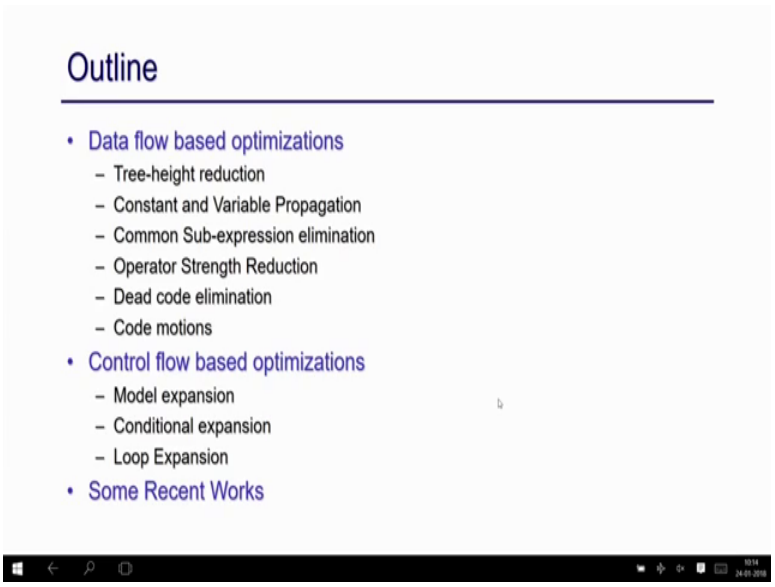
Optimization Techniques for Digital VLSI Design

Prof. Chandan Karfa
Dr. Santosh Biswas
Department of Computer Science & Engineering
Indian Institute of Technology, Guwahati

Lecture - 05
Impact of Compiler Optimizations on High-level Synthesis Results

Welcome everyone, today we are going to discuss on the impact of compiler optimization on high level synthesis results ok.

(Refer Slide Time: 00:37)



Outline

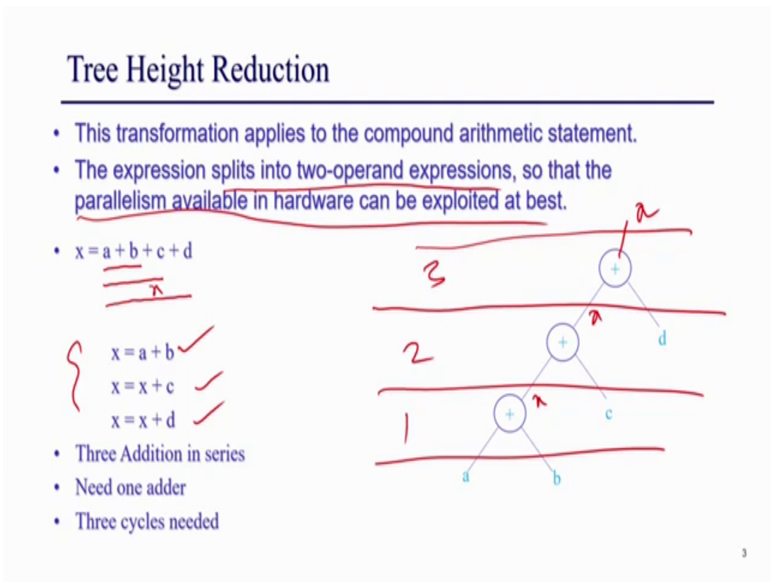
- Data flow based optimizations
  - Tree-height reduction
  - Constant and Variable Propagation
  - Common Sub-expression elimination
  - Operator Strength Reduction
  - Dead code elimination
  - Code motions
- Control flow based optimizations
  - Model expansion
  - Conditional expansion
  - Loop Expansion
- Some Recent Works

So, if we you look into this this software domains; where if you just say any compiler like C compiler this is your any other domains. So, you have a lot of optimization techniques are applied on that ok, because we have a C code, and that is going to map to a assembly level code by a compiler, and maybe that code is not optimized wanted. So, there are a lot of research happened and there is a very very rich domain of area in compiler optimizations that are actually applied and we just see how they are affecting on process, that I mean that execution in the processor or of a behavior.

So, similarly when you are thinking about high level synthesis; so, we have gone to obstruction level as high as C code right. So now, we are starting from a C code and we are generating hardware. So now, the question again how those compiler optimization techniques can have an impact in the hardware that is given to synthesize through high level synthesis. Because we have starting from a C code and so, we have the scope to exercise this particular practice just to see how this optimizations can have impact on the generated hardware through high level synthesis.

So, that is kind of the discussion topic today. Specifically, we are going to discuss various optimization techniques basically on data flow-based optimizations; like, tree height reduction, constant and variable propagation, this common sub expression operator strength reduction dead code elimination code motions, techniques like that also whenever you are leaving. And also, certain compiler optimization technique like a module expansion conditional expansion loop expansions. And also, some recent works on this area, basically just to review this techniques primarily with some examples. And then we try to see their impact in the generated hardware through high level synthesis.

(Refer Slide Time: 02:32)



Tree Height Reduction

- This transformation applies to the compound arithmetic statement.
- The expression splits into two-operand expressions, so that the parallelism available in hardware can be exploited at best.
- x = a + b + c + d

  x = a + b

  x = x + c

  x = x + d
- Three Addition in series
- Need one adder
- Three cycles needed

x
3
x
2
x
1
+
+
+
a
b
c
d
3

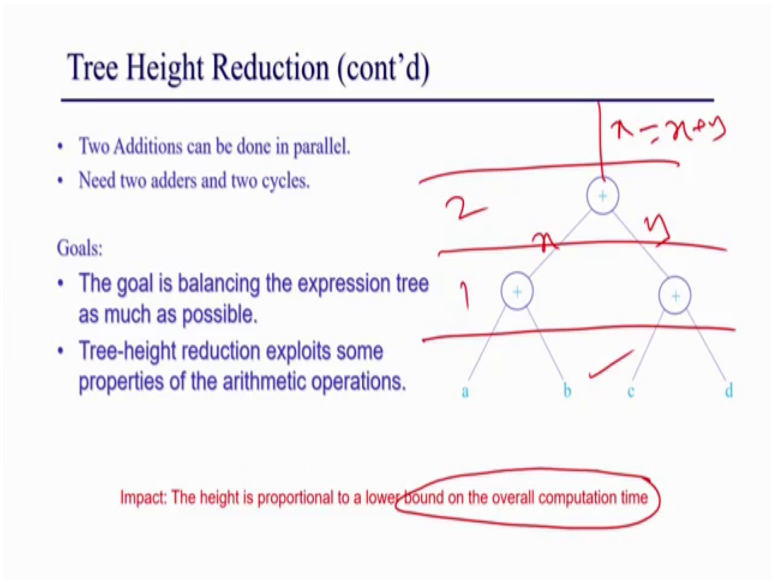
So, we will start with this data flow-based optimization. Specifically, we will start with this tree height reduction. So, if you remember, when we start discussing on high level synthesis in the pre-processing step what we do is just whenever there is a big expression say arithmetic expression, we try to split this expression into small 3 address operation right. So, that is we have already discussed during this high-level synthesis and discussions right. So, and so, that is something just to do just to because that whole expression is not going to be execute in one cycle. So, try to split that big expression into small small, expressions and just to execute the small expression when each clock each clock right.

Now, this whenever you are actually breaking this expressions, there are various way to do it right. So, whenever you do in certain way, and there are various way to do it and not all of them result in same kind of critical part of the number of types time times required to execute that behavior right. So, that is what is related about this tree height reduction. So, whenever you are arithmetic expressions, you just try to split them into 2 operand expression that is it is 3 address code, such that we can exploit the parallelism available in the hardest at best ok. So, for example, you take this example of this very simple example a plus b plus c plus d. So, what we can do? We can do this first right, then we can do this is say x, and then we can do this plus this. And then this whole plus d, right.

So, that is what a x equal to a plus b, then x equal to x plus c. And then this is x now then x into x plus d right. So, this is what I am doing. So, if you just draw that dependency here. So, we compute a plus b here. So, this is x now, and then x plus C this is x now and then this is final x. So, you can see we need basically there is a sequential dependence among these operations, and I need 3 cycle because I have I can execute this in time step one, then this is in time step 2 and this is in time step 3. So, I need 3 cycle to execute this software behavior right. Similarly, so, and also since in each time step, I have only one addition operation is happening. So, I need only one adder right, but I need 3 cycle to execute this right.

On the other hand, if we just execute in this way that I do this a plus b first and c plus d in parallelism.

(Refer Slide Time: 05:08)



Tree Height Reduction (cont'd)

- Two Additions can be done in parallel.
- Need two adders and two cycles.

Goals:

- The goal is balancing the expression tree as much as possible.
- Tree-height reduction exploits some properties of the arithmetic operations.

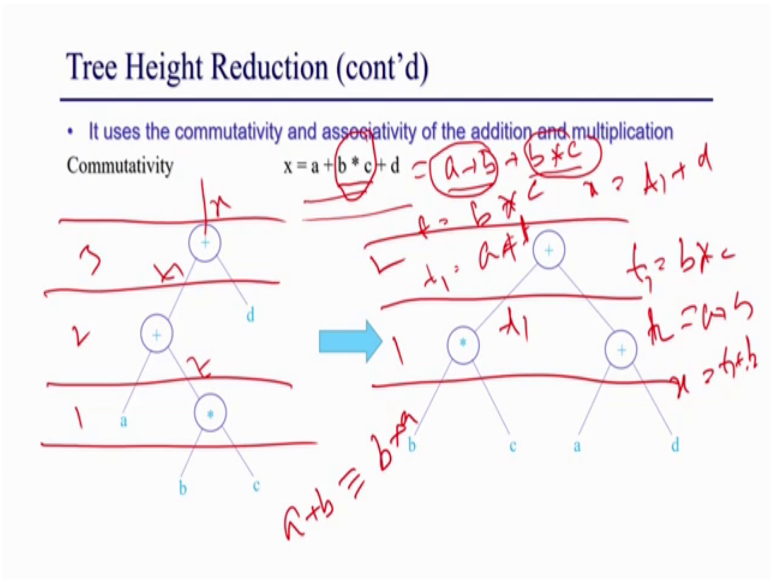
Impact: The height is proportional to a lower bound on the overall computation time

Because we can do that and then I just merge these results right. So, that is what is shown here. So, I just do a plus b here and then c plus d here, so, this is y and then I am doing x plus y, this is x equal to x plus y which is doing the same thing, right. So, this is doing the same thing, but now I need 2 cycle right. So, in the first cycle I can do this to in parallel and this is second cycle right. So, but I need 2 adders right, because now or 2 additional operation is happening in parallel. But suppose you have this to adder available, then the then probably this is a better choice. Because now this is taking one one cycle less than the other one, right.

So, the idea instead is like this tree height reduction is to balancing the expressions as much as possible. So, that we can execute operations in parallel, which is not dependent, right and also and what kind of impact it is having is like the height is proportional to the lower bound of the overall computation time. So, whatever the tree you are getting syntax tree, the height is basically it determines the number of times (Refer Time: 06:13) which is latency right so, or the overall or lower bound on the computation time you need. At least number of time steps to execute that behavior. So, if the height is too high; that means, your computation time or the number of time cycle times required or the latency is required to execute too high.

So, on the other hand, if you can paralyze you can do execute this in parallel, now you can actually have a serious impact on the number of time steps required right. So, this is something a technique which primary did you try reduce the computation time right, improve the computation time of this generator design right. And also in some cases, this is very simple operation, but in some cases this kind of reduction cannot be done directly. So, we might have to apply this arithmetic properties like say associativity, commutativity, or say distributivity, those kind of properties of the arithmetic operator's just to utilize this tree height reduction.

(Refer Slide Time: 07:19)


Tree Height Reduction (cont'd)

- It uses the commutativity and associativity of the addition and multiplication

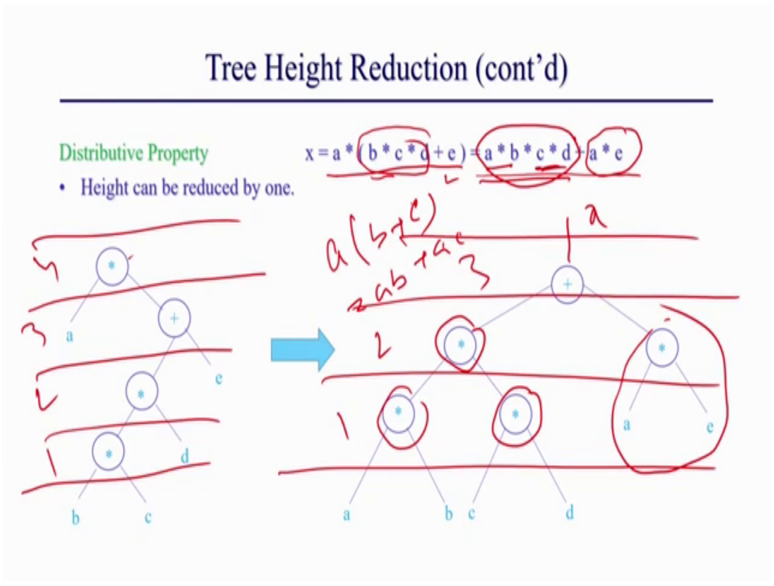
Commutativity $x = a + b * c + d$

So, here is some examples, say for example, you have this right a plus b this ah. So, you have in this case you have to execute this this first, because this has the higher presidence, then you can do this and then this right. So, what is happening here? So, b plus c is happening here, then you are doing a plus this say suppose this is say d, and then this is happening here. So, you are doing T equal to b star c, and then you are doing say T 1 equal to this is say T 1, equal to say a plus T a plus t. And then finally, x equal to x equal to T 1 plus d right. So, this is what is happening, but again in this case you can see I need 3 step right, I can do this. But if you are just try to directly do this height reduction on this, this is 1 2 3 do this it is not possible because it has the higher precedence you have to do it. But since you know this adder is a addition is a commutative operation right. So, basically a plus b is equivalent to b plus a o. So, this is there.

So, I can do I can rewrite this expression like this is basically equal to a plus b plus b star c. So, I can do this. So, if you do this now I can do this operations parallel that these 2 operations. So, what is happening here? So, I am I am doing this b star c here. So, I am doing T equal to b star c, say this is a T 1 and T 2 equal to a plus b I can do this, and this operation now in parallelism ok, right. This is what is happening here T 2 equal to say a plus b, and then x equal to T 1 plus T 2 right. So, then you can actually do this in 2-time steps. So, this is time step one, this is time step 2. So, this is something an example just to highlight this tree height, height reduction is not always possible it can be directly applied or because of the precedence of the operations right.

The multiplication has the higher precedence over additions and so on. So, we cannot apply them directly, but you can actually manipulate this expression using the this arithmetic operations property; like, commutativity, distributivity, or assosiativity, just to reduce that tree height reduction right.

(Refer Slide Time: 09:32)



Tree Height Reduction (cont'd)

Distributive Property

- Height can be reduced by one.

