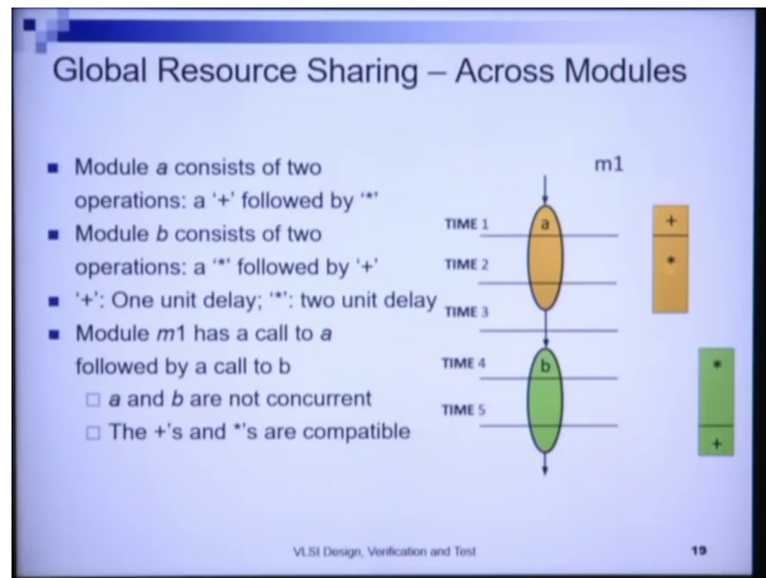
Embedded Systems – Design Verification and Test
Dr. Santosh Biswas
Prof. Jatindra Kumar Deka
Dr. Arnab Sarkar
Department of Computer Science and Engineering
National Institute of Technology, Guwahati

Lecture – 10
Hardware Architectural Synthesis – 5

Welcome to module 2 of lecture 7. In the last module we saw that for operations within a simple data flow graph within a basic block and for temporary registers within a simple operation constraint graph within a basic block, their corresponding resource sharing models can be form using interval graphs. And we can find the minimum number of registers required to implement a set of temporary variables within the operation constraints graph or the minimum number of functional units required for each type to implement the behavioral operations within an operation constraints graph can be obtained in polynomial time using graph coloring of the conflict graph corresponding to the operations or the registers in the operation constraint graph within a simple basic block.

We use the left edge algorithm to obtain such an optimal coloring of the interval graph in n log n time. We saw that here n is the number of operations or the number of temporary registers that we have in the graph. And we need to sort the elements at the start of the algorithm in terms of the left edges of that intervals and this amounts to the n log n complexity of the left edge algorithm. In this module we will see a bigger scenario.

(Refer Slide Time: 02:18)



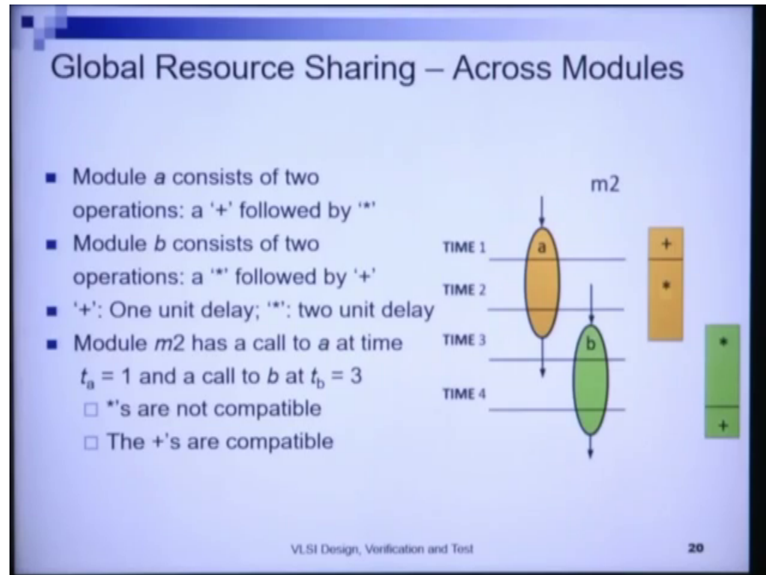
We will see the resource sharing problems that exist, when we consider not only a single operation constraints graph, but we consider a different operation constraints graph across modules. Now, we need to understand that within a single operation constraint graph there are no controls statements there are no mutually exclusive operations there are no branches there are no loops, but; however, when we consider resource sharing, when we attempt to consider resource sharing across modules across operation constraints graph in different modules, we need to also consider the control flow structure the loops the branches in those modules.

So, in this we will look at this broader problem. And we will we will take simple examples to understand what is the actual problem that we have in hand. The first scenario is the most simple scenario. Let a module a consists of 2 operations are plus followed by a star. So, this is module a it is here I have a plus and here I have a star a star a plus followed by a star. And I have a module b which consists of 2 operations a star followed by a plus.

And we said that plus uses 1-unit delay and star consumes 2-unit delay, right. Now, let us consider a module m 1 here which calls module a followed by module b. So, it first calls module a and then calls module b. We see that here this is very simple a and b are not concurrent and hence all operations are also not concurrent. And we can apply the simple

interval graph method to solve it and the left edge algorithm will give me the maximum resource sharing for module 1.

(Refer Slide Time: 04:29)



Now, let us complicate the situation just a bit. I have a module a it consists of 2 operations plus followed by a star very similar to the previous one, and also module b very similar star followed by a plus. Now here the only difference is that the module b is called before module a finishes. So, I have an operation constraint graph that defines module a, I have an operation constraints graph that defines module b. And I in the first example both this operation constraints graph were separated in time.

The complete lifetimes of the operation constraints graph were separated in time they were non concurrent. Here the operation constraints graph are overlapped. Module 2 here is a module 2 that calls both module a and module b. Module 2 has a call to a at time t equals to 1 and a call to b at time t equals to 3. Now, here we see that the stars are not compatible. The star operation that is a multiplication operation takes 2 time units to execute and hence this multiplication operation and this multiplication operations are overlapped in time and hence I cannot use a single instance of a multiplier resource to execute both these operations. They become incompatible.

However, the plus operations are compatible. They are separated in time and; obviously, they are both additions and can be implemented by the adder. So, hence the same instance of an adder can be used to execute both these behavioral operations.

(Refer Slide Time: 06:19)

Global Resource Sharing – Across Modules

- Module *a* consists of two operations: a '+' followed by '*'
- Module *m*3 has two calls to *a* at time t_a = 1 and t_a = 5
  - *m*3 has three other *'s
- Two non-contiguous intervals for the '*' in *a*
- The conflict graph is not an interval graph
  - It is not even a chordal graph
  - Vertex colouring becomes intractable

Conflict Graph

VLSI Design, Verification and Test 21

Now let us take the next small complication higher complications. Now we have a loop. And we to simplify a loop we just say that there is a module which calls another module 2 times, ok. Module a consists of 2 operations a plus followed by a star similar to the previous 2 examples. Now module 3 has 2 calls to a at time t a equals to 1 and ta equals to 5. Now there is a module 3 which calls a at time which calls a at time t equals to 1 and then again calls a at time t a equals to 5. Now, m 3 has 3 other multiplication operation.

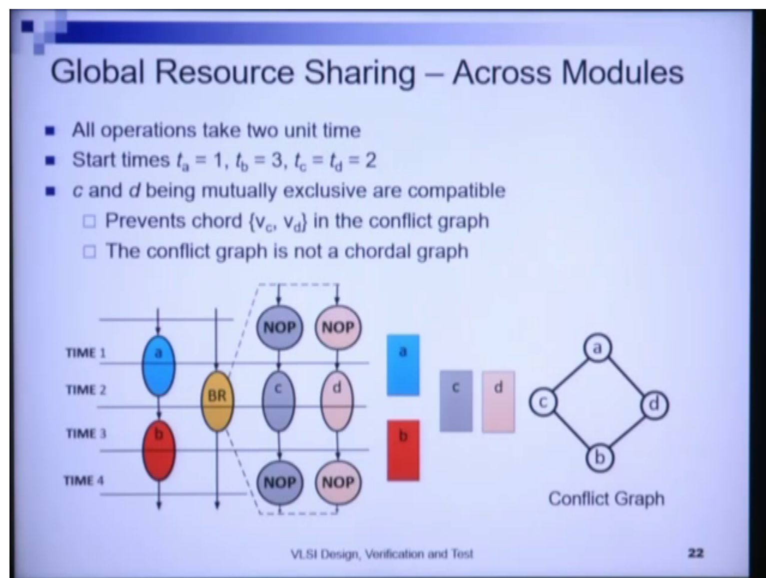
So, the 2 noncontiguous intervals of star we have 2 noncontiguous intervals of star in a, so, here this star operation in a non-compatible with the star operation 2 here. Again this star operation is not is not compatible with 4 here. And here this 2 is not compatible with 3 this 4 is again non compatible with 3. So, what happens that, this module a becomes non compatible with 2. Why? Module a is non compatible with 2 because module a does not complete before 2 begins.

So, 2 is a distinct multiplication operation a is another module. So, a and 2 are conflicting. And hence I have an edge in the conflict graph between a. And 2 similarly I have an edge between 2 and 3 because 2 and 3 overlap. I have an edge between 3 and 4 right I have an edge between 3 and 4 because 3 and 4 overlap I have an edge between a and 4 because 4 and a overlap in time.

But what we see here is that there are 2 distinct intervals now for this module a. And it is no longer a single continuous interval. And hence the conflict graph is no more an

interval graph. It is not even a chordal graph. We have a cycle of length 4 and we do not had any chord which connects nonconsecutive edges nonconsecutive vertices. And hence this is not a chordal graph. So, this therefore, the addition of this simple call here, we call a module 2 times with this with these 3 additional multiplication operation makes this graph non makes this graph are non-chordal graph. And hence the graph coloring algorithm on this conflict graph therefore, becomes np complete and we need to apply enumerative techniques.

(Refer Slide Time: 09:31)



Now, let us take another type of complication which is branching. Now we assume that all operations there are simple one type of operation in this example all operations take 2 units' times.

So, start times of the operation t a is at time step 1, t b starts at time step 3, t c and td are 2 mutually exclusive operations and they both start at time step 2. Now, c and d being mutually exclusive or compatible, although they execute at within the same time step because they are mutually exclusive they can be implemented by the same resource instance. Because either c or d will execute, both c and d will never execute they are mutually exclusive by this branch.

And because of this mutual exclusion between c and d, there is not a chord between vc and vd in the conflict graph. And therefore, again we have a non-chordal graph here. Why? Let us see how. There is an a here; a and b are compatible; a and b are compatible

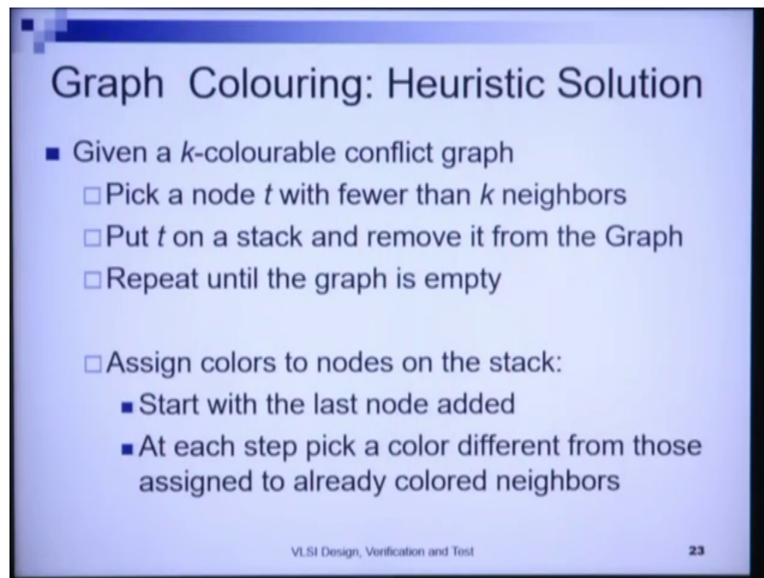
and hence there does not exist an edge in the conflict graph, a and c are not compatible and here a and c have an edge. Similarly, a and d are not compatible and hence they are they have an edge why are they not compatible because they overlap in time.

Again b and c are not compatible because they overlap in time, b and d also are non-compatible because they overlap in time. And hence in this conflict graph this conflict graph again becomes a non-chordal graph just by the introduction of a simple branch and therefore, resource sharing over. So, modules which contain branches and loops become non trivial and cannot be solved in polynomial time and I need enumerative techniques for graph coloring, because this problem becomes np complete for general graphs non-chordal graphs.

However, we can apply heuristic techniques to solve general graph coloring problems. Although these although; obviously, such graph coloring techniques will not be optimal may not be optimal such graph coloring may not give the minimum number of colors. However, they are often essential when the problems of are of very large sizes. For np complete problems as we have seen branch and bound etcetera similar types of enumerative approaches can be applied to the graph coloring problem as well when we are not studying it here.

We have taken we have understood a flavor of handling such big np complete problems in with exponential complexity using combinatorial search approaches like branch and bound in previous lectures and we will not consider them here. Rather we will understand a simple heuristic approach for solving the graph called the general graph coloring problem.

(Refer Slide Time: 12:55)

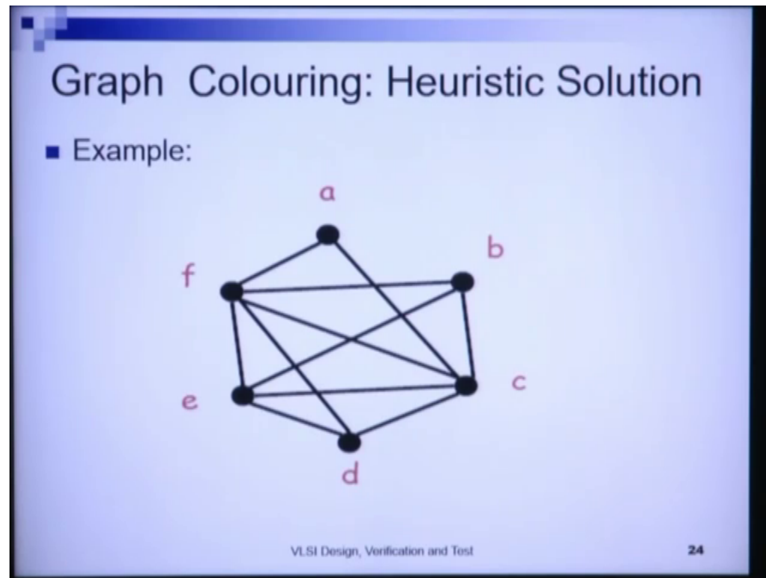


The solution proceeds as follows. Let us say we want to find out whether a conflict graph is k colorable? So, we want to find out and allocate colors to a k colorable conflict graph. So, if the graph is k colorable, we will be able to assign colors to the vertices of this conflict graph. So, how do we proceed? We pick a node t with fewer than k neighbors. So, in the conflict graph we pick a node t with fewer than k neighbors. Then we put t that the node that is the node we have chosen and put it on the stack and remove it from the graph. So, after we take this node out of the conflict graph, the edges adjacent into it also move away, alright. Now, this exposes a few more nodes with less than k neighbors possibly.

Now, if we can proceed and go on doing this we will ultimately obtain the empty graph; that means, fewer will go on taking up nodes pushing it onto the stack and until all nodes in the graph have been put into the stack right. And then the graph becomes completely empty, we may get stuck in between as well because we do not get the neighboring nodes with less than k neighbors. And that does not mean that the graph will not be k colorable. We should understand that this is a heuristic approach. Now, we assign colors to the nodes in the stack one by one. We start with the last node added and at each step we pick a color different from those assigned to already colored neighbors.

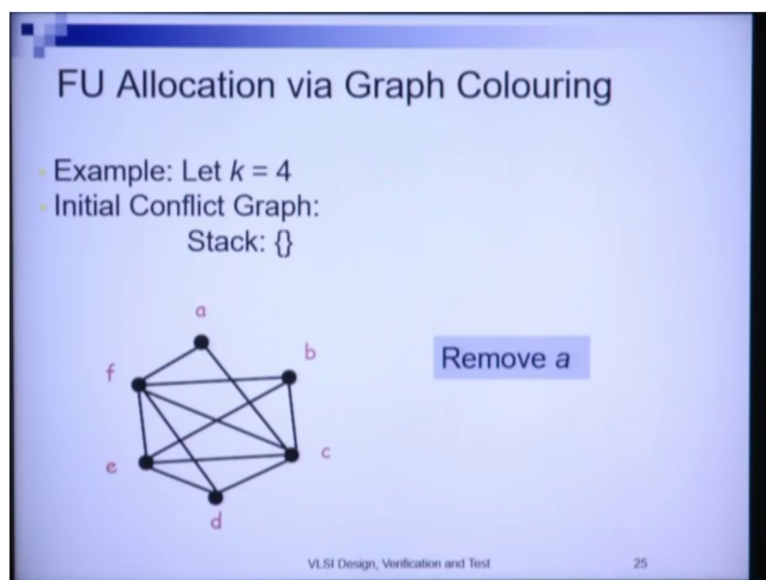
So, if I have neighbors previously colored neighbors with a certain color, when I take the another node out of the stack, I cannot assign a color which is same as that of it is neighbors right. We will take an example to see this.

(Refer Slide Time: 15:03)



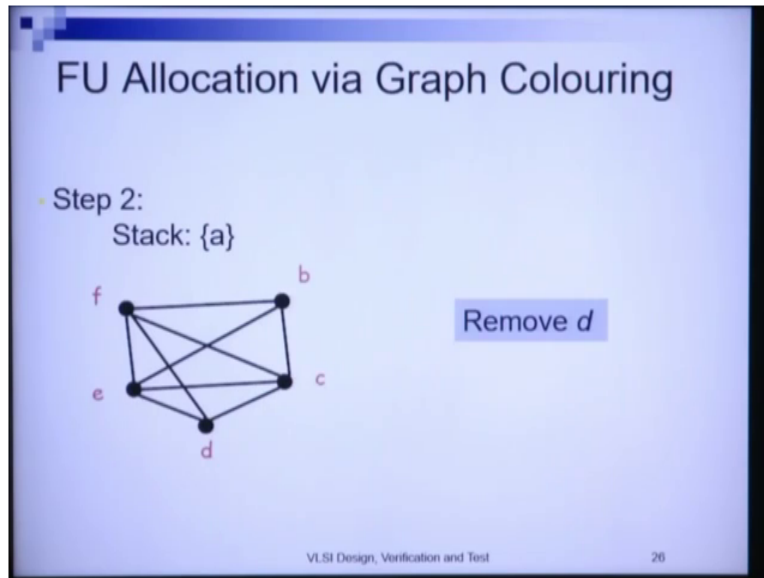
So, let us say we are given this conflict graph. So, this conflict graph say has a, b, c, d, e, f operations and we want to allocate all these operations are of the same type and we want to allocate the minimum number of functional units necessary to color this graph.

(Refer Slide Time: 15:35)



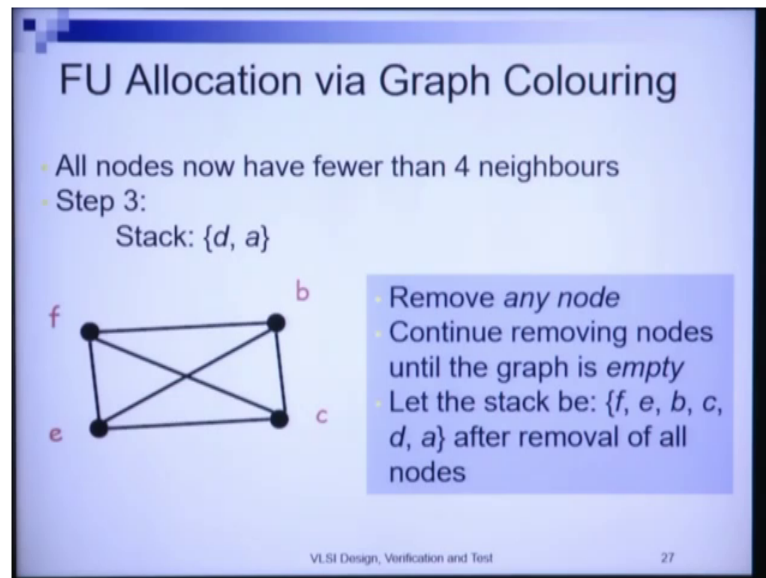
So, what do we have? We have k equals to 4. First, we will search for a node with less than 4 neighbors. So, a is such a node with less than 4 neighbors. In fact, a has 2 neighbors. So, I can remove a from this graph.

(Refer Slide Time: 15:50)



When I remove a from this graph, we this is the residual graph that residual conflict graph that remains, a goes to the stack. Now, we have to find another graph which has less than 4 neighbors. So, we are assuming that the graph is 4 colorable and hence we are trying to find out neighbors which are less than 4. So, we choose d, d has 3 neighbors less than 4 neighbors and we choose d.

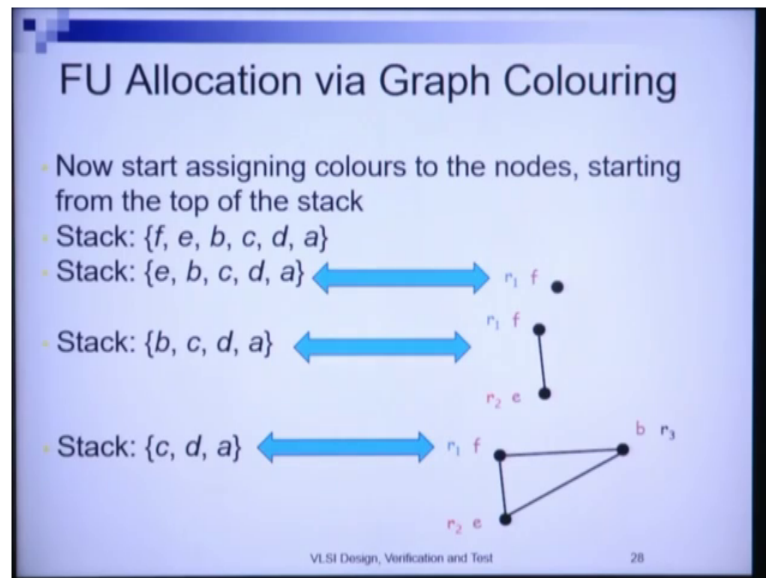
(Refer Slide Time: 16:19)



And after we have chosen d, the residual graph is this one; this is the residual graph that remains and a d and a goes into the stack. Now, we see that all the remaining nodes have less than 4 neighbors and hence I can choose in any order. So, we can remove any node. We continue removing nodes until the graph is empty. So, febcda is a certain order of choosing the nodes and the stack will be this after removal of all nodes.

