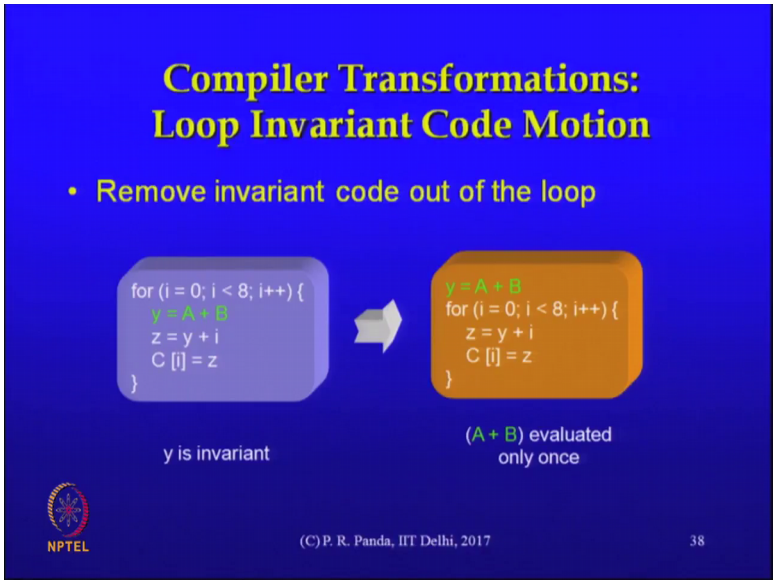
Synthesis of Digital Systems
Dr. Preeti Ranjan Panda
Department of Computer Science & Engineering
Indian Institute of Technology, Delhi

Lecture – 11
Compiler Transformations in High Level Synthesis: Loop Unrolling and Function Inlining

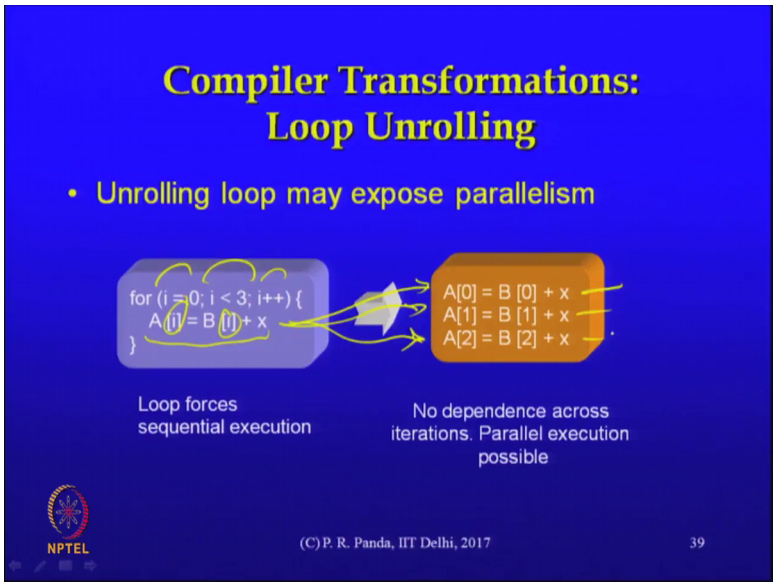
Let us continue, we were on this topic of high level transformations before actually running the synthesis algorithms.

(Refer Slide Time: 00:25)



And while we studied the loop invariant code motion this is a simple optimization, but in the context we also defined quite a few things, we discussed the memory mapping strategies, which are all very interesting because there are many different choices and appropriate choice has to be made depending on the resources the architectural flexibility that we have and any other constraints and whatever is the optimization criteria.

(Refer Slide Time: 01:10)



Continuing on this topic, let us look at one more very interesting optimization called loop unrolling. This too comes from the compiler domain, it is a high level optimization. The objective of such an optimization is that it exposes some opportunities, for later optimization the example is this. We have a loop i going from 0 to 3 and there is a loop body in which there is some computation there are some memory accesses and so on.

The observation of course, is that written in this way and implemented in a straightforward fashion the loop forces a sequential execution, because the semantics are sequential of the for loop, what optimization opportunities could be there in the loop. We have 4 iterations of the loop i going from where 3 iterations, i going from 0 to 2 and in each iteration, there is a loop body that is being executed. So, where is the optimization opportunity?

Student: (Refer Time: 02:21)

Observation is that the different loop iterations are all independent of each other that can be exploited in many different ways, independent me first of all means that you do not have to respect the order in which the user has written the code as long as the iterations are independent. We could reverse the order of the iterations and get the same result, but more important is that we could execute all of them in parallel and therefore, save time if we had the resources. So, that is the loop unrolling optimization conceptually it is actually very simple you just blow away the enclosing condition, the loop and the

condition and the iteration, and directly copy that loop body over into what we have on the right.

The unrolling can be done as long as you are able to resolve these conditions properly the initial condition, the final condition, and the updating of i as long as you could do that statically that is you are able to see how many iterations you are going through and the value of i in each iteration, all that is needed is you propagate the corresponding value of i into the loop body, wherever i occurs you would have to make an appropriate substitution.

So, that is what it leads to the multiple copies of the loop body and with the right values of i being filled in that is what we have that ok. Question is now how does it help this is just a high level transformation as of now we have not optimized anything, we have only blasted away the loop into multiple copies of all its iterations, now what happens where is the opportunity for improvement?

Student: (Refer Time: 04:24) parallel now.

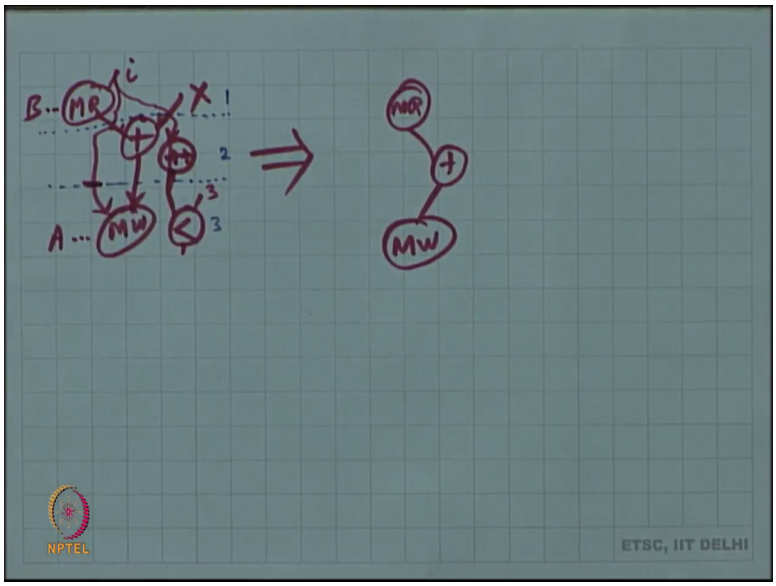
There is the possibility that we can execute this in parallel, where do you see that when you construct the dfg for this entire basic block now consisting of the 3 bodies there is no conditional among them.

