

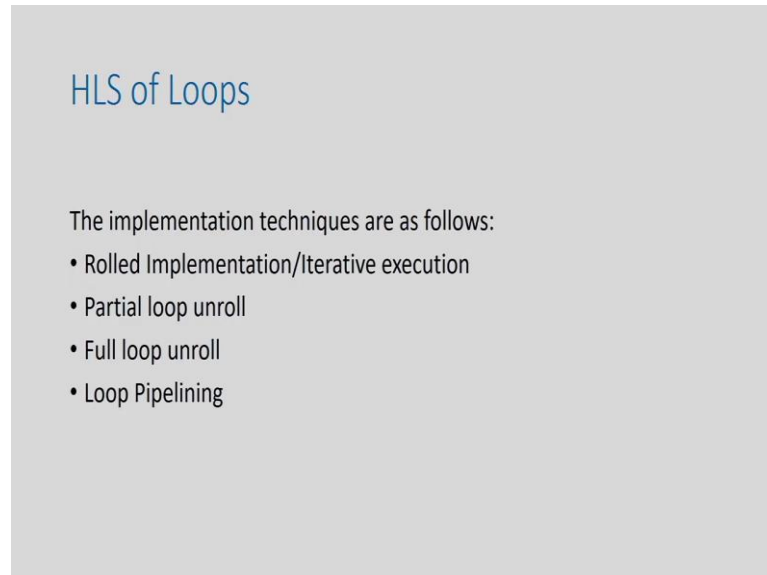
C-Based VLSI Design
Dr. Chandan Karfa
Department of Computer Science and Engineering
Indian Institute of Technology, Guwahati

Module - 07
Efficient Synthesis of C Code
Lecture - 24
High-Level Synthesis of Loops

Welcome everyone to my course C based VLSI design. In today's class, we are going to learn about High Level Synthesis of Loops. So, as you know this high-level synthesis is an process of converting a C code into equivalent RTL code.

So, in the C code one of the important construct is loop right and the in this class, we will try to understand how this loop will be effectively mapped into hardware and since loop is something is iterative and its efficient executions is very important, otherwise it might has hindered the performance ok.

(Refer Slide Time: 01:19)



HLS of Loops

The implementation techniques are as follows:

- Rolled Implementation/Iterative execution
- Partial loop unroll
- Full loop unroll
- Loop Pipelining

So, the loops are actually implemented in the hardware in four different ways. One is the rolled implementation or the iterative executions and then, partial unrolling, full complete loop unrolling and loop pipelining. In today's class, we are going to discuss about this rolled implementation, partial loop unrolling and full unrolling and in the next class, we are going to discuss in detail about the loop pipelining ok.

(Refer Slide Time: 01:49)

Some important Terms

- **Initiation Interval (II)**: Number of cycles between starting of two consecutive iteration.
- **Latency**: Number of cycles needed to process one set of inputs.
- **Throughput**: Number of cycles between two outputs. The interval in which outputs are produced.

Before going into the discussion I am going to introduce certain terms which is important in the context of these loops and their execution. In hardware, the first term is called Initiation Interval ok. So, the initiation interval says that how number of cycles between starting of two consecutive iteration right. So, in the loop there are multiple iterations, right. So, the II the or initiation interval says that how frequent the iteration starts ok.

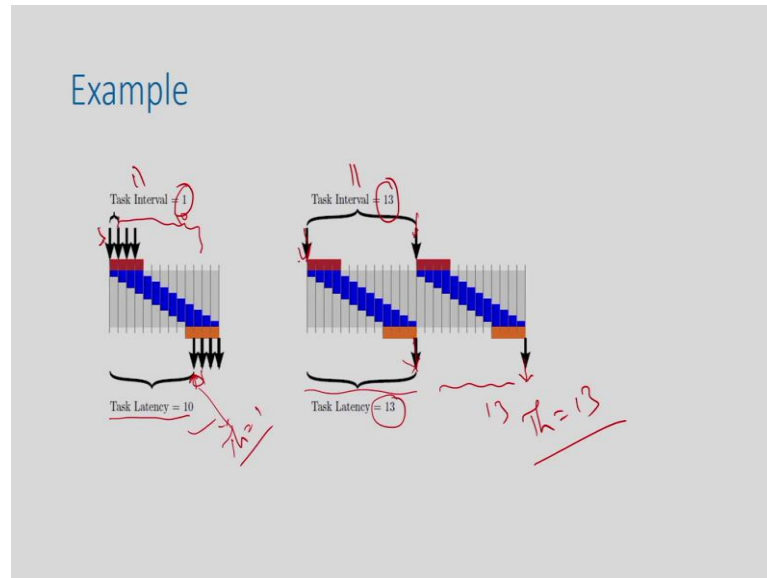
And Latency is says the number of cycles needed to process one set of inputs. This is a number of clock cycle. Once you give an input, what are the number of cycles you need to get the corresponding output ok. So, that is called latency and then the another important term is called Throughput. Throughput is a there is the number of cycles between two outputs right. So, you are getting output 1 corresponding to input set 1 and we are getting output 2 corresponding to input set 2.

So, what is the time difference between these two outputs? Remember this latency may and this throughput is always not the same because a latency is something say you are giving some input and it needs a 10 cycles to get an output.

So, the latency of the design is kind of 10 cycle, but your throughput can be 1 because if you design the circuit as a pipeline mode, then what is going to happen that your whenever your first output come in the next clock, your second output might come if it is a optimal pipelining right.

So, your throughput will be 1 although your latency is 10, ok and initiation interval also not always same as latency because so it depends on how you are going to execute the design, ok.

(Refer Slide Time: 03:31)



So, let me take two examples to explain these two terms in more detail say you take the left hand side example where I am actually having a pipeline execution of the loop body, ok. So, what is happening say suppose that as the example. I have given that you would say that if you give an input, you need ten cycle to get the output ok to process the input to obtain the output in a 10 cycles, ok.

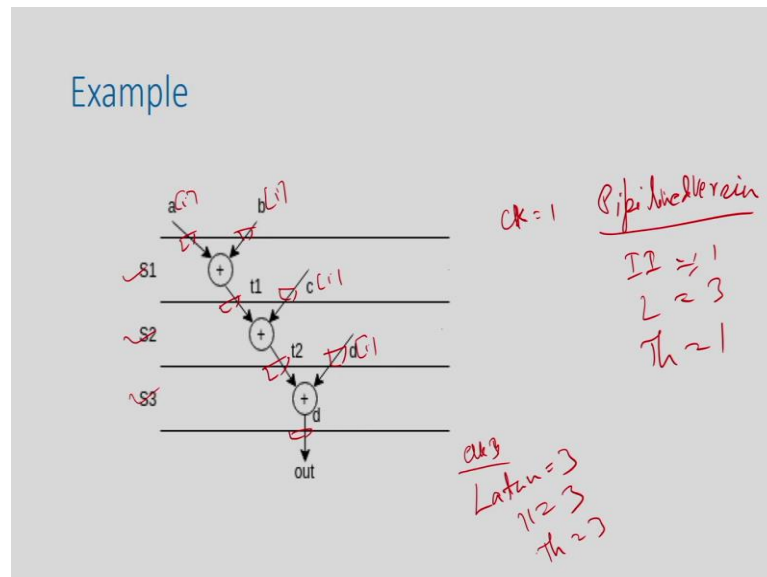
So, you are giving the input one here and you are getting your first output here, ok. So, that is the latency 10. You are giving your input here and you are getting your output again. It is 10 cycle right. So, distance between the first input to first output, second input to second output is 10 cycle.

