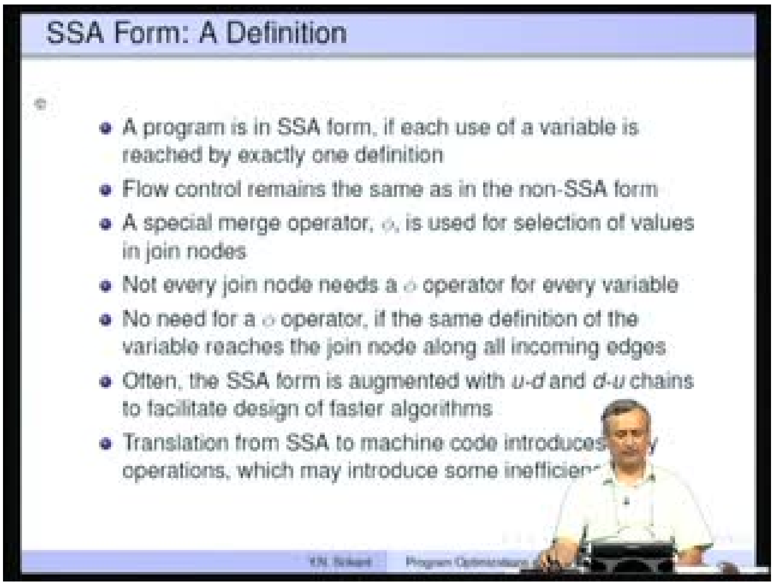
Compiler Design
Prof. Y. N. Srikant
Department of Computer Science and Automation
Indian Institute of Science, Bangalore

Module No. # 13
Lecture No. # 32
The Static Single Assignment Form:
Construction and Application to Program Optimizations - Part 2

Welcome to part 2 of the lecture on the SSA form.

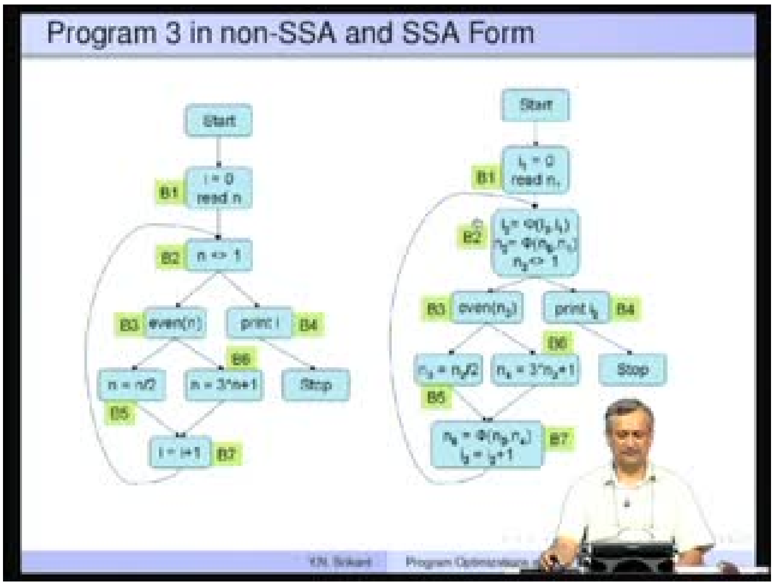
(Refer Slide Time: 00:22)



Let us recapitulate a little bit. A program is said to be in SSA form, if each use of a variable is reached exactly by one definition. The flow control remains exactly as in the non-SSA form, but there is going to be a special operator called the phi operator, which is introduced into the join nodes. The phi operator is useful for the selection of values in join nodes. So, there may be a… because we have a condition that there is exactly one definition for each use. There may be many definitions of the same variable reaching a point and which one to choose is the question. This is resolved by the phi operator.

Not every join node will need a phi operator. If the same value is coming through all the paths, all the edges into a join node, then we really do not need a phi operator there.

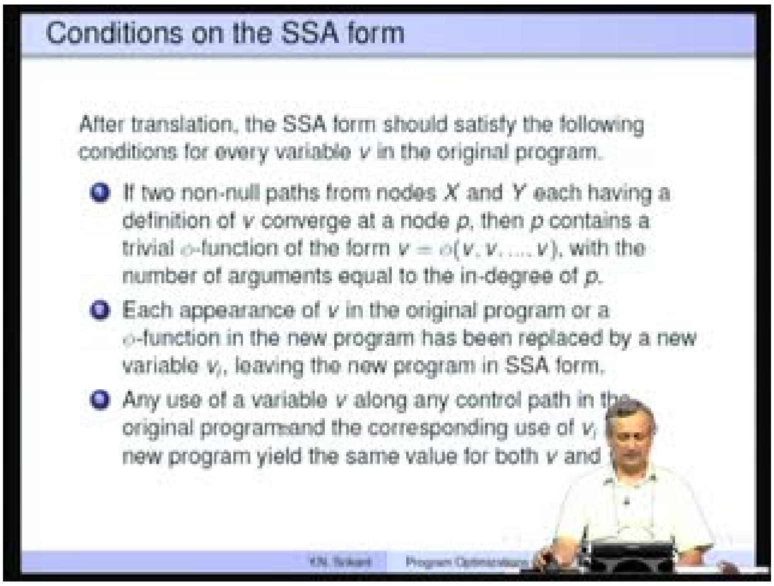
(Refer Slide Time: 01:34).



Here is an example of a non-SSA form and the SSA form. Here this is the non-SSA and this is the SSA. You can see that there are two definitions of i coming here: one is coming this way and the other is coming this way. We have phi operator for i. Similarly, there is a definition of n coming into this and then these definitions of n are again entering this. So, we have a phi operator for n here. Here is a phi operator for n because of these two definitions coming in. These are the salient points of an SSA form.

The semantics is simple. When the phi operator is supposed to execute depending on the arc through which this node is entered, appropriate parameter is chosen. For example, if we enter through this (Refer Slide Time: 02:32), then the first parameter is chosen and if we enter through this, the second parameter is chosen. That is assigned to the left hand side.

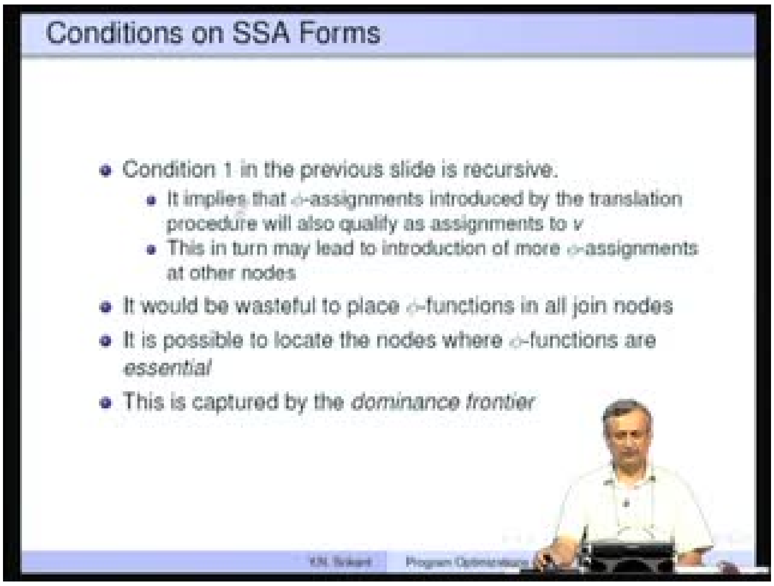
(Refer Slide Time: 02:40).



There are some conditions on valid SSA forms. The first one says - if two non-null paths from nodes X and Y each having a definition of v converge at a node p, then p contains a trivial phi function of the form, v equal to phi v comma v comma etcetera.

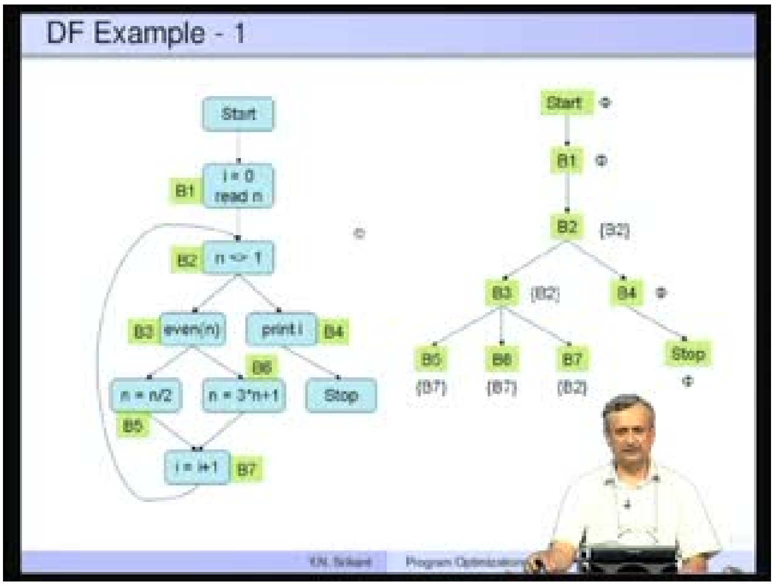
Each appearance of the variable v in the original program or a phi function in the new program has been replaced by a new variable v i, leaving the new program in SSA form. In other words, renaming of variables in this trivial phi functions and otherwise have been performed already. So, the values of the new variable and the old variable must match. That is what the third condition would say.

(Refer Slide Time: 03:28).



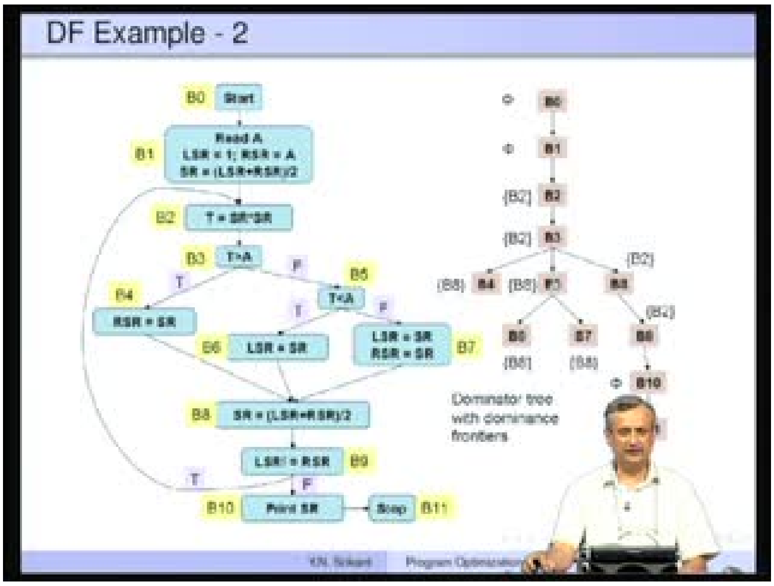
Condition 1 also says - the assignments to the phi functions will also qualify as assignments to the same variable. Therefore, they may in turn introduce more phi functions. The dominance frontier really tells you exactly where phi functions are to be included including the recursive nature of this condition 1.

(Refer Slide Time: 03:58).



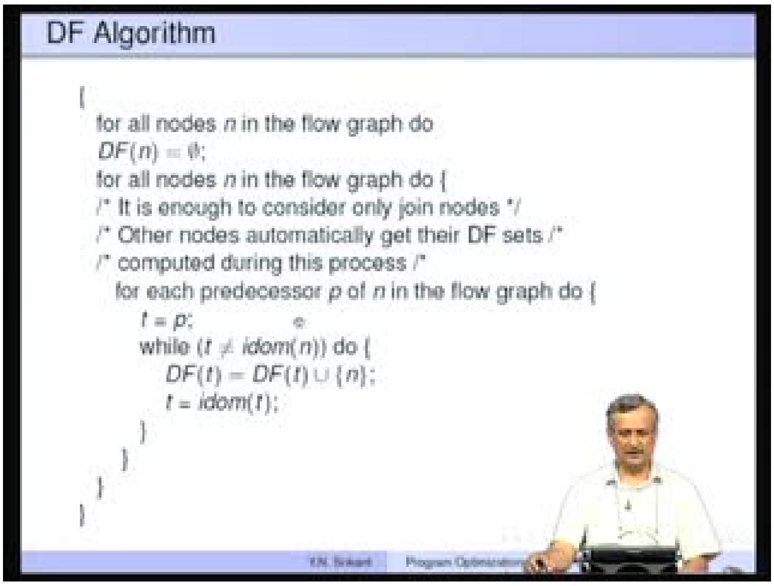
Here is an example of the computation of dominance frontier. Intuitively, dominance frontier is the set of nodes for a particular node. It is a set of nodes, which are just beyond the region, where the node dominates. So, for example, this node and this node, they dominate all other nodes. For example, start dominates all nodes. So, there is no other node to which we can reach. So, it is D F function is phi. Whereas, let us look at B3. Here is B3 and then it dominates B5, B6 and B7. So, B5, B6 and B7. After passing these, the next node we come to is B2. So, B2 is on the dominance frontier of B3. This is the meaning of the dominance frontier. Its use is, it tells you exactly where phi functions are to be introduced if we consider those nodes, where the assignment statements exist.

(Refer Slide Time: 05:03)



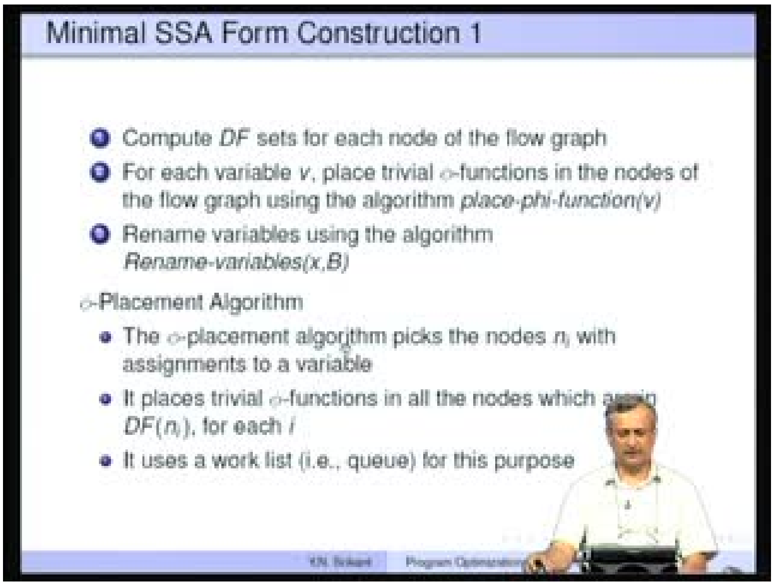
Here is another example, which is very similar. If you consider B5; so, B5 is here, B4, then B5, B6. So, B5 really dominates B6 and B7 only. So, the next node we come to is B8. That is why B8 is the dominance frontier for B5.