So, you have the whole thing being represented as a single dfg there you will find that these are actually different data elements and there is no obvious link between the 3 iterations. So, as long as you have the right resources you could actually schedule all of them in parallel, what resources are needed to implement this.

Student: (Refer Time: 05:10) the volume of the (Refer Time: 05:13) are required (Refer Time: 5:15) the signals are simply biased.

That if we actually try to do all of these things in parallel then you need multiple copies of the resources. So, first of all you need multiple adders, but what is the dfg corresponding to first the original loop I had.

(Refer Slide Time: 05:34)



A class this is x and what about the Ai and Bi.

Student: (Refer Time: 05:40)

Right, I have Bi translates to in general a memory read operation and we saw what that might look like that is memory read, the output of that goes to the adder; i is an input and what is the other input is there any other input.

Student: So, there should be an address (Refer Time: 06:11)

So, there is an address computation, which has to occur before the memory read operation and that address actually goes as input to the memory address. So, we did study the different possible memory architectures it could be that A and B are actually in different memory modules if that is. So, then i is enough as the address and there is an a separate decision that this memory read corresponds to the array B, the memory module that contains B then you do not need anything else.

On the other hand if A and B were both to map to the same memory, then the address is not merely i there is a more complex address, right and that is generated that is what I have on the left, but the loop unrolling well there is a write also which is a memory write this is the data, memory write has another input which is the address and that address comes.

Student: (Refer Time: 07:22)

From you know whatever that i could come there as the address, but this is actually a different memory. And somehow that has to be separately annotated as a dfg node it is fine the 2 actually correspond to different memory models fine that is all I have in that original loop iteration. So, 1 iteration when it is scheduled could result in perhaps 3 different cycles, assuming that the memory accesses and addition all of them take 1 cycle each then that is the first cycle this is the second cycle and this is the third cycle. What does the unrolled loop look like, where does the dfg of the unroll loop look like.

Student: (Refer Time: 08:26)

So, you actually have the same structure, but it is just lubricated this is a read and then there is a addition is still there, but to the memory read operation you still have to provide an address it is just that you know the addresses, since you have expanded i out in the original loop there are a few other things actually there are they we didn't show it because they are not in the critical path, but you are comparing i with 3 in every iteration you are also doing i plus, plus in every iteration. Where does that show up in the dfg that is still computation that needs to be represented and scheduled?

Student: (Refer Time: 09:15)

The general while loop we looked at how to handle write, what you are scheduling is actually just the loop body, but result from that schedule that comparison is also an operation that is scheduled in the loop and the result of that comparison is used to decide whether to get out of the loop or not. So, you still have to schedule it which means that there is that operation that needs to be performed right. And less than operation means that i is 1 of the inputs 3 is the other input and that result has to be there at least by the end of the last cycle i should have that result and there is that i plus, plus. When do i do that?

Student: after this.

That after this with.

Student: (Refer Time: 10:15) there should be above all the process it should not be in the last cycle because in that (Refer Time: 10:18)

There are some things like I said when we talked about the while loop earlier this comparison was actually repeated before you entered the loop for the first time. If you take care of that then it is to do it once at the end, but there is one more thing we need to do which is the i plus, plus i needs to be incremented as part of the loop body after that the i less than 3 is actually compared. So, how do i fix this dfg to include the i plus, plus where should i put the i plus.

Student: before the comparison.

Before the comparison here is that ok.

Student: yes.

I do this and so.

Student: (Refer Time: 10:59)

This is plus or i equals i plus 1 and that result.

Student: (Refer Time: 11:09)

Is what I would use is that fine.

Student: sir we should reverse it.

Reverse the comparison and the increment with what is the semantics of the for loop do you first increment and then compare or you first compare and then increment.

Student: (Refer Time: 11:29)

Really.

Student: no that we will never.

You do compare before you get into the loop.

Student: (Refer Time: 11:35)

Yes, but first we are saying well do separately you could do this in different ways, but here we are saying that the comparison happens at the end at least that is 1 valid way of

doing it. So, that is what it would look like there is a dependency both. So, i is an input to both the increment and to this which means. What does that imply as far as the ordering of these is concerned.

Student: (Refer Time: 12:05)

Yes. So, actually the problem is that if you increment i here, then that is the value that becomes the address and that is not intended you do want the increment to be at least logically split from everything else that is happening or wherever the users are there of i in the loop body so.

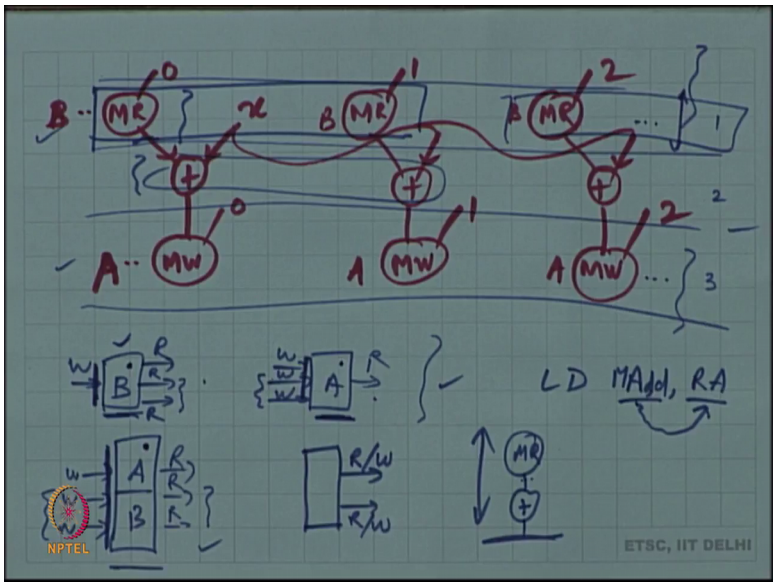
Student: (Refer Time: 12:36)

Right., so if this increment is actually changing the value of i right there, then there is actually a dependence that is created and you cannot have it later, but you could actually do it in a different way where you do the increment here you, but the result is stored in a temporary variable you do need to update i at the end. Maybe that update could be happening in the last cycle, but the comparison there is enough information for you to do the comparison.

Student: second one

You this could act for example, it could be that you just determine i plus 1 the value of i plus 1 and that value is used for the comparison, which is fine and the actual update of the i happens perhaps at the by the end of this because you could resist this is registered right and therefore, simultaneously with the memory right you could have the updating of i, but for the unroll loop what would the dfg look like I still have my memory reads.

