Compiler Design

Prof. Y. N. Srikant

Department of Computer Science and Automation

Indian Institute of Science, Bangalore

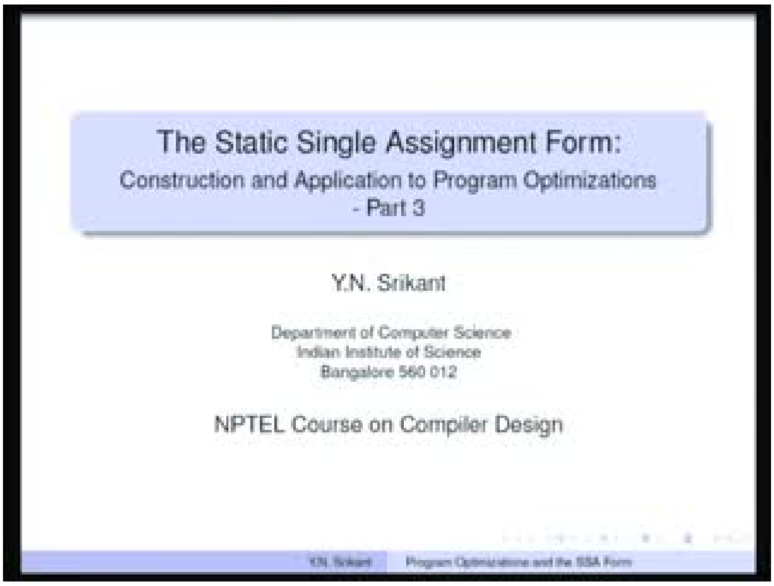
Module No. # 13

Lecture No. # 23

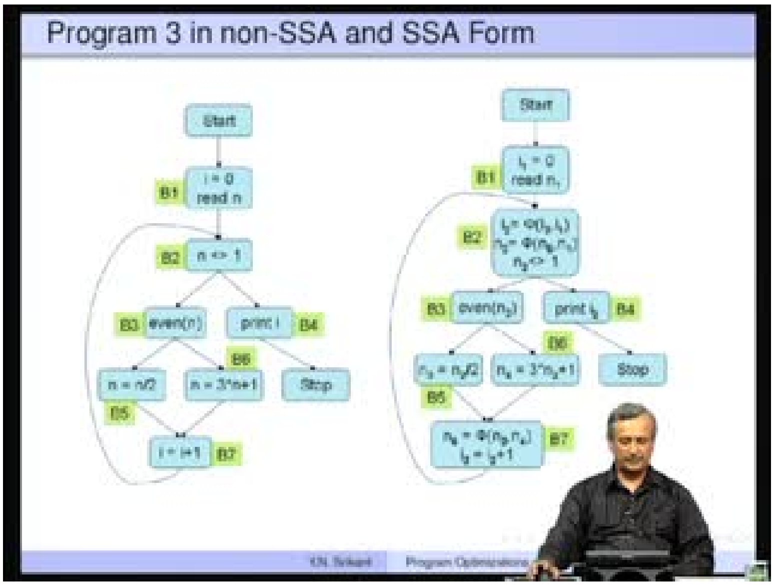
The Static Single Assignment Form: Construction and Application to Program Optimizations Part 3

Welcome to part 3 of the lecture on the Static Single Assignment form. To recap a bit, here is an example of the SSA form and the non-SSA form as well.

(Refer Slide Time: 00:18)



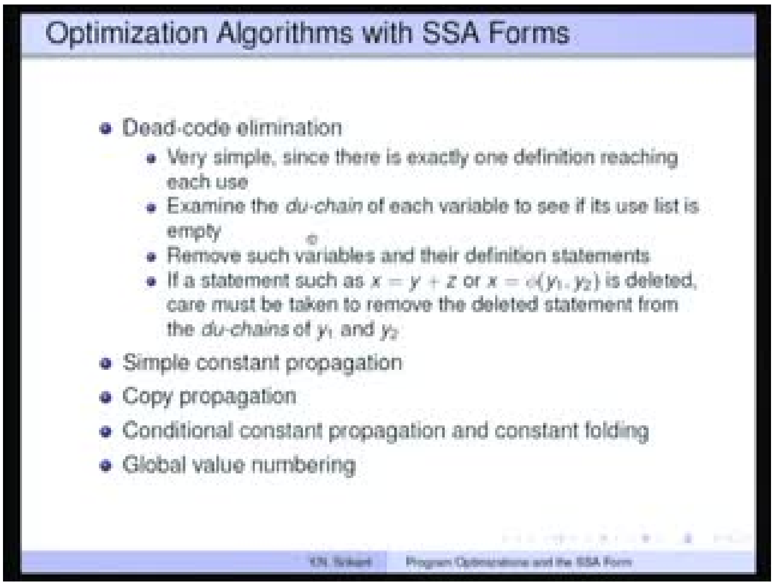
(Refer Slide Time: 00:31)



The non-SSA form is just a flow graph; whereas, in the SSA form some of the join nodes will have phi functions for the incoming parameters. For example, i in the original flow graph flows from B1 into B2 and also from B7 into B2. So in B2, the SSA form we have a phi function which really takes two parameters I_3 and I_1; I_3 corresponds to the first parameter coming from B7 and I_1 corresponds to the second parameter coming from B1. Similarly, the variable n is read here, so that is like a definition and later, we have definitions of n in B5 and B6 as well.

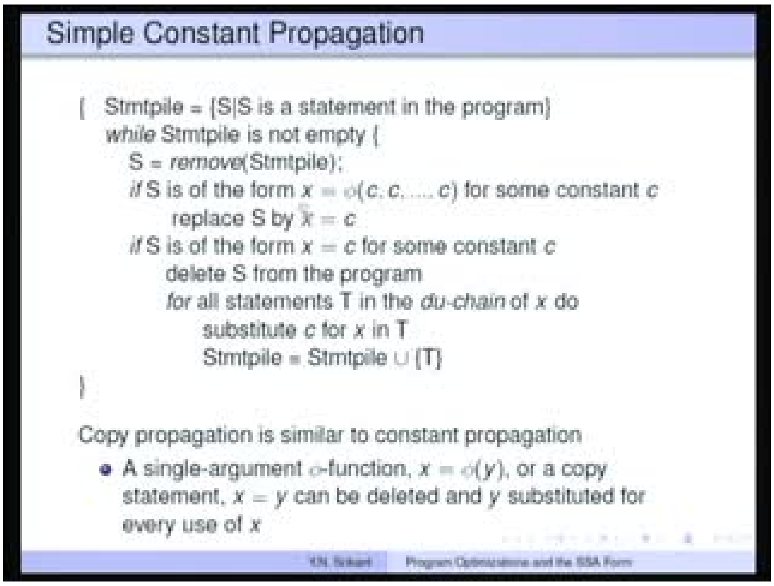
So in B7, we have a phi function for n which has two parameters n_3 and n_4, corresponding to these two predecessors. Then, in B2 we have another phi function for n, which takes care of n coming from B1 and this n_6 which is coming from B7? So that is how phi functions are. So, we also saw how to insert phi functions and how to rename the parameters of the phi function etcetera.

(Refer Slide Time: 01:53)



So, we were looking at the optimizations with SSA forms. There are many optimizations which are very fruitful on such forms. For example dead-code elimination, simple constant propagation, copy propagation, conditional constant propagation, constant folding, global value numbering these are all optimizations which can be done very effectively on the SSA form.

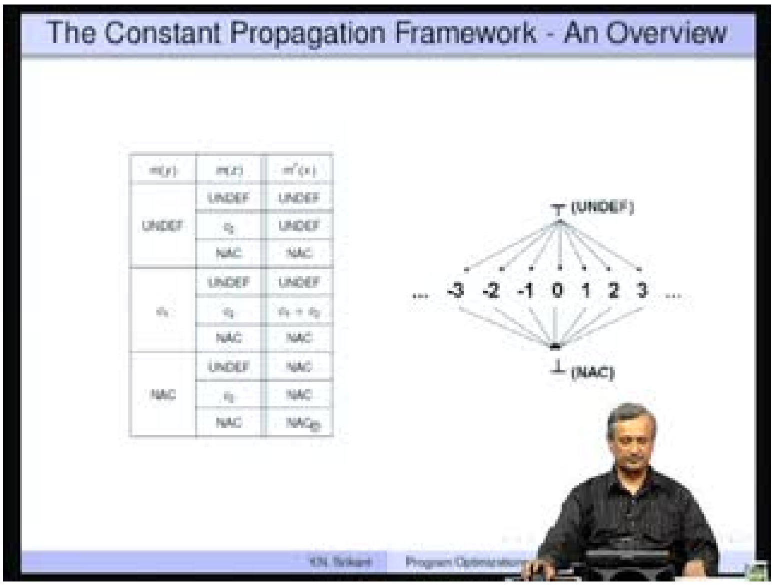
(Refer Slide Time: 02:19)



Simple Constant Propagation is really simple. Take all the statements put them in a statement pile and then take one at a time from the statement pile. If they are trivial phi functions with all parameters being equal and constant, then such statements can be replaced by x equal to c. Otherwise, if there is a statement x equal to c then we take the du-chain of that particular x. Then for all uses of x, we can really substitute c and then the new statement is also added to the statement pile, so that we can propagate constants further.

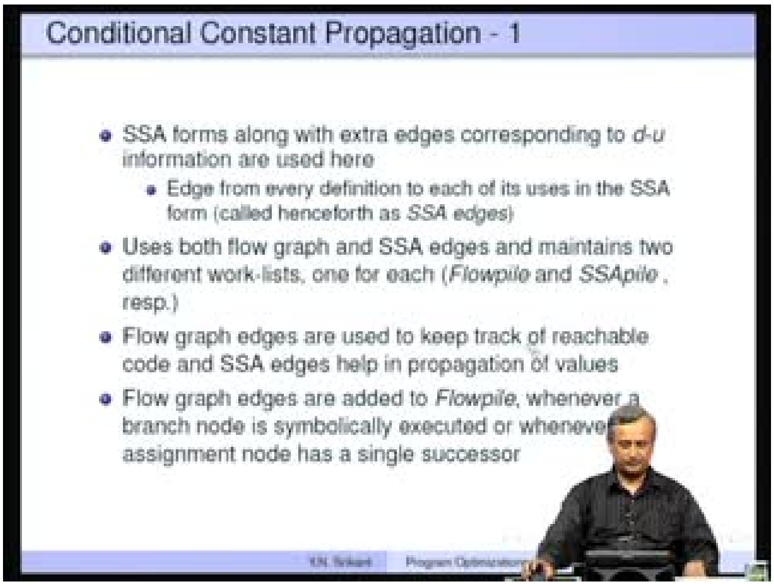
Copy propagation is very simple, if there is a single argument function x equal to phi y or a copy statement x equal to y, these can be deleted. We can substitute y for every use of x. This is possible because every use is read exactly by one definition in the SSA form.

(Refer Slide Time: 03:24)



The Conditional Constant Propagation is special. Let us recapitulate the transfer function of the Conditional Constant Propagation Frame work. So this is monotonic, but it is not distributive and here is the lattice of the constants. So, all the constants are in comparable there is an undefined value at the top and not a constant value at the bottom.

(Refer Slide Time: 03:54)

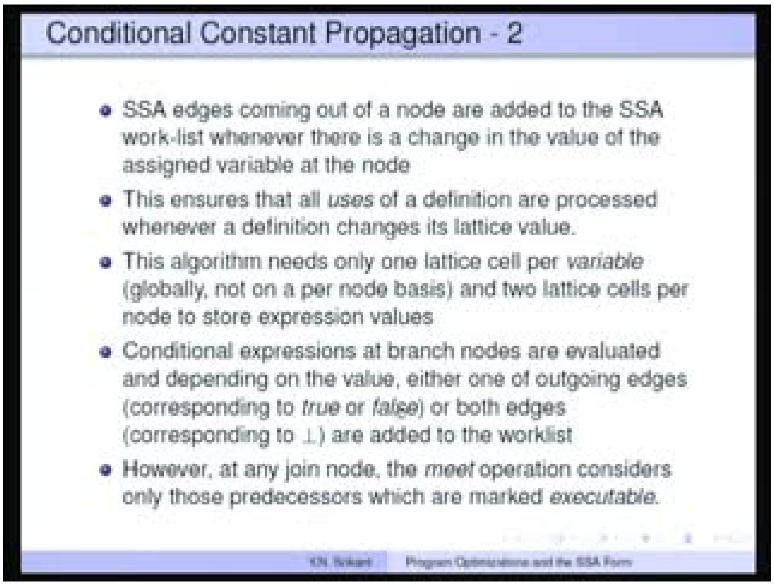


So, how does conditional constant propagation on SSA forms work? So, SSA forms along with extra edges corresponding to the definition use information are used here. So edge from every definition to each of it uses in the SSA form hence, we call these as SSA edges. So, they are also available in the graph. It uses both flow graph edges and the SSA edges and maintains two different work lists. This is a work list based approach exactly like simple constant propagation. So, we have a Flowpile and an SSApile.

So, it is important that flow graph edges are used to keep track of reachable code. So, whatever code cannot be reached will have all the edges incoming into it marked as non-executable and therefore, we can never reach that node.

SSA edges are helpful in the propagation of values, whenever there is a change of value in a node; the SSA edge is used to activate that particular node and put it into a SSApile. So, the flow graph edges are added to the flowpile, whenever a branch node is symbolically executed or whenever an assignment node has a single successor.

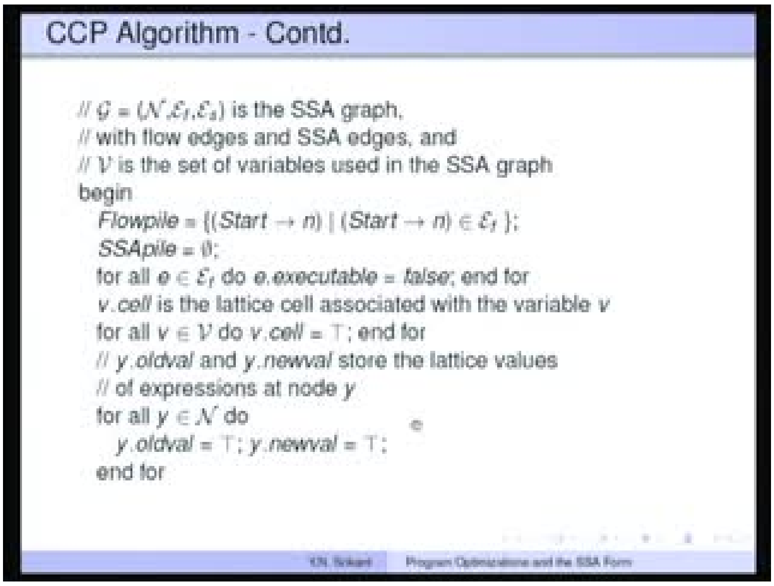
(Refer Slide Time: 05:12)



Whereas SSA edges coming out of a node are added to the SSA work list, whenever there is a change in the value of the assigned variable at the node. So the algorithm really needs only one lattice cell per variable not on a per node basis and two lattice cell per node to store expression values, so not too much of extra space.

The Conditional expressions at branch nodes are evaluated and depending on the value, either one of the outgoing edges corresponding to true or false or both edges corresponding to not a constant are added to the work list. So, if you are able to evaluate it to either true or false, only one edge is added otherwise, both edges have to be added. At any join node, the meet operation considers only those predecessors which are marked executable. So that is an extra point to be noted here.

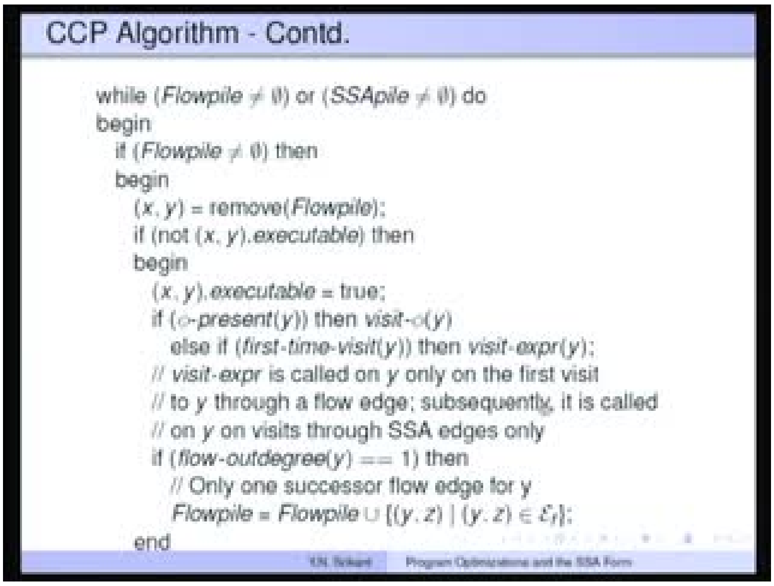
(Refer Slide Time: 06:05)



So let us look at the algorithm, we saw an example. We will learn through that example again a little later. So, G is the SSA graph N E_f and E_s; so E_f is the flow graph edges, E_s are the SSA edges. Now, V is the set of variables used in the SSA graph that is the program itself. So, we initialize the flowpile with the first edge start to n, so all the edges which go out of the start node are added to the flowpile, whereas the SSA pile kept empty.

For all the edges in the E_f set that is, all the flow graph edges; they are made as e.executable equal to false, so that nothing is marked as true to begin with. Then v.cell is the cell associated with the variable v, so we must initialize that also it is initialized to top, that is the undefined value. Then y.oldval and y.newval store the lattice values of expressions at a particular node y. So these need to be initialized as well. So both y.oldval and y.newval are initialized to top the undefined value.

(Refer Slide Time: 07:27)

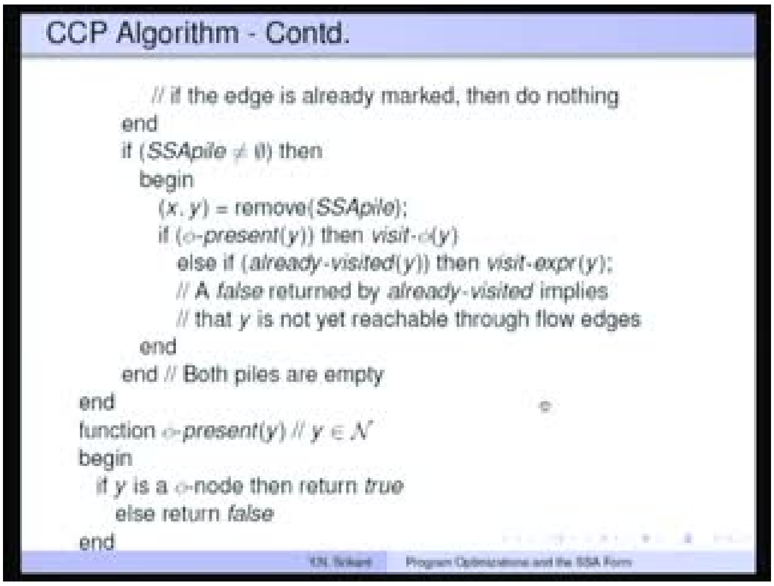


Then there is a big loop which goes on until both flowpile and SSA pile become empty. So the first one is, if flowpile is not equal to phi that is flowpile is not empty, what we do? We remove an edge from the flowpile. If the edge is not marked as executable that means, we are now entering through an edge which is not yet seen before. Now mark that particular edges executable that is executable equal to true for that edge.

