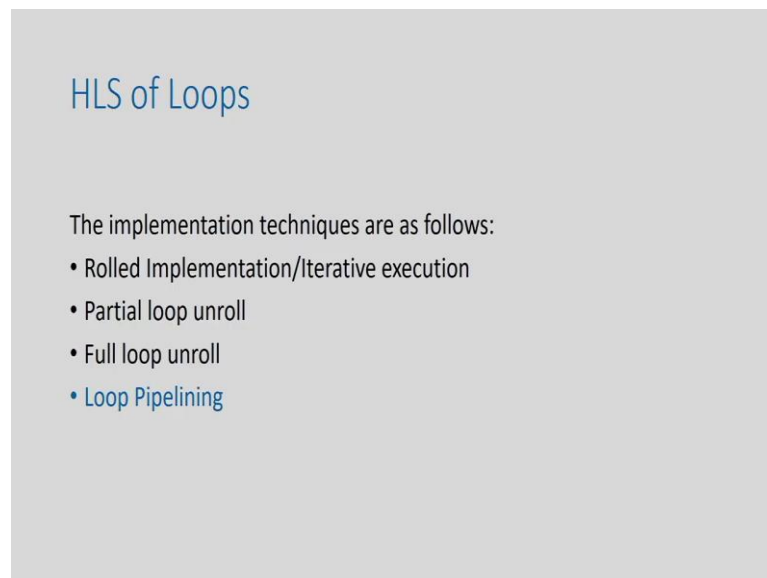


C-Based VLSI Design
Dr. Chandan Karfa
Department of Computer Science and Engineering
Indian Institute of Technology, Guwahati

Module - 07
Efficient Synthesis of C Code
Lecture - 25
High-Level Synthesis of Loops-Pipelining

Welcome everyone to my class C-Based VLSI Design. So, in this week we are discussing on this High-Level Synthesis of Loops and in today's we are going to discuss on Pipelining ok.

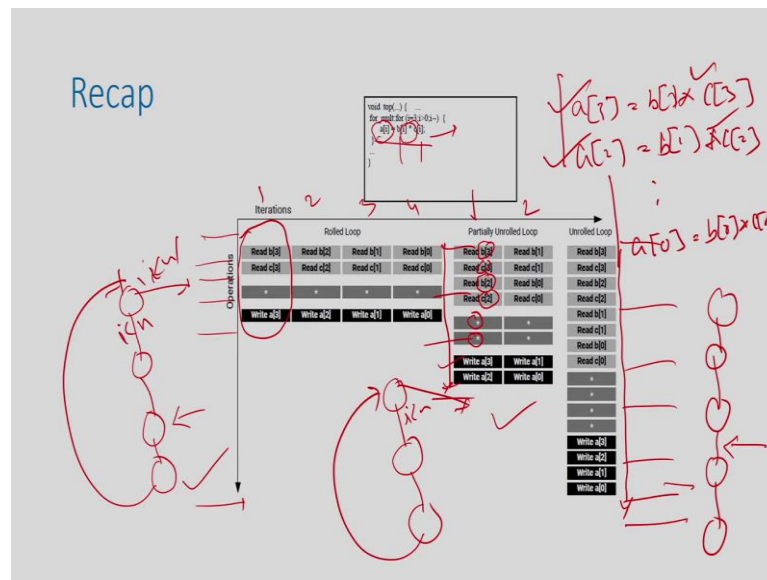
(Refer Slide Time: 00:57)



The slide is titled "HLS of Loops" in blue text. Below the title, it states "The implementation techniques are as follows:" followed by a bulleted list of four items: "Rolled Implementation/Iterative execution", "Partial loop unroll", "Full loop unroll", and "Loop Pipelining". The last item, "Loop Pipelining", is highlighted in blue.

So, we have already seen that for loop there are 4 type of way we can implement the loops into hardware. One of them is basically the iterative executions, then partial unroll, completely unroll and loop pipelining. So, we have already discuss about the first three. So, today's topic is Loop Pipeline.

(Refer Slide Time: 01:13)



Before going into the discussion loop pipelining just recap what we have understood about this first three options ok. So, the first one is the rolled implementation or the iterative implementations. So, if we have a loop we what we are going to do? We will take a one iteration of the loop. We will going to schedule it and then I am going to have a controller which is going to repeat the same thing for all iterations right.

So, if you just see this diagrams. So, if you have a loop like this where you have a operation here. So, you are basically reading b and c and then you are doing a multiplication and then you are writing to a right. So, the way it is going to happen is basically you read. This is my iteration 1, this is my iteration 2, this is iteration 3 and 4 right.

So, I am going to take this operation, I am going to decide the schedule. So, this might be schedule in 1 cycle or it might be schedule in say 1, 2, 3, 4 cycles. It is up to the way we actually implement it right. So, we will take a iteration, we will do a schedule and I will identify number of state needed to execute one of the iterations. Say I need 4 iterations and then I am going to repeat it for n times right.

So, when this i less than n and if not i less than n then I am going to execute it. So, this is the basic level of implementations and we understand that this actually need lot of time right. So, there is no parallelism is happening right. So, then we have a

complete unroll. So, where we exactly completely unroll the loop, there is no loop at all.

So, we will have 4 operations like $a^3 = b^3 \star c^3$, $a^2 = b^2 \star c^2$, $a^1 = b^1 \star c^1$. So, that is the complete unroll and that is the implementation. That means, all the operations are available with me right in the 1 clock in at the same time. It is not that different iterations are happening.

So, we have all operations are actually available with me and then I can actually try to schedule them parallel, right. So, I can actually schedule say in say 5 cycles. So, then my FSM will look like this, alright. So, and that is all. There is no loop.

So, this unroll is actually good in the sense that there is no repetitions of the iterations because all the operations are available with me. So, I can actually do a parallelization during scheduling, I try to schedule maximum operation in one time step. But we can understand that this need maximum resource right, and this need minimum resource, but maximum time

And then in between comes the partial unroll. So, instead of running this loop for four times I am going to run two times and then every iteration I am going to do the work of two iterations right. So, this is exactly this diagram shows.

So, in the first iteration 1 and this is my iteration 2. Iteration 1, I am going to read both $a^3 = b^3$ and $a^2 = b^2 \star c^2$, and then I am going to do the both operations both this one and this one. So, I am going to read these two and then next time I am going to do this. So, I am going to do these two multiplications, this and this and then I am going to write both a^3 and a^2 right. And the next iteration I am going to do this.

So, now once I have this I can have a schedule which is maybe because there are more operations available. My schedule can parallelize the operations and say suppose I able to do it in say 3 iterations right. So, I just decided to do the like this ok, so some logic and then this is going to iterate for two times right. So, i less than n and again the same conditions.

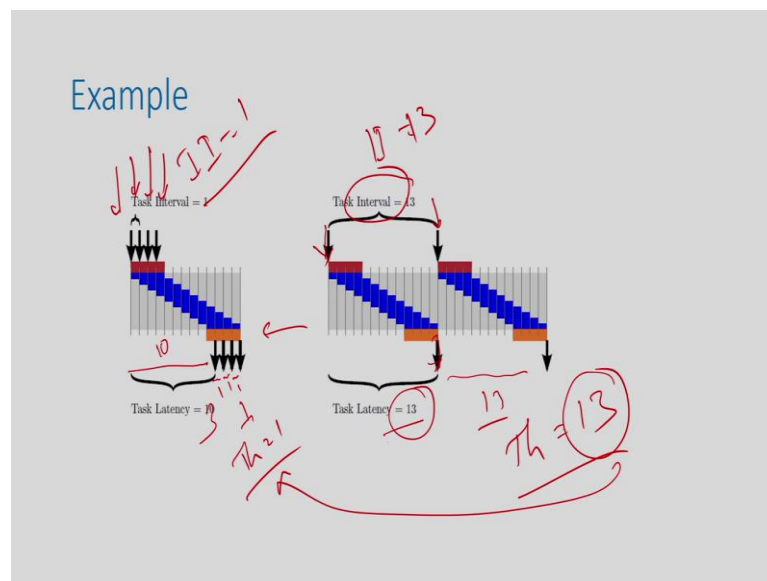
But this is where since the number of state reduce and the number of iteration also reduce, so total time to execute this partial unroll is something much better than this

and but it may be more than this. But the resource requirement is basically is less than this and it is more than this. So, it is basically a tradeoff between this completely unrolled versus completely rolled right. And then we have seen these are the three options we have.

