Electronic Design Automation
Prof. Indranil Sengupta
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture No #12
Synthesis: Part 5

So now we shall be talking about high level synthesis.

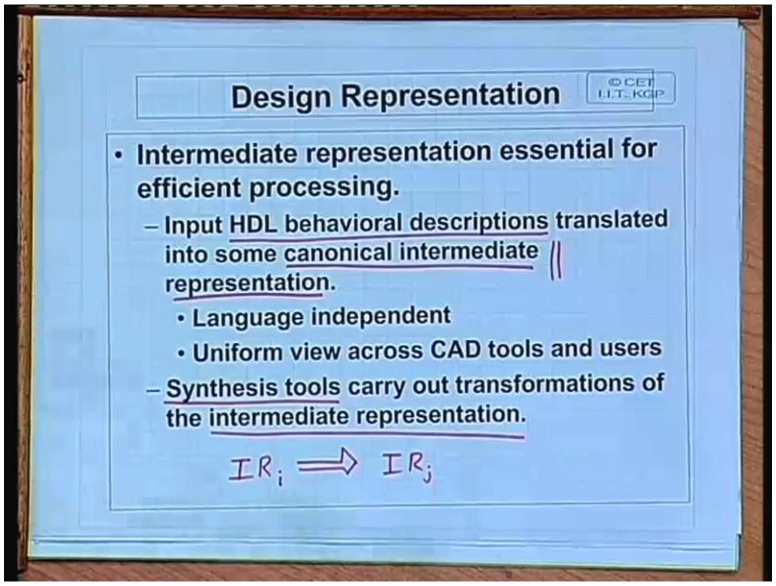
(Refer Slide Time: 01:13)

© CET
I.I.T. KGP

High Level Synthesis

Now as I had mentioned in our last class basically high level synthesis means that starting from the behavioral specification you are trying to arrive at a register transfer level net list for the given design. And in fact you will see that in order to do this transformation there are a number of different steps you have to go through. Most importantly you will have to understand what is the kind of intermediate representation or intermediate form that will lead to represent our design specification so that it is easy to manipulate. So first let us talk about the design representation.

(Refer Slide Time: 01:52)



Design Representation

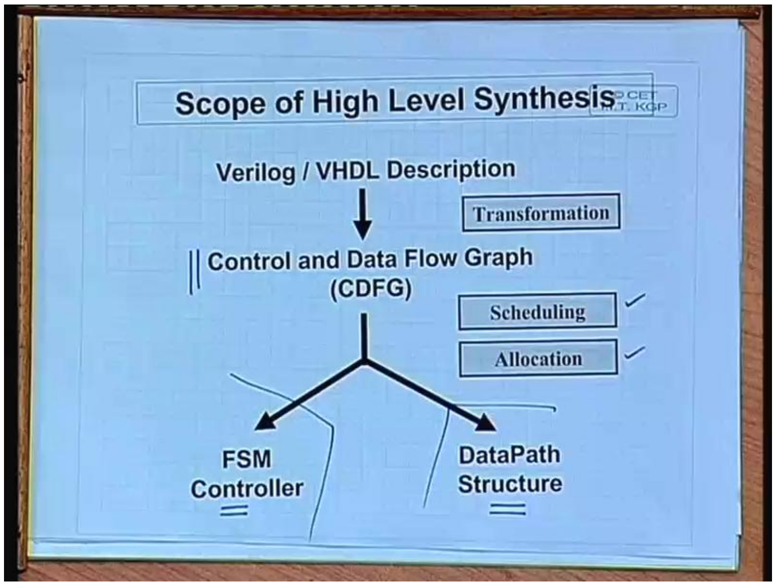
- Intermediate representation essential for efficient processing.
  - Input HDL behavioral descriptions translated into some canonical intermediate representation.
    - Language independent
    - Uniform view across CAD tools and users
  - Synthesis tools carry out transformations of the intermediate representation.

$IR_i \Longrightarrow IR_j$

We will try to illustrate with examples and subsequently we will see that using these design representations how we can do a number of different manipulations. So of course this representation will be essential for efficient processing for obvious reasons and all input that we are feeding is some hardware description language behavior description. It can be Verilog VHDL or any other language so our first step will be to translate this into some canonical intermediate representation. Okay. Which will be easy to process and efficient to represent. Now this canonical representation should be such that it should not be dependent on any particular hardware description language. This is of course very important from the point of view of portability across CAD tools. Suppose you have 1 part of the design coded in VHDL other part coded in Verilog. So if the intermediate representations are same you can mix the designs and you can synthesize them.

So this intermediate representation should be language independent and should be should give an uniform view across a number of CAD tools and systems and the typical synthesis tools which work on a design for synthesizing will be working on this intermediate representation and will be carrying out several transformations on them. What I am saying is that. Suppose I have an intermediate representation version i from there some CAD tool will be transforming it into another intermediate representation version j. The representation will be the same but will be carrying out a sequence of transformations on them so as to you can say improve the performance of the resulting hardware that will be getting finally. Okay. So let us try to briefly try to understand what is the scope of high level synthesis.

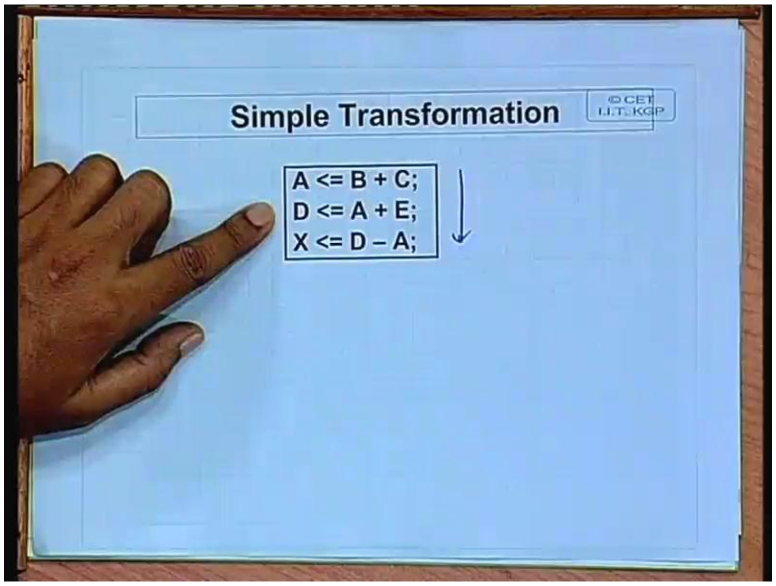
(Refer Slide Time: 04:04)