(Refer Slide Time: 05:25).



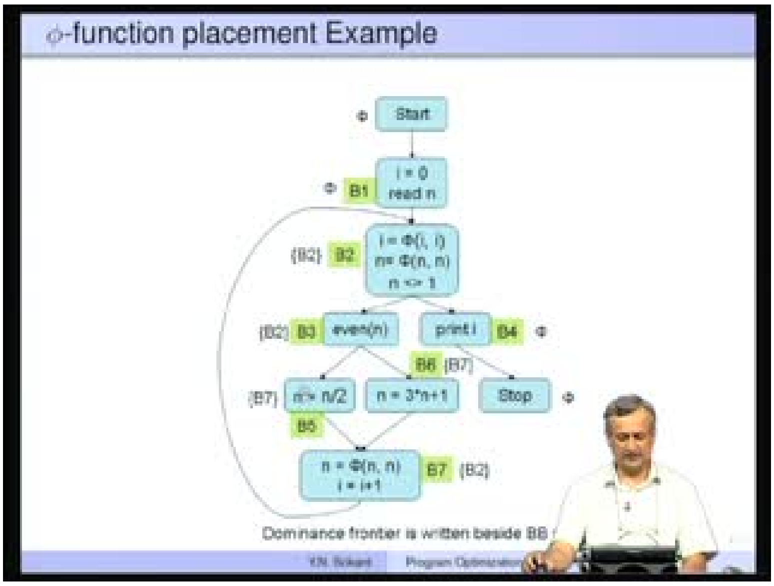
The algorithm is straight forward. We looked at this algorithm in the previous lecture already. We always start from the predecessor of a node and then go on climbing in the dominator tree until we meet the immediate dominator of that node. So, this is the algorithm.

(Refer Slide Time: 05:47).



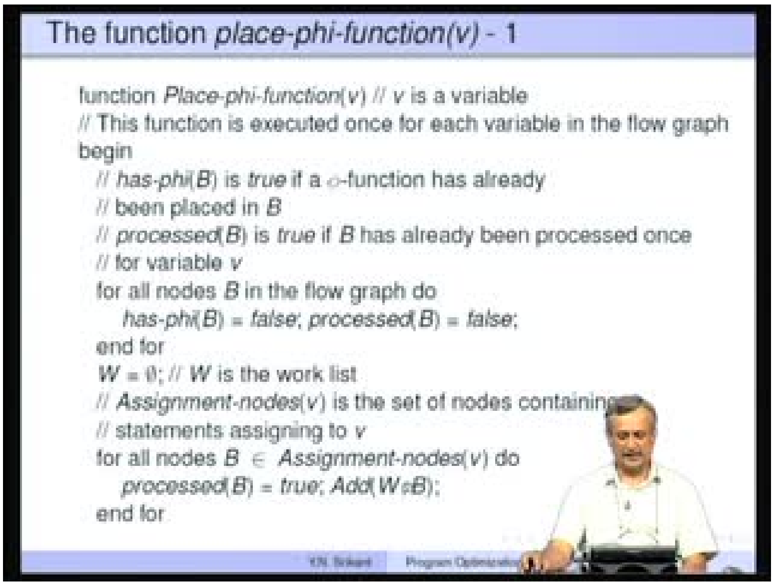
Now, the first step is placement of the phi. This requires computation of the dominance frontier of each node in the flow graph. For the phi placement algorithm, we pick the nodes n i with assignments to a variable. Then, place trivial phi functions in all the nodes, which are in the dominance frontier of that node. To do this, we use a worklist

(Refer Slide Time: 06:16)



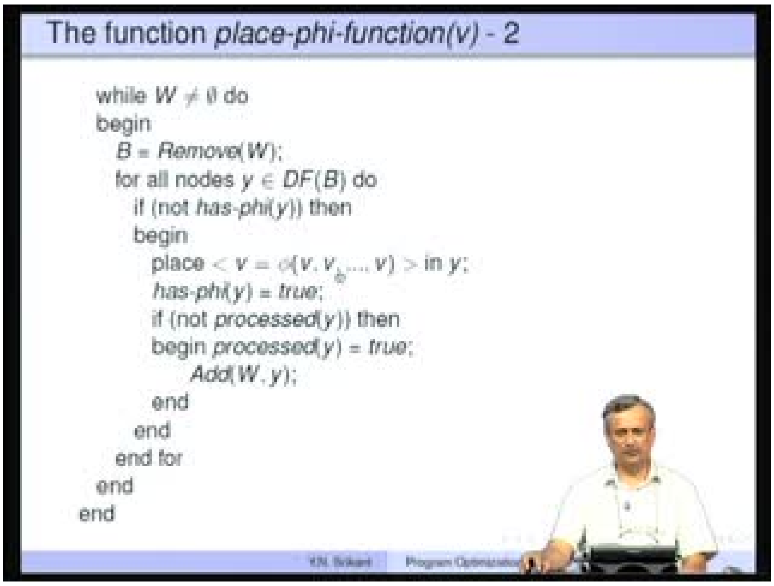
This is how a program would look like after placing trivial phi functions. The arguments are not yet renamed and the variables are not yet renamed. Everywhere it still remains as i equal to, n equal to, etcetera.

(Refer Slide Time: 06:33).



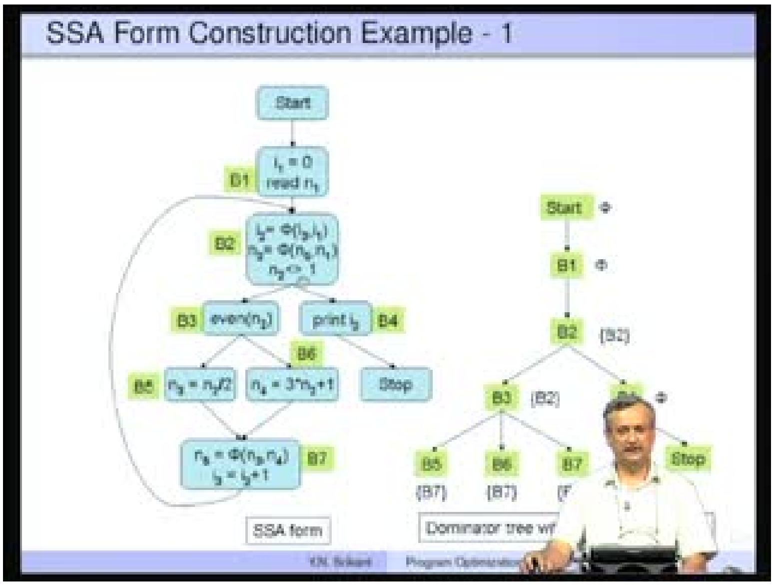
Let us look at the Place-phi-function program and quickly see how it runs. This is executed once for each variable in the flow graph. has-phi is true if a phi function has already been placed in the basic block B. processed B is true if B has already been processed once for variable v. These two are set to false and initialized for the whole program. Now, the worklist, W is made empty to begin with and we add all the assignment nodes, which are nothing but the set of nodes containing assignment statements assigning to the variable v to this particular worklist. So, we say - processed B equal to true and add that to the worklist.

(Refer Slide Time: 07:30).



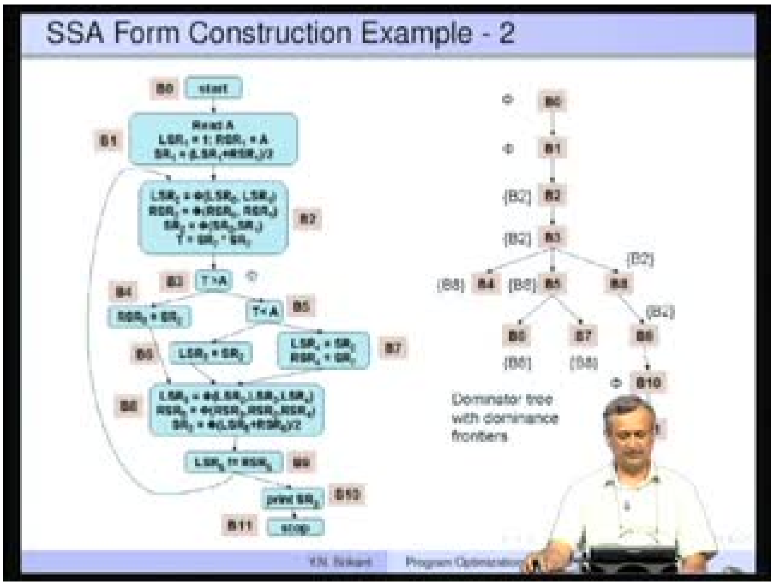
Now, in the loop, we remove a node from the worklist. Then, take the dominance frontiers of that particular node. If a phi function has not been placed in that particular node y, we place one and then make has-phi true. If the new node has not been processed yet, then we add it to the worklist and set process as true. So, this is the one, which takes care of the recursion in and make sure that addition of new phi nodes, which may result in addition of some more phi nodes is taken care of.

(Refer Slide Time: 08:17)



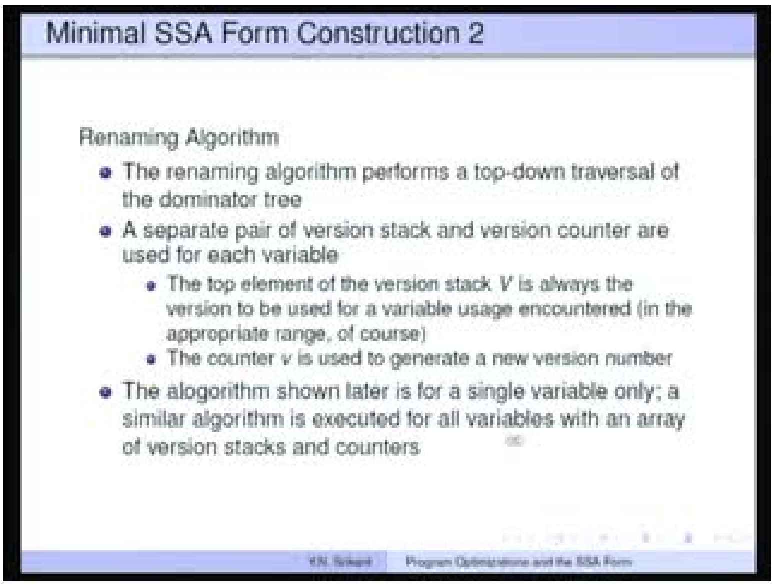
After the phi construction process is over, we need to rename the variables. For example, this is the final form that we need to produce i 2 equal to phi of i 3 comma i 1, etcetera.

(Refer Slide Time: 08:34)



Similarly, in this example as well. So, we are going to refer to these examples as and when necessary.

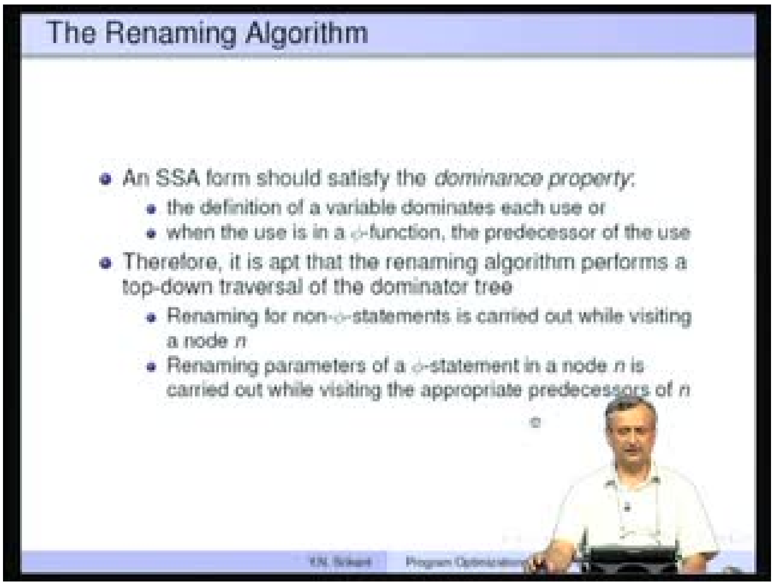
(Refer Slide Time: 08:41)



How does the renaming algorithm work? First of all, the renaming algorithm performs a top down traversal of the dominator tree. It does not travel along the flow graph, but it traverses the dominator tree. Whenever it goes to a particular node in the dominator tree, it processes that particular node in the flow graph. It uses a pair of version stack and version counter. So, this is one pair. For each variable, you have a pair of this kind. The top element of the version stack V is always the version of the variable that we have to use. In other words, there will be versions of variables V 1, V 2, V 3, V 4, etcetera. So, as we get a new definition for the variable in the program, we are going to use a new version for that particular variable. However, the reason why we require the stack is that the variable, which has been renamed, must be used for all the uses of that particular definition. So, it is not enough to just rename the definitions, but we also need to rename the uses and that must be done for the particular definition and its uses. So, the top element is always the version to be used for a variable usage encountered in the appropriate range, of course,

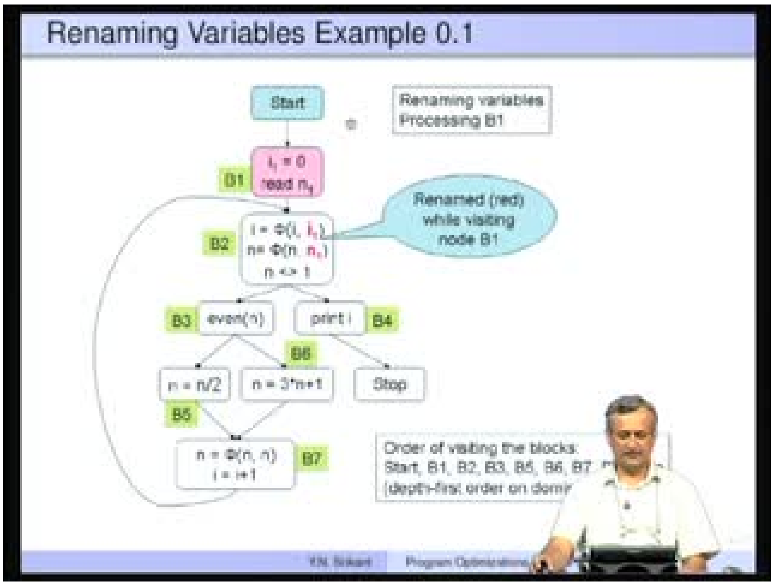
Here the counter V is used to generate a new version number. That is it. We are going to show the algorithm for a single variable, but a similar algorithm is executed for all the variables. We just have to use an array of version stacks and array of counters.

