

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

**Module No. # 13**

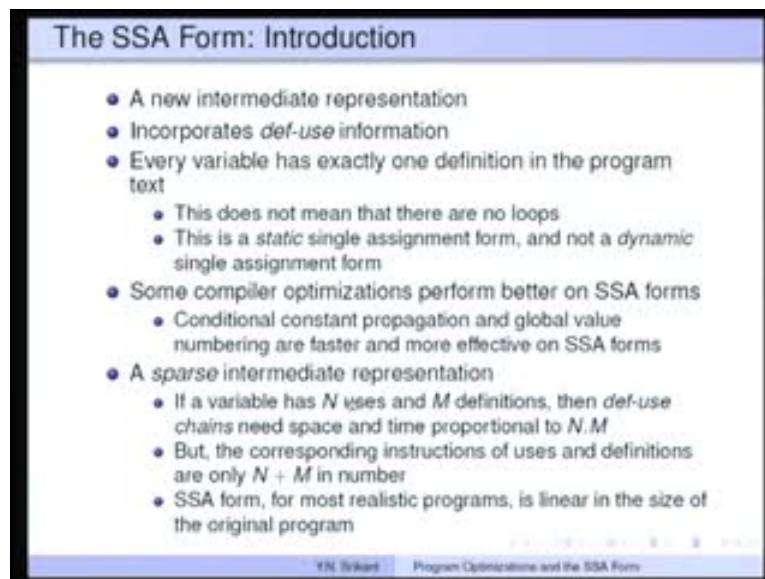
**Lecture No. # 21**

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

Welcome to the lecture on Static Single Assignment Form.

We are going to study the construction of the form known as the SSA - Static Single Assignment. We will also look at a couple of applications of this form, which are very useful in program optimizations.

(Refer Slide Time: 00:36)



The SSA Form: Introduction

- A new intermediate representation
- Incorporates *def-use* information
- Every variable has exactly one definition in the program text
  - This does not mean that there are no loops
  - This is a static single assignment form, and not a dynamic single assignment form
- Some compiler optimizations perform better on SSA forms
  - Conditional constant propagation and global value numbering are faster and more effective on SSA forms
- A sparse intermediate representation
  - If a variable has  $N$  uses and  $M$  definitions, then *def-use chains* need space and time proportional to  $N.M$
  - But, the corresponding instructions of uses and definitions are only  $N + M$  in number
  - SSA form, for most realistic programs, is linear in the size of the original program

Y.N. Srikant Program Optimizations and the SSA Form

What exactly is the Static Single Assignment form and what are its advantages? It is a new form of intermediate representation that is used in compilers. Basically, we still maintain the control flow representation, but the basic difference between a control flow graph and the SSA form is that it incorporates the definition use - def-use information. The second important property, which differentiates it from the control flow graph is that every variable has exactly one definition in the program text. So, I must hasten to add

that - this does not mean that there are no loops. They are definitely loops; otherwise, there is no way you can write non-trivial programs. This is a static single assignment form, but not a dynamic single assignment form.

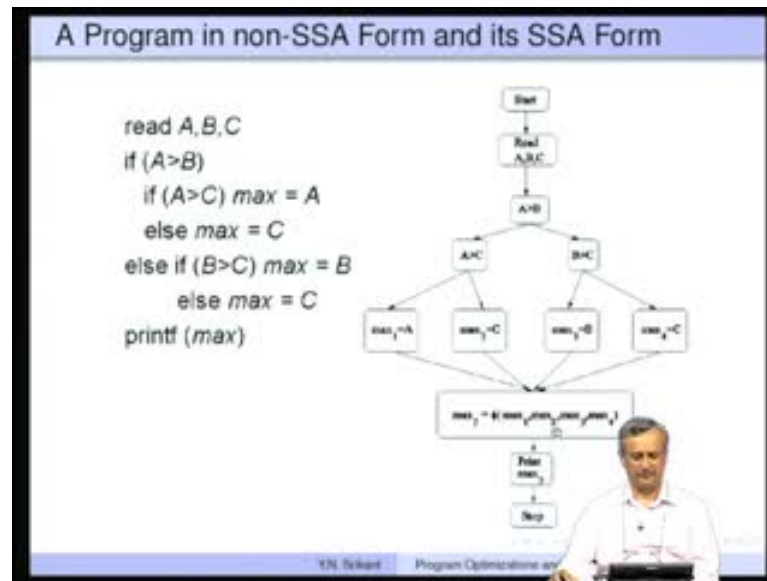
In other words, there will be definitely loops and within the loop there will be exactly one definition for each variable. The loop may iterate and at runtime the same variable may get many values, but when you look at the program statically; that is, as the compiler sees it, the loop will have exactly one definition for each variable. Of course, elsewhere in the program as well there is going to be exactly one definition per variable. Because of this property, some of the optimizations are going to be much simpler to perform, much more efficient, effective and all that.

**One very big in using** SSA form is on the constant propagation. We are going to look at constant propagation in great detail later, but conditional constant propagation and global value numbering are much faster and more effective on the SSA forms. In other words, the time required to perform the constant propagation is much smaller on the SSA form than on the control flow graph. The global value numbering that we have already seen with basic blocks is performed with little more effectiveness in the case of SSA forms.

SSA forms are also known as sparse intermediate representations. Why are they called sparse? If a variable has  $N$  uses and  $M$  definitions, then the def-use chains require space and time proportional to  $N$  into  $M$ . This is fairly well known. So, you have  $M$  definitions and  $N$  uses. So, there may be intertwining among these definitions and uses. So, you will possibly require  $N$  into  $M$  amount of space and time to consider these def-use chains.

The corresponding instructions of uses and definitions are obviously only  $N$  plus  $M$  because every definition is one instruction and every use in another instruction. So, you have  $N$  plus  $M$  number of instructions corresponding to these. The SSA form, for most realistic programs is linear in the size of the original program. So, it does not require quadratic amount of space in either the number of edges or the number of nodes in the graph. So, this is the basic advantage. We will see later that the time required to perform constant propagation, etcetera is also much smaller.

(Refer Slide Time: 04:47)



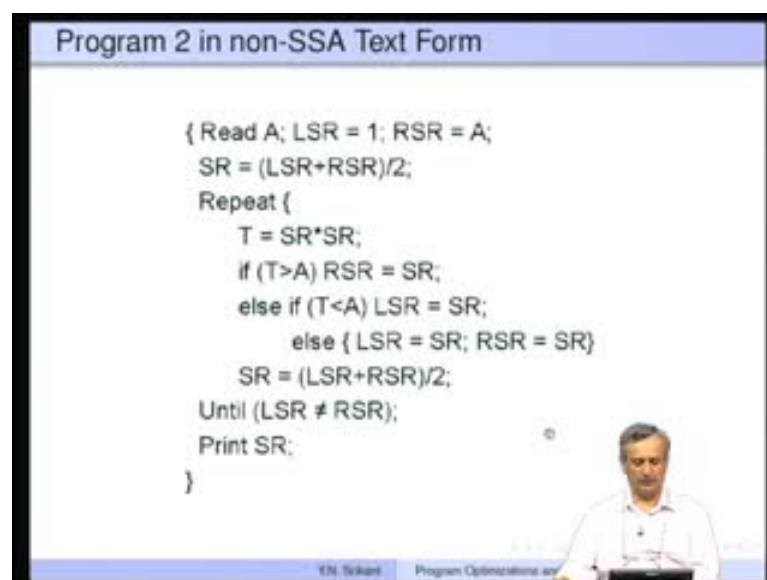
Let us look at a simple example and understand the features of SSA form. Here is a program in non-SSA form; the usual programs that we write. Here is the SSA form. It is not as if the SSA form cannot be written in text mode. Text forms of SSA programs are also possible, but it is easier to understand the salient features of these forms if they are shown in the form of a flow graph.

This is just a max finding program. Read A, B, C. If A greater than B, then check A greater than C. If so, max equal to A; otherwise, max equal to C. Else if B greater than C, max equal to B, else max equal to C. So, printf max. When this is in SSA form, so far so good; you start off, then read A, B, C, then check A greater than B. If so, check A greater than C. Then, max 1 equal to A. Let me tell you what this max 1, etcetera are; otherwise, max 2 equal to C. So, if B greater than C, max 3 equal to B and otherwise, max 4 equal to C. At this junction point (Refer Slide Time: 06:12), we have used only one variable called max and it is being updated in the whole program at various points in time. However, here in this SSA form, we are not allowed to have more than one definition per variable. These would have been max equal to A, max equal to C, max equal to B, max equal to C, etcetera. Then, here we would have had printf max, but in the SSA form, this is not possible. So, we have to introduce different variables. Let us say that this definition is max 1 equal to A (Refer Slide Time: 06:48), this is max 2 equal to C, this is max 3 equal to B and max 4 equal to C.

So far, things have been very easy, but when we come to this merge box (Refer Slide Time: 06:56), the join node, we have 4 values of max coming from the 4 edges, which particular value among these is relevant to us will be known only at runtime. If the control flow actually is along this path, then it is max 1; otherwise, it is max 2, etcetera. This choice is exercised by actually putting in place what is known as a phi function. A phi function has as many parameters as the number of incoming edges. So, we have max 1, max 2, max 3 and max 4. The semantics of this phi function are straight forward. So, the order of these parameters actually corresponds to the order of these incoming edges. So, the first parameter corresponds to the first edge, the second parameter corresponds to the second and so on.

At runtime, the implication is – if the computer is supposed to execute the phi function and has let us say – hardware to do it, then if the control flow comes along this particular edge, the phi function returns max 1. If the control flow is along this edge (Refer Slide Time: 08:15, it returns max 2, and so on and so forth. So, appropriately, this max phi, which is being assigned; this phi function here will get one of these values only at runtime. Thereby, max phi will record the maximum of A, B and C. So, this is a simple example.

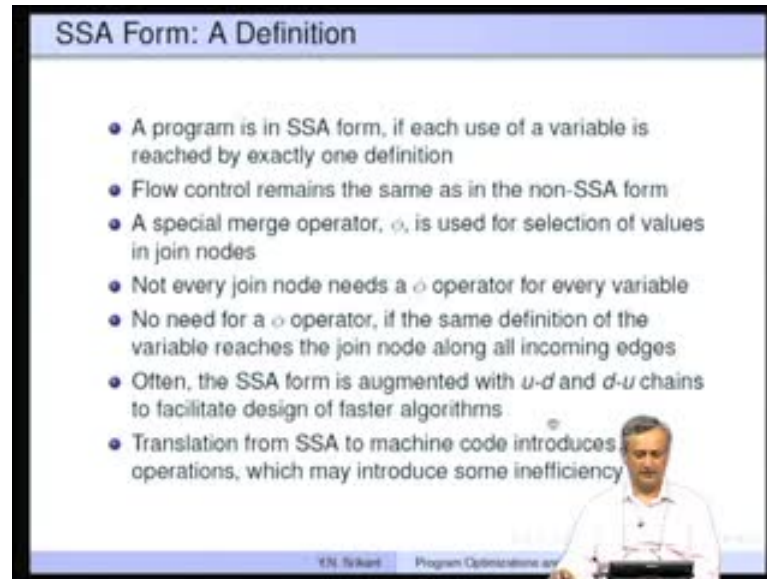
(Refer Slide Time: 08:40)



```
{ Read A; LSR = 1; RSR = A;
SR = (LSR+RSR)/2;
Repeat {
  T = SR*SR;
  if (T>A) RSR = SR;
  else if (T<A) LSR = SR;
  else { LSR = SR; RSR = SR}
  SR = (LSR+RSR)/2;
Until (LSR = RSR);
Print SR;
}
```

We are going to see more examples; let me show you one right away.

