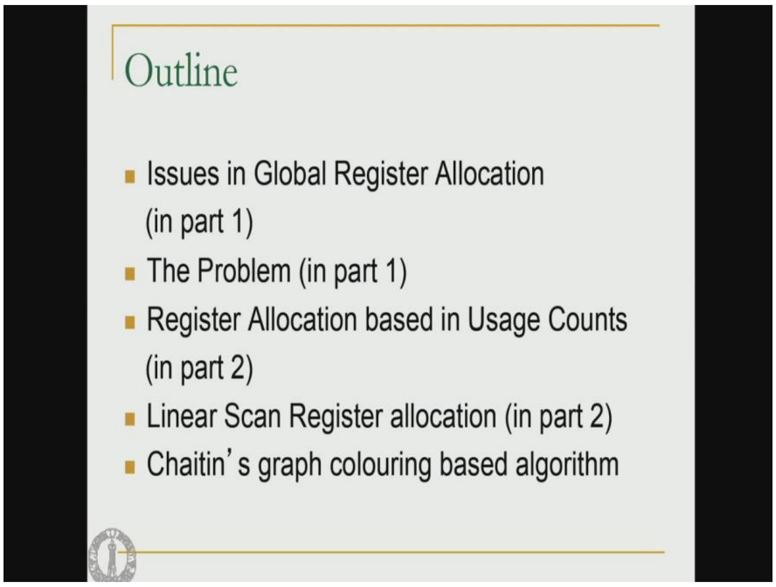
Principles of Compiler Design

Prof. Y. N. Srikant
Department of Computer Science and Automation
Indian Institute of Science, Bangalore

Lecture - 30
Global Register Allocation Part – 3

(Refer Slide Time: 00:30)

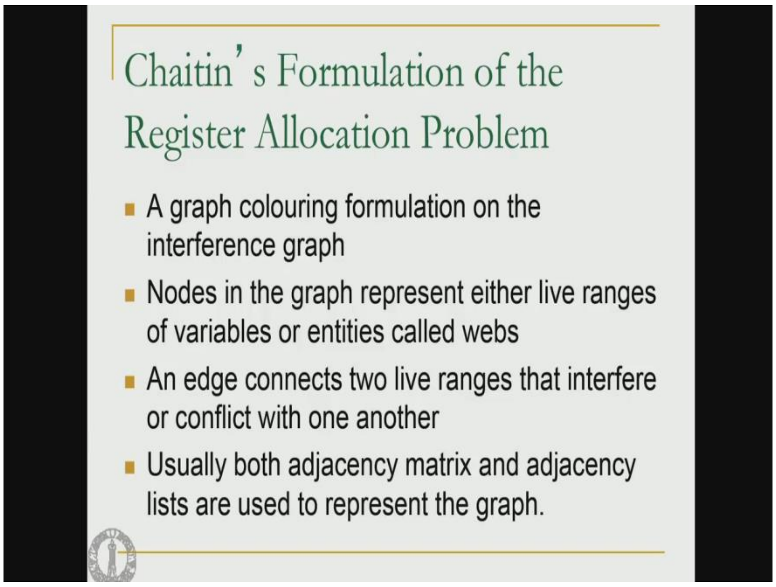


Outline

- Issues in Global Register Allocation (in part 1)
- The Problem (in part 1)
- Register Allocation based in Usage Counts (in part 2)
- Linear Scan Register allocation (in part 2)
- Chaitin's graph colouring based algorithm

Welcome to part three of the lecture on a Global Register Allocation. So, in part 1, we looked the issues in global register allocation and we also discussed the definition of what is global register allocation and the problem itself. We also saw register location for loops, which is based on the principle of usage counts that was in part 2, and a very fast linear scan register location was also discussed. Now, we today we are going to discuss graph coloring based algorithm due to Chaitin, so this is probably the most complicated and most you know efficient register locations scheme that is possible for programs.

(Refer Slide Time: 01:18)



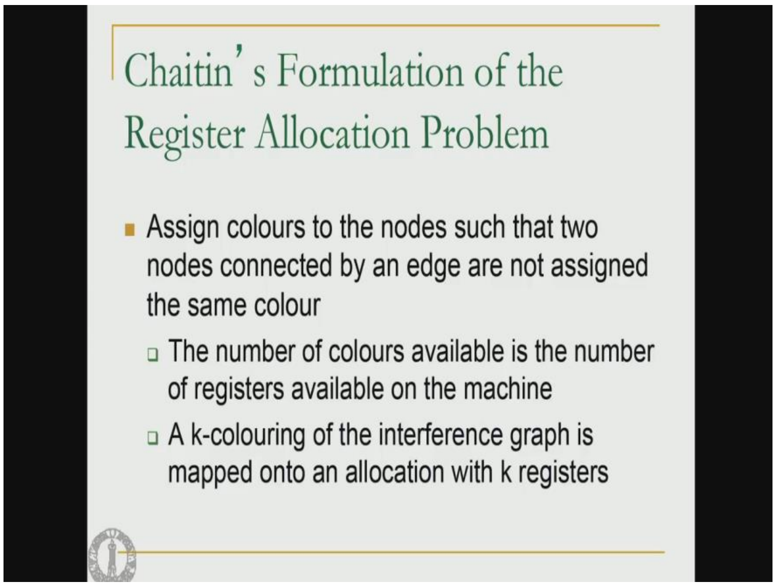
Chaitin's Formulation of the Register Allocation Problem

- A graph colouring formulation on the interference graph
- Nodes in the graph represent either live ranges of variables or entities called webs
- An edge connects two live ranges that interfere or conflict with one another
- Usually both adjacency matrix and adjacency lists are used to represent the graph.

The formulation of the problem is based on graph coloring and the graph underlining the problem is what is known as interference graph. So, when we say interference graph we also need to mention, which are the nodes and which are the edges in the interference graph, nodes in the graph are either the live ranges of variables or entities called webs. So, we know what live ranges are these are the points at which the particular variable is live, so from the definition you know to the last use of that particular definition, so that is called as a live range, so nodes in the graph represent live ranges of variables.

Webs are extensions of these and we will not deal with them in this lecture, an edge connects two live ranges that interfere or conflict with one another. So, basically if there are two variables and both of them are active in the program at the same time, it is very clear that we cannot assign the same register to both these variables. So, the same concept can be extended to the live ranges, so if there are two live ranges which are active at the same time they are said to interfere or conflict with one another, so we add an edge in the interference graph between two nodes which are conflicting. And as I mentioned in the last part we require both the adjacency matrix and the adjacency list to represent the graph.

(Refer Slide Time: 03:01)

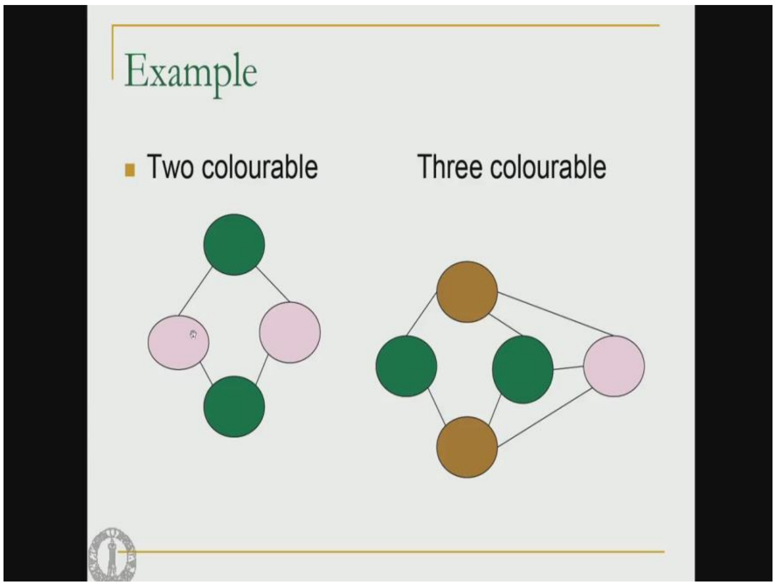


Chaitin's Formulation of the Register Allocation Problem

- Assign colours to the nodes such that two nodes connected by an edge are not assigned the same colour
  - The number of colours available is the number of registers available on the machine
  - A k-colouring of the interference graph is mapped onto an allocation with k registers

The basic idea is to assign colors to the nodes, such that node to nodes connected by an edge or you know assigned the same color. So, you know if you assign the same color it implies that we are assigning the same register that is the reason why interfering nodes cannot be assign the same color. The number of colors available is the number of registers in the machine and k coloring of the interference graph can be mapped onto a graph you know a register location problem with k registers.

(Refer Slide Time: 03:41)



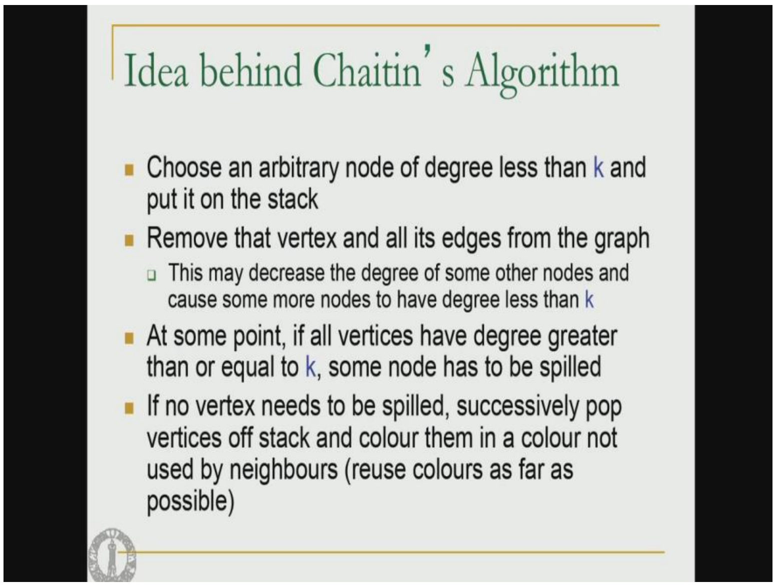
Example

- Two colourable

Three colourable

So, I showed you this example in the last part also this is a two colorable graph; that means, the program corresponding to this graph actually uses only two registers and the allocation of the registers using this fashion. The variable corresponding to this will be given a register corresponding to this violet rather light pink color, this is also given the same register this live range, but these two live ranges are given a different register. This is a three colorable program or the interference graph corresponding to the program and it is similar in you know spirit. So, these two are given the same register these two live ranges are given the same register and this is given a different register.

(Refer Slide Time: 04:40)



Idea behind Chaitin's Algorithm

- Choose an arbitrary node of degree less than k and put it on the stack
- Remove that vertex and all its edges from the graph
  - This may decrease the degree of some other nodes and cause some more nodes to have degree less than k
- At some point, if all vertices have degree greater than or equal to k, some node has to be spilled
- If no vertex needs to be spilled, successively pop vertices off stack and colour them in a colour not used by neighbours (reuse colours as far as possible)

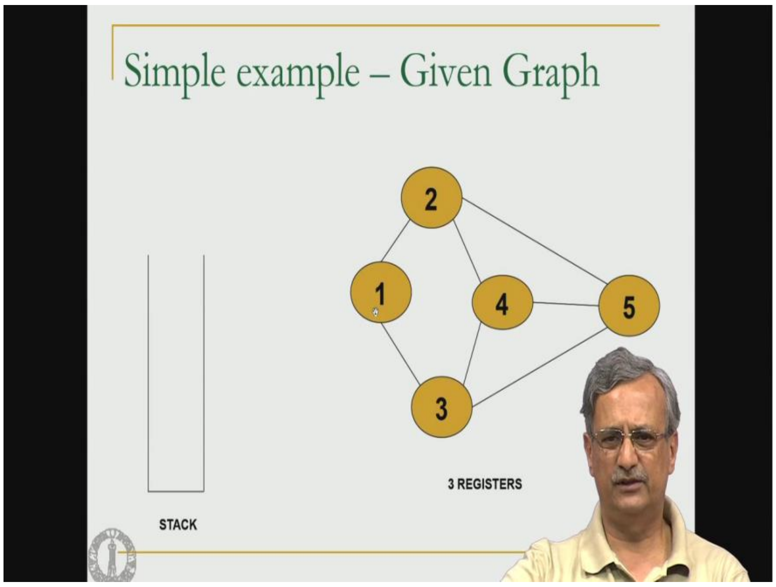
So, what is the idea behind Chaitin's algorithm, so basically the idea behind Chaitin's algorithm is one of you know graph reduction. So, by graph reduction we mean picking up the arbitrary nodes of the graph which have degree less than k, you know and then we remove the nodes. So, typically we pick an arbitrary node of degree less than k and first put it on the stack, so that is what this says.

So, why degree less than k and why not equal to or greater that would be the question, we will see the reason for this very soon. But, basically the idea is if we remove any arbitrary node of degree less than k and then color the graph of with the remaining nodes and edges, then we can show that it is possible to color the original graph with this node and corresponding edges included.

So, when we remove the vertex we have to remove all the edges connected to that vertex as well and this process may decrease the edges of some other nodes and cause more nodes to have degree less than k. So, this is a repetitive process we go on doing it and we reach a point where there are no more nodes in the graph, so that is one possibility that if we reach that stage, then you know we can simply color the nodes appropriately we will see how with an example. And the other possibility is at some point all the vertices have degree greater than or equal to k. So, in this case we cannot continue the reduction process further and we have to resort to what is known as spilling.

