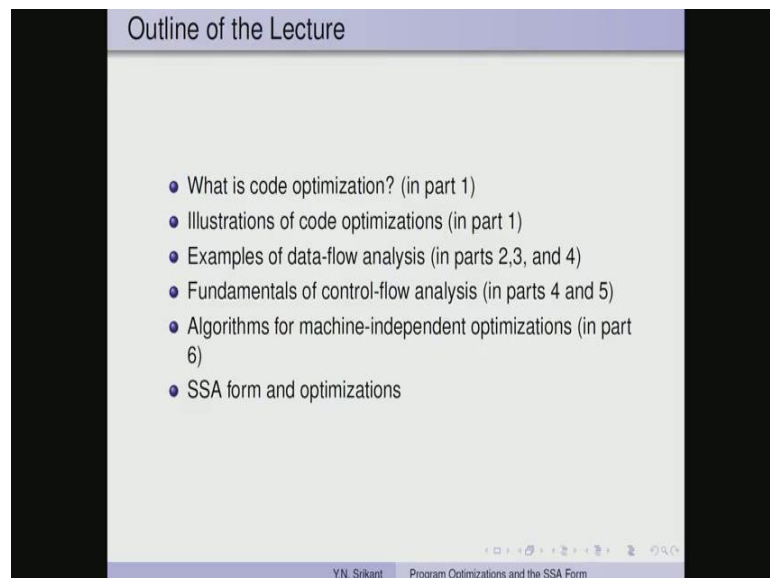


Principles of Compiler Design
Prof. Y. N. Srikant
Department of Computer Science and Automation
Indian Institute of Science, Bangalore

Module - 10
Lecture - 37
Introduction to Machine-Independent Optimizations Part-7
Instruction Scheduling and Software Pipelining Part-1

Welcome to part 7 of the lecture on machine independent optimizations. Today we will continue our discussion on static single assignment form and its application to optimizations.

(Refer Slide Time: 00:37)



Outline of the Lecture

- What is code optimization? (in part 1)
- Illustrations of code optimizations (in part 1)
- Examples of data-flow analysis (in parts 2,3, and 4)
- Fundamentals of control-flow analysis (in parts 4 and 5)
- Algorithms for machine-independent optimizations (in part 6)
- SSA form and optimizations

YN. Srikant Program Optimizations and the SSA Form

(Refer Slide Time: 00:45)

SSA Form: A Definition

- A program is in SSA form, if each use of a variable is reached by exactly one definition
- Flow of control remains the same as in the non-SSA form
- A special merge operator, ϕ , is used for selection of values in join nodes
- Conditional constant propagation is faster and more effective on SSA forms

Y.N. Srikant Program Optimizations and the SSA Form

So, the content of today's lecture would be to look at the, you know constant propagation algorithm, but the variety that we are going to look at is the conditional constant propagation algorithm. To do a bit of recap, so a program is in the static single assignment form if each use of a variable is reached by exactly one definition. So, this is something I already mentioned. The flow of control remains the same as in the non SSA form. Then there is a special merge operator phi, which is used for the selection of values in joint nodes and of course the conditional constant propagation is going to be faster and more effective on the SSA forms.

(Refer Slide Time: 01:28)

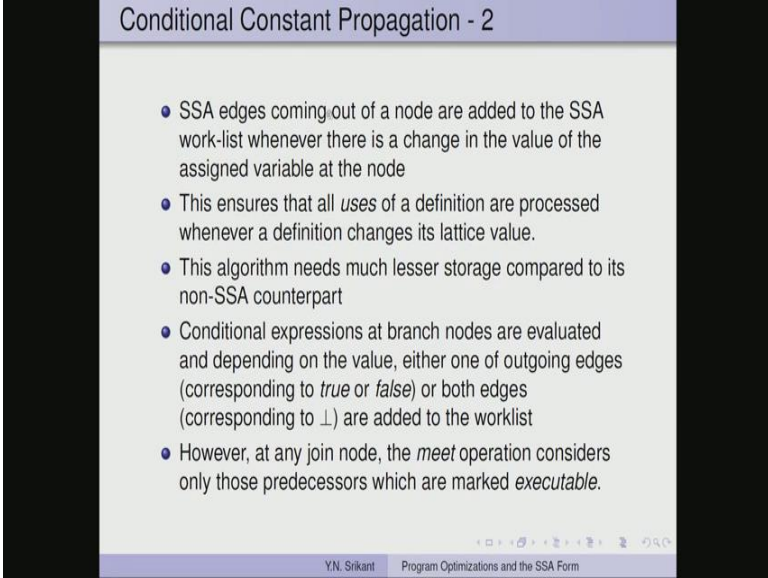
Conditional Constant Propagation - 1

- SSA forms along with extra edges corresponding to $d-u$ information are used here
 - Edge from every definition to each of its uses in the SSA form (called henceforth as SSA edges)
- Uses both flow graph and SSA edges and maintains two different work-lists, one for each (*Flowpile* and *SSApile*, resp.)
- Flow graph edges are used to keep track of reachable code and SSA edges help in propagation of values
- Flow graph edges are added to *Flowpile*, whenever a branch node is symbolically executed or whenever an assignment node has a single successor

Y.N. Srikant Program Optimizations and the SSA Form

So, the conditional constant propagation algorithm itself, it uses a SSA form with extra edges corresponding to the definition use chains. So, these are called SSA edges, we use both the flow graph and SSA edges and maintain two different work lists; one for the flow graph edges, and the other for the SSA edges. So, these are called flow pile and SSA pile respectively. Flow graph edges are used to keep track of the reachable code and the SSA edges are used to help in the propagation of values flow graph. So, edges will be added to the flow pile whenever a branch node is symbolically executed or whenever an assignment node has a single successor.

(Refer Slide Time: 02:22)



Conditional Constant Propagation - 2

- SSA edges coming out of a node are added to the SSA work-list whenever there is a change in the value of the assigned variable at the node
- This ensures that all *uses* of a definition are processed whenever a definition changes its lattice value.
- This algorithm needs much lesser storage compared to its non-SSA counterpart
- 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 \perp) are added to the worklist
- However, at any join node, the *meet* operation considers only those predecessors which are marked *executable*.

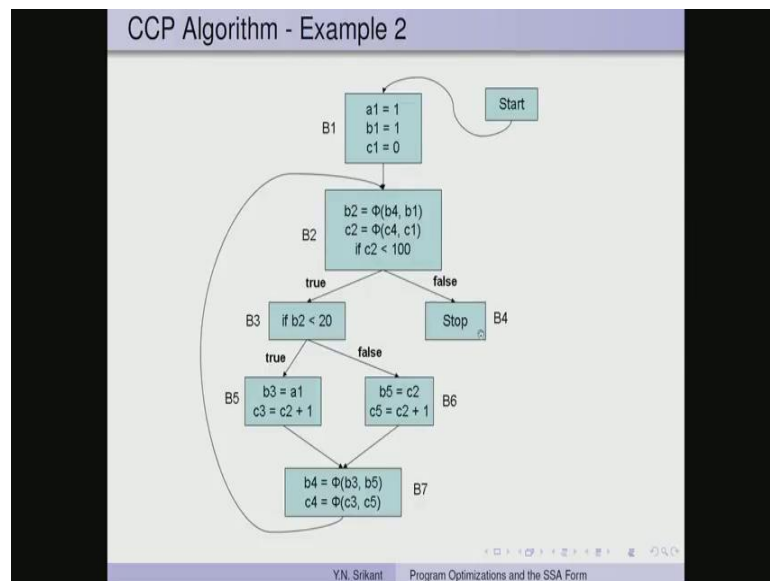
Y.N. Srikant Program Optimizations and the SSA Form

The SSA edges are added coming out of a node are added to the SSA, form SSA work list whenever there is a change in the definition value you know. So, there is the change in the value of the assigned variable change in the value of the definition. So, now the reason why we do this is to make sure that the node which is affected by the change in this value is processed as soon as possible.