And so pipeline, what it does? It basically try to execute two iterations of the loop in overlapped manner right. So, that it is not only try to execute one iteration at a time, it try to overlap the execution of two iterations of the loop or multiple iteration of the loop at a time.

So, as a result, obviously, my resource requirement will be more because if you want to execute more things at the time, so, your resource requirement will be more. But, your throughput or the initiation interval both will improve right. So, that something is this diagram.

(Refer Slide Time: 06:02)



So, for normal iterative iteration suppose you are giving an operation here and you are getting output after say 13 cycle which is the latency of the design and then you are going to give the next input. So, your initiation interval II equal to 13 and your latency is also 13. And then the next output you are going to get after again 13 cycles. So, your throughput is also 13.

So, this is the possibility. So, basically this is the my iterative iterations, iterative execution of the loop and which is has the my initiation interval is very large. My latency is high and then my throughput is also very low right, because I am getting one output in every 13 cycle.

So, what it does the pipelining? So, it still I need say latency is 13 say I mean, I have such a way I designed that by still I have need the desire in latency 13, but what I am going to do? I am going to give my input in every clock right. So, I want to give my input every clock. So, my II become 1 in best case.

And my throughput is also I am going to also after sub type. So, for the first output I have to wait for some time and then after that in every clock I am going to get the output right. So, my throughput will also become 1. So, this is what is the pipeline design.

So, you can understand that this is something is really useful because your throughput really become high from 13 to 1, my initiation interval I. So, I can actually take input in every clock and it is actually running very faster. And we will show that the this particular pipeline design need much less resource than compared to the complete unrolled one ok. So, let us try to go into that.

(Refer Slide Time: 07:45)

Loop Pipelining

- Loop pipelining allows the operations in a loop to be implemented in an overlapping manner.

```
void func(m,n,o) {
  for (i=0; i<n; i++) {
    op_Read;
    op_Compute;
    op_Write;
  }
}
```

`#pragma HLS PIPELINE II=1`

The diagram shows two timing diagrams, (A) Without Loop Pipelining and (B) With Loop Pipelining, overlaid on a clock signal. In (A), the operations op_Read, op_Compute, and op_Write are performed sequentially, taking 3 clock cycles to complete. In (B), the operations are pipelined, with each stage (op_Read, op_Compute, op_Write) taking 1 cycle and overlapping with the next stage, resulting in a total of 4 cycles for the loop body. Handwritten red annotations include $a[i] = b[i] \times c[i]$ and $a[i], b[i], c[i]$ with arrows pointing to the pipeline stages.

(A) Without Loop Pipelining (B) With Loop Pipelining

So, the basic idea of loop pipelining as I mentioned it is basically allows the operation of loop to be implemented in overlapped manner right. So, let us take the example. So, suppose we have a for loop where we are actually doing the same thing you can think about this is basically a i equal to $b_i \star c_i$ right. So, this is basically you are reading b and c and then you compute this multiplications and then you are writing to a , right.

So, this is the high level code and this is my iterative execution right. So, this is my clock. So, I just read the data, then I compute the multiplication, then I write into a ; this is my for iteration 1, then this is my iteration 2, this is my iteration 3. So, suppose this loop is going to execute for 3 times. So, I need kind of 9 cycle right because this is every time I am going to write something.

In the pipeline what I am going to do? I am going to give I am for the first iteration I just do the read of a , b and c . I will calculate in the this b into star into next clock this is my next clock and then I am going to write it. In the second clock I am going to compute this b_2 into c_2 , but I am going to read b_1 and c_1 .

So, if you see here this is 2. So, in this read is basically b_2 and c_2 . So, in the first clock I can see I am only reading this element right. In the next clock what I am doing is I am doing this b_2 into c_2 in this operations and I am also reading the next input. So, I am going to read b_1 and c_1 , right.

In the next clock in the third clock what I am doing; I am going to do I am going to write this a_2 because I have already computed this. So, I am writing the data into a_2 for the first data, for the next data which is already written I am doing the multiplication. So, I am doing $b_1 \star c_1$ and also I am started reading the next data right. So, I am going to write I am also reading b_0 c_0 right.

So, you can understand that so, basically multiple iterations are overlapped. So, if the number of iteration is more this overlap will give you more benefits, because it will continue this three operations in parallel for many cycles. And then you will get maximum benefit right.

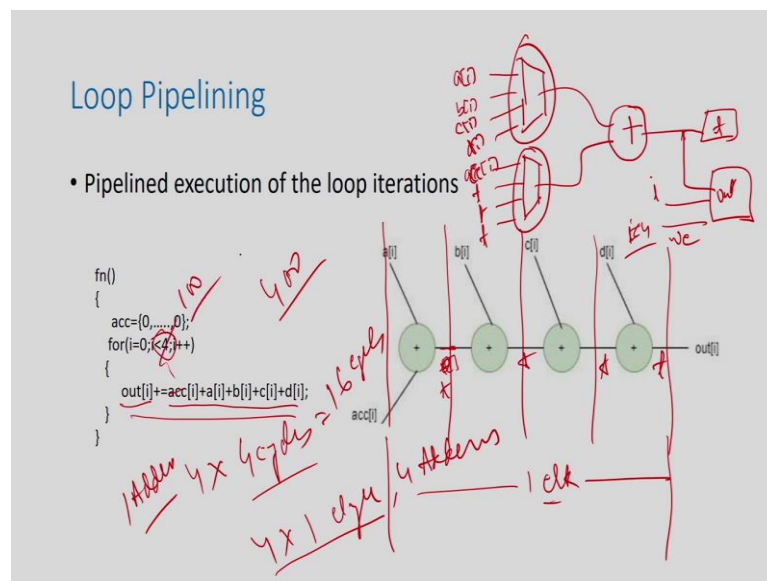
So, this is how; so, you can actually see that every clock I am able to give the inputs which so, that my initiation interval become 1 whereas, it was 3 because you are

giving inputs here, inputs here, inputs here. So it was 3 earlier, now it become 1. And also for the first input I have to wait for 3 cycle, but for every next input I am going to give the getting the output in every cycle.

See here I am getting output here, getting output here, I am getting output here. So, my initiation interval is 1, throughput is 1 and the latency is still 3 because for each input I need 3 cycle to I mean complete the loop, but the overall performance is drastically improved using loop pipelining ok.

And in since we will going to take this Vivado HLS for our experiment purpose. So, the Vivado HLS if you just specify this pragma HLS pipeline with the initiation interval 1 it will exactly give you this diagram. So, this pragma if you just put inside the loop it will pipeline that loop automatically. So, we do not have to do much ok. Only thing you have to understand what is the impact and how things works ok.

(Refer Slide Time: 11:15)

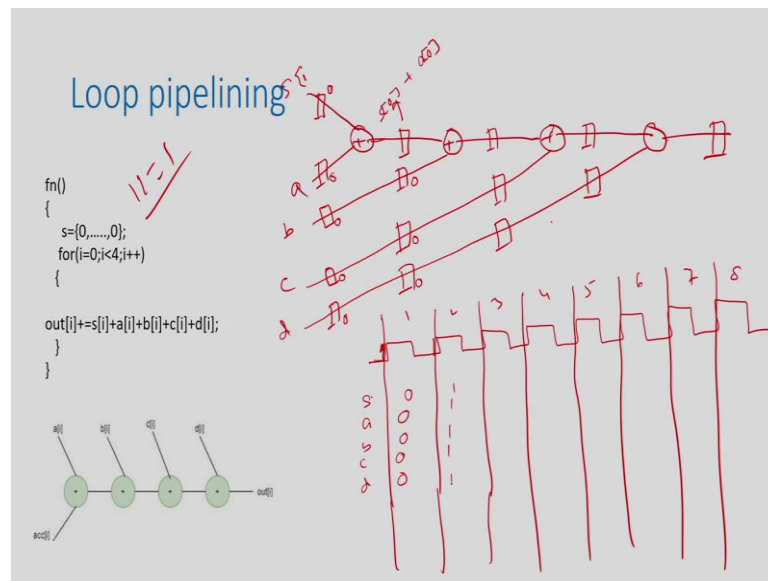


So, what I am going to do? I am going to take an example now and I am going to explain the resource requirement right. So, that is very important to understand the what is the total number of resource require for loop pipelining compared to the iterative iteration right, and unroll will take more. So obviously, that is more, but try to understand that resource requirement for iterative execution versus pipeline.