Scope of High Level Synthesis
Verilog / VHDL Description
Transformation
Control and Data Flow Graph (CDFG)
Scheduling
Allocation
FSM Controller
DataPath Structure

Now here from this diagram you can see that we start with a behavior specification in a hardware description language verilog VHDL or any other language. From this we come to the intermediate representation which is CDFG this is the most popular representation which people use control and data flow graph. See as the name implies it is not a single graph it is actually two graphs control flow graph data flow graph. But both of these taken together this is called CDFG. So this transformation step is needed at the beginning and from the CDFG there are a number of sequence of steps you need to follow through which you will be synthesizing the basic hardware comprising of the data path. That means the arithmetic logic unit bus registers etcetera and also the circuit which will be controlling them that means an FSM controller.

There are again several steps that have to be gone through the most important of these are scheduling an algorithm there are other steps also we will be talking about them later. So as you can see from the given description we will have to carry out a number of intermediate steps. Finally we will be arriving at a data path structure and an FSM controller. So at this stage we can say that our hardware has been synthesized. So we have the basic hardware ready we also have the control unit which will be controlling the hardware both are there. Typically from CDFG we separate out FSM and data path and synthesize them separately because the way they are synthesized or the kind of you can say the basic building blocks they use them use in them they are quite different. Okay so let us first take an example and illustrate how this intermediate form looks like and what is the control part of it and what is the data flow part of it all right. So let us take a very simple example.

(Refer Slide Time: 06:22)



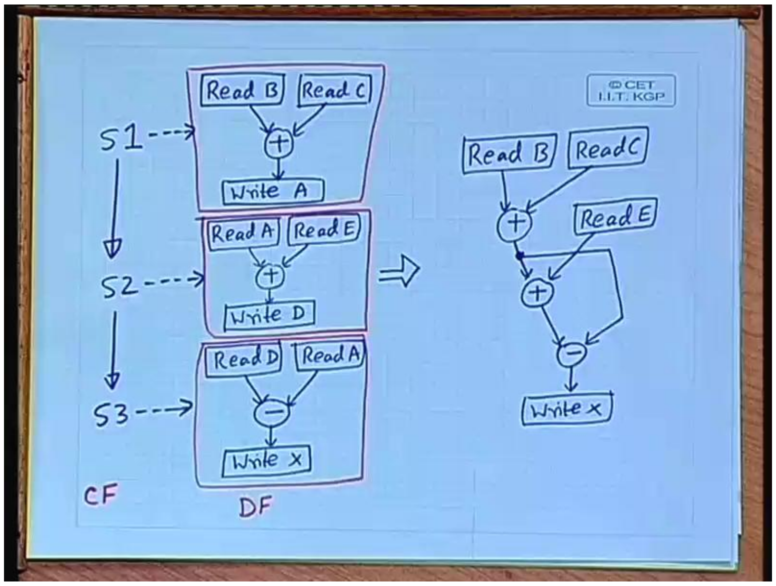
Simple Transformation

©CET I.I.T. KGP

```
A <= B + C;
D <= A + E;
X <= D – A;
```

Suppose there are three statements in a descriptive language I am not talking or concentrating on any particular hardware description language. This can be any generic description b c are two values typically storing registers they will be assigned to a, after adding a plus c assigned to d, d minus a to x. This is in sequence so the value of a, which is getting here that is used here d computed here will be used here. Okay. So now let us see how this particular thing can be mapped into a data flow description okay.

(Refer Slide Time: 07:05)



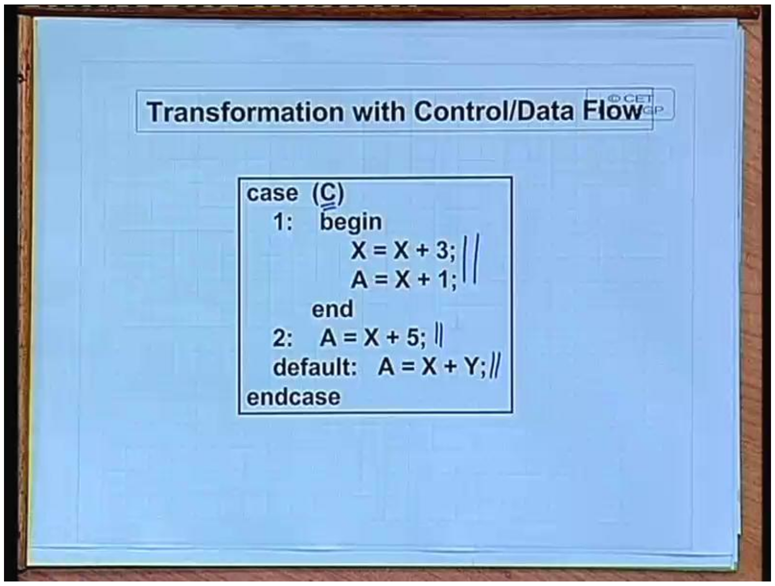
First I am trying to draw the data flow description. Data flow description will contain each of the intermediate steps which you need you see the first step takes b and c as input and compute a. So I am trying to show that part it reads the value of b and c it performs an addition and it writes it into a. This is the first statement this corresponds to the first statement similarly you have the second statement where the operands are A and E you do an addition again and you compute the value of D. So this is the second step. Similarly third one you see the third step is x equal to D minus a so in the third step you read d read a subtraction this is the operator and write into x. See these are the three statements corresponding to which these are the data flow graphs this is called the data flow graph. This shows how data flows through the operators and another thing these are the individual data flow graphs for the three statements but we have not told or mentioned anywhere that these three statements have to be executed in sequence. So in addition there has to be some kind of specification that this is statement 1 this is statement two and this is statement three and these statements must be executed in sequence.