So, the latency of this design is 10 cycle. So, what is the initiation interval? Because you are giving the inputs in every cycle because its a pipeline executions. So, the initiation interval or task interval is 1, ok and also you can see the difference between the output 2, output is also 1 throughput is also 1 for this design. So, this is the pipeline, a execution of the loop right.

Now, you think about the iterative iteration of the loop. So, you are giving the first input here and you are getting your first output here and say you need some time to read and all. So, total latency is 13 cycles ok and after 13 cycle, you are giving the second input and you are getting the output again. So, again the latency is 13 cycle.

Now what is the Initiation Interval? It is basically 13 because you are giving input here and then you are giving input after 13 cycle, right. So, here initiation interval is 1, here initiation interval is also is 13 ok and the throughput is also 13 in this case because you are getting throughput every 13 cycle right not in every cycle, ok.

(Refer Slide Time: 05:28)



So, let us take an another example say suppose this is the loop body, ok. So, this is the loop body and this is the schedule you need 3 cycles to execute this, ok and assume these are all array basically some data which is basically available right. So, now what is the if you execute in an iterative manner iterative execution. So, what is going to happen? So, you give one set of data at clock 1 right, clock 1 and your output will come so 1 2 3.

So, after third clock cycle, only your clock 4 you are going to get the first output at the start of clock if you assume there is a register here, right. So, otherwise you can assume that it is after third clock is available let us say there is no register, ok. So, that means here and then after. So, your latency of this design is 3 because there are 3 cycles ok to process the data and since this design is not pipelined.

So, your initiation interval, that means you have to wait until these things are processed. You have to wait and then only you can give your next set of input, ok. So, then the IR, II is also 3 and throughput is also 3 because every third cycle you are going to get the output. Now suppose I make this design pipeline, right. So, I am adding this pipeline stages everywhere, then what is going to happen?

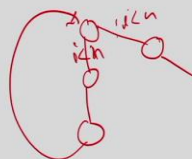
So, now I can give input in every cycle because this stage 1 will process one set of data, this stage 2 will process another set of data and stage 3 will process another set of data, right. So, for pipeline version we have the initiation interval is also 1 and then latency is also latency is 3 because for every set of data you need 3 cycle to get the output, but your throughput will be 1, ok. So, this is how we calculate these values and these are the important parameters in context of loop execution, ok.

(Refer Slide Time: 07:38)

Rolled Implementation/Iterative Implementation

Steps:

- Schedule the loop body.
- Allocate the resources accordingly.
- Execute the loop in $n*k$ cycles, where 'n' is number of loop iterations and 'k' is number of cycles required for each iteration.



Now, let us move to the first version of the loop which is called Rolled Implementation or Iterative Implementation. So, this is the by default executions for loop. So, loop is basically have iterations, right. So, its running for say n iterations and 1 iteration, there may be some 10 operations, right.

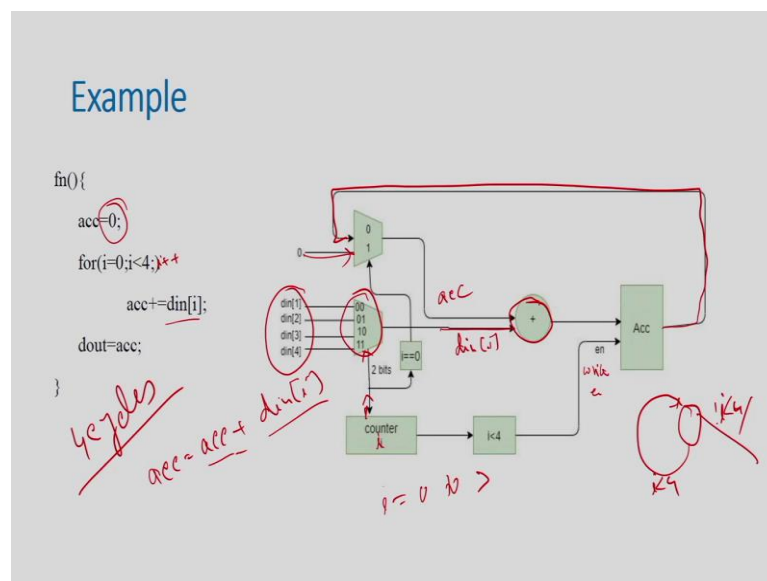
And now you based on your resource availability, you have to you can schedule that particular loop body and let us say that scheduling that loop body 1 iterations of the loop take k cycle. So, that means you give one set of input, you run k cycle. One iteration is

over, then you run the next cycle and so on. So, the overall time it will take is n into k , right.

So, the overall time will be taken in this way is basically n into k and it is the most simplest version of the loop, right. So, you have you the schedule the loop body whatever the say it takes a 3 cycle to execute and then, you just loop it for n times right. So, if n if i less than n you do this and otherwise I less not n , then you go to the other part right.

So, this is how the controller FSM will be generated and it will keep executing the loop for n into k minus cycles and the rest of the things, we will wait until the loop complete its execution right because this is how I am going to execute the loop, ok. So, this is how this is what the basic or the default implementation of the loop.

(Refer Slide Time: 09:07)



Let us take example to understand better say I have this loop given where I am doing some accumulations of this din and since this din is the small array. I am assuming this is flattened, ok. These are actually put into registers ok. Now what I am going to do? So, basically in every iteration what is happening?

So, acc equal to acc plus d in i , right. So, I need one of addition operation and that this is operation. So, the scheduling we will take only 1 cycle right since I am assuming this array is splitted into registers. So, this particular for this adder one input will be the

accumulator, right. So, this accumulator is coming here like this. So, I have just put 0 and otherwise because the accumulator initially should contain 0.

And whenever the initially I can reset this accumulator with 0, so I just choose a multiplexer to reset this. So, this is your basically accumulator and the other input is basically din and now, you have to choose this among din_1 to din_4 . So, you need a mux, right.

So, that will choose the appropriate value from din_1 to din_4 and then, this will send it here. So, this is my din_i basically, right and this calculation will be done and it will put into the accumulator right and we have a counter i which is basically will count from 0 to 3, right.