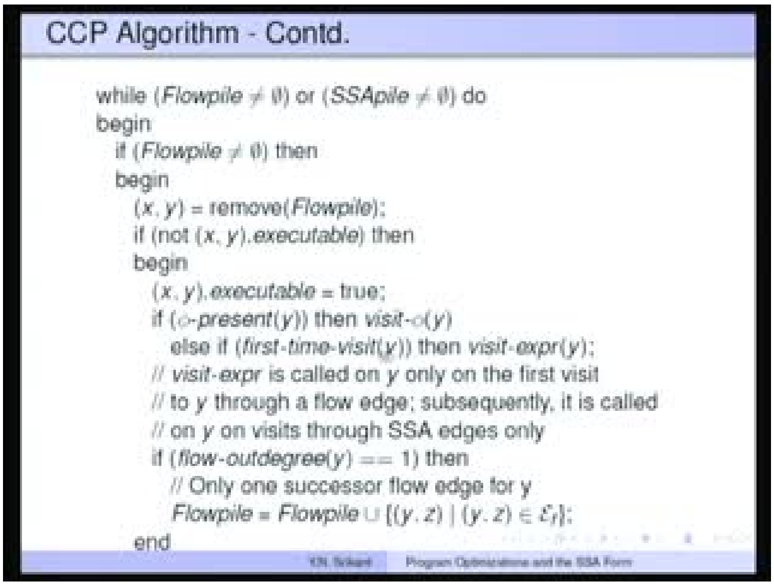
Now, we check couple of things; is it a phi node so if phi present y, so this is true if it is a phi node then we call the function visit phi y, we are going to see some details little later. So if it is not a phi node, then it is an ordinary expression node. So, first time visit y is checked and is saying are we visiting it for the first or are we visiting it second third time etcetera.

If it is first time then visit expression; the point is visit expression is called on y only on the first visit y through a flow edge. Subsequently it is called on y on visits through SSA edges only. So, if first time visit y is false then we do not visit that expression right now, we are going to visit it later, if any values of the parameters in the expression change.So, if flow-outdegree is 1; that means, we have exactly 1 successor for the node after doing some processing, then we just add that to the flowpile.

(Refer Slide Time: 09:19)

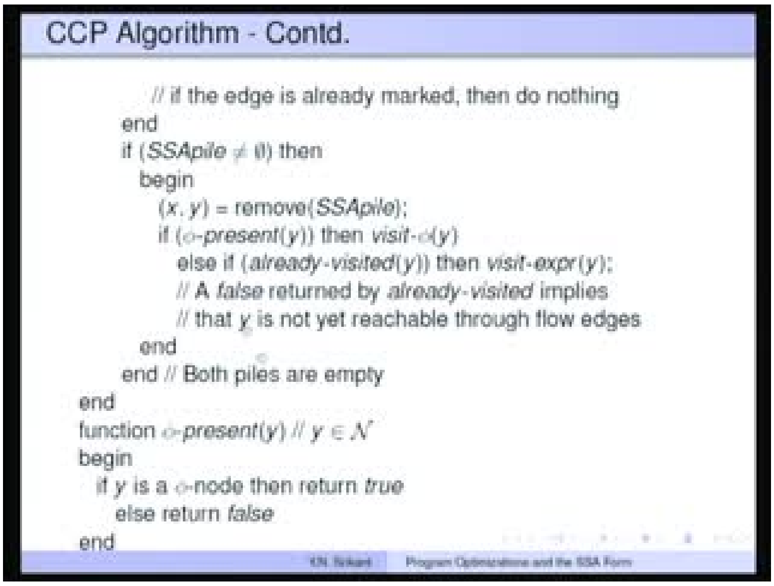


(Refer Slide Time: 09:24)



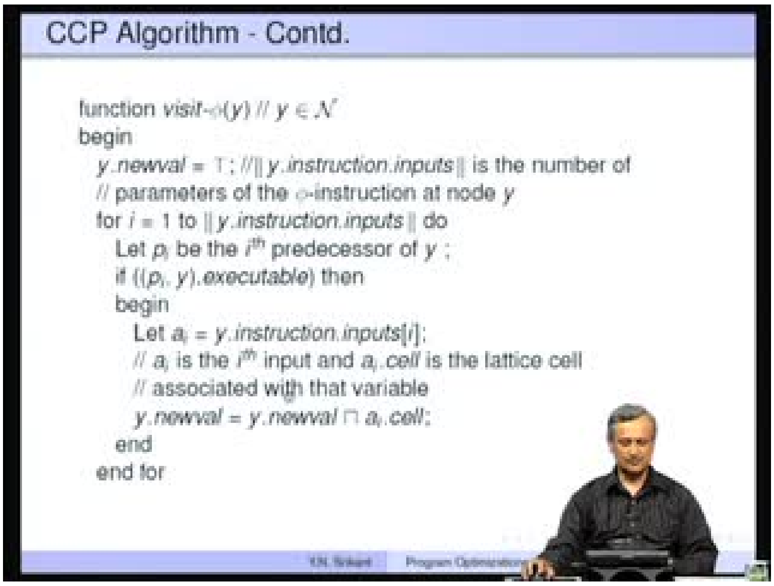
Then, if the edge has already been marked that is this part so is it first time y etcetera is where we are.

(Refer Slide Time: 09:32)



Now we look at this SSApile. Alternately, we are going to look at the SSApile and the flowpile. So, remove an edge from the SSApile again check whether it is a phi node then call visit phi y; if it is already visited then visit expression again because, we are now coming through the SSA, so nothing wrong with that. The point is, if it is not visited already - the node y - is not visited already that means, it is not yet reachable through any flow edges. Therefore, we are not going to visit it at all. Unless a node is already visited, we are not going to visit it when we visit through a necessary edge because it may not be reachable at all. So this loop goes on until both piles are empty. Now let us look at the details.

(Refer Slide Time: 10:36)

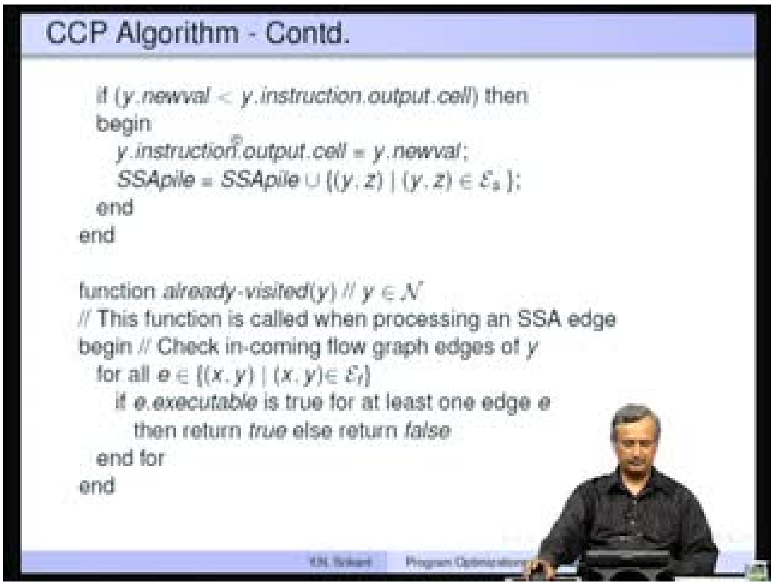


So phi present y is simple; if y is a phi node then returns true otherwise return false. So what does visit phi y do? So y is a node, take y.newval as undefined value. Let y.instruction.inputs be the number of parameters of the phi instruction at the node y. Now, we are going to take each one of the inputs and then check whether the edge corresponding to that input is executable.

Why we want to actually take the meet of only those parameters, whose corresponding edges are marked as executable otherwise we do not want to touch them. So, let p_i be the ith predecessor of y, i running from 1 to y.instruction.inputs. If p_i.y is executable then take the corresponding input y.instruction.inputs I, so the ith input is taken in the i.

Then take y.newval and meet it with a_i.cell, so you get the new y.newval - updated value. This is done for all the inputs and we leave out those inputs which come through edges not marked as executable, we do not want to touch them. So it is easy to see that if a node has not been visited at all not even once, then nothing gets done at the phi node, this loop will run many times but in an empty fashion doing nothing.

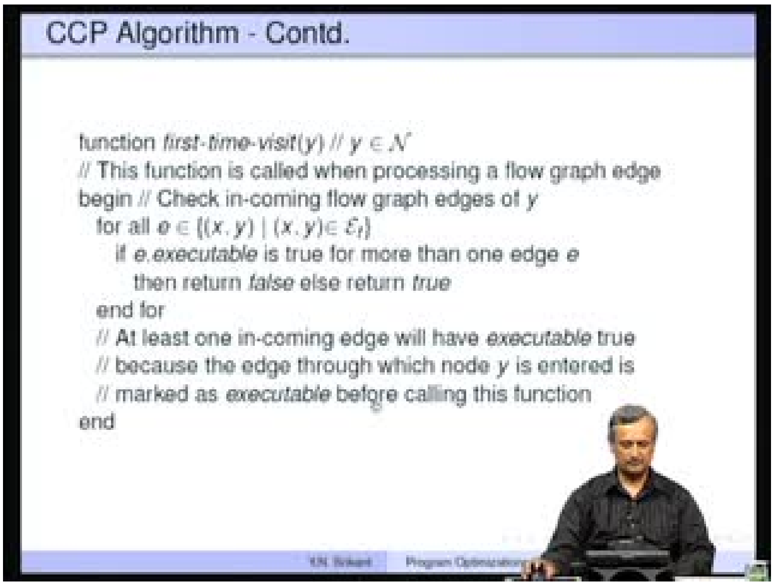
(Refer Slide Time: 12:13)



Now, if the new value is less than y.instruction.output.cell the old value that means, the value has changed. Remember, we go from undefined to constant to not a constant that is downwards in the lattice cell that is y is less than. If y.instruction.cell equal to y.newval, take the new value as the output value and add the outgoing SSA edges to the SSApile. So, y comma z, where y comma z is in e_s, so all the edges going out of phi are added to the SSA pile because the value has changed. If the value has not changed, there is nothing to do, you do not add anymore edges to the SSA pile.

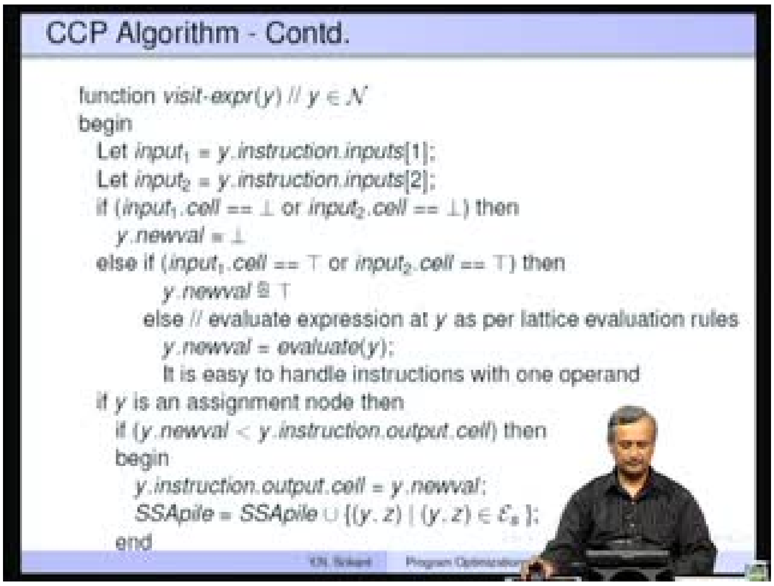
So already visited it is simple, it simply checks the incoming edges of y. If one of them is marked as executable then it is already visited otherwise it is not visited at all. Check incoming edges of y for all e in x y such that x y in e_f, so y is in our node so x y is the incoming edge. If e.executable is true for at least one edge e, then return true otherwise return false. So, it is a fairly straight forward function.

(Refer Slide Time: 13:34)



What does first time visit y do? Exactly one of the edges must be marked as executable and other should not be. So, if e.executable is true for more than one edge then return false, otherwise return true. At least one incoming edge will have executable true, because the edge through which the node is entered is marked as executable before calling this function. The first time you come to y, it will be one of the edges - incoming edges - will be true, so you will get something as true. But if more than one edges is true then we are entering for the second time, so this will be returned as false.

(Refer Slide Time: 13:34)

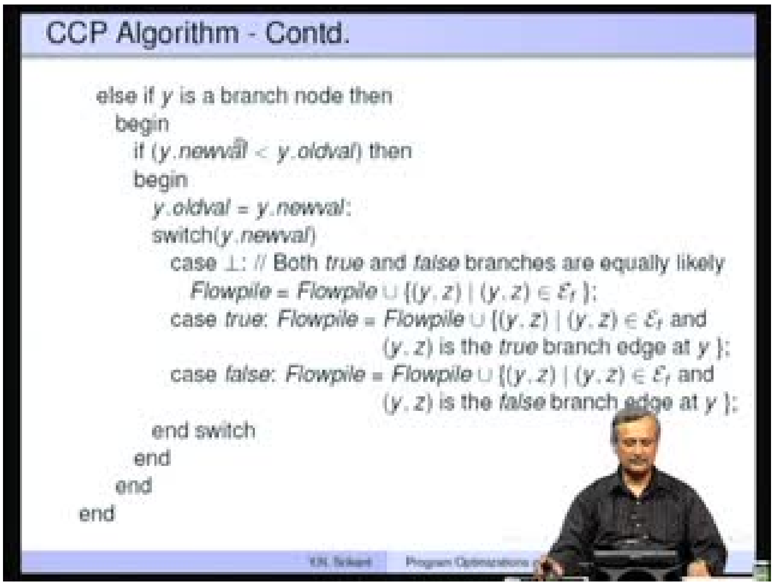


What does visit expression do? It really processes the expression whether it is an assignment statement or a branch condition. Take the 2 inputs; input one equal to y.instruction.inputs 1 and input two, as y.instruction.inputs 2. If I recall the transfer function, if one of the inputs in x plus y either x or y is not a constant, then the output is also not a constant. So, if input 1.cell equal to n a c or input 2.cell equal to n a c then y.newval is n a c not a constant.

We have covered the n a c part. If one of them is undefined input 1 or input 2 is undefined, then y.newval is undefined. If this is also not true then both are really neither top or bottom values in the lattice, so we can do some evaluation. Evaluate the expression as per the lattice evaluation rules, evaluate y, so the expression is evaluated. Of course, it is easy to modify this to handle instructions with one operand. So copy instructions are easy to handle.

If y is an assignment node then if y.newval is less than y.instruction.output.cell. So the newval that we got here by evaluating the expression is less than the old value. That means, the value has changed. So remember, we always go down in the lattice. Take the new value as y.newval, store it in the instruction.output.cell of y and add all the edges going out of y to the SSApile so for it is as we did before.

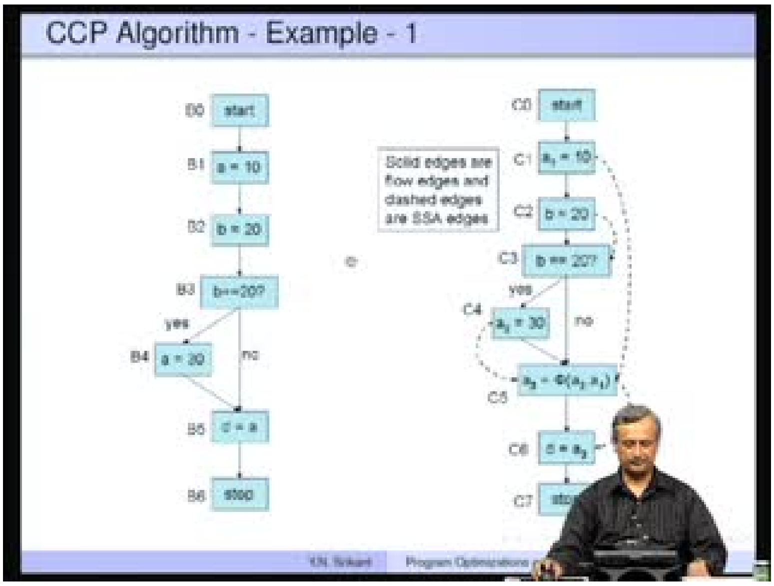
(Refer Slide Time: 16:31)



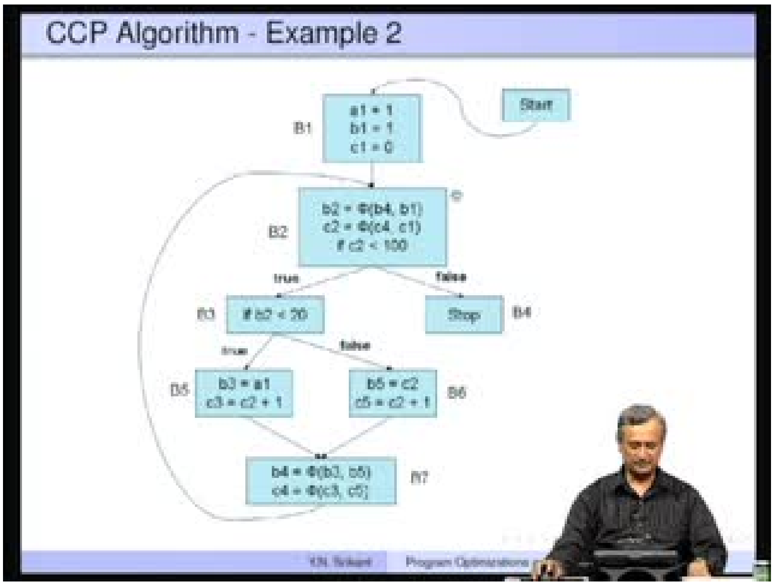
What if it is a branch node if the value has changed, y.newval is less than oldval then y.oldval is equal to y.newval. Now check whether, what is the value of y.newval, if it is n a c not a constant that means, both a true and false branches are equally likely. We add both branches to the flowpile. If it is evaluated to true then we add only the true branch edge to the flowpile; in the case of false, we add the false branch edge to the flowpile.

This is where, if the condition has become a constant and as evaluated to either true or a false. Then, we can avoid some of the nodes which can be entered through either the true or false edges. So, we actually remove those as read code finally.

