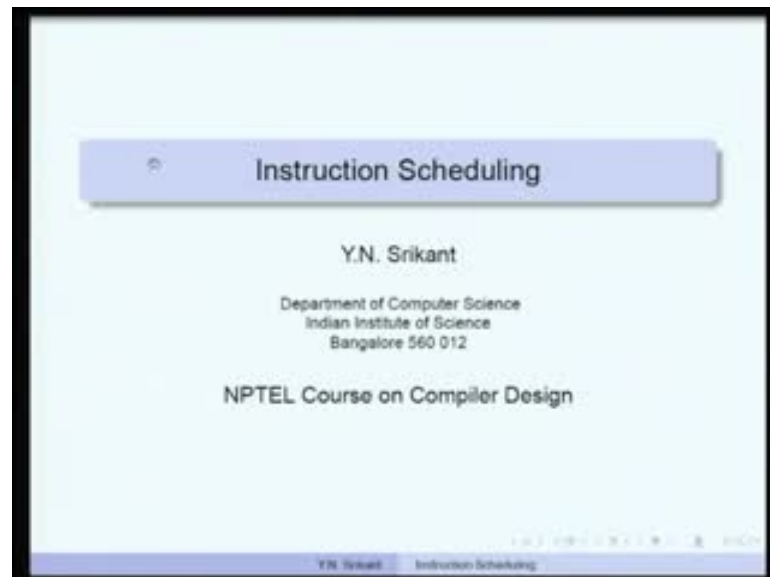


Compiler Design
Prof. Y. N. Srikant
Department of Computer Science and Automation
Indian Institute of Science, Bangalore

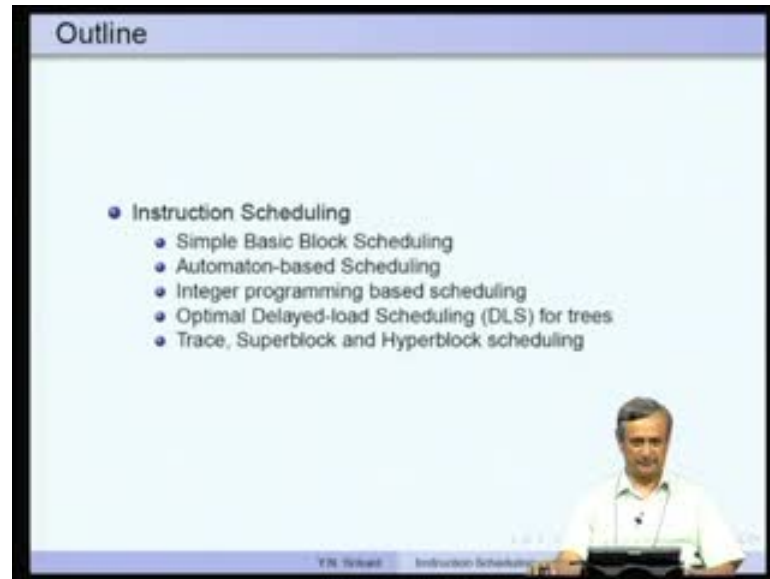
Module No. # 15
Lecture No. # 28
Instruction Scheduling

Lecture on instruction scheduling.

(Refer Slide Time: 00:24)



(Refer Slide Time: 00:26)

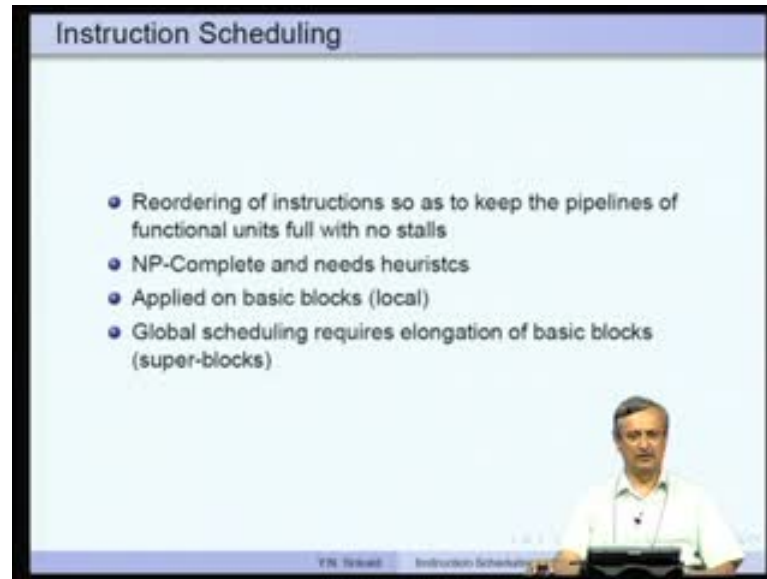


In this lecture, we will be looking at machine dependent optimization; Instruction Scheduling is one of them. We have seen one type of machine dependent optimization before - that is the v pole optimization. That was a very simple optimization - looking at a window of instructions to find patterns and then you know replace that pattern with more efficient sequence of instructions and so on.

Instruction Scheduling is different. There are many types of instruction scheduling mechanisms. We are going to see a few of them. Basically all instruction scheduling mechanisms consider a sequence of instructions in one or more basic blocks - the minimum of course is one basic block - then based on different criteria they try to reorder the instructions.

Why should these instructions be reordered? Basically, we want to make sure that the pipelines in the processor remain full most of the time and any hiccups in this pipeline are avoided. So, we see how best it can be done by using instruction reordering. Simple basic block scheduling, automation-based scheduling, integer programming based scheduling, optimal delayed-load scheduling for trees, and then trace, superblock and hyperblock scheduling are some of the instruction scheduling mechanism algorithms that we are going to consider.

(Refer Slide Time: 02:12)



So, what exactly is instruction scheduling? As I mentioned, it is nothing but reordering of instructions so as to keep the pipelines of the functional units of a processor with no stalls, so no hiccups. Instruction scheduling, in general, is an empty complete problem and therefore there is a need for heuristics to take care of instruction scheduling. If it is applied on basic blocks, then it is called local instruction scheduling and if it is applied on a sequence of basic blocks, then it is called global instruction scheduling. So, global scheduling requires elongation of basic blocks and these are called superblocks, hyperblocks, etcetera.

So, let us look at some motivation. Why exactly instruction scheduling is needed? What are its advantages? Consider this sequence of instructions here. We have seven instructions and we are considering a load-store architecture. In other words, any operand must be in a register. So, we have to load the operand into a register and then use it. That is the model that we are considering here.

(Refer Slide Time: 03:08)

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 * r4
i7:	r7 ← r5 + r6

(a) Sample Code Sequence

(b) DAG

The DAG shows nodes for instructions i1 through i7. i1 and i2 are load nodes with two outgoing edges each. i3 depends on i1 and i2. i4 is a load node with one outgoing edge. i5 depends on i3 and i4. i6 depends on i3 and i4. i7 depends on i5 and i6.