(Refer Slide Time: 13:50)



I have the addition, I have the x and I have the memory right. This is again for A this is for B and address is 0, here address is 0 both unknown what else do i have this is the iteration for i equals 0.

Student: index would be 0, but address would be what we (Refer Time: 14:26)

Here we are assuming that the memory module for storing A is different from the memory module that is stored in B that memory has only a nothing else. So, the index is same as the address if it was anything more complex than you would have to do an address calculation there what else is there I just duplicate this.

Student: (Refer Time: 14:57)

So, I have 3 copies, these are changing the addresses are changing, but they are changing in a way that they are statically known what else should I do to this data flow graph this x is actually common to all the three. So, I can actually move it here like that.

Student: (Refer Time: 15:37)

For now I just label them, this is just the dfg dfg is still correct irrespective of how you are going to synthesize it. So, what optimization opportunities are there let us take a look, but here is a memory right into A, memory right into A, memory right into A, addresses are different, but that is my dfg, but that is not the end of it as you are

identified. Now if I schedule this how many cycles will this take, one useful thing that we have done is in the process of the unrolling, we have thrown out some of the operations, we have thrown out every iteration there was the increment operation that is thrown out because everything is known statically right.

We have annotated the 0 1 and 2 directly, then the comparison is also removed from here that is also redundant there is no need to do a comparison all the information was statically available. So, that simplifies in some way, in fact, there are some bottlenecks that are implied by the increment operation, those we have actually simplified and this is actually more amenable to parallelism to the extent that you are providing the parallelism in the resources, one kind of parallelism we could provide is that we can have actually multiple adders and that would allow us to execute multiple addition simultaneously.

But what other possibilities are there for parallelism fundamentally can we do better than 3 cycles per iteration, which is a total of 9 cycles. If we gave more resources first question, second question is what resources then are needed for us? If we had enough resources we could do this kind of a schedule, certainly as far as the execution is concerned if you gave me three adders I could schedule all the three additions in parallel, could I also schedule the memory reads in parallel.

Student: sir if i can have something like of (Refer Time: 17:45)

What resource from the memory is needed for me to be able to schedule all the three memory reads into the same cycle.

Student: (Refer Time: 17:55)

Multiple read ports I need to have a 3 read port memory, if I had that remember all these correspond to be these are different locations of that memory that are being accessed. If it was actually a 3 port memory then we could actually schedule it in this way, same with rights if I had at least 3 read ports for the memory that stores B 3 read ports, from which I could read data out simultaneously.

Then I could schedule all of these into the first cycle, similarly if I had 3 write ports into the memory that stored A. Then to I would be able to realize all those memory rights in the same cycle, if I had combined A and B into just 1 memory module, what is the

implied property of the resource. If both of them were there in the same A was there and B was also there, then I would need a memory with 3 write ports to realize all of these in the same cycle, and also 3 read ports to realize all of these in the same cycle.

Any questions about these are the extremes and you would need all these kinds of resources, these possible it is just that you have to have the appropriate resources, which leads to a natural trade of what kind what metric versus what metric are we trading of.

Student: sir power.

First of all area versus delay this is a delay optimization performance optimization; obviously, were trying to do things in parallel means we are trying to optimize performance. Asking for a memory with 3 read ports as opposed to the standard 1 that has just 1 read port, and 1 write put means that you are asking for a larger memory you are still assuming that the access time of the 3 port memory is still within the clock cycle that is available, right otherwise it will not fit into 1 clock cycle.

You might still get a performance advantage, but this stage might actually split into multiple clock cycles perhaps same might happen here. So, that is an area versus delayed trade of what about here well actually, here I could have done with a memory that had just 1 write port, but multiple read ports at least for this part of the code. Similarly here I could have done with a memory that had a single read port, but if A and B were clubbed together into the same memory module, then I would need it to have 3 read ports and 3 write ports which is even more expensive in terms area, right there was a comment about power. What is the analysis with respect to power?

Student: (Refer Time: 21:04) 6 ports, so 3 read and 3 write in that case I had 2 instances of memory.

Right.

Student: and I had 4 ports each. So, the ports were 8.

Student: So, area would.

I had 2 memories, here I had 1 read port and 1 write port remember 1, right port is minimum you can’t have a memory with just read ports and no write port, because then

you can't store data in the memory, same thing with a read port there is no point writing into a memory if you can't read out of it and therefore, 1 read and 1 write is of course, expected.

So, it is a comparison between 2 additional read ports here, 2 additional write ports here versus 2 additional read, and 2 additional write ports which is greater area from here it is not obvious, but we did talk a little bit about it in the previous class what are involved here each cell of the memory has to change. Now, in fact each of the cells here now has 4 ports, here 2 every cell has 4 ports, the total size is same in terms of number of cells, but the complexity of each cell is changing in what ways. So, here in the previous solution in this solution every cell has 4 ports here every cell has 6 ports.

So, from that point of view just the cell area point of view this 1 is a more expensive, but this is not all like we had said there are other components that we also have to argue in terms of there is this sense amplifier here, remember some of this doesn't apply if these are very small memories, but we are just arguing in terms of the conceptual problems that show up when you have larger memory.

Student: again picture i also needs to decide for example, if I am doing a read on 0 so.

Right.

Student: this 0 belongs to A or this belongs to B assuming A and B are (Refer Time: 23:80)

There is an address computation fortunately in this unrolled example all the addresses are statically computed. So, right whatever the address is whether it is 0 or 3 or 1 or 4 and so on. We are able to calculate there is no need to schedule a computation for that address, in this particular case because all of them are known, they might show up as address calculations just like in the other case we had the i plus, plus and so on, but this is a situation in which the address is known, it may not be as simple as 0 and 1, but it is known also remember what we talked about the decoder here, there are some arguments with respect to the larger decoder is the some of the smaller decoders plus a little decoder.

If you know how a decoder is designed, in terms of pure logic, but that is 1 argument the, but the other is also that when it is split like this, the storage is split like this there is routing related overhead, that is usually big when it is a compact decoder like this the area overhead could be less. In general when you split a larger memory into multiple smaller memories you expect an area overhead, for this case where you have smaller memories because of the decoder reason and because of the sense amplifiers getting duplicated; however, what counter balances this is that we have a 6 port cell, every cell is 6 ported whereas, in this you have every cell being 4 ports.

So, by itself it is not obvious, but these are the parameters that affect the area related, argument there is also a delay related argument the access time for the smaller memories is smaller than the access time for the larger memories that is hard to argue about without looking at the context.