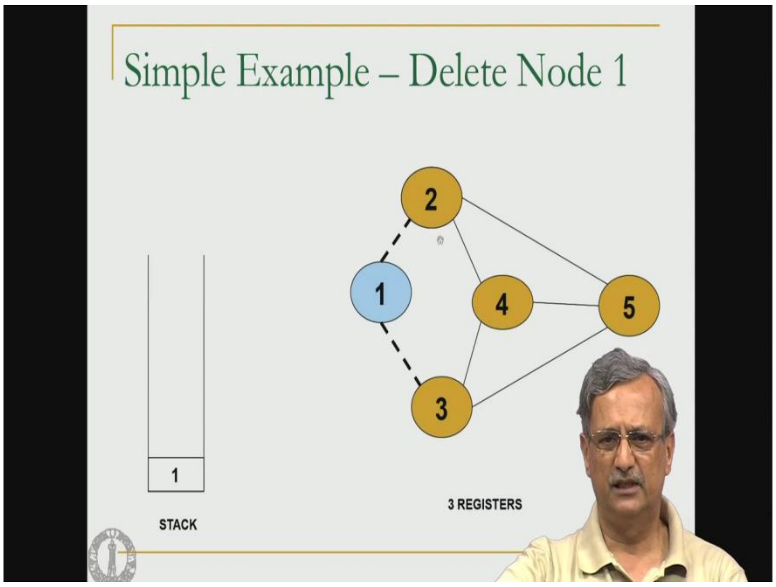
So, spilling implies that the live range or the variable corresponding to that particular node we will not be assigned a register, but it will be actually placed in memory location all the time, so it is not going to get a register at all. So, if a vertex actually gets spilled then you know we are going to remove the vertex from the graph as if it was a reduction process and then continue further.

(Refer Slide Time: 07:16)


Simple example – Given Graph

Let us a look at this simple example to begin with the stack is empty, we have this small interference graph and let us assume that there are three registers; that means, for the reduction process, we must look at those nodes which have degree a you know two or less. So, node one has degree 2, this node has degree 3, so that is not eligible for the reduction and same is true for 3 as well 4 also has degree 3 and 5 also has degree 3, so we are actually you know restricted to node number 1. So, we when we remove node number 1 the two edges corresponding to this node number 1 attach to it are also remove and node number 1 is going to be placed on stack.

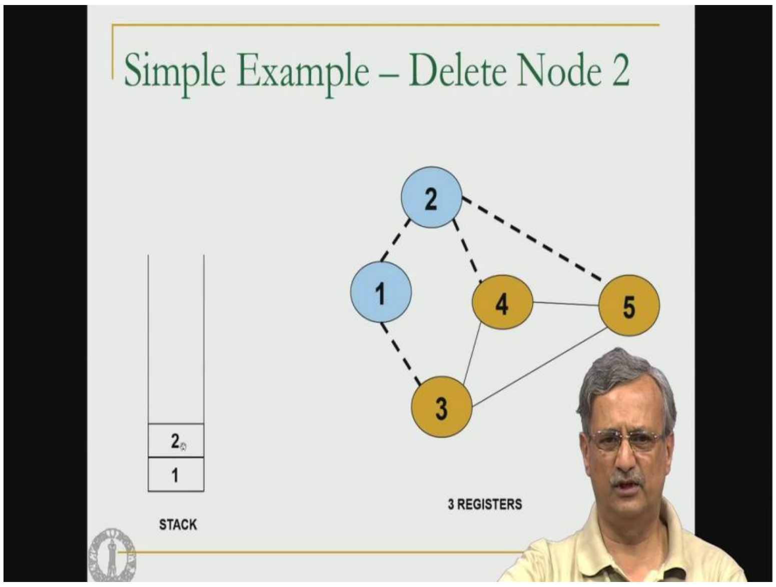
(Refer Slide Time 08:05)



Simple Example – Delete Node 1

So, one is placed on stack, so the dotted edges imply that they have been removed. Now, node number 2 and node number 3 have just two edges each, because the third one is gone, so we can pick either node 2 or node 3 and continue the reduction operation.

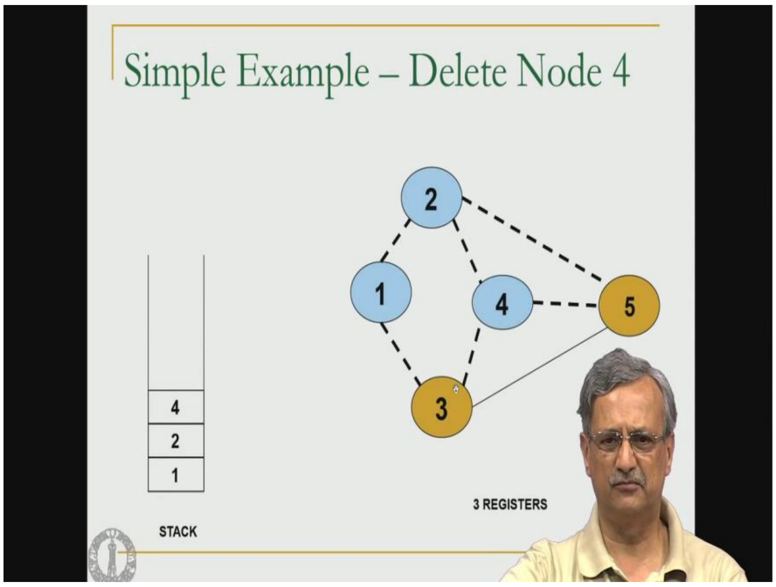
(Refer Slide Time 08:27)



Simple Example – Delete Node 2

So, let us choose node number 2 remove node two and the two edges you know attach to it. So, that leaves us with the graph of three nodes 3, 4 and 5 each of the nodes in this small graph actually have just degree 2, so any one of them can be chosen for the reduction process for the next step in the reduction process.

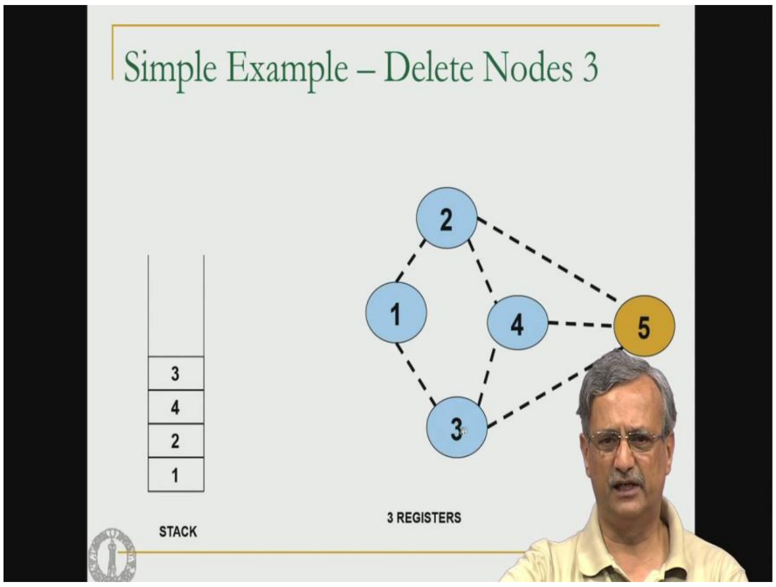
(Refer Slide Time 08:49)



Simple Example – Delete Node 4

Say we choose four we remove 4 and the two edges related to it 4 is put on the stack, so we have 3 and 5.

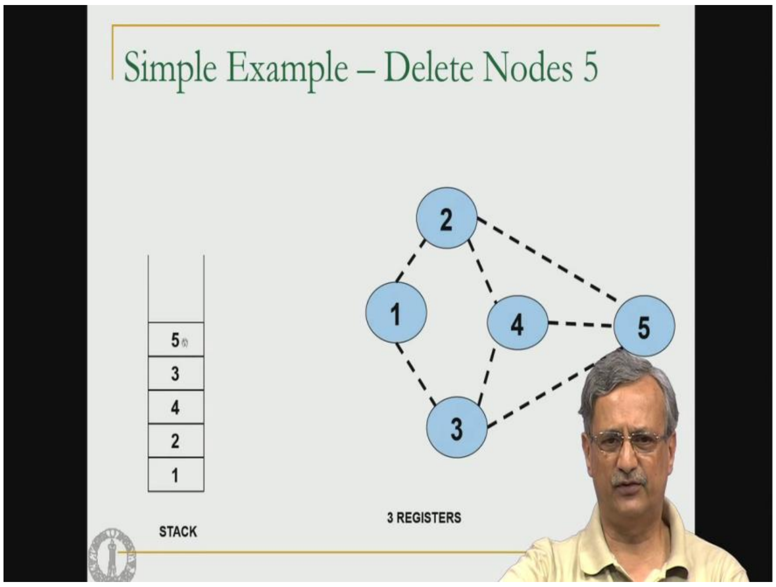
(Refer Slide Time 08:58)



Simple Example – Delete Nodes 3

So, we remove three put it on stack, remove the edge you know related to it have just node number 5.

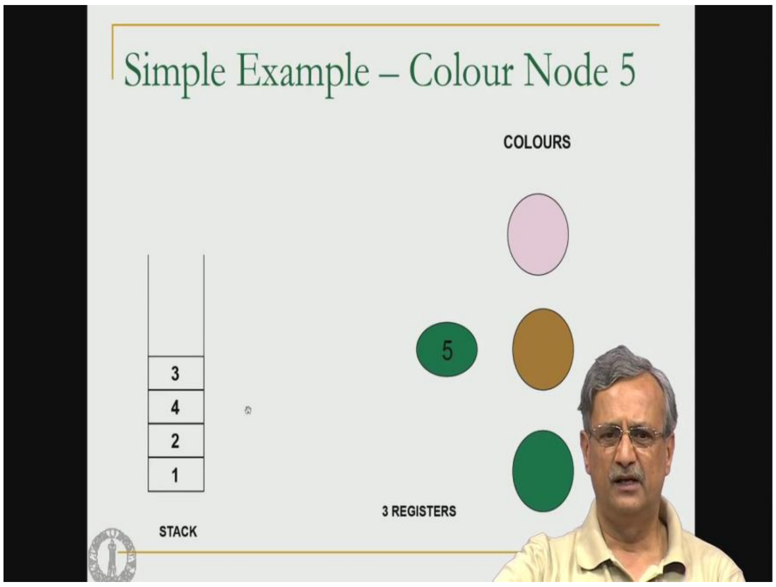
(Refer Slide Time 09:07)



Simple Example – Delete Nodes 5

So, 5 can also be remove and placed on the stack, now the graph is empty, so this is a completely you know reduce it could reduced completely. So, the graph can be colored with three registers, the process of coloring the graph starts in the a order of nodes placed you know the reverse order of nodes that have been placed on the graph; that means, we will start coloring with five then go to 3, 4, 2 and 1, let us see how it proceeds.

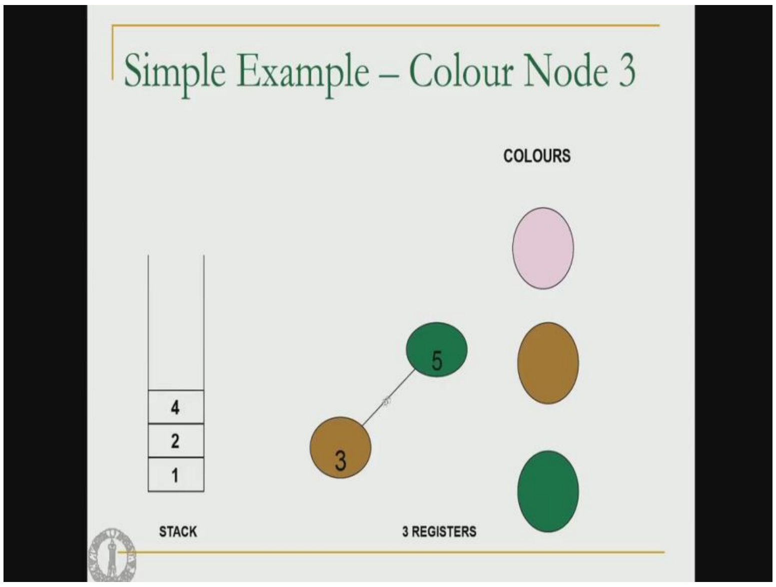
(Refer Slide Time 09:36)



Simple Example – Colour Node 5

So, we pick node number 5 and that is restored to the graph, so and we assign a color to it there are three colors that we have, so node number five is given a particular color say green right, so that is allocating it a register.

