb and carry will result which we will add with msb of these two at this second clock. Then concatenate 2 to make one complete S11 sum there. Like this, we will have four such results.

We will add these two and this time, in the second stage of addition, we have to shift left by two bits; I will explain why in a minute. Similarly, S13 and S14 are added up and once again this S14 is shifted left by two bits and then added up. Once again, this lsb will be added first in clock 3. There is an extra 3 because the bit precision has now increased internally and it will increase progressively. So we want to give a little more quotient. Therefore, we spend one more clock. You can experiment with this to see whether we can reduce the clock and there may be some scope for you to reduce. You may get even six pipelines instead of eight and you can experiment with it. In fact, this is not really required because what we will do is pipeline the whole, not only the multiplier but also the entire process. Even a complicated

algorithm can be totally pipelined. All these pipeline delays will only be a onetime affair. Therefore, we are not really bothered about extra pipelines. The higher the pipelines you do, the higher will be the throughput, that is, the speed of operation will be much higher. So that is a desirable thing and, of course, you have to pay a price for additional pipeline registers. That will only be a marginal increase so you are free to use as many pipelines as possible. We will explain why these two bits here. We take the sum of the first stage.

(Refer Slide Time: 43:30)

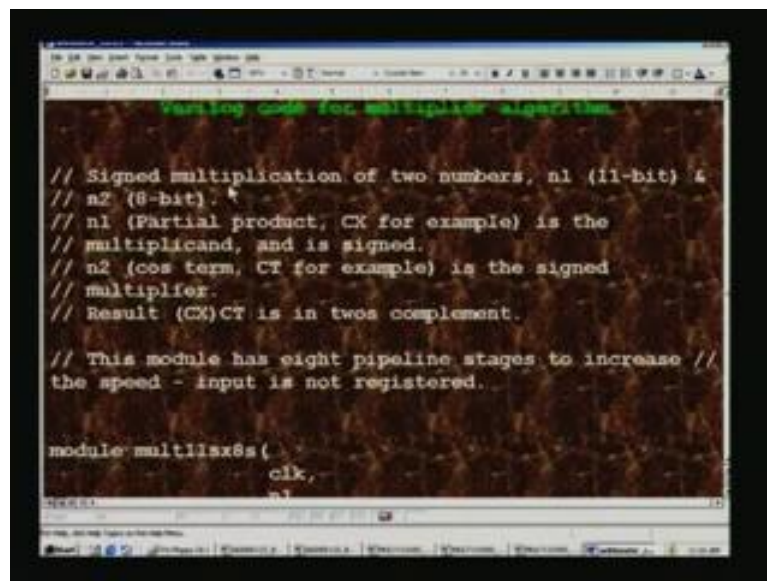


If you look at this example, once again what we do is we add this and out comes this result. We add this and there will be a result here. You are trying to add those two results. So, naturally, the first result is taken as such whereas the second result is shifted by two bits, because if you see the second group, it is shifted by two bits from the first group. This is the first one and the corresponding first partial product is this one. Therefore, there is a one and then two shifts here. Therefore, the second stage of addition will have two shifts.

Likewise, those two, when added in the second stage, will produce one more output. Similarly, you will get one more output from this. So note that, in the last stage, there will be two inputs and those inputs are staggered by four bits because it is the result of these four, first here and these four here. So, when compared to the first one, in this second group you have four bits. So the third stage will be four bits shifted and that is how it is. This is the specialty of this Novell algorithm for the multiplier. (Refer Slide Time: 44:48) There are scores of other algorithms. You are free to look into this and you can make a comparison. In fact, all algorithms are good enough and you are free to use any of them. Once you use this

pipelining approach, it does not really matter which particular algorithm you use; what is important is how efficiently you use it. You can see two bit is shifted before the partial product first stage and final stage when you add the two numbers. Once again, there is a left shift of two bits here for this second number so, in this case here, when we add these two numbers, the second is shifted by two bits. This is the second stage input. For the final stage, we see that it is shifted by four bits for the reason we have explained earlier. Finally, the result is in 19 bit as we have seen before. This explains the algorithm.

