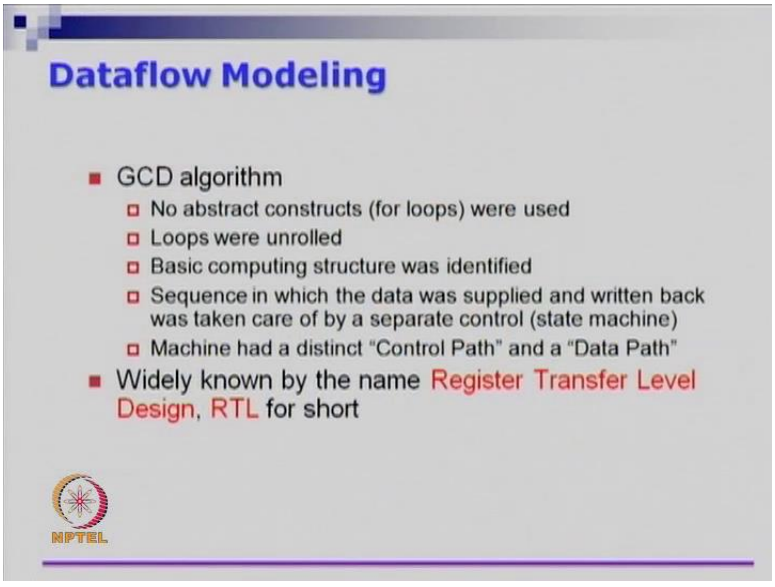
Digital Circuits and Systems
Prof. Shankar Balachandran
Department of Electrical Engineering
Indian Institute of Technology, Bombay
And
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Module - 47
Pipelining (and Verilog)

Welcome all of you. So, we are at module 47 of week 8 and this will be the last module of this week. And as I mentioned in the first video of this week I said, we will have one more week of lectures to cover some of the topics, that is students have given feedback about specifically related to number representations and arithmetic circuits. So, in this concluding module of this week, what we will do is, we will connect all the things that we have seen so far.

In the past 4 modules, we looked at what pipelining is, what interleaving is and so on. But in the last module we looked at what are the different intricacies, when we involve blocking verses non blocking statements. So, what I want to do in this class is, connect pipelining to the way we model circuits in Verilog. So, this will be the concluding module for this week.

(Refer Slide Time: 01:13)

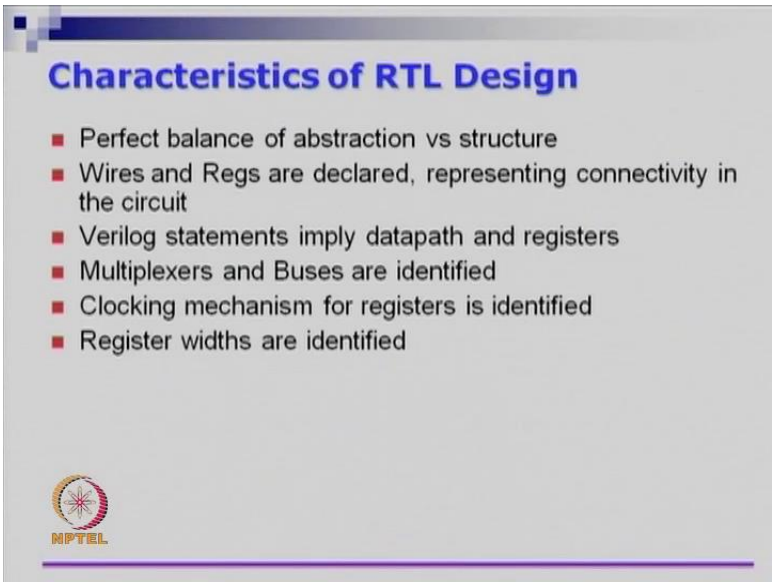


So, let us take a step back and look at the GCD machine that we designed. So, the GCD machine that we designed was actually quite nice and some way. So, first of all, we actually used fairly simple concept. So, we started with the algorithm and then we went and did what is called unrolling the loop. So, there was a while loop in it and we figure out that, we need a structure, which will keep the loop iterating. And we had a state machine, which took care of how things were done.

And finally, what we did is, we said we identify the basic computing structure and we had a sequence, which was running through it. So, something like that, where we take a problem and divide that into a distinct data path and a control path, such a design would be called RTL design. So, this is something that you may see in job posting and what not. So, RTL design essentially means register transfer level design.

So, the implication is that, you actually start looking at basic modules and these basic modules in turn are connected. So, you have basic registers and you have various arithmetic units and logic units and so on, which are all connected together, which form the data path and a control machine or a state machine which controls the sequence of these events. So, such a level of design is called RTL and we are going to see RTL models for pipeline circuits in this lecture.

(Refer Slide Time: 02:43)



So, the chief characteristics of RTL design is that it is a perfect balance between abstraction verses structure. So, we could go all the way down and do various things, but

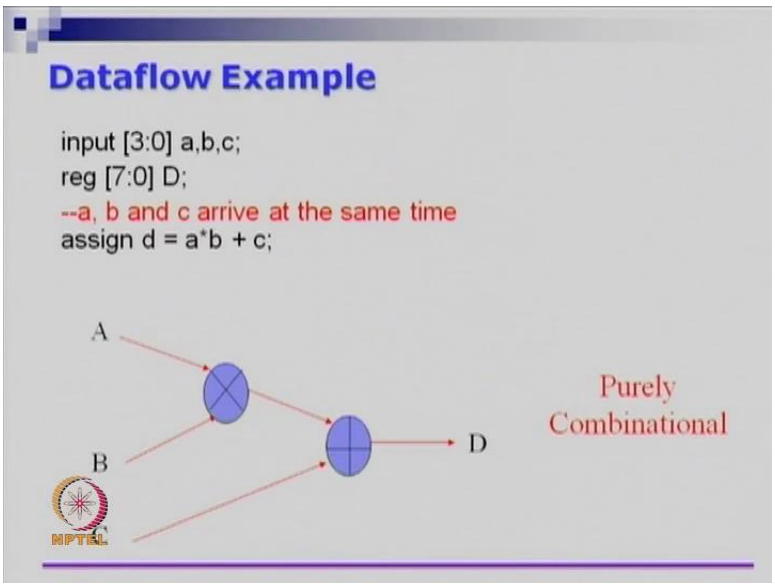
then if you notice, the subtracted that we designed and the comparator that we designed was not in terms of basic gates. So, we use the operator minus for subtraction and for comparison, we just compared 4 bits with 4 bits and so on.

So, such a level of design, where we do not bother with the finite details of the actual hardware, but bother with the abstraction or the program is intent. So, this is the perfect balance that you get from RTL design. So, what we do in RTL design is, we declare the wires and regs, so that we know all the connectivities that we need and we have Verilog statements that imply the data path elements and the registers.

We also identify all the multiplexers and buses that are needed. So, even the GCD example if you remember, we had a two multiplexers to choose, where the inputs are supposed to come from. We decided a clocking mechanism and we also identified the register width. So, by register width I mean, so in the GCD problem, we said we will look at GCD of 8 bit, two 8 bit numbers.

