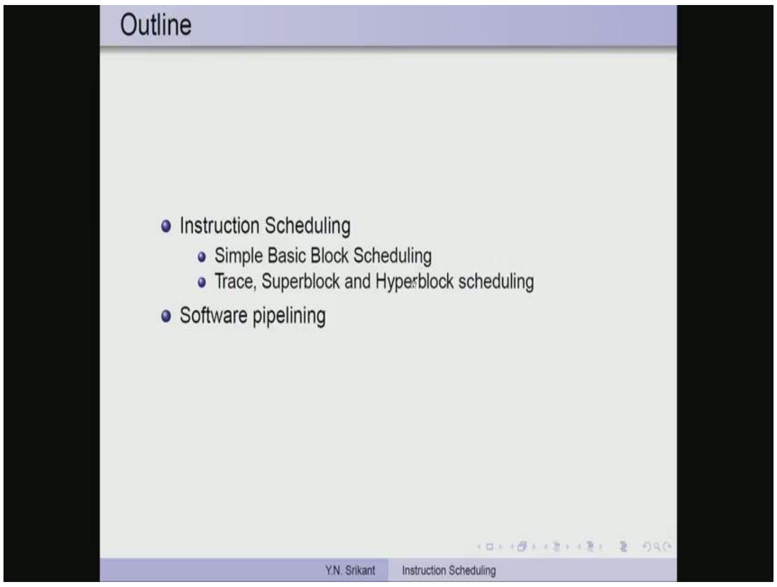
Principles of Compiler Design

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

Lecture - 38
Introduction Scheduling and Software Pipelining Part – 2

Welcome to part two of the lecture on instruction scheduling and software pipelining.

(Refer Slide Time: 00:25)



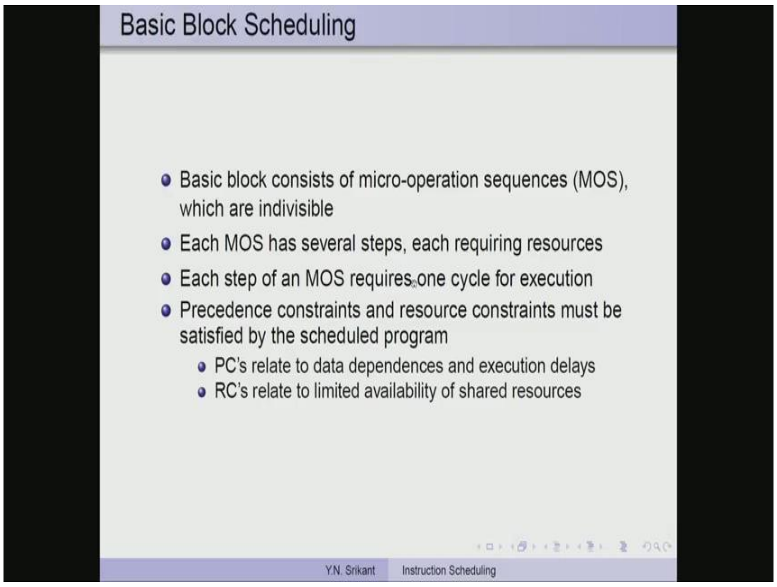
Outline

- Instruction Scheduling
  - Simple Basic Block Scheduling
  - Trace, Superblock and Hyperblock scheduling
- Software pipelining

Y.N. Srikant Instruction Scheduling

So, we discussed you know one part of simple basic block scheduling last time, we will continue with the today and then go on to trace, superblock and hyper blocks scheduling.

(Refer Slide Time: 00:41)



Basic Block Scheduling

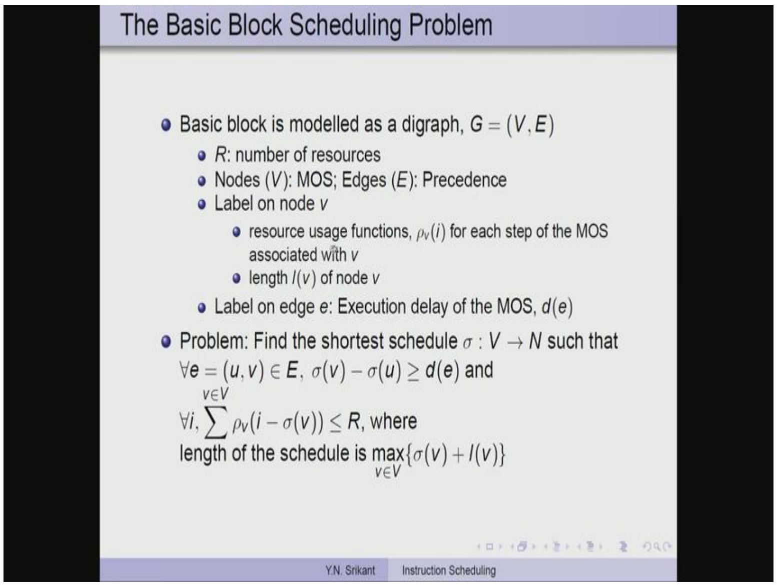
- Basic block consists of micro-operation sequences (MOS), which are indivisible
- Each MOS has several steps, each requiring resources
- Each step of an MOS requires one cycle for execution
- Precedence constraints and resource constraints must be satisfied by the scheduled program
  - PC's relate to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

Y.N. Srikant Instruction Scheduling

To do a bit of recap basic block scheduling consist of a you know, a basic block consists of micro operation sequences. These are the instructions of the machine and this micro operation sequences of are indivisible. That means, the micro operations which constitute the MOS cannot be scheduled separately. Each MOS of course, has a several stapes and each of these steps requires one cycle for execution.

So, it is possible that different instructions have a different number of micro operations within their MOS. And therefore, they may require different number of cycles and also different number of resources. So, we have two constraints for basic block scheduling; one is the precedence constraint, and other is the resource constraints. The precedence constraint relates to data dependences and execution delays whereas the resource constraint relates to the availability of shared resources.

(Refer Slide Time: 01:51)



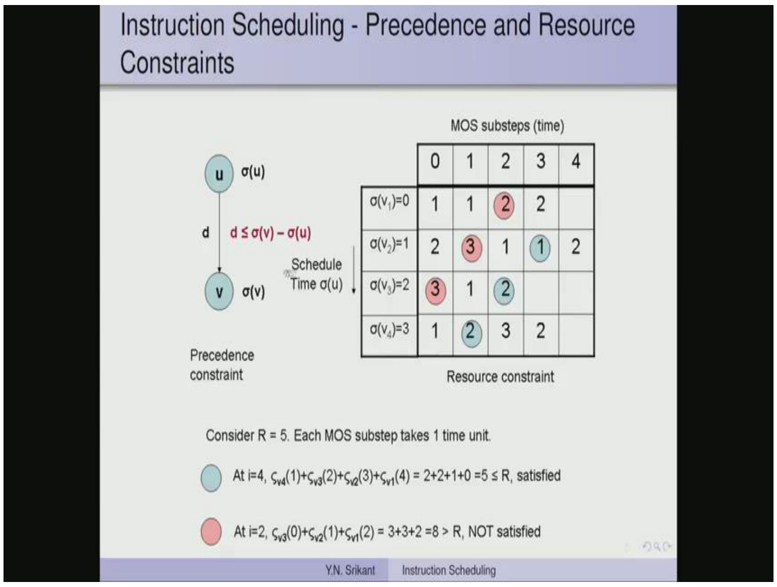
The Basic Block Scheduling Problem

- 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, here is the formal description of the basic block scheduling problem, it is actually basic block is model as a directed cyclic graph. And here the nodes or the MOS are the instructions, edges are the precedence constrains. And the label on each node tells us about the resource usage of that particular node or the MOS. And it does so for every micro operation of the MOS. And of course we have also length of the node which is nothing but the length of the resources vector. The problem is to find the shortest schedule sigma, which is a mapping from the nodes to the natural numbers this n nothing but the timeline. Such that, the precedence constrains are met and also the and also the resource constraints are met.

(Refer Slide Time: 02:59)



Instruction Scheduling - Precedence and Resource Constraints

	0	1	2	3	4
$\sigma(v_1)=0$	1	1	2	2	
$\sigma(v_2)=1$	2	3	1	1	2
$\sigma(v_3)=2$	3	1	2		
$\sigma(v_4)=3$	1	2	3	2	

Consider R = 5. Each MOS substep takes 1 time unit.

🔵 At i=4, $\varsigma_{v4}(1)+\varsigma_{v3}(2)+\varsigma_{v2}(3)+\varsigma_{v1}(4) = 2+2+1+0 = 5 \le R$, satisfied

🔴 At i=2, $\varsigma_{v3}(0)+\varsigma_{v2}(1)+\varsigma_{v1}(2) = 3+3+2 = 8 > R$, NOT satisfied

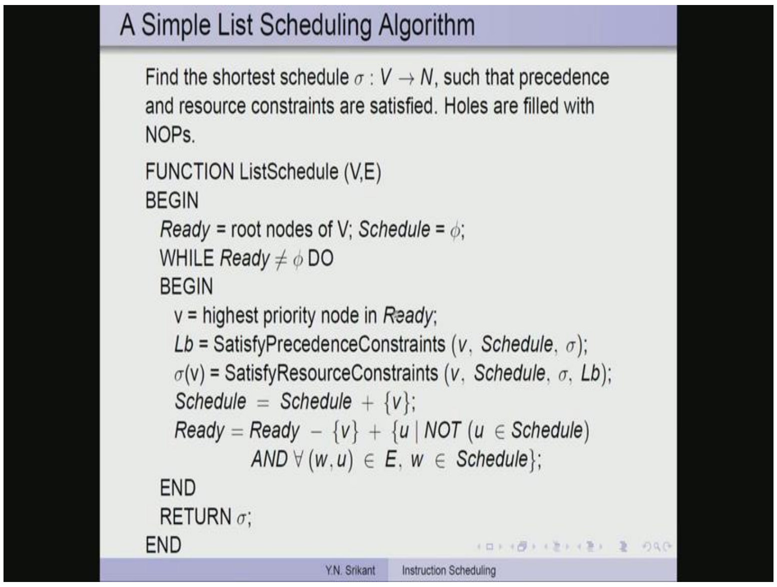
Y.N. Srikant Instruction Scheduling

The precedence constraint simply you know can be shown diagrammatically as follows. So, this is the node which has been scheduled already and this is the node which is to be scheduled and the delay on the edge u v is d. So, this constraint simply says that sigma v cannot be scheduled, d plus sigma u steps you know before d plus sigma u steps are completed. So, this is quite clear because this instruction takes d cycles to complete.

Similarly, for the resource scheduling this is the timeline in which the nodes are schedule on which this nodes are schedule, so v 1 is schedule at 0, v 2 at 1, v 3 at 2, v 4 at 3 etcetera. This is the step MOS sub step again in times of time because each MOS sub step requires only one cycle. So, if the resource constraint simply says add up the resources along the diagonal and that should be less than or equal to the number of resources available.

So, here we have shown only one resource and the number of resources is available is 5. And there is a clear violation here 3 plus 3 plus 2 being 8, that is because v 1 is still active in this sub step 2, v 1 v 2 is active and it is in sub step 1 v 3 is active and it is in sub step 0. So, the resource requirements of the sub step are 3 3 and 2 respectively so that adds up to 8. So, this schedule is not a feasible schedule and for this step you know 2 2 plus 2 plus 1 is fine so the resource constraints are satisfied, but since there is no satisfaction of constraints here, this schedule cannot be used.

(Refer Slide Time: 04:59)



A Simple List Scheduling Algorithm

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

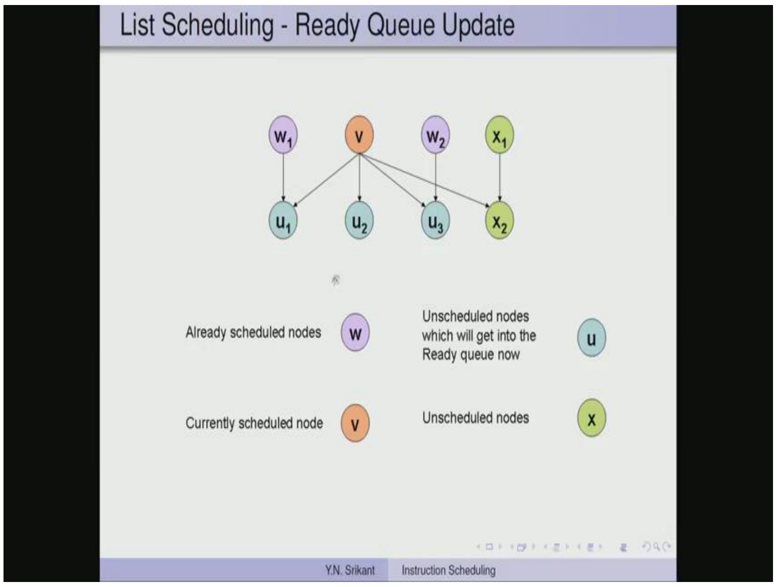
```
FUNCTION ListSchedule (V,E)
BEGIN
  $Ready$ = root nodes of V; $Schedule = \phi$;
  WHILE $Ready \neq \phi$ DO
  BEGIN
    v = highest priority node in $Ready$;
    $Lb$ = SatisfyPrecedenceConstraints ($v$, $Schedule$, $\sigma$);
    $\sigma$(v) = SatisfyResourceConstraints ($v$, $Schedule$, $\sigma$,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

Y.N. Srikant Instruction Scheduling

The algorithm for list scheduling is quit straight forward it is a topological sort based algorithm, what we do is a we pick up the root nodes of the directed acyclic graph as the starting point. There are put it to queue called ready queue and we keep doing it until the ready queue is not empty. So, we pick the highest priority node in the queue and then we find the we find the lowest times slot in which the precedence constraints are satisfied.

And then from that l b onwards, we find the slot in which the resource constraints are satisfied, as i explain before. So, that would be the schedule for the node v now, v goes in to the set of schedule nodes and the ready list is updated, by removing v and adding those successors of the schedule nodes. For example, u not of u in schedule, so that means you should have already you know u is not a schedule node already. And then we also pick up the successors of the schedule node w is scheduled, so this are all ready to be placed into the ready queue.

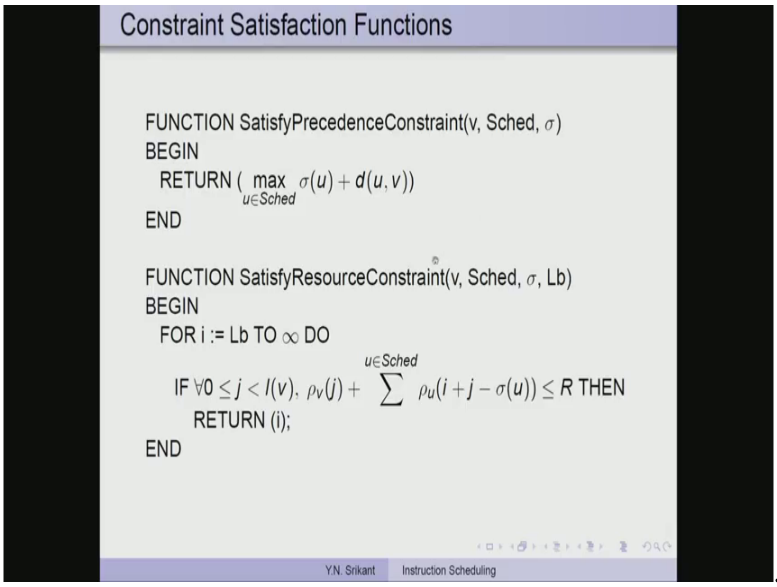
(Refer Slide Time: 06:17)



List Scheduling - Ready Queue Update

So, the ready queue updated is very simple you know as follows, so if this are the schedule nodes and we already know that you know v has been schedule. So, u 1 u 2 u 3 are ready to be put it into the ready queue, that is because the predecessor of u 1 u 2 u 3 have already been schedule, whereas the predecessor of x 2 is not yet schedule.

(Refer Slide Time: 06:44)



Constraint Satisfaction Functions

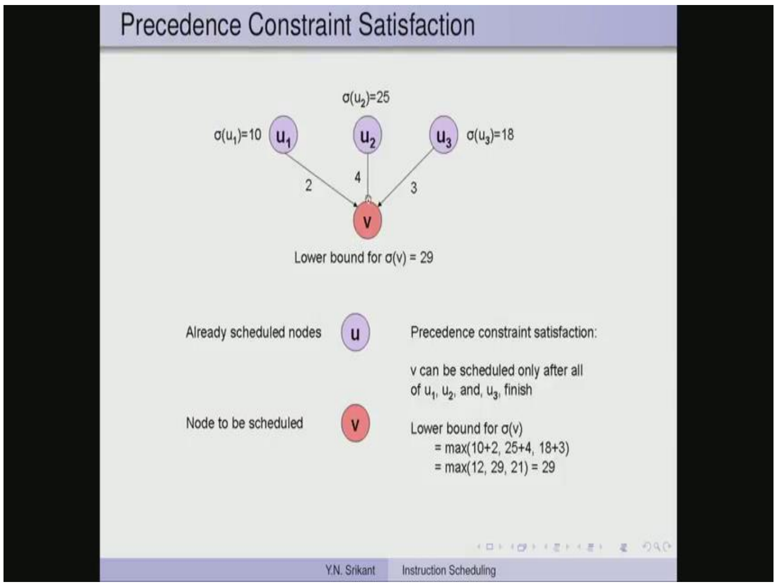
```
FUNCTION SatisfyPrecedenceConstraint(v, Sched, $\sigma$)
BEGIN
  RETURN ( $\max_{u \in Sched} \sigma(u) + d(u, v)$)
END

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

.

So, this is these are the two functions which I explained you know corresponding to the satisfaction of constraints, precedence and resource.