Because our context here is the clock period, whether it makes a difference to the scheduled or not depends on whether the access time of for example, the larger memory and how it compares with the clock period. So, all of these are the factors that 1 would have to consider, when arguing about the performance advantages of loop unrolling, but it is a powerful optimization particularly in hardware.

Because you have the ability to unroll the loop to large very large extents, and possibly get significant amounts of speed ups performance benefits. If you had the resources to spare, but what if I had constrained you in terms of the memory inputs, I say I do not give you a 3 port or a 6 port or a 12 port memory instead of the loop iterating from 0 to 3.

What if it was i trading from 0 to 100, I could still do the unrolling of the dfg in a way that is correct, but what is then what after that what is the synthesis related argument. If you want a similar performance, then you need a 100 port memory not just that, but the access time should also fit into the same clock cycle which might not happen, but the way to formulate the problem then could be that you are provided memory resources.

Let us say these are dual port memories, or 4 port memories or something it is fixed memory properties are fixed, then how would you argue essentially all of these 100 operations do not proceed in parallel. if I give you only a 2 port memory, then these 2 out of the 100 independent read operations that are there you can schedule in parallel in the

next clock cycle while you are doing this addition perhaps you can schedule the other 2 and so on. This is a scalable design methodology it is just that resources are constrained and therefore, in every clock cycle you are able to pick up only those of the independent operations, which your resources permit it is still an advantage from a performance point of view it is still an advantage.

Student: sir memory has to allowed 3 (Refer Time: 27:34) if memory is same. So, if here the second option a B common.

Right.

Student: memory has to allow both read and write in same cycle the way we.

When we say that there are so many read ports and so many write ports, what we are saying is that all of these operations can happen in parallel otherwise you call. It what is a read writes port, so maybe there are 2 ports each of them is read write means that you can do either read or write on to it, but not both.

Student: So, even the buses could be different.

Right.

Student: but still memory only allows either read or write in a cycle that is a very common memory.

Yes, but that is not the only way we didn't specify the technology here, register files are a class of memories wherein every clock cycle you might actually do both reading and writing perhaps splitting it internally into different phases of the clock cycle, but as far as the synthesis sees that is also a kind of memory into which you are doing both reading and writing, conceptually you are doing both reading and writing irrespective of how its internally implemented.

All of these designs are likely to complain, if you try to read and write into the same location of the memory, then it is not clear what you yourself are doing this is the case in which these are different locations, but we are reading and writing into it.

Student: case first which we are different A and B. So, we are reading from 1 memory and we are writing at different.

Right.

Student: in 1 memory.

Right.

Student: So, they can go in parallel they are two different.

They can certainly go in parallel, but the question was about the other 1 in which we are saying there are read ports and there are write ports.

Student: sir 1 more doubt.

Student: in typical microprocessor (Refer Time: 29:24) and then there is a memory to fetch the instruction and it contain its (Refer Time: 29:26)

Right.

Student: now here the m r (Refer Time: 29:29) register file or they or these are (Refer Time: 29:30)

In a processor there is a fixed protocol with respect to the relationship between memories and registers, when you say a load instruction of the CPU, what are the operands to the load instruction.

The load instruction has 2 operands 1 is an address and memory address other is.

Student: register.

The register address register file address, implying that you read from the memory and store the data in the register, where do you store it in the register file you have to specify it. Here we do not say where it goes into, but what is the implication when I do the schedule, where does that data go into.

Student: goes into some register.

They go into a register, which is implied from the fact that this operation takes place in 1 clock cycle and the result is used in a different clock cycle, a separate register is inferred, but that inference is just like any other computation, when you have an edge that is

crossing a clock cycle boundary a register is inferred. You do not say anything if for example, you had enough time to do a memory read.

And an addition in the same clock cycle, then there is no need for a register you could argue if the memory is designed in a way that the operand is provided, then there is perhaps no need for it you store it here at the end of the clock cycle if you need it for the future right. So, this is treated in a way that is identical to the other operations and there is no implicit requirement that if you read from the memory it must go into a register file we haven't even defined any register file here.

Student: but at least in this case it was required, because it is in second cycle that it is going, so register has to.

Yes that register was required here, but only because of the normal synthesis rules. So, memory is a little more complex operator and it has many different parameters, one of the parameters could be that the output is registered and therefore, maybe there is a clock to the design and so on. So, several other parameters size of the memory, how many ports are there of the memory and remember the delay is a function of the size it is a function of the port and so on.

So, the number of properties that you would need as part of the memory library element is a lot more because it is more complex this two is simple, you could have phases in the memory you could have more complex memories like a d ram or something in which there are many different modes of operation. All of those somehow have to be exposed to the synthesis tool, if we want to make automatic decisions about the mapping of the data about scheduling of the operations in the data.

Student: one more question.

Student: typical design all the adders parallel adders would be exposed to all the registers is it true and everything would have to.

Student: come through a mux lets say there was 10 16 ports, 16 registers.

Right.

Student: then all the 16 registers would be mux to every adder then the select lines will make sure that the appropriate register data goes to the appropriate adder.

Right so if you had a number of adders, the muxes are inferred, but when are they inferred not at this stage you would have to do the scheduling the extent of the sharing of the inputs of the adders determines the maxes that are there. So, there is no additional rule that one needs at this particular stage, it is just assumed that the adder is a combinational circuit.

Student: (Refer Time: 33:41) where we are saying hardware is realized for this particular loop.

Right.

Student: but if it is a generic hardware then all must mux.

It depends on what you are calling generic what you are calling specific, but we are saying here this is the more general architecture. In which you are free to not have a connection, not have a mux if it is not needed.

The processor is the other extreme, where you need to provide paths from every part of the register file to every adder perhaps and therefore, you need a particular muxing strategy busing strategy and so on for the data to reach. Here, we are saying that we will make the inferences if it turns out that is 16 to 1 mux is necessary at every adder, because all those paths are there considering the sharing of that particular adder then we will insert those muxes system, if they are not necessary then we will not.

Student: (Refer Time: 34:42) if we have parallel hardware we will have specific registers dedicated for that hardware will not.

Registers we are not dedicating it, unless the library element itself comes with an inbuilt register then you cannot throw it out and therefore, you use it, but otherwise parallel hardware is there.

A necessity for muxing of data, reaching that hardware, would always be there, but it is there to the extent that its needed, but in general this is a fine argument which you can say that I will go to this extreme, where everything is inferred as discrete components,

you could also say that I have something like a platform that I have already defined parts of which may mean maybe I have a register file, I have some definition of a bus, I have some even processor components and so on. I fixed part of it, but the other part that is not fixed is what is getting inferred by the synthesis tool.

Student: (Refer Time: 35:45)

Student: should the behavioral synthesis be aware of implementation limitations, the physical implementation limitations like routing overhead, because if I put too many muxes.