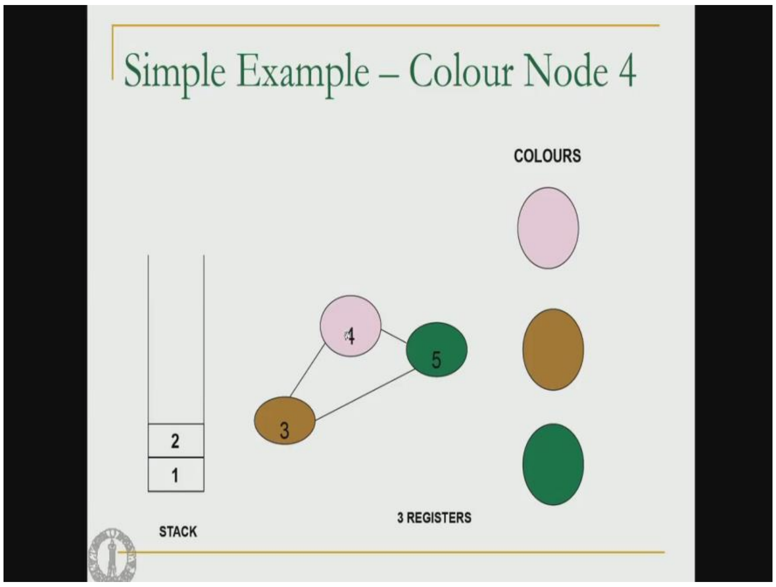
(Refer Slide Time 09:55)



Simple Example – Colour Node 3

Now, restore node number three and the edge with it, so we have three it cannot be given the same color as 5, so we have we pick another color from the free colors here and then assign the color to 3.

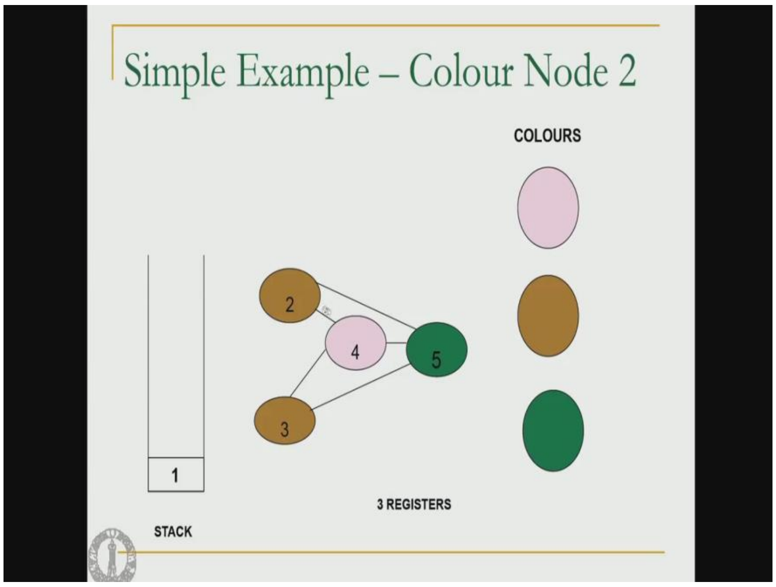
(Refer Slide Time 10:12)



Simple Example – Colour Node 4

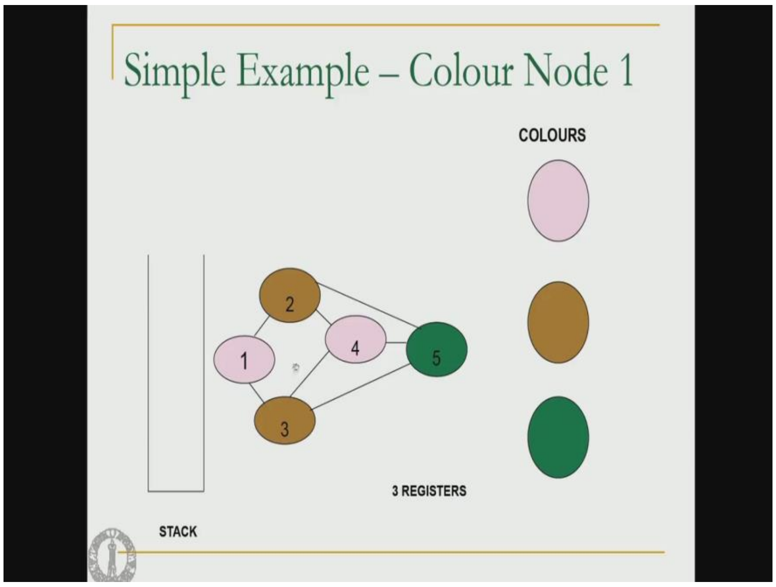
Then number four is reintroduced into the graph along with a edges, so it is connected to both 3 and 5, so we need a third color for 4 which is available assign that to node number 4.

(Refer Slide Time 10:25)


Simple Example – Colour Node 2

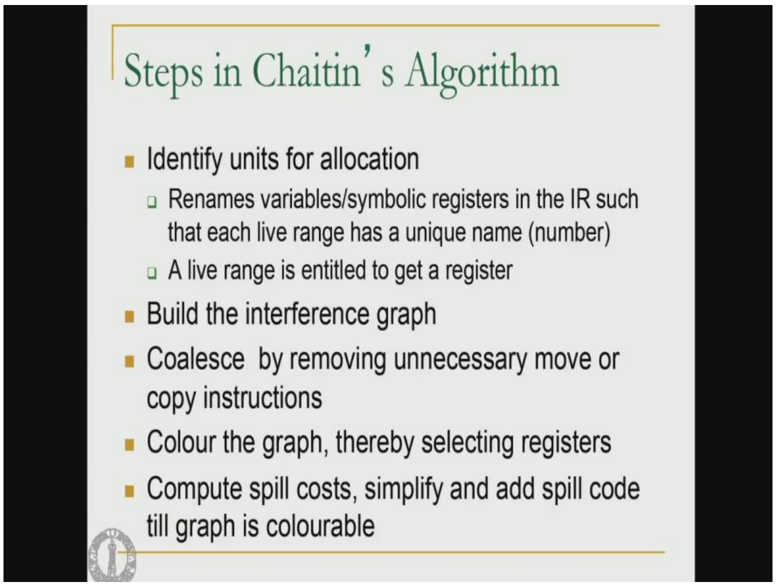
Now, 2 is introduced 2 is connected to 4 and 5, so we must give a color which is different from that of 4 and 5 and that can be brown.

(Refer Slide Time 10:38)


Simple Example – Colour Node 1

Now, we introduce 1 again 1 is connected to both 2 and 3, so you cannot actually give the color of either 2 or 3 we can give one of the free colors. So, usually we try to you know assign a color which is more frequently used, so that is 4, so we assign 1 to it and that completes our example.

(Refer Slide Time 11:03)



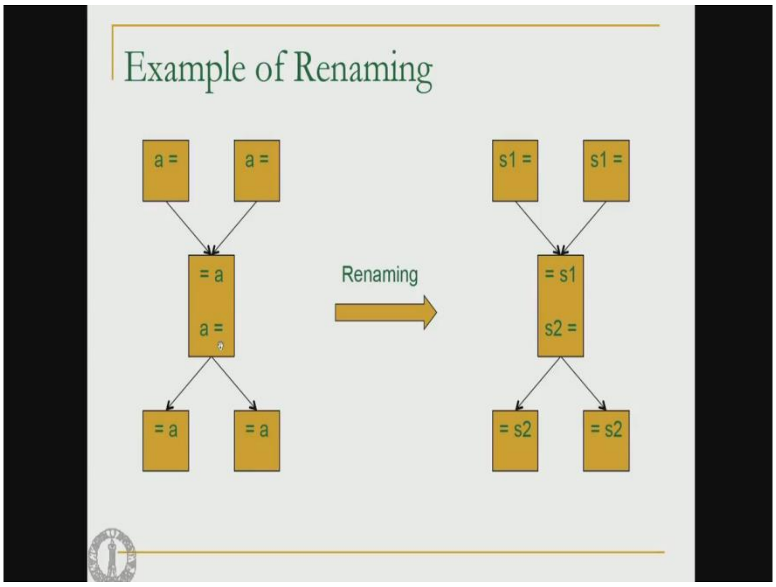
Steps in Chaitin's Algorithm

- Identify units for allocation
  - Renames variables/symbolic registers in the IR such that each live range has a unique name (number)
  - A live range is entitled to get a register
- Build the interference graph
- Coalesce by removing unnecessary move or copy instructions
- Colour the graph, thereby selecting registers
- Compute spill costs, simplify and add spill code till graph is colourable

So, let us look at the steps in Chaitin's algorithm in more detail, the first step is to identify the units for allocation. So, in doing this there is a process of renaming the variables which have I have to be under taken, so it is possible that the same definition you know same variable has more than one live range attached to it. In other words it is possible that I define a variable and then have a couple of uses for that variable.

Define the variable once again have a few more uses, use define the variable a third time have some more uses, this can go on any number of times each of these definition, use ranges qualified to be a live range on it is own. So, we make these three live ranges as three different you know variables in the program, so that the program now does not reassign a variable you know unless it is absolutely necessary, so I will show you when it becomes absolutely necessary with an example. So, once the renaming of the variables you know is completed we have each live range having a unique name and a live range is now entitled to get a register.

(Refer Slide Time 12:44)


Example of Renaming

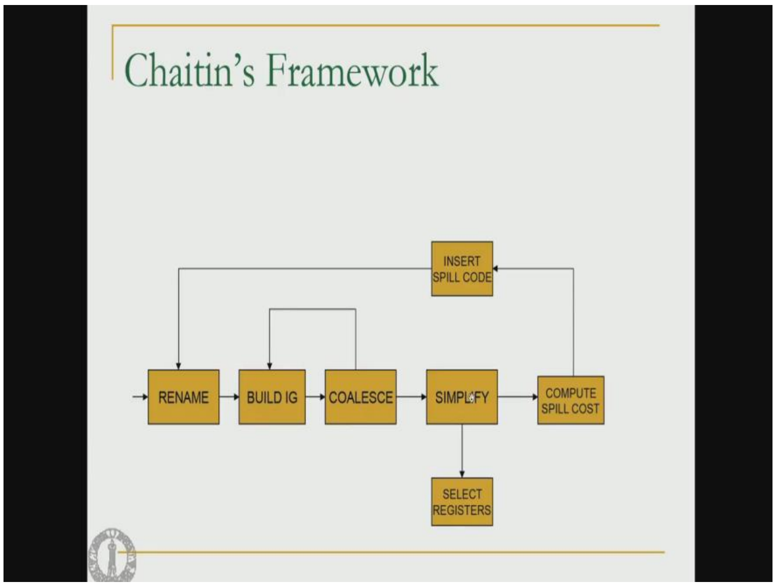
So, let us look at the example, so in this small example we have a equal to an assignment, a equal to another assignment, equal to a usage of a, a equal to one more assignment, equal to a usage of a and here is equal to a usage of a. The renaming process makes both these definitions as s 1 equal to and s 1 equal to that is because the value which is reaching this a can be either from this branch or from this branch.

So, even though these two are two different definitions, we actually place them in the same live range and so this, this and this ((Refer Time: 13:28)) together come in to the same live range and therefore, they have the same variable s 1. Whereas the second definition of a here rather the third definition of a is not related to any of these and it can be assigned a unique name s 2 and obviously, the two uses will also be marked as s 2.

So, there is there are two live ranges here this is one live range and this is another live range. So, this has the name s 1 and this has the name s 2. So, once the live ranges have been renamed we build the interference graph, so we are going to look at the steps needed to build the interference graph very soon. After building the interference graph there is what is known as coalescing of the copy instructions or this is also called as removing the unnecessary move or copy instructions. So, I will give you examples of this as well, after that we color the graph that is the reduction process is applied thereby we select the registers.

Suppose, we get stuck during the coloring of the graph and the graph cannot be reduced further we need to you know spill one of the variables as I mentioned. So, we spill we computes spill cost then chose a particular node to spill then remove that nodes simplify the graph add the spill code to the original program. And now we again do all these steps once more you know a not the here, but this particular step from building the interference graph onwards until the graph becomes completely colorable.

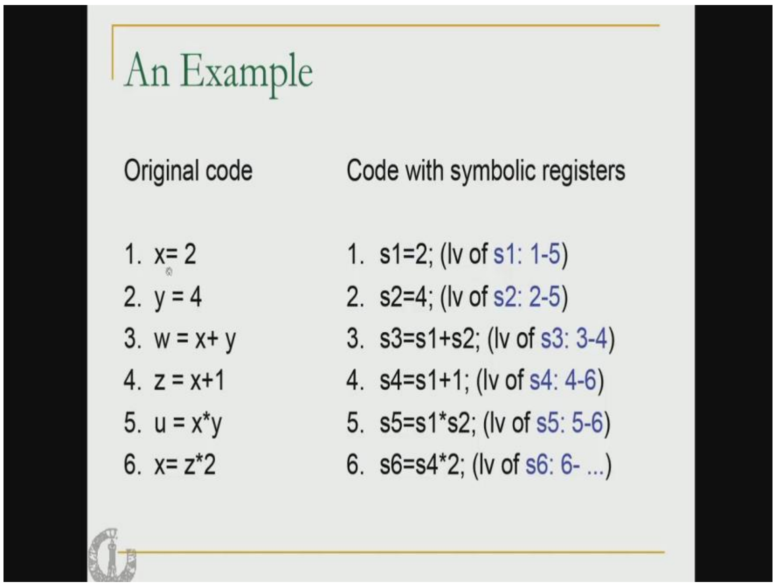
(Refer Slide Time 15:23)



Chaitin's Framework

So, let us now look at these steps in more detail, so this is the iterative process rename build coalesce is in a loop and then simplify is the reduction process, if it is complete then we select registers, otherwise we compute the spill cost insert the spill code and go back to renaming building etcetera, etcetera if renaming is not necessary we could directly go to build.