(Refer Slide Time: 08:45)



The slide is titled "SSA Form: A Definition" and contains the following bullet points:

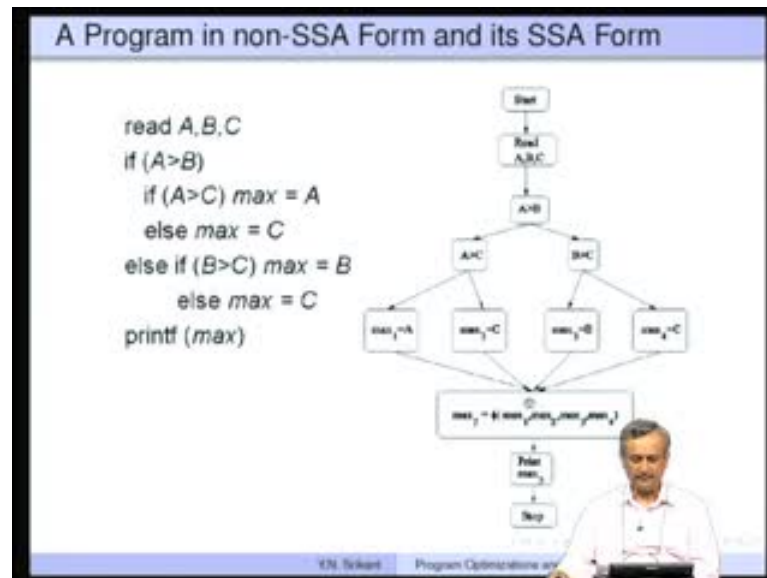
- A program is in SSA form, if each use of a variable is reached by exactly one definition
- Flow control remains the same as in the non-SSA form
- A special merge operator,  $\phi$ , is used for selection of values in join nodes
- Not every join node needs a  $\phi$  operator for every variable
- No need for a  $\phi$  operator, if the same definition of the variable reaches the join node along all incoming edges
- Often, the SSA form is augmented with  $u-d$  and  $d-u$  chains to facilitate design of faster algorithms
- Translation from SSA to machine code introduces operations, which may introduce some inefficiency

In the bottom right corner of the slide, there is a small video inset showing a man in a white shirt speaking. At the bottom of the slide, the text "EN Wikid Program Optimizations and" is visible.

Before that, let me give you a definition of an SSA form; maybe some of the details that I mentioned are already here. A program is in SSA form, if each use of a variable is reached by exactly one definition. I mentioned this already; every variable has exactly one definition. This is static not dynamic.

Flow control remains exactly the same as in the non-SSA form. We saw that example already. A special merge operator, phi is used for selection of values in join nodes. I showed you this join node and the phi operator. Not every join node needs a phi operator for every variable. This is important. Just because it is a join node and there are three variable values coming in, does not mean we need three phi functions. There is no need for a phi operator, if the same definition of the variable reaches the join node along all incoming edges.

(Refer Slide Time: 09:54)



In other words, along these edges, if exactly the same definition; let us say there was a definition  $t$  equal to  $\phi$  here. Then, the same definition actually arrives along all the paths and reaches this point. There is no need to have different versions of  $t$  and a  $\phi$  operator for  $t$  in this particular join node.

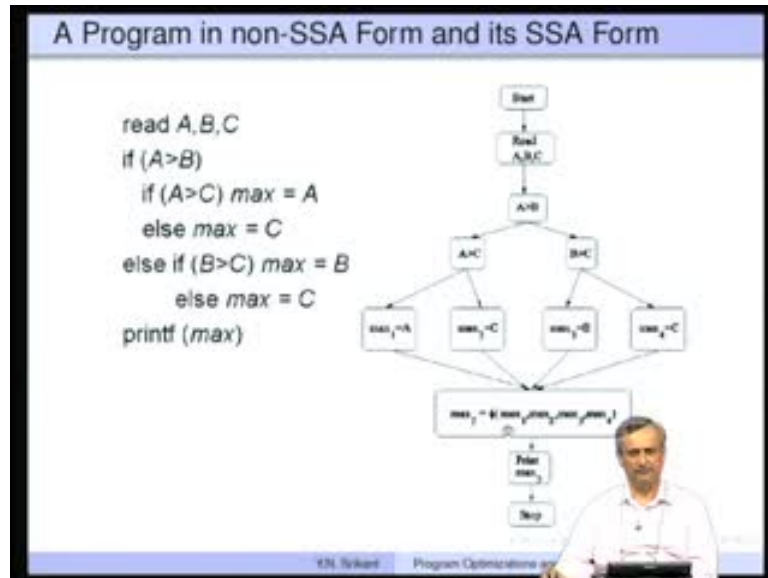
(Refer Slide Time: 10:21)

- 
- SSA Form: A Definition
- A program is in SSA form, if each use of a variable is reached by exactly one definition
  - Flow control remains the same as in the non-SSA form
  - A special merge operator,  $\phi$ , is used for selection of values in join nodes
  - Not every join node needs a  $\phi$  operator for every variable
  - No need for a  $\phi$  operator, if the same definition of the variable reaches the join node along all incoming edges
  - Often, the SSA form is augmented with  $u-d$  and  $d-u$  chains to facilitate design of faster algorithms
  - Translation from SSA to machine code introduces operations, which may introduce some inefficiency

That is what this is really saying. Often, the SSA form is augmented with  $d-u$  and  $u-d$  chains to facilitate design of faster algorithms. So, in the case of conditional constant propagation algorithm, we will need the  $d-u$  chain in order to make the algorithm much

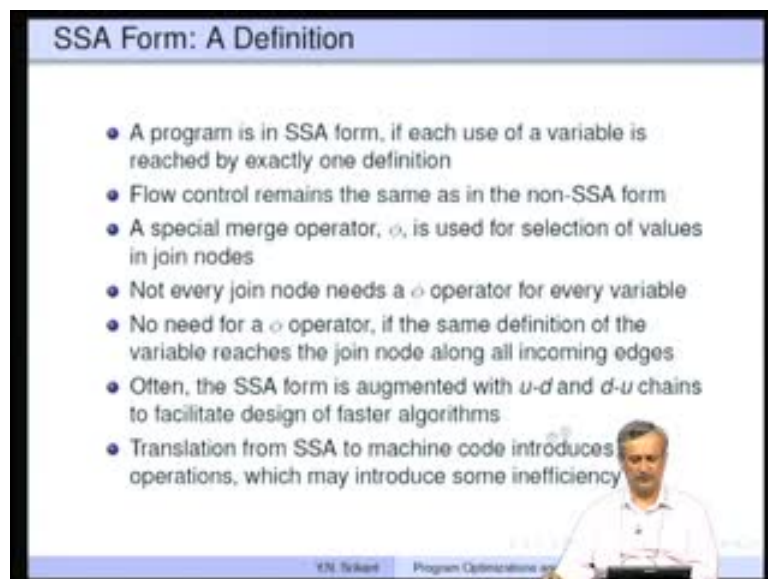
faster than others. Translation from SSA to machine code introduces copy operations, which may introduce some inefficiency.

(Refer Slide Time: 10:56)



You observed here that the phi operator is special and there will be possibly no machine, which can execute this as a machine instruction. We need to convert the phi operator into ordinary machine instructions.

(Refer Slide Time: 11:20)

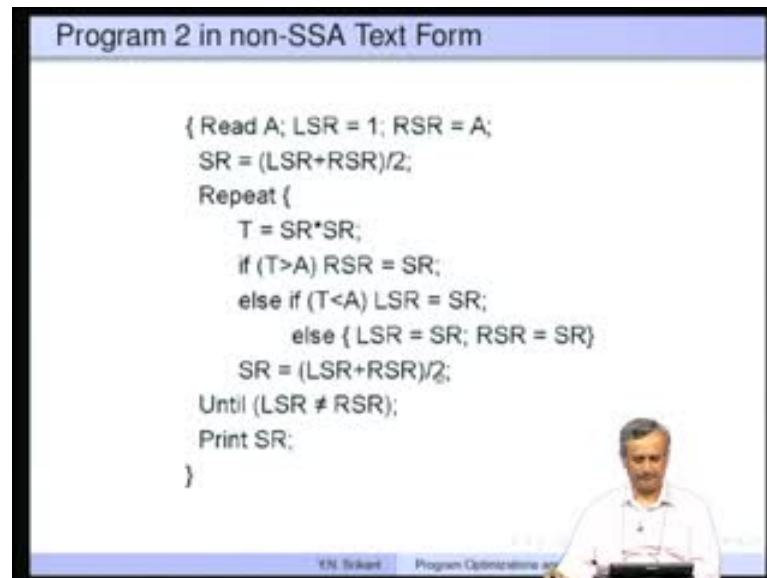


We will see how this can be done a little later, but I should mention here that the conversion operation will introduce a few copy operations. Therefore, there may be some



inefficiency, which is introduced, but hopefully the other optimization such as copy propagation and register allocation will handle this inefficiency and it is not going to be very visible.

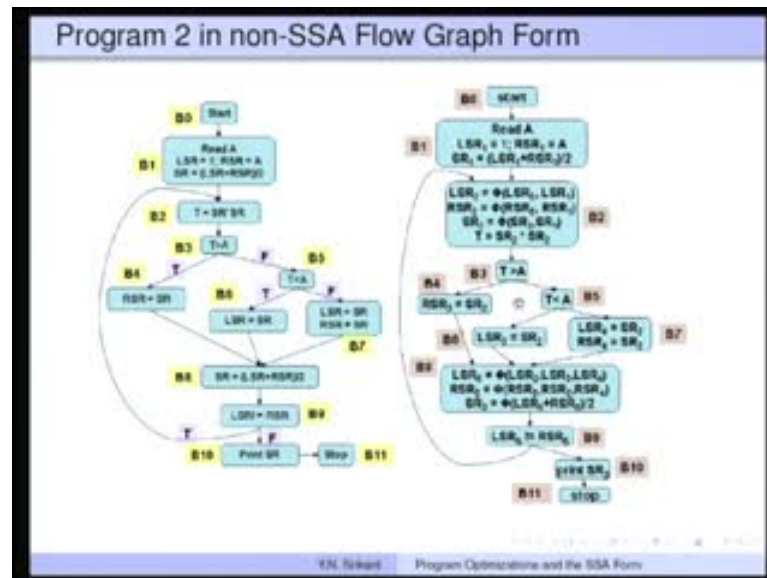
(Refer Slide Time: 11:40)



Let us look at more examples. This program is almost a meaningless program. It is very similar to binary search in syntax, but not necessarily in semantics. It has been concocted to show the specialties of the SSA forms. Here is SR equal to LSR plus RSR by 2 after initialization and reading A. Then, we have a repeat until loop. You compute T equal to SR star SR. Then, if T greater than A RSR equal to SR, if T less than A LSR equal to SR, else LSR equal to SR, RSR equal to SR and finally, SR equal to LSR plus RSR by 2 until LSR not equal to RSR. Finally, print SR.



