

Design Verification and test of Digital VLSI Designs
Prof.Dr. Santosh Biswas
Prof. Dr. Jatindra Kumar Deka
Department of Computer Science and Engineering
Indian Institute of Technology, Guwahati

Module - 1
Introduction
Lecture - 3
Transformations for High Level Synthesis

So, in the first 2 lectures, what we have seen? In the first lecture, we have seen that there is standard VLSI flow; I mean design flow for the digital VLSI circuit. So, we start of its specification. Then, we go for high level synthesis. Then, we go for get level synthesis and so on.

Then, we told that the first module of the course would be mainly discussing about the high level synthesis part, in which case there will be a specification following. Then, we will get what you say the architecture design, a block block block level design of the circuit.

Now, in the in the next lecture, what you have seen that we also discuss that the the code main emphasis of the course will be to lead you to algorithms and tools, which will help the designer to automate this design or get some, I mean automated implementation of this specification by different cad tools and algorithms. So, that is my main focus of output.

So, in the next lecture, what we have seen? We have seen that in the next lecture that whenever you have to process some some some inputs or some specification by algorithm or procedure, things have to be represented in a formal way. So, I mean in the case of circuits, if there are specifications, so we have write to this specification in a formal language. So, it becomes it becomes unambiguous. Everything can be normalized. It enhances to some standard. So, with a discussion about that, there are several languages to do that. So, you can do it using verilog, VHDL system etcetera. Verilog is a well-known, I mean approach is used. So, we just gave some idea about the verilog code.

Then, we have seen that CDFG that is control and data flow graph is a very well-known formal model, which can capture your verilog designs. It can be represented in a formal

state state transition type of a diagram using CDFG that is control and data flow graph. So, you have already, we have seen in the last 2 lectures that you have given a specification. Now, from the specification, you want go for a high level design. For that, what we have to do? We have to first represent it using a standard format or standard or for formal model, so that there is no ambiguity and that is it can be efficiently processed by some algorithm and so forth.

Then, next what so, then we have seen that verilog is a very good language or is a hardware definition language, in which you can represent your specification and form. We can convert it into a control and data flow graph, which is a formal representation of the specification. Then, we can do lot of processing on that. So, what will be the looking today looking will be today? We will be looking at transformations for high level synthesis.

So, what is that idea? So, idea is that some some these specifications, if you see are written manually. So, after that, all the processing starts with automated cad tools, but the idea is that specifications are derived from requirements. You want to level a calculator. So, we write a specification that I need adder, subtractor, multiplier and so forth. It is a business calculator.

Then, you need logarithmic exponential factorial etcetera. So, based on requirements, you will write down your own specification. So, that is the manual part of it. Then, specification can be represented in in some kind of automated, I mean so formal model like this CDFG etcetera. So, it is a not very automated process. It is much automated process. I mean you can map your specifications to any formal model.

So, there may not be much error in that, but you can understand that when when these specifications are converted into RTL, when these specifications are converted into verilog or VHDL that you are representing your specification, they were in general English or general notion in your mind.

Now, your converting it into some verilog or VHDL and which you want to converted into CDFG, go ahead in the design flow. So, it is step where you are writing the code. So, represent your specification in a fully a manual process. We develop this specification based on requirement and our internal analysis. So, there can be lot of errors or we can say inefficiency etcetera in the coding. So, in this case, today's lecture,

we will see what transfer machines are. We can do so that your code that is your specification becomes more precise or it can be become more efficiently in terms of hardware etcetera.

So, I mean again, I was meaning again reemphasize that again in a c goal. So, if you are going to write for a given specification, if you are going to write c code, again if there is some inefficiency etcetera, which is discovered later part, when the product is in the market. So, always you can go for a patch or always you can change your code a bit, so that it become efficient and all. In case of hardware, once the specification has been developed, we have written the RTL that is verilog for that. It has been say synthesized and fabricated sent to the market.

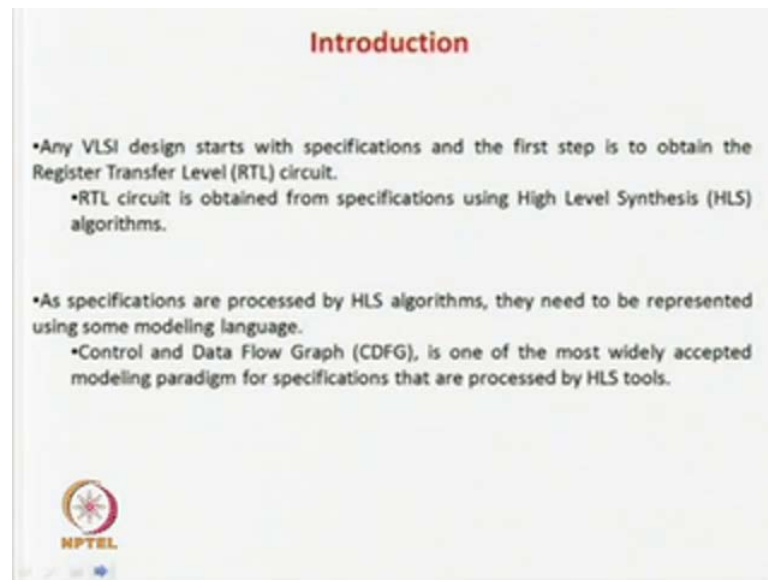
So, even if you find that there are some kind of in efficiencies like this captured, shear can be could have been some optimization etcetera, they cannot be again redone. This is because hardware is once fabricated. Then, it is sold in the market. So, there is no step in the redoing anything over here. So, that is why, here you should be very very precise on what we are doing.

We should not be living behind. We should not think that there can be any if there is any possibility of any optimization at any point of time, then we should be we should be able to take care saying that that whatever optimization could have been possible or whatever error was there, we have to debug all. I mean as much as possible, we have to optimize or code as much as possible before we go for the next level in the design flow.

So, that design which comes out is coming or only once at once and for all. After that, there is no modification possible. After the fabrication has been done, so what do we do? What we have to do would be very sure that the design is very much optimize, the error free and so forth. So, in the first step when that is the manual path, that is the most risky part of the design. This is because all other steps are more or less I mean automated by cad tool.

So, there is not much risk or problem of errors as in the case of when you are writing a specification and computing in mind. What you want to do? Writing a verilog or VHDL code for that, there is the place where the maximum errors can be minimized because that is a purely manual process.

(Refer Slide Time: 05:37)



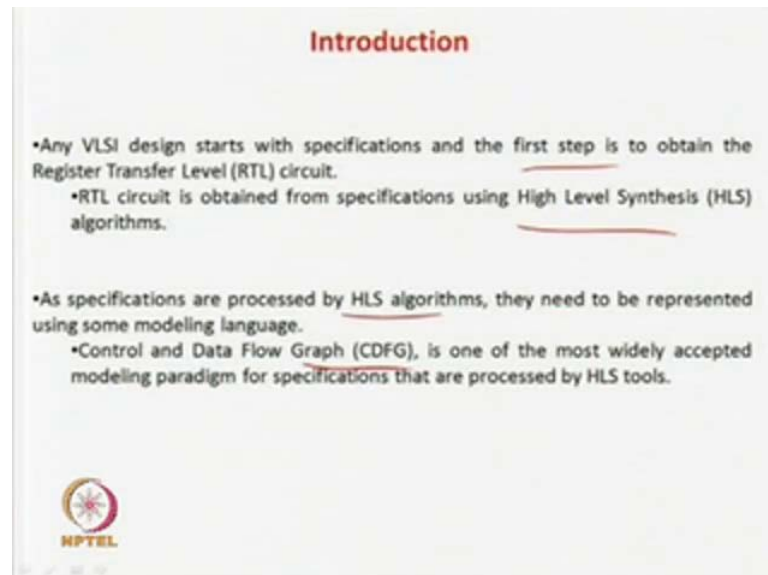
So, today, we will see what the standard measures of debugging that is c are. As in the case of c, you run your verilog code with different type of test benches that is the input. Hence, whether these operate correct or not; in the second module of the course, we will see that there are formal verification methods. Also, in this case, you need not apply an each thing, but you can formally or mathematically verify that your design is proper. Your design matches this specification. There is a formal way of doing it, but there also some other steps that we will see today like there optimization steps you should not quality error. So, whatever today we will see are not errors, but you can do lots of optimization.

Let me give an example. Many of the examples we elaborated in the lecture. So like say for example, if you have 3 expressions like a equal to b plus c, a equal to b plus c and a equal to b plus c at different parts of the code, if I know that a and b values do not change in between this codes and in between this instructions. So, you can always say temporary variable t 1, which is equal to a plus b.

Whenever you want to assign a plus b to some variable, we did not compute a plus b. Once again, you just apply t, so this is one kind of an optimization. So, this is not an error. It is a performance optimization. In this case, you will be having only one 1 instead of 3 adders. If you have being 3 repeated operations, so elimination of redundancy is an


optimization. So, in this lecture, we will be mainly looking at part optimizations are possible in the code, which is written by a human designer.

(Refer Slide Time: 06:54)



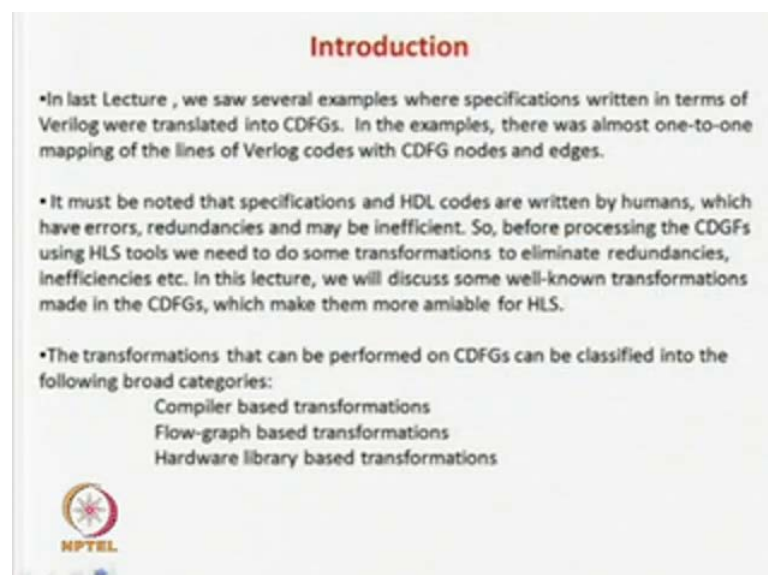
Introduction

- Any VLSI design starts with specifications and the first step is to obtain the Register Transfer Level (RTL) circuit.
 - RTL circuit is obtained from specifications using High Level Synthesis (HLS) algorithms.
- As specifications are processed by HLS algorithms, they need to be represented using some modeling language.
 - Control and Data Flow Graph (CDFG), is one of the most widely accepted modeling paradigm for specifications that are processed by HLS tools.




So, the first introduction you already said that first step is high level synthesis. So, in high level synthesis, we represent is a control and data flow graph.

(Refer Slide Time: 07:02)



Introduction

- In last Lecture , we saw several examples where specifications written in terms of Verilog were translated into CDFGs. In the examples, there was almost one-to-one mapping of the lines of Verilog codes with CDFG nodes and edges.
- It must be noted that specifications and HDL codes are written by humans, which have errors, redundancies and may be inefficient. So, before processing the CDFGs using HLS tools we need to do some transformations to eliminate redundancies, inefficiencies etc. In this lecture, we will discuss some well-known transformations made in the CDFGs, which make them more amiable for HLS.
- The transformations that can be performed on CDFGs can be classified into the following broad categories:
 - Compiler based transformations
 - Flow-graph based transformations
 - Hardware library based transformations



This is because this is one of the most widely used models for that one. Then, we emphasize that that says I mean verilog codes were converted into CDFGs. So, that is an automated flow, but the verilog codes are developed from the human specification.

