C-Based VLSI Design

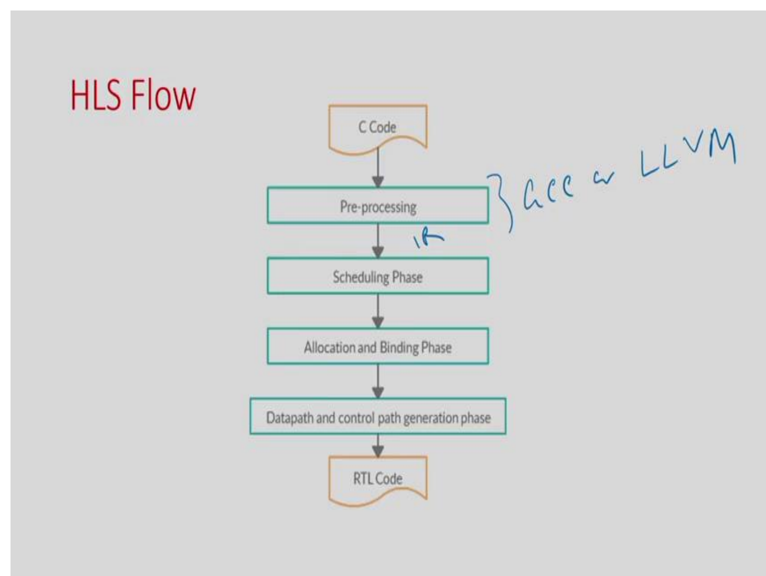
Dr. Chandan Karfa
Department of Computer Science and Engineering
Indian Institute of Technology, Guwahati

Module - 09
Impact of Compiler Optimizations in Hardware
Lecture - 29
Frontend Optimizations in High-level Synthesis

Welcome everyone in today's class, we are going to discuss on Frontend Optimization in High level Synthesis. So, if you remember the high level synthesis flow, it is basically we start with a input behavior written in C C plus plus and then we do them preprocessing, then we do scheduling, then allocation binding and then data path and controller generation right and finally, we will get the RTL.

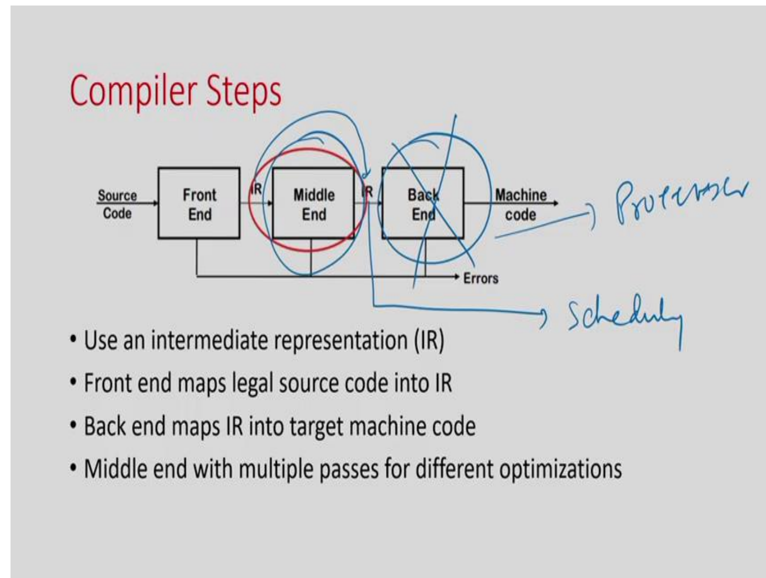
So, in the preprocessing step, we have already discussed that in the preprocessing step we have primarily use the C compiler right, just to parse the input C and create some intermediate form right and from that we extract certain information and then you start the scheduling in some point, right. So, now if you look into this that preprocessing part little bit closer, let us look into this compiler right, what whatever the compiler is getting use in the pre processing part.

(Refer Slide Time: 01:42)



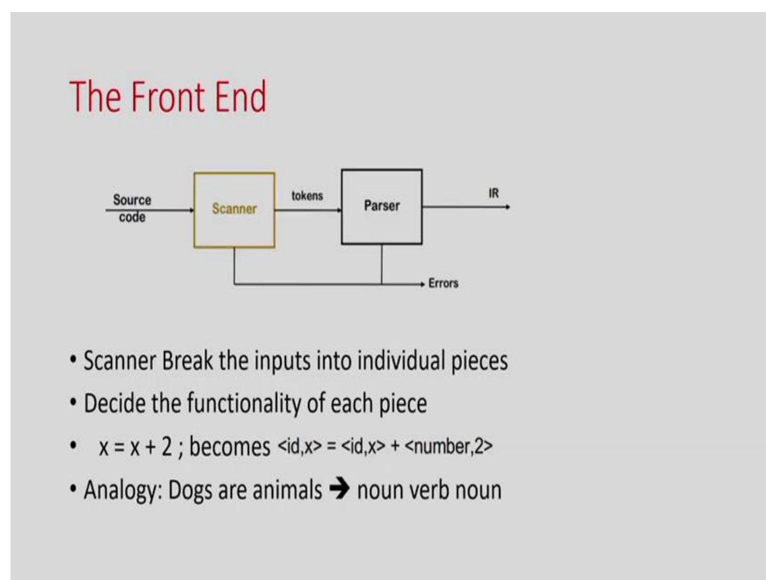
So, as I mentioned here, we usually use GCC or say LLVM kind of compiler just to create some IR, right. So, which will create some intermediate representation and with that information representation that scheduling and everything starts, right.

(Refer Slide Time: 02:04)



So, now if you look little bit closely on this GCC or any other C compiler, it has kind of three primary phases, right. So, front end, middle end and back end. And in the front end what we do? In front end we usually do this scanning and parsing, right.

(Refer Slide Time: 02:14)



So, basically it is a syntax analysis and semantic analysis, right. So, what we do is, we have the input C code, we parse it, we break this into token right; we have to identify individual terms or the and then identify their types. And in the semantic analysis is just we have to just make sure that is actually satisfy the syntax of the C code, right. So, that is what is really is done in the front end, right. So, this is very conventional steps.