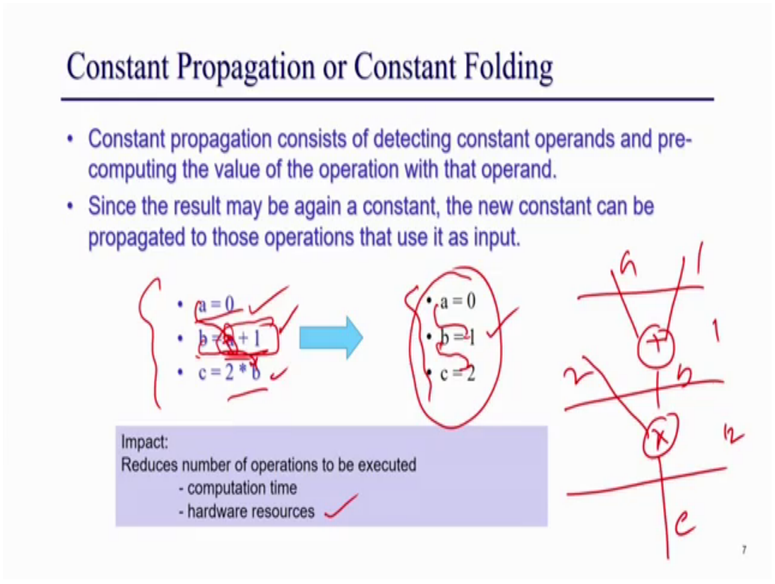
$x = a * (b * c * d + e) = a * b * c * d + a * e$

So, similarly here is another example for this distributivity. So, if you have this expression you have to do this first. So, you have to do this b star c. and then because this is a multiplication right. So, I have to do this and then this additional will happen. So, I am going to do this b star C then b star C into d, then I am doing addition. So, this is happening here and then I am finally, I am multiplying this with this multiplication right. So, I need basically 4-time step. So, I need 4-time step right, because I need basically for 1, 2, 3, 4.

On the other hand, if I just do the distributive property because multiplier is a distributor operation. So, I can multiply this I can get this expression right. So, then what is happening? So, basically distributive in say a into b plus means ab plus a c. So, this is what I am doing here. So, I just multiply a this is b star c, star b and a into e. Now I can do this a into e in parallel with this multiplication right. So, here also I can do this and this operation in parallel. So, I am doing a into b a into b here, and then c into d here, and then multiplying this whole thing to get a a into b into c into d. here I am doing a a into e in parallel, then I am doing the addition right.

So now you can see you can see I can do the whole thing 3-time step, right so, 1 2 3. So, again this height is reduced, but directly from that expression I am not able to reduce, but I can only reduce just up by applying this distributive of distributive operation right. So, this is just to this is one kind of technique which is called tree height reduction, and I just somebody we can actually just apply try to apply to paralyze these operations. And just to reduce the computation time right. So, the number of times step required to execute the behavior, it has a great impact on that, right. And another point is there just to reduce the height we have to apply this, arithmetic operations arithmetic properties like commutative distributive or associative operations just to make it reduce the height of the tree as much as possible.

(Refer Slide Time: 11:45)



Constant Propagation or Constant Folding

- Constant propagation consists of detecting constant operands and pre-computing the value of the operation with that operand.
- Since the result may be again a constant, the new constant can be propagated to those operations that use it as input.

a = 0
b = a + 1
c = 2 * b
a = 0
b = 1
c = 2
a
1
+
1
2
b
×
2
c

Impact:
Reduces number of operations to be executed
- computation time
- hardware resources

So, we will move on to the next kind of compiler optimizations which is called constant propagation or constant folding. So, as the name suggest it is basically if you have a constant sum, right some very, very basic will become a constant. So, you should apply all the operands of that variable with that constant value ok. It is a simple as that. So, for example, here suppose after certain kind of optimization your behavior become like this. So, equal to 0 so, a is 0, so, a is a constant now. So, you should apply wherever the a occurs you should apply this 0 directly, right so, for example, here if I just apply this a equal to 0 b will become 1, right.

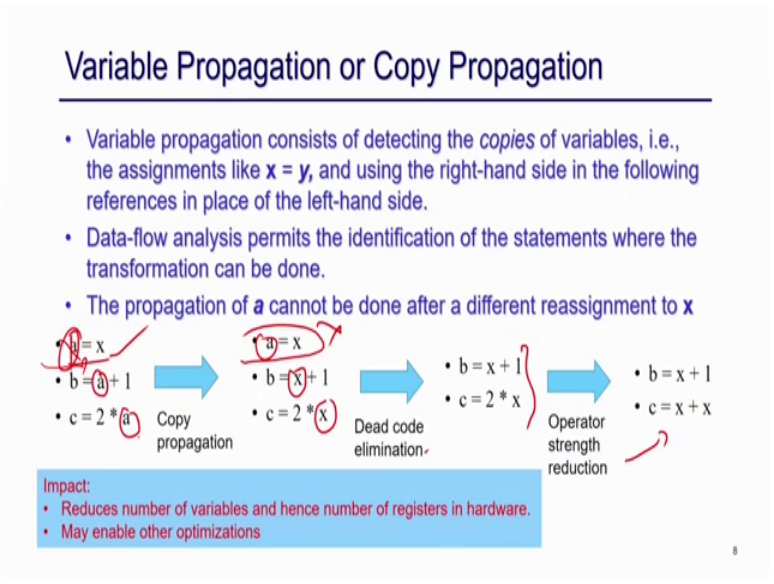
So, this is something is called constant propagation. So, I am propagating a to this location. So, you b also become a constant. And the important factor is that whenever you propagate constant sum other very well also become a constant might become a constant tight. For example, here although b is a expression b is a expression a plus 1, but because I am propagating a as a constant. This right-hand expression evaluator it becomes a constant. Right now, I can I should propagate b also right. So, I will propagate b to this expression. So, b equal to 1. So, this will become 2 into one it will be basically 2. So, all these things become constant right. So, this is what is called constant propagation or constant folding.

So, so, this is something we always do this because what is the advantage of having it. So, basically if you have this a plus 1 operation or 2 2-star b these are the operations this has to executed right in the hardware. But if you propagate some constant, and some expression becomes again constant you do not have to execute that operation right. So, it has an impact on the resource as well as in the computational time, right, because if we try to execute this operation. So, for example, this so, you have to execute b a a this is say a is coming. So, this is a plus 1. So, in this first time step this is become b, and then you have to do this multiplication, right.

So, you can you have you need at least this is 2. So, this will become C because see this operation depends on b. So, you need at least 2-time step to execute this right. So, on the other hand, if we just propagate this constant these all are kind of variable assignment I mean this is can be done just in one sided right. So, this is the advantage of having constant propagation. And it has impact both on in the resource as well as in computational time because I can improve the constant time by just not executing those operation because those are becoming constant. As well as I do not have to store these variables. Now right I do not have any some temporary variables or some intermediate result I do not have to store, or say I do not need an adder or a multiplier in the hardware.

So, it is basically having impact on the resource as well. So, it might reduce the number of registers in my reduce the number of adder multiplier, that was in somewhere in the hardware resources, multiplexes as well as and also computational time. Because I do not have to execute those operations in the hardware now right. So, this constant proportion and constant folding is kind of or a constant folding is, and very simple operation, but it has a some very significant or significant impact on the generator hardware.

(Refer Slide Time: 15:02)



Variable Propagation or Copy Propagation

- Variable propagation consists of detecting the *copies* of variables, i.e., the assignments like **x = *y*,** and using the right-hand side in the following references in place of the left-hand side.
- Data-flow analysis permits the identification of the statements where the transformation can be done.
- The propagation of ***a*** cannot be done after a different reassignment to **x**

Impact:
- Reduces number of variables and hence number of registers in hardware.
- May enable other optimizations

8

So, we will move on to the next kind of optimization which is called variable propagation or copy propagation.

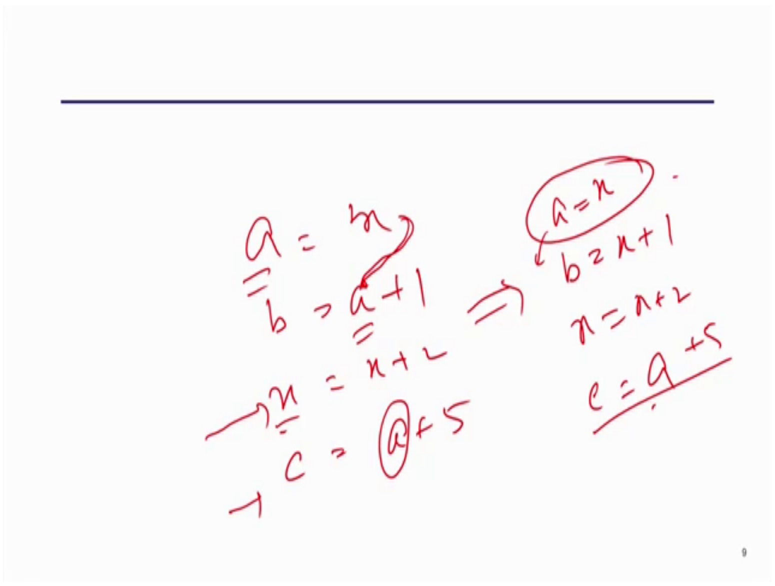
Just like constant propagation here this some variable is assignment operation, right, say a equal to b. This kind of operation then we can utilize either a or b, or say I can use only b right; I do not need a right because that is they are the same value, right. If a equal to b, then this what say for example, in this example a equal to x. So, x and a are basically the same way same value. So, in this case I can use only x I do not need a right. So, that is what is called value propagation a sorry variable propagation or copy propagation. Because whenever some copy operation is there assignment operation is there, you just take the right-hand variable. And you just replace the all the operands of the left-hand variable in the in the next operations. So, what is there here is also an example. So, a equal to x so, I can what I can do I can just eliminate b, I can use x wherever a occurs right.

So, I replace this a by x, this a by x, and I just ah. So now, this operation is basically become a dead code, right. Because I do not need this so, I can delete this. So, this is called dead code (Refer Time: 16:13). So, finally, this will become this right. So now, we can do some other operations is something operator strength reduction that I will discuss later. But essentially this is what is called value propagation proposition or copy propagation right. So, so, what is the impact definitely it has an impact on the register, because I am removing some of the variables a.

So, I do not need a register to store is this definitely have some impact on registered number of registers ok. And also, the important factor this is like once you have done this it might enable other optimizations, right. For example, here you enable dead code elimination. And in some scenario, it might also enable operator strength reduction or constant propagation or other kind of optimization. So, in general and we do not have to maintain 2 different kind of variable, 2 registers their interconnections, right, all those compress it will come. So, it has an impact directly on the number of resource we need right. So, this is something and important operation simple, but useful operation right.

So, also one point to be noted here is that we cannot just replace binaurally all occurrence of a, right, because if a is really fine here right. So, then you cannot just use it or another way if the different reassignment of x for example.

(Refer Slide Time: 17:37)

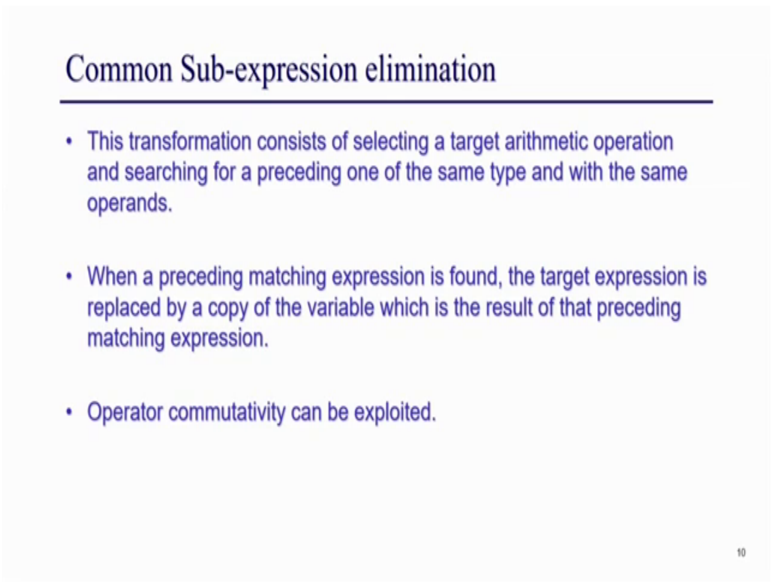


Suppose a equal to say x and then say b equal to a plus 1, then x equal to say x plus 2 then say again c equal to a plus 5. I cannot, I can replace this, right. I can replace this because now this x is same because the same value is happening here. So, I can just do a equal to x then b equal to x plus 1 this is a copy propagation. And since now this x become x plus 2 I cannot use here x, right. This is not allowed, right then because this x and this a is not same. Because whatever the value of x here is modified here. So, I cannot use this right.

So, I cannot just remove this operation this is not a dead code, but I can again I can always do this copy propagation. But I should avoid doing this replacing this a by x. So, it should be remain as a, right. So, this should remain as a, and I cannot delete this as a dead code right. So, this is what is this constant copy propagation, but you have to keep in mind that I cannot just replace all the occurrences of a. Because in sometime, that x might be reassigned or re redefined then that has a different value now. So, we cannot just use x for the subsequent operations.

So, those things can be just checked by data flow analysis, right. So, those comma or though are comfortable with this compiler. Then all those time (Refer Time: 19:05) are there they know how to doing this kind of check, right, which can be done which cannot be done right.

(Refer Slide Time: 19:16)



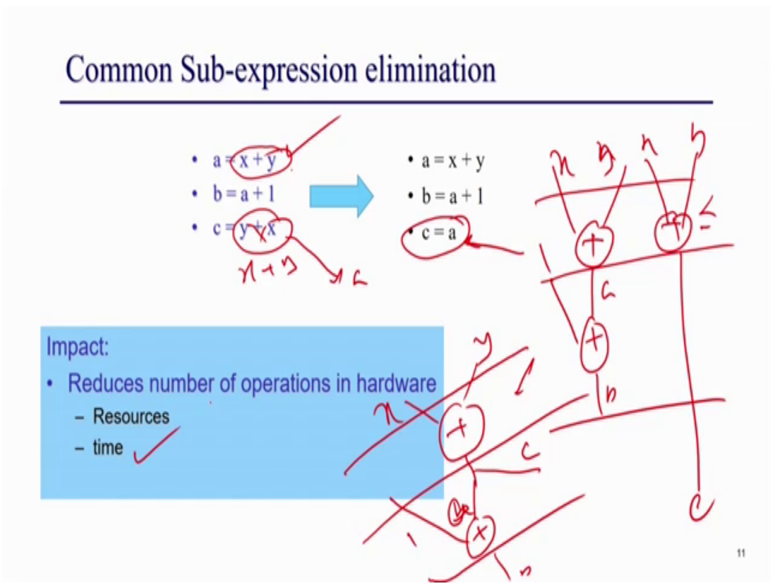
Common Sub-expression elimination

- This transformation consists of selecting a target arithmetic operation and searching for a preceding one of the same type and with the same operands.
- When a preceding matching expression is found, the target expression is replaced by a copy of the variable which is the result of that preceding matching expression.
- Operator commutativity can be exploited.

10

So, I will move on to the next set of operation optimization called common sub expression elimination. So, as the name suggest, and whenever there is a some sub expression which is same, right, common I have 2 operations where the right hand expression is same. I can use only I can execute that sub expression only once right. So, that is what is called common sub expression elimination. And I can use that particular value for the subsequent requirement, right.