(Refer Slide Time 15:48)



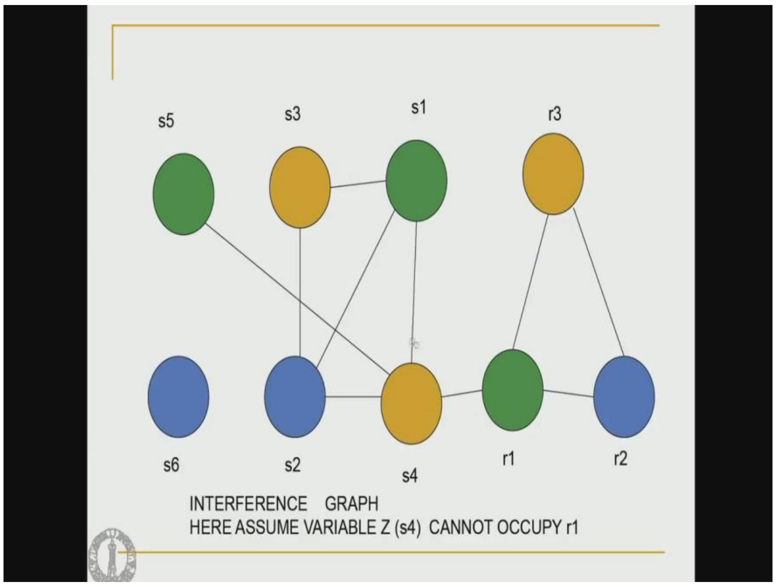
An Example

Original code	Code with symbolic registers
1. x= 2	1. s1=2; (lv of s1: 1-5)
2. y = 4	2. s2=4; (lv of s2: 2-5)
3. w = x+ y	3. s3=s1+s2; (lv of s3: 3-4)
4. z = x+1	4. s4=s1+1; (lv of s4: 4-6)
5. u = x*y	5. s5=s1*s2; (lv of s5: 5-6)
6. x= z*2	6. s6=s4*2; (lv of s6: 6- ...)

So, let us take a simple example for the renaming, so we have x equal to 2 and we have x equal to z star 2 we have y equal to 4 and then we have w z and u being assign. So, let us just rename them, so this x gets s 1 and this x gets a different name called s 6, so the others of course, could have been retained as it is. But, just to make sure that the intermediate code is inform we have assigned new names to all of them otherwise it was not really essential only this first x and the last x needed different names.

Now, what about the live ranges, so if you look at the definition of s 1 till you know the last use of s 1 is the live range of s 1. So, from instruction 1 to instruction 5, we have the live range of s 1. So, similarly for s 2 it is says it is from 2 to 5, 2 and then s 2 last uses 5 l v of three is 3, 4. So, it is just that you know instead of just one instruction we have just said the instruction 3 and the next 1 and 4 it is 4 to 6 here s one is used and 5 also you know there is no usage of s 1 s 4 here sorry usage of s 4 here and then there is usage of s 4 here as well, so 4 to 6 right. So, then we have s 5 5 and 6, so again instead of marking just one instruction we have marked 5 to 6 and the usage of s 6 will be from this point onwards we do not know about that, so this is the way we calculate the live ranges.

(Refer Slide Time 17:56)



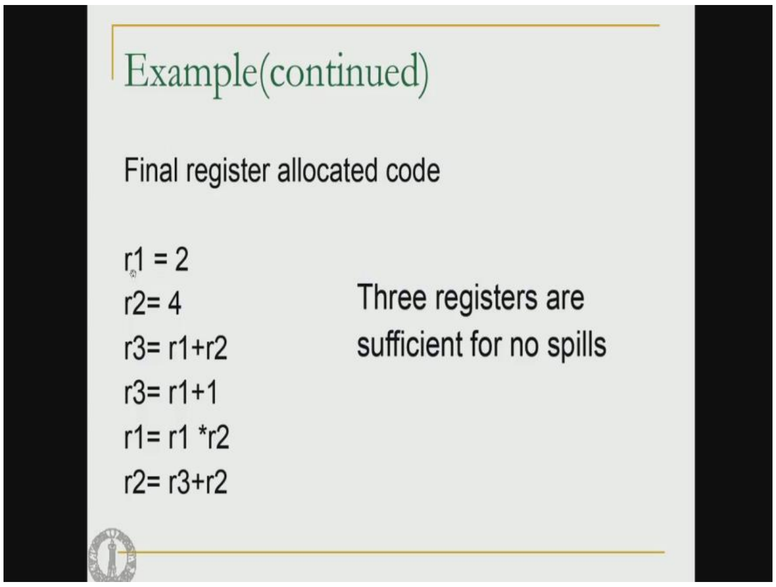
And once the live ranges are calculated we you know compute the interference graph. So, the nodes of the interference graph the, you know symbolic registers, unique symbolic registers that we have shown in the previous example. So, and the edges between these show the interferences, so I will come to r 1, r 2 and r 3 shortly. So, let us focus on a s 1 it says it interferes with s 3, s 2 and s 4, let us verify it with our example here.

So, when I have a say interfere you know we just check whether there is the overlap of the live ranges. So, for example, here this s 1 interferes with two because 1 to 5 and 2 to 5 they have overlapping ranges and then s 3 again you know it is 3 to 4, so that also overlaps with this right. So, so s 1 interferes with s 3 it interferes with s 2 and it interferes with 4 also. So, finally, we have s 4, which is going from 4 to 6, so that also interferes with s 1 right, so s 2, s 3 and s 4 see that 5 to 6 does not interfere with this and 6 onwards also does not interfere with s 1. So, this is the way we actually look at the interference, similarly s 5 interferes with s 4, s 4 interferes with s 2 and of course, s 1 and so on and so forth.

Now, so coming to the of course, the colors would not have been assign and right in the beginning, so that is important, coming to these three you know nodes which are marked as r 1, r 2 and r 3, r 1, r 2 and r 3 correspond to the three physical registers, which are present in the machine. Whenever there is a constraint that one of the registers cannot be used or interferes for with a particular live range, then we introduce these physical registers and an add a edge between that particular live range and the register.

So, in this case the constraint is that s 4 cannot be given the register number r 1, so we have added an edge between s 4 and r 1. If there were no restriction of this kind and any variable could have could be given any register then adding these three registers to the interference graph is not really necessary, but in the absence of that we need to add this as well.

(Refer Slide Time 20:57)



Example(continued)

Final register allocated code

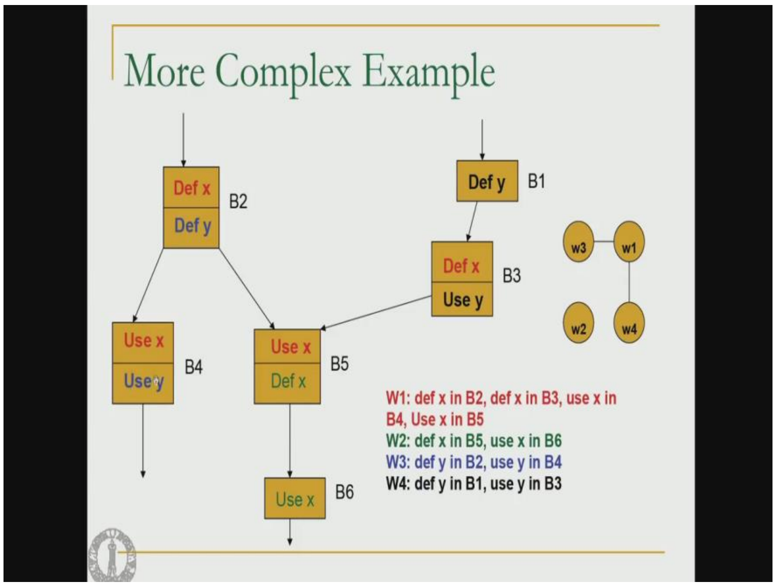
```
r1 = 2
r2= 4
r3= r1+r2
r3= r1+1
r1= r1 *r2
r2= r3+r2
```

Three registers are sufficient for no spills

Continuing this example after the register location or the coloring is performed. So, coloring actually ((Refer Time: 21:08)) has been done very simply with one, two and three colors. So, for example, you know we could start with this push it on to the stack, then start with this you know, then go on to this push this also on to the stack, we have three registers. Then you know this edge will also go we take out this and the two edges then we are left with these rest of it, we could take out either this or this and then you know the other one. Finally, we take out this and of course, we have already assigned the colors to r 1, r 2 and r 3, because they are physical registers.

So, the reverse process will assign the colors as I explained in the previous example, having done that we could assign the registers corresponding to these colors and rewrite the code in the form of using a registers. So, now, r 1 equal to 2 and r 1 equal to r 1 star r 2 indicates that the assignment you know for the same variable is taking place in the same register, this is just a coincidence; it is possible that you know r 1 was assigned some other live range you know you know once it becomes not useful.

(Refer Slide Time 22:31)



More Complex Example

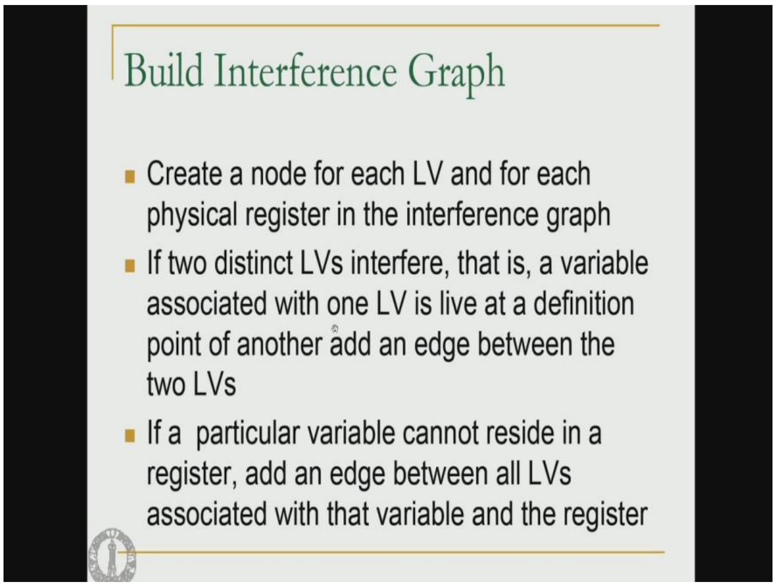
W1: def x in B2, def x in B3, use x in B4, Use x in B5
W2: def x in B5, use x in B6
W3: def y in B2, use y in B4
W4: def y in B1, use y in B3

Let us take one more example, so here this is a basic block these are all basic blocks B 1, B 2, B 3, B 4, B 5, B 6. So, in this basic block we have a definition of x and a definition of another variable y here we have a usage of x and then followed by usage of y, here we have a usage of x and then a definition of x this has a usage of x here is a definition of y a definition x and usage y.

So, if you look at the live ranges right, so the live ranges here are called W 1, W 2, W 3 and W 4. So, if you look at W 1, it says def x in B 2 def x in B 3 use x in B 4 and use x in B 5 all red. So, this, this ((Refer Time: 23:28)), this and this, these four together form one live range, because these are inter related you known we cannot assign a different variable to these two definitions. Whereas this x now can be assign a different live range, because it does not interfere with any of these.

The second one is here def x and use x, then the third one is here def y and use y and the fourth one is again def y and use y none of them interfere with each other, so they are made into different live ranges you know. So, once we have these different live ranges they are associated with different variables W 1, W 2, W 3 and W 4 the interference is also noted W 1 interferes with both W 3 and W 4. So, if you look at that red, which is W 1 it is very clearly interfering with def y use y that is nothing but W 3. And then you know it is also interfering with the last one which is W 4, so here, so this is a the way we actually compute the interference graph.

(Refer Slide Time 24:52)

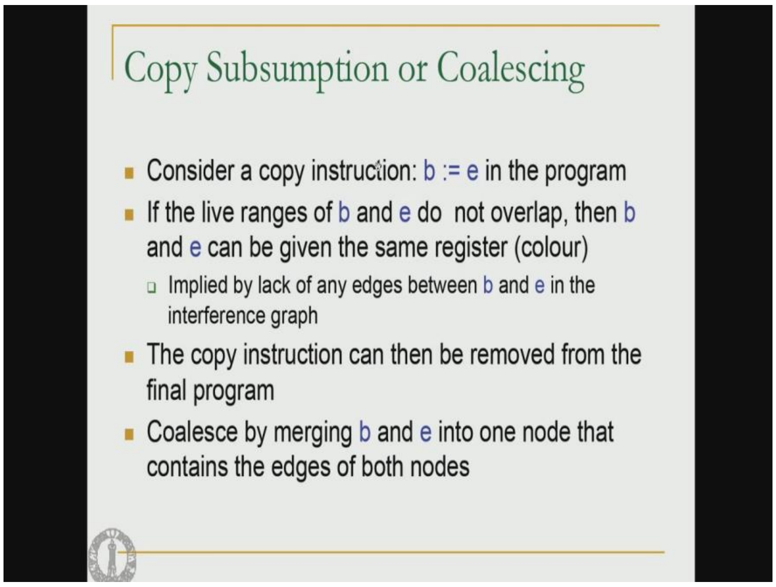


Build Interference Graph

- Create a node for each LV and for each physical register in the interference graph
- If two distinct LVs interfere, that is, a variable associated with one LV is live at a definition point of another add an edge between the two LVs
- If a particular variable cannot reside in a register, add an edge between all LVs associated with that variable and the register

So, here is a description of what I just now explained, so create a node for each live variable and for each physical register in the interference graph, if two distinct live variables interfere that is a variable associated with one live variable is live at the definition point of another add an edge between the two l v's. So, this is what I just now mentioned, if a particular variable cannot reside in a register then add an edge between all l v‘s associated with that variable and the register, so this is the you know extra that we do for physical registers.