(Refer Slide Time: 12:33)



The flow graph form of the program and the SSA form of the program are shown side by side to give you a good comparison. This is the initialization block and then you compute the initial values of SR as well and then so on; T equal to SR star SR and so on and so forth. This is a join point and this is again a join point because there are multiple incoming edges here. Each of these join points definitely has phi functions. Observe that this T here is SR equal to SR, but the join node here has many phi functions. Let me tell you why this is required.

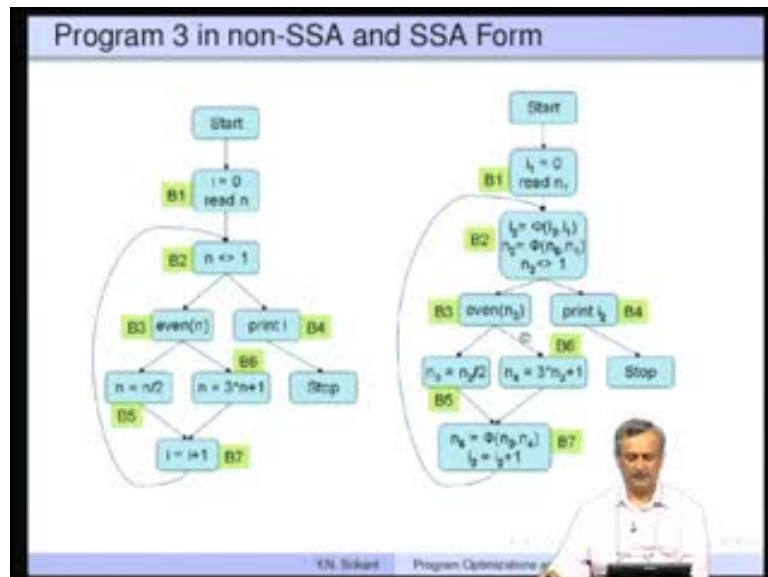
For example, this SR star SR; when the control flow is through this, we will require this SR value, which is computed in basic block B1. Later on, when the control flow iterates like that, it will receive SR value through this statement (Refer Slide Time: 13:57); that is B8. That is the reason why you require a phi function for SR in this case.

Later, we will also see how exactly to place these phi functions, but this is the first join node (Refer Slide Time: 14:13), where the LSR value from B1 and the LSR value from these two: LSR equal to SR; this B6 and B7. They actually meet. So, the value here and the value here will definitely require a phi function here. Then, that value in turn will be propagated to this basic block B2. That is what is shown here (Refer Slide Time: 14:44). LSR 5 is mentioned here. That is actually for **these two**: LSR 2 along this direction, LSR 3 along this direction and LSR 4 along this direction. Similarly, RSR. There is one value coming through this, another value coming through this, and a third value coming

through this. This may be a little puzzling, but phi function is also treated as a definition. That will again in turn give you a value. That is how we introduced phi functions and so on and so forth. We will see this later. This and this require phi functions for LSR, RSR and SR variables.

Now, you can see that if the control arrives here (Refer Slide Time: 15:32), it is going to LSR 1, if it arrives here; that is second argument, and LSR 5, if it arrives along this particular path. Similarly, for RS 2, if the control arrives along this path, it is RSR 1 and if it arrives along this path, it is RSR 5, etcetera.

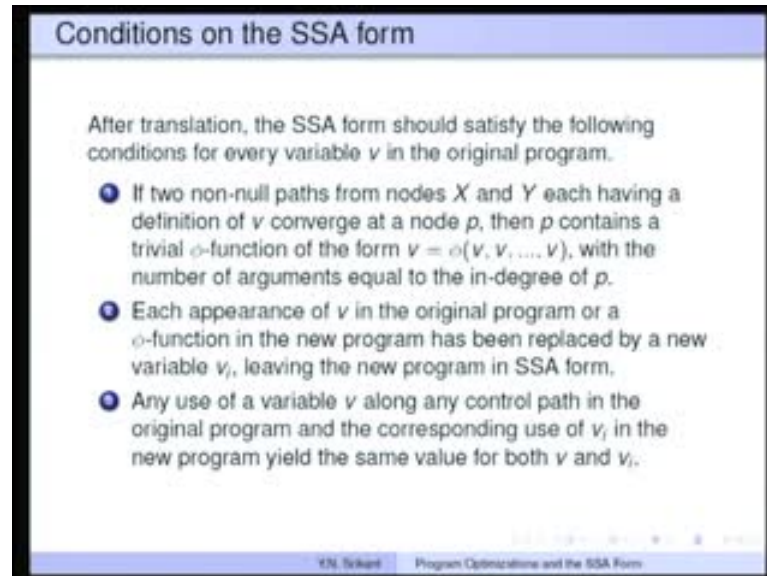
(Refer Slide Time: 15:56)



There is a third example. Here is the non-SSA form. It has two junction points: 1, 2; B2 and B7. Each of these join points do not require phi functions for both i and n. B2 requires two phi functions: one for i and another for n, but B7 does not require any phi function for i; it requires a phi function only for n. For example, here the value of i can come from here, or from this i 3. That is why there is a phi function here. Similarly, the value of n can come from here or it can come from this n 6, whereas, in this particular case, we are only computing a new value of i 3 and we are not making a choice of i 3, which actually is arriving from this edge or this edge. The value of i 2 computed in B2 actually arrives along both paths. That is the reason why we do not require a new phi function for i in block B7. So, this is the SSA form for this particular program. The choice of values here - along this path it will be n 3 and along this path it will be n 4. So,

depending on which path we take during execution,  $n$  will get that particular value. That is how the phi function operates.

(Refer Slide Time: 17:38)



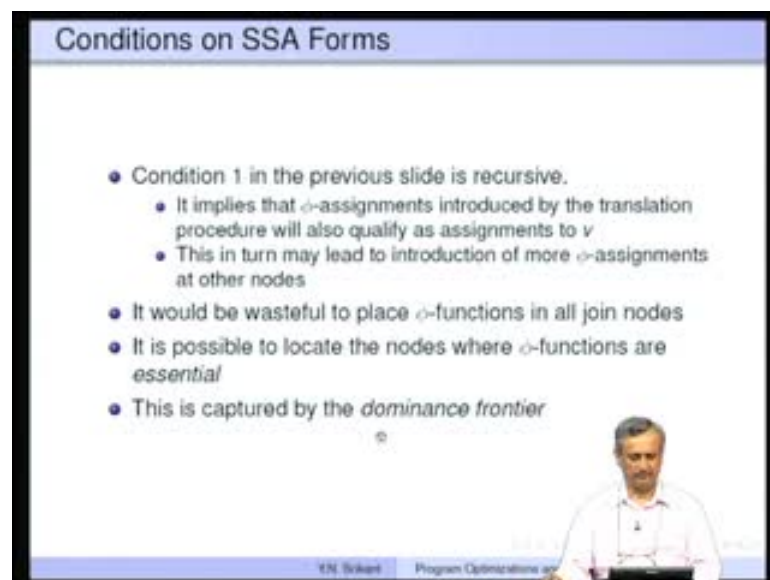
What are the conditions on the SSA form? In other words, we are given actually a non-SSA form program and we will go through an algorithm, which converts this non-SSA form program to SSA form. After the translation, the SSA form should satisfy the following conditions for every variable  $v$  in the original program. What are these conditions? 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 of  $v$  comma  $v$  comma, etcetera, with the number of arguments equal to the in-degree of  $p$ . So, what this says is – the trivial phi function is of the form phi of  $v$  comma  $v$  comma, etcetera, with all the variables being mentioned as  $v$ . The renaming operation making each of these distinct as  $v_1, v_2, v_3$ , etcetera, happens a little later in the algorithm and when that happens, the rest of the properties will be satisfied. So, this simply says when two definitions converge at a node, then we require a phi function at that node.

The second condition is – each appearance of  $v$  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. So, the phi function is also an assignment  $v$  equal to phi of  $v_1, v_2, \dots$ . So, this  $v$  (Refer Slide Time: 19:45) also needs to be changed into a new variable. Precisely, that is what this is saying. All the definitions will require to be changed to different variables and this

$v$  equal to  $\phi$  of etcetera, also needs to be changed into a new variable, but we must make sure that the new program is in SSA form. That is exactly one definition per variable.

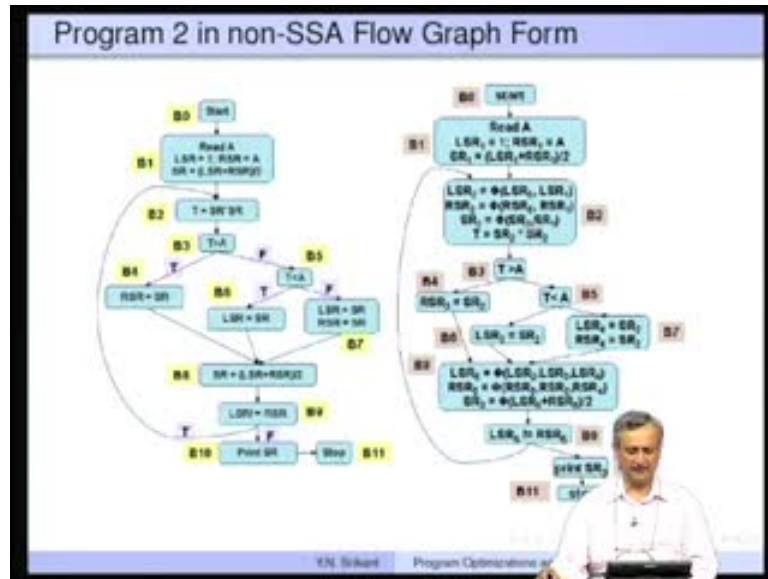
Any use of a variable along any control path in the original program and the corresponding use of  $v_i$  in the new program yield the same value for both  $v$  and  $v_i$ . So, this is making sure that the semantics of the program does not change when it change into SSA form. So, each instance of the variable, which is in a definition, will get a new variable name. That variable name should have the same value as in the original program at any control point. That is what this is trying to say.