(Refer Slide Time: 06: 56)



Precedence Constraint Satisfaction

$\sigma(u_2)=25$

$\sigma(u_1)=10$ u_1 u_2 u_3 $\sigma(u_3)=18$

2 4 3

v

Lower bound for $\sigma(v) = 29$

Already scheduled nodes u

Node to be scheduled v

Precedence constraint satisfaction:

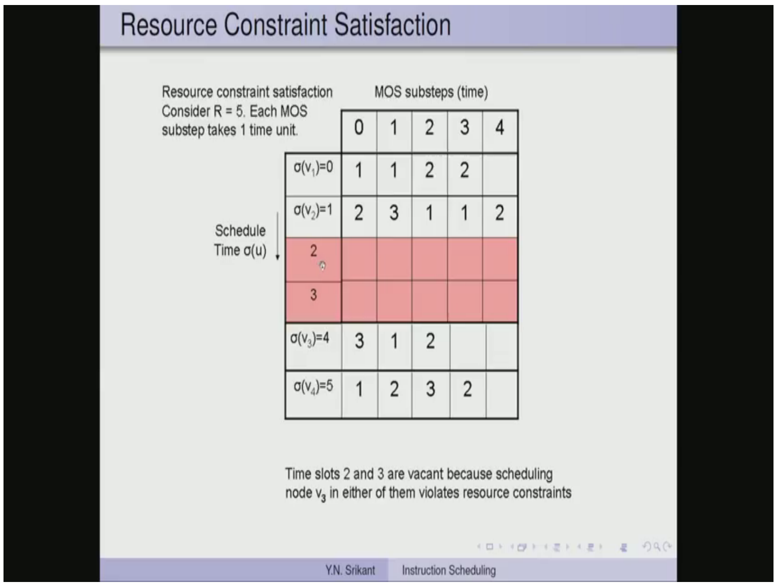
v can be scheduled only after all of u_1, u_2, and, u_3, finish

Lower bound for $\sigma(v)$
= max(10+2, 25+4, 18+3)
= max(12, 29, 21) = 29

Y.N. Srikant Instruction Scheduling

So, precedence constraints satisfaction simply says, we find a slot for v which is maximum of a sigma u and plus 2 sigma u 2 plus 4 or sigma u 3 plus 3. In this case, it happens be 29. So, this is the earliest time at which v can be schedule and that is Lb as returned by the function satisfy precedence constraints.

(Refer Slide Time: 07:21)



Resource Constraint Satisfaction

Resource constraint satisfaction
Consider R = 5. Each MOS substep takes 1 time unit.

MOS substeps (time)

Schedule Time $\sigma(u)$

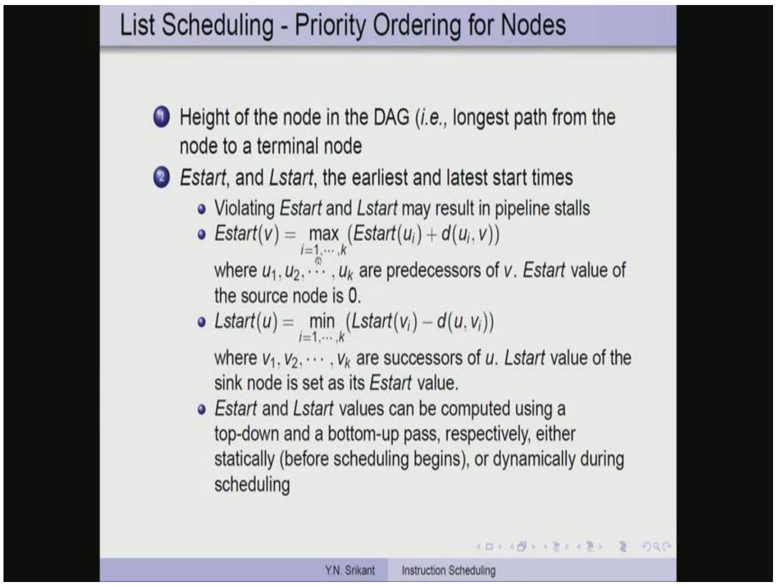
	0	1	2	3	4
$\sigma(v_1)=0$	1	1	2	2	
$\sigma(v_2)=1$	2	3	1	1	2
2					
3					
$\sigma(v_3)=4$	3	1	2		
$\sigma(v_4)=5$	1	2	3	2	

Time slots 2 and 3 are vacant because scheduling node v_3 in either of them violates resource constraints

Y.N. Srikant Instruction Scheduling

As far as the resource constrain satisfaction is concerned, we check at every times slot. Whether the requirements of resources of various kinds and for the various sub steps is indeed under control. So for example, if we schedule this node here, as we have already seen we exceed the number of resources available. So, this slot is left free the same is true for this slot as well so for example if we place this here we get 3 then 1 and 2 so that is 6 which exceeds 5. So, 3 is also kept vacant and then we can schedule 4 and 5 here, so these are the no up slots.

(Refer Slide Time: 08:06)



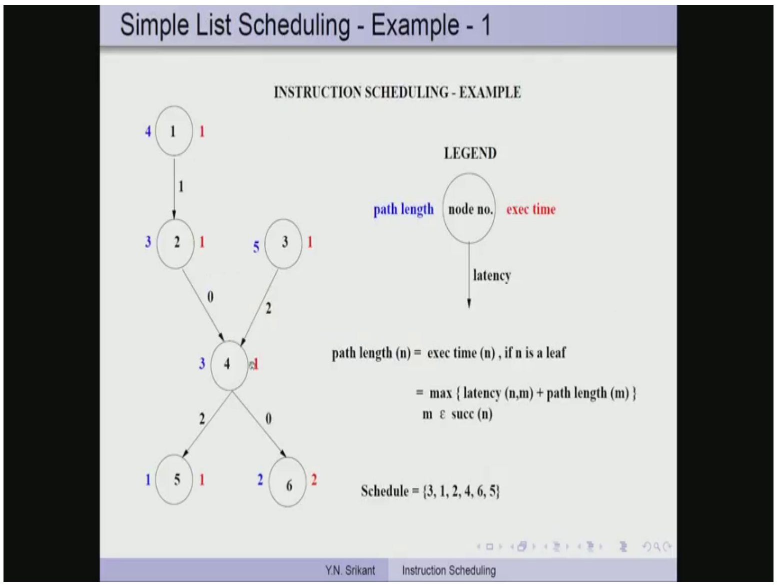
List Scheduling - Priority Ordering for Nodes

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
   - $Estart(v) = \max_{i=1,\cdots,k} (Estart(u_i) + d(u_i, v))$ where $u_1, u_2, \cdots, u_k$ are predecessors of v. *Estart* value of the source node is 0.
   - $Lstart(u) = \min_{i=1,\cdots,k} (Lstart(v_i) - d(u, v_i))$ where $v_1, v_2, \cdots, v_k$ 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

Y.N. Srikant Instruction Scheduling

Now, the last issue that we need to look at before we look at examples, is how do we order the ready queue by priority, what is our priority ordering function. So, one possible priority ordering function is the height of the node in the directed acyclic graph. That is, the longest path from the node to a terminal node, I will give you an example of this to explain it.

(Refer Slide Time: 08:41)



Simple List Scheduling - Example - 1

INSTRUCTION SCHEDULING - EXAMPLE

LEGEND

path length (n) = exec time (n) , if n is a leaf

$$= \max_{m \,\varepsilon\, succ(n)} \{ latency\ (n,m) + path\ length\ (m) \}$$

Schedule = {3, 1, 2, 4, 6, 5}

Y.N. Srikant Instruction Scheduling

So, suppose we consider this graph right, so this is a directed acyclic graph. So, the legend says the left side the blue is the path length that we need to compute. Whatever, is inside the circle is the node number and then what is return to the right of the circle in red, is the execution time, and the label on the edge latency of the instruction. So, this a very generalized you know model, it allows execution time to be attach to the node and latency to be attach to the edge.

The reason why this may become important you is one example of that is the load instruction. So for example, immediate if there are many load units available, then you know after one of the loads is started and which requires one cycle of execution time. We can actually load you know schedule other loads on a other load units, but for this particular load which was started it may require several units of time to make the result available.

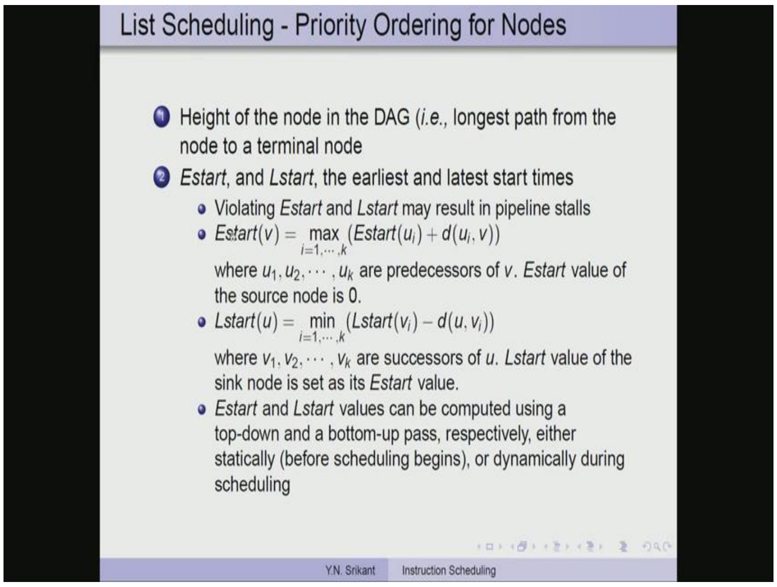
So, the load execution time is 1, whereas the load latency is 2 that could be an example a instruction, which has an execution time and a latency as well. The path length is commuted as the execution time if n is a leaf. So, in this case these two are the leaves, so we have the execution time as 2 and 1, so the path length is initialized to 2 and 1 respectively here. Otherwise we take the maximum of the latency of the edge and added to it the path length of the target node.

So for example, if we take we have computed the path length for these two as 1 and 2. Now, we consider the node and these are the two successors which have already been scheduled. So, we take the execution time and of this node alright, so this is the node for which we want to compute the path length. So, what we do is we take the latency of this instruction, with this instruction which is given as a 2 and we consider the path length of this particular node.

So, that is 1 so 2 plus 1 is the path length as computed along this path. Along this path we have a latency of 0 and a path length of 2, so this is 2 plus 0 equal to 2. So, the maximum of these two is 3, so that is the path length of 4. So, you can see that quite easily so we have to use 2 plus 1 here.

And then we move on to this node and for that node the execution time you know the delay or the latency is 0 and the path length here is 3, so 3 plus 0 is 3. So, that is denoted as the path length here, for this node latency is 2 and the path length is 3. So, we note 5 as the path length here, and for this node this is 3 plus 1 that is 4. So, this is how we compute the path lengths we will see another example little later.

(Refer Slide Time: 12:31)



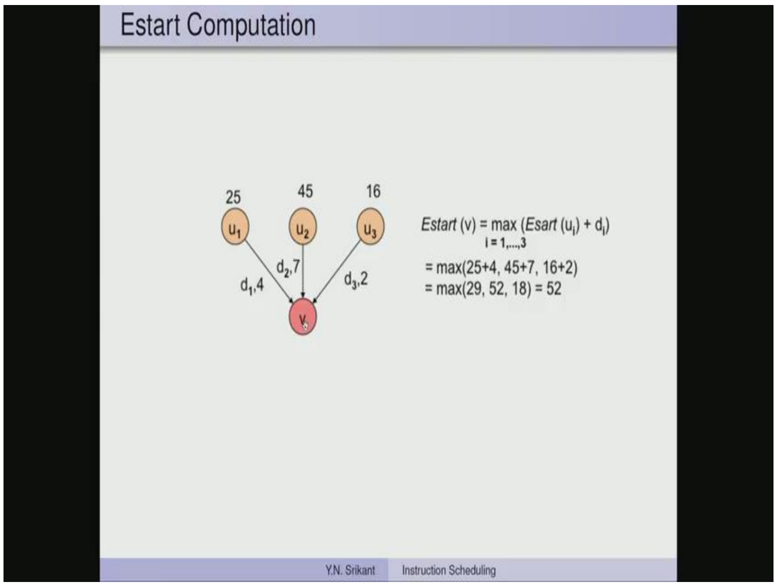
List Scheduling - Priority Ordering for Nodes

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
   - $Estart(v) = \max_{i=1,\cdots,k}(Estart(u_i) + d(u_i, v))$ where $u_1, u_2, \cdots, u_k$ are predecessors of v. *Estart* value of the source node is 0.
   - $Lstart(u) = \min_{i=1,\cdots,k}(Lstart(v_i) - d(u, v_i))$ where $v_1, v_2, \cdots, v_k$ 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

Y.N. Srikant Instruction Scheduling

The second possibility is to use early start and late start, latest start times as the priority ordering values. So, E start is the earliest time at which a node can be scheduled and L start is the latest time, at which the load can be scheduled. So, violating E start and L start may result in pipeline stalls, so we may have to introduce no ops in that case. So, every node can be scheduled between E start and L start, so that is the idea. So, how do we compute E start, E start of a node is the maximum of E start u i plus d of u i comma v. So, this is the delay and this is the E start of the node, so where u 1 to u k are the predecessors of v. So, let me and E start of the source node is 0, so let me show you an example of this.

(Refer Slide Time: 13:43)



Estart Computation

$Estart(v) = \max_{i=1,\ldots,3} (Esart(u_i) + d_i)$

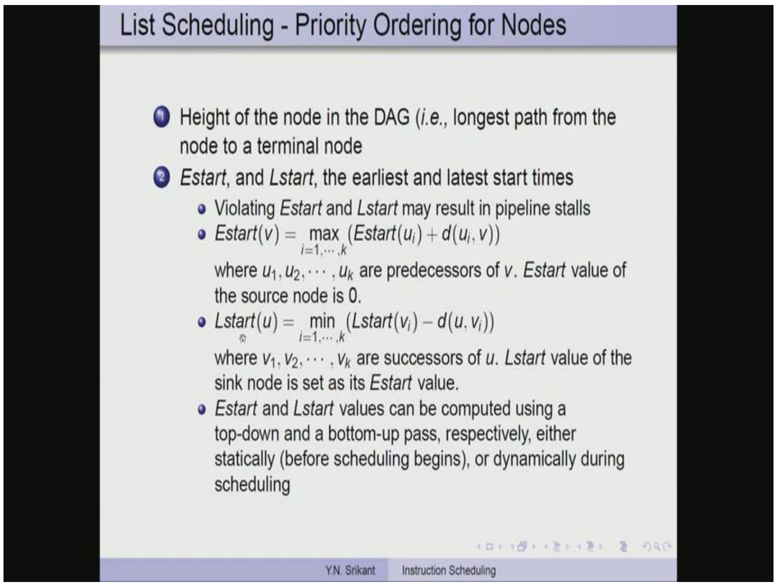
$= \max(25+4, 45+7, 16+2)$

$= \max(29, 52, 18) = 52$

Y.N. Srikant Instruction Scheduling

We want to compute the E start of v that is, the earliest time at which this can be started. So, to do that we know the E start of the predecessors, so we add delay to it and find out the maximum, so 25 plus 4, 45 plus 7 and 16 plus 2, so the maximum comes to 52.

(Refer Slide Time: 14:07)



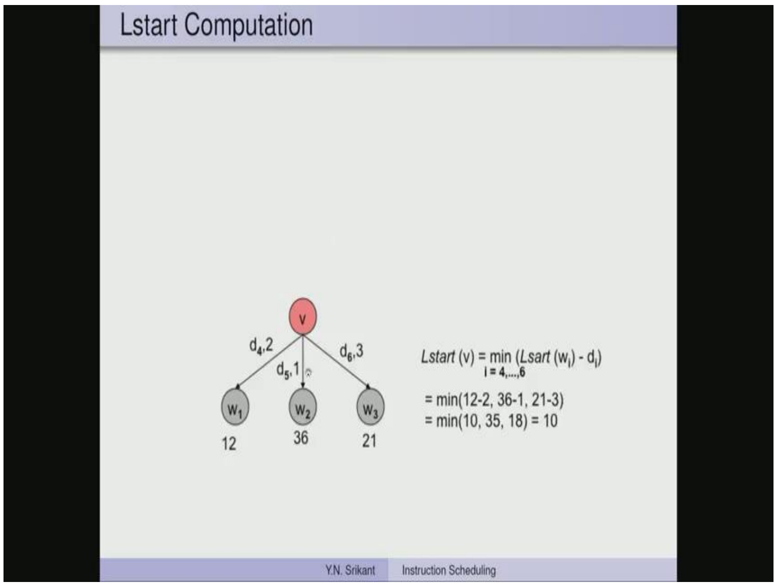
List Scheduling - Priority Ordering for Nodes

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
   - $Estart(v) = \max_{i=1,\cdots,k}(Estart(u_i) + d(u_i, v))$ where $u_1, u_2, \cdots, u_k$ are predecessors of v. *Estart* value of the source node is 0.
   - $Lstart(u) = \min_{i=1,\cdots,k}(Lstart(v_i) - d(u, v_i))$ where $v_1, v_2, \cdots, v_k$ 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

Y.N. Srikant Instruction Scheduling

So, in a similar way the latest start time can be computed as the minimum of L start v i minus of d u i. Where v 1 to v k are the successors, so the L start of the sink node is set as the E start of the node itself.

(Refer Slide Time: 14:27)



Lstart Computation

$Lstart(v) = \min_{i=4,\ldots,6}(Lsart(w_i) - d_i)$

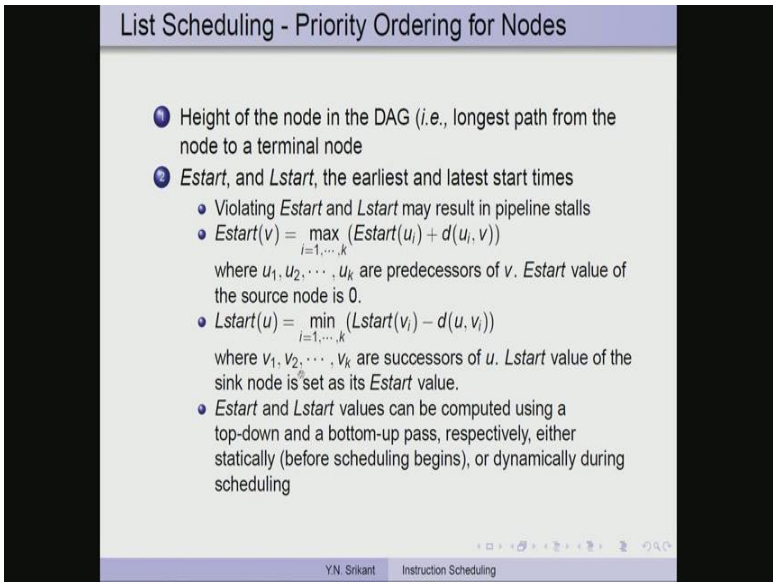
$= \min(12\text{-}2, 36\text{-}1, 21\text{-}3)$

$= \min(10, 35, 18) = 10$

Y.N. Srikant Instruction Scheduling

So, again taking this example, we want to compute the L start of v for that we know the L start values of the successors, so w 1 w 2 and w 3. So, we compute 12 minus 2, 36 minus 1 and 21 minus 3, so that comes to 10 as it is returned here. So, basically we are working backwards to compute the L start starting from the sink whereas, for the E start we start from the top and go towards the sink.