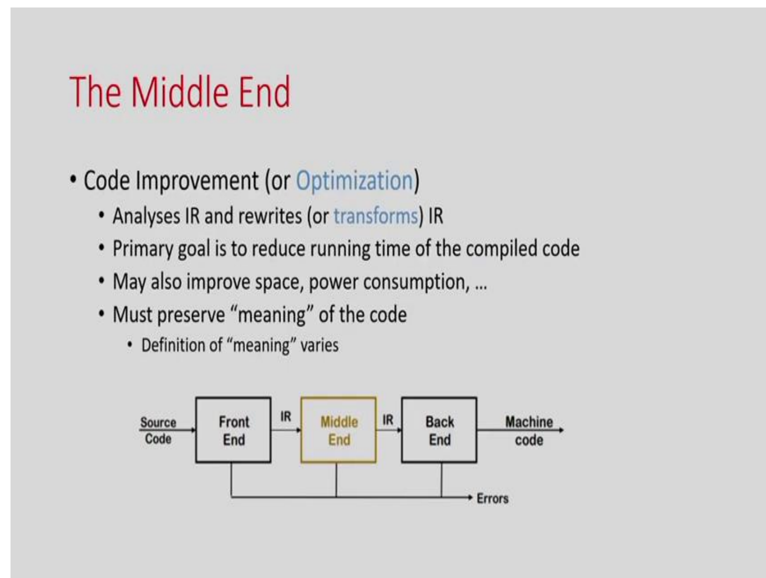
And we no need to go into much detail into this; because this is already done by some existing compiler, which is integrated to with some high level synthesis tool, right. And it generates some kind of intermediate representations or IR, ok. So, this IR is something now goes for the next phase, which is called the middle end, right.

So, that middle end what it does is, basically apply various kind of optimizations, right. So, the primary objective of this optimizations is to improve the performance in terms of area, power or timing whatever the optimization goals and then it again modify this IR into another form of IR, right. So, we have that IR. And if you take the normal the GCC flow, now this IR will be converted into the machine code or assembly language, right.

So, which converted into assembly or then it will converted into machine language which is something by done by the back end and then that will run in some processor, right. But we do not need this step; because we do not need this, because that convert those C code into instruction, a set of instruction which can be executed in a target processor. What we are going to do? We are going to take this and then we will start the scheduling stuff, right.

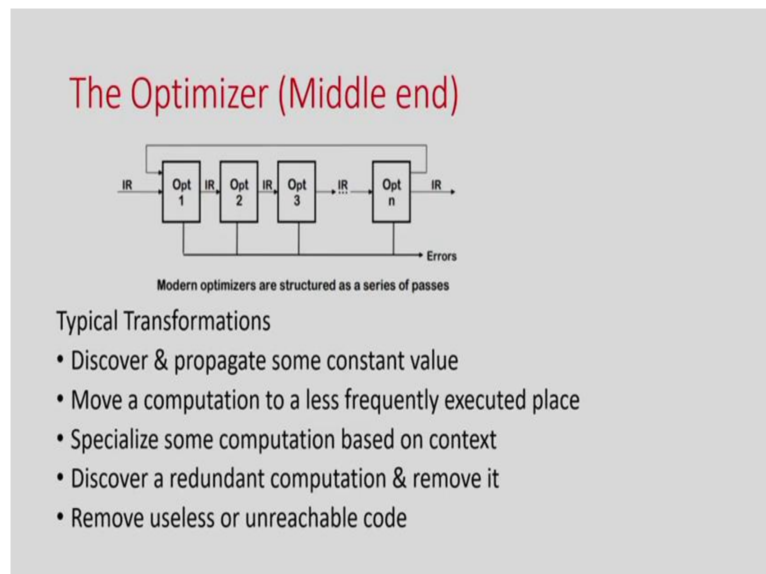
So, this is what is the process. So, we do not need to go into the back end stuff, how it gets happen; but the point of concern or interest is our is the middle end, where the optimization case applied, right.

(Refer Slide Time: 04:14)



So, if you look into this middle end, which is basically is also called the code improvement or optimization phase, where we actually analyze the IRs and rewrite it to another IR, right. And primary goal is to improve performance that I have already talked about.

(Refer Slide Time: 04:34)



So, in this optimization phase is not a single phase, it has various optimization phase, right. So, if you look into the LLVM or say in GCC, we have optimization phase 1, 2, 3, 4 many other phases are there right and every phases if certain kind of optimizations is

targeted, right. So, what kind of term, optimization usually this particular this optimization phase works. So, it basically do this five kind of optimizations; it is basically identify the constant value and it propagate the constant right and it actually we will see how it impacts.

So, that is one kind of transformations; it basically identify certain optimal operations which is actually happening in some, it is place some places where it is unnecessary computing some things which is a repetitive thing, probably it moves to some less frequent execution place, so that I do not have to execute that particular thing many times, ok. And also sometime it does the specialized computation based on context, right.

So, say for example, operator, strength reductions and so on I am going to talk about that and then also discuss on the identify the redundant computations and remove it, it remove the unreachable code or useless code or dead code and remove this, right. So, this kind of optimization usually this optimization phase does, right.

Now, in the context of high level synthesis, we term this optimization as a front end optimization; because this is what happened in the before the actual processing of the things done in the high level synthesis right, the schedule I mean we can consider the scheduling allocation binding and data path control which is the primary phases of high level synthesis and this is the front end, right. And since I we integrate this GCC or any other compiler into the HLS flow, we can always enable or disable this optimizations right and they will impact the high level synthesis, right.

So, the purpose of this in today's class to understand what are the common such optimizations applied during this optimization phase and how they are going to be impacted in the hardware, right. So, once we understand that part and then if say in a particular applications you are actually processing; you found that this kind of optimization should not be applied here and hence you can actually turn it off, right.

So, during the high level synthesis run, you can actually options to turn it off a particular phase of the optimization say a one or two or three whatever it is, you can actually turn them off. So, or you can enable them in either way, right. So, that is something is we should understand the impact and then based on our application goal, we can do that, ok.

In addition to that there are certain optimizations which is not usually done by the compiler, specifically the loop transformations which has a significant impact in the high level synthesis results. And it you may need to apply them manual in the source C code, right. So, the by front end optimizations I mean both kind of optimizations; one set of optimizations which is done by the C compiler in the optimization phases and certain kind of optimizations which is not usually done by this optimization phases; but they are actually can be done manually, you can rewrite your C code to by applying those kind of optimizations and it has a significant impact in the high level synthesis results.

