Synthesis of Digital Systems

Dr. Preethi Ranjan Panda
Department of Computer Science & Engineering
Indian Institute of Technology, Delhi

Lecture – 9
Compiler Transformations in High Level Synthesis: Constant Folding, Dead Code Elimination, Constant Propagation, & Strength Reduction

(Refer Slide Time: 00:36)

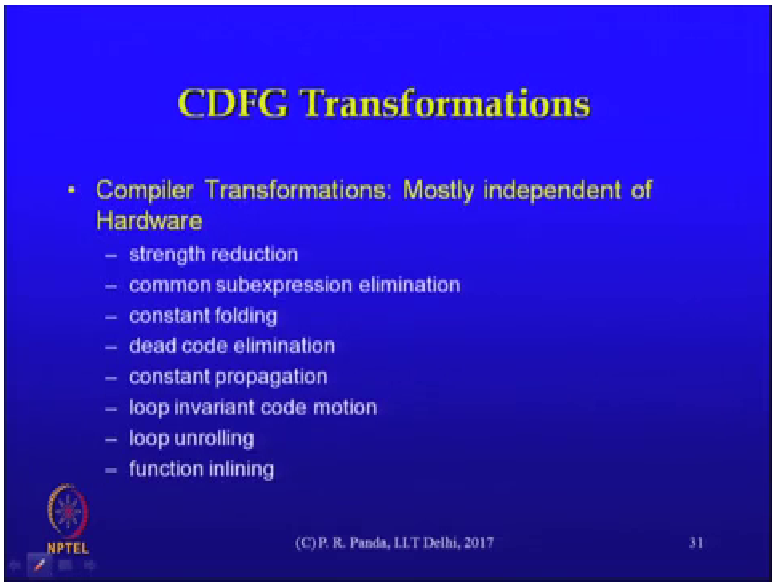
Contents

- Introduction
- HLS tasks
- Design representation
- Compiler transformations
- Hardware optimisations
- Scheduling
- Register allocation
- Challenges

(C) P. R. Panda, I.I.T Delhi, 2017 30

So, let us move onto a new topic. This is the early transformations or compiler transformations that are performed on the input to higher-level synthesis after we have appropriately represented it in some intermediate format.

(Refer Slide Time: 00:42)



CDFG Transformations

- Compiler Transformations: Mostly independent of Hardware
  - strength reduction
  - common subexpression elimination
  - constant folding
  - dead code elimination
  - constant propagation
  - loop invariant code motion
  - loop unrolling
  - function inlining

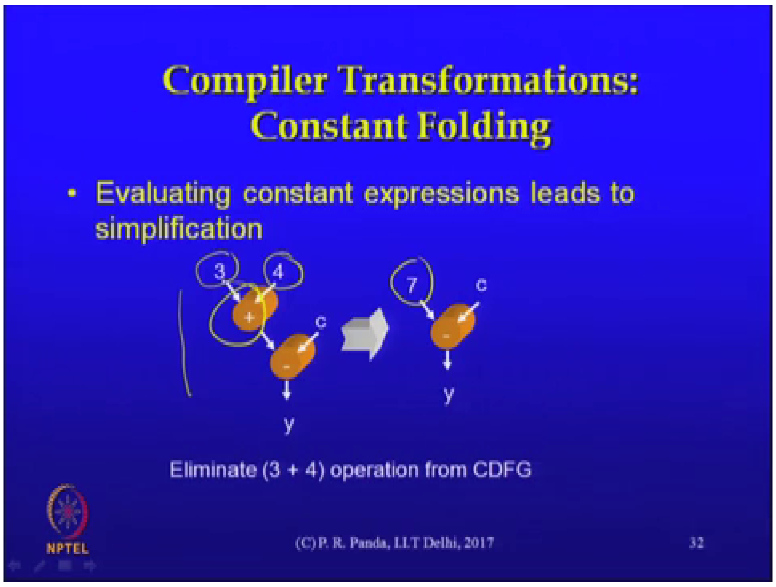
(C) P. R. Panda, I.I.T Delhi, 2017 31

If we have captured it as a CDFG then we are just calling it a set of CDFG transformations. These are early transformations they are also called here compiler transformations simply because all of these techniques have something in common with the typical early optimizations that happen in a compiler. Some of them are straightforward, some of them make sense perfectly well in both the context irrespective of whether the output is instructions in a compiler or hardware as it is in synthesis. Some of them are a lot more interesting conceptually they make sense particularly the later transformations here like unrolling, but many of them are of the simplification variety, the user has provided some input; and the objective of the transformation is to simplify that input in some way.

Now, simplification normally should lead to a better quality output typically irrespective of whether it is a software output or a hardware output, so that is what these early transformations are about. And even though I say it is mostly independent of the hardware that is what makes it common with the set of compiler transformations. Some later transformations are actually dependent on some of the other assumptions we are making like resources that are available to us.

The approach that we will take in the discussion of these early transformations is that we will highlight the possibility, where is the opportunity for those kind of transformations that is what we will identify. We would not necessarily go into too much detail on the algorithms for performing each of those optimizations. As you will see those optimizations make intuitive sense. The moment we illustrated with an example, but some of them may be not so trivial when it comes to implementation. We will not talk in too much detail about the implementation and the algorithms for doing so, but we will highlight some of the interesting aspects of it that is the approach that we will take. Mostly these are illustrations of the possibilities for optimizations, but maybe as part of the project or assignments, we could explore one or more of these optimizations.

(Refer Slide Time: 03:35)



Compiler Transformations: Constant Folding

- Evaluating constant expressions leads to simplification

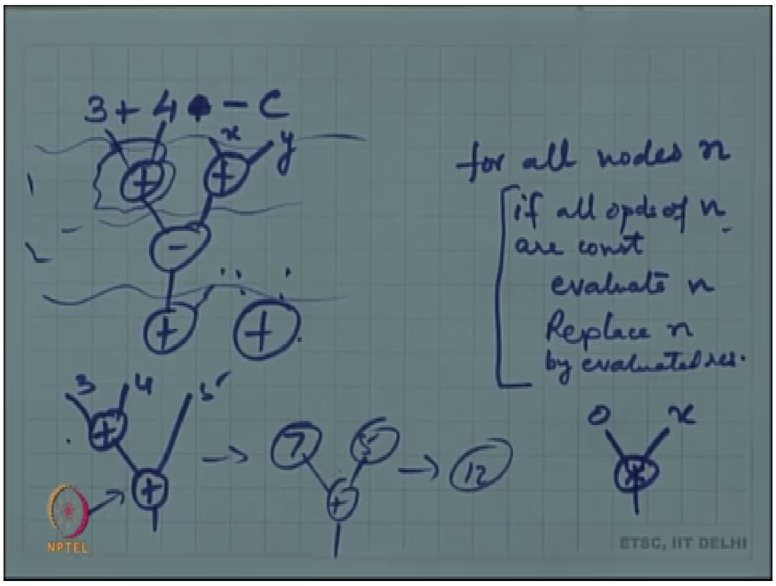
Eliminate (3 + 4) operation from CDFG

(C) P. R. Panda, I.I.T Delhi, 2017 32

In the class of simplifications, let us start with compiler transformation called constant folding. Just refers to evaluating constant expressions at compile time as opposed to runtime in a software context, here in the hardware context, what it just means is that you have some expression. There is an opportunity to statically evaluate that expression within our synthesis tool itself, instead of scheduling that operation at runtime, which will cost something, it will cost time, it will cost to resources and so on. This simple example should make it obvious, how this sub graph got derived we are leaving out it should be sort of obvious what the original HDL statement would have been that lead to the creation of a graph like this.