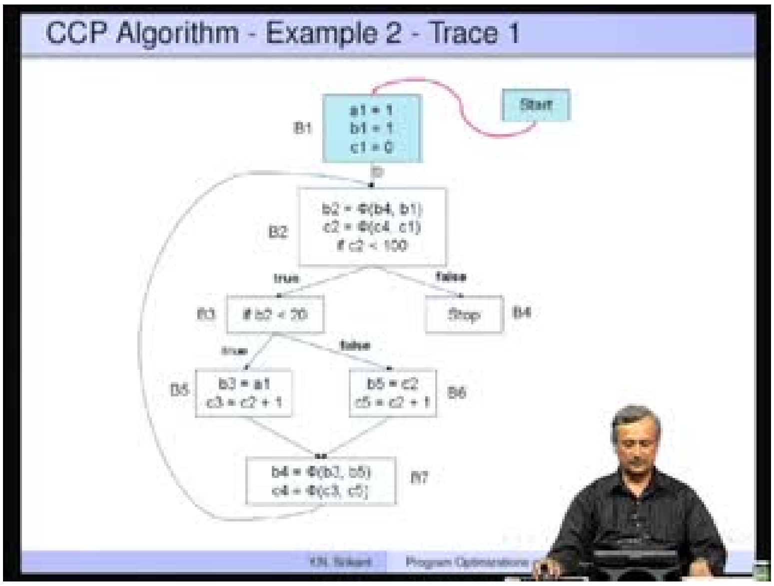
(Refer Slide Time: 17:31)



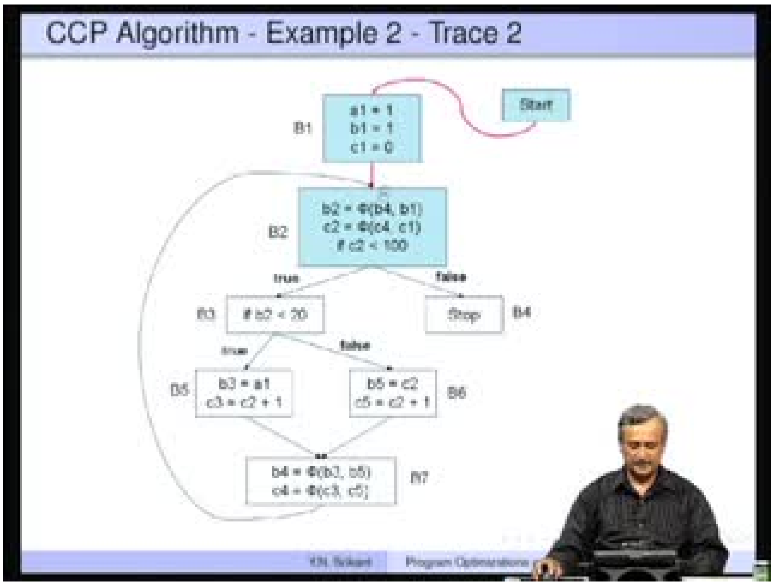
(Refer Slide Time: 17:41)



(Refer Slide Time: 17:45)

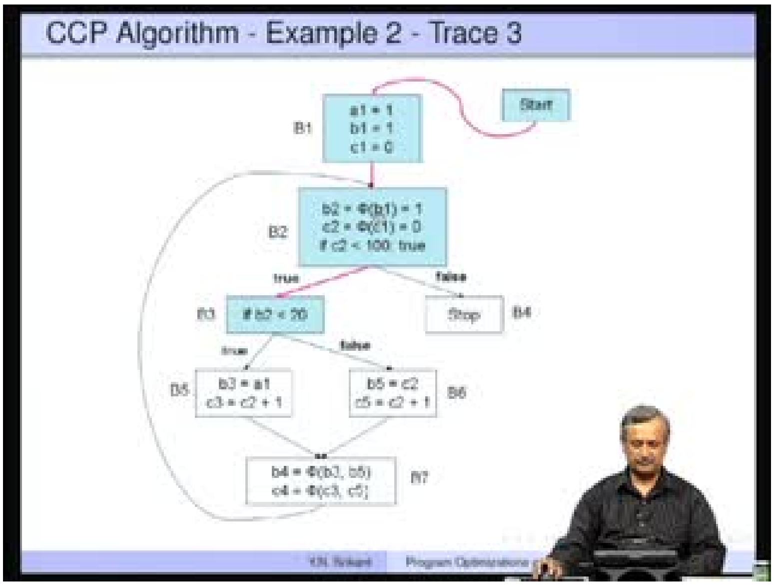


(Refer Slide Time: 17:57)

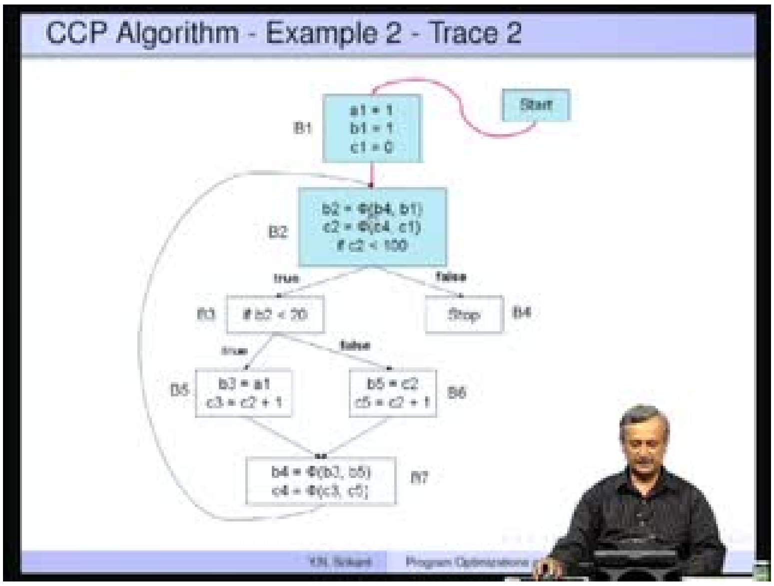


So here is a simple example, let us run through the more difficult example, because this is a2 simple example, which we ran through last time. So this example, we start with the first node B1 after start, a1, b1, c1 are all initialized. This particular edge actually is the only one coming of B1, we add this edge to the flowpile and that makes this particular node to be interpreted next.

(Refer Slide Time: 18:13)

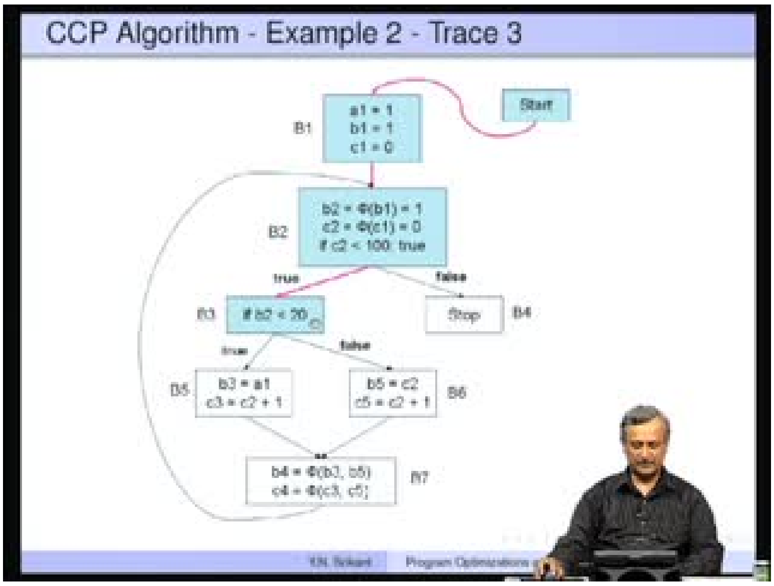


(Refer Slide Time: 18:15)



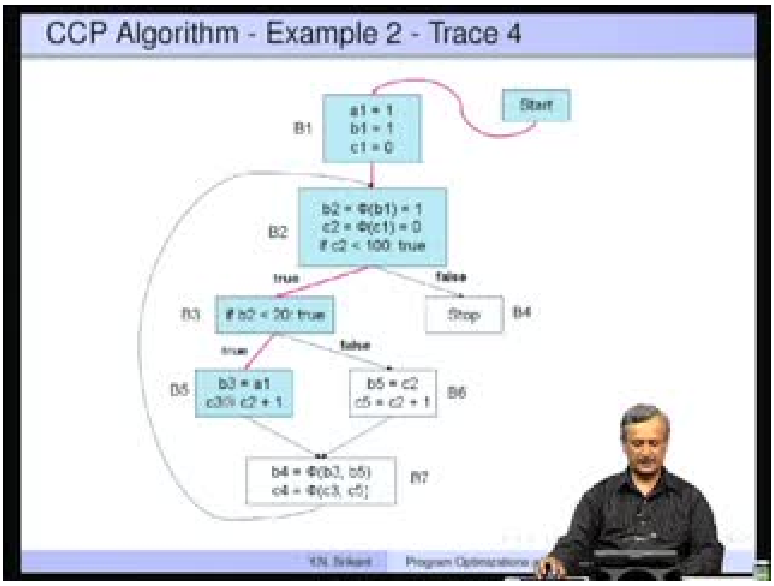
So what happens here, see that b4, b1; b1 is the only parameters which is defined. This part is not yet to marked as executable, so nothing is coming out of this.

(Refer Slide Time: 18:26)

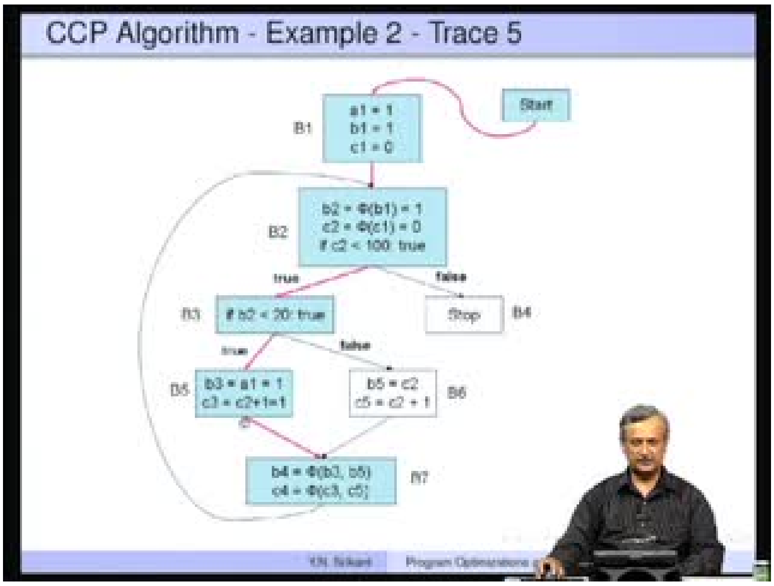


So only b1; b1 is 1, so b2 becomes 1 phi b1 of is just b1, which is 1. Similarly, phi c1 is c1 is 0, so c2 becomes 0. Therefore, c2 less than 100 is obviously true, 0 less than 100 and that makes this particular edge to be added to the flowpile and this is not yet added; it does not mean it will never be added it may be added little later.

(Refer Slide Time: 18:58)

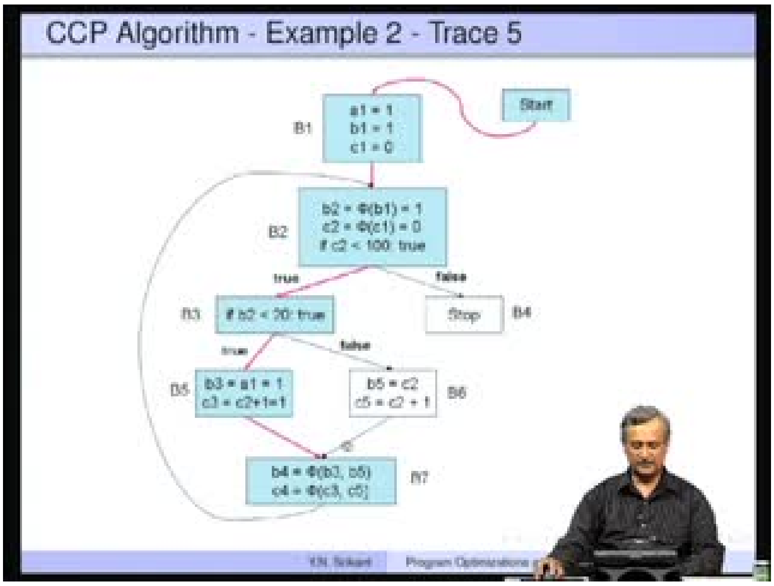


(Refer Slide Time: 18:26)

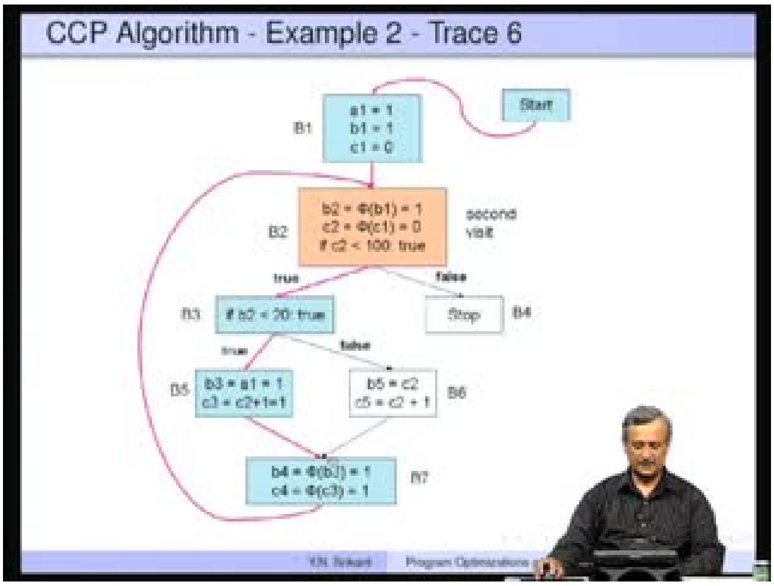


Now, we evaluated b2 less than 20 that is also true, so again the true part is true edge is taken and become to B5. Now in node B5 b3 evaluates to 1 and c3 evaluates to 1 both are constants as so far. This particular edge which is the only successor is taken we come to B7.

(Refer Slide Time: 19:23)

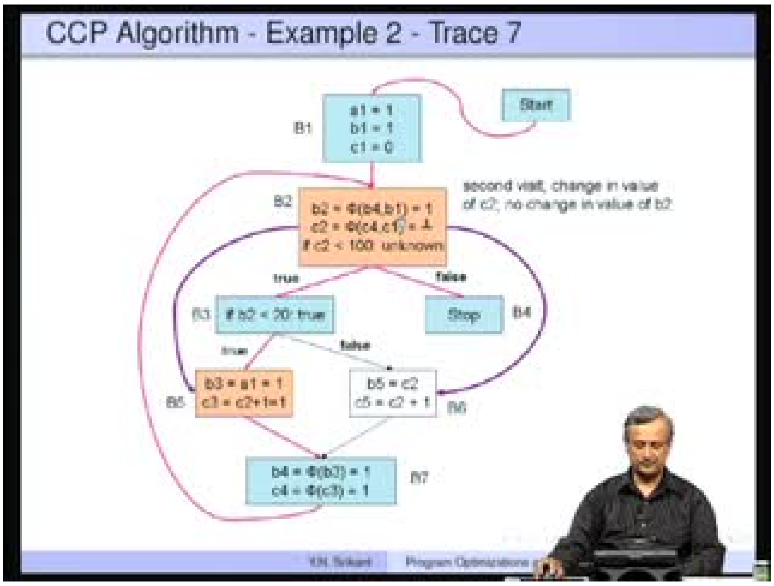


(Refer Slide Time: 19:30)



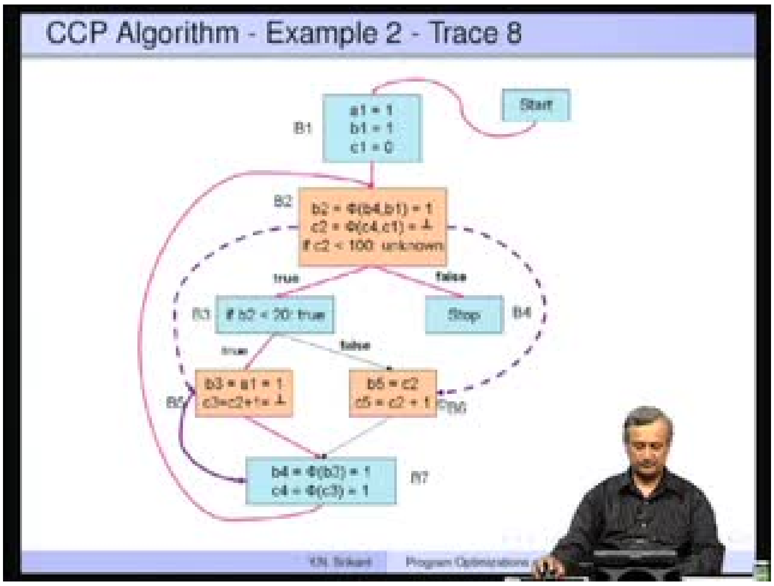
Once we evaluate B7, again this parameter is not yet available, only B3 is available and similarly, only c3 is available when evaluate b4 it becomes 1 and c4 becomes 1. This edge is added to the flowpile that means, we come to B2 once more a second visit.

(Refer Slide Time: 19:41)

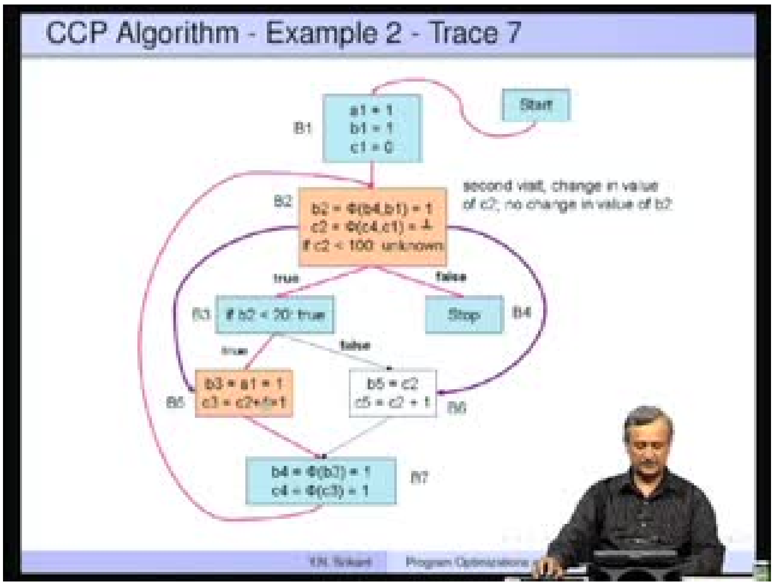


So second visit change in the value of c2 to not a constant, but there is no change in the value of b2. Even with this available b4 is 1, b1 is 1, so this b2 becomes 1 but, in the case of c2; c4 is 1 but c1 is 0, so phi function is not a constant.