So, there is no need to go through all you know all the edges in the flow graph before we arrive at this particular node. So, I usually towards the end of the algorithm there will be very few changes in values and very few nodes will be affected. So, towards this phase of the algorithm it is beneficial if the affected nodes are inform directly, so this ensures that the all the uses of a definition are processed whenever a definition changes its value.

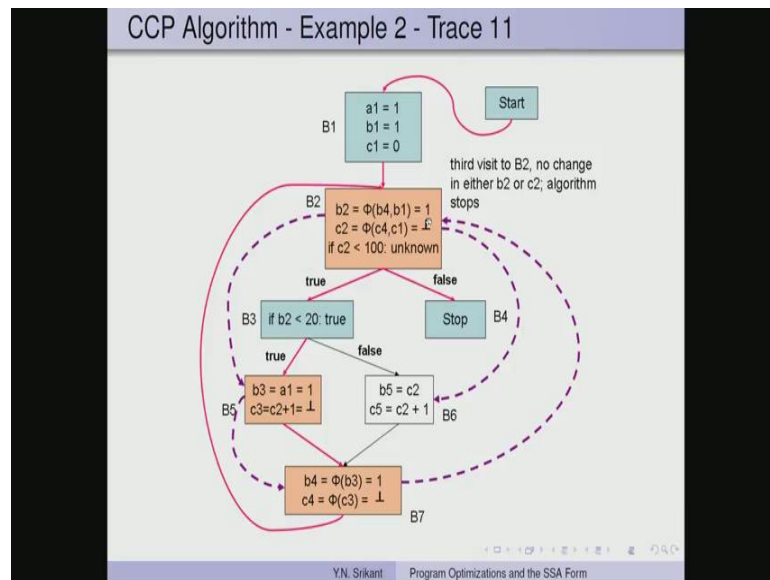
So, this algorithm requires much lesser storage compared to the non SSA counterpart conditional branches at branch nodes are evaluated. So, they have either true value in which case the true edge is added to the, you know flow pile if the false edge is the one to be taken then the false edge is added to the flow pile. But, if the value is not known then both the edges are added to the flow pile, but at a join node the meet operation considers only those predecessors which are marked as executable. So, this allows us to actually you know remove some code which can never be reached.

(Refer Slide Time: 04:15)



So, I gave you this example last time, but we did not work through it, let us do that this time we have initialization of the three variables a 1, b 1, c 1 this is already in the SSA form. So, here we have a phi operator for b 2 to choose the value coming from the back edge or from the top. Similarly, c also has a phi operator to choose the value coming from this side or from that side, so let us see how this works.

(Refer Slide Time: 04:54)



Now, to begin with we add this edge to the flow pile, so when we extract this edge from the flow pile this node is going to be processed. So, we symbolically assign the lattice value as constant 1 for a 1, constant 1 for b 1 and constant 0 for c 1 then, since this is the only outgoing edge from b 1 this will be added to the flow pile. So, correspondingly when this node is when this edge is processed this node will be taken up for processing as well.

So, here we have, you know a phi function, now this node is not yet marked as executable only the red ones are marked executable. So, the value coming from this point is irrelevant to us at this time, so the phi function will consider only the value coming from the top that is b 1. So, b 2 is evaluated as phi of b 1 obviously with just one parameter, it gives you the value of that parameter which is 1 c 2, similarly will give us 0.

Now, c 2 less than hundred can be evaluated to true because c 2 is a value 0 and once that is done the true branch is the only one which is relevant and that is added to the flow pile. So, when we process this edge we process this node and we see that b 2 less than 20 is really true because b 2 has a value 1. So, again the true edges added to the flow pile and when we take up processing b 5 the two assignments are going to be evaluated.

So, that gives us b 3 equal to a 1 equal to 1 because of this value a 1 which has not changed and c 3 is c 2 plus 1 c 2 is again just 1 just 0. Therefore, we get c 2 plus one as 1

and then we add this node to the flow pile and when we actually process this edge this will this node b 7 will be processed. So, when we process this node we get again exactly one executable edge as incoming edge the other one is not yet marked as executable. So, b 4 becomes phi of b 3 corresponding to this incoming edge and that is value of b 3 which is one c 4 becomes phi of c 3 which is the value 1 again.

Now, this edge is marked as executable and that brings us to the node b 2 for a second visit, now during the second visit the value of b 2 actually you know has no change. So, because the value coming from this side is also 1 coming from the top is also one, whereas the value of c 2 changes c 4 from here is 1. Whereas, c 4 from the top is 0, so the meet of these two values is actually not a constant 1 and 0 you know, so from the two incoming edges that would be marked as not a constant. So, once that is marked as not a constant its value has change, so the two SSA edges will be added to the SSA pile, the value of c 2 less than 100.

Now, is unknown because this is a not a constant, so both true and false edges will be marked as executable we had already process this true. So, there is no need to add it to the pile again, but this will be added to the flow pile again, now from, now on since this has already been marked. So, this will not be added processed once more unless the value changes because of the SSA edge, it indeed happens in this case. Now, the flow pile consists of only this particular edge and nothing else and obviously there is nothing to do in a stop node.

So, there is no change as far any of the values are concern, so when we consider the SSA edges and process them, we would be actually looking at the node number b 5. So, when b 5 is evaluated we see that b 3 is now a 1 which is 1 and c 3 is c 2 plus 1 which is not a constant. So, previously b 3 of course was still 1, but c 3 was also 1, now because of c 2 changing its value to unknown rather not a constant, now c 3 becomes not a constant. So, this leads to a change in value for c 3 and that would be you know made known to the algorithm by adding the SSA edge from b 5 to b 7 to the SSA pile.