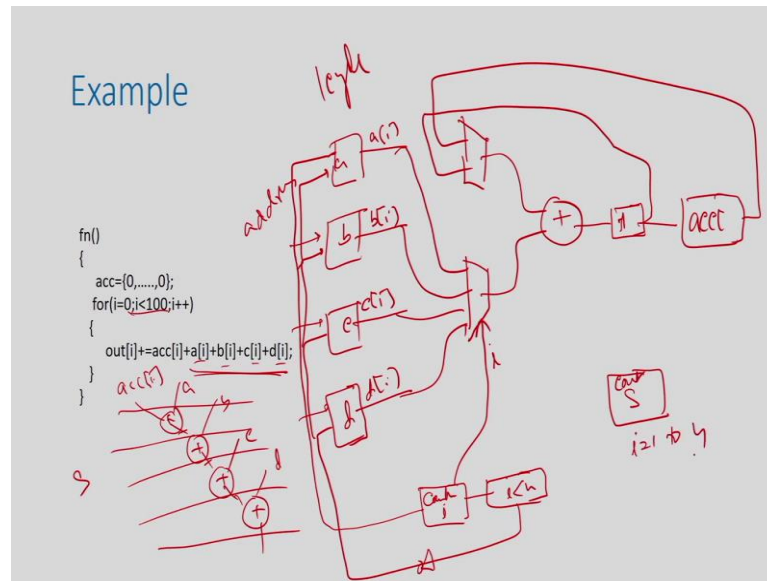
So, whenever this i so it will automatically increment from 0 1 2 3. So, whenever it is basically become 4, it will automatically reset to 0 right. This is how that this counter will work and that particular counter this counter this direction should be in the reverse.

So, this will be the selection of this data right because if it is i equal to 0, I am going to, so if it is 1, I am going to select this and so on right. So, this is how I can select the data based on this i value and also this i less than 4 will be the write enable for this accumulator register accumulator is a register.

So, till the time it is less than 4, I am going to write after that I am going to stop. So, this is how I am going to execute the behavior in the hardware, ok. And you can understand that this is basically a single loop operation and this is basically looping here right. So, for i less than 4 and once it is not, I will go to the rest of the thing right. So, this is how I can actually store the when execute this and it will need since there are four operations.

So, you need 4 cycle, right. So, you need 4 cycles to execute this. So, this is very simple and you need minimum resource. The important factor here is that it need minimum resource because you are taking only one iteration of the loop at a time and you are actually assigning resource for that right, but it takes lot of time because it is take maximum possible time to execute the big array, right.

(Refer Slide Time: 12:03)



So, let us take an another example where I can actually do some resource sharing. So, I will just show the execution of this part. So, since this is the big array and these arrays are big, so I cannot put them into, I should not split them into register because it will consume lot of registers right.

So, I have now 4 arrays array a b c and d right. So, I should have that counter. So, I am writing some abstract design. So, this counter will decide the address, right. So, this is the address right. So, this is address basically.

So, if I give this i, so it will just select a i right and there should be some write enable. So, I can actually just check whether i less than n or not. So, that will be my this enable signal. So, this is read enable right. So, this would be, so I am going to say that this is basically a negation of this data.

So, I will say that this is a read you just read from this data and then, I will get this a i b i c i and d i here, right. So, this is how I should access this. I am assuming this arrays are mapped to block ram ok. So, now I have all this data and it will need 1 cycle right because these are all 4 parallel ram. So, this is one, this you need 1 cycle right and then I want to schedule this. So, this scheduling can be you can actually draw different kind of thing right.

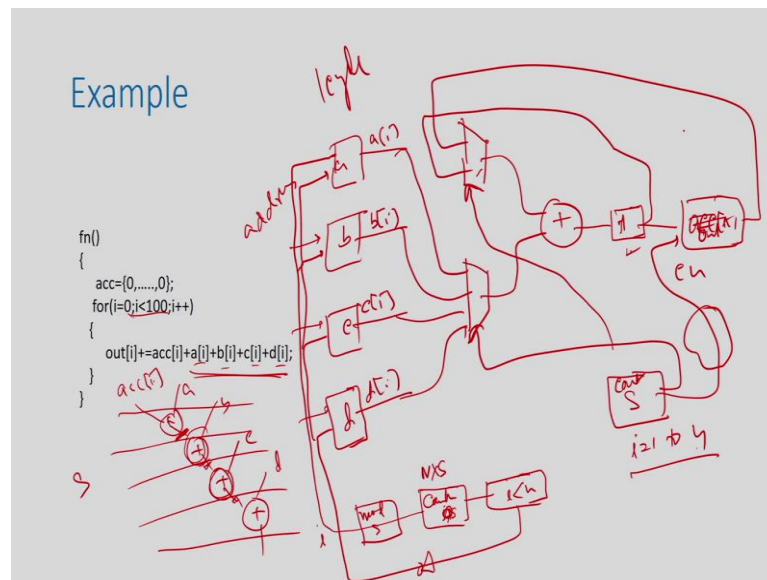
So, basically it can be like this right. So, you are doing say acc a, then b, then c, then d right. So then if you try to do this, you need basically 4 cycle right. So, you need 4 cycle, but you can actually do these things using a single adder because it is a single addition operation and you can actually mux multiplex the input, right. So, let me try to do that only. So although there are 4 operations, I can actually execute in a single resource. So, let us say I have one adder.

So, what is the input? So, I have the left input is basically is that accumulator. So, whatever the accumulator so basically here this is accumulator i, right. So, I am updating the accumulator i, but what I can do is I can store this register in a temp register, right. So, I am going to store this register in a temp register right. So, this is my temp register and then I have a mux which will select between this accumulator or this temp, right, so or actually I do not need this accumulator here.

So, I can actually just select this temp because initially this temp value ok I also need the let us put it like this. So, this is a abstract design do not get much. So, there may be some minor bug, but it is ok the basic idea I am just try to convey. So, this is what is the mux, right. So, this is again controlled by this i right. So, this i sorry this is ok. This is basically not I you should have a timer for this scheduler, right.

So, there is a scheduler time. So, there is s which is basically there is a counter s which is learning from 1 to 4 because there are 4 cycle, right. So, that will be control by this schedule time ok. So, then what is happening here?

(Refer Slide Time: 15:50)



So, and this I can also have some way you can control whether you take the acc i or t, right. So, what is happening in the first clock, you read all those things. So, the read will be done in say first clock. In the next clock, you are actually performing this acc plus a i right.

So, this mux will select acc i and this will select a i and that summation. This result will be stored in this t right and then in the next clock, this s will decide and this basically this counter would not increase until. So, this is counter is basically s into i into s, right. So, this is not basically n into s and you can actually do a mod here, right.

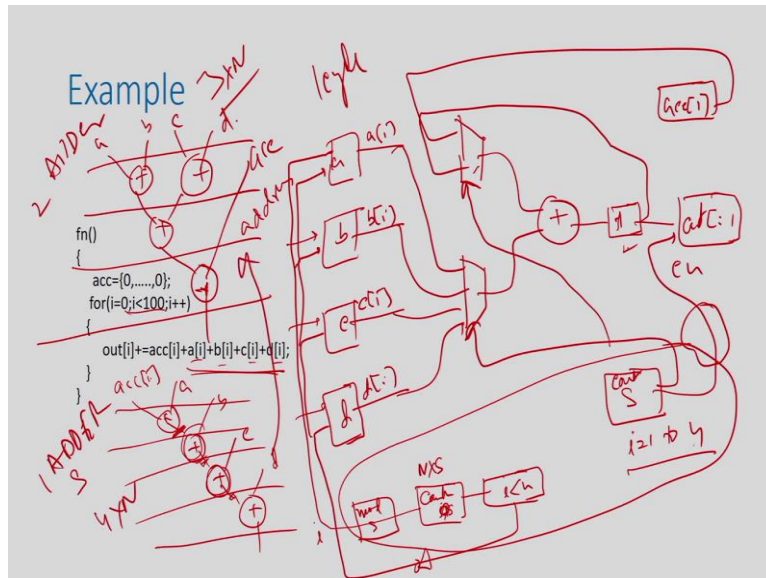
So, s mod here mod s because this I will increase once one execution is over. So, the so basically you can assume this counter is n into s and this I will be this is my i which is can be obtained by just do a mod s, right.

