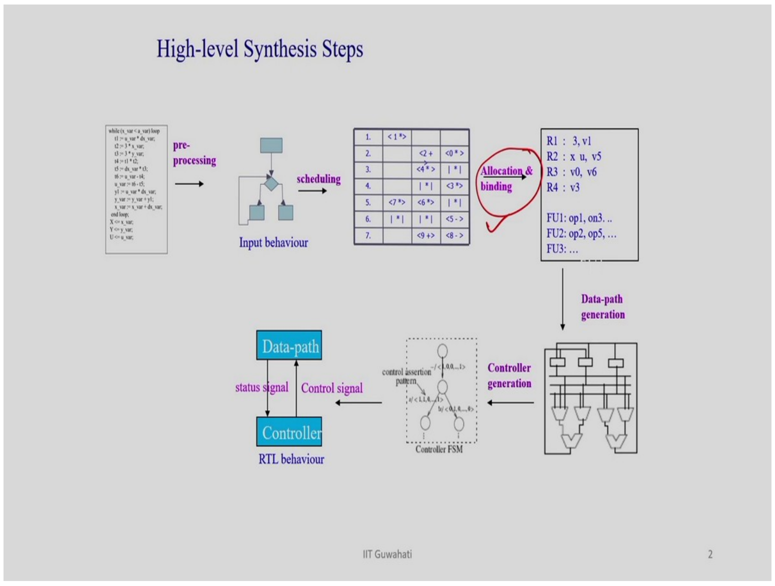
C-Based VLSI Design
Dr. Chandan Karfa
Department of Computer Science and Engineering
Indian Institute of Technology, Guwahati

Module - 06
C-Based VLSI Design: Allocation, Binding, Data-path and Controller Generation
Lecture - 20
Register Allocation and Binding

Welcome everyone, in today’s class, we are going to discuss about Register Allocation and Binding.

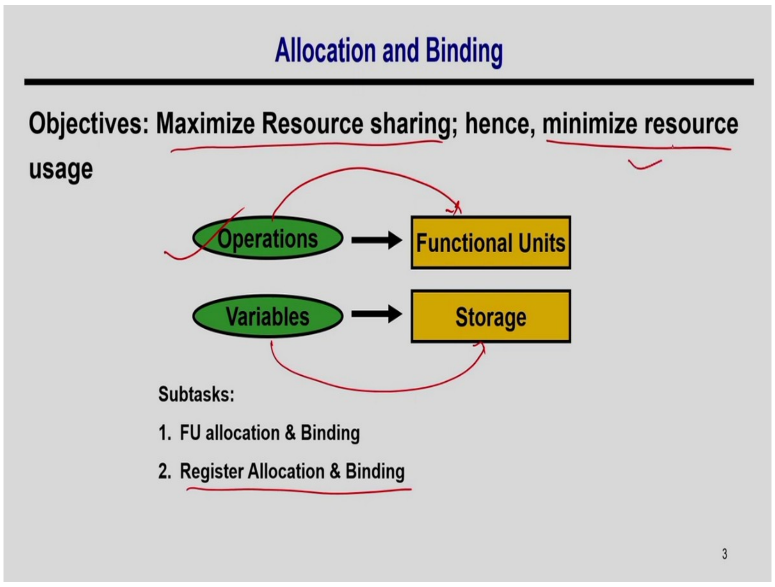
(Refer Slide Time: 01:00)



So, if you remember the high level synthesis, which converts a C code to an equivalent register transfer level behavior, consist of various phases like preprocessing, scheduling, allocation binding, data path ,and controller generations and finally, generates the RTL, right. So, we are now in this phase of allocation and binding; so that means at this particular point, we have that scheduled behavior is available to me.

So, we know exactly in which time step which operation is going to be scheduled, that is already done and is available to us and what we have to do now; I have to map those variables to the registers and operation to the functional units. So, that is what is called allocation and binding. So, that is the part we are discussing.

(Refer Slide Time: 01:39)



Allocation and Binding

Objectives: Maximize Resource sharing; hence, minimize resource usage

$$\begin{tikzcd} \text{Operations} \arrow[r] & \text{Functional Units} \\ \text{Variables} \arrow[r] & \text{Storage} \end{tikzcd}$$

Subtasks:

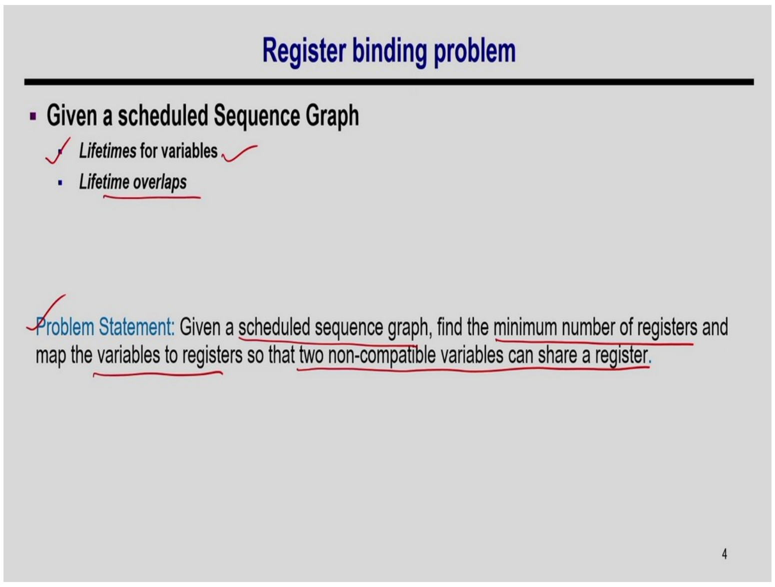
1. FU allocation & Binding
2. Register Allocation & Binding

3

So, in allocation and binding, our objective is to map the operation to the function units and operation to the variable to the storage; it is basically registers and RAM, ROM. So, we have already discussed about this operation to function unit allocation and binding. In today’s class, we are going to specifically discuss on this register allocation and binding problem.

That means, how I can map these variables of the behaviour to the registers with an objective is to minimize the number of register usage, right. So, that means our objective will be used to maximize the resource sharing or register sharing and to reduce the minimum use of the registers, ok.