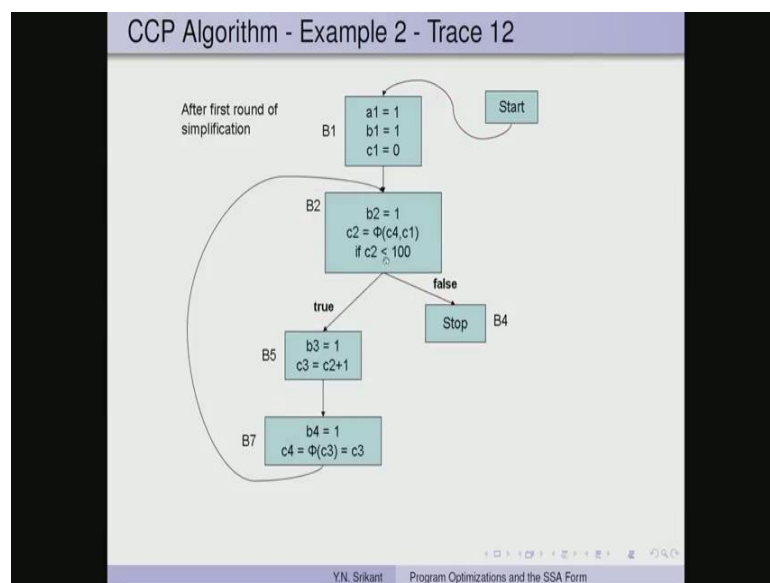
Now, of course we must process this node also because it has been affected by the change in b 2, but it so happens that the incoming edge is not yet marked as executable. So, this processing of this node will not be taken up because it is still unreachable via

executable edges, so nothing happens in b 6 then we process you know we take up this edge right and we process this node b 7.

So, when we process b 7, b 4 is computed as ϕ of b 3 because the other one is still not marked as a executable c 4 now gets the value not a constant because of this value. Now, the c 4 previously had value 1 and b 4 had value 1, now b 4 retains the value 1, but c 4 now gets the value not a constant. So, you know at this point we again have you know, for example there is a change in the value of c 4 and that will be affecting this particular node, so the SSA edge from here to here will have to be taken up.

So, if the third visit to b 2 there is no change in either b 2 or c 2 and of course this expression also remains the same and since there is no change and there are no more edges to be processed in the either the SSA pile or the flow pile the algorithm stops. So, once the algorithm stops we can actually do some optimization such as dead code elimination, etcetera.

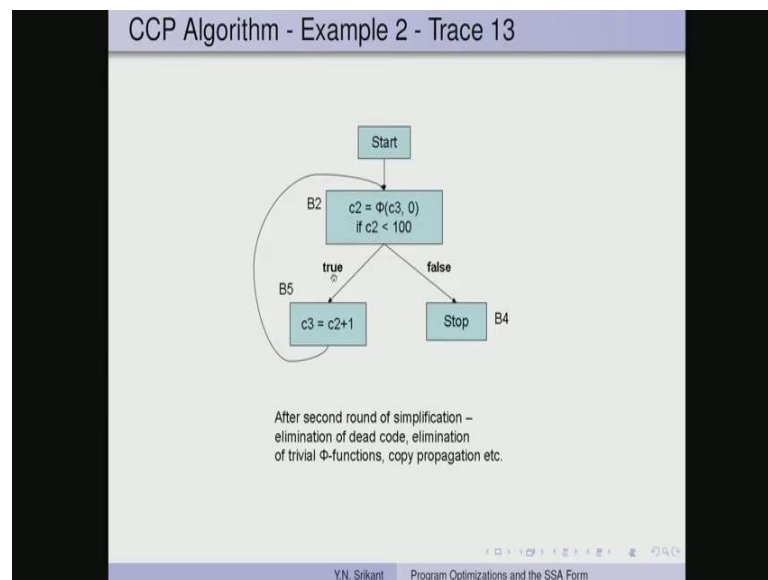
(Refer Slide Time: 12:17)



So, after the first round of simplification we have you know in this case, for example we have b 2 equal to 1 and c 2 of course is not a constant. So, it remains as it is c two equal to phi of c 4, c 1 and this expression could not be evaluated, so it remains as it is both the true and false edges have been added. Now, b 3 remains as 1 and c 3 expression remains as c 2 plus 1 because it became not a constant, here again b 4 becomes remains as 1 and c 4 can be simplified as phi of c 3 because there is only one incoming edge here.

So, that is c_3 itself, so after this round of simplification there is more simplification possible. So, we could eliminate some you know for example b_2 is 1 here, b_3 is 1 here, b_4 is 1 here, so but none of these have been really used anywhere, so we can remove such code.

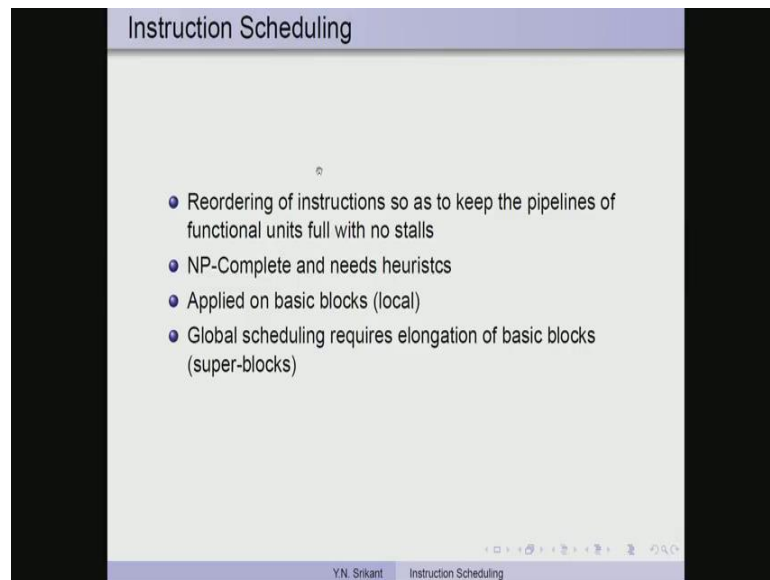
(Refer Slide Time: 13:22)



So, all the constants here are not used, so they can all be kind of thrown away and here we could replace this by a constant itself. So, if we do all this simple modifications to the program using dead code elimination trivial phi function elimination copy propagation etcetera. So, we get the final form of the code which is a very compact piece of code, but remember this is still in SSA form it has a phi function. So, this is how and hyper block scheduling which are useful for you know multiple function unit processors and so on vector processors and so on.

Now, the second type of optimization on machine code is called software pipelining, so instruction scheduling and software pipelining are actually machine dependent optimizations. So, they depend on the machine architecture and machine instructions, so they cannot be performed very effectively before we perform code generation.

(Refer Slide Time: 15:37)



So, what exactly is instruction scheduling well it is just reordering instructions, so as to keep the pipelines of functional units full with no stalls. So, this is the goal of instruction scheduling, so it is nothing but reordering of instructions I will give you an example to show what this means. So, the problem of this reordering is as usual like all good problems or difficult problems in computer science it is an N P complete problem and once it is N P complete. So, we can only apply heuristics to overcome the exponential explosion and the heuristics will obviously be you know will not produce optimal results.

But, they will produce decent results if it is applied on basic blocks alone then it is called local instruction scheduling and if it is applied on several basic blocks at a time such as superblocks. Then it is called global scheduling this requires elongation of basic blocks similar to the extended basic blocks that we studied long back.