So, the implication of that is, the multiplexer, the subtractor, the comparator, all of them were supposed to be 8 bits. So, once you identify the structures, we need put those parameters in and take it all the way down. So, these are the chief characteristics of what is called the RTL design.

(Refer Slide Time: 04:11)



So, let us look at a various examples of a data flow modeling. So, let us look at this picture here and the corresponding Verilog description. So, I am not providing the complete Verilog description, we are only seeing the part of the Verilog description. So, let us say we have input 3 colon 0, a, b, c and we have a reg 7 colon 0 D and we have assign d equals a into b plus c.

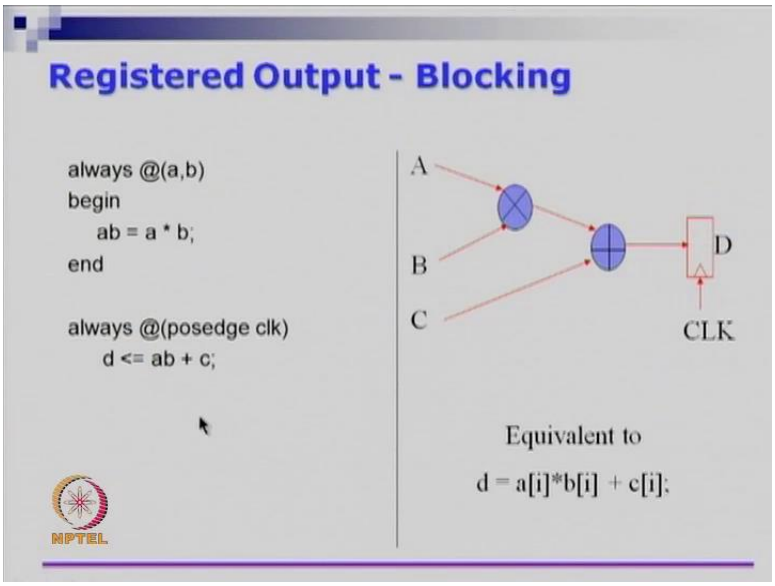
So, what this would mean is, we are looking at d, being assign a times b plus c. So, a times b is 4 bit number multiplied by another 4 bit number, the result is 8 bit number or to which we are adding a 4 bit number, so the result is a 8 bit number. So, d is actually 8 bit number and if you wrote something like this, so forget the declarations. If we had anything like assign d equals a times b plus c.

Then, the implication is we have A, B, C which are the inputs and D is the output and A times B, so there are tools which are called synthesis tools, which can look at this line of code. Understand that, star means multiplication and actually throw in a multiplier there and also understand that, plus means in addition and throw in an adder there. So, this plus is adder and this cross is multiplier, you could infer something like this by writing a line of code like this.

So, you could also go and write this whole thing in a structural manner. So, by that for the adder, you may actually use a ripple carry adder, for a multiplier, you may come up with your own way of doing multiplication of two numbers and so on. But if you do that the programmer is intent is buried inside lots of lines of code, whereas this is very clean and many tools can immediately understand that, you are looking for multiplication and addition.

So, forget some of the definitions and so on. So, there is some mismatch in capital letters, this is lower case letters and what not. So, the intent is not about the syntax here, what about what is the inference from the Verilog that we are writing. So, for instance all of these must have been small case letters and what not. So, let us ignore that for a while. So, this is called data flow.

(Refer Slide Time: 06:29)



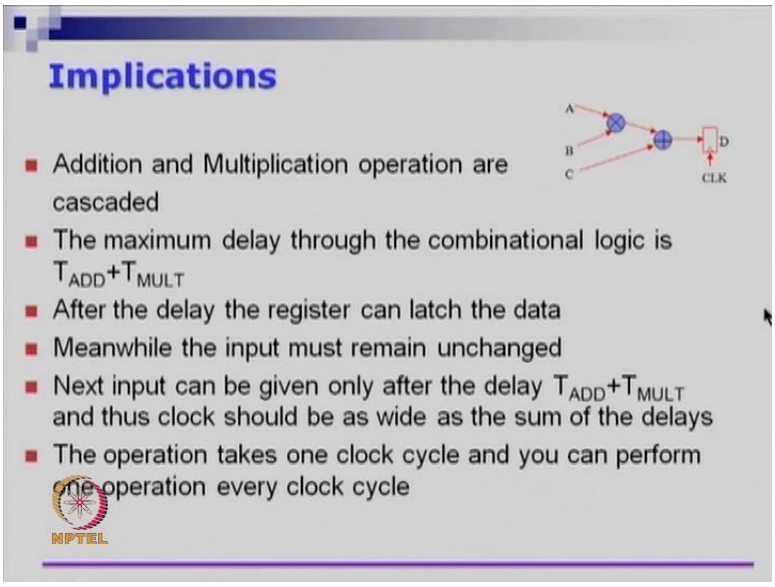
Now, we could do something else. So, let us look at the thing on the left side, always at a comma b, a b is a star b, always at posedge of clock, d is a b plus c. So, if you notice this a and b are inputs and we did always at a comma b, which means any change in any of the bits of a comma b, will result in this always block begin triggered, which means, the multiplier, if there is any change in input of the multiplier, there will be a change in the output of the multiplier.

So, this star means multiplies, so a b is a times b in phrase this part of the circuit, where you have A and B as inputs and there is a multiplier, this wire here is essentially A B. And what we have here is, here we have d equals a b plus c. So, this is A B added to C should give D and the reason why D is a register is, because we have always add posedge of clock, which means this D will be a 8 bit register and there are 8 lines going out of the adder, which all get register in D.

So, this is the hardware that will be inferred if you write code like this and you can see that, this is combinational in nature and this is sequential and we have adopted all the policies that, there is a blocking equal to here and a non blocking here and so on. So, it is fairly easy to understand what this is doing, but more importantly from a… If you want to think about it as a software, what you are actually doing is, d of i is a of i into b of i plus c of i.

So, you are taking inputs which are all given at one instant of time; that is what this i means. So, this i is not subscript within a, this a of i is time. So, at time i whatever a, b and c; if they are all kept stable, then it takes those inputs A and B multiply, added with C and that will go as input of this flip flop here. In the next cycle, this value will get in to the register, when the next clock pulse comes in, it will go to the register. So, the hardware inferred is essentially this.