(Refer Slide Time 25:34)



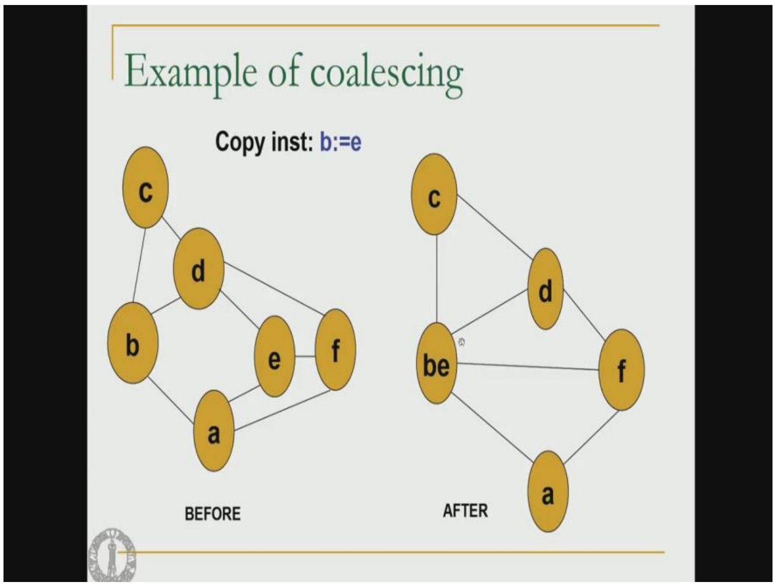
Copy Subsumption or Coalescing

- Consider a copy instruction: b := e in the program
- If the live ranges of b and e do not overlap, then b and e can be given the same register (colour)
  - Implied by lack of any edges between b and e in the interference graph
- The copy instruction can then be removed from the final program
- Coalesce by merging b and e into one node that contains the edges of both nodes

Then the next step is called as copy subsumption or coalescing copy coalescing. So, what happens is that the copy instructions, which are present in the program for example, of the kind b equal e this is a copy. So, whatever is in e's copy to b, so both of them carry the same value there is no other computation, if the presence of such instructions sometimes it is possible to merge the nodes corresponding to the live range of b and the live range e. So, if we are able to do this merging then this called as a copy subsumption operation, if the live ranges of b and e do not overlap; that means, the there is no edge between the live range of b and the live range of e in the interference graph.

Then; obviously, b and e can be given the same register, so they do not overlap we can give them the same register. So, this is the implied by the lack of any edges between b and e in the interference graph I m going to show you an example very soon. The copy instruction can then be removed from the final program and we merge b and e into one node that contains the edges of both the…

(Refer Slide Time 27:04)

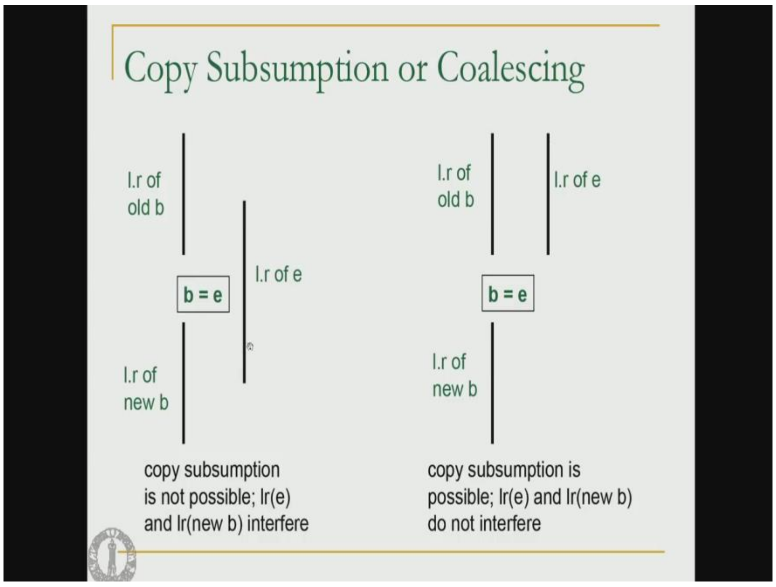


Example of coalescing

Copy inst: b:=e

So, b equal to e is a copy instruction, this is the original interference graph and since there is no edge between b and e; that means, the live range of b and e do not overlap. So, we could merge b and e into a single node, so that becomes the node b e all the edges which are connected to b and e will now go to will connect to the node b e. For example, d c and a connect to node b, so d c and a connect to node b e as well now node f connects to e along with a and d. So, a and d are already connected to b, so they are connected and f also, now connects to the node b, e

(Refer Slide Time 27:58)



Copy Subsumption or Coalescing

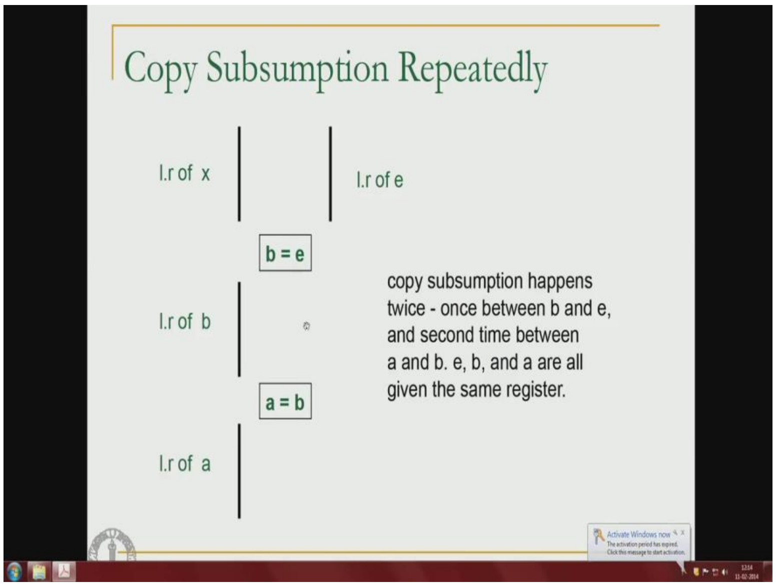
copy subsumption is not possible; lr(e) and lr(new b) interfere

copy subsumption is possible; lr(e) and lr(new b) do not interfere

So, why should we do this, so before understanding why this helps few more examples will help to clarify the situation. So, there is a live range for this old b, then there is a copy operation and there is a different live range of the new b, so here this is the live range of e. In this case the live range of e is actually overlapping the live range of you know new b right, so this is see this and therefore, after we copy it is not possible to assign the same register to both the live range of e and the live range of new b; that means, we cannot actually get rid of this copy instruction, copy subsumption in this case is not possible.

What about this case the live range of old b and the live range of e overlap, but the l r of e and l r of new b do not overlap. Therefore, whatever is the value of e here can be safely assumed to be the value of b since there is no other modification to the variable e. So, copy subsumption is possible here, because they do not interfere and in this case we remove the copy instruction.

(Refer Slide Time 29:31)

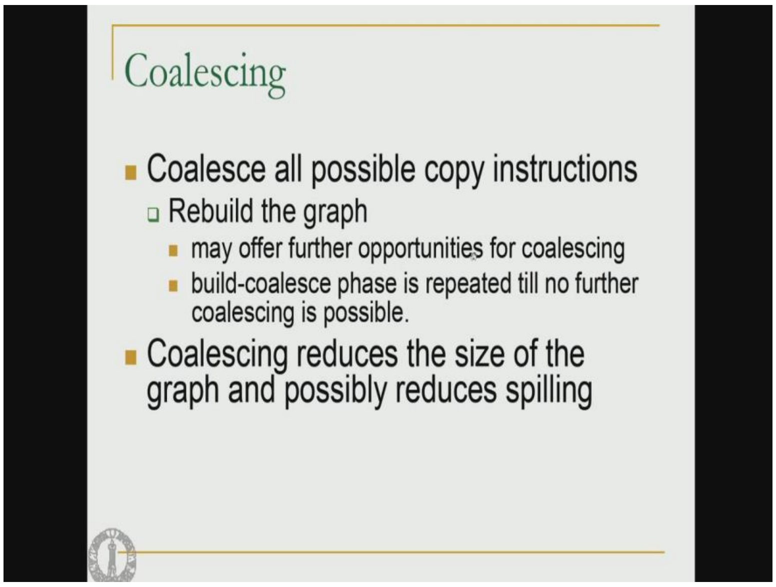


Copy Subsumption Repeatedly

copy subsumption happens twice - once between b and e, and second time between a and b. e, b, and a are all given the same register.

So, in this case copy subsumption you know is required to be applied repeatedly here is x, here is e and here is old b or rather b, so b equal to e, so we can do away with this copy operation because b and e do not have any overlap. After this there is another copy operation a equal to b since the live ranges of a and b do not overlap we can do away with this copy operation as well and assign the same register to both a and b. So, in effect a, b and e are all given the same register, but of course x gets a different register. So, this shows that copy subsumption perhaps should be applied more than ones in order to get the best benefit.

(Refer Slide Time 30:28)

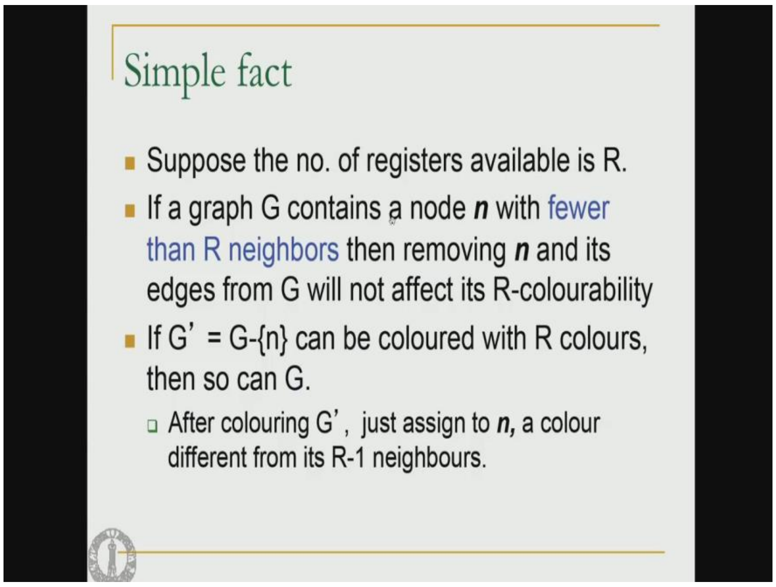


Coalescing

- Coalesce all possible copy instructions
  - Rebuild the graph
    - may offer further opportunities for coalescing
    - build-coalesce phase is repeated till no further coalescing is possible.
- Coalescing reduces the size of the graph and possibly reduces spilling

So, the coalescing operation, coalesce all possible copy instructions the way I just now discuss and then it builds the rebuilds the graph, which may offer further opportunities for coalescing, we do coalescing only you know for one instruction at a time and then rebuild the graph. Build coalesce phases is repeated till no further coalescing is possible, basically coalescing reduces the size of the graph and possibly reduces the spilling operation.

(Refer Slide Time 30:28)



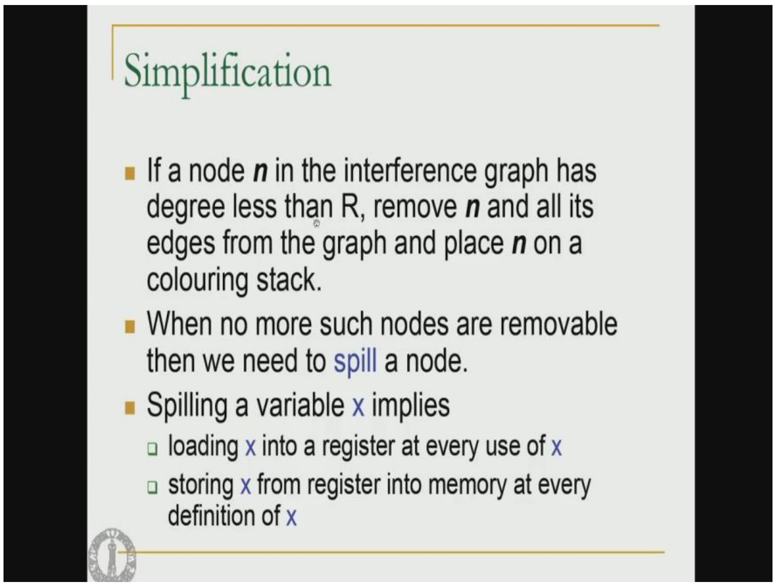
Simple fact

- Suppose the no. of registers available is R.
- If a graph G contains a node ***n*** with fewer than R neighbors then removing ***n*** and its edges from G will not affect its R-colourability
- If G' = G-{n} can be coloured with R colours, then so can G.
  - After colouring G', just assign to ***n,*** a colour different from its R-1 neighbours.

Now, we move on to the step of reduction and we also provide a justification why reduction is correct suppose the number of the registers available is capital R, if a graph G contains a node n with fewer then R neighbors. Then the statements that removing n and it is edges from G will not affect its R color ability, so let us understand why. Basically if g prime which is nothing but G minus n can be colored with R colors then. So, can G let see why basically now you know from G from G we have removed a node and it is a adjust right now that is our G prime.