So, the way you are writing your verilog code will decide on the view that is the input to your all the cad 2 algorithms or the formal algorithms. The input is the verilog code. So, if there is no optimization in there are, if you mean out, if you have some redundancy is there all, so your circuit will be analyzer larger. So, we will be mainly emphasizing on the verilog codes and CDFG models. So, if there are some redundancy or some optimization is possible etcetera, so that is actually called transformation.

So, we will transform or verilog codes or we will transform CDFGs because CDFG is a formal model. So, it is very easy to see them and do go for the transformations. So, we will see that this well-known transformation, if you are applying to CDFGs, so our code, we should not call it bug free. This is because we are not catching bug catch, hold bug.

We are actually thinking and code will be optimized. It will take less hardware. We will be more efficient over here. So, what will be one of the different optimization or transformation? We will see we will see compiler driven transformation flow graph transformation and hardware library based transformations. So, we will see one by one. So, let us start with the compiler based transformation. So, what is the compiler based transformations?

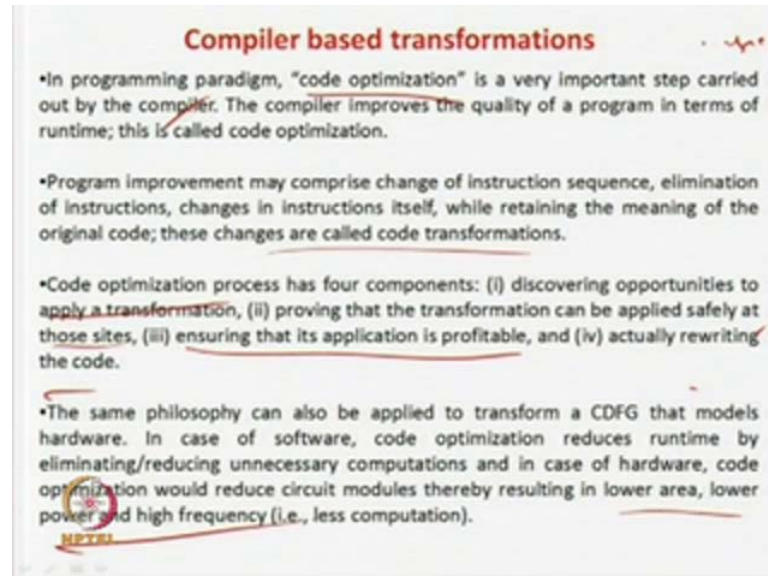
So, if you are grid about the compiler design in your undergraduate class, so you know that our compiler also tries to optimize your codes in the base possible manner. Again, the difference is that even if you missed your optimization in the c language once, then you can again redo it. This is because it is nothing hardcoded or fees that one times.

So, even if will be customer says that the null optimized, you can again go for optimization. But, still we do optimization, lot of optimization even in the software level. This is because you want to give a good product to the customer there. Then, every time getting complains and repairing them or modify optimizing them. In case of hardware, optimization is more, much more severe problem and more important to do. This is because once a hardware is fabricated these, no chance anymore to do or undo anything and optimize it.

So, what is the compiler base optimization? In compiler base optimization, what we try to do is we have already read that we eliminated redundant elimination. Redundant computations are eliminated. Partial redundant computations are eliminated. We go for dead code elimination etcetera, so that your code becomes very efficient in that. Now,

the same structure will also be applied at the hardware level. So, that is actually the compiler based transformations.

(Refer Slide Time: 09:12)



So, there same concept if you write this, so we will do so. As I already told this code optimize in compiler base optimization, what we do? The code optimization is a very important step by compiler because by doing this the code, which is written by the human human designer or human coder will be optimized in terms of performance, one very important actually. So, what is the idea?

So, what are transformations in this case that is you transforming code a to code b by doing some optimization or doing some redundant elimination etcetera? But, you should be very careful that whenever doing the transformation, there should not be any changes function ready of the code. This is because you are doing only optimization. You are not debugging. You are not trying to say when operations you are not doing anything, on that round only, we are doing try to do some performance.

So, you are in the other way, you are actually trying to optimize the code. It is assured in optimized way or your hardware requirement is less in case of VLSI. The input output behavior should remain the same. So, how can you go for code optimization? First, you have to find out what are the, where is the opportunity to discover code transformation like if you are redundant expressions, so they are actually opportunities to do that. Then, you have to prove that the transformation can be applied safely at those sizes. Now, what

is that say for example, if you have in one step, $a = b + c$ after that a lot of codes are there. Again, here you will have $a = b + c$ you have $d = b + c$.

Then, what you can say that $a = b$, $a = b + c$. This you can also store in a temporary variable t . Now, instead of d , if you write $b + c$, you can write $d = t$ that is one place where you can apply the transformation. If you want to prove the transformation is safe that means you have to say that this b and c , they are not overwritten. At any point from this part of $a = b + c$ to this part of the code, if it is done, then b and c without change in the path. If you want again, write back the old value of $b + c = d$ that will be an error which will change the behavior of your code.

But, if you say that here we applied $a = b + c$ after that $b + c$ has never been changed throughout this code from this instruction. So, this instruction, very safely you can apply the value of $a + b$, which is equal to stored in a temporary variability to d .

So, you have to prove that is safe. Then, ensure, you have to ensure that the application is profitable like in this case $a = b + c$ were eliminating one addition operation here in case of hardware. This is because eliminating 1 adder, so that is the very good profit. This is because saving some computation, and then actually you have to rewrite the code. So, in that case, you have to reduce $a = t = b + c$ and then here $d = t$ something like this.

So, the basic idea is that first we have to find out where optimization can be applied. Then, we have to prove that the optimizations really do not harm. They preserve the equivalence level. That is what I want. What I mean to say is that there is no behavioral change at that level. Then, you have to ensure that the application is profitable that is you are going you can do lot of transformations, but they are not going to profitable. They are going to harm view may be. So, in that case, you do not go further. If you find out that you can say some computations and something, so you can go about that. Finally, you have to rewrite the code.

So, that is actually the idea of the code transformation using compile based optimization. So, what will we achieve in hardware? You will achieve in hardware lower area, lower power, higher frequency, less computation. In case of software, what do we achieve by this? You get it will take I mean the code will be executed faster. It will take less time in the ram and so forth.

(Refer Slide Time: 12:17)

Loop Invariant computations and code hoisting

An expression evaluated inside a loop that uses operands whose values do not change from iteration to iteration is called a loop invariant computation. So the evaluation of a loop invariant expression can be moved out of the loop.

If the loop's body executes more than once, this should reduce the number of times the expression is evaluated. Further, the scheme also speeds up overall system clock because time required to evaluate a loop decreases (as hardware for computing the invariant code is removed out of the loop).

```
module CFG_tr_example(A,B,C,D,E);
input [3:0] B,C,D,E;
reg [7:0] A,F;
output [7:0] A,F;
initial begin A = B * C + D; end
while (A > 0) begin
  A = A - 1;
  F = E + B;
end
endmodule
```

Handwritten annotations on the slide include:

- A red box around the code line `F = E + B;` with a question mark `?` next to it.
- A red box around the code line `while (A > 0) begin` with `(A > 0)` written next to it.
- Red text `a = b * c` and `C = 1` written to the right of the code.
- A red arrow pointing from the `F = E + B;` line to the `while` loop.

So, basically the philosophy is said the entire performance requirement is different. Now, we will see one by one what are the compilers, it remain optimizations. So, first is actually loop invariant computation and code hoisting. So, what is the idea? So, here the idea is if you have say a loop, some loop is executing every time, and then if there is some variable like say $A = B + C$. The value of $A = B + C$ does not get changed in this loop. So, $B = \text{something}$ and here $C = \text{something}$ happens inside this loop B where you B plus. It does not get affected.

Then, what is that you are unnecessarily going in the loop every time you are computing $B + C$? It is assigned to the value of A , which remains same. So, the idea is that if B and C are not changing inside the loop, then why wants to keep $B + C$ equal to inside the loop? What you can do? You can keep the loop $B = A = B + C$ over here or just after the loop, you can assign that $A = B + C$ depending or some situations.

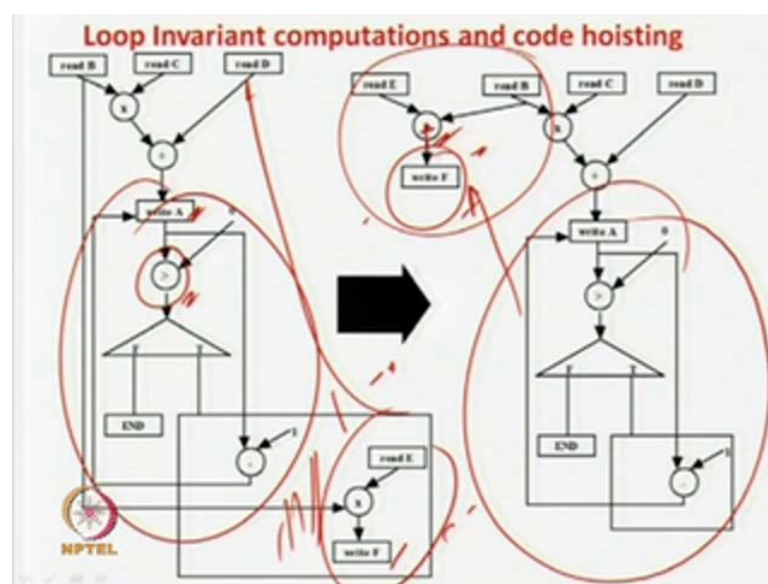
So, what is the idea here? So, the idea here is that if there are some variables or some operations, which is in a loop, but the loop, at deteriorations of the loop are not effecting the computation of some variables or some operations in that, then why unnecessary one to repeat those computation the loop n number of times as mean times the loop is executing. Better, you bring that stuff out of the loop, so that the number of times the loop is operated is saved by the computation, which is being repeated, but having no effect.

Let us see the example. So, whatever I am saying is actually written over in these steps. So, let us see the verilog code. So, in the verilog code, we say the initially A equal to B star C plus C that is the operation. Then, we are saying that y a is greater than 0, A equal to A minus G and you say that F equal to E plus B. So, what happens?

So, these are loop, which were actually executing a number of times. What is A is equal to B plus B star C plus C? Another is a computation over here. So, what is saying that F equal to E plus B? So, we can easily observe that there may be some other computations here also. If you just look at this computation F equal to E plus B, it is nothing to do or nothing is effecting in their loop, which is executing a number of times. So, why unnecessarily I want to put F equal to E plus B? Unnecessarily, I want to execute it n number of times or I want to compute n number of time n number of times inside.

So, what I can do is that I can either bring this computation down here. So, I can bring this computation down here. I can bring this computation out of the loop. So, in this case, what happens? Evaluate the computed only once. The loop will execute and come out with the required number of operations. We will be saving n a minus 1 number of types operating the solution. That is the basic philosophy. Now, we will see this. So, we model this whole stuff in a control and data flow graph. So, you can see that first we had A equal to B plus C star D.