So, time to load is two cycles and each operation requires one cycle. In this code, this is the dependency graph directed as cyclic graph for this particular code. If you observe this code with the latency that is given here, two cycles for load and one cycle for half - we immediately see that there are two stalls, one at i3 and another at i5. Let us see, why?

So, i3 loads - you know r1 plus r2 computes r1 plus r2 and puts it into r3. r1 and r2 are loaded from a and b in the instructions i1 and i2. The instruction i1 would have completed by the time we reach i3, because two cycles would have been completed. But the instruction i2 which loads b into r2 would not have completed, when we come to i3, so, there is a need to skip one cycle here and that is the stall that we are talking about.

Once the result of load instruction in i2 is ready, the computation of r1 plus r2 can take place. Similarly, the instruction i5 - it has r5 equal to r3 minus r4. r4 is being loaded from a memory location c in the instruction i4 - it is not yet complete and therefore, there is a stall at this point. We have to skip an instruction and then execute the instruction i5. So, these seven instructions will really require nine cycles in order to complete.

Now, suppose we want to get rid of these stalls, is it possible to reorder the instructions such that the dependences as given in the directed cyclic graph are still satisfied. In other words, we definitely want to wait for the loads to be completed before we come to i3 - that is the operands must be ready before we come to i3. Similarly, we want to make sure

that r4 is ready before we come to i5; only thing is some reordering of instructions may be possible.

(Refer Slide Time: 06:17)

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

Let us see how? So, here is a new sequence of instructions. There are no stalls here, but dependences are indeed satisfied. How are the stalls eliminated? i1 remains as it is, i2 remains as it is, but i4 is actually replacing the instruction i3 and i3 has been pushed down by one cycle in the step one. Now, when we arrive at the instruction i3, which is r1 plus r2 - both r1 and r2 are ready - the reason is r1 takes two cycles. So, it will be ready by the time i4 is executed - r2 takes two cycles, so that will also be ready by the time i3 is executed.

So, r1 and r2 will now contain operands. There is no stall. r4, of course, would not have completed by the time we execute i3, but it is not required the instruction i3. So, once we finish i3, we execute i5 which requires r4 and r4 would have got its operand by the time we reach i5, because 2 cycles have elapsed.

So, this particular sequence of instructions actually completes just in seven cycles. It does not take nine cycles at all. That is, because we have just swapped these instructions i4 and i3 - all thus have remained the same. So, i4 is in this slot with a load instruction and i3 is in this slot with the computation instruction.

So, this is a very simple example, which shows that a certain change in the order of your execution of the instructions will lead to removal of the stalls. That does not mean every stall can be removed. There will be some stalls which cannot be removed, as we will see in later examples. We will have to insert **NOPs** in those cycles.

(Refer Slide Time: 08:20)

Definitions - Dependences

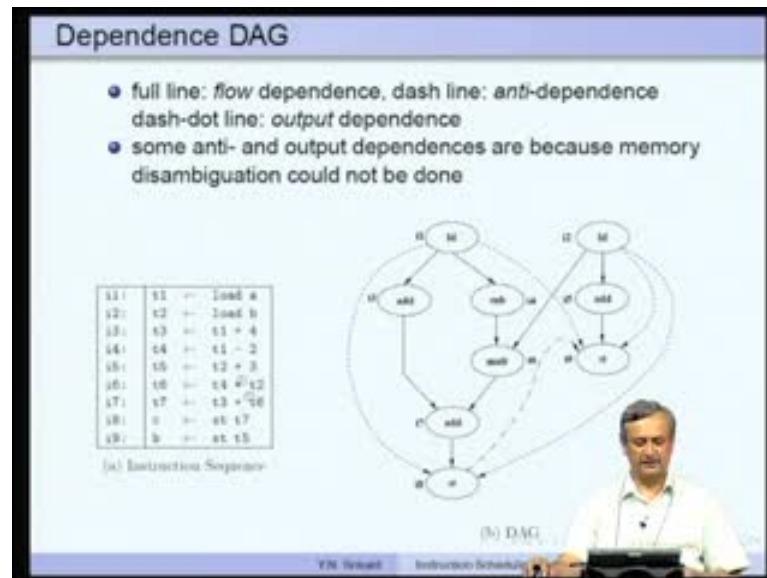
- Consider the following code:
 - $i_1 : r1 \leftarrow \text{load}(r2)$
 - $i_2 : r3 \leftarrow r1 + 4$
 - $i_3 : r1 \leftarrow r4 + r5$
- The dependences are
 - $i_1 \delta i_2$ (flow dependence) $i_2 \bar{\delta} i_3$ (anti-dependence)
 - $i_1 \delta^o i_3$ (output dependence)
- anti- and output dependences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

So, quickly to recap at what we need here, we definitely want to look at the same dependences, flow and t and output as we studied in parallelization. But these are now at the register and the load levels; they are not at the higher levels but between statements etcetera. They are at the machine instruction level, for example, i_1 is $r1$ is loaded with $r2$, so then i_2 is $r3$ equal to $r1$ plus 4 and i_3 is $r1$ equal to $r4$ plus $r5$.

So, here the value which is actually loaded into $r1$ is used in i_2 , so $i_1 \delta i_2$ holds. Similarly, whatever is read from $r1$ is replaced in instruction i_3 so there is an anti-dependence between i_2 and i_3 , so that is $i_2 \bar{\delta} i_3$ and then we also have $i_1 \delta i_3$, because we are actually loading into the same register $r1$. i_1 and i_3 are related by output dependence.

So, anti and output dependences can be eliminated by register renaming; we have seen some examples of this in parallelization. For example, here we do not use $r1$ we just use say may be $r6$ or something like that then automatically there is no output dependence between these two, so that automatically eliminates this anti dependence as well.

(Refer Slide Time: 10:09)



So, what is a dependence direct acyclic graph? It is the same dependence graph as we studied in parallelization - it is just that here we have at the machine instruction level. Here is a simple example. This is the sequence of instructions which is running example that we are going to use.

i1 to i9 - nine instructions. There are nine instructions here. Nine nodes here. These are the nodes of the direct acyclic. Each instruction is a node, there is a solid arc from one node to another node and that shows a flow dependence. For example, between i1 and i3, t1 is equal to load a, t3 equal to t1 plus 4, so there is a load flow dependence. Similarly there is another flow dependence between i1 and i4 and so on and so forth.