(Refer Slide Time: 19:42)



Common Sub-expression elimination

- a = x + y
- b = a + 1
- c = y + x

- a = x + y
- b = a + 1
- c = a

Impact:

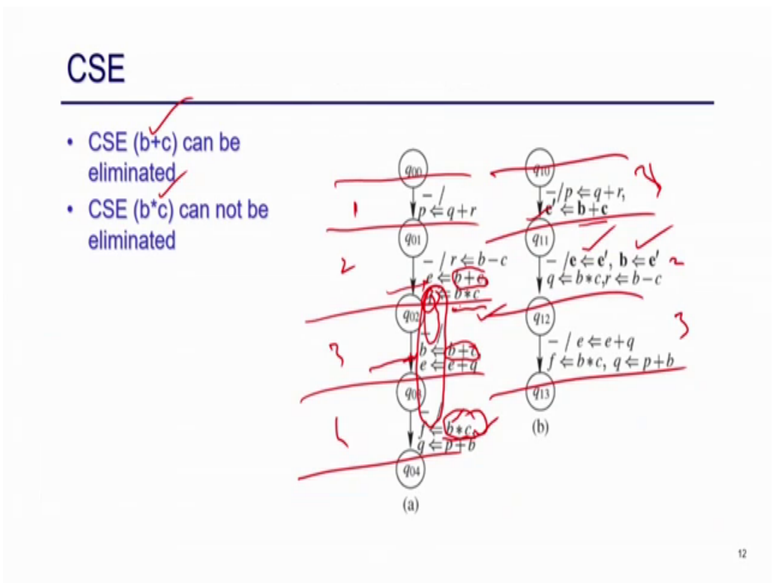
- Reduces number of operations in hardware
  - Resources
  - time

So, that is what is called common sub expression elimination. For example, here you can see I have x plus y, I have y plus x. But this is commutative operation, this is a single x plus y. So, these are the common sub expression. So, I can just only execute once and whenever I have this operands I can replace this by a, right. This is what is happening here. So, this is what is called common sub expression elimination. So, it is so, and what is the impact of it have in the hardware; obviously, can see that I do not have to execute this is 2 times so; obviously, it is saving resource as well as computational time, right, because here if you just try to do it it will take 3 cycle. And because or maybe in 2 cycle for example, here you are doing this say x plus y right. So, you are doing x plus y you get a here.

Then you are doing here plus 1 is your getting b and then say you are doing c plus. So, you we all although. So, this is you can do here again this is your C right. So, this is again the same x y right. So, x y, x y, x y, x y. But we need 2 cycle on the other hand, if you just do it I am not going to do this operation. So, again I know I do not need this resource. Instead what I am going to do I am going to do this once x and y, and I am going to assign this to both. A and c, right. And then in the next cycle, I can do this plus 1. So, although I am not reducing the number of time step here, but I am actually reusing this results to define the both the variable right.

So, effectively I am actually reducing the number of resource. At least for this example, but in some scenario we might see in that the number of common sub expression is remaining. So, that expression maybe in 5 places I do not have to execute in 5 times. So, it might also save your time. So, it can also impact in the number of computational time as well right. So, there is something is there.

(Refer Slide Time: 21:47)



CSE

- CSE (b+c) can be eliminated
- CSE (b*c) can not be eliminated

So, let s take an another example of common sub expression elimination here. So, you can see here, one common sub expression elimination expression is b plus c and b plus C is the common sub expression.

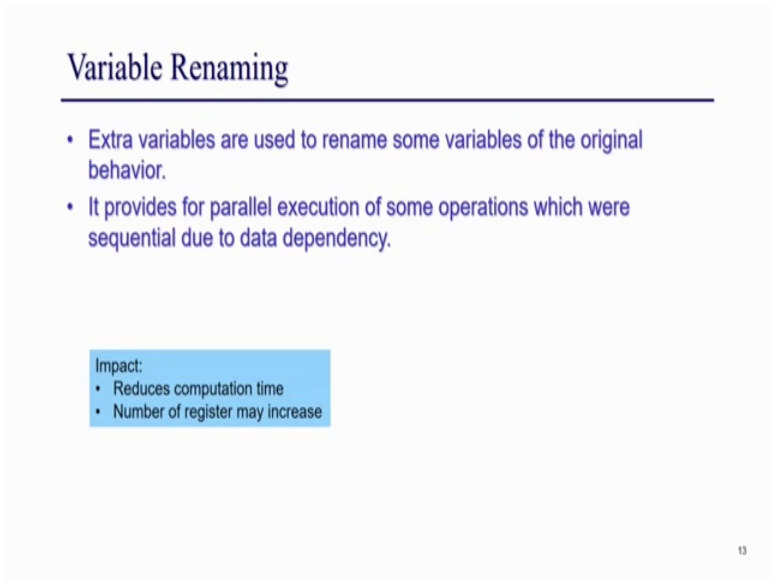
Similar I have another common sub expression is b star c, right there are 2 common sub expression b plus c as well as b star c. So, can we replace this b plus c? yes, because in this case b and c is not updated in between, right. If b and C is updated in between then this b plus c, and this b plus C is different right. So, but here in the in this portion of the code b plus b or c is not updated right. So, I can use I can store this b plus c into some temporary well e, e dash and I can use that e dash for both this expression and this expression right. So, this e and a e and b both can be defined using this right. So, this is what I am I have done right. So now, you can see here I can do this things in parallel because since I have dependency on this some other operations, in other cases, I have to I cannot execute all of them in one-time step, but now I can do even in one-time step; which is saving your own time step right. But for the case of b star c here b star c and this b star C is not same why because b ah. So, because in this portion of the code. Your b and C should not change, but here b is updating right.

Since b is updating so, this b and this b is not same right. So, I cannot replace this expression by this this value of e f q, right. So, this b star c cannot be eliminated. So, this is also again we have to keep in mind that we just cannot blindly just replace all those expression. We have to specifically do some data flow analysis just to find out whether those b and c are same in all the places then only we can do that. So, we can see in this example was doing simplification, here I need one this is one-time step f really this is another time step this is another time step. 4-time step right 1, 2, 3, 4 here I need 1, 2, 3-time steps right. So, I actually saving one time also, right one-time step also.

So, that is what I just. So, he your common sub expression can have impact on both resource as well as in time ok. But you have to always keep in mind that I cannot just replace all the sub expression without checking the data flow or data dependency among the operations. The basic bottom line is that between 2 operation, all the variable that occurs in the right, that sub expression should not be modified right they should not be redefined again between the 2 2 sub expressions right. So, in that case we cannot just replace. And this example just clarify that particular point.

So, I will now move on to the next set of operation optimization called variable renaming.

(Refer Slide Time: 24:45)



Variable Renaming

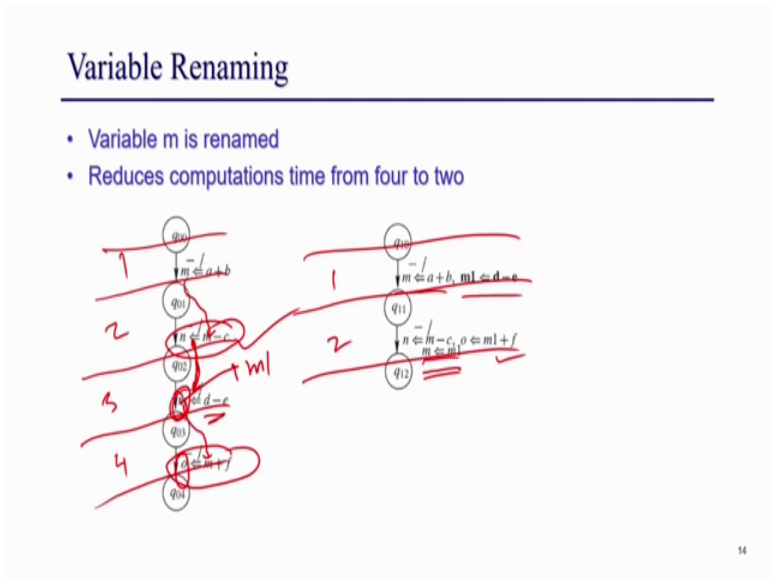
- Extra variables are used to rename some variables of the original behavior.
- It provides for parallel execution of some operations which were sequential due to data dependency.

Impact:
- Reduces computation time
- Number of register may increase

13

So, again this is a operation where you just rename some of the variable. But what is the impact of that? It might just reduce the dependency. Some sequential dependency is there among some operations you just rename some variable and that that so, that this dependency goes and we can execute this to in parallel ok.

(Refer Slide Time: 25:09)



Variable Renaming

- Variable m is renamed
- Reduces computations time from four to two

14

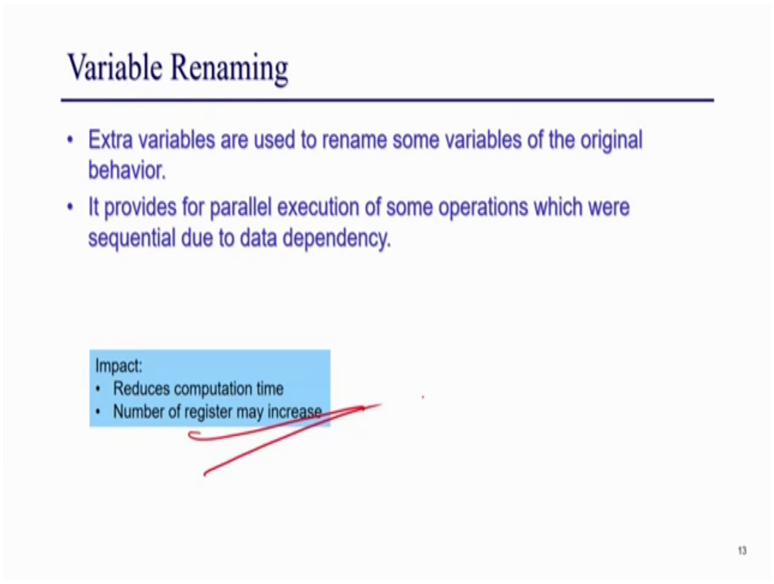
So, that is what is called variable renaming, you can see here. So, what is happening here? So, m is equal to a plus b, right, and then this m is using here. And again, is m is defined here, and is m is going to use here.

So, you can see here I cannot do all this operation parallel. Because this operation depends on this, unless this operation execute I cannot execute this, unless this execute, I cannot execute. So, I need 4-time step right. So, you can see here. So, this is time step 1, this is time step 2, this is time step 3, and this is time step 4. And I need 4 times to exect this behavior, because there is a dependency. But on the other hand, if I just redefine this variable, because this m does not depend on this instead of m I can just make m 1, right. How does it matter? Because this is not the final output right. So, if I assume that final output of this, or I can just assign this here ah. So, I can just redefine this variable. So, that I can just break this dependency right this right after it right.

So, this is right after it the dependency which I can break here. So, so that I can just execute this m equal to a plus b, and this m 1 equal to to d minus a parallel. Because this 2 do not depend to each other. And then in the next cycle, I can do this these assignment and this assignment, right. So, n equal to m minus c and o equal to m plus f. I can do this execute. And since my final value of m is this this value, I can just do m equal to m m 1, right. Because this is the my final value. But you can see was doing this variable renaming I just reduced this right after dependency as a result I can execute this operations in parallel, and I can execute the whole behavior into 2 cycle. So, this is variable definitely helps in improving the number of time step required to execute an behavior. So, hence the computation time.

On the other hand, I am actually defining some variable. So, I may need some extra registered to store that variable.

(Refer Slide Time: 27:01)



Variable Renaming

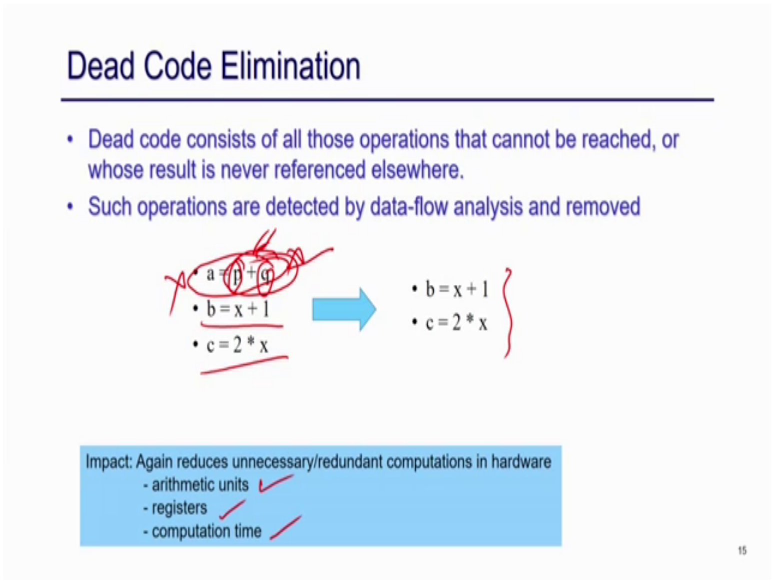
- Extra variables are used to rename some variables of the original behavior.
- It provides for parallel execution of some operations which were sequential due to data dependency.

Impact:
- Reduces computation time
- Number of register may increase

13

So, that may increase the number of registers right. So, this might have some little bit side effect, but, but it has if you are a lot of this write after read after sorry write after dependencies then if definitely this is going to break that particular dependency, and it can you can execute your behavior in less number of time step right. So, this is what is called variable renaming. Next set of optimization is dead code elimination. As the name suggest, if some code is dead; that means what that a value that is defined there is never used, right.

(Refer Slide Time: 27:32)



Dead Code Elimination

- Dead code consists of all those operations that cannot be reached, or whose result is never referenced elsewhere.
- Such operations are detected by data-flow analysis and removed

- a = p + q
- b = x + 1
- c = 2 * x

$\rightarrow$

- b = x + 1
- c = 2 * x

Impact: Again reduces unnecessary/redundant computations in hardware
- arithmetic units
- registers
- computation time

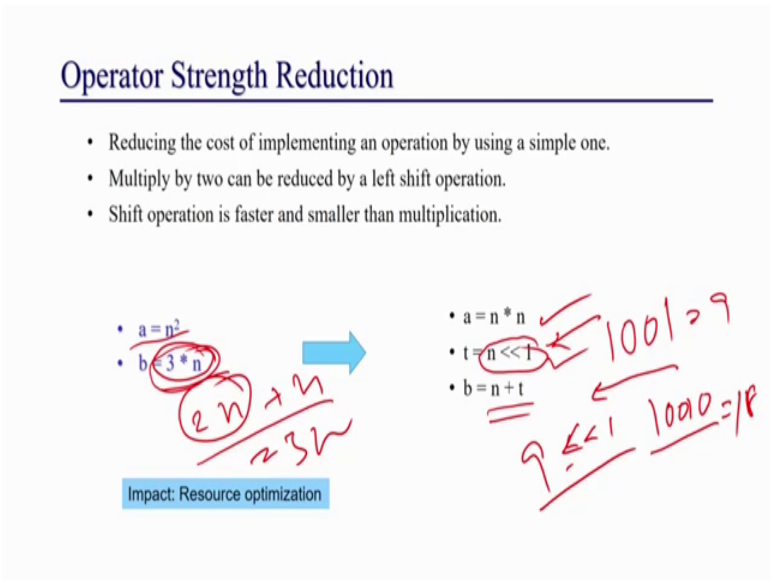
15

So, whatever the definition you are actually defining some expression is basically dead right it does not have any use in your behavior. So, in that case you can actually remove that, simply remove the particular dead code, right, because it does not have any impact in the generated output right. So, for example, here you can see a equal to p plus q, and then you have b equal to x plus 1 and C equal to 2 x 2 into x this does not have any impact here right. So, I can simply delete this dead code, and when final operation will become b into b equal to x plus 1 plus and C equal to 2 into x.

So, what kind of impact it might have this dead code elimination; obviously, I do not have to execute this p plus q. So, it will reduce the number of arithmetic units, because I do not need to execute that. And also, I do not I do not have to execute this. So, I need I may step some computational time, because I do not have to need any another resource to execute that. So, that might increase the number of time step required to execute. Similarly, whatever the express variable that it is in the right-hand side they may also be didn't that they do not may have some use in other places. So, if I just reduce it delete this eliminate this dead code, I do not have to store p and q as well right. So, this may also reduce the number of registers.

So, dead code elimination in general impact in all precise resource in the sense number of arithmetic units number of registers as well as in the computational time right. So, this is; what is dead code elimination.

(Refer Slide Time: 29:18)



Operator Strength Reduction

- Reducing the cost of implementing an operation by using a simple one.
- Multiply by two can be reduced by a left shift operation.
- Shift operation is faster and smaller than multiplication.