(Refer Slide Time: 20:12)

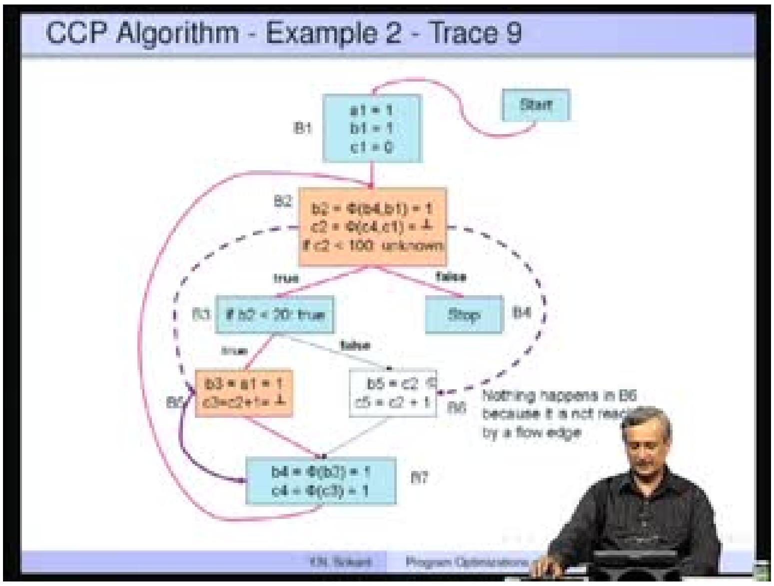


(Refer Slide Time: 20:21)

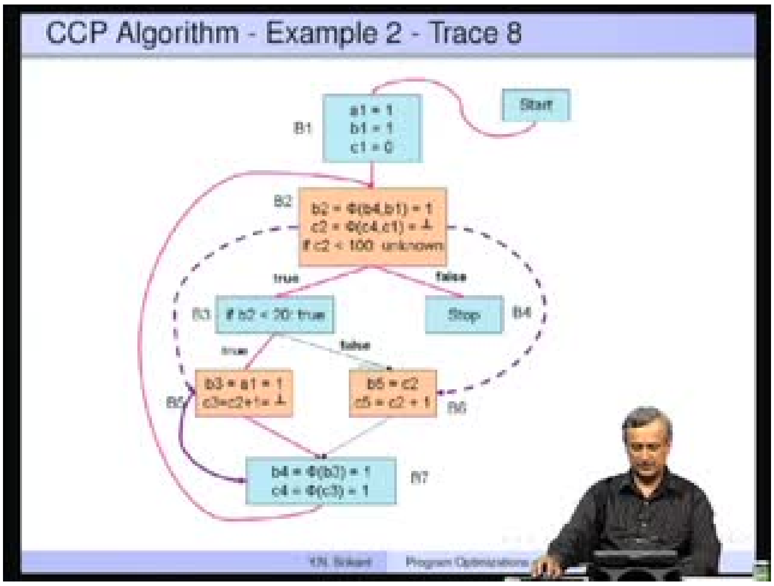


Now, this ((s h j)) edge actually activates the node B5 and B6 both of them. Let us take this particular node, b3 now evaluates to 1 and c3 becomes not a constant because c2 plus 1 is the value; c2 is less than 100 which is not known. So at this point, we do not know whether c2 is 100 or not it has been evaluated to bottom. Actually, we add both edges to the flowpile, so this edge and this edge both are added.

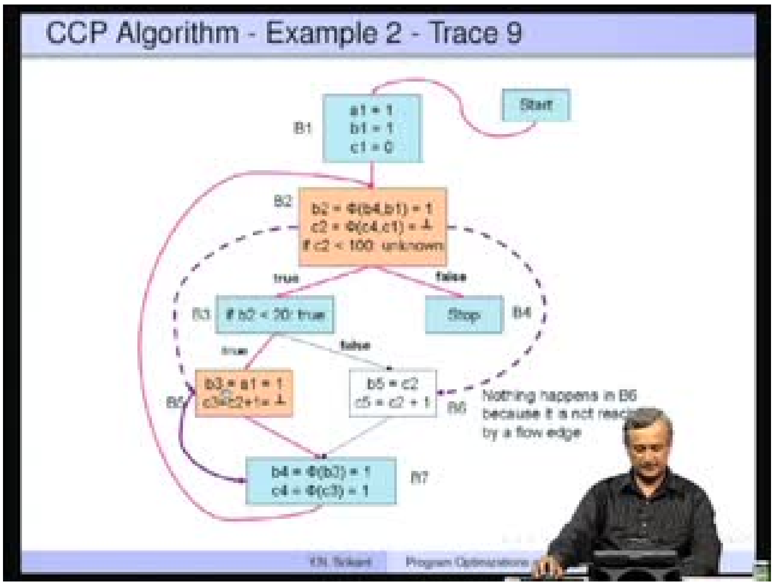
(Refer Slide Time: 20:50)



(Refer Slide Time: 20:55)

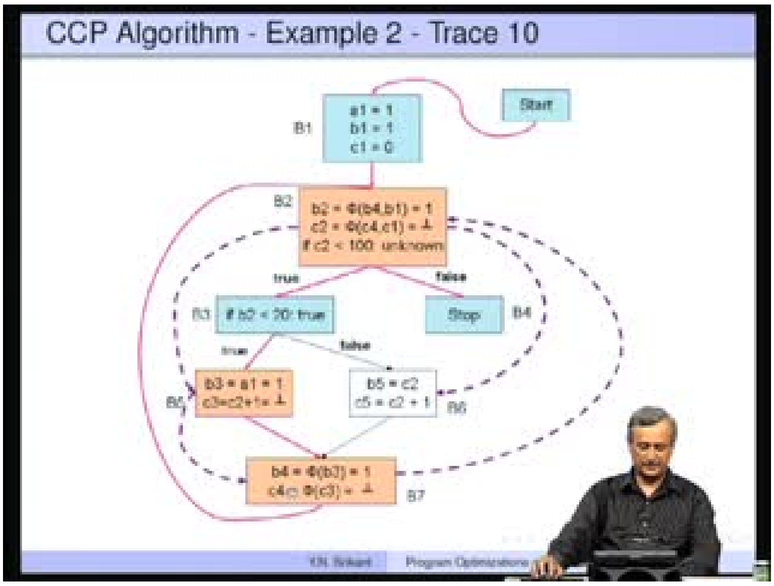


(Refer Slide Time: 21:09)



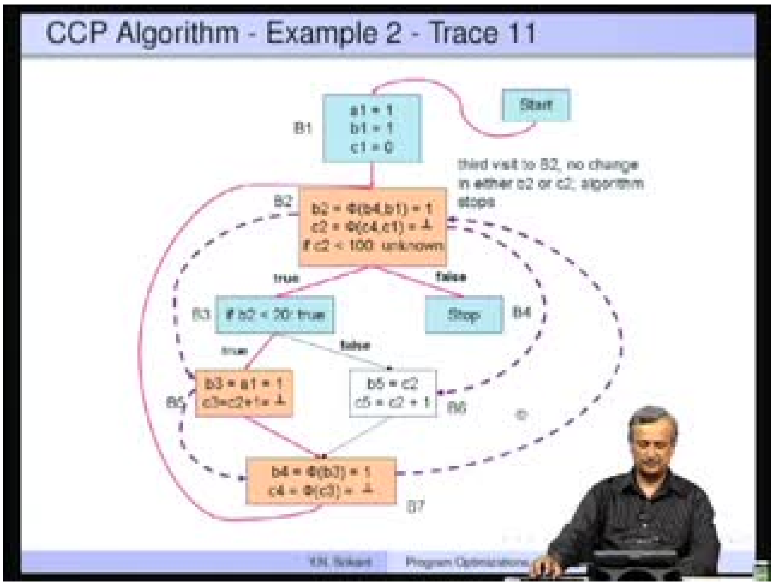
We actually, in this particular basic there is nothing happening which has been entered through this particular SSA edge. So B6 there is nothing happening because this particular edge is not marked as executable. After this we evaluated this B5, c3 becomes not a constant again, b3 is a constant. We would have already come here, through the SSA edge we come here again.

(Refer Slide Time: 21:24)



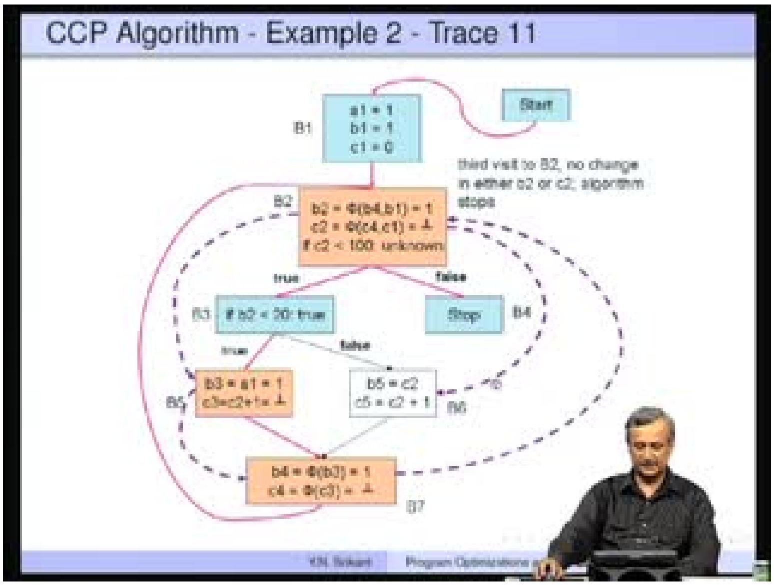
Now evaluate B7 again, so when we evaluate B7 as a second time, b4 remains as 1 but c4 becomes not a constant. There is change value that means we need to propagate the value to this through the SSA, this goes on the SSA pile.

(Refer Slide Time: 21:40)

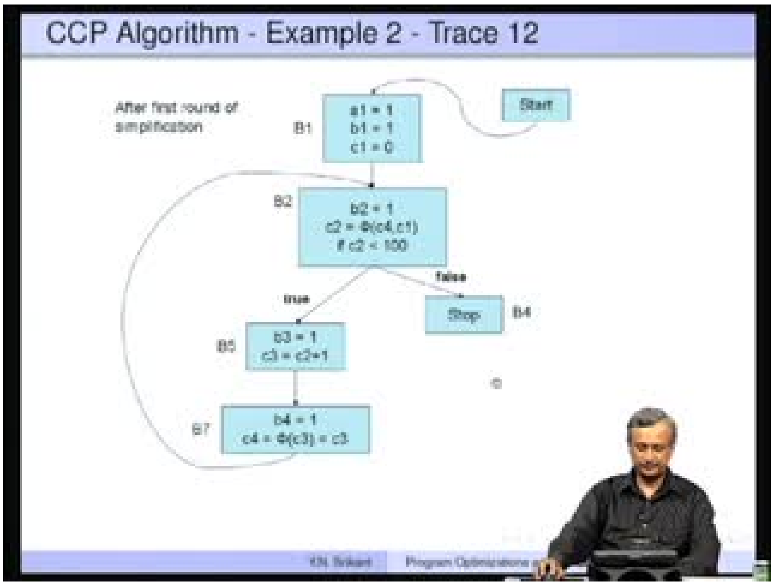


Now, we come to the third visit of B2. There is no change in b2 and there is no change in c2. The b2 remains at 1 c2 remains at not a constant value. So there are no more SSA edges added to the SSApile, there were no flowpile edges added either this was evaluated long back or nothing was done for this particular edge.

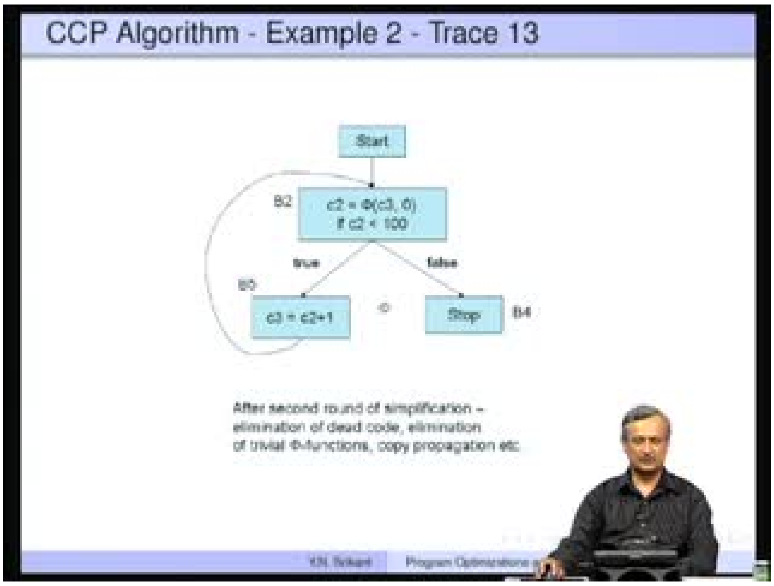
(Refer Slide Time: 22:12)



(Refer Slide Time: 22:18)



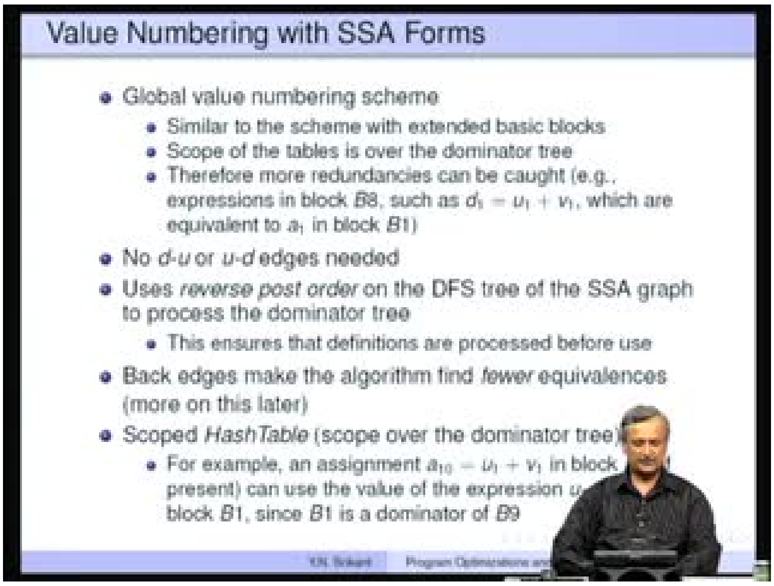
(Refer Slide Time: 22:56)



Finally, we can remove some of these dead edges; this is dead, this is dead so this node is removed (Refer Slide Time: 22:14). Then we get a simplified SSA graph. Here b2 is 1, c2 has not been evaluated, so it became not a constant and it remains as phi of c4, c1. If c2 is less than 100 remains as it is, in this case b3 was evaluated to 1; c3 was not a constant so it remained as c2 plus 1.

Here b4 was evaluated to 1 and c4 was evaluated as phi of c3, which is c3 itself but no for the value as such. Now we can do some copy propagation and removal of course such as b2 equal to 1, b3 equal to 1, b4 equal to 1 and simplify the entire flowgraph to every small one like this. You see that the b variable is completely drawn; this was possible because of conditional constant propagation.

(Refer Slide Time: 23:13)

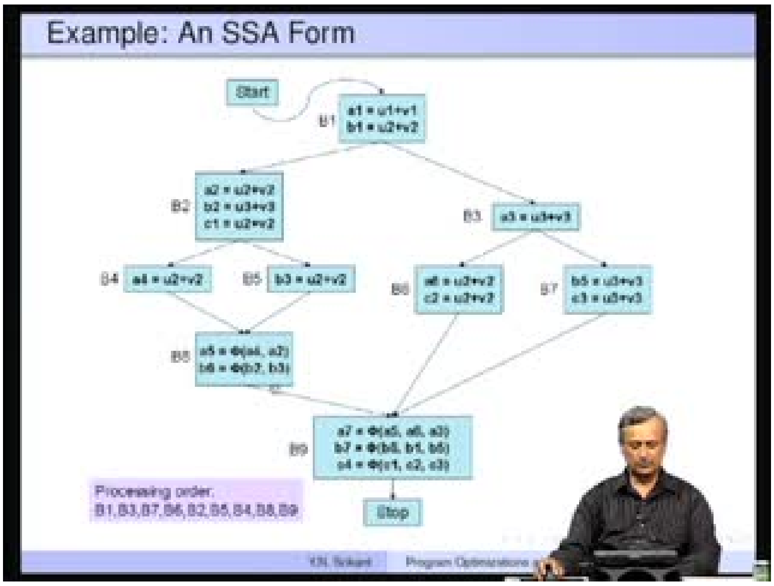


So that is the conditional constant propagation algorithm, that we saw until now. Let us move on to the next optimization known as value numbering. We have seen value numbering before; we did value numbering on basic blocks. We used a hash table, we entered expression into that hash table and whenever we found another expression with the same value number we said these expressions are identical.

So, the variables are also entered into the table but not exactly the same table, it was entered into the name table. Then we saw value numbering with extended basic blocks that actually found a few more cost the commerce of expressions and did few more copy propagation etcetera. The reason was the scope of the expressions and so on and so forth were extended.

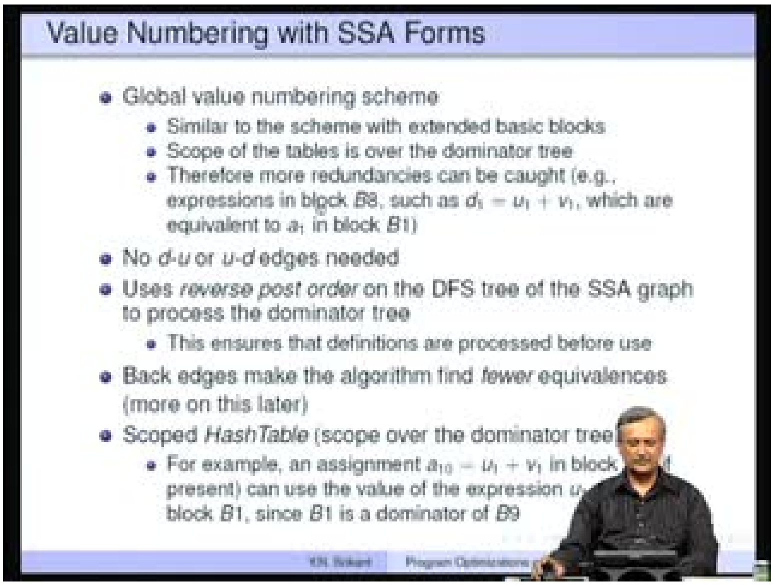
Now with SSA forms we can do even better. This is a global value numbering scheme which is very similar to the scheme with extended basic blocks. But the scope of the tables is over the dominator tree. So it is not the extended basic block that rules scope of the tables but it is the dominator tree. Therefore, more redundancies can be caught for example, I am going to show you picture now, in block B8 suppose you had d_1 equal to u_1 plus v_1 as extra which are equivalent to a_1 in the block B1.

(Refer Slide Time: 25:15)



Suppose, let us stay look at this picture, in block B8 we had d_1 equal to u_1 plus v_1 we had something here and a_1 equal to u_1 plus v_1 is here, so B1 dominates B8. If we had d_1 equal to u_1 plus v_1, we could have use to a_1 as the value of d_1 directly that was possible.