(Refer Slide Time: 10:37)



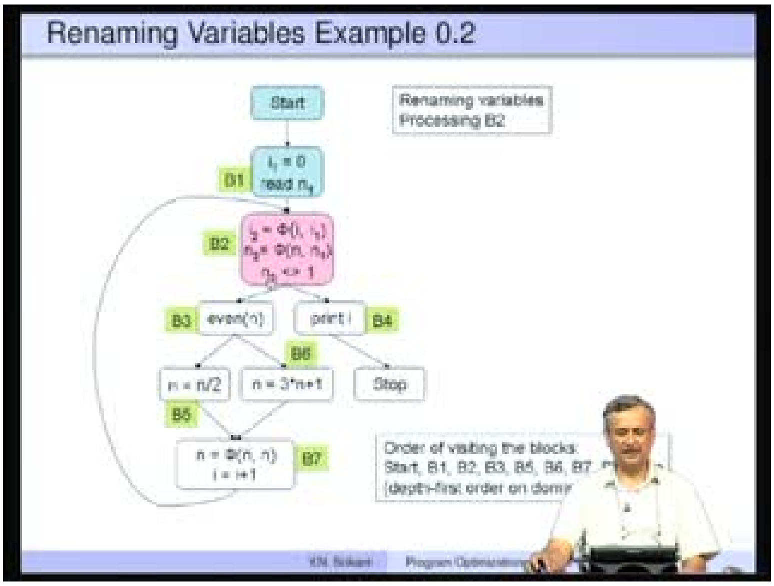
There is something very important here. First of all, we still have not explained why we need to do a top down traversal of the dominator tree. The important property that an SSA form should satisfy is called the dominance property. The definition of a variable dominates each use. So, all the uses of a variable are dominated by… Those nodes are dominated by the appropriate definition; otherwise, if we are looking at a phi function, then the definition of a variable dominates the predecessor of the use. So, this is the way it is. This is called as dominance property. Because of this, it is apt to say that the renaming algorithm performs a top down traversal of the dominator tree. How does it do it? Once we want to actually process the uses after we meet the definition and because the definition dominates all its uses, we must process the definitions first. We can do it by traversing the dominator tree from the top. So, renaming for the non-phi-statements is carried out while visiting a node, particular node n. Whereas, renaming parameters of a phi-statement in a node n is carried out while visiting the appropriate predecessors of n. This will become very clear now as we go along.

(Refer Slide Time: 12:15)



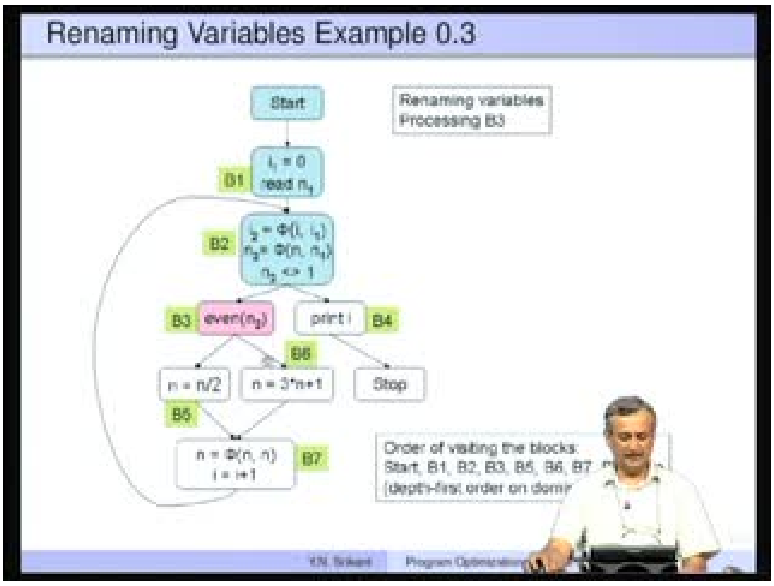
Let me show you an example first and then go back to the algorithm itself. Let us run through an animation of this program. The start node will be processed to begin with and then it leads to the basic block B1. In the basic block B1, we first rename the definition i 1. Then, read n 1 is also a definition. So, that is also renamed. Then, when we are visiting B1, we also have to look at the phi functions in the successors of this particular node, B1. So, that is in B2. So, we are going to rename the appropriate parameter corresponding to i 1; that is, the second one because it is the second arc that is coming into the node B2. So, this i from the trivial phi function is renamed as i 1 and this n is renamed as n 1. That is all, nothing else happens during the visit to B1 apart from renaming i 1 and n 1 here.

(Refer Slide Time: 13:21).



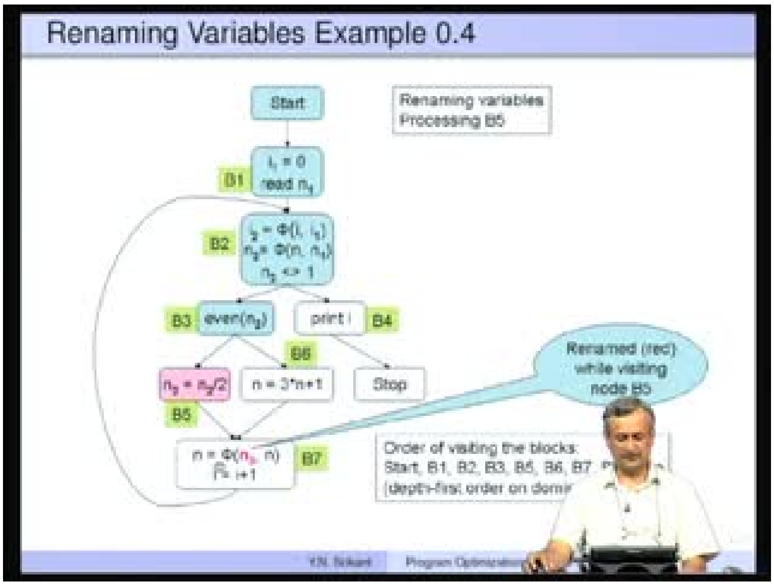
Next, B1 dominates B2. The next node we visit is B2. Here we are renaming i 2 and n 2. These are the two definitions; these are two new definitions of i and n respectively. Nothing happens as far as the first two parameters of the phi function, they remain as i and n. These will be renamed when we are visiting the node B7, which actually has an incoming arc to B2. So, this is a usage of n (Refer Slide Time: 13:53). This is supposed to be n 2. So, we make it n 2. That is all there is to it.

(Refer Slide Time: 13:59)



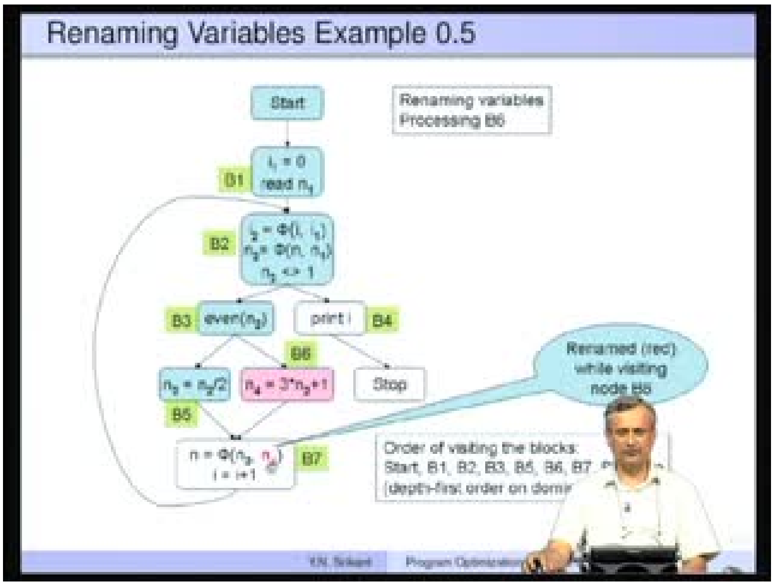
Then B2 dominates B3. So, we go to this. This is a usage. The appropriate obviously, the definition is n 2. So, this is named as n 2. This is where the version stack comes into picture. The top most entry will have n 2 and n 1 is below that.

(Refer Slide Time: 14:21)



After B3, we come to B5. It is the first edge kind of a traversal. So, when we rename n 3… Before that, we rename n 2 because n 2 is a usage corresponding to the old variable n 2. Now, there is a new definition n 3. Now, n 3 is feeding into B7 and that is the first edge. So, the parameter of the phi function in B7 is renamed here as n 3.

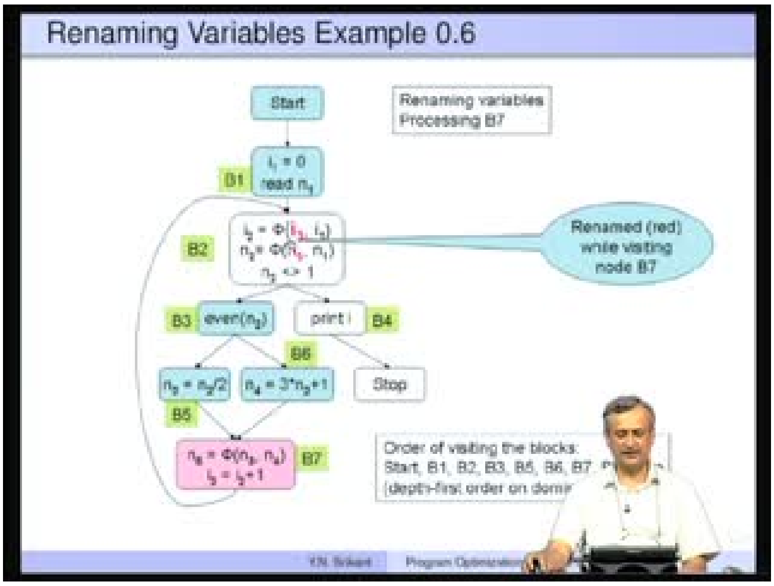
(Refer Slide Time: 14:51)



Then, B6 is traversed and that would rename n 2 and n 4 appropriately. See here that this is n 3, but for this, it will be a new definition called n 4, but the usage here comes from this n 2. So, this is still n 2.

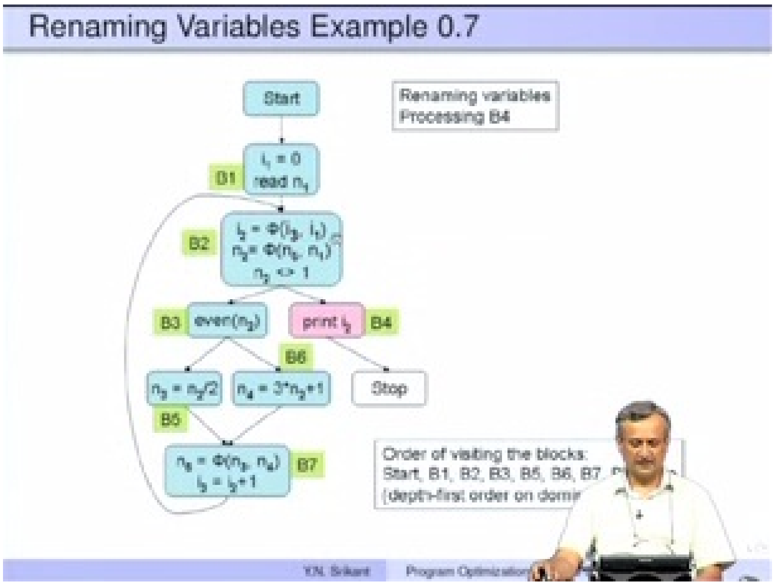
Here (Refer Slide Time: 15:10), we have the incoming edge as the second one and n_4 is renamed here.

(Refer Slide Time: 15:16)



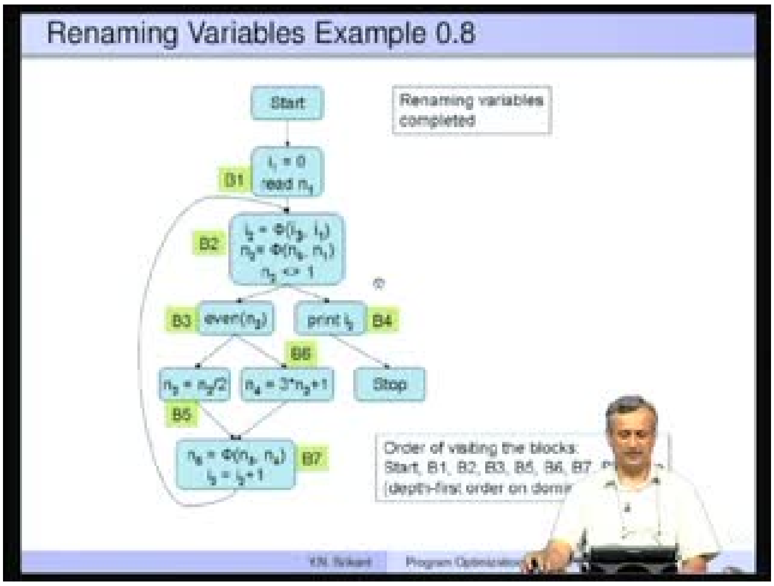
Then, we go to B7 and here we rename n 5 and i 3. Then, i 2 will be renamed based on this particular definition i 2. The original stack being different, this i 2 would have been in a different stack altogether and that is easy to use here, and this edge is coming into B2. So, the first 2 parameters here are renamed as i 3 and n 3.