So, let me taken the example. So, there is a function say here which is actually computing this addition. So, basically each out i is basically acc 0, let us say I. So, acc 0 plus acc is a 1 plus b 1 b i plus c i plus d i right. So, this is a 4 iteration loop and I am doing this right. So, and if you see the dependency for one loop this is the dependency graph right.

So now, let us try to understand if I want to execute this in iterative manner first ok.

(Refer Slide Time: 12:06)



So, if you want to execute in iterative manner what I told? I just told you that I am going to take one iteration. So, this is my dependency graph or sequence graph, I am going to schedule it. So, if I if say I schedule it in 4 cycle right. So, I schedule it in 4 cycle. So that means, one of one iteration of the loop will take 4 cycle right. So, if it take 4 cycles, so, what will be the adder requirement here?

Since in every cycle, I am going to do one operations at a time, so I can utilize only 1 adder to do all 4 operations and I can multiplex the inputs right. So for so, if I just try to do this way, so, I need a 1 adder and I need to multiplex the input because right. So, let me just try to draw the diagram. So, what I am doing? In the first iteration I am doing a i plus acc i right. So, I have to give here a i and here I am giving acc i.

Next iteration what I am doing? I am doing b_i here and I have to give some name here because this is something we will store somewhere right. So, this is say t_0 right. So, some temp I ; some variable t right, so, which is a register. So, then I am going to give t here right. So, this is my t basically right. So, every time I am doing these operations I am going to storing some register.

Next iteration I am going to do c_i into again t . So, I am going to give t again here right. So, then this is this and the next iteration I am going to give d_i here and again the t here right. So, this is how this will look like. And then once this 4 iteration is over so I have some counter i which will I designate this and that will store to my out right. So, then I will give i and then some write enable will decide by this once this 4.

So, when is basically i less than 4, so, sorry i equal to 4. So, this is the write enable and I am going to write this whatever the data I am computing here t to this data right. So, this is what the iterative execution of this loop body. I need only 1 adder, but we need two marks just to multiplex the inputs right. So, that is the idea.

Now, as I told you that once you have this loop and you can actually decide the schedule yourself right. So, here you need 4 cycle, if you just try to do 1 adder something, but you can also do the operation chaining right. So, you can actually decide to do everything in 1 clock right. So, this is the 1 clock itself.

So, if you try to do that you need only 1 cycle to execute one iteration of the loop and hence the total execution time is 4 cycle right. And for this the previous one I discuss this is 4 iterations. So, there are 16 cycles right. So, for this using 1 adder to implement the whole thing, you need 16 cycle to complete this.

But if you want to do iterative, but in every iteration I want to do all these 4 operation in chained manner then I need basically 4 adders, right. So, I need 4 adders because I cannot share; I mean share the resource among them. So, I need 1 cycle, but 4 adders and I too complete the complete loop I need 4 cycles right. So, here I need 4 adders, here I need 1 adder right.

So, this is you understand that for iterative execution. If you do not do operation chaining we need only 1 adder, but we need 16 cycle right. Now, let me take the

same example and try to do the pipeline implementation ok. So, if you remember that this; I let me try to show how things will work. So, the pipeline execution, how it will work.

So, I you have this dependency graph right. So, what is I am going to do is this. So, this is my; so, I just remove this s is I rename the s this acc to both this just to this s ok for simplicity for less writing. So, this is my a, this is my b, this is my c and this is my d right. This is what is happening. So, what is the pipeline implementation?

So, the pipeline implementation is that since I want to do the overlapping of iteration I cannot share this adders right in best case. If your II is 1, you cannot just share this adders. But if your II is not 1, probably we can share that I we can discuss later. But let us say your II is equal to 1; that means, in every iteration want to get some data.

So, then what is the implementation? So, you have I am just going to put registers here ok. So, I am going to put the registers here and then let me just explain what is the pipeline data structure. So, I am just going to put this is my pipeline stages right. So, these are all resisters. So, I am going to put resister in after every stage ok. So, this is my structure ok.

So, this is my stage 0, this is my stage 1, this is stage 2, this is stage 3, this is stage 4 ok. So, this is a 4 stage pipeline ok. And then what I am going to do? So, I am going to think about the clock as well. So, let me just draw the clock. This is how the clock happens. So, this is my clock 1, this is clock 2 right, this is clock 2, this is clock 3, this is clock 4, this is clock 5, clock 6, clock 7 and so on right. So, 1, 2, 3, 4, 5, 6, 7, 8, ok.

So, what I am going to do? In the first clock I am going to give this all 0 right; s 0, s 0 everything 0 right. So, I am going to give 0. Then what is going to happen? So, in the first clock this set of resister will contain. So, I am just going to give the index here for less writing otherwise my graph will become too congested. So, this 0 means it is containing b 0, 0 means it is containing c 0 right.

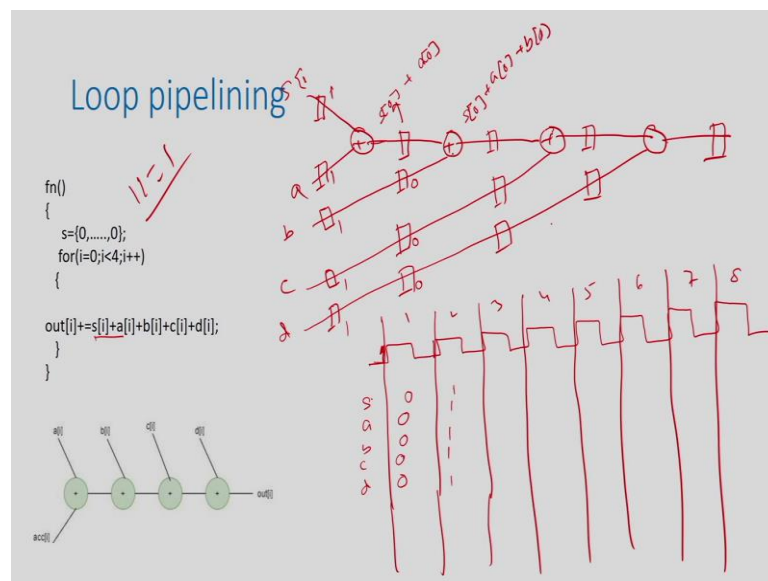
So, what is going to happen? So, my that a, this first stage I am going to give the input s 0 b. So, this is my s, a, b, c and d. So, I am giving the inputs 0 0. So, a 0 this is 0 means s 0 a 0 b 0 c 0 and d 0 right. So, since in the first clock, so when so,

whenever this positive is come here so that data will update this register, so that data is available here right.

So, now this particular adder what is going to do is going to do is doing this addition because these two available is there. So, here what is going to store? It is contain s 0 plus a 0 right. So, this register will contain. So, this calculation will be done and at this point s 0 plus a 0 will be available. So, at this data line it is b 0 at this line it is c 0 at this line it is d 0 right.

So, whenever the next clock comes what is going to happen? So, this register will get the b 0, this register will get c 0 and this register is going to get d 0 right and this register will get this data right. And what I am going to do in the next clock? I am going to give this all 1 right; so, s 1, a 1, b 1, c 1 and d 1. So, what is going to happen?

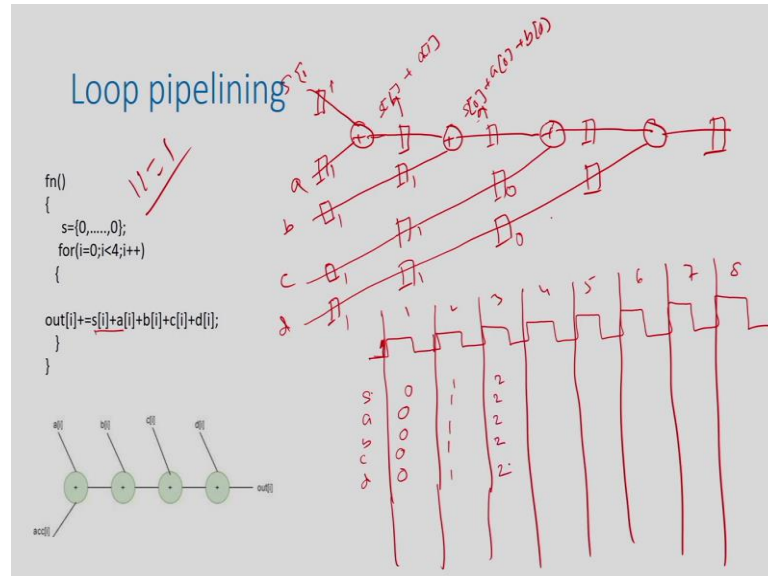
(Refer Slide Time: 20:21)



Then this will this register will contain, this s 1, a 1, b 1, c 1 and d 1 right. So, you understand that in the first clock only the addition s 1 b 1 happen for the first set of data right. Now, you can understand since this particular register has this s 0 b 0 and this is contain b 0, so, it is what is going to happen here; s 0 plus a 0 plus b 0. So, this addition is going to happen here right.