Student: it may not be routable right so.

Right, this is a very interesting and an important question at the at this stage we are making decisions with respect to sharing, say right operations are getting shared onto a particular adder let us say without noting, the fact that.

If the degree of sharing increases then the delay is actually increasing, in the scheduling formalism we are not necessarily looking at we are just looking at it at the library components delay that that delay is not static that delay fundamentally does depend on a number of things, if 16 operands have to find their way to the input of that adder, then you need a 16 to 1 mux at the input who accounts for the delay of that mux similarly, that output of that adder is going to 20 different places.

Then the fan out cause's additional delay, how do we account for that these are very fundamental questions that ought to be answered by the synthesis process, but we will look at ways to handle it later on. Let us go ahead with a reasonably simplified model initially, but we will try to make it more sophisticated understanding that there are such limitations that are there, such assumptions of constant delay have been made, but we may need to revisit that later on .

So, that is the loop unrolling it is a very fine optimization it comes from a particular domain of compilers, there to there are many different limitations. Which make loop unrolling a very interesting optimizations your loops may run from 100 iterations to 1000 of iterations? And 1 extreme is that you completely unroll the loop at 1000 times, there would be some scalability reasons both scalability related weaknesses both in the

compiler context and also in the synthesis context, although those weaknesses are very different in the compiler context.

So, you have instructions being executed on a processor does it help to do unrolling, this is what we started off with and. In fact, this optimization was borrowed from there how does it help to unroll a loop if you have a instructions executed on your compiler, which is this the same optimization that we studied.

Student: (Refer Time: 38:43)

This architecture of the processor is fixed though; assume that it is a processor that is able to execute only one instruction at a time single pipeline. If you had the ability to execute multiple instructions at a time, then since the unrolling is exposing some parallelism there would be some advantage, there too there is a limitation to how much unrolling is good though.

Student: it depends on.

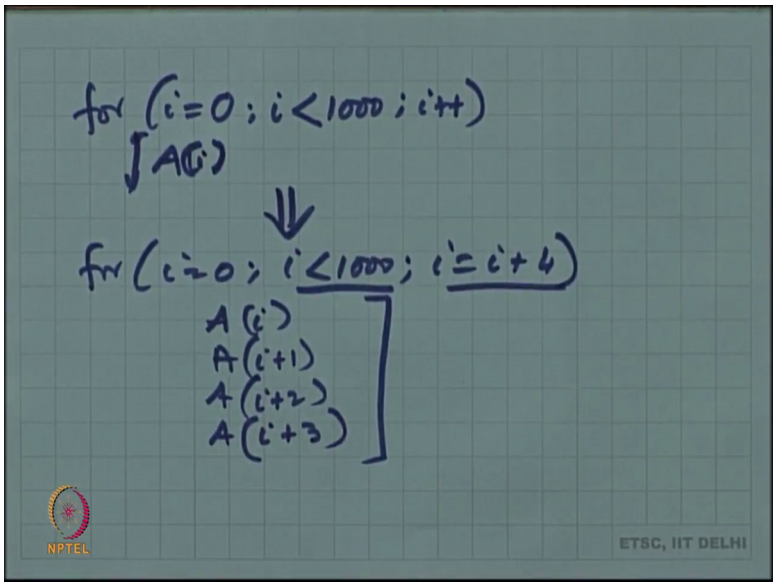
It depends on what is the degree of parallelism that the processor is actually giving you.

Student: (Refer Time: 39:23)

Then whatever the architecture is there would be a limit to the extent of parallelism that it supports, and that does influence the degree of the loop unrolling, just because you have a loop from 0 to 1000, doesn't mean that you must unroll all you must completely unroll the loop. The performance will not scale a 1000 times, right because you do not have a 1000 adders you do not have a 1000 port memory and so on.

It would be a limited extent to which you might do unrolling and that is usually done it is a partial unrolling that you would perform. So, maybe 4 iterations at a time you would unroll the loop by what do you mean by 4 iterations at a time.

(Refer Slide Time: 40:11)



If you have a loop that goes from.0 and that is the loop body. How do i unroll this 4 iterations at a time, it does depend on some of the hardware parameters, but you may say that up to 4 consecutive iterations i can unroll. Beyond that there is not much value because the resources aren't there, and the exploitation of that resource isn't really leading me anywhere.

So, I can say 4 at a time, what does the loop look like unrolling this 4 at a time means what does the resulting loop look like, it is still a loop transformation, I do not unroll it all the way from 0 to 1000, but to 4 at a time. So, I just do a loop transformation.

Student: 6 increment by 3 and right the loop.

I can increment by 4 that is 4 at a time, I could just have this and say i equals i plus 4 and.

Student: (Refer Time: 41:17) i i plus 1 i plus 2.

Right, so if I just say i a i just refers to the basic block that is a function of i. So, I have a i, i could do that there are some things that I have gained by doing this unrolling even if it is limited, remember the I plus, plus. I have got rid of how many increment operations are there one forth the number of increment operations, the comparisons have also come down by one forth. So, there is certainly some benefit even, if you are just counting the number of instructions that are executed there are fewer instructions due to unrolling.

There are other very subtle aspects of unrolling that one needs to be aware of and also because of which you can't afford to just randomly unroll to very large extents, what happens if you take it to the extreme and unroll this a 1000 times, it might seem as though you do not have the resources. So, you are not able to benefit in any way from that extreme unrolling, but could you lose something.

Student: size of A.

The code size increases, because that body is being duplicated a 1000 times, code size increasing has some very subtle effects of what nature, 1 is it the program now occupies more space if you were limited in terms of space then there is an issue.

Student: sir need to fetch more instructions. So, the whole loading of a program by a processor rate initially it has to fetch the instructions now I have more number of instructions.

There is a fundamental effect on memory hierarchy, at the cache is polluted with a large number of copy instruction cache is polluted with a large number of copies of the instructions that are just doing the same thing, but some of the parameters are possibly different.

So, if there is competition for the space of the instruction cache, then this one is wasting, space in the process you might be throwing out some other valuable instructor its possible for example, that the expanded a code does not fit into the cache can certainly happen and therefore, the performance can be very poor, because you start getting cache misses whereas, the original code that was compact here might all have fit into the cache. So, there are some other subtle aspects that need to be taken into consideration when deciding unrolling.

So, that was the context of the compiler, in the synthesis context there are also some very interesting tradeoffs we already saw the resource related argument, but what happens in synthesis, if you expand the loop a 1000 times. There is no cache, there is no instruction could there still be some drawback. Let us assume the context here is synthesis, but I had 1 basic block and this is a new basic block, but expanded a 1000 times, we want to see what is the implication in synthesis?