(Refer Slide Time: 15:04).



List Scheduling - Priority Ordering for Nodes

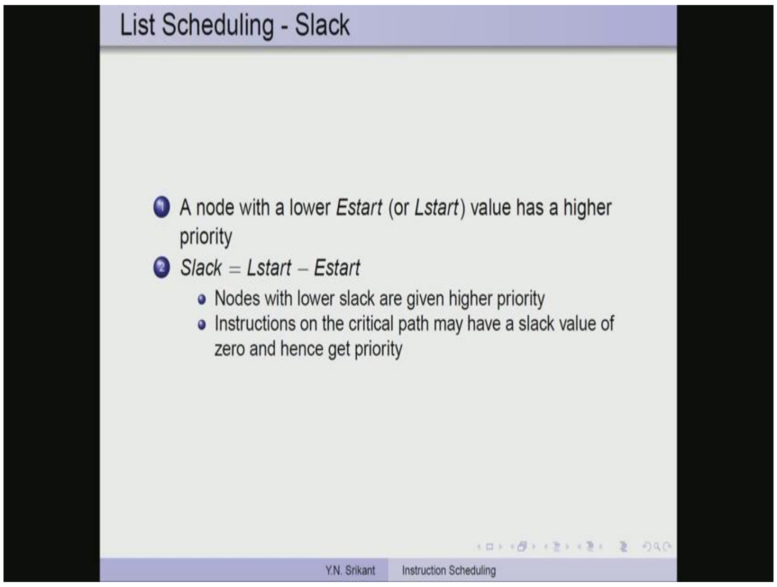
1. Height of the node in the DAG (*i.e.*, longest path from the node to a terminal node
2. *Estart*, and *Lstart*, the earliest and latest start times
   - Violating *Estart* and *Lstart* may result in pipeline stalls
   - $Estart(v) = \max_{i=1,\cdots,k} (Estart(u_i) + d(u_i, v))$ where $u_1, u_2, \cdots, u_k$ are predecessors of v. *Estart* value of the source node is 0.
   - $Lstart(u) = \min_{i=1,\cdots,k} (Lstart(v_i) - d(u, v_i))$ where $v_1, v_2, \cdots, v_k$ 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

Y.N. Srikant Instruction Scheduling

So, E start and L start values can be comfortably computed using a top down pass and a bottom up pass. So, basically we start the E start computation from that top go to the sink node and then initialize the L start value of the sink node, to its E start value and work backwards to compute the L start value. So of course, this is during the this can be done either during the you know before the scheduling begins or it is possible to do it dynamically during the scheduling itself.

So, the different between these two is you know quite important if we do it before the scheduling begins, then we cannot alter the priority of the node during the scheduling process. Whereas, if we do it during the scheduling process itself, then the priority of the node can possibly be altered.

(Refer Slide Time: 16:05)



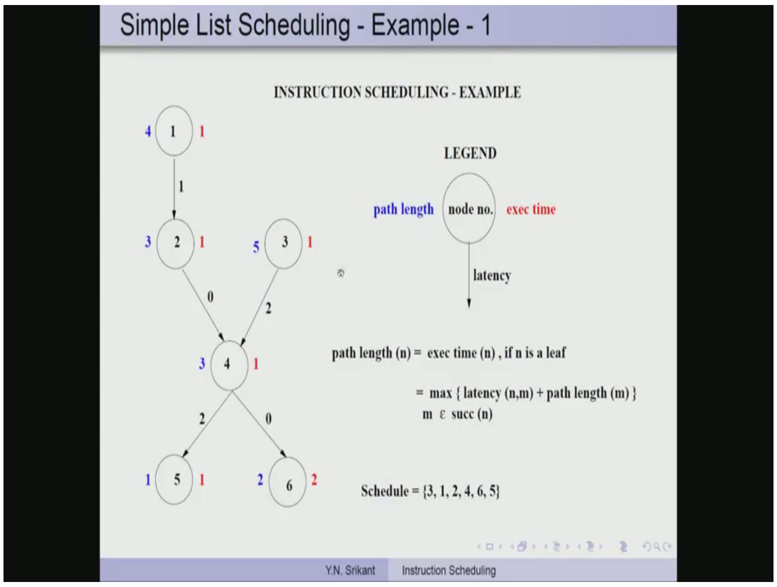
List Scheduling - Slack

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

Y.N. Srikant Instruction Scheduling

So, it is also possible to use a slack value as another priority item, so before we look at this slack we could also use you know E start or L start value has the priority item. So, if the lower the E start value, the higher the priority and the lower the L start value again it has a higher priority. In the case, of slack which is nothing but L start minus E start, the nodes with lower slack are given the higher priority. And instructions on the critical path may have a slack value of 0, and therefore automatically they will get the priority.

(Refer Slide Time: 16:49)



Simple List Scheduling - Example - 1

path length (n) = exec time (n) , if n is a leaf

= max { latency (n,m) + path length (m) }
m ε succ (n)

Schedule = {3, 1, 2, 4, 6, 5}

Y.N. Srikant Instruction Scheduling

So, let us look at an example, of scheduling with the path length first and then using slack as second example. So, we already computed the path length for this example a few minutes ago now, let us try to schedule using the path length. So, to begin with the node number you know 1 and 3, these are the roots of the dag, so they will be put it in to the ready queue.

Let us assume, that the resource constraints are always satisfied in other words there are enough numbers of resources in the system. So, we do not even have to check the resource schedule, resource constraint at this point we will see that in the next example in this case, there are no resource constraints. So, between 1 and 3 we need to pick one of the nodes to be schedule in the first time slot.

So, to do that again we look at the path length this has 4 and this has 5 so the higher path length instruction is picked up first so that gets scheduled at 3. So, the sorry in the first time slot so number 3 gets scheduled in the first time slot. Therefore, the now ready queue has this 1 right, ready queue cannot contain 4 at this time even though it is the successor of 3 because 4 has a predecessor 2 which is not yet scheduled.

So, we can only put those nodes whose predecessors are already scheduled into the ready list. So, after we schedule 3 we are force to schedule 1, there is no other option and after we schedule 1 we can schedule 2. The reason is so 1 we start this node 3, 1 is independent of it, so it can be schedule in the second time slot and node number 1 executes with a just one delay. So, in the next third time slot we can schedule node number 2.

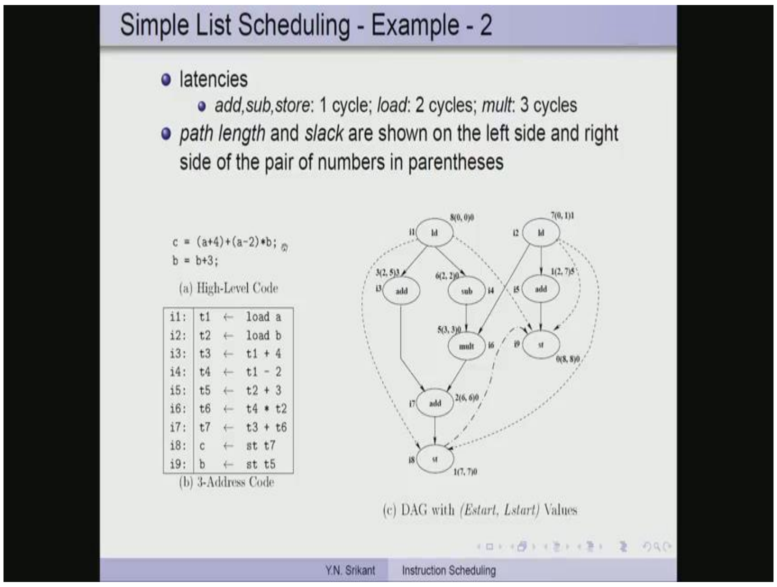
So, once we do that of course, this seems to require 0 number of slots, and node number 3 requires 2 slots to complete. So, since this was scheduled in slot number 1 you know in this is number 2, this is number 3 slot number 4 is available for instruction number 4, both 3 and 2 would have completed by that time. So, we can schedule you know 4 in slot number 4 as we have done here.

Now, there are two possibility after slot number 4 for the slot number 5 we can schedule either 5 or 6, but it so happens that 5 you know requires a 2 cycles. And it has a path length of 1 this requires a 0 cycles and it has a path length of 2. So, since that path length indicates that this can should be scheduled first, we can put that into the slot number 5 and by the in the slot number 6 you know 4 would have completed and 5 can be schedule.

So, the difference between 4 and 5 is actually 2 time slots so that is sufficient for this instruction to complete. So, we could not have placed 5 into slot number 5 because the delay involved in completing 4 is 2 cycles. So, if we had try to place 5 we would have placed a no op in cycle number 5 and then we would have placed 5. That would have been inefficient schedule, but our heuristic of using the path length takes care of it and says we can schedule node number 6 in slot number 5.

So, this is a path does schedule no ops in between so that is assuming there are no resource constraints. So, this is how we produce the schedule you know and this is how we use the path length in order to brake any conflicts. So, there was a conflict here and there was a conflict here as well.

(Refer Slide Time: 21:14)


8(0, 0)0
i1
ld
7(0, 1)1
i2
ld
3(2, 5)3
i3
add
6(2, 2)0
sub
i4
i5
add
1(2, 7)5
5(3, 3)0
mult
i6
i9
st
0(8, 8)0
i7
add
2(6, 6)0
i8
st
1(7, 7)0

Simple List Scheduling - Example - 2

- latencies
  - *add,sub,store*: 1 cycle; *load*: 2 cycles; *mult*: 3 cycles
- *path length* and *slack* are shown on the left side and right side of the pair of numbers in parentheses

```
c = (a+4)+(a-2)*b;
b = b+3;
```

(a) High-Level Code

```
i1: t1 ← load a
i2: t2 ← load b
i3: t3 ← t1 + 4
i4: t4 ← t1 - 2
i5: t5 ← t2 + 3
i6: t6 ← t4 * t2
i7: t7 ← t3 + t6
i8: c  ← st t7
i9: b  ← st t5
```

(b) 3-Address Code

(c) DAG with *(Estart, Lstart)* Values

Y.N. Srikant Instruction Scheduling

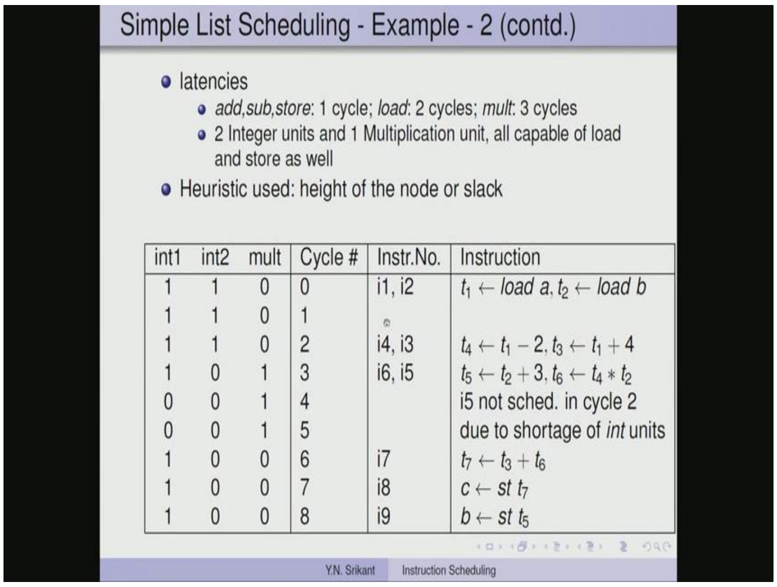
Now, we go on to the second example this is the example, we had studied before. Here, the latencies of the add sub and store instructions are 1 cycle each, the load instruction has 2 cycles of latencies and the multiply instruction has 3 cycles of latencies. So, we actually this is the dag with all this extra you know anti and output dependences already marked.

So, if you look at the instruction sequence i 1 to i 9, we are not sure whether we can actually schedule everything without any no op, so this dag does not tell you anything. Now, the early start E start and L start values can be computed in a top down pass and in a bottom up pass quite easily. So, we start with the E start value of this as 0, then you know using that simple computation we assign E start values to these two. And then this cannot be assigned E start value this 2 cannot assign E start value immediately.

So, we will have to assign E start of 0 to this and then once that is assign we can assign the E start value for this as well. So, the E start value is indicated as the first component of this parenthesis and once we complete the computation of the E start values up to this point and then this point. We assign the L start value of this node as the E start value and then we backwards in order to produce the E start values. So, this is a fairly straight forward pass so I will not spend too much time in explaining how it is computed.

Then the number to the left of the parenthesis is the path length. So, you know that is easy to compute again you know we start with a path length here as 0 and work backwards. The number to the back words the number to the right of the parenthesis is the slack which is nothing but the L start minus E start value. So, to schedule this so let me show you how this scheduling can happen.

(Refer Slide Time: 24:00)



Simple List Scheduling - Example - 2 (contd.)

- latencies
  - *add,sub,store*: 1 cycle; *load*: 2 cycles; *mult*: 3 cycles
  - 2 Integer units and 1 Multiplication unit, all capable of load and store as well
- Heuristic used: height of the node or slack

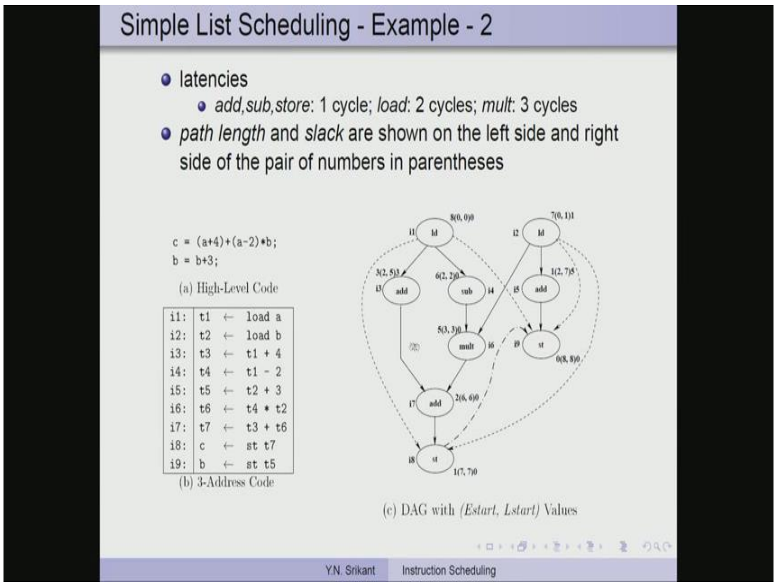
int1	int2	mult	Cycle #	Instr.No.	Instruction
1	1	0	0	i1, i2	$t_1 \leftarrow load\ a, t_2 \leftarrow load\ b$
1	1	0	1		
1	1	0	2	i4, i3	$t_4 \leftarrow t_1 - 2, t_3 \leftarrow t_1 + 4$
1	0	1	3	i6, i5	$t_5 \leftarrow t_2 + 3, t_6 \leftarrow t_4 * t_2$
0	0	1	4		i5 not sched. in cycle 2
0	0	1	5		due to shortage of *int* units
1	0	0	6	i7	$t_7 \leftarrow t_3 + t_6$
1	0	0	7	i8	$c \leftarrow st\ t_7$
1	0	0	8	i9	$b \leftarrow st\ t_5$

Y.N. Srikant Instruction Scheduling

So, let us assume that we have two integer units and one multiplication unit and all this 3 units are capable of handling the load instruction and store instruction. The heuristic used you know whether we use the height of the node or the slack value the schedule comes out to be the same.

So, let us begin to begin with a in this diagram we have both this you know i 1 here and i 2 here both this are eligible to be scheduled. So, among these one of them can be scheduled on integer unit 1 and other can be scheduled on integer unit 2, there is no need to wait. So, both this loads are scheduled in the same cycle number 0 and in cycle number 1, we cannot scheduled anything. The reason is t 1 is needed in i 4 and you know it is also needed in i 3.

(Refer Slide Time: 25:19)



Simple List Scheduling - Example - 2

- latencies
  - *add,sub,store*: 1 cycle; *load*: 2 cycles; *mult*: 3 cycles
- *path length* and *slack* are shown on the left side and right side of the pair of numbers in parentheses

```
c = (a+4)+(a-2)*b;
b = b+3;
```

(a) High-Level Code

i1:	t1 ← load a
i2:	t2 ← load b
i3:	t3 ← t1 + 4
i4:	t4 ← t1 - 2
i5:	t5 ← t2 + 3
i6:	t6 ← t4 * t2
i7:	t7 ← t3 + t6
i8:	c ← st t7
i9:	b ← st t5

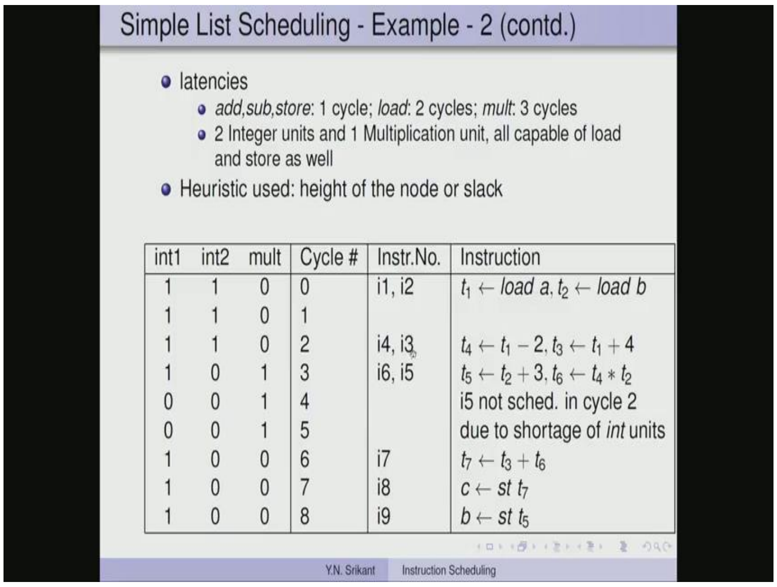
(b) 3-Address Code

(c) DAG with *(Estart, Lstart)* Values

Y.N. Srikant   Instruction Scheduling

So, both this instruction require this value of t 1, so t 1 plus 4 and t 1 minus 2. So, the next cycle after this has to be left vacant, we have to introduce a no op here there is no other option. And by the time we arrive at cycle number 2 the loads have completed. So, we have you know this load and this load has been completed so we have this, this and this all the 3 as possibilities for scheduling they get it into the ready queue.