(Refer Slide Time: 16:55)

Instruction Scheduling - Motivating Example

- time: load - 2 cycles, op - 1 cycle
- This code has 2 stalls, at i3 and at i5, due to the loads

i1:	r1	←	load a
i2:	r2	←	load b
i3:	r3	←	r1 + r2
i4:	r4	←	load c
i5:	r5	←	r3 - r4
i6:	r6	←	r3 * r5
i7:	d	←	st r6

(a) Sample Code Sequence

(b) DAG

Y.N. Srikant Instruction Scheduling

So, let us take this example, so we have several instructions here, so there are two load instructions t 1 gets a and t 2 gets b then we have t 3 as t 1 plus t 2. So, we have t 4 as load again t 5 is a another add instruction t 6 rather minus instruction t 6 is a you know multiplication and then d gets the value via a store. So, this is the instruction sequence i 1 to i 7 that we are trying to execute if you look at this sequence and then see how the results are used we get this type of a graph this is a dependence graph.

So, for example we have t 1 and then t 2, so t 3 uses t 1 plus t 2, so it has to necessarily wait this computation has to t 1 plus t 2 has to wait until loads of both a and b are complete. So, that is indicated by adding these two arcs to from the two loads to this particular plus add, so similarly the value of this addition is used later in you know in this i 5. So, again we have this you know edge and the load also feeds a value to this operator, so we have another edge from here.

Finally, we have you know an edge from this to this indicating that there is a use of this t 5 in this right and, finally t 6 is used in the store instruction. So, that is used that is indicated by this edge, so the value which is produced here is also used by this multiplication, so that is seen here easily so t three is used here as well.

So, this is the dependence diagram that is relevant for this basic block, now as I mentioned the evaluation of t 1 plus t 2 cannot take place until both t 1 and t 2 are ready that is both these loads are completed. So, if we assume that load requires 2 cycles and if

any other operation requires only 1 cycle then after t 1 is initiated we can go and initiate t 2. Now, t 1 is still in progress at this point when we try to evaluate t 1 plus t 2 t 1 has completed because it has finish 2 cycles, but t 2 has not yet finished.

So, we cannot execute any you know we cannot evaluate this particular plus operator, so this is said to have a stall at this point because we need to introduce a nop instruction to take care of being a, you know idle. Then there is another load here r 4 and the result of that load t 4 is used immediately in the next cycle and because load requires 2 cycles we really cannot evaluate t 3 minus t 4 in this cycle you know. So, immediately after t 4 we have to wait for one cycle and then go to then evaluate it, so at i 3 and i 5 we have two stalls. So, we need to introduce 1 nop after i 2 and another nop after i 4 to make sure that the code executes properly. So, the purpose of instruction scheduling is to try and eliminate such stalls, so let us see how we can eliminate these stalls by reordering the instructions.

(Refer Slide Time: 21:13)

Scheduled Code - no stalls

- There are no stalls, but dependences are indeed satisfied

i1:	r1	←	load a
i2:	r2	←	load b
i4:	r4	←	load c
i3:	r3	←	r1 + r2
i5:	r5	←	r3 - r4
i6:	r6	←	r3 * r5
i7:	d	←	st r6

Y.N. Srikant Instruction Scheduling

So, we have the same sequence of instructions, but then you know these load instructions have been changed, the sequence of these load instructions have been changed. So, we have r 1 here, r 2 here and instead of r 3 we have r 4 right, so here we had r 1, r 2, r 3 and in this we have r 1, r 2, r 4. So, this is the difference, so after r 1 and r 2 actually are initiated, we initiate the next load instruction rather than you know we going directly to the addition.

So, this gives enough time for this load instruction to complete, so at this point, now both r 1 and r 2 are ready they have their values available. So, of course if we had used r 4 here, instead of r 2 we would not have had that value ready, but we are not using it. So, r 1 and r 2, r 1 plus r 2 can be executed directly, now after this by the time we reach r 3 minus r 4 this load would also have completed.

So, r 3 minus r 4 can also be executed in the next cycle of course r 3 star r 5 and store r 6 can all be executed in the following cycles. So, this code requires only 7 cycles and has no stalls at all whereas the previous code it has 7 plus 2 nop, so 9 cycles and it had 2 stalls. So, this is the purpose of instruction scheduling we try to eliminate as many stalls as possible we try to introduce as few nop as possible into the instruction sequence.

(Refer Slide Time: 23:13)

Instruction Scheduling - Motivating Example

- time: load - 2 cycles, op - 1 cycle
- This code has 2 stalls, at i3 and at i5, due to the loads

11:	r1	←	load a
12:	r2	←	load b
13:	r3	←	r1 + r2
14:	r4	←	load c
15:	r5	←	r3 - r4
16:	r6	←	r3 * r5
17:	d	←	st r6

(a) Sample Code Sequence

(b) DAG

Y.N. Srikant Instruction Scheduling

So, if we did have stall here after r 2 and another one just before r 5, this i 5 then you know the pipeline would have been kind of stuck at that point. So, it cannot proceed further until the load is ready, so this is the this will take more cycles and, therefore the speed of the program requires more time to execute.

(Refer Slide Time: 23:38)

Definitions - Dependences

- Consider the following code:
 $i_1 : r1 \leftarrow load(r2)$
 $i_2 : r3 \leftarrow r1 + 4$
 $i_3 : r1 \leftarrow r4 + r5$
- The dependences are
 $i_1 \delta i_2$ (flow dependence) $i_2 \bar{\delta} i_3$ (anti-dependence)
 $i_1 \delta^o i_3$ (output dependence)
- anti- and output dependences can be eliminated by register renaming

Y.N. Srikant Instruction Scheduling

So, let us go through some definitions before we take up the algorithm for instructions scheduling, so let us say there are three instructions i_1, i_2, i_3 , r_1 is load of r_2 , r_3 is r_1 plus 4 and r_1 is r_4 plus r_5 . So, what is significant here is that r_1 is computed into and then used here then you know r_1 is used here and then computed into again r_1 is computed into here and here. So, these are 3 different types of dependences, the first one r_1 being computed into and then used is called as a flow dependence it is indicated as $i_1 \delta i_2$.

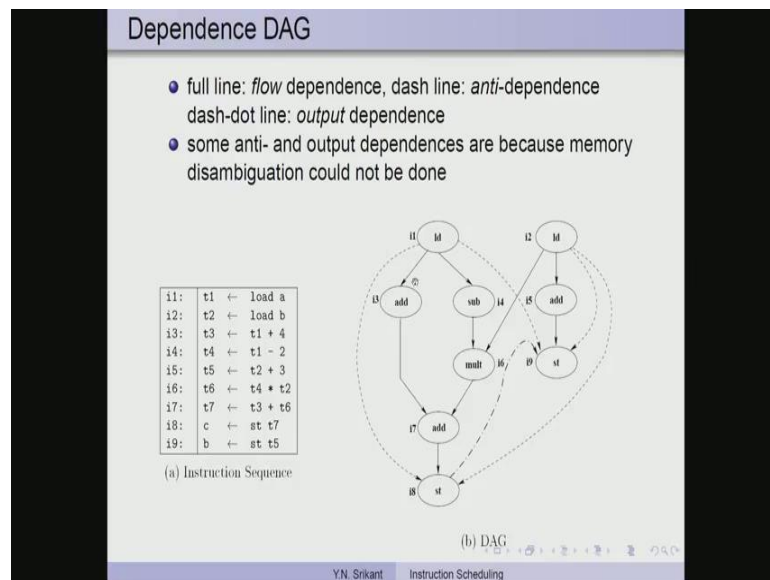
So, the second one you know i_2 and between i_2 and i_3 , we are using r_1 here and then writing into r_1 . So, this is called anti dependence and it is indicated as $i_2 \bar{\delta} i_3$, the third one between i_1 and i_3 for the same r_1 is called as output dependence. So, the dependence is always indicated between instructions here, so $i_1 i_2, i_2 i_3, i_1 i_3$ etcetera the reason why such dependences become important is that parallelization later cannot be performed that is one secondly instructions scheduling.