As you can see this is a sub graph, this is not a complete graph, we have for example, for these three and four; we did not create separate nodes. Usually in a graph, you do not have an edge without two nodes one at the source and one at the destination. What this is typically indicates is that it came from somewhere maybe this is just part of the graph that we are taking about. So, this is an expression that would have originated from a statement that had these operations.

(Refer Slide Time: 05:04)



So, minus c something like that that is what it came from hopefully, it is obvious from the graph it should be obvious what kind of HDL statement it might have originated from. Anyway all these behavioral statements need to be addressed in synthesis, we have to create the respective hardware. We illustrate the transformations through just a transformation on the internal representation, not necessarily either the source code or the hardware which is going to implement that source code, but this intermediate illustration should make it obvious.

First of all what its origins, where it came from and also because of the simplification what kind of hardware efficiency is obtained; I should be able to connect the dots, but we will talk about mostly the transformations that happen on the intermediate code. So, what happens before and what happens later we can discuss if it is non trivial, but in most of these cases hopefully the implications are obvious right.

Here what is the implication, why is this a good idea to eliminate that operation 3 plus 4 and replace it by that partial computed result of 7, what is the point how does it help?

Student: Resources are being reduced.

Resources are being reduced, how?

Student: (Refer Time: 06:30).

So, this 3 plus 4, if I had not simplified it, I would have scheduled that operation in one of the cycles which means that it is taking a clock cycle that resource that is performing the addition that adder resource is blocked for this operation during that time period right. So, if we have enough information to actually evaluate it statically at compilation time, then the obvious question is why not right, it will say something. So, that is a kind of simplification that is behind a number of these early optimizations, the argument is that this simplification makes sense irrespective of the details of the hardware architecture, irrespective of how many resources are available to you right. Make sense means what you might not always end up in let us say we are talking about it about time being saved, because part of the computation we have already done during the compilation time. Is time always saved as a result of this or only sometimes?

Student: Depends on the availability of number of (Refer Time: 07:47).

It depends on the availability, of course, in this particular example, it is obvious that we are saving time, but there is this question of should I be looking for am I actually saving time or when I find an opportunity for a constant folding I just go ahead and do constant folding.

Student: Should always do.

Should always do, but does it always help?

Student: No sir, like let us say (Refer Time: 08:12) synchronous circuit.

Yeah.

Student: Well, like addition has to be performed at one cycle and then subtraction is in the next cycles, then probably I will be shorter cycles and may resulted to some errors in the circuit.

This particular example, it is clear if these were single cycle operations then instead of two cycles, I am taking one cycle, but.

Student: This linear dependency is not there.

I may have a more complex, let us say this is my graph right in which 3 plus 4 refers to just one part; but this on the right, I had x y variables, I cannot get away from scheduling them. Then it is not clear that you are saving time; even if you discard this part of the computation, because there was enough information at compile time. You still need that much time cycle one, cycle two are still needed. The argument though is that the solution you expect would be better than or equal to the solution without doing constant folding.

You might not be able to prove that it is always better, particularly we are doing it early on right at this stage, we have not done a number of things, we have not done scheduling for example, right. So, without doing a schedule, it is hard to argue that this particular local transformation that you made is definitely going to save me time right. But similarly in terms of area from an area point of view is it worth doing the constant folding operation?

Student: Yes.

Student: Yes sir.

Always?

Student: No.

Why, no?

Student: Means I might anyway need let us say three adders in this case.

Because we are talking about these two cycles, but maybe those two adders are there as my resources, anyway in some other cycle I am going to use two adders. So, then it is not clear that you are actually saving on the resources, but the general argument is that it may help, but it is hard to argue that it make it worse.

Some of the optimizations are of that nature, they are very interesting ones and we should look at a few of those, but this kind of simplification can only lead in one directions. The other part of the discussion is how do you do constant folding. I should in order to implement this intuition, I have to write an algorithm for performing constant folding. What would that algorithm be input to that algorithm is what is the representation it is in the form of that control data flow graph and so we understand what that looks like and the what I have here on the left a sub graph of the data structure. What is an algorithm for doing constant folding. Let us try to understand some of the issues involved by just writing a very simple algorithm. What is a very natural algorithm for doing constant for?

Student: First level would be just to parse the whole graph and look for the constant and remove the (Refer Time: 11:21), and replace it with a and other constants.

Student: Constants.

So, let us traverse the graph one node at a time. The order is a little tricky, but let us say I just picked them in some order. I have for all nodes n what should I do. So, I am let us say either here or one of these nodes, and what should I do for each of the nodes, it can be a very simple strategy. So, for each of the nodes I should.

Student: Look for incoming edges.

Look at all the incoming edges and.

Student: Check.

Check it if all of them are constant. Remember if one of them is constant it does not necessarily help if you have 1 plus x constant folding does not immediately apply. So, if I say if all operands of n are constant then

Student: (Refer Time: 12:23).

Then you just evaluate that node right there you have enough information to perform the evaluation, then we will say you evaluate. And I am just informally writing it; obviously, all operands involves what kind of operation, this also may involve in the general case loop over all the incoming edges of that operation. But anyway number of incoming edges for an operation may not be too many maybe it is a binary operation, you have only two or max maybe three not too many. So, I am leaving that detail out, but in the general case depending on the way you have designed the graph, this may be actually a loop inside the outer loop. Just understand that in the general case, it is you have a loop there, but a loop with not too many iterations is expected there, you evaluate them. After evaluation?

Student: (Refer Time: 13:31).

Yes, there is a replacement that happens you take part of the graph right here we took a part of that graph and replaced it by some other graph. So, in the general case, you are just replacing maybe it is n itself by the evaluated result. There is some patching of the graph that needs to be done, so some edges have to be deleted a node has to be deleted and it is may be replaced by a new node all of which I am not writing in this algorithm, but at the high level that is what I would be doing. What else after that?

Student: Sir

After that, I just move to the next node right, so that is my action for one vertex one node. If it turns out that all the operands are constant then I can go ahead and do this constant folding; if not then I just move to the next node. So, we can start off with an algorithm like this. Can we comment on it is effectiveness, how good is it? These are called optimization parses each individual optimization that we perform may be called one pass through this is the pass that we are doing over the entire graph right. We are looking for all opportunities that exist for performing that optimization in a local sense and we are performing.

So, an optimization parse is good if it can detect and act upon all the opportunities that are there in the users designs. Can we comment upon is this good enough or it needs further improvement? Can you think of expressions that we write in which it is clear when we look at it manually that there are some opportunities for folding constraints, but this algorithm does not catch it.

Student: Sir, we are not deciding on the direction in which will move in graphs, so let us say start from the top.

We have not decided on the direction, yeah, yeah, how does it tell now knowledge of the direction.

Student: So, it is more of like let us say I have (Refer Time: 16:17), if I have (Refer Time: 16:21) bottom of the graph, and then start folding from them I may miss out one of the opportunity which could have folded if I started in the sorted.