So, we color G prime and definitely the assumption is it can be colored with R colors after this is over, we look at the node n you know. So, now, there are R minus 1 neighbors for n, so they would have be given in the worst case R minus 1 colors, so the R'th color is still remaining we can assign it to n and thereby complete the coloring. So, the important point is that reduction operation does not affect the color ability of the graph, so after reduction if you are able to color the graph, then we could have done it, before the reduction operation as well this is the justification for the reduction operation.

(Refer Slide Time 32:54)



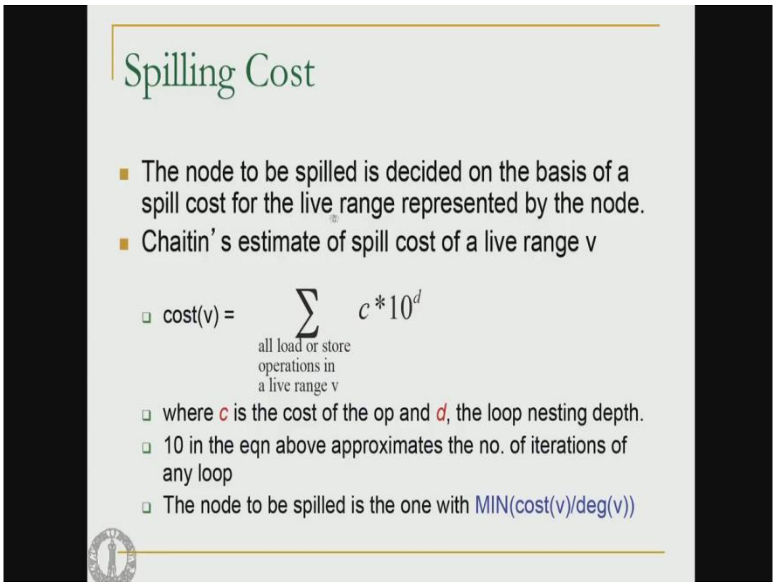
Simplification

- If a node ***n*** in the interference graph has degree less than R, remove ***n*** and all its edges from the graph and place ***n*** on a colouring stack.
- When no more such nodes are removable then we need to spill a node.
- Spilling a variable x implies
  - loading x into a register at every use of x
  - storing x from register into memory at every definition of x

So, if a node n in the interference graph has degree less then R remove node and all its edges from the graph and place n on a coloring stack, so this is the reduction operation, when no more such nodes are removable we need to spill. So, what does spilling mean, spilling a variable x let us look at the fine print now, it implies loading x into a register at every use of x. So, the reason is we say x will not get a when we spill a node it does not get a register, so it goes into its always resident in memory. So, whenever we use a variable x, we will have to load it into a register at every use of x.

So, we cannot do any operation without loading a value into register, that is why we want to move x into a register that is the usage point. And then perform the operation on it using it, at the definition point storing x from register to memory is necessary, because finally, the value of x must reside in the memory location corresponding to x.

(Refer Slide Time 34:12)



Spilling Cost

- The node to be spilled is decided on the basis of a spill cost for the live range represented by the node.
- Chaitin's estimate of spill cost of a live range v
  - cost(v) = $\sum_{\text{all load or store operations in a live range v}} c*10^d$
  - where *c* is the cost of the op and *d*, the loop nesting depth.
  - 10 in the eqn above approximates the no. of iterations of any loop
  - The node to be spilled is the one with MIN(cost(v)/deg(v))

So, I will show you the affect of spilling very soon you know the spill code that is needs to be introduced will be shown very soon, but now let us understand how to chose a node that is to be spilled, so that is done based on what is known as the spilling cost. So, the node to be spilled is decided based on the spill cost for the live range represented by the node. So, how do we compute the spill cost, so here is the live range v, it is cost is sum of all the you know factors c star 10 to the power d over all the load or store operation in live a range v.

So, this is the you know only cost that, why we are considering the load or store operations in a live range, the point is if we spill a node then you know at a every use we need to load and that every definition we need to store. So, these are basically the use and def operation inside the code; what is the cost of each of these load or store operations it is c into 10 to the power d. C is the cost of the operation it could be an addition or you know, it could be a multiplication or division etcetera, etcetera, so multiplied by 10 to the power d.

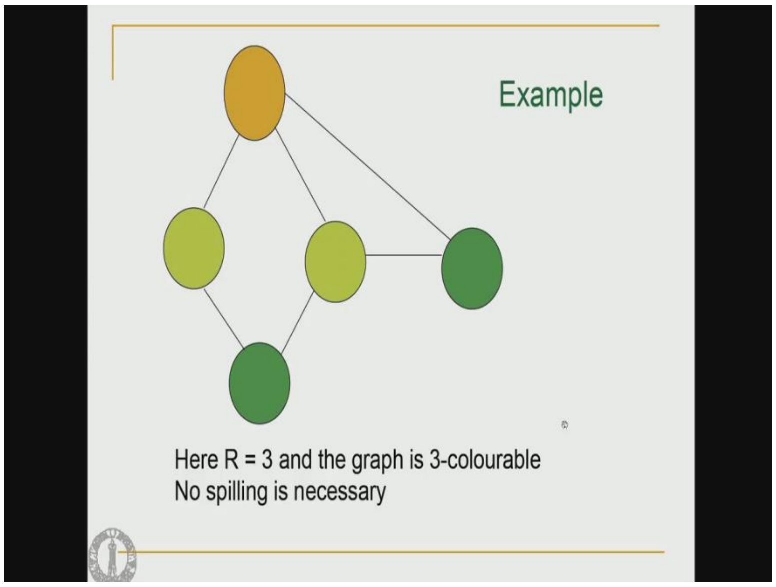
So, d is the loop nesting depth why should we multiply it by 10 to the power d let us understand this by looking at loop nesting of zero, that is the instruction are not nested at all in any loop. So, this d would become 0, 10 to the power 0 is 1 so; that means, we are looking at only the cost of that particular operation that is a load you know for that particular operation. Suppose the instruction is present inside a loop, then obviously, the loop is going to be executed a large number of times.

So, the number of times the use or def you know needs to be loaded or store will also increase depending on the number of times, the loop operates. Experimentation has shown that you know the exact number of times the loop iterates is not really necessary, you know we could simply assign a numbers such as 5 or 10 or 50 or 100, 10 is just a number it could be 50 or 100 as well. It approximates the number of iteration of any loop, so even if you place 100 here and evaluate the you know this cost function.

It is not going to actually make a difference as far as the register allocation is concerned the cost will be; obviously, different we only want to differentiate the number of times the cost of variable which is used in inside a loop as compare to the cost of a variable which is used outside the loop. And this d is necessary to take care of the number of times the, you know the loop nesting has been done. So, if there is a doubly nested loop; obviously, then you know we need to 10 into 10, so this becomes 10 to the power 2 if it is a triply nested loop then they will be 3, because the loop would be executing 10 into 10 into 10, 1000 number of times.

So, this is the way we actually compute the cost of spilling loop, now after we compute the cost for each one of the variable, we compute another quantity called the cost v per degree v, so divided by degree v. And then take the minimum of these and that is the node which is to be spilled. So, this cost v per degree v kind of distributes the cost to the various edges of that particular node v that is why that is what the division operation really indicates degree is the number of neighbors. So, we distribute it to the number of neighbors for that particular live range or node v. So, once we do this the; obviously, the node that has a minimum cost or degree is the node that is to be spilled, because any spilling operation which is very expensive should be avoided the minimum cost spilling operation is a the one which is to be encouraged.

(Refer Slide Time 39:18)

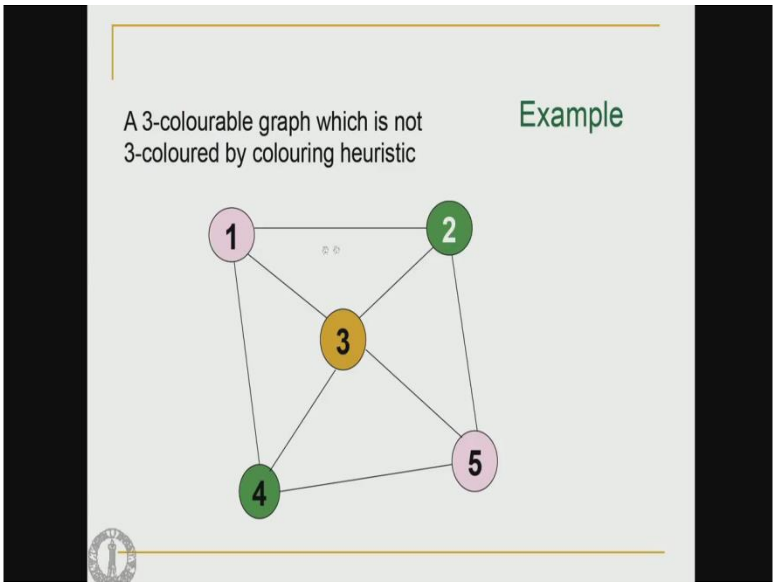

Example

Here R = 3 and the graph is 3-colourable

No spilling is necessary

So, here is a graph which can be colored with a three colors without any spills, so that is easy to see.

(Refer Slide Time 39:31)

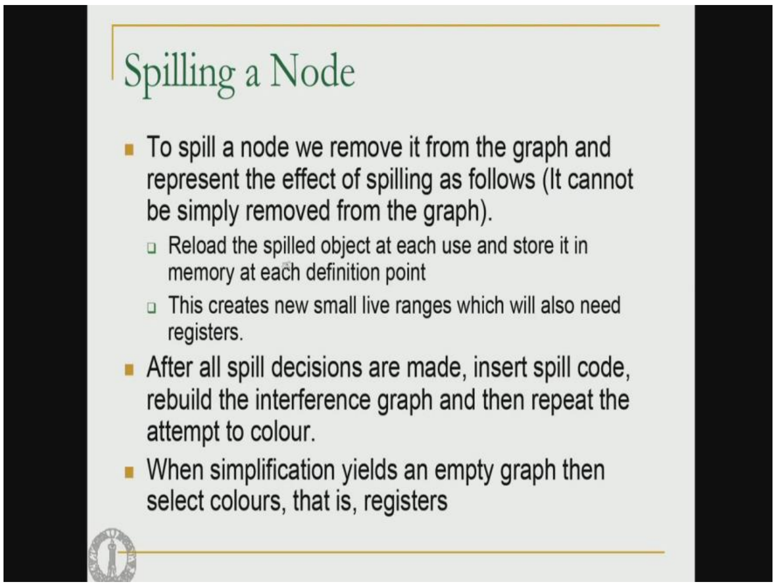

Example

A 3-colourable graph which is not 3-coloured by colouring heuristic

And as I mention in the early part of this register location lecture the graph coloring algorithm, actually graph coloring is n p complete problem in general. And we have used a heuristic which is by way of spilling the node when the graph coloring cannot proceed and because it is a heuristic which has it is shorts comings one problem is there r graph such as this, which can be three colored, but our heuristic shows that it cannot be colored using three color. So, here is a such a graph; obviously, by doing a manual coloring of the graph, we have been able to color it.

But, if we apply the algorithm then you know it is very clear with three colors none of the nodes have degree less than three. So, we cannot actually start the reduction process at all we have to spill a node before the coloring process, reduction process begins. So, whereas, by coloring like this we are able to color it without any spills, so this is a short coming of the limitation of the coloring heuristic.

(Refer Slide Time 40:57)



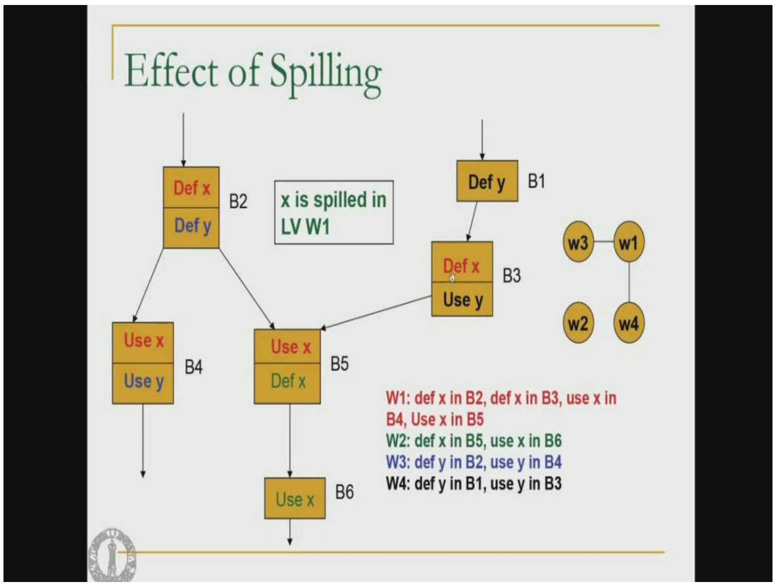
Spilling a Node

- To spill a node we remove it from the graph and represent the effect of spilling as follows (It cannot be simply removed from the graph).
  - Reload the spilled object at each use and store it in memory at each definition point
  - This creates new small live ranges which will also need registers.
- After all spill decisions are made, insert spill code, rebuild the interference graph and then repeat the attempt to colour.
- When simplification yields an empty graph then select colours, that is, registers

So, what exactly do we mean by spilling a node, so to spill a node we remove it from the graph and represent the effect of spelling you know as follows, we reload the spilled object at each use; that means, we introduce a load instruction for every use and we store it in memory at each definition point. That means, we introduce a store instruction at after the immediately after the definition is over. So, remember we have a load instruction just before usage and we have a store instruction immediately after a definition. So, it, so happens that this is creates very small live ranges for the load and store points and again these we cannot perform any operation without registers, so this load and store also require registers.