(Refer Slide Time: 20:48)



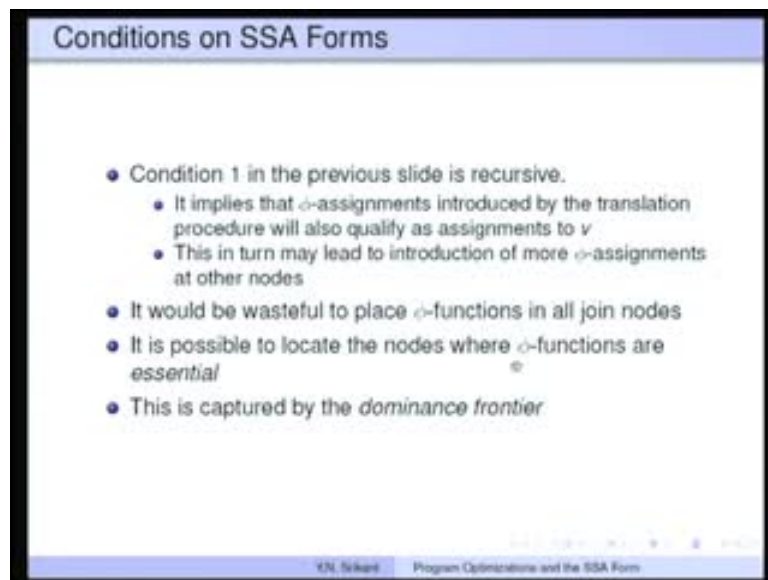
Condition 1 in the previous slide; that is, the trivial  $\phi$  function placement is recursive. It implies that  $\phi$  assignments introduced by the translation procedure will also qualify as assignments to  $v$ . This is what I was saying. So, you have to place  $\phi$  functions and then look at them, treat that also as a new assignment to  $v$ . So, you have to introduce a new variable for that assignment as well. In fact, it may so happen that introduction of one  $\phi$  assignment will lead to the introduction of another  $\phi$  assignment.

(Refer Slide Time: 21:30)



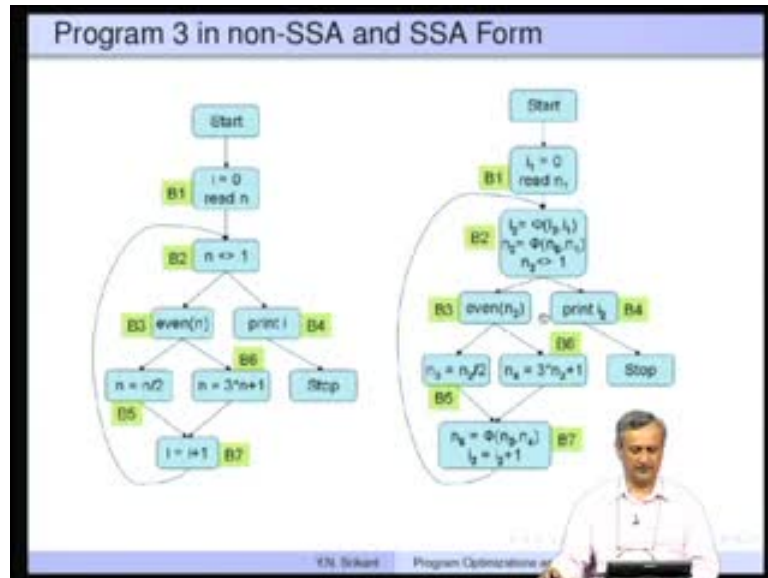
I will show you an example in this particular case. See this particular phi function, which is introduced here is because of these three edges. There are two LSR definitions coming in here and this LSR function is fed here. Now, we have two LSR values coming: one from here and another from here. Therefore, we will have to place a phi function for LSR in this case as well. Like this, introduction of one phi function may require introduction of one more phi function and things of that kind. That is precisely what this is trying to say.

(Refer Slide Time: 22:06)



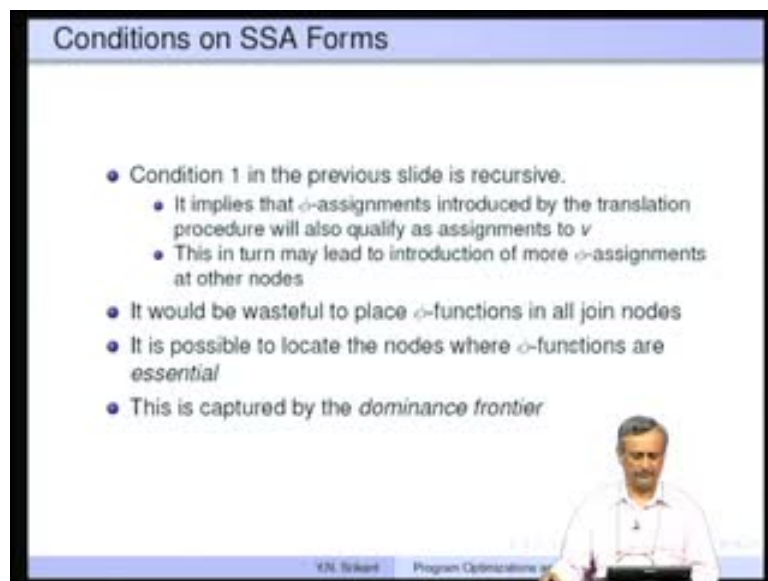
We have to go on collecting the phi assignments also as definitions and see if more phi functions become necessary because of these. We already saw that it is wasteful to place phi functions in all join nodes.

(Refer Slide Time: 22:29)



Some of them do not require phi functions. For example, here  $i$  does not require any phi function because the value of  $i$  is the same along all paths.

(Refer Slide Time: 22:38)

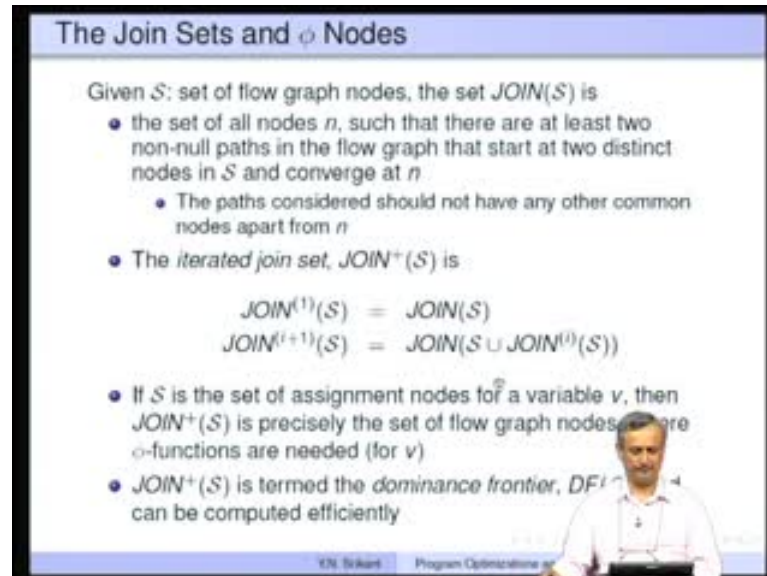


It is possible to locate the nodes where phi functions are essential. So, it is possible to compute through an algorithm and enumerate the nodes where phi functions are



essential. This information is captured by the dominance frontier property, which we are going to study in some detail.

(Refer Slide Time: 23:01)



**The Join Sets and  $\phi$  Nodes**

Given  $S$ : set of flow graph nodes, the set  $JOIN(S)$  is

- the set of all nodes  $n$ , such that there are at least two non-null paths in the flow graph that start at two distinct nodes in  $S$  and converge at  $n$ 
  - The paths considered should not have any other common nodes apart from  $n$
- The iterated join set,  $JOIN^+(S)$  is

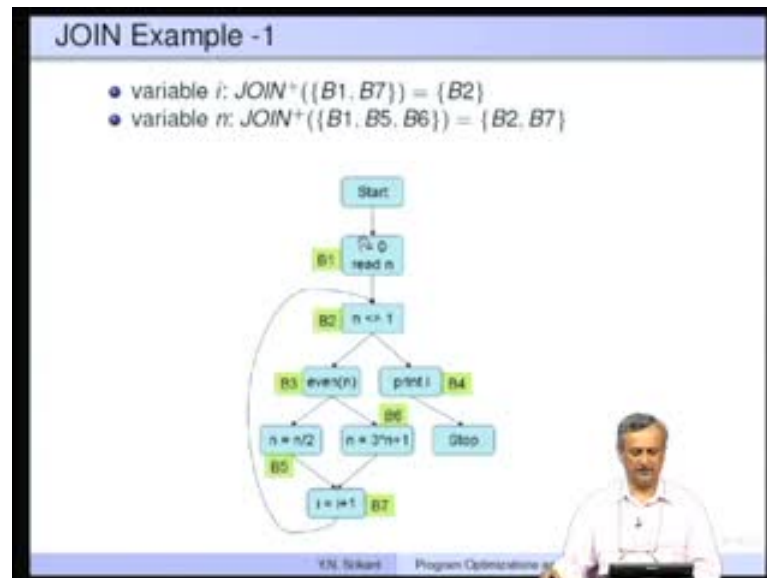
$$JOIN^{(1)}(S) = JOIN(S)$$
$$JOIN^{(i+1)}(S) = JOIN(S \cup JOIN^{(i)}(S))$$

- If  $S$  is the set of assignment nodes for a variable  $v$ , then  $JOIN^+(S)$  is precisely the set of flow graph nodes where  $\phi$ -functions are needed (for  $v$ )
- $JOIN^+(S)$  is termed the *dominance frontier*,  $DF(v)$  and can be computed efficiently

Let us understand what are known as join sets and phi nodes; the relationship between these and then go on to the dominance frontier. Let us say we are given a set  $S$  of flow graph nodes, then what is  $JOIN S$ ? The  $JOIN S$  is the set of all nodes  $n$ , such that there are at least two non-null paths in the flow graph that start at two distinct nodes in  $S$ . So, you are given  $S$  and then you are picking two nodes, both of them have definitions of variables; that we will see later, but these are two distinct nodes and they have non-null paths in the flow graph that converge at  $n$ . So, that is what is important for us. The paths considered should not have any other common nodes apart from  $n$ .



(Refer Slide Time: 24:06)



Let us look at some simple examples. Let us take the set  $S$ , which is  $B1$  comma  $B7$ ; so,  $B1$  and  $B7$  here. We want to compute the JOIN; read this as JOIN it does not matter; JOIN plus will be defined very soon.

From  $B1$ , let us look at some paths and then from  $B7$  also, let us look at some paths, and see where they converge. We should take the first node where they converge. So, this goes to  $B2$  and from  $B7$ , using the back edge, we also arrive at  $B2$ . So, this is the first node where we converge. This is in the JOIN set of  $B1$  comma  $B7$ .

What about  $B1$ ,  $B5$ ,  $B6$ ? So,  $B1$ ,  $B5$  and  $B6$ ; let us take two at a time. Suppose you consider  $B1$  and  $B5$ ,  $B1$  is here (Refer Slide Time: 25:09) and  $B5$  is here. From  $B1$ , we arrive at  $B2$ , from  $B5$ , we go via the back edge and again arrive at  $B2$ . That is the first node where they collide. So,  $B2$  is in the join set of  $B1$  and  $B5$ . Let us take  $B1$  and  $B6$ . If you consider  $B1$  and  $B6$ , you still end up with  $B2$  because  $B1$  comes to  $B2$  and  $B6$  also goes through  $B7$  and then comes to  $B2$ . So, there is nothing more to be added here, but if you consider  $B5$  and  $B6$ , the first node they converge to is  $B7$ . So,  $B7$  is also in the set of join nodes.