(Refer Slide Time: 15:46)



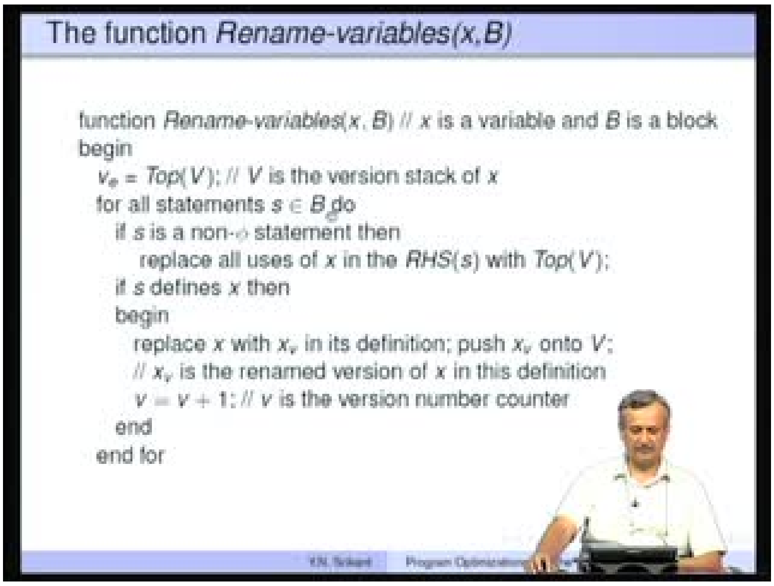
Then, we process B4 and rename this as i_2.

(Refer Slide Time: 15:50)



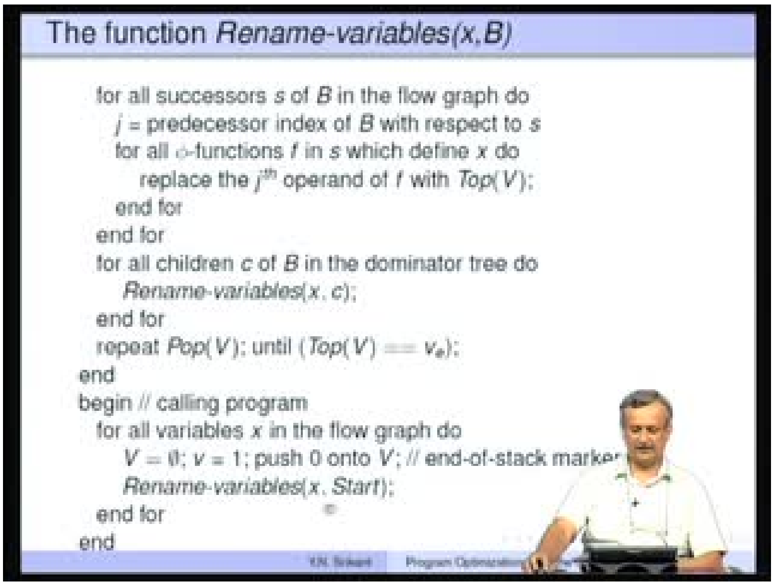
Finally, we stop. So, this is our sequence.

(Refer Slide Time: 15:59)



Let us go through the algorithm and see how it works. Function Rename-variables. Here is the top of the version stack. The version stack to begin with will be empty. So, this will act as the kind of… If the stack V is empty, then this will be the bottom of stack marker. If this is reached, we are going to stop; otherwise, somewhere in the middle, this will be the most current version that we want to use. For all statements s in B do.

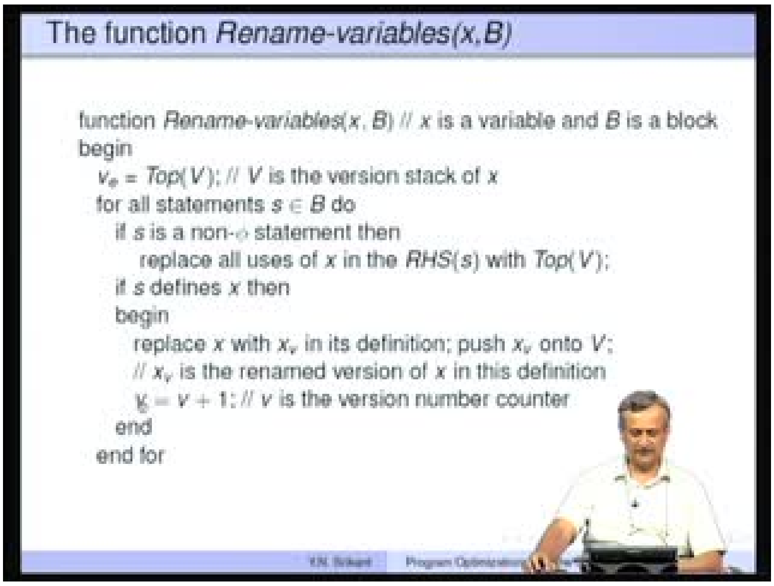
(Refer Slide Time: 16:37)



Let us look at the main calling program to begin with. For all variables x in the flow graph do. So, the version stack is empty, version counter is initialized to 1, and we push

0 on to V. This is the end of stack marker. Then, we call Rename-variable x comma Start.

(Refer Slide Time: 16:54)



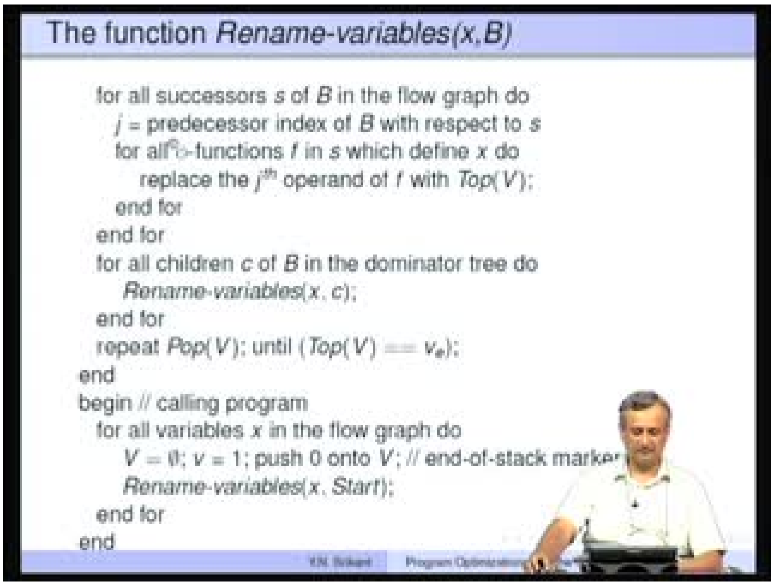
That is why we come here. The stack is empty to begin with. Now, for all the statements s in B, we have a basic block. Right now, if it start, there will be no statements in it. So, none of these will be executed. For the first start block, nothing is executed here.

Nothing is executed here (Refer Slide Time: 17:13) and none of these are executed. Then, we come to this - for all children c of B in the dominator tree do. Call Rename-variables. So, we are going to call (Refer Slide Time: 17:24) on B1. Start did nothing. So, we go to B1.

Now, we come here (Refer Slide Time: 17:31). We have B1 and the stack still contains only the top of stack marker, but there are statements s in B. So, s is a non-phi-statement here (Refer Slide Time: 17:44) – i 1 equal to 0 and read n 1 are both non-phi-statements. Replace all uses of x in the RHS with top of V. There are no RHS variables to be renamed here (Refer Slide Time: 17:56). There is nothing at all. Therefore, nothing is done here (Refer Slide Time: 18:01). If s defines x; basic block B1 has two assignment statements. Read is also an assignment. So, i equal to 0 and n equal to 0. So, replace x with x v in its definition. So, v is 1. So, we are going to have i 1 equal to 0 and push x v on to V.

Now, the new variable, which is generated, the version is pushed on to the appropriate version stack. Remember that there is one version stack for each variable. So, i 1 is pushed on to i's stack and n 1 is pushed on to the n's stack. Now, increment the version counter appropriately. So, i's version counter and n's version counter are incremented as far as v 1 is concerned.

(Refer Slide Time: 18:47)



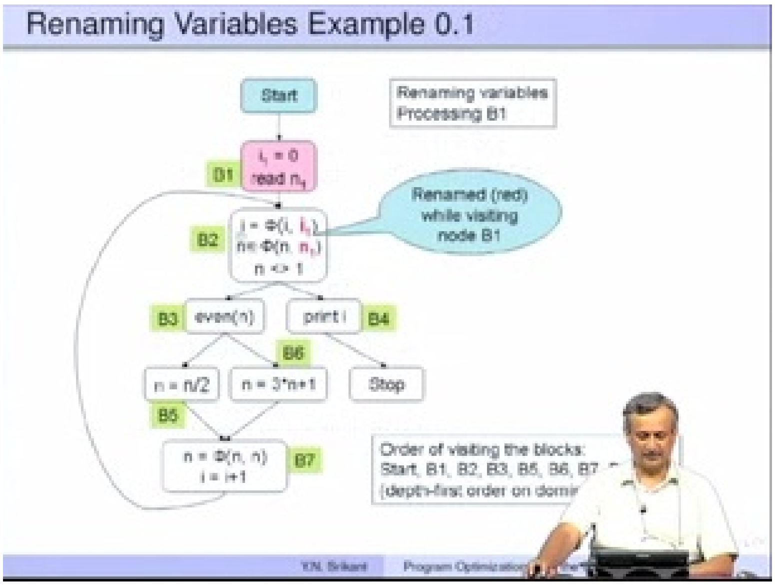
Now, look at the successors of other basic blocks (Refer Slide Time 18:51). So, this is the successor B2. It has a phi function. That is what we want to see. So, j be the predecessor index of B with respect to s. That is, we are looking at which particular arc this is (Refer Slide Time: 19:05) the first arc or the second arc. That is the j that we are considering.

For all phi functions f in s, which define x do. So, we are looking at the phi functions in this successor (Refer Slide Time: 19:18) i and n. So, replace the j th operand of f with top of V. So, appropriately here we are going to replace this (Refer Slide Time: 19:27) with i 1 and this with n 1.

So, the replacement for phi functions is over. Now, this process continues with the other children of the basic block V. So, in this case (Refer Slide Time: 19:43), from B1, we call B2 and then B3, etcetera as I explained.

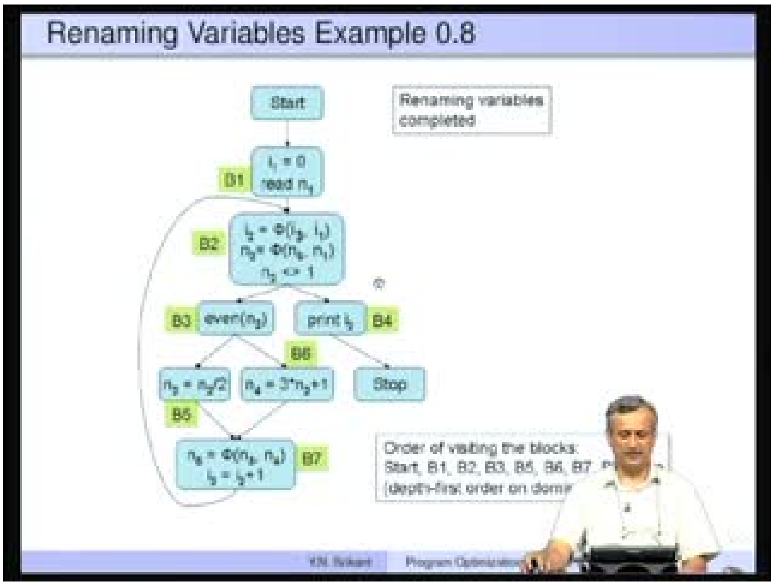
Once all the children have been exhausted, the version stack is popped until it reaches the element V, which we actually write here from the top (Refer Slide Time: 20:05) of the stack. So, we enter the function with a particular version variable and then we also exit that function when we reach the same configuration of the stack. This is how renaming of variables happens.

(Refer Slide Time: 20:24)



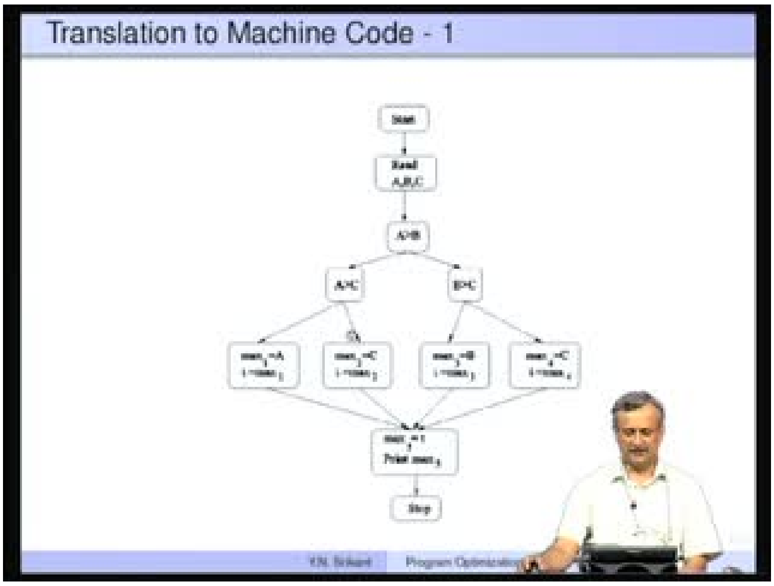
To summarize, within a block, we first look at the RHS and rename variables. Then, we look at the LHS and rename the variables. Then, we look at the successors of the basic block and rename the phi function parameters, appropriately. So, this whole thing happens during a traversal of the dominator tree.

(Refer Slide Time: 20:50)



This is the final product after renaming. So, this would have been taken care of.