Yes I think both of you are talking about similar limitation say I had this 3, 4, 5 that was my graph. It is obvious that there is an opportunity for folding out everything my whole graph, but that algorithm does it actually do that or not? No.

Student: No.

Student: If we start with.

It depends on the order in which you process the nodes. If you start with that node and replace this by 7, then in a second phase you can replace the whole thing by just one result. On the other hand, if you start from here, then what happens?

Student: (Refer Time: 17:16).

You actually fail this condition does not hold because all the incoming edges are not constants, therefore, this is not good enough. So, at this point, we are just trying to identify what it is weaknesses are you could try and fix it, but this is one kind of weakness. What other kind of weakness would be there, before that let us try to resolve how do you fix this?

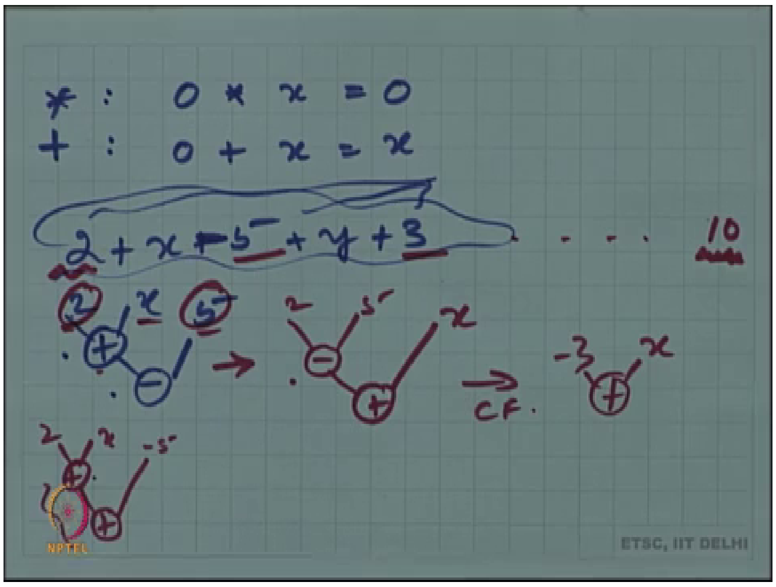
Student: Sir, recursively the next in this example, so now if I look at my so the parent of this node, I see if my parent is already visited or processed by my algorithm, if it is not we checked for that first, if it can be optimized if not similar.

There are traversal orders of the graphs that we can follow to resolve situations like this. Let us say cycles are not there in this graph, which is right in our CDFG the individual DFG there are actually no cycles. So, this if we apply on an individual DFG then somehow we should proceed from top to bottom and not from bottom to top intuitively, but the way to achieve that is this kind of orderings of the graph traversal will make sure that if there is a dependent node that is what there is a dependency in these two. So, if there is a dependent node, then the node that is dependent on some other node waiting for a result from some other node should be computed later. So, though we should start from those nodes that have the least dependencies; if we do that then we can resolve a situation like this, but that is just one limitation. Can we think of other examples that this algorithms not is not good enough to detect here?

Student: Sir if form an upper block we have a computation which has variables, but always returns in a constant, for example, if we have a multiplier as block and we have to zero as one of the inputs and any variable that is the other inputs. So, the output (Refer Time: 19:15)

So this is actually one could call it as part of constant folding, the fact that you have some kind of operations in which the knowledge of one of the operands is enough for us to know what the result is. So, then you could think of a set of specific values that help in this kind of simplification. If you have one of the operands zero to a multiplier, you know what result is. Similarly, in this class, you can think of a number of other simplifications for addition what rule can be used these are like rules that one can actually specify to the system to aid in the simplification. For addition, in what rule can you think of?

(Refer Slide Time: 20:11)



For multiplication, it is this rule. If you have 0, and it is multiplied by x, the result is 0, this certainly helps in simplification. For addition, what kind of rules apply?

Student: 0 plus x is x.

So, all of those simplifications, so if I have some such thing and associated with it there are bunch of such rules you can think of. For every operation you might come up with some such special cases that one can easily do an early look up to see whether the situation is such that it permits the application of these rules of simplification. Any other kind of situation, we are trying to make that constant folding algorithm more and more aggressive. So, we have already found some cases that I did not say how we adjust the algorithm, but it should be clear, hopefully. What is the central traversal mechanism and processing each of the nodes that does not change, you can still keep it the same thing, but the actions that you are performing inside can be more complex. You can look out for more patterns than just the simple one that we looked at.

There are some other things that are also applicable. There is a complex expression could it be that you detect the application of some local constant folding opportunities that does not necessarily get addressed by our simple algorithm. Maybe we have fixed it for a simple expression, which is just one binary operator say, what if you had ten different operands and a complex expression. The opportunities for optimization may lie buried in different parts of the expression. What is an example of that? In different parts, for a single operation, when you look at both its operands the folding opportunity does not apply, but we could do something to that expression to discover the opportunity for folding.

Student: Getting the operation into the sub parts.

Right, but what is let us just start from an example, what kind of a pattern are we looking for?

Student: (Refer Time: 22:47).

If I have an expression like this minus 5 right; soo you see that the algorithm does not work? Because it leads to that kind of a graph and obviously, in both the nodes both this and this simple rule that we have does not work. Of course, you can expand it to a more general expression in which constants are there, but they are not all nicely bunched together for us to discover, but the opportunity is there. So, somewhere so these are separated out in the expression, but the opportunity is there, it is just that that simple algorithm does not catch it. What can we do about this?

Student: (Refer Time: 23:53).

You are on the right track. We have to restructure that graph in some way without changing the meaning of the expression of course, but the restructuring might actually help. What we conceptually want to do is bring these constants closer in the graph, so that we detect an opportunity our basic algorithm need not change right. Finally, we want to apply that same pattern search which is that for an operation do you find both its operand all its operands constant. But we may need a pre processing stage where these two are actually brought closer to each other. What could that be, how can I restructure this graph so that the opportunity is more clear?

Student: Why do not we do this when we are creating this DFG usually parsing testing?

Actually, this transformation is done very early on, but what information do you have at the time that you are creating the DFG.

Student: So, when I am creating for this expression, so a simple rule like ok first I will take into the process the constants from the expression, and then the variables the graph will be created like 2, 5.

The creation of that graph even though it seems as though with an example you are thinking of a process that solves that particular example, but if we make it more complex there. In the general case, there is a complex expression and there is an opportunity to do something to combine something here with something that is there. You do need an algorithm that is general enough. If it is simple enough then we could do it during parsing, but this is a discussion that we did have during the parsing, which is if it is reasonably complex, then you do not want to burden the parsing process. Remember during the parsing there is an elementary thing that is not known; optimization opportunities are not known they exist and so on, but what is the other very elementary thing that is not known during parsing.

Student: (Refer Time: 26:09).

You do not know whether there is an error in the there is a syntax error that is going to show up there. If it is going to show up then you would rather flag it as early as possible and not really proceed to optimization steps right. Remember a optimizations are always considered little optional, certainly in early stages as we are verifying our design for correctness, we are not yet bothered about optimization, we are first bothered about whether it is actually working clearly or not that is the reason why too much sophistication is typically not introduced into parsing and very early stages. But this having said that this is still very early stage that we are talking about, we have not done anything we have only captured it in the representation, and all these analysis is happening on the representation, but every other step does come later. These are simplification steps that we start off with indeed. Yeah.