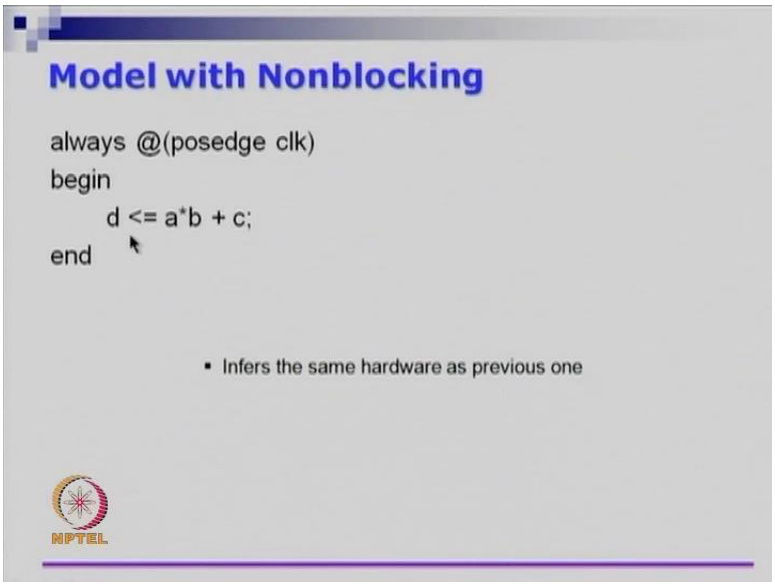
(Refer Slide Time: 08:49)



So, the implications are that, addition and multiplication are cascaded, because we have one combinational block followed by another combinational blocks. So, we have those two cascaded and if I want to look at the rate at which I can give these inputs, this is from global inputs to flip flops. So, this is input to flip flop, the longest path in that will be through the multiplier and the adder. So, the maximum delay through this combinational logic is propagation delay of addition plus propagation delay of multiplication.

So, after that delay the register can actually register the data, but meanwhile, you want to keep the inputs stable, so that this will not change or upset this flip flop. So, the next clock pulse, we have to give only after T ADD plus T MULT. So, remember, we did this timing sequential circuit timing analysis in probably week 5, I believe. So, this operation takes one cycle and we can give one set of inputs in every clock cycle. So, all of that is probably very clear to you.

(Refer Slide Time: 09:53)

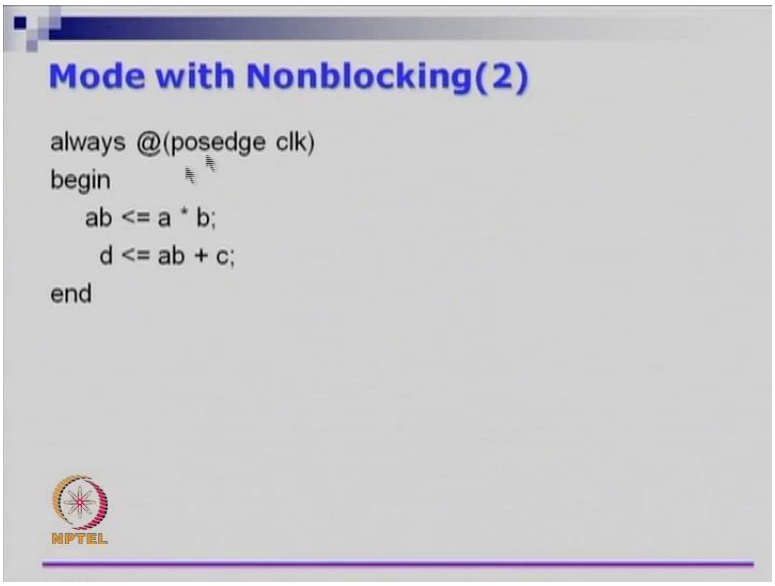


Let us now look at the different kind of a statement. So, here what we have done is, always at possible of clock, d equals a times b plus c is put inside the always statement. So, you could write something like this, this is still in some sense relatively straight forward; you have a times b plus c. So, if you write a single statement like that, Verilog both the simulator and the synthesis tools.

So, simulators go and show you waveforms, so that you can go check everything is correct and so on and synthesis tools actually take this infer gates for you and can actually spit out a net list. So, d equals a times b plus c is a template that, these tools can understand and it can infer the same hardware as the previous one. So, this is an essentially a multiplier followed by an adder, followed by a register and you can start seeing, why it should have a register and why this a into b itself is not a register.

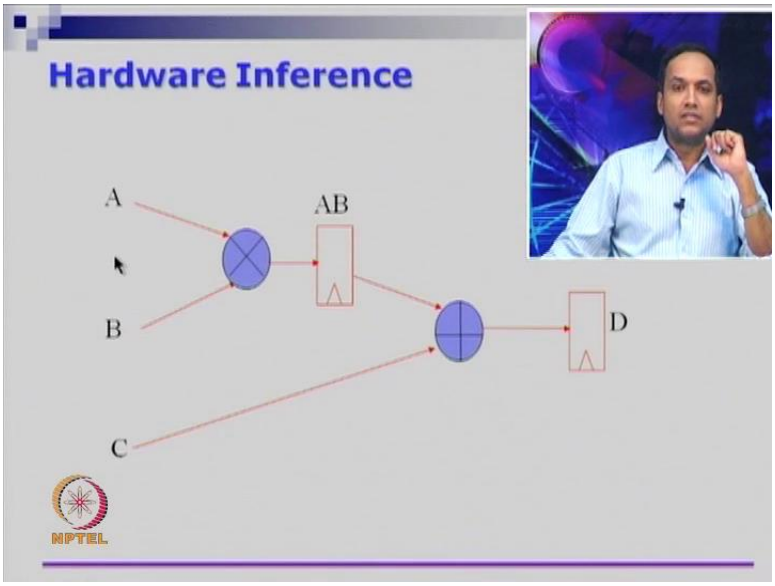
So, if you look at this one, only the result of this addition is registered, not the result of the multiplication followed by the result of the addition. So, this requires only one level of circuit and ((Refer Time: 11:07)) what we have here is essentially a one stage pipeline and remember one stage pipeline is no difference from combinational circuit in terms of overall latency. You still have to wait for one full clock cycle.

(Refer Slide Time: 11:20)



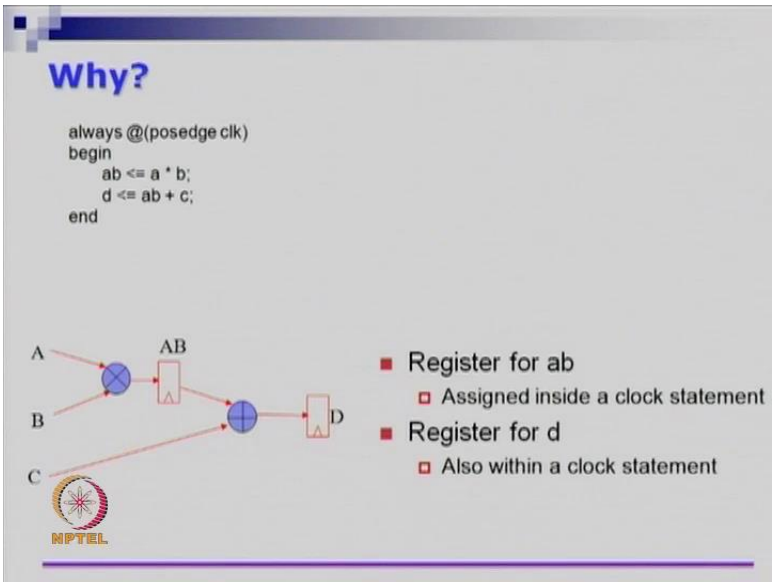
Now, let us look at another model. So, this model is always at posedge of clock, a b is a times b, d is a b plus c. So, let us say we wrote something like this, a b is a times b and d non blocking equals a b plus c. So, take a while and think about, what it should really being inferring first. So, if you notice a b and d are on the left side and both of them are in posedge of clock, which means a b will be registered, which means the result of the multiplication will get registered and the result of the addition will also get registered. So, now go and think about, what will be the overall circuit. So, once you have done with it, comeback to this.