(Refer Slide Time: 25:55)

**The Join Sets and  $\phi$  Nodes**

Given  $S$ : set of flow graph nodes, the set  $JOIN(S)$  is

- the set of all nodes  $n$ , such that there are at least two non-null paths in the flow graph that start at two distinct nodes in  $S$  and converge at  $n$ 
  - The paths considered should not have any other common nodes apart from  $n$
- The iterated join set,  $JOIN^+(S)$  is
  - $JOIN^{(1)}(S) = JOIN(S)$
  - $JOIN^{(i+1)}(S) = JOIN(S \cup JOIN^{(i)}(S))$
- If  $S$  is the set of assignment nodes for a variable  $v$ , then  $JOIN^+(S)$  is precisely the set of flow graph nodes, where  $\phi$ -functions are needed (for  $v$ )
- $JOIN^+(S)$  is termed the *dominance frontier*,  $DF(S)$ , and can be computed efficiently

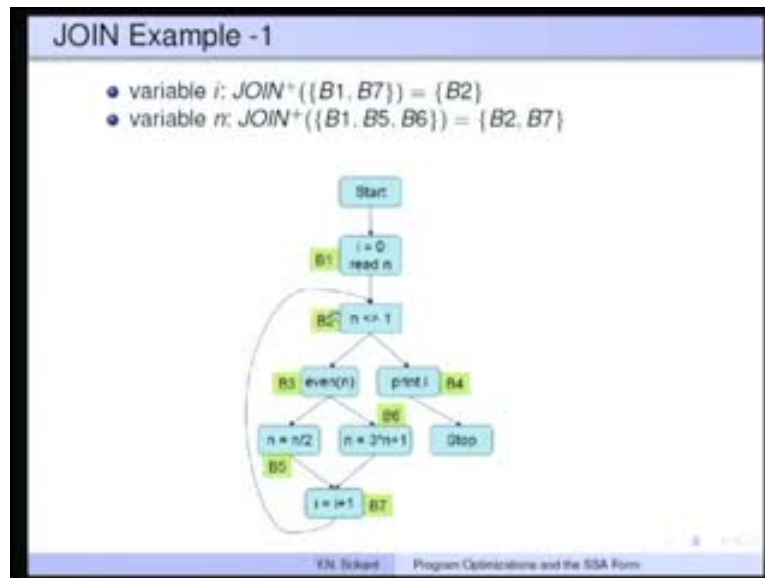
EN Wikid Program Optimizations and the SSA Form

This is the way we compute JOIN, rather define JOIN. What is an iterated JOIN set? JOIN plus S. This join plus is needed to take care of phi functions introduced by phi functions. So, let us define it. Recursion again; JOIN 1 S. This is the terminating condition equal to JOIN S and JOIN of i plus 1 S is JOIN of S U in JOIN of i of S. So, you combine S and JOIN I of S and take the JOIN of that, then you get JOIN i plus 1 S. That is called as JOIN plus; this is the closure.

If S is the set of assignment nodes for a variable v, then JOIN plus S is precisely the set of flow graph nodes where phi functions are needed for v. This is the reason why we wanted to define the JOIN set.

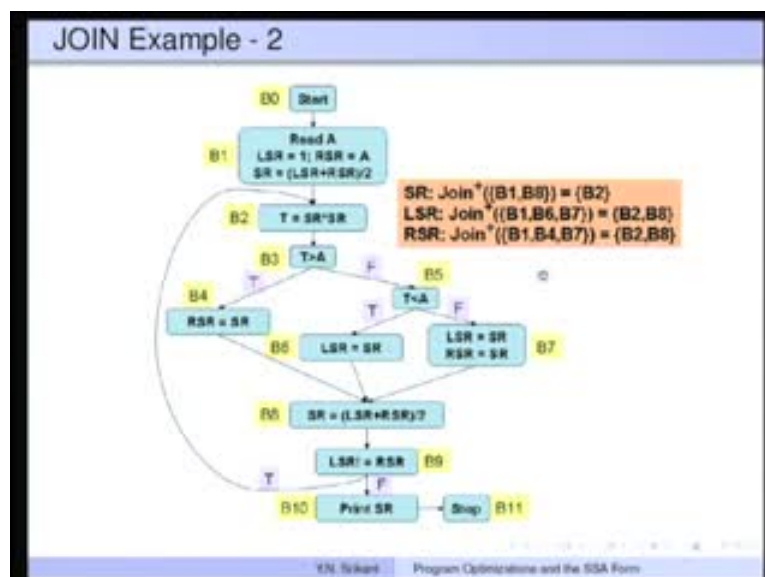
JOIN plus S is termed as the dominance frontier, DF of S and definitely it can be computed efficiently. So, we are going to look at some algorithm to compute the dominance frontier. The central concept in translating an ordinary flow graph to SSA form is the concept of dominance frontier. So, we need to understand how to define and compute dominance frontier.

(Refer Slide Time: 27:33)



To continue with the same example, by adding B2 into B1 comma B7; that is, S union B2, we really do not get any extras. So, if you take B1 and B2, it gives you nothing extra. Taking B2 and B7 also gives you nothing more. Similarly, taking B2 and B7 into S and then computing JOIN still does not add anything to this particular closure. So, we are actually going to look at the JOIN plus as B2 comma B7 itself.

(Refer Slide Time: 28:17)



Second example: Again, let us compute the join. We are looking at B1 and B8; so, B1 is here and B8 is the other join node. Let us take only the join nodes because they are the

only ones which are of great interest to us. These are the only places where we need to put phi functions. So, these two converge at B2; very similar to the previous case. So, this goes through this arc and then comes to B2. This is for the variable SR. For the variable LSR, let us consider the nodes where LSR is defined. LSR is defined in B1, then L S R is defined in B6, and it is also defined in B7. So, we are going to consider B1, B6 and B7 for that particular purpose.


We need to take two at a time. So, let us take B1 and B6. Actually, we can take a path along this particular edge (Refer Slide Time: 29:31) and come to B8; B6 directly comes to B8. So, B8 gets included. Remember – we are not supposed to use common paths, they should be disjoint paths; they meet only at the common node. Then, B6 and B7 again meet at B8. What about B1 and B7? B1 again comes here (Refer Slide Time: 30:00) and B7 also comes here. Suppose we add this B8 into B1, B6, B7; we have now added here, we are taking S union this join. Then, B1 is here, B8 is the extra. So, B1 and B8 actually meet only in B2. That is how B2 comes into the set. So, B8 remains in the set; B2 and B8 now form the Join plus.

Similarly, the RSR; it is very similar. So, RSR is from B4, then B1, and then B7. So, these three (Refer Slide Time: 30:53) will again converge at B2 and B8. The Join plus operation uses B2 and B8. So, B2 and B8 are the places where we are going to place the phi functions for both variables, LSR and RSR. SR will have one phi function in B2. That is how the placement takes place.

(Refer Slide Time: 31:20)

### Dominators and Dominance Frontier

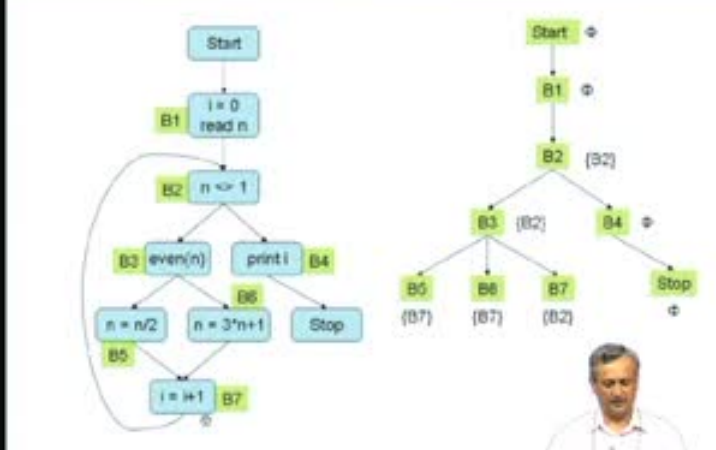
- Given two nodes  $x$  and  $y$  in a flow graph,  $x$  dominates  $y$  ( $x \in \text{dom}(y)$ ), if  $x$  appears in all paths from the *Start* node to  $y$
- The node  $x$  strictly dominates  $y$ , if  $x$  dominates  $y$  and  $x \neq y$
- $x$  is the immediate dominator of  $y$  (denoted  $\text{idom}(y)$ ), if  $x$  is the closest strict dominator of  $y$
- A dominator tree shows all the immediate dominator relationships
- The dominance frontier of a node  $x$ ,  $DF(x)$ , is the set of all nodes  $y$  such that
  - $x$  dominates a predecessor of  $y$  ( $p \in \text{preds}(y)$  and  $x \in \text{dom}(p)$ )
  - but  $x$  does not strictly dominate  $y$  ( $x \notin \text{dom}(y)$ )




How do we compute the dominance frontier? To compute the dominance frontier, we require dominator information. So, let us do a recap on dominators that we studied a couple of lectures ago. Given two nodes  $x$  and  $y$  in a flow graph,  $x$  dominates  $y$ , if  $x$  appears in all paths from the start node to  $y$ . So, that is the definition of dominator.

(Refer Slide Time: 31:56)

### DF Example - 1



The flow graph shows a loop structure. The dominator tree illustrates the dominance relationships: Start dominates B1 and B2; B1 dominates B2; B2 dominates B3 and B4; B3 dominates B5, B6, and B7; B4 dominates Stop.




If you look at this example, here is the dominator tree information for this particular flow graph. So, B2 dominates all these nodes because all the paths starting from the start node to any one of these nodes has to pass through B2. B3 dominates B5, B6 and also B7. The

reason is – if you look at this... These two are very simple. So, every path from start goes through B3, but even to reach B7, you have to actually pass through B3. Neither B5 nor B6 will dominate B7 because you can always find an alternate path to reach B7 bypassing the node B5 or B6.

(Refer Slide Time: 32:47)

### Dominators and Dominance Frontier

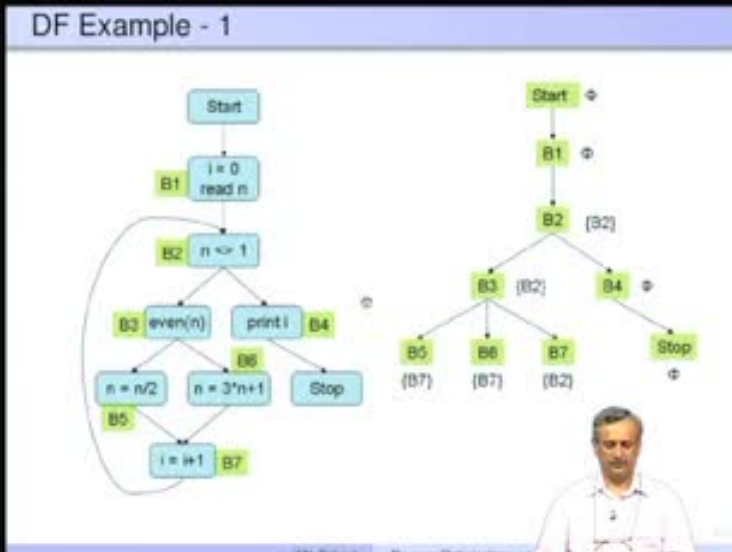

- Given two nodes  $x$  and  $y$  in a flow graph,  $x$  dominates  $y$  ( $x \in \text{dom}(y)$ ), if  $x$  appears in all paths from the Start node to  $y$
- The node  $x$  strictly dominates  $y$ , if  $x$  dominates  $y$  and  $x \neq y$
- $x$  is the immediate dominator of  $y$  (denoted  $\text{idom}(y)$ ), if  $x$  is the closest strict dominator of  $y$
- A dominator tree shows all the immediate dominator relationships
- The dominance frontier of a node  $x$ ,  $DF(x)$ , is the set of all nodes  $y$  such that
  - $x$  dominates a predecessor of  $y$  ( $p \in \text{preds}(y)$  and  $x \in \text{dom}(p)$ )
  - but  $x$  does not strictly dominate  $y$  ( $x \notin \text{dom}(y)$ )



The node  $x$  strictly dominates  $y$ , if  $x$  dominates  $y$  and  $x$  is not equal to  $y$ . This is one is of strict domination. So, we do not want  $x$  equal to  $y$ .  $x$  is the immediate dominator of  $y$   $\text{idom}$ ; this is important for us, if  $x$  is the closest strict dominator of  $y$ .