Similarly, since this s 1 and a 1 is available here, so what is going to happen? This addition will be done.

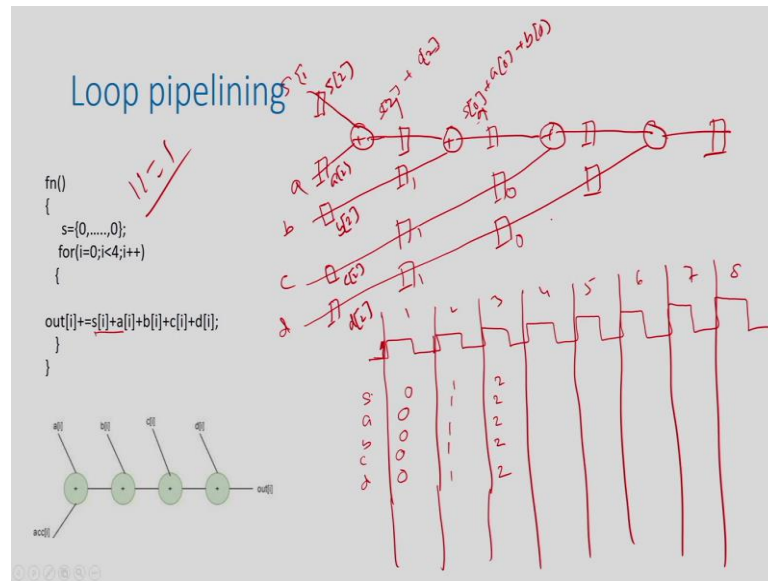
(Refer Slide Time: 21:01)



And here it will store s 1 and a 1 right and this is going to happen. So, whenever the next clock come the 3rd clock come since this data is available here. So, this will stored in this register this 0 will store here. So c 0, this d will stored here right And then since this register input is d 1.

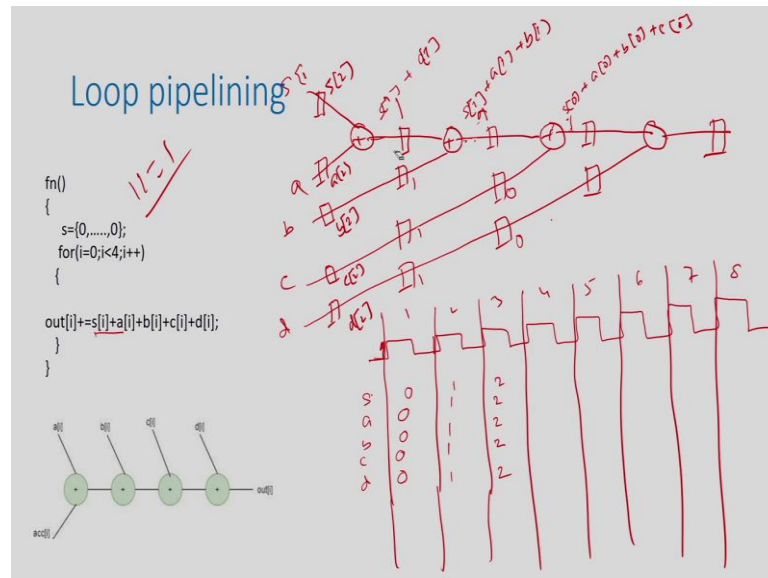
So now, d 1 will store here and then this register input was c. So, the now c 1 will store here and b 1 will store here right. So, this yes. So, b 1 will store here. And then what is going to happen? And this particular register will contain s 1 plus a 1 and in the same time I am also giving the 3rd input. So, s 2, a 2, b 2, c 2 and d 2. So, then this register will now contain the next set of inputs.

(Refer Slide Time: 21:56)



So, this will contain s 2 right, this is s 2, this is a 2, this is b 2, this is c 2 and this is d 2 right. So, as a result what is going to happen? Now, this s 2 plus a 2 is going to happen by this register. So, this particular register will go to do now s 2 plus a 2 right. And then since this particular adder we have this s 2 plus.

(Refer Slide Time: 22:25)



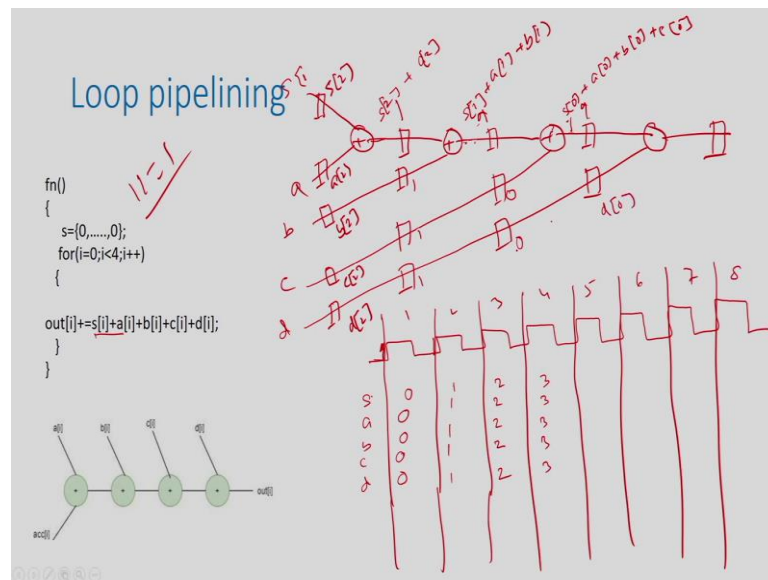
So, this still contain s 1 and a 1. So, I will update later. So now, what is going to happen? Let me just try to. So, this particular content s 0 plus s 0 plus b 0 and this is

content c_0 . So, this adder will now calculate s_0 plus a_0 plus b_0 plus c_0 right. So, this is understood. So, this is it will be available here.

And now this is done. So, basically this at this point in this is s_1 and a_1 is there and this b_1 is available here. So, this will be now the value is s_1 plus b_1 plus c_1 right. So, this will be s_1 plus b_1 plus a_1 plus b_1 right because this is s_1 plus a_1 at this point and this is b_1 . So, this addition will come here and whenever the next clock come it will update this register.

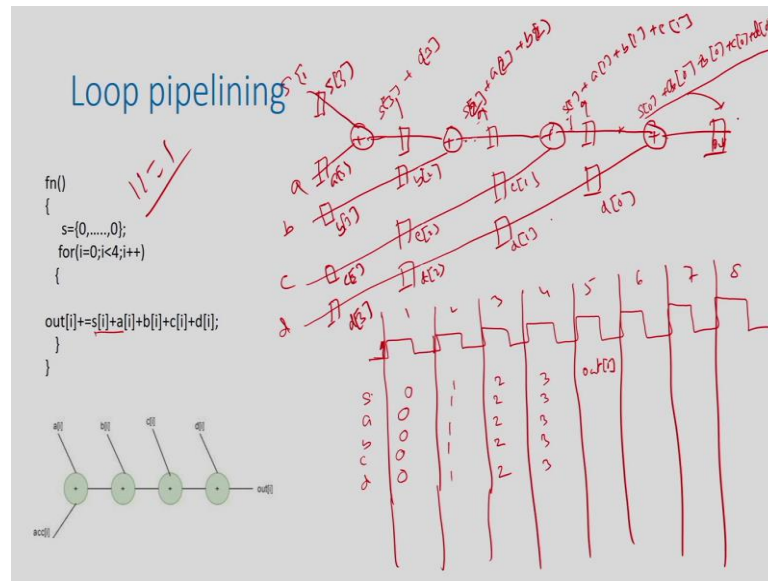
So, similarly this register and this since s_2 and a_2 is available. So, at this point I have s_2 plus a_2 right.