So, we really cannot schedule this instruction ahead of this instruction because of this dependence then here is another important point. So, output dependences can be eliminated by register renaming flow dependence is also called as a true dependence and it cannot be eliminated by any transformation. But, anti and output dependences can be for example suppose we use r_1 here and r_1' here right, so the r_1 and r_1' let

us say are two different registers. So, in that case there is no ant dependence between these two, similarly suppose we use r 1 here and this is r 1 prime.

So, these two are two different registers and thereby the output dependence is also eliminated such register renaming in some of the machines can be done by the hardware at runtime as well. But, otherwise compilers can perform this register renaming and then eliminate such as well, so in fact if you recall during chaitins register location algorithm. So, we clearly said every live range has exactly one variable, so the second writing etcetera automatically is eliminated you know this is output dependence automatically gets eliminated. So, similarly since we are doing register renaming in that algorithm this anti dependence also gets eliminated.

(Refer Slide Time: 26:49)



So, let us look at the dependence directed acyclic graph a full example, so here is the basic block corresponding to this directed acyclic graph it has these nine instructions. So, we have shown the flow dependence has a full line, for example i 1 to i 2, i 1 to i 4 etcetera and then we have shown the anti dependence in the form of these dash edges. So, the output dependence in the form of these dash dot edges, so what has why we should indicate so many dependences in this particular diagram. So, it so happens that there are two load instructions here and then there are two store instructions and the compiler has not been able to determine that a, b and c are all distinct memory locations.

So, what has happened is the compiler assumes that it is possible to have a b and c as the same memory location, so it says lets add edges between the instruction load instruction. So, the store instruction to indicate that there is a dependence anti dependence, similarly between i 1 and i 8 as well, of course between i 2 and i 8 and i 2 and i 9. Then the output dependence between these two is added because of the same reason, so if the two store instructions use b and c. But, we have no idea that rather the compiler has no idea whether b and c correspond to the same memory location or they correspond to different memory locations.

So, it adds an output dependence edge from this to this, so this is the complete dependence dag and whenever a scheduler goes through this dependence graph and tries to schedule instructions. But, it cannot violate any of these constraints either the flow dependence constraints or the anti dependence constraints or the output dependence constraints. So, let us see how to schedule such basic blocks the reason we want to consider basic blocks is that they are kind of independent entities.

But, we will not consider the effect of the instructions before the basic block and after the basic block we will just assume that each of these instructions in the basic block are connected according to these dependence constraints and then schedule them. So, if we really want to do better work then we will have to look at other basic blocks also that we will do in the advanced instructions scheduling parts of the lecture.

(Refer Slide Time: 29:46)

The slide is titled "Basic Block Scheduling" and contains the following text:

- Basic block consists of micro-operation sequences (MOS), which are indivisible
- Each MOS has several steps, each requiring resources
- Each step of an MOS requires one cycle for execution
- Precedence constraints and resource constraints must be satisfied by the scheduled program
 - PC's relate to data dependences and execution delays
 - RC's relate to limited availability of shared resources

At the bottom of the slide, there is a footer that reads "Y.N. Srikant | Instruction Scheduling".

So, each basic block consists of instructions which are called as micro operation sequences.

(Refer Slide Time: 29:54)

Basic Block Scheduling

- Basic block consists of micro-operation sequences (MOS), which are indivisible
- Each MOS has several steps, each requiring resources
- Each step of an MOS requires one cycle for execution
- Precedence constraints and resource constraints must be satisfied by the scheduled program
 - PC's relate to data dependences and execution delays
 - RC's relate to limited availability of shared resources

Y.N. Srikant Instruction Scheduling

So, each of these micro operation sequences is indivisible in other words the micro operations within the sequence cannot be individually scheduled they will have the entire sequence has to be scheduled as 1 unit. So, that is why because they are indivisible then each M O S has you know micro operations these are the several steps in the instruction and each requiring resources. So, of course each step of the M O S requires one cycle for execution how does all this relate to real instructions in a machine.

So, the M O S is nothing but the pipeline stages of the various micro operations correspond to the various pipeline stages of you know of the machine. So, obviously each pipeline stage requires resources and each pipeline stage executes in one cycle. So, this is a fairly a realistic assumption regarding the operation sequence and this is a fairly realistic modelling of the operation sequence. But, as well there are two types of constraints that we need to show one are called the precedence constraint and the other is called as the resource constraint.

So, the precedence constraints they relate to the data dependences that I already mention you know flow anti and output dependences and they also relate to the execution delays possible because of these dependences. So, load may take two cycles multiply 3 cycles etcetera the resource constraints relate to the limited availability of shared resources. So,

for example the function units you know adders, subtractors, multipliers, load store units, etcetera these are all the various shared resources. So, depending on the availability of these the schedules vary, so the resource constraints in a scheduling problem relate to the availability of limited availability of shared resources.

(Refer Slide Time: 32:13)

The slide titled "The Basic Block Scheduling Problem" lists the following parameters and constraints:

- Basic block is modelled as a digraph, $G = (V, E)$
 - R : number of resources
 - Nodes (V): MOS; Edges (E): Precedence
 - Label on node v
 - resource usage functions, $\rho_v(i)$ for each step of the MOS associated with v
 - length $l(v)$ of node v
 - Label on edge e : Execution delay of the MOS, $d(e)$
- Problem: Find the shortest schedule $\sigma : V \rightarrow N$ such that
 - $\forall e = (u, v) \in E, \sigma(v) - \sigma(u) \geq d(e)$ and
 - $\forall i, \sum_{v \in V} \rho_v(i - \sigma(v)) \leq R$, where
 - length of the schedule is $\max_{v \in V} \{\sigma(v) + l(v)\}$

The slide footer includes the name "Y.N. Srikant" and the course "Instruction Scheduling".

So, what is exactly the formulation of the problem as such, so the basic block is modelled as a digraph G equal to V comma E . So, the nodes V and the edges E have to be explained, now the nodes are nothing but you know micro operation sequences. So, M O S and the edges are nothing but the precedence constraints that we already mentioned of course we are also going to use other notation. Now, for example r is the number of resources in our formulation there is also a label on the node every node has a label.