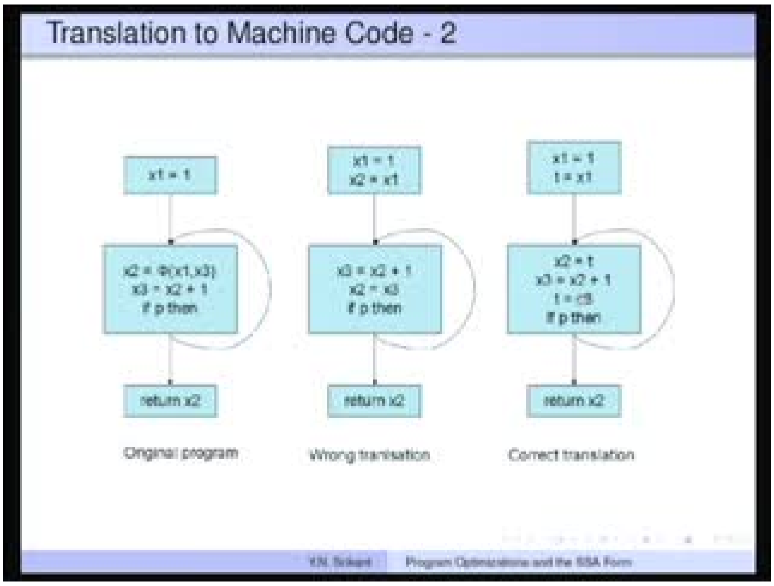
(Refer Slide Time: 21:00)



The next issue that we need to worry about before we look at optimizations of various kinds is that the phi functions cannot be executed on a machine. So, that is a concern. We must translate the phi function to appropriate machine code. How do we do that? There is a fairly straight forward scheme. If you recall, we would have had a phi function here - max phi equal to phi of max 1 max 2 max 3 max 4. That is what we would have had.

Now, we introduce a temporary t, copy max 1, max 2, max 3, max 4 in the appropriate predecessors to t, and then say max phi equal to t. So, this scheme will always work. We need to apply another set of transformations on it later on. For example, copy propagation. So, max 1 equal to a and t equal to max 1. So, this becomes (Refer Slide Time: 22:17) t equal to a and so on and so forth. Apart from that, this scheme will work.

(Refer Slide Time: 22:26)



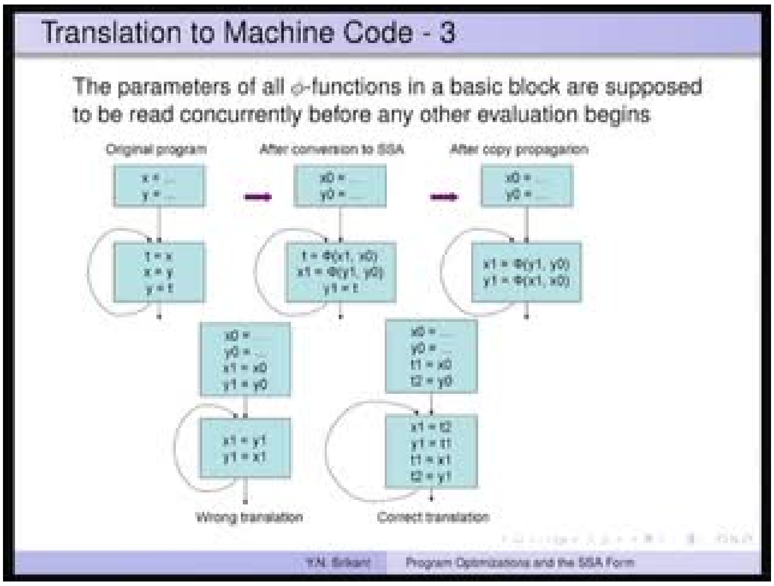
Some other scheme, which one can think of sometimes does not work. Let me show you an example. Here is a program within the SSA form x1 equal to 1, x2 equal to phi of x1 comma x3, x3 equal to x2 plus 1, and then if p then there is branch; otherwise, go out.

Instead of generating a temporary t and then saying along this path, t equal to x1 and then along this path, t equal to x3 and so on and so forth, let us try to be cleverer and then straight away take this variable x2 and assign it x1 here (Refer Slide Time: 23:12). Instead of t equal to x1 and then x2 equal to t here we said x2 equal to x1 directly. So, this statement (Refer Slide Time: 23:22) is not needed any more because we are making an assignment to x2 equal to x1 here. We will have to make an assignment x2 equal to x3 just after this statement x3 equal to x2 plus 1. Why? That is because this arc will be taken only after one iteration. So, let us go through one iteration, execute x3 equal to x2 plus one, and then say - x2 equal to x3, but this is a wrong translation.

This would not do at all. This is x2 equal to x1 (Refer Slide Time: 23:55). Then, you have x3 equal to x2 plus 1, then you have x2 equal to x3, and then you go back. So, this gives an incorrect translation because the first time you come here (Refer Slide Time: 24:11), x2 should have taken the value x1. If we do not iterate and then return, it would give you x2. Whereas, here we said x2 equal to x1. So, x3 equal to x2 plus 1. Therefore, now, x2 equal to x3 makes the value as x2 plus 1. Whereas, in the previous program, it would have been x2 equal to x1, if we did not iterate at all.

The value returned here is one more than what it would be returned here (Refer Slide Time: 24:42), if we did not iterate through this particular program. In other words, even if we iterate, it will always have one more than the previous versions. So, this is a wrong translation. The correct translation is exactly the way I showed you - take a temporary, assign x1 to it and then take a temporary, assign x3 to it. So, here (Refer Slide Time: 25:09) x2 retains x2 equal to t. So, here x3 equal to x2 plus 1 and then t equal to x3, but if we go through without any iteration, we still return x2, which is the old value. So, the new value is not used immediately. This actually is the correct translation.

(Refer Slide Time: 25:34)

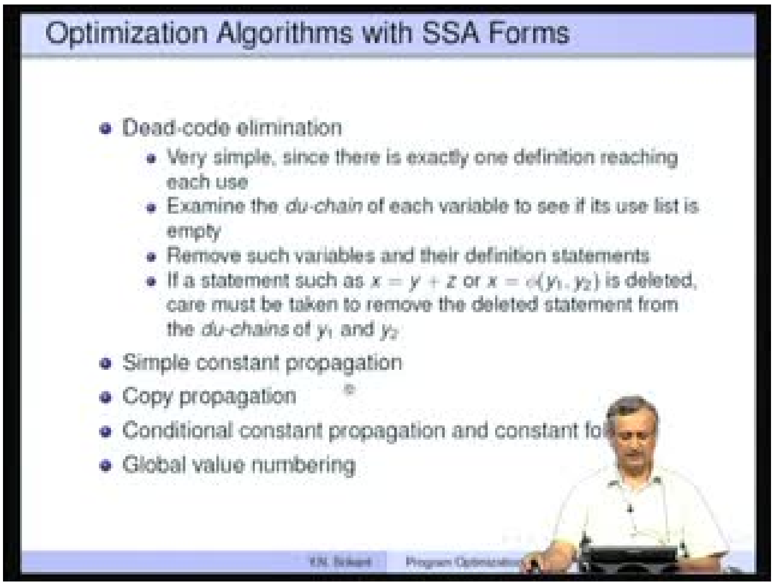


Let me show you another example of what can go wrong. We have the original program here - x equal to, y equal to, and then we are swapping. We simply swap in a loop again and again and again; that is all; t equal to x, x equal to y, y equal to t. We convert it to SSA. Now, t actually gets a phi function because this x here can be from here or it could be from here. So, we have x naught comma x1. This is x naught and y naught. Then, x1 also gets a phi function here because there are two values of y coming in: one through this and another through this. So, phi of y naught comma y1 and then y1 equal to t. This is the new y1. So, y1 equal to t.

With this, we can do a copy propagation because t can be replaced by phi of x naught comma x1. So, we have x1 equal to phi of y naught comma y1 and y1 equal to phi of x naught comma x1. This is a correct SSA form; no problem. However, if we try to hasten and then say - let me do assignment to x1 and y1 right here (Refer Slide Time: 26:55) and then immediately after wards at this point, like in the previous case, we get a wrong answer – x naught equal to, y naught equal to, x1 equal to x naught, y1 equal to y naught, x1 equal to y1, y1 equal to x1. So, this obviously gets the same value into x1 and y1. So, this is wrong. It is not swapping at all. So, we need to introduce a temporary t1 equal to x naught, t2 equal to y naught, x1 equal to t2, y1 equal to t1, t1 equal to x1 and t2 equal to y1. So, this is a correct translation. In this case, you cannot really do too much of copy propagation; a little bit yes, but not too much.

y1 equal to t1 and t2 equal to y1 will become t2 equal to t1, but beyond that not too much of copy propagation will happen here. However, this is the correct translation. In other words, one has to be very careful and introduce temporaries in the predecessors of the phi function so that appropriate translation takes place. Then, leave it to the optimizer to remove the copies if it can.

(Refer Slide Time: 28:12)

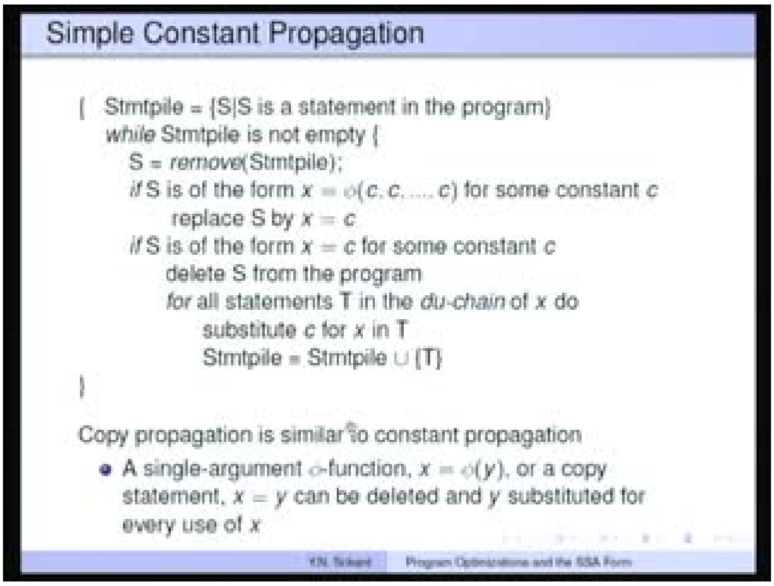


What are the various optimizations, which are possible with SSA forms? Let us look at some of them. The first one is dead-code elimination. Dead-code elimination is extremely simple. Why? You have exactly one definition reaching each use. So, if the du-chain of a variable is empty, then there are no uses of that particular variable. Therefore, the definition has nothing, no effect, nothing to do. So, examine the du-chain of each variable to see if its use list is empty. If it is so, remove such variable and their definitions statements as simple as that.

If a statement such as x equal to y plus z or x equal to phi of y 1 to y 2 is deleted, what happens? It is not that you can just delete the statement as such, but then there are definitions of y 1 and y 2. For example, x equal to y plus z may not have any use. In other words, I have not used x later on at all, but what happens to the d u chain of y 1 and y 2. So, in that, this statement will be present. So, we have to actually remove those statements from the du-chains of y 1 and y 2 or the du-chain of x. So, we must take care to do that as well; otherwise, there would be a statement, which is deleted, but is present in the du-chain. Some processing would actually issue some error.

We can do simple constant propagation, we can do copy propagation, we can do conditional constant propagation, constant folding, global value numbering, etcetera. Let us look at each of these in sequence.

(Refer Slide Time: 30:29)



We have already seen enough constant propagation. This is probably a much simpler version of constant propagation. The more complicated version called the conditional constant propagation; we will discuss very soon. In simple constant propagation, you are only going to look at statements of the form x equal to c. So, wherever x occurs, we will try to replace it by c. That is what we want to do.

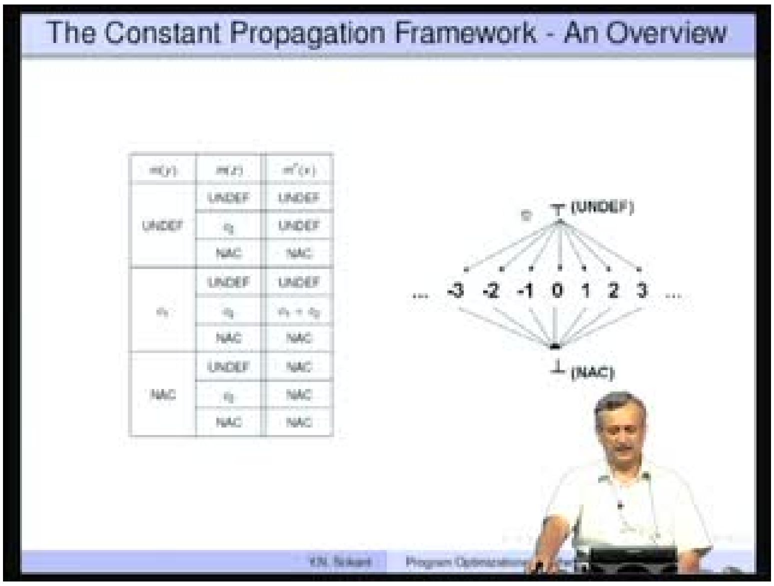
For this, again we are going to use a queue. So, this is called as a statement pile (Refer Slide Time: 31:11). This is initialized to S such that S is a statement in the program. So, you put all the statements in the program into the statement pile. While the statement pile is not empty, take a statement if S is of the form; it is of the form of a phi statement, x equal to phi of c comma c comma c. In other words, all the parameters of the phi statement are constants. These need not happen right in the first instance. It can happen after some constant propagation is carried out. That means, the same constant value is arriving through each of its edges; in preceding edges. So, we can replace this comfortably by a statement x equal to c; no harm done.

If S is of the form x equal to c for some constant c, delete the statement from the program. For all statements in the du-chain of x, substitute c for x in the statement T and then add T to the statement phi. In other words, what we do is - we take the statements in the du-chain, examine it. Obviously, there will be some usage of x there. We remove that x from the statement, we put c in its place; the constant c. So, we have done constant propagation replacing x by c. Now, we need to process that statement as well because that may lead to further replacements and the things of that kind. So, we keep that on the statement pile and go ahead; that is it.

This is a very simple constant propagation. The constant flows down the program. Now, the point is - each statement in which a variable is replaced by a constant may actually in turn induce other statements to become targets for constant propagation. That is why this is necessary. So, first time you visit a statement, there may be nothing to do. It may be of the form x equal to y plus z, but then it is possible that there is y equal to c and z equal to c. y plus z eventually becomes a constant. This particular simple constant propagation is not very effective because we are not even evaluating expressions here. Even it becomes y plus z and it is c 1 plus c 2, we are not evaluating here. So, the next version of constant propagation called conditional constant propagation will do not only this, but a little more. We will see that soon.

What is copy propagation? It is very similar to constant propagation. So, a single argument phi function such as x equal to phi y or a copy statement x equal to y, we can delete it and y is substituted for every use of x. So, in x equal to c, wherever we had x, we substituted by c. Here wherever we had x, we substituted by y (Refer Slide Time: 35:03). So, that is copy propagation; very simple copy propagation.