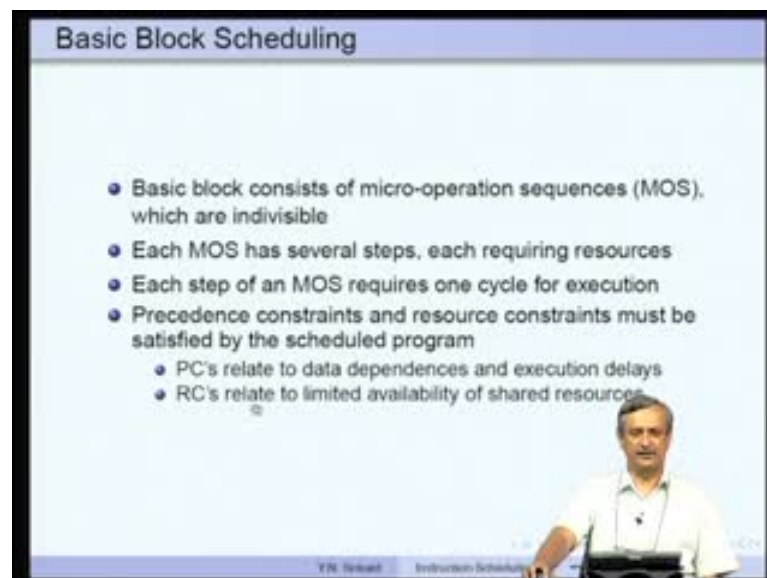
So, all the flow dependences are shown in solid lines; then the anti-dependences are shown using the dash line. This is the dash line from i1 to i9. Here is i1 and here is i9. You may wonder, why load a in i1 and b equal to store t5 in i9 have been linked by an anti-dependence. The reason is memory disambiguation, which says that this a and this b are different, has not be done, so no algorithm has been applied to find out whether a and b are the same location are or different locations.

So, because of that conservatively we will have to assume that there is an anti-dependence from i1 to i9, because a and b could be the same. Similarly, b of course, is the same so, i2 and i9 also have a dependence. This is i2 and this is i9.

Similarly, you know **i1 and i8**. i1, i2 and i8 also have similar anti-dependence here and here. So, these are the anti-dependences that need to be depicted in this dependence diagram. Then, there may be output dependences so those are shown by dash dot line. So, here is an output dependence, an output dependence is between these two and again memory disambiguation has not been done, so we do not know that b and c are different memory locations, so there is an anti-dependence from i8 to i9.

So, the dependence diagram shows all these. Now, the problem of instruction scheduling would be to rearrange instructions in this sequence so as to respect all these flow, anti and output dependences, yet minimize the number of stalls that may occur in the sequence.

(Refer Slide Time: 13:27)



So, the first algorithm that we are going to study is the basic block scheduling. What is our model? We need to study the model of a basic block. A basic block is assumed to consist of what are known as micro operation sequences MOS which are indivisible.

In other words, if an instruction contains three or four micro operations in it, once we start that instruction, all the micro operations in it will have to be completed and there is no way we can divide this sequence of four micro operations into two each and schedule them separately.

So, once the instruction is started, all the micro operations will run one after another. They are indivisible. Each MOS has several steps and each one of these steps will require resources. Each step of an MOS requires one cycle for executions. This is the assumption. There are precedence constraints and there are also resource constraints. These must be satisfied by the reordered or scheduled program.

We have the diagram here. This shows the precedence constraints. Once we say there is a delay between these two, the delay on this edge could be the time of execution of the load instruction, automatically this becomes a weighted directed acyclic graph and that shows all the precedencies that must be satisfied.

The precedencies relate to data dependencies and execution delays as I mentioned just now. But the dependence diagram does not show the resource constraints. So, resource constraints relate to limited availability of shared resources. In other words, I want to execute this, add, sub and this add and there is only one adder, even though we may find that these three can be executed at the same step, it is not possible for us to schedule them in the same step, because there is only one adder. Compulsorily we may have to schedule them in three successive cycles. That is what resource constraint is all about.

(Refer Slide Time: 15:59)

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 $\max_{v \in V} \{\sigma(v) + l(v)\}$

Here is the basic block scheduling problem. First of all, the basic block is modeled as a digraph G with a set of nodes V and a set of edges E and R is the number of resources that we are going to have in the system in the processor. The nodes of the graph G are

micro operation sequence and the edges of graph E show the precedences. There is also a label on the node V ; instead of having on the edge alone we also have a label on the node.

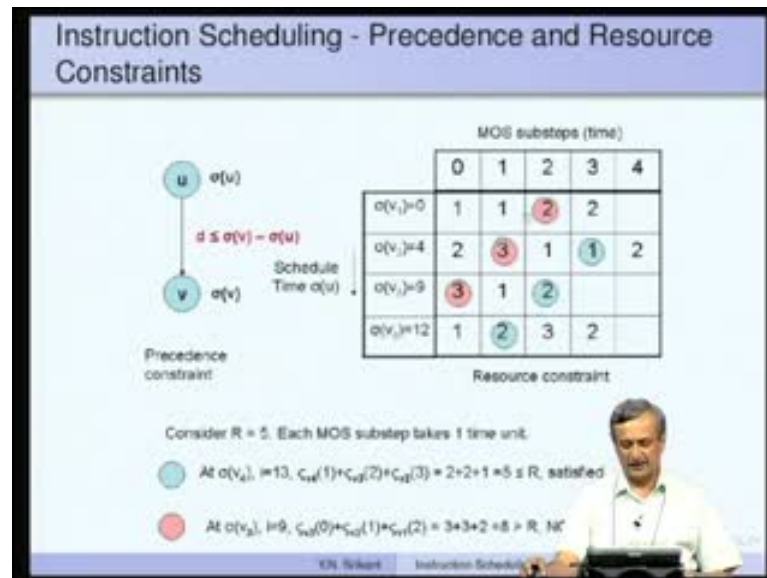
So, the label on the node indicates a resource usage function. What is this? There is a resource usage function $\rho_{v,i}$ for each step of the MOS associated with i with the node v . In other words, if there are four steps in an MOS associated with node v , each of these four steps will require resources, so $\rho_{v,1}$ says how many resources of a particular type are needed for this particular micro operation sequence step, $\rho_{v,2}$ similarly, for the second MOS step and so on and so forth.

We are considering only a single type of resource; r is the total number of resources, so it is a simplified problem as such. If we have many types of resources, it is definitely possible to extend this problem to take care of those as well. It is just a question of checking all the resource constraints at a particular point in time.

So, instead of just one vector of resources, rather one, number indicating the number of resources, we would have a vector of these numbers saying how many are there in each particular type. So, the length l_v of a node v shows how many substrates are included in the micro operation sequence at node v . Label on edge e , any particular edge e shows the execution delay of the MOS and it is denoted by d of e . We have the resource requirements of a particular node v , e indicated by $\rho_{v,i}$ for each sub step i that node and we have on the label on the edges coming out of the node v or incoming edges of node v , the execution delay of a particular that MOS.

Now, the formal notation and description of the problem is to find the shortest schedule σ . σ is a mapping from v to n . For each node v , we need to find time slot. The time slot is nothing but a natural number. Which is the time slot in which v can be executed - that is the schedule that we want to find. For each node v , we want to find a particular number of that kind.