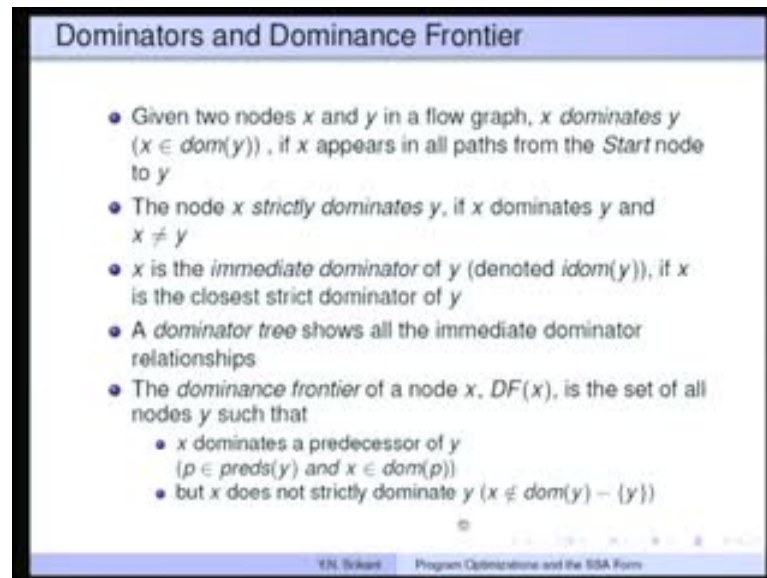
(Refer Slide Time: 33:11)

### DF Example - 1

This particular tree information that is given here shows only idom information, immediate dominator information. It does not show the transitive relationship, it should be inferred using the descendants of a particular node. B2 for example, immediately dominates B3 and B4, but it also dominates B5, B6, B7 and stop, but they are all descendants; that is it.

(Refer Slide Time: 33:37)



A dominator tree shows all the immediate dominator relationships. Now, coming to the dominance frontier; we have defined the join set where two definitions converge for the first time. Dominance frontier is exactly that, but we are going to define dominance frontier in a structural way so that computation is simpler; we are going to use dominator information for that.

$DF(x)$  is the set of all nodes  $y$  such that  $x$  dominates a predecessor of  $y$ , but  $x$  does not strictly dominate  $y$ . This is the definition of dominance frontier. So, it should dominate a predecessor of  $y$ , but  $x$  does not strictly dominate the node  $y$  itself.



(Refer Slide Time: 34:32)

### Dominance frontiers - An Intuitive Explanation

- A definition in node  $n$  forces a  $\phi$ -function in join nodes that lie just outside the region of the flow graph that  $n$  dominates; hence the name *dominance frontier*
- Informally,  $DF(x)$  contains the *first* nodes reachable from  $x$  that  $x$  does not dominate, on *each* path leaving  $x$ 
  - In example 1 (next slide),  $DF(B1) = \emptyset$ , since B1 dominates all nodes in the flow graph except Start and B1, and there is no path from B1 to Start or B1
  - In the same example,  $DF(B2) = \{B2\}$ , since B2 dominates all nodes except Start, B1, and B2, and there is a path from B2 to B2 (via the back edge)
  - Continuing in the same example, B5, B6, and B7 do not dominate any node and the first reachable nodes are B7, B7, and B2 (respectively). Therefore,  $DF(B5) = DF(B6) = \{B7\}$  and  $DF(B7) = \{B2\}$
  - In example 2 (second next slide), B5 dominates B6 and B7, but not B8; B8 is the first reachable node from B5 that B5 does not dominate; therefore,  $DF(B5) = \{B8\}$

Y.N. Srikant Program Optimizations and the SSA Form

However, this is still not a great definition if you want to compute it. So, let us first understand intuitively what exactly this dominance frontier is; it does not seem to be so clear. A definition in node  $n$  forces a  $\phi$  function in join nodes that lie just outside the region of the flow graph that  $n$  dominates; hence the name dominance frontier. Informally, the set DF of  $x$  contains the first nodes reachable from  $x$  that  $x$  does not dominate, on each path leaving  $x$ .

(Refer Slide Time: 35:20)

### DF Example - 1

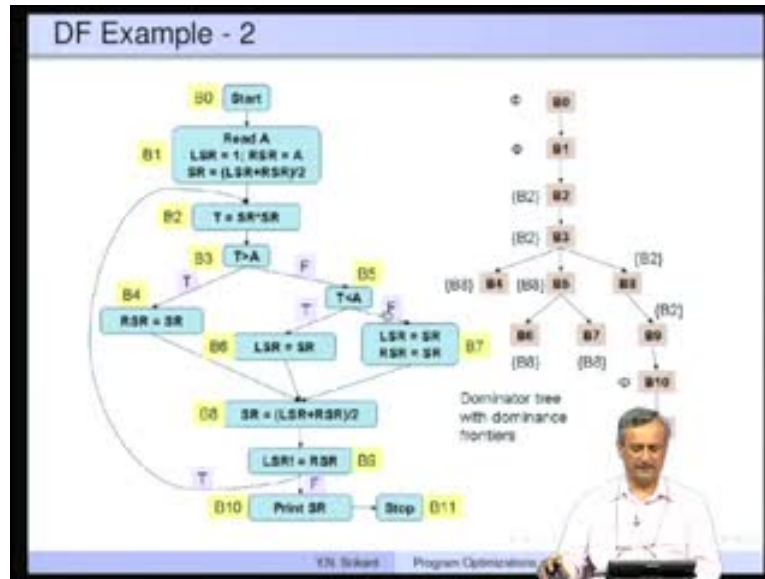
Y.N. Srikant Program Optimizations

Let us look at this example. If you look at start node, it dominates every node. So, there is no question of reaching a node, which is not dominated by start. Therefore, the dominance frontier, which is indicated just beside the node here is  $\phi$  for start. The same is true for B1. There is no node, which is not dominated by B1, but is still reachable. Start is not dominated by B1, but there is no path to start node.

Suppose you consider the node B2, B2 dominates again every other node here B3, B4, B5, B6, stop and B7, but there is a path from B2 using the back arc back to B2. Since we are looking at strict domination only, B2 dominates itself is not considered for us in our algorithm. So, there is a path from B2 to itself through the back arc and B2 does not dominate itself in the strict dominator sense. Therefore, the dominance frontier of B2 consists of the node B2.

Let us consider B3. You are looking at B3 dominates B5, B6 and B7 (Refer Slide Time: 37:00); B5, B6 and B7. Let us start taking a path out of these, you again go to B2. So, this is (Refer Slide Time: 37:10) B2; dominance frontier of B3 is also B2. Let us look at B5. B5 does not dominate any node. So, the immediate node, which is reachable from B5 is B7 and that is its dominance frontier. The same is true for B6 as well. If you look at B7, it does not dominate any other node. The next node reachable is B2. So, it is B2. B4 dominates stop and there is no other back arc. So, B4 has dominance frontier as  $\phi$  and same is true for stop as well. So, this is what is explained in this particular slide (Refer Slide Time: 37:53).

(Refer Slide Time: 37:56)



Let us look at the second example. Bnaught and B1 are similar. They have dominance frontier information as phi because start and B1 dominate every node in the flow graph and no other node is reachable from either start or B1. That is, no other node, which is not dominated by them is reachable from these two nodes. Because of the loop, B2 again is reachable from itself. So, its dominance frontier is B2. The same is true for B3. So, you go through this path and then the first node you come to is, which does not dominate is B2. So, the dominance frontier for B3 is also B2.


What about B4? B4 is here (Refer Slide Time: 38:54) and it does not dominate any other node. So, the next node is B8 and hence, DF is B8 for B4. Similarly, for B5 as well DF is B8. So, it dominates B6 and B7, it comes here (Refer Slide Time: 39:11); this is not dominated by B5 because there is a path like this. So, B8 is on the dominance frontier of B5. For B8 itself, it dominates all other nodes here, B9, B10 and B11. So, we reach B2 and that is in DF. For B6 and B7, it is just B8 and then for B9, you can again go back. So, it is B2 and for others it is phi.

(Refer Slide Time: 39:46)

### Computation of Dominance Frontiers - 2

- 1 Identify each join node  $x$  in the flow graph
- 2 For each predecessor,  $p$  of  $x$  in the flow graph, traverse the dominator tree upwards from  $p$ , till  $idom(x)$
- 3 During this traversal, add  $x$  to the  $DF$ -set of each node met

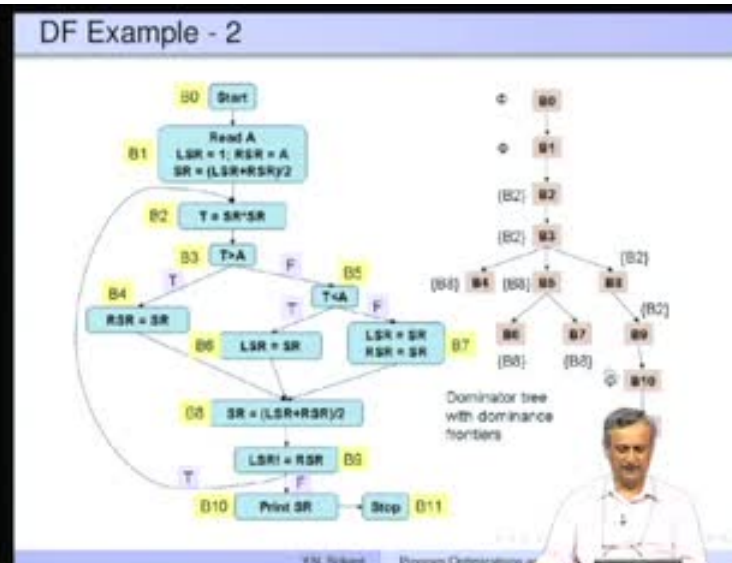
- In example 1 (second previous slide), consider the join node B2; its predecessors are B1 and B7
  - B1 is also  $idom(B2)$  and hence is not considered
  - Starting from B7 in the dominator tree, in the upward traversal till B1 (i.e.,  $idom(B2)$ ) B2 is added to the  $DF$  sets of B7, B3, and B2
- In example 2 (previous slide), consider the join node B8; its predecessors are B4, B6, and B7
  - Consider B4: B8 is added to  $DF(B4)$
  - Consider B6: B8 is added to  $DF(B6)$  and  $DF(B5)$
  - Consider B7: B8 is added to  $DF(B7)$ ; B8 has already been added to  $DF(B5)$
  - All the above traversals stop at B3, which is



This is how the dominance frontier information is intuitively understood.