(Refer Slide Time: 35:08)



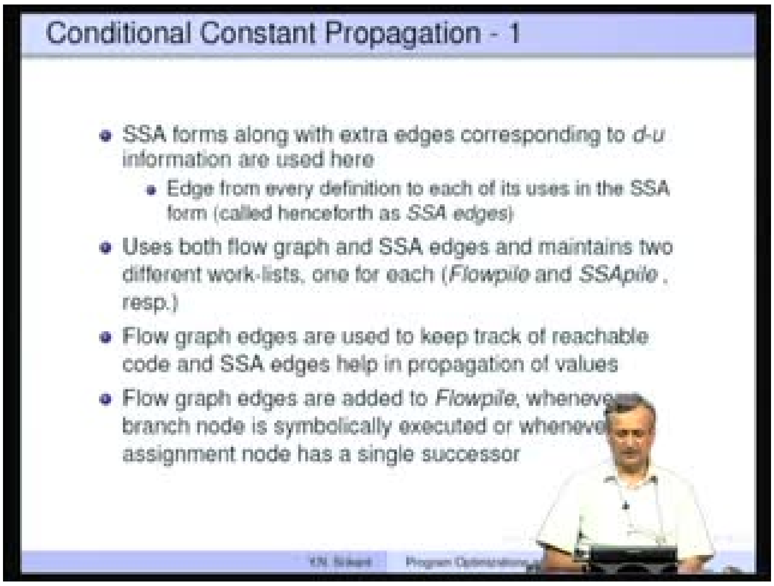
Now, we come to conditional constant propagation. Let us do a recap on the constant propagation framework that we studied some time ago. The constant propagation framework had a lattice for its variables. So, the variables could take 3 values: one was UNDEF; that is, to begin with, the variables do not contain any value. So, they are undefined. So, UNDEF. The variables also could take any of the constant values assuming they are integers - minus 3, minus 2, etcetera, or 1, 2, 3. These constant values are incomparable. So, this is the lattice that we have.

All constant values are grouped as constant. So, that is the middle abstraction. Then, the third abstract value is not a constant. So, we have determined that the variable is not a constant anymore. For example, for a particular node, the incoming predecessors give you y equal to 2 along one path and y equal to 3 along another path. Then, y cannot be a constant at all. It cannot be a constant, it can neither be 2 nor 3, or something else. In such a case, y can be given the abstract value; NAC; not a constant.

If you have a statement x equal to y plus z, then here we have listed the effect of the transfer function for x equal to y plus z. As we said, in the previous lectures, the product of these lattices - one for each variable is the domain of data flow values for the constant propagation framework. Here it suffices to see the transfer function effect. Suppose y takes the value either UNDEF or constant or NAC. So, that is what m y gives you. y would be the actual value, but m y gives you the abstract value.

Depending on what m z is, UNDEF, c2, or NAC, m prime x, the new abstract value for x would be either UNDEF or NAC. In other words, unless all of them are constants, c 1, then c 2; x will not be c 1 plus c 2. In other words, we start from the top, we can only go downwards and we never go upwards. Once we have determined that a variable is not a constant, its value can never change, but if the variable had a undefined value, it could become defined and carry a constant. If it had a different constant value along two paths, it could become not a constant in some join node. So, you can only go downwards. That is what is shown here (Refer Slide Time: 38:20). If it is UNDEF, then it is UNDEF, UNDEF, or NAC. If it is constant, then UNDEF, c 1 plus c 2, or NAC, but if it is NAC then it can be nothing but NAC. So, we do not go upward in the lattice. So, this is the constant propagation framework that we had already studied. We are going to use the same frame work for our conditional constant propagation as well.

(Refer Slide Time: 38:46)



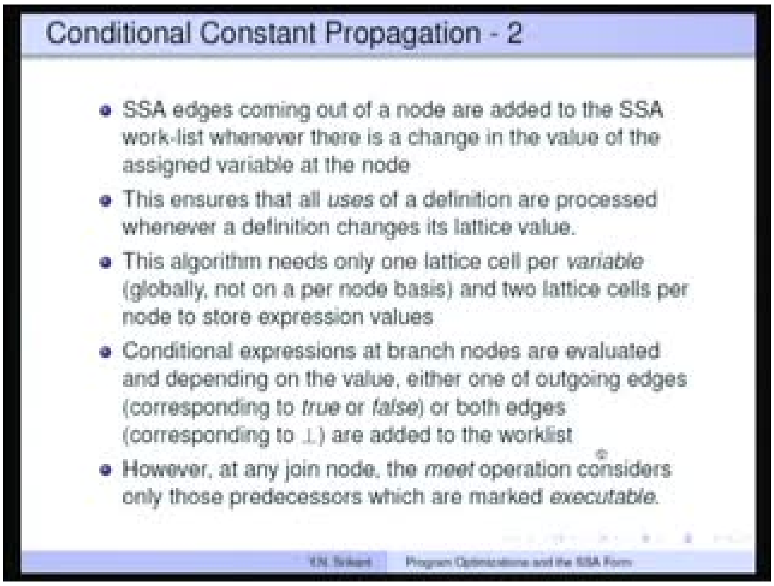
SSA forms along with extra edges corresponding to the d-u; definition use information are used here. Edge from every definition to each of its uses in the static single assignment form. Hence forth, called SSA edges; is used here.

Actually, we use both SSA edges and flow graph edges. We are going to use two different work-lists or queues, one for each flowpile is a queue corresponding to flow graph edges and SSA pile is a queue corresponding to SSA edges. So, I must point out a difference here in the simple constant propagation. If you recall (Refer Slide Time: 39:41), we went by basic blocks or the statements in the program, whereas in the case of conditional constant propagation with SSA form, we are using edges. Unless we traverse an edge, actually the node, which is the target of that particular edge, is not reachable. So, that is the important point here.

Flow graph edges are used to keep track of reachable code. As we go on, as we say each edge is visited, we can visit appropriate nodes as well. The SSA edges are used for propagation of values. So, once we reach a particular node for the first time, we visit that particular node a second time only if some value, which is feeding into that particular node changes. So, if that happens in the definition corresponding to that particular variable in the node, then the SSA edge would be responsible for the flow of this particular information. This will become very clear as we go on.

Flow graph edges are added to flowpile whenever a branch node is symbolically executed or whenever an assignment node has a single successor. This is very clear. So, if we have a single successor after finishing a particular node, the next node would be added. As I said, we are going to visit each node only once through the flow graph edges, we will be visiting a second time only if a value changes in that particular node. In the case of a branch node, we are going to evaluate the condition in that branch node and then add either the true edge or the false edge to the work-list, appropriately.

(Refer Slide Time: 41:47)



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 that particular node; not otherwise. This ensures that all uses of a definition are processed whenever a definition changes its lattice value. This is how SSA form becomes powerful. You are making sure that nodes, which change values are processed, but if we do not use SSA edges to reach that particular node, we may have to go through many other nodes in the flow graph. So, traversal of the flow graph again and again would be necessary in order to actually process that particular node.

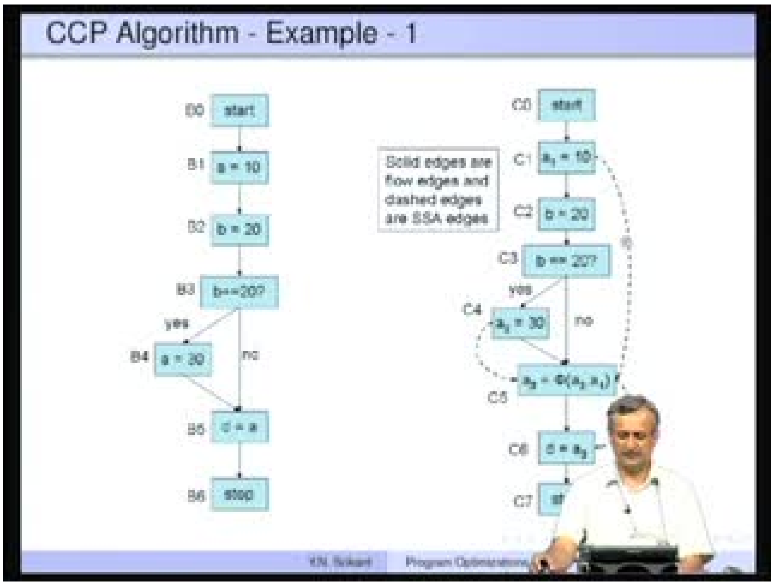
If this happens, the amount of time that is needed for the algorithm actually becomes very high. That is the advantage we have in the case of conditional constant propagation with SSA form. The time needed to process the program, do the constant propagation is much lesser than the time needed to conditional constant propagation with just the flow graph.

This algorithm needs only one lattice cell per variable and not on a per node basis. So, previous versions of this algorithm, which worked on the flow graph required actually one lattice per node per variable. So, there was too much storage necessary and it also requires two lattice cells per node to store expression values; the old and new values of an expression.

Conditional expressions at branch nodes are evaluated and depending on the value, either one of outgoing edges corresponding to true or false or both edges corresponding to NAC are added to the worklist. So, if only true part is true, only that edge is added, if only false part is holding, that edge is added; otherwise, both edges are added to the worklist.

However, at any join node, the meet operation considers only those predecessors, which are marked executable. So, this is important for a phi function because in a phi function, there are many parameters, each one corresponds to the preceding edge. So, we do not consider any of the edges, which are incoming and are not marked executable. We consider only those edges, which are marked executable. So, this makes sure that we catch more constants, some dead-code, and some unreachable code, etcetera are eliminated and so on.

(Refer Slide Time: 44:48)

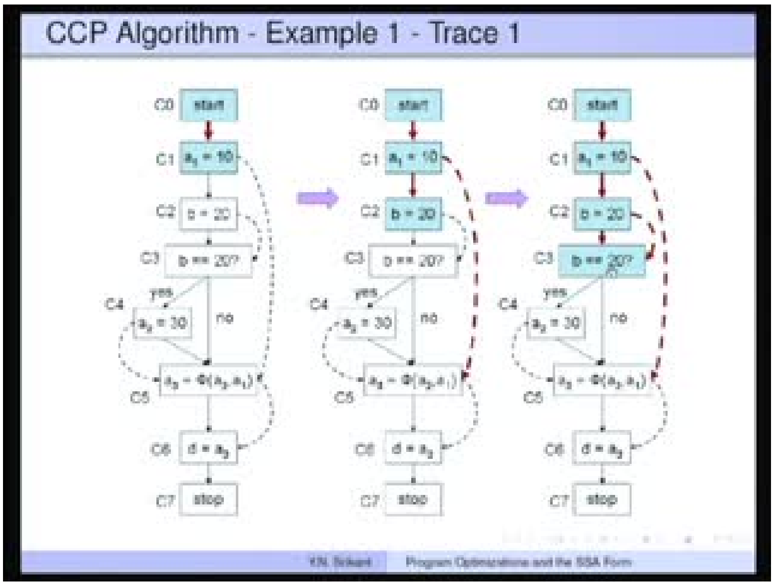


Let me give you an example, which is slightly away, but it is ok, we will come back. Here is the example. Here is the program; a very simple program. Start, a equal to 10, b equal to 20, then there is a test is b equal to 20? Yes; a equal to 30, no; we go straight. If it is after assigning a equal to 30 here, we say - c equal to a and stop. So, this is easy to comprehend. At this point, after a equal to 10 and b equal to 20, obviously is b equal to 20 is true. So, only this branch (Refer Slide Time: 45:25) will be executed with run time. Now, we assign a equal to 30. So, this branch is never executed and we come here so see d equal to a will make the value of d as 30 and then we stop. This is the original program.

Here is the SSA form (Refer Slide Time: 45:48). There are two assignments to a. So, we have a 1 and a 2. We have just one assignment to b. So, we are going to retain it as b; only one assignment to d. So, this will be retained as d. So, a 1 equal to 10, b equal to 20, then is b equal to 20? The test; then, this becomes a 2 equal to 30. Here we have a phi function a 2 along this path and a 1 along this path. Then, d equal to a 3 and then stop.

The solid edges are all flow graph edges. Now, this a 1 (Refer Slide Time: 46:35) is used here. So, this is an SSA edge. This a 3 is used here. So, this is an SSA edge. This a 2 is used here. So, this is another SSA edge. Actually we should have shown more SSA edges here, but just to avoid clutter I did not do it. So, this is b equal to 20. Is b equal to 20? There is a usage here. So, this will be another SSA edge.

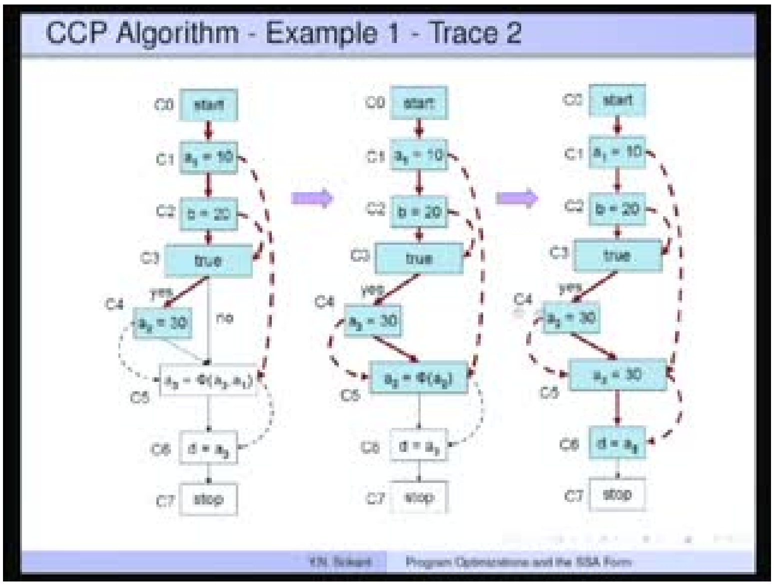
(Refer Slide Time: 47:00)



How does the CCP algorithm work? With start, nothing happens. Then, you have a 1 equal to 10. So, you do a symbolic execution. Now, the value of a 1 is from undefined, changes to 10. So, this SSA edge actually is added to SSA pile. Then, we go to this (Refer Slide Time: 47:32). So, the statement b equal to 20, when executed will change the value of b from undefined to constant 20. This will change the value of b. The lattice value changes from undefined to constant. So, this SSA edge is also pushed on to the stack.