(Refer Slide Time: 23:30)



So, I will have s_2 plus a_2 right. So, now, in the next clock I am going to give the data s_3 a_3 b_3 c_3 and d_3 right. What is going to happen? So, this register will now contain this because this is already updated. This particular register will now contain d_0 right. And this will now contain d_1 because its input was d_1 .

(Refer Slide Time: 23:55)



So, this will be d 1 this will be c 1 input to this register was d 2, so now, it will store d 2 this will contain c 2 this will contain b 2 right. And then since this is I have already given the input now, so this will contain now a 3, b 3, c 3 and d 3 right. So, this will now contain d 3, c 3, b 3 and a 3.

So, we can understand this. So, in the 4th clock it is going to happen? So, this adder one input is d 0 and other input is this. So, it will calculate this whole summation right. So, s 0 plus b 0 plus sorry; this is a 0 plus b 0 plus c 0 plus d 0 right. So, this calculation will available at this point.

So, this adder one of the input is s 1 a 1 plus b 1 and the other was c 1. So, this will now calculate s 1 plus b 1 plus c 1 plus d 1, right. So, this will now contain s 1 plus b 1 plus c 1 plus d 1 right; so, s 1 plus a 1 plus b 1 plus c 1 plus d 1.

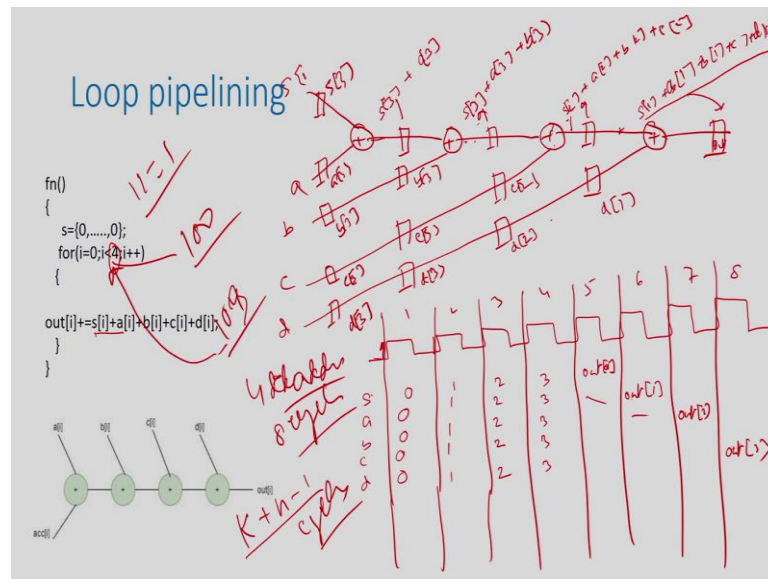
So, for this adder the input is s 2 plus a 2 and this is b 2. So, it will now update this s 2 a 2 and b 2 right. For this adder the input is basically a 3 s 3 and a 3 and s 3. So, this will calculate now s 3 plus a 3 right. So, it will store now s 3 plus a 3 right. So, this is understood.

So, whenever the next clock comes whenever I am going to give the next of the input, so if, so, since there is no other input I am going to stop it here. So, what is going to happen? So, it will contain still this data. So, you can have some enable

signal just to say there is no valid data. So, you can stop that. So, whenever the next clock comes what is going to happen? This output will store this value right. So, this will get this value now right.

So, for the first iteration data I am going to get the out 0 here right because that is my out 0.

(Refer Slide Time: 26:11)



And then this particular register will get d 1 because its input was d 1, this register will get now s 2 plus a 2 plus b 2 because c 1 was there. So, this will now store I am quickly doing this now. So, this will take s 2 plus a 2 plus b 2 plus c 2. So, this will be updated with c 2 because its input was c 2. So, let me just update this data first. So, this will now c 2 and this was input was 3. So, this will all update with 3 right. So, this will be 3. So, it will now contain b 3 c 3 and d 3.

So, now we can correlate. Now this will do this s 2 plus a 2 plus b 2 plus c 2. So, this will do s 3 plus a 3 plus b 3. So, here I am going to write s 3 plus b 3 plus c 3 and for this there is no input. So, whatever it will do something, but it is immaterial now, so because this loop is only for 4 iterations.

So, what is going to happen now? In the next clock for this one it is basically d 1 plus this. So, this all will replace by 1, right. So, this will all replace by 1. So, this

will be $s + 1 + b + a + 1 + b + 1 + c + 1$. So, in the next clock what I am going to get? I am going to get out 1 because this is all 1, right. So, you understand that.

So, this way all things will be shifted every time by 1 and then in the next clock I am going to get output 2 and the next clock I am going to get output 3 and that is what is there. So, if the loop is more, so, this whole process will continue. So, what we understand from here? I need 4 register 4 adder, right. So, I need 4 adder. And how many clocks I need?

So, first output is coming after 4 clock and then in 8th cycle I am going to execute everything, because first output is coming in after 4 clock because I have added 4 stage pipeline and then after that every iteration I am going to get output. In the previous diagram what I have drawn here? Here I am going to get out 0 in clock 4, clock 8 I am going to get out 2 or clock 12 I am going to get out 3 and clock 16 I am going to get the this out 4 right.

So, here it is actually in every iteration, I am going to get only for the just we need to fill up the whole pipeline stages. So, the first output will come after. If it is K stage pipeline K for the rest n for the rest n minus 1 output, it will come in every cycle. So, this is the total cycle requirements right. So, you understand that.

So obviously, the resource requirement is more because you need 4 adders still and also this pipeline registers. But if this particular loop iteration is very high say 100, I am going to get a huge benefit, because after 4 stage I am going to get one output and it is basically will need only 104 iterations right. So, that is something, but sorry 103 iterations to complete this. Whereas, for the previous one it is 4 into 100, so 400 iterations.

So, you can understand that if this 4 is 100, the benefit is from 400 to 100. This is basically so, number of time now is needed exactly order of the number of iterations of the loop right. So, I hope I am able to make it clear the resource requirement for pipelining.

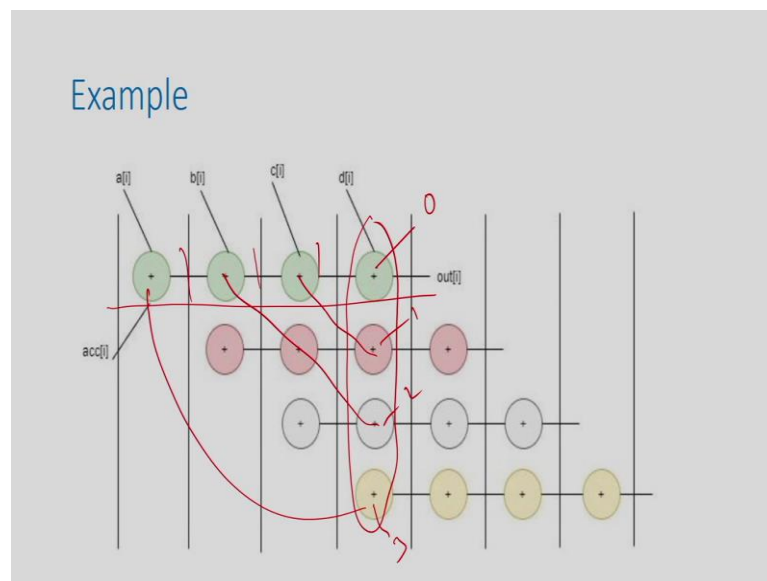
So, here if the pipeline initiation interval is basically 1, you cannot actually share this resource for every stages, but if the this for some dependencies where this

complete things is not possible, so in that case there may be some your initiation interval cannot be 1 right.

So, for example, if your one iteration depends on the previous iteration. So, loop current dependencies are there. So, until that particular data is ready I am not able to do the corresponding operation in the next iteration. So, there should be some kind of stall in the pipeline and as a result your II may not be 1.

In that case we have some scope to share the operations. But that is example I am not taking here, but I understand I mean I hope you understand the concept there. And your resource requirement might be little bit less if your II is not 1, but for complete pipelining and there is no loop carried dependencies you can actually give your input in every clock you can actually get your output in every clock and your resource requirement is actually maximum ok.