(Refer Slide Time: 45:43)

A screenshot of a Verilog code editor showing the implementation of a multiplier. The code is titled "Verilog code for multiplier algorithm" in green. It includes comments explaining the signed multiplication of two numbers, n1 (11-bit) and n2 (8-bit). The code defines a module named "mult11x8s" with an input "clk" and "n1".

```
Verilog code for multiplier algorithm

// Signed multiplication of two numbers, n1 (11-bit) &
// n2 (8-bit).
// n1 (Partial product, CX for example) is the
// multiplicand, and is signed.
// n2 (cos term, CT for example) is the signed
// multiplier.
// Result (CX)CT is in two's complement.

// This module has eight pipeline stages to increase //
the speed - input is not registered.

module mult11x8s(
    clk,
    n1
```

We will look into the verilog code for this algorithm. We have two numbers n1 and n2; n1 is 11 bit and n2 is 8 bit. Both of them are signed numbers. Internally, we will remove this sign temporarily and then apply the algorithm only on the magnitudes. Having got the final results, we will sandwich the sign bits later on at the fag end. Here we have n1 and n2, which are 11 bits and eight bits; 11th and 8th bit will be the sign bit. This particular thing, as I mentioned before, is for DCTQ application. In the DCTQ application, we need to evaluate 8 by 8 matrices. For example, CX is a matrix we will have to evaluate and it will create some partial product which is what is being used as n1. And this is multiplicand and is signed.

In this case, second number is a cos term. So in DCTQ, it is a cosine transform that we will evaluate. We need some cos term for that and these are all stored in the ROM table, the design of which we have already seen before in the ROM design. We had also mentioned that it was with the end goal in mind that we had taken all these examples such as ROM, RAM, Dual RAM and also the adder what we have seen as well as the multiplier that we will be

trying to look in to. All these applications are with the end view of using in DCTQ application which is for video compression used for JPEG and MPEG. This is the signed multiplier and we have a multiplicand and multiplier and the final result which is sum is an evaluation of three matrices CX and CT. We had assumed CX is already available and CX can also be evaluated by using a similar multiplier which we will consider later on.

(Refer Slide Time: 48:14)



```
// Result (CX)CT is in twos complement.
// This module has eight pipeline stages to increase //
the speed - input is not registered.

module multi11x8s(
    clk,
    n1,
    n2,
    result
)

input          clk ;
input [10:0]   n1 ;
input [7:0]    n2 ;
output [18:0]  result ;
```

This result is in two's complement. Now, we will see the actual design. This module has eight pipeline stages to increase the speed and we should make a mention that input is not registered; straightaway, it may be registered elsewhere. For example, these are all the inputs supplied from the ROM table or from the dual ROM table that we have already seen. If you remembered correctly, you could see that there are registers at the output for both dual ROM as well as RAM, which we had seen in the design before. Because the registers are already there we do not have to register the inputs. So we can take them straightaway. This is the module name that we have declared here. So, module multiplier 11 into 8, S stands for sign. This is the module design name we have given; implying that it is signed eleven bit. We have clock as input. Then there must be two numbers as inputs, n1 and n2 and the final result is also listed.



(Refer Slide Time: 49:20)