The resource related arguments we have already made, but let us say we have a resource in which our resources consists of 4 adders and maybe a 4 port memory, or something like that not a 1000 port memory, but 4 port memory 4 port memory, means that maybe you schedule 4 read operations at a time or 4 and 4 write operations at a time and so on. Resources are limited we define it to be whatever they are maybe it is a single read operation and so on, but could there be any negative implication on the synthesis on the quality of the synthesized output.

Student: So, before that there will be an impact on let us say. So, scheduler if I am writing, I am simulating this code, VHDL code then the number of parallel events which now I have to tackle would be more would that be an impact.

It is an impact not on synthesis on simulator it is a different impact; it does impact the compilation time the synthesis time is impacted, because now I have a graph which is 1000 times larger right. So, everything every operation every optimization that I do later on now has to deal with a larger graph than a smaller graph.

So, the synthesis time increases, synthesis time increases means that indirectly the synthesis quality might increase, because might decrease because many of these synthesis runs we might limit the amount of total time that we give to the tool, but could there be other impact on the quality of the synthesis.

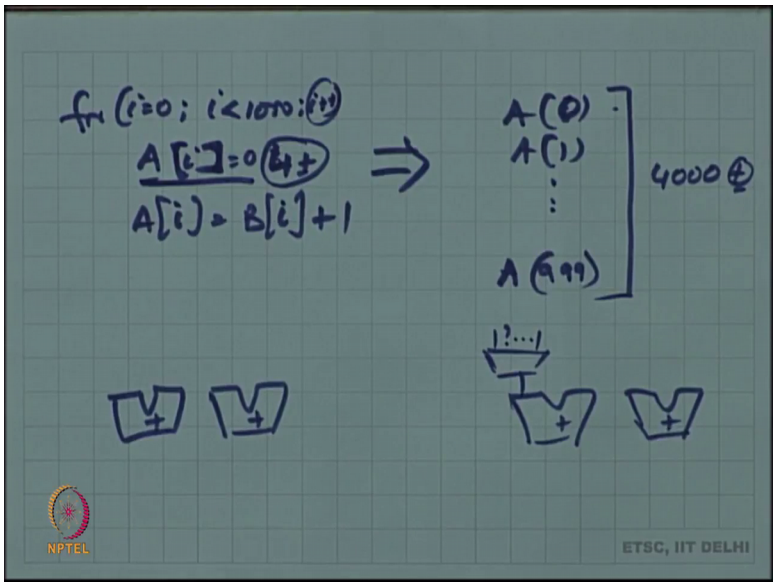
Student: Sir routing overhead like.

Routing overhead why would there be routing overhead.

Student: Because.

Because we said that our resources are the same in both cases right, so I had for.

(Refer Slide Time: 46:40)



I am going up to 1000 and the other was I had no for loop, I just had 0 A 1. So, remember these are the loop bodies right these are the copies of the loop bodies up to A 999 here I had A i.

So, i generate some synthesis result here, I generate a synthesis result in this case the data path part of it could be essentially the same although it is not obvious, but let us say I gave to a adders 2 this scheduler, I also give 2 adders to this scheduler that just that there are maybe there are 3. Let us say there are 4 add operations, in this loop body this code has how many add operations, if you exclude the i plus, plus because that is not there here this expanded loop has how many add operations.

Student: (Refer Time: 47:56)

So, this is 4000 adds, but the resources are the same let us say, it is the resource constraint synthesis where we say you take 2 adders. In both cases I take 2 adders, what else is the other implication, we want to compare ultimately the 2 circuits that are the outputs of the synthesis and possibly make a decision on which is better I said 1000, but you can extend it to large numbers and see if you can predict an impact.

Student: (Refer Time: 48:38)

Multiplexers, if you need a mux here; what is the width of this mux?

Student: 5000.

What determines the width of the mux?

Student: Total number of inputs that can come.

Total number of inputs where would the inputs come from.

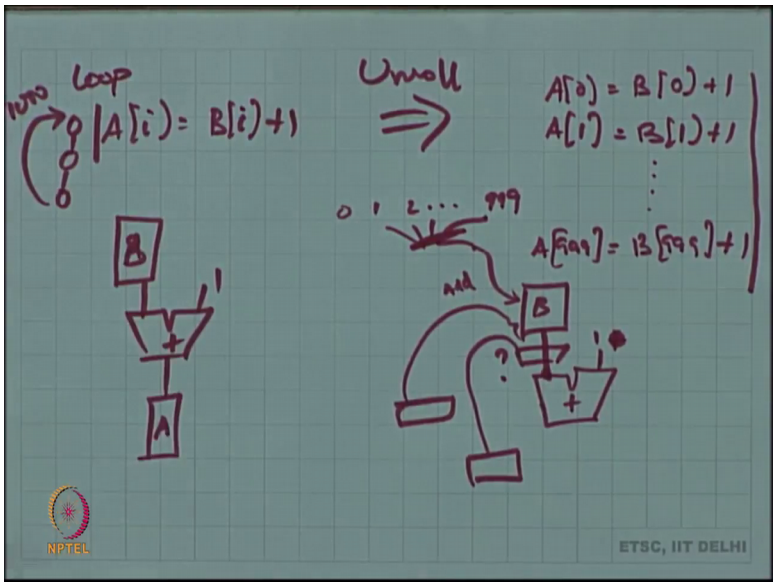
Student: (Refer Time: 49:02) A 999.

Right this is not the complete design I didn't say what is there in A, but let us say there is a memory access, right each iteration you are getting a memory access. And the array is in fact, A let us say what hardware is that. So, this is an A i let me make it, nothing make it like that A i equals 0 is my is actually that wouldn't have an adder, but let us say I have A i equals, some such design right what kind of circuit is that here.

Student: (Refer Time: 49:55)

Right, let us move out here so I have A i plus 1 that is my loop.

(Refer Slide Time: 50:04)



Student: (Refer Time: 50:10) this is a loop where instructions can be went (Refer Time: 50:15)

Let us have an adder, you need only 1 adder for this just the loop body is being represented and scheduled, I have 1 here and this comes from B i. So, it is actually a memory and this is also going into a memory say these are different memories, and the i is are appropriately provided, but the data path part of it looks like this in the unrolled loop.

Student: we were constrained by a single adder.

We were constrained by a single adder, I am skipping the dfg and the representation and so on, just to have the picture in mind regarding the final hardware what would it be.

Student: (Refer Time: 51:28)

Where will they come from all the B 0 to B? So, the other one is just other is just 1. So, that is fine, so these 1000 elements come from where.

Student: (Refer Time: 51:46)

They come from the memory.

Student: yes.

So, what do I.

Student: signal to store.