(Refer Slide Time: 19:45)



There are now restrictions. What are the restrictions? For all the edges e , say u comma v in the set of edges e , $\sigma(v)$ minus $\sigma(u)$ must be greater than or equal to d of e . What does this tell us? Let us look at the picture.

So, here is a node u , here is a node v , u has been scheduled at the time slot $\sigma(u)$, let us say, v has been scheduled at the time slot $\sigma(v)$, obviously, $\sigma(v)$ comes later than $\sigma(u)$, because u must be executed before v . There is a precedence constraint, rather edge connecting u and v , which says u must be completed before v .

The delay on the edge u v is d . So, this delay d must be than less than or equal to the schedule of v , which is $\sigma(v)$ minus the schedule of u , that is $\sigma(u)$. This is quite easy to understand, because we cannot start v before we complete u . How much time does u take? It starts at the slot $\sigma(u)$ and then it requires at least d slots more in order to complete. So, $\sigma(u) + d$ is the minimum time at which we can start. That is why, d less than or equal to $\sigma(v) - \sigma(u)$.

(Refer Slide Time: 20:54)

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 $\max_{v \in V} \{\sigma(v) + l(v)\}$

So, that is the precedence constraint and sigma v minus sigma u greater than or equal to the delay on the edge e. What about resources? This says, if you sum up all the resource requirements at a particular point in time, all the resource requirements must be less than or equal to r.

So, let us understand this better. Let us say, we have this matrix; each row indicates a MOS corresponding to a node, say v1 v2 v3 v4 etcetera. So, each one of these is a MOS and since we know that one time slot is required for each one of these MOS sub steps, so here is the time slot of the MOS sub steps.

This is not a valid schedule. It violates certain conditions, but this will tell you what the resource constraint is all about. Let us say, the first MOS, v1, has been scheduled at time zero, second MOS at time four - one, two, three, four - you know 4 steps are needed, 4 time slots are needed for sigma v1 to complete. Sigma v3 cannot be started before the fourth time slot. Then, we require five time slots for v2 so v3 starts at nine, then we need three for v3 so v4 starts at twelve. We have taken care of all the precedence constraints, the delays and so on, but then this did not check the resource requirements.

So, let us consider the red diagonal and the blue diagonal. First look at the blue diagonal, let us say, we are looking at v4 in its second time step rather sub step, so, at this point the elements in the matrix really show how many resource elements are needed for that

particular sub step. For example, σv_1 in time slot zero requires one resource in time slot one require one resource, in time slot two requires two resources etcetera.

So similarly, v_4 requires one, two, three and two resources in the time slot zero; one, two and three are starting at twelve; so at this point in time, we are looking at this time slot twelve, time slot thirteen, fourteen, and fifteen. In time slot twelve, we require two resources because of v_4 , but remember v_3 has not yet completed; v_3 started at nine, so it is actually requires nine, ten, eleven - so these are all the time slots that are needed for this particular node to complete. So, what we really say is let us see what are the various timing requirements and resource requirements of these slots.

So, for example, this requires two here, two here and two here - these are all the concurrently executing threads that we have, so sum of all these is five. So, assuming that there are five resources in this particular system, so the number of resources that we are using at a particular time, let us say, is five; so, we have five resources and utilization is also five. So, the resource constraints are satisfied.

But when you look at the red resources - red circles here - we find that there is a violation. If you add up these three, and three, and two - we really make it eight. Because there are eight resources being used, but there are only five - that means the number of resources available is less than the number of resources that we are trying to use. So, there is a violation.

So, that is precisely what I said. This is not a valid schedule. If these were to be a valid schedule, this would not have occurred. You would actually a summed up these things to be less than or equal to five; so, this is not a valid schedule, because the resource requirements at a particular point in time are actually beyond what is available in the system.

So, that is what this is. $\rho v_i - \sigma v_i \leq r$. That is what it should be, but it is not. The length of the schedule, a valid schedule would be maximum of $\sigma v_i + l v_i$ where v_i is in v . So, we just take the maximum σv_i and $l v_i$, that is not the maximum of whole thing, we know one of the previous steps may have a smaller σv_i but a larger $l v_i$, so that may contribute to the maximum of the schedule.

So, we have demonstrated that at some point in time, there may be satisfied resource requirements, but at some other points in time, there will be violation of resource requirements. So, this should never happen, resource requirements must be satisfied at every point in time.

(Refer Slide Time: 27:11)

```

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  $\forall (w,u) \in E, w \notin$  Schedule};
  END
  RETURN  $\sigma$ ;
END

```

So, how do we schedule the instructions so that the resource constraints and the precedence constraints are both satisfied? Here is a simple algorithm. This is a variation of the very well-known list scheduling algorithm that people have studied in operating systems, in job shops scheduling and so on. The requirement is to find the shortest schedule sigma which is a mapping from v to n such that precedence and resource constraints are satisfied and whenever there are holes we assume that they are filled with NOPs..

So, here is a basic outline of the function called list schedule, which takes the dag as its input v comma e. How does it work? It really does a topological sort of the graph. It starts with those nodes in the graph which do not have any predecessors, those are the root nodes of v, so the dag may not be a single component, you know, it may be a forest.

So, in such a case, the number of root nodes of the graph will be more than one. There is a queue called, ready queue which shows all the nodes which are eligible to be scheduled. Root nodes of v are eligible to be scheduled, because they have no predecessors and when we begin with the root nodes of v, all the resources are assumed

to be available. It does not mean that every node in the ready queue can be scheduled in the same time slot, they still have not found time slot for all the nodes in the ready queue. So, we just come from the top and include nodes as we go along.

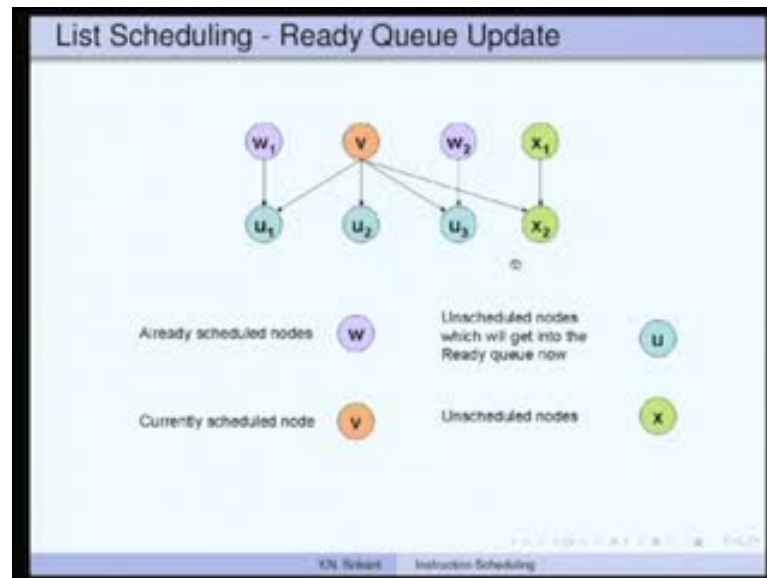
The schedule to be begin with is five, it has nothing. So, there is a loop while ready not equal to five do, in which we actually take out nodes from the ready queue and then schedule it, put something else in to the ready queue and keep doing this until all the nodes in the graph are scheduled. So, in that case ready node cannot get anymore nodes, so we stop when ready becomes five.