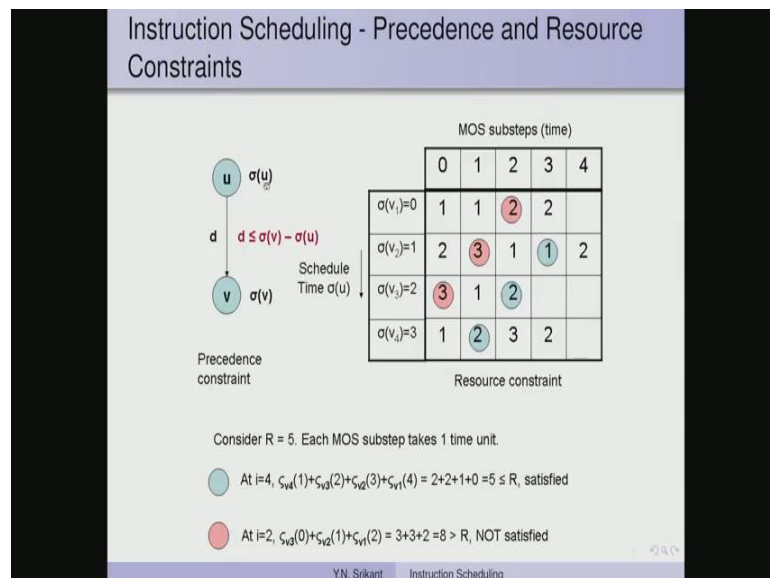
So, what is that label the label is a resource usage function row V of i for each step of the M O S associated with the node V . So, I said there are many micro operations within each sequence right, so each of these micro operations require some resources. So, row V of i for each i , you know will actually tell us the resources use for that particular micro operation if there are 5 micro operations in an M O S then i will range from 1 to 5.

So, we are going to mention the resources required for each of these micro operations for this row V of i of course we also need the length l_v of the node that is nothing but the number of steps in the micro operation sequence. So, there is a label on the edge as well this is the execution delay of the instruction and that will be denoted as d of E . So, for

example the load instruction requires a 2 cycles multiply instruction requires 3 cycles, so these are the delays associated with the M O S the problem is to find the shortest schedule sigma.

So, this schedule is actually a mapping from v to n, so in other words the this n is nothing but the time, so we are going to assign each node to a timeslot such that for all the edges in the graph the precedence constraint is satisfied. So, $\sigma(v) - \sigma(u) \geq d$, I am going to explain it very soon and the resource constraint is also satisfied. So, I will explain this also very soon, now once the schedule is found the length of the schedule is nothing but maximum of $\sigma(v) + 1$. So, take all the nodes find this sum and take the maximum, so that is going to be our schedule.

(Refer Slide Time: 35:20)



So, if you consider the precedence constraints say this is the node u this is the node v and between these two nodes is a delay d. So, this is the label on this particular edge the node u has been assigned a number sigma u that is its schedule the time step node v has been assigned a number sigma v which is its time step at which it can execute. So, this simply says this delay d must be less than or equal to sigma v minus sigma u, so this is quite understandable it simply says that once sigma u is completed.

So, that will once u completes and that will happen only after sigma u plus d steps right, because once u is initiated it requires these steps to time steps to compute. So, sigma u

plus d is equal to σ_v that is the earliest that σ_v can start, so that is what this says σ_v greater than or equal to d plus σ_u .

So, of course it may have to start later than this, you know minimum value of σ_u plus d because of resource constraints. But, that is, now let us understand the resource constraints, so what we have shown here is a table on this side is then you know M O S sub step. So, each M O S sub step actually micro operation executes in one cycle, so we can say that this is also time in, you know the units are time units and on this side are the nodes which have been assign the time units again.

So, let us assume for our example that σ_{v_1} is 0 in other words node v_1 starts the instruction at node v_1 starts its execution at time step 0 then it has 4 micro operations in it. So, at time step 0, the first you know micro operation executes and it requires one resource let us assume that there is only 1 type of resource. So, it requires one resource unit one unit of resource in the second micro operation sequence which starts at micro operation rather at time step 1.

So, this is 0, so this is 1 it requires again 1 unit of resource the micro operation sequence number 2 rather 3 starts at time unit 2 and it requires 2 units of resource. So, the at time step 3 we have the fourth micro operation which requires 2 units of resources, suppose just for the sake of example we assume that node v_2 or the instruction at node v_2 has been scheduled at time step 1. Then for the various it has 5 micro operations in it and each of these require 2, 3, 1, 1 and 2 resources respectively, similarly v_3 has been scheduled at 2.

So, its micro operations require 3, 1 and 2 resources respectively, finally if v_3 is scheduled at 3 and its micro operations require 1, 2, 3 and 2 resources respectively. Now, if you look at the total number of resources available in the machine, let us say it is 5 there is only one resource that is necessary for execution and it is available in you know 5 units. So, when we start this at this point there are no other instructions which are executing so we have requirement of one resource and there are 5 of them.

So, this can execute well at this point the previous step has completed and again we require only one resource out of 5 and this resource has been released after its completion. So, this can also execute and, now when this particular instruction you know number v_1 is executing in its time its second micro operation.

Now, we have already begun the second instruction and that is executing its first micro operation, so in some you know actually whenever we look at the diagonal they correspond to the same time step. So, this is time step 1 micro operation 0 and this is you know, this is micro operation 0 and time step 1, so these two are actually in the same slot of time. So, this requires 2 resources, so at the at time step 1 we have a requirement of 2 plus 1, 3 resources which is still because we have 5 of them.

Similarly, if you continue at time step 2, we have the node v 3 which is executing its micro operation step 1 or a micro operation 1 v 2 is executing its micro operation number 2 and v 1 is executing its micro operation number 3. So, the resource requirements of all these three have to be added up because if this is the same timeslot of 2, so 2 plus 3, 5, 5 plus 3, 8. So, we require eight units of resource what we have available is only 5, so actually speaking such a schedule is not possible let us just for the sake of the example continue.

So, at this point we have v 3, this is micro operation 0, this is micro operation 1 and along this diagonal we have 1 plus 2 plus 2 which is fine you know. So, we require 5 units of resources and that is available in fact even at this point we have 1 plus 1, 2 plus 1, 3 plus 2, 5, so this is also, but unfortunately this required more than what is available, so this schedule does not work, so we may have to introduce some you know dummy instructions in between without rather the nop instruction. So, at these points in order to make sure that the resource constraints are taken care of and schedule comes you know is proper.

(Refer Slide Time: 42:20)

The Basic Block Scheduling Problem

- Basic block is modelled as a digraph, $G = (V, E)$
 - R : number of resources
 - Nodes (V): MOS; Edges (E): Precedence
 - Label on node v
 - resource usage functions, $\rho_v(i)$ for each step of the MOS associated with v
 - length $l(v)$ of node v
 - Label on edge e : Execution delay of the MOS, $d(e)$
- Problem: Find the shortest schedule $\sigma : V \rightarrow N$ such that $\forall e = (u, v) \in E, \sigma(v) - \sigma(u) \geq d(e)$ and $\forall i, \sum_{v \in V} \rho_v(i - \sigma(v)) \leq R$, where length of the schedule is $\max_{v \in V} \{\sigma(v) + l(v)\}$

Y.N. Srikant Instruction Scheduling

So, that is what has been said here, so this sigma really adds up the various resource requirements of the various micro operation sequences at in the various states of their execution. So, that is what this minus operation is really doing, so this gives you the entire resource requirement for the whole machine with many instructions operating at in different states.