(Refer Slide Time: 25:55)



Simple List Scheduling - Example - 2 (contd.)

- latencies
  - *add,sub,store*: 1 cycle; *load*: 2 cycles; *mult*: 3 cycles
  - 2 Integer units and 1 Multiplication unit, all capable of load and store as well
- Heuristic used: height of the node or slack

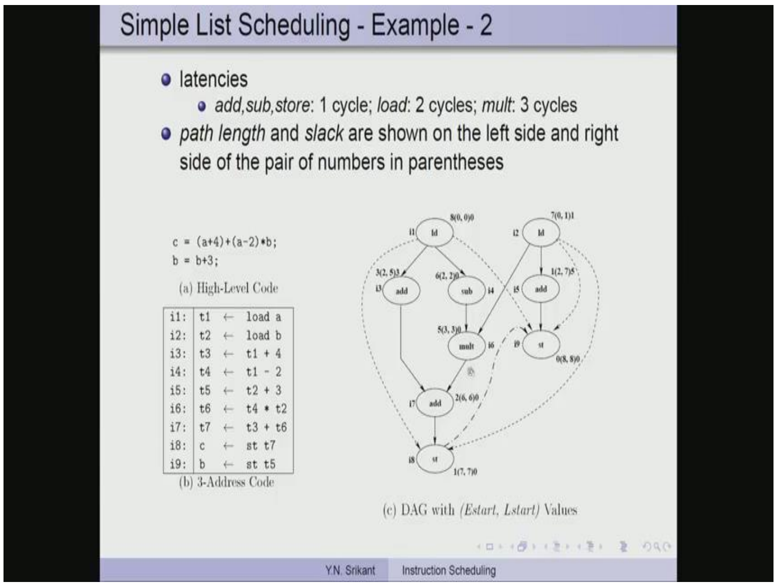
int1	int2	mult	Cycle #	Instr.No.	Instruction
1	1	0	0	i1, i2	$t_1 \leftarrow load\ a, t_2 \leftarrow load\ b$
1	1	0	1		
1	1	0	2	i4, i3	$t_4 \leftarrow t_1 - 2, t_3 \leftarrow t_1 + 4$
1	0	1	3	i6, i5	$t_5 \leftarrow t_2 + 3, t_6 \leftarrow t_4 * t_2$
0	0	1	4		i5 not sched. in cycle 2
0	0	1	5		due to shortage of *int* units
1	0	0	6	i7	$t_7 \leftarrow t_3 + t_6$
1	0	0	7	i8	$c \leftarrow st\ t_7$
1	0	0	8	i9	$b \leftarrow st\ t_5$

Y.N. Srikant Instruction Scheduling

Now, we choose to schedule i 3 and i 4 because they have higher priority so that would be they would be scheduled again on integer units int 1 and int 2, we are still not use multiply. So, these are the two instructions so they have been scheduled in cycle number 2 and there is no need to make them sequential here. So, cycle number 3 now, we have a you know i 5 and i 6 available to us right, i 5 was available in cycle number 2 also, but you know there was no availability of resources here.

So, the 2 integer units are already taken and we required another integer unit to perform this t 5 equal to t 2 plus 3 this had to be perform. So, we did not have another integer unit here therefore, we priority choose i 3 and i 4 the whether the priority it is a same here the choice does not change really. So, once we have chosen this i 5 can be scheduled in the next slot, i 6 also available you know was ready to be scheduled in this slot so we can see that here.

(Refer Slide Time: 27:25)



Simple List Scheduling - Example - 2

- latencies
  - *add,sub,store*: 1 cycle; *load*: 2 cycles; *mult*: 3 cycles
- *path length* and *slack* are shown on the left side and right side of the pair of numbers in parentheses

```
c = (a+4)+(a-2)*b;
b = b+3;
```

(a) High-Level Code

i1:	t1	←	load a
i2:	t2	←	load b
i3:	t3	←	t1 + 4
i4:	t4	←	t1 - 2
i5:	t5	←	t2 + 3
i6:	t6	←	t4 * t2
i7:	t7	←	t3 + t6
i8:	c	←	st t7
i9:	b	←	st t5

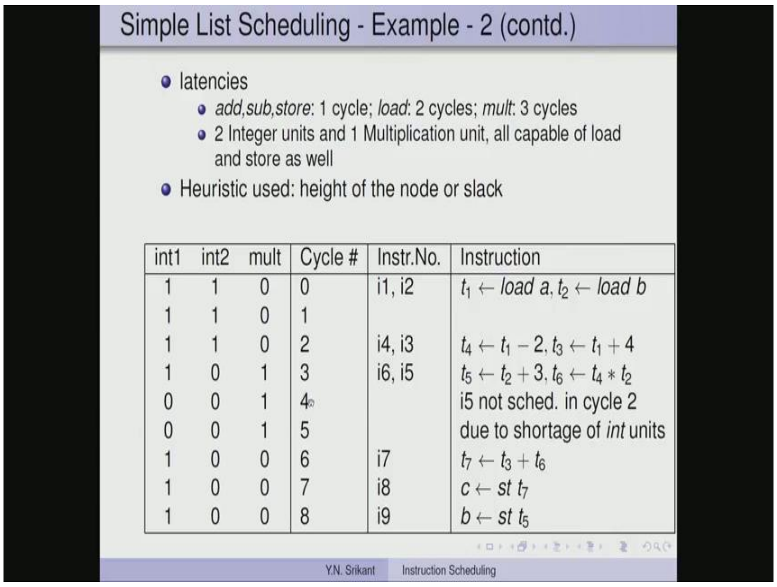
(b) 3-Address Code

(c) DAG with *(Estart, Lstart)* Values

Y.N. Srikant Instruction Scheduling

So, this i 1 i t2 i 3 i 4 i 5 so this is this multiply can be schedule now, so that is what we do so this not yet ready because it dependence on this.

(Refer Slide Time: 27:39)



Simple List Scheduling - Example - 2 (contd.)

- latencies
  - *add,sub,store*: 1 cycle; *load*: 2 cycles; *mult*: 3 cycles
  - 2 Integer units and 1 Multiplication unit, all capable of load and store as well
- Heuristic used: height of the node or slack

int1	int2	mult	Cycle #	Instr.No.	Instruction
1	1	0	0	i1, i2	$t_1 \leftarrow load\ a, t_2 \leftarrow load\ b$
1	1	0	1		
1	1	0	2	i4, i3	$t_4 \leftarrow t_1 - 2, t_3 \leftarrow t_1 + 4$
1	0	1	3	i6, i5	$t_5 \leftarrow t_2 + 3, t_6 \leftarrow t_4 * t_2$
0	0	1	4		i5 not sched. in cycle 2
0	0	1	5		due to shortage of *int* units
1	0	0	6	i7	$t_7 \leftarrow t_3 + t_6$
1	0	0	7	i8	$c \leftarrow st\ t_7$
1	0	0	8	i9	$b \leftarrow st\ t_5$

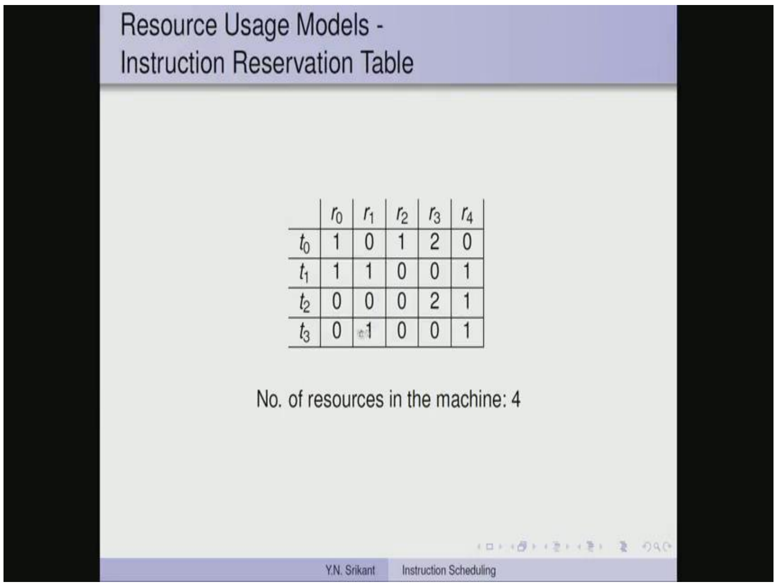
Y.N. Srikant Instruction Scheduling

So, we schedule i 5 and i 6 so 1 is on integer unit that is i 5 i 6 is multiplies so that goes on the multiply unit. Then until the multiplication is complete we really cannot do anything, so we have to wait. So, there is no other instruction that we can schedule. So, we just wait multiply takes 3 cycles we just have to wait for 2 more cycles. Then in slot number 6 the cycle number 6 we are ready to schedule i 7, there is no conflict here so we just schedule i 7.

Then the in the next cycle we i 8 is ready, so we schedule i 8 there is no conflict again and in i 9 we schedule in cycle number 8 and that is again schedule on one of the integer units. So, this is the way in which we work out the schedules starting from cycle number 0 we walk on a cycle by cycle basis, look at the instruction which are in the ready queue pick up those which are higher priority and schedule them.

So, what you must observe here is because the number of you know function unit this is more than one, we have scheduled more than one instruction. So, what we do is we just pick up one at a time based on priority and schedule them the available function unit, but we do not have to increment the cycle number because the resource is not a constrained at this point.

(Refer Slide Time: 29:15)



Resource Usage Models - Instruction Reservation Table

	r_0	r_1	r_2	r_3	r_4
t_0	1	0	1	2	0
t_1	1	1	0	0	1
t_2	0	0	0	2	1
t_3	0	1	0	0	1

No. of resources in the machine: 4

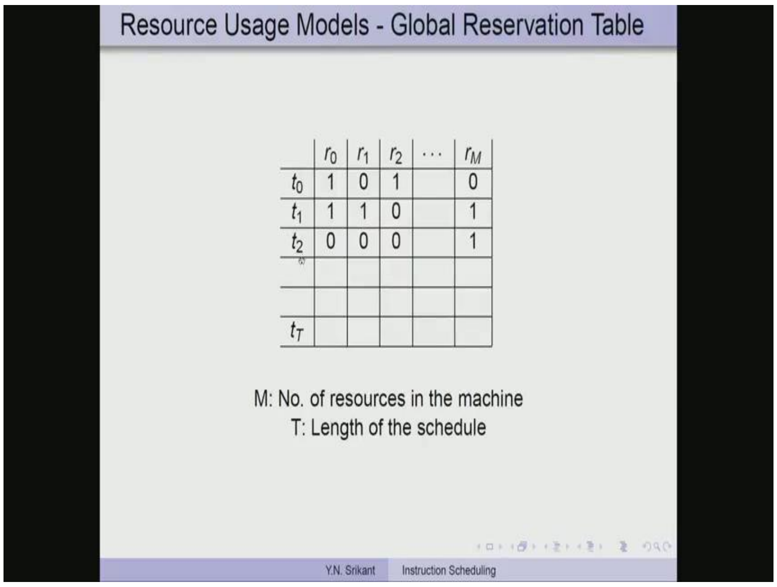
Y.N. Srikant Instruction Scheduling

Now, how do we provide an input to the scheduling algorithm in terms you know there resource rather, how do we provide the reservation the instruction resource requirements to the scheduling algorithm. So, this is provided in the form of instruction reservation table, this is a very simple table for each instruction we have a table of this kind. Let us, assume that the number of resources are in the machine is really 5 not 4 so r 0 r 1 r 2 r 3 r 4 so 0 to 4.

So, and then the instructions require maximum of 4 cycles right and in each one of this slots, we mention the number of resources that is required for that particular time slots. So, this instruction requires 4 time slots to complete and in the first time slots it requires r 0 r 2 and r 3, it requires 1 of r 0 1 of r 2 and 1 of r 3.

So, similarly, in the next cycle it requires r 0 r 1 and r 4, in t 2 it requires only r 3 and r 4 and in t 3 it requires r 1 and r 4. So, these are the resource requirement of this particular instruction and this table which is called as instruction reservation table, tells us the resource requirements of the instruction. So, there is going to be one table for each instruction in the machine.

(Refer Slide Time: 31:05)



Resource Usage Models - Global Reservation Table

	r_0	r_1	r_2	$\cdots$	r_M
t_0	1	0	1		0
t_1	1	1	0		1
t_2	0	0	0		1
t_T					

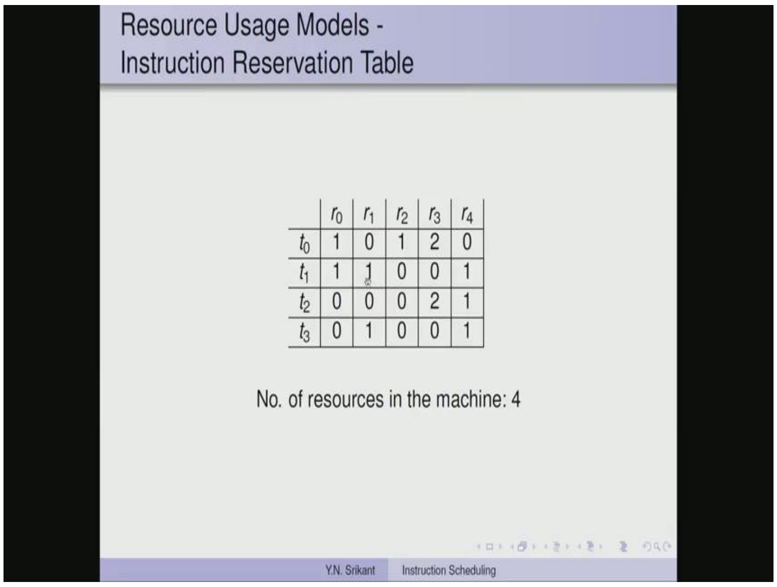
M: No. of resources in the machine
T: Length of the schedule

Y.N. Srikant Instruction Scheduling

Now, how do we keep track of the usage of the resources during the scheduling because if the number, if there are many resources we need to you know keep some kind of a table, which shows the usage of this resources. So, this indeed happens this called as a global reservation table, so it has as many column as the number of resources in the machine.

And the number of rows is equal to the length of the schedule. So, to begin with we do not know the number of rows, but once the schedule is complete this table will be t rows in length, in size. So, basically we start with the you know the slot t 0, then depending on the instructions which are ready to be scheduled that is there available in the ready queue and we the one with highest priority picked.

(Refer Slide Time: 32:17)



Resource Usage Models - Instruction Reservation Table

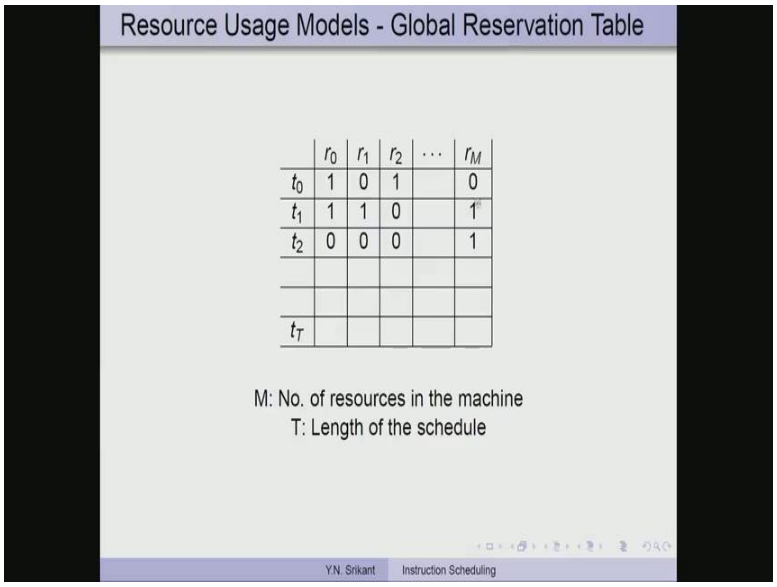
	r_0	r_1	r_2	r_3	r_4
t_0	1	0	1	2	0
t_1	1	1	0	0	1
t_2	0	0	0	2	1
t_3	0	1	0	0	1

No. of resources in the machine: 4

Y.N. Srikant Instruction Scheduling

So, and that instruction reservation table will look something like this.

(Refer Slide Time: 32:21)



Resource Usage Models - Global Reservation Table

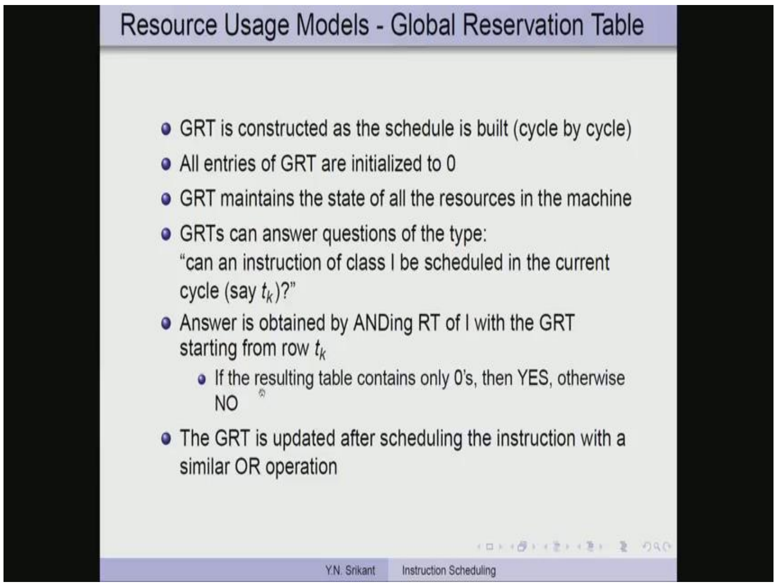
	r_0	r_1	r_2	$\cdots$	r_M
t_0	1	0	1		0
t_1	1	1	0		1
t_2	0	0	0		1
t_T					

M: No. of resources in the machine
T: Length of the schedule

Y.N. Srikant Instruction Scheduling

So, we super impose the instruction reservation table on this global resource reservation table and check whether the resource requirements of the instruction are met, how do we do that.