They require they have small live ranges of their own, which will also have to be given registers and this is the reason why after the spill decision are made, we insert the spill code rebuild the interference graph and repeat the attempt to color. So, this make sure that even this small live ranges get the registers appropriately, when simplification yields an empty graph then it is time to select the colors that is nothing but the registers.

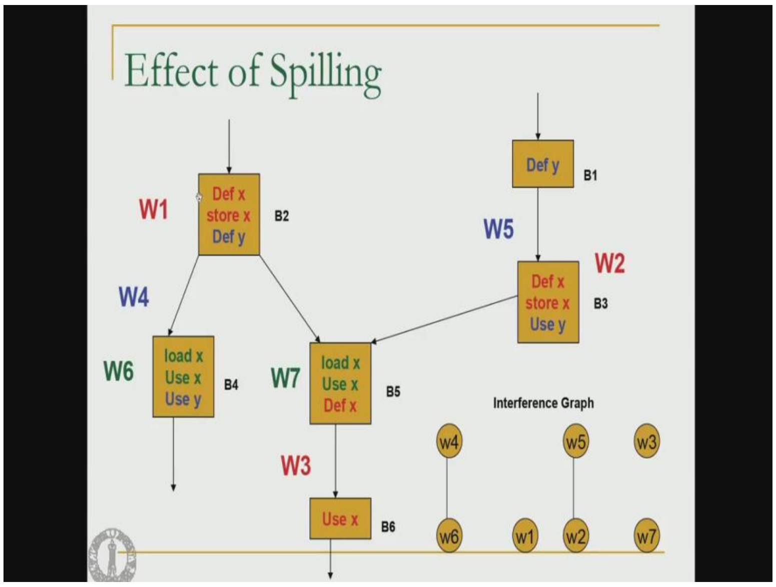
(Refer Slide Time 42:34)



W1: def x in B2, def x in B3, use x in B4, Use x in B5
W2: def x in B5, use x in B6
W3: def y in B2, use y in B4
W4: def y in B1, use y in B3

So, I want to show you the effect of spilling here, so this is our original example, you know with W 1, W 2, W 3, W 4 as the four live ranges. So, let us assume that the variable x is spilled in the live range W 1; that means, just before just after this definition of x, you must have a store instruction, just before the use of x we must have a load here also we must have a load just before the use and just after this def you must have a store.

(Refer Slide Time 43:15)

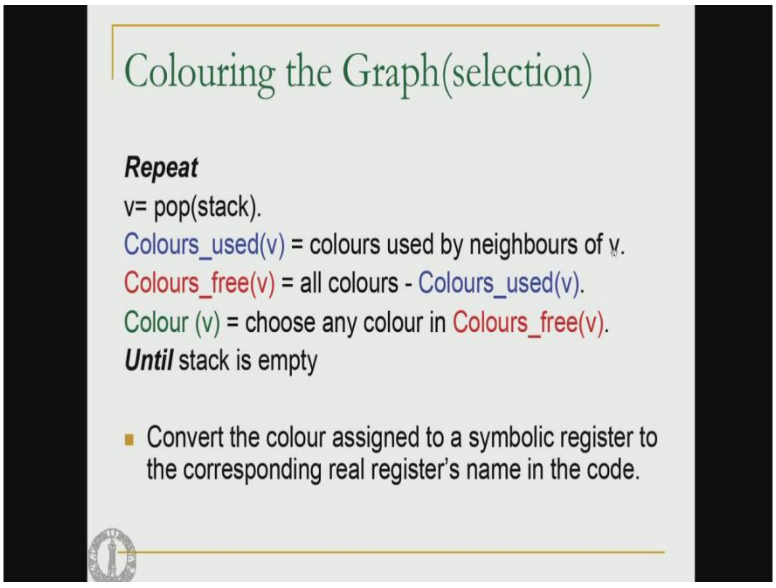


So, def store load use, load use and again def and store. So, this are the small live ranges that I actually mention you know this could be like x equal to something say x equal to 5, that is the definition right and then we have a store instruction. So, we need a small you know, we need a register for doing this computation. And then of course, the register will be freed after the store instruction, but nevertheless this is a small live range, which requires a register for each operation to be completed, the same is true for this as well right and even this, I need a register to load the value of x and then use it here.

So, this load and use requires a small is a small live range require a register, this is another small range which is requires on more register. So, by spilling the variable x in this live range, we have actually introduce many more live ranges, but hope fully the original interferences are gone and the graph is easier to color, in this case of course, the number of interferences is approximately same. So, it does not help, but this is an example to show the affect of spilling.

So, the graph now will have eight of this nodes rather W 1, W 2 you know nodes 7 node and the interferences between this nodes are also slightly different. So, for example and W 6, they have an interference, so we have a W 4 right. So, this is def and use that is one of these and then W 6 is the small live range they interfere W 5 and W 2 also interference. So, this is a W 2 and this def and use is W 5, they also interfere and there is no interference among the other live ranges at all.

(Refer Slide Time 45:35)



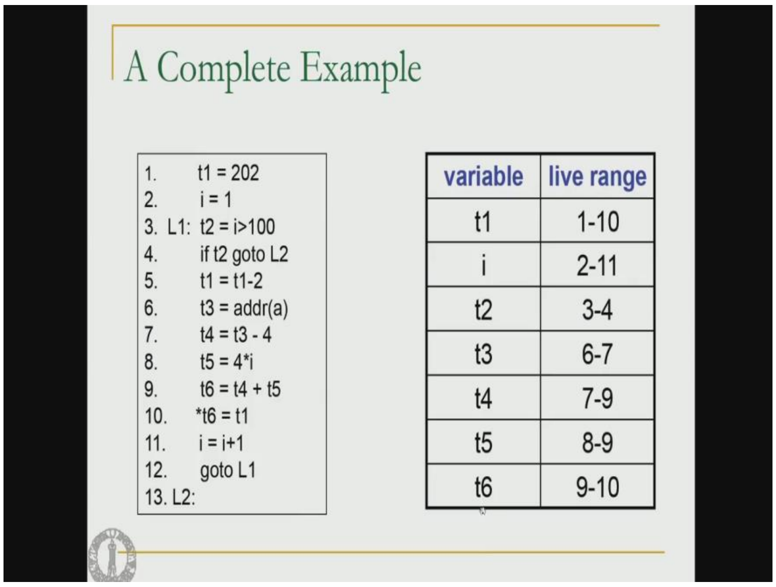
Colouring the Graph(selection)

Repeat
v= pop(stack).
Colours_used(v) = colours used by neighbours of v.
Colours_free(v) = all colours - Colours_used(v).
Colour (v) = choose any colour in Colours_free(v).
Until stack is empty

- Convert the colour assigned to a symbolic register to the corresponding real register's name in the code.

Then what do you mean by coloring the graph or selecting registers, for the program. So, we have a repeat until loop. So, all the node have been placed on the stack, so we pop the stack get a live range, colors used v this is the set is equal to color used by the neighbors of v. So, neighbors of v have been assign some colors, then you know that is the set we are looking at initially none of the colors have been used. So, this would be empty, colors free is all colors minus colors used to begin with all colors will be become colors free as well. Now, choose any color from color free and assign it to node v. So, once we have done this right the you know we go back check whether the stack is empty then the next operation you know next live range is obtained from the stack and the same process is repeated. So, convert the color the assigned to a symbolic register to corresponding real registers name in the code and rewrite the code with the physical registers, so this is the coloring or selection of registers.

(Refer Slide Time 47:03)



A Complete Example

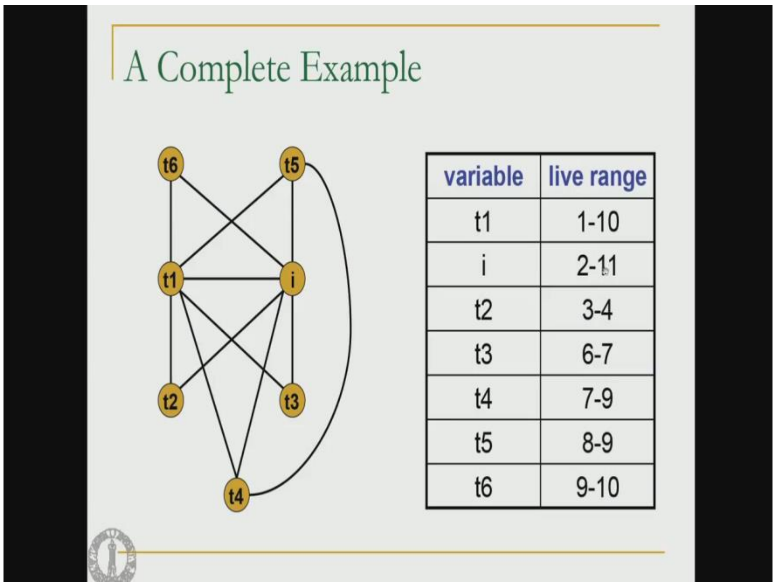
```
1.        t1 = 202
2.        i = 1
3.  L1:  t2 = i>100
4.        if t2 goto L2
5.        t1 = t1-2
6.        t3 = addr(a)
7.        t4 = t3 - 4
8.        t5 = 4*i
9.        t6 = t4 + t5
10.      *t6 = t1
11.       i = i+1
12.       goto L1
13. L2:
```

variable	live range
t1	1-10
i	2-11
t2	3-4
t3	6-7
t4	7-9
t5	8-9
t6	9-10

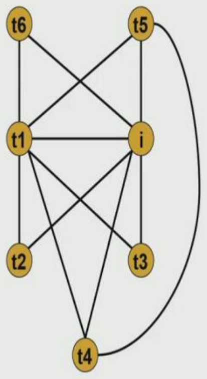
So, let us look at a complete example starting with a program you know and then and going through the various steps of the building the graph coloring it and so on. So, this program has many variable i is a programmer define variable where as the others are all temporaries, but still for coloring operation all the variables are equal. So, we do not differentiate between programmer defined variables and temporaries. So, t one has a live range of 1 to 10. So, here is the first definition of t 1 and the last use of t 1, so that is 10 I has the range from 2 to 11.

So, there is a definition of i here and there is a last use of i here of course, this definition is also i. So, and both these i's are given the same live actually assign the same live range t 2 is between 3 and 4. So, t 2 is defined here and the last use is here t 3 is 6 to 7. So, t 3 is define here and used here t 4 is between 7 and 9 t 4 is defined here and used here last time, t 5 is between 8 and 9. So, t 5 is defined in it and last use is 9, t 6 is between 9 and 10. So, t 6 is defined in 9 and then you know star t 6 is is usage actually t 6 is used it is a pointer. So, star t 6 is the address where t 1 is placed, so this is a usage right, so this is this are the various live ranges, so it is easy to see the interference.

(Refer Slide Time 49:00)



A Complete Example

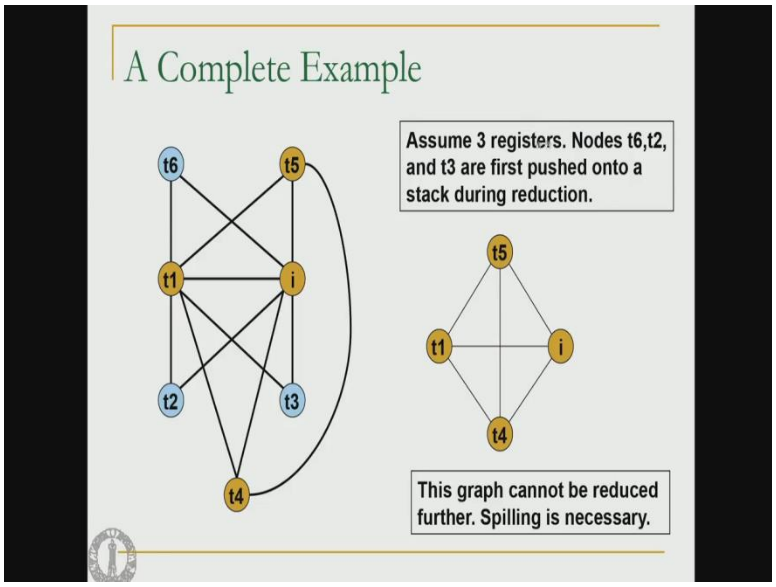


variable	live range
t1	1-10
i	2-11
t2	3-4
t3	6-7
t4	7-9
t5	8-9
t6	9-10

So, t 1 and; obviously, i they interfere and t 1 also interferes with the t 5. So, where ever there is an overlap with the range, so t 2 interferes, then t 3 interferes, t 4 interferes, t 5 interferes and t 6 also interferes. So, t 1 is connected to all of them, so it is connected to all of them. Whereas t 3 is connected to only t 1 and i, so t 1 has a 6 to t 3 has a 6 to 7 as it is live range, so obliviously i and t 1 are only to which interfere with it, similarly t 2 is connected to only t 1 and i, t 6 is connected to only t 1 and i so on and so forth.