- $a = n^2$
- $b = 3 * n$

Impact: Resource optimization

- $a = n * n$
- $t = n << 1$
- $b = n + t$

2n + n
= 3n
1001 = 9
9 << 1
1000 = 18

So, next optimization is called operator strength reduction. So, what does it mean? So, there will be some operation; which is complex right so, as multiplier. So, multiplied by constant right; so, this kind of operations can be replaced by a simple shift operation. So, I actually doing the same of same operation. But I am doing using a less complex operation right. So, I am just reducing the strength of an operator or say I have x square. So, I need a multiplier x into x, but instead of doing x into x I can just do x plus x 1, sorry, 2 into sorry this case I cannot do, but what I can do sorry this constant multiplication is a plus. So, I can do it so, 3 into n.

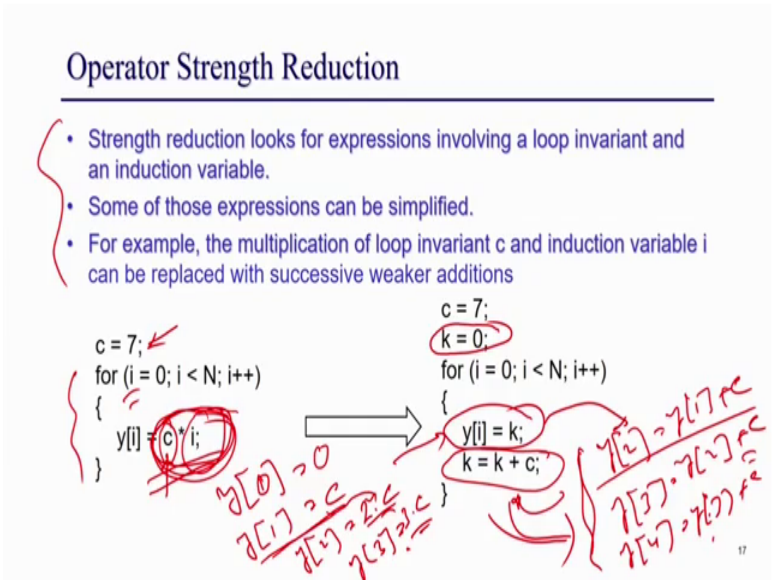
So, there what we can do; so, here is an example. So, I have say n into n square so, I can do n into n. So, I can then, what I can do this I have 3 into n right. So, I can do 2 into n then plus n; which is basically 3 n, all right. So, whenever you have 2 n, I can just do the left shift right. So, whenever your are shifting some of by value of say if I have say 5, right, 1 0 1 1 0 0 1 is what is 9 ok. This is 9 if we just do it left shift 9 left shift by 1 what will happen? So, we are shifting the whole thing by one bit so, 1 0 0 1 0 so, what is that? So, this is basically 18.

So, basically when you are shifting some valuable it is just doubling the value. So, I just make a shift. So, I will get 2 n then I can just do the addition right. So, instead of doing a using a multiplier, I am using a less costly operation shift. Shift is just you do not need a basically any operator, right you have to just do the bit manipulation you have to just shift the whole thing by one bit you do not need any adder multiplier, or any kind of the resource, just to do this. You just shift and you reassign the value to this bits, right. You have to just do a 1 equal to a 0, a 2 equal to a 1, a 3 equal to a 2 and so on, right. You have to just do that it will automatically do the shifting, right. So, this is something is done. So, I just break this 3 in 2 into plus, and there do this by shifting, and then I just do make a addition. So, multiplier can be replaced by a shifting an additional operation right.

So, essentially you can see it has a impact on the resources optimization, right in the resource, right because a multiplier when you are going to implement in a hardware it is a complex mix circuit. On the other hand, if the shift is nothing just a bit manipulation bit assignment. I need just a simple adder right. So, the adder is much simpler (Refer Time: 31:01) and a multiplier. So, this operator strength reduction is useful just to reduce their resource of your design, right.

So, and some other example is this is one kind of thing while you can actually replace this multiplier by shift and add kind of operation.

(Refer Slide Time: 32:20)



Operator Strength Reduction

- Strength reduction looks for expressions involving a loop invariant and an induction variable.
- Some of those expressions can be simplified.
- For example, the multiplication of loop invariant c and induction variable i can be replaced with successive weaker additions

```
c = 7;
for (i = 0; i < N; i++)
{
    y[i] = c*i;
}
```

```
c = 7;
k = 0;
for (i = 0; i < N; i++)
{
    y[i] = k;
    k = k + c;
}
```

So, on the other hand, if you have some operations which is basically inside the loop and which depends on that inductive variable i. So, i is the inductive variable, which is basically i (Refer Time: 32:30). And some variable which is basically invariant to the loop. C is not changing in the loop right, so, this is a loop invariant. So, if some expressions are there which is depend on the inductive variable as well as some loop invariant, I can replace those kind of expression by a simpler operation right. So, what is happening here you can see here so, C is initially 7.

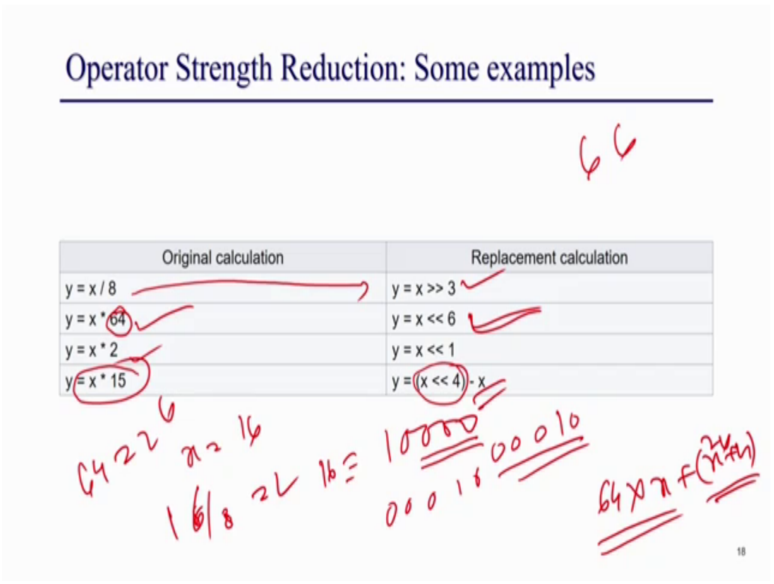
What I am doing here y 1 equal to basically, y 1 equal to what sorry y 0 equal to i equal to 0. So, this is 0 so, y 1 is what? This is basically c, right now this is this I equal to 1. So, one into c is C, then y 2 is what 2 into C, right y 3 is what? 3 into c and so on. So, this is what is happening. So, I now we can see here this successive expressions instead of doing this multiplication, I can compute this 2 C from this only right. So, I can do just do this whatever the value here y 1 this I can do y 2 equal to y 1 plus c, right.

Similarly, y 3 what I can do? This is 3 C this is 2 C plus C. So, I can do y 2 plus c, right. So, I can just to this way; so, y 4 equal to y 3 plus c. So, instead of doing this multiplication, I can just do this right I can do the same thing by an by an addition operation. So, this is what I am doing here. So, I have initially k equal to 0 because this is

C is 0, and then this is y equal to k, and I am just calculating this directly I can do this or I can just calculate this because my k is that the previous value. And then I just add C, and then I am assigning these 2 here right. So, if it is affecting you bring this right this is affecting into this.

So, what I am doing here is instead I replace a costly operation or a complex operation multiplier by a addition operator. And this is applicable to this this inside the loop, right. this specifically this kind of strength reduction looks for expression involving a loop invariant and a inductive variable, right. And just some of these cases this can be replaced by a with a addition operation ok. So, that is what is happening in this particular cases. We might have some other cases of operators strength reduction.

(Refer Slide Time: 35:02)



Operator Strength Reduction: Some examples

Original calculation	Replacement calculation
y = x / 8	y = x >> 3
y = x * 64	y = x << 6
y = x * 2	y = x << 1
y = x * 15	y = (x << 4) - x

66
64 = 2^6
x = 16
16/8 = 2
16 = 10000
00010
64 × x + (x...)
18

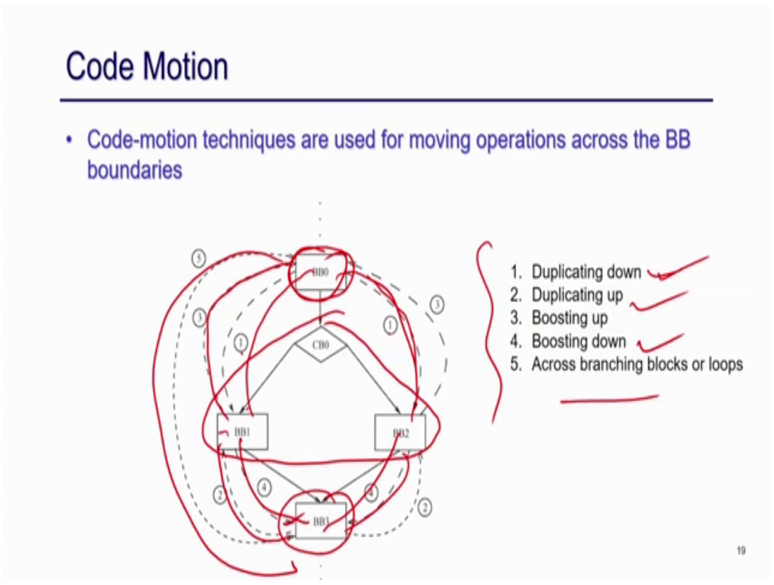
Like say if you have say division operation right. So, division is basically right shift.

So, for example, say suppose your x equal to say 16 right. So, you are doing x by 8, 16 by 8, this will be 2 right. So, this 16 means what? This is 16, right this is 16 and 2 means what right. So now, if you just do this 3 writes into this so, this 3 will go. So, this 1 0 and 3 extra 3 0 which is basically 2 right. So, this division by some number which is basically a power of 2 can be replaced by just simply right shift operation, right, similarly what I just seen that multiplication can be replaced by a left shift operation.

So, this and this factor can be this multiplication with power of 2 right, 64 is basically 2 to the power 6. So, I can just replace with a left shift of 6, right. So, but as I seen in this example if it is not multiplication of this then I can find the nearest a power of 2 and plus some addition operation, right. For example, suppose you have in this case is 6 is of 66 you can do, 2 64 into x plus x into x plus x into x right or maybe 2 into x. Then I can do this 6 times. And then I also I can shift by one bit and then I can just add it ok. So, this way I can do also this is also shown here. Or if you have say 15, then I can do this is what I am saying I can do the 16 the nearest power which is 16, and then I can do this 1 minus. Because this becomes 16 x, and then I just make a minus x. So, that it becomes (Refer Time: 36:49).

So, the idea is that whenever you have this division or multiplication operation. I can always replace them by and shift an addition operation, right. And idea is that if it is exactly multiplier of 2 sorry power of 2, then I can directly do this thing if it is not then I can find the nearest power of 2. And then I can do either addition or subtraction based on the value we obtained right after shifting. And also, we can do some operator strength reduction by eliminating this inductive variable, and this loop invariant variable right in some in case of loop.

(Refer Slide Time: 37:28)



Code Motion

- Code-motion techniques are used for moving operations across the BB boundaries

1. Duplicating down
2. Duplicating up
3. Boosting up
4. Boosting down
5. Across branching blocks or loops

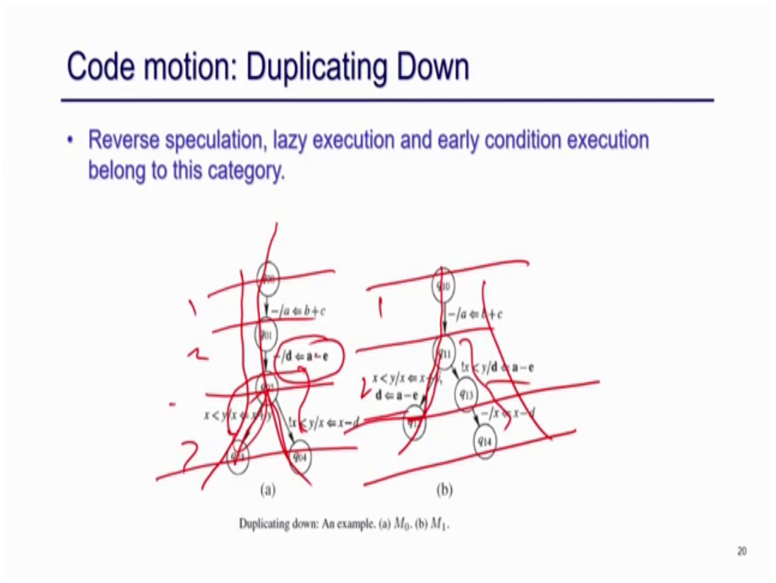
So, these are this operator strength reduction, and this usually have an effect in generating hardware in terms of replacing a complex big operation by a simpler big operation. Right. But affecting doing the same functionality you are accepting the same functionality in the design, right. This is what is called operator strength reduction. So, I will now move on to the next set of optimization which is code motion and it has very good impact on the generator hardware.

What is the code motion? As the name suggest, it is basically moving the code right moving the code in your behavior. And your movement can be anywhere right. So, you can be before loop to after loop from the loop to outside or from the outside to inside the loop or maybe there is an offence block, and you are moving some operation before loop before. The offense to inside the (Refer Time: 38:19) of from e fells to outside, right both is possible. So, all kind of possible possibility is there, and you based on that we actually can have a different kind of code motion is equal to duplicating down basically you suppose I just consider a if else here we can have also a loop right. So, this is your if else block right, this is your if else block in this.

So, you are moving some operation from before the loop, before the; if else block to inside the ff block this is 1 2 right. So, this is one. So, this is duplicating down. So, duplicating up is I mean whatever we. So, this is your effectively this is your if else block sorry this is your if else block ah. So, this is your if else block, right, this is if else block and if some code after that you can move inside the if else this is called duplicating up.

Boosting up is what? Boosting up it is basically from the loop to outside, right. Or boosting down is from the if else to after the loop. This is called boosting down. Or it can have a across right something here, you can move it after the; if else or this may be loop you can do that right. So, you can do across the loop as well. So, this again various kind of code motion and it might have some good impact ok.

(Refer Slide Time: 39:36)



Code motion: Duplicating Down

- Reverse speculation, lazy execution and early condition execution belong to this category.

Duplicating down: An example. (a) M_0. (b) M_1.

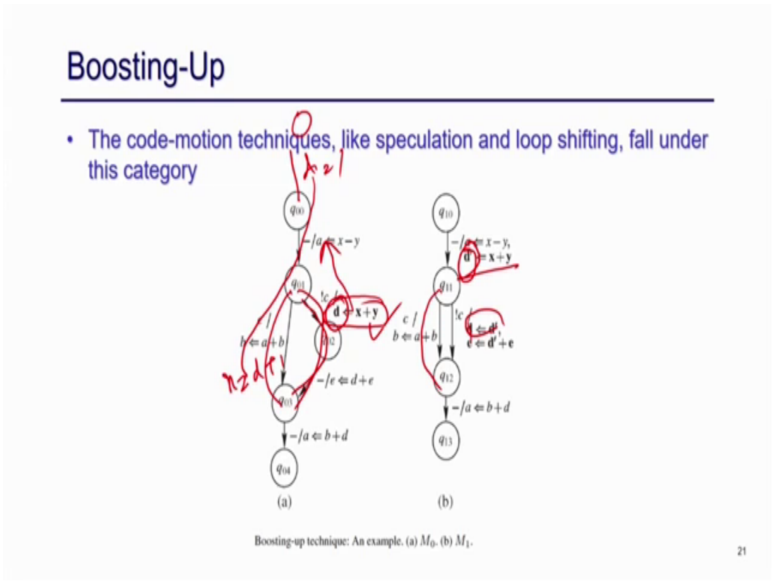
20

So, here is an example. So, duplicating down so, here you can see there is a operation here which is before the if else, right. What I am doing here? I am just moving this duplicating in both the path, right. What is the impact of that? You can see I just moved this operation to this branch and this branch also.