Now, after that, we come here (Refer Slide Time: 47:57). This edge was added to the flowpile to begin with and then we added this edge to the flowpile. Now, we added rather the SSA pile. So, this edge was next added to the flowpile. Now, this edge was also added to the SSA pile, but it suffices to say that these two SSA edges have no effect at this point. Why? When we look at this parameter a 1 (Refer Slide Time: 48:26), this node has both its preceding edges as non-executable, they are not marked executable. So, there is nothing we can do here. This particular SSA edge has no effect because before reaching this node this cannot be used. This is because, this edge would have been marked not yet executed, but once we execute it, this SSA edge (Refer Slide Time: 48:53) is of no use again because the value does not change any further. We have processed it once and we are not going to process it again unless b equal to 20 changes to b equal to 30, or something like that.

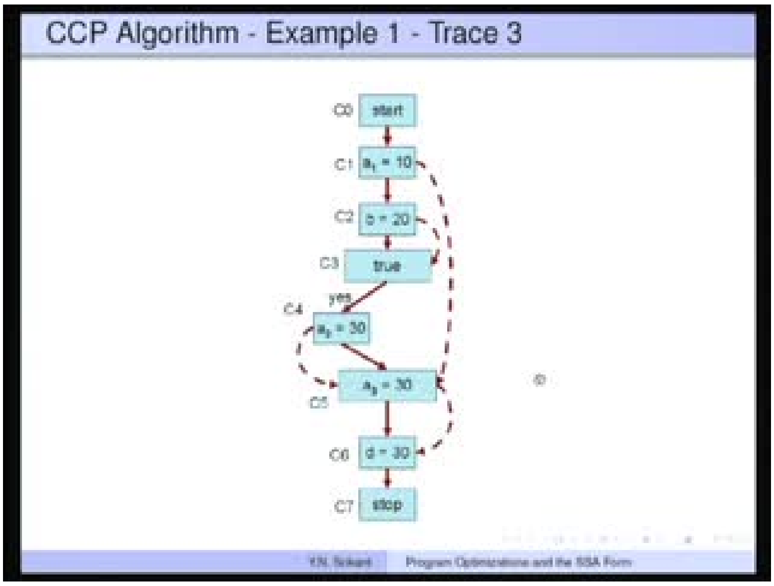
(Refer Slide Time: 49:07)



This is the next step. Then, we take this edge. Why? b equal to 20 is true (Refer Slide Time: 49:14). b has a value 20; symbolic execution. Checks 20 equal to 20. So, this is true. So, only the true edge can be taken. We take the true edge. This will be put on the flowpile. Then, in the next step, we check the assignment a 2 equal to 30. The value of a 2 changes from undefined to 30. Now, again the value has changed and this goes on to the SSA pile. We come to this via this particular edge. So, this is put on the flowpile. When we take out that edge, this node will be executed.

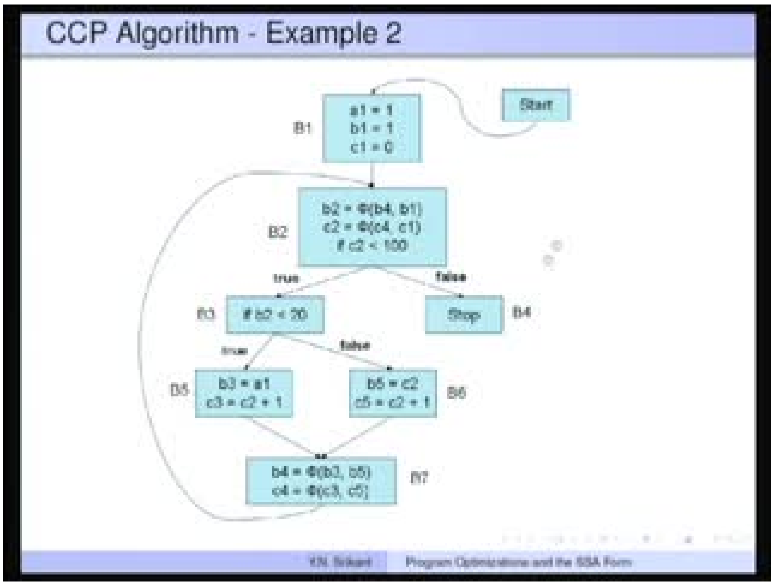
This particular node, when we come here (Refer Slide Time: 50:06), please observe that this particular edge is not yet marked executable. So, we are not going to actually consider this edge, when we consider the phi function here. We are going to consider only this particular edge. So, this will be ignored. That is why the node is returned as a 3 equal to phi of a 2. Once we evaluate this phi, it is very easy to see that this edge was taken. So, it is a 2 and value of a 2 is 30. a 3 equal to 30 is the statement to be executed next.

(Refer Slide Time: 50:40)



Once we do that, a 2 equal to 30 is executed and then d equal to a3 (Refer Slide Time: 50:47) becomes d equal to 30 and then we stop. The SSA edges in this particular example do not play any significant role.

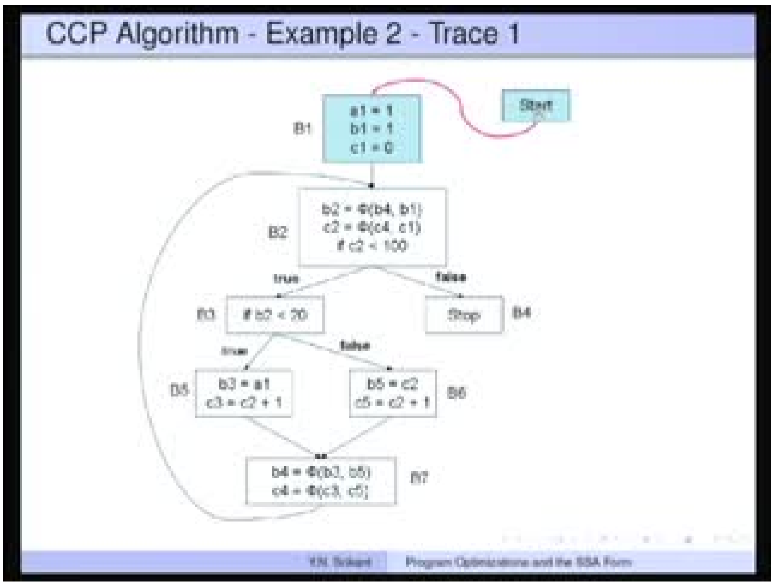
(Refer Slide Time: 50:59)



However, in the second example, they are going to play a very significant role. Let me show you the second example also and then we will go on to the algorithm itself.

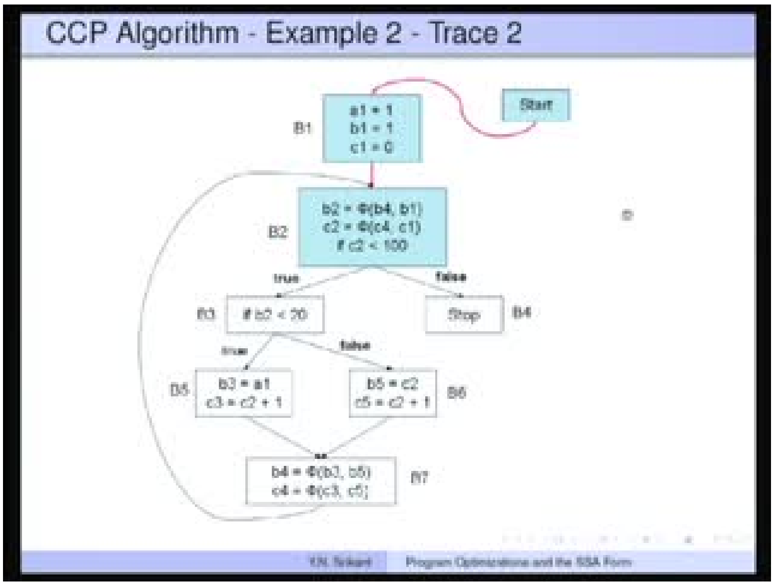
This is a slightly more complicated example. We have a1 equal to 1, b1 equal to 1 and c1 equal to 0 here. Then, we have b2 equal to phi of b4 comma b1, c2 equal to phi of c4 comma c1, and if c2 is less than 0, etcetera. If b2 less than 20, b3 equal to a1, etcetera. False; we come to b5 equal to c2. Then, these two merge (Refer Slide Time: 51:38) and there is a loop. So, this is our example.

(Refer Slide Time: 51:41)



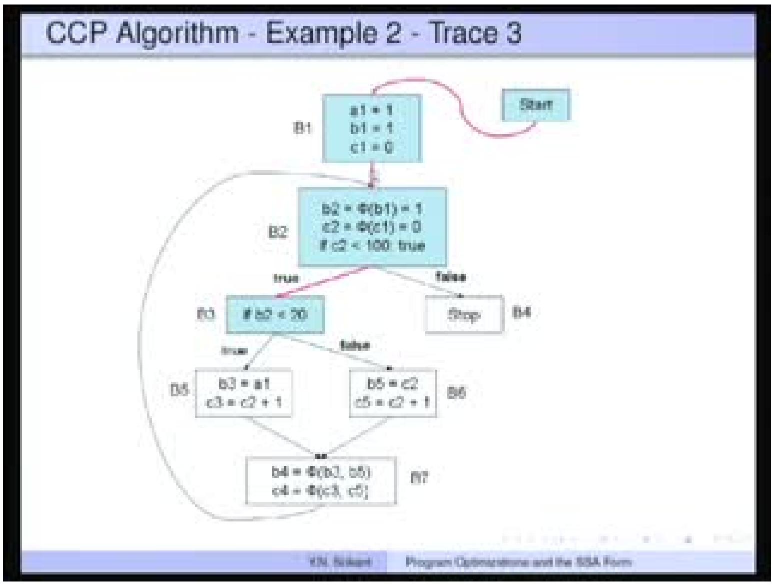
We start and then we execute this node. So, a1, b1 and c1 change their values to these constants.

(Refer Slide Time: 51:49)



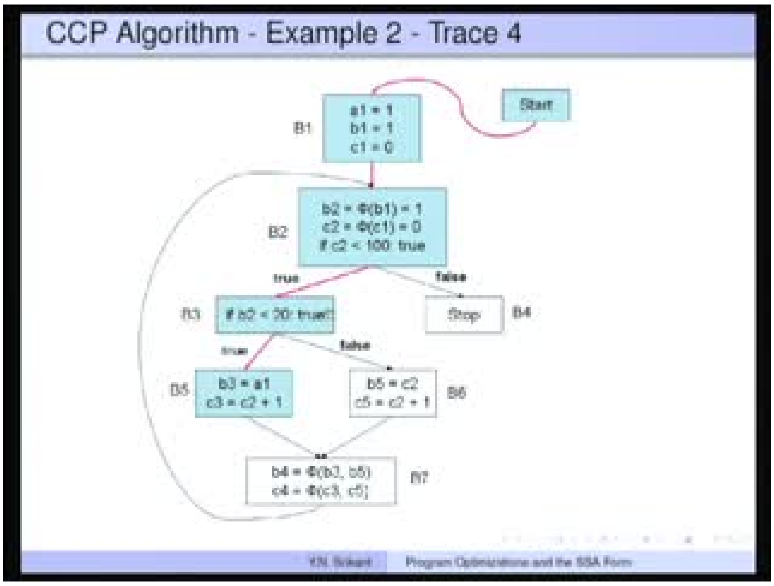
Then, we have to execute this particular node because this edge will be added to the flowpile.

(Refer Slide Time: 52:00)



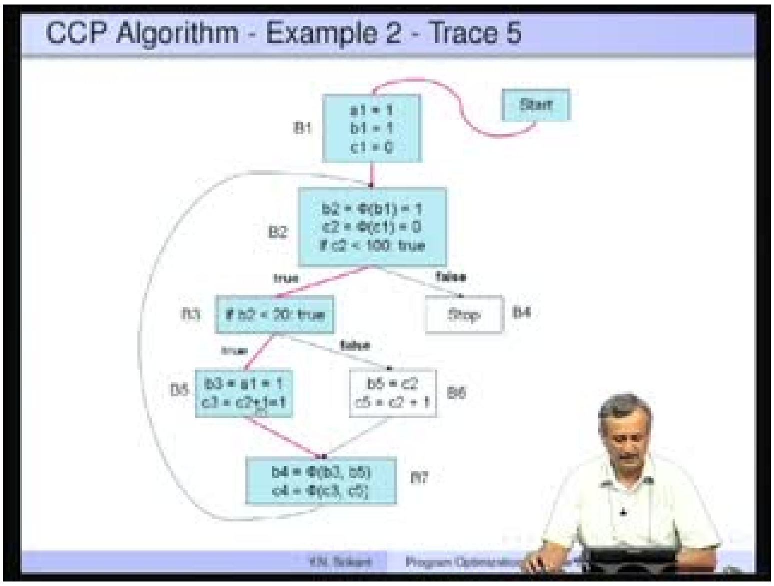
Remember that because this particular edge is not yet executed, we actually will consider only this edge and the parameter corresponding to it. That is the second parameter (Refer Slide Time: 52:11). So, phi of b1; there is only one now. So, phi of b1 is trivially b1 and that value is 1. c2 is phi of c1 and that value is 0. Now, if c2 less than 100 becomes true because c2 is 0, 0 less than 100 is true. So, we only take the true edge and come to this. This particular node will be executed next.

(Refer Slide Time: 52:34)



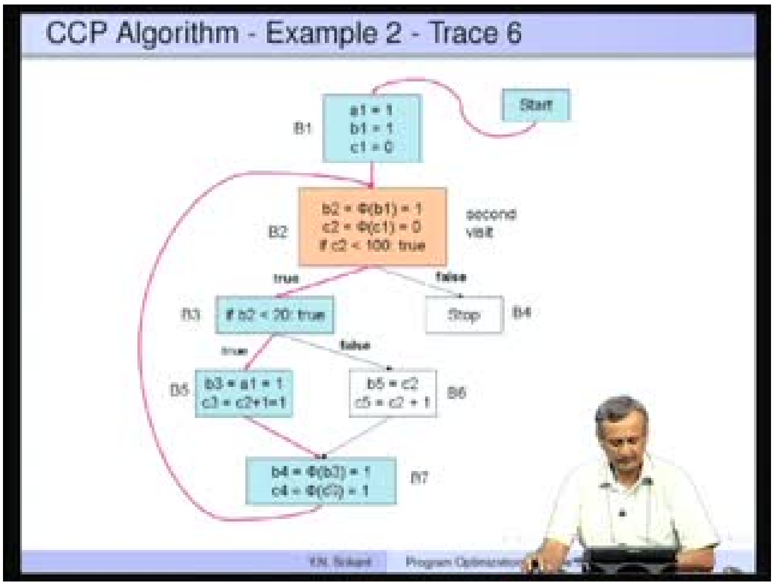
b2 less than 20 is also true because b2 is 1 and 1 less than 20 is true. So, we take only the true branch. Remember that the false edges have not yet marked as executable. Once this is marked as true, we come to this node (Refer Slide Time: 52:49). So, this becomes b3 equal to a1 and c3 equal to c2 plus 1.