(Refer Slide Time: 12:04)



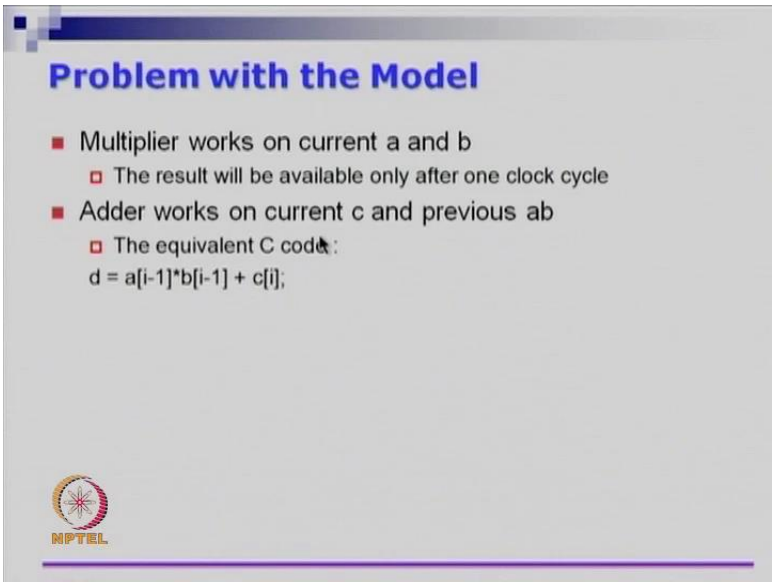
This would be the hardware, A and B would be multiplied and that would be stored in a register A B and C would be multiplied with the register value of A B and that would given a, give a D. So, the first thing that you have to, now go and think about is, what is this circuit really doing, is it going to be correct and if it is not going to be correct, what is the issue with it. So, a quick clue is that, this is not correct, go and think about one, then come back to the next slide.

(Refer Slide Time: 12:39)



So, let us see why it is not correct. So, let us see a b equals a into b and d equals a b plus c. The first thing is, there is a register inferred for a b, because it is inside posedge of clock, there is also registered inferred for d, because there is a posedge of clock. So, you have two registers and this multiplication will give you this and adder will give you this. So, it looks likes whatever code we wrote, the hardware that is inferred is actually correct, but is this capturing the intent of the programmer.

(Refer Slide Time: 13:10)

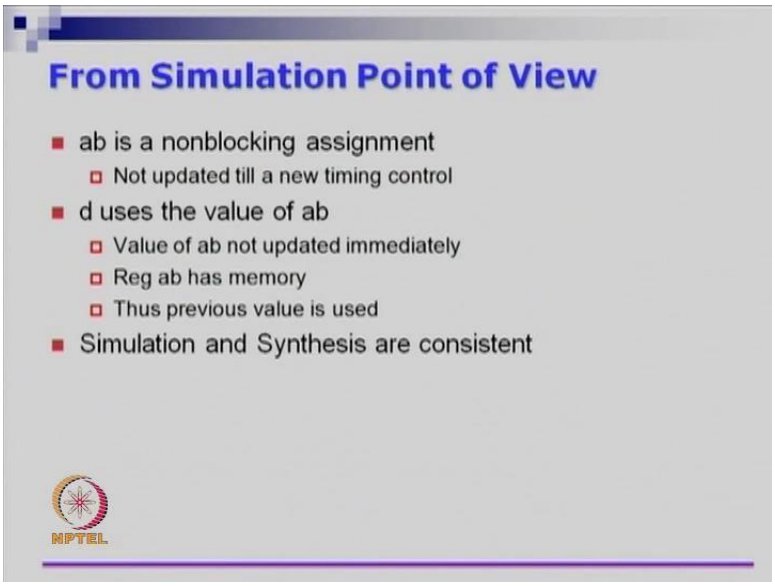


So, the problem with a model this essentially this, the multiplier will work on the current a and b that is given, but adder works on the current c and previous a b. You go and look at the picture, ((Refer Time: 13:24)) if you go and look at what the adder is adding, let us say at some point I gave A, B and C, which all change together. At time instant i, let us say I gave a fresh set of inputs A of i, B of i, and C of i, these are the inputs that I given a time instants i.

This multiplier will be looking at A of i and B of i and you gives C of i here, but this adder will take C of I and A B of i minus 1, because is the register, it is holding the previous value of the multiplication of A and B. So, it takes A B of i minus 1 and C of i and adds it, this is the problem, because the current input A and B multiplied together and add it to C.

However, we are adding the previous value of A and B is product with current value of C. If this is not programmer's intent, then this hardware is incorrect. So, in this case, we want to do a b plus c and this is doing a of i minus 1 times b of i minus 1 plus c of i and this is not correct, this is not what the intent was.

(Refer Slide Time: 14:28)

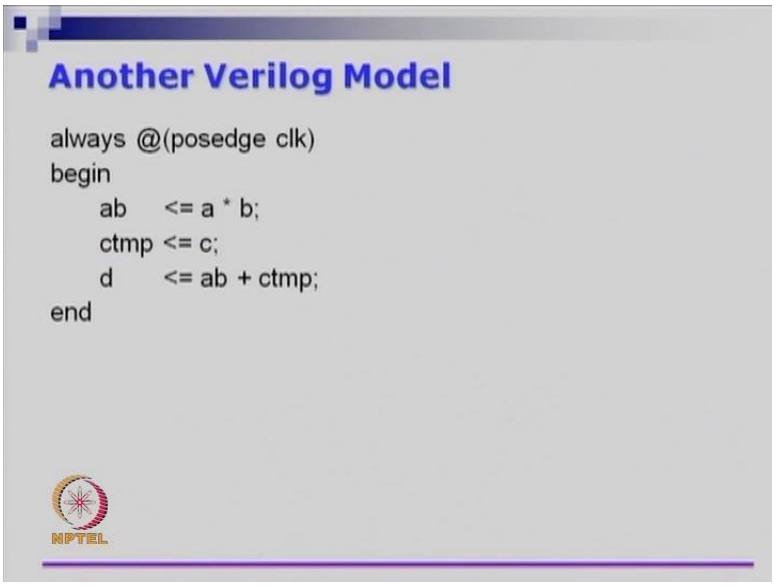


So, from a simulation point of view let see why this is consistent first of all. So, the hardware that we saw was the picture that I showed earlier, let see from the simulation perspective, whether this is supposed to be correct. So, a b was a non blocking statement. So, a b non blocking equals a times b. So, it will not be updated until a new timing control comes through.

So, remember, ((Refer Time: 14:54)) if you go back to this Verilog code here, so you go and look at all the right sides, the right sides will all be calculated first. So, when the new clock comes, it will take current a, current b, multiply it, you keep the result a side to assign to a b in the future. Then, you take current a b and c add them and put it for d, but the current value of a b that you have is something that you had in the previous side. So, that is the problem.

So, a b is a non blocking statement, it is not updated till a new timing control comes through. So, d uses the value of a b, but it is not updated immediately. So, the simulation and synthesis view that we have for this is correct. So, essentially, the model that we have is wrong. The synthesis and simulation matches in what they are a showing and the model that we have is incorrect for the problem that we started.

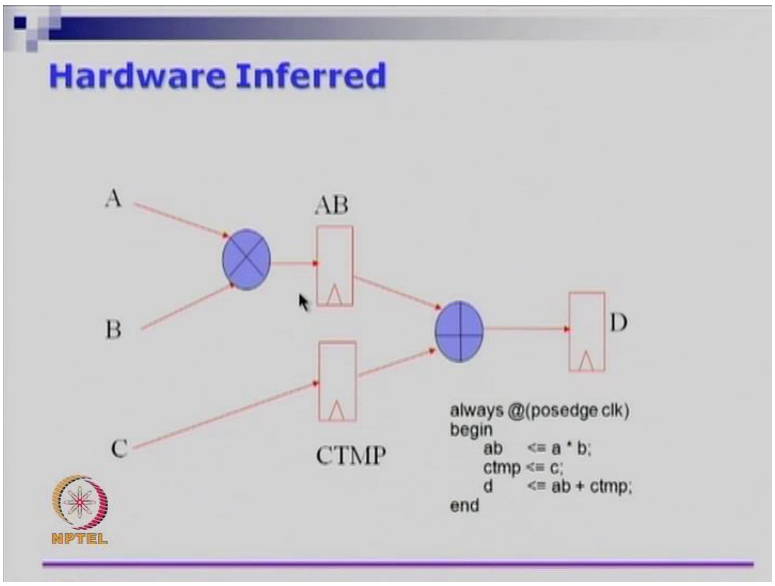
(Refer Slide Time: 15:49)



Let us look at another Verilog model. So, in fact, ((Refer Time: 15:55)) if you look at this, what you have is, this is not properly form, it is not a well formed pipe line circuit. So, this problem with this is, it is not a well form pipe line circuit, because A and B go through two levels of registers, whereas C goes through the only one level of register. So, there is the problem with the path from C to D.

So, I cannot put a new flip flop here and the path from C to D, because that will also increase the path of the flip flop from the length the number from A to D. So, what that means, is, in this path from C to the adder, I should put a flip flop. In that case what happens is, it makes it a well form pipe line circuit and the next Verilog model is essentially that. So, let us look at that Verilog model always at posedge clock a b is a times b and we have something called ctmp, we say ctmp is non blocking equal c and d non blocking equals a b plus ctmp. So, the key thing is, you have ctmp here and not c.

(Refer Slide Time: 16:57)

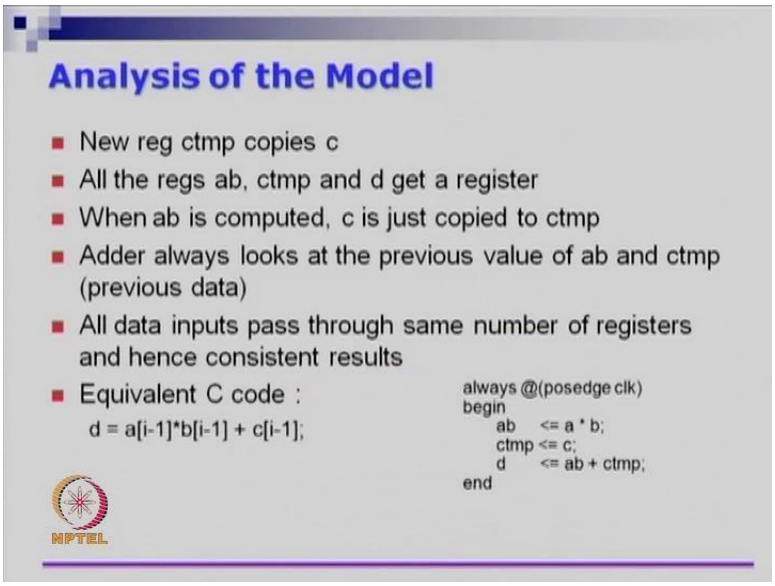


So, let us see what the hardware inferred for that would be. So, again the Verilog code is showed in the right bottom here. So, a b, ctmp and d are all on the left side in an always statement, which as a posedge of clock, which means, a b ctmp and d will all in for flip flops or registers. So, A B is a register CTMP is a register and D is a register and what are the operations that we need, we need a multiplier and we need in an adder. So, there is a multiplier and there is an adder.

Now, let us go and look at how these things get connected? So, the product of a and b should be given as input to a b. So, the product of A and B is given as input to A B, c is given as it is as input to ctmp. So, we give C as it is as CTMP and a b and ctmp should have an adder in between. So, the addition of the should go to D. So, that is what you have here A B and CTMP are added to gather in the addition goes to D.

So, what we have is, we have we have taken this always block and inferred this piece of logic. Now, let see whether this is well form pipe line circuit. So, from A to D, there are two register, two levels of flip flops, from B to D again there are two levels, from C to D again there are two levels. So, for all the intermediate levels also there is no miss match. So, if you go and look at A B itself, so if you go and look at the wire A B, it is after one level. So, there is no miss match anywhere else from all the inputs to an outputs, we have two levels. So, this is the two stage pipeline and this is a well from pipeline.

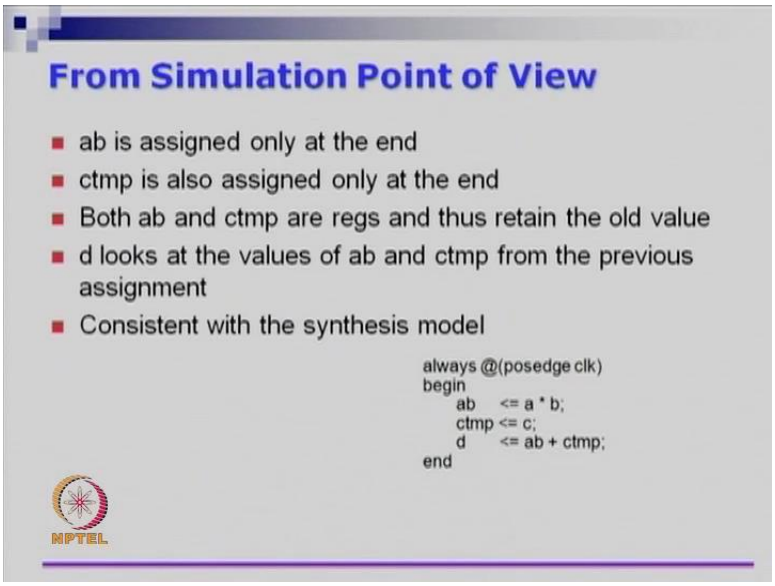
(Refer Slide Time: 18:37)



So, again the Verilog code is showed in the right bottom, analysis of the model shows that new reg ctmp copies c. All the regs a, b, ctmp and d get a register and when a b is computed, c is just copy to ctmp and adder always looks at the previous value of a b and ctmp, all the inputs pass through the same number of registers. So, the equivalent thing that we have is, you looking at… So, if I go and give input i; it is going to take two cycles are a latency of two cycles to come to the output. ((Refer Time: 19:17))

So, the D input that you are going to see here is actually looking at A B of i minus 1 plus CTMP of i minus 1, which is actually equivalent to C of i minus 1. So, the equivalent code would have in software would have return it as d is a of i minus 1 times a of i minus 1 plus c of i minus 1, which is perfectly fine. As long as, all the input that we are having or suppose to be coming from the same instant, only that the output as coming two cycles late that something that we have to watch out for that is all.

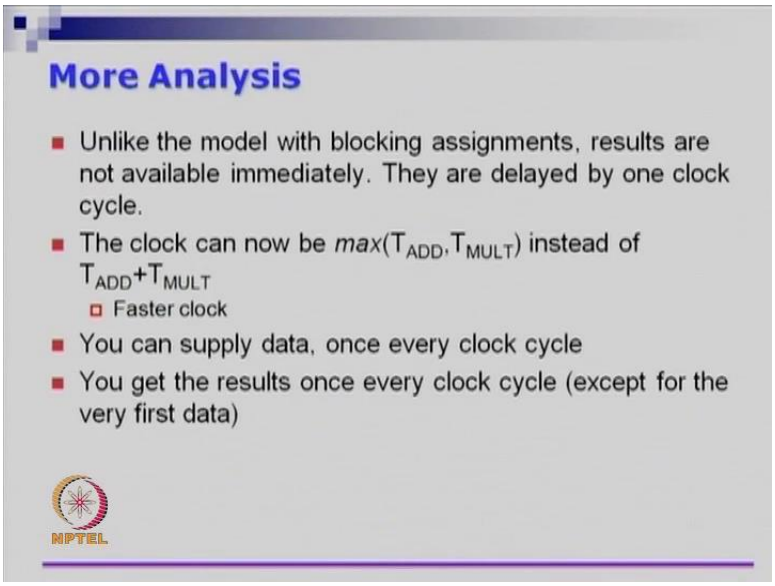
(Refer Slide Time: 19:50)



From a simulation point of view also this is perfect, because you go and look at all the right side, you compute the current, you take the current a and b multiply it, but you want give it in the future. You take the current c, whatever c is storing and put it in ctmp, but whatever a b and ctmp are storing right now, add then put it in d. So, we evaluate the right side first a b that a times b should be given to a b in the future, you do not do it right away.

Whatever value of c should be given to ctmp in the future, you do not give it to right away that whatever value a b and ctmp are right now add them and assign in to d in the future. And the future will come, when this always block is suspended as in all the assignment are evaluated all the right sides are evaluated. Now, we are ready to go and assign to the left sides. So, all of them will have a consistent view.

(Refer Slide Time: 20:44)

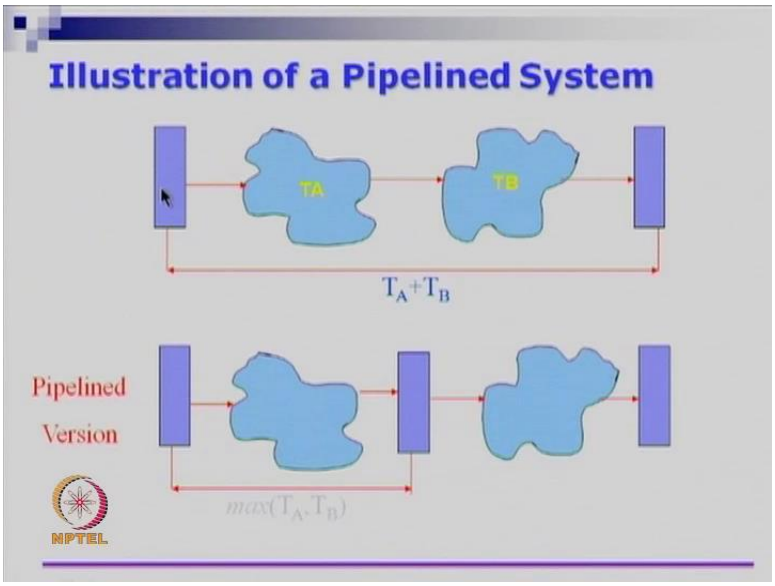


So, a little more analysis, unless unlike the model with the blocking statements, the results are not available immediately, I hope that something that you got from this module from this week's module. The clock can now actually be the maximum of T ADD comma T MULT, instead of T ADD plus T MULT. So, T ADD plus T MULT as long as both are positive maximum of T ADD comma T MULT is guarantee to be less than or equal to T ADD plus T MULT.

Essentially, what we have is, we have a clock that can run faster, we can supply data every once every cycle, but and you also get the results once every cycle, except for the very first cycle. So, essentially there is a latency of two cycles, there is through put of one input; that is being process one set of inputs; that is being processed every cycle. So, this is what we call pipe lining, when the multiplier is working on the current set, the adder is evaluating the results from the previous set.

So, the adder is getting consistent set of input, but it is operating on the previous thing that you gave. So, that is the data path elements are actually working in tandem and this is what we called pipe lining. So, the data is an essentially marching through from in some sense left side to the right side or from the primary inputs to the output register, under the control of the clock. And this is facilitated by small combination block that we have put in between, namely the multiplier and adder.

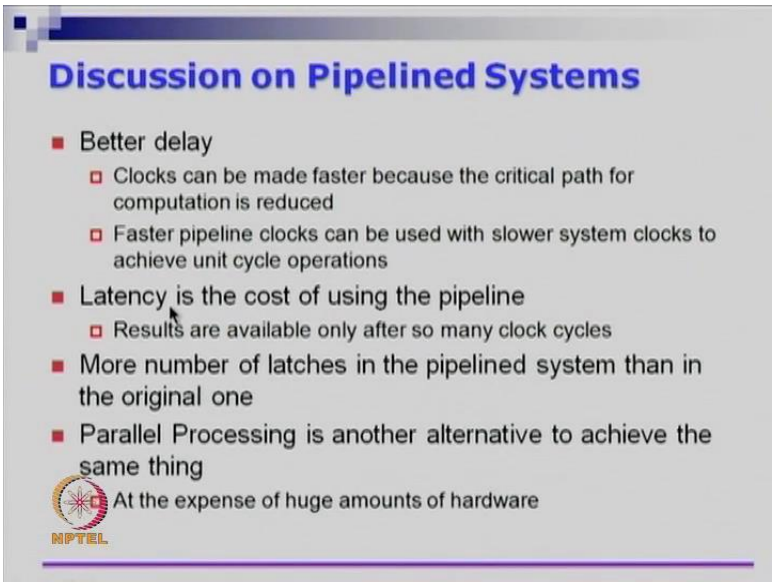
(Refer Slide Time: 22:14)



So, in terms of picture usually for pipe lining, people draw what is called cloud diagram. So, let say this is the block A, which is which could be the multiplier and this is block B, which could be the adder. Then, without pipe line the set of inputs going through some combinational logic going through this would have a delay of T A plus T B. So, let say this is some combinational logic, this is some other combinational logic. Then, a set of inputs that is firing from here will need T A plus T B time plus set of time here plus Clock to Q delay here to settle down.

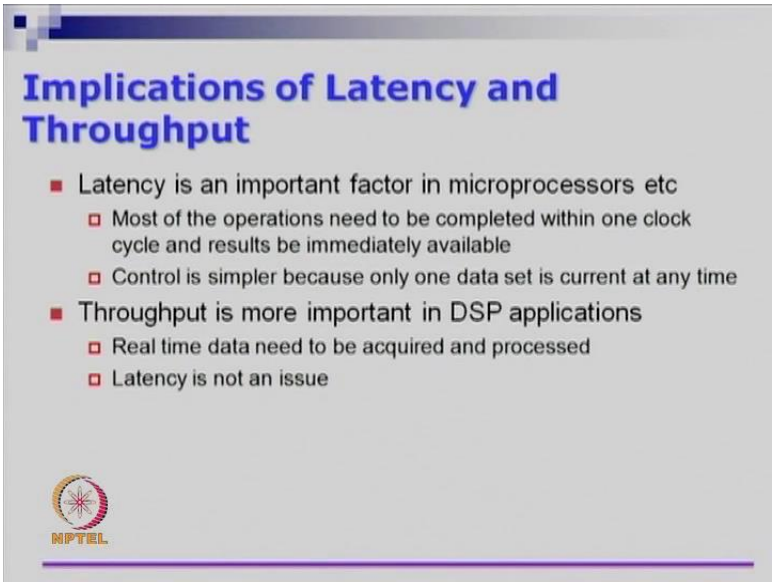
Whereas, the movement you put a flip flop in between, then this to this flop is T A plus setup time plus Clock to Q delay here and this to this, would have been Clock to Q delay here plus T B plus setup time. If Clock to Q and setup time are the same, then we are essentially looking at the rate at which the whole circuit can run is maximum of T A comma T B not T A plus T B. So, this is the key thing, this is the most use most important thing about pipe lining. The movement you start putting pipe lining registers in between, a cutting down the combinational delay and you are enabling fast a clock.

(Refer Slide Time: 23:32)



So, you get better delay, but you get at the cost of what is call latency, latency is the cost with which we are using the pipe lining. So, the results are not going to be in the current cycle itself, we are going to take two cycles are, if it is are three stage pipe line to be three cycles and so on. And if you start putting in more and more pipe line registers, the latency will become higher and higher. We also discuss that parallel processing and interleaving at very two different alternatives with which you can do similar things. So, you can achieve the performance of pipe line circuits by also doing either interleaving or by parallelism.

(Refer Slide Time: 24:05)



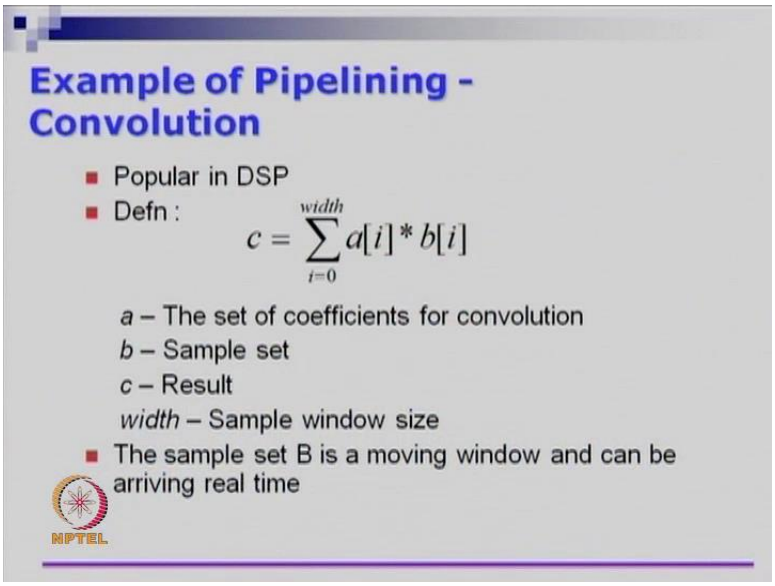
So, latency is important for circuits like microprocessors, so processors like CPU's and so on. Latancy is really important, because when a set of inputs come in, how quickly the result comes out in terms of cycles, this is important for microprocessors. And most of the operations in fact inside a CPU, needs to finish with in one clock cycle, things like addition, multiplication, logical AND, logical OR and so on, you would want all of them to finish with in one cycle.

So, which means, what we have to do is, you have to design a circuit that is really, really fast. Remember, if you given only one cycle to do addition or multiplication, then we cannot say, I will pipe lining it and I get the results after five cycles. We need the result in one cycle, the whole thing should be combinational and should be fast enough to accommodate, let say 1 nano seconds delay. So, that is all you can get for 1 Giga hertz clock that is use for a processors.

In fact if you using 3 Giga hertz or 4 Giga hertz processor, you are looking at either 0.33 and seconds of 0.25 nano seconds to implement the whole adder. So, are multiplier and what not. So, latency is really important for micro processors, whereas if you go and look at DSP, Digital Signal Processing are anything like that, through put is a lot more important. You looking at what is rate at which I can process video frames or what is rate at which let say spoken wave forms or process and so on.

So, usually you are acquiring real time data and you are processing it, latency may not be a big issue. However, through put, I should be able to process everything that is coming in as input and I should be able to take one frame after another every so many time units and so on. So, for such things like DSP through put is lot more important that latency.

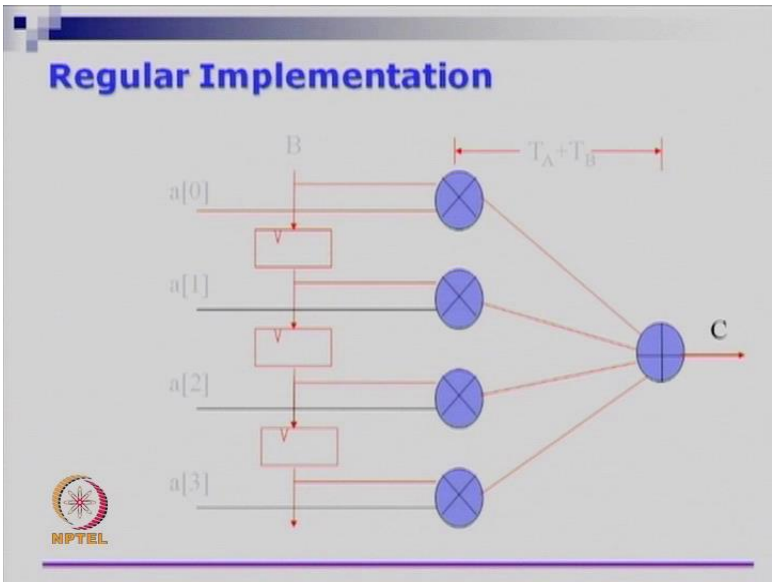
(Refer Slide Time: 25:55)



So, let us look at a simple example in signal processing call convolution. So, if you are in the 4th semester you might have of learn what convolution is. So, it is simplify is essentially as simple as this you take a of i and b of i in this case, we are looking at different values not time a of i into b of i and i equal 0 to it. So, you are this is essentially equivalent to multiply add and accumulate.