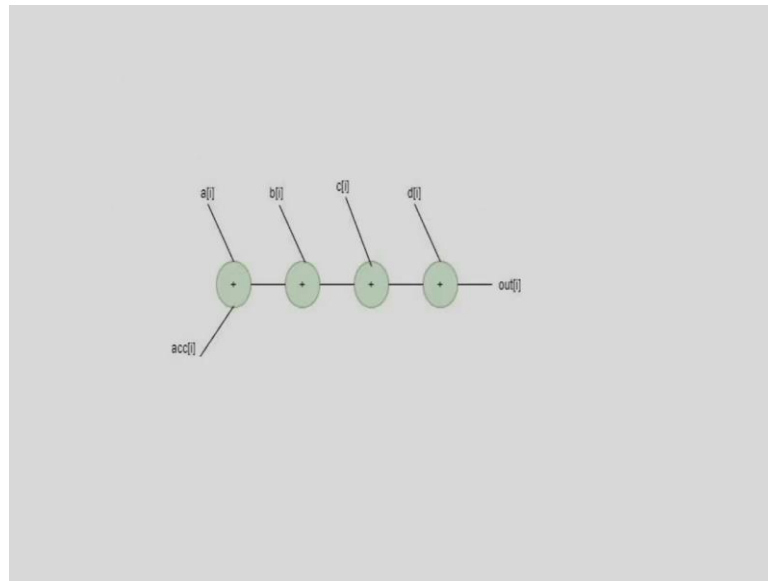
(Refer Slide Time: 30:39)



So, that is all about the resource requirements. So, what I am going to do is so, just to complete this diagram. So, actual execution is look like this right because this will. So, there are various stages. So, every stages 4 operation is 4 adder is actually active. So, this adder is nothing but this adder, this adder is nothing but this adder, this adder is nothing but this adder.

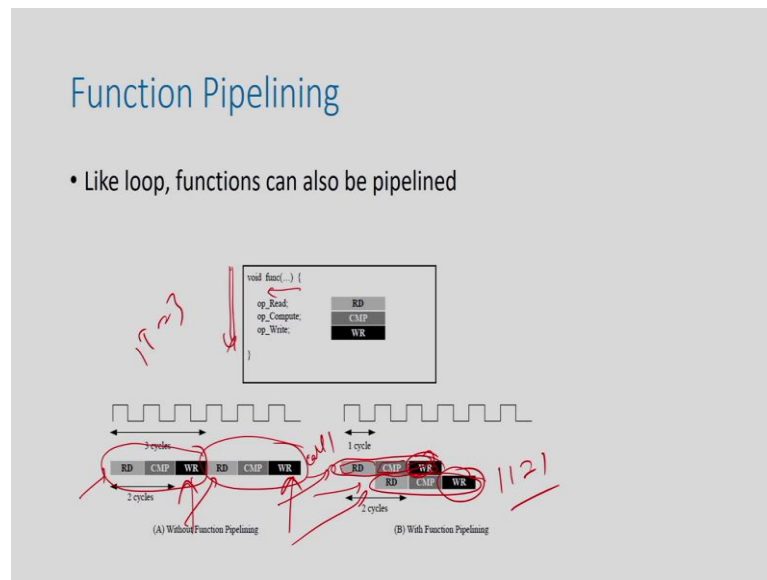
But this is working on the 3rd set of data, this is working on the 2nd set of data, this is working on the 1st set of data, this is working on the 0th set of data. So, this is what is the maximum overlapping of the iterations. So, the adder is actually connected this way because of this pipeline stages their execution is actually parallel right. So, they are now actually have that multiplexing of the inputs, but they are actually running in parallel.

(Refer Slide Time: 31:27)



So, this is what is the pipeline execution looks like for the example that I have taken ok.

(Refer Slide Time: 31:29)



Now, I will move on. So, similar to the loop pipelining you can actually do the function pipelining. So, whenever you have a function, so, in the function you can actually apply again the pragma loop pipeline. Then what is going to happen? The behavior is exactly the same the way I just explained for the loop. So, basically it will do the read, compare and write this is for the call 1, right.

And then you do a second call. So, when you are going to apply from the loop function pipelining; if you assume this there is a streaming application that you write a top function and every and the input is continuously coming. It is not that you just want to do things for one set.

So, one is one whenever there are one set of inputs whenever the output is ready you want to give the next set of inputs and so on right. So, basically it is a streaming data. So, for that function pipelining is really useful ok. So, it is basically you can understand the same thing. So, in normal time if you just try to call it two times, this is the execution of the first call and this is the execution of the second call.

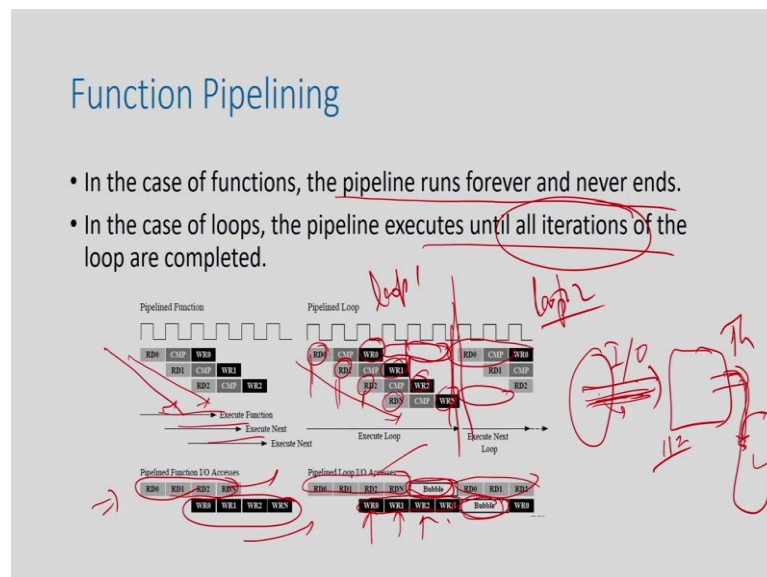
So, it will basically just like your iterative execution of the function. But if you just apply a loop pipelining, so, your this is the 1st call, this is the 2nd call, but they are now overlapped and you can actually give the input in every clock. Here you will give your input in 1st clock and the 4th clock right. So, because there is no function

pipelining implemented. So, your II will be 3 right because every after 3 clock you are going to give the inputs here its every clock your II become 1.

So, function pipelining, for functions for streaming application is really helpful because here your initiation interval is improves. As a result throughput is also improving right. So, now actually in every clock you are getting a output right. So, where earlier you are getting output in every 3rd clock.

So, this is what similar to functional loop pipelining. And here the loop function is basically the function pipelining indicates that I want to execute this different call of the functions in overlapped manner, right. So, if it is the application or the requirement you should do this ok.

(Refer Slide Time: 33:31)



So, here is the diagram that actually shows the difference between the function pipelining and loop pipelining. So, basically for the function pipeline we assume that this run forever right. So, I am assuming it is a streaming application continuously some inputs are coming I am going to process this, so, which is basically shown by this diagram.

So, your continuous this is the 1st call, this is 2nd call, this is 3rd call and so on and it is actually happening in pipeline manner. For loop it is a fixed iteration right. So, basically the pipeline is going to happen for the number of iterations right. So, this is

my loop 1 right. So, this is my loop 1, this is my loop say 2 this is a different loop right, this is not the same loop.

So, for the 1st loop the number of pipeline things will execute based on the number of iterations of the loop. Once it is done I am going to start the next code whatever is there after the code right. So, it is basically not that loop is running forever. It is going to pipeline for the number of iterations.

And based on that you are actually having; the based on your II value you will decide how often you are going to read the data it may be 1 or maybe 2 or 3 and so on. So, it depends on that. So, and if you think about that IO interface because once you apply pipeline you have to so, this is your loop body right. So, there may be some other part of the code which is giving the inputs here right and whatever the output is producing the other part of the MOS circuit you are going to get it.

So, you need to make sure that the previous circuit is actually giving the input the way your II requirement here right. And similarly whatever your throughput for this loop the data is ready for the next part. So, you have to make a synchronize between the components which is previously before the loop body or the hardware corresponding to the loop body. And similarly the code that is or the hardware that is after the loop body.

So, you need to make a IO synchronization right. So, for loop what is going to happen? So, this is if you see here I am going to read every cycle. So, this particular block which is producing the input for this loop should produce the input in every clock right because I am going to read in every clock. So, this is my this read pipeline access so every clock. So, this particular block has to give me this inputs in every cycle.