Student: What that argument that we do not want to load initial step and then we want see if it is working (Refer Time: 27:17).

Yeah.

Student: So, we can provide the optimization options so for example, in typical GCCs, you have optimization options with one you will.

Right

Student: So, similarly in such front end.

But that does not change even to a synthesis tool, you can provide an optimization levels of optimization like you would to a compiler that does not affect parsing that affects what subset of the available optimizations would be performed. But we still do not do it during parsing, but anyway, but the restructuring is easier said than done. How do you restructure?

Student: Sir, can we try to change the like if have x minus 5 and if make it minus 5 plus x and then I.

But, how do you just first of all getting back to the simple example, how do you restructure it in a way that we can discover the constant folding opportunity more easily. Two constants are there, you add two and you subtract five, what is the simplification. It seems as though as a result of whatever we do we should subtract three from this, but I should just be able to achieve that formally through some algorithm. So, what restructuring can I do, it should be the output of a series of transformations on that structure that is my input.

Student: Example one simple thing can be so when I start parsing my tree, I look at my incoming edge if one of any incoming edge is the constant then I look at my children.

But you are talking about the algorithm I have not got there just give me the restructured graph that will lead to a constant folding opportunity.

Student: (Refer Time: 28:58).

Why are you thinking so much [laughter], it is dangerous too try and impute complicated motives when somebody is asking a simple question. This is good enough.

Student: No, (Refer Time: 29:20).

Minus, but that comes later first I have to restructure the graph and then apply my old algorithm because that algorithm would work on this graph. Because it has now found an operator for which all the operands are constants, this would lead to 3, the constant folding optimization we wanted to. But what is also being pointed out is that the subtraction operation can be a little tricky sometimes for what reason it does not have that associativity property you cannot just interchange the order and expect everything to be fine.

So, the suggestion was that you have any way this is a constant that is being subtracted. So, you have this 2 x and you actually replace it with a plus and that constant instead of 5 will just make it minus 5, it is an integer and negative integer are fine. But then with this kind of a structure if let us say all our nodes were plus operations then it would be much easier, we do not have to worry about correctness and we can try out many different structures, some of which might lead to that optimization.

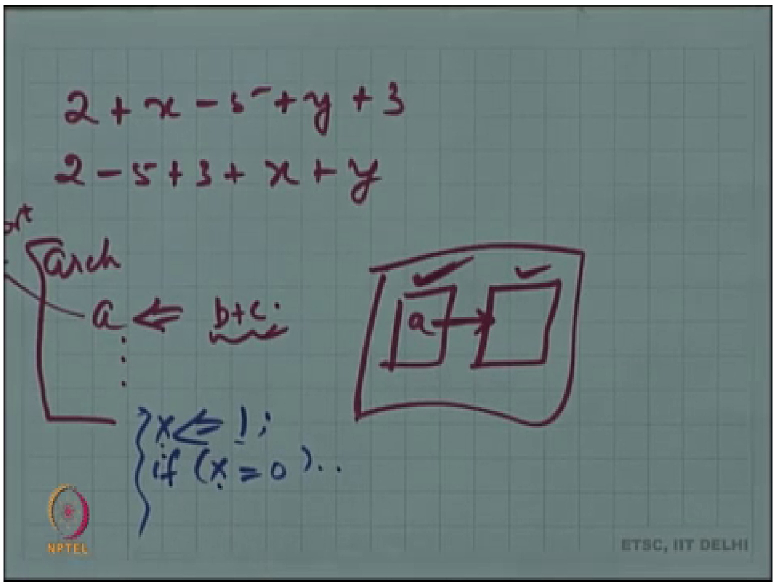
Student: Complex expression we have to take care of what (Refer Time: 30:48) is the.

So this is just illustrating the intuition that there exists the opportunity for performing these kind of optimizations. Like I said any restructuring of course, has to maintain the correctness of the specification, you cannot change the meaning of the statement. But how many such restructurings are possible? If you have one node then how many possible structures are there, how many graph structures we are saying that nodes are there. But the way they are connected is up to us let us say all the nodes are either variable nodes or constant nodes or these operation nodes and let us say for simplification they were all the same kind of operator, all of them are additions.

Even then there would be a very large number of combinations number of different graph structures which would be equivalent with respect to the functionality, but there would be different structurally that number you can actually easily see will be exponential in the number of nodes. Why is it exponential? And take that as a homework problem, it should be easy to see. Even if you fixed the addition structure operation structure, and you see that there are n slots all of those inputs all my variables constants and all of them can fit you can see one ordering correspond to n slots that are there n factorial orderings are there. So, it should be at least n factorial all of them are not necessarily legal.

If you have other kinds of operation that are not associative and so on, but in general there would be a very large number of these restructurings. So, we need some simplification in that procedure also, knowing that restructurings are there even though if I have ten operands there in my expression then I cannot necessarily afford to try out all the restructurings, and in the process find out which is the one that is applicable for constant folding.

(Refer Slide Time: 33:20)



One kind of heuristic that is often performed is you have an expression of this nature 2 plus x minus 5 plus y some such expression is there. You try to push all the constants to one side of a larger expression. In the process, you hope that you can discover as many possibilities for constant folding as possible. So, from here you can say 2 minus 5 plus 3, these we try to shift to one side. And that the variables you keep to the other side that might actually help us converge faster with respect to finding the opportunities for constant folding. It is not full proof the moment you have subtraction and other kinds of operations, you cannot necessarily shift everything to one side, but it is a reasonable heuristic that is often used.

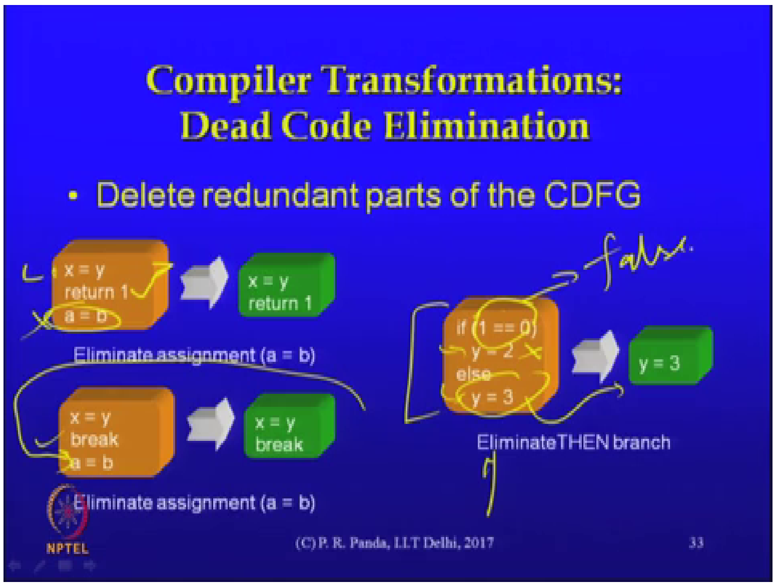
So, anyway that was just an illustration of what seems like a very obvious optimization from an intuition point of view, but to actually apply it aggressively you do need a little more careful thought in the algorithm. So, on one hand, you would like to extract and operate upon as many such opportunities as are present in the code. On the other hand, you do need that algorithm to be scalable such that if the expression is large or the number of statements is large then you still finish within a reasonable amount of time. Such a trade off is always there in any kind of optimization, this is just a simple optimization that illustrates that trade off. How much time you spend in discovering all the opportunities for optimization versus how good the quality of the optimization is.