Pick a node of highest priority, node from the ready queue and call that as v . How to do this? We will see a little later. Let us assume there is a prioritization of nodes in the ready list and one of the highest priority nodes is picked.

Then, we need to compute the lower bound on the time slots at which the node v can be scheduled. So, this is done by the function SatisfyPrecedenceConstraint which we are going to see very soon. v comma schedule comma σ . Once we have found the minimum time slot in which v can be scheduled, this is done by looping at the precedence requirements, we need to check whether that the minimum time slot satisfies resource constraints, whether all the resources which are needed are available to us and if so, fine - otherwise, we need to actually keep that slot vacant, take the next slot - check whether resources are available at that point and so on and so forth.

So, this is done by the routine satisfy resource constraints v comma schedule comma σ comma Lb . Once we have found σv , which is the time slot, which satisfies both the precedence and resource constraints, we include that node as a scheduled node, so schedule equal to schedule plus variable.

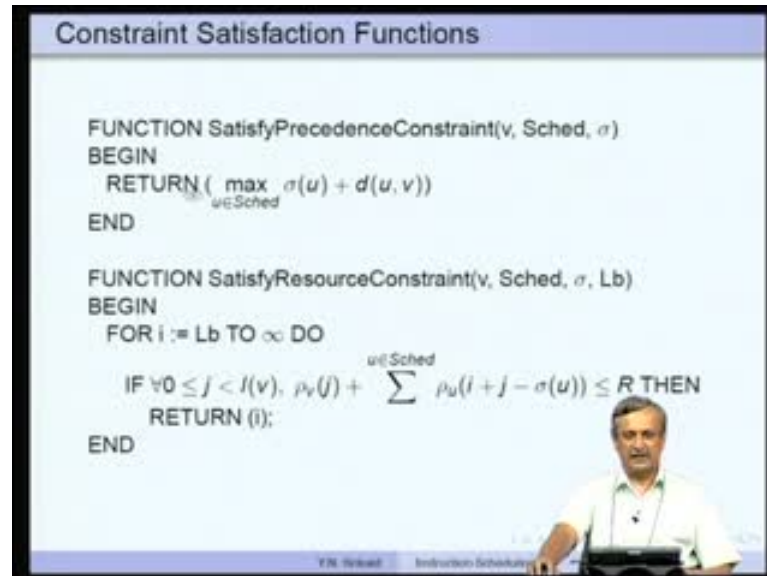
(Refer Slide Time: 32:11)



Then the ready list needs to be updated. What do you mean by updating? We take out the node v from the ready list here, remember we did not take it out, take it out then you try to add those successors of the node v , which actually are now ready to be scheduled. In other words, their own for example, v is the node to be is scheduled, which has been just now scheduled - there are three successors u_1 u_2 and u_3 for this particular node, then w_1 and w_2 are two other predecessors of u_1 and u_3 , they have already been scheduled - so u_1 was waiting for v_2 be scheduled, now v also has been scheduled - so u_1 is now ready for scheduling. u_2 has only v as its predecessor so u_2 is also ready for its scheduling. u_3 is similarly, ready, because w_2 is already scheduled. v is now just now scheduled, so this is also ready for scheduling.

But another successor of v which is x_2 , has a predecessors x_1 , which is not yet scheduled. So, even though v has been scheduled, x_2 cannot be scheduled, because x_1 has not yet been scheduled. It can be scheduled only after x_1 is scheduled, so that is what we do here. We take out v and then include all those nodes u , such that u is not in scheduled list, so obviously u should not have been scheduled already. Then for all w comma u in e , uw must be already scheduled. So, for this node, all its predecessors must be scheduled. w is a predecessor of u and u is now being considering, putting into the ready list, so all its predecessors must already be scheduled. That is what we said here, x_1 is not yet scheduled, so x_2 cannot be included in the ready list, but u_1 , u_2 , and u_3 will be included in the ready list.

(Refer Slide Time: 34:02)



Constraint Satisfaction Functions

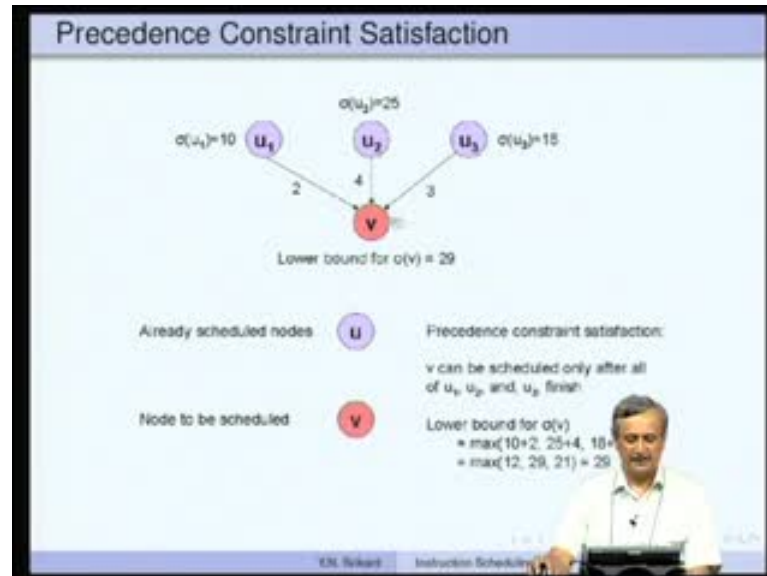
```
FUNCTION SatisfyPrecedenceConstraint(v, Sched, σ)
BEGIN
  RETURN ( maxu ∈ Sched σ(u) + d(u, v) )
END

FUNCTION SatisfyResourceConstraint(v, Sched, σ, Lb)
BEGIN
  FOR i := Lb TO ∞ DO
    IF ∀ 0 ≤ j < l(v), ρv(j) + ∑u ∈ Sched ρu(i + j - σ(u)) ≤ R THEN
      RETURN (i);
  END
```

Now, what are the two functions - precedence constraints and satisfy resource constraints do? They consider $\sigma(u) + d(u, v)$ over all the schedule nodes u and then return the maximum of this value. Let us see what it really means.

We are looking at v and now we are looking at all the predecessors of v , so that is what this said, $\sigma(v)$ is our node we are looking at u comma v , where u is a predecessor, so all the predecessors which have been scheduled - you know that they are scheduled because otherwise we would not have included v in the ready list - so these are all the scheduled predecessors already, now they have been scheduled at 10, 25, and 18.