Similarly, the write is also happening in every clock right. So, you can see here the write is happening in this clocks. So, this particular model should be ready to as a receive the output in every clock right. So and so, that because it is also doing something on the output its producing right. So, it may you have to make sure that that particular module should have sufficient storage or the way it is been implemented. It should able to consume the output in every clock right.

And then once this is done there should be a bubble because there is no read here right. So, there is no read happening in this clocks right. So, in this clocks. So, there is a bubble here and then the next loop starts. So, then again the read happens. So, there is some bubble in the input as well as the output because whenever the next loop start there is no output for few cycles right. So, this is the bubble states.

So, this is what is the IO looks like and you once you design your whole complete circuits you have to make sure that your circuit actually is able to manage this IO access pattern. For function it since it is a forever thing its continuous happen there is no bubble right. So, it is going to happen continuously and this is write is also going to happen continuously. So, that is the difference ok.

(Refer Slide Time: 37:00)

Iterative Implementation of FIR

```

#define NUM_TAPS 4
void fir(int input, int *output, int taps[NUM_TAPS])
{
    static int delay_line[NUM_TAPS] = {};
    int result = 0;
    for (int i = NUM_TAPS - 1; i > 0; i--) {
        delay_line[i] = delay_line[i - 1];
    }
    delay_line[0] = input;
    for (int i = 0; i < NUM_TAPS; i++) {
        result += delay_line[i] * taps[i];
    }
    *output = result;
}

```

So, now I will move on. So, I have one example here for this that iterative implementation of the function versus this the function pipelining ok. So, I have a FIR implementation here. So, I will just briefly explain this. So, it has the number of tap is 4. So, what is happening here is there is a shift register here right. So, whatever the tap is there the delay line contain so, it will be shifted by 1 right. So, basically all the data is shift by 1 bit right. So, that is what (Refer Time: 37:30).

So, 0 1 2 3 and that this, so this 0 data will become. So, basically from this to this is happening. So, it is basically i equal to i minus 1 right. So, it is basically right shift

sorry left shift. So, whatever it is. So, basically we are shifting the data and then I am actually for d 0 I am taking new input right. So, what is happening here?

So, I have d 0 d 1 d 2 and d 3 and then in the next iteration. So, this is my new input d 0 and this is my whatever d 0 was there and whatever d 1 was here and whatever d 2, d 3 is discarded right. This is how the thing is happening. And then what I am doing is basically I am just doing a multiplication of the delay line with some tap value ok. Tap is another say constant value ok.

So, if you try to implement this as a iterative iteration. So, you need basically one multiplier and one add right. So, that is what is this circuit. So, you have a multiplier and there is a add. And then what is happening here? So, every clock you give this delay line i, this is my delay line implementation and then you do this tab 0. You are doing a multiplication then adding with the result and it was storing in the results. So, this is my result right. This is what is happening.

And then every whenever the next input comes then that input will come to that 0th location to the shift register right. So, for this iterative implementation you can understand that since it is a 4 tap it will take, so input will come in every 4 intervals. So, i your II is basically 4 and your latency is also 4 because you need 4 cycle to complete this. So, your latency is also 4 and throughput is also 4 right, because you are getting output in every 4 clocks right. This is the basic implementation.

(Refer Slide Time: 39:17)

Function Pipelined Implementation of FIR

```
#define NUM_TAPS 4  
void fir(int input, int *output, int taps[NUM_TAPS])  
{  
    #pragma HLS PIPELINE (1);  
    static int delay_line[NUM_TAPS] = {};  
  
    int result = 0;  
    for (int i = NUM_TAPS - 1; i > 0; i--) {  
        delay_line[i] = delay_line[i - 1];  
    }  
    delay_line[0] = input;  
  
    for (int i = 0; i < NUM_TAPS; i++) {  
        result += delay_line[i] * taps[i];  
    }  
  
    *output = result;  
}
```

Task Interval = 1
Task Latency = 1
4 times

loops in the hierarchy below are automatically unrolled.

Now, suppose you apply this pragma pipeline at the function level not inside the loop. You remember if you try to pipeline this loop you have to apply here, but here what you are doing, you are applying this at the function level ok. And also you remember that whenever you apply pipeline in the higher hierarchy the internal things will be unrolled ok. This is something is followed most of the high level synthesis tool.

So, as a result this loop will be unrolled right. There is no loop now. There are 4 multiplication addition operation and you can understand that this is given by this. So, this is the architecture I am going to get that now, since I have apply this pipeline here this is unrolled and this so I need 4 copy of this body. So, which is basically this, so, 4 multiplier followed by 3 adder.

This is the structure I am going to get, because initially my delay line is 0. So, that is why this adder is not needed. I can remove this adder because it is just the 0 right. So, this is why this optimize. So, this is the unroll version of the loop body. And the what is the advantage here?

So, I am doing everything in 1 clock, but I am going to give my input in every cycle. Because if you pipeline this with pipeline interval 1 so, you basically unroll the things. You now the requirement of the adder and multiplier is basically four times right.

So, there are four times more multiplier error needed because you have unroll the loop because I want to pipeline this circuit with II equal to 1. And similarly my latency will become 1 because I want to I have to execute everything in 1 cycle and then my throughput is also 1 ok.

So, this is what is the impact of this function you can understand. This is not a good circuit at all because your this delay of the circuit is addition of this 4 multiplier plus 3 adders. So, now it will be very slow circuit right, but this is. So, if you apply this II equal to 1, it will create like this.

So, what I am try to convince through this example that once you apply this pipeline the top level functions and the (Refer Time: 41:21) you need to be really careful about applying there. So, it may be because of this pipeline although your II become

1, it is accepting every clock, you are getting a latency become 1, but this circuit has lot of delay. So, it will be run very slowly your clock speed cannot be achieved with this ok.

Whereas this will take only it is only 1 adder 1 multiplier and you need 4 cycle, but it is at least 4 x faster than this circuit ok.

(Refer Slide Time: 41:49)

Nested Loop

Which loop to pipeline?

```
#include "loop_pipeline.h"
dout_t loop_pipeline(din_t A[N]) {
    int i, j;
    static dout_t acc;
    LOOP_1: for(i=0; i < 20; i++){
        (LOOP_2) for(j=0; j < 20; j++){
            acc += A[i] * j;
        }
    }
    return acc;
}
```

pipelining the innermost loop gives the smallest hardware with generally acceptable throughput for most application

When a loop or function is pipelined, any loop in the hierarchy below the loop or function being pipelined must be unrolled.

400
1 cycle

So, now since we are talking about the loop pipelining. So now, if you have a nested loop, so, where you should apply pipeline and what is the impact that should be understood ok. So, for the example that we have seen is basically only a single level of loop right. So now, you have a nested loop. So, there is a loop inside the loop. So, there is a loop i inside there is a loop j ok and I am doing some computation inside ok.

So, the idea for this loop let us try to say suppose I apply this pipeline here ok.

(Refer Slide Time: 42:23)

Loop Pipelining – nested loop:

Case 1: inner-most loop (loop_J) is pipelined

- Only one copy of hardware (i.e., 1 Multiplier, 1 Adder) is required.
- 1 cycle to read A[i], multiplier and Adder
- Total time required = 20 * 20 cycle
- Effectively all upper loops are also pipelined

Multiplier	Adder	Time
1	1	20 * 20 cycles

So, I just apply the pipeline at the innermost loop. So, then what is going to happen? So, effectively I am going to pipeline this loop. So, let us say I need some pipeline circuit, which need only 1 cycle for simplicity say to execute this. So, then how many cycle we need? I need 20 into 20 cycles because this both the loop is executing 20 times ok. So, I need 400 cycles, but I need only 1 adder and 1 multiplier right, I need 400 cycles.

So, the idea if you try to pipeline the innermost loop what is going to happen? Effectively the complete loop is basically a pipeline, because now every clock we are going to give inputs and nothings is going to happen. So, your resource requirement is actually minimum ok.