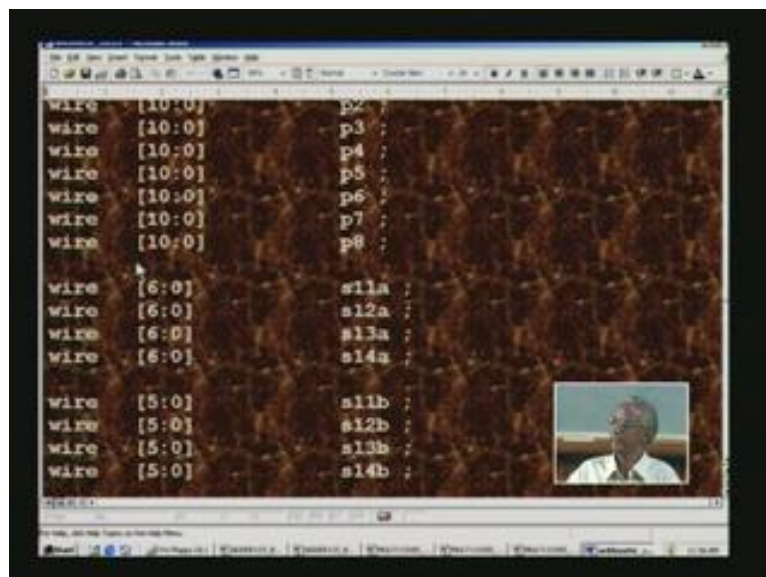
```
module adder
    n2
    result
)
input          clk ;
input [10:0]   n1 ;
input [7:0]    n2 ;
output [18:0]  result ;

wire          n1orn2s ;

wire [10:0]   p1 ;
wire [10:0]   p2 ;
wire [10:0]   p3 ;
wire [10:0]   p4 ;
wire [10:0]   p5 ;
wire [10:0]   p6 ;
```

All I/Os are listed and widths are also mentioned. For example, input here, then n1 and n2 are inputs, eleven bits and eight bits, final result is nineteen bits. Whether there are two numbers n1 and n2, whether it is a 0 or not, we have a separate flag for that. That is what is declared here. We have seen that the partial products are of eleven bits. We need to declare them as wire as they are all evaluated as an assign statement.

(Refer Slide Time: 50:04)



```
wire [10:0]   p2 ;
wire [10:0]   p3 ;
wire [10:0]   p4 ;
wire [10:0]   p5 ;
wire [10:0]   p6 ;
wire [10:0]   p7 ;
wire [10:0]   p8 ;

wire [6:0]    s11a ;
wire [6:0]    s12a ;
wire [6:0]    s13a ;
wire [6:0]    s14a ;

wire [5:0]    s11b ;
wire [5:0]    s12b ;
wire [5:0]    s13b ;
wire [5:0]    s14b ;
```

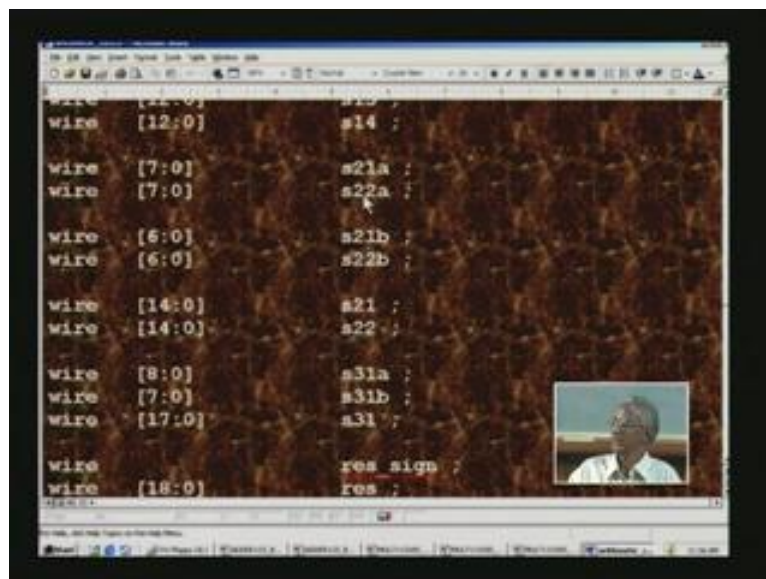
In the case of first stage addition, we have seen S11 through S14. They are also combinational to start with, especially the lsb portion. Even the msb portion will be primarily an assign statement and later on we will be registering in the succeeding clocks.