(Refer Slide Time: 34:29)



So, what is the time slot at which v can be lower bounded at which v can be scheduled? Look at this, this is ten, the delay is two, so ten plus two is twelve, this is twenty five plus four is twenty nine, and eighteen plus three is twenty one, so twenty nine is the large value, so before the time slot twenty nine, we cannot really schedule v . Why? If you do that, then u_2 would not have completed by that time, so we take twenty eight for example, u_2 completes at twenty nine, so the values computed by u_2 will not be available to v , therefore, twenty eight is an illegal slot.

So, that is what this really is all about. So, the max in this case was this twenty nine, others did not contribute to the maximum. So, this gives you the lower bound at which the node v can be scheduled.

But the precedence constraint on its own will not tell us whether the resources available for executing v at that lower bound value, say of twenty nine. The function SatisfyResourceConstraint checks this availability. So for example, we started at 1, at every time slot, $Lb + 1$, $Lb + 2$, $Lb + 3$ etcetera from that point onwards is checked, this says infinity, but you really do not go to infinity, because there are only finite numbers of instructions. Each instruction has a finite number of MOS steps and we know that every instruction terminates.

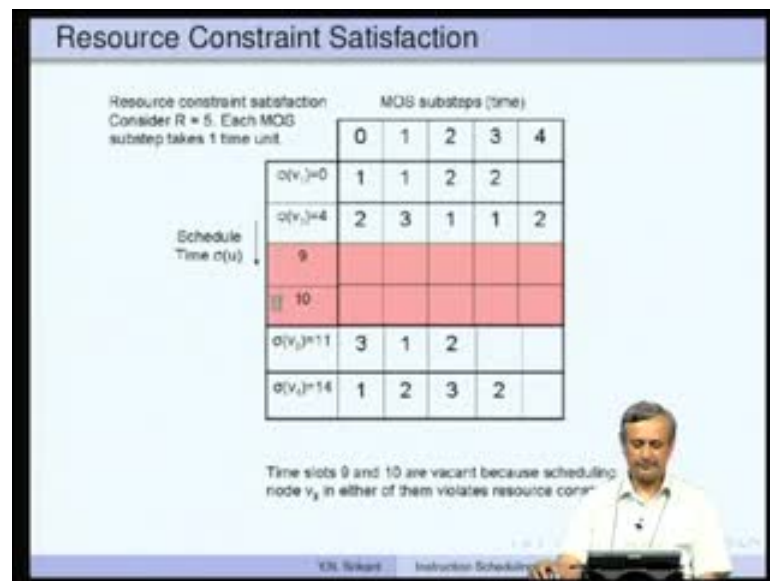
So, eventually after a certain number of steps, every instruction preceding v would have completed execution. So, resources must become available. So, this will never be an

infinite loop. It is just shown that we do not know how many instructions, how many times slots we need to check. That is why, this says L_b to infinity.

So, from L_b onwards, we check L_b , L_b plus 1, L_b plus 2, L_b plus 3, etcetera and what we check are exactly the same as what I showed you beforehand - the sum of the resource requirements of various times steps must be less than or equal to r , so this is the sum of the resource requirements of the other instructions which are concurrently executing with v - $\rho_{v,j}$ is the resource requirements of the j th sub steps of node v - so together tell you about the total resource requirements of the entire sequence.

So, this must be true for all values of j from zero to l_v , that is for all sub sets of the node v . Once we find a time slot i at which the resource requirements are also satisfied, and that may be L_b , L_b plus 1, L_b plus 2, etcetera we return that value i as the slot at which v can be scheduled.

(Refer Slide Time: 38:32)



So, for example, take the previous schedule itself, so here, this is σv_1 , this is v_1 , this is v_2 , etcetera, this is v_3 and this is v_4 - why did we show these two in red? This is time slot zero, this is time slot four, and this is time slot nine, ten, eleven, fourteen etcetera. These two time slots have been left free, nothing has been scheduled simply, because scheduling this instruction v_3 in any one of these violates the resource constraint.

So, if we leave these two free, then you know the resource constraints will not be violated. It is just for the sake of example. You know it may be one slots, it may be two slots or three slots in practice, but here I have just shown, these two as vacant saying that it is possible the two slots would be left free and then you know resource constraints would be satisfied if we schedule v_3 at eleven and v_4 at fourteen. So, that is why, we have left these two slots free. So, the Lb would have been nine because of the precedence constraints. So, this is 0 then plus four, then plus five is nine.

But Lb and Lb plus 1 are not suitable for scheduling v_3 , because the resources are not available. Let us say, resources are available at eleven, that is eleven, twelve, thirteen - these three times step resources become available therefore, we can scheduled them at that time slot and you know then resource constraints will be satisfied. No violation occurs.

(Refer Slide Time: 40:32)

List Scheduling - Priority Ordering for Nodes

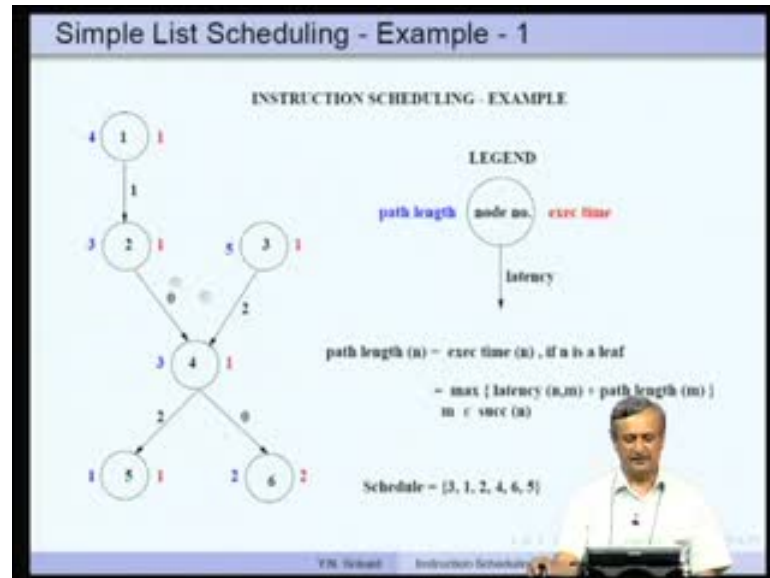
- 1 Height of the node in the DAG (i.e., longest path from the node to a terminal node)
- 2 *Estart*, and *Lstart*, the earliest and latest start times
 - Violating *Estart* and *Lstart* may result in pipeline stalls
 - $Estart(v) = \max_{i=1, \dots, k} (Estart(u_i) + d(u_i, v))$
where u_1, u_2, \dots, u_k are predecessors of v . *Estart* value of the source node is 0.
 - $Lstart(u) = \min_{i=1, \dots, k} (Lstart(v_i) - d(u, v_i))$
where v_1, v_2, \dots, v_k are successors of u . *Lstart* value of the sink node is set as its *Estart* value.
 - *Estart* and *Lstart* values can be computed using a top-down and a bottom-up pass, respectively, either statically (before scheduling begins), or dynamically during scheduling

Now, how do we actually order these nodes? In the priority ordering here, I said pick the highest priority node in the ready list, we did not say what is the priority ordering that we used, that is what we want to see now.