So, after every 4 cycle, you increase i right. This is how I am trying to mean this. So, next I will increase this s and then now it will select this t here and this b i here and then, it will do this operation and store this result into t again, right.

And then again in the next clock this t will be come here and c i will come here and it will do this operation and you can do this and then, last cycle you can do this and it will come here in the next cycle. Somehow you can actually use this s to put a enable here.

I mean some version of some version of this, so that you write this data into acc I. This is the final value right or this is actually I am sorry this is actually out i right and your acc i is something another input basically sorry.

(Refer Slide Time: 17:55)



So, this is actually out i because this is the output and say acc i is another array which will store like this and then this will store in out i only, right. So, this is contain all zeros basically. So, this is an very abstract level design. So, this counter control things. We have to accurately design, but the overall idea of this counter control thing is that I will increase only to i plus 1 once this one iteration is over, right.

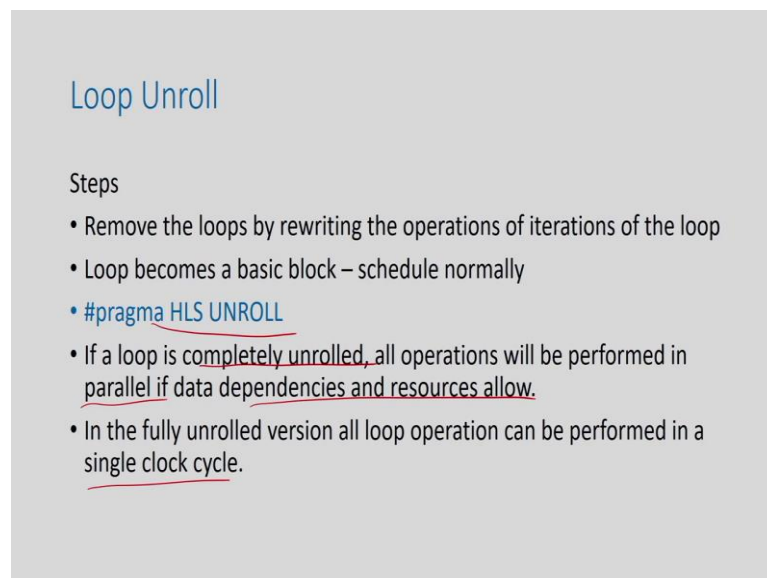
S is done and then, I can actually take i 2 and again I will allow it for 4 cycles, but the important factor is that although there are so many operations and i since I try to do this in say 4 cycle, these operations I can actually execute these things using a single adder, right. So, the basically the objective abstract of this discussion is to emphasize that even if there are many operations and if I do a rolled implementation my resource requirement is minimum, ok.

You can also take this diagram and actually try to draw this version of this dependency right. So, you can actually do this right. So, this is basically a i b i c i and d i right. So, you do these things over here and then basically you do this addition here. So, then this is done and then you actually add with acc right. So, then you just do add with acc. So, then in this version you need 3 cycle instead of 4 cycles.

So, this is what is called tree height reduction. I reduce the height of the tree. So, since here there are two parallel operations, I need at least two adder right. For this design I need only one adder, but here I need two adders because there are two operations running in parallel. So, based on your dependency or you can actually modify the dependency based on your resource availability and actually you can improve the performance.

So, this will take you can understand if it is a 4 cycle 4 into n cycle, it is 3 into n cycle right and if you have two function which is available, why not doing this right. So, this is the idea.

(Refer Slide Time: 20:09)



Loop Unroll

Steps

- Remove the loops by rewriting the operations of iterations of the loop
- Loop becomes a basic block – schedule normally
- `#pragma HLS UNROLL`
- If a loop is completely unrolled, all operations will be performed in parallel if data dependencies and resources allow.
- In the fully unrolled version all loop operation can be performed in a single clock cycle.

So, the key takeaway from this discussion is this loop rolled. Implementation of the loop is the basic way of implementing things. It takes maximum number of iterations, but it takes the minimum resource ok and based on your resource available, you can actually modify the dependency to improve the time requirement per iteration.

So, I just modify these to this to reduce the time from 4 to 3, right. So, that was the idea I will now move to the other extreme, right. So, this is the rolled implementation and I am going to talk about the full unroll implementation, right.

So, that means in the full unrolled version there is a no loop. I am just removing the loop and I am writing the all-iteration operations of all iteration together and I will just

completely remove the loop, right. So, then what will happen, it is basically become a basic block right. It is a series of sequence of operations which is something is there. So, that is what is the another extreme end.

So, you can understand here that in practice if there is no dependency between 2 iteration, this the iteration i and iteration i plus operations are in parallel right. So, if I assume that one iteration take 1 cycle. So, I can actually execute the complete loop body in single cycle right.

So, the earlier time it was n cycle. If I assume each iteration, take 1 cycle then it is n cycle, but here it is 1 cycle. So, it is a complete extreme end, but there I need only say 1 unit of resource here, you need a any unit of resource because this all n iterations are running in parallel ok.

So, this is the another exchange. So, usually you can actually do, you unroll yourself or you can use the pragma HLS UNROLL of Vivado just to do that you just mention this within the loop it will completely unroll the loop.

So, the point here is that if you is since you are completely unrolling, so it is actually improved the parallelization because if the loop iteration there is no inter loop dependencies. The whole loop is basically running in parallel right.

So, if your you have this resource allow, then all operation can be done perform in parallel right. All iterations can be done in parallel and in best case, it can actually complete the whole loop in single cycle. If I assume that there is an operation chaining allowed and one iteration take 1 cycle to execute, ok.

(Refer Slide Time: 22:29)

Example

```
for(int i = 0; i < N; i++) {  
    a[i] = b[i] + c[i];  
}
```

For unroll, N must be static.
Arrays will be mapped to registers.
Effective for small loops

$a[0] = b[0] + c[0]$
 $a[1] = b[1] + c[1]$
 $a[2] = b[2] + c[2]$
 $a[3] = b[3] + c[3]$

4
1 cycle
4 Adder

So, let me take examples. So, suppose you have this loop and one important point here is that for unknown you need to know the it should be static, right. So, unless you know this is fixed value I cannot unroll the loop, right. So, that is the requirement and then now you assume say there is this loop, right. So, now I suppose this is 4, right and.

So, then in the 4 iterations so if I unroll the loop, content will look like there is a no loop. The code will look like this a_0 equal to b_0 plus c_0 a_1 equal to b_1 plus c_1 a_2 equal to b_2 plus c_2 and a_3 equal to b_3 plus d_3 , right. So, there is no loop. I have this and since there is no dependency between this, so I can actually add I have say 4 adder operator and I can actually do this.

