Digital VLSI System Design

Prof. Dr. S. Ramachandran

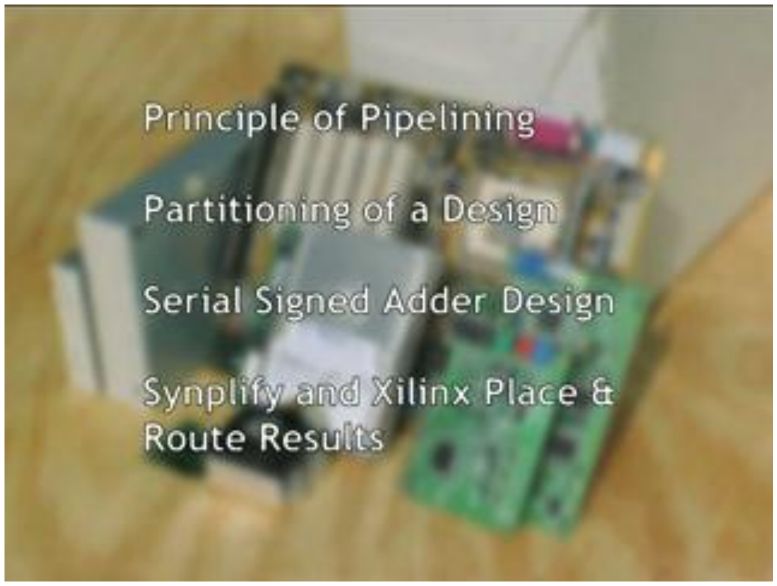
Department of Electrical Engineering

Indian Institute of Technology, Madras

Lecture – 40

Design of Arithmetic Circuit

(Refer Slide Time: 01:42)



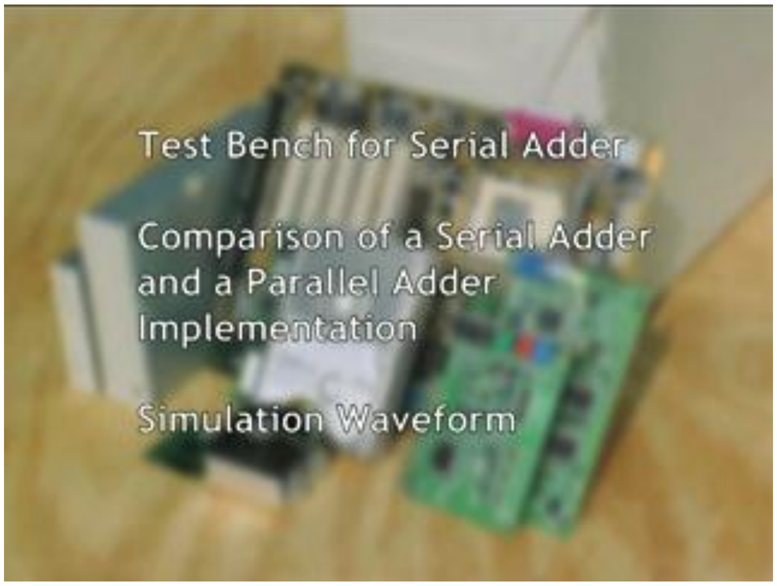
Principle of Pipelining

Partitioning of a Design

Serial Signed Adder Design

Synplify and Xilinx Place & Route Results

(Refer Slide Time: 02:15)



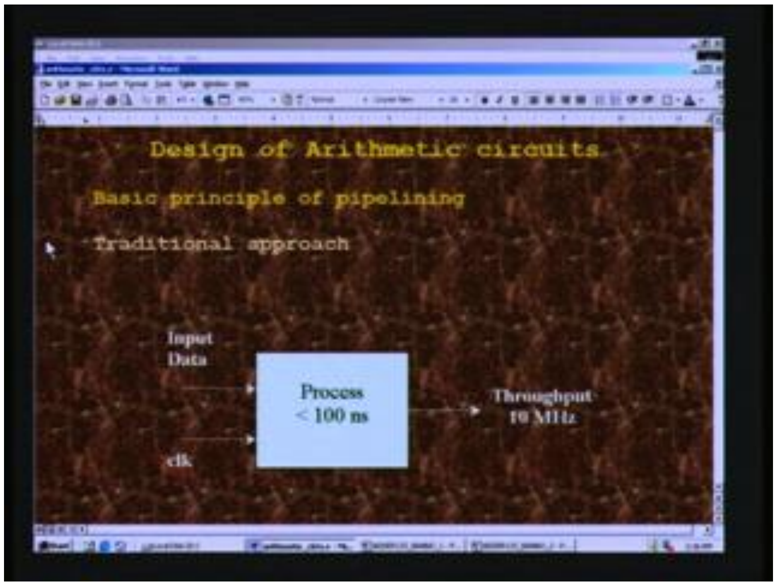
Test Bench for Serial Adder

Comparison of a Serial Adder and a Parallel Adder Implementation

Simulation Waveform

Some of the most important circuit that we need to design for FPGA ASIC implementations are arithmetic circuit, basic arithmetic circuits are add, subtract, multiply and divide and so on.

(Refer Slide Time: 02:31)



Design of Arithmetic circuits

Basic principle of pipelining

Traditional approach

All these circuits are very computationally intensive and therefore, the conventional methods of approach will not be sufficient. We will have go to for what is known as pipelining, and we will explain in a minute what pipelining is and before that, let us see what we are going to cover in this lecture. This applications for arithmetic circuit are basically from the point of view of implementing DCTQ, which will be covering later on, in design applications and whatever we require for that application will be covering in the next two lectures.

To start with, let me also make a mention that, I will be covering only fixed point arithmetic, and not floating point arithmetic for simple reason that, fixed point arithmetic is much less complex, that is takes much less chip area. That is the reason why this has been more popular in FPGA sequent permutation.

The traditional approach what we have is a process, this process can be a simple add, subtract or multiply or complex algorithm, which is the sum total of all the arithmetic circuit that we have just now said mentioned. This process let us say for example, it takes one of the processors takes 100 nanoseconds. The question is- how we going to deal with speeding up the system? It is vital because certain applications such as video scaling, and video compression and the like of it, are very computationally intense operations and requiring specialised algorithms to solve. All these can be done in real time. That is to say, you can process a motion picture only, if you can do the computational very fast. The conventional approach will not help will not take you anywhere near to real time application, therefore, the need for the pipeline approach. Before we go for the pipeline approach, let us see the traditional approach.

For example, we have a multiplier and let us say it takes 100 nanoseconds or it will be less than 100 nanoseconds. What do we do? We apply if it is a real time application we need to apply input data and at periodically. Let us say with respects to a rising edge of the clock. Every positive edge of the clock you need to input data and outcomes the processed output here. Since the total process takes about 100 nanoseconds, you will get only 10 mega hertz through.

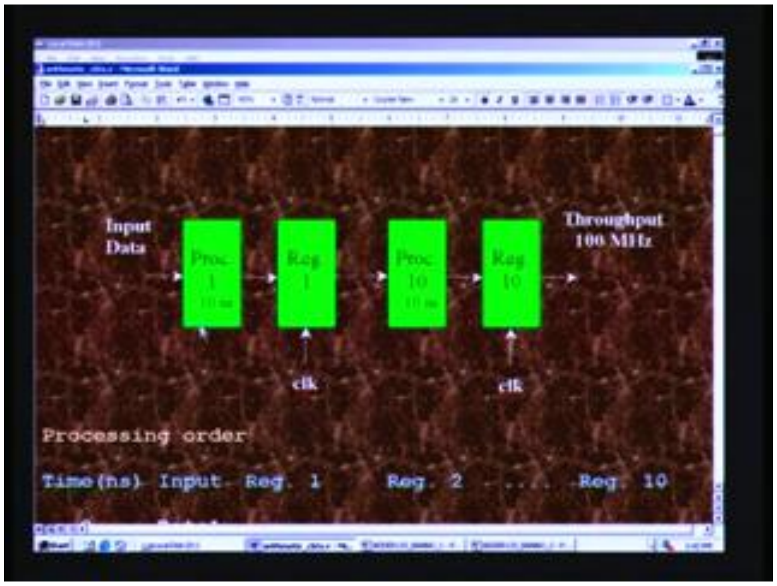
That is a 1000 by 100 is 10 mega hertz and naturally 10 mega hertz will be the clock frequency as well. That is to say, we can apply only input data every 10 mega hertz every 100 nanoseconds. If a multiplier takes this much time, and we cannot do anything worthwhile as far as real time processing is concerned. What we do is, we adopt what is called pipeline approach and this approach basically has advantages as well as disadvantages. The advantage is that throughput increases dramatically.