(Refer Slide Time: 32:38)



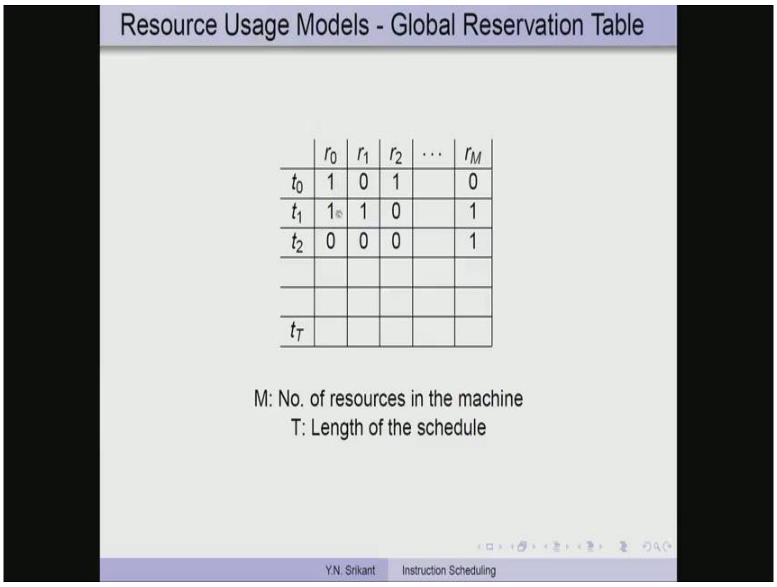
Resource Usage Models - Global Reservation Table

- GRT is constructed as the schedule is built (cycle by cycle)
- All entries of GRT are initialized to 0
- GRT maintains the state of all the resources in the machine
- GRTs can answer questions of the type: "can an instruction of class I be scheduled in the current cycle (say t_k)?"
- Answer is obtained by ANDing RT of I with the GRT starting from row t_k
  - If the resulting table contains only 0's, then YES, otherwise NO
- The GRT is updated after scheduling the instruction with a similar OR operation

Y.N. Srikant Instruction Scheduling

So, for that this is just a description to the GRT, it is constructed as the schedule is built cycle by cycle and all entries of the GRT are initialize to 0. So, the GRT maintain the state of all the resources in the machine and it can answer questions of the type, can an instruction of some class be scheduled in the current cycle say t k. How do we obtain this answer, this is obtained by ending the reservation table of the instruction with the GRT starting from that particular row. If the resulting table contains only 0's then yes otherwise, it is a no.

(Refer Slide Time: 33:21)



Resource Usage Models - Global Reservation Table

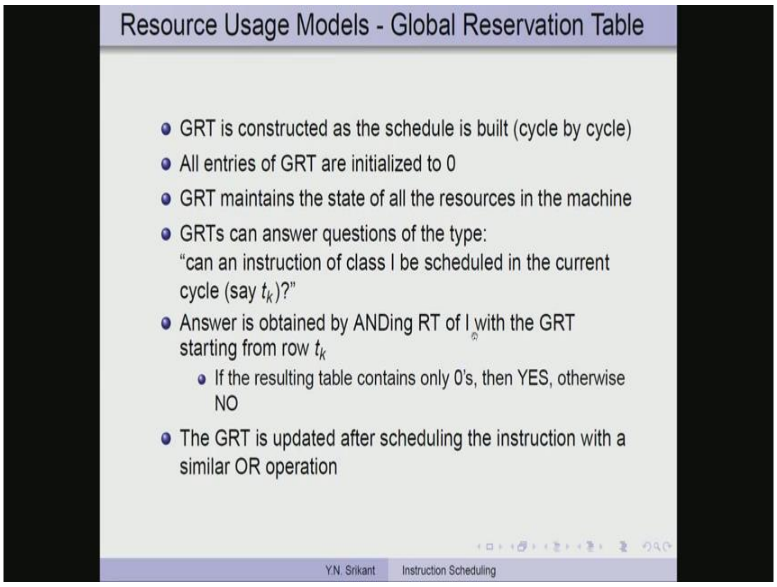
	r_0	r_1	r_2	$\cdots$	r_M
t_0	1	0	1		0
t_1	1	1	0		1
t_2	0	0	0		1
t_T					

M: No. of resources in the machine
T: Length of the schedule

Y.N. Srikant Instruction Scheduling

So, that is what I meant here, so u and v we keep the reservation table here and then and the appropriate entries. So, if the so that means, a you know if the reservation table of the instruction requires a particular resource say r 0 and the GRT already has a 1 here, anding of these two will produce a 1 that means, the resource is busy. So, what we require is a 0 here, if this were to be a 0 whereas, instruction reservation table had a 1 here, the anding operation would have produced 0 indicating that the resource is not busy.

(Refer Slide Time: 34:04)



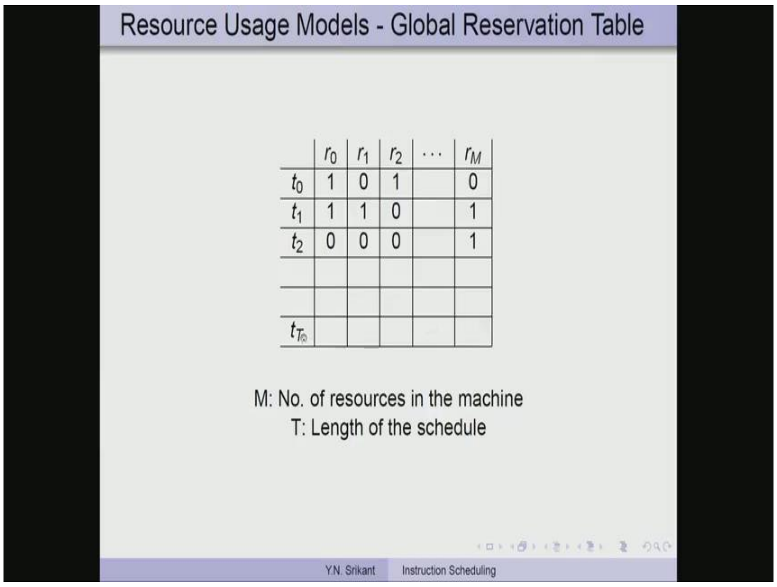
Resource Usage Models - Global Reservation Table

- GRT is constructed as the schedule is built (cycle by cycle)
- All entries of GRT are initialized to 0
- GRT maintains the state of all the resources in the machine
- GRTs can answer questions of the type: "can an instruction of class I be scheduled in the current cycle (say t_k)?"
- Answer is obtained by ANDing RT of I with the GRT starting from row t_k
  - If the resulting table contains only 0's, then YES, otherwise NO
- The GRT is updated after scheduling the instruction with a similar OR operation

Y.N. Srikant Instruction Scheduling

So, that is precisely what he said here, if the resulting table contains only 0's for the all rows and columns of the reservation table of the instruction you know that so many columns and so many rows. Then obliviously, all the resources required by the instruction are available and the instruction can be schedule otherwise, no. If the instruction can be scheduled after checking this, the GRT has to be updated. So, if this instruction is scheduled.

(Refer Slide Time: 34:40)



Resource Usage Models - Global Reservation Table

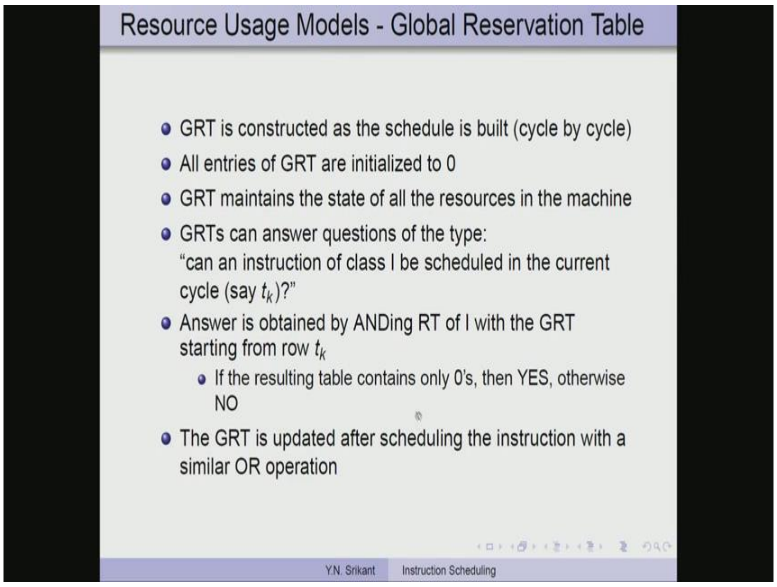
	r_0	r_1	r_2	$\cdots$	r_M
t_0	1	0	1		0
t_1	1	1	0		1
t_2	0	0	0		1
t_{T_0}					

M: No. of resources in the machine
T: Length of the schedule

Y.N. Srikant Instruction Scheduling

Then we similarly, you know we place the reservation table at the appropriate time slot, the reservation table of the instruction and do an or operation. So, then we actually make those resources, which are used by the instruction to be busy, so we put a one in all those places.

(Refer Slide Time: 34:58)



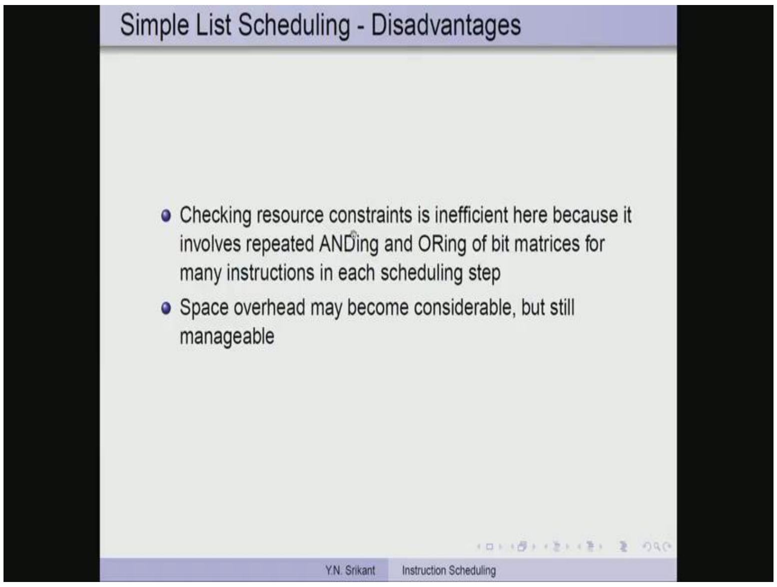
Resource Usage Models - Global Reservation Table

- GRT is constructed as the schedule is built (cycle by cycle)
- All entries of GRT are initialized to 0
- GRT maintains the state of all the resources in the machine
- GRTs can answer questions of the type:
  "can an instruction of class I be scheduled in the current cycle (say t_k)?"
- Answer is obtained by ANDing RT of I with the GRT starting from row t_k
  - If the resulting table contains only 0's, then YES, otherwise NO
- The GRT is updated after scheduling the instruction with a similar OR operation

Y.N. Srikant Instruction Scheduling

So, that precisely how the GRT is updated after the scheduling the instruction. So, that is about you know scheduling, instruction scheduling using the basic block scheduler.

(Refer Slide Time: 35:09)



Simple List Scheduling - Disadvantages

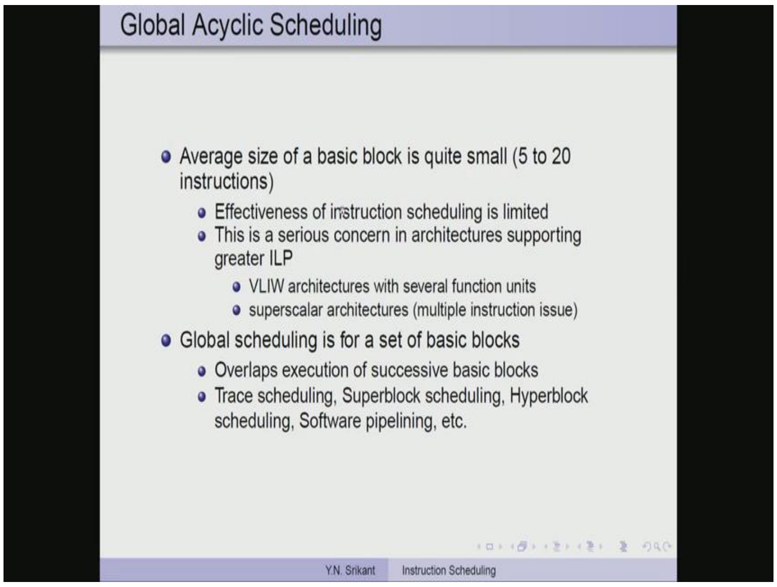
- Checking resource constraints is inefficient here because it involves repeated ANDing and ORing of bit matrices for many instructions in each scheduling step
- Space overhead may become considerable, but still manageable

Y.N. Srikant Instruction Scheduling

Such, a simple list scheduling strategy has some disadvantages, the first disadvantage is see checking the resource constraints is a very inefficient process here, because it involves repeated anding and oring of the of the bit matrices is for many instruction in each scheduling steps. So, this is a bit of inefficiency, but is not that bed space overhead may considerable, but this is not a serious issue.

The checking of resource constraints is the slower operation compare to space problems created by this, but still it is a very simple algorithm very effective and allows building you know building introducing many heuristic in to it for computation for the priority so it is still very validly used.

(Refer Slide Time: 36:09)



Global Acyclic Scheduling

- Average size of a basic block is quite small (5 to 20 instructions)
  - Effectiveness of instruction scheduling is limited
  - This is a serious concern in architectures supporting greater ILP
    - VLIW architectures with several function units
    - superscalar architectures (multiple instruction issue)
- Global scheduling is for a set of basic blocks
  - Overlaps execution of successive basic blocks
  - Trace scheduling, Superblock scheduling, Hyperblock scheduling, Software pipelining, etc.

Y.N. Srikant Instruction Scheduling

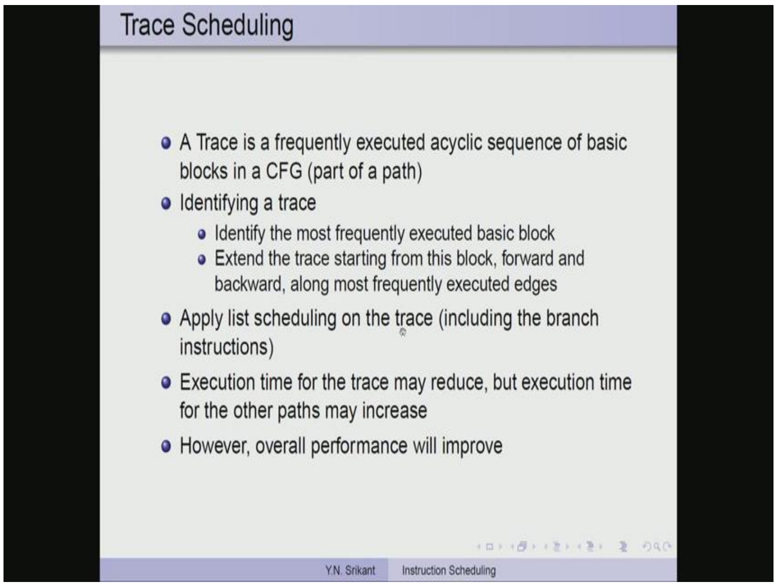
Now, you know let us move on to the next you know scheduling strategy, that is the global acyclic scheduling strategy. Let me explain why such a strategy is the in the required. It is so happens that the average size of basic block is small, in the in most applications say between 5 and 20 instructions. So, the instruction scheduling is not that effective, when we actually schedule very small number of instructions.

In other words you know there are not n of choices for the various slots so we may be forced to put no ops there so this. So, this is a serious concern in architecture supporting greater instruction level parallelism. So for example, VLIW architectures have several function units, superscalar architectures have multiple instruction issue possibilities. So, on such architectures, when we can initiate you know either more than one instruction percent cycle or at least one instruction per cycle. And then use the pipelining available in the machine to execute them in various phases.

So, in such machine very small basic blocks make the you know bring down the efficiency of the machine and make the program run slowly. So, global scheduling is actually go in you know is in the same spirit as the value numbering that we did for extended basic blocks. So, we take a set of basic blocks and try to schedule the instructions of basic blocks, a set of basic blocks as if there were a single basic block.

So, this overlaps execution of successive basic blocks and there are several techniques for it, one is straight scheduling, the second is super blocks scheduling, third is hyper blocks scheduling and forth is software pipelining. There are many more of course, and we will deal with only four of this in our discussion. So, I hope that kind of clarifies why we require looking at more than one basic block.