(Refer Slide Time: 42:53)

A Simple List Scheduling Algorithm

Find the shortest schedule $\sigma : V \rightarrow N$, such that precedence and resource constraints are satisfied. Holes are filled with NOPs.

```
FUNCTION ListSchedule (V,E)
BEGIN
  Ready = root nodes of V; Schedule =  $\phi$ ;
  WHILE Ready  $\neq \phi$  DO
  BEGIN
    v = highest priority node in Ready;
    Lb = SatisfyPrecedenceConstraints (v, Schedule,  $\sigma$ );
     $\sigma(v)$  = SatisfyResourceConstraints (v, Schedule,  $\sigma$ , Lb);
    Schedule = Schedule + {v};
    Ready = Ready - {v} + {u | NOT (u  $\in$  Schedule)
      AND  $\forall (w, u) \in E, w \in$  Schedule};
  END
  RETURN  $\sigma$ ;
END
```

Y.N. Srikant Instruction Scheduling

So, the list scheduling algorithm is stated here, so the purpose is to find the shortest schedule v to n such that precedence and resource constraints are satisfied if there are

any holes they are filled with nop. So, function list schedule takes a dependence graph this is a actually this is a topological order in sorting algorithm really. So, there is a queue called ready which is used by this algorithm the ready queue consists of all the root nodes of v which do not require anything to execute.

So, these are the top level nodes in the dag, so as I said this is a topological sort, so we start from those nodes which do not require anything to any do not have any precedence require. So, you know constraints that is there are no incoming nodes for this particular root, these root nodes, so those are the only ones which can be executed in. So, you know to begin with right it does not mean that all the nodes in ready queue will be assign. So, the same timeslot not necessarily we have to check many other conditions of resource constraints as well.

So, the schedule to begin with is empty and we continue till the ready queue is empty we get the highest priority node in the ready queue. So, how to assign priorities is the next thing that we need to understand we will do that in a few minutes then for this particular node v which we have picked from the ready queue.

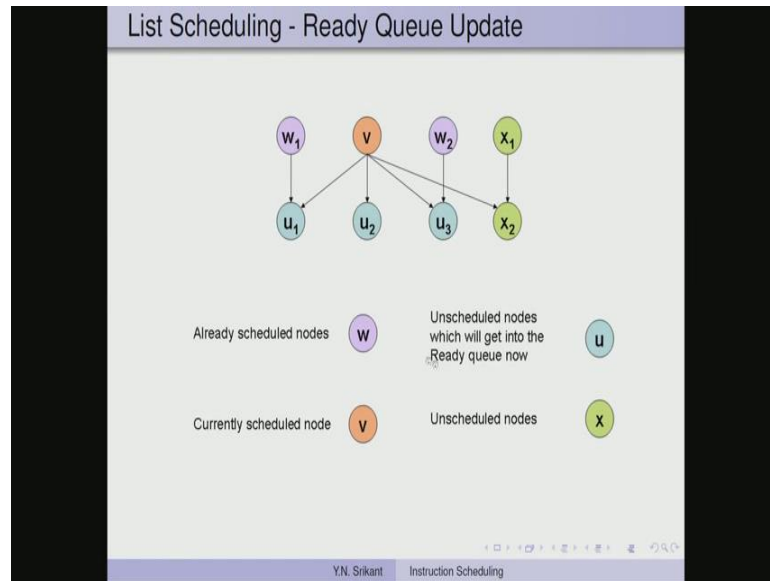
But, we want to find a slot timeslot to schedule it, so to do that there are two things to do, first is find the lower bound for the timeslots of v which satisfy precedence constraints. So, this is a function which is in the next slide, so we will see that v , schedule, σ , so it takes these parameters the node the partial schedule which has been obtain so far. So, of course you know rather these sets schedule that has been a set of schedule nodes and then σ is the schedule itself that is the mapping from the nodes to the timeslots.

So, this gives you this function gives you the lower bound on the time at which it can be the node v can be you know scheduled. So, it does not mean that this is the place this is the timeslot at which we are going to schedule v we still have to check the resource availability. So, σv the slot at which v will be schedules is obtained using the satisfy resource constraints function which takes v . Then the set of schedule nodes the partial schedule σ and also $l b$ as parameters, so depending on the resource constraints which we have mentioned.

Here, you see in this we already mention the resource constraints here the function satisfy resource constraints we will assign a particular slot for v , so it could be $l b$ it could be $l b + 1$ $l b + 2$ etcetera. But, definitely something will be found for this, so

the schedule, now is you know gets v as an extra member because we have already scheduled v at σ . Now, the ready queue losses v , but then we also add all the nodes u which are actually now eligible to be scheduled. So, u is not yet already scheduled that is number 1 not of u in schedule and we have all the edges as w, u such that w is scheduled, so we take all the...

(Refer Slide Time: 47:14)



Now, I will show you a picture here, so this is you know currently scheduled node, now it has been assigned a slot. So, these w are all the already schedule nodes, these 2, now the ready queue will lose v of course, but then it has other possibilities of you know actually to be taken care of. So, u_1 , u_2 and u_3 are 3 nodes which are emanating from v and whose other predecessors w_1 and w_2 have already been scheduled.

Whereas, the successor x_2 still has another node x predecessor x_1 which is not yet scheduled, so this is not yet ready to be schedule. But, u_1 and u_2 , u_2 and u_3 have all their predecessors already schedule this and of course this, so this has only this predecessor this has this and of course this. So, these two have already been scheduled this is just now scheduled, so u_1 and u_2 and u_3 will all be added to the ready queue.

(Refer Slide Time: 48:25)

A Simple List Scheduling Algorithm

Find the shortest schedule $\sigma : V \rightarrow N$, such that precedence and resource constraints are satisfied. Holes are filled with NOPs.

```
FUNCTION ListSchedule (V,E)
BEGIN
  Ready = root nodes of V; Schedule =  $\phi$ ;
  WHILE Ready  $\neq \phi$  DO
  BEGIN
    v = highest priority node in Ready;
    Lb = SatisfyPrecedenceConstraints (v, Schedule,  $\sigma$ );
     $\sigma(v)$  = SatisfyResourceConstraints (v, Schedule,  $\sigma$ , Lb);
    Schedule = Schedule + {v};
    Ready = Ready - {v} + {u | NOT (u  $\in$  Schedule)
      AND  $\forall (w,u) \in E, w \in$  Schedule};
  END
  RETURN  $\sigma$ ;
END
```

Y.N. Srikant Instruction Scheduling

So, that is what this says so this is done for the entire ready queue 1 at a time and then we return sigma.