(Refer Slide Time: 39:54)

### DF Example - 2



Dominator tree with dominance frontiers


You are really looking at the node just immediately beyond the nodes, which are dominated by the particular node. That is why, it is some kind of a frontier just beyond the dominator region and hence, it is called a dominance frontier.

(Refer Slide Time: 40:13)

### Computation of Dominance Frontiers - 2

- 1 Identify each join node  $x$  in the flow graph
- 2 For each predecessor,  $p$  of  $x$  in the flow graph, traverse the dominator tree upwards from  $p$ , till  $\text{idom}(x)$
- 3 During this traversal, add  $x$  to the  $DF$ -set of each node met

- In example 1 (second previous slide), consider the join node  $B_2$ ; its predecessors are  $B_1$  and  $B_7$ 
  - $B_1$  is also  $\text{idom}(B_2)$  and hence is not considered
  - Starting from  $B_7$  in the dominator tree, in the upward traversal till  $B_1$  (i.e.,  $\text{idom}(B_2)$ )  $B_2$  is added to the  $DF$  sets of  $B_7$ ,  $B_3$ , and  $B_2$
- In example 2 (previous slide), consider the join node  $B_8$ ; its predecessors are  $B_4$ ,  $B_6$ , and  $B_7$ 
  - Consider  $B_4$ :  $B_8$  is added to  $DF(B_4)$
  - Consider  $B_6$ :  $B_8$  is added to  $DF(B_6)$  and  $DF(B_5)$
  - Consider  $B_7$ :  $B_8$  is added to  $DF(B_7)$ ;  $B_8$  has already been added to  $DF(B_5)$
  - All the above traversals stop at  $B_3$ , which is


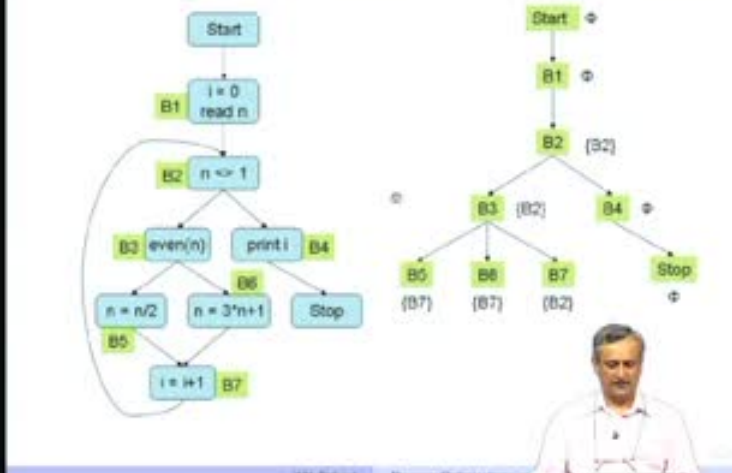


How do we compute the DF? Whatever we said so far is not an algorithm, it is only an intuitive explanation of what DF is.

Algorithm is simple. Identify each node  $x$  in the flow graph. For each predecessor,  $p$  of  $x$  in the flow graph, traverse the dominator tree upwards from  $p$ , till  $\text{idom}$  of  $x$ . During this traversal, add  $x$  to the  $DF$ -set of each node met.

(Refer Slide Time: 40:55)

### DF Example - 1



Let us walk through our examples and compute the DF information accordingly. Again, we really need to consider only the join nodes:  $B_2$  and  $B_7$ . If you compute the DF


information for them using this algorithm, automatically, the DF information for all others also gets computed. Let us see how.

(Refer Slide Time: 41:23)

### Computation of Dominance Frontiers - 2

- 1 Identify each join node  $x$  in the flow graph
- 2 For each predecessor,  $p$  of  $x$  in the flow graph, traverse the dominator tree upwards from  $p$ , till  $idom(x)$
- 3 During this traversal, add  $x$  to the DF-set of each node met

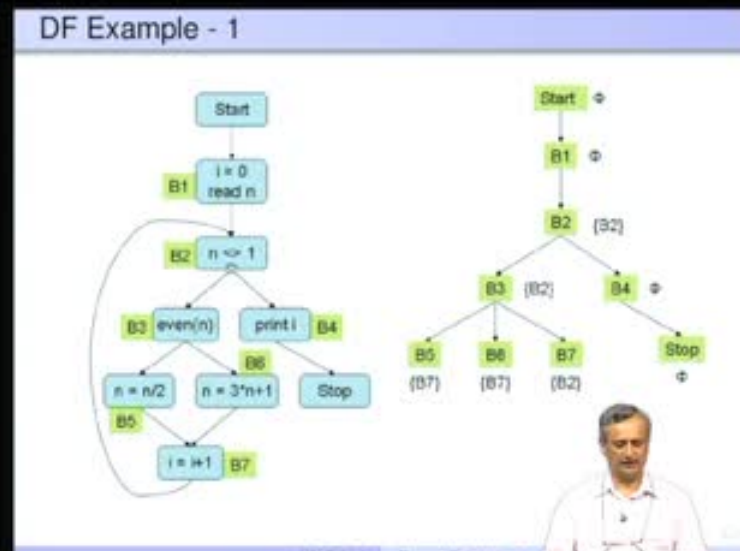
- In example 1 (second previous slide), consider the join node B2; its predecessors are B1 and B7
  - B1 is also  $idom(B2)$  and hence is not considered
  - Starting from B7 in the dominator tree, in the upward traversal till B1 (i.e.,  $idom(B2)$ ) B2 is added to the DF sets of B7, B3, and B2
- In example 2 (previous slide), consider the join node B8; its predecessors are B4, B6, and B7
  - Consider B4: B8 is added to  $DF(B4)$
  - Consider B6: B8 is added to  $DF(B6)$  and  $DF(B5)$
  - Consider B7: B8 is added to  $DF(B7)$ ; B8 has already been added to  $DF(B5)$
  - All the above traversals stop at B3, which is



The algorithm says – identify each join node for each predecessor,  $p$  in the flow graph, traverse the dominator tree upwards from  $p$ , till you reach  $idom$ .

(Refer Slide Time: 41:34)

### DF Example - 1



Let us say we are in B2 because that is the first join node that we have. If there are two predecessors in the control flow graph, for this node B2;



[No audio from 41:48 to 41:54]

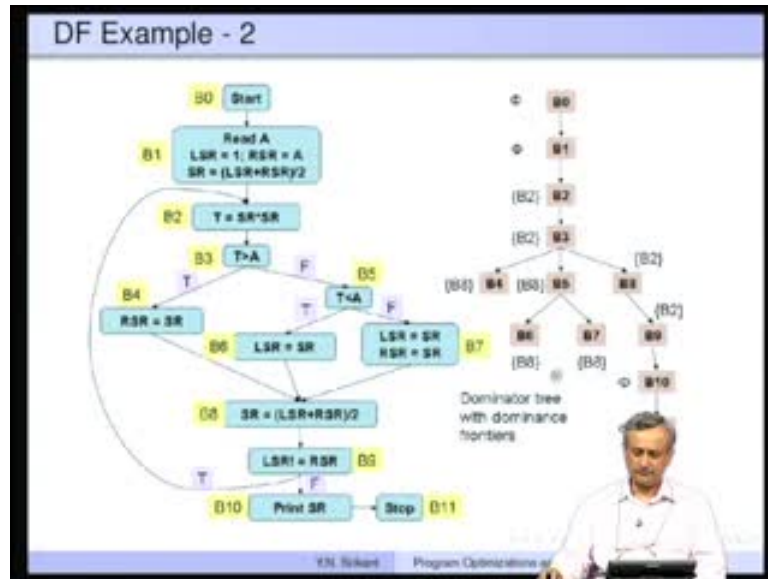
Go to the dominator tree, look at B1. B1 is indeed the immediate dominator of B2. So, we do nothing. We have to stop. We cannot go beyond the immediate dominator. So, you come to B7. B7 is here (Refer Slide Time: 42:17). Now, we can traverse to B3, B2, and then to B1. That is the point where you stop. As we go on traversing these parents in the dominator tree, we are going to add the node B2 into the dominance frontier of each node that we meet in the dominator tree. That is what this is saying (Refer Slide Time: 42:43).

During this traversal, add  $x$  to the DF-set of each node met. So, start with B7, add B2 to its DF, go to B3 its parent in the dominator tree, add B2 here (Refer Slide Time: 43:00), go to B2 the parent of B3, add B2 there as well, and then go to B1. That is the idom of the B2 here (Refer Slide Time: 43:08). So, you stop. Now, we have looked at both the paths for B2. Let us look at the node B7. This is the other join node in the graph. It has two predecessors: B5 and B6. So, let us go to B5 first. We can start from B5 and then the immediate dominator of B7 is B3. So, in one step, we reach B3. So, we can only add the node B7 to the DF information of B5, then we go to B3, and then we need to stop. We cannot do anything more.

Similarly from B6, we add B7 to the DF information of B6, go to B3, and then we stop. This is how dominance frontier is computed in as an algorithm. So, observe that whatever is left over, all others are  $\phi$ . You can actually start from some other node. Let us take for example, B4. So, B4's predecessor is B2, but B2 also happens to be the immediate dominator of B4. So, there is nothing you can do there. This is how all other leftover nodes actually get  $\phi$ ; others have been filled up already.



(Refer Slide Time: 44:39)



This is also a very interesting flow graph. It has two join nodes. Let us look at B2, which is similar to the B2 in previous case. This also has two predecessors: B1 and B9. If you look at B1, it is the immediate dominator of B2. So, nothing gets added. So, we go to B9. From B9, you can go on adding the node B2 to every one of these parents. So, you have added it to B9, you have added it to B8, then you go to B3, you have added it to B3, then you go to B2, you have added it to B2 as well, then you go to B1, **and** it is time to stop. Why? B1 is the immediate dominator of B2. Therefore, we need to stop. Now, we have covered both the predecessors of B2: one is B1 and the other is B9. There is one more join node; that is B8. B8 has many predecessors: one is B4, the other is B6 and the third one is B7.