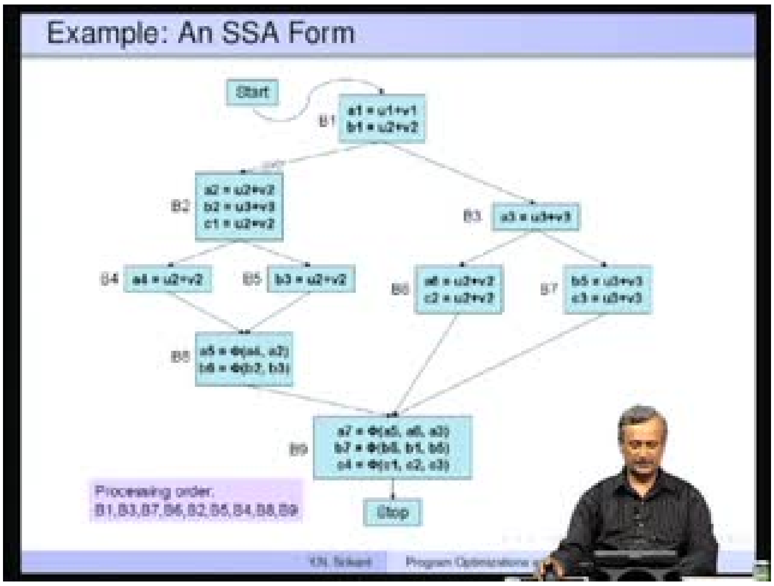
(Refer Slide Time: 25:43)



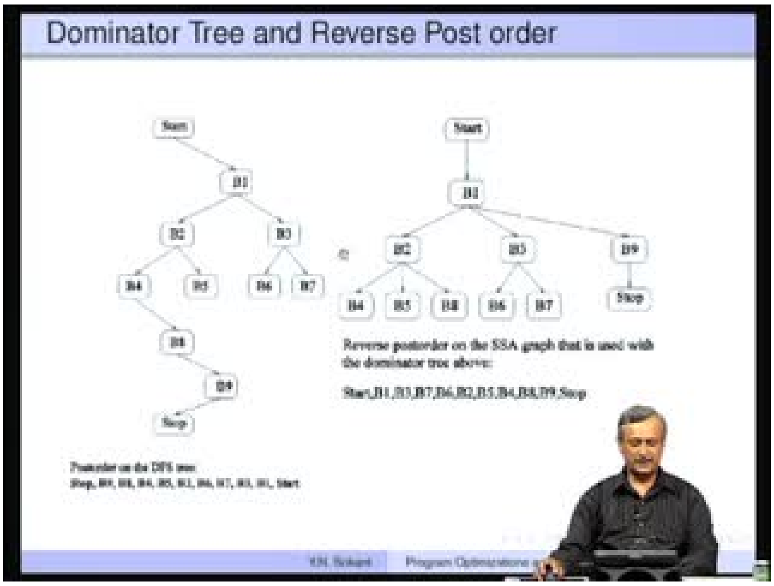
So with dominator as the leading theme; the scope of the table is over the dominator tree. Therefore, we can catch a few more redundancies. We do not need any d-u or u-d change; definition use or use definition, changes are edges are not needed here. They were needed for the conditional constant propagation but they are not needed here.

Another interesting feature is it uses the reverse post order on the DFS tree, so what we will do is we take the SSA graph I will show you that also in a minute.

(Refer Slide Time: 26:35)



(Refer Slide Time: 26:38)

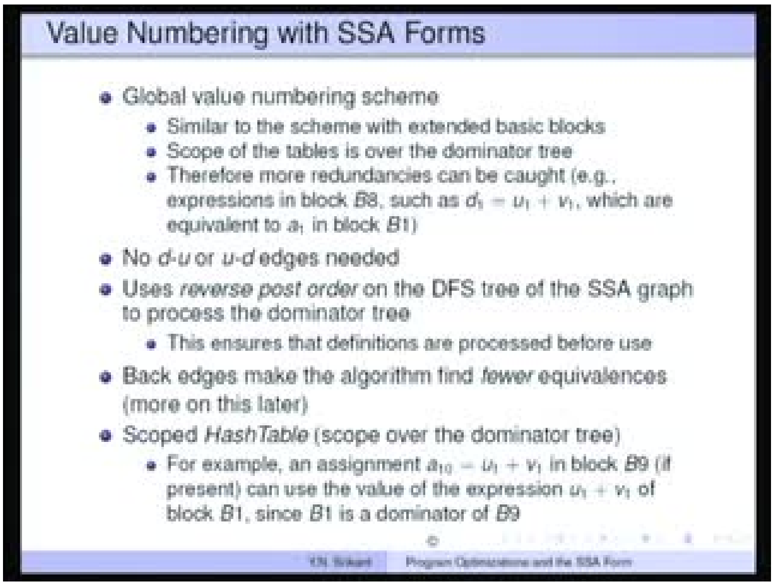


This was the SSA graph, so we do a DFS-depth first search on this. This is the DFS tree, now do a post order on this DFS tree, we get stop B9, B8, B4, B2, B6, B7, B3, B1, and start. Now take the reverse of this so that gives your start B1, B3, B7, B6, B2, B5, B4, B8, B9 and stop. So, start from the start node and use this order on the dominator tree.

We can see that we do a start, then we do B1, then we do B3, then we do B7, then we do B6, then we do B5, then B4then B8, then finally B9 and then stop.

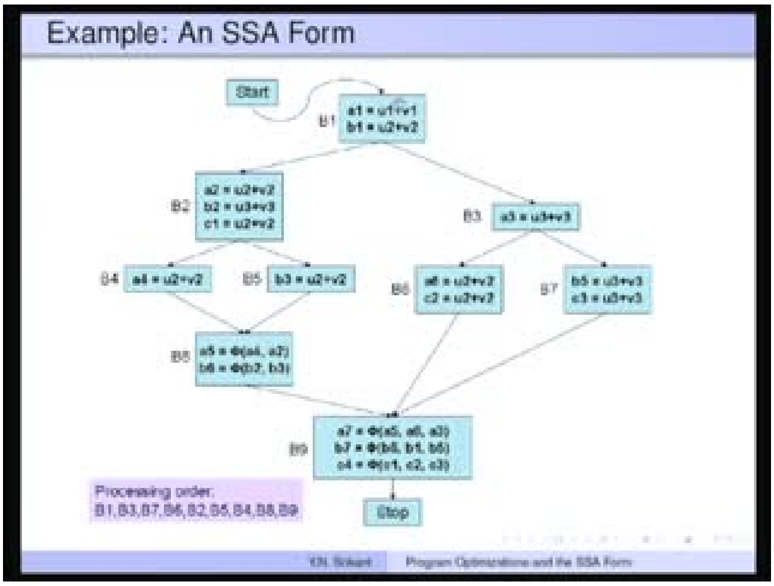
The idea is by the time you actually look at these children; you would have finished the processing of their dominators. Therefore, the expressions which were defined in these dominators are all available for use in these children, so that is what really is the basis of this ordering.

(Refer Slide Time: 28:03)



This ensures that the definitions are processed before use that is a succession clip. The back edges they may be present in the SSA graph, our example does not have a back edge right now, but later I will show you an example with a back edge. So back edges make the algorithm find fewer equivalences. So some expressions which we know are equivalent will not be marked as an equivalent when there is a back edge. This is bad but there is not much we can do about it.

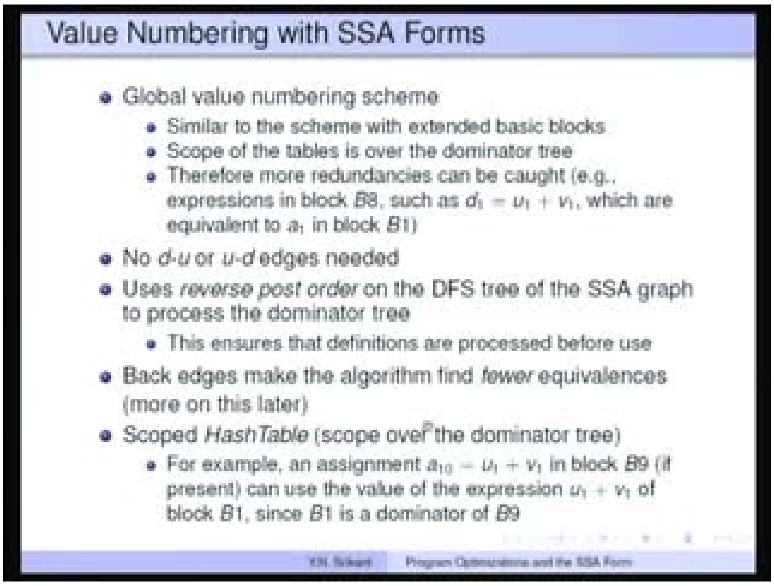
(Refer Slide Time: 29:08)



The hash table that we used to store expressions is scope over the dominator tree. For example, an assignment a_{10} equal to u_1 plus v_1 in block B9, if it is present, this is not present in the example but suppose it present, it can use the value of the expression u_1 plus v_1 of block B1 since, B1 is the dominator of B9.

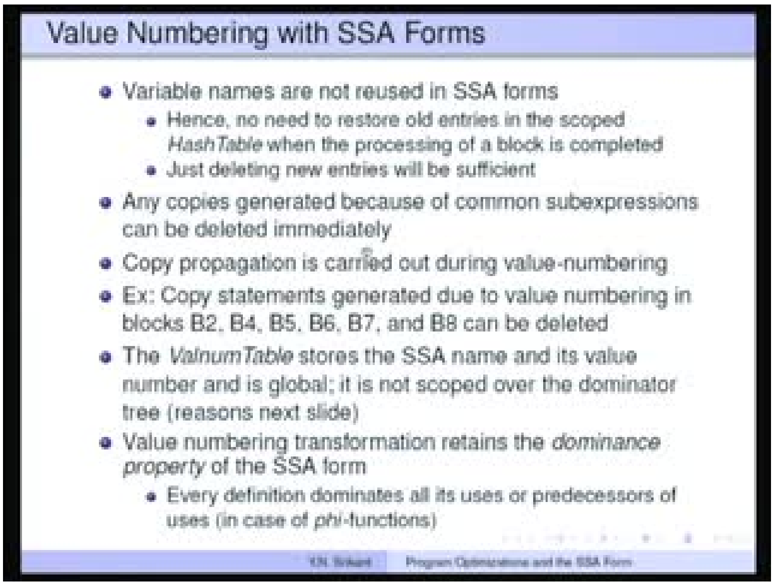
Let me show you that if you had any d_1 equal to u_1 plus v_1 or what was that? That was a_{10} equal to u_1 plus v_1 here. B1 is the dominator; it defines u_1 plus v_1. So, we would have use that directly need not have defined yet and again we could have just used a_1 in place of a_{10} this is possible because B1 dominates B9.

(Refer Slide Time: 29:35)



So that is how the hash table works. If you had not used to scoping over the dominator tree, we would not have caught this.

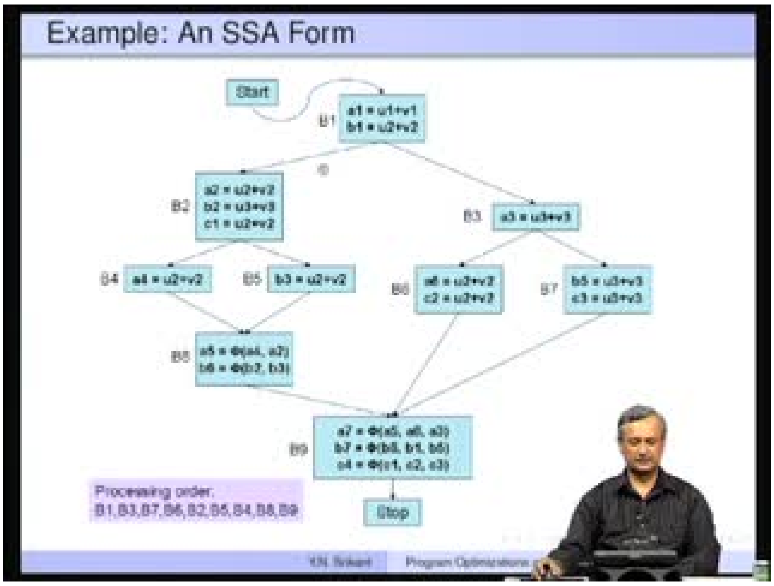
(Refer Slide Time: 29:46)



Now recalls that each name is unique in an SSA form, so variable names are not reused in SSA forms at all. There is no need to store old entries in the scoped hash table when the processing of a block is completed. Remember in the recall that in the case of extended basic blocks, we actually had to remove all the new entries and restore the old entries when we went out of scope. So when we return to the parent we had to remove the new entries which were inserted by the children and then we had restored the old entries also.

That is not necessary here because the old entries are corresponded to old definitions of the same variable, which were redefined in the new scope, here that cannot happen. The each name is going to be defined exactly once so no more redefinitions, just deleting new entries will be enough there is no question of restoring old entries here.

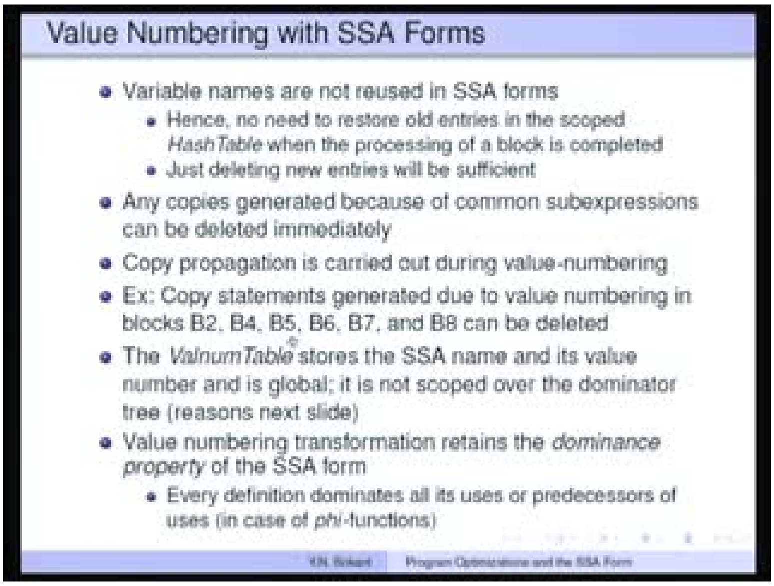
(Refer Slide Time: 31:09)



So any copies generated because of common sub expressions can be deleted immediately. For example, we will see that a_2 equal to u_2 plus v_2, b1 is u_2 plus v_2 so this becomes a_2 equal to b1; b1 is a-eter of b2 see. This is a copy we do not have to retain this copy at all wherever a_2 occurs we will be able to use b1 directly, we do not have to worry whether there is a conflict of interest are something like that.

So how we do that? We are actually going to replace the value number of a_2 with the value number of b1, so whenever we want to search for a_2 automatically get b1.

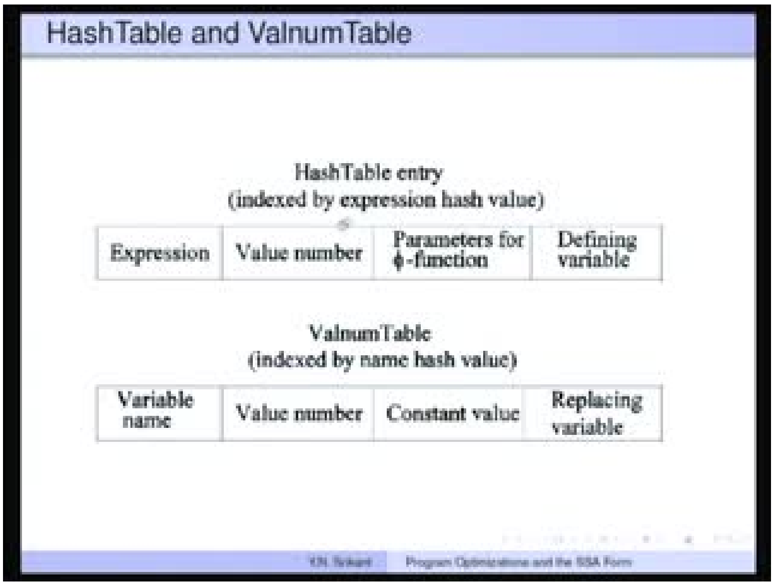
(Refer Slide Time: 31:46)



So that is how these copies can be deleted immediately. Copy propagation is carried out during value numbering itself the way I just now mention. Copy statements generated due to value numbering in the blocks B2 B4 B5 B6 B7 B8 can be deleted. So we are going to see how the deletion happens? The valnum table store the SSA name and its value number and is also is a global table. It is not scoped over the dominated tree I will show you the reasons for it very soon.

Value numbering transformation retains the dominance property of the SSA form. What is the dominance property? Recall this every definition dominates all its uses or predecessors uses in the case of phi functions. So the condition constant propagation did not violate any of this, it actually preserved the dominance property the value numbering transformation, also preserves the dominance property.

(Refer Slide Time: 33:01)

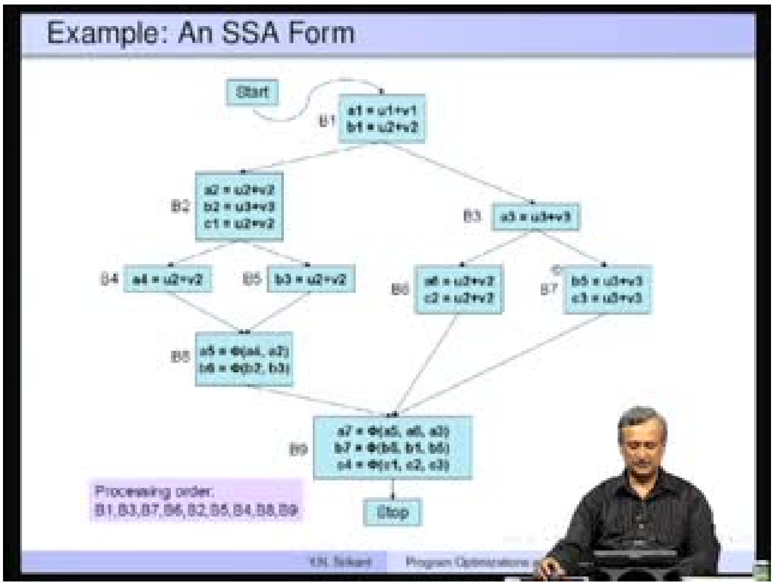


So I wanted to show you the picture of the valnum table that is here, so hash table entry has expression and it is indexed by expression hash value; when we say an expression, a phi function is also an expression; along with the value number of the expression we must also have the parameters of the phi function, there are many parameters those are also stored these will be useful later.

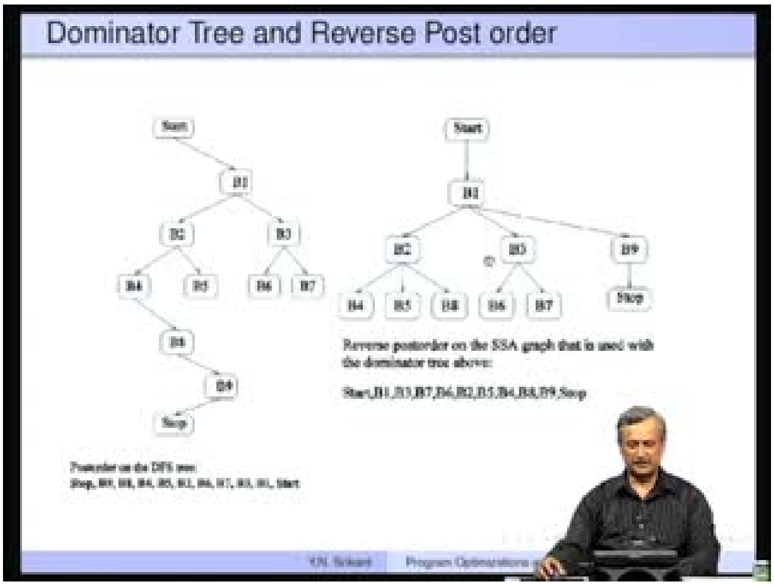
There is something called defining a variable that we need to store here. So the first time that the expression has occurred and of course we take that and the variable on them left hand side of the expression is stored here. Whenever we find expressions equivalent to the expression here within the scope of course, we can use the defining variable in place of the new variable that we have encounter.

The valnum table is simple; it stores the variable name and it stores the value number also, it is actually indexed by name hash value. This constant or not, etc is stored here; if it is a constant, it is a constant value and then the replacing variable. As I told you copies need not be kept, for each copy that variable name we need to have the variable replacing the variable as well.