This corresponds to s 1, this corresponds to s 2, this corresponds to s 3. So this simplistic description the portion on the left where we are saying that this s 1, s 2, s 3 must be executed in sequence. This is the control flow part of it here where we are talking about the actual data reading and writing from registers doing the operation adding subtracting this is the data flow right. So these two taken together will give us the total picture. See but I am also illustrating with respect to this example that during synthesis this intermediate representation is the first step. But this is not the end of this end of the story just you can carry out very simple manipulation on this. Keeping this control flow description in mind and carry out some optimization what I am saying is that this DFG can be optimized looking into the fact that s 1, s 2, s 3 has to be executed in sequence and you can get an optimized version like this I am showing this. B and c are read first they are fed into an adder okay this write a immediately you are reading so you need not store it anywhere intermediate you can straight away use the output of the adder as the input to that. So if you have another adder the first input to the adder can come straight away from the output of this adder the other input is read e and similarly this d straight away you can feed it to a subtractor.

Of course the output of the subtractor is x write x. But the other input is a; a is the same value which you have computed here so the same value out here this will also have to come here. So this is the modified and the compact data flow graph for the same problem. Now these are techniques which are already used by compilers and this are fairly standard techniques that you have several statements you just parse them try to minimize them. You try to minimize the number of variables you required for execution. It is like this this a and d are really not needed straight away the output of 1 can be fed to the input of the other. So this is a very simple example I have taken just assuming that there is a sequence of statements we have executed the data flow graphs and how we can go about constructing the data flow graph. (Student Noise Time: 12:53)

Yes, yes. We are using the control information for optimizing it. If the control stop said that these three are concurrent statements then we could not have combined it like this. So implicitly control information is also used to do carry out this minimization. (Student Noise Time: 13:16) This is still a data flow graph but this data flow graph also keeps you can say it is consistent with the control information which you have specified. Okay so this in a sense it combines the control and the data flow together (Student Noise Time: 13:39) No one after the other why we have drawn the data flow see if you simply follow these arrows. (Student Noise Time: 13:52) No I am assuming that the hardware description language in which you have written this specification which these statements mean that 1 after the other they have to be executed that is the assumption I am making. Now let us take slightly more complex example.

(Refer Slide Time: 14:09)

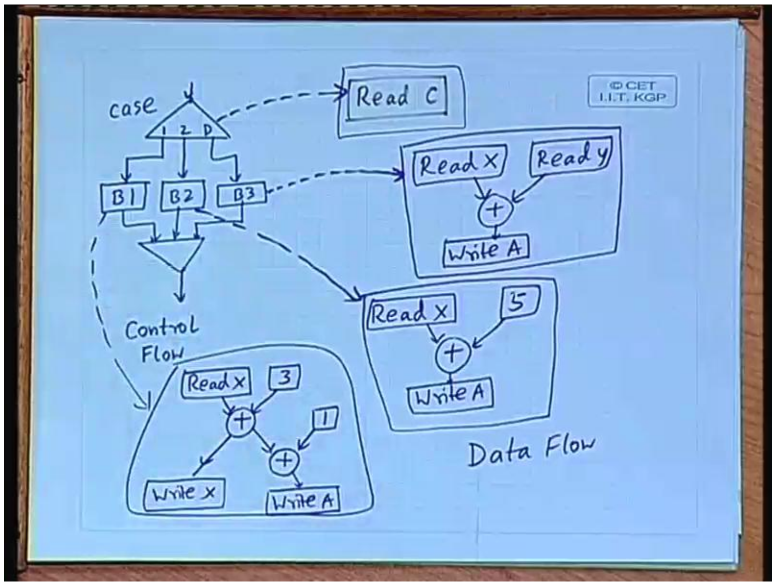


Transformation with Control/Data Flow

```
case (C)
   1:  begin
          X = X + 3;
          A = X + 1;
       end
   2:  A = X + 5;
   default:  A = X + Y;
endcase
```

Where there is some control not just simple sequence of execution just like if then else kind of things. Here we have a case statement and if the value of c is 1 you add three to a variable x you use the same value of x add 1 to it write it to a. If it is true a is equal to x plus five and otherwise a is equal to x plus y right now in this case you will have to explicitly specify the control flow graph because that is important. Because here the control flow graph will have to check for the value of c. Data flow graph will contain this information where the actual data is flowing register to register addition subtraction etcetera. But control flow graph will specify that I will have to check the value of c if it is 1 call that if it is two call that if otherwise call that. Okay so let us try to show that diagram first the case statement c will check 1 two or default.

(Refer Slide Time: 15:30)



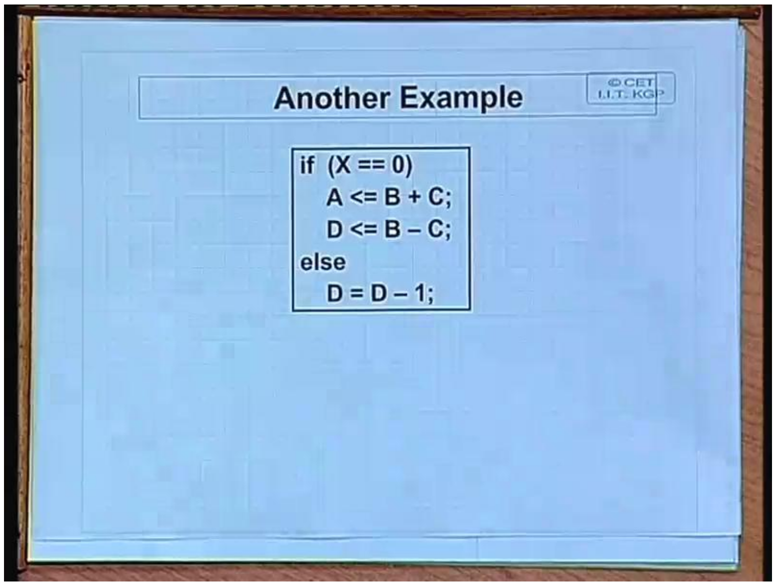
So this we are or this is typically represented like this this is a case block where there are three branches out 1, 2 and default these are the values. So if it is 1 you will be executing a block b 1 if it is two you will be executing a block b 2 otherwise block b 3. Now after that the control will be merging these three traces and it will be single trace of control it will merge. Now this is the control flow description now control flow description actually this will be checking the value of case so there will be some kind of a multiplexer or decoder out here. It will be checking the value of the variable c depending on the variable the value of c it will be generating control signals to activate b1 or b2 or b3. Now this is only the part of it this is not complete in order to complete it, we will also have to show the data flow part like the first data flow part will be read c.