So, let us look at it t 4 is 7 to 9 right, so we have a t 1 and i as obvious candidate, so t 1 and i, but it also connecter to t 5 which is between 8 and 9. So, this is 7 and 9, this is 8 and 9, so they overlap, so this is our interference graph for which we need to assign colors.

(Refer Slide Time 50:36)



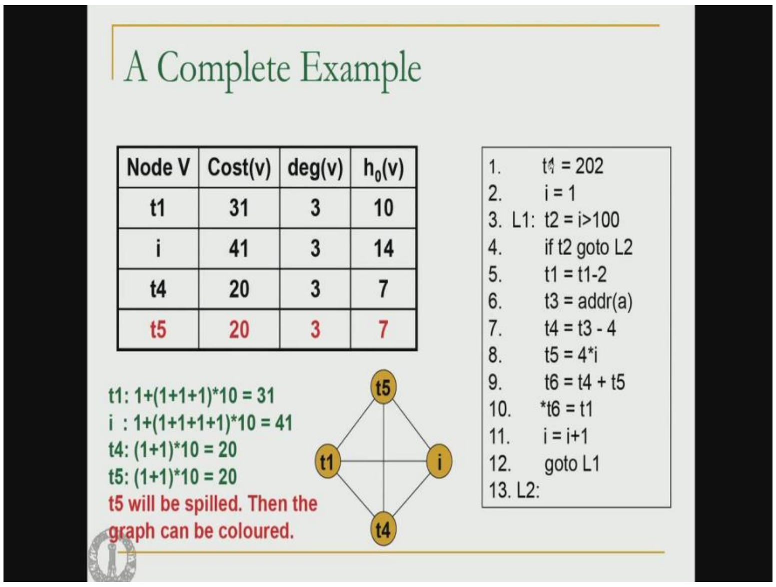
A Complete Example

Assume 3 registers. Nodes t6,t2, and t3 are first pushed onto a stack during reduction.

This graph cannot be reduced further. Spilling is necessary.

So, we start you know let us assume that there are three colors three registers. So, we can you know look at t 6 t 2 and t three this have degree 2, which is less than 3 all have 3 or more. So, we could simply take t 6, t 2 and t 3 and place them on a stack, so once we do that and we remove the nodes along with their edges the graph that is left is this graph. Unfortunately all the nodes in graph have degree 3, you know this has 3, this has 3, this has 3 and this also has 3 ((Refer Time: 51:23)), so the reduction process cannot proceed spilling will be necessary.

(Refer Slide Time 51:27)



A Complete Example

Node V	Cost(v)	deg(v)	$h_0(v)$
t1	31	3	10
i	41	3	14
t4	20	3	7
t5	20	3	7

t1: 1+(1+1+1)*10 = 31
i : 1+(1+1+1+1)*10 = 41
t4: (1+1)*10 = 20
t5: (1+1)*10 = 20
t5 will be spilled. Then the graph can be coloured.

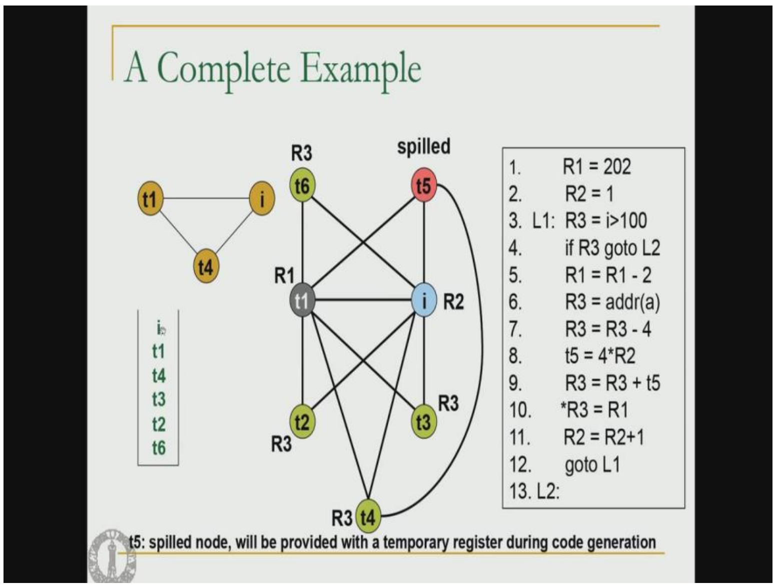
```
1.      t1 = 202
2.      i = 1
3.  L1: t2 = i>100
4.      if t2 goto L2
5.      t1 = t1-2
6.      t3 = addr(a)
7.      t4 = t3 - 4
8.      t5 = 4*i
9.      t6 = t4 + t5
10.     *t6 = t1
11.     i = i+1
12.     goto L1
13. L2:
```

Let us compute the cost of spilling for each of the nodes in this particular graph. So, if you look at node t 1, then you know in the live range for t 1, we have one operation which is outside the loop, so that is cost one and there are three operations associated with t 1 inside the loop. So, there is a t 1 here right, there is a t 1 here and there is another t 1 is here, so these three operations are inside the loop, so the it is multiplied by 10, you know based on our spilling heuristic, so here, here, so multiplied by 10, so the cost total cost is 31.

Similarly for i, we have one operation outside, so that is the this instruction number 2 and then we have 1 you know then 3 right then 3 and 4, 4 operation inside the loop, so that is multiplied by 10, so 40 plus 1 is 41. Then the third one is a t 4, t , has two operations inside the loop. So, t 4 is here and t 4 is here two operations, so that cost is 20. Similarly t 5 also has two operation inside the loops, so that cost is also 20, now the degree of each of these vertices is three.

So, divide and you know round of, so 31 by rather trinket, so this is 10, so this is 14 and then this is 7 this also a 7, so right. So, we are really rounding of in this cases, so this also less then 0.5, then take the floor otherwise take the ceiling value. Out of these, these are the two with least value, so any one of them can be chosen for the spilling operation, so let us choose t 5 for the spilling.

(Refer Slide Time 53:50)

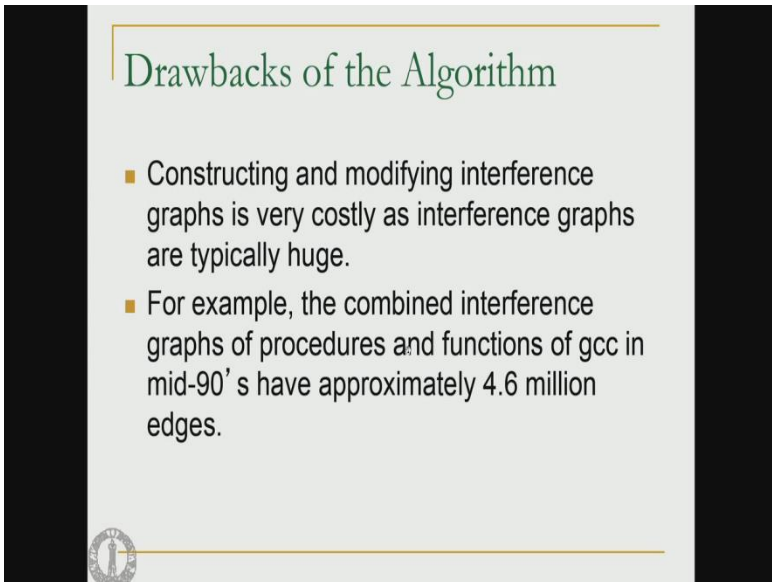

A Complete Example

```
1.      R1 = 202
2.      R2 = 1
3. L1:  R3 = i>100
4.      if R3 goto L2
5.      R1 = R1 - 2
6.      R3 = addr(a)
7.      R3 = R3 - 4
8.      t5 = 4*R2
9.      R3 = R3 + t5
10.     *R3 = R1
11.     R2 = R2+1
12.     goto L1
13. L2:
```

t5: spilled node, will be provided with a temporary register during code generation

Then the, we remove that you know, we get actually this small graph which can be easily colored using three colors. So, now the coloring stack has t 6, t 2, t 3, t 4, t 1 I, so all the nodes are placed here, we can color them in the reverse order note that t 5 is not present here it is spilled, so it will be permanently in memory. And it will be let us assume that temporary register can be provided to this particular node are variable during the code generation process. So, the global register allocator need not provide any register for this particular variable t 5, so otherwise the coloring part is quite straight forward, so we get the coloring as shown in this picture.

(Refer Slide Time 54:40)

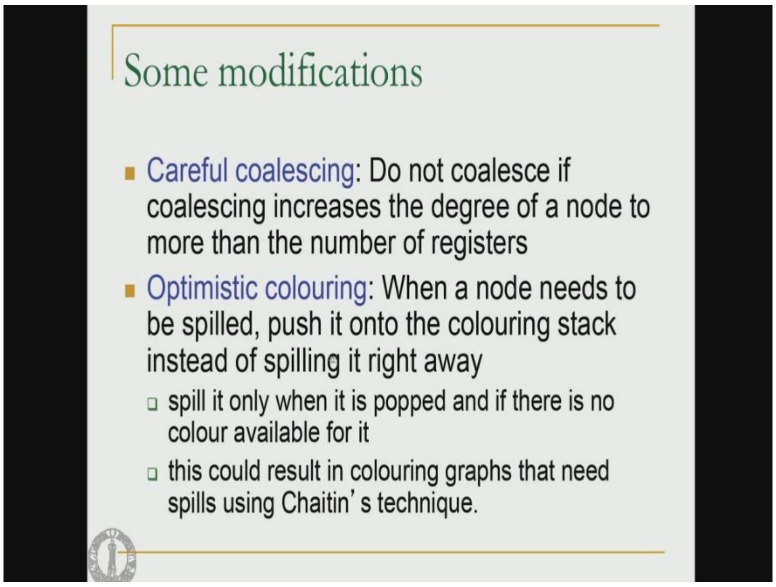


Drawbacks of the Algorithm

- Constructing and modifying interference graphs is very costly as interference graphs are typically huge.
- For example, the combined interference graphs of procedures and functions of gcc in mid-90's have approximately 4.6 million edges.

So, after this the let us look at the you know of course, then we rewrite the program with the registers t 5, written as it is this will be given a register at the time of machine code generation, these are still not machine operations, but we have assigned physical registers. Drawbacks of the algorithm constructing and modifying the interference graph is very expensive because the graphs can become very large for example, the combined interference graphs of procedures and functions in g c c in mid 90's; the older version of g c c approximately has 4.6 million edges, so these are extremely large, so graph coloring takes a large amount of time.

(Refer Slide Time 55:27)

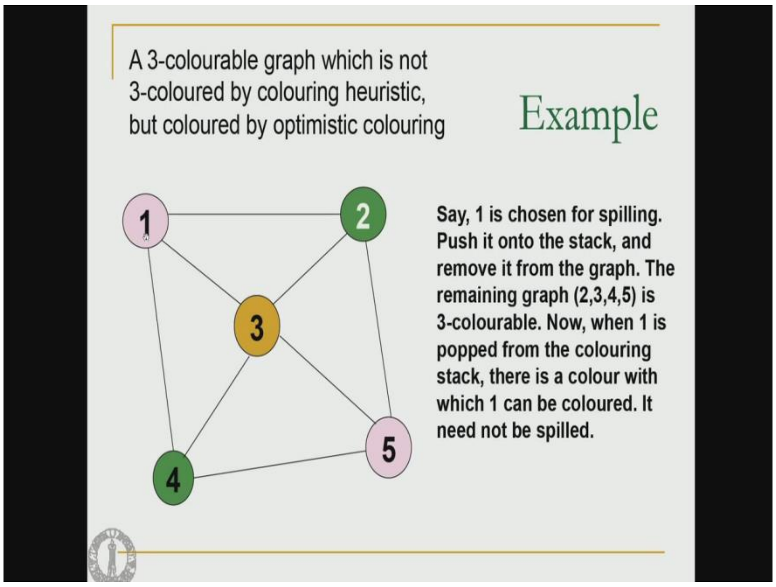


Some modifications

- Careful coalescing: Do not coalesce if coalescing increases the degree of a node to more than the number of registers
- Optimistic colouring: When a node needs to be spilled, push it onto the colouring stack instead of spilling it right away
  - spill it only when it is popped and if there is no colour available for it
  - this could result in colouring graphs that need spills using Chaitin's technique.

So, some modifications are possible for example, we do not have to do copies subsumptions, if the degree of the node becomes more than the number of registers. And we do optimistic coloring, that is when a node needs to be spilled we do not you know we push it onto the coloring stack instead of spilling it right away, so and then try to give it a color, when we pop the node.

(Refer Slide Time 55:53)



Example

A 3-colourable graph which is not 3-coloured by colouring heuristic, but coloured by optimistic colouring

Say, 1 is chosen for spilling. Push it onto the stack, and remove it from the graph. The remaining graph (2,3,4,5) is 3-colourable. Now, when 1 is popped from the colouring stack, there is a colour with which 1 can be coloured. It need not be spilled.

So, let us look at an a example, so here say node one is chosen for spilling we just put it on to the stack and remove the corresponding node in the edges. The rest of the graph now becomes colorable with three colors, because the possible to do it you know spill two and then sorry reduce two reduce five and so on and so forth. So, let us say we finished the coloring for this part of the graph, now when we come here we find that you know there is color free, which can be assigned to node one.

So, this is again a heuristic which is called optimistic coloring and it may not work in all cases. Suppose we had assigned two different colors to these nodes then you know in the coloring process, then this would now have had it color free; so this completes our decisions on register allocation.

Thank you.