(Refer Slide Time: 34:30)

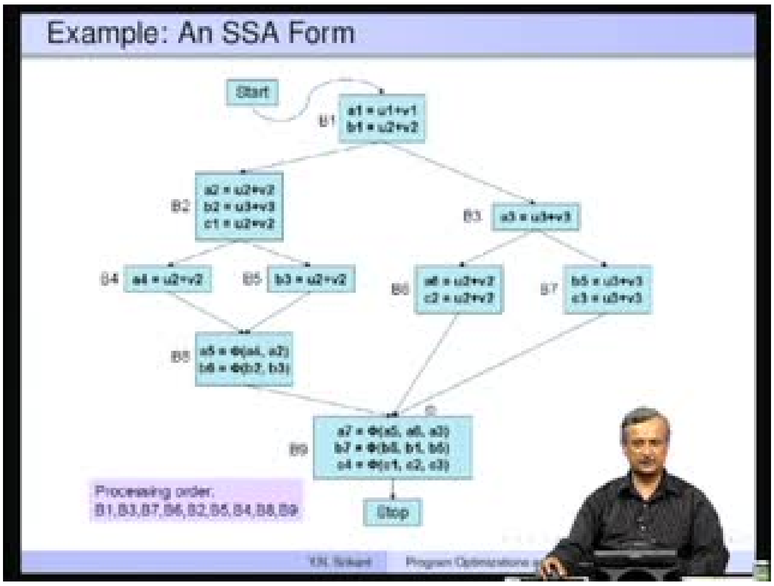


(Refer Slide Time: 34:53)



This is our example that we are going to run through; it has many many redundancies so u_1 plus v_1 is defined here, u_2 plus v_2 is defined here, so u_2 plus v_2 is here, again here, where u_3 plus v_3 another u_3 plus v_3, but remember B2 and B3 do not dominate each other nothing at all. For example, see here B2 and B3 are not dominators of each other.

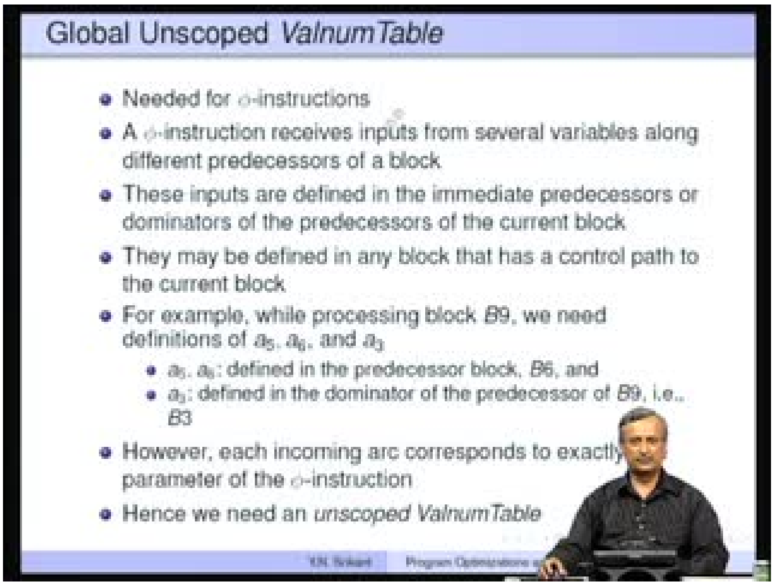
(Refer Slide Time: 34:57)



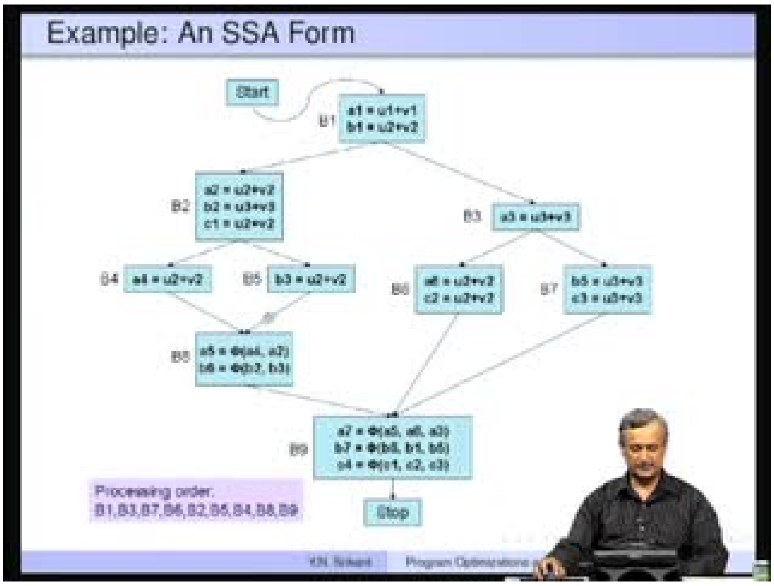
So the u_3 plus v_3 which has defined here and B2 cannot be reused here, if it was actually define here like u_2 plus v_2 it could have been reuse, but not now. Now whatever is defined in B1 can be used in B2 and whatever is defined in B1 and B2 can be used in B4 the reason being B1 and B2 dominate B4. Similarly, whatever is defined in B1 and B2 can be used in B5.

Finally, what about B8? B1 of course dominates B8 so whatever is here can be used here, B2 dominates B8 so whatever here can be used here, but neither B4 nor B5 dominate B8, so whatever is used here defined here, cannot use as a commerce of expression here. But we must remember that B4 and B5 supply parameters to the phi function here. For example, a5 is phi of a4 comma a2, a4 comes from here and a2 comes along this path. So it would actually come with this way, a2 is defined here, but it comes this way.

(Refer Slide Time: 36:47)



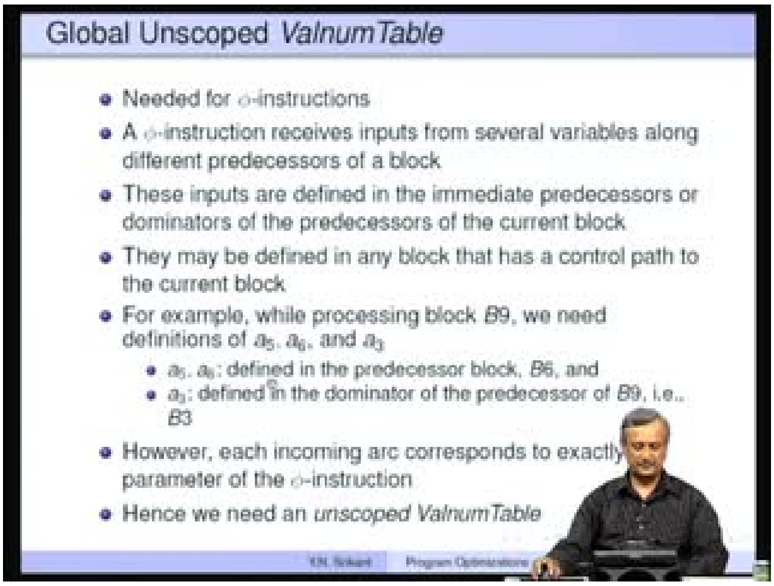
(Refer Slide Time: 37:36)



Similarly, B6 gets B2 from here like that and B3 directly; the same is true for B5, B6 and B7. When we come to B9 we get one parameter from here that is for a7, a5 is defined in B8 so that comes here, a6 is defined in B6 that comes here, and a3 is defined in B3 is at comes via B7, for B7 it is similar, c4 is also similar (Refer Slide Time: 36:30).

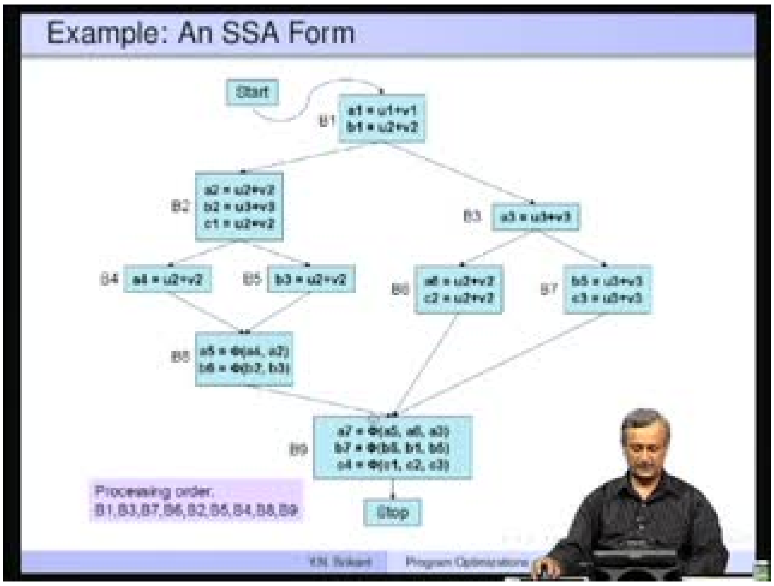
Let us see why the valnum table should be unscoped and then run through the example, so the unscoped valnum table is really needed for processing phi instructions. For example, a phi instruction receives inputs from several variables along different predecessors of a block; these inputs are defined in the immediate predecessors or the dominators of the predecessors of the current block.

(Refer Slide Time: 37:46)

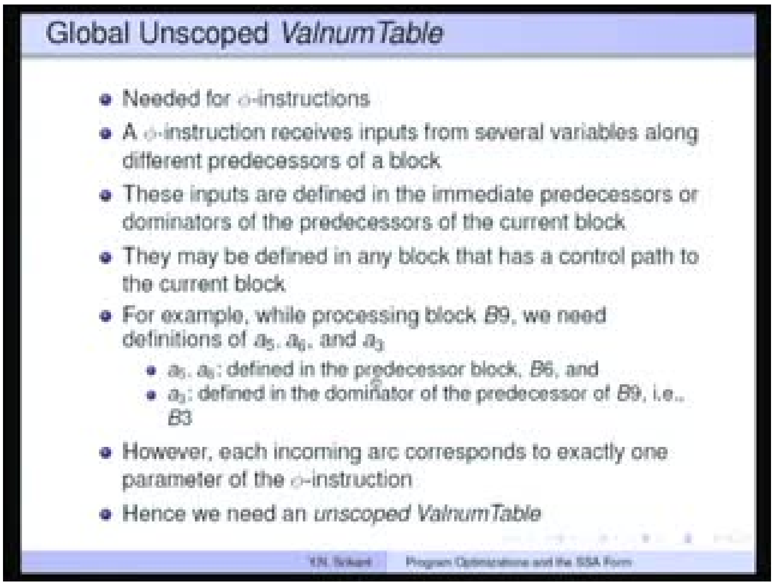


This is obviously true because of the dominance property, so phi gets many variables along its incoming edges and they could be defined in immediate predecessors. For example here a4 is defined in the immediate predecessor, but a2 is defined in the dominator that is upwards that is what it is saying or dominators of the predecessors of the current block.

(Refer Slide Time: 38:09)



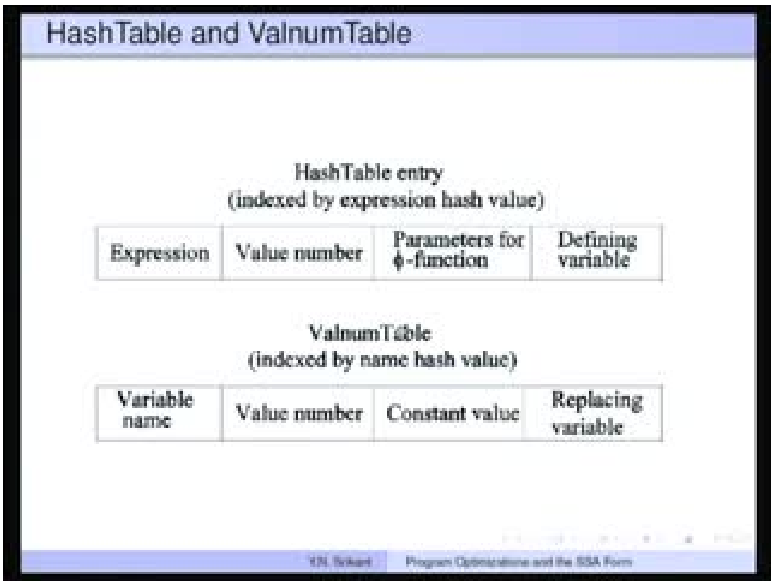
(Refer Slide Time: 38:38)



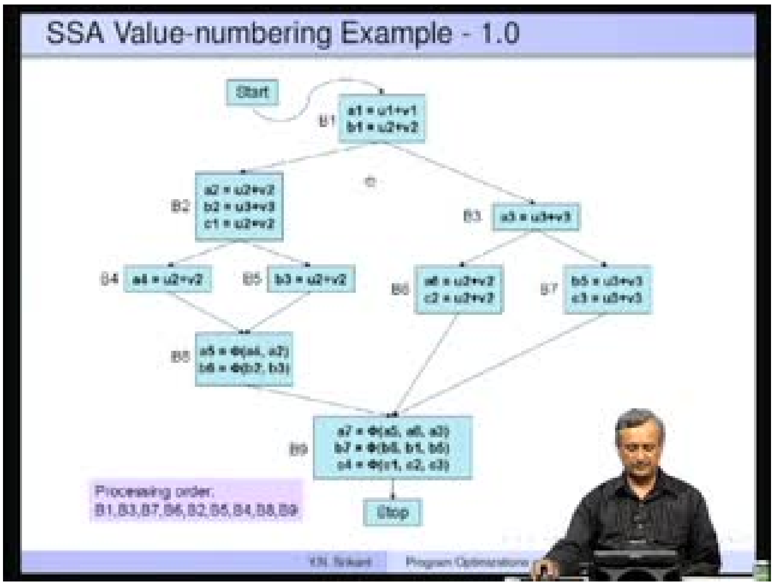
Of course, they may be defined in general in any block that has a control path to the current block subject to the dominates property. For example, while processing block B9 we need a5, a6, and a_3. Let us look at that a5, a6, and a3. I already showed that a5 comes from here, a6 also comes from here, but a3 comes from this point (Refer Slide Time: 38:10). Whereas if you take b7, b6 comes from here, b1 actually comes from all the way from the top and b5 comes from here.

So in general the variable could come from anywhere subject to the dominants property of course, anyone of the dominators. However each incoming arc corresponds to exactly one parameter of the phi instruction. Since, the parameter can come from any of the dominators or the predecessors, we need an unscoped value number table it is not possible to use a simple scoped table, we need an unscoped valnum table for this.

(Refer Slide Time: 39:06)

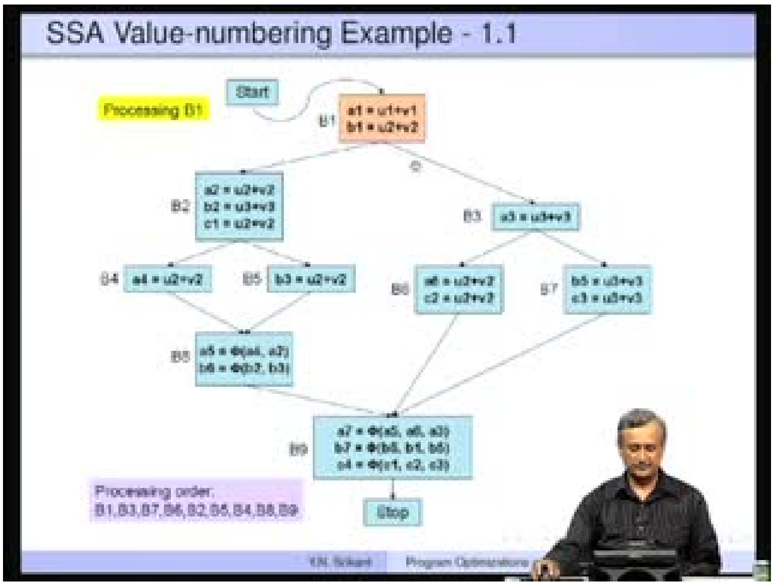


(Refer Slide Time: 39:10)



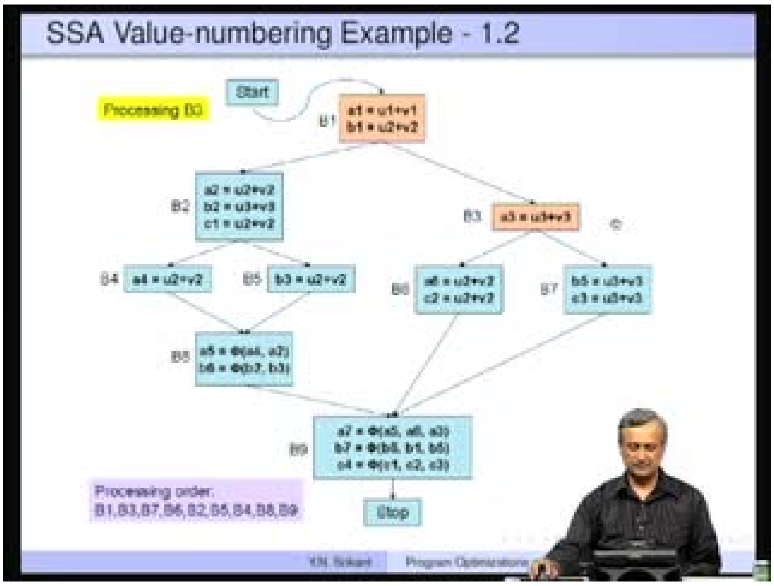
This is the picture just now I showed you, let us run through this particular example and see exactly how value numbering happens. There are more points to be noted as we go along and we will define them at the end of the example.

(Refer Slide Time: 39:27)

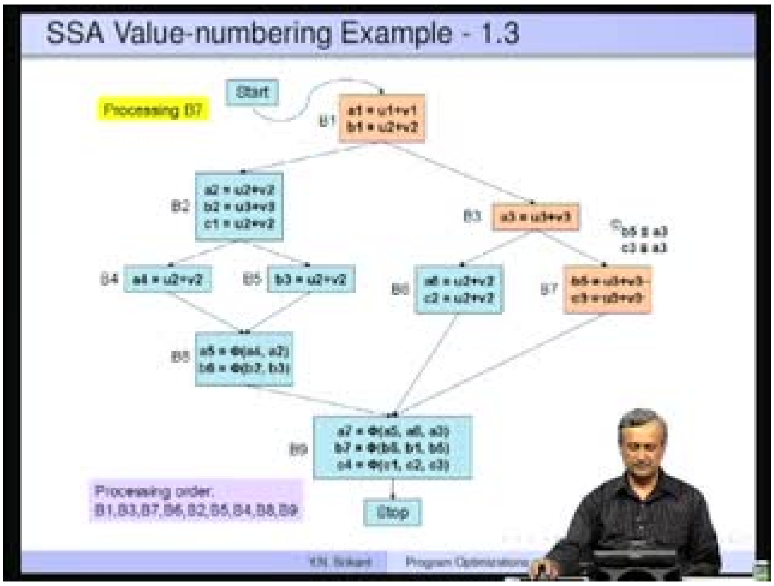


The processing order is always given here, we are now processing the block B1 (Refer Slide Time: 39:32). It has two assignment statements a_1 and b1 equal to u_1 plus v_1 and u_2 plus $v_{2.}$ So u_1 plus v_1 and u_2 plus v_2 are entered into the hash table and then a_1 and b1 actually get entered into the valnum table no problem at all.