This is one data flow description this will be linked with this. Because the case statement is working on c so we will have to read the value of c that is a data flow operation so that is one data flow block out here which is assigned to this node of the control flow block. Similarly I am okay I am showing for b 3 I am showing first b 3 is a equal to x plus y very similar read x read y add them up write a. So this is your b 3 so generating control signal for b 3 means you are activating this piece of the hardware. Similarly for b 2 it is a equal to x plus y read x a constant value 5 add them up right here. This is how b 2. Similarly for b 1, b 1 is a two statements x equal to x plus three and a equal to x plus one. So this also similarly you can have you can combine combining them straight away add x and 3 write x and the same value you can take directly from here you can feed it to another adder with a value 1 and you can write it to a. So this corresponds to b 1 so this part is the data flow graph this part is the control flow graph.

So actually the high level synthesis system will be first parsing your behavioral description and will be generating both the control flow and the data flow. Data flow graph once you have generated diagrams like this. There will be some processing on this to find out how many registers you require. How many arithmetic logic unit adder subtractors we require? What is the kind of hardware sharing you can use what is the total number of computation steps or the clock cycles you need that is the design of the data path? But once it is done the control unit this can be implemented as an FSM. The control unit will be simply generating the control signals to activate the data path right. So these two descriptions are generated concurrently so that they can be synthesized also together right. Yes (Student Noise Time: 20:24) One more control block means this one. This is not really a control block this is like a connector notation.

That means depending on some condition you are doing one of many things and after you are doing 1 of them you again come to the same point and from there you continue this is just a flow of control. Control flow there is a single thread of control which is there. So beyond the point there is a if then else kind of thing you go to 1 of them again you come back to the same point and continue from there yes anything else. (Student Noise Time: 21:09) Yeah, yeah. Because this case with this block with an arrow to it implies that the data flow block where you are giving the signal the value coming down on to it will be used here that is implicit. So this notation means something this is the case where you are implying something you are computing some value that value will be used to take the decision. There will be some comparators also here okay. So let us take another simple example.

(Refer Slide Time: 21:52)



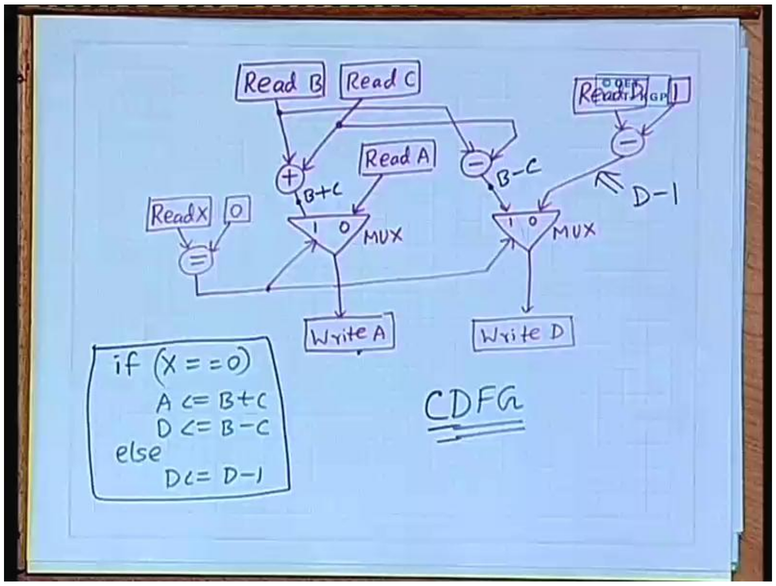
Another Example

```
if  (X == 0)
    A <= B + C;
    D <= B – C;
else
    D = D – 1;
```

So this is not a case this is a if statement if a variable x is 0 then you compute a equal to b plus c and d equal to b minus c otherwise d equal to d minus one. So this is also very similar in fact this is much simpler than the earlier example and I am showing the picture if x equal to 0 a equal to b plus c and d equal to b minus c. This path is shown on the right side here this is the x equal to 0 part and you look at the control flow part there is an if statement. So this is a condition evaluation now condition evaluation what is the condition evaluation it is if x equal to zero. Now x equal to 0 you can see this is also a part of the data flow read x 0 compare is the operator here. So the value of this is evaluated and this is used by this if block to take a decision the value will be either 0 or 1 false or true. So if it is true then you activate this block if it is false you activate this block right. So control and the data flow graphs they are being generated together side by side and what ever you have for the control there will have to be a segment of the data flow graph corresponding to each of them each of those nodes.

No, this if equal to 0 do this otherwise do this this is the block when you execute x equal to 0 this is a comparator the comparator will be returning true if x is 0. It will be returning false if x is not equal to 0. Okay, so this is how a simple if statement in the behavior description can be handled. Now there are certain ways in which you can optimize the piece of hardware and we can get you can say a more efficient read. Suppose you consider the same scenario the same example there are three blocks 1 is a comparator block this is a computation block or subtractor this is a computational block adder subtractor. But suppose we want to combine these three together, so this combined together how we do this. So actually what we are trying to do is that we are trying to well again the control flow description was different data flow description was defined. Now we are trying to synthesize I am showing you typically the output of the synthesis that, how this control and data flow can be taken both together and register level net lists can be generated. So for this particular example I am showing the final diagram.

(Refer Slide Time: 21:52)

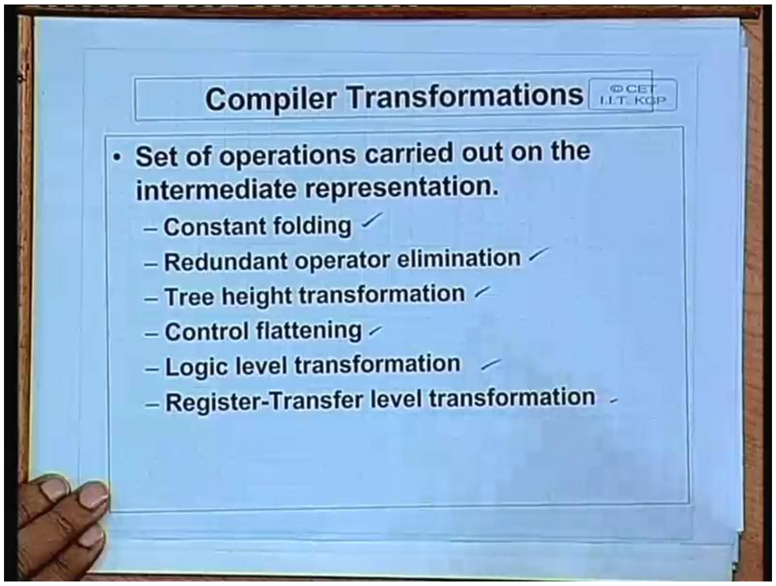


```
if (X = = 0)
    A <= B+C
    D <= B-C
else
    D <= D-1
```

Just for your reference this was the description. Now in the earlier one you see this was the comparison. Okay, this is there read x 0 compared with this and the else part there is a d equal to d minus 1. This is being computed here read d subtract 1 from it. So this computes d minus 1 okay. Similarly if x equal to 0 you have an equal to b plus c d equal to b minus c b plus c so here you have computed something which is the value of plus c and b minus c you are computing here this is b minus c. Right. Now the synthesizer will come to know that if x equal to 0 the value of a plus b has to be assigned to a, and there is no conflict here no other assignment to a. So you put a multiplexer here depending on the result of the comparison if it is true 1 then this b plus c will be stored into a. But if it is false see this description says if it is false I am not assigning anything to a. So actually as part of the data path either there are two ways either you disable write a, or you just introduce a dummy read a through this multiplexer write back the value of a both way can be used. Similarly for the other path for writing d if the result of the comparison is true then you write b minus c otherwise you write b minus 1.

So this diagram essentially combines both control and data in fact this is CDFG control data flow graph both together. So this I have drawn just for the sake of convenience actually in the implementation these registers will not be scattered like this there will some register bank like kind of thing there will be some reads and writes generated for that. It will be more complex but just for understanding the basic concept I have shown you these examples now one thing I told you that after this data flow or the control flow graphs are constructed. Well control flow graph will reflect the control information now the data flow graph or the different segments of the data flow graph can be combined using control flow information as this example I have shown because earlier we had different segments of the data flow graph. Using control flow information we have been able to combine them into a single one. Now after we have done this we can carry out several transformations on this in order to optimize. Now in fact these kinds of optimizations are fairly standard and are already used since a long time by compilers. So we call them compiler transformations we will see what kind of typical transformations are carried out.

(Refer Slide Time: 29:13)



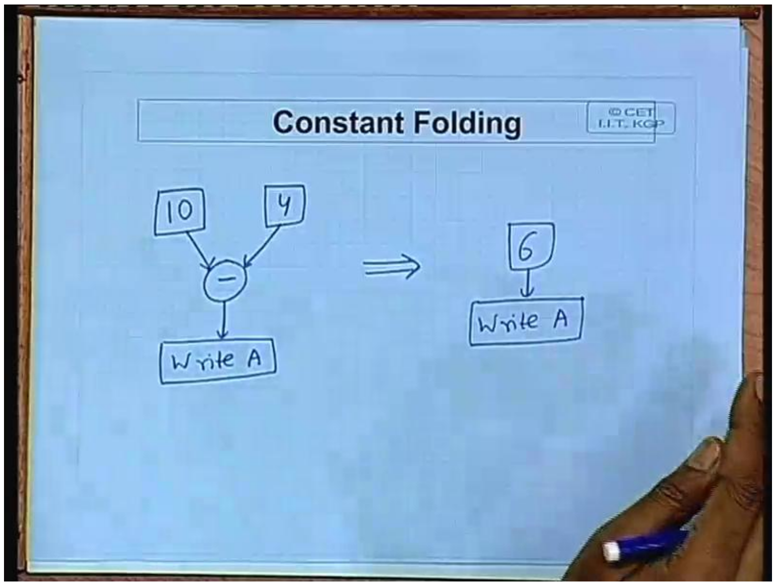
Compiler Transformations

©CET I.I.T. KGP

- Set of operations carried out on the intermediate representation.
  - Constant folding ✓
  - Redundant operator elimination ✓
  - Tree height transformation ✓
  - Control flattening ✓
  - Logic level transformation ✓
  - Register-Transfer level transformation

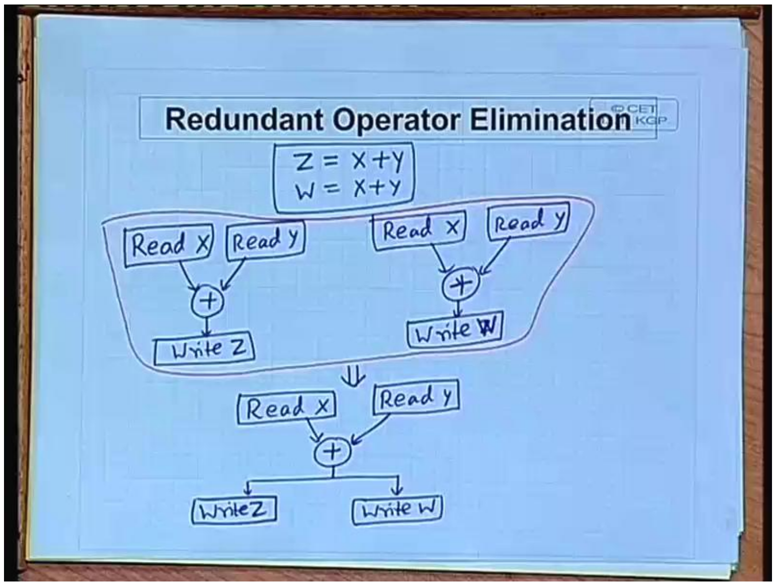
Now as you can see that some of the transformations are constant folding redundant operator elimination tree height transformation control flattening logic level and register transfer level transformations. So these are fairly standard things which are applied on a given CDFG in order to get another CDFG which is better. So intermediate representation still remains CDFG okay CDFG to CDFG transformation. Okay. First let us try to explain constant folding. It is a very simple thing.

(Refer Slide Time: 29:50)


Constant Folding

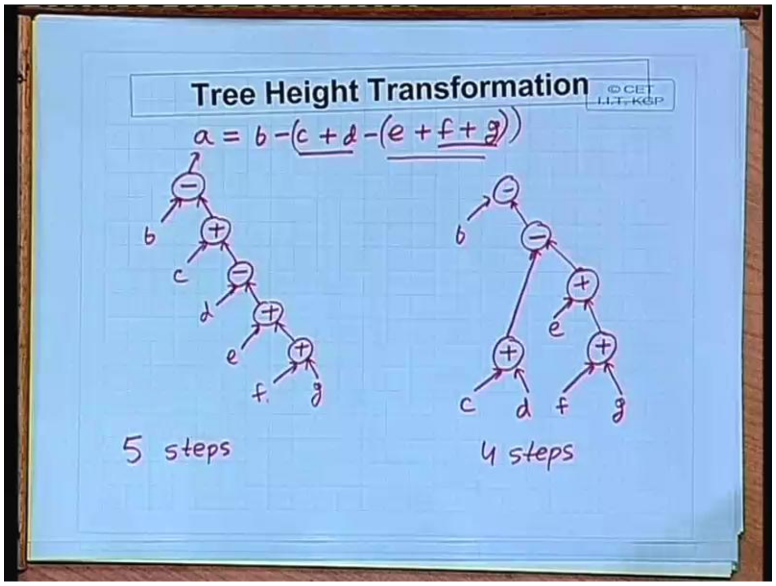
It says that in many cases you will see that there is a portion of CDFG where there are some portions which can be immediately evaluated. There is a constant ten and constant four suppose there is an operator subtraction which you are writing somewhere. So the processor can immediately find out that there is an operator for which the operands the values of which are already known. So, it can evaluate it at the compile time itself and straight away you can transform into a constant value six which you are writing into a. So you wrote in a subtractor. So you do not need places to hold ten and four straight away you can write six into a simple. Redundant operator elimination.

(Refer Slide Time: 30:56)



Well this already we had illustrated in an example earlier. But let me show you again with an example. Suppose you have two statements z equal to x plus y and w equal to x plus y. So obviously you can see that this x plus y is a common operator so with respect to the DFG the kind of transformation that will take place is this. In the original diagram it will be something like this write z this is for the first one and for the second one there will be again a similar block generated sorry write w. Okay so these will be the graphs which will be generated initially before any optimization steps but the processor can find out that there is a portion of the tree read x read y plus read x read y plus these are identical. So it can identify this redundant operator common operator or common sub expression what ever you call and it can transform this graph into another which will be like this read x read y and write z write w. The same value you are writing in both places. So this is one standard method which is used fine.

(Refer Slide Time: 33:30)



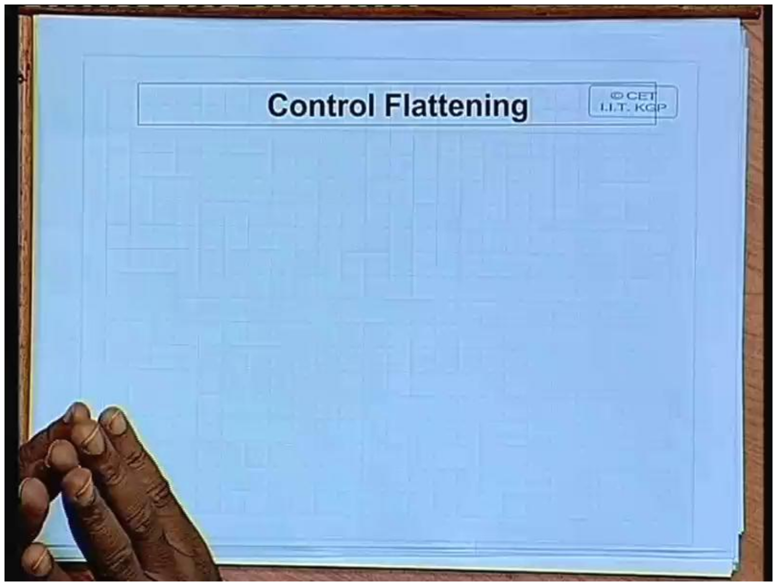
Tree Height Transformation

There is another method which is called tree height transformation. This is actually important from the point of view of efficiency in execution of the synthesized hardware. See this also I am explaining with an example suppose I have a statement like this an equal to b minus c plus d minus e plus f plus g. Suppose corresponding to this well I am just showing the DFG after parsing the DFG is constructed say the DFG has been constructed like this. I am just showing the DFG I am not showing the entire part of it subtraction I am showing in the reverse order okay b addition then c again subtraction d again addition e another adder f and g. So in this way the computations are carried out f plus g plus e d minus this c plus this b minus this so this finally will go into a. Okay so this okay this in terms of the intermediate computation how many steps do you need for this of course you need 1 2 3 4 5 steps.