So, this is a $0 b_0$, I am just writing in short, $a_1 b_1$ this is $a_2 b_2$, this is $a_3 b_3$ and you will get sorry this is, sorry this will be this should be b and c right. So, this is $b_0 c_0 b_1 c_1 b_2 c_2 b_3 c_3$ and this is $a_0 a_1 a_2 a_3$, right. So, this is what we can just do and you can actually understand that I can actually do the things in single cycle, ok but your resource requirement is 4, right.

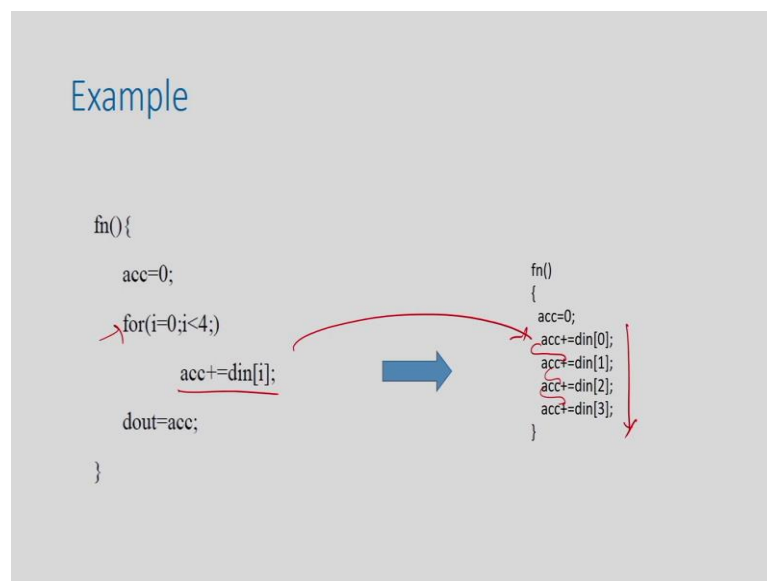
So, if you try to execute this in an iterative manner, you need only one basically adder and you need basically a mux which will basically or basically you have a ram, you basically give I you will get b_i and you have a ram which is storing say c and this is b , you give the address, you get c_i right and you just add it and this will be stored in a and you give the address i , right.

So, this is the execution. So, this will take n cycle. So, n say 4, so I will say you just take 4 cycle, but 1 adder right. So, the 1 adder here it is 1 cycle 4 adder, right. So, this 1 cycle is the extreme end because I am assuming that there is no dependency between the inter iterations.

But if there is a dependency between iterations, so you may not able to execute everything in single cycle and that is a next example highlight that, but try to understand that this although I get the maximum timing benefit by my resource requirement also increase in the same factor, right.

So, the performance increment in terms of timing or latency is same, as the same factor of resource requirement ok. So, that is something obvious because you want to execute in parallel.

(Refer Slide Time: 25:51)

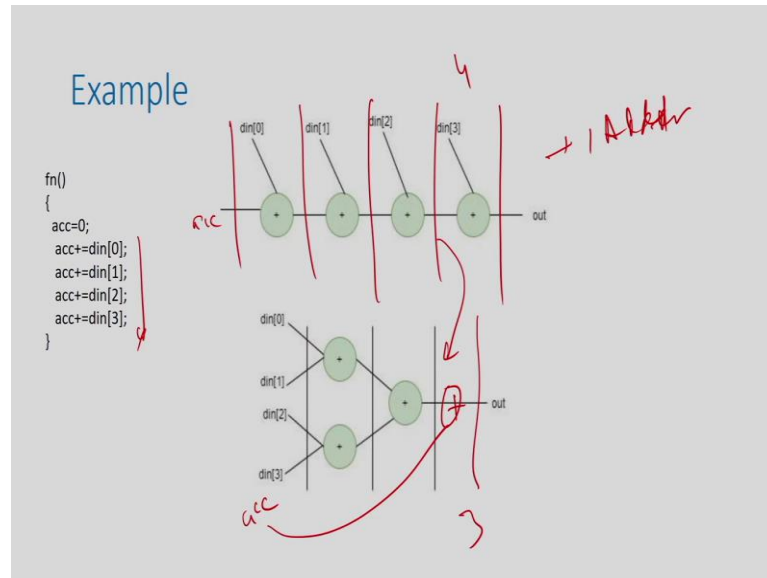


So, now I take the second example or the first example that I have taken earlier that you just do this accumulator d in and if you unroll here, you will get this right. So, basically there is a no loop, right. So, you just have there is no loop. It become a straight line of code, but here you can see that this iteration that because it is accumulator equal to accumulator plus d in.

So, whatever I am calculating accumulator in i 0 i 1 will depend on that, right. So, it is basically you can see here this accumulator is getting used here, this accumulator is

getting used here, this accumulator getting use here right. So, as a result so there is a inter iteration dependencies. So, I cannot execute this if you want to execute. So, the dependency looks like this right.

(Refer Slide Time: 26:32)



So, you have this, but this is your accumulator. So, there is a dependencies. So, in general I have to schedule in 4 cycle right because if I assume there is that operation chaining is not allowed, then you have to execute in 4 cycle. So, your although you unroll, you do not get any benefit in this case right. So, apparently and you can actually still execute this using 1 adder because you can actually multiplex this inputs time division multiplexing of the inputs of this adder, ok.

So, that is one a possibility, but usually once you have these, you can actually rewrite this dependency into this right. So now you can see here instead of 4 cycle, I can execute the things in another there is a mistake here. So, the acc will come here in 3 cycle right.

So, it was 4 cycle, this is 3 cycles. So, after doing this although there is a loop carried dependencies after unrolling, I can actually restructure this dependency graph to improve the performance as well right. So, if the basic idea is that if there is a loop carried dependency unroll does not help much right.

So, that is the basic take away from this discussion right, but if the loop is small and the loop there is no dependency among the operations or the iterations operation into the

even iterations, so then this unrolling give you the maximum benefit, but as I shown in this example that if there is a loop carried dependency unrolling may not be that much helpful, ok. So, I will move to the middle path what is called partial unrolling, ok.

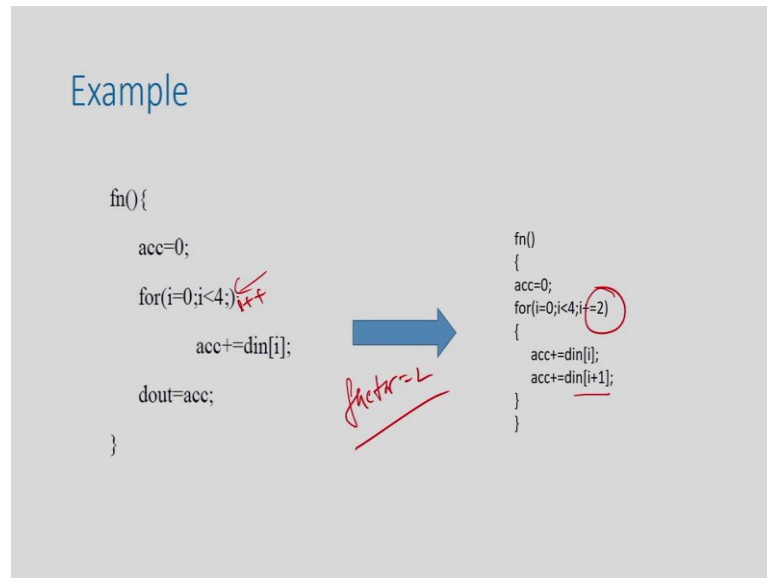
(Refer Slide Time: 28:12)

The slide is titled "Partial Loop Unroll" in blue text. Below the title, it lists "Steps to be followed:" with two bullet points: "Unroll the loop by unroll factor." and "Follow the same steps as rolled implementation technique." At the bottom, there is a code snippet: "#pragma HLS UNROLL factor=8". The number "8" in the code is circled in red. To the right of the code, the number "64" is written in red and underlined.