Student: Sir, if the operator is fixed then they can simply iterate through all the edges have and every time we find the constant, we can give to what for something and then we can create a graph (Refer Time: 35:18).

If you have more information about the kind of expression, it is you can be more efficient with respect to evaluating it. Of course, yes, but just that in the general case I do not assume that operators are fair start they have the certain properties that allow any kind of reordering and so on. We discovered that as we traverse the graph and the algorithm should be such that it is able to handle all possibilities, all the operators it is able to handle in one integrated framework. So, yeah that is what constant folding is about.

Let us also take a look at a few other ah optimizations not all of them to the same level of depth, but we will point out what the opportunities are. The other thing that happens is you may be needed to iterate over a set of optimizations that should be clearer with the second one that we illustrate a second and third one.

(Refer Slide Time: 36:17)



Compiler Transformations: Dead Code Elimination

- Delete redundant parts of the CDFG

x = y
return 1
a = b
x = y
return 1
Eliminate assignment (a = b)
x = y
break
a = b
x = y
break
Eliminate assignment (a = b)
if (1 == 0)
y = 2
else
y = 3
y = 3
Eliminate THEN branch
false.
NPTEL

(C) P. R. Panda, I.I.T Delhi, 2017 33

Let us introduce this other from a control flow point of view simple optimization called dead code elimination. Idea is you delete what you discover are redundant parts of the code. When can you assert that a code is redundant some statement is redundant, there are a number of such opportunities, but the simple one here is that there is a control flow here at that second statement is actually taking you somewhere else. So, because control flow is going to take you somewhere else, if it is this is part of a function and it is taking you to the end of the function, you can assert that this statement there will never be executed. You will never reach that statement. Since you cannot reach why do not we just throw out that statement?

What benefits are there those are a different question, but at least one obvious benefit from a compilers point of view is that it has simplified your CDFG right. Now, there are fewer nodes in the CDFG, and therefore a number of other things that you do there would be dependent on the complexity of the CDFG, the size of the CDFG in terms of edges nodes and so on, from that point of view at least it is worth performing the dead code elimination kind of optimizations.

So, there are a number of such control structures which might lead to some statements becoming redundant. Break is a similar kind of statement it is. So, return was taking us out of the function, break takes us out of a loop or something like that. And in that process, it would be that maybe we can assert that some particular statements become redundant. Of course, as we make these decisions we do need to take care that it is not possible for us to jump from some other part of the code to that statement that we are about to drop that check does need to be made. How do you make that check? Often compilers do the following they look at whether there is a label that you have attached to some statement. If you have attached it may mean that you have a jump statement from some other part of the code to that label.

The definition of the basic block which we defined as sequence of instructions with one entry point and one exit point that is influenced by the presence of labels because it may indicate an alternative entry point. So, you do need to check to make sure that it is not possible to enter that in some other way as long as that is there you could do this kind of redundancy removal. So, from a control flow point of view, this is a simplification. And once you have identified it, it should be clear how to simplify it.

The other kind of simplification is the following. This kind of a statement has some redundancy in it. What is the redundancy that can be anticipated at the compile time, where is the redundancy? That condition can be pre-computed right. Since, it can be computed statically I already know whether I will be entering this branch or this branch, there is nothing dynamic about it. Even if it is part of a loop, every time I come to the if statement. In fact, I would never reach that part. So, I could in fact throw out the complete conditional and just replace it by the statement or list of statements that are going to be executed. So, this is a generalization of the dead code elimination the part of the graph becomes dead code.

Let me conclude this. By itself it is not clear that it is dead code. On this particular example, you have to do one other thing before you conclude that some code is dead what is it?

Student: The first optimization wave.

Yes, we have to perform a constant folding I have to recognize that there is an opportunity for constant folding, and this 1 equal to 0 translates to false that is the simplification intermediate step that was not shown here. And then you say that if false, then something else something else therefore, you can perform that elimination so that implicit constant folding opportunity is there. It has to be performed first if you directly start with dead code elimination, you will not detect it, because it is not false it is just you see an equality operator. And if your algorithm was coded as if it is if I know it is true or false, then I proceed otherwise I will discard it then you might not discover it.

Student: Sir.

Yeah.

Student: If I am writing through a signal, but I am not reading that signal anywhere (Refer Time: 41:12) that would be a part of dead code elimination.

If you are writing into a signal like that, but that signal is not used anywhere in the code is it dead code. So, this is the architecture. The architecture has a signal or there is a port called a.

Student: Sir we can (Refer Time: 41:34).

Fine; so let us have a port. So, this a is actually a port for us, and I did a equals 1. And I have some other things and that is my model. Is that statement redundant?

Student: No.

Student: But we if it is output port then it is input port for any other entity.

Right.

Student: And I am not reading that input port any other (Refer Time: 41:59).

Yes, it is an output port from here, and it does form an input port to some other entity and.

Student: I am not reading that input. So, I am not using.

In that other entity, you are not using it in any way; therefore can you throw out this assignment.

Student: (Refer Time: 42:22).

Why?

Student: (Refer Time: 42:23) in an entity.

Should it show up in a waveform that we plot?

Student: Yes.

The value of one being assigned to a, should show up, if the simulator gives me a way of ah plotting all the signals all the ports right, should I show up? It shows the right values should show up.

Student: Sir simulation it makes up, but synthesis occur (Refer Time: 42:47).

So, in simulation, so I cannot throw it out for the simulation. For synthesis as you are generating hardware for these modules would you assign the one value to the port or not?

Student: Sir we will assign.

You will assign, why?

Student: sir we cannot leave the port (Refer Time: 43:07) even if is not used it will be floating I mean it will cause some problems.

Student: Floating what.

Student: Sir, port optimization is not.

Well.

Student: It is optional, it cannot be it can be.

Say I had b plus c, would you still perform the addition? There is a methodology issue that the port has to be connected to something, therefore maybe I just connected, but should I perform that operation knowing that ultimately that result is not being read by anyone, should I perform it or not?

From in overall methodology point of view we cannot throw it out for the following reason. You may not have as you are synthesizing a piece of specification, you actually may not have enough information about all the contexts in which that design is going to be used. It seems as though here we know everything about it not only about that particular entity, but also that it is part of a bigger system about which we are assuming full knowledge. And during that synthesis we have not only full visibility into identity, but also into that other entity, in order to conclude that it is useless assignment.

All of those assumptions often we do not make as we perform synthesis. Not as we perform compilation, remember yeah as you are compiling a large piece of code the code might be separated out into multiple files. As you analyze one function, you have no way of knowing whether that function is going to be called from some other file or not we process things one translation unit at a time and it is often that translation unit is a file. In fact, the compiler generates it is output on a per file basis might not assume any knowledge about all the contexts in which the functions that are visible externally in that file are used. In fact, that is considered good design practice right, to not assume anything about how somebody is going to use. If you are exporting something then you assume that it will be used in that kind of situation the equivalent of the file in software here is that entity.