(Refer Slide Time: 38:44)



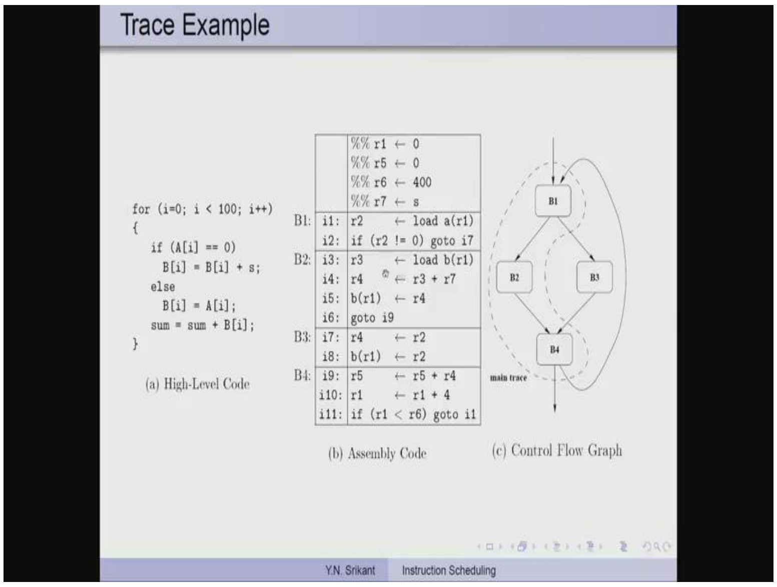
Trace Scheduling

- A Trace is a frequently executed acyclic sequence of basic blocks in a CFG (part of a path)
- Identifying a trace
  - Identify the most frequently executed basic block
  - Extend the trace starting from this block, forward and backward, along most frequently executed edges
- Apply list scheduling on the trace (including the branch instructions)
- Execution time for the trace may reduce, but execution time for the other paths may increase
- However, overall performance will improve

Y.N. Srikant   Instruction Scheduling

So, trace scheduling is a fairly widely applicable method and trace is a frequently executed acyclic sequence of basic blocks in a control flow graph, so that is part of a path. So, how do we identify a trace. So, we identify the most frequently executed basic block and then extend the trace starting from this block forward and backward along the most frequently executed edges.

(Refer Slide Time: 39:26)



Trace Example

```
for (i=0; i < 100; i++)
{
   if (A[i] == 0)
     B[i] = B[i] + s;
   else
     B[i] = A[i];
   sum = sum + B[i];
}
```

(a) High-Level Code

```
         %% r1  ← 0
         %% r5  ← 0
         %% r6  ← 400
         %% r7  ← s
B1: i1:  r2     ← load a(r1)
    i2:  if (r2 != 0) goto i7
B2: i3:  r3     ← load b(r1)
    i4:  r4     ← r3 + r7
    i5:  b(r1)  ← r4
    i6:  goto i9
B3: i7:  r4     ← r2
    i8:  b(r1)  ← r2
B4: i9:  r5     ← r5 + r4
    i10: r1     ← r1 + 4
    i11: if (r1 < r6) goto i1
```

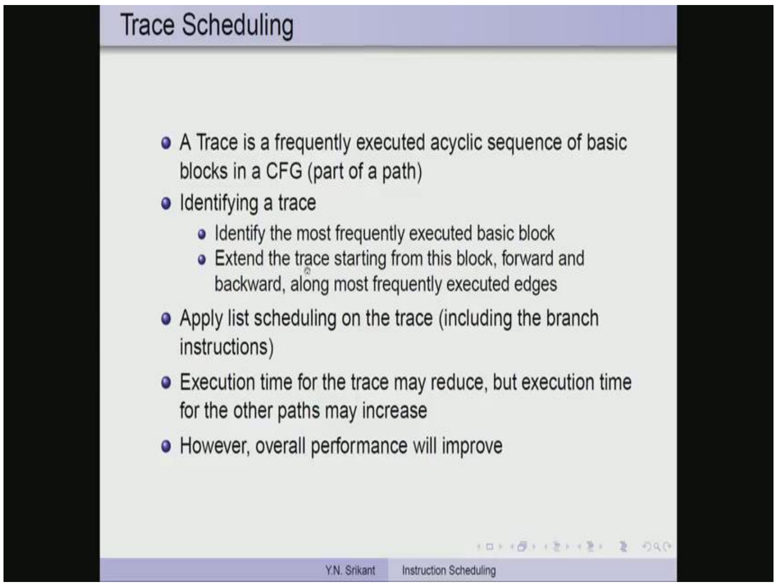
(b) Assembly Code

B1
B2
B3
B4
main trace

(c) Control Flow Graph

So, to show you an example, so this is the control flow graph. So, let us say this was the most frequently executed block, then grow it backwards and forwards and include this entire path as the main trace.

(Refer Slide Time: 39:43)

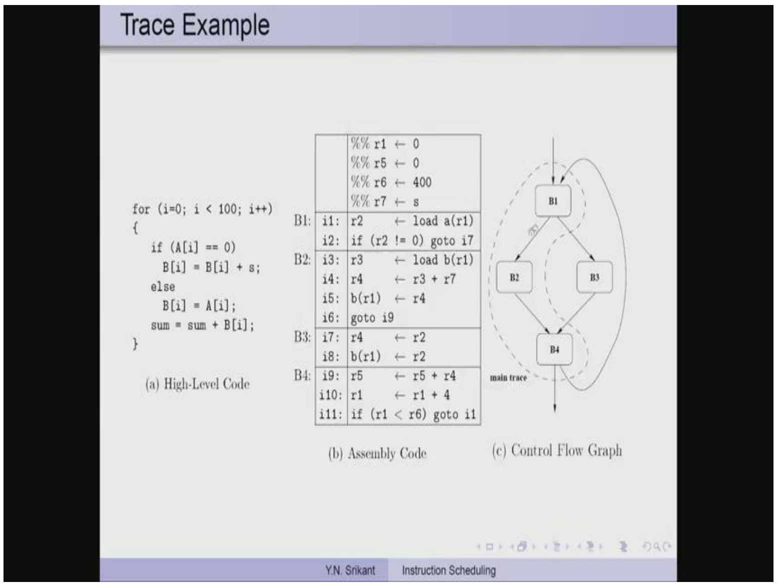


Trace Scheduling

- A Trace is a frequently executed acyclic sequence of basic blocks in a CFG (part of a path)
- Identifying a trace
  - Identify the most frequently executed basic block
  - Extend the trace starting from this block, forward and backward, along most frequently executed edges
- Apply list scheduling on the trace (including the branch instructions)
- Execution time for the trace may reduce, but execution time for the other paths may increase
- However, overall performance will improve

So, once we identify traces using profiling and you know this simple algorithm. We can apply list scheduling on the trace including the branch instructions of course, execution time for the trace may reduce now, but the execution time for the other path may increase, I will show you why this happens, but the overall performance will certainly improve.

(Refer Slide Time: 40:15)



Trace Example

```
for (i=0; i < 100; i++)
{
   if (A[i] == 0)
     B[i] = B[i] + s;
   else
     B[i] = A[i];
   sum = sum + B[i];
}
```

(a) High-Level Code

```
             %% r1   ← 0
             %% r5   ← 0
             %% r6   ← 400
             %% r7   ← s
B1:  i1:  r2      ← load a(r1)
     i2:  if (r2 != 0) goto i7
B2:  i3:  r3      ← load b(r1)
     i4:  r4      ← r3 + r7
     i5:  b(r1)   ← r4
     i6:  goto i9
B3:  i7:  r4      ← r2
     i8:  b(r1)   ← r2
B4:  i9:  r5      ← r5 + r4
     i10: r1      ← r1 + 4
     i11: if (r1 < r6) goto i1
```

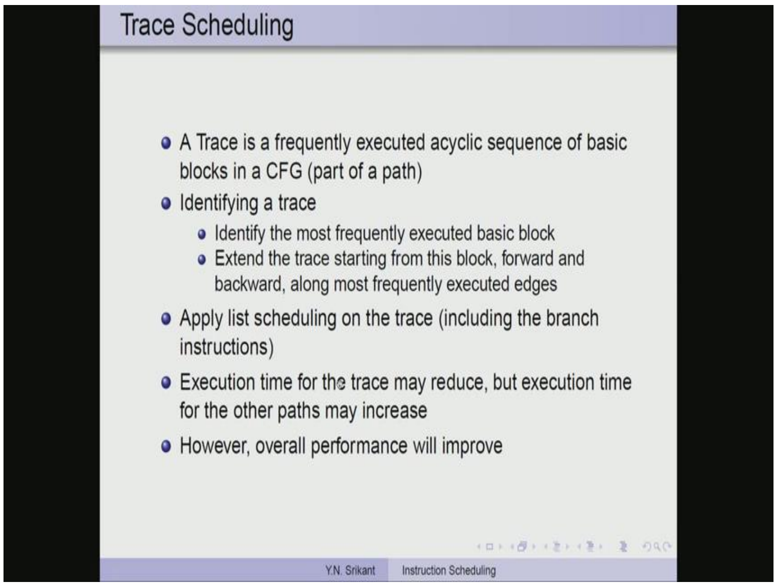
(b) Assembly Code

(c) Control Flow Graph

So for example, this is our main trace, what we really do is a we consider these 3 basic blocks as one unit or one basic block and try to schedule the instructions. So, we can move instructions between these basic blocks and that introduces you know some compensation code that we are going to see little later. So, let us assume that we can move this instruction among these basic blocks by doing so since we have many instruction we will probably reduce the execution time of this set of basic blocks, but then execution most of the time goes along this path, but sometimes it also goes along this path.

So, this is the outside of the trace, so this block the outside the trace. So, if we jump to this block apart from you know some compensation code etcetera, which needs to be executed we will see that later, jump into this block kind of brakes this pipeline. So, we may have to execute this block at a higher cost compare to what it was before.

(Refer Slide Time: 41:43)



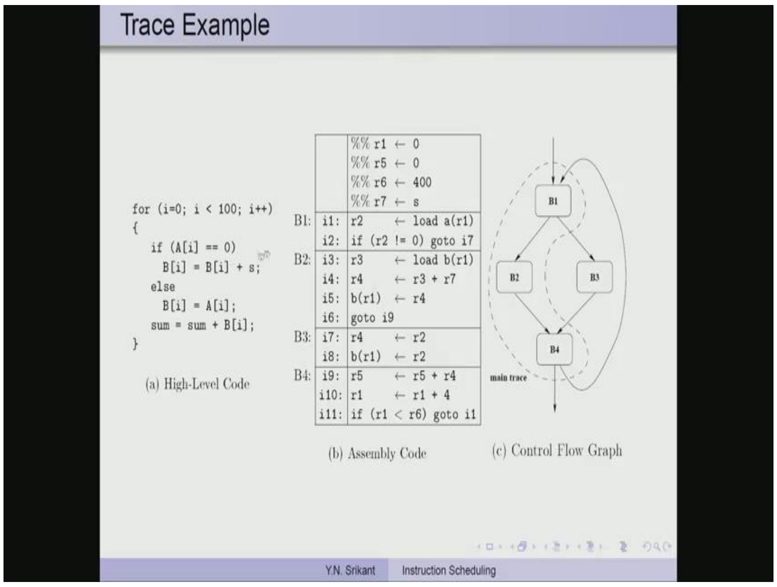
Trace Scheduling

- A Trace is a frequently executed acyclic sequence of basic blocks in a CFG (part of a path)
- Identifying a trace
  - Identify the most frequently executed basic block
  - Extend the trace starting from this block, forward and backward, along most frequently executed edges
- Apply list scheduling on the trace (including the branch instructions)
- Execution time for the trace may reduce, but execution time for the other paths may increase
- However, overall performance will improve

Y.N. Srikant Instruction Scheduling

So, that is what I was trying to explain here the execution time for the trace may reduce, but the execution time for the other path may increase, because of compensation code etcetera.

(Refer Slide Time: 41:51)



Trace Example

```
for (i=0; i < 100; i++)
{
   if (A[i] == 0)
     B[i] = B[i] + s;
   else
     B[i] = A[i];
   sum = sum + B[i];
}
```

(a) High-Level Code

```
         %% r1 ← 0
         %% r5 ← 0
         %% r6 ← 400
         %% r7 ← s
B1: i1:  r2     ← load a(r1)
    i2:  if (r2 != 0) goto i7
B2: i3:  r3     ← load b(r1)
    i4:  r4     ← r3 + r7
    i5:  b(r1)  ← r4
    i6:  goto i9
B3: i7:  r4     ← r2
    i8:  b(r1)  ← r2
B4: i9:  r5     ← r5 + r4
    i10: r1     ← r1 + 4
    i11: if (r1 < r6) goto i1
```

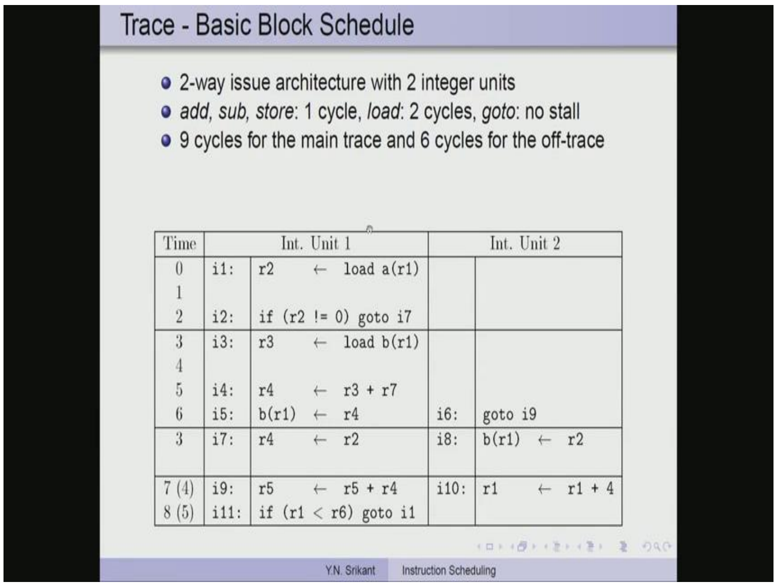
(b) Assembly Code

(c) Control Flow Graph

Y.N. Srikant Instruction Scheduling

So, let us consider this example so here in this example we have and if then else condition as well. So, in the if part we execute this and in the else particular we execute this and after the if-then else is over, we execute this. So, there are 4 basic blocks corresponding to these things, so this is the you know conditional block. Then we have the then part, we have the else part and we have the join corresponding to sum equal to sum plus b i. So, and here are the instruction corresponding to the 4 blocks?

(Refer Slide Time: 42:37)



Trace - Basic Block Schedule

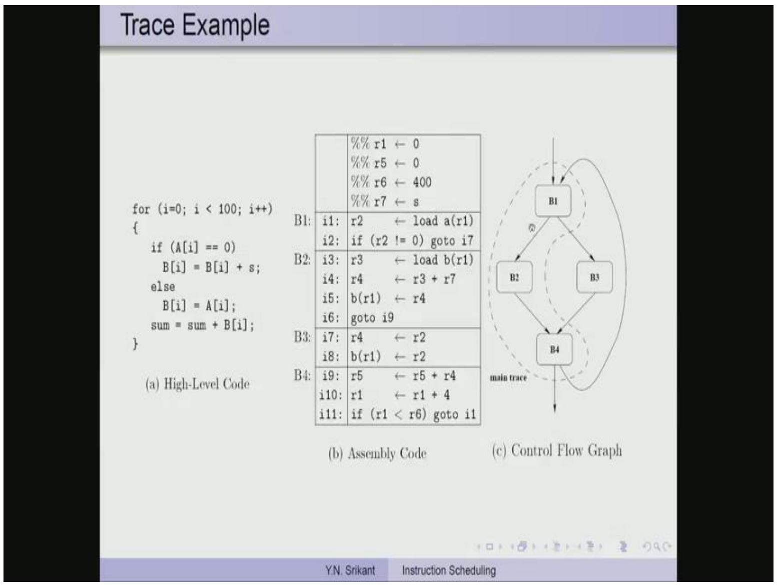
- 2-way issue architecture with 2 integer units
- *add*, *sub*, *store*: 1 cycle, *load*: 2 cycles, *goto*: no stall
- 9 cycles for the main trace and 6 cycles for the off-trace

Time	Int. Unit 1		Int. Unit 2	
0	i1:	r2 ← load a(r1)		
1				
2	i2:	if (r2 != 0) goto i7		
3	i3:	r3 ← load b(r1)		
4				
5	i4:	r4 ← r3 + r7		
6	i5:	b(r1) ← r4	i6:	goto i9
3	i7:	r4 ← r2	i8:	b(r1) ← r2
7 (4)	i9:	r5 ← r5 + r4	i10:	r1 ← r1 + 4
8 (5)	i11:	if (r1 < r6) goto i1		

Y.N. Srikant Instruction Scheduling

So, suppose we take that trace and then apply our basic blocks scheduling algorithm a very simple, we have not found we still have not separated them into main trace and the site trace etcetera. We just take each block and schedule it using the basic blocks scheduling algorithm that is all. So, if we do that then you know we are actually will be force to introduce a no op here, a no op here and therefore, the number of instructions taken for this particular program you know. So, 9 cycle for the main trace and 6 cycle for the off trace.

(Refer Slide Time: 43:31)



Trace Example

```
for (i=0; i < 100; i++)
{
   if (A[i] == 0)
     B[i] = B[i] + s;
   else
     B[i] = A[i];
   sum = sum + B[i];
}
```

(a) High-Level Code

```
          %% r1  ← 0
          %% r5  ← 0
          %% r6  ← 400
          %% r7  ← s
B1:  i1:  r2     ← load a(r1)
     i2:  if (r2 != 0) goto i7
B2:  i3:  r3     ← load b(r1)
     i4:  r4     ← r3 + r7
     i5:  b(r1)  ← r4
     i6:  goto i9
B3:  i7:  r4     ← r2
     i8:  b(r1)  ← r2
B4:  i9:  r5     ← r5 + r4
    i10:  r1     ← r1 + 4
    i11:  if (r1 < r6) goto i1
```

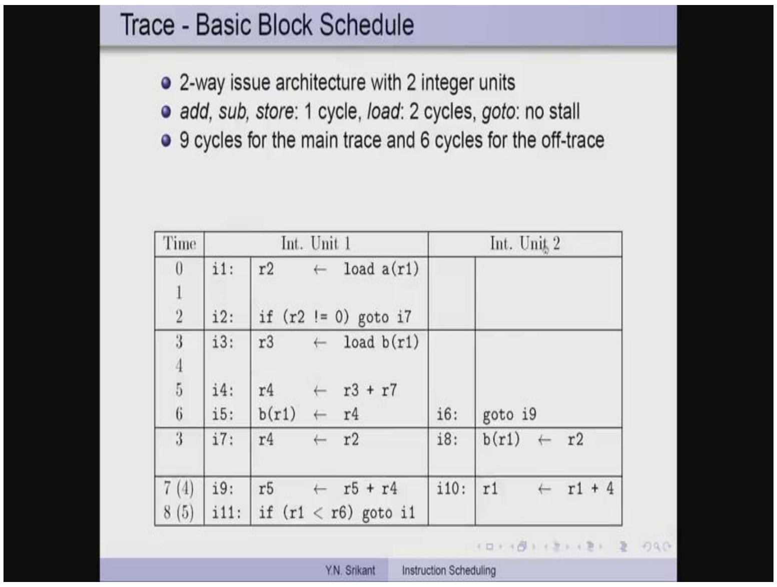
(b) Assembly Code

(c) Control Flow Graph

Y.N. Srikant Instruction Scheduling

So, remember we have identify that trace, but that is this is the main trace and this is the off trace, but this is the block in the off trace, but we did not apply any scheduling algorithm combining these basic blocks. We are still applying basic block scheduling algorithm separately for b 1 b 2 b 4 and b 3 that is the idea.