So, I also so, this is here and this is here. And what is the impact? Because this d is not using in this branch. So, I can do this operation, and this operation in parallelism, right? On the other hand, since this d is using here so, I need 2 cycle here. So, advantage is here is that if the behavior is executing through this branch, then what is happening? So, I need 1 cycle, 2 cycle and 3 cycle, right. So, I need 3 cycle to execute the way we are. But here you can see this is happening through this branch I need one cycle and another cycle so, one cycle.

So, I am saving one cycle in conditional execution, but whenever the execution happening through this I need still 3 cycle, right. Because I have to do this in this. But at least if because in your behavior this if else might execute multiple times. So, in some time you are actually saving one one clock, right, that is what is the advantage of having duplicate down right. So, because in some scenario that particular operation may not depend is not using some particular branch of a if else and there you can do that a operation in parallel. So, that wills give you a saving here. So, as I have shown here, I have one times of saving in the if branch, right.

(Refer Slide Time: 41:07)


Boosting-Up

- The code-motion techniques, like speculation and loop shifting, fall under this category

Boosting-up technique: An example. (a) M_0. (b) M_1.

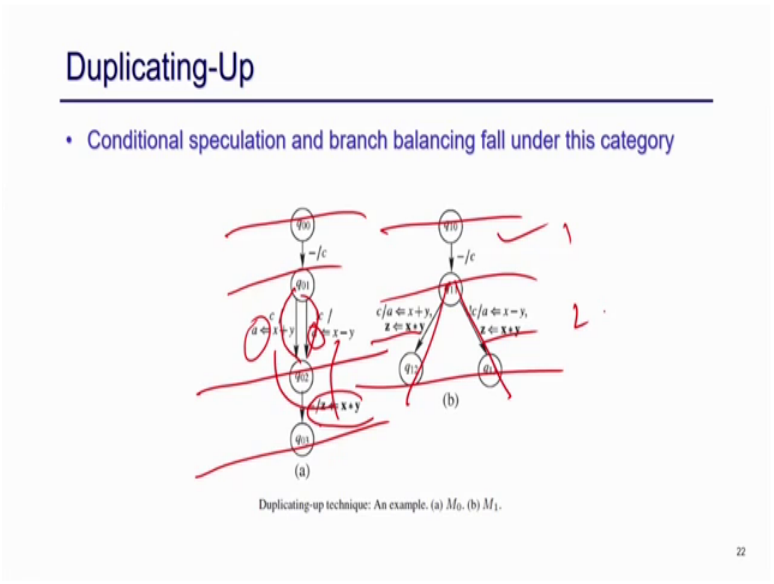
21

Boosting up giving me some example of them boosting up is something, suppose you have an operation here d equal to x plus y I want to move it to before the if else. So, this is your; if else block, right. And if you just try to do it because this variable is used only here right. So, if we just put d equal to x plus y here that d might be used here right. So, there may be some operation here which is basically say x equal to d. So, then this d plus 1 say so, this d is not this right there maybe some other d defined here d equal to 1. So, I am going to use this d here, but if you move directly this operation here then this value of x will come here, which is wrong, right.

So, that is whenever you do the boosting operation you need to store that is a temporary variable, right. What I am doing here this d equal to x plus. So, I am boosting of as to do it if you put the if else, but I am storing this into d dash, and I am going to use this d dash here, right, and this d dash have no impact here again. You can see here that this particular operation I am boosting of and it might impact though if this is not visible in this example that maybe I am actually saving, some time step because this particular yeah. Actually, I am doing this because I need 2 cycle here. In this case, you can see here in the if this these branch right in this branch I need 2 cycle in one cycle I have to do this, in the next cycle have to do this. Right. But in after scheduling after boosting up I can do this operation in one cycle. So, if there is an execution through this path after boosting up, I am saving one-time step, right.

So, this boosting up also again you can actually you can improve your computational time by just putting some operation before the if else block from one branch. But you have to keep in mind that when you are moving that you cannot just use the same d because said, that me a way that d might have some different use in the other branch; which might be affected by this d.

(Refer Slide Time: 43:13)



Duplicating-Up

- Conditional speculation and branch balancing fall under this category

q_{00}
$-/c$
q_{01}
$\bar{c}/a \Leftarrow x+y$
$c/a \Leftarrow x-y$
q_{02}
$-/z \Leftarrow x+y$
q_{03}
(a)
q_{10}
$-/c$
q_{11}
$c/a \Leftarrow x+y, z \Leftarrow x+y$
$\bar{c}/a \Leftarrow x-y, z \Leftarrow x+y$
q_{12}
q_{13}
(b)
1
2

Duplicating-up technique: An example. (a) M_0. (b) M_1.

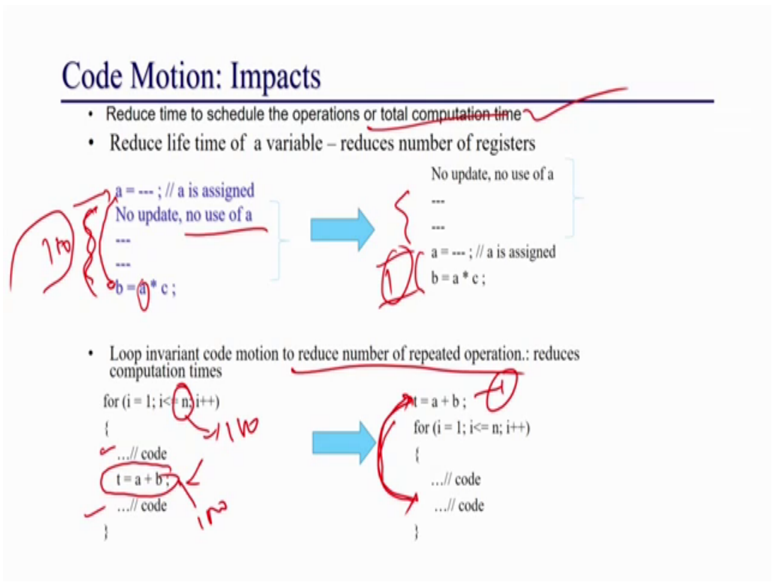
22

So, I have to stored in some temporary variable, right. This is what is the idea of boosting up just 2 example give example of duplicating up here there is an operation after the if else. So, this is your if else and I am just duplicating up both of them, right.

So, what I am doing here I am just moving this to this branch and this branch. Here you can see this operation is not depend on this values this a is computing here. So, I can do this domain parallel. So, earlier how many time step was required one, for any branch I need one branch to do the if or else, and this 3 there are 3-time step now I need only 2, right. One-time step for this one-time step for either going doing this or this right. So, I have need only 2-time step instead of 3. Again this duplicating down can improve you computation time right, duplicating out sorry. So, and this way we can actually give other kind of a example, but this is just to highlight that this kind of duplicating down or say boosting up or say duplicating up is kind of examples, have an impact in the directly impact on the computation time, but in conditional branches, right.

It is a not directly if you if you just so, saw binaurally you may not see the effect. But if your behavior execute through a particular branch then your computation time might improve right.

(Refer Slide Time: 44:30)



Code Motion: Impacts

- Reduce time to schedule the operations or total computation time
- Reduce life time of a variable – reduces number of registers

```
a = --- ; // a is assigned
No update, no use of a
---
---
b = a * c ;
```

```
No update, no use of a
---
---
a = --- ; // a is assigned
b = a * c ;
```

- Loop invariant code motion to reduce number of repeated operation.: reduces computation times

```
for (i = 1; i<= n; i++)
{
  ...// code
  t = a + b;
  ...// code
}
```

```
t = a + b ;
for (i = 1; i<= n; i++)
{
  ...// code
  ...// code
}
```

So, that discuss that the reduce the schedule of operation or total computational time definitely, right. There is another big impact of code motion is something it reduce the lifetime of a variable right. So, what I am doing I am moving actually I am moving an operation from one place to another place, right. The idea is that if some operation defined here right very early. And it is if I use very late. So, that particular definition has to store in some register for a long period of time. Because unless this is going to use here, I cannot remove that particular value from the register, right. Because then that value is required here so, defined and used.

So, I have to store that variable for that long, but if I move the definition and this use closed by, then I do not have to store it for long time, right. This is just to give you an example, suppose if I define a here and there is a long sequence of code. So, there is say a 100 line of code where I do not use a at all when I am not updating a at all, and then I am using it here right. So, that means, I have to store this a for say this is say 100 cycle for 100 cycle. But in code motion suppose I just moved this operation, and to near to this right. So, this operations where this there is no use of a or there is a there has got a definition of a here right. So, that a is this part of operation in independent of a. So, what I can do I just moved this operation near to this definition.

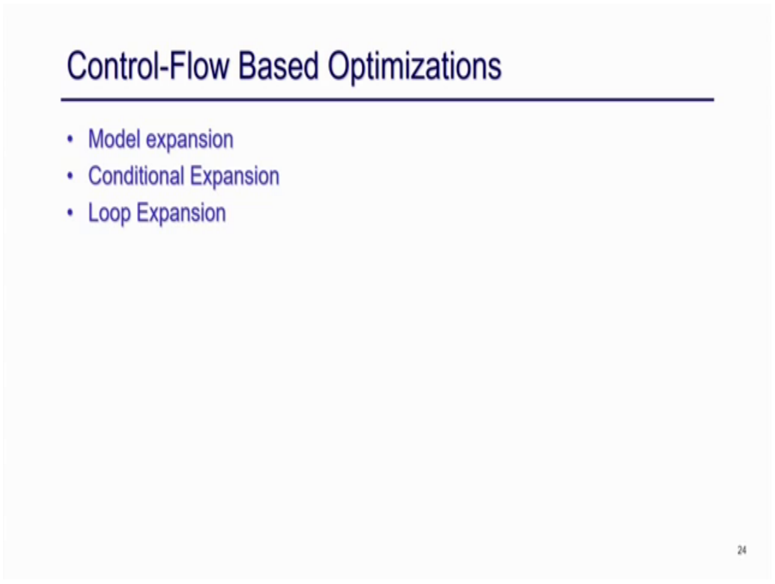
So now I I so, maybe I have to just in one cycle, I can just define I can use it. So, instead of storing this for 100 cycle I can just store the variable a for one cycle, right. So, that means, this code motion infecting this reducing the lifetime of a variable. And effectively, this is reduce the number of register as we have seen that we can stroke 2 register if their lifetime is not overlapping, right, that we have already discussed during high level synthesis discussion. So, so, we can actually reduce the number of register that is going to be required to store all the variables, right.

That is another impact of code motion. The third impact of code motion is something is the number of operations also right the repeated operation also then reduced. For example, here suppose I have a loop in various operation e equal to a plus b. Because a and b is not updating here right so, this is a constant operation. So, instead of doing this per n times, I can do it only ones, right, and I can use this T insect, right. And then I do not have to do it for say this n is a 100, then I am not doing this 100 times I am doing only one times. So, this is also saving computation time, right, because I do not have to do this operation for multiple number of times I am just doing only ones.

So, this is also have a good saving because if you if you just have a utility iteration implementation of this loop, and effectively say be 100 number of cycles, right, because I do not have to do it 100 times. If this operation particularly inside the loop then it is going to execute n number of times or 100 number of times, but if it is outside the loop it is going to be execute only once right. So, so, code motion is very useful technique and there are lot of work actually happening recent time just to just to show that what are kind of impact, right in of code motion in generator hardware. And it has impact on number of computation time number of operation to be executed and also as well as the lifetime of a variable, and hence the number registered required equal to store this variable.

Those was interested you can go into this kind of technique like reverse speculation you can search internet, lazy execution and early condition execution; like, speculation loop shifting right, or say branch balancing or conditional speculation. So, there are a lot of specific type of this code motion, I am not going to detail of each of them because this is can be whole new topic the code motion different kind of code motion. And there are a lot of work actually going on they are actually showing the impact of say conditional speculation in generator hardware or say branch balancing in generated hardware through high level synthesis. Or say, speculation loops shifting early condition execution say reverse speculation. So, those are specific type of for code motions and there are lot of studies happening on that. So, those interested you can search for those kind of work right.

(Refer Slide Time: 48:54)



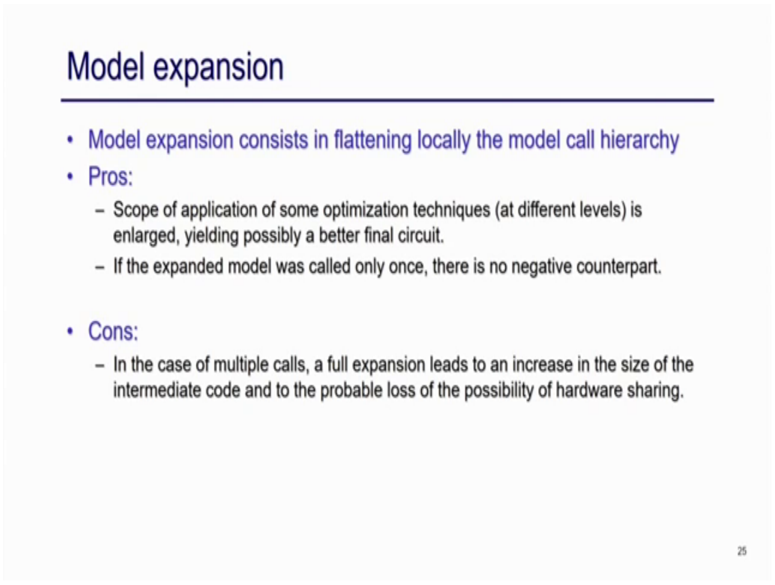
Control-Flow Based Optimizations

- Model expansion
- Conditional Expansion
- Loop Expansion

24

So, now I am going to talk about this control flow-based optimizations, and here I am going to talk about specifically 3 kind of technique module expansion conditional expansion and loop expansion.

(Refer Slide Time: 49:05)



Model expansion

- Model expansion consists in flattening locally the model call hierarchy
- Pros:
  - Scope of application of some optimization techniques (at different levels) is enlarged, yielding possibly a better final circuit.
  - If the expanded model was called only once, there is no negative counterpart.
- Cons:
  - In the case of multiple calls, a full expansion leads to an increase in the size of the intermediate code and to the probable loss of the possibility of hardware sharing.

25

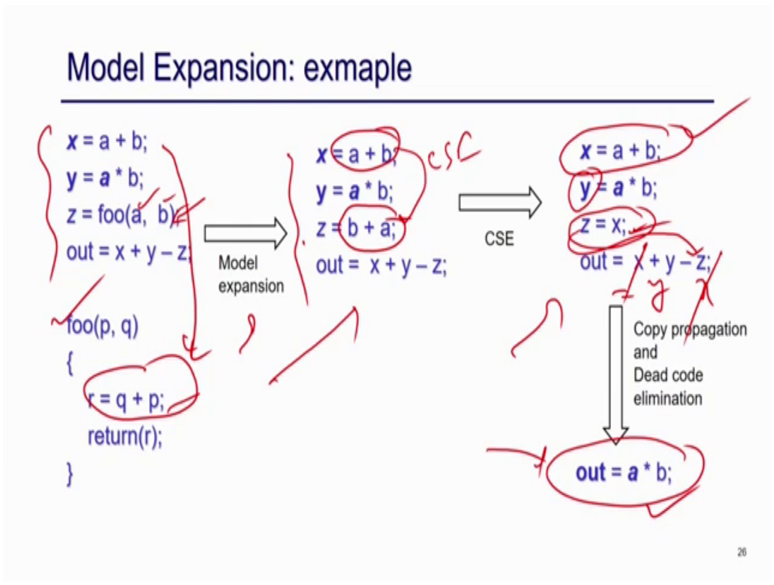
So, what is that module expansion? It is basically just replacing the function body by the body right. So, if you remember when I am talking about this code this coding style. So, what I have discuss that whenever there is a function it will create a module in the hardware, right. And it that function is going to call multiple places, I am going to reuse that particular module if they are non-overlapping, right. If they are schedule is non-overlapping. So, that is how this function is going to synthesized. But in certain scenario there is a small small function is only called once, right they are not effectively calling multiple times.