So, within the entity, we may perform all the aggressive optimizations; across entities sometimes we could if we flatten the whole structure and optimize it together. But if the designs that we are producing are on a per entity or synthesizing things one entity at a time which might be the more common situation, then you do not assume about who is connected and how they are going to be using it.

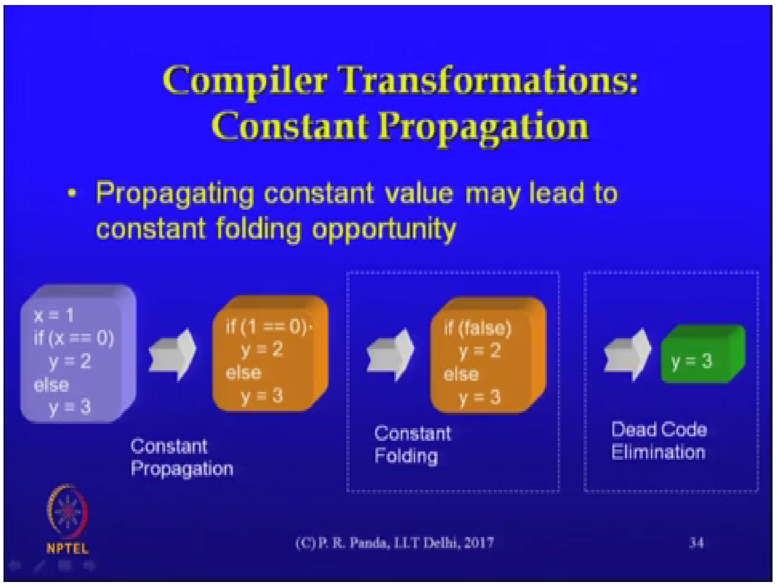
Student: A typical (Refer Time: 45:56) design process we explicitly tell ourselves to preserve hierarchies and there the port optimization should not happen, but where.

Right.

Student: Say flatten as you said do not preserve.

Yeah, it might in conclusion we might actually leave it as an option to the tool to either perform make that assumption or not make that assumption, but it often would be that if we want to reuse the design for anything else then we would like (Refer Time: 46:23), but the option might be there for us.

(Refer Slide Time: 46:30)



Compiler Transformations: Constant Propagation

- Propagating constant value may lead to constant folding opportunity

x = 1
if (x == 0)
y = 2
else
y = 3
if (1 == 0)
y = 2
else
y = 3
Constant Propagation
if (false)
y = 2
else
y = 3
Constant Folding
y = 3
Dead Code Elimination

(C) P. R. Panda, I.I.T Delhi, 2017

34

Move onto a related transformation called constant propagation that is yeah.

Student: All these optimization tricks that you are talking about they have an inherent dependence in terms of sequence in which they.

Yes.

Student: Can be executed.

There is dependency.

Student: Different benefits. So, for example, without constant propagation, we may not know that if 1 equal to equal to 0 there is a way I mean if we first propagate the (Refer Time: 47:00)

So, we are pointing this out as independent optimizations, but this is mostly from an engineering simplicity point of view and also from the point of view of presentation. What is the right ordering to use, what is the optimal ordering for a set of compiler optimizations, it is a difficult problem and we had problems.

Student: Sir.

Yeah.

Student: What if it was (Refer Time: 47:24) in a same problem, I mean we (Refer Time: 47:27) discuss it for a port if it was a single input have been redundant and part of that.

So, this decision has to be first made as we write this kind of code we are assuming that these are all variables with local scope, you can do whatever you want throwing out something is ok; often throwing out something is not ok because of reasons like that. Signal assignment you may want to preserve the history, a simulator definitely is obliged to preserve the history of a signal assignment, all through the simulation. Even if the only change only assignment happened only at time zero then too we are obliged to preserve it. Synthesis we looked at this example of whether we should keep it or not, but for signals and ports we have to define at a higher level of abstraction whether we should optimize.

Nevertheless even the optimization opportunities exist like all of these things can still be optimized. If HS of a signal all of these opportunities do apply there, but whether a assignments can be completely thrown away or not is more complex question. Signal assignments we may not want to throw it away, but much of this discussions we can assume that these are variables. As you can see the syntax here used is a little loose deliberately because I do not want to make language specific assumptions there, but we can assume that these are variables right.

A different illustration constant propagation refers to propagation of the knowledge of a value of a variable to it is use even though in the context of it is specified use it is actually a variable. You may actually be able to establish that the value of that variable at the time that it is used is already known; and it is known to be a constant so that is what is called a constant propagation. If I knew that this one that I have here is what flows into x and only that one is going to be flowing, then I can replace it by something like this where I say the x which was a variable is replaced by a constant. Why do I do that how does it help?

Student: (Refer Time: 49:50).

Both the other optimization that we talked about, first the constant folding is applicable, it is discoverable only if I will perform the constant propagation; otherwise it is not even applicable.

Then once constant folding application is known there is also a for the dead code elimination. You can see that these optimizations are interdependent, one might help the other right. Practically it might happen that we arrange these in the form of some sequence, and you iterate upon that sequence until there is no further improvement that is visible. This is a practical example compilers typically do this; optimizers do this. You perform one set of optimizations you iterate over the same set again maybe they create another set of opportunities for optimization go over it again. So, hopefully this does not take too many iterations to converge convergence just means that I do not identify any more opportunities for optimization right.

Student: Sir.

Yeah.

Student: What if x is a signal, which is used in some other process when we can do this.

What if x was a signal that is visible in some other process then is still valid or not valid, replacing this code by this code is valid or not?

Student: (Refer Time: 51:21).

Throwing out that assignment is not valid, but after that the rest of it is actually valid, the propagation is not an issue. In fact, I have simplified the discussion a little bit here, the propagation just refers to taking the value recognizing that this is the value and substituting that value for the constant that is all that we do when we say propagation. Throwing that assignment out is part of some other optimization I have just combined them in the interest of simplification, but that could be part of a different argument whether we should be throwing out or not. If nobody needs it then we throw it out; if somebody needs it then we keep it, but that does not invalidate the rest of the optimization, which is the application of the constant folding application of the dead code, these are still applicable.

Of course, through all this you remember that in the signal assignment, when you do x equals 1, it is not immediately visible in the same delta.

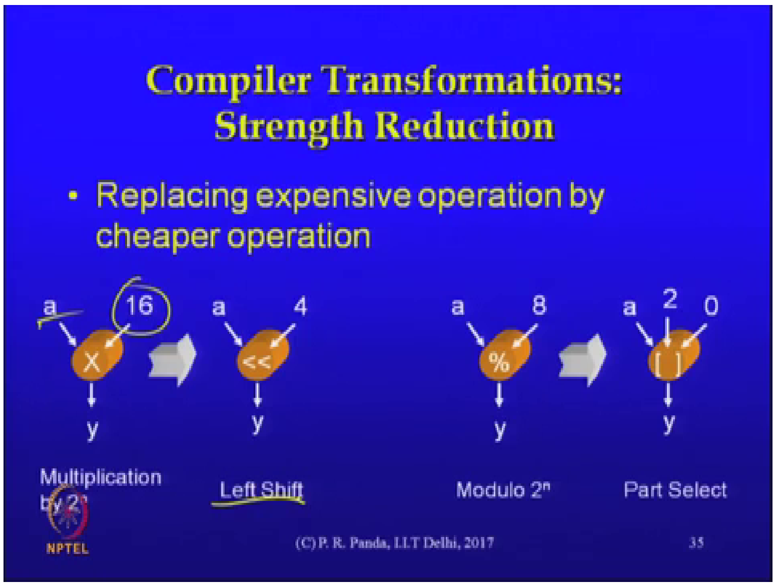
Student: (Refer Time: 52:19).