So, I am going to talk about all these kind of optimizations very briefly. So, the first and one of the important optimization is a loop invariant code motion, ok. So, as the name suggest, it is basically you have something invariant to the loop, right. So, what do you mean by the invariant? Invariant is means it does not change within the loop, right. So, that means we if we in the loop body, you have certain expressions which is invariant to that loop and you are calculating it multiple times, because loop will execute many times, right.

Now, if you execute that particular operations within the loop is basically is a redundant computations right and every time if you execute this in hardware, it needs some time right, some clock steps some resource. So, why not you actually move that particular operation out of the loop right or that expression out of the loop, you compute it once and reuse within the loop, right. So, that is what is called loop invariant code motion. So, code motion in the sense moving the code, right.

So, you are moving some code, which is invariant to the loop body and out of the loop body, right.

(Refer Slide Time: 09:04)

Loop-Invariant Code Motion

```
do i = 1,n
  a[i] = a[i] + sqrt(x)
end do
```

(a) original loop

```
if (n > 0) C = sqrt(x)
do i = 1,n
  a[i] = a[i] + C
end do
```

(b) after code motion

- When a computation appears inside a loop, but its result does not change between iterations, the compiler can move that computation outside the loop.

So, let us take an example which will clear your the concept. So, if you have this code where you are actually having a loop, where you are calculating a i with a i plus square root of x, right. What is the invariant here? You can clearly see is that the square root x, right. So, there is no relation with square root x with the loop iteration.

So, for i equal to 1, I am going to calculate square root x; because x does not change within this loop body and hence the square root of x is also invariant to the loop body, right. So, for i equal to 1, the value will be same; i equal to 2 value will be same and so on, right. So, what I can do is actually, I can calculate this square root before the loop, right. And because I need only this if I if my loop bound is greater than 0, otherwise I do not even not need that, right.

So, I will not going to the loop also. So, I just have a condition if n equal to 0; then only I need this square root and I am going to store in a some temporary variable and within the loop I am using C, right. So, I have I computed square root in C and I am going to use it here. So, the advantage what I am going to get it here is, earlier it was the square root is a very complex operation in hardware and it was going to happen say n times and in n can be hundred, right.

So, 100 time your computer going executing the square root, but here you are actually doing exactly once, right. So, that is something obviously you can understand; it is a will give you the benefit in terms of resource as well as the computation time, ok.

(Refer Slide Time: 10:32)

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

So, let us move on to the next one. So, the next one is very important is called tree height reduction. So, if you remember in the initial classes when I take an example of diff q, I take a big expression, right. And then I told you that particular big expression that executing in a single clock is something is not feasible; in the sense that it need that computational delay that I am going to get for that circuit will be too high, as a result the achieve target clock will be very low, right.

So, usually what it does? It is break that big expression into three address form; that means in small small parts sub expressions and combining the sub expression, you will get the big expression, right. Now, the question is that, when you are going to break this big expression into sub expression; how do you break it, right? So, you can have you can break it in many ways, right. So, and then for different kind of breaking it, you might achieve different different of results, ok. So, let me take an very simple example, suppose you have doing this a plus b plus c plus d, right.

So, how many way you can break it? So, one option is that you break it a plus b, then you are adding with c, then adding with d, right. So, that is what I am doing here; x equal to a plus b, then you are doing x equal to x plus c, then you are doing x equal to x plus d, right. So, this is what I am doing here. And if you just break this way, the tree or the dependency graph will be like this right; that you have this a plus b. So, I mean you can do like this way also, that a plus b.

So, you are doing a plus b and then you are doing c here and then you are doing d here x plus d, right. So, you are doing d here, right. So, now, you think about if I do this way, I am going to get this, right. And now, if you since the height of the tree is three and if you do not apply this operation chaining, you need kind of three kind of time step, right. So, you need three time stamp, right.

So, you need three clock cycle to execute this operation; whereas if I just break this way, say I am going to take the same expression and I am going to break it this way ok, say I just do this, say this is my t 1, this is my t 2 and then I am doing this, ok. So, then it will be breaking like this. So, the expression is like.

So, what I am going to do it now is like this. So, I am going to do this a plus b and I am going to do c plus d and then finally, I am going to add this two into this right; that is will be my x 1, this is my t 1, this is my t 2. So, the same expression could have been break in this way also, right. And advantage of breaking this way is you can see here, I can do the things in to two time stamp, right.

So, the same expression, the way I just break the expression, sub expressions is directly impacting the number of clock cycle it is needed to execute it, right. So, this is what I am talking about this tree height reduction that, whenever you have big expression; you try to break it such a way that, the height of the tree will be minimum, right. So, that is what is the objective. And if you assume that, so the tree height is kind of a good measure of the number of clock cycle needed.

So, if the tree height is reduced, the overall schedule time and hence the overall latency of the circuit that will be generated will be less, right. So, this is what this tree height reductions and it is very important optimization.

(Refer Slide Time: 14:02)

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.

```
int x = 14;
int y = 7 - x / 2;
return y * (28 / x + 2);
```

```
int x = 14;
int y = 7 - 14 / 2;
return y * (28 / 14 + 2);
```

```
int x = 14;
int y = 0;
return 0;
```

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

10

Another important optimization is called constant propagation or constant folding; the idea from the name you can understand that you try to propagate the constant, right. So, if you have expression, you have certain variable may be constant right at some point of time. And if you write an expression, if you just try to calculate the expression, it will incur some resource right. Whereas, if you just replace those variable by the corresponding value, probably you do not have to calculate that expression or you may you may your the overall computation things will be reduced, right.

So, just to justify this, I taken very simple example; say suppose I have a expression like this. So, I have y equal to 7 minus x by 2 right and then return y star some big expression right, 28 by x and this. So, if you try to execute this in hardware, so I need one divider; then when one subtractor and then in this next step I need one adder, one divider and one multiplier, right. So, many operations is happening.

So, basically 1, 2, 3, 4, 5, 5 operation is happening in this particular expression and you have to schedule this 5 operations in your during scheduling. But you can understand here that this x is 14 is a constant, right. So, what I can do? I can just replace this x by 4. So, I am going to replace this 14, I am going to replace this by 14, right. So, if I just do this, what is going to happen? My y become a fixed value, right. So, y become now 0 right, because 7 minus 7 is 0.

So, y becomes 0 and if I now replace y by 0 in the next step again constant propagations, it can actually happening in recursively. So, now, if you just put it 0, the whole return value becomes 0; because 0 multiplies something is 0. So, this give you the idea that, you have 5 operation to be scheduled; but just propagating a certain constant, I do not have to do any operations right, I have to just assign the value of these variables, that is all.

So, this is what is called constant propagation or constant folding and you can understand very easily that it applying; this is very simple, but very important optimizations, which will directly impacting the number of clock cycle or the latency of the design as well as the hardware resource both, right. So, you need to execute less operations and hence you need less resource and also you need a less number of time; because you are not actually executing certain operations in the hardware, ok.

(Refer Slide Time: 16:26)

So, this is what is constant propagation. Similar one is the variable propagation or copy propagation; instead of the constant value, you might have two the some value, there are two copy of the same value, right. So, I can remove one of the copy right; that is it that is what is the copy propagation. So, and so it let us take an example. So, suppose you have this code right and you have a equal to x; so that means I have a two copy of the value, a and x is basically the same thing.

And in the next of the expression, somewhere I am using a, somewhere I am using I could have used x also. So, you might use both or you may use only one. So, what I

understand that, I no need to keep both n x in my code; I can just remove one of them, right. So, what I did, I just basically remove everything with x, right. So, I just remove a, right.

So, what I am going to do? I am going to replace everything by x. So, then I can actually, I do not need this code; so this become dead code and I can remove the dead code as well, right. So, now, this will be my code. So, this is the idea that, you have two copy of the variable one value and use one of them.

So, you remove one of them, right. So, that is what is the idea and you can have multiple copy also; not only two, you can have many copy of the same value. So, idea is that, you remove all the redundant copies. And you can understand from this example that this actually helps in reducing the number of variables and hence the number of registers in the hardware right; because if you have the less number of variable means, less number of register in hardware.

So, this copy progression directly impact on the registers; not on directly on the number of operations or number of the function unit, but it directly impacting the number of registers, ok. And also it might enable some other scope of applying other, other optimization also, right. So, for example, here after doing this, I just do the dead code elimination and then I just do a operator strength reduction from this to this, right. So, I just replace this multiplier by a addition, that is what is called operator strength reduction.

(Refer Slide Time: 18:37)

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

12

So, that we understood, let us move on; the next one is the common sub expression elimination. So, as you understand from the name is the common sub expression; that means if you have sub expressions, which is common and I just eliminate those other expressions, right. So, I am going to compute it once and I am going to reuse it. So, for example, here you see x plus y and x plus y. So, basically y plus x, so they are basically same. But if you execute this code in the hardware, you are going to do this two times, right. So, I do not need to do that.

So, what I can do, it is basically I can just do a equal to x plus y 1 and here I can just use a right, that is the idea. I am going to identify the common sub expressions, so which is something happening multiple times in the code and I just remove the occurrence of those sub expressions and I am going to compute only once, right. And we have to understand one thing is that, the functionality should not change between; say for example, if this is x, then these are not the same one, right.

So, if you just see this code, say suppose you have this and you this is basically x equal to a plus 1. Now, this x and this x is different, right. So, this x is maybe some value, now this x is basically x plus y. So, in that case this and this is not common sub expressions. So, I can only say do two expressions common sub expression; if the expression involves certain variables and they are actually not modified in between, between two sub

expressions, then only they actually represent the same functional, same functionality, ok.

So, the this x operates, this expression is identifying such a functional equivalent sub expression and remove them right, that is all. And you can understand the impact immediately right, the number of operation less; so hence the number of resource less in the hardware and also the latency, because you are not going to execute those expression multiple times in the hardware, ok.

(Refer Slide Time: 20:31)

Variable Renaming

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

m = a + b
n = m − c
m = d − e
O = m + p

m = a + b
n = m − c
m1 = d − e
O = m1 + p

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

So, this is also very common expression, then the next one is the variable renaming ok; this is also very interesting, but important optimizations, ok. So, what is basically you sometime rename certain variable to reduce this write after write dependency, ok. So, let me explain that write after write. So, you see here, I have a code here, where I am just writing m and then I am going to use it here and then again writing m ok and then I am going to use it here, right.

So, this is something, if you try to schedule, it need kind of four clocks, right. So, you are going to do this a plus b here, it is your m; then only you can do this m minus c right and then again you can do this d minus e here only, right. So, you cannot do that early. So, this is your d and this is your e, this is your d and then this is your, this is again your m and this is your p.

So, you need kind of four clocks; because this m is getting used here, so I cannot do this in parallel to this operation, right. So, as a result, so this will end up having more clocks; but I can understand this is kind of a dependency which can be removed easily, by just renaming this variable, right. So, what I am doing here, I do the same code; instead of using this m, I am just using m one for the next one. So, now, since there is no correlation between this value and this value, now I can execute this part and this part in parallel, right.

(Refer Slide Time: 22:21)

Variable Renaming

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

m = a + b
n = m – c
m = d – e
O = m + p

m = a + b
n = m – c
m1 = d – e
O = m1 + p

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

So, I can move this part from this place to this place, right. So, I can now do this here, right. So, this is my m, right. So, this is my m and this is my, this is my n and this is my o, right. So, this is what is called variable renaming. So, it basically there are some kind of false dependency arise; the write after write dependencies, because of this the one variable getting redefined, right.

So, what I can do is that, I can just re use a different variable and in some sense this is basically SSA right, which is called static single assignment form, right. So, static single assignment form is that, every variable only going to define only once, right. So, it cannot assign two times. So, here this m was defined two times, right. So, which is not SSA.

So, what I can just do? I can just use a different variable for this definition. So, this is kind of static single assignment form and it is also kind of renaming the variable; but

effectively I am doing the SSA here only right and it is actually ensure, it actually removes certain kind of false dependencies, right. So, and it will improve the performance of the high level synthesis result; in the sense that my total schedule time will be less and hence the latency will improve, ok.

But since now I am executing this in parallel, so my number of resource might increase, right. So, number of ALU, FUs and number of registers might increase here; because earlier it was happening here, so it was kind of sequential, I could have used one ALU to do all these four operations, but now I need at least two FUs right to do that. So, it might increase the resource little bit, but the latency will reduce, right. So, that is what is the variable renaming.

(Refer Slide Time: 24:21)

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

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

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

14

So, let me just move on to the next one, next one is the dead code elimination. From the name you can understand that there is some dead code, which is not have no impact on the output right; the code that has no impact on the output is dead code right, it has no use basically, you just remove it, right. So, for example, here you take this example, now you can see here the final output say this is out. So, this is my say out is the final output, ok. And so, this a has no use, right. So, a is never used and this definition is dead code, right.

Because this a is neither output or it has no impact on the final output. So, I can just remove this line and it will be my code, right. So, by doing this live variable analysis and

the program analysis, you can always identify that undisabled code or say dead code and you can actually remove them, right. So, this is what this dead code elimination is all about. And you can understand that it immediately impact on the number of ALU, number of registers and the total latency as well, ok. So, move on, next one is operator strength reduction.

(Refer Slide Time: 25:29)

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.

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

Impact: Resource optimization

So, this is a very important optimizations, is basically I have some computational intensive operator, ok. But in some context, it is a context specific optimizations; I can actually replace that powerful or computation intensive operator by a less powerful or less computational intensive operator, right. So, the good example is that, if you multiply with a constant right; you multiply with a constant, so you need a multiplier. I can replace that multiplier with a shift operation.

Left shift and left shift you remember in the hardware, you do not need any resource right, it is basically you just bit assignment, right. So, suppose you have this say 0 to 6 bits and you want to left shift by 1, right. So, what is basically is this? So, you just put whatever the value is say x, x, x, x, x, x; you are going to put this x, x, x 5 times and you just put a 0 here, right. So, it is basically shift.

So, it is basically the bit assignment; this is coming here, this is coming here, this is coming here, this is coming here, this is coming here and the 0 is coming here. So, the

multiplier is very it is a computation intensive of operation; I just replaced by a shift operation, which is nothing but the bit assignment and its kind of no resource, right.

So, multiply by constant, divide by constant; so those are all is kind of can be replaced by a shift operation. So, for example, x by 8 is nothing but right shift by 3 bits; x into multiplier, so it is a 2 to the power 6, so you just left shift by 6 bits. To a 2 is basically left shift by 1 bit, right. So, multiply 2 is basically left shift by 1, right. So, now, if you have a not 2 to the power of, it is not maybe power of 2. So, what I can do? I can just 2 to the power 4 minus 1, right. So, 15 is basically 2 to the power 4 minus 1.

So, what I can do? I can just left shift by 4 and then I do a minus x, right. So, it is basically x into 2 to the power minus 4; so it is basically 2 to the power 4 into x minus x and this is basically left shift by 4 bits, right. So, I can actually; even if the constant is not power of 2, I can actually readjust and I can actually do it by less expensive operation, ok. So, this is very important operation; because you can actually remove certain kind of expensive operations from your final hardware right, so obviously it impact on the resource.

(Refer Slide Time: 28:05)

Code Motions

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

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

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

So, the other one is that code motion is basically is as I name suggest that, moving the code and one of the things I have already discussed the loop invariant code motion, right. So, loop element code motion is already discussed. So, and in some cases its may not be that loop invariant, the code is not within the loop; but there may be some scenario,

where you actually define a operations here a is equal to something and this is line number 1. And in say line number 30, I am going to use this right say to define say f equal to a plus 5.

So, that means from line number 1 to 30 for this whole period, I have to keep this a into register right; you have to keep this variable live, because this is the lifetime of the variable. But it may be possible that, you can actually move this operation little bit early; say I can actually move this operation to line number 5, right. So, without violating the final output right, without touching the final output.

So, it means now the lifetime become for only 5 units. So, I am not saying this is the 30 cycles, I am trying to say that if there are 30 operation; if you try to schedule them, it need more time right, number of clock cycle will be more. And if it is only 5 operations, number of cycle needed to schedule them will be less, ok. So, as a result, so the lifetime of this variable will reduce if we just move the code. So, this is what is called move, code movement, right.

So, this is called code motion. So, and it is actually you can understand directly that, it basically directly impact on the lifetime of a variable and hence the number of registers; because I do not need to store this a for now 30 clocks or the clock that need to be execute 30 operations, rather I can need to store this in hardware to only for the clocks needed to schedule 5 operation, ok. So, this is reduce the number of registers and loop invariant code motion I have already discussed that it reduce the computation time and resource significantly, ok.

So, this is very important optimizations and this code motion can only not only happen in the straight line of code; you we can actually have some many instances that you move the operations from the branch or you move the operation into the branch, from the branch to after the branch, there are many so many kind of optimizations possible in the coordination context and it is interesting to learn that part, right. But in general code motion is very powerful optimizations, which actually impact both in the number of registers as well as in the computation time, ok.

(Refer Slide Time: 30:46)

Loop Transformations

- Induction variable Strength reduction
- Loop fusion
- Loop inversion
- Loop interchange
- Loop-invariant code motion
- Loop nest optimization
- Loop unrolling
- Loop splitting
- Loop unswitching
- Software pipelining
- loop tiling

- Extremely important in HLS context
- Improves locality of references
- Reduce memory access
- Improve pipeling performance

Needs to be applied manually

17

So, these are the optimizations which I have talked about is the, the optimization that is already enabled inside the GCC compiler or C compilers, ok. And you can actually enable or disable them based on your needs, ok. And there are another set of optimization, which is called loop transformations. So, it is basically optimizing the loop and if you look into any compiler book, you might get at least fifty such loop optimizations, right. So, I just list them some of them here; but in general this number of loop transformation is huge and most of the compiler does not do it automatically, ok.

So, but they have a very significant impact in the high level synthesis results, ok. You can understand that say suppose there are two loops which has the same bound and actually number of iterations is same. So, what I can do? I can just merge this two loop into one, ok. And as a result, so earlier it was time say if the one loop was taking n cycle; so the previous code was taking 2 n cycles, now it would going to take much much less than 2 n cycles right, because you have the operations same, but you can parallelize them, right.

So, in best case you can actually schedule the both the loops in now in n cycle, ok. So, that is what I just give a simple example, but there are many such optimizations, ok. And some of the optimizations actually done by the high level synthesis tool itself; like loop unrolling or say loop pipelining, right. So, this already is implemented in high level

synthesis, right. So, we have already discussed those things in when we are discussing the loops.

But there are many such optimization, specifically this loop tiling and loop interchange; these are very important optimization in the high level synthesis context, probably you have to do it manually, right. So, there is no kind of automations there and if you have certain applications, where it is kind of kind of become inevitable to apply this kind of optimization to get a better results, ok. So, what I am trying to point out that, we should understand this loop optimizations very clearly and their impact on the higher level synthesis results.

There are various studies happen, so interested reader can actually I mean Google it and identify that, the impact of various kind of loop term optimizations in high level synthesis results. And based on your application, you can actually choose which one actually best fit to you, ok. So, what I am going to do in today's class; instead of discussing all, because it will take multiple classes. So, I am going to take an example and I am going to see how this two optimizations can get actually make an impact on that particular application, ok.

(Refer Slide Time: 33:27)

Loop Tiling and Loop Interchange

- locality optimization are increased by interchange and tiling

```
for(i=0; i<Bi; i++){
  for(j=0; j<Bj; j++){
    for(k=0; k<Bk; k++){
      C[i][j] += A[i][k] * B[k][j];
```

```
for(ii=0; ii<Bi; ii+=Ti){
  for(jj=0; jj<Bj; jj+=Tj){
    for(kk=0; kk<Bk; kk+=Tk){
      for(i=ii; i<ii+Ti; i++){
        for(j=jj; j<jj+Tj; j++){
          for(k=kk; k<kk+Tk; k++){
            C[i][j] += A[i][k] * B[k][j];
```

So, let us be take that matrix multiplication, right. So, it is very common well known applications. So, are you actually multiply two array, say two matrix say A and B; this is my A and this is B, you are multiplying and storing the resultant C, ok. So, this is what

where is very well known code; you basically have a three nested loop and what is actually happening here. So, to compute this value what I am going to do; I am going to take the first row and of this and I am going to take the first column of this and I am going to multiply, right.

So, this into this will give you me this code, right. So, that is what is happening here, right. So, if you see here i is fixed, k is varying. So, it is basically you are going this way and here k is varying this way, so your row is varying. So, this is right. So, you actually take multiple take this element and this element multiply, right. So, then you take this element, sorry you take this element and this element and multiply and then add this, right. So, this into this and getting added with this, right. So, you say suppose this is your x and this is say y.

So, you multiply x into y, then you multiply. So, this is a and this is b. So, you multiply a into b and store; then you take this element and this element, again multiply and add, right. So, this way you are going to get only this element right and you have to do this for both adding all row and all column, right. So, you can think about i is row and j is column and this is actually calculating one element, right. So, this is calculating one element of C and this is you have to identify for all element.

So, this is something is given to you and you can understand here is that, there are lot of optimizations, right. So, basically for one value if say, I if I just assume this is n cross n and this is n cross n; so say let us let say the dimension is same. So, for one value, you have to do how many multiplication? You have to do mult n multiplication plus n addition, right. So, for one value, you have to do n mult operation plus n add, right. This is for one value and how many values are there? There are n square values, right. So, it is basically n square into this.

(Refer Slide Time: 35:54)

So, there are n square n cube number of multiplication, n cube multiplication operation plus n cube addition operation is getting performed here. So, there are lot of optimizations happening and also you can see here that this if this A, B and C is a big matrix; say for smaller matrix, they might be mapped to registers. But if this they are the big matrix, what is going to happen; they are going to store in block ram right and there are lot of memory access also.

So, the first point is that lot of computations, which we already understood; the next part is the memory access. To calculate one value, you are accessing n data of A and n data B, right. So, that means, 2 n axis of A and B and similarly you are actually accessing that C i multiple times also. So, it is kind of kind of 3 n access for one data and you are doing for n square. So, there are order of n cube memory access.

So, n comma n cube order of n cube multiplication n cube order of n cube addition order of n cube memory access. And another important point to be noted here is; so once you calculate this value, you take this row and you take this column multiply. To calculate this value, I am going to take this row and I am going to take the next column; so that means this particular row is getting reused, right.

So, this in fact this particular row is going to be used for all element of this row, right. So, I am going to multiply this with this, I am going to this going to get this; I am going to multiply this with second column, I am going to get the second element of the first

row, first row third column third element of the first row, first row fourth column fourth element of the matrix and so on. So, what I have understood here is that, this particular value is getting going to repeat or access multiple times, right. So, and it is getting reused, ok.

So, this is also another important point here, ok. So, now, the question is that, since I am actually accessing this many times, ok. So, I may try to do something, such that I the number of time I am going to access the block ram should reduce, right. So, if you access it every time you calculate this; it will take time, because the memory read takes time. But my objective should be, when I am going to implement or realize hardware for this kind of a problem; if that particular value is getting access many times, I try to minimize the access, right.

So, that I am going to read it once, I am going to use it multiple times ok, that is the idea, ok. So, the way I am going to do this things is that, the way I am going to calculate this equation; I am not going to, my objective will be to reduce the memory access, so that I do not have I mean, so that memory access time get reduced, ok. So, one another you show; so there are this is the second point that you under, second point is the memory access right, memory access.

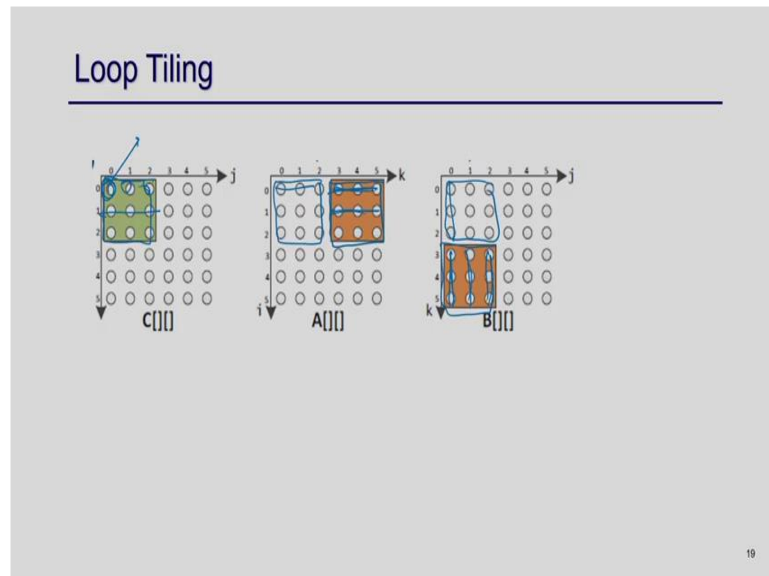
The third point which you should note here, which actually enable certain things that; although I am actually calculating this value by multiplying, basically taking this row and taking this column and multiply element wise the and in this particular loop, you are actually doing the whole thing and after the one iteration of this for loop, the I will get the final value of this. But it is not necessary to do this whole computation at a time; what I am try to mean is that, say I just take these three element and multiply with these three element and I will store the value here. So, that it is not completed yet, right.

So, after some time, so then I am going to take say this say this and I am going to multiply these three; I am going to put it there, after sometime I am going to take this part of the array, right. So, in the first part I am going to take this part of the array and I am just partially calculating the value of C i j, I am storing that result here. So, when I am going to take this part of the matrix, then I am going to multiply this with this again, right.

So, I am going to take this part of the array. So, I am going to take multiply this with this and I am going to add that with the previously computed value, because this is all getting added, right. So, finally, it does not matter whether I have calculate all this n multiplication addition at the same time or say n by 4, n by 4, n by 4, n by 4. So, I just, I can do that right; finally once the my whole loop ends, my matrix should be correct, right.

So, I should not bother about what is the intermediate value of those C i C j. So, that is what is called loop tiling, right. So, what I am doing here is, I actually instead of taking the whole matrix at a time, because then that reuse does not happen.

(Refer Slide Time: 40:57)



So, what I am going to do; I can actually break my matrix into small small component, right. So, I can take a small part of the array B, A and say this part of this and what I am going to do? Once I going to multiply; so multiply this with this, I am going to add those element into this notation, right.

So, it is basically if I multiply this with this and then I am going to get some part of this matrix right, so corresponding to this part. When I am going to take this part and this part and I will multiply this with this and if I add that with the previously computed value, that will give you the correct value right and that is going to happen right eventually. So, the idea is that, this is what is called loop tiling that; instead of taking the complete array

at a time, you basically take a small part of the a tile of that matrix and then you multiply and do the multiplication there for my all columns, right.

So, I am going to do this say basically with this with this, with this, with this, with this, with this and I am going to get this three value; immediately I am going to do this, with this, this with this and this with this, I am going to get this row. So, this way, so when I am going to take this window, I am going to finish all the computation that this particular window involves, ok. So, that will give me the partial computation of that matrix for this window only; then I am going to take the other two window and I am going to complete all the operation that involved in that window, ok.

So, this way what I am doing, actually I am doing improving the locality of references; I am taking a window and where I am going to use the same data multiple times what I can do, I can store that window in a buffer and I do not have to read it multiple times, right. So, that I am going to do it here and that loop tiling gives you. Similarly this loop interchange what it does is, basically you can actually exchange this i j k in any order; finally once this loop will end, you will get the correct value, right.

So, the order of operations, order operation will change, if you change this i j k. So, you can actually make it say k, j and sorry k this is k, j and i or say j i and k or say i k and j, right. So, any order if you execute this three for loop; result after completions of this the whole loop, your C i j will matrix will be the same. It does not matter which order you are calculating; because that order change of order actually only what is the intermediate value may not be correct, I mean it may not be the same.

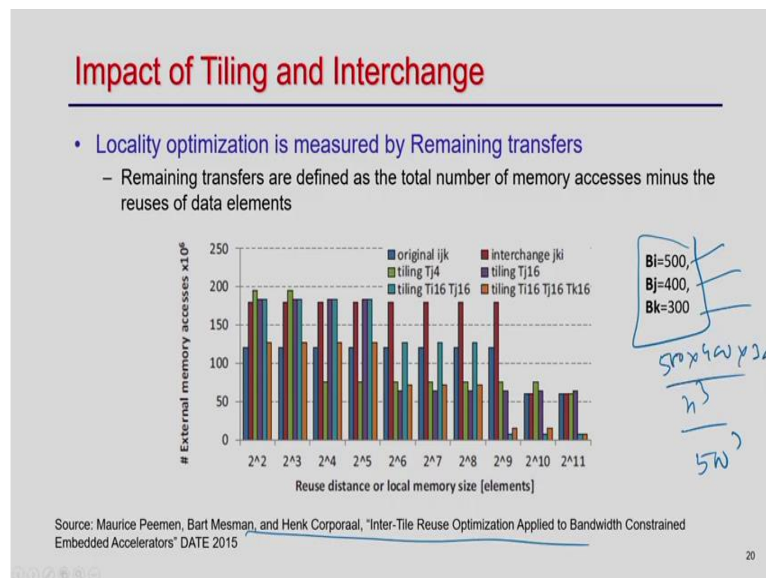
But it does not matter, because I am bother about the final outcome matrix multiplex. So, so from this example what I conclude that, doing loop tiling instead of taking the whole column and whole row at a time; take a small window, identify the all the sub value which involves those window, complete the executions and then you move to the next window. So, this is how I am going to do that and this is what is this code all about, right. So, here this is the window for the row i; this is the window for the row j and this is the window for the row k, right.

So, this is what is happening here. So, basically to be precise, I this is the window for i, this is for window for j and this is the computation happening for this window, right. So, you cannot have a window for k, because you have two dimensional matrix. So, it is

basically this basically says that for all values, you identify window by specifying that some i j value and then for that window, you calculate the partially calculate the multiplications for that window using this k 1, right. So, this is what is loop tiling and loop interchange I have already talked about.

So, there are many such combinations right; the first of all what is the window size, right. So, window size is important; you can take a size of 4, you can size of 8, you can type of size of 16 and so on. So, your window size can differ and this order can differ, right.

(Refer Slide Time: 44:55)



So, now I just put you some results which I have taken from this paper basically, they had done a very interesting study. So, they take this matrix multiplication, they take a very big array. So, their i is basically 500, j is 400, and k is 600. Now, we can understand that n cube is how much, right. So, n is say, so 500 time of 3, right. So, basically n is basically 500 into 400 into 300; so lot of multiplication, lot of addition, lot of memory access, right. And then what they have done is very interesting stuff. What they have done is, they basically do this loop interchange, so they change the loop order, right.

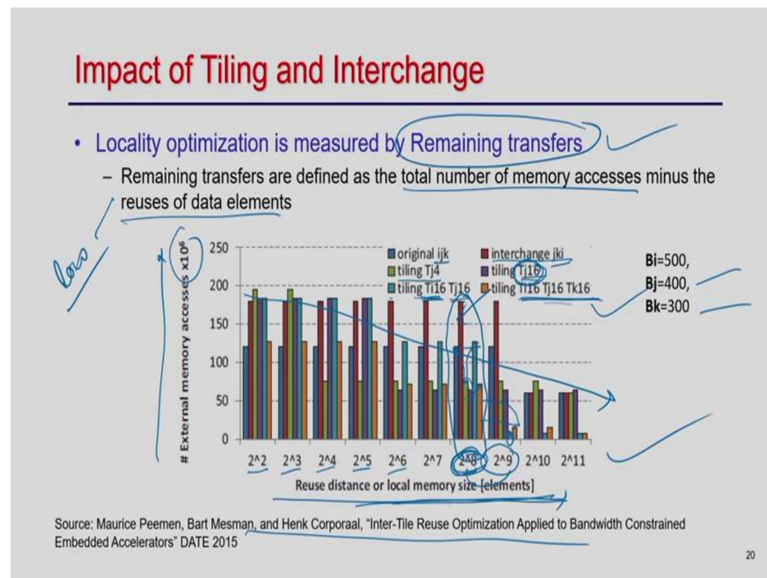
So, original order was i j k and they take another say j k i and also they do the tiling and in the tiling they take different different combinations; as I mentioned that you can tile only i, you can tile only j, you can tile both i and j or you can tile by window of 4, you can tile window of 8 or 16 and so on, right. So, they take different such combinations; one of the combination the green is basically only tile the j with 4, there is the blue is the

tiling i the i and j both with a window of size 16, this is j with 16 and this is tiling i j k all with 16, ok.

And then they actually measured the improvement using the remaining transfer, this is very interesting terminology. What is remaining transfers? It says that are defined as the total number of memory access minus the use of the data element right, so that means you need n cube order of memory access. And if you take a window, I am going to reuse certain kind of memory content, right.

So, if I remove this reuse part from the total access, the rest of the things is the again the memory access, extra memory access which is not getting reused.

(Refer Slide Time: 46:48).



So, if this value is low; that means there are lot of reuse happening, right. So, your reuse is very high and if this value is remaining transfer is high; that means this kind of optimization is not improving your locality of preferences, ok. So, this is the parameter which they have calculated and put it in the table. And the another the x that x axis actually give you the window size, right.

So, in the harder, so we can think about the; in the processor there is a cache right, so cache size is fixed. So, you and in the hardware, if you think about the high level synthesis; we can think about that, I am going to take as part of the matrix, I am going to

put in some registers locally. And that windows that window size is something say 4; you can only can store 4 element right or 2 to the 3 8 or 2 to the power 4, 16 or 32 or 64.

So, they are saying that, if I just have a capability to in the HLS that, I can only scape 4 element from that matrix and stored in the HLS or 8 element from the memory and stored in there from that window the tiling window and so on. So, this is my the local memory size and this is the kind of 10 the number into 10 to the power 6 number of memory access, external memory access right. So, that the remaining transfer right, they have putting it here.

So, now you can see here that the number of remaining transfer is not same for all cases, right. So, you just take one scenario, I am not going to explain everything. So, let us say you take this versus this, right. So, you take this. So, you if you take this one, so the first one is the i j k normal code the left hand side what I have taken. So, it is kind of taking some 120 into 10 to the power 6 number of memory access, right. So, this one; but if you just do this inter change j k, your memory access actually improves a lot, right.

So, that means, sorry increase a lot. So, it does not give you any benefit when your local memory size is 2 to the power 8, right. So, whereas, this green the tiling j gives a good benefit, right. So, from the original one it reduce a lot; so that means your locality of reference increase, it increase further when you use the tile of 16, right. So, it is understood; but if you just do this tiling both i and j, it does not improve, so that is very interesting. So, just doing the tiling of j, gives you good benefit; but whereas if you just do both i and j, it does not give you the benefit, right.

It actually increase the results and is actually worse than the original one also, right. And the last one is that if you tile all, your performance is also not as much as just doing tiling 6. So, this actually gives you the good measure that blindly doing the tiling of i j k, may not give you the best results; whereas just doing the tiling of size 16, give the best results for this when the window size is 16. And you can understand that though if you increase the local memory size, the number of locality of reference increases and hence the external memory access get reduced and your performance getting improved, right.

So, from there you can gradually it getting reduced, that you can understand. But and basically say if an increase from close to this, you can see the performance get hugely improved, right. So, for example, when you have this 2 to the power 8, 2 256 your local

memory tiling all i j 6 does not give you good benefit right; it does it is worse than actually, if you just only doing tilings by j by 16. But if you take a window 2 to the 9, you can see that it is give a huge improvement, right.

So, your improvement when you just tile all, you have a huge improvement and of the best result you are going to get, if you just tiles i and j, right. So, that this is give you the best results. So, I bring this result in our discussion is that, your this number of tiling possibilities is many; tile size whether I am going to tile all parameters or only one parameters, the you can actually interchange the loops that is also another parameter and also another parameter is the local buffer size. So, among all these three, your performance may be impacted severely, right.

So, your performance can change severely based on these three parameter and say based on your application, probably you have to choose the right combination, right. So, that is what I am try to highlight with this example that, this loop tiling and loop interchange can give a huge benefit if you choose the parameter correctly. And this you remember you have to do manually, right. So, there is no compiler which will give you say; if you do this tiling only j, it will you get the for best performance or not, right.

So, this is something where this human expertise coming to picture and basically you have to choose the right combination of this optimizations to get the efficient hardware; it is not that you choose all, choosing all would not would not give you the good performance always right, that I have already discussed. So, this actually highlight this about the loop, loop interchange and loop tiling; but there are many many loop transformations and I just request you if you are interested, you can actually go into dig deep.

And you actually can understand that this optimizations can have a great impact on the efficient the hardware that you generate through high level synthesis, ok. So, with this I conclude today’s class. So, as a summary what I understood here is that, we should know what are the optimization that getting applied in the front end, the compilers and based on my application, probably I have to either use or disable them based on my requirement and specifically the loop transformation actually gives a huge benefit in terms of the performance of my design.

And we should must, we must aware the impact of those optimizations and we should apply them during efficient hardware generation. And specifically for loop transformation, it is not that apply all also always give you the good performance; probably you have to take the right balance of the optimization to get the best hardware, ok. So, with this I conclude today's class.

Thank you.