Let us say for instance, you have only 10 megahertz throughput. How about jagging it up tenfold? Is there any trick we can play, so that this throughput increases tenfold? Yes, there is, and the answer is the pipelining which we are already mentioned about.

This is the greatest advantage that we have, because we can now process say video compression and the rest of the applications of that sort, which demand high throughput. Otherwise, with the conventional approach it was not feasible, and that is the advantage main advantage you have.

To pay up price for that and the price is not very heavy. It is just increase of area chip area in the sense that, we need to have what are known as pipeline registers which will be seeing shortly. Because of those registers, this chip area will increase. In addition to this, even in the conventional approach, if you apply data outcomes the result only after this delay of 100 nanoseconds, this delay is still pre prevalent in the pipelined approach as well. We will see why it is so? The delay is referred to as a latency, and this are the 2 disadvantages that we have that is chip area and latency but this is only a small price we need to pay in order to get very high performance in terms of processing speed. Now let us see what a pipeline is.

(Refer Slide Time: 07:59)



Processing order

Time(ns) Input Reg. 1 Reg. 2 .... Reg. 10

As you all know, we have a pipe through which we run either oil or the like of it to transport from one place to another. In order to bring about this, you have to switch on a motor and start pumping the oil or water as the case may be. Once you switch on, you do not get on the water supply immediately, it comes only after a delay and that delay may be referred to as the latency here and water flows or water or any other liquid flows through the pipe.

In digital rams instead of water we can assume data being flowing into a pipe which is composed of not only process but also registers. We will see in a minute, why these are separately indicated and that is the analogy that we have, we can draw for this digital pipeline.

Water pipeline is the analogy. What we claim is, we mentioned earlier is there any possibility of improving this speed. We also mentioned that dramatically, it can change say, from 10 mega hertz speed of operation to 100 mega hertz operation, just by pipelining and having the very same hardware and how to do this?

Let us say, the previously we had a single process it could be or multiple process. This process can be simple add or subtract, multiply or combination of all these or including even a complex algorithm which we may need for video applications. All we need to do is, divide this process into some convenient small time consuming process. Say, it is mere question of division of the entire process strategically.

Let us say we have some 10 processes, and each of these processes roughly 10 nanoseconds or less than 10 nanoseconds. If you add all these, it will amount to 100 nanoseconds which is the same as this process. We will just divide this process 10 times. We will have process one less than 10 nanoseconds. Then process 2 here I have put 10. That means, a dotted line implying that similar blocks are there for the rest of it, and ultimately it is processed here.

You would notice that immediately after process, you will have a register here in order and clock by the usual system clock, and at the positive edge of the clock, whatever is available as the process one result; these are all intermediate results that will be registered here.

For example, whatever is registered here, with the arrival of the next clock that will go on to a similar assuming that, this is process at 2? It will go in, it will process 2, will be computed. When the second clock arrives, it will be registered in the process 2, and so on it travels in the same fashion that water flows over the pipeline. After several clock cycles, you will get the output here, that means to say it has what is called the delay, or what we in pipelining process, we refer it as latency. You would notice that in order to convey it to through 10 registers you need 10 clock pulses actually. Suppose the clock pulse is 10 nanoseconds, you need 100 nanoseconds latency and at the end of which only, the output starts manifesting.

The advantage here is we can have a clock 100 mega hertz now, because the total process, all this when you add process 1 process 2 up to 10.All will add up to 100 nanoseconds as we started with traditional approach. In this case, we need to apply fresh data every 10 nanosecond and the processing order for the same, and that is how you get a throughput of 100 megahertz. This will be clear if you follow this processing order.

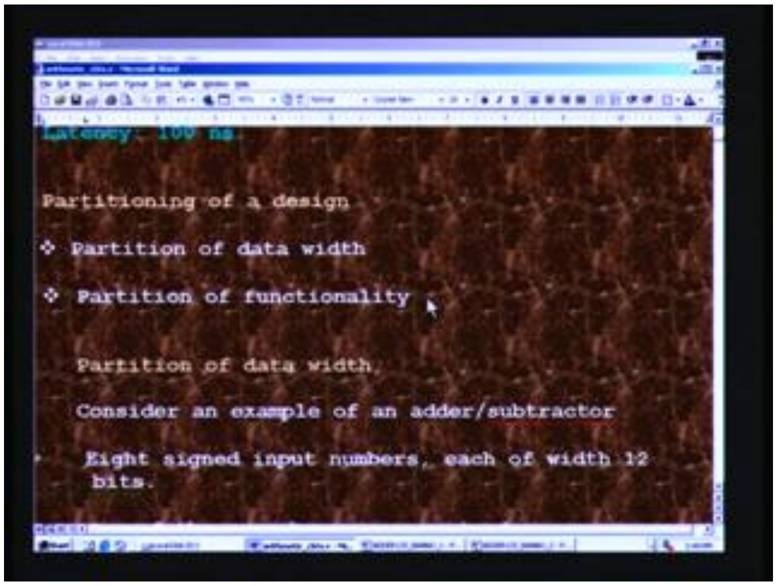
We will look at time access on the y axis here and register contents at what point of time it would register different processors. For instance at 0 time, let us say we have applied the input data 1, we are going to apply serially 1 data after another. For example data 1 data 2 in that order, we are going to apply at every 10 nanoseconds at the input here. Once you apply here, the process 1 will start immediately getting processed, and after 10 nanoseconds that process will be complete. At the end of which, we can register in the REG 1 here and that happens only here.

Because clock will strike after this only at after 10 nanoseconds, at 10 nanoseconds, it not only registers in 1 the process 1, and this is the nomenclature for the data. For example, if it is data 1 and process 1, we this must be interpreted as corresponding to the data 1. Data 1 was processed completely, as far as process 1 is concerned and process 2 will be done for data 1 at this stage here. In fact, it happens in between but the actual process result is registered only with the arrival of the second clock that is at 20 nanoseconds. When this happens, this data 2 which was processed in process 1, will be registered in register 1 here, and that is the reason why 2 is appended here. This 1 denotes the data 1, data 2, data 3. Whereas, process 1 pertains to the register 1, which sources the intermediate result of process 1.

In this fashion, process 1, process 2, up to process 10 are all stored in register 1, through 10. Naturally at register 10, if you examine the output, you will get the final result. While this is happening, you will be at every instant of clock, you will be reading a fresh data - data 1 data 2 data 3 and so on. At 100 nanoseconds we have read data 11. Simultaneously, processed process 1 corresponding to data just previous data here, and previous data of this is processed in second stage and so on it goes. At the register 10, is available at the processed output of processed 10 which happens to be nothing other than the data 1. That means, to save we started the computation here only at 100 nanoseconds this process has output here. This is nothing other than the final output, as we have seen here and this is the one.

Naturally, you start getting the first output only after 100 nanoseconds and that happens to be the latency here, and that is how the data flows via the digital pipeline so to say. There are 2 issues in partitioning, any particular design normal conversional text refer this as horizontal partitioning and this as vertical partitioning.

(Refer Slide Time: 15:38)



Latency: 100 ns.

Partitioning of a design

- Partition of data width
- Partition of functionality

Partition of data width.

Consider an example of an adder/subtractor

Eight signed input numbers, each of width 12 bits.

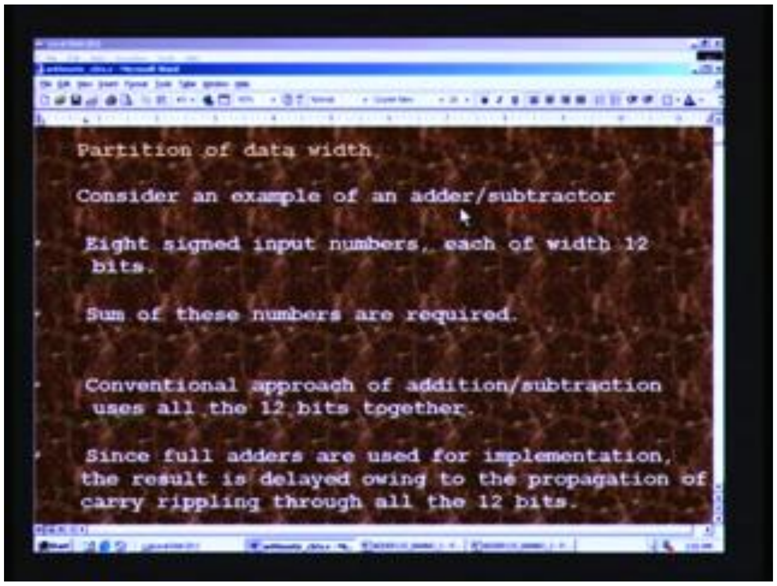
We will speak in terms of the actual functionality, say for example, we have an adder or any other data processing which processes let us say 16 bits. You can add two 16 bit numbers and if you do that, 1 it will be time consuming as we will see little later. A better way of doing it is, you just bifurcate that 16 bits into two 8 bits and then add only 8 bits is at a time, that will be faster than adding 16 bits.

You can pipeline here, you can start adding the LSB first, and then the second clock in the second register, that is process 2. You can add the MSB in this fashion you can distribute your entire width data width in this fashion. There is no hard and fast rule for such division, you have to experiment with it and then arrive at the best possible bifurcation or partitioning. This is the partition by data width. And next is partition of functionality.

The functionality is like add, subtract, multiply, etc. In terms of this addition, you have to club all similar functionality together and partition accordingly.

That is to say, add must be partition different from the subtraction, which in turn must be different from the multiply and so on. We have already described this data width. I will just read out this will be the summary of that.

(Refer Slide Time: 17:18)



Partition of data width.

Consider an example of an adder/subtractor

- Eight signed input numbers, each of width 12 bits.
- Sum of these numbers are required.
- Conventional approach of addition/subtraction uses all the 12 bits together.
- Since full adders are used for implementation, the result is delayed owing to the propagation of carry rippling through all the 12 bits.

Consider an example, of an adder subtractor. In this adder subtractor; we want to add 8 numbers that 8 signed input numbers each of width 12 bits. What we need is the sum of these 8 numbers and each notice 12 bits here, and the conventional approach of addition subtraction uses all the 12 bits together. That is 2 numbers are taken and added with all the 12 bits as a single entity. That is the poor way of implementing, because there is a propagation delay involved and which will see little here.

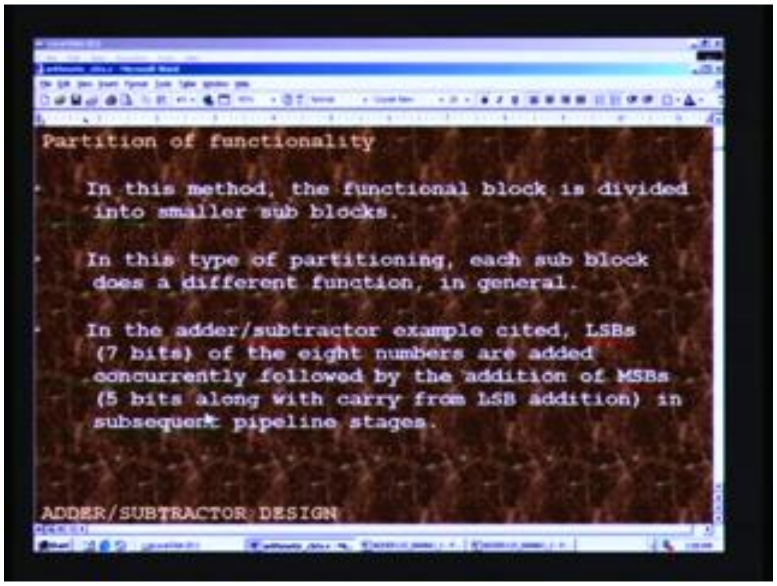
Since, full adders are used for implementation, the result is delayed owing to the propagation of carry rippling through all the 12 bits. When you add 2 bits it will generate a carry, that carry is added to the next second bit, and so on the carry propagates. That is the reason, why the whole adding process is quite delayed more the number of bits, more will be the delay and therefore the processing speed goes down.

You can bi-furcating this data to lesser values say, half of that 12 may be 6 or 7 bits and you can improve this speed. Even the usage of carry look ahead-circuit, does not help in speeding up the computations. Since a large number of gates and inputs are required in this case.

There are certain circuit is which can compute, carry far ahead of time. Disadvantage of using such circuit is that, they have lots of gates to be realized and each of these gates, have a large number of inputs. If you really do put it, test when compared to a little carry of the same width, and when you check it over FPGA platform or ASIC platform, using synopsis dc compiler. You will get figure, which is far less and compared to even the ordinary repel carry and that is the reason why, this carry look ahead-circuit does not really help and this is my practical experience.

I have examined both in FPGA as well as synopsis platform that is for ASIC, and found that ordinary repel carry it. It is faster than the carry look ahead for the reasons listed here. Then what is the answer? The answer for this problem is to divide the data widths into smaller chunks, and introduce pipelining that is what we have discussed so far. We also need to introduce pipelining in the light of the basic principle that we have considered for the pipelining. Mention may be made here, in the data width partitioning approach, all sub blocks do the same function. For example, if it is data width so if it is add it merely adds irrespective of what width, whether LSB or MSB or adding. It is basically the same function that is being done here.

(Refer Slide Time: 20:28)



Partition of functionality

- In this method, the functional block is divided into smaller sub blocks.
- In this type of partitioning, each sub block does a different function, in general.
- In the adder/subtractor example cited, LSBs (7 bits) of the eight numbers are added concurrently followed by the addition of MSBs (5 bits along with carry from LSB addition) in subsequent pipeline stages.

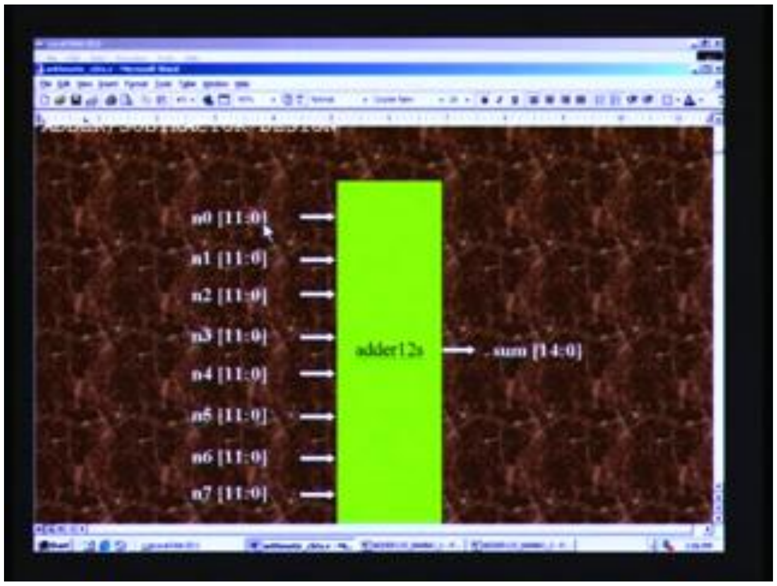
ADDER/SUBTRACTOR DESIGN

Partition of functionality, is another category this also referred to as a vertical partitioning. In this method, we partition in terms of add, subtract, multiply and so on. In this type of partitioning, each sub block does a different function in general.

This is different from the previous data width partitioning, because it does a different function such as add, subtract etc. In the adder subtractor example, cited LSBs 7 bits are taken of the 8 numbers are added concurrently followed by the addition of MSBs 5 bits, along with carry from LSB addition in subsequent pipeline stages.

This is to say, what we do is we take only 7 bits of LSB and add in the first pipeline stage and register it and in the when the that is in the first clock pulse, and when the second clock pulse arise, we take the carry of this LSB addition and then add it with the MSBs. MSB in this case will be 5 bits and that is why we save 5 bit is along with carry from LSB addition in subsequent pipeline stages we have that. The adder subtractor design is as follows. First let us have this pictorially here. We have n 0 through n 7 as 8 numbers, and each of which is of width 12 bits as you mentioned before.

(Refer Slide Time: 22:00)

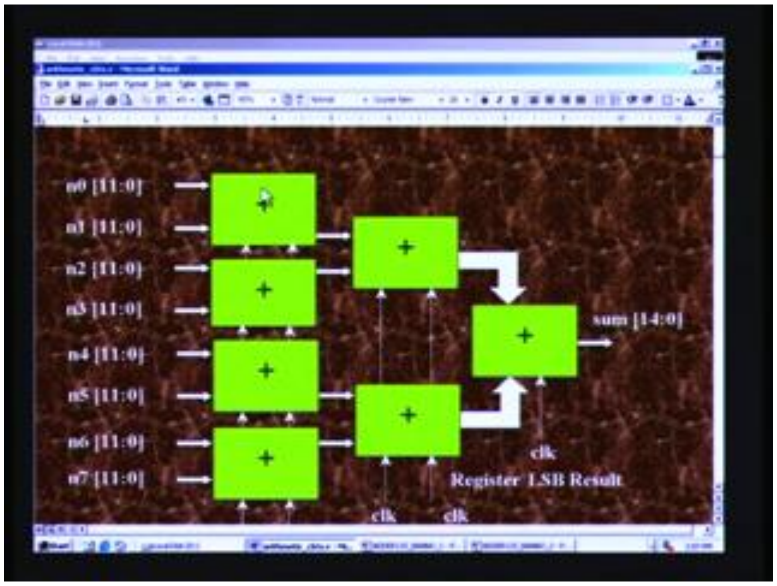


MSB is 11th bit. This is the MSB and so is the case for all these and it is basically an adder come subtractor provided you have a signed addition. You can use two complements in this which will be seeing the funda shortly. Why the width has increased, we can easily reason out there are n 0

0 through 7 means you need 3 bit. If you have to add 2 bits at a time, you naturally need 2 to power of 3 which is 8. 3 is the 1 we have to add because, 3 bits are causing this addition we will explain this little clearly later on. As a result you will get whatever is 12 bits here.

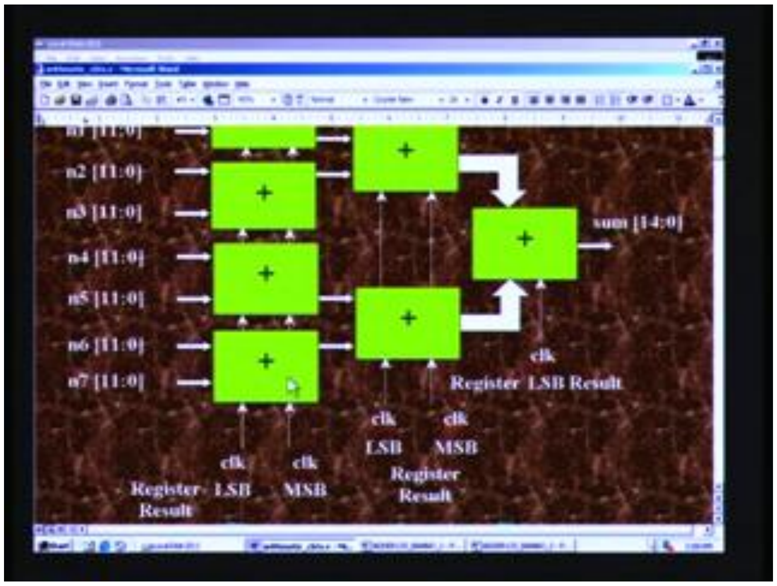
It will add by 3 bits that is what you have it here. The result will be 15 bits and there is a pipeline therefore there is a clock here. This is the partition of the design can be in this fashion.

(Refer Slide Time: 23:19)



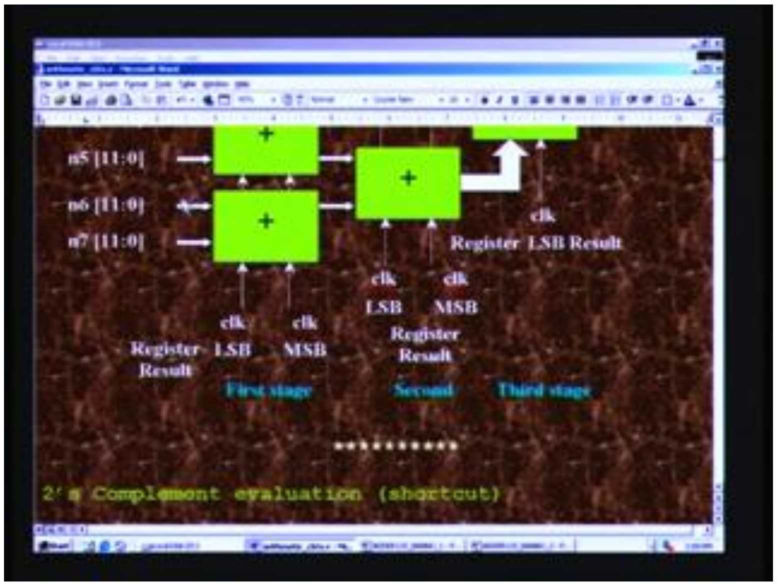
This the algorithm, evolved by this speaker for a sack of DCTQ application. Here in this case, we have 4 adders adding at 1 stroke that is concurrently, and thereby speeding up the process. Also having pipeline registers internally, so these are all the clock inputs here. That means, there is a register inside here as well as here. This register will add n not n 1 LSB alone here, and register at the first register here, When the first clock strikes and similarly in parallel, we will also be adding n 2, n 3, n 4, n 5, n 6, and n 7.With the arrival of the second clock pulse, we will be adding the MSB together with the carry generated during the LSB phase.

(Refer Slide Time: 24:17)



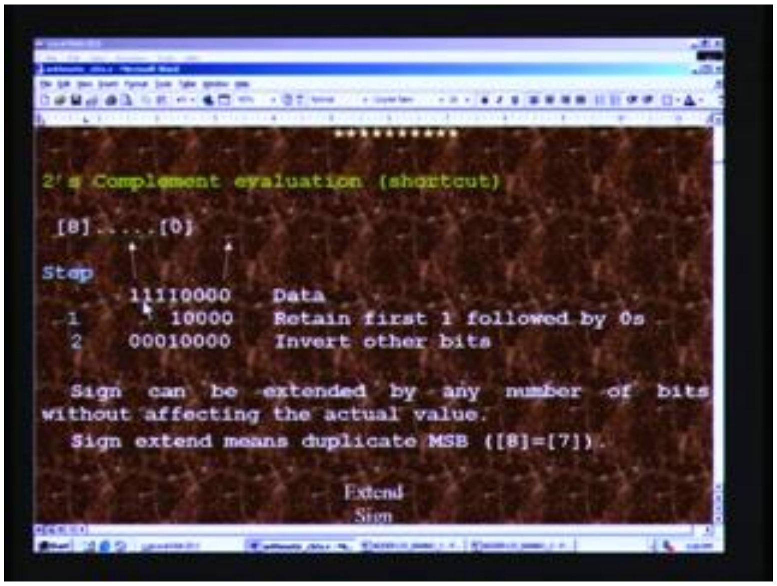
The outcome four results here, and we repeat this exactly the same thing here by 2 more adders in the second stage. This is the first stage, this is the second stage, this is the third stage, and once again there is a clock input here. At the arrival of third clock, what would have done is we would have added LSB of these 2 here. With the arrival of fourth clock, we will be adding the MSB of these 2 inside and register here. 2 outputs emerge from these here, and you can find once again add in third stage here. This addition is, there is internal register once again and the final output is not registered. Because, that is the requirement for the DCTQ application, and how 15 bit is we will reason out right from here. We start with 12 bits when you add 2 12 bits, it will generate a career that means it is totally the 13 bits will be the result. Here also it will be 13 bit is when you add 3 2 13 bits outcomes 14 bits it itself, and when you add 14 bits to 14 bits, final output will be 15 bits. That is why 14 through 0 is put here and this is how we pipeline.

(Refer Slide Time: 25:36)



You see that, every function is divided and LSB, MSB are basically performing that add operation. In this case, all of them are add. So you will not know the functionality partitioning although functionality partitioning is also implied here. LSB, MSB is treated inside; whereas, the actual functioning, we can take treated as this is 1 adder, this is another adder and so on. That is how different functionality is also partition - therefore this particular design has partitioning of not only data width but also the functionality. Let us know, what a two's complement evaluation is before we start on the code. Let us take a simple data of 8 bits here.

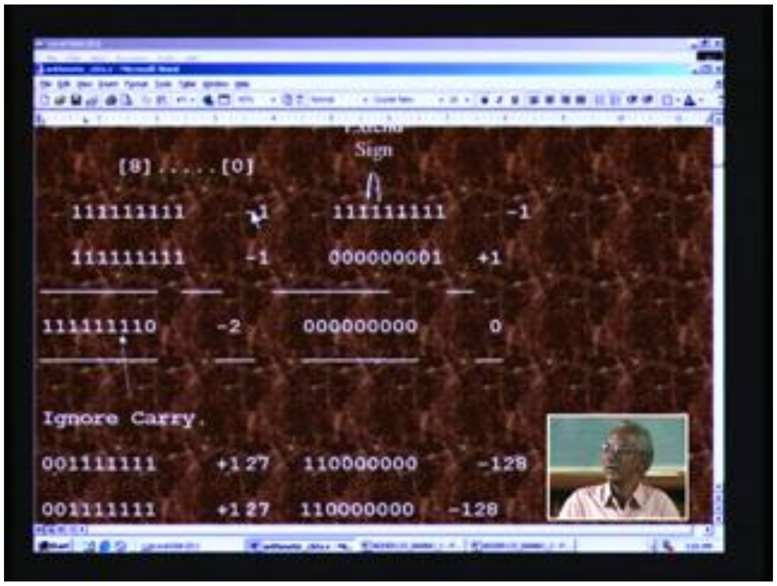
(Refer Slide Time: 26:28)



```
**********

2's Complement evaluation (shortcut)

 [8].....[0]

Step
       11110000   Data
 1        10000   Retain first 1 followed by 0s
 2     00010000   Invert other bits

   Sign  can  be  extended  by  any  number  of  bits
without affecting the actual value.
   Sign extend means duplicate MSB ([8]=[7]).
```

This is the data here, so how to evaluate 2s complement in a short cut will see here. In order to do this one, look for 1 first here, this happens with the first 1.What you do is just duplicate all these numbers, right of 1 including 1 and in the second step here, this is already done. What you do is, go to the other bits invert each of the bits and put here. That is to say although we say as 2 steps. You can at 1 stroke we can straight away arrive right from here. All you have to do is, look for first 1 one 0 0 0 put it here, then look here, invert each of these bits right here, so it is actually 1 step process. This is the easy way of evaluating 2's complement. 2's complement is nothing other than inversion of each of the bit is the result is added with 1.That can be done in short cut in this fashion. Sign can be extended by any number of bits without affecting the actual values. What we say is the MSB is a sign bit, and you can append this sign bit repeat any number of times. Thereby, the actual value will not be changed and extension means duplication of MSB see for in this case 7th bits, duplicated as 8th bit. 8th bit reference to actually 9th bit because, we start with a 0.To illustrate with examples let us consider minus 1.

(Refer Slide Time: 28:00)



```
            Sign
[8].....[0]
 111111111    -1    111111111    -1
 111111111    -1    000000001    +1
_________   ____   _________   ____
111111110     -2   000000000      0
_________   ____   _________   ____

Ignore Carry.
001111111    +127  110000000   -128
001111111    +127  110000000   -128
```

We will add 2 minus 1, so the result must be minus 2 in decimal and when you add this here, it is simple addition here 1 plus 1 is 0 and so on. Finally, a carry will emerge ignore that carry. If you drop the carry and interpret this straight away, you can easily interpret as minus 2 because this is normal addition without regard to any sign. By looking at the MSB, you can distinguish whether this is in 2’s complement. You can recognize this, as a negative number that is why you put a negative number here. If you want evaluate 2’s complement of this value and outcomes the magnitude.

Let us look at the 2s complement. We have to retain the first 1 here, and replace all these by 0 that means it boils down to nothing other than 1.It is magnitude is 1, and that is what we have put here, because the MSB sign bit being 1 for sign negative sign we put here. Same is the case here and same is the case here too, and if you take the 2s complement of this you will get this as 1.All will become what? This need not be done, we are just nearly adding here and we will see this is MSB sign bit. Minus is there and first 1 we encountered is this here. We retain this 2 bits and make the other bit 0 that is the short cut notation we already seen. This is nothing at other than all 0es 1 0 which implies 2 so the magnitude is 2 that is what we put here. Similarly you can put for all these things minus 1 one so add the 2 it is mere simple addition without regard to any sign. If you deal with 2’s complement it is a plain addition of binary numbers here. You can see for 1 20 7 1 20 7 the final result is this. You can work it out yourself.

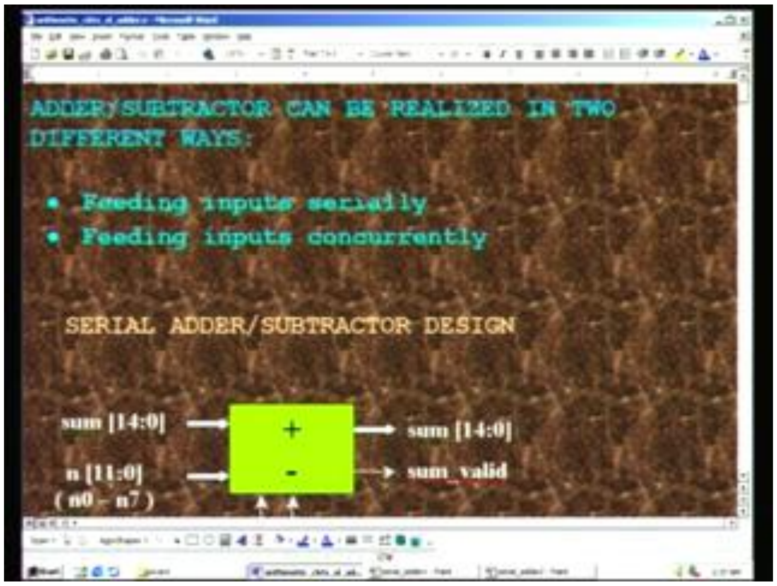
So is the case for, minus 1 20 8 1 20 8 and notice that for this 2’s comp this MSB’s minus 1 minus here. If you take the 2’s complement, it turns out to be the same thing which is nothing other than 256, because, this is in 9th bit here 4 plus 4 8 here. 256 here, and without the sign extension the MSB will be mistaken as a negative number for high positive values, such as plus 254.

What we mean here is this 254 if you add here, you have already added here. What will happen, it will result in 8 bits result, it will be 8 bits, see for example 4 here 4 8 here, we have extended this sign therefore, there is 1 more bit here. We have seen that, by extension no value is changed and in fact, we can have a look at this. I will come to that after explaining this, so what we said here is without sign extension the MSB that is the 7th bit, if you just take 8 bit number this is the MSB.

We will be mistaken, as a negative number for high positive values such as 254 why we say is, if it is 254 the number is here. If you see this is the 7th bit as the mentioned here. Instead of interpreting plus 2 as plus 254, the system will be reporting it as minus something else. That is not a desirable thing and hence, the need for extension of the MSB to the next location and that is why we have a sign extension that is the meaning here. We will prove that by one of these examples.

For example, let us see here we will take minus 1.In this case, you have 9 ones here. What if I have just 2 ones alone? Will it still be minus 1? The answer is yes. I can duplicate the MSB which is the sign bit here in this case. Any number of times I can duplicate. The proof is very clear - even if it is infinity the last will be 1.Therefore we recognized as minus here and now we start with this first 1 we retain this 1 and do I mean? Invert all other bits right up to infinity. Magnitude naturally is, 1 for indefinite number of and there and therefore it is a negative value and as well as the magnitude 1. This proves that, you can by extending the sign bit which is nearly duplicating the actual sign bit to the 1 more bit extra. That is how we have added and seen the results, we will go into verilog code for adder subtractor. We wish to add 12 bit numbers 8 of them and each of them is a sign number.

(Refer Slide Time: 33:16)


ADDER/SUBTRACTOR CAN BE REALIZED IN TWO DIFFERENT WAYS:

- Feeding inputs serially
- Feeding inputs concurrently

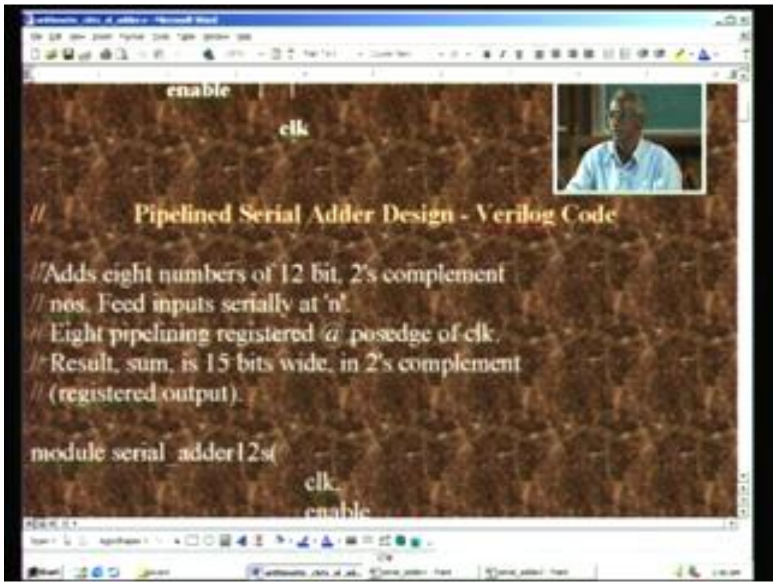
SERIAL ADDER/SUBTRACTOR DESIGN

In essence, what we are doing is add or subtract and this is our goal. This can be realized in 2 different ways. One is we can feed the input serially and one after another. For example, we have 8 inputs. Let us say here, in this block diagram which does the addition or subtraction. We have only in serial adder, we have only 1 set of input and you will have to apply one after another here, at every positive edge of the clock.

The synchronous of the clock, you will have to apply different numbers starting with n not. First we apply n not, then n_1 n_2 and so on right up to n_7, 8 of them we apply one after another. This is what is implemented in a serial adder. In contrast to this, we have parallel adder in which case we can apply all the 8 inputs at a time at one stroke. Of course, with reference to a positive edge of the clock and these are all the two things that we have seen. That is, a serial adder that 1 possibility, another is concurrent or parallel adder. If you want very high speed processing naturally, you go in for this concurrent adder, where chip area is of great concern.

We go for a serial adder and let us have a look at this serial adder. We have basically 2 numbers added here, and in fact all the 8 numbers are going to appear through this input. The accumulator result, will be available in register internally. There is a register, and that is what output here as sum. Since we have 8 numbers of 12 bits each, and we need to have 15 bits totally, and difference note is that the 3 bits between the input and this 1, this is clearly arising on account of the fact that we have 2 to the power of 3 number of inputs. That is the reason why three bits extra are coming here, and when the sum is valid, it is indicated by another binary signal here, and all this transaction will take place only, when enabled that is when enable is high. We can write a verilog code for this particular application it is basically a pipeline approach, although it is pipeline we are not going to use very many registers but single register will do the job.

(Refer Slide Time: 35:54)



```
            enable
                        clk

//            Pipelined Serial Adder Design - Verilog Code

//Adds eight numbers of 12 bit, 2's complement
// nos. Feed inputs serially at 'n'.
// Eight pipelining registered @ posedge of clk.
// Result, sum, is 15 bits wide, in 2's complement
// (registered output).

module serial_adder12s(
                        clk,
                        enable
```

In addition of 8 bit, numbers are accounted at here as I mean comment as usual and so pipeline serial adder design. Of course, verilog code is what we are talking about. It is in 2s complement we will cover this 2's complement in greater depth, when we take up the concurrent adder prior to taking the concurrent adder.

The goal is to add 8 numbers 12 bits each and it is in sign. You have to seat, feed the inputs serially at n input that we have seen earlier. It has basically 8 levels of pipelining, and this registering will happen, at the positive edge of the clock. Outcome result, there are two ways of expressing the result, either as a sum or as a result. The difference we will see what exactly there when we as we progress further. Naturally, it will have to be 15 bits width and in 2s complement and this is going to be a registered output. This is the module decoration for the serial adder and we give apt name here, because it is serial adder we give this name and it is 12 bits signed, that is why this name has been given.

You have to list all the i/o's here and here according to the block diagram, we have already seen clock enable and n are the 3 inputs, and sum valid and result are the 3 outputs. Sum and result are primarily the same output, the difference is that, the sum will be valid only for 1 clock pulse duration.

Whereas, this result will be valid for 9 clock pulses and that is the difference between the two. This is in fact, while writing the code you do not write twice like this, either you restrict to sum here or result, you do not have to put both.

Just for simplicity sake, when we see the waveform to aid us, to understand both have been separated out like that. Once you have declared the module, you have to declare all the i/o's here, inputs are declared here, and clock enable and n are the inputs, and note that, these are all single bit signals. Similarly, the output this is a single bit signal which is indicating when the sum is valid. Final result is here, and the same result is also available as sum. This is valid only for 1 clock pulse duration and the number of bits that we are taking, reckoning is also indicated here.

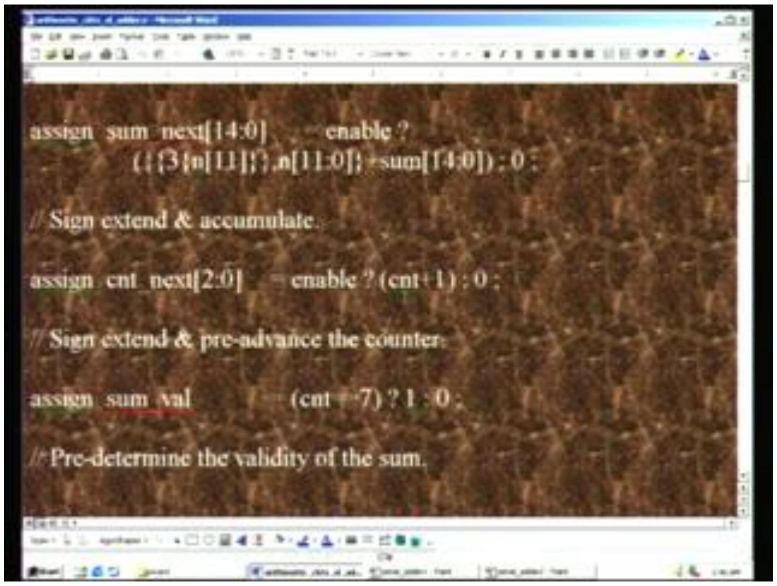
The wire as you know is actually the assign statements outputs, these are all the intermediate outputs that we need prior to registering later on. You have a 1 to 1 correspondent between sums, next which will do the accumulation of the different numbers. For example, n not emerges first then this will be n not after summing, and before prior to that 1 we need to initialize this to 0. We will see, that will be done only in the some case which is registered case, as the progress we will see the difference. We also have to keep track of the number of inputs, that we have already processed.

For example we are inputting only n not, and then we need upgrade the counter and it will go right from 0 1 2 3 and so on up to 7. Therefore, we need 3 bits here, in order that we may keep track of number of inputs that we feed here. In this particular design, it is n not through n 7 which is actually 8 numbers and therefore 3 bits adequate. We also need to declare sum valid, which is basically the same as this sum valid and with note the difference in spelling here. This is an assign statement and that is why it is declared as wire here. We also finally, have need the computer sum and that is 15 bits as we have already seen. It is going to appear in the always positive edge of the clock block and therefore it is declared as REG. So is the case with the counter, that is also in a positive edge of the clock and this is only 3 bits in width.

This is also a register sum valid, which indicates when the sum is valid and this is going to be valid only for 1 clock duration. The result is same as the sum, except that it is going to be valid for 9 clock cycles. When we do the computation of next set of numbers of I mean consisting of another 8 inputs then we will be updating this. This is the very first statement assign statement in order to compute the sum next. This is done by using a MUX which you are already familiar and enable is the one, which will enable the computation of the sum actually.

To start with, when we have n not input, it will appear here and sum is initialized to 0, which will notice little later on. To start with, we should not forget to clear the sum, otherwise we will be adding unwanted number. Therefore, 0 plus n not will be n not and this will be assigned to the sum next, and this will be registered only with the arriving positive edge of the clock, next subsequent to this. When enable is not high, actually 0 goes through the sum next. That is what we want. This will be registered during the always block positive edge of the clock and similarly we need to update the counter, and for that we have a count next here. This is like once again derived from the enable signal if it is 1 then this also a MUX realization.

(Refer Slide Time: 41:52)



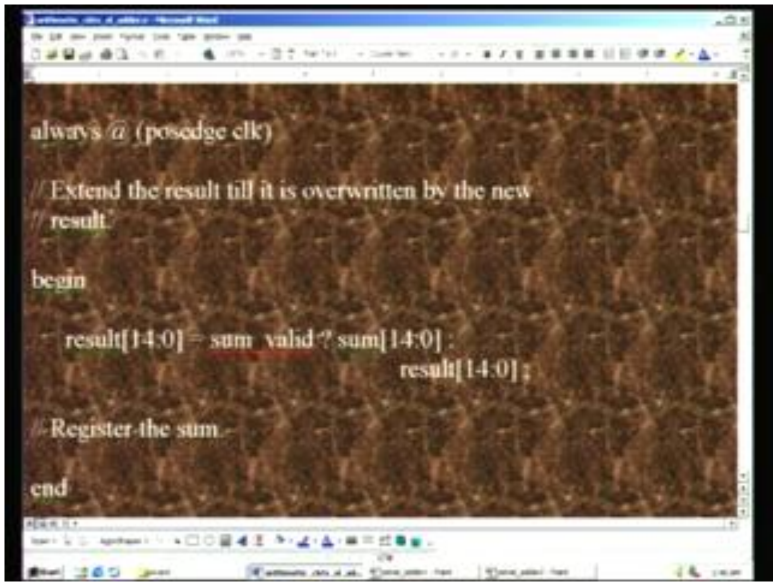
```
assign  sum_next[14:0]      = enable ?
            ({{3{n[11]}},n[11:0]}+sum[14:0]) : 0 ;

// Sign extend & accumulate.

assign  cnt_next[2:0]      = enable ? (cnt+1) : 0 ;

// Sign extend & pre-advance the counter.

assign  sum_val            = (cnt == 7) ? 1 : 0 ;

// Pre-determine the validity of the sum.
```

We increment the counter by 1, and then assign it to the count next, and this is basically a pre-advancing of the counter or pre increment, and otherwise you initialize to 0. The next statement is the sum valid and this is also a signal, which is ahead of the actual sum valid and which will be registered later on.

This is assigned to using an assign statement and only when the counter is 7. All the 8 inputs are already fed and when n 7 is arriving. At that point of time, we make the sum valid because with the subsequent clock, the entire computation is over that is after 8 clock cycles. Only when the 7 is encountered then this 1 is assigned to the sum value, otherwise it is going to be 0.For all the other clocks right from first to 7 clocks naturally, this 0 will be forced to this 1. Only with the arrival of the count value 7 it will be made 1.

Next is the always block positive edge clock, we are taking the final sum and merely assigning here. This is what is already computed and we merely assign to the final output which is sum, and similarly the count next which was pre incremented earlier is assigned to this count-register. This is for advancing the count. Naturally, the sum valid which assigned earlier is actually output to the register and here we have one more for the result.

(Refer Slide Time: 43:38)

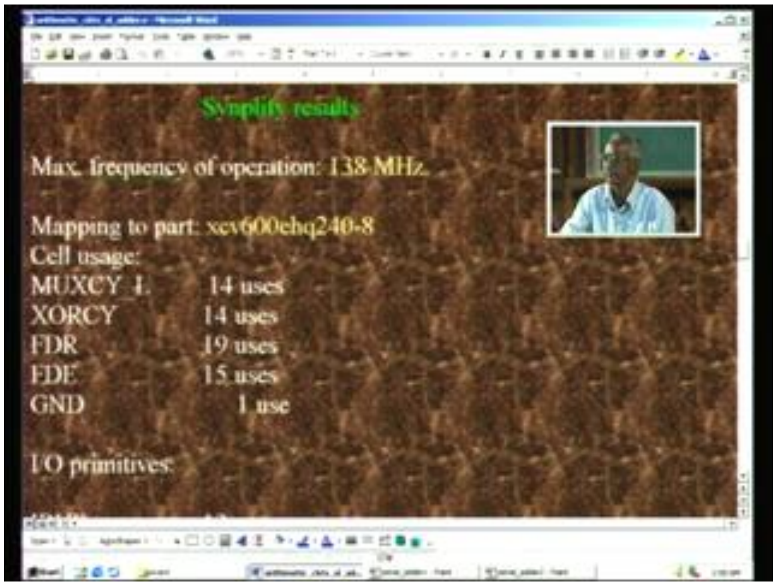


```
always @ (posedge clk)

// Extend the result till it is overwritten by the new
// result

begin

   result[14:0] = sum_valid ? sum[14:0] :
                                    result[14:0] ;

// Register the sum

end
```

This is not really required in actual design, and this is only to make it make things clear, it has been put. In this case, what we do is this is a parallel circuit to the other always block which we have seen, and we merely assign the result here, and depending upon the sum valid. Sum valid will be 1 only with the arrival of the positive edge of the clock, which we saw here in this particular condition. This is positive edge of the clock here, and subsequent clock only that will be reckoned here and when that happens, it is merely to delay the final result by one clock pulse. This actual sum which was registered in the previous clock edge will be assigned to the final result. If not, if sum is not valid that means, it is not going to be valid for a count right from 0 through 7 and it will be valid only for 7. Only for one clock duration, this sum will be assigned here which is the final adder up values of sum of all n_0 through n_7.

Otherwise, what will happen? The sum is not yet ready. What we do is, merely whatever was preserved the previous cycle that is for previous set of inputs added that only will be assigned here. If sum is not valid, is this clear to you? I hope so. This completes the verilog coding, we are not going to have a look at the test bench, because quite a simple thing. We will have look at the concurrent adder subsequent to this one. Before that, we will just see summarize the simplified results. The result of synthesis it operates at 138 mega hertz and this is the device used here XCV 600.

(Refer Slide Time: 45:29)



Synplify results

Max. frequency of operation: 138 MHz

Mapping to part: xcv600ehq240-8
Cell usage:

MUXCY_L	14 uses
XORCY	14 uses
FDR	19 uses
FDE	15 uses
GND	1 use

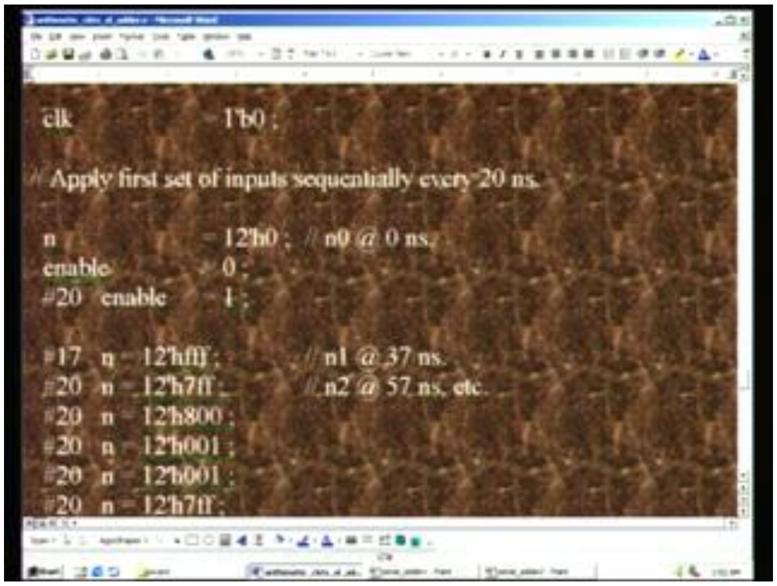
I/O primitives:

Various MUX and other primitive cells are listed here including the buffer out. The number of LUTs it consumes very little. For this device it is 0 percent that means, fraction of less that 1 percent. Place and route results for this are here, and you notice that gate count is quite low of the order of 464 here. JTAG gate count is also quite high but for higher sizes it will come down. The frequency now it reports is much higher and the test bench for the serial adder.

I said that I will not cover this, that is also there, will quickly cover the test bench here. We need to initialise - 10 nanoseconds as a half period for the clock to run. This is the standard practice which we have been following throughout the course here. We have to include the design which is the serial adder, which we have just now seen. The test bench will have to be declared as a module here. That is what we are doing here.

Consolidate all the outputs here and so we merely declare all outputs. Then clock and enable and input n, are all to be declared as register in the case of a test bench, which we have already seen a number of times earlier. We need to invoke the actual design, which is serial adder 12s. We instantiate just one time and list calling port by name, and next thing is we initialize and apply the actual stimulus or the actual inputs. For example, clock is initialized here and first input n not value is 0 and we are going to enable only at 20 nanoseconds here, and then apply all the other inputs.