(Refer Slide Time: 14:49)



So, equal to $b + c * d$, so this you can see $B * C + D$. So, $B * C + D$, so this is equal to $B * C$. This is D . So, you are getting this operation. So, that is equal to A . So, there is the loop. So, you can see that is a computation computational loop. So, if this is false, you start. If it is true, then you have to decrement the value of A . Again, you have to go back. Inside this loop, there is one operation, which is actually saying that $A = F$ equal to A . This is the plus. So, $A * F$ equal to $A * E$ kind of a thing. This is E , you can say.

So, what I am meaning to say is that in this computation $A B C D$ are there. In this in this loop, it is executing a number of times. So, in this case, what is saying I am computing? F which is equal to nothing but A equal to F equal to $A + E$ whatever. So, we think is that this computation has no affect in this loop? This you can very easily find out because E is being read and F is being written. Nothing of this is being used to compute this when the case that this computation is affecting this case or some other part in the loop. So, you can figure out that this part is very, very important to computing the loop.

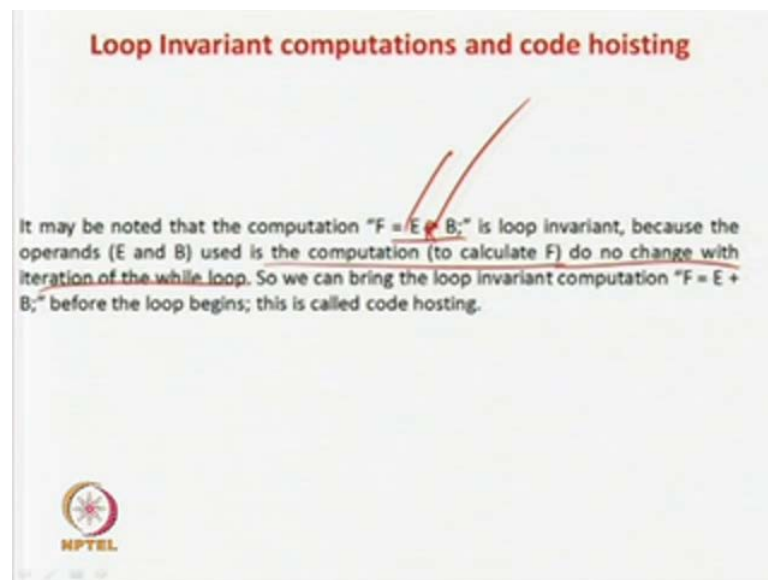
But, if you can just see this inside this loop, inside this loop, this is your loop part of it. This is the computation in the loop. So, this is not affecting this loop anyway. So, what we can do is that we can bring it out and bring it in the top; because then we will be saving the amount. They can be some other computation, which is not so near for simplicity. They can be some other computations in the loop, which may be affecting this way.

So, you can have a computation like some G is equal to A star one kind of a thing. So, every time A is being decremented, so the value of G is affected by the loop. So, you can also say that G equal to $A + G$ equal; also G equal to $A * G + A$. This also you can write.

So, what is happening every time the value of A is added to itself? So, it is equal to $A + A - 1 + A - 2$ dot, dot, dot. So, this computation will be affected in the loop, so that you cannot bring out of the loop that is very obvious. Here, if you see this computation is not of having any affect in the loop, so this you can bring it out over here So, that is what this is transformation that has been done over here. If you can see, so they have brought this transformation over here. So, $E = F$ equal to $E + B$ or $E * B$, whatever may be the condition. So, this is you writing it over here and this is your loop.

So, inside this loop, you are not doing the computation. So, the loop is executing a number of times. So, by bringing or a hosting this code over here, so what you are achieving? You are saving the number of additions, number of additions here by A minus 1 number of times.

(Refer Slide Time: 17:30)



So, A this loop is as given, A times. So, once you have to at least here, so what is the amount of saving? A minus 1 is the amount of time you have seen. So, that is what I have written here. So, we can note that the computation this one or I mean if you say this plus or if we in this case, if the star, so whatever is the case, we can also see the star is loop invariant.

This is because the operands E and B are used in the computation to calculate. They do not change with the iteration of the loop. That is value of E and B does not change in the loop. So, unless why you want to put it 1, so this one we will start this, 1 you can put it before the loop. So, this computation before we start of the loop actually; this is called code hoisting. So, what is the basic idea?

(Refer Slide Time: 18:01)

Loop Invariant computations and code hoisting

The conditions required to determine loop invariant computations are as follows.

Any computation (or expression) can be written as
"assigned variable = operand1 OPERATOR operand2... OPERATOR operandn"

In terms of a CFG, in the computation, "assigned variable" and "operand1" through "operandn" correspond to storage nodes while "OPERATOR" corresponds to operational nodes.

A computation "assigned variable = (operand1, operand2, ..., operandn)" that is inside a loop, is loop invariant if

1. If all the operands are constants
2. All the computations that assign values to the operands are located outside the loop
3. All the computations that assign values to the operands are themselves loop invariant

In the above example (Figure 1), "F = E + B;" is loop invariant because computations that assign values to the operands (F and B) are located outside the loop (taken from inputs).

Handwritten annotations in red:
- "operand1" is circled and labeled "kain".
- "operand2" is circled and labeled "d".
- "operandn" is circled and labeled "2".
- "OPERATOR" is circled and labeled "op".
- "operandn" is circled and labeled "op".

So, basic idea is that if you have any kind of an operation, which use values or input variables or whatever is computation is independent in the loop, then better is that you forget about whatever you are having over there. I mean what they mean, if you find out, and then these computations are irrelevant in the loop. So, what is the idea? You take out that stuff and either put it before I mean loop starts or you can better put it also after the loop start. I mean anywhere you can put it depending on situation. So, that is actually we have. We generally move it up in the code. So, that is we actually call code hoisting.

So, formally speaking, let us see what we mean. So, in this case, the condition required to determine loop invariant computations formally, how do you go about it? So, in this case, it says that if there is an operation like assigned variable, this 1 operator 1, 2; this one is this case. So, in CFG generally, we will have this on this one. We will all have operation nodes. For operator 1, operator 2 and operator 3, we will have the storage nodes. In this case, this is your storage node. This is your operation node. That is very obvious. Then, what you have to find out?

If I mean these guys, these guys, and these guys are constants if these guys, then how do you find out layer loop invariant? So, if they we either constant it like for example, if we have a loop and you are saying that something like say F is equal to 2 plus 3, so it will be 2. It will be 3 kind of a thing that means just because make it clean. So, in the loop, you may think that F is equal to 2 into 3. So, you can have 2 here and you can have 3 here.

So, this is again a constant. So, obviously 2×3 is equal to 1×6 . So, unnecessarily why do I want to have a 6? Over here, I can write directly 6 in F. So, again the plot can be brought out of it.

So, what it says that if the operands are constants, then there is nothing nothing to do in the loop. They are totally I mean irrelevant to keep in the loop. We can bring it out the computations there assigned values of the operands are located outside the loop. So, that whatever values whatever computations assigned values to this operators, operations 1, 2, 3, 4 are outside the loop.

That means what? So, any computation what is like say this is A is equal to A plus B . This is equal to D . This is equal to $2 \times F$ something like this and but all this A plus B D and F , they are all outside the loop and nothing effects. A , B , C , D , E , F , or whatever are unaffected by the loop. So, that means what whatever I mean these operations are they are allocated values, which is outside the loop. So, then therefore, if these values are these operators are operands are assigned values outside the loop, this is obvious that inside the loop, their values will not be changed.

So, if this is the case, then why do you want to unnecessarily keep these things inside the loop? Better, also you bring this whole expression outside the loop. Then, third computation that assigned values to operations is there is an insert invariant and there is a recursive designation. We have said that I mean the basic idea is that if you find out that, these operands are actually something whose values do not change inside the loop.

So, either they are constants, or then their values will not be changed inside the loop. If they are being applied the values like operand 1, operand 2, operand 3 etcetera are assigned values, which are outside the loop. Then, also their values will not change inside the loop. Also, it may be saying that operand 1 says the operand 1 and operand 2 or assigned outside the loop an operand end.

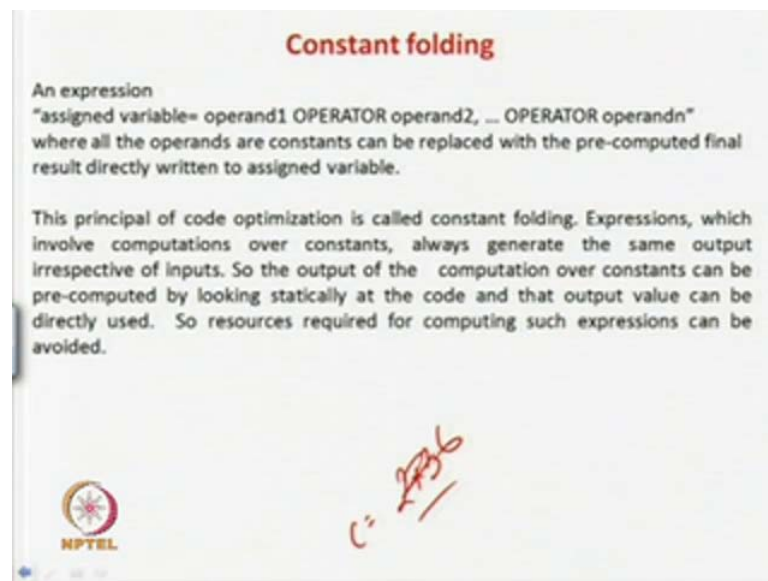
So you can say operand n is equal to operand one plus operand 2 then what happens the recursive definition the third case so this values of these 2 operands are assigned outside the loops and their again assigned up end it would be assigned operand end. So, again obviously the operand n value will also not change inside the loop. So, the recursive definition, we find out such a case in which case, the operand values are not changing inside the loop due to any reason like n or like constants. They are assigned outside the

loop. Again, they are they are being assigned by such kind of operation whose values are again assigned outside the loop and so forth and recursive manner you can define.

Then, you can think that these these values will not change anything inside the loop. Therefore, they will not change anything inside. These values will not change inside the loop. So, better what you do is that you do not keep such kind of operations inside the loop. Better you through it out so up hoisted.

So, those values will be completed only once, which is outside the loop. Then, will be save from n number of say this, so how many operations you are saving in this case operator 1, 2, 3, 4. So, if there n operators, n, n into A minus 1, if is your loop is running n number of times, so that many number of operations we will be saving.

(Refer Slide Time: 22:19)



So, that is one kind of an operation. Second operation is called the constant folding. So, that is like I said that if you have some operation 2 star, 3 C equal to 2 star, 3, so as a designer, you may write this by mistake or not be so much an experience designer. You may write line something like this.

So, in this case, what happens, which state? We require a multiplier, but if you have some kind of a constant like this, so what you can do is better you can better. The tool can find out the value is 6 and directly you can assign the value of 6 just like let us do that example. So, in this case, we have written A equal to verilog a verilog code. So, we

have written A equal to 1 plus, so this is CDFG for that. You can easily go and write about the constant 2 and write A . So, I mean we are showing very small examples of code.

So, you may think it very trivial, but the case is I think of a code, which is 10s of 1000s of lines of code. They are written by different different people. Then, you are actually merging it to 1 and combining them into 1 by proper interfacing and all. So, if you are a person developing such a big amount of code by a team, multiple people are involved and all. So, such type, I should say some type of minus stuff may be sometimes over loop by the people. Then, actually they lead to lots of I mean optimization problems in the code.

