Compiler Design

Prof. Y. N. Srikant

Department of Computer Science and Automation

Indian Institute of Technology, Bangalore

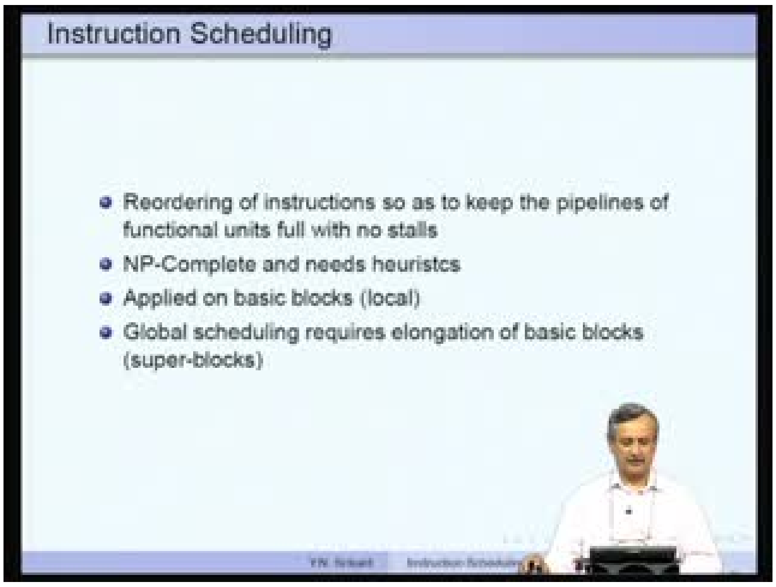
Module No. # 15

Lecture No. # 29

Instruction Scheduling-Part 2

.

(Refer Slide Time: 00:20)

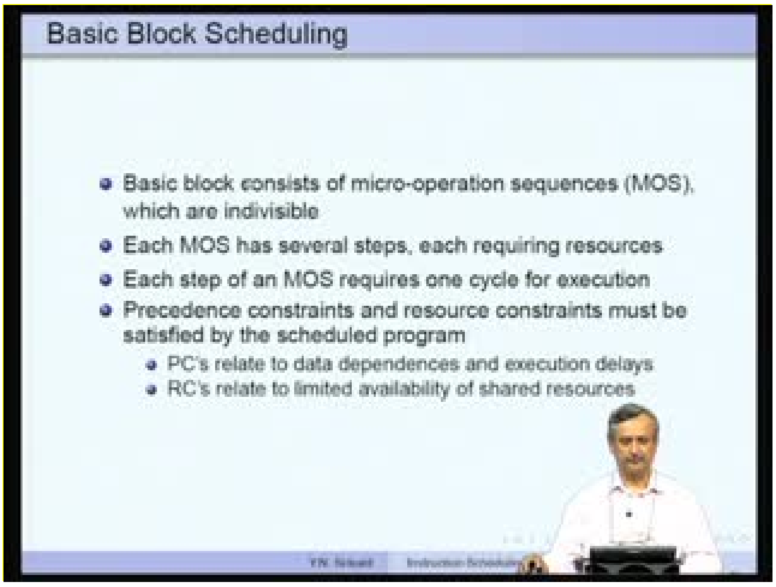


Instruction Scheduling

- Reordering of instructions so as to keep the pipelines of functional units full with no stalls
- NP-Complete and needs heuristcs
- Applied on basic blocks (local)
- Global scheduling requires elongation of basic blocks (super-blocks)

Welcome to part 2 of the lecture on instruction scheduling. So, to recapitulate a little bit, instruction scheduling is nothing but reordering of instructions so as, to keep the pipelines of functional units full and make sure that there are no stalls. This is an NP-Complete problem, as are actually all good problems in computer science. So, we need heuristics, in order to solve this problem and each heuristic will have some efficacy and some may be better than the others. We can apply heuristics on basic blocks and that would become local instruction scheduling; if we consider extended basic blocks called super blocks, then the scheduling strategy would be called as a global instruction scheduling strategy.

(Refer Slide Time: 01:16)



Basic Block Scheduling

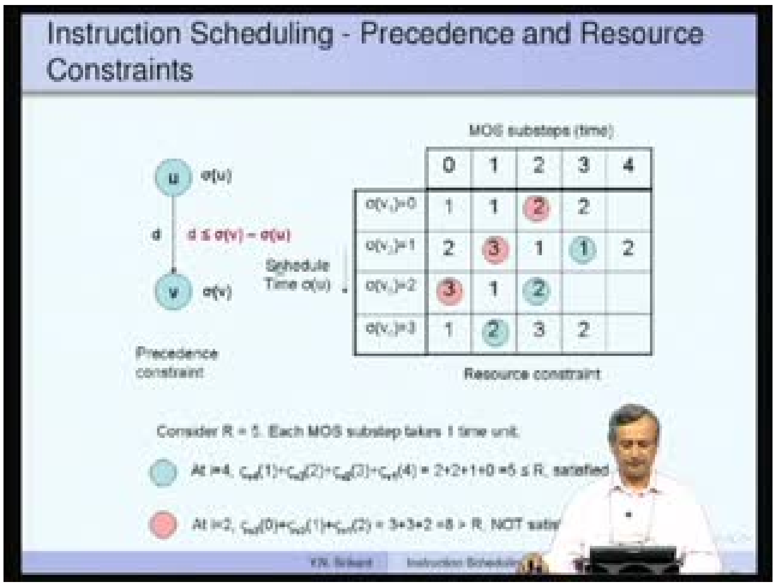
- Basic block consists of micro-operation sequences (MOS), which are indivisible
- Each MOS has several steps, each requiring resources
- Each step of an MOS requires one cycle for execution
- Precedence constraints and resource constraints must be satisfied by the scheduled program
  - PC's relate to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

What is basic block scheduling? We consider a model of basic blocks to consist of various micro operation sequences; these are the instructions that we have. Each MOS is indivisible; so it is pipeline, but it is indivisible. Each MOS has several steps and each of these sub steps will require resources. We assume that each MOS sub step will require one cycle for execution.

There are going to be constraints on the reordering process. So some of them are precedence constraints; some of them are resource constraints.

Precedence constraints arise because of data dependences and execution delays whereas; resource constraints arise because of the availability or unavailability of the shared resources.

(Refer Slