INDIAN INSTITUTE OF TECHNOLOGY GUWAHATI

NPTEL

NPTEL ONLINE CERTIFICATION COURSE
An Initiative of MHRD

VLSI Design, Verification & Test

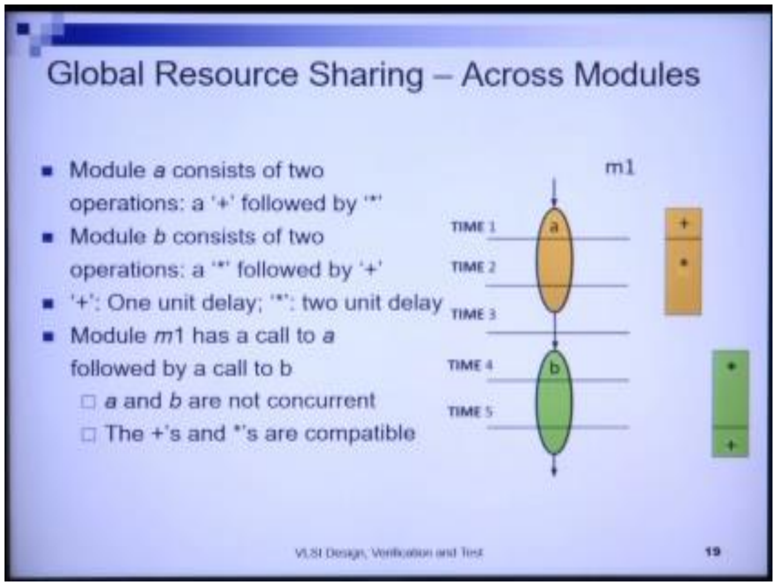
Dr. Arnab Sarkar
Department of CSE
IIT Guwahati

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 formed using interval graphs and we can find a minimum the minimum number of registers required to implement a set of temporary variables within the operation.

Constraint graph or the minimum number of functional units required for each type to implement the behavioral operations within an operation constraint 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 nlog n complexity of the left edge algorithm. In this module we will see a bigger scenario we will see the resource sharing problems that exist.

(Refer Slide Time: 02:14)


Global Resource Sharing – Across Modules

- Module *a* consists of two operations: a '+' followed by '*'
- Module *b* consists of two operations: a '*' followed by '+'
- '+': One unit delay; '*': two unit delay
- Module *m*1 has a call to *a* followed by a call to b
    - *a* and *b* are not concurrent
    - The +'s and *'s are compatible

m1
TIME 1
TIME 2
TIME 3
TIME 4
TIME 5
a
b
+
*
*
+

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 and there are no loops but however when we consider the source 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 two operations are plus followed by a star so this is module it is here I have a + and here I have a *, a * a + followed by a * and I have a module B which consists of two operations a * followed by a + and we said that + uses one unit delay and * consumes to unit delay, right.

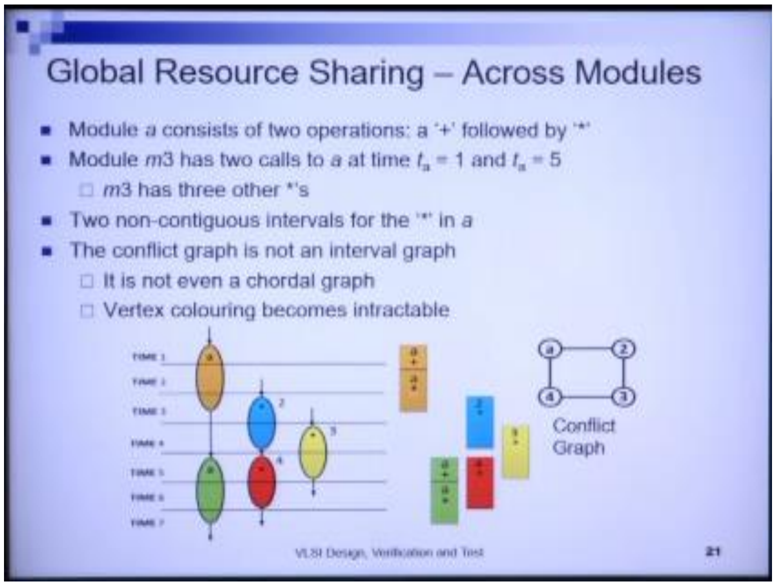
Now let us consider a module m1 here which calls module A followed by module B, so it first calls module A and then calls module B we see that here this is very simple a and B are not concurrent and hence all operations are also not concurrent and we can apply the simple interval graph method to solve it and the left edge algorithm will give me the maximum resource sharing for module 1. Now let us complicate the situation just a bit I have a module A it consists of two operations + followed by a * very similar to the previous one and also module will be very similar * followed by a +.

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 given 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 * are not compatible the * operation that is a multiplication operation takes two 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: 06:13)



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

TIME 1
TIME 2
TIME 3
TIME 4
TIME 5
TIME 6
TIME 7
a
2
3
4
Conflict Graph

VLSI Design, Verification and Test

21

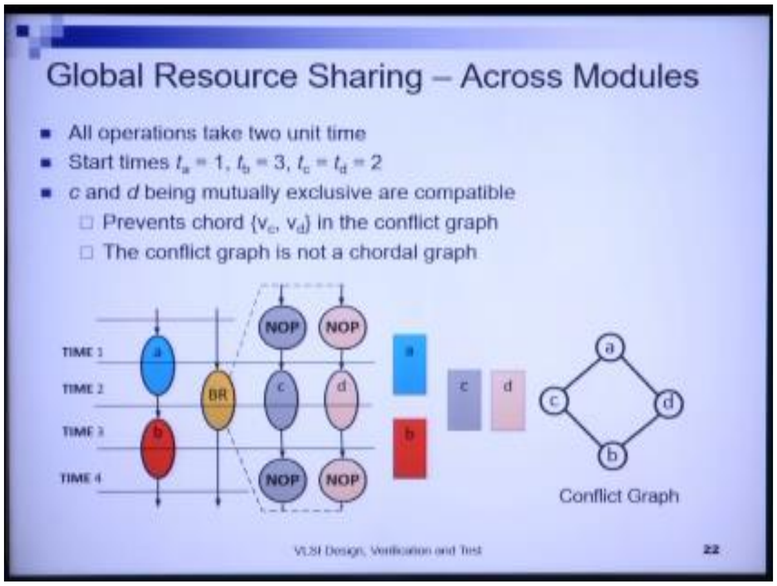
Now let us let us take the next small complication higher complication now we have a loop and we to simplify a loop we just say that there is a module which calls another module two times okay, module A consists of two operations are + followed by a * similar to the previous two examples. Now module three has two calls to A at time ta equals to1 and Ta equals to 5 now there is a module three which calls A at time which calls A at time T equals to 1 and then again calls A at time T equals to 5.

Now m3 has three other multiplication operation, so the two noncontiguous intervals of * so we have two non contiguous intervals of * in A, so here this * operation in A is non compatible with the * operation to hear again this star operation is not is not compatible with 4 here and here this 2 is non-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 two and three 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 two 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 have any chord which connects non-consecutive agents.

In non-consecutive vertices and hence this is not a chordal graph so this therefore the addition of this simple call here we call a module two times with this with this three additional multiplication operation makes this graph a 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: 09:26)



Global Resource Sharing – Across Modules

- All operations take two unit time
- Start times $t_a = 1$, $t_b = 3$, $t_c = t_d = 2$
- *c* and *d* being mutually exclusive are compatible
  - Prevents chord $\{v_c, v_d\}$ in the conflict graph
  - The conflict graph is not a chordal graph

VLSI Design, Verification and Test 22

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 two units so start times of the operation Ta is at time step 1 Tb starts at time step 3 Tc and Td are two 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.

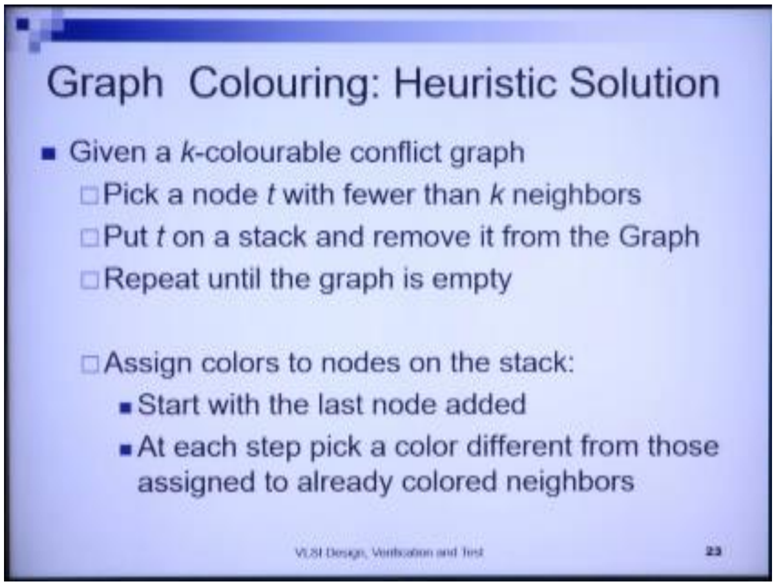
Both c and d will never execute they are mutually exclusive by this branch and because of this mutual exclusion between c and d there is not a cord between vc and vd in the conflict graph and therefore again we have a non-chordal graph here why, let us see how there is an A here A and B are compatible 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 have an edge why are they not compatible because the overlap in time again b and c are not compatible because they overlap in time b and d also are not compatible because the 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 such 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 become NP-complete for general graphs non-chordal graphs.

However, we can apply heuristic techniques to solve general graph coloring problems 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 extra 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: 12:51)



Graph Colouring: Heuristic Solution

- Given a *k*-colourable conflict graph
  - Pick a node *t* with fewer than *k* neighbors
  - Put *t* on a stack and remove it from the Graph
  - Repeat until the graph is empty
  - Assign colors to nodes on the stack:
    - Start with the last node added
    - At each step pick a color different from those assigned to already colored neighbors

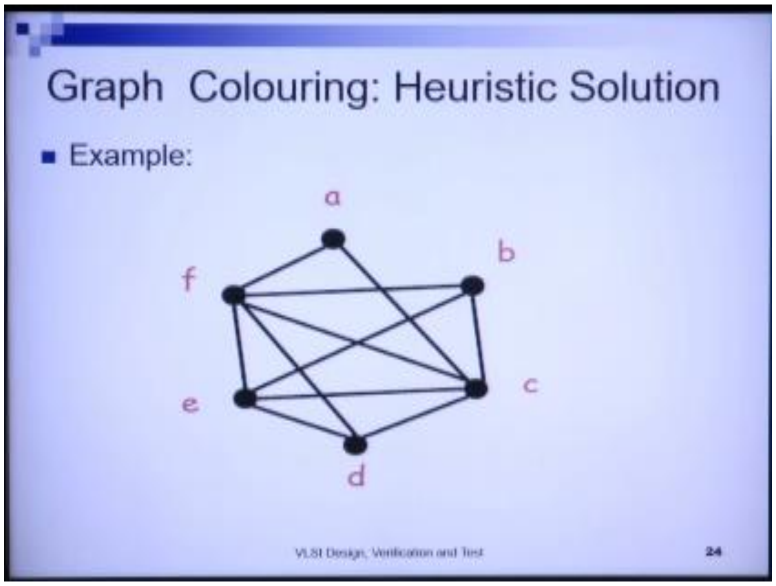
VLSI Design, Verification and Test 23

The solution proceeds as follows, let us say we want to find out whether the 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 anode t with fewer than k neighbors, so in the conflict graph we pick a node t with fewer than k neighbors then we put that the note 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 adjusting to it also move away, right 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 we 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 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 its neighbors, right.

(Refer Slide Time: 14:59)



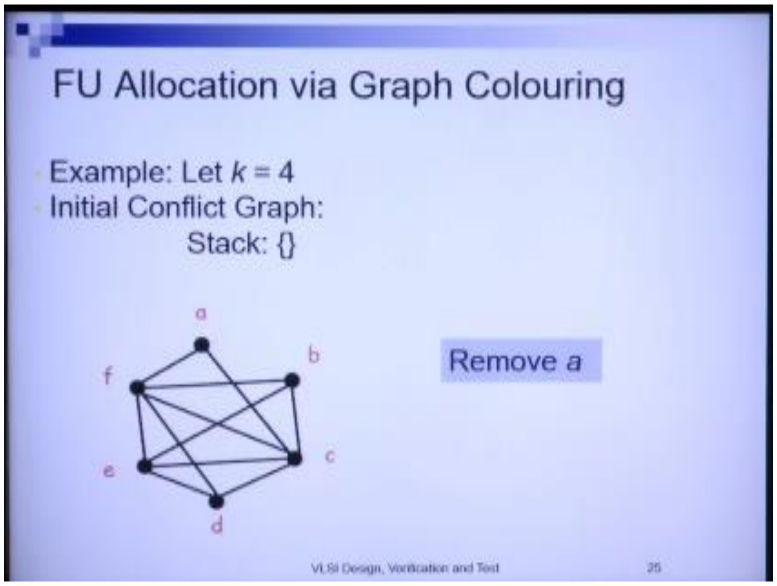
Graph Colouring: Heuristic Solution

- Example:

VLSI Design, Verification and Test 24

We will take an example to see this, so let us say we are given this or 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 to color this graph.

(Refer Slide Time: 15:29)


FU Allocation via Graph Colouring

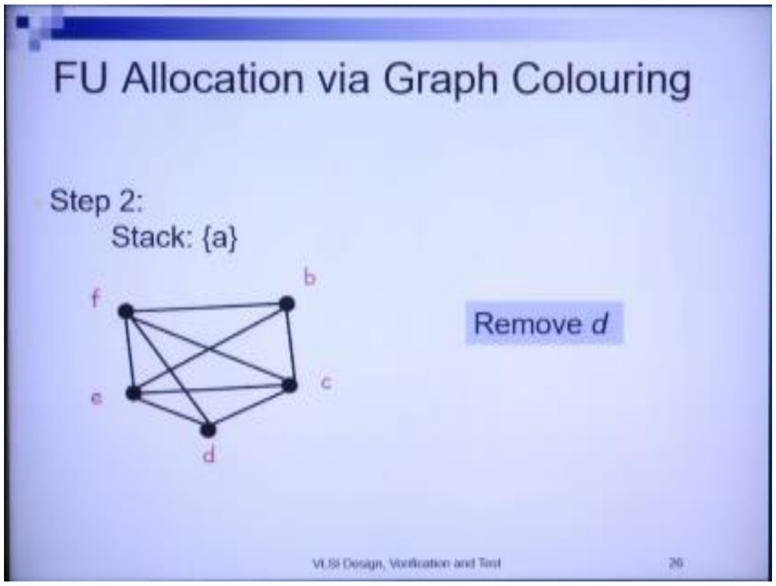
- Example: Let $k = 4$
- Initial Conflict Graph:

Stack: {}

VLSI Design, Verification and Test 25

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 k from this graph when I remove a from this graph.

(Refer Slide Time: 15:46)



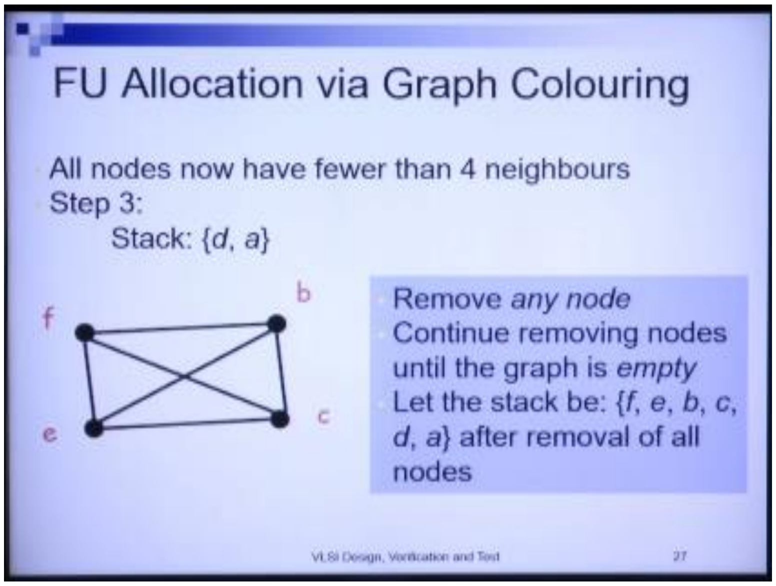
FU Allocation via Graph Colouring

- Step 2:
  - Stack: {a}

VLSI Design, Verification and Test 26

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 for 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: 16:15)



FU Allocation via Graph Colouring

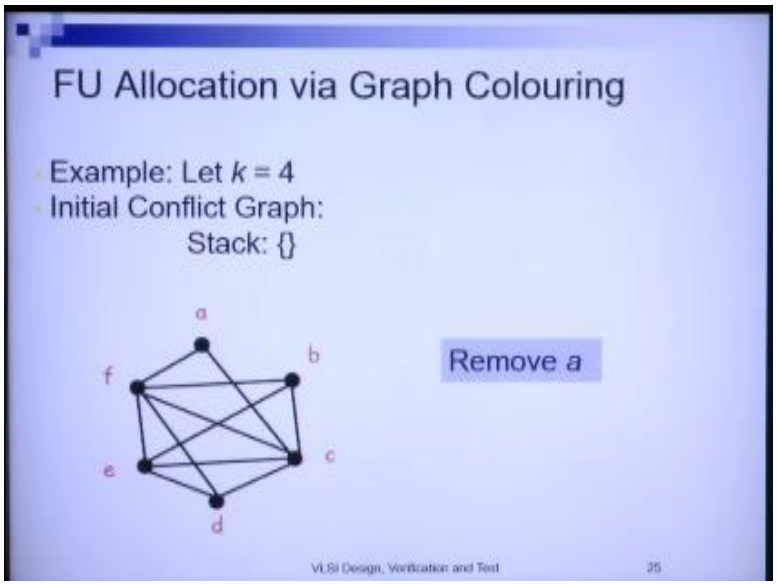
- All nodes now have fewer than 4 neighbours
- Step 3:
  - Stack: {*d*, *a*}

- Remove *any node*
- Continue removing nodes until the graph is *empty*
- Let the stack be: {*f*, *e*, *b*, *c*, *d*, *a*} after removal of all nodes

VLSI Design, Verification and Test 27

And after we have chosen d the residual graph is this one, this is the residual graph that remains and ad 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 f,e,b,c,d,a is a certain order of choosing the nodes and the stack will be this after removal of all nodes.

(Refer Slide Time: 16:50)



FU Allocation via Graph Colouring

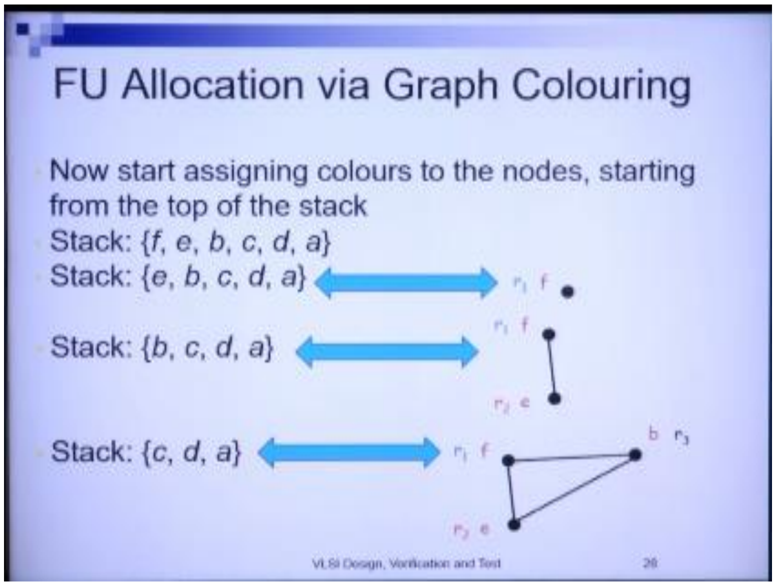
- Example: Let $k = 4$
- Initial Conflict Graph:

  Stack: {}

a
b
c
d
e
f
Remove *a*

So before the start of the algorithm we see that if 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.

(Refer Slide Time: 17:17)


FU Allocation via Graph Colouring

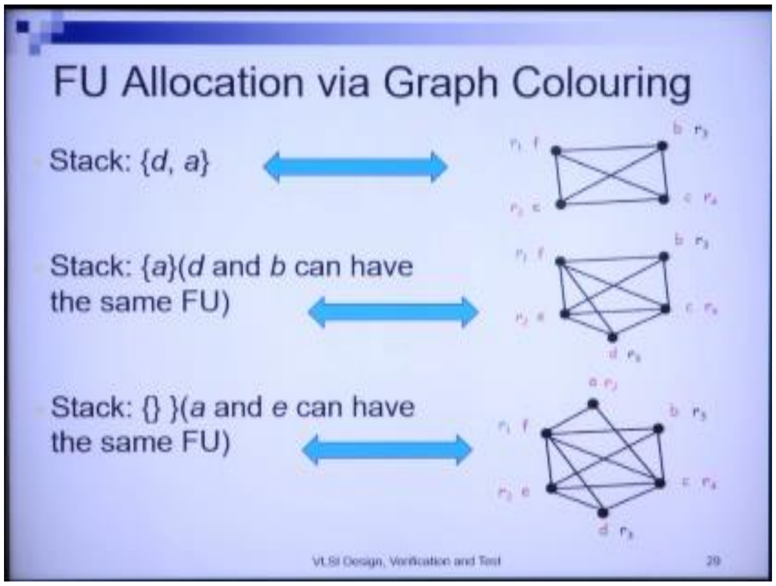
- Now start assigning colours to the nodes, starting from the top of the stack
- Stack: {*f*, *e*, *b*, *c*, *d*, *a*}
- Stack: {*e*, *b*, *c*, *d*, *a*}
- Stack: {*b*, *c*, *d*, *a*}
- Stack: {*c*, *d*, *a*}

r_1 f
r_1 f
r_2 e
r_1 f
b r_3
r_2 e

However I have chosen this order and have come to this place 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 be then I took it order cdef now when I have taken out c I have allocated resource r1 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 and hence I allocate another distinct resource r2 resource 2 color it red and I use a distinct resource because they share an edge.

Now I takeout edge be 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 three instances of the resource for executing a fb and e then I remove c.\

(Refer Slide Time: 18:31)



r1 f
b r3
r2 e
c r4
d r3
a r2

FU Allocation via Graph Colouring

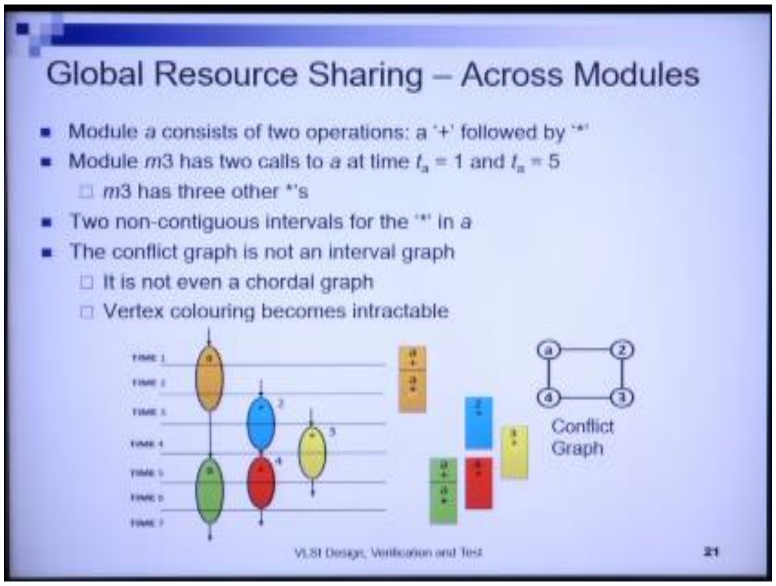
- Stack: {*d*, *a*}
- Stack: {*a*}(*d* and *b* can have the same FU)
- Stack: {} }(*a* and *e* can have the same FU)

VLSI Design, Verification and Test 29

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 colors a 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 resource instance are three 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 craft 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.

(Refer Slide Time: 20:00)



Global Resource Sharing – Across Modules

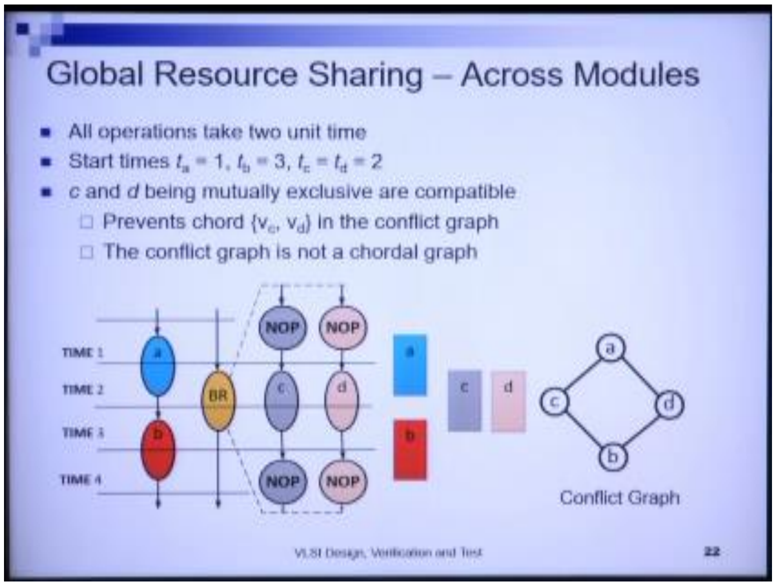
- Module *a* consists of two operations: a '+' followed by '*'
- Module *m*3 has two calls to *a* at time t_a = 1 and t_a = 5
  - *m*3 has three other *'s
- Two non-contiguous intervals for the '*' in *a*
- The conflict graph is not an interval graph
  - It is not even a chordal graph
  - Vertex colouring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

Conflict Graph

Conflict graph becomes non-chordal and becomes a general graph and polynomial algorithms do not exist for coloring similar for functional units we studied that this similar thing happens for registers as well because let us say this is an operation instance multiplication and this operation floats is output on a particular register suppose I allocate a functional unit f1 to this multiplication operation then there are two distinct intervals in which the functional unit must hold operation 1.

Similarly there will be two distinct intervals for the corresponding temporary register at the output of this multiplication operation and hence the registered implemented will have two distinct intervals and will not be a simple interval graph anymore.

(Refer Slide Time: 20:55)



Global Resource Sharing – Across Modules

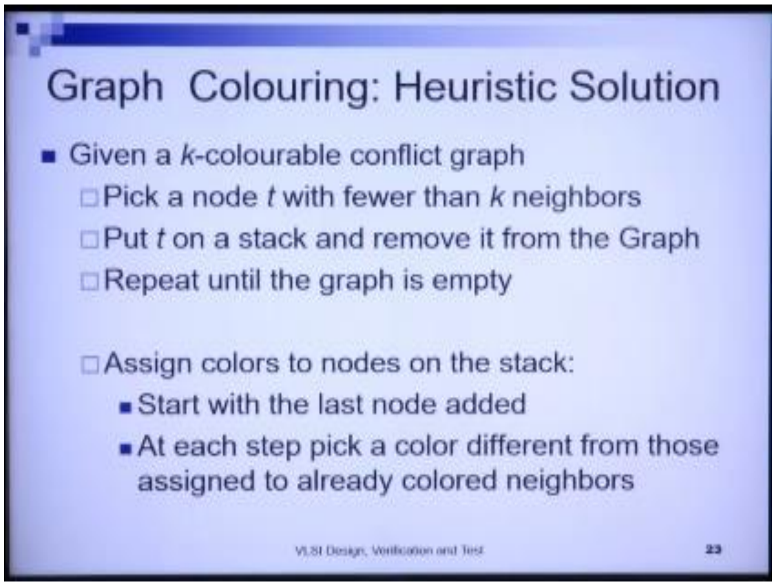
- All operations take two unit time
- Start times $t_a = 1$, $t_b = 3$, $t_c = t_d = 2$
- *c* and *d* being mutually exclusive are compatible
  - Prevents chord $\{v_c, v_d\}$ in the conflict graph
  - The conflict graph is not a chordal graph

TIME 1
TIME 2
TIME 3
TIME 4
a
b
BR
NOP
NOP
c
d
NOP
NOP
a
b
c
d
a
c
d
b
Conflict Graph

VLSI Design, Verification and Test 22

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 a general graph is by the graph coloring method.

(Refer Slide Time: 21:09)



Graph Colouring: Heuristic Solution

- Given a *k*-colourable conflict graph
  - Pick a node *t* with fewer than *k* neighbors
  - Put *t* on a stack and remove it from the Graph
  - Repeat until the graph is empty
  - Assign colors to nodes on the stack:
    - Start with the last node added
    - At each step pick a color different from those assigned to already colored neighbors

VLSI Design, Verification and Test 23

That we just studied we come to the end of module 2 of lecture 7.

Rekha Hazarika

CET Administrative Team

Susanta Sarma

Swapan Debnath