So, you are multiplying a of i and b of i that is the multiply and we are adding for several values of i and we are accumulating the results in c. So, this is usually called Mac operation and w is usually the window size and the result is eventually stored in c. So, the sample set is essentially moving window and can be arriving in real time. So, if you change i equal to 0 to width, whatever the width of the window is a processing that and then you move the window, and then process this once more and so on; that is what convolution is…

(Refer Slide Time: 26:59)

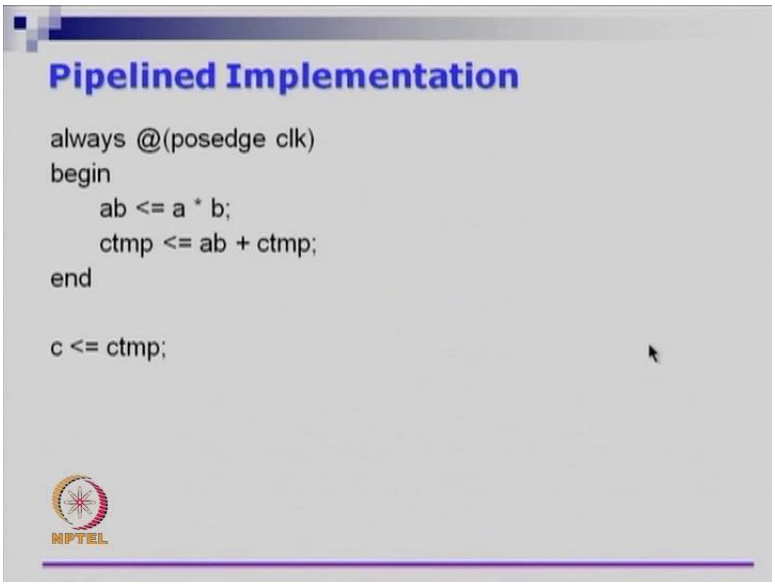


So, if you want to do hardware directly right you may actually end of something like this. So, B is the set of inputs it is coming through and let say, we have doing convolution over of a window of size 4 So, 0 to 3 is a window of size 4, you may end of doing something like this. So, a of 0, so you get B, which is set of coefficient and we passed then B is data that is coming in, a is a set of coefficient.

So, B is real time data that is coming in. So, it is coming one sample every clock cycle and so let say zeroth time instance it is here, all these flip flop have incorrect values, at 1, it will be here and 2, it will be here and 3, it will be here. So, at time step 3, B of 0 would be here, B of 1 would be here, B of 2 would be here and B of 3 would be here and when you take the product of those, you have multipliers for those.

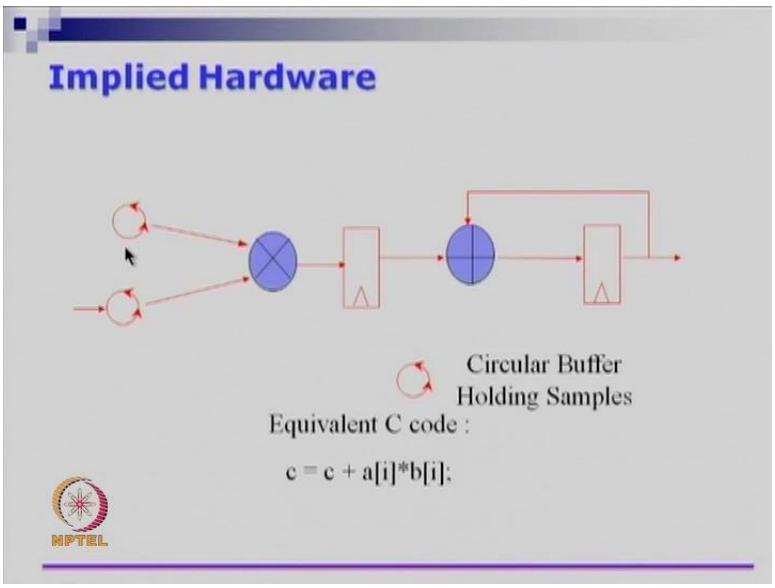
And you have an adder for that, if you go and look at the combinational delay, so you are looking at what is coming from the flip flops, through the combinational logic. Let see, even if I put a flip flop here, then I have a multiplier, and then adder in the longest path.

(Refer Slide Time: 28:11)



So, one way to break that is to be able to do something like this. So, a b is a times b, ctmp is a b plus ctmp. This is what we have been see all along, a times b and we are adding with ctmp and we are putting the result back in ctmp itself, finally, we assigning c to be equal to ctmp. So, the hardware inferred would be something like this.

(Refer Slide Time: 28:37)



So, we go and look at this, a b is a registers ctmp is a registers, so we have a register called A B, we have register called CTMP. So, we have a multiplier and an adder, which feets in to CTMP and A B respectively. If A and B are going to come from circular

buffers, this multiply will take one set of value at a time. So, circular buffer is nothing but set of values that are in a ring.

So, this set of values is for A, because we are looking at only the coefficients, this set of values is for B, because it suppose to come from the external world, you take one at a time from here and here, multiply that, put the result here. And take the previous value of CTMP added to the A B; that will give you the next value of CTMP. Once, you do that, then this buffer rotates by one step this rotates by one step that gives you a new input along with the new coefficient. And if you keep doing this, this is equivalent to doing c equal c plus a of i into b of i.

So, a pipe line implementation is as simple as that, if you go and look at this one, this assign c equals to ctmp. So, there must be an assign here, assign c equals c temp a combinational, all it is doing as it is stopping the register call ctmp and giving it to the external world this brings me to the end of week 8. So, we are end of module 47. So, for what we have done in this week is, we started with the notion of pipe lining.

Remember, pipe line is a acyclic circuit and which means, there is no feedback path from any register to any other register. This is always in some sense, you can draw the pictures from left to right or top to bottom and so on, without having to go back in the reverse and once, you have such a circuit, then it is nice, it has a lot of interesting properties. So, we looked at what is a latency, what is through put and we also looked at, what is a K stage pipe line and what are the careful thing that, when we need to do when we design a K stage pipe line.

And then we look at a whole lot of Verilog and the implications of equal to blocking equal to statement verses non blocking equal assignment statement. So, once you understood all the implications of that, the pipe lining using Verilog becomes much more easier. So, this brings me to the end of week 8, the assignments in this week as I mentioned in the beginning of the course are slightly lagging behind, whatever is there in the course otherwise.

So, the Verilog assignment that you are going to see in this week or going to be based on whatever we see till week 7, not involving the current weeks lectures. So, thank you and I will see you next week. Whatever I cover in the next week will, so I will put everything

together and give a complete over view of whatever you have seen in the course and I will also cover arithmetic circuits and number representations in the next week.

So, thank you and I will see you in next week.