So, then instead of making a function or module for a function, I can replace that call function called directly by that function body, not will be the advantage because whenever you have module you have to keep telling me some control; that means, a extra control signal. So, that will also other complexities come into that just maintaining, right, and it just have only one use. So, there is no point maintaining or doing all those extra things just to maintain that module for one call, right. What you can do you just replace that that call by the function body, the advantage of that is that now I solve the optimization that we talked about so far can be applied on the whole code right.

Whenever there is a module what about the optimization that will be local to that module, but and whatever the optimization is happening in the main function that is local to that. So, what about the optimization? That does not go into this module to optimize something because that is a different module right that might have some other use somewhere. So, I cannot do this context specific optimization. But if I replace that function call by that function body in that place, now I have more scope to apply optimization in the whole setup code right. So, that is a big advantage that will actually give you some benefits on in terms of resource number of devices. But the bad side of this particular or say negative side of this particular optimization is that if the particular function in multiple calls then I should not do it, right.

Then basically if I just do increase the code size, right, instead of one module I will if I there are say 10 calls I have 10 body to be replaced and if the body is weak then it increase the code size, and hence the number of operation to be executed, right. Because if it is between a module and it is only one hardware you got for that or I need 10 copy of the hardware if I just replace the function body right. So, that is something the negative side of this and you should do this kind of module expansion only if we have only one or 2 call of a function right in this.

(Refer Slide Time: 51:41)



Model Expansion: exmaple

```
x = a + b;
y = a * b;
z = foo(a, b);
out = x + y – z;

foo(p, q)
{
  r = q + p;
  return(r);
}
```

Model expansion

```
x = a + b;
y = a * b;
z = b + a;
out = x + y – z;
```

CSE

```
x = a + b;
y = a * b;
z = x;
out = x + y – z;
```

Copy propagation and Dead code elimination

```
out = a * b;
```

26

So, here is an example suppose I have this is my main code, and I have a function calling here. And this is doing nothing just doing say p plus q, it is q plus p .

So now I can replace this function call by this function body, right. If I just do it, since I call through a b, I can replace this q plus b by a plus b right. So, this if I do the model expansion my code will become this. Now by applying this common sub expression this common sub expression I I can replace this 2-common sub expression I can replace them right. So, I can do this x equal to a plus b and I can just do z equal to x so, that is actually improving. So, in this case I cannot apply common sub expression elimination, because this tend to different body and these 2-different expression I cannot do that, but once I apply this model expansion now, it increase the scope to apply the other optimization like common sub expression elimination in this case.

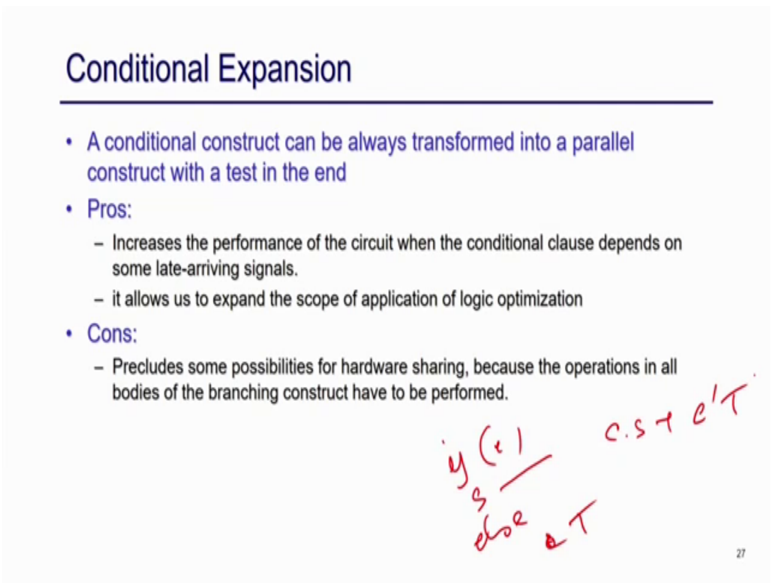
I can replace this common sub expression by a single a, right. That is what I have done. And I am not done yet I can have some other optimization also. So now, this is become a copy operation I can do the copy propagation, right. So, I can just do this do here. So, I can just do it x here. So, once I do this x equal to f then I can replace this, right, then this out will become out become only y, right. Then once I do this x equal to a plus become dead code and because that have no use. So, effectively this become out equal to a star b. Again, I can replace this y because I do not need to store that because it is happening.

So, effectively this whole code is doing a out equal to a star b. So, this is just to highlight an example just to highlight that if you do the model expansion, that will increase the scope of application of the other optimizations, and that might reduce the code by a significant amount. So, that is the advantage of model expansion, but you have to always keep in mind that if the particular function says by a multiple pluses, then it is not a good idea to replace all the all the call by a body because it will you have to execute this operation those many times. So, there should be some tradeoff between this model expansion where whether we should do or not even in certain scenarios.

Or maybe it if that particular function has a say 10 calls. So, we might just replace one or 2 calls because that might if we see there is a very high chance of have other optimization possible of or some other optimization in certain places we can actually replace those calls. But I can keep other calls there so that I can use that function body for other calls, right. I can reuse that particular function body or the hardware corresponding into the function body for those other calls ok.

So, there might be trade off and that is has to be analyzed based on your application or the example that you have right.

(Refer Slide Time: 54:33)



Conditional Expansion

- A conditional construct can be always transformed into a parallel construct with a test in the end
- Pros:
  - Increases the performance of the circuit when the conditional clause depends on some late-arriving signals.
  - it allows us to expand the scope of application of logic optimization
- Cons:
  - Precludes some possibilities for hardware sharing, because the operations in all bodies of the branching construct have to be performed.

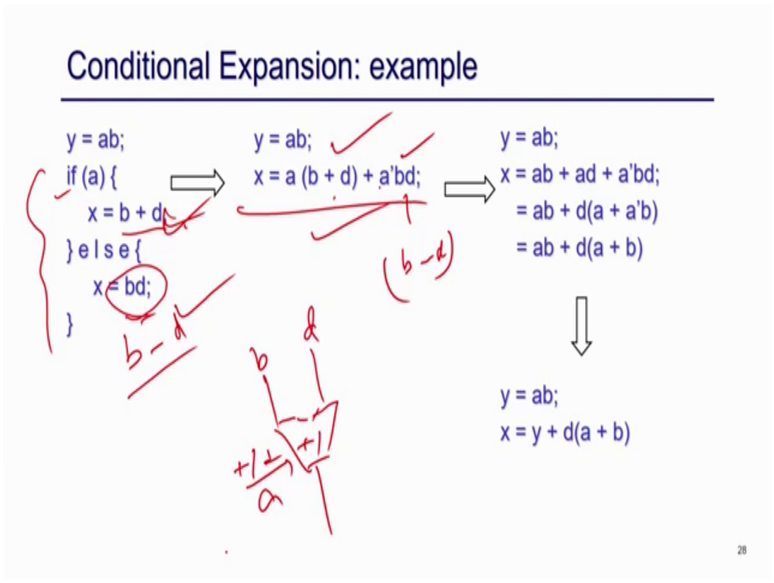
if (c)
s
else T

$c.s + c'T$

27

So, I will move on to the next topic is called conditional expansion is; basically, when you have a say a conditional expression if say C say are doing something at else I am doing something right. So, I can always do say I am doing say here s. So, I can always do c to s plus c bar into c bar into say this is say T right c bar into T. I can always do that. So now, instead of doing this in sequential I can do it in parallel.

(Refer Slide Time: 55:10)



Conditional Expansion: example

```
y = ab;
if (a) {
    x = b + d;
} e l s e {
    x = bd;
}
```

y = ab;
x = a (b + d) + a'bd;

y = ab;
x = ab + ad + a'bd;
= ab + d(a + a'b)
= ab + d(a + b)

y = ab;
x = y + d(a + b)

So, here is an example. So, so suppose I am doing if a equal to true, then I am doing this else (Refer Time: 55:14) with this.

So, I can rewrite this whole thing by this, right. Instead of doing this in conditional, I can do everything in parallel right this is a into b plus d, and then a bar into b T, right. This is what I can do. So, this is what is called conditional expansion. So, I am replacing the conditional body because whenever you have conditional body you have some marks, some decision maker whether you could do this or that. So, all this control signal will be enumerated. I can do it directly, right, this is what is the advantage of the conditional expansion. But the disadvantage is that earlier in hardware I had to do either this, or this I do not have to do both, right.

Suppose this is also b minus d. probably I can use a same ALU which can do plus or minus to do both the operation, right. Because they are mutual expressive operation either this will be executed or this will be executed. So, in hardware I can share a resource for this right. So, for this if I just suppose this is b minus d, I can just do b and d. And I have only one conditional control signal that will decide whether plus or minus ok. Then it will just do this either b plus d or b minus d based on a right this is if this is basically a equal this is a or not right. So, if it is a then it will be done or like this right.

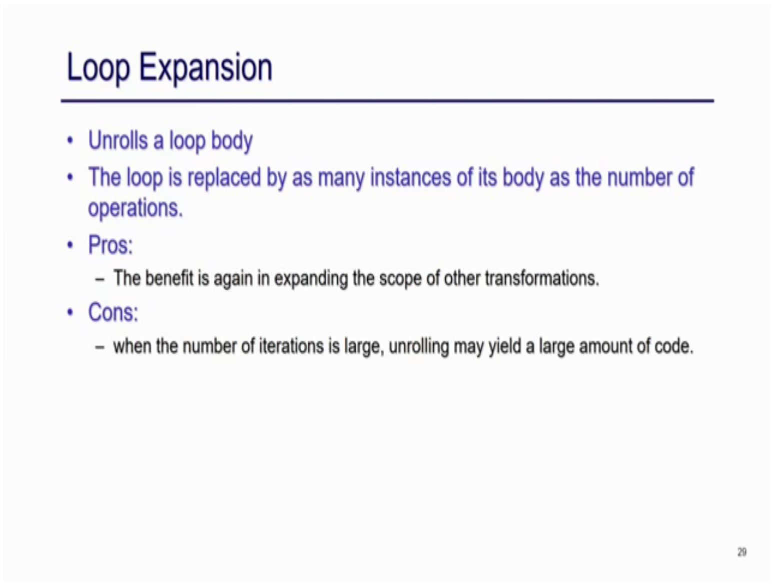
So, but I can share this recourse for both the operation, but if you just do it here I have to do both the operation right. So, if instead of b b d, I can do b b minus d. So, I have to do both b plus d as well as b minus d just to in this case.

So, this is the negative part of this, it will increase the resource, but I can do instead of this things I can do in the all these things in parallel. And another advantage of doing this conditional expansion that it is if these variables are Boolean variable right. So, it will actually increase the scope of application of logic optimization. For example, suppose this is a Boolean expression, I can do certain kind of optimization, right some manipulation Boolean manipulation and finally, I can reduce this big expression by a smaller expression, right. This is what I can do. So, that is another advantage of conditional expansion in case of Boolean variable. So, I can apply some Boolean logic optimization just to get a simpler version of expression, right. Instead of doing all this bigger expression I can have a simple operation to be done right.

That is another advantage of conditional expansion.

(Refer Slide Time: 57:37)

.



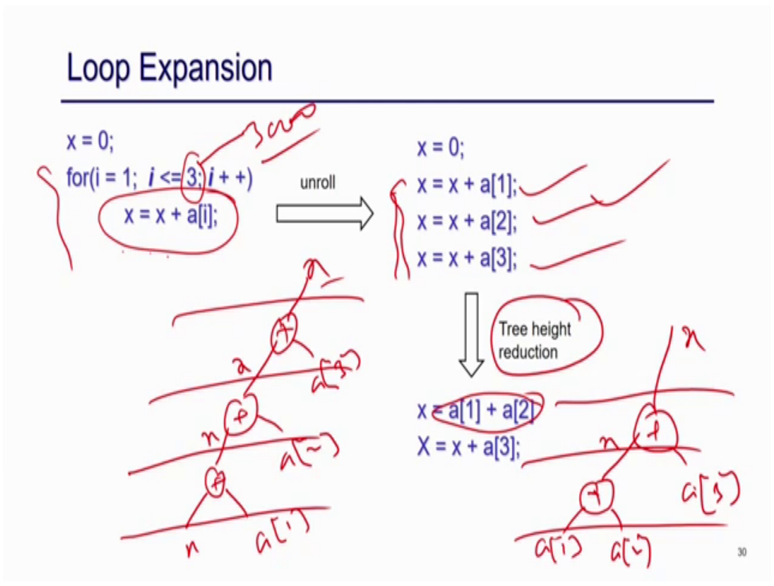
Loop Expansion

- Unrolls a loop body
- The loop is replaced by as many instances of its body as the number of operations.
- Pros:
  - The benefit is again in expanding the scope of other transformations.
- Cons:
  - when the number of iterations is large, unrolling may yield a large amount of code.

29

So, I will move on to the next topic which is called loop expansion. So, this loop expansion is nothing but loop unrolling, right. If you have a loop body you just unroll it and replace by the whole loop body by a series of operation right.

(Refer Slide Time: 57:52)



Loop Expansion

```
x = 0;
for(i = 1; i <= 3; i++)
    x = x + a[i];
```

unroll

```
x = 0;
x = x + a[1];
x = x + a[2];
x = x + a[3];
```

Tree height reduction

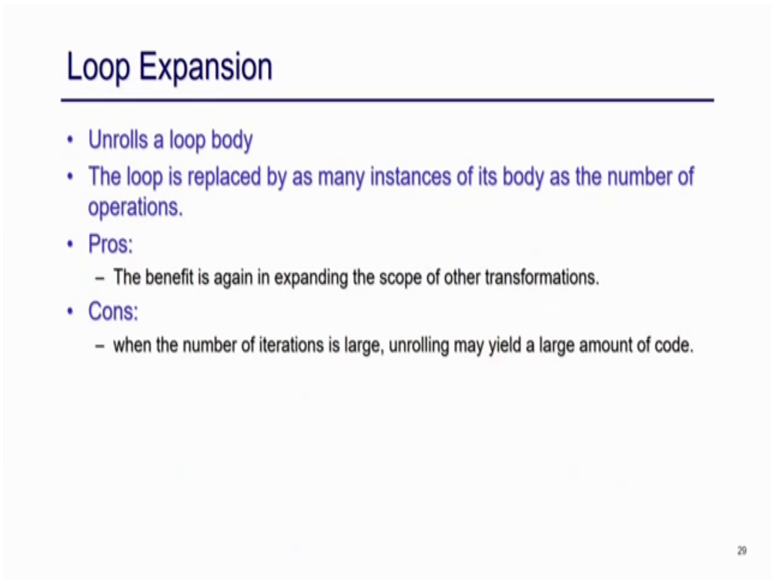
```
x = a[1] + a[2];
X = x + a[3];
```

3cw
x
a[3]
a[2]
a[1]
+

For example, here suppose we have this loop, where this I is going from one to 3, I am doing this I can replace this loop by this set. So, I can just do x equal to x plus a 1, then x equal to x plus a 2, and then x equal to x plus a 3. So, I just replace this loop by this what is called loop unrolling. And what is the advantage of having this? the advantage is that once you have done this, probably you can actually apply the other kind of optimization for example, here I can apply tree height reduction because this is nothing but doing these things in series right.