So, the partial unrolling basically is a tradeoff between iterative execution versus complete unrolling, right. So, we have seen that the iterative execution is used minimum resource, maximum time. Complete unrolling take minimum time, maximum resource right. It is a complete end. So, can I make a tradeoff between this? So, what is the trade off? It is the partial unrolling. So, instead of complete unrolling, you do the partial unroll so by a factor right.

So, you mention that if my loop is a running for 64 iterations, I make it factor 8. So, now my loop will iterate for 8 iterations and consecutive 8 iterations operation will be executing 1 iteration, ok. That is the basic idea. So, it is decided by the unroll factor. So, you can give your unroll factor based on your requirement ok.

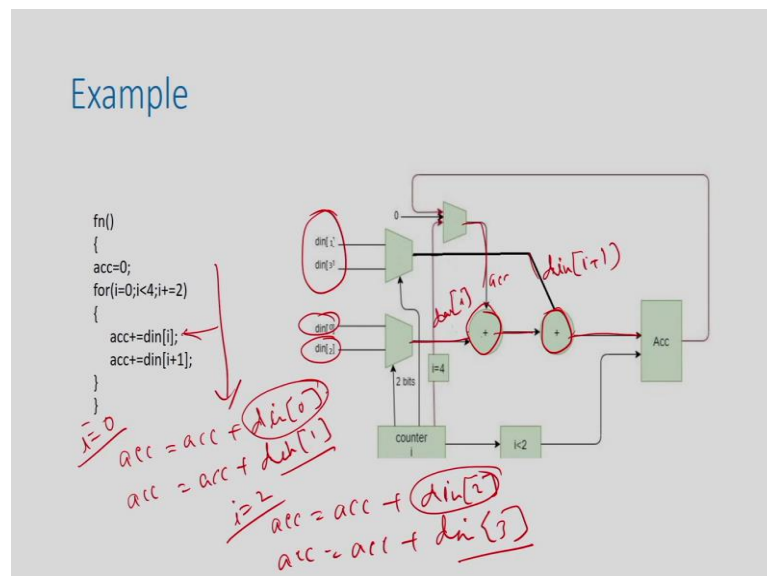
(Refer Slide Time: 29:03)



So, let me take the same example the accumulator example. So, I have 4 iterations I just say [FL] unroll factor is 2, right. So, what I am doing is basically I want to do 2 iterations into 1 iteration, right. So, what I can do is basically I am doing this.

So, accumulator is equal to d in accumulator d in i plus 1 and I am incrementing i equal to i plus 1. Here it is i plus plus right. So, now this loop will iterate 2 times because I reduced the number of iteration of the loop by a factor of 2. As a result, I unroll the loop by the unroll factor, ok.

(Refer Slide Time: 29:38)



So, this is how unrolling works and if you can understand that this is basically middle path and based on your resource available, you decide the unroll factor. If you have lot of resource available, you give a better the higher factor, then the number of iteration that will be running in parallel will be more, but if you have less resource, then you give the unroll factor less, ok. So, that the number of operations to be running in parallel will be less and you can utilize your resource right.

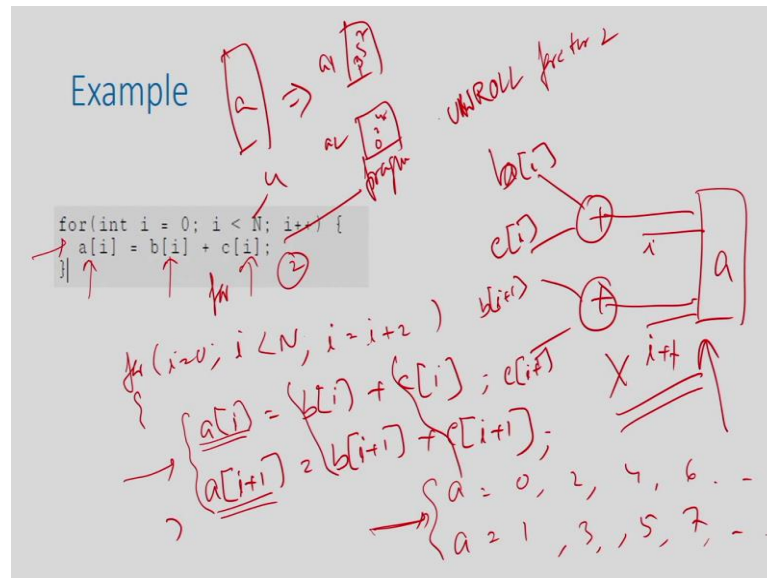
So, if you just take the same example that accumulator, now I have two operations. So, this is the first operation where I am doing this $acc + d$ in and this is that output and this is my $d_{in} + 1$ and this is d_{in} . So, then the result is going there, right. So, I am now selecting you can understand here that here in these operations acc equal to $acc + d_{in} 0$, right and here then I am doing acc equal to $acc + d_{in} 1$, this is my i equal to 0 and i equal to 0, then my i will be 2, right.

So, when i equal to 2 what I am doing acc equal to $acc + d_{in} 2$ and then acc equal to $acc + d_{in} 3$, right and I am done. So, for this input what I am going to select? I am going to select between this and this because these are the things is going to added here right. So, this is i that is why I am selecting $d_{in} 0$ and $d_{in} 2$, right and here for this second output this $i + 1$, $i + 1$ is 1 and 3, right.

So, this is my 0. So, $i + d_{in}$. So, $i + 1$ is basically $d_{in} 1$ and $d_{in} 3$. So, this is why this mux is selecting between 1 and 3 and this is my $i + 1$, ok. So, this is what try to. So, how you should multiplex the this arrays and I am assuming this array is basically mapped to registers, ok. They are all available in parallel ok.

You can understand this partial unroll; it depends on your resource availability. You should decide your unroll factor and it will improve the performance, ok without much increment of the resource requirement ok.

(Refer Slide Time: 31:51)



So, if you take the same example of this and if you just make a factor equal to 2, you can understand that and then the rewritten code will be like this $i = 0$, $i < N$, then $i = i + 2$, right and then here I am going to do $a[i] = b[i] + c[i]$ and then, $a[i+1] = b[i+1] + c[i+1]$, right. So, this is what the unroll factor equal to 2, right

So, if you just write here pragma in this code if you just write pragma unroll 2 factor, 2 sorry this factor equal to 2 unroll factor 2, so it will automatically be converted internally like this or you can write yourself, ok. So, both it is fine and I am not drawing the diagram, but you can understand that you can use two operations to do these two things, right.