Yeah.

Student: read from memory.

So, this is the B memory which is still the same that is not changing, now what kind of connections are implied between that and ultimately this, even if you unroll the loop. In fact, you do not need anything more you can directly connect the output here there is no sharing.

So, this is a little tricky in inferring the width of that mux is not obvious just from here, it depends on how many independent paths are there from different sources to that particular input of the operand. In this extreme actually there was even no need for a

max, but if there were some registers all of which somehow at different stages in the schedules had their outputs routed into the adder then there would be a mux over there.

Student: or let us say if the data path leg path from B to adder.

Yeah.

Student: is faster than the other leg.

Student: let us assume a hypothetical case in that scenario I can have multiple signals to read memory. So, that I already have read the memory let us say for 10 signals then I need a mux too.

What this is just pointing out is that there is a dependence on the sharing of the operands, which is not obvious immediately what makes this problem complicated is that there is a phase ordering issue you need information that is actually resolved at a later stage.

Student: (Refer Time: 53:32) input address of the (Refer Time: 53:35) because there we are changing from 0 to 999.

At they at the at the address here this is the address.

Student: yes.

What needed at the address, anyway this a similar kind of a an analysis you would have to do if it is needed it is fine. So, if I actually treat this as independent constants you need to provide paths over here.

Then its alright you are stuck with that large mux, there does not show up here, but it actually does show up here.

Student: (Refer Time: 54:12)

Right, so there but the mux is necessary if I proceed in this straightforward way there is that mux such muxing might actually show up as a bottleneck, you do not have to of course, you know that there is a better way to design this it is just that at this extreme it seems as though it is not a good design.

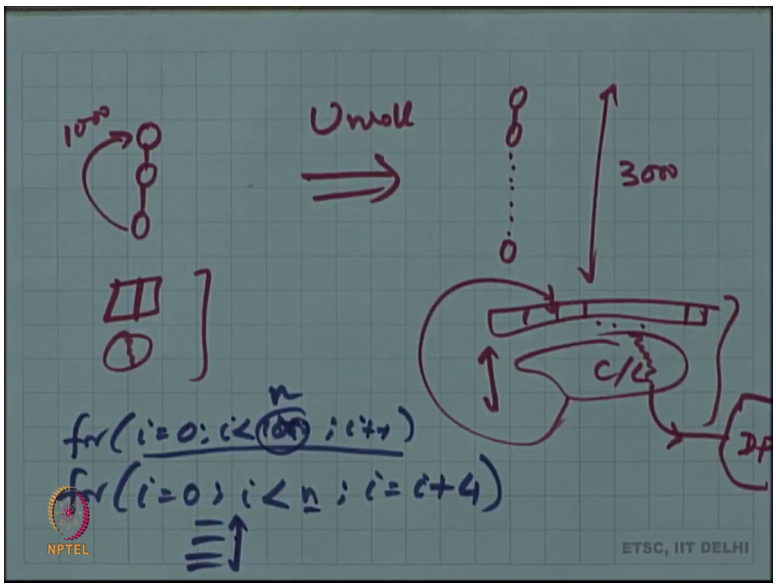
Student: (Refer Time: 54:35)

You remember that hierarchical design that we did where we nested the loop and we said that the inner loop is what would be unrolled that to could possibly lead to a difference the implication of that i plus, plus is a counter, but 1 other thing I want to point out its good that this resource sharing does lead in a non trivial way to different mux widths and we need to worry about it.

The other implication is supposed whatever the loop body was and that resulted in a 3 cycle schedule. That unrolled loop body would result in what kind of a schedule this is a 3 cycle schedule, but actually you are going through it a thousand times ok.

The under loop body it is possible that you have

(Refer Slide Time: 55:29)



That's my original loop, so in when you unroll it, what would it look like this is the fsm. we looked at the data path part of it, but let us look at the fsm part of it that is also part of the circuit that is being generated, when you unroll what is the fsm look like. I have simplified this, but essentially it is this kind of a structure loop with the iteration count telling us how many times you go through here, what will it look like?

Student: (Refer Time: 00:00)

it is a sequence yet there is no loop here right, now it is possible that because of the unrolling you do not need 3000 cycles, because there is some parallelism out there perhaps you are able to do this in 2000 cycles, you are still limited by resources that are

available, but perhaps it is a 2000, since 2000 cycles, but what is that translate to in terms of the fsm.

Student: (Refer Time: 56:26)

That's what it becomes, at least by default unless you do something about it, that is what it becomes. This is the equivalent of our other argument that there are 3000 instructions, when you do it in the processor here the equivalent of instruction is what happens in each state right. So, do you actually have a very large number of instructions maybe by default it is 3000 cycles as you just expand, the original loop what kind of implication does it have on the quality of the fsm that is generator, and fsm that has three states versus an fsm that has 3000 states.

Student: more area.

It will take more area why, will it take more area the 3000 state fsm. Even to represent those 3000 states you need how many bits. Let us look at just the state register, state register has only 2 bits here to accommodate 3 states, but if you had 3000 states then you would need your 12 bits or something just the state register requires 12 bits, implemented in this straightforward way.

We have just taken it to the extreme and unrolled everything and associated with it of course, there would be some combinational logic that would naturally be more complex than this even without knowing what is there here you would expect this fsm to be a lot more complex than this, area there is a difference. In fact, delay also there is a difference delay through the fsm means this is the sequential part, but this combinational logic among other things is generating what output.

Student: of the fsm.

Those this next state indeed, but what does what else is going next state is the next state logic output consists of those select signals control signals that are going back into the data path right. So, this is actually part of the clock cycle you have to allow for the signals to pass through the combinational logic of the fsm and into the data path.

Therefore here that combinational logic was simpler, here it was more complex and it means that essentially you have to subtract that much from what is available for the data

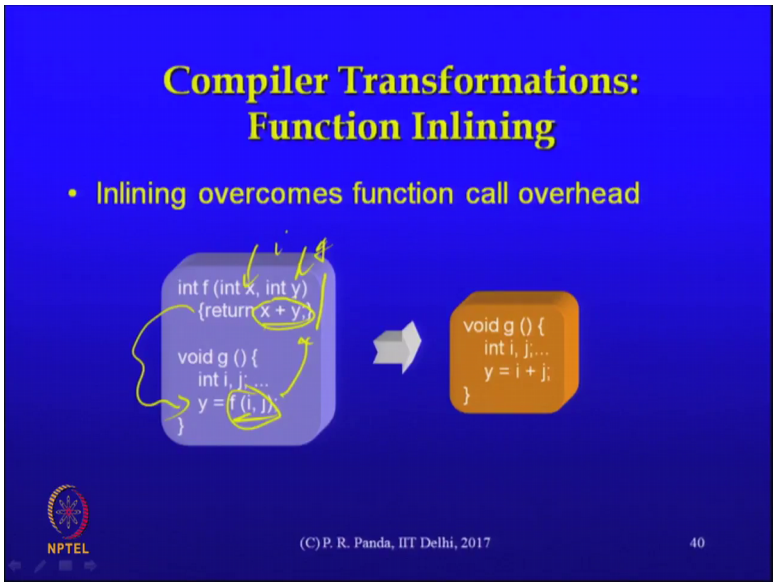
path components. The effective clock width that is available might actually go down in that case where you have a very large fsm.