So, you are doing x plus this is your x you are doing this is a 1, then you are doing this is x you are doing a 2, and then you are doing this is your x. And this is a 3 right and then you are getting this. So, instead of doing this, we can do in parallel right. So, I can do this x equal to this is basically you can do a 1 and a 2. And this is your x and then you can do a 3 this is your another addition this is your a 3 equal to x. So, instead of this 3, time step I need basically now 2 times step, right. I can do it in 2-time step. So, basically idea is that when you do this unrolling, it actually widens the application of other optimization. So, that you can optimize your code ah, but and also the up and also another advantages is that you have to maintain all this condition this control signal will be more here, right because we have to check whether this I less than 3 or not those things you have to maintain, but here you do not have to maintain.

(Refer Slide Time: 59:45)



Loop Expansion

- Unrolls a loop body
- The loop is replaced by as many instances of its body as the number of operations.
- Pros:
  - The benefit is again in expanding the scope of other transformations.
- Cons:
  - when the number of iterations is large, unrolling may yield a large amount of code.

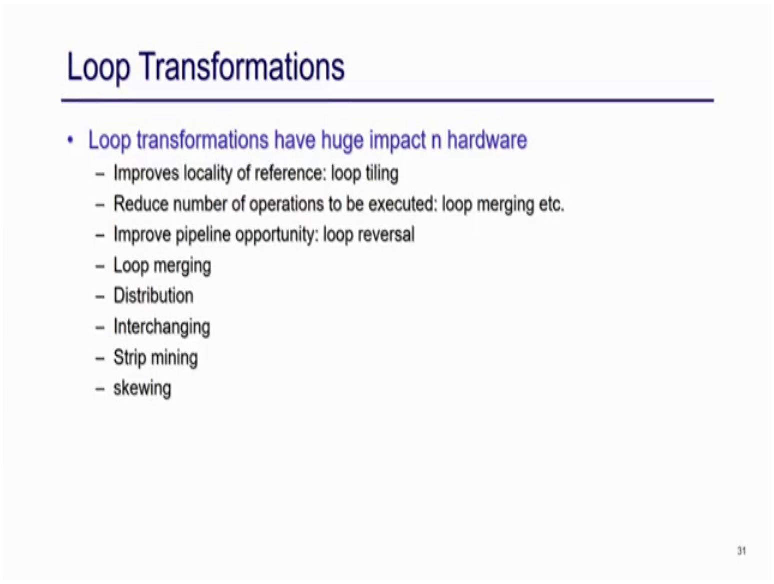
29

So, your the complexity of the control will also get reduced if you just do a loop expansion. But you understand that in this loop is very the number of iteration of the loop is very high. So, if you just unroll blindly it will increase the code size by and large, right. And that is not something is good right because if you say this instead of 3 you have 3,000. So now, you have 3,000 operation it has to be executed, right. That is may not be a good idea to do a execute instead of doing it it is better to do it truly.

So, you should have some have some have some trade off. So, you sometime we just look for partial number. Instead of whole things unrolled. So, there also as I had mentioned I I unroll it say your loop is 300 3,000 times and in 3,000 times I can unroll by 3 only, then I I am doing 3 operation in one loop and I am going to execute the whole loop of thousands time thousand times. So, I can also go for partial unrolling if you are by loop interest is too high, right.

So, you should always think about a tradeoff between the code size versus. this control size or the kind of benefits that you are going to have in a loop body. What is called loop unrolling?

(Refer Slide Time: 60:47)



Loop Transformations

- Loop transformations have huge impact n hardware
  - Improves locality of reference: loop tiling
  - Reduce number of operations to be executed: loop merging etc.
  - Improve pipeline opportunity: loop reversal
  - Loop merging
  - Distribution
  - Interchanging
  - Strip mining
  - skewing

31

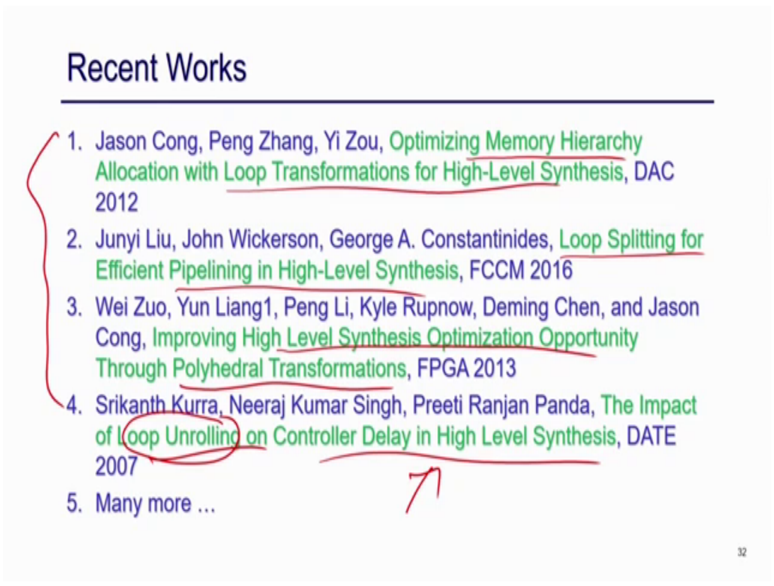
So, in fact this loop transformation is very is very important in the context of others synthesis, because loop has a huge impact on generating hardware right. So, a lot of work has been done on this people have tried to check what kind of loop transformation is going to improve your locality of references. So, that during successive operation you do not have to read multiple data from your from adding. So, that whatever operations happening through some variable values that are going to be done together right .

As I have mentioned earlier, also that whenever you have a a loop or say a loop and you are add is going to mapped into the memory. And in memory we do not have multiple port, right, we have only one or 2 ports. So, if you have say multiple access 4 5 access at a time I cannot execute all of the in 1 plus so, I need multiple. On the other hand, if I found a 5 is going to use say 5 scenarios. So, I can do those 5 operations at together right. So, then I can read a 6 and whatever, the value operation is going to depend on a 6 I am going to do that.

So, if we just do some certain kind of loop transformation so that all this operation which has a locality of references, they have greater locality operations happen in successive iterations that will improve your number of memory read, and hence the number of time required to execute that right. So, that is something I mean a lot of work has been done and looping is one kind of operation or loop skewing another kind of operation which actually improve this locality of references, also people try to do it x merge to loop or loop merging try to merge 2 loops so that you can do some kind of paralyzation, or number of time step is required to execute that can be eliminated or other kind of optimization just to improve the pipe training opportunity like loop reversal and certain like this.

Actually, in fact, there are a lot of loop transfers and techniques are there and there are various work actually done on this just analyzing whether how hard type kind of effect how certain loop transformation have in the in the generator hardware. And that is in fact, is a another discussion topic that maybe another lecture just to discuss on that. So, instead of going into detail of that because this is just to give you a highlight that loop is an important factor, and there are a lot of optimizations and that has a significant impact in the hardware right.

(Refer Slide Time: 63:24)



Recent Works

1. Jason Cong, Peng Zhang, Yi Zou, Optimizing Memory Hierarchy Allocation with Loop Transformations for High-Level Synthesis, DAC 2012
2. Junyi Liu, John Wickerson, George A. Constantinides, Loop Splitting for Efficient Pipelining in High-Level Synthesis, FCCM 2016
3. Wei Zuo, Yun Liang1, Peng Li, Kyle Rupnow, Deming Chen, and Jason Cong, Improving High Level Synthesis Optimization Opportunity Through Polyhedral Transformations, FPGA 2013
4. Srikanth Kurra, Neeraj Kumar Singh, Preeti Ranjan Panda, The Impact of Loop Unrolling on Controller Delay in High Level Synthesis, DATE 2007
5. Many more …

71

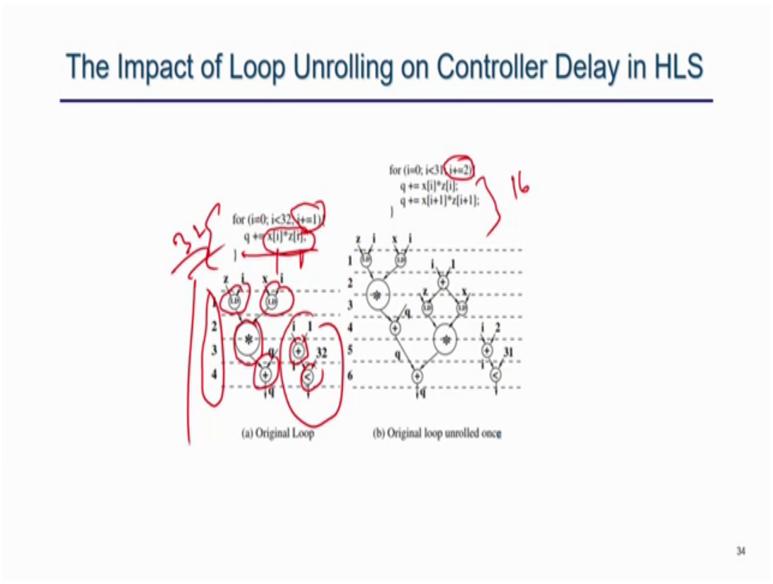
32

So, that is just I am going to highlight it and those who are interested going to much detail of this research area. Just for your references, I just copy some 4 recent work that actually working on this loop transformations and their impact on high level synthesis.

This work (Refer Time: 63:34) for example, is working on the optimization and memory hierarchy allocation for loop transformation for high level synthesis. Similarly, this is working on loop splitting for efficient pipelining in high level synthesis. This work is working on say a high-level synthesis optimization opportunity through poly hydral transformation, polyhydral is another way of applying loop transformation or representing in loop, and this paper works on that how to improve the optimization opportunity through loop transformations. Another work is something impact of loop unrolling in controller delay in high level controller delay in high level synthesis.

There are many more so, this is just as a starting point for you. And since we have already discuss about this loop expansion and loop unrolling, I am going to discuss this work briefly. Just to give you the; what kind of impact of loop unrolling on the controller delay ok. So, if you take an example of this right. So, you have an example where I am doing this for 32 times, I am doing this operation right.

(Refer Slide Time: 64:30)



The Impact of Loop Unrolling on Controller Delay in HLS

```
for (i=0; i<32; i+=1)
    q += x[i]*z[i];
}
```

```
for (i=0; i<31; i+=2)
    q += x[i]*z[i];
    q += x[i+1]*z[i+1];
}
```

(a) Original Loop

(b) Original loop unrolled once

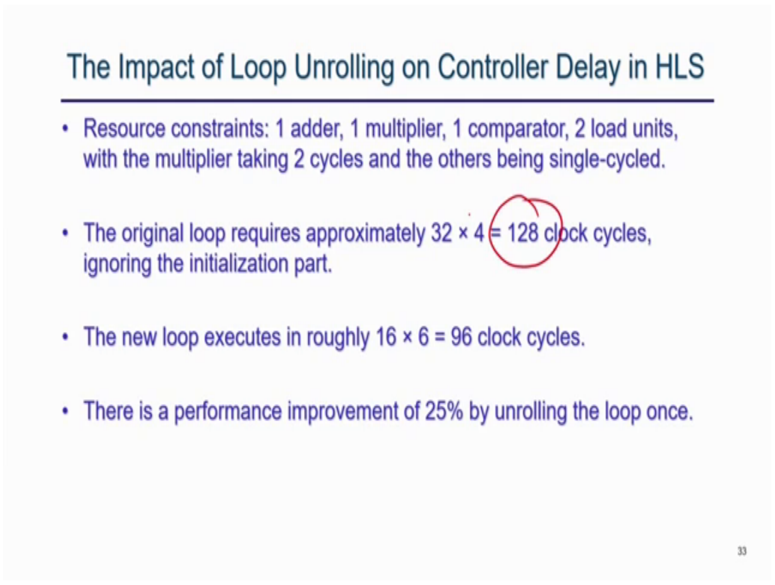
34

So, I just as I have mentioned I am going to partially unroll, I will just unroll by 2. So, what I am doing here? I am doing 2 operations in the loop, this 2-successive iteration, I am increasing the loop iteration by true early I am increasing it by one. And I am doing it 16 times, right. Earlier it was I am doing it 32 times I am now doing 16 times. And if you just draw the data flow graph, I am doing this loading operation loading is basically memory read right.

So, I am just read z i and x i and then I do this multiplication. I am doing this multiplication I am assuming that I have multiplication is of 2 cycle operation and this load addition comparator all are using a single cell operation. And I have one multiplier, one adder, one comparator and 2 loader available, right. And then I just do this x into z is after 2 cycle, I just do q equal to q plus this right this is what I am doing here. And in this place, I am just doing I equal to I just do I plus 1 and then check whether this is less than

32 or not. So, what I do? I just to execute this I need 4 cycles just to satisfy that dependency.

(Refer Slide Time: 65:49)



The Impact of Loop Unrolling on Controller Delay in HLS

- Resource constraints: 1 adder, 1 multiplier, 1 comparator, 2 load units, with the multiplier taking 2 cycles and the others being single-cycled.
- The original loop requires approximately 32 × 4 = 128 clock cycles, ignoring the initialization part.
- The new loop executes in roughly 16 × 6 = 96 clock cycles.
- There is a performance improvement of 25% by unrolling the loop once.

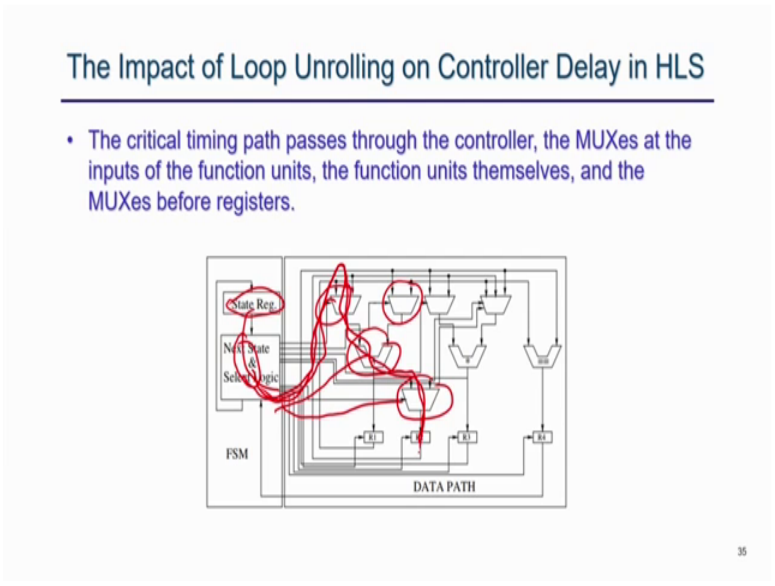
33

Since this is going for 32 iteration, roughly, I need 120 8 clock cycles. I need someone or 2 clock step to initialize all those I just ignore that. But roughly I need 130 32 into 4 128 clock cycles. On the other hand, if you have this partial unroll, I have this load here xyz, then I am doing this xi to zi. And then I am doing q equal to q plus, right. Q plus q plus this is xyzy that I have done. And here I am doing this i equal to i plus because this i plus is used for i plus 1. Then I am reading this xi plus 1 here, and there is a xi plus 1 here, and then I am doing this multiplication for 2 cycle. And then finally, add this result with this ok.

This is my final que, and this part is similar to this I am doing I plus 2, because now it equals to I plus 2 it is I plus 2, then you just checking whether this is less than 30 or not right. So, you can see just to satisfy the data dependency here I need 6 clock steps, right. And this is I just mentioned is going to execute 16 cycle. So, 6 into 16 I need 96 clock cycles. We can see directly I have savings of 25 percent of clock cycle. If I just do it partial unrolled by 2. So, it is good right I have a very good. In fact, of unrolling in the number of clock cycle we need to execute this division. So, it is good, but let us now try to show in another angle that what is the kind of impact of this unrolling because this unrolling factor can be different, right.