(Refer Slide Time: 43:49)



Trace - Basic Block Schedule

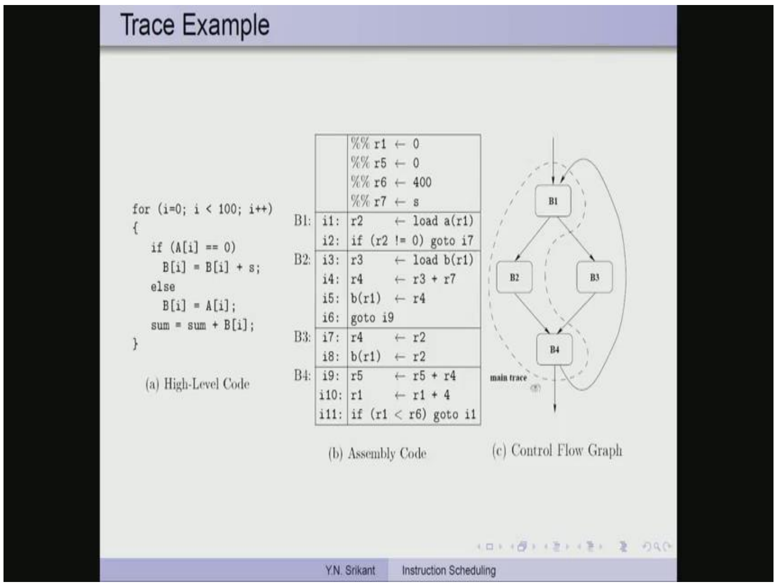
- 2-way issue architecture with 2 integer units
- *add*, *sub*, *store*: 1 cycle, *load*: 2 cycles, *goto*: no stall
- 9 cycles for the main trace and 6 cycles for the off-trace

Time	Int. Unit 1		Int. Unit 2	
0	i1:	r2 ← load a(r1)		
1				
2	i2:	if (r2 != 0) goto i7		
3	i3:	r3 ← load b(r1)		
4				
5	i4:	r4 ← r3 + r7		
6	i5:	b(r1) ← r4	i6:	goto i9
3	i7:	r4 ← r2	i8:	b(r1) ← r2
7 (4)	i9:	r5 ← r5 + r4	i10:	r1 ← r1 + 4
8 (5)	i11:	if (r1 < r6) goto i1		

Y.N. Srikant Instruction Scheduling

That is the comparison part so how does this take a 9 cycles for this basic block we require 3 cycles and then if we go to i 7 that would be i 7 is here, that would be the off trace. So, if we continue here that would be the main trace, so 0 1 2 3 4 5 6 all these corresponds to the main trace, and then we also have you know i 7. So, after this 6 we have a go to instruction which will bring us to 7 that is here i 9.

(Refer Slide Time: 44:34)



Trace Example

```
for (i=0; i < 100; i++)
{
   if (A[i] == 0)
     B[i] = B[i] + s;
   else
     B[i] = A[i];
   sum = sum + B[i];
}
```

(a) High-Level Code

		%% r1 ← 0
		%% r5 ← 0
		%% r6 ← 400
		%% r7 ← s
B1:	i1:	r2 ← load a(r1)
	i2:	if (r2 != 0) goto i7
B2:	i3:	r3 ← load b(r1)
	i4:	r4 ← r3 + r7
	i5:	b(r1) ← r4
	i6:	goto i9
B3:	i7:	r4 ← r2
	i8:	b(r1) ← r2
B4:	i9:	r5 ← r5 + r4
	i10:	r1 ← r1 + 4
	i11:	if (r1 < r6) goto i1

(b) Assembly Code

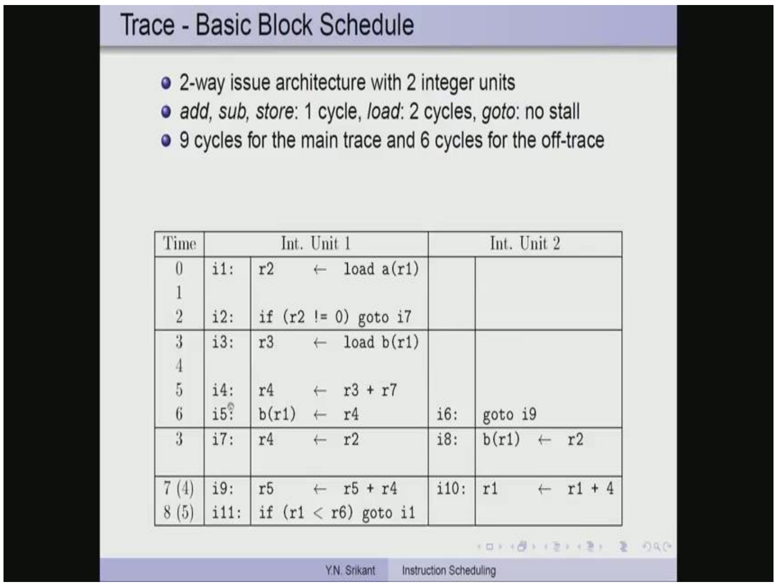
B1
B2
B3
B4
main trace

(c) Control Flow Graph

Y.N. Srikant Instruction Scheduling

So, this corresponds to the merger block here, this block, so we execute this we execute this and then jump to this or we jump right in the beginning, jump to this point execute this and then fall through to b 4.

(Refer Slide Time: 44:48)



Trace - Basic Block Schedule

- 2-way issue architecture with 2 integer units
- *add*, *sub*, *store*: 1 cycle, *load*: 2 cycles, *goto*: no stall
- 9 cycles for the main trace and 6 cycles for the off-trace

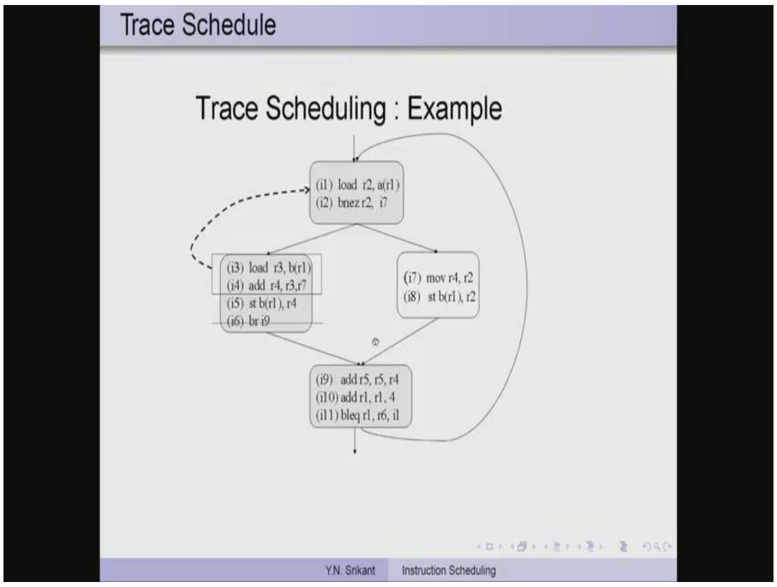
Time	Int. Unit 1		Int. Unit 2	
0	i1:	r2 ← load a(r1)		
1				
2	i2:	if (r2 != 0) goto i7		
3	i3:	r3 ← load b(r1)		
4				
5	i4:	r4 ← r3 + r7		
6	i5:	b(r1) ← r4	i6:	goto i9
3	i7:	r4 ← r2	i8:	b(r1) ← r2
7 (4)	i9:	r5 ← r5 + r4	i10:	r1 ← r1 + 4
8 (5)	i11:	if (r1 < r6) goto i1		

Y.N. Srikant Instruction Scheduling

So, these are the two possibilities so if we fall through here then we execute b 1 and b 2 then we jump to b 4 execute this. So, after 6 we have 7 and 8 so that means, we have 9 cycles main trace, for the off trace we defiantly have to execute b 1 then we go to i seven execute this instruction and then fall through and execute the join. So, that would require a 3 cycle here you know, so this is the fourth cycle, fifth cycle and sixth cycle so that that is what we require.

So, we require a 6 cycles for the off trace and 9 cycles for the main trace so that is our scheduling. And we have 2 integer units available, so we can schedule instruction freely on either one of them based on the dependences. So, this is a 2 way issue architecture with 2 integer units. So, we can issue instructions 2 instructions in the same cycle also and of course, it requires 1 cycle for add sub store, 2 cycles for load and goto has no stall so we can actually schedule this and something else also.

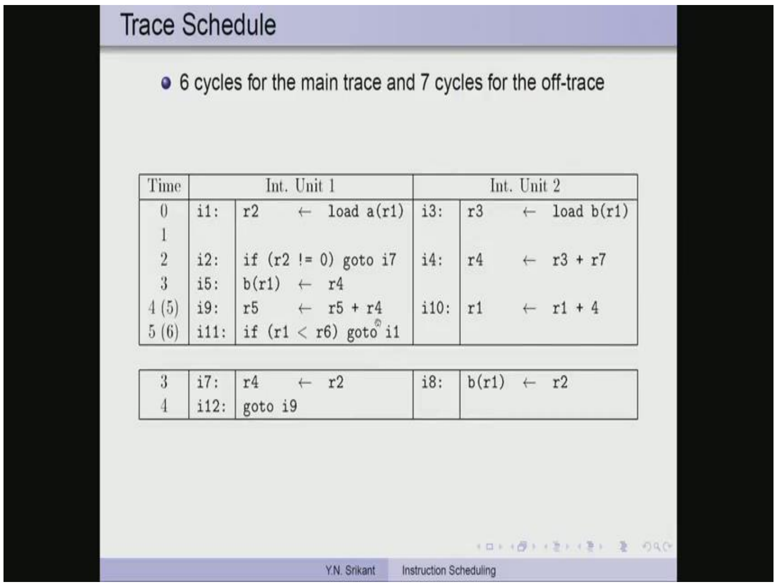
(Refer Slide Time: 46:09)



Suppose, we consider these 3 blocks as a single block and schedule so this is the trace scheduling what we did so far was not trace scheduling. So, we have in effect we would be moving some of the instruction from here, to this part and then we are also kind of deleting this branch because the flow of control will be maintained like this. So. we are not we are kind of falling through from execution from here to here, we do not have to jump.

Whereas, for the off trace it is going to be different, so these instructions will be the effect is to move them here and then we have a store and then we have this three instructions that would be our main trace. And then the off trace would jump to this point execute this and then jump to this and execute this, so it takes more time. So, here there are no jumps whereas, on this part there are jumps, so this is the where the trace scheduling would happened.

(Refer Slide Time: 47:17)



Trace Schedule

- 6 cycles for the main trace and 7 cycles for the off-trace

Time	Int. Unit 1		Int. Unit 2	
0	i1:	r2 ← load a(r1)	i3:	r3 ← load b(r1)
1				
2	i2:	if (r2 != 0) goto i7	i4:	r4 ← r3 + r7
3	i5:	b(r1) ← r4		
4 (5)	i9:	r5 ← r5 + r4	i10:	r1 ← r1 + 4
5 (6)	i11:	if (r1 < r6) goto i1		

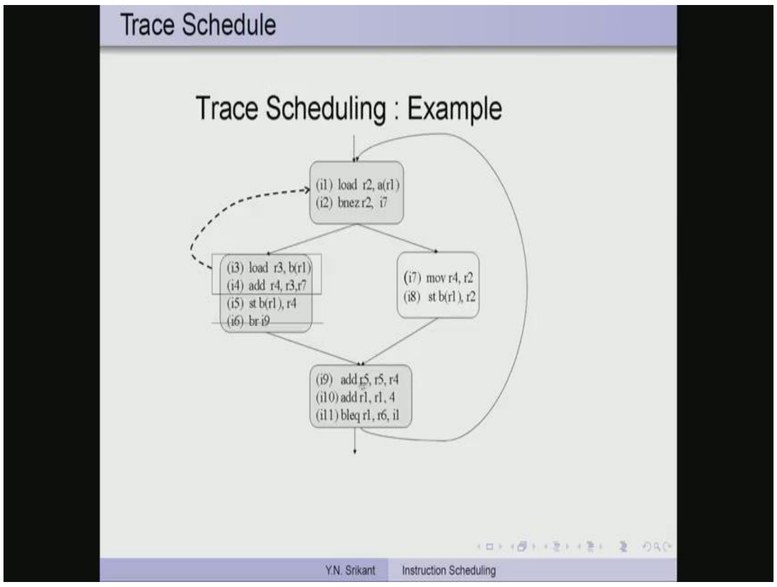
3	i7:	r4 ← r2	i8:	b(r1) ← r2
4	i12:	goto i9		

Y.N. Srikant Instruction Scheduling

So, let us look at it, so it requires 6 cycles for the main trace and 7 cycles for the off trace. Whereas, we had required 9 cycles for the main trace and 6 for the off trace in the normal scheduling you know application. So, this is the integer unit 1 integer unit 2, so this our main trace and this is the off trace block. So, how does the you know control go, so we have been able to schedule the instruction in a mixed manner.

So, if this is the condition so if the condition is false we go to i 7, so that is the braking the main trace. So, otherwise we fall through we continue and this is the loop actually. So, if r 1 less than r 6 go back to i 1, so as long as we are executing the main trace we will be doing it very fast. So, the number of cycles required 0 1 2 3 4 and 5 so that would be for the main trace. Now, how does the off trace part execute so that is this part.

(Refer Slide Time: 48:33)



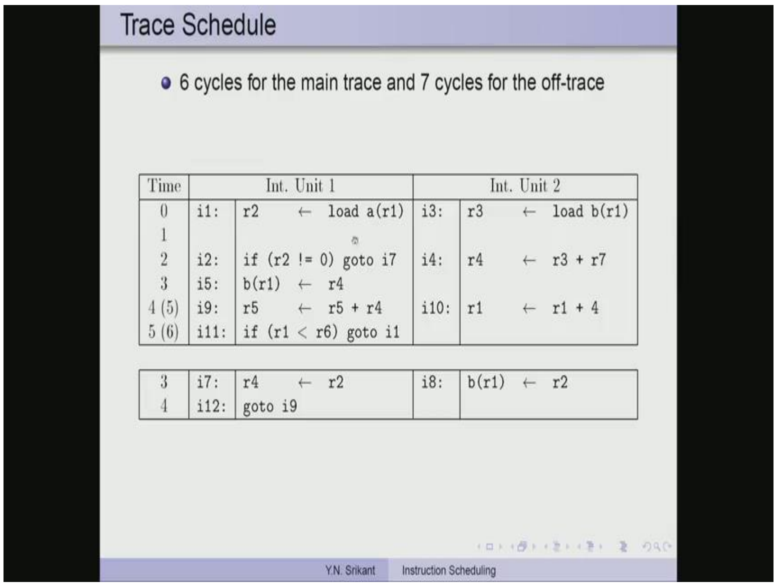
Trace Schedule

Trace Scheduling : Example

Y.N. Srikant Instruction Scheduling

So, as I told you we have to jump to this part, execute this and then jump again into the middle of the main trace, execute these instructions and then get out.

(Refer Slide Time: 48:47)



Trace Schedule

- 6 cycles for the main trace and 7 cycles for the off-trace

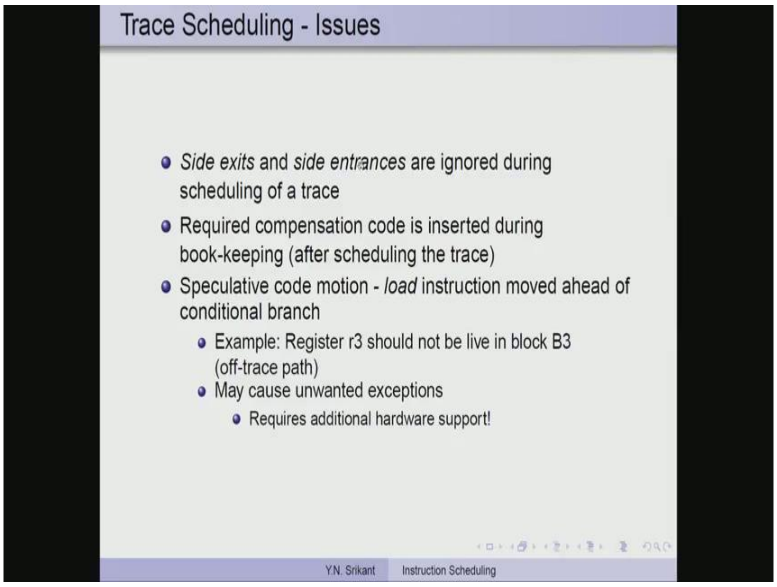
Time	Int. Unit 1		Int. Unit 2	
0	i1:	r2 ← load a(r1)	i3:	r3 ← load b(r1)
1				
2	i2:	if (r2 != 0) goto i7	i4:	r4 ← r3 + r7
3	i5:	b(r1) ← r4		
4 (5)	i9:	r5 ← r5 + r4	i10:	r1 ← r1 + 4
5 (6)	i11:	if (r1 < r6) goto i1		

3	i7:	r4 ← r2	i8:	b(r1) ← r2
4	i12:	goto i9		

Y.N. Srikant Instruction Scheduling

So, here we go to i 7 execute r 4 equal to r 2 then we go to the middle of the main trace so that is i 9 that is here, execute this instruction, this instruction and also this jump instruction. So, at this point you know obviously we require 0 1 2 so 3 cycles then you know 3 4 that is 5 cycles and then 5 6 that is 7 cycles. So, that is what is here you know the main trace is very fast, but the off trace because of the two jumps requires more time to execute.