So, before the start of the algorithm, we see that f has more than 4 neighbors more than 3 neighbors, e has 4 neighbors, c has 4 neighbors. So, I can choose either a or b or d. I had chosen a that can be done randomly; however, I have chosen this order and I have come to this place.

(Refer Slide Time: 17:21)

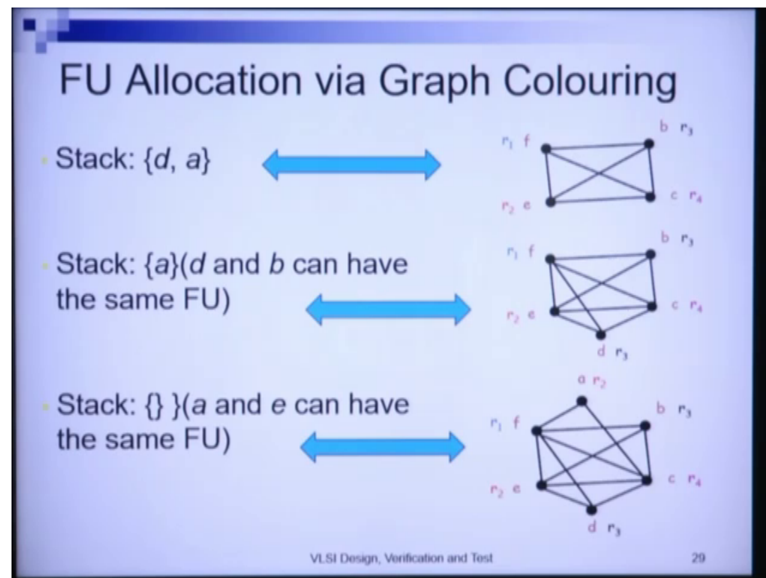


And for now I have an empty graph. Now, when I have an empty graph and all nodes have been put into the stack, I will start assigning colors to the nodes starting from the top of the stack. And at the top of the stack I have f which is the last node I have added.

So, I had first added a, then I had added b, then I took it order cdef. Now, when I have taken out c, I have allocated resource r 1 with the blue color to f at this step. Then I take out the next color e and we see that e and f share an edge hence I allocate another distinct resource r 2; resource 2 and color it red. And I use a distinct resource because they share an edge.

Now, I will take out edge b. And we see that b has an edge with both f and e and hence I cannot use the colors blue or red and I have used black here. So, I need another instance of the resource. So, I have already used 3 instances of the resource for executing f b and e, then I remove c.

(Refer Slide Time: 18:35)



And I see that c also shares an edge with each of the previous nodes each of the previous operations and therefore, I have to allocate another different color say pink to this node.

Now, I take out d. We see that d and b do not share an edge and hence d and b can have the same color. So, the same resource instance can be used for both operations b and d. Resource instance r 3 black can be used for both b and d. And at last I take out a, we see that a and e can have the same color because they do not share an edge and the same resource can be assigned to both of them.

Hence, I have been able to allocate all the operations of my operation constraints graph using 4 colors from the corresponding conflict graph that I have by obtaining a coloring of the conflict graph using 4 colors; this is a heuristic graph coloring methodology for general graphs. Now, we must understand that like we said that across modules, when there are loops as well as branches conflict graph becomes non chordal and becomes a general graph and polynomial algorithms do not exist for coloring.

For functional units we studied that a similar thing happens for registers as well. Because let us say this is an operation instance multiplication. And this operation floats its output on a particular register. Suppose, I allocate a functional unit f 1 to this multiplication operation then there are 2 distinct intervals in which the functional unit must hold operation 1. Similarly, there will be 2 distinct intervals for the corresponding temporary register at the output of this multiplication operation.

And hence the register implemented will have 2 distinct intervals and will not be a simple interval graph anymore. And hence the register coloring problem or the register allocation problem will also become a general graph coloring problem which is np complete and one of the methods by which we can color such general graph is by the graph coloring method that we just studied. We come to the end of module 2 of lecture 7.