(Refer Slide Time: 52:55)



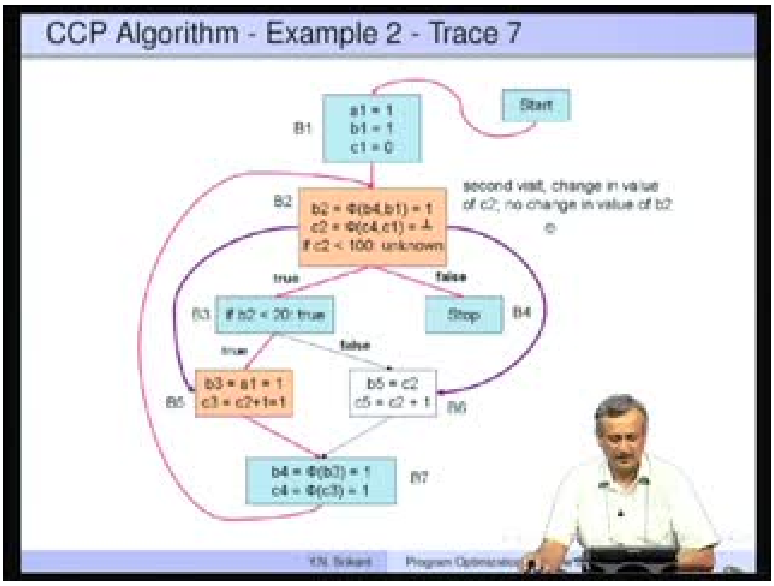
We get b3 equal to 1 and c3 equal to 1 taking these constant values along the way. That again leads us to this particular node. This is not yet marked (Refer Slide Time: 53:05). So, remember that. Therefore, we do not consider this particular edge, when we take the phi function. We consider only this particular edge. So, there is only one parameter that is b3 and another parameter c3 for the second one.

(Refer Slide Time: 53:21)



Here phi of b3 again is just b3. So, value of b3 is 1. So, b4 gets a value 1. phi of c3 is c3; c3 has a value 1. So, c4 also gets a value 1. This edge is now marked executable and this is a second visit for this particular node. That is why this has been shown in a different color. Previously, it was so. Now, this edge is also marked as executable and this edge is also marked as executable. So, what happens?

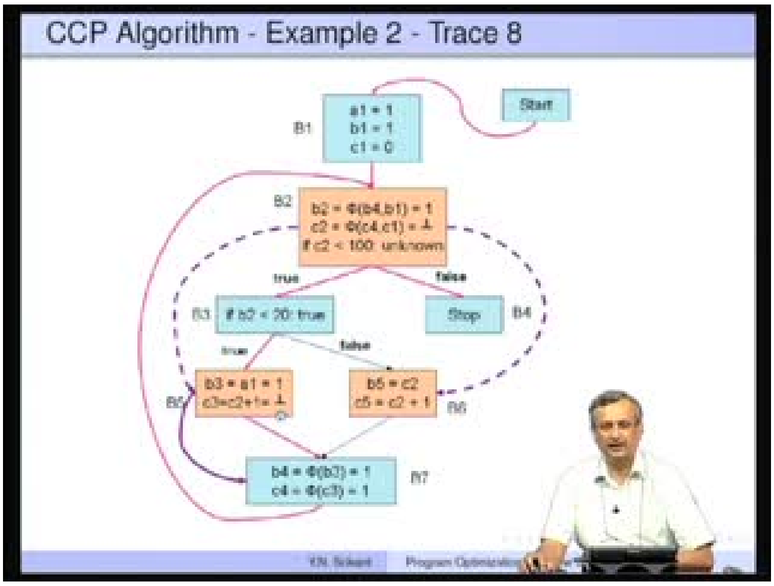
(Refer Slide Time: 53:56)



phi of b4 comma b1; b4 has a value 1, b1 has a value 1. So, phi of b4 comma b1 is phi of 1 comma 1. So, that is ok; that is still a value 1, but if you look at c2, it was previously a constant value 0 (Refer Slide Time: 54:12). Now, phi of c4 comma c1; c4 is 1 and c1 is 0. So, along one path, you have a constant value. Along another path, you have a non-constant value. Now, c2 takes a value not a constant; NAC. Therefore, c2 less than 0 becomes unknown and we need to add this edge also.

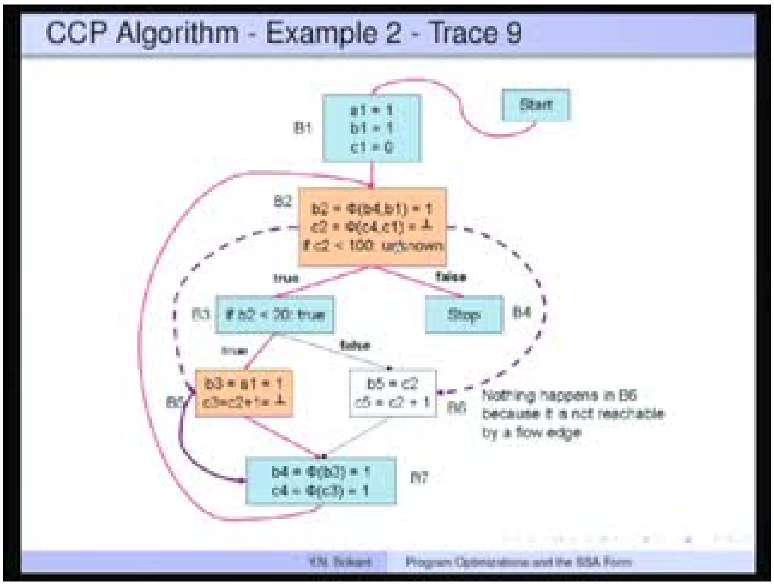
In the previous case (Refer Slide Time: 54:35), this edge was not yet executed. Now, we mark this also and put it on the work pile. This part is not yet marked (Refer Slide Time: 54:44). So, we are not going to traverse this edge again and again unless the value changes. The value has changed here. So, c2's value has changed and b2 has not changed. So, the usage of c2 is here in this. We are going to actually process this b5 using this SSA edge now. So, no change in the value of b2.

(Refer Slide Time: 55:12)



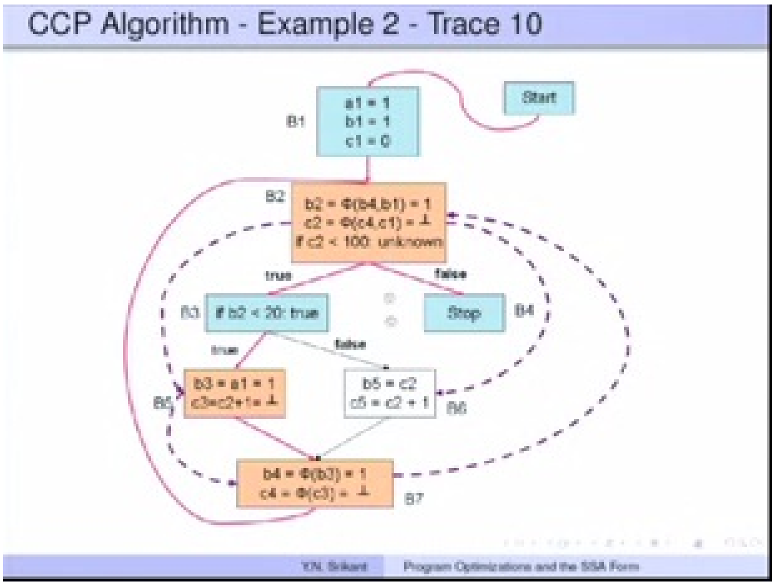
Now, c3 changes value from 1 to not a constant because c2 is not a constant. So, this c3 has changed a value. It has changed from (Refer Slide Time: 55:18) 1 to not a constant. So, this SSA edge is also going to be added to the SSA pile. Now, with this, there is no change here. For example, there is no change here. b4 and c4 do not really change. We do not have to execute this particular node again and again because b5 is here. So, only this particular value, which has changed will be used here. This edge (Refer Slide Time: 55:57) has not been executed at all, but c4 has changed the value.

(Refer Slide Time: 56:04)



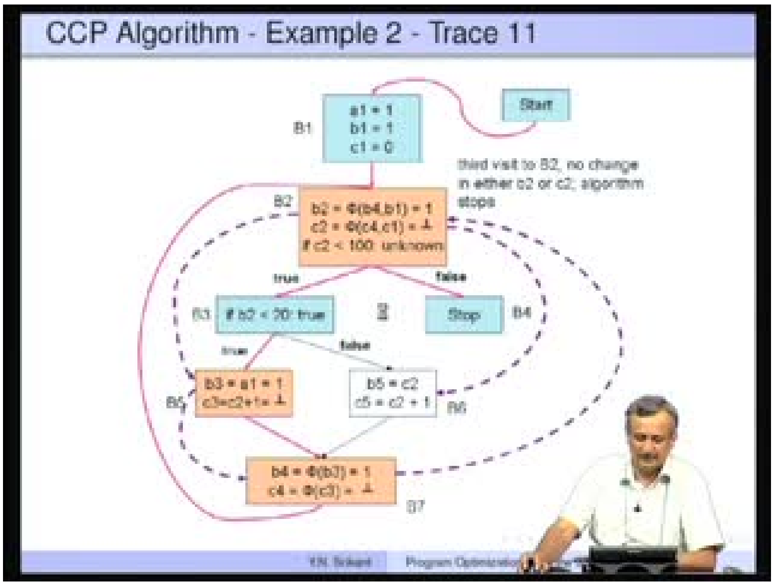
Now, nothing happens in b6. Next is this particular node, which is used and this particular change in c4. This is not a constant really (Refer Slide Time: 56:16). So, this change in c4 will introduce some changes in this. Supposed to introduce some changes in this, but it does not.

(Refer Slide Time: 56:26)



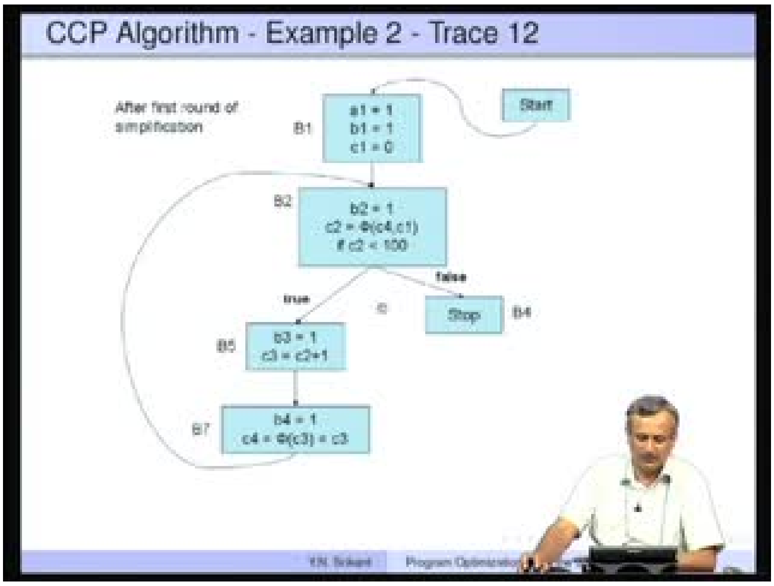
This has become not a constant. It has actually gone to this, but c2 does not change anymore. c2 was not a constant (Refer Slide Time: 56:36). Even though this c4 now changes to not a constant, this SSA edge does not change the value of c2. So, this part does not help because this is not yet executed.

(Refer Slide Time: 56:47)



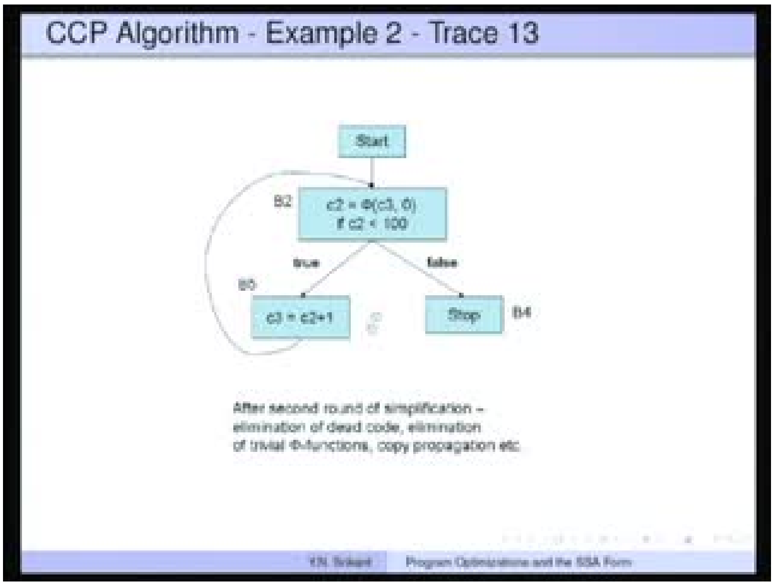
Finally, this is the third visit to b2. No change in either b2 or c2 and algorithm stops. So, this is the place where the algorithm has stopped.

(Refer Slide Time: 56:59)



This shows that after the first round of simplification, we get this flow graph. I am going to show this flow graph in the next lecture as well. So, finally, we have b2 equal to 1, c2 equal to phi of c4 comma c1, c2 less than 0, etcetera.

(Refer Slide Time: 57:14)



After some more simplification, the flow graph becomes like this. We will discuss this example along with algorithm, in the next lecture again.

Thank you.