So, even if very small code, if you as a very human being and human, I also you can look it out and discover there is a problem. If your codes are very large and very complex in nature, so such type of things can easily creep in and lead to problems. Now, that is why, even all the techniques we are illustrating by small examples. This is because if I if you take very big examples here, you know very difficult to understand and for us to speak out in a class and also show in this thing.

So, basically the concepts we are doing are illustrated over here. So, whenever, if you take some practical example or if you are going to inertia to work on this, so whenever you will get some large codes like this, so this concept will be in your mind. So we will be less prone to make some such type of mistakes. Then, even if there are some mistakes, there will be codes to tell you that this is the problem when you go for an optimization mean. In case of transformation, the code there will be to, what are the cad tools will do? The cad tools will tell you that say designer see this is a problem over here.

Here is the redundant computation or here is the code code invariant computation. Do you want to post it? If you say yes, then it will do the multiplication for you like in this case. You may say that it may say that you have designed a code like this, like this one. It may say that is a constant value. So, do you want to write E equal to 3, and then transform your code. So, this type of warnings will be given. So, there are some kind of errors and some kind of warnings.

So, error means where there is code, bugs like you forgot to move a percentage, forgot to give semicolon and so forth. By these types of things are warnings, so it may say that

these are some of the warnings. You can look at the warnings and you can modify your code if required. So, that is what the next constant folding. So, in the constant folding, we are I mean operation 1, operand operation 2 and so forth. So, if all these are constants, then what you can do or somehow you can also pre compute them. It can be also be like that like C A equal to A 2. Then, B is equal to 3 and C equal to A plus B.

So, in this case, what you can do is that if you find out that A value of A plus B, it is always constant. Do not change anything like this. After that, we value of A and B is nowhere used. You can think that the value of A or B is never changed. Also, you can say that you better remove this and make that A equal to 5 kind of a thing. So, that is computed. You can if you can pre compute the value of some final expression, then you can do that when this actually called constants.

So, the expressions, which are combination of constants, always generate the same output irrespective of inputs. In this case, you can say that you can fold your constants and make it value. So, you can in this case you say when order sometimes, you can see multiplier so forth.

(Refer Slide Time: 25:42)

Constant folding

Given below is a Verilog code having an expression where all the operands are constants. It may be noted that the computation "A = 1 + 2;" always results in A having the value of 3, irrespective of inputs. Also the final value of A (i.e., 3) can be pre-computed by statically looking at the code. Finally, the expression "A = 1 + 2;" can be replaced with "A = 3;", which saves an adder and a register.

```
module CFG_trexample (A);
reg [3:0] A;
output [3:0] A;
initial begin A = 1 + 2; end
end
endmodule
```

So, now we will go to the second level, third policy of optimization.


(Refer Slide Time: 25:45)

Redundant computation elimination

If there are two expressions
"assigned variable1=operand11 OPERATOR11,...,OPERATOR1(n-1) operand1n"
and
"assigned variable2=operand21 OPERATOR21,...,OPERATOR2(n-1) operand2n"
where,
operand11=operand21, OPERATOR12=OPERATOR22, ..., operand1n=
operand2n

and the operands of the computations do not get modified in between evaluating "assigned variable1" and "assigned variable2", then we can perform the computation (i.e., "operand11 OPERATOR11OPERATOR1n operand1n") only once and assign "assigned variable2"= "assigned variable1".

This reduction is called redundant computation elimination. Obviously, redundant computation elimination reduces hardware requirements.




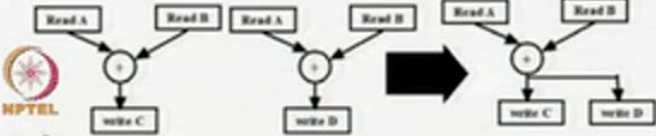
It is called redundant computation. That is very simple, but actually, we always make this step for mistake.

(Refer Slide Time: 25:56)

Redundant computation elimination

Verilog code below has an expression with redundant computation. It may be noted that computations "C = A + B" and "D = A + B" will result in same value assigned to C and D. This is because operands and operators of the both the computations are same and operands A, B are not changed in between "C = A + B" and "D = A + B".

```
module CDFG_tr_example (A,B,C,D);  
input [3:0] A,B;  
reg [3:0] C,D;  
output [3:0] C,D;  
initial begin  
C = A + B;  
D = A + B;  
end  
endmodule
```



So, we will see first the example. Then, we will come back to the formal definition somebody as did not code. Then, we are saying that C is equal to A plus B. Then, after sometime E C D C equal to A plus D. So, in this case, you may say that no one there will be no full to write this but actually do not think of a case like this. Think that there are the big very big codes in between after a long time is also writing A and B.

So, what I mean in this case say mean there is say, another 10,000 lines in between in this. So, after writing this $A + B$, then of these 10,000 lines, so may have written in over a week or something like this. 10,000 lines means there are lot of procedures in between. It may not be a flat 10,000 lines code.

There will be a lot of procedure etcetera, etcetera, etcetera. Then, I mean procedure modules and so forth. Then, after say 7 days again, you forgot that you have already computed $A + B$. Once you could have reuse that, but say again. You are writing A equal to D equal to $A + B$. so, there you can see there is a redundant computation.

You have to also observe one thing that this is a redundant computation. If all these nowhere 10,000 lines of code, A is here and B is here nowhere here assigned A or B new value. This should not be possible. So, you have written C equal to $A + B$. Then, after some 5,000 lines of code, you are writing A equal to something B equals. Sometimes even if you reuse the values and all, if you are writing $A + B$; then it is equal to the same C .

Then, you will be in a big problem because that may not be true, but if it is the case, then that A and B equal to A and B have returned. This is because as they are, you makes A and B some constants, which are taken as inputs. They may not be constants, but they are always taken as inputs. Then, you do not change them like they will be multiplier and multiplied, which you never change. Then, after 10,000 lines of code, never you have assigned a new value to B A , never you have assigned any new value to A . So, these things are not there.

Then, what is the case? Then, obvious the value of A and B has not change and you could have very easily written D equal to capital C . So, now it is very easy. Here is 1 line just after another. So, nobody will right like that, but if there would have been practical consider a very week chunk of code, here is the very high chance that you may have forgot. You have written in case of D equal to $A + B$, so you could have written D equal to $A + B$. Again, there is a redundant computation. Then, what you actually do?

So, in this case, see the transformation. This transformation is very simple, $A + B$ equal to C . D , you can write it over there. You have to very very careful that the value of A and B should not change in between these 2 expressions. So, that is operation; A and B

should not have change between A and D. Therefore, actually the compiler will tell what you have time in 2 will tell you the redundant expression.

Then, you have to go and think about that, whether you have change there. You can search and find out whether I mean, advance tools can even tell you that whether you have A and B have changed over here or not. So, if there are no changes of A and B, then you can do for this redundant computation elimination, some of the formal of doing.

(Refer Side Time: 28:24)

Redundant computation elimination

If there are two expressions
"assigned variable1=operand11 OPERATOR11,...,OPERATOR1(n-1) operand1n"
and
"assigned variable2=operand21 OPERATOR21,...,OPERATOR2(n-1) operand2n"
where,
operand11=operand21, OPERATOR12=OPERATOR22, ..., operand1n=
operand2n

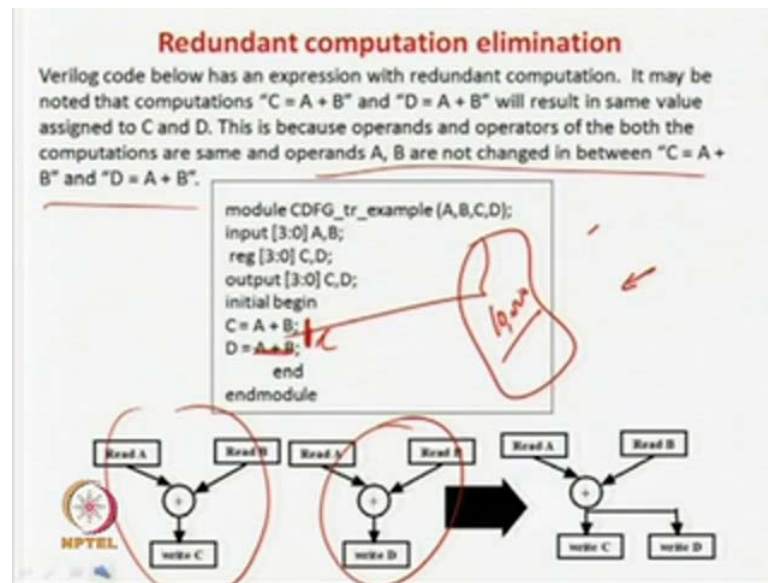
and the operands of the computations do not get modified in between evaluating "assigned variable1" and "assigned variable2", then we can perform the computation (i.e., "operand11 OPERATOR11OPERATOR1n operand1n") only once and assign "assigned variable2" = "assigned variable1".

This reduction is called redundant computation elimination. Obviously, redundant computation elimination reduces hardware requirements.

NPTEL

So, formal doing says that assigned value is equal to operation this 1, operand operator 1, 1, this 1 dot, dot, dot, and assigned variable 2 is this 1. Now, this 1 should be equivalent to this 1. This should be equivalent to this 1. This 1 should be equivalent to this 1. Similarly, all the operators should be same. All the operand and operand should be same as well as that nowhere, after this expression, nowhere operand 1, 1, operand 1, 2, operand 1, 3, dot, dot, dot, operand 1, n should change in between. So, that is the value of operand 1, 2, 3, 4, 5, 6, n should be intact from this expression to this expression. Then, only you can say that this expression, I will store in temporary variable. I will be applying theorem. So, that is the actually the idea.

(Refer Side Time: 29:04)



Redundant computation elimination other ways says that we compute some expression and after that, the expression does not change. Again, why do you re-compute the expression? Let us use this expression. Once again, there is the idea of this redundant computation elimination. But, we have to be very careful here. Why? This is because you are using A equal to say, you are using A, I mean A equal to B plus C over here. C equal to D plus over at the reusing that, but anywhere in this middle of the code, if A has been changed or B has been changed, then it will be very big problem.

This is because in that case, you have to mean instead of A equal to B plus C D will be writing as C. You may be storing in some temporary variable. You will be writing it over here. Then, it will be very big problem because some other values will be assigned. So, in the course of optimizing your code, you may lead to erring bugs. So, I mean here in this case, you are writing D equal to C. So, that case is also very risky. Then, you have to be also very careful that A, B and C do not change over time. What people do is they put some temp variable like t e m p 1, and then they will write D is equal to temp instead of writing D is equal to C because I mean A, B and C are user variables.

So, it is very, very possible that user will be always using them. User will be always using them. I mean 245 kinds of assignments etcetera, etcetera, and etcetera. So, if you are using a temporary variable, then you know that I will not be using that anymore. I

mean I will not be actually, I mean I will be taking that because they are all already defined for compiler optimization.

So, what happens is that you start with initial begin. Then, you see the temp equal to 1. Then, temp equal to A plus B, you know that variable A plus do not change. So, even if value of C changes anywhere in the code, you are not bothered. So, D equal to temp 1, you can very write happily. You will be done with this one. Now, this was about what do you say is that redundant computation elimination.

Now, we will go for a sub case of this one. So, for example, if you have a very big expression like A equal to B plus C plus D plus something, something, something. Again, you have something like that. Now, you can say that partially they are equivalent like say for example; we will take an example over here.