So this will need five computational steps because the way you have drawn the computation tree the computation has to be done in sequence. (Student Noise Time: 35:31) Yeah. Okay. Yeah, like this. Yeah this fine e plus f plus g together d minus this again here, here, yeah right fine. The other alternative what you can do? We can try to find out what are the things that you can do in parallel in this example for instance you can see f plus g and c plus d u can do in parallel. So we can have an adder with c and d we can have another adder with f and g these two can be done in the same step. So we have done this we have done this. So now we will have to add e to this we need another adder with we have computed this this minus this. Similarly lastly there will be another subtractor b minus this. So here we have been able to save 1 step four steps now in general the number of steps can be reduced even more.

But one thing you try to understand here because this is also very important part of high level synthesis. The point to note is that by rearranging the way we are carrying out the computation we have been able to reduce the number of steps. But in the process are we having to pay something more? (Student Noise Time: 37:47.062) is fine. One thing you note in previous case since the operations had to be done in sequence then we could have used a single adder subtractor hardware for the purpose. But here we would be needing two parallel hardware's two adders are needed but we can save time. But we will have to use multiple copies of computational hardware so this is the price to be paid. This is some kind of a trade off that if you need higher speed you will have to invest more hardware. So we will see that there are some trade-off involved. There is a step called control flattening of course.

(Refer Slide Time: 38:41)



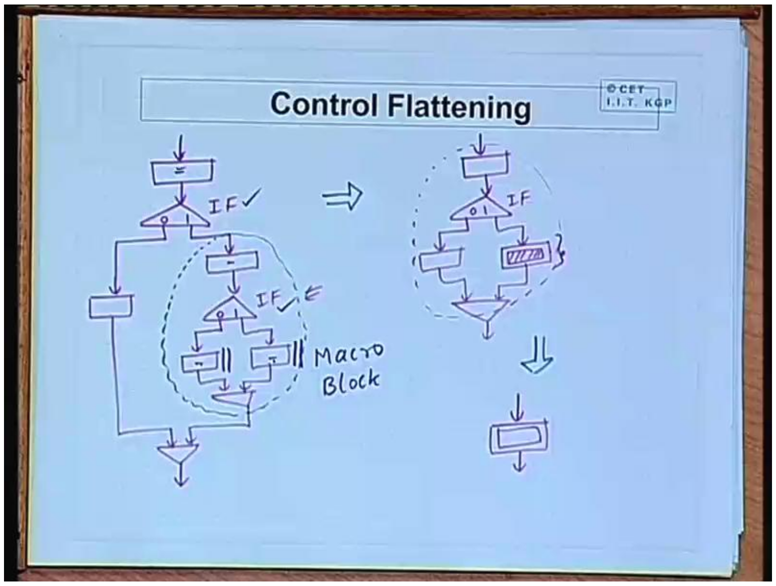
Control Flattening

© CET I.I.T. KGP

See this step of control flattening is useful only for the purpose of control unit design. The idea is simple see when ever you have a very big system or circuit and you are trying to design the control unit for it there are two approaches. One is that you generate the control specification for the whole circuit and you start or you go about generating the FSM for generating the control signals all in one go. But in many systems particularly for large system processors you will find that there is a hierarchy of control which comes very naturally like I am giving the example. Suppose at the highest level you can say that in order to perform a task I need these four things to be done in sequence. So let us generate four control signals for these two, one after the other.

Now the third control signal for example this is meant to excite a piece of hardware which in turn is a slightly a complex system which requires four steps internally. So those four steps I could have embedded in my original control specification also. But as an alternative I can also have some kind of a hierarchy that in the high level description there are four steps but within the third step there are again four sub steps. So if we can specify it like this our control specification and implementation may become easier otherwise our control specification will become very huge and we can end up in designing 1 single very big FSM or instead several smaller FSM's may be more efficient both in terms of area and execution. So control flattening I am illustrating with an example this I have the diagram here.

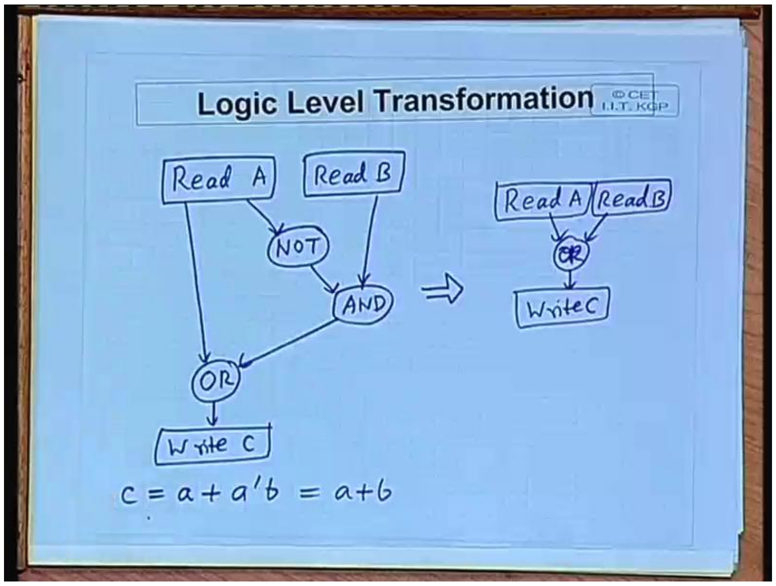
(Refer Slide Time: 40:36)


Control Flattening

Suppose we have a control flow graph like this. But this is not really a control flow graph I have actually shown both of them also data flow computations are shown here. Suppose this is a data flow computation block after that there is a if statement depending on the outcome of if either you come here or come here. So here again there is some data flow block another if statement data flow block here data flow block here another if statement you come out. So you can take this entire graph at 1 go and try to generate the control signal but for that purpose you try to understand what will happen when we are trying to generate signals to activate say these blocks you will have to know the outcome of this if statement you will have to know the outcome of this if statement you will have to AND them up together. Then you will have to generate the signal for the multiplexers what ever more you have nested if's and those things this problem will become more and more complex. So it is not just 1 if there are actually several conditions you actually need to see together in order to decide whether you have to do this or do this.