Therefore, that new value if this was a signal assignment it would not reach the x equals 0, is that clear?

Student: Yes sir.

I have a x, if I had this x equals 1 and immediately after if you had if x equal 2, so this is VHDL, so some such statement then that new value does not reach this because we did not have an intervening weight or something. You have to wait until the next delta, and therefore, that does introduce some complication here. Our rules are still the same which is that this x when we say it actually uses the old value then the assignment is made to some other value. Remember, it is a the way to understand signal assignments we said there pretend that the assignment is happening to a temporary variable. Since, it is happening to a temporary variable it is not visible to that immediately following statement and cannot be optimized away the constant propagation does not work right.

(Refer Slide Time: 53:19)



Compiler Transformations: Strength Reduction

- Replacing expensive operation by cheaper operation

Let us talk about a few other such simple transformations strength reduction refers to finding ways of replacing an expensive operation; however, we define expense in terms of area or power or time with a cheaper operation that does the same thing. Example would be that you have a multiplication with a number that is a power of 2. This is an integer and this is a multiplication with the power of 2. I have with me a set of rules that tell me that these two are actually equivalent. One can do a local replacement of a multiplication with the left shift operation that does the same thing. Will it make sense, why are we saying this is a cheaper thing to do?

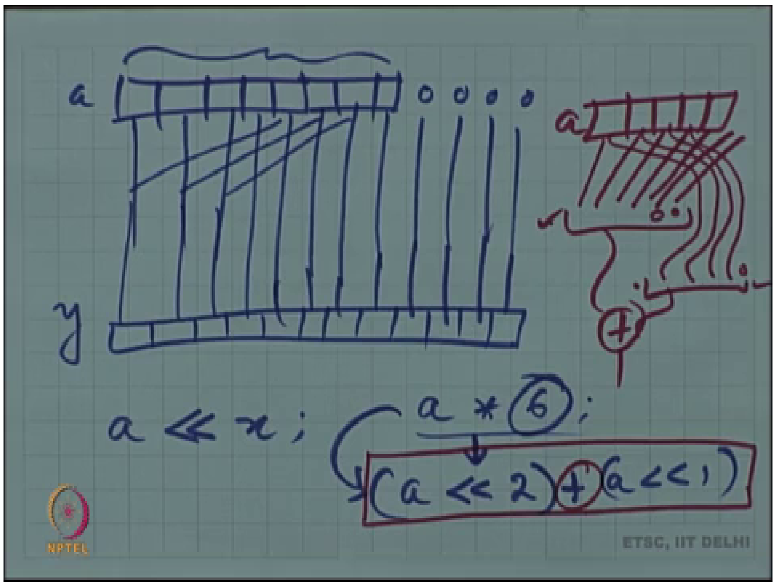
Student: Because linear register or that.

How do you do multiplication operation, you need a multiplier right. If you do left shift a is stored in a register say, how do you do left shift by four?

Student: So, we just (Refer Time: 54:30)

Student: Actually in case of multiplication.

(Refer Slide Time: 54:36)



So, let us say I have a five bit register a is here and so far better make it some more bits 8 bits, how do you do left shift by four?

Student: (Refer Time: 54:49).

So, somehow it seems that I should take these four bits or at least I can assume that there are these four zeros out there, and it depends on what it is that I am assigning it to, but it could be that if I am assigning it to it 12 bit number then that is. So, what I have right. So, what hardware was needed to perform the left shift?

Student: (Refer Time: 55:28).

Is it shift register that is needed, can you do it something else that is simpler than a shift register. This is my a, and this is my y, which is a shifted left four bits you could use a shift register, but what else could we use.

Student: (Refer Time: 55:53).

Where is the surface where should I put my shift register here?

Student: No sir, in case of shift register, only the carry flag. So, the (Refer Time: 56:07) goes to the carry and other eight bits are like same (Refer Time: 56:13) like throwing the zero flag from the right side so again and again four time you will throw the zero to the register. So, the register will be same of the eight bits of it and the carry (Refer Time: 56:24).

Whether it is 8 bits or not it depends on what I did not say what is that type a, what is the type of b. Let us say we had allocated 8 bits for a and 12 bits for y, then these 8 bits of a still become the higher 8 bits for y, but the lower 4 bits can become 0. Or whatever the algorithm is I can perform the appropriate manipulation in the way I pick up the input bits, but what hardware is needed and y, y needs to be stored here. I need if this y is to go into a register then this is a 12 bit register that is needed. So, where should I put the shift register, you remember if y was 8 bits then what would we do.

An algorithm is clearly there what the shifting by 4 bits or 4 bits, or whatever it is just trying to find out what is the hardware. So, that we have to be able to argue when I say cheaper I have to have a hardware structure in mind right. So, that I can say it is cheaper than a multiplier right. So, what hardware are we talking about here?

Student: Sir, if y is a 12 bits, we do not need a shift register because we can just extract the last 8 bits.

Yeah.

Student: So, we do not need as such.

So, there is no hardware, this is only an interconnection.

Student: Yeah.

That should be understood, there is actually nothing there is no operation, there it is just a matter of me picking up the right bits from somewhere, I may need to handle the msb bit where if there is a sign bit or whatever I do need to worry about it. But there is no shifter that is obviously, implied by a an operation like a left shift or a right shift. There is a shifter circuit in general you have to instantiate that when, for what kind of expression will you actually need to instantiate a shift circuit?

Student: 16 is replaced by a variable.

Yes, if I had a left shifted by x then you need a proper shifter circuit because only at runtime you know how much to shift it by and so the circuit is more complex. Since, it is complex you have to schedule that operation and that takes some time, it takes some area and so on. But this particular kind of class of operations which is just multiplication by a constant, translating to a shifting by a constant, the hardware required is trivially small right. Remember there is a power related argument in these optimizations that sometimes we overlook. So, one argument counter argument could be let us say you already have multiplier as a resource.

Student: Yes, that is what it is.

Suppose, you already had at that multiplier because you needed it for something else, then is it still a good idea to perform this or not? Since, from a you are not saving area at least component area you are not saving because that multiplier is already there. but in spite of that it might make sense well I have to argue area if it is just component area I am worried about perhaps it is the same, but there might be other benefits there might be delay related benefits because this is a much faster thing. But also there are there could be power related benefits because if you were to do that multiplication to perform multiply the multiplication by 16 as opposed to just extracting the bits to perform multiplication by 16, the power consumption might be very different input, yeah.

Student: Let us say if I have multiply a with some number let us say a into 6, then I will need a substation also (Refer Time: 60:25), so I will.

If you had this kind of a multiplication.

Student: (Refer Time: 60:28).

Which is not a power of 2 nevertheless is realizable in terms of an adder would be enough to perform that kind of multiplication, you are right. So, I should be looking for such opportunities, I take that constant, this is a constant. So, I can do a lot of things, a lot of static analysis could be done to find out what kind of hardware do I really need or what is the cheapest hardware with which I can perform that computation. So, I could actually realize this in terms.