(Refer Side Time: 31:05)

Common Sub-computation elimination

There is a small change in common sub-computation elimination procedure compared to redundant computation elimination. In common sub-computation elimination procedure we need to have an extra temporary variable which stores the value of the common sub-computation for future use.

```
module CDFG_tr_example (A,B,C,D,E,F);
input [3:0] A,B,C,E;
reg [3:0] D,F;
output [3:0] D,F;
initial begin
D = A + B + C;
F = A + B + E;
end
endmodule
```

The Verilog code has a common sub-computation. It may be noted that computations "D = A + B + C" and "F = A + B + E" have "A+B" as the common sub-computation. This is because operands and operators of the both the sub-computations are same and operands A, B are not changed in between "D = A + B + C" and "F = A + B + E".

So, in this case, you see are somebody as in verilog code over here. Now, you see this is equal to A plus B plus C and S is equal to A plus B plus C. Assume that the very big chunk of code over here, which is in this case, D in every whole expressions are not equivalent. But, what is equivalent? Equivalent part is A plus B and A plus and A.

These 2 parts are equivalent. Still, you can also save something provided that A and B values do not change. In this part of the code, they are assuming that they are not changing. Then, what you can say there? I can say that D is equal to A plus B plus C.

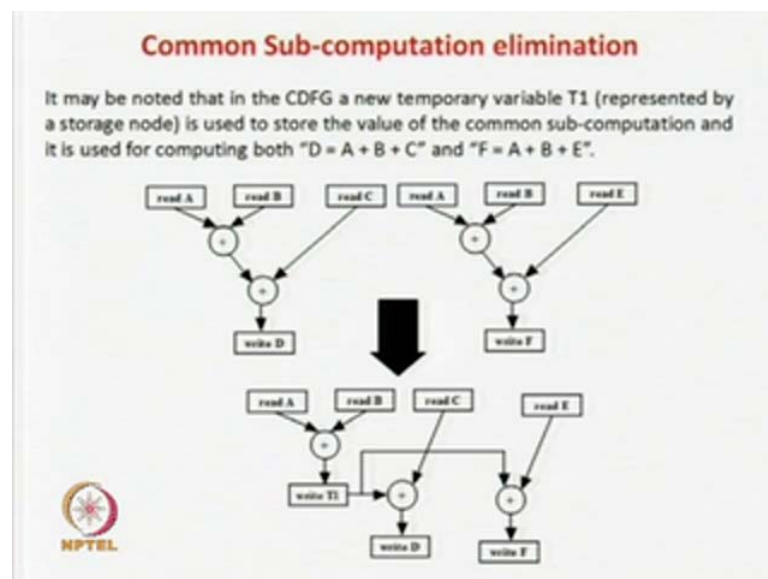
Then, I can and one more steps here. Then, t equal to A plus B , I can say that is by temporary variable.

Now, here instead of A plus B , I can put the value of t over here. So, it will be t plus E . so, I mean in this case, I mean the saving will not be there as much because obviously the saving is there. This is because you calculate A plus B , then you apply D equal to t plus C and in this case, apply t plus C . So, instead of 1, 2, 3, 4 it is something like that. It will be temp equal to A plus B . D will be equal to t plus C and dot, dot, dot. After that, F will be equal to t plus E .

Now, you see 1, 2, 3 multipliers adders. In this case 1, 2, 3, 4 adders you are going, so you are saving one annual in this case. So, like previous case, you have to be very very careful that the value, this is the part you are using for mean redundant computation in redundant some computation. In this case, you are saying it is equivalent. So, we have to be very careful that the value of A and B do not change in between.

So, that is having to very carefully this thing. So, what is the formal star? So, if you have some expression like this some expression like this, then you find out a subpart of it like say operate and operand 1 operand. This part is equivalent to this part or this part may also be equivalent to a sub part over here.

(Refer Side Time: 33:10)



So, those things you can use it. This part you can say in a temporary variable t_1 . All do not compute it. Again, here use the value of temporary variable one over here, but it should be very careful that operand 1, 2, 3, 4 or whatever over here and between these 2 codes, where we are applying, the equivalence should not be changed or assigned anywhere. In this case, this was like very similar to the case of common sub expression elimination. But, in this case, it is not the full expression. It is only a part of the expression.

Now, just like looking at this code, if you see it is equal to $A + B + C$ and in this case again, $A, A + B + C$. so, this is the 2 expression. So, in this case, you can see this part was common like this part was very common. So, if you just look at a graph level stuff, so it is very easy to travel the graph. You can find this equivalent to this equivalent to you can go for graph some kind of graph isomorphism kind of algorithm. If I know that graph sub graph isomorphism, then these 2 graphs are equivalent in nature.

So, you can use them, and then we need to also write. It is added to B over here in it. So, in this case, operand in C, this case operand in this part is different part of it. So, we can easily go for common some expression elimination. So, till now, we have seen above how many. We saw that loop invariant and lusting. Then, you have seemed some expansion elimination, and then common full expression elimination. Then, also we have seen constant folding. Another very important thing we always here in software as called dead code. Many times code has some dead code, dead code computation.

So, what is dead code computation? Dead code computation in other words says that you have written a very, very large code. Some of the expression you thought that it may be a very useful expression, I want to something like that or other that you might have realized that this variables are not at all required of this computation are not are required.

Then, you forget to remove it or remove those some parts of the combination, which are not at all affecting your code. They remain still they because you thought of you logic as something. Then, you start writing your code. Then, after sometime, you find out that this logic is no longer required in your code. You are not that important. So, you do not use them at all in your code anymore.

So, it is, but is still remains because it is a very big code. You forget to go above, go back and exactly eliminate which part of the variables or which part of the computation

with thought to be used. It is not required anymore. So, you could have gone there or eliminate. This is a very difficult problem because generally speaking, if you may write some expression, then you can go out of computations.

You can find out that. That computation becomes tautology. Tautology means it should be always true like you can. Then, very complex Boolean expression you can think this is true. If a man goes to heaven, then if you tell true, then something something happens very big.

So, you are writing a logical code. Then, what happens is that you can find out. Then, based on all the input cases, you will be a tautology. That is always answer will be always true. All should be false kind of a thing. So, in that case, you do not require such a big computation because answer is always 0 or always 1 kind of a thing.

It is very very difficult sometimes. We have been proved mathematically or for a human being to find out that that is always the case. That is always true or false. That is the case redundant. You need not go for such a computation there. Just write 0 or a 1 in that case. So, sometimes you may also find out that whatever you are doing, something what doing as a very big expression and 1, 0.

So, that means is no importance; should not and it will be 0. This is because ending with 0 has no meaning. Some computation will always lead to 0. You are ending with something that is a very difficult mathematical problem. To find out that in what conditions, it will be there and then proof that are all possible input conditions.

We find values, then it will not change or it is always 0 or 1, always 2 or false. So, that means that could mean a code which is lines is your code, but is having no impact on your code. It is always either giving 2, result always giving a false redundant kind of thing.

So, finding such code in a very generalize environment is very difficult problem. Again, it is always if you are writing a code as a human being, which sometimes like code whose values or always constant or whose condition results are always constants, it will be easily done away with this code. So, such type of codes can be thrown out of your main code, which we are submitting for design. They use the unnecessarily lead to area and etcetera. In this case, I tell you.


(Refer Side Time: 36:35)

Dead-computation elimination

Dead-computation elimination is one of the most simple transformation where a computation that has no effect on the output of a code is eliminated. In the CDFG the corresponding nodes are eliminated.

```
module CDFG_tr_example (A,B,C,D);
input [3:0] B,C,D;
reg [3:0] A,E;
output [3:0] A;
initial begin
A = B + C + D;
E = A + 1;
end
endmodule
```

It may be noted that computation "E = A + 1;" is dead because it has no effect on the output of the code.

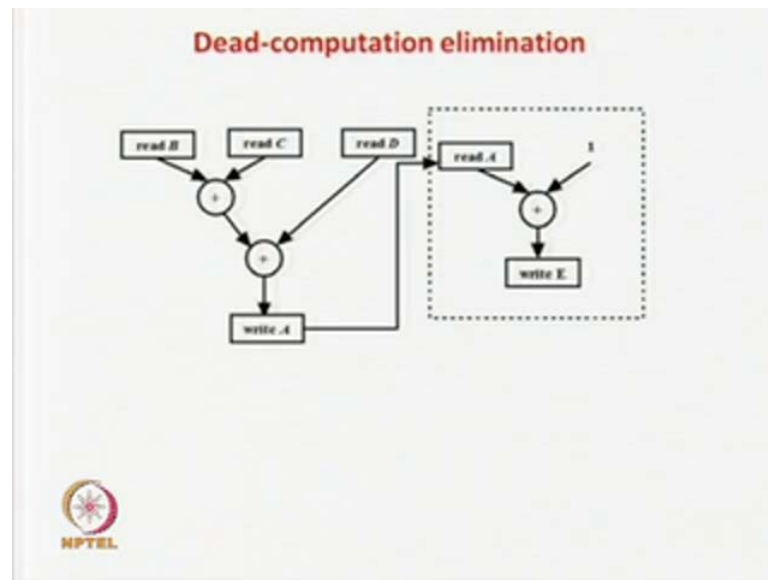


So, you are A plus B plus have some people are so code, so initially they are saying A equal to B plus C plus D. then, here it is written that A equal to A plus. It might have been the case that might have thought that E is another output. In the case, you might have thought. So, in this case is output should be, what do you calls output is A. Also, you could have thought that E is another output. He might have thought. Then, you have written in this line E equal to A plus 1, but there after sometime, he might have changed this specification and found out that the output is not at all required. There is enough output. Second output is not required.

So, here eliminated the value of E, but he has actually forgot to do this. Then, still he has define registered A, E. So, he forgot to remove this definition. So, A, E is remaining a variable in, but is not in an output. So, he just eliminated. That E is not required to be output. So, he eliminates this here and also as output. A, E was an also eliminated. Another depth definition that we state that is the variable in might have forgotten into eliminate. Then, also you might have forgot to eliminate A equal to A plus 1.

So, this code unnecessarily will be in your computation. It will lead to an adder, and a register for E. It will be unnecessarily hanging and making area operate. So, you can actually eliminate this part. You can eliminate this part. This is actually called dead code elimination.

(Refer Side Time: 37:42)



This is because it has no effect on the output of the code. So, like this for example, you can have looked at it. So, this is the input and output. So, this is your output. This is not at all output, but still so this part of the computation is not required. You can through this. So, dead code elimination actually talks of eliminating such a code, but actually it is a very simple example.

So, you can understand this. Sometimes, it may happen that this is a lot of condition over here and you are comparing it over here. Now, it may happen that and this may also be, if this is true, then you execute something. It may be the case, but as told within, it may be the computation. Here, it will be a very very complex stuff like if A, then this, then so forth. You can find out that for all input cases.