(Refer Slide Time: 48:37)

Constraint Satisfaction Functions

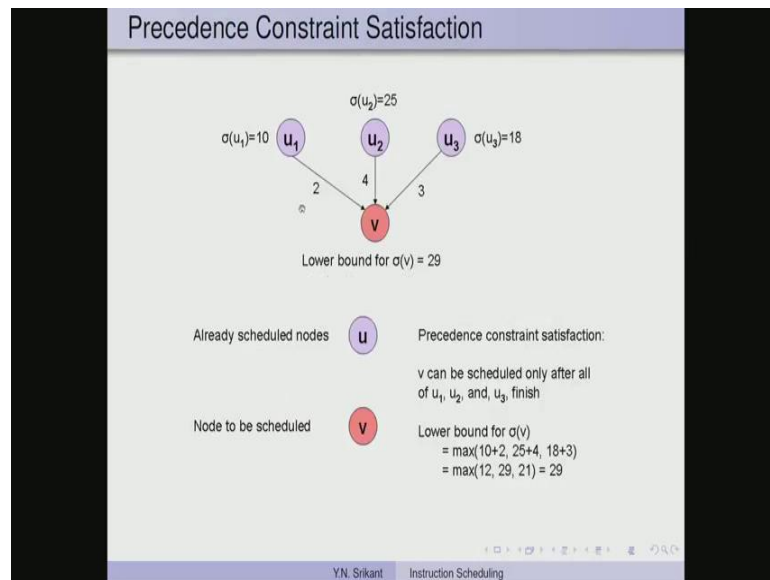
```
FUNCTION SatisfyPrecedenceConstraint(v, Sched,  $\sigma$ )
BEGIN
  RETURN (  $\max_{u \in Sched} \sigma(u) + d(u, v)$  )
END

FUNCTION SatisfyResourceConstraint(v, Sched,  $\sigma$ , Lb)
BEGIN
  FOR i := Lb TO  $\infty$  DO
    IF  $\forall 0 \leq j < l(v), \rho_v(j) + \sum_{u \in Sched} \rho_u(i + j - \sigma(u)) \leq R$  THEN
      RETURN (i);
  END
```

Y.N. Srikant Instruction Scheduling

Now, let us look at the constraint satisfaction functions satisfy precedence constraints, so this you know simply considers sigma u plus d u, v for all the schedule nodes and fix the maximum.

(Refer Slide Time: 48:53)



So, let us see what it means, so this is the precedence constraint satisfaction, so we are here right and u_1, u_2 and u_3 have already been scheduled. So, their schedule timeslots are 10, 25 and 18 the delays for the u_1, u_2 and u_3 those instructions are 2, 4 and 3, so the earliest that we can schedule v is 10 plus 2 or 25 plus 4 or 18 plus 3 the maximum of these. So, obviously it is 25 plus 4 which is 29, so until you know that is because even though these 2 complete, earlier u_2 does not complete earlier than 29 cycles and that is the minimum at which v can be scheduled.

(Refer Slide Time: 49:45)

Constraint Satisfaction Functions

```

FUNCTION SatisfyPrecedenceConstraint(v, Sched,  $\sigma$ )
BEGIN
  RETURN (  $\max_{u \in \text{Sched}} \sigma(u) + d(u, v)$  )
END

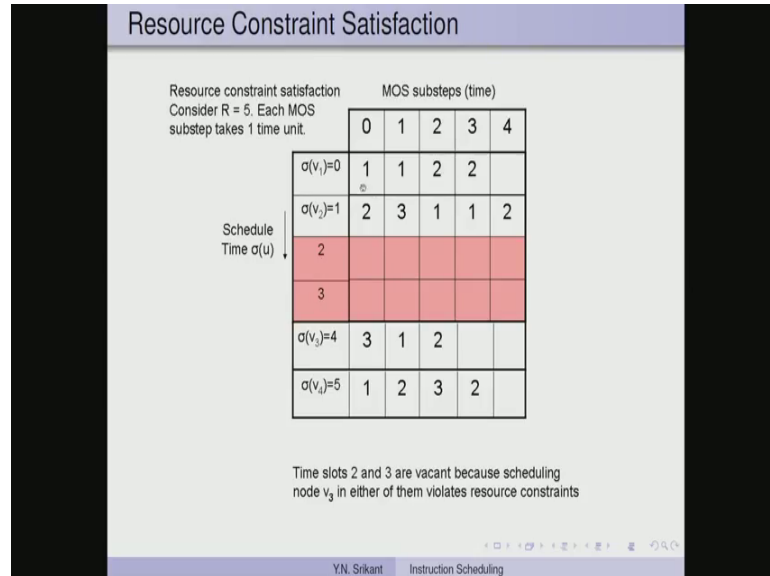
FUNCTION SatisfyResourceConstraint(v, Sched,  $\sigma$ , Lb)
BEGIN
  FOR i := Lb TO  $\infty$  DO
    IF  $\forall 0 \leq j < l(v), \rho_v(j) + \sum_{u \in \text{Sched}} \rho_u(i + j - \sigma(u)) \leq R$  THEN
      RETURN (i);
    END
  END

```

Y.N. Srikant Instruction Scheduling

So, that is what this satisfy precedence constraints really tells us what does satisfy resource constraints tell us, so let us look at the picture again.

(Refer Slide Time: 49:56)



So, the same picture that we had used before you know we had a problem here, these two are not present, these two dummy slots were not present, and this row was actually present at this point. So, we had 3 plus 3, 6 plus 2, 8 as resource requirements, now to take care of the resource constraints we have actually made these 2 dummy slots there are nop instructions here. So, now the resource constraint of 5 is satisfied, so here any diagonal, now has only 5 or less, so this is 5, so this is 2 plus 1, 3 that is it and this has 3 plus 1, 4.

So, here this has 2 plus 1, 3 plus 1, 4, this has 2 plus 2, 4, this has 3 and this has 2, so this schedule with blanks here is a proper schedule and this is precisely what the resource constraint satisfaction function checks. So, it checks at this point whether the total resource requirements are satisfied if not it increments the counter to by 1 and checks again, so it did that for these two found that only 4 is a feasible slot.

(Refer Slide Time: 51:13)

The slide displays two pseudocode functions for constraint satisfaction. The first function, SatisfyPrecedenceConstraint, returns the maximum value of $\sigma(u) + d(u, v)$ for all u in the set of scheduled instructions. The second function, SatisfyResourceConstraint, iterates from a lower bound Lb to infinity, checking for a slot j where the resource usage, calculated as $\rho_v(j) + \sum_{u \in Sched} \rho_u(i + j - \sigma(u))$, is less than or equal to the resource limit R . If such a slot is found, it returns i .

```
FUNCTION SatisfyPrecedenceConstraint(v, Sched,  $\sigma$ )
BEGIN
  RETURN (  $\max_{u \in Sched} \sigma(u) + d(u, v)$  )
END

FUNCTION SatisfyResourceConstraint(v, Sched,  $\sigma$ , Lb)
BEGIN
  FOR i := Lb TO  $\infty$  DO
    IF  $\exists 0 \leq j < l(v), \rho_v(j) + \sum_{u \in Sched} \rho_u(i + j - \sigma(u)) \leq R$  THEN
      RETURN (i);
  END
```

So, that is precisely what it is doing it is trying from Lb to infinity and checks this inequality and then returns the slot at which resource constraints are satisfied. So, we are definitely certain that we will find some slot because you know even though we do not reorder every instruction has a finite amount of time requirement. So, every instruction must finish after a few cycles and after that we will definitely have get a slot to assign to this particular v . We will stop here and continue with this part in the next lecture.

Thank you.