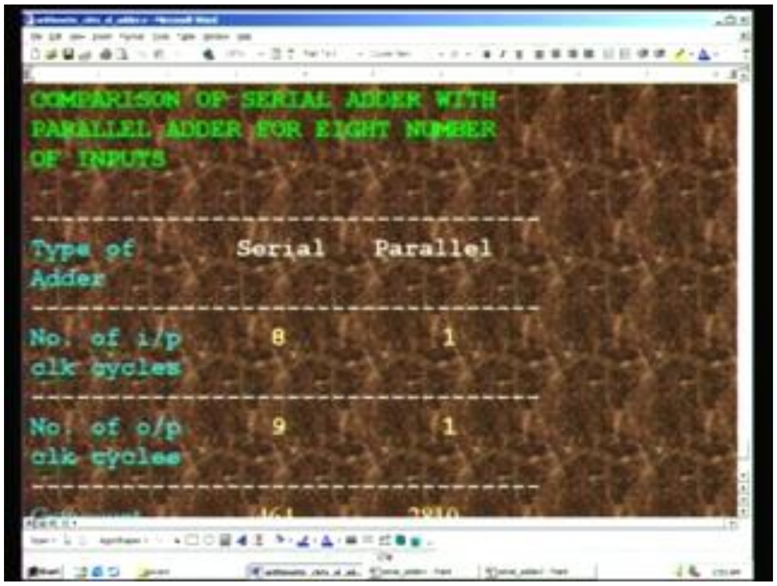
(Refer Slide Time: 47:31)



```
clk             = 1'b0 ;

// Apply first set of inputs sequentially every 20 ns.

n               = 12'h0 ;  // n0 @ 0 ns.
enable          = 0 ;
#20  enable     = 1 ;

#17  n = 12'hfff ;          // n1 @ 37 ns.
#20  n = 12'h7ff ;          // n2 @ 57 ns, etc.
#20  n = 12'h800 ;
#20  n = 12'h001 ;
#20  n = 12'h001 ;
#20  n = 12'h7ff ;
```

This is n_1, n_2 and so on. In this order, n n_1 is minus 1 notice this one. This is in 2s complement and all these things are in different values. The last value is just a dummy thing, because, last n7 value is this 1. n0 value is this one. Every 20 nanosecond we apply and the data we offset. As we have done before, that data is always ready when the clock strikes. After this, we disable for a while for just 1 clock duration and then enable again, and apply the second set here of numbers here. You can just remember this one here, and finally after applying all these numbers n_0 through n7.We disable here, and after about 100 nanoseconds we stop finally. Then we run the clock by toggling the clock here, which is once again the same thing that, we have already seen this runs at 50 megahertz. This is the end of the test bench. And we will have a quick glance of what a serial comparison between serial adder and parallel adder.

(Refer Slide Time: 48:38)



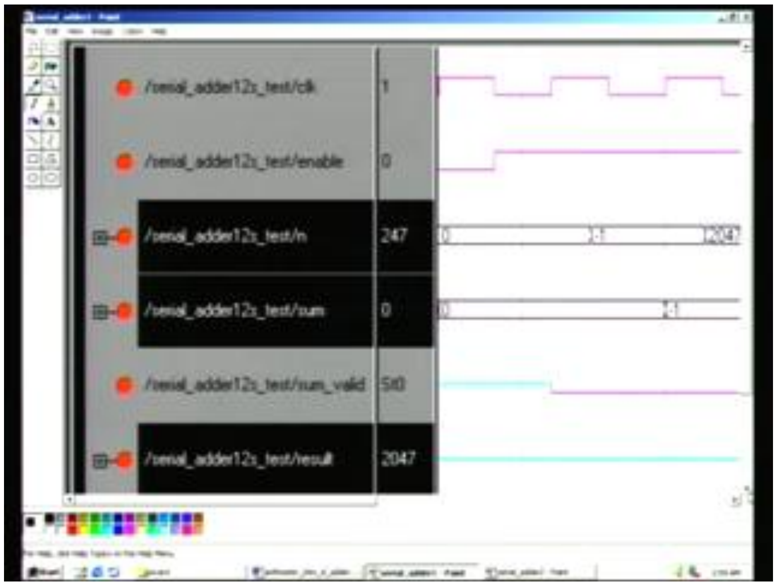
COMPARISON OF SERIAL ADDER WITH PARALLEL ADDER FOR EIGHT NUMBER OF INPUTS

Type of Adder	Serial	Parallel
No. of i/p clk cycles	8	1
No. of o/p clk cycles	9	1

For a serial adder which we have already considered, you require 8 clock cycles for inputting the n0 through n7. Whereas in parallel in 1 stroke 1 clock pulse you can do that.Similarly, the output will take 9 clock cycles, because we have to after having input to all the 8.We have to make the enable low for 1 clock pulse, then only the sum will be cleared and that is the reason why 1 additional clock cycle is required. Naturally, the serial adder is much slower of the order of 9 times slow. In other words, parallel adder is faster by 9 times and at the price that you have to pay, you have to pay in terms of chip areas 6 times what you have here.

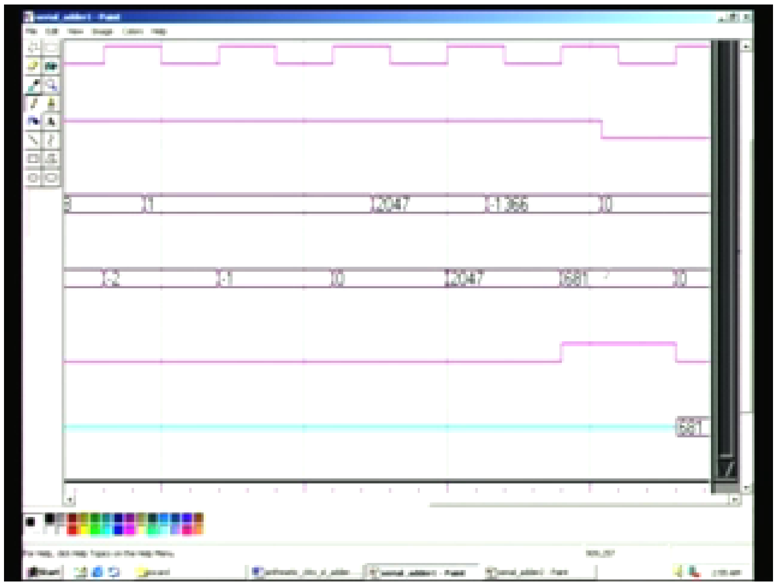
In actually, if you see the parallel concurrent adder which you are going to see next, we will see that basically 7 adders are required.Although 7 adders are required actually the chip count is only 6 times here, because optimization tool is being a very efficient job there. So also, the JTAG count and maximum frequency operation here is 174, in parallel, which we will be seeing subsequent to this at almost close to that. This is not a handicapped and we will see the waveform here.

(Refer Slide Time: 49:54)


/serial_adder12s_test/clk 1
/serial_adder12s_test/enable 0
/serial_adder12s_test/n 247
/serial_adder12s_test/sum 0
/serial_adder12s_test/sum_valid St0
/serial_adder12s_test/result 2047
0
-1
2047
0
-1

Before we conclude this serial adder, and here you can see that clock enable, n sum are all applied here. This is the final result here and we can see this. This is the first set of inputs. 0 is applied then minus 1 20 47 and so on. You remember minus 1 was triple f there and you see here, is the final result. 0 minus 1 is minus 1 minus 1 plus 20 47 is 20 46 and keep on adding like this. Finally, for the entire set you would have done. The last 1 final result is 681 for the first set of numbers. You can see that here.

(Refer Slide Time: 50:40)



The final result is valid only for the 681. We has also seen a result that is being registered 681. It will be valid for 9 clock cycles, which will be seeing in the next waveform here. This is the second set of inputs that we are going to apply and here what you have is 100 2 100 etc. We have applied you remember that 100 then 2 100. The result is, 300 then plus 300 and then 400000 and thousand 500 and so on, then 1000 600 here, 1000 800 then finally 20 47.

Once again this goes high some valid goes high and final result is available at 20 47 you note that one. If you take this timing it is coming at 350 nanosecond here; whereas, in the first one what we have seen is basically it is 1 7. This is corresponding to 170. 350 minus 170 is 180. Each is 20 nanoseconds -that means 9 clock cycles it has taken. This proves that serial adder is 9 times slower when compared to the parallel adder or concurrent adder which will be seeing next.

Thank You.