(Refer Slide Time: 47:50)

**Computation of Dominance Frontiers - 2**

- 1 Identify each join node  $x$  in the flow graph
- 2 For each predecessor,  $p$  of  $x$  in the flow graph, traverse the dominator tree upwards from  $p$ , till  $idom(x)$
- 3 During this traversal, add  $x$  to the  $DF$ -set of each node met

• In example 1 (second previous slide), consider the join node  $B2$ ; its predecessors are  $B1$  and  $B7$

- $B1$  is also  $idom(B2)$  and hence is not considered
- Starting from  $B7$  in the dominator tree, in the upward traversal till  $B1$  (i.e.,  $idom(B2)$ )  $B2$  is added to the  $DF$  sets of  $B7$ ,  $B3$ , and  $B2$

• In example 2 (previous slide), consider the join node  $B8$ ; its predecessors are  $B4$ ,  $B6$ , and  $B7$

- Consider  $B4$ :  $B8$  is added to  $DF(B4)$
- Consider  $B6$ :  $B8$  is added to  $DF(B6)$  and  $DF(B5)$
- Consider  $B7$ :  $B8$  is added to  $DF(B7)$ ;  $B8$  has already been added to  $DF(B5)$
- All the above traversals stop at  $B3$ , which is

VM Wilent Program Optimizations

Let us start from  $B4$ . If you are somewhere here (Refer Slide Time: 46:19), from  $B4$ , you can add  $B8$  to it, and then you go to  $B3$ . So,  $B3$  is the immediate dominator of  $B4$ . So, you need to stop. Then, the other predecessor is  $B6$ .  $B6$  is here. So, you add  $B8$  to  $B6$ , go to its parent, add  $B8$  to it, and then you go to  $B3$ , which is the immediate dominator of  $B8$ . So, you need to stop. The same is true for  $B7$  as well. You add  $B8$  here (Refer Slide Time: 46:59), go to  $B5$ ; nothing to be added, then you go to  $B3$ , and you need to stop. This is because, for  $B8$ ,  $B3$  is the immediate dominator and we need to go up to that point only; nothing beyond that.

Now,  $B7$  is remaining. So, we go to  $B7$ , add  $B8$ ;  $B5$ , we add  $B8$ , then we go to  $B3$ , and we need to stop. So, we have covered all the three predecessors of  $B8$  and we have actually added dominance frontier information for all these nodes. The rest are all going to be  $\phi$ . So, we saw (Refer Slide Time: 47:54) the algorithm on the example.

(Refer Slide Time: 47:59)

```
DF Algorithm

{
  for all nodes n in the flow graph do
    DF(n) = φ;
  for all nodes n in the flow graph do {
    /* It is enough to consider only join nodes */
    /* Other nodes automatically get their DF sets */
    /* computed during this process */
    for each predecessor p of n in the flow graph do {
      t = p;
      while (t ≠ idom(n)) do {
        DF(t) = DF(t) ∪ {n};
        t = idom(t);
      }
    }
  }
}
```

EN Wikit Program Optimizations

Let us look at the algorithm in the algorithmic format. For all nodes  $n$  in the flow graph, initialize the dominance frontier to  $\phi$ ; that is it. So, to begin with, we have already initialized to  $\phi$ . So, (Refer Slide Time: 48:18) these will remain as  $\phi$  all others will get modified.

For all nodes  $n$  in the flow graph do. Actually, as I mentioned, it is enough to consider only the join nodes. There is no need to consider other nodes at all. Other nodes automatically get their DF sets on the way during this process. Now, the main loop is here (Refer Slide Time: 48:44), for each predecessor  $p$  of  $n$  in the flow graph do say – store  $t$  equal to  $p$ , then go on till  $\text{idom } n$ . So,  $t \neq \text{idom } n$  do. So, add to the DF of  $t$  the node  $n$ . So,  $DF$  of  $t$  union  $n$  and then go to the parent  $t$  equal to  $\text{idom}$  of  $t$ . This keeps happening until you go to the immediate dominator of the node  $n$ . So, on the way, you have updated all the DF sets with  $n$ .

(Refer Slide Time: 49:26)

The slide is titled "Minimal SSA Form Construction 1" and contains the following content:

- Compute *DF* sets for each node of the flow graph
- For each variable *v*, place trivial  $\phi$ -functions in the nodes of the flow graph using the algorithm *place-phi-function(v)*
- Rename variables using the algorithm *Rename-variables(x,B)*

$\phi$ -Placement Algorithm

- The  $\phi$ -placement algorithm picks the nodes  $n_i$  with assignments to a variable
- It places trivial  $\phi$ -functions in all the nodes which are in  $DF(n_i)$ , for each  $i$
- It uses a work list (i.e., queue) for this purpose

At the bottom right of the slide, there is a small inset image of a man in a white shirt, likely the lecturer, and a footer that reads "K.M. Srikant Program Optimizations".

Now, we are ready to discuss the construction of the Static Single Assignment form. So far, we have learnt how to compute the dominance frontier information for each node in the flow graph. So, compute using that algorithm. For each variable, *v*, place trivial phi functions in the nodes of the flow graph using the algorithm *place-phi-function, v*. Rename the variables using the algorithm *Rename-variables*. So, these are the three concrete steps in the construction of the minimal SSA form. Why is it called a minimal SSA form?

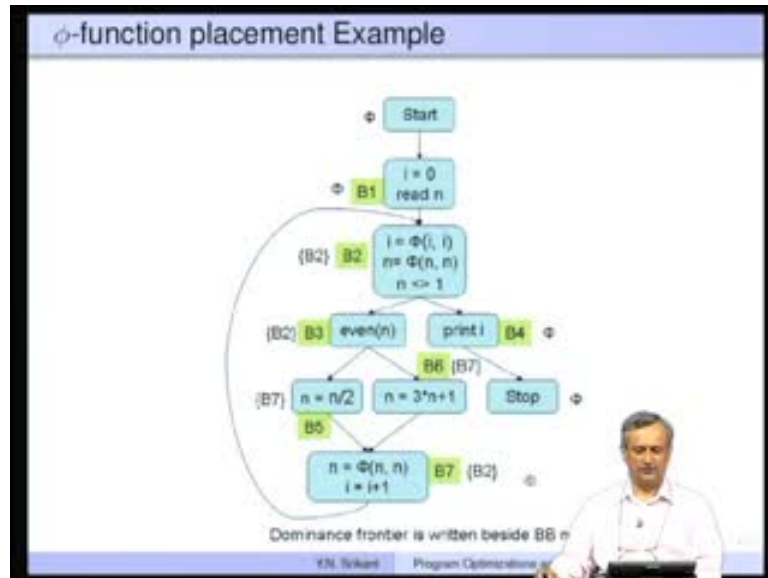
A non-minimal SSA form would possibly place phi functions, which are redundant. It will just place it in every join node for every variable, which enters that node and some of these are redundant; whereas this minimal SSA form places phi functions in only those join nodes where it is essential.

What is the outline of the phi placement process? We are going to look at the detailed version of the algorithm, but let us first get the outline.

The phi placement algorithm picks the nodes with  $n_i$  with assignments to a variable. So, there may be many nodes, which assign to a variable *v*. So, each of these nodes is picked up. It places trivial phi functions in all the nodes, which are in the DF of  $n_i$ , for each  $i$ . So, if there are three nodes where a variable *a* is being assigned, look at the dominance frontier of each of these nodes, go to the DF node and place a trivial phi function there. Trivial phi function is of the form  $\phi$  of *v* comma *v* comma *v*, etcetera. The number of

operands for the phi function is equal to the number of predecessors of that particular join node. This process uses a work list; that is a queue.

(Refer Slide Time: 51:59)



Let us look at an example to understand how it works and then go to the details of the algorithm. In the original example, we had a statement  $i$  equal to and  $n$  equal to, and here we have  $n$  equal to and  $i$  equal to. So, these are the two join nodes that we had seen. When we place phi functions, the trivial function implies  $i$  equal to phi of  $i$  comma  $i$ . So, we want to choose  $i$ , but we have still not renamed the operands of the phi function. That is why this is called as a trivial phi function.

Similarly,  $n$  equal to phi of  $n$  comma  $n$ . We had a statement  $i$  here and we had a statement  $i$  here. So, the dominance frontier of each of these nodes is just this. So, we placed a phi function for  $i$  here. There was a statement read  $n$ , which is an assignment to  $n$  here and there is another statement  $n$  equal to something here and  $n$  equal to something here. So, the dominance frontier of all these happens to be this node and this node. So, we need to place two trivial phi functions in these two (Refer Slide Time: 53:33). So, there is an  $n$  equal to phi of something and  $n$  equal to phi of something here. Similarly, for  $i$ .  $i$  is assigned here and  $i$  is assigned here. So, the dominance frontier of this B1 and B7 is B2. That is why there is a trivial phi function in this particular node; no other node has any trivial phi functions. So, algorithm is simple.

(Refer Slide Time: 53:58)

The function *place-phi-function(v)*

```
function Place-phi-function(v) // v is a variable
// This function is executed once for each variable in the flow graph
begin
  // has-phi[B] is true if a phi-function has already
  // been placed in B
  // processed[B] is true if B has already been processed once
  // for variable v
  for all nodes B in the flow graph do
    has-phi[B] := false; processed[B] := false;
  end for
  W := W; // W is the work list
  // A component node(v) is the set of nodes containing
  // statements assigning to v
  for all nodes B in Component-node(v) do
    place-phi-function(B);
  end for
  while W != {} do
    begin
      B := Remove(W);
      for all nodes y in DF(B) do
        if (not has-phi(y)) then
          begin
            place-phi-function(y);
            has-phi(y) := true;
            if (not processed(y)) then
              begin
                processed(y) := true; Add(W, y); end
            end
          end
        end for
      end
    end
  end
end
```

Dominance frontier is written beside BB in:

Let us study the algorithm in some detail. Place-phi-function; v is a variable. This says – the function is executed once for each variable in the flow graph. So, there is a has phi B, which is a predicate, which is true if a phi function is already been placed in B. Processed B is another predicate, which is true if B has already been processed once for variable v. There is initialization, for all nodes has phi B equal to false and processed B equal to false.

The work list begins with phi and then assignment nodes v is the set of nodes containing statements assigning to v. For all nodes B in assignment nodes v, processed B is true and add it to the work list. Now, pick up a node from the work list, for all nodes y in the dominance frontier of B, if it does not have a phi function correspondingly for this particular variable, place a trivial phi function there, make has phi true. If y has not been processed, now assign it processed equal to true, add this to the work list. This is where some explanation is necessary.

This last statement – Add (W,y) implies that the new phi statement, which has been added to the program may cause more phi statements to be added to the program. That is the reason why we are adding it to the work list. So, one-by-one, the nodes where assignments to the variable v are made is taken out of the work list, its dominance frontier is checked. For each of the nodes in the dominance frontier, you place a trivial phi function. So, if these nodes have not been processed, the ones in which phi functions

have been placed are added to the work list once again and the entire process continues until the work list becomes empty.

This is the example (Refer Slide Time: 56:22) we have seen. So,  $i$  equal to 0 and  $i$  equal to  $i$  plus 1 are the two nodes. Look at the dominance frontier that is here, add a phi function here, but then processing this will not yield anymore phi functions. Similarly, read  $n$ ,  $n$  equal to  $n$  by 2,  $n$  equal to 3 star  $n$  plus 1 are all nodes where we have an assignment to  $n$ . So, look at the dominance frontier of these. Those are these two places (Refer Slide Time: 56:53). We have added trivial phi functions in these two places, but these two together again do not yield new nodes where phi functions have to be assigned. So, that is how phi functions are assigned.

Let us stop the lecture at this point and in the next lecture, we will take up Renaming of Variables. Thank you.