(Refer Slide Time: 49:33)



Trace Scheduling - Issues

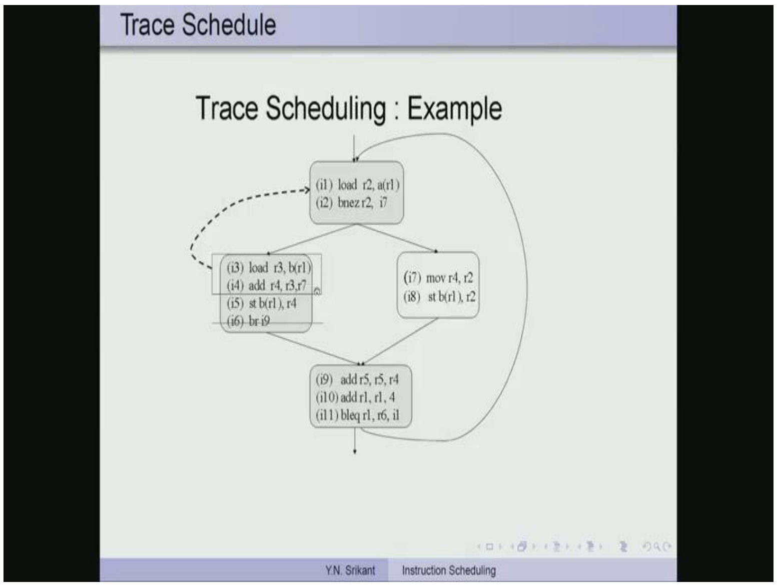
- *Side exits* and *side entrances* are ignored during scheduling of a trace
- Required compensation code is inserted during book-keeping (after scheduling the trace)
- Speculative code motion - *load* instruction moved ahead of conditional branch
    - Example: Register r3 should not be live in block B3 (off-trace path)
    - May cause unwanted exceptions
        - Requires additional hardware support!

Y.N. Srikant Instruction Scheduling

So, this is the trace scheduling, but then as I told you we require some extra code to be introduce into the various blocks. So, the side exits and side entrances are ignored during the scheduling of a trace and this requires compensation code to be inserted, during the book keeping phase after this scheduling of the trace. So, basically for the main trace we do the scheduling then check whether the instructions have been displaced from their original position.

And then and see if extra code has to be introduced and we introduce it in the off trace. So, there are also other possible side effect, so one is the book keeping code which I am going to show you very soon. The speculative code motion load instruction moved ahead of conditional branch, so in our example so the register r 3 should not be live in the off trace path.

(Refer Slide Time: 50:44)

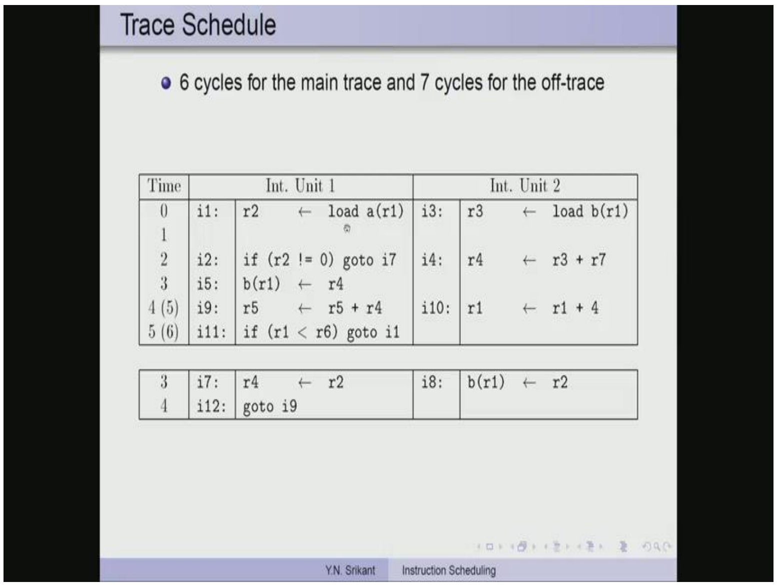


Trace Schedule

Trace Scheduling : Example

So, let me show you so here as I said this instructions are all going to be effectively moved here. So, that means the load instruction also moves here and the register r 3 would be loaded and after that suppose we take this branch right. So, the load has already kind of happened so now you can see that.

(Refer Slide Time: 51:08)



Trace Schedule

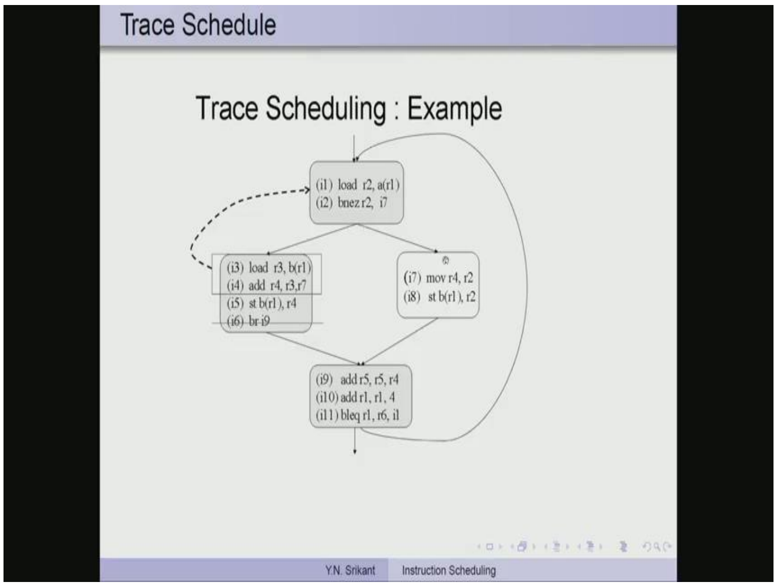
- 6 cycles for the main trace and 7 cycles for the off-trace

Time	Int. Unit 1		Int. Unit 2	
0	i1:	r2 ← load a(r1)	i3:	r3 ← load b(r1)
1				
2	i2:	if (r2 != 0) goto i7	i4:	r4 ← r3 + r7
3	i5:	b(r1) ← r4		
4 (5)	i9:	r5 ← r5 + r4	i10:	r1 ← r1 + 4
5 (6)	i11:	if (r1 < r6) goto i1		

3	i7:	r4 ← r2	i8:	b(r1) ← r2
4	i12:	goto i9		

So, the load is going on its own, so it has been scheduled in parallel with this load. So, the load of r 3 has completed by the time you jump to the off trace so that is here. So, this r 3 is still live at this point because the load has already completed at this point.

(Refer Slide Time: 51:29)



Trace Schedule

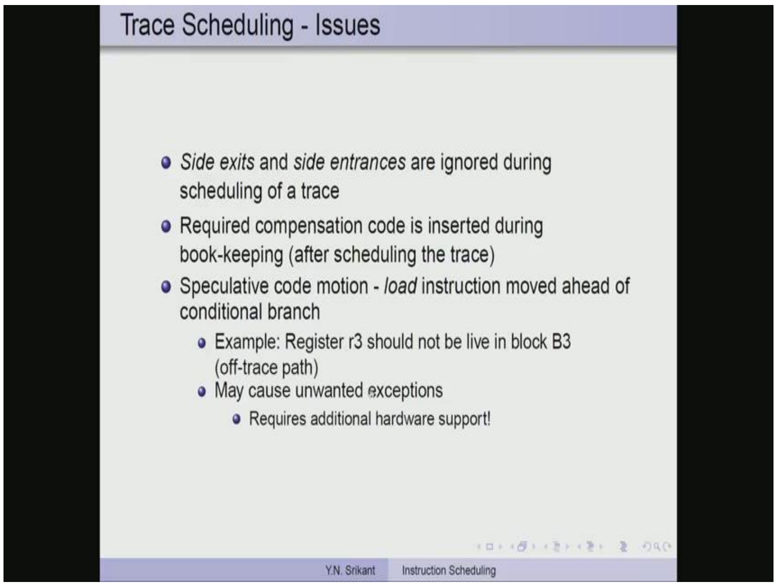
Trace Scheduling : Example

```
(i1) load r2, a(r1)
(i2) bnez r2, i7

(i3) load r3, b(r1)
(i4) add r4, r3,r7
(i5) st b(r1), r4
(i6) br i9

(i7) mov r4, r2
(i8) st b(r1), r2

(i9) add r5, r5, r4
(i10) add r1, r1, 4
(i11) bleq r1, r6, i1
```

Y.N. Srikant Instruction Scheduling

So, we come here, but we find that r 3 is live whereas, in the original control flow diagram r 3 was loaded here in some other block. So, if we had made an exit at this point r 3 would not have been live. So, this is a side effect the load has been moved speculatively to this point assuming that the main trace should be taken, but the main trace was not taken. So, the register r 3 is live even in this block.

(Refer Slide Time: 51:59)



Trace Scheduling - Issues

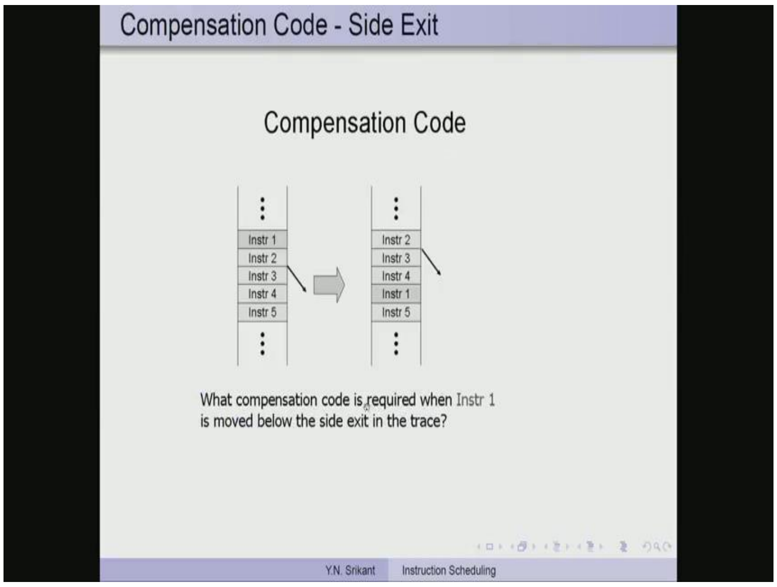
- *Side exits* and *side entrances* are ignored during scheduling of a trace
- Required compensation code is inserted during book-keeping (after scheduling the trace)
- Speculative code motion - *load* instruction moved ahead of conditional branch
  - Example: Register r3 should not be live in block B3 (off-trace path)
  - May cause unwanted exceptions
    - Requires additional hardware support!

Y.N. Srikant Instruction Scheduling

The side effect of that is possibly some unwanted exceptions, so this is not easy to take care of. It requires additional hardware support to detect such exceptions and make sure that some repairer is cost executed repair is performed. So, trace scheduling requires some extra hardware support to take care of such unwanted exceptions.

These are not supposed to have been cause, but because of the main trace being scheduled separately this has been caused, but it should not cause difficulty when the off trace is taken. So, such unwanted exceptions should be caught and delt with appropriately by the hardware.

(Refer Slide Time: 52:47)



Compensation Code - Side Exit

Compensation Code

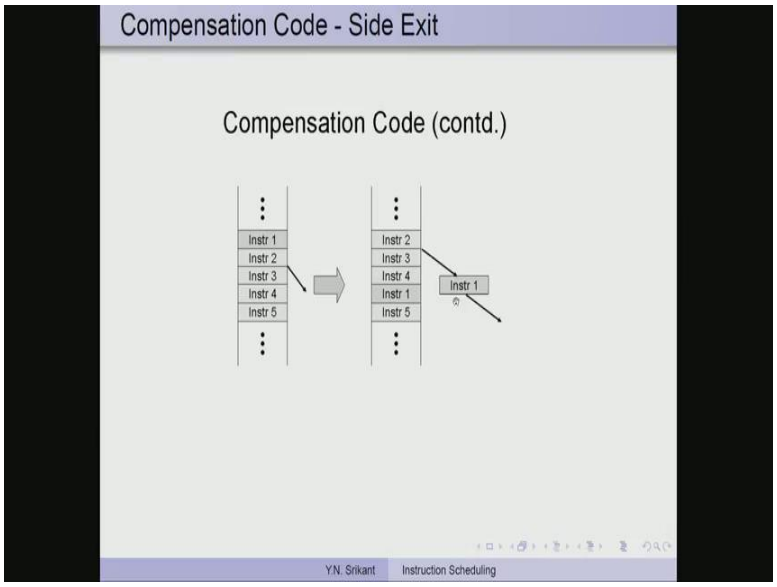
What compensation code is required when Instr 1 is moved below the side exit in the trace?

Y.N. Srikant Instruction Scheduling

Now, the compensation code, so this is the original sequence of instructions, instruction 1 2 3 4 5 and these are the instructions which possibly corresponds to many blocks of the main trace. Now, suppose the instruction sequence here is modified to this, so we 2 3 4 and then instruction 1 and then followed by the instruction 5.

So, if we were actually exiting after instruction 2 here, right so in the original sequence we would have a executed 1 then 2. And maybe would have a executed exited to the off trace in the some iterations. Whereas, in this case now we do not execute instruction 1 at all, we simply execute instruction 2 and then we exit, so this is incorrect.

(Refer Slide Time: 53:42)



Compensation Code - Side Exit

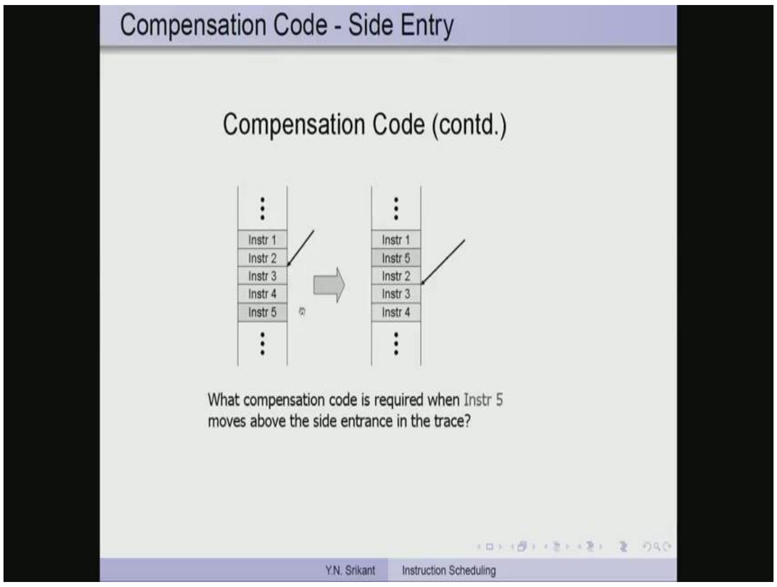
Compensation Code (contd.)

Instr 1
Instr 2
Instr 3
Instr 4
Instr 5
Instr 2
Instr 3
Instr 4
Instr 1
Instr 5
Instr 1

Y.N. Srikant Instruction Scheduling

What we need to do is to insert instruction 1 in this path along this edge. So, this is the extra compensation code that is executed and there is nothing wrong in executing instruction 1 after instruction 2 because if we had use the main trace, we would have still executed instruction 1 after instruction 2. So, dependence is permitted so there is nothing wrong in inserting instruction 1 at this point. So, this is one time of compensation code.

(Refer Slide Time: 54:11)



Compensation Code - Side Entry

Compensation Code (contd.)

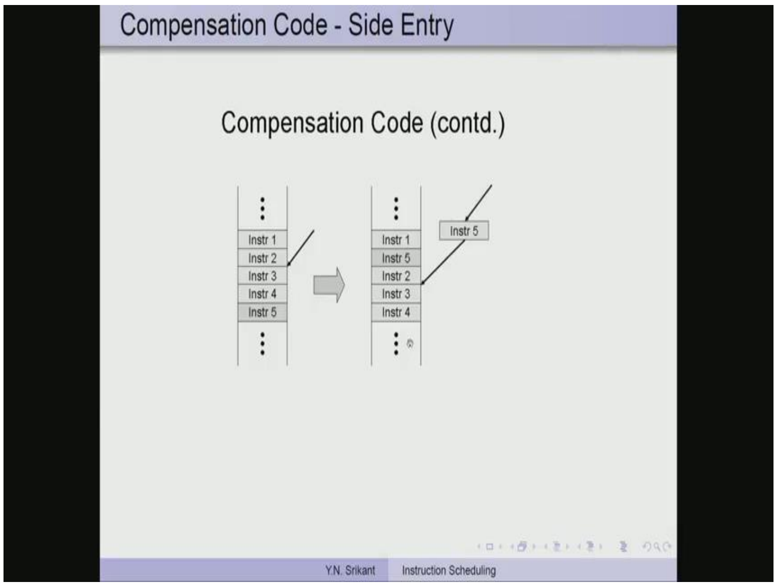
Instr 1
Instr 2
Instr 3
Instr 4
Instr 5
Instr 1
Instr 5
Instr 2
Instr 3
Instr 4

What compensation code is required when Instr 5 moves above the side entrance in the trace?

Y.N. Srikant Instruction Scheduling

Suppose, we had same sequence 1 2 3 4 5, so there was possibly a jump into the middle of this code to instruction 3 from outside. So, we would have now change the order of these instructions for the main trace so it has now become 1 then 5, then 2, then 3, then 4. So, what compensation code is required when instruction 5 moves above the side entrance in the trace? So, what the problem now is if we actually enter through this you know edge, we execute 3 and 4, but we do not execute 5 at all, 5 has mood up.

(Refer Slide Time: 55:01)



Compensation Code - Side Entry

Compensation Code (contd.)

Y.N. Srikant Instruction Scheduling

Obviously, the compensation code that is instruction 5 has to be inserted along this edge. And there is nothing wrong in inserting it here and executing it before 3 because even in the main trace we have scheduled instruction number 5 before instruction number 3. So, this is the compensation code that has to be inserted. So, compensation code actually can become quite large in some cases and this is the one of the disadvantage of trace scheduling. We will stop here and continue with the other type of scheduling in the next part of the lecture.

Thank you.