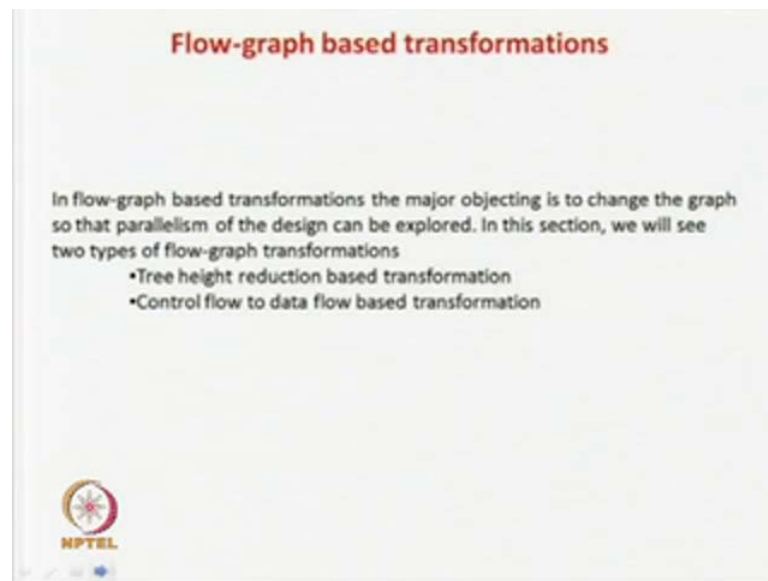
The value always is going to be 1. So, say you are comparing it with the value of 1, so you are saying that if this is 1, and then the output of this 1 is 1. This is true. There are you do something, but you may find out that either because of the lot of inputs, all possible input computations are whatever complex equation you have written. Always, the values will be 1, always the value will be 0.

So, the answer is always true or always false. So, this is very well known, so need not go for such big computation and computation. So, you can throughout all this and you can write the value as true and false. So, this is actually dead computation, but still is very,

very difficult to find out the dead computation. This is because it is having no effect. Only your code, you are just saying 0 and 1.

So, like ending with the 1, ending with the 0 or oaring with the 1 basically wrote have any meaning. So, in such cases to find out those parts of the code an eliminated is that very difficult. So, in this case, was the very simple example. So, you can find out that this code has no effect on the output, but really to debug and find out.

(Refer Side Time: 39:09)



This logical part of the code is having been no effect on the output is very difficult problem. But, still I mean you are always thinking of how we can minimize or code or how we can throughout the part, which is not required. So, that you can get a very good what do I say is a very good transformed code.

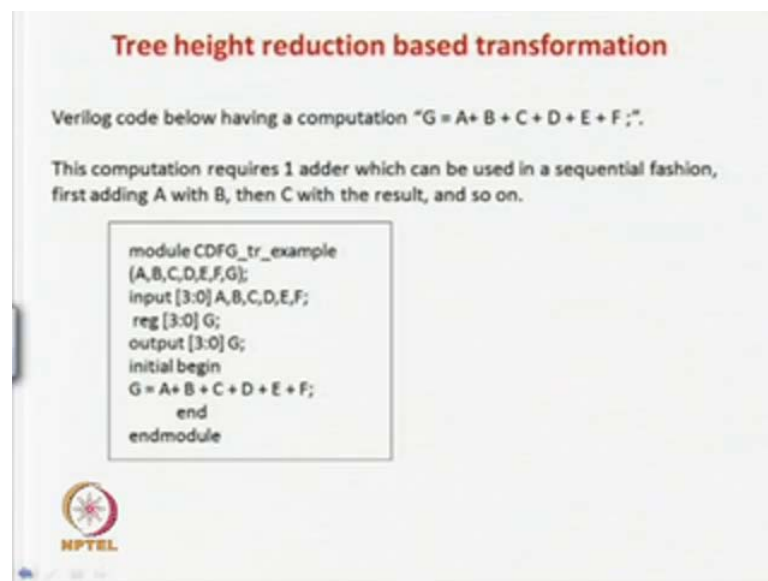
So, that it is I mean you get a very good. What do I say much optimized? I mean argument for this like you code have oppose in women, sub expression such are even save some part of you will put. if you can find other very big chunk of code, very big module of the code is dead, it is not in have any effect or the effect is very redundant kind of a thing on unnecessary like ending with 0 oaring with ending with one oaring with 0 kind of things you are you are coming into picture.

Then, you can, you could have thrown out this code. This is because this code is having been is not any effect. Now, there is actually very difficult to do problem to solve. Still,

people are finding a way to find search porches of the code, give warning to the user and so forth. So, these were all about compiler driven transformation. Now, we will see some graph flow base transformation that is we have drawn these. What do you say? We always have drawn our this control and data flow graph.

So, now what we do? Based on looking at the graph structure, how can we optimize our code? So, that is actually called the flow graph based transformation. So, true we will see that is actually tree height reduction transformation and is control to data flow. These already we have seen in the last lecture that if you convert your data flow, control flow diagram to data flow diagram CDFG. Then, you can save some redundant element. Now, let us see. So, first we will go for this height reduction. Then, we will again see what the theory behind it is.

(Refer Slide Time: 40:44)




Tree height reduction based transformation

Verilog code below having a computation "G = A+ B + C + D + E + F;".

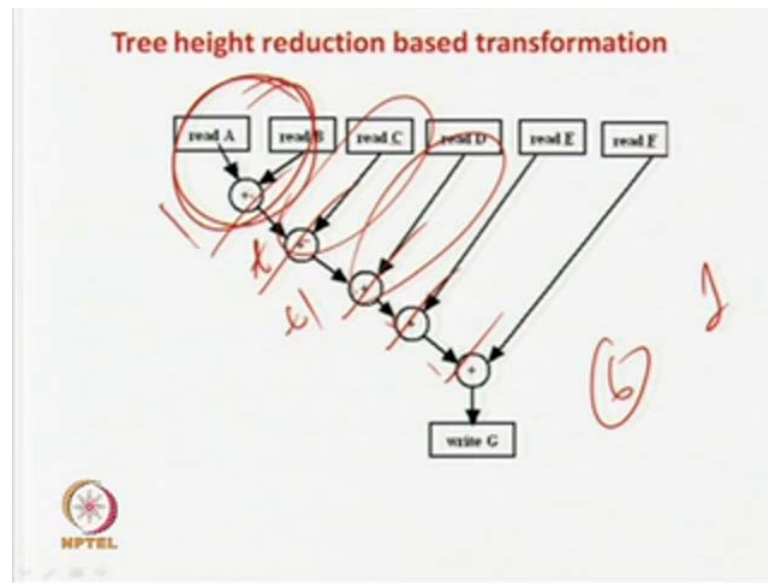
This computation requires 1 adder which can be used in a sequential fashion, first adding A with B, then C with the result, and so on.

```
module CDFG_tr_example
(A,B,C,D,E,F,G);
input [3:0] A,B,C,D,E,F;
reg [3:0] G;
output [3:0] G;
initial begin
G = A+ B + C + D + E + F;
end
endmodule
```

 NPTEL

We have a computation like this. So, what is the computation? Computation is equal to A plus B plus C plus D plus E plus A. So, it is a very long computation. This is your example.

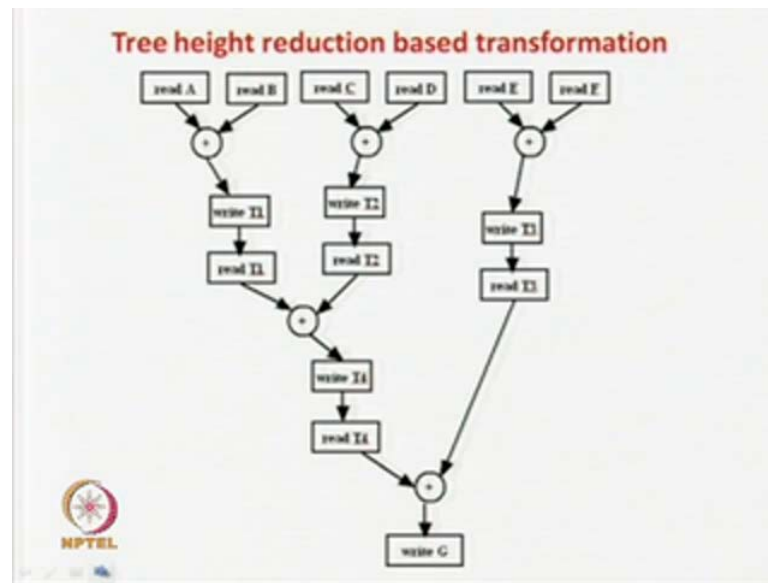
(Refer Slide Time: 40:58)



So, it is a very sequential kind of an operation. So, let us just see C D F, so this is CFG. First you do A plus B, this result. Then, you go for C plus D. Then, that is value go and so forth. So, in this case, how many other you require? So, in this diagram, you do get 1, 2, 3, 4, 5, 6 adders are required. In fact, you can also due within 1 adder. How? First you do A plus B, store the values of somewhere same adder. You go for t that is the temporary variable for A plus B.

Then, what you do? You take the value of t and at the value of C, same adder. Then, you make is A t 1, then the t 1 is stored somewhere in the same adder. You reuse for adding t 1 and D and so forth. So, you require only 1 adder to solve the problem. In this case, you require how many steps 1, 2, 3, 4; at least 5 plus 1, 6 steps may be requiring to do your problem for once. First step 1, 2, 3, 4, 5 additions are there. So, that is 5 steps, you will be having 5 additional steps require solving the problem. So, this actually the tree height is very large. Now, what can you do? We can go for some time of optimization.

(Refer Slide Time: 41:50)



In this case, we can say that we will have multiplier adders; you can do operations in a faster way. So, here what we are optimizing here? What we are optimizing? It is speed at the cost of area. So, we are giving more area and we are saying that optimization is possible.

So, now in this case, what we are doing? So, in this case, C we are doing A plus B in one go, C plus D is one go and E plus F in one go. Then, we are adding the value of t 1 plus to this temporary variables that is in the under step. In third step, we are adding the value in this one. So, how many adders we have put? We have 1, 2, 3, 4, and 5.

So, in this, all these 3, you require 3 adders for this one. These were doing in parallel, and then one adder here. So, same adder you can use and again in the same adder, you can be used for this one. So, in fact, you are having 3 parallel steps. So, A plus B, then C plus D and E plus F to be a together, then one of the adders you are using for adding intermediate values for this one. The same adder can be reused for doing this one.

So, 3 adders are required. So, in the previous case, 1 adder was required. Now, we are using 3 adders, parallel adders and you are doing the operation in 1, 2 and 3 steps. You are getting their answer. In the previous case, you have the 5 step also. So, again the tree height has been reduced from 5 to reduce to 3 steps. So, what essentially we have done in this case, suffer some very big expression or very long expressions. If you are using a single tree to do that a very long tree to do that the hardware requirement will be less.

In that case, the very sequential kind of operation and your time taken will be as many numbers of operators operands operators are there. So, that might number of steps will be required. Now, if you can paralyze it that is your rezoning the hide of the tree, then you are thinking more hardware.

(Refer Slide Time: 43:18)

Tree height reduction based transformation

Tree height reduction based transformation uses the commutative and the distributive properties of expressions to decrease the length. Let there be an expression "assigned variable1=operand11 OPERATOR11OPERATOR1n operand1n", which can be reduced in length by breaking it into sub-computations and storing their outputs in temporary variables.

Finally, operations are done on the temporary variables to determine the value of "assigned variable1".

It may be noted that evaluation of a long expression involves more steps because operations are done in sequence (on order of precedence of the operators). On the other hand, if we break the expression into sub-expressions then they can be evaluated in parallel, there by speeding up the computation.

However, in the case of sequential evaluation, less hardware is required as the same circuit can be reused (in different steps), which is not possible in case of parallel evaluation.