(Refer Slide Time: 50:20)



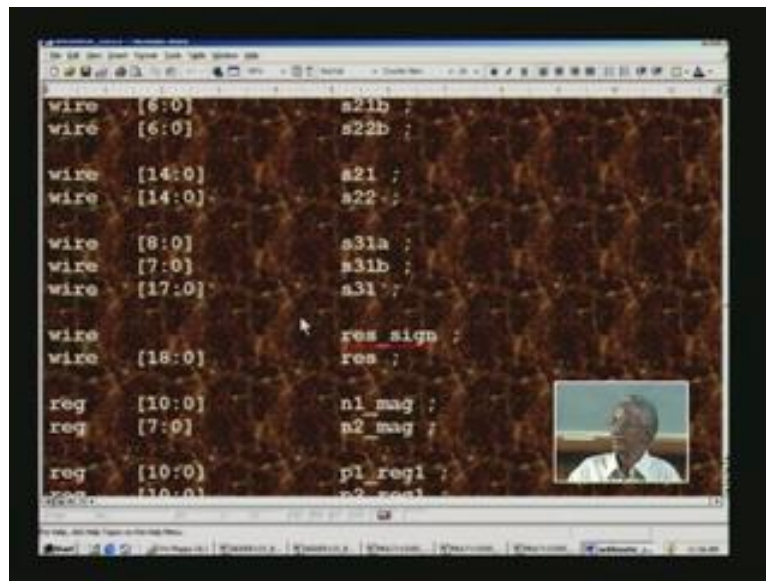
You can see that so many signals like this are used.

(Refer Slide Time: 50:38)



That is for the first stage and this is for the second stage here.

(Refer Slide Time: 50:45)



```
wire [6:0] s21b ;
wire [6:0] s22b ;

wire [14:0] s21 ;
wire [14:0] s22 ;

wire [8:0] s31a ;
wire [7:0] s31b ;
wire [17:0] s31 ;

wire res_sign ;
wire [18:0] res ;

reg [10:0] n1_mag ;
reg [7:0] n2_mag ;

reg [10:0] p1_reg1 ;
reg [10:0] p2_reg1 ;
```

And this is for the third stage.

(Refer Slide Time: 50:52)



```
wire [18:0] res_sign ;
wire [18:0] res ;

reg [10:0] n1_mag ;
reg [7:0] n2_mag ;

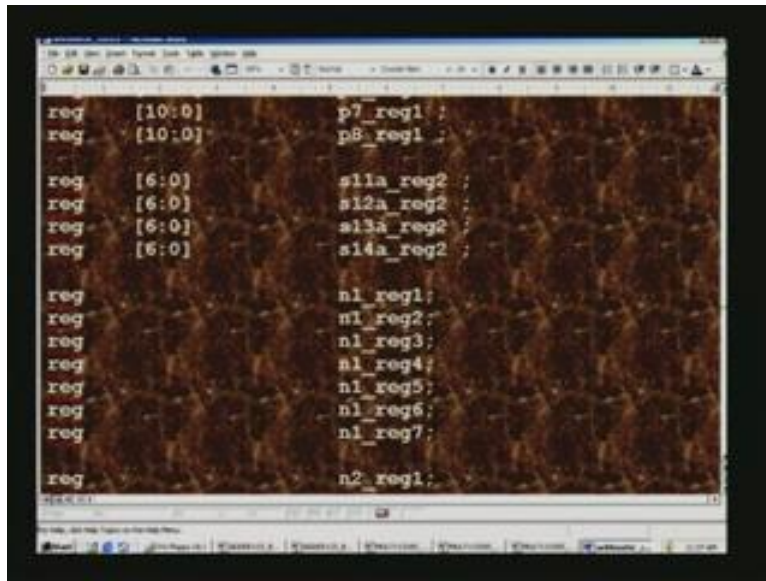
reg [10:0] p1_reg1 ;
reg [10:0] p2_reg1 ;
reg [10:0] p3_reg1 ;
reg [10:0] p4_reg1 ;
reg [10:0] p5_reg1 ;
reg [10:0] p6_reg1 ;
reg [10:0] p7_reg1 ;
reg [10:0] p8_reg1 ;

reg [6:0] s11a_reg2 ;
reg [6:0] s12a_reg2 ;
```

You then have a final result. This sign is also required; we have to keep track of it because we are evaluating the magnitude using the algorithm. The magnitudes for the two numbers  $n1$  and  $n2$  are declared as registers which we will be evaluating. We also declared other register which happened to be in always block and they are all at different clock. For example, we give an extra reg 1 for any registering you do for partial product at clock 1. That is the nomenclature we adopt.