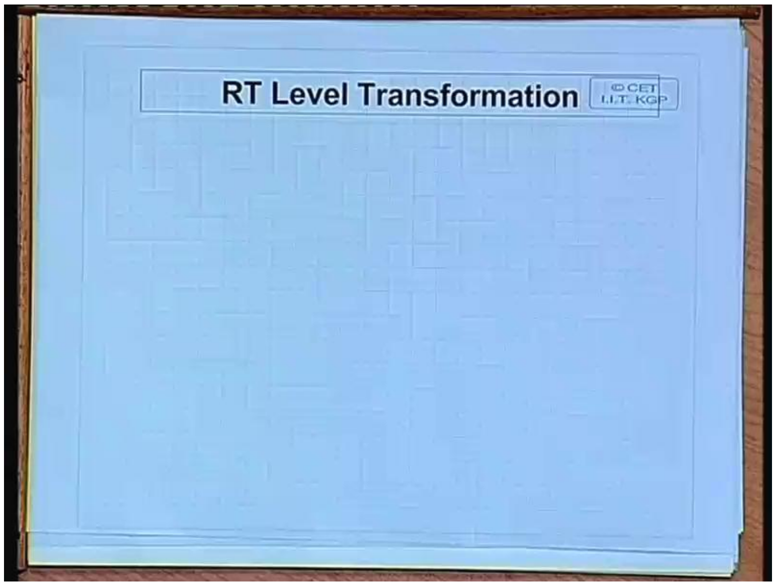
Because, ultimately this is a piece of hardware which has to be activated using a single control signal just unlike a program execution where you can have several if statements. So what you can do you can define this sub block as a some kind of a macro block and at the high level you can treat the diagram as like this. This is a macro block so if the if statement is true you activate the macro block. So at this level your control becomes very easy but inside the macro block there will be another level of control checking of this and activating this will be inside this macro block. So actually you are now having some kind of a hierarchy of control signals being generated in a hierarchy of modules this helps in many cases instead of having a very big. See bigger the size of FSM more number of flip flops you have to need to store the state more amount of hardware will be need to generate the control signals out of those states. But if you have this hierarchy which can come naturally from a description it can help a lot. And you can have some kind of logic level of transformations.

(Refer Slide Time: 43:32)



Which of course is fairly trivial like I am giving a very simple example. Suppose in a portion of the data flow graph you have a segment like this say you are reading the values of two values a and b. Suppose you are doing a logical NOT on 1 of the values then you are doing a logical AND on this value and b then we are doing a final OR on this and this. So generate say a value say c suppose this diagram came out directly from your hardware description. So actually what you tried to specify was c equal to a OR a prime b now the synthesizer can very easily find out this expression optimize it. It knows that this is equal to nothing but a OR b and it can simply modify this diagram as sorry this is OR. So this kind of simple logic level transformation carried. But as you can understand for this purpose you need a logic optimizer like Socrates or MIS which you had already mentioned. This will be also part of the story the synthesizer will be invoking them to optimize the logic part of it to get an optimized version of it okay.

(Refer Slide Time: 45:37)

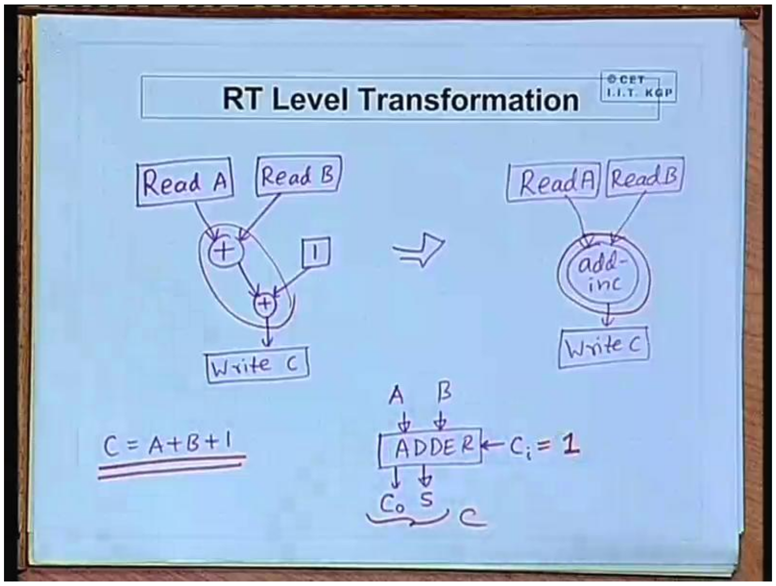


RT Level Transformation

© CET I.I.T. KGP

And finally we can carry out some register transfer level transformations. Now let me illustrate this with an example which you have already in the diagram. So register transfer level transformation means see there can be some peculiarities or some special functionalities of the basic hardware means you are using and you can utilize them to get a better realization like you look at this example.

(Refer Slide Time: 46:17)



Say in this example you have two operands a, and b which you add them together finally you add 1 to it and write it to c. So you are actually trying to compute a equal to c equal to a plus b plus 1 so if you want to directly map this into hardware you will be needing two adders. The first adder will be adding a and b the second adder will be adding 1 to it well either you do this or you do it in two different steps you are doing in the same adder. But the synthesizer will observe but will the adder that we have that also has a carry input. So if I say the carry input to the 1 then this addition by 1 is taken automatically this can be automatically taken care of. So I do not need this additional step of adding 1 by using a separate function unit so rather these two adders I can combine by a special operator add increment which also I can implement using the available adders.

So this can be transformed into an add increment order kind where you will be add increment write to c so in this case the carry input will be actually set to 1. It will be coming and the output this will be c; c will be computed so this kind of transformations as you can understand are very much dependent on the capability of the available hardware with the synthesizer we will be using right. So today in this lecture we had tried to give you some idea regarding the CDFG and what kind of transformations can be carried out to optimize the CDFG. But we have not yet talked about the systematic procedure. But one may follow or use in order to synthesize the hardware starting from the CDFG. This discussion we shall be starting from our next lecture thank you.