So, this discussion came early on, but I did have some discussion scheduled for this at the later stages where we were identifying such phase ordering related issues in synthesis, but we do need to worry about this. So, in both the cases compiler and synthesis cases it does not make sense to arbitrarily unroll a loop even if you have all the information sometimes you do not have enough information.

We had this loop for i equals 0 i less than 1000 i plus, plus everything was known about that loop. And even then it wasn't worth it sometimes you do not know this, right you do not know how many times it is going to iterate and therefore, you cannot of course, completely unroll the loop because you do not know the value of n.

The partial unrolling is still possible right our rewriting of this as equals i plus 4 then the 4 loop bodies this is still possible you do have to put in some checks because, this n might not be divisible by 4 you have to have some patch up code at the end, but it is certainly possible performance benefits are there at least across those 4 consecutive iterations. So, even if n was not known unrolling applies as an optimization. So, since we did that in detail let us move to the next.

(Refer Slide Time: 60:43)



Related transformation called in lining, much of the arguments are similar, but essentially this is an optimization where there is a function call. And in that function call is simple as is the case here, there is not much happening in that function then the call to that function could be replaced by just the body of that function why is this useful.

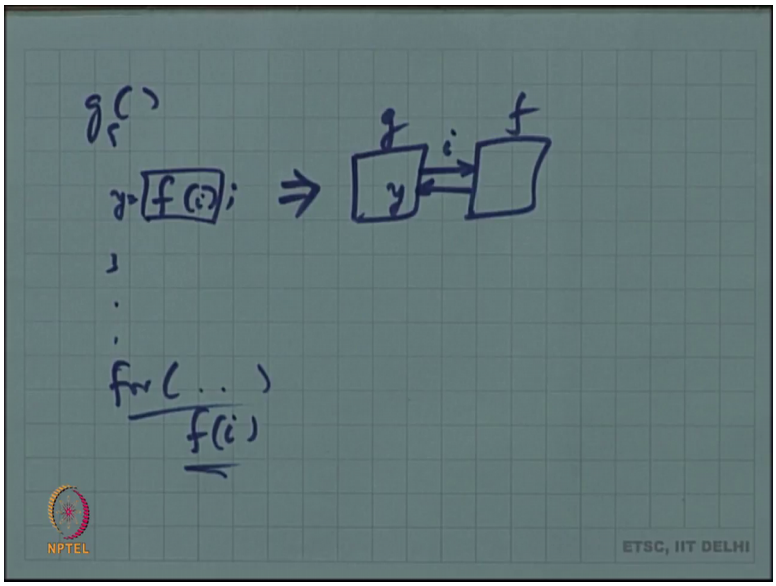
Student: sir stack instruction stack.

The function call is associated with some overhead certainly what is that overhead i j needs to be pushed onto the stack. They need to be retrieved here and similarly there is some protocol that returns that value as we resume in the calling function. So, certainly some few additional instructions are required to achieve the function call. So, f goes away completely and f i j just becomes i plus s.

So, i and j are propagated through these formal parameters and all the appearances of x and y those formal parameters inside are now substituted by occurrences of i and j. So, at least wherever it is simple it is an optimization that is often performed by compilers, the same thing in the context of synthesis at least its applicability is the same. We didn't say how functions would be synthesized there could be a couple of ways, but at least 1 way to synthesize functions would be.

In fact, to inline all the functions and then you just build a data flow graph for a single function and go ahead with the synthesis, the explicit idea of a stack and whatever the protocol is for a processor based design need not be respected in the synthesis. But one way is you just inline everything not all functions are always in linable in that way, and you could also take the view that you have a function g within that you call some function f.

(Refer Slide Time: 62:54)



And in fact, this function could merit a separate piece of hardware by itself that you could also design it in that way, g is 1 module and f is actually a different module and the call from let us say you pass parameter is here y is equal to f of i. The hardware transformation could be this g is synthesized into something, f is synthesized into a different function i is something that goes as an input from g to f.

And the result is what is captured in y value of g you could treat it as this also particularly if that f is available as a module that someone else designed and you are just instantiating that, that is another way to implement the function either way there would be some benefit from in lining it that of course, has to be weighed by different considerations that are there, but if this is to be separately.

Synthesized you could say that if this was simple enough then there is no need to have a separate instantiation of a design I just make it part of this, but this may not be a good idea if this was actually a complex function that deserved its own maybe you already have access it is an optimized implementation, hand optimized implementation. You have of a particular library component like an fft or something like that it may not be a great idea to take that function and inline everything here, because some optimization possibilities might be lost perhaps it is a manually optimized design.

So, the argument tends to be similar in that if the function that is being called is relatively simple, then there is merit in considering in lining that function in the place that it is

called. As it becomes more and more complex the value of the inlining has to be more carefully analyzed for example.

F might be called a 100 times at different places, if you inline every one of those calls then it may lead to increased hardware increased complexity. So, this has to be balanced by different parameters that affect the inlining, but what should just be recognized is that there is a possibility of simplifying the specification by doing an inlining. Just like there was there danger of the code becoming too complex and therefore, the design becoming resulting design becoming complex in the unrolling case, in the inlining case also there is the danger that if you arbitrarily in line a complex function that is being called a large number of times.

Which is not unusual you may have a loop, in which you are calling a function if you decide to unroll that loop then the calls become a very large number to that function f and therefore, we do need to separately consider whether it is worth inlining the function at all you had a question.

Student: (Refer Time: 66:60) is something similar to flattening during rtl synthesis.

The yeah flattening is the same as in the terminologies is different here, but inlining and unrolling are all essentially the same an rtl synthesis tool would. In fact, work by unrolling a loop completely or similarly in lining of a function completely since, that tool does not have the flexibility to expand 1 clock cycle to multiple clock cycles, whatever you have specified within 1 clock cycle has to be done in that same clock cycle.

You can’t afford all of this, remember all of this was possible just because you are allowing the synthesis tool to choose, how many clock cycles to take if it must be done in a clock cycle you have to generate combinational logic for that loop; however, complex it is and therefore, you have to unroll the loop you have to inline the function right.