So, this is to I will just do it unroll by 2 I can do it unroll by 4, 6, 7, 8, 9, 10, 11, 12 anything. I can do partial unrolling by anything so, we will try to, this one try to this one specifically try to find out what kind of impact of this unrolling in the latency the number of clock cycle is required to execute this. And also, it is impact on this controller delay ok.

(Refer Slide Time: 67:43)



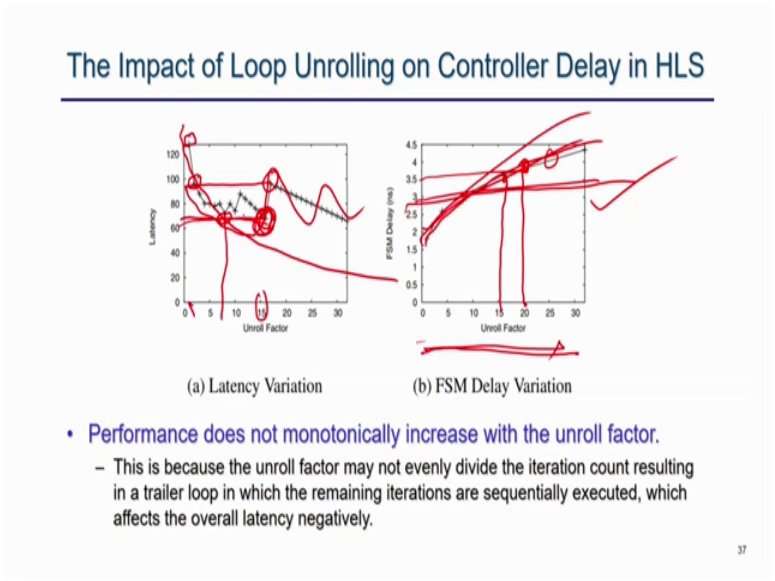
The Impact of Loop Unrolling on Controller Delay in HLS

- The critical timing path passes through the controller, the MUXes at the inputs of the function units, the function units themselves, and the MUXes before registers.

35

So, they have plotted this that in this graph right.

(Refer Slide Time: 67:47)



The Impact of Loop Unrolling on Controller Delay in HLS

(a) Latency Variation (b) FSM Delay Variation

- Performance does not monotonically increase with the unroll factor.
  - This is because the unroll factor may not evenly divide the iteration count resulting in a trailer loop in which the remaining iterations are sequentially executed, which affects the overall latency negatively.

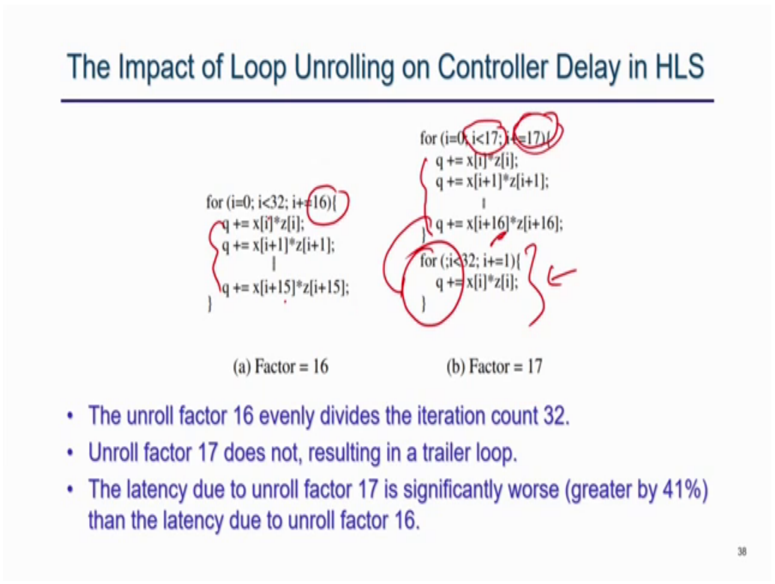
37

So, what is that? So, let me just figure it out. So, the let us so, let us put this latency variation. So, as I mentioned this for unroll factor 2 I have get 128. So, initially I have there is no unroll this is 128, right, if you unroll by one, then I am getting 96.

So, if I just plot this ideally if you just unroll by 3 this should go down right. So, it is going down and then finally, it is become not exactly this right. So, I expecting like this, but your number of time step required is not reducing exactly this way, right. Why because this is not linear. So, why this is the case? Because if you just do unroll with the because the actual loop is executed 32 times if you just do unroll by 2 now it going to do exactly 16 times and you can do 2 operation at a time, but if you just do it unroll by 16 again, you can do this 16 operations at a time. And I can iterate the loop 2 times.

But if I just unroll by 17, what will happen? I can do only 6 17 operations at a time when I and the remaining operations 15 I have to do without loop right. So, another loop I have to do the tailing loop I have to write.

(Refer Slide Time: 68:56)



The Impact of Loop Unrolling on Controller Delay in HLS

```
for (i=0; i<32; i+=16){
  q += x[i]*z[i];
  q += x[i+1]*z[i+1];
          |
  q += x[i+15]*z[i+15];
}
```

(a) Factor = 16

```
for (i=0; i<17; i+=17){
  q += x[i]*z[i];
  q += x[i+1]*z[i+1];
          |
  q += x[i+16]*z[i+16];
}
for (;i<32; i+=1){
  q += x[i]*z[i];
}
```

(b) Factor = 17

- The unroll factor 16 evenly divides the iteration count 32.
- Unroll factor 17 does not, resulting in a trailer loop.
- The latency due to unroll factor 17 is significantly worse (greater by 41%) than the latency due to unroll factor 16.

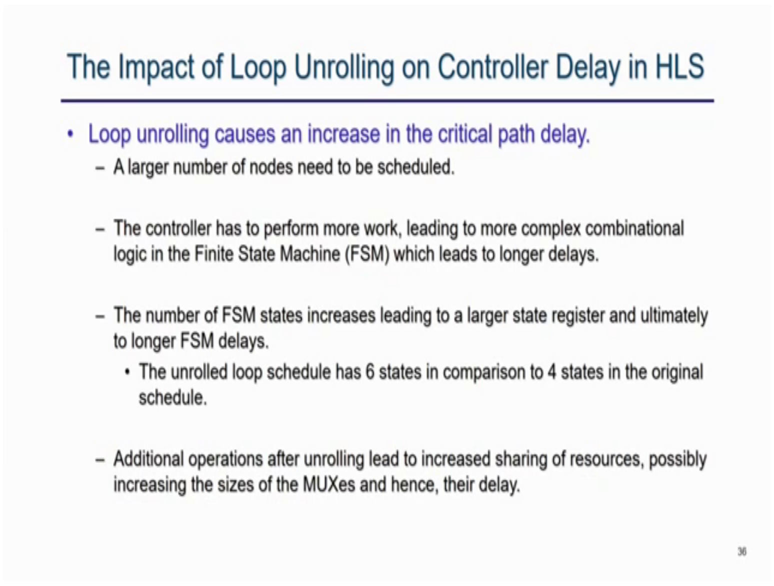
38

So, that is what is given here. So, if I just do unroll by 16 either 16 operations, and the loop is going increase by 16. So, this loop will execute 2 times it is fine, but whenever this is a 17. So, I have this 17-operation written here, you can see here this 6 i to 16 and i to 15 because there are 17 operation, but if I increase the I to 17, and this I less than 17 will not be true so, I have to come out.

So, from this 17 18 to from 17 to 30 those operation is still remaining that has to be executed right. So, that is a trailing loop, what is called and that has to be executed. So, this is cannot be this has another separately, and that is actually causing this latency in this right.

So, you can see in this figure 16 has a good around you need around 64 or something right 679. But for 79 need a almost a 90 or 95 or something. So, the number of latency does not linearly decrease with the unroll factor. That is clear right, but the; what is the controller delay that let us understand now.

(Refer Slide Time: 70:02)



The Impact of Loop Unrolling on Controller Delay in HLS

- Loop unrolling causes an increase in the critical path delay.
  - A larger number of nodes need to be scheduled.
  - The controller has to perform more work, leading to more complex combinational logic in the Finite State Machine (FSM) which leads to longer delays.
  - The number of FSM states increases leading to a larger state register and ultimately to longer FSM delays.
    - The unrolled loop schedule has 6 states in comparison to 4 states in the original schedule.
  - Additional operations after unrolling lead to increased sharing of resources, possibly increasing the sizes of the MUXes and hence, their delay.

36

So, if you just see a generic data path that is generated we have a, few and there have been mux at the input of a few and the output of a few if go to register.

And there will be some mux again right we have discussed all this things right and whenever. So, what is the controller delay? So, based on the current stage you generate the next stage. So, there will be some delay here and that signal will come here it will come here and it will come here. So, this signal effective propagate to this multiplexer this a few this a few to register. So, this is the controller delay path right. Now if you increase this unroll what will what will happen here? So, just see the number of state increase right.

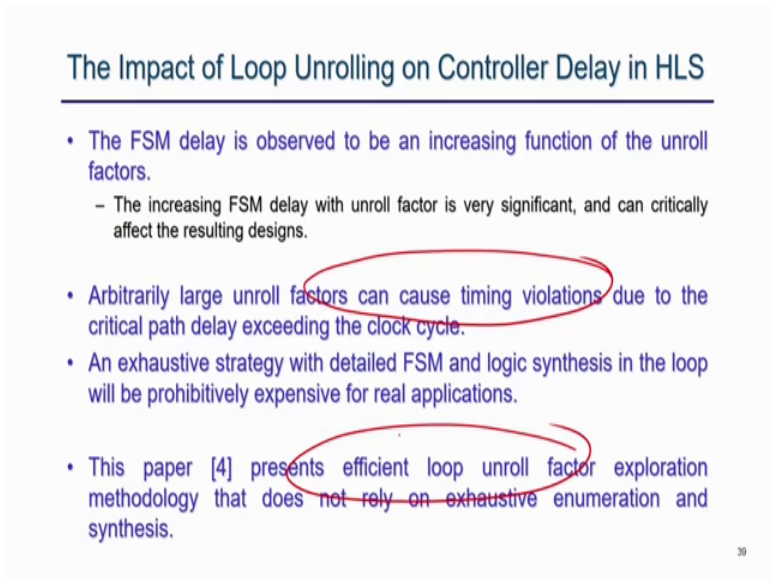
So, the complexity of the controller will increase. So, the controller complexity of delay of this particular thing will increase right. Because now a lot of state the decision making here is bigger. So, the controller delay of the ff will increase. And also, similarly if you see here I am doing earlier only 2 operation in chain now 3 operation in chain I am now doing 1 2 3 4 operation in chain. So, this chain operation chaining length is increased so that then you have to now it put extra mux here and those cases. So, that delay of the data path will also increase right. Because that multiplexer chain will be increasing sharing of research possibility increasing the size of mux right since the delay.

So, if you have to do the unroll effectively this delay here also will increase, and delay of this path will increase because the number of mux the mux will come when this path will high and also the number of operator will come high right. So, the total delay of these controller will increase right. So, unrolling is not good for controller delay right. If you just so, delay will increase and that is actually plotted here. You can see whenever the unroll factor is increasing your delay is actually monotonically increasing, which is not like this. So, we can understand that so, you should have a trade off right you can adjust a do a arbitrary number of unroll, because your computation delay may go beyond your clock period. And in that case, you basically the kind of target clock you may violate the target clock, because your if you computational delay go very high say suppose your target clock is 3 millisecond if it becomes 4, then you are actually violating the time.

So, your design is not meeting the time step right you just unroll by say 20, you are violating the target clock (Refer Time: 62:26) right. So, that is a serious violation. So, there should be some trade off. So, you can see here probably this is the best choice, right in this case your latency is minimum which is 16, but in this 16 also the total delay is say 3.5 or something. And suppose your target clock is 3 point you are meeting the time (Refer Time: 72:47) ok. So, this is something that is trade off is among this all this unroll factor probably, I should change this 16 which is a better choice or maybe something else.

So, based on the target clock, you are target clock is not that high probably you have to choose this right-angle factor of say 8. So, that is the point here right. So, basically given of design and your target clock your what this unroll factor you choose right, so that your controller delay is not going so high that is meeting that does not meet the timing; on the other hand, you have a good benefit in the latency as well.

(Refer Slide Time: 73:24)



The Impact of Loop Unrolling on Controller Delay in HLS

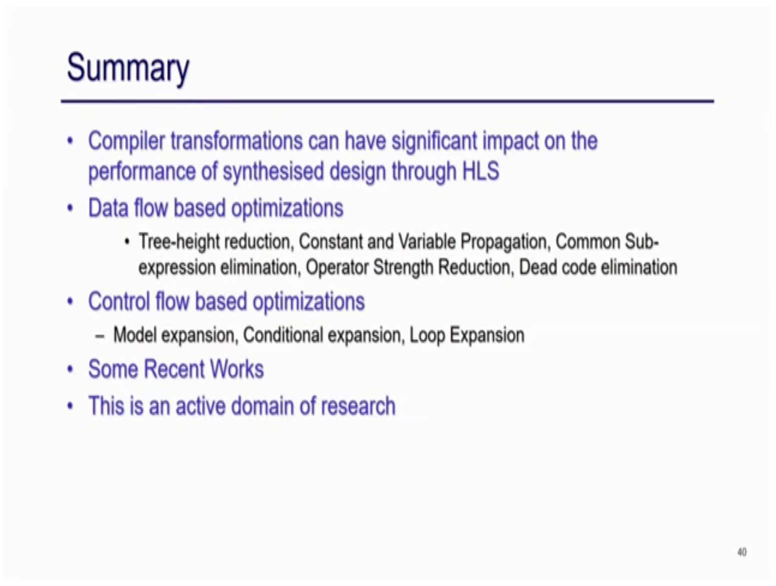
- The FSM delay is observed to be an increasing function of the unroll factors.
  - The increasing FSM delay with unroll factor is very significant, and can critically affect the resulting designs.
- Arbitrarily large unroll factors can cause timing violations due to the critical path delay exceeding the clock cycle.
- An exhaustive strategy with detailed FSM and logic synthesis in the loop will be prohibitively expensive for real applications.
- This paper [4] presents efficient loop unroll factor exploration methodology that does ~~not rely on exhaustive~~ enumeration and synthesis.

39

So, that is what is work all about this work specifically propose an algorithm that we will have to find out the tradeoff between this and this latency number of latency required versus the number of this computational delay. So, the; it meets the target clock, but you achieve the minimum latency right.

This is what is all about. So, similarly if you looking into this other work. They are also tries to do this kind of advanced kind of checking. Just to see for example, this work actually work on memory hierarchy right. So, how do you improve the memory hierarchy using loop transformation? So, these are the kind of techniques are there so, this can be used just to utilizing this can be used to utilize your improve your performance of a design right.

(Refer Slide Time: 74:16)



Summary

- Compiler transformations can have significant impact on the performance of synthesised design through HLS
- Data flow based optimizations
    - Tree-height reduction, Constant and Variable Propagation, Common Sub-expression elimination, Operator Strength Reduction, Dead code elimination
- Control flow based optimizations
    - Model expansion, Conditional expansion, Loop Expansion
- Some Recent Works
- This is an active domain of research

40

So, this is another area of research. So, in summary, we can see that this compiler optimizations are have a significant impact on the generator hardware. And we have specifically discussed 3 kind of work specifically the data flow-based optimizations. We have also discussed some control flow optimizations. And also, we have discussed some recent works and loop transformations their impacts on hardware. And I can say this and active domain of research, and people are trying to find out the impact of this optimization techniques various area like in memory in optimizations of the bit width optimizations of the computational time, optimizations of power optimizations of routing length.

And various aspect you can always think of having implementing or applying this kind of optimizations. And see their impact on the generator hardware. So, with this I am going to conclude this session, and in the next session, we are going to discuss about this article optimization technique for powers timing.

Thank you.