(Refer Slide Time: 51:30)



```
reg [10:0] p7_reg1 ;
reg [10:0] p8_reg1 ;

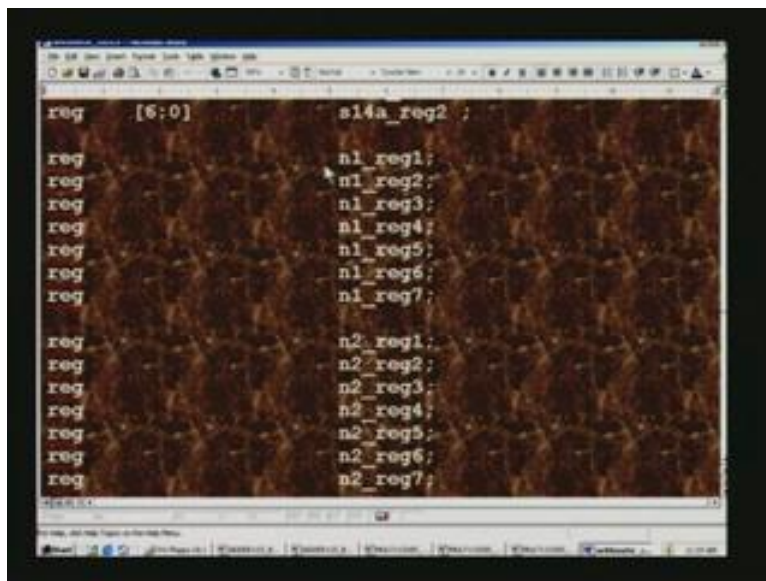
reg [6:0] s11a_reg2 ;
reg [6:0] s12a_reg2 ;
reg [6:0] s13a_reg2 ;
reg [6:0] s14a_reg2 ;

reg n1_reg1 ;
reg n1_reg2 ;
reg n1_reg3 ;
reg n1_reg4 ;
reg n1_reg5 ;
reg n1_reg6 ;
reg n1_reg7 ;

reg n2_reg1 ;
```

Similarly, for other pipeline stages, when clock 2 strikes you will have intermediate results in reg 2 and so on.

(Refer Slide Time: 51:42)



```
reg [6:0] s14a_reg2 ;

reg n1_reg1 ;
reg n1_reg2 ;
reg n1_reg3 ;
reg n1_reg4 ;
reg n1_reg5 ;
reg n1_reg6 ;
reg n1_reg7 ;

reg n2_reg1 ;
reg n2_reg2 ;
reg n2_reg3 ;
reg n2_reg4 ;
reg n2_reg5 ;
reg n2_reg6 ;
reg n2_reg7 ;
```

We also need to propagate the number sometimes, n1 as well as n2.



(Refer Slide Time: 51:53)

A screenshot of a code editor window with a dark background. The code is written in a light-colored font and consists of two groups of instructions. The first group has seven lines, each starting with 'reg' followed by a space and then 'n2\_reg' followed by a number from 1 to 7. The second group also has seven lines, each starting with 'reg' followed by a space and then 'n2rn2r\_reg' followed by a number from 1 to 7. The editor window has a standard Windows-style title bar and toolbar at the top, and a taskbar at the bottom.

```
reg          n2_reg1 ;
reg          n2_reg2 ;
reg          n2_reg3 ;
reg          n2_reg4 ;
reg          n2_reg5 ;
reg          n2_reg6 ;
reg          n2_reg7 ;

reg          n2rn2r_reg1 ;
reg          n2rn2r_reg2 ;
reg          n2rn2r_reg3 ;
reg          n2rn2r_reg4 ;
reg          n2rn2r_reg5 ;
reg          n2rn2r_reg6 ;
reg          n2rn2r_reg7 ;
```

And what they are, we will see when we start analyzing the code. We will continue with this in our next class. Thank you.