(Refer Slide Time: 39:52)



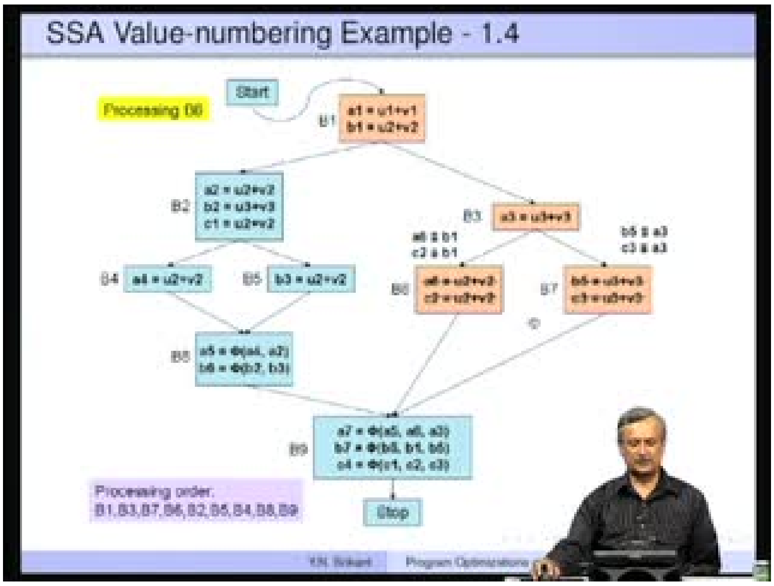
(Refer Slide Time: 40:03)



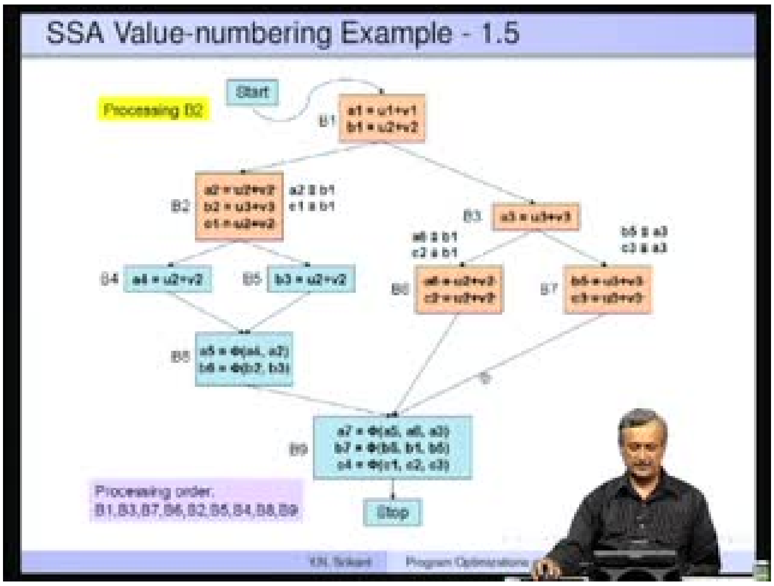
Then the next node to be visited is B3, so a3 is defined here as a same treatment u3 plus v3 is entered into the hash table and into the valnum table. Then B7, so this defines u3 plus v3 which was already defined before, because the hash table is scoped over the dominator B3 dominates B7. We find this b5 equal to u3 plus v3 is nothing but defined already, so we have b5 equal to a3 as equivalent a3 is right here then c3 is equal to u3 plus v3 that is also equivalent to a3 (Refer Slide Time: 40:30).

So we delete these two statements because copy propagation can be directly done, we can store in place of b5 and c3 as a3 itself and that value number automatically takes care of these two usages later.

(Refer Slide Time: 40:56)



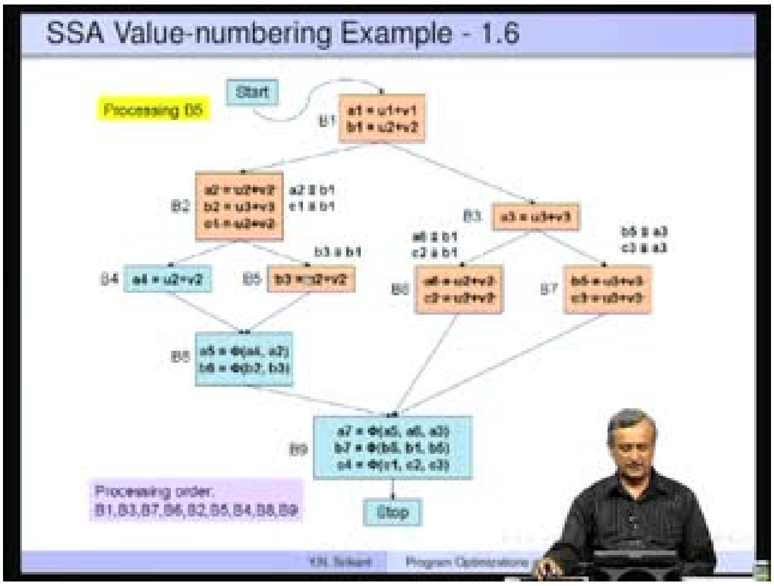
(Refer Slide Time: 41:13)



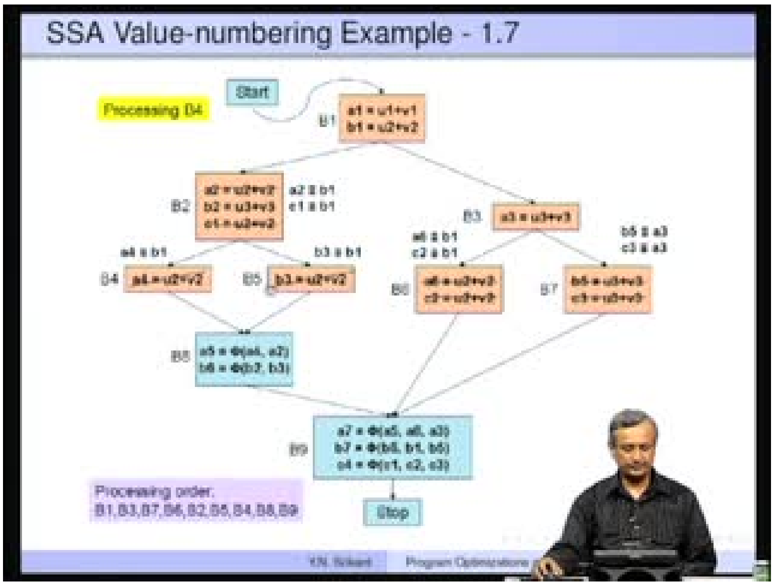
Then we process B6 in that reverse post order, here we find a6 as b1 so u2 plus v2 which is defined here in the dominator scope and c2 as b1 again, so that is also defined here and we can delete these two (Refer Slide Time: 41:10).

Then we process B2, so here we find a2 as b1 which was defined before, b2 as u3 plus v3 which is not defined before, because this is a different dominator scope as such so B3 would have already gone out of scope, B2 is a new dominator scope so this is retained as it is and entered into the tables, but c1 as u_2 plus v2 is already equivalent to b1, c1 is equivalent to b1, because u2 plus v2 is already defined.

(Refer Slide Time: 41:54)

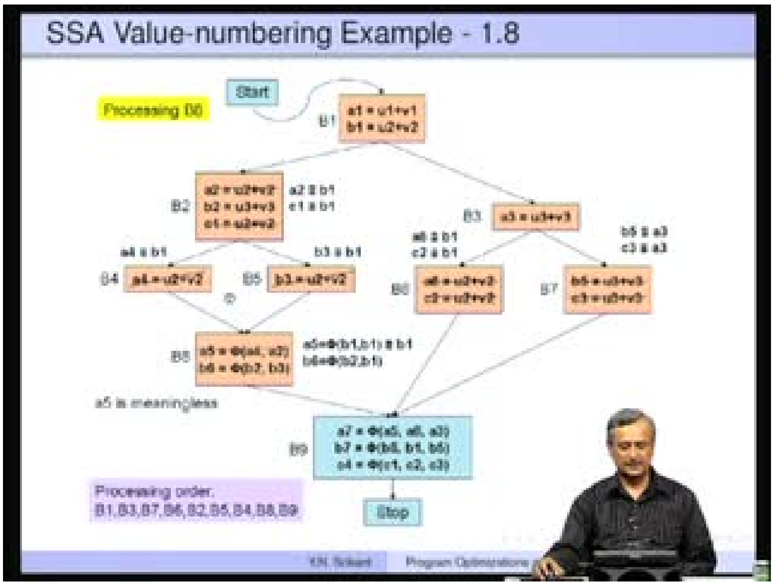


(Refer Slide Time: 42:03)



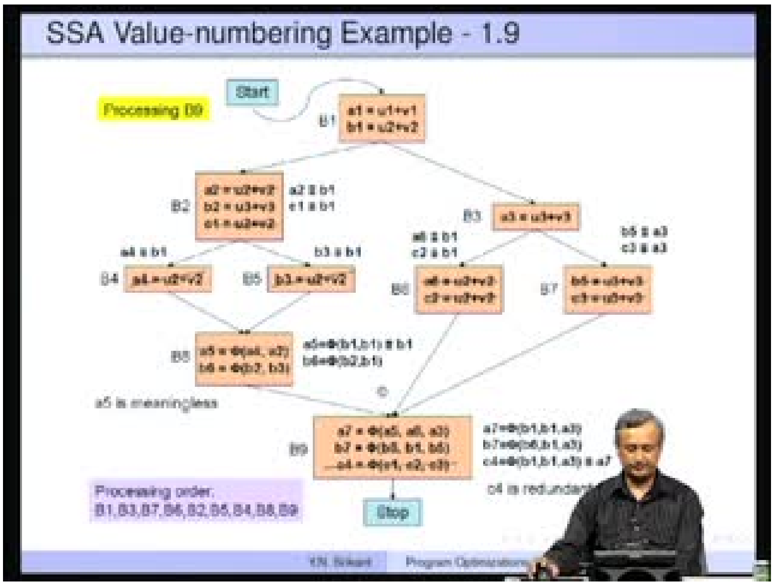
Then we come to B5 here we find u2 plus v2, b3 is equivalent to b1 so this is deleted. Here in B4, a4 is equivalent to b1 this is also deleted.

(Refer Slide Time: 42:09)



Then we come to B8, so in B8 a5 is a phi function with a4 and a2 as parameters. If you trace back when you search the valnum table a4 is equivalent to b1, so that is what we have replaced here and a2 is equivalent to b1 again, the second parameter is also b1. So such phi functions which have all parameters as equal the same are meaningless phi functions. So a5 is meaningless phi function we do not need it here, we can simply replace the phi function by the parameter b1. So a5 is equal to b1, but b1 is already there before. This is a copy statement and we can delete this as well.

(Refer Slide Time: 43:06)

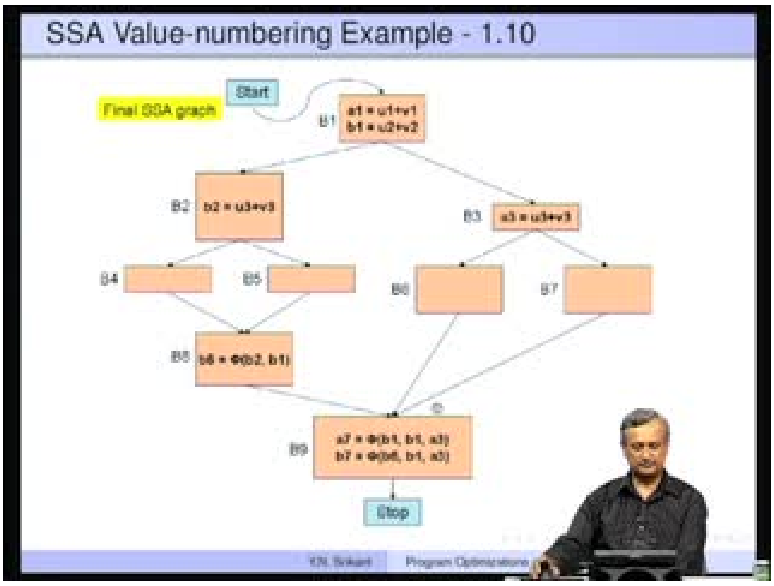


But b6 is not so b6 is remains as phi of b2 comma b1. When we come to B9, a7 is a phi function with a5, a6, and a3 as parameters; a5 is famous b1 so the first parameter is b1; a6 is same as b1 second parameter is also b1; a3 is a3 there is no change, so it remains as it is the phi function does not become meaningless.

The second phi function b7 has b6 b1 and b5 as 3 parameters; b6 is as it is; b1 is as it is; and b5 is equivalent to a3, it is replaced by a3 so this phi function is also not meaningless it remains as it is. Whereas c4 equal to phi of c1 comma c2 comma c3 c1 c2 c3 are equivalent to b1 b1 and a3; c1 is b1 here, c2 is b1 and c3 is a3.

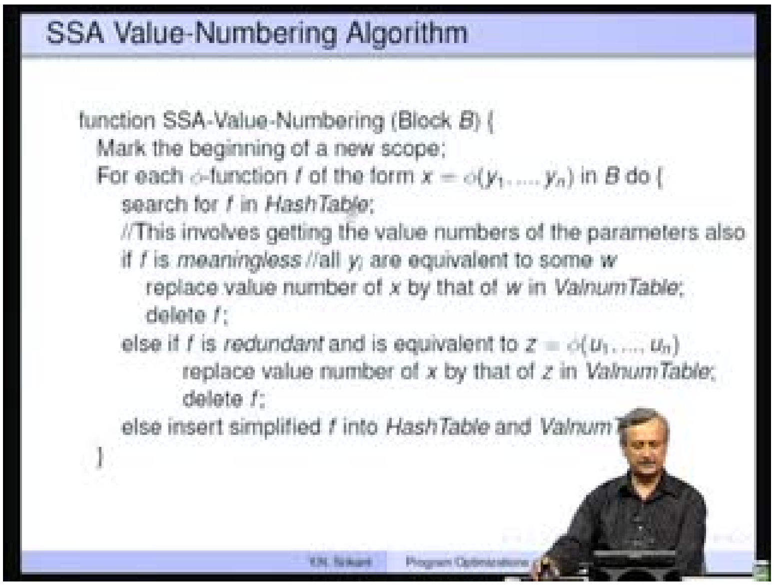
So now observe that a7 and c4 are exactly identical. Therefore, one of these needs to be retained and the other can be removed c4 will be removed and it is called as a redundant phi function, which is already covered by some other phi function in the same basic block.

(Refer Slide Time: 44:34)



This is the simplified SSA graph after the copy statements and redound expression etc are all removed. So this is how we are able to simplify.

(Refer Slide Time: 44:47)

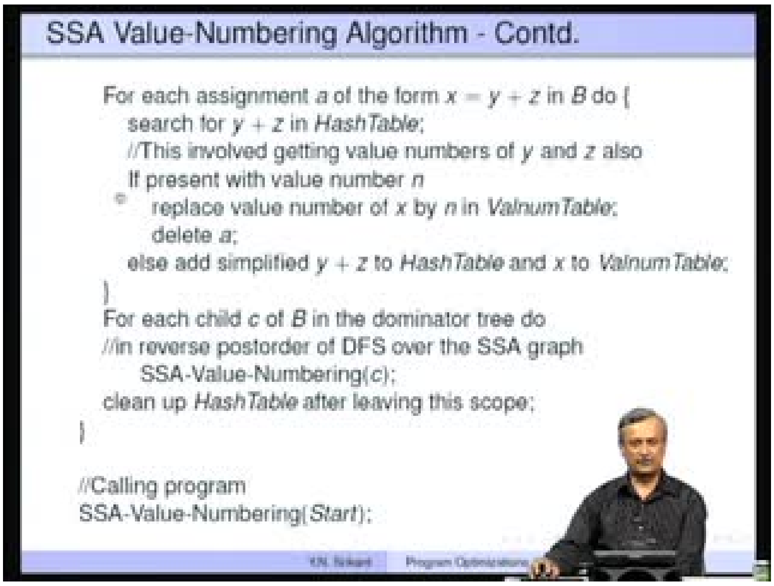


Let us look at the SSA value numbering algorithm in a formal manner mark the beginning of a new scope, so the basic block B is the parameter for each phi function f of the form x equal phi of y_1 to y_n in the basic block B do search for the function f in the hash table name is x and the parameters are y_1 to y_n. So this involves getting the value numbers of the parameters, you have to dig into valnum tables get their parameters etc etc. Then use a special hashing function for phi and there were many number of them available and then enter into the hash table.

Suppose you find that f is meaningless this will be defined later, all y_i are equivalent to some w that is all the parameters are equal. Now replace the value number of x by that of w because this becomes x equal to w, all these are w in the valnum table and delete f suppose, it is not meaningless but it is redundant, so redundancy is there is some other phi function, which is equivalent to this f of the form z equal to phi of u_1 to u_n in the same basic.

So replace the value number of x by that of z in the valnum table and delete f so I already showed you this otherwise, simplified f into hash table and the valnum table.

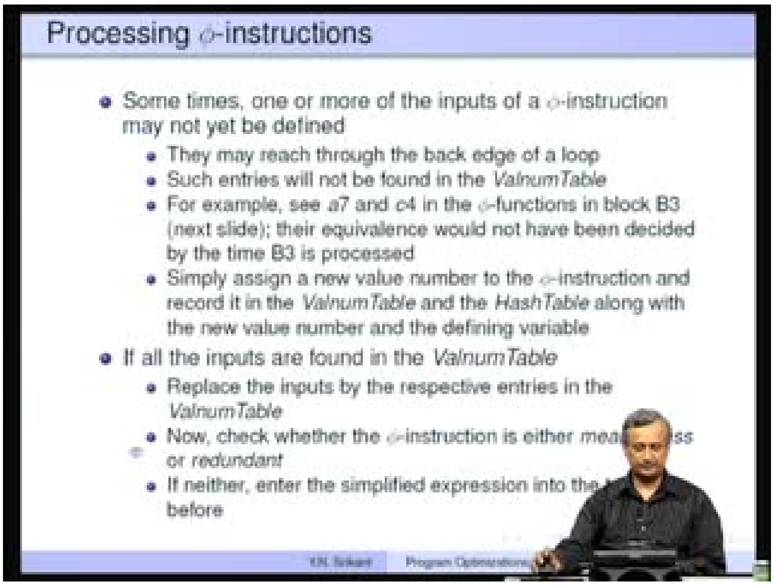
(Refer Slide Time: 46:25)



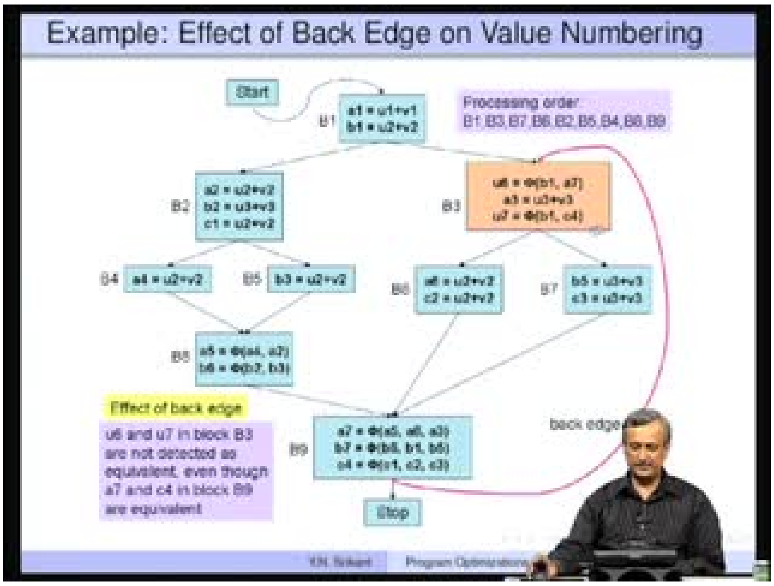
Then this is for phi function, what do you do for assignments? If it is x equal to y plus z search for y plus z that implies take the value numbers of y z etc etc, apply a hash function and search a hash table. If it is already present with value number n then replace the value number of x by n in the valnum table and delete a. I showed this if there is an expression already defined we do not have to keep the copy later. Otherwise, add the simplified y plus z to hash table and x to the valnum table, so this is as before.