So, this is $a[i]$, sorry this is $b[i]$, this is $c[i]$, this is $b[i+1]$, this is $c[i+1]$ and this result is actually going to the a array. So, there are two data basically. So, this you write into i , this you write into $i+1$. So, this is how I should execute this and you can understand that my rolled implementation will take one complete unroll will take say if $n = 4$, it is 4 adder and my partial unroll is taking 2. It is basically completely the middle one, ok.

So, that is what. So, one important point to be noted here is that with this unrolling, the access of array it is getting increased. So, right so you can see here that earlier in this here every clock I am going to access one time a 1-time b 1-time c , right. So, now if this

a b and c are mapped to say blocked ram and then if I assume that this if I just doing the factor by 2, I can see now actually in every iteration I am accessing two value of a 2 value of b 2 value of c right.

So, if that particular block ram is single port, then this will create a bottleneck because you cannot access two. So, this benefit you will not get, right. So, you have to still do it in a sequence. So, if that particular block ram is dual port, it is not a problem because you can actually access two data right, but if the unroll factor is 8 10 or so on right in that case your dual port or single port does not help.

So, how I can improve that? How can I make sure that even if I partially unroll and I am accessing more than two data of a particular array I can actually achieve my required performance by doing the partitioning of the array ok? So, I will take a concrete example and I will show you how to do that, ok.

So, even if you take this example. So, since suppose there is a single port array ok, there are two accesses and you can as see that what is that data is you are getting accessed here a 0 2 4 6 and so on, right and here I am going to access 1 3 5 7 and so on.

So, what I can do? I can actually split the array a into two arrays, right array a to a 1 and a 2. Here I am going to put the data of 0 2 4 6 and those things and here 1 3 5 7 and those index data. So, now there are two arrays. These two arrays can be have to two different block ram and I can actually access those data because these are two different array.

So, based on the access pattern I have to decide how to partition the array into multiple arrays. So, the key takeaway or the key summary that I want to make from this slide is that just doing blindly unrolling, partial unrolling may not help because your array increment because loops are always come with arrays and since the array access is increasing. So, if you assume that I am just doing unrolling, I will give get the desired performance improvement. It will be bottlenecked by the array accesses.

So, always once we apply this unroll, partial unroll you have to make sure that you partition the array based on their access pattern, right. So, the example that I tried to show here and if you just partition the array into that way, then only your improvement required performance benefit would be obtained otherwise you won't get that, ok.

(Refer Slide Time: 36:49)

Unroll factor

- Since unroll increase array access, the array also needs to be partitioned appropriately to achieve the desired performance

```
#define CHANNELS 8
#define SAMPLES 400
#define N CHANNELS * SAMPLES

void foo (dout_t d.out[N], din_t d.in[N]) {
    int i, rem;

    // Store accumulated data
    static dacc_t acc[CHANNELS];

    // Accumulate each channel
    For_Loop: for (i=0; i<N; i++) {
        rem=i%CHANNELS;
        acc[rem] = acc[rem] + d.in[i];
        d.out[i] = acc[rem];
    }
}
```

So, I take a another good example say suppose I have this code, right. So, this code you can see here, there are two-factor what is channel and one is samples, right and the total number of iteration is basically channel into samples and what is happening here you see carefully.

So, basically what I am doing here for every i i just get the remainder value which is mods channel, right. So, the remainder value will be 0 1 to 7 one of the values right and then, what I am doing corresponding that accumulator I am updating the adding the value din, ok.

So, what you going to happen here so effectively and then that value I am going to return out to v out. So, basically there are 8 channels, right. So, 8 channels and the corresponding data. So, whenever some i comes and I am going to put that 0 8 16 24, these data's their corresponding d in value, I am going to put into that acc is 0. In the channel 0, this is my channel 0 right.

Similarly, 1 9 17 25, this index values I am going to add, I am going to put into channel 1, right. This is my acc 1. You can understand this way I have the 8 channel which is acc 7 which will store 7, then 15 23 31 and so on those corresponding inputs, ok. So, this is how this code is working ok. So, you can understand here that there is a good chance because this acc there are 8 channel, right.

So, there is a good chance that I can actually do a unroll of this loop by a unroll factor of 8 because these channels, so that I can actually parallelize this operation. This is my 1 iterations of 1 channel. So, basically 1 channel content right. So, then this set of data will go into the channel, right.

So, the same channel. So, what I can do, I can parallelize this. So, I can actually put a unroll factor 8, so that the iteration 1 to 7 will execute in one time in 1 iteration. 8 to 15 will be executed in the next iteration, 16 to 23 will be executed in next iteration and so on, ok. So, this is what I can do. So, I am going to do that.

(Refer Slide Time: 39:12)

Partial Unrolling

- Partially unrolling the loop by a factor of 8 will allow each of the channels (every 8th sample) to be processed in parallel if the input and output arrays are also partitioned in a cyclic manner to allow multiple accesses per clock cycle

```
void foo (dout_t d_out[N], din_t d_in[N]) {
#pragma HLS ARRAY_PARTITION variable=d_in cyclic factor=8 dim=1 partition
#pragma HLS ARRAY_PARTITION variable=d_out cyclic factor=8 dim=1 partition
    int i, rem;
    // Store accumulated data
    static dacc_t acc[CHANNELS];
    // Accumulate each channel
    For_Loop: for (i=0; i<N; i++) {
#pragma HLS UNROLL factor=8
        rem=i%CHANNELS;
        acc[rem] = acc[rem] + d_in[i];
        d_out[i] = acc[rem];
    }
}
```

So, I am going to do that that I am going to put a unroll factor equal to 8 into this code, right. So, if you just do that, the tool automatically will rewrite this code into that way, right. So, basically you can understand that there will be 8 copy of this code and this i i plus 1, i plus 2, i plus 3, i plus 4 to i plus 7 will be used here right.

So, I am not writing that, but I hope you understand the corresponding loop body, ok but with this what is the problem what the bottleneck I am getting here, you can see here that now since I am going to execute this 8 copy of the body in same, so that d will be accessed d in will be accessed 8 times now and d out is also going to be used 8 times now. Earlier it was only one. So, now whatever the way you map into d in into 1 block ram, you cannot access all 8 data right.

So, you although we try to execute this one, one complete channel of execution in 1 iteration, you cannot achieve it because of this array access problem, right. So, now I have to split the array and you can understand how you should split into the array. I can split the array into 8 copies right and such that this data I am going to put of d in into array one this data. That means, index 0 8 16 24 and so on 32 and so on.

This I am going to put in the array 1, din 1 din 9 din 17 din 15 din 33. This I am going to put say array 2 or d in 2 and so on. So, I am going to split this array or partition, the array into 8 arrays and partitioning is also very important. It is a cyclic manner right. So, I am going to partition the array such that 0 goes to partition 1, 1 goes to partition 2, 2 goes to partition 3 and so on right.

So, this is how I am going to cyclic way I am going to partition. You cannot just partition blindly, then also it would not work. You have to be see the access pattern and you have to make sure that the iteration that are now I am going to running in parallel. The corresponding data must be in the different different arrays. They cannot be in the same array, then their performance improvement won't be achieved ok.