Let us consider two or three of these the varieties. One of them is the height of a node in the directed acyclic graph - that is longest path from the node to a terminal node - longer the path, higher the priority. Why? You know, if the path is longer that means there are

many instructions which depend on this particular node, so it is better, if that node is scheduled early that is the heuristic that I use.

(Refer Slide Time: 41:37)



So, let me show you an example. Here is a simple directed acyclic graph, the legend says inside is node number, left side is path length, and right side is execution time, below is latency. If you look at the nodes here, these are the leaf nodes. For the leaf nodes, we say that execution time of the node is the path length, so execution time is one here, so path length is also one, execution time is two here, so path length is two.

When we go this particular node, so we take the execution time here, path length here, add the delay, so that becomes 3, we take the path length here, add the delay, that is zero, so it becomes true. So, the maximum of these really becomes the path length for this particular node four - that is what this says. Latency n comma m plus path length of m , so, max over all the successors of the node n .

Then for this series, three plus zero, so that is three, and three plus two, that is five here path length and this becomes three plus one that is four, so this is how path lengths are computed. Let us say, when we start scheduling, let us say we consider these two nodes, then four and five, so this has a longer path length, so this node would be scheduled first and not this particular node.

Then, we can even look at this scheduling of this particular dag before we go to the other heuristic. So, the algorithm says, consider the root nodes of the dag so that is one and three, these are the only two, which will be put into the ready list. So, one and three are in the ready list, we need to pick the highest priority node from that ready list, that is path length, so five and four, so node three which has path length five is picked, because it is higher, so that is why the schedule indicates nodes three as the first node.

Let us assume that there are a number of resources to execute any number of instructions in parallel, so we do not have to worry about resource constraints in this simple example, the next example we look at that also.

Then three is completed, but then the successor instruction is four that cannot be executed until two is completed, and two cannot be completed unless one is completed so one is the only instruction in the ready queue right now. So, we are going to include one in the schedule it, in the next time step after three.

You know if the number of resources available is very large, no limitation. So, there is no problem about using resources, so one can be scheduled in the next time slot because one does not depend on three.

So the next one after one, the node number two can now be included in the ready list, because two has only one as its predecessors, two can be now included in the ready list. The minimum time slot at that which two can be scheduled, this was scheduled - let us say at time slot zero, a plus one, so that is the time slot at which two can be scheduled, so we can compute the time slots at which these the instructions can be scheduled in this manner.

Whereas, if we assume that only instruction can be issued in every cycle - that is you cannot have more than one instruction each cycle, then you know we have scheduled three at time slot zero, one at time slot at one. So, the minimum available times slot for this node number two is sigma of one plus one, so that is time slot number two so, after at the time slot two, we schedule this node two, because it has no predecessors and once two is complete, you know after time slot zero that four can be scheduled, the reason is node number three, which was initiated in time slot zero will complete in two time slot so by the time we reach time slot two, three would have completed.

So, four can be scheduled in time slot three, because both its operands are ready and once four is completed, we again have five and six put into the ready queue. Which of these should be picked? Again, we use the path length as the criterion two and one, so node six has path length two, which is higher than path length of one of node five, so scheduled node six, first, and followed by node five. Now, the ready list is empty. We have assigned at a time slot for each one of the nodes in the directed acyclic graph. So, that completes this particular scheduling process.

(Refer Slide Time: 47:13)

List Scheduling - Priority Ordering for Nodes

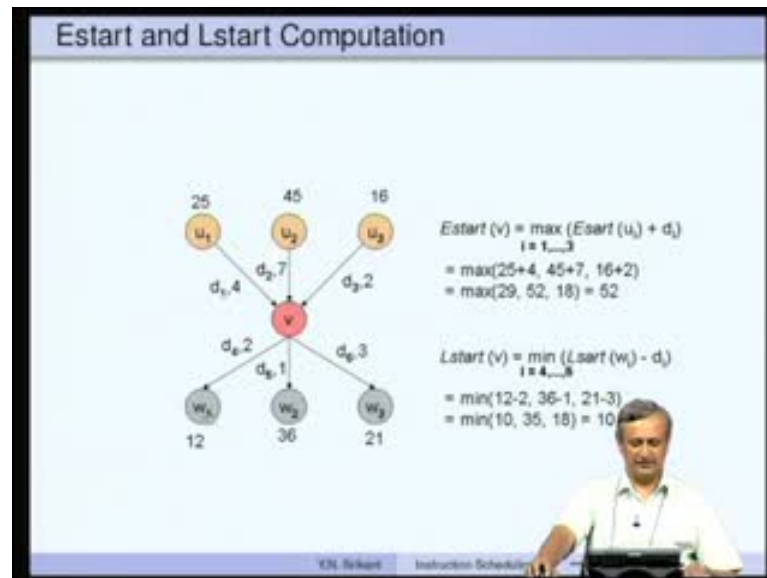
- Height of the node in the DAG (i.e., longest path from the node to a terminal node)
- *Estart*, and *Lstart*, the earliest and latest start times
 - Violating *Estart* and *Lstart* may result in pipeline stalls
 - $Estart(v) = \max_{i=1, \dots, k} (Estart(u_i) + d(u_i, v))$
where u_1, u_2, \dots, u_k are predecessors of v . *Estart* value of the source node is 0.
 - $Lstart(u) = \min_{i=1, \dots, k} (Lstart(v_i) - d(u, v_i))$
where v_1, v_2, \dots, v_k are successors of u . *Lstart* value of the sink node is set as its *Estart* value.
 - *Estart* and *Lstart* values can be computed using a top-down and a bottom-up pass, respectively, either statically (before scheduling begins), or dynamically during scheduling

Now, what is the second heuristics that can be used? We can compute what are known as earliest start time and latest start time, *Estart* and *Lstart* for each of the nodes. What exactly is *Estart*? It tells you that you cannot schedule a particular node earlier than the value of *Estart*. *Lstart* tells you that this is the latest time by which you should schedule the node, and if you violate either the *Estart* value or the *Lstart* value, and schedule it outside these limits, this may result in pipelines stalls.

So, as far as possible, we should try to stick to the time slots between *Estart* and *Lstart* and schedule the nodes. But *Estart* and *Lstart* are really based on precedence constraints; they do not consider any resource constraints. Therefore, we may still have to violate the *Estart* and *Lstart* value sometimes, because of resource constraints, but then since there are not many resources available, there will be pipeline stalls and that cannot be helped.