Student: Left shift

Student: 3 minus.

Student: (Refer Time: 61:02).

Something like this. Right one shift by two, and one shift by one which would translate to multiplication by 4 and multiplication by 2; and essentially I need only one adder to do this. So, yes I should be looking out for such opportunities. The generalization of that is something like you have a set of variables, a set of constant this happens in certain DSP kind of computation. So, you have constant coefficients with which you multiply a variable data. Let us so several such multiplications are there, what is the best circuit to realize that set of multiplications. All of which are of this nature a variable multiplied by a constant.

The default one would be if you just had a multiplier for each one of them or you have one multiplier, and you sequence all the operations over that multiplier, but these other interesting solutions might be there. It is expected that a synthesis tool is smart enough to do some little bit of exploration to find out what is the best way to realize a circuit. So, when you have constants like this, you do have such opportunities that you can explore.

Student: Sir in this case.

Yeah.

Student: It is assumed that the multiplication is more than two cycles.

It is assumed that the multiplication is more than two cycles?

Student: No more than one cycle.

Why?

Student: The next statement should it take.

The transformation is valid anyway we have to argue whether it is efficient or not are you saying that it is efficient only if the multiplication takes.

Student: (Refer Time: 62:51).

Why?

Student: Statement takes how many cycles a shift operator.

This statement takes how many cycles, I have to first define what is the delay of all my operations, but what do you suggest is the time required to do that computation. Adder takes one cycle let us assume. What should be assumed about the time required for the shift?

Student: If (Refer Time: 63:17).

We should assume zero, well it is not actually zero because there are wires and so on, but from a component delay point of view we are saying it does not require any component it is a matter of us just taking the right bits from here one set of bits. So, I have my a, shift left by two means that this is one operand, where these are zeros. And the other operand is shift left by one which is right here, and make that zero that is my other operand. So, as of now I have not taken any time right there is no gate that was required to derive this. After that I have to do an addition that is what it is.

Student: Sir, registers are usually reused; in this case the interconnection to happen and register could (Refer Time: 64:13) take the interconnection part.

Where is the register?

Student: A

Which register this one?

Student: Yes.

Right a is stored in a register right.

Student: After the shifting.

After the shifting is there a need to store, these intermediate values in a register.

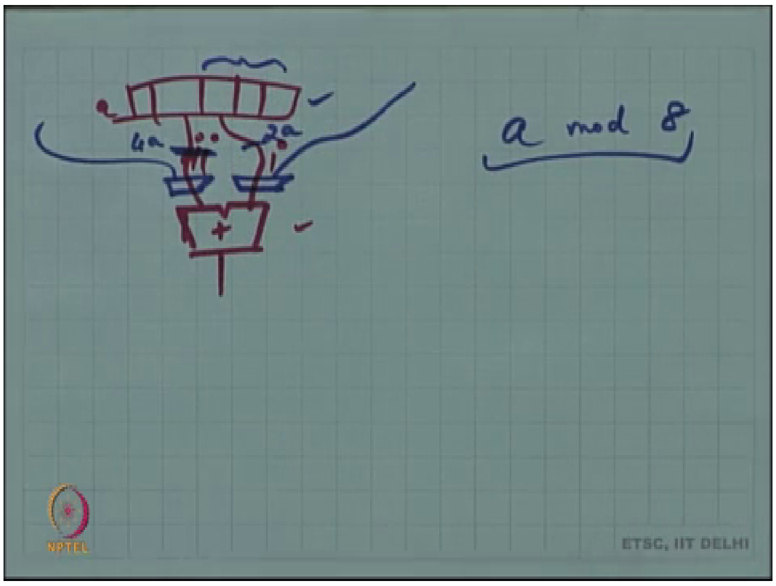
Student: To feed it to the adder (Refer Time: 64:33) into a register.

Why?

Student: You will have a dedicated adder just for this, even a bit even a reasonable set of data flow in that sense would such transformation makes sense (Refer Time: 64:47)

Let us find let us go to the bigger problem later on, but let us just concentrate on this, this small problem here. Question is do I need storage for a shifted by two and a shifted by one or can I just take directly.

(Refer Slide Time: 65:08)



I had my a, and I had my adder this is my adder circuit. And essentially I had two inputs there that somehow this was yeah this had two more zeros and it is actually the same thing that is going here with the addition of another zero. Is this a valid circuit or not? It is a combinational circuit so far.

Student: Yeah.

I did not store these in any way. In fact, I had these five bits that were coming from a instead of that I have two more bits that I have just added as inputs to the adder, here is one more. Is this a valid circuit as of now? It is valid. So, when does it become not valid. So, suppose I had that adder was reused in some other cycle to do something else not this particular addition.

Student: (Refer Time: 66:00).

That will lead to us placing a mux here and a mux there. So, the second operand comes from some other part of the design

Student: (Refer Time: 66:14).

This is actually no different from the standard resource allocation that we have already talked about. As of now unless I am going to need that four a or well it is hard to argue that I need it in a way that if a is still there, then 4 a it can still be derived or 2 a can still be derived 8 a can still be derived. If I have reused that register for a in the meantime that decision as of now we have not taken. Remember as we are doing these operations here high level transformations, we have not gone through all the synthesis steps of scheduling and resource allocation and register allocation and so on.

So, unless I have lost that space for storing a itself, I can certainly derive a shifted by a constant number of bits whenever I need it, but it could be that at some time 4 a is what I am storing, but a is not needed. So, let us say I reuse that register for something else then maybe 4 a is needed later on and I need to store that value that was derived here right 4 a and 2 a, but it is often possible to do all this in just a simple combinational circuit that does not require additional storage.

You can easily identify a subset of these operations for which there would be one default way of realizing them modulo division is also a similar example. The default one would be performing division and getting the quotient and the remainder, but it could be that often if the other operator there for the modulo is a power of 2, then of course I can realize it in a very simple way what would that be. So, a 8 eight can be simplified when it comes to hardware realization as what.

Student: Checking the last LSBs three.

So, a mod 8 would be you just take this three LSB bits it is what is called a part select. We can include a dummy operation like a part select it happens in zero time as far as gate delay is concerned; for the same reason that the shift operation by a constant also happens in zero time. So, strength reduction means that we have moved from a more complex operation to a simpler realization of that same operation. And there are some obvious benefits there, because the hardware would be simpler which means that usually there would be an area benefit, there would also be a performance or a power benefit.

Most of these optimizations that we are talking about at this stage belong to that class where we can argue that it makes sense to perform that optimization. Even though we do not know a lot of details of what is coming up, proper synthesis has not started yet like what the resource constraints are all of these things are not known scheduling and all of those steps are not performed yet. Yet we can argue that a certain set of optimizations actually makes sense, these are the class of optimizations that somehow lead towards simplicity that simplicity it makes sense to argue about.

Even if for example, you do not know anything about the target architecture, whether it is an ASIC, whether it is an FPGA, whether it is a processor all of these things usually can be ignored the he target architectures can be ignored for most of these optimizations. Some of them are interesting because they may represent tradeoffs, but often the simplification operations make sense to move in that direction. Remember that they may or may not result in improvement all the time that does have a dependence on how you realize it, but it would not be worse it would usually be better than or equal to. Let us stop here.