So, if I just do that, so that is what can be done by this pragma that you partition the array d in by a factor 8 and you partition the dimension one, right. So, dimension one means it is only one dimension. So, you can actually do that. So, basically this way it will actually partition this way and you can this is actually important is the cyclic. If you just write the cyclic, it will actually do that that. 0 will be proved in partition 1, 1 to partition 2 and so on right.

So, this is what I try to emphasize that your unroll factor allow each channel to process in parallel that 8 channel every 8 sample if only if the input and output array are also partitioned into cyclic manner to allow the multiple access per clock cycle. So, whatever I just explained, the same thing has to be done for d out as well because d out is also going to access 8 times in 1 iteration.

So, if you just do both unroll and partition cyclic partition, then only you can actually run the 8 iterations of this loop in 1 cycle right and you can actually achieve the desired performance benefit, ok.

(Refer Slide Time: 42:40)

Unroll Factor

- Partial loop unrolling does not require the unroll factor to be an integer multiple of the maximum iteration count

```
for(int i = 0; i < N; i++) {
    a[i] = b[i] + c[i];
}
```

→

```
for(int i = 0; i < N; i += 2) {
    a[i] = b[i] + c[i];
    if(i+1 < N) a[i+1] = b[i+1] + c[i+1];
}
```

Handwritten notes: $a[i+2] = b[i+2] + c[i+2]$, $a[i+1] = b[i+1] + c[i+1]$, $i(i+1) \leq N$ break, $a[i+2] = b[i+2] + c[i+2]$, $a[i+1] = b[i+1] + c[i+1]$.

So, one more important small important factor that sometime suppose I want to put a unroll factor which is not exactly divisible of this value. So, suppose this is my say 15 and I just put a unroll factor equal to 2. So, that means it is not that I am able to partition these particular iterations into evenly, right.

So, because 15 is not divisible by 2 so, but that is actually not a problem because you can actually write this way that you because I want to put a unroll factor two. So, I just put i equal to b plus c and i . If I just put a line if i plus is greater than you just break here, otherwise you do this and it is i plus 2, right.

So, what is happening is when i equal to 0 a 0 equal to b 0 plus c 0 going to happen a 1 plus b 1 plus c 1 going to happen. When is i equal to 2 a 2 plus b 2 plus c 2 is going to happen and this a 3 plus b 3 plus c 3 is going to happen, but when i equal to is 15, so the only a 15 plus b 15 plus c 15 is going to happen, but this line won't be executed because I am going to break it out from there, ok.

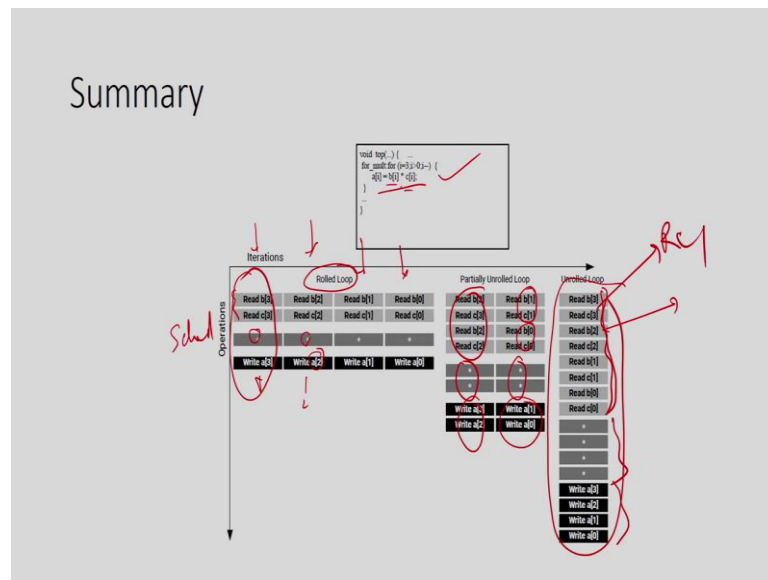
So, basically the takeaway from this particular slide is so if you just put a unroll factor, you should not bother about whether it is a exact divisible or not because even if it is not, tool will make sure that it would not execute unnecessary things, ok. It will stop unnecessary thing but it is a additional check.

So, you try to factor it such a way that you should not have this kind of code because whenever you add this code, it will always check. You should have a conditional checking right which will take some resource and time, but it does not stop you from factoring the loop, but if you try to do it in manually, this whole chain you have to physically add this kind of line right.

So, similarly if you just try to do by factor 3 here, so you have to add a line like if $i + 2$ is greater than equal to N , then you break right. Otherwise, you do $a[i + 2] = b[i + 2] + c[i + 2]$, right. So, basically after every operation you have to check whether the $i + 1$ or $i + 2$ or $i + 3$ is greater than equal to N , then you should break otherwise you do the next operation, ok.

So, this is how you can actually manually do that or if you do not do that, if you put unroll factor which is not perfectly multiple of this for the actual number of loop iteration, it will automatically do this in general but it is not advisable to do that because it actually take extra condition check in the hardware, ok.

(Refer Slide Time: 45:14)



So, with this I am going to summarize this class. So, I have seen that in this class that that loop the 3 way of executing the loop. One is the complete unrolling, partially unrolling or complete rolled implementation right. So, if you take this loop, this diagram summarize that. So, if you have a complete rolled implementation. So, you in iteration 1,

you will read $a[3]$ $c[3]$ and then, do the addition operation and then you write $a[3]$. So, this is your clock 1.

In clock 2, you do read $b[2]$ $c[2]$, do the error multiplication operation and write $a[2]$. So, this is your clock 2 and then clock 3 and clock 4, right. So, you need 4 clocks and I am assuming this. So, basically this is the iterations and you might need multiple clock cycle to do that, right. So, because your read might take some cycle, I am just saying that this is the operations and you have to schedule them also, right

But this is how the iterations works right, but if you partially unroll say by 2. So, read of $b[3]$ and $c[2]$ is going to happen 2 multiplication is going to happen and 2 write to $a[3]$ and $a[2]$ is going to happen in the next iteration reading of $b[1]$ $c[1]$ $b[2]$ $c[0]$ and this two-addition multiplication corresponding multiplication is going to happen and you are going to write $a[1]$ and $a[0]$ and you need basically 2 iterations and if you completely unroll, everything is single cycle right.

So, all 4 reads from b and c , all 4 multiplications and all 4 write to a is going to happen in single iteration and how many clock cycle is needed for these operations, it depends on your resource availability right. So, it might 1 cycle or k cycle based on your resource availability and also, how these arrays get mapped that will. So, right so how this arrays goes map to block ram, also it also is a deciding factor because based on this access pattern you have to see right.

For example, here even if you do this and if you map this block ram into this b and c into ram, then you cannot access 4, right. You have to wait for 4 cycles which a single port. Even if you put into register, then you can access all them in parallel right. So, this is what two factor depends and based on that your schedule time will be determined, ok. So, with this I conclude this discussion.

In the next class, we are going to see how the loop will be pipelined in the hardware, ok.

Thank you.