(Refer Slide Time: 43:10)

Loop Pipelining – nested loop:

Case 2: Outer loop (loop_I) is pipelined and inner loop is unrolled

- Each iteration of loop_I needs to be executed as single entity
- The 1 cycle to carry out 20 memory access, 20 multiplications, 20 additions
- Total cycle required = 20 cycle

Multiplier	Adder	Time
20	20	20 cycles

So, now if you just try to give a pipeline to the upper loop right, so, loop I right. So, I want to pipeline this loop now, not this loop ok. So, then whenever you try to apply the pipeline here the by default this loop will be unrolled. So, this loop is not there. So, there are 20 copies of this addition and multiplication operations right. So, and now you want to pipeline the things. I want to give your pipeline your input in every clock. Then what is going to happen?

You need 20 copy of the multiplier and 20 copy the adder, because that is what you are doing because you are effective means of applying this pipelining to the loop i means you are unrolling loop j. And hence this 20 operation is running in parallel, so you need 20 copies of the resource. So, you need 20 adders 20 multipliers, but you need only 20 cycle because the outer loop is only 20 cycles ok.

So, now if you apply this pipeline here right not in this loop on this loop on the function level. So, if you apply there what is going to happen? Both the loop will unroll right. So now, there are 400 operations. So, 400 multiplication operation, 400 addition operations and you want to pipeline it, because you want to give your input in every clock. So that means, you need to perform this 400 multiplication and 400 addition in the same clock right.

(Refer Slide Time: 44:27)

Loop Pipelining – nested loop:

Case 3: Function is pipelined

Multiplier	Adder	Time
20*20	20*20	1 cycle

- Not feasible to generate hardware with 400 multipliers

So, then the resource requirement will become 1; so, 400 adder 400 multipliers and only 1 cycle. So, you can understand that this is not a feasible circuit. So, if you just give this function pipelining at the top, probably your resource available in the target architecture may not be sufficient to map all this 400 multiplier and so.

So, what I am try to convince is that whenever there is a nested loop pipelining the innermost loop is most of the times its acceptable and that is what people usually do. And that will give you the smallest hardware, but generally the acceptable throughput right. Because you are actually executing using minimum resource and throughput it will be good because pipeline will actually overlap the multiple iterations.

So, the example that I have taken here its only 1 cycle, but if it is a K stage pipeline you will see the benefits right. So, if you assume this in the body will take K stages the advantage you can actually see over the iterative iterations. If it is a 1 cycle it is actually become iterative iterations right. So, there is no benefit you are going to get. But once it is some stages K stages or 5 stages 6 stages then the benefits you will see over the iterative executions.

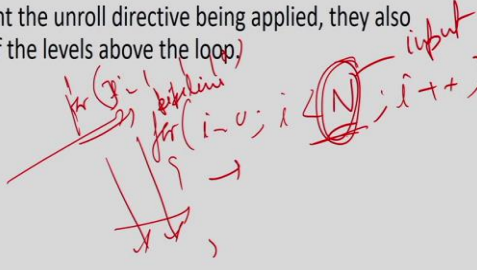
So, for nested loop the smart idea is to pipeline the innermost loop because if you do not if you try to do the upper one the inner one will unroll and your resource requirement will become more. And if you try to do the function level pipelining

your resource requirement become infeasible may be infeasible ok. And so, that is the conclusion of this slides.

(Refer Slide Time: 45:58)

Variable Loop Bounds

- The first issue with variable loop bounds is that they prevent HLS from determining the latency of the loop.
- Variable bounds loops cannot be unrolled.
- They not only prevent the unroll directive being applied, they also prevent pipelining of the levels above the loop.



So, now one more important point to understand is the variable loop bound. Because, many of the time we will write a for loop where I just write $i = 0$ $i < N$ $i++$ and this N is an input. If it is an input then it is a problem right. So, if it is an input what are the problem I am going to see? If I want to unroll this loop, I cannot right, I do not know how many times.

It might be unrolled 10 times for when N equal to 10, it might be unrolled 10000 times when N equal to 10000. So, basically if it is N is an input or it is basically variable. And if you try to unroll tool cannot unroll. It will simply say I cannot unroll because this loop bound is not a static value, it is a variable value ok. And as a result only the iterative iterations is possible.

There is one more problem. So, once you try to pipeline this design and say you have another upper loop say which is say for i equal to 1, j equal to 1 to 10 say and I want to pipeline this loop right. If you want to pipeline this loop the by default this will be unrolled right and since I cannot unroll I cannot apply the pipeline here also right.

So, it will actually stop me pipelining the upper or outer loops as well right. So, that variable loop bound not only stop unrolling, but because unrolling is not possible the applying pipeline to the outer loop is also restricted because of this variable number. Also because this N is not known, the total latency also cannot be estimated right.

So, from the runtime once you synthesize the hardware the tool cannot tell me what is the total latency of the design, because you do not know the N right. So, if you say it has come up with a iterative implementation, so, you do not know how many clock cycle it will take to complete the design at the static time. So, if you give some value N it will takes a 10 cycle, if you give 20 it will take 20 cycles and so on. So, even this latency cannot be determined for the generator hardware ok. So, this is what is the problem.

So, in general the idea of in the hardware synthesis that we should not use variable loop bound, because for my applications I know what is the upper limit of the value and let us use that ok.

(Refer Slide Time: 48:13)

Variable Loop Bounds

- The loop bounds are explicitly set to the maximum value of variable width and the loop body is conditionally executed:

```
#include "ap_cint.h"
#define N 32
typedef int8 din_t;
typedef int13 dout_t;
typedef uint5 dsel_t;

dout_t code028(din_t A[N], dsel_t width) {
    dout_t out_accum=0;
    dsel_t x;
    LOOP_X_for (x=0; x<width; x++) {
        out_accum += A[x];
    }
    return out_accum;
}
```

→

```
#include "ap_cint.h"
#define N 32
typedef int8 din_t;
typedef int13 dout_t;
typedef uint5 dsel_t;

dout_t loop_max.bounds(din_t A[N], dsel_t width) {
    dout_t out_accum=0;
    dsel_t x;
    LOOP_X_for (x=0; x<N; x++) {
        if (x<width)
            out_accum += A[x];
    }
    return out_accum;
}
```

The diagram illustrates the transformation of a loop with a variable bound (width) into a loop with a fixed maximum bound (N) and a conditional body. Red circles highlight the variable 'width' in the original code and the constant 'N' in the transformed code. Red arrows indicate the mapping of the loop body and the conditional execution logic.

So, I am given a very good example here which basically says that suppose even if you do not know the value, but you know the maximum range. So, how can you rewrite the code, to make sure that your code actually solve all the problem that is written here, ok? So, for example, say suppose in this code I have loop and this is the it depends on the input, right.

So, there is a function and this is a variable loop bound and I know this maximum value is actually $N/32$. So, the number maximum value this loop can execute 32. So, what I can do now? I can run this loop for 32 times and I am just put a condition here that if condition is less than width which is my input you do this operation right, you do this operation.

So, then even if you try to unroll it will unroll the loop right because this N is 32 fixed. So, also I can actually assume and I can actually guess the latency and also I can do the pipelining. Although there will be a conditional check which is a little bit of additional hardware, but the problem of unrolling problem of applying pipelining the outer loop or estimating the latency of the design all can be done, right.

So, the bottom line here is that if we should use a static loop bound if possible if it is not possible you should estimate what is the maximum limit and I should rewrite my code this way so that it does not stop me applying the loop optimization ok.

(Refer Slide Time: 49:39)

Summary

- Pipelining a loop gives best performance in terms of II, throughput and area.
- Pipelining the innermost loop gives the smallest hardware with generally acceptable throughput for most application
- When a loop or function is pipelined, any loop in the hierarchy below the loop or function being pipelined must be unrolled
- Variable loop bound may hamper the application of loop optimizations like unrolling, pipeline

So, just to summarize. We understand that pipelining of loops give the best performance in terms of throughputs initiation interval and also area ok. And also what I understand that whenever try to pipeline the innermost loop that give you most of the even the best hardware, because it is the minimum hardware resources and it is most of the time the if the number of pipes line stage is more and then the performance is also acceptable ok.

And whenever there is a nested loop or nested function, so, if you apply the outermost loop the rest of the things inner loop will be unrolled and hence you might have a see the exponential growth of the resource requirement. So, you need to be very careful about applying pipelining on the outer loops, because that is may give a exponential or a huge resource increment ok.

And also we understand the variable loop bound actually causes problem in unrolling and pipelining and latency calculation or estimation. So, most of the time if possible we should make our loop static bound, if not we should rewrite our code with a upper bound given to the code so that resolve those problem ok.

So, with this I conclude today's discussion.

Thank you.