For each child, now we have finished processing B, so what about the children of B? In the dominator tree in the reverse post order of DFS as I told you about the SSA graph call SSA value numbering for each of the children. Finally, once we want to get out of B clean up the hash table after leaving this particular scope, so initially we supply start as the parameter and then call the function.

(Refer Slide Time: 47:38)



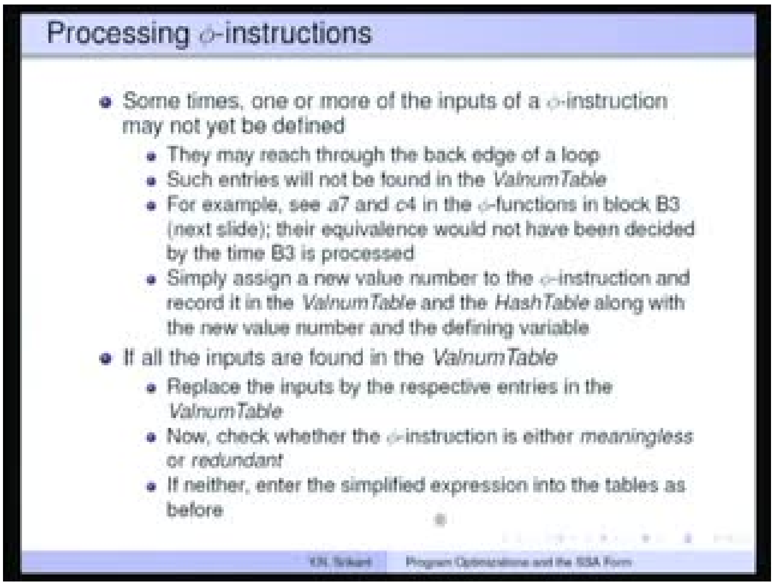
(Refer Slide Time: 47:56)



Let us look at some details of how phi functions are processed? One or more inputs of the phi functions may not yet be defined for example, they may reach through a back edge of phi of a loop, and such entries will not be found in the valnum table. Let me show you an example, this B3 have been replaced by a new block which has u6 and u7 as new definitions, the old one had only a3 equal to u3 plus v3.

Now u6 is phi of b1 comma a7 and u7is phi of b1 comma c4. Unfortunately, we found that a7 and c4 even though are equivalent we have not processed this block, so we have not found that a7 and c4 are equivalent. We are processing this block we have not processed this block and there is a back edge here. So because of that we do not find u6 and u7 as equivalent, they are not redundant phi functions at all because a7 and c4 are not yet found as equivalent.

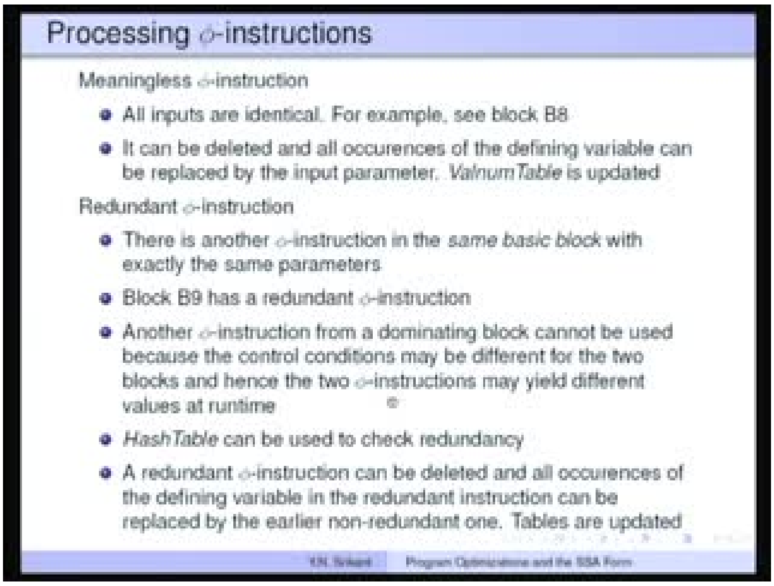
(Refer Slide Time: 49:05)



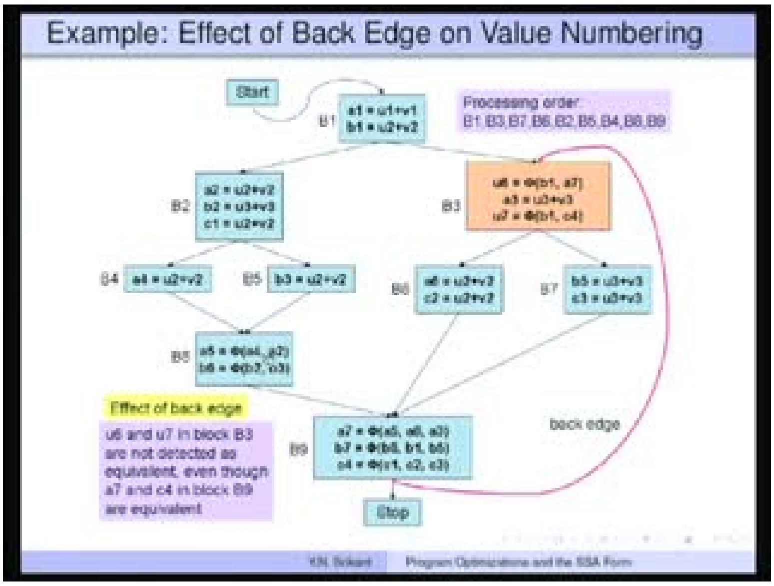
So because of this back edge a7 and c4 will be treated as distinct, separate. Therefore, this u6 and u7 will be actually entered into the hash table as if they are two different phi functions, which are not equivalent to each other. So simply assign a new value number to the phi instruction and record it in the valnum table and the hash table along with new value number and the defining variable that is what we do.

So we do not really go through the value numbering scheme again and again. So that is why this is not done, we just want to do it once it takes too much time. If all the inputs are found in the valnum table, then replace the inputs by the respective entries in the valnum table. Check whether the phi function is either meaningless or redundant if neither enters the simplified function into the tables.

(Refer Slide Time: 49:44)

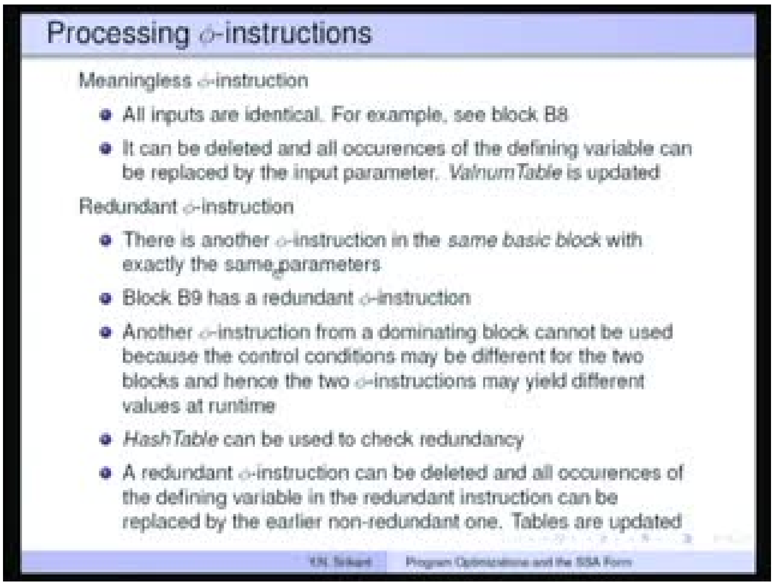


(Refer Slide Time: 50:11)



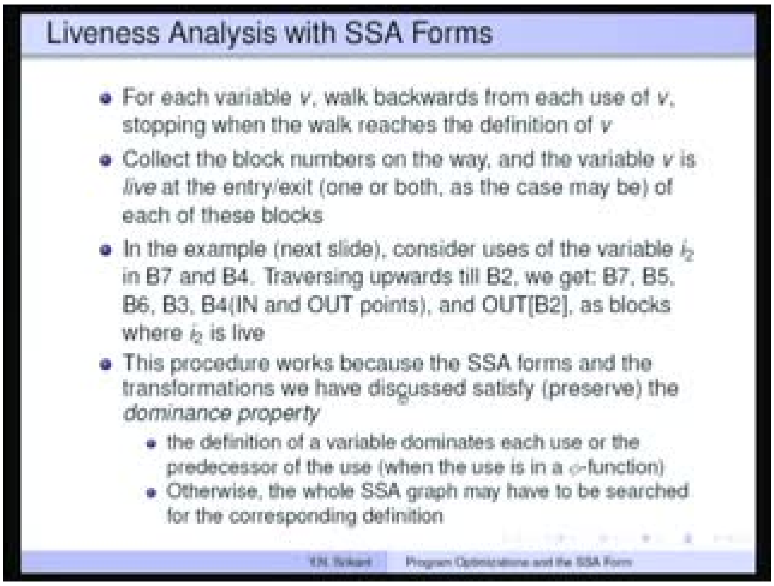
Now what about meaningless and redundant phi functions? All inputs are identical so for example, in block B8 as I showed you if all the inputs are equal, then they can be deleted this particular instruction can be deleted. Occurrence of defining variable can be replaced by the input parameter; only valnum table needs to be updated. We saw this example already for example, here this and this they become meaningless therefore, both this parameter become equal (Refer Slide Time: 50:15).

(Refer Slide Time: 50:22)



Redundant phi instruction means, we have already discussed this instructions in the same basic block with exactly the same parameters. So redundant phi instructions can be deleted in all occurrences of the defining variable in the redundant instruction can be replaced by the earlier non redundant one tables are updated.

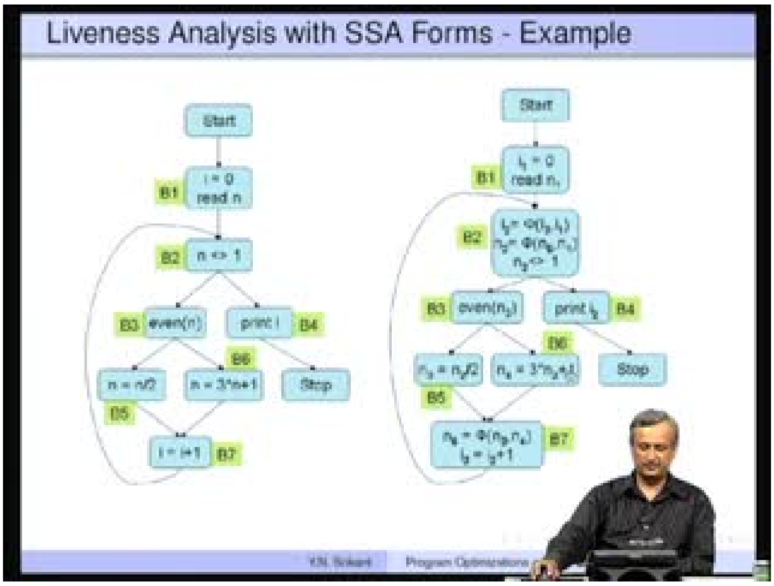
(Refer Slide Time: 50:50)



So this is how value numbering actually happens in SSA forms. Now let us look at something somewhat interesting we saw value numbering on SSA graphs, which actually preserved the dominance property. Now we will see how important it is for Liveness analysis.

What is liveness? Liveness is whether a variable is going to be used at a particular sometime later that is, informally what liveness is of a variable? Really is for a variable which is defined is there any use later on. So how do you find it in the SSA graph? It is very simple for each variable v; walk backwards from each use of v, stopping when the walk reaches the definition of v. So, collect block numbers on the way and the variable v is live at the entry or exit, one or both as the case maybe, of each of these blocks.

(Refer Slide Time: 51:57)

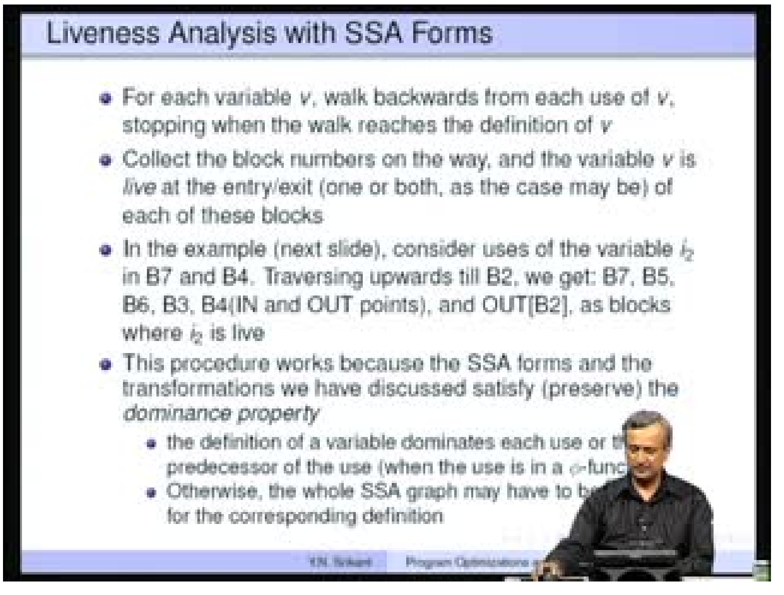


So let me show you a simple example, here is our original flow graph and here is our SSA flow graph. Let us look at some usage let say $i_{2,}$ i_2 is used here and i2 is used here also (Refer Slide Time: 51:60). Now we want to find out all the blocks where i_2 is live. Let us take this which is very simple, we go backwards till the definition of i so we go up to his point i_2 is defined here from here to here.

So in block B4 it is actually live at the entry of block before it is live, but the end of block it is not because there are no more usages here. At the output of block B2 it is live and at the entry of block B2 it is not live, because i_2 is defined immediately. So B4 in and B2 out is collected, then we start here so B7 in is collected, then B5 out is collected, B5 in is collected, B3 out is collected, B3 in is collected, and then B3 out is already there and finally, we have reached the definition point of i_2 so all these blocks are deemed as live blocks for i_2.

We can do the same thing in along the other path also from here B7 in B6 out B6 in then B3 onwards we have already collected. So B2 and B3 both in and out, B4 only in B5 B6 both in and out and finally B7 in these are the points where i_2 is live. What about n_2? So you can look at n_2 which is here, which is using here also, for n_2 you go on you just defined here so B5 in B3 out B3 in and B2 out.

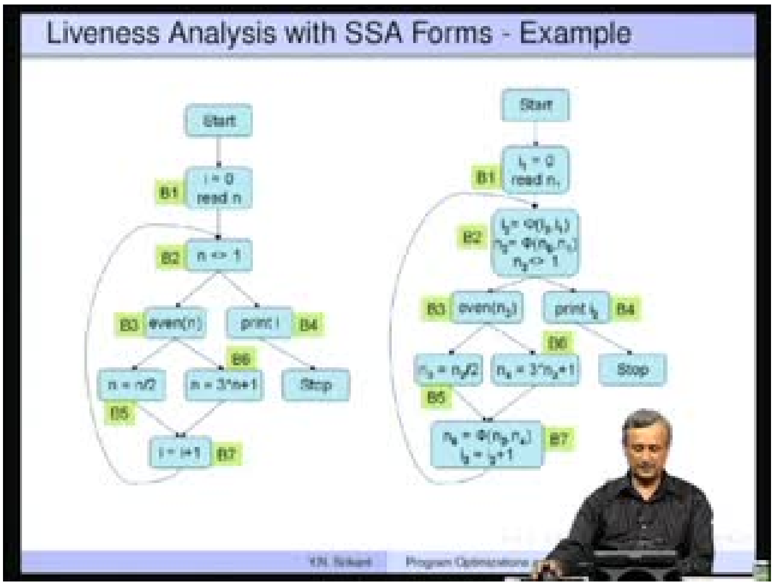
(Refer Slide Time: 54:20)



Similarly, here B6 in and then onwards we have already collected. Whereas if you look at n_3 or n_4 we start n_3 is defined here, so only this much n_4 is defined here. This is how the liveness is computed? We just go walk backwards from each use of v stopping when the walk reaches the definition of v, collect the basic block numbers on the way and the variable v is live at the entry and exit, one or both of these basic blocks. So I already showed you this particular example.

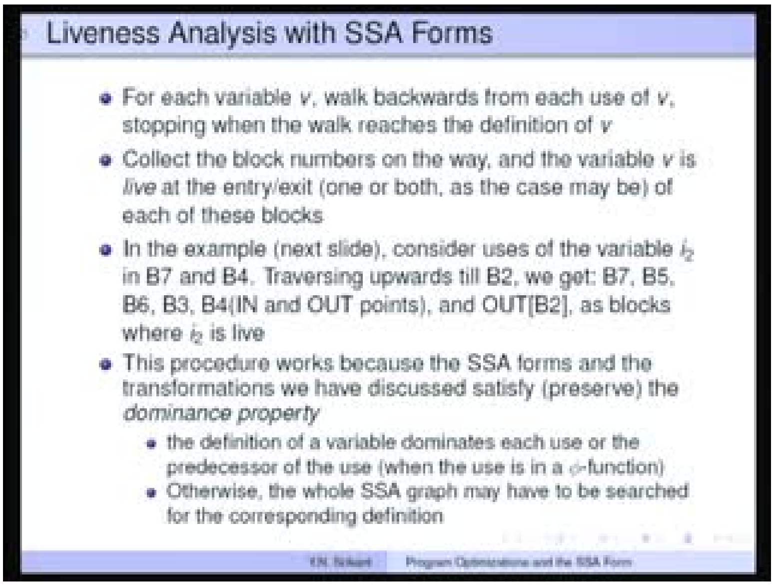
Why this procedure works? It works because the SSA forms and the transformations we have discussed satisfy or preserve the dominance property, so again we must recall that the dominance property is definition of variable dominates each use or the predecessor of the use, when the use is in a phi function.

(Refer Slide Time: 55:27)



Suppose the dominance property was not satisfied then there is a problem what may happen is that the whole SSA graph may have to be searched, we do not know dominance property is not satisfied. Here we are 100 percent certain that if we reach the definition that is sufficient because the definition dominates all the uses or at least the predecessors of the uses.

(Refer Slide Time: 55:41)



So because of that we stop at the definition, we do not have to go beyond that whereas if the dominance property is not true, then the definition could be anywhere it is not necessarily up to the dominator only. So the whole SSA graph may have to be searched and to find the corresponding definition.

This is a sample of how liveness analysis is possible. It is also possible to actually do partial redundant elimination and a few others, but they are much more complex than what I have presented so far those are outside the scope of this particular, of course. So this is the end of the lecture and in the next time we will look at parallelization thank you.