(Refer Slide Time: 02:23)



Register binding problem

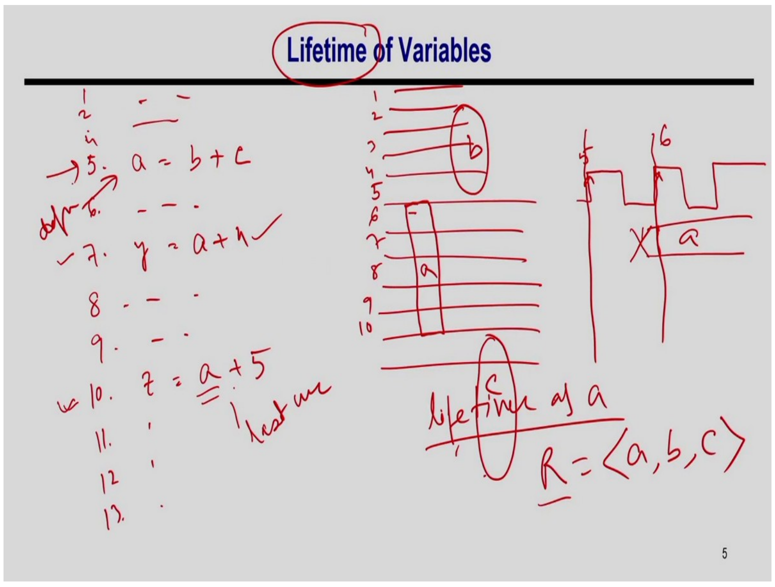
- Given a scheduled Sequence Graph
  - *Lifetimes* for variables
  - *Lifetime overlaps*

Problem Statement: Given a scheduled sequence graph, find the minimum number of registers and map the variables to registers so that two non-compatible variables can share a register.

4

So, let us try to define the problem first, right. So, what is given to us? We have a given a scheduled sequence graph, ok. From that what we can do? We can identify the lifetime of the variables.

(Refer Slide Time: 02:43)



Lifetime of Variables

5

So, let us try to understand, what is a lifetime of a variable? So, let us consider a behaviour like this say; suppose in time step 5, operations a = b + c schedule and say operation 6 some other operations is there; in time step, 7 it is say some y = a + x and then say time step 8 something is there, 9 something is there.

In time step 10, again say z = a + 5 say right and then you have time step 11, 12, 13, where some other things are there and you have time step 1, 2 to 4, where some other operations are there, right.

So, our objective is to identify what is the lifetime of a variable, right. So, that is something we try to define. So, in a C program, whenever you define a some variable say a = b + c and you have some value defined, your particular variable always stores this value a for the rest of the program.

If I am assuming there is no function call or within the function body, the value of the variable is available, right. But in hardware, once you think about this hardware. So, the point where I define ‘a’ and the point where I am using the ‘a’ last time, it is the space or the time where I need that particular variable, right.

For the rest of the time, I do not need that variable, right. So, what is the lifetime I will define? So, at time step 5, a is defined; because it is the first line where a is defined, I assume these operations are not related to a.

So, the rest of the operation where the a is not either defined or use, I just keep them blank, right. So, that means, so I know that the variable a defined at time step 5, right. So, what is happening here is, at time step 5, it is getting defined. And so, if you just think about that that way. So, if you just think about the clock, right.

So, if it is if this is the 5th clock. So, this is my 5th clock, right. So, let us try to understand the use of ‘a’ in this particular program, right. So, a is defined first time at time step 5 and it is again used in time step 7 and in time step 10, right.

So, and the rest of the time step, I some operation which are not related to ‘a’ are there, so I am not writing them for brevity, ok. So, that means, a is getting defined at time step 5. So, what you happen in hardware?

So, this 5th clock comes, because it is the 5th clock and then the whatever the value of a b and c, that will go to the a FU that adder unit and it will do b + c. And the results of a plus b will be available kind of at this point, right. So, by end of this clock, the value of a will be ready the a=b + c, right.

So, now whenever the 6 clock comes, so from that point this value a will be available to use, right. So, if you just think about this way. So, the value of ‘a’ is defined from this point, right. So, for clarity of presentation or solving the problem; we can assume that every value of the lifetime of a start from at the end of time step 5 and hence from the time step 6, right.

So, similarly, so I can say that this value of ‘a’ is starting from time step 6 to time step t right; because it is getting used in 7 and time step 10. And in time step 10, it is getting used; it means that it has to, the register has to hold the value of ‘a’ throughout the whole clock of time 10th clock, otherwise, this adder will get some wrong value, right. So, we will consider the lifetime of ‘a’ is from 6 to 10; because it is exactly defined at the end of time step 5.

So, for clarity, I will do that from time step 6, ok. So, what is, so this is the lifetime of variable a, right. So, this is lifetime. So, now, what is the scope, right? So, what is our objective first of all? I want to map these variables to registers. So, the basic solution obviously, you map each variable to individual register; but that does not give you the best solution, because then if there are twenty variables, you have twenty registers in the hardware.

But we what are the whole purpose of this discussion is, can I use less than twenty register to store all these variables, right. And to do that, what we define the term is called lifetime and the lifetime is the term that is exactly the time where the value is first time available or the define to the time step where it is exactly used last time, ok.

So, it may use many times in between, but this is the last use, right. So, this is the last use, right. So, this is the definition. So, the difference from this definition to the last use is the lifetime of the variable. But, how does it matter?

It matters because now I can store this value of ‘a’ in a register in the hardware only between 6 to 10 in this time frame; the rest of the time I can use that particular register to store something, right. So, if some variables say ‘b’ is getting defined at this places, so I can store ‘b’ and say some other variable which is getting defined at here ‘c’. So, I can use a single register R to store ‘a’, ‘b’ and ‘c’; because the lifetime of variable ‘a’, ‘b’, and ‘c’ is non-overlapping, right. So, that gives the advantage.

So, if two variables' lifetime is not overlapping, I can store these two variables in a same register in the hardware. And what is the implications? Although in C program the value of 'a' remains there throughout the program; but its hardware, since it is a single register which is storing either value of ‘a’, ‘b’ or ‘c’, the register contain the value of ‘a’ only between 6 to 10 time step and from 2 to 5 in that time step, it will store the value of ‘b’.

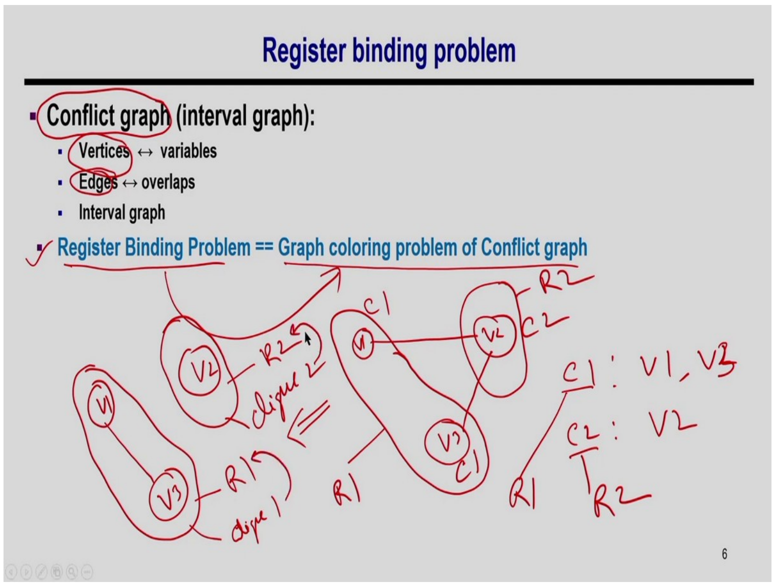
So, after 10, the value of ‘a’ is not available in the hardware; but it does not matter, because it is not going to use anywhere after that, right. So, in that sense this is perfect and that is where we have a scope to share register, ok.

So, if I now can define the problem, the given a schedule sequence graph I, our objective is to find the minimum number of registers and map those variable to the register, such that two non compatible variable can share a register, right. What does it mean that, compatible means two register, which whose lifetime does not overlap. So, here ‘a’, ‘b’ and ‘c’ are compatible, right.

So, our objective here is to find the minimum number of registers and map the variable to the register, such that two variable which has overlapping lifetime should not share a register, right. In other way if two variable whose lifetime is non overlapping, can share a register right, this is the problem statement.

So, from the sequence graph, we will identify the lifetime of the variables and we will identify their overlapping things and from there I can identify the variables which can be mapped to a register; if their lifetimes are not overlapping right, this is the overall problem, that is what we are going to solve.

(Refer Slide Time: 11:12)



Register binding problem

- Conflict graph (interval graph):
  - Vertices ↔ variables
  - Edges ↔ overlaps
  - Interval graph
- Register Binding Problem == Graph coloring problem of Conflict graph

Now, the question is how we can solve this problem, ok? So, we will show that this register binding problem can be solved using the graph coloring problem of conflict graph; it is similar to the FU allocation and binding problem, but we will explain how this problem of register binding can be mapped to graph coloring problem, ok. So, what we can do? So, for each variable, so if just like this; from each variable, I can construct a node, right.

So, I can construct a node or vertices and then if two variables' lifetime overlap, I will add an edge, right. So, if there is a variable say v_1 and there is a variable v_2, say there is a variable v_3 and say v_1 and v_2's lifetime is overlapping and v_2 and v_3's lifetime is overlapping, right. So, I know what is their lifetime and from their lifetime, I can construct this graph and this is called conflict graph, ok.

If this conflict graph, we can construct, then what is the problem? Our problem is to map these variables to the registers, a minimum number of registers, right. How can I do that? Now, you see this is the conflict graph right and what I am saying, if there is an edge between these two nodes; that means these two particular variables cannot be mapped to same register.

So, I should use different color for them, say suppose I use say color 1 and this is color 2; because I can use two different colors, because they cannot be mapped to this. And now for this v_1, I cannot use color 2; but I can use color 1, because there is no edge between this.

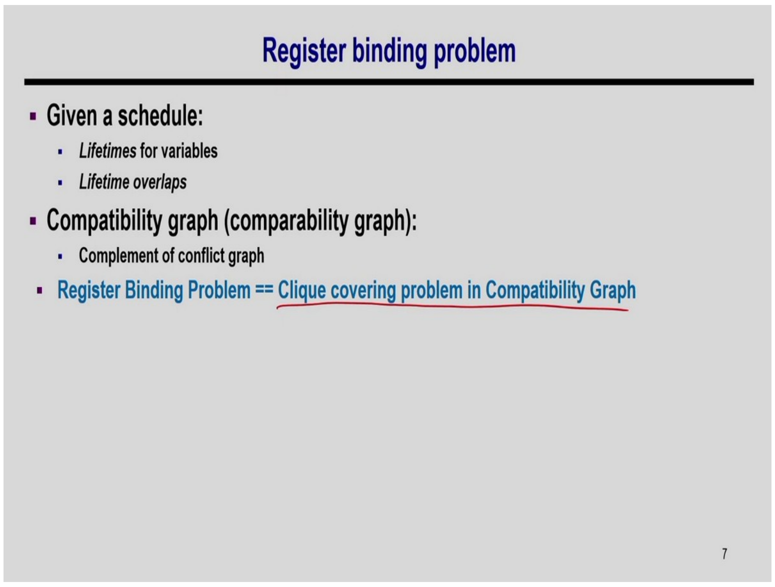
So, how many colors are needed? Two colors, right. So, in C_1 color is v_1 and v_2 and for color 2 is the color of node v_2. And what is this? So, basically it says that, this v_1 and v_3 sorry this is v_3; v_1 and v_3 have the same color and since their lifetime is non-overlapping, so they can be mapped to same register. So, this can be R_1, right. And similarly, since these two nodes has a conflict with this, this should be mapped to a different color or register, this is R_2.

So, I can understand that C_1 is nothing, but R_1 and C_2 is nothing, but R_2, right. So, in that way from the lifetime of the variable; I can construct the conflict graph, where each node represents the variables of the program. And if their lifetime overlaps, I have a edge in this particular graph and this is how I can construct the conflict graph.

And then coloring the nodes with a minimum number of color, such that two nodes adjacent to each other, should have a different color right; that is what is your graph coloring problem, use minimum color to color the nodes of a graph, such that two adjacent node have a different color.

So, identifying that minimum color is equivalent to finding the minimum number of register, right. So, what I have shown here that, this register binding problem can be mapped to graph coloring problem of conflict graph, ok. So, this is how I can solve the problem.

(Refer Slide Time: 14:38)



Register binding problem

- Given a schedule:
  - *Lifetimes* for variables
  - *Lifetime overlaps*
- Compatibility graph (comparability graph):
  - Complement of conflict graph
- Register Binding Problem == Clique covering problem in Compatibility Graph

7

Similarly I will show that this register binding problem can also be mapped to clique covering problem of compatibility graph, right. So, let us consider again. So, now, I am going to construct a complement graph of the conflict graph, right. So, this is my conflict graph. So, what is the complement graph of this?

So, the say v_1, v_2 and v_3 and I am saying that v_1 and v_2 is overlapping, so there is a edge here. So, I will have only edge this right; because these two are compatible to each other, so that means their lifetime are not overlapping, right.

So, this is the complement of this graph and this is the compatibility graph, right. And then what is clique here? Clique is the complete sub graph in this graph, right. So, what is the maximum complete sub graph? This is the one clique right, which is maximum complete sub graph of two nodes ok and this is one more clique.

So, what I am doing here is, basically in the compatibility graph, I try to cover all the nodes using a clique, right. So, I want to cover all the nodes of the compatibility graph using minimum number of cliques, right.

So, what I am going to do? I am going to find out the maximum possible clique size, after that for the rest of the node I try to find out the next maximum possible clique and so on; this way I am going to cover all the nodes. And what is that? Since these two nodes are compatible to each other; that means their lifetime is not overlapping, I can map this to register 1 and this I can map to register 2.

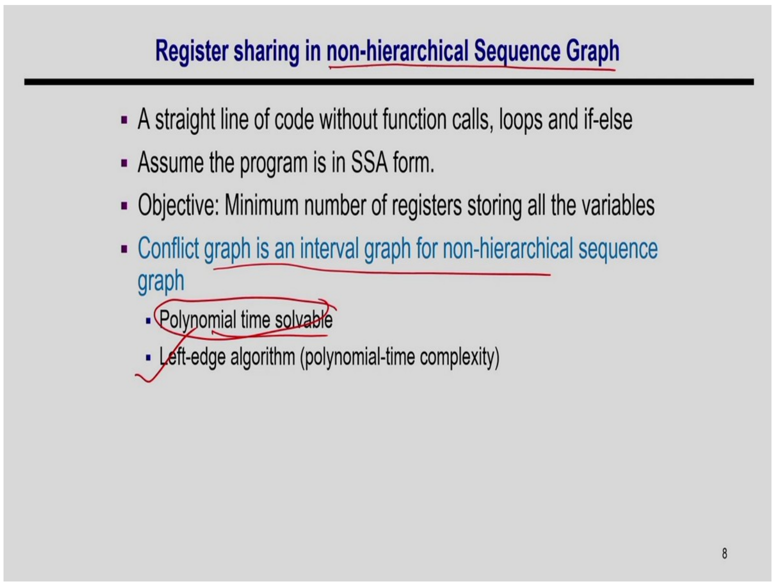
So, this is my clique 1 right, see clique 1 and this is my clique 2. So, clique 1 is nothing, but register 1 and clique 2 is nothing, but register 2. So, what I have shown here that, register binding problem can be mapped to the clique covering problem of compatibility graph, right. So, this is how I can solve the problem, right.

So, in general the solution strategy is like this, you take the scheduled sequence graph, you construct the lifetimes, you identify the lifetimes of the variable; from the lifetimes of the variable, you construct the conflict graph or compatibility graph.

And then in the conflict graph, you solve the problem of graph coloring, you try to color the conflict graph with minimum number of colors and that each color gives one registers and the nodes having the same color will map to the same register.

Similarly, I can take the compatibility graph; in the compatibility graph, I try to solve the clique covering problem. Hence, each clique will represent one register and the nodes that belong to a single clique, will be mapped to that particular register, ok. So, this is how I can map or solve the register allocation and binding problem using either graph coloring problem or clique (Refer Time: 17:43) partitioning problem, ok.

(Refer Slide Time: 17:45)



Register sharing in non-hierarchical Sequence Graph

- A straight line of code without function calls, loops and if-else
- Assume the program is in SSA form.
- Objective: Minimum number of registers storing all the variables
- Conflict graph is an interval graph for non-hierarchical sequence graph
  - Polynomial time solvable
  - Left-edge algorithm (polynomial-time complexity)

8

So, now, what I am going to do is, the sequence graph that we have already seen earlier that this may be non-hierarchical or hierarchical, right. When it is non-hierarchical? When we have a straight line of code right; so straight line of code means, there is no if-else, there is no loop, there is no function call and so on, it is a straight line of code. Then it is a non-hierarchical problem, where it is nothing, but a single basic block right, sequence straight line of codes.

So, whether this particular problem, how this problem is solvable NP-complete in polynomial time or not, that we are going to see, ok. So, in general we have already discussed in the previous class that, this clique cover problem or graph coloring problem is NP-complete for general graphs, right.

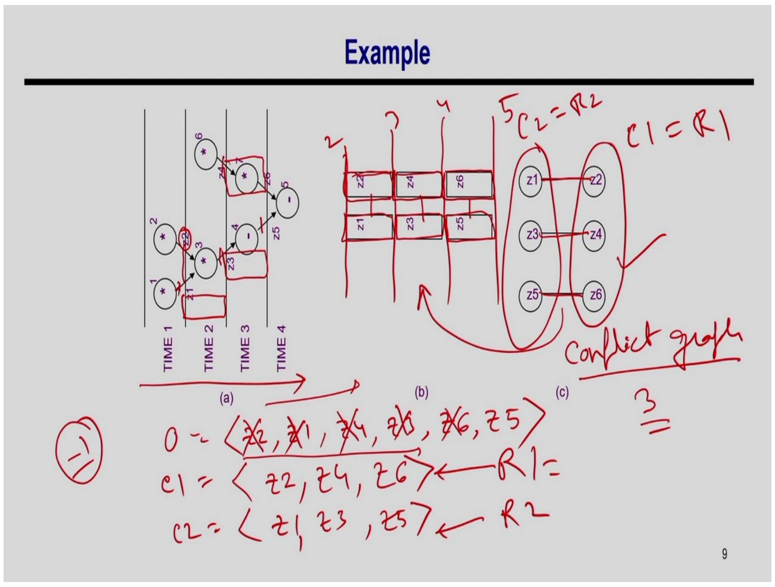
And we have also seen that, for a special type of graph where the graph is interval graph, the conflict graph is an interval graph; the problem is polynomial time solvable, because it's complement graph the compatibility graph is also comparable graph, ok.

So, if we can show that for a particular case, the conflict graph is an interval graph; then I can solve it in polynomial time, right. But in general, if it is not the case, I have to go for heuristic algorithm; because the problem in general is NP complete problem, ok.

So, now, what we are going to do it now is, I am going to check for various kind of sequence a graph, whether the problem remain polynomial time solvable or not, ok. So, I am going to discuss in the next few slides. So, first we will be going to consider the non-hierarchical sequence graph, right.

So, what is non-hierarchical sequence graph? It is a straight line of code, ok. So, in the straight line of code; there is no if-else, there is no loop, there is no function call. So, it there is no control flow basically, it is a data flow ok and which we can consider as a single basic block, right. So, within a basic block, if there is a operation; we can think of this particular problem is basically, we will show that this conflict graph will remain an interval graph and hence this is polynomial time solvable.

(Refer Slide Time: 20:09)



So, let us try to show with an example. Say suppose this is my scheduled graph. So, I am putting the time in the x-direction; because, so that we can represent this in the left edge, in terms of the left edge, right. So, suppose these are the operations 1, 2, 3, 4 and the variables are z_1 to z_6, right.

So, now, what is the lifetime of the variable? So, I can just put the time line here. So, this is 2, this is 3, this is 4 and you can think about this is rest, right. So, see z_1 is getting defined at the end of time step 1, right. So, it's lifetime is basically here right in the time step 2 and after that it is never used.

So, this is the only time where it is getting used. So, the lifetime of z_1 is basically in this part only, right. So, this is my z_1. Similarly for z_2 also, right. So, this is my z_2; z_3 also it defined at the end of time step 2, so it is only getting use it here. So, the lifetime of z_3 is this.

Similarly, z_4, it is defined at the end of time step 2 and it is getting used here, right. So, this is the lifetime of z_4, right. And similarly, z_5 and z_6 is getting defined at the end of time step 3, so their lifetime is starting from 4 to 5, I can assume, right. So, this is the lifetimes, ok.

So, now, you can see what is their conflict graph. So, I can see that this z_1 and z_2 are conflicting; because they are actually defined at the, they are going to use in the same time, so there is an edge between them.

This z 3 and z 4 are conflicting each other, so there is an edge between them in the conflict graph right, this is my conflict graph. And z_5 and z_6 again having have a non-overlapping lifetime and hence they conflict to each other.

So, what I have seen that, so this particular conflict graph represent exactly the interval graph, ok. Hence this conflict graph is basically an interval graph. So, when in this particular case, when this conflict graph represents the intervals of the nodes exactly; then this problem we already discussed, that it is polynomial time solvable and hence we can apply algorithm like left edge algorithm, ok.

So, let us try to solve this problem using left edge algorithm and see how many registers are needed, right. So, in the left edge algorithm, what we do; we actually sort this lifetime of those variables in terms of the left edge, right. So, then the sorted order will be the order is.

So, z_2, z_1; because their left edge is the this and then z_4, z_3, then z_6, z_5, right. And then what it does? It takes color 1 right and then it tries to scan this list and try to find out.

So, initially it assumes it's previously selected interval is the right edge is -1, right. So, initially nothing is selected, so the right edge -1. And then from this list is try to find out the first candidate, whose left edge is greater than the previously selected one.

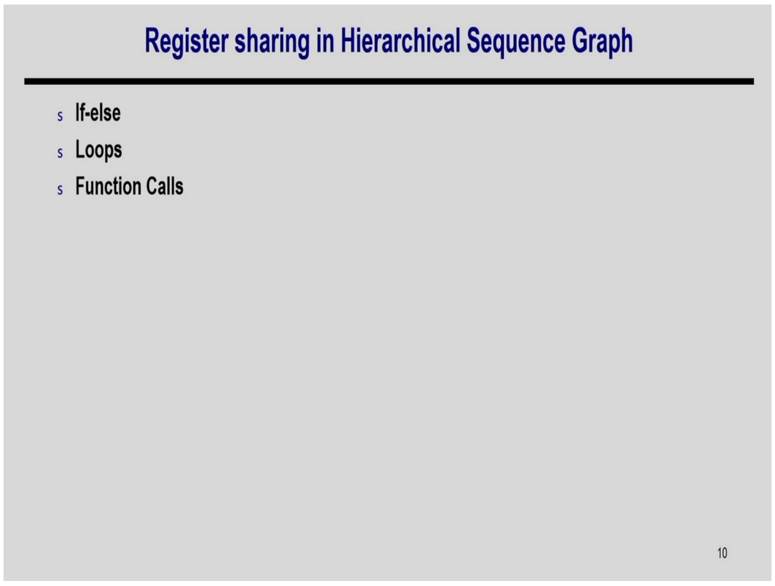
So, initially I can select z_2; because z_2 obviously it is a starting one and it is obviously the $2 > -1$. Now, this the previously selected is z_2 and its right edge is basically 3 right and I am going to select the next one, which is greater than equal to this 3, right.

So, I cannot select. So, z_2 is selected; I cannot select z_1, because it is not overlapping, it is actually overlapping with z_2. But I can select z_4 right; because z_4, the left edge of z_4 is 3, which is >= left edge of z_2, right. So, I can select z_4. In the same way I cannot select z_3, but I can select z_6, right. So, this is my color 1 and I have scanned the whole list and I found that this z 2, z 4 and z 6 should have the same color 1, then I will go for color 2.

Again, I am going to do the same thing; initially it is -1 and obviously, I can select z_1. And once I select z_1, the previously selected intervals the right edge is actually 3 and then I can select, you can understand that I can select z_3 and I can select z_5, right.

So; that means, I can color this node with C_1 and hence it is the register 1 and I can color this node with color 2 and hence this is my register 2. And so, that means register 1 will store z_2, z_4 and z_6 and register 2 will store the variables z_1, z_3 and z_5, ok So, in summary for non-hierarchical sequence graph, I can solve the problem in polynomial time using left edge algorithm, ok.

(Refer Slide Time: 25:43)



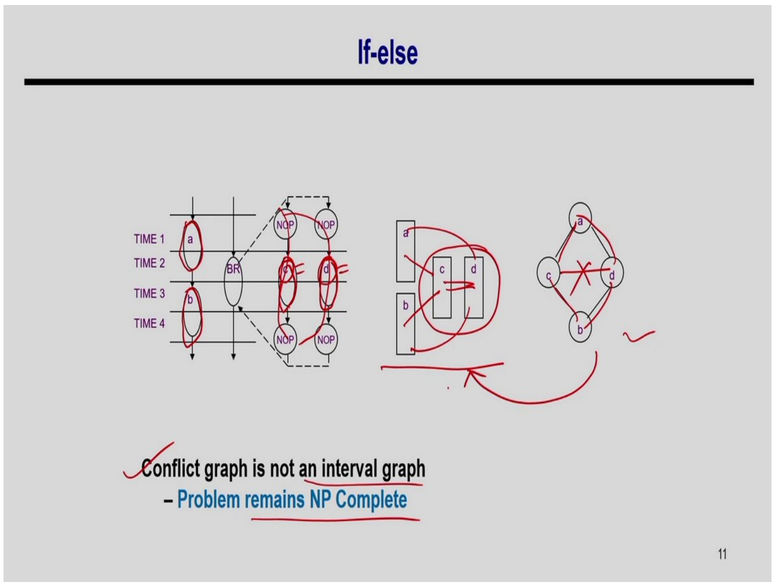
Register sharing in Hierarchical Sequence Graph

- If-else
- Loops
- Function Calls

10

I will move on. So, for hierarchical sequence graph, when the sequence graph become hierarchical; when you have the control flow, either if else, loops and function calls, right. And then we are going to see what is going to happen for if-else, loops and function call to the that conflict graph; whether that particular conflict graph will remain interval graph or not, that we are going to see now, ok.

(Refer Slide Time: 26:06)



If-else

Conflict graph is not an interval graph

– Problem remains NP Complete

11

So, now first you consider if else and if you understand the if-else, where we have a branch node and in the branch node, we have a if branch and the else branch, right. So, this is the if branch and this is the else branch. Now, suppose this variable ‘c’ is getting defined in the if branch and variable ‘d’ is getting defined in the else branch, ok.

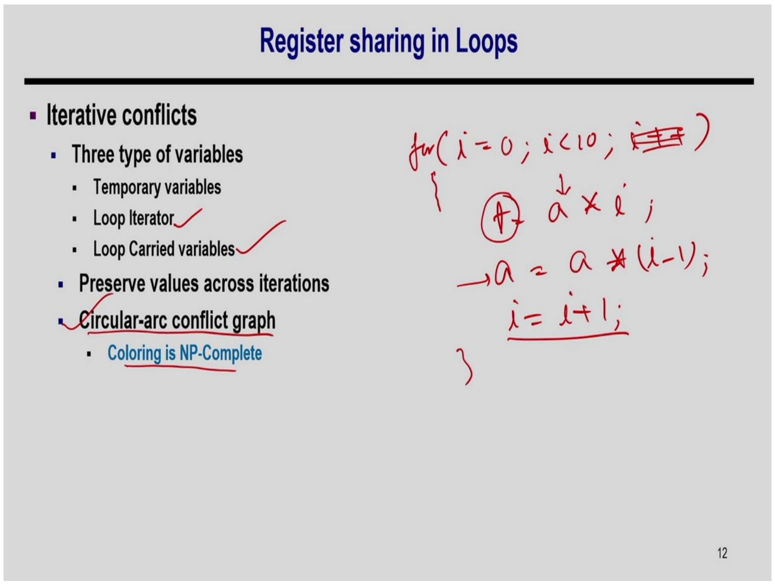
And this is the say lifetime of the variable ‘c’, right. So, it is the lifetime of ‘d’ also; so that suppose it is getting defined here and it is getting used here, in abstract level. So, then the corresponding and there is other two variable ‘a’ and ‘b’, whose lifetime is like this, right.

So, this is the variable ‘a’, whose lifetime is this and this is the variable ‘b’, which is the lifetime is this. Now, think about their inter corresponding intervals. So, you can see that c it’s interval is basically 2 to 3 and this is the intervals and then a, variable a’s interval is 1, 2 and variable b’s interval is 3, 4, right. So, now, we try to construct the conflict graph, right.

So, now you see here, so ‘a’ and ‘c’ is conflicting, so there is an edge; ‘b’ and ‘c’ is conflicting, there is an edge; ‘d’ and ‘b’ is conflicting, there is an edge; ‘a’ and ‘d’ is conflicting, there is an edge. But ‘c’ and ‘d’ is also conflicting, but there is no edge; because these two are mutually exclusive branch right, because in a if else program, either if branch will execute or else branch will execute.

So; that means, although the lifetime of variables ‘c’ and ‘d’ is actually overlapping, but there is no edge in the conflict graph. And hence, this conflict graph does not represent the intervals; hence it is not a not an interval graph. So, for if-else, in general the conflict graph is not an interval graph and hence the problem remain NP complete. So, I cannot solve it using polynomial time algorithm like left edge, ok.

(Refer Slide Time: 28:02)



Register sharing in Loops

- Iterative conflicts
  - Three type of variables
    - Temporary variables
    - Loop Iterator
    - Loop Carried variables
  - Preserve values across iterations
  - Circular-arc conflict graph
    - Coloring is NP-Complete

```
for(i=0; i<10; ~~i++~~)
{
    t = a * i;
    a = a * (i-1);
    i = i+1;
}
```

12

So, now, let us move on to the loops. So, for loops we have. So, if you just try to write a loop. So, basically what is loop? So, suppose I just write for(i =0; i < 10 ; i++) and then say what I do say suppose t = a * i and then say a = a * (I – 1) and then basically we have i=i+1.

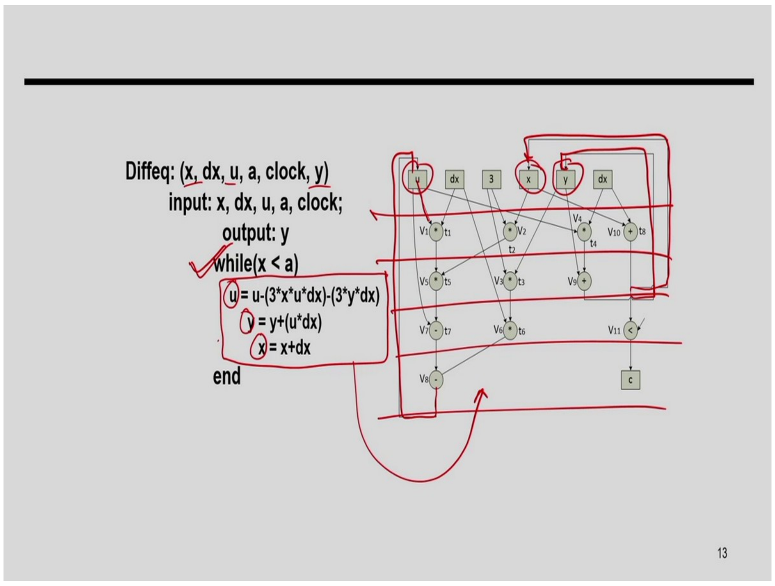
So, in loop basically there are three type of variables, ok. So, one type of variable is t, which is a temporary variable, which basically define and getting using within the same iteration, ok. There is another type of variable, which is type a which is defined in say iteration i and getting used in iteration i+1, right. So, it is basically loop carried dependencies rights; the variable which is has dependency across loops ok and also we have the loop iterator right, which is i = i + 1.

So, there are three types of variables and their lifetime I cannot just simply represent using the normal the way I explain for the non-hierarchical sequence graph. Here because of this across loop across iteration dependencies, the lifetime actually shows using circular arc conflict graph ok, I will explain with an example in the next slide.

And in general, for circular arc conflict graph, the problem of register graph coloring remains NP complete, right. And hence, when we try to solve the problem for loops; the problem remains NP complete, so we cannot again apply this polynomial time algorithm like left edge, ok.

So, let me just explain one loop example and how their lifetime looks like and how to solve the problem.

(Refer Slide Time: 30:05)



```
Diffeq: (x, dx, u, a, clock, y)
    input: x, dx, u, a, clock;
        output: y
        while(x < a)
          u = u-(3*x*u*dx)-(3*y*dx)
            y = y+(u*dx)
             x = x+dx
        end
```

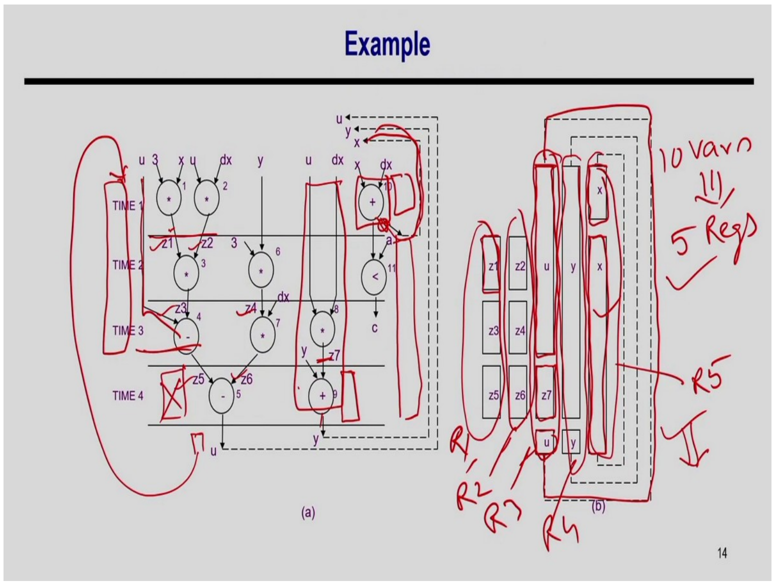
So, let us take the same diffeq example; if you see here, so far, we have discussed the scheduling corresponding to this block, right. So, this block is represented as a sequence graph like this and say I can schedule this behavior.

Say this is my one schedule right, say suppose this is one possible schedule. And now I want to get the lifetime of the variables and I try to find out the minimum number of registers needed to store this variable of using this program, ok. So, that is the purpose. But you see here this body is actually executing under a while loop right; so, this that means it is iterative in nature.

So, and what will happen because of that? You can see here this u it is getting used, it is a basically an input right; u, x, y they are actually x, u and y they are actually the input to the program, but they get redefined; u, x and y it is getting redefined, so that means they are actually going back, right. So, for the next iteration, so they are the loop carried dependencies here right, so x even y.

So, these three variables is available at the start of the program; they are getting used in the program and after that for some their lifetime is not there, but at the end of the program, they are going to be used again. So, they have a circular lifetime, ok.

(Refer Slide Time: 31:35)



So, let us now try to construct the lifetime for one possible some schedule. So, this is one the one of the schedules; you see here this z_1, z_2, z_1, z_2, z_3, z_4, z_5, z_6 are the temporary variables and their lifetime is exactly the way I explained in the last slide that, this z_1 is actually defined at the end of this time 1, so it is lifetime from 2 to in the time step 2 and after 2, it is not used, so and so and so on.

This is the exactly same thing I use in the same slide; but interesting to see about u, x and y, right. So, let us see u. So, it is obviously is available from time step 1; because it is the input, input to the program, right. You see here it is last used at the time step 3 and in time step 4, it is not needed right, it is not needed.

So, it is lifetime is basically from 1, 2, 3 and there is nothing in time step 4 and at the end it is again defined. So, it is actually having a loop carried dependencies, right. So, which is represented in the circular conflict graph is like this. So, your u is actually starting from 1, it is still y it is defined.

And again at the end of time step 4, it is getting defined, which is represented by the circular thing, right. And you can see here; so that means although u is a variable which is loop carried, that means it is actually used across loops.

But still for some point of time, they it has no definition and I can actually share z_7, because z_7 is defined here. So, it is actually lifetime is only for time step 4, which is this. So, I can share a same register to store u and z_7 which is very interesting.

So, we have already seen this is say basically register 1(R_1), this is register 2(R_2) and this is register 3(R_3). So, this is my register 1, this is register 2, and this u and z_7 is stored in register 3; for y it is not possible, because y is getting used in time step 4 also.

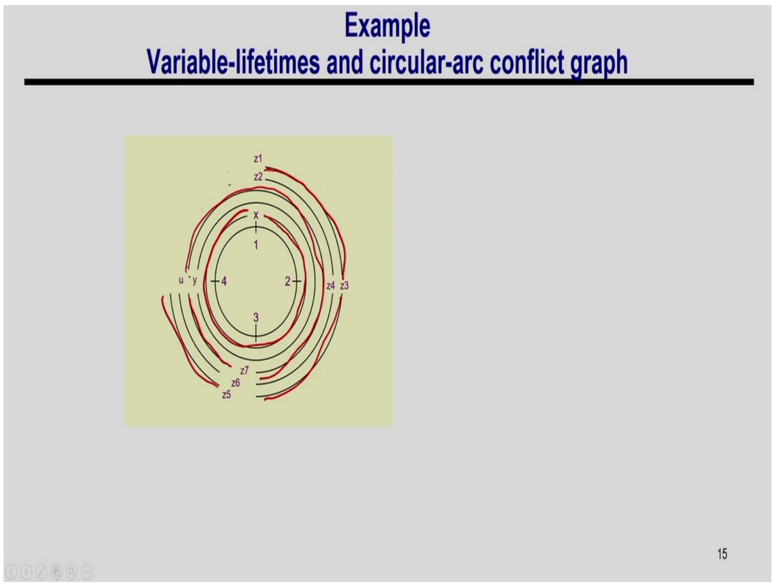
So, that means, the old value is getting used still time step 4 and it is defined at the end. So, I need a separate register to store y, right. So, that means this is my register 4(R_4) and x is very interesting. So, x is defined use only in the first step, right.

But since is the loop carried dependency, this value will remain active for the rest of the program, this is where the main problem comes, right. So, although x has no use in the rest of the program; but since this value of x is going to use the next iteration, I need to keep this x live for the whole program.

So, this is the old value and this is the new value; but this is something necessary, so that means I cannot share any other variable with x in the register, x has to remain active or remain valid throughout the iteration of this loop. And I can use say register 5(R_5) to store x, right. So, how many registers I needed? R_1, R_2, R_3, R_4, R_5. How many variables are there? 6 + 3= 9 plus 1=10; so, 10 variables are mapped to 5 registers, right.

So, this is how I can actually map these loops into registers; but in general, this problem cannot be solved using the left edge algorithm, because it is represented by the circular thing and which does not be represented by the conflict graph. So, their conflict graph cannot represent the circular dependencies, ok.