How do you compute Estart v? E start v is computed as the maximum overall the nodes 1 to k - you are really looking at the predecessors of the node v u i comma v - so u i are all the predecessors, there are k of them. So, you consider Estart value of u i and the delay of the edge u i comma v, take the max overhead that gives you Estart of v. Let us see how this is done.

(Refer Slide Time: 49:07)



So, you have the node v here, for which you need to want to compute Estart. These are the three predecessors so u1 u2 u3, so they have their Estart value already computed, so that is 25 45 16, let us say, the delays are d1 d2 d3, 4 7 and 2 respectively, so you look at 25 plus 4 then 45 plus 7, and 16 plus 2, so that means 29 52 18, the max is obviously 52.

So, this is how you compute the Estart value. This tells you that v cannot start before time slot fifty two, so that is that is make sense, because we are really looking at the time at which all the three predecessors of the node v complete that is forty five plus seven.

Now, the Estart value of the source node is taken as zero. We begin the computation of the Estart value from the top assuming the source node has a zero Estart value. What about Lstart? It is from the bottom, so Lstart u is computed as the minimum overall the successors. So, you want to compute the Lstart value of u. There are successors for u, those are all the v i nodes one to k, take the s start of the successors node subtract the latency and take the minimum of this particular computation overall the successors.

So, let us see what we need. There are 3 successors, we want to compute the Lstart value for v, w1, w2, w3, their Lstart values are 12 36 and 21. What is the Lstart value for node v? Twelve minus two, thirty six minus one and twenty one minus three, so minimum of all these is ten, so that is the latest time at which node v can be scheduled. So, if you schedule later than that, then you know we are going to create some pipeline stalls. That is very obvious, is it so? Let us say ten so ten plus two is twelve here and ten plus one is thirty six and ten plus three is thirteen.

So, if this can be scheduled at time slot two, that is why ten plus two is twelve, so if you actually exceed 1, then this node will not be scheduled at twelve, it will be scheduled at some other value, so there would be some kind of a stall, but they cannot be helped if there are many resources available.

(Refer Slide Time: 51:56)

List Scheduling - Priority Ordering for Nodes

- ④ Height of the node in the DAG (i.e., longest path from the node to a terminal node)
- ④ **Estart**, and **Lstart**, the earliest and latest start times
 - Violating *Estart* and *Lstart* may result in pipeline stalls
 - $Estart(v) = \max_{i=1, \dots, k} (Estart(u_i) + d(u_i, v))$
where u_1, u_2, \dots, u_k are predecessors of v . *Estart* value of the source node is 0.
 - $Lstart(u) = \min_{i=1, \dots, k} (Lstart(v_i) - d(u, v_i))$
where v_1, v_2, \dots, v_k are successors of u . *Lstart* value of the sink node is set as its *Estart* value.
 - *Estart* and *Lstart* values can be computed using top-down and a bottom-up pass, respectively, either statically (before scheduling begins), or dynamically during scheduling

Now, once we compute the Estart value starting from the top, the Lstart value of the sink node is set at this Estart value, and we start computing the Lstart value from the bottom towards the top, so Estart and Lstart can be computed using a top down and bottom up pass respectively either statically before scheduling begins or dynamically during the scheduling itself.

So, you know if you actually want to schedule the precedences rather the Estart and Lstart values during scheduling process, because the graph is going to change as and

when we schedule certain nodes, it can be done, but that requires a little extra computation at schedule time.

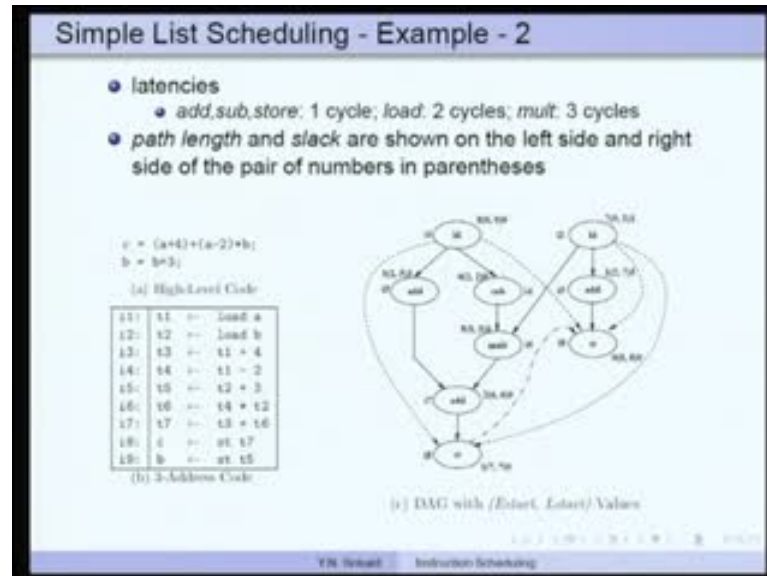
(Refer Slide Time: 52:49)

List Scheduling - Slack

- 1 A node with a lower *Estart* (or *Lstart*) value has a higher priority
- 2 $Slack = Lstart - Estart$
 - Nodes with lower slack are given higher priority
 - Instructions on the critical path may have a slack value of zero and hence get priority

So, a node with a lower *Estart* or *Lstart* value has a higher priority that is one possible heuristics, but a slightly better heuristics would be to consider slack which is *Lstart* minus *Estart*, you are really looking at the interval *Lstart* minus *Estart*, during which you want to schedule a node, so it make sense to consider nodes with a lower slack and give them higher priority - that is because you know *Estart* and *Lstart* are very close, there is not much gap between them. So, those nodes get higher priority and those which have higher slack value they have many more slots during in which they can be scheduled, so get a slightly lower priority. Instructions on the critical path may have a slack value of zero there *Lstart* and *Estart* values actually may coincide and hence they get the maximum priority.

(Refer Slide Time: 53:54)



So, this is an example which we saw already. Here is another example. Let me show you how the Estart values and Lstart values are really computed. For example, if you look at this particular graph, the add, sub and store values are supposed to have one cycle, two cycles and three cycles respectively, rather add sub store has one cycle, latency load has two cycle latency and mult has three cycle latency. The path length and slack are shown on the left side and right of the parentheses. So, eight is the path length and zero is the slack whereas, within the parentheses we are showing Estart and Lstart, so we start with a value of 0 for Estart and then you know, we compute zero plus two, as the Estart value for add, zero plus two for the Estart value of sub, and from here, two plus one, three, is one possibility for this particular add node, whereas, on this side two plus one is three, is the only possibility for the node mult and then we have you know for this particular node we have three delay slots for multiply, so three plus three, six is the Estart value for add, so this side we get three and this side we get six, so Estart really is the max of these. So, it is computed as 6 and when plus one is seven for this particular node s.

So, this is how Estart values are computed. We will stop at this point, continue this example in the next lecture, where we are going to discuss in more detail how the Lstart values are also computed and used for scheduling. Thank you.