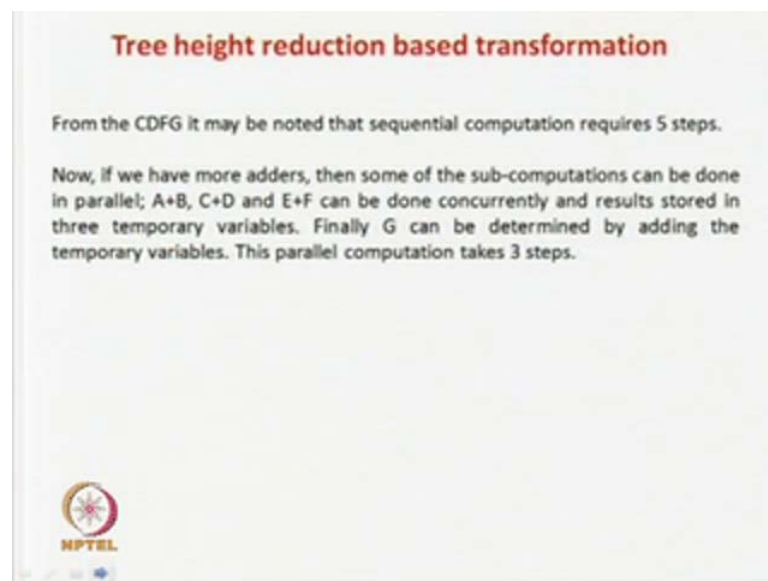
Now, you are optimizing on the number of hardware say you are optimizing on the number of steps on the time required. So, formally, we say that tree height reduction is base on transformation that are uses commuted, even distributive properties and so forth. So, let there be an expression like we say that variable 1, then operator 1, then this is so and so forth.

So, this is a very verilog operation as you can see. Now, what we can do? So, if you go on sequential from this, this, this, then what happens? It takes a very long time and the pre length is very large. So, what we can do? We can break it up some computations, and then we store their outputs in some temporary variable. That is the idea.

So, you have very long expression like A plus B plus C plus D plus something. So, we are breaking them into B plus C, D plus C, E plus F and so forth than variable in temporary variable. So, sub expression, you are breaking this one, this one and this one you are operating in parallel. So, again these 2, 2 temporary variables, the output will be again and again, the output of these 2 other variables will be there. Then, this will be also happening in parallel.

So, what is the idea essential in very big expression? You have to take 1, 1, 1, 1 and do about these, one way of going about, otherwise that you break it up into small sub expressions and do that parallel this. So, the intermediate results again you process whatever is possible in the intermediate results paralleled. Again, you read this thing. So, you can reduce the height of the tree. You can also get a very quick solution for this one, but what is you are losing? You are losing the amount of time.

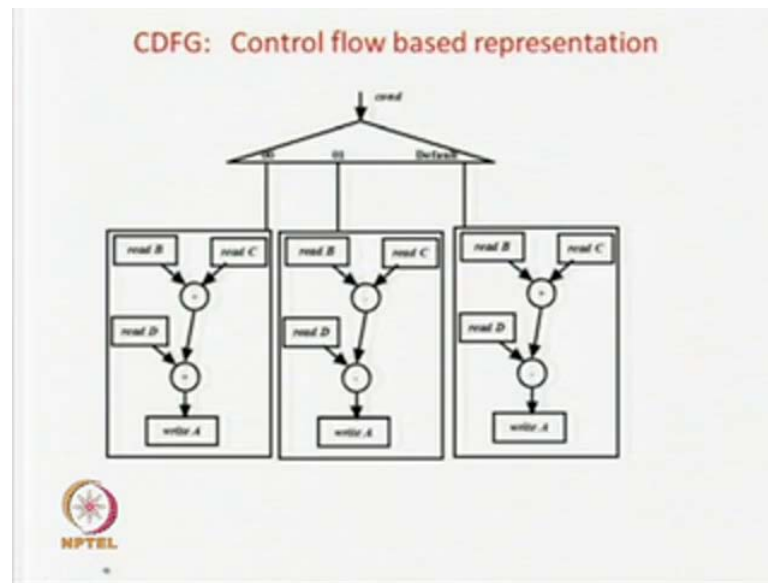
(Refer Slide Time: 44:39)



So, time loss in this case is; I mean time gain is very high, but you lose in the area. This is because whatever you are parallelizing because you are breaking into sub expressions, some expressions you have breaking whatever sub expression, you are broken up and you are eliminating and you are executing in parallel. So, that many amount of hardware is required. So, in this case, 3 adders, 3 additions you could use. You brought down into 3 sub expressions or 3 sub additions.

So, 3 parallel adders are required, but now in second step, you can use any one of them in the fourth step. Also, you can re use any one of them, kind of second step and third step you can reuse. In this case, in this long tree heighted, long height tree, you are using the adders one after another. So, I mean storage space or I mean the variable requirement is very, I mean this adder requirement is very very less in this case and one in this case it is 3.

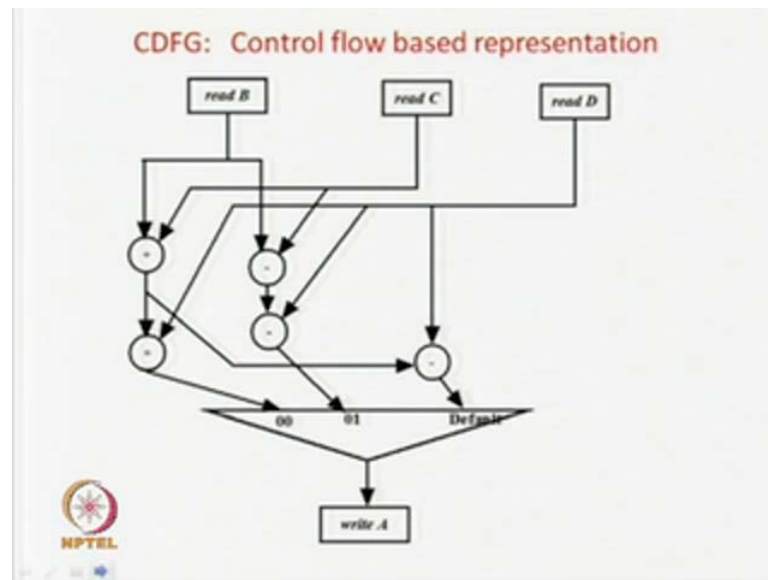
(Refer Slide Time: 45:20)



So, I mean if you go for tree height reduction, you will get faster operations, but what do you we lose; obviously, your time. You are gaining and you are losing in the area only. So, that is about the 3 hybrid transformation. So, next one is actually control flow and data flow based transformation. So, already we have discussed in the last lecture. So, we had some based on some conditions, you either do these computations, either you go for this computation and we go for this computation. Now, these are control flow base representations.

So, I have given a verilog code. It can be hardware definition language. It can be directly transform to this one and you are going to get your solution. Then, we say that if I mean that is CDFG can be directly obtained from you are verilog code that is actually control flow base. Already, this was discussed in the last class. Now, you see the lot of redundancy like B C, B C, B C are already there. Then, these Ds are already redundant.

(Refer Slide Time: 46:14)



If you can go for a control flow base diagram or control flow base series, the problem is solved. You read B once, you read C once, and you read D once. Then, you do the computation, but you forget about all the redundancies. A eliminate redundancy like B plus C is here once and B plus C here once.

So, B plus C and B plus C that is B plus C. You can see this. This is B plus C. So, this is computed once used here as well it is also used here. So, 2 times it is used.

So, if you go for this, this is your control flow. This is a data flow. This is a data flow not control flow. So, this data flow base representation, already we have seen. So, in this case, data flow representation redundancy the eliminator, but again what with the difficulty? We saw that there is there is no one to one representation between the verilog code and data flow based representation.

So, what we have to do? We have to go for flattening of the code. We have to go take some tool or do manually redundancy elimination and so forth. Still, I mean your representation is better; area requirement is better and so forth. So, this was about your graph base transformations. So, this can lead to either a very fast action or your hardware will be very fast at the cost of more number of hardware.

In this case, you redundancy is reducing. The reducing of a redundancy, there is the control and data flow base stuff you reducing. You make ever complex representation,

but at the cost of that, you would go for manual intervention because in data flow base CDFG verilog codes are h d l codes. They cannot be directly mapped if the nested loops and also you have to go for flattening and that entire thing that you have already seen. That type of transformation is hardware library base transformation.

So, what is that? So, your library has some kind of special case. What is the design library? So, if you are fabricating your circuit, you should have a library like I have a 2 input. I get a, I have a 3 input and get, but you may not have a 100 input and get, but some of the expression may be like $i_1, i_2, i_3, \dots, i_{100}$.

So, you can think that I will have 100 input and get an operation will be done. So, in digital design sometimes, we have seen that our digital design and implement 10 input and get, but is a real case design. You not have such a huge number of end gates and all. So, a design library is given. So, people are we studying about fabrication? It is very very difficult to fabricate in VLSI gates having more than 4 inputs. Then, the gate is slow and sever other problems are there.

So, you can read out any standard VLSI, the digital design books, the c most design books. What is the problem and all so base takes for me trying granted that designing anything of more than 4 input and gates, 4 inputs, any kind of gate is a very difficult problem. Now, what is what do you have to do? So, what is the design life is design life will tell you that I have 2 inputs and gates, 3 input and gates have or gate and flip flop etcetera. This is the speed; this is the area and etcetera.

So, all the definitions, you can also have some special stuff like you can have an increment as you can have a comparator. You have multiplexer; you have carry for or adders. So, you say I built some very good design blocks, which are actually fast take low power and very generalize. It may not be which should not be very specific, but generalize I can flip flop a multiplexer or register the comparators, which every people use flip flop etcetera. It is very commonly used. So, there can I say that this design is very good. So, better keep in the library.

So, there any percent can use it, so that becomes a design library. Sometimes based on this design library, some transformations can be done. For example, if I say there, I want to compare a 6 bit number with 0, any 6 bit number compare to be 0, so how can you do that? So, you can use the comparators. So, it is very simple because you have to use a

comparator. So, this is your number 0. This is your 6 bit number, which we are taking of this are 6 bits 0.

(Refer Slide Time: 49:35)

Hardware library based transformations

- Any operational node of a CDFG has a corresponding circuit capable of performing the operation e.g., adder to do addition, comparator for equality checking etc.
- This mapping of an operational node with hardware circuit depends on circuits available for implementation in the design library. For example, checking equality of a variable A (6 bit number say) with zero can be done using a 6-bit comparator or simply feeding all the 6 bits of A to an OR gate (if A is 0 then all the bits are 0 thereby generating 0 from the OR gate).
- The OR gate implementation consumes less hardware than the comparator, however, depends of availability of a 6 input OR gate in the library.
- Hardware library based transformations are modifications in the operational nodes of a CDFG, depending on availability of corresponding circuits in the design library, for achieving efficient circuit implementation in terms of area, frequency, power etc.

The slide includes a hand-drawn diagram showing a 6-bit number '0' being connected to an OR gate. The OR gate has six inputs, each labeled with a bit of the number '0'. The output of the OR gate is labeled '0'. To the left of the OR gate is a hand-drawn comparator symbol. The NPTEL logo is visible in the bottom left corner.

Now, the 6 bit number you want. So, if it is all 0, you get the answer as one and so forth. Another very simple way of doing it is what you connect all the 6 bit numbers. All the 6 bits of your number to or gate, if everything is 0, then you get the answer as 0, so make in or gate. So, if all the input bits of the 6 bit number are 0, then you get the answer 1 and so forth. So, very easily you can say that 6 bit nor gate, these are comparators for 0 of a number.