(Refer Slide Time: 35:18)

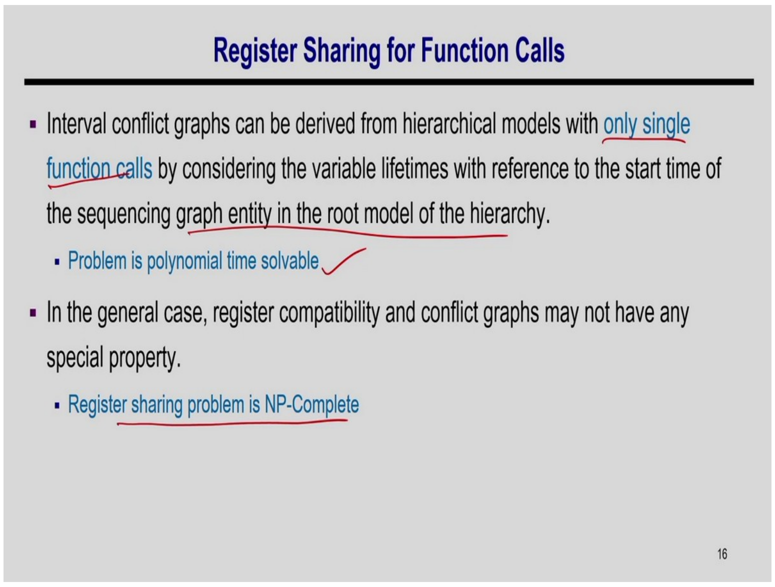


Example
Variable-lifetimes and circular-arc conflict graph

So, for circular dependencies, we usually use this kind of circular arc type of graph, right. So, you can see here the x; we already defined the x is active for throughout the cycle, right. So, there are 4 timestamp and say z_1 is only active for. So, you can actually see z_1.

So, z_1 is actually active for 2, so you can actually define that way, right. So, the z_1 is active for only one half, which is share with z_3 and z_5 and u is defined. So, this is my u, which is defined for three parts of the circle, which is shared with z_7, right. So, this is how I can actually represent, but when this also perfectly captures the lifetimes of the variables, ok.

(Refer Slide Time: 36:12)


Register Sharing for Function Calls

- Interval conflict graphs can be derived from hierarchical models with only single function calls by considering the variable lifetimes with reference to the start time of the sequencing graph entity in the root model of the hierarchy.
  - Problem is polynomial time solvable
- In the general case, register compatibility and conflict graphs may not have any special property.
  - Register sharing problem is NP-Complete

16

So, what we understand for loops, the problem is not polynomial time solvable, ok. Now, we will talk about the function call. So, if the function is only a single call, right. So, what we can do? We can just inline the function and all the variables inside the function is basically get mapped to the main program, right.

And in that case, this it is basically and if you assume that in the function call there is no if else or loops; because I am talking about a hierarchical graph, where we just only talk about the function call.

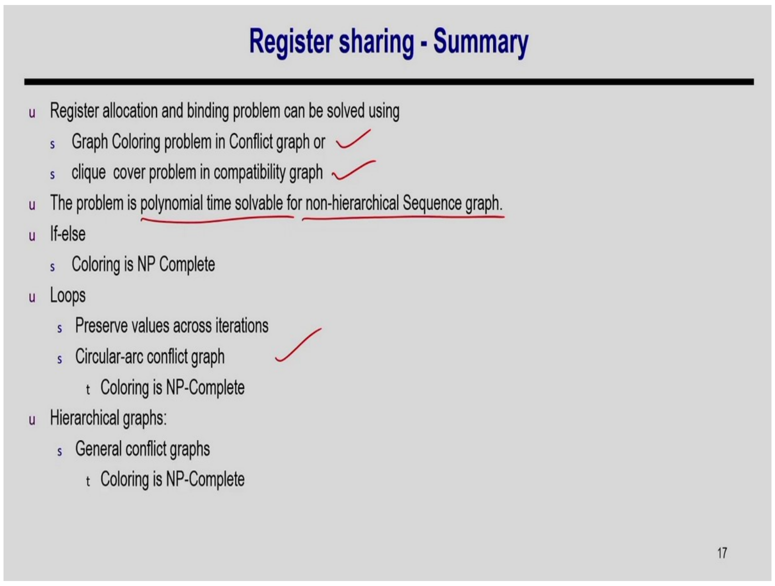
So, then it becomes a non-hierarchical code, there is no straight line of code and hence the problem will remain I mean we can actually show that their compatibility, their conflict graph is actually interval graph and hence the problem is polynomial time solvable.

But once this particular function called multiple times or there are multiple functions; what is the problem? If one function calls multiple times, so our objective, so the variables inside the function get duplicated; even if you are inlined, it will be duplicated multiple times, right.

So, now, I cannot consider that particular variable as a single instance, it has multiple instances. And for such cases, this particular this conflict graph, is not an interval graph. And hence the problem cannot be solved in polynomial time. And hence, the register sharing problem will remain NP complete in such cases, when we have multiple calls to a single function.

So, in general I can say that for function call also, this problem is not polynomial time solvable, ok. So, we have seen that for all cases of non-hierarchical graph, this problem is not polynomial-time solvable; only for non-hierarchical graph, the problem is polynomial time solvable, because there is no control flow, ok.

(Refer Slide Time: 38:05)



Register sharing - Summary

- Register allocation and binding problem can be solved using
  - Graph Coloring problem in Conflict graph or ✓
  - clique cover problem in compatibility graph ✓
- The problem is polynomial time solvable for non-hierarchical Sequence graph.
- If-else
  - Coloring is NP Complete
- Loops
  - Preserve values across iterations
  - Circular-arc conflict graph ✓
    - Coloring is NP-Complete
- Hierarchical graphs:
  - General conflict graphs
    - Coloring is NP-Complete

17

So, to summarize what we have seen that, the register allocation binding problem is the problem where we try to find the minimum number of registers, which store all the variables of the program, such that two variables can share a register when their lifetime is non-overlapping, right.

And we have seen that, this particular problem of register allocation binding can be solved using graph coloring problem of conflict graph or clique cover problem of compatibility graph, ok.

And we have seen that for non-hierarchical sequence graph, this problem remains polynomial time solvable and hence we can apply left edge algorithm to solve the problem. For hierarchical graphs consist of if-else, loops and function call; this particular problem remain NP complete and hence we cannot apply polynomial time algorithm like this left edge and which has to be solved using some heuristic algorithm, ok. So, with this I conclude today’s discussion.

Thank you.