So, if all bits of a number are 0, then the answer is 0. Then, number is 0. So, if you feed or number is the bits of number to an n bit, but you can n bit number to a n bit nor gate. So, the output is 1. Then, you not at the number is 0 else it is not, but for that you design library should have a input nor gate or you should be able to take some gates and make it as a 6 input nor gate or whatever n input nor gate kind, otherwise if you do not have such a kind of a nor gate say for the time is say 5 n input of 6 input nor gate is not available. Then, have to do you have to take comparators. So, comparator available in the design library should take a comparator feed the 6 bit numbers 6 feed the value of 0.

So, if the number is equivalent, then we will get one else not. If you if you would library have neither say 6 inputs nor gate, it specially design as a comparator, then will be very

very happy that you can use this. So, what is the better thing is that if you use the comparator, then comparator actually uses these are lot of x or gate for computing.

So, that means what so if you want to that this that is a 3 bit, 2 bit numbers are there say. This is a 0 a one and a 2 and say 3 bit numbers b_0 b_1 and b_2 3 bit to 3 bit numbers there the compact for equivalence what you do you have to x or a 0 with b_0 a one with b_1 one and a 2 with b_2 if all the x or values are 0 that is both of all the bits are individual equivalent then only the answer is equivalent so compare will actually involve all all x or x oaring with all the input with that you have already land in digital design and make you course so a general comparator been have for the 6 bit comparator then you require 6 x or gates to do that

But if you want to compare with 0 so should not require so many x or gate so you can easily take care of a nor gate and solve the problem but, for that your design library should have these specialized nor gate which can compare with 0 and comparing with 0 is very much required in most of the case of very general purpose circuit because if we implement a loop then we say that for i equal to one to n or you can say that for 0 equal to for some value equal to 0 to n or you can say that for n to 0 anything you can do that so many times you require comparator comparison with 0.

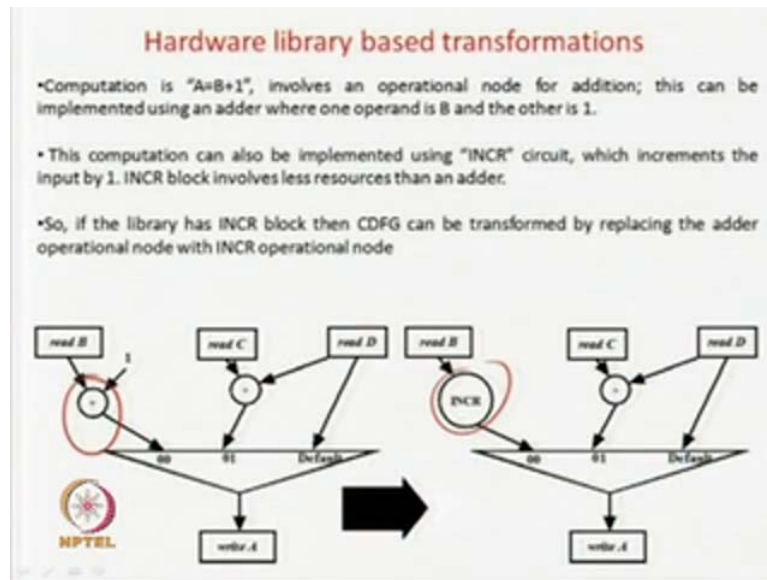
So it is very you can you can think very generalized to a which should be placed in the library so if you design library has a comparator as well as a as a 6 input nor gate kind of a thing so you can change you are a simple comparator as you can in instead of the comparator you can bring in this stuff write being in what stuff you can bring in you are a 6 input nor gate for a comparator. So that means one based on something available in your library you can always go for some kind of optimization let us take another example like say for example, if this is CDFG where you have one operation you say that d equal to b plus 1.

So, what is the one thing you can do? You can take an adder. You can do that. So, that is stuff; but actually there is something called increment. So, just the hardware is very very simple if you can go to the digital design. I mean what you call undergraduate course. So, you can find that increment is nothing but it increments the number by 1.

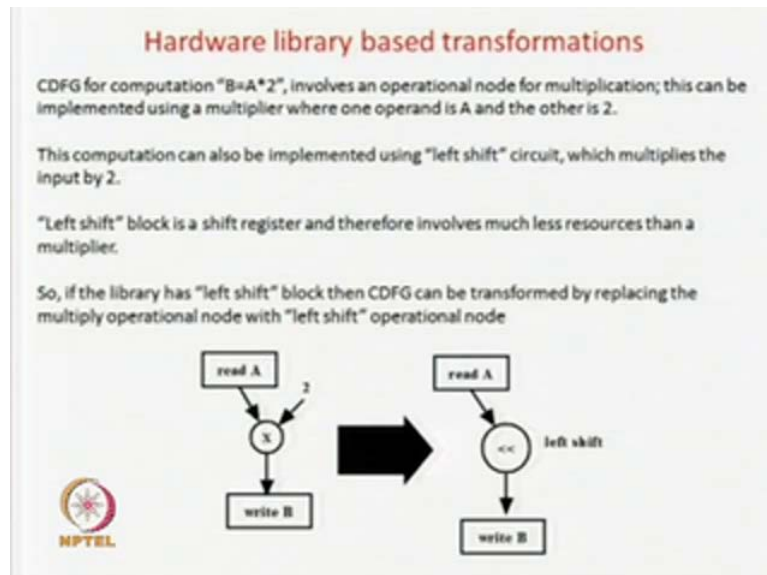
That is it. Lots of anything it does not do any kind of full addition. So, there is no kind of full adder and half adders involve there. It is just a simple circuit. Increment the value by

1 that is it incrementing the value of 1 sometimes is also required in computing tools complement and so forth. So many times, you have to add only one number. So, in that case, this is calling a increment. So, if in this if you a library design library, you have an increment, so do not put have full adder over here.

(Refer Slide Time: 53:02)



(Refer Slide Time: 53:17)



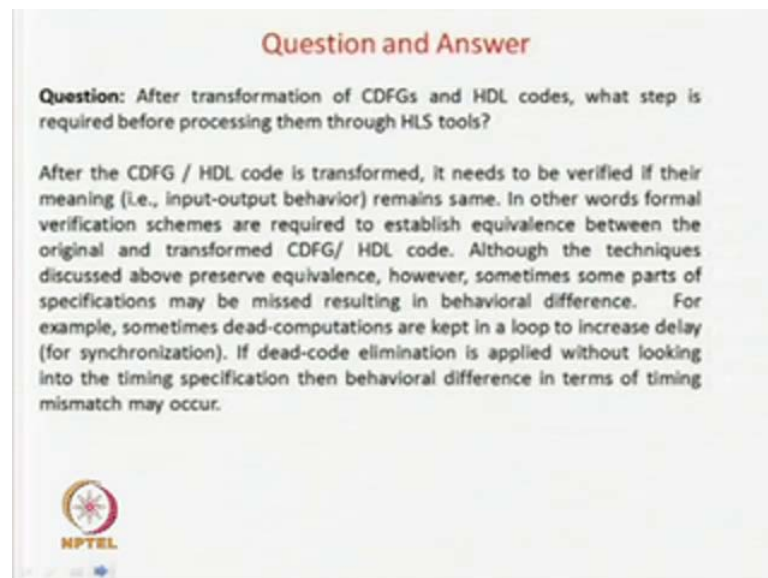
You just put an increment or a circuit will be smaller. So, this transformation is possible if and only if you have an increment in your library. So, this transformation or this betterment in your design is possible based on sub hardware, which is available in your

library. So, they are called library base transformation. We will take another example like say multiplication with A 0.

So, you have a computation like B equal to A plus 2. So, you always know that multiplication by 2 is nothing. You should not a put a multiplier because multiplier is very very complex hardware. Multiplying by 2 is nothing but you have to go for a left shift. For example, if you have 0, 0, 0, you want to multiply it by to it will be A, so it will be 0, 0, 0, 1. You go for 1 bit left shift.

So, if you have a left shifter in your library, you should put a multiplier, you go for a left shifter here. So, in this case, your hardware will be less and optimize. It is possible only if you have a left shifter in the library or you have to design it and put it in the library. So, these types of transformation are possible.

(Refer Slide Time: 53:51)



If you have are, what do you can called some specific stuff in your library, which is helping you or optimization. Do not using a general purpose comparator or general purpose multiplier, general purpose adder is not require, but some specific cases, some other circuits are available, which you can use it.

So, this was about all the transformations possible. As I told you, this is the manual step. You write your code before going into automated cad tools, which will be dealing in the next lecture. Once you say that you design is complete, it has no bugs or every

transformation has been done in the code is very optimized through, then go for automated tools for high level synthesis, gate level synthesis and so forth.

But, this stage as it is manual, so you have to apply all these transformation techniques, debugging techniques etcetera. Debugging technique, you will be looking more details in the verification module of the course, but very manual or rudimentary kind of a debugging is we apply some input patterns and see everything is proper or not.

So, once everything is done, then you can go for design of what do you can call high level synthesis. That is from this verilog on CDFGs, we will go for a block level architecture of the circuit. That we will be seeing in the next module of the codes. Here, it is very important because the codes are written manually. So, you have to go for transformations, so that you get optimized stuff in already.

So, before you stop because the question answer code questions a session. So, it says that the question is say simple question like after transformation of the CDFG is code, what is required? Why it is required to preprocess them through high level synthesis tool? Once it is required before preprocessing just like I was saying that a free you transform CDFG and HDL codes to some automaton like what you have already done. You have taken HDL code like that log. We have many CDFG. Then, in the next module, we say that we will be processing them to high level synthesis tools.

Then, what step is required? So, there are 2 things that I said HDL, and then these types of codes are transforming. These we verified that input output behavior remains the same that is you are verifying for characters that is your some input specification. From the input specification, you have developed the verilog code. Then, you have transformed it by logging out of optimization. So, transformation is very much required that is you transform them and optimize them and so forth.

Then, before you feed it to your optimizer, before your tool high level synthesis tools, then what step is required? That is actually the verification tool that is you have taken tools, you have taken your CDFG, you have taken your verilog code converting into CDFG by doing all these transformation. As we have seen, you may have given some inputs outputs and debug the code.

When you said this code is ready, now you can go for automated tool for high level synthesis get levels are fabrication. Now, you should be very careful here. You should go for verification that it is you should verify formally that is the specifications are equivalent always. Specifications are all satisfied by this stuff your delivering because in the process of transforming your code, in the process of optimizing your code, you may sometimes need to some kind of issues, which may lead to our specification values. In very simple example, I just told in the answer is sometimes you may put some kind of dead computation to increase the length of a loop.

This is because sometimes we may find out; we may give a loop, which has to be data synchronized with some other stuff. Now, further you might have put some dead code. So, that delay case unnecessarily increase or sleep kind of a thing in c language. Now, because we transformation, if you remove the code, then the delay will be decreased. Then, there will be synchronization problem.

So, in such case, even if the transformation is there to optimize your code, but in this case of because it may lead to non-synchronization, you may lead into problems. Therefore, before going for anytime of transformation debugging etcetera, finally, transforming your design, you should be very careful that there is no behavioral change from this specification.

So, that has been verified and formal techniques should do that. It will be landing, the verification part of the module. The next part of this course that is next module will be looking at the cad tools. I mean what you call CDFG, which is optimizing, correct specification of data. How do you go for architectural design directly using kind of tool? So, the algorithms required to do the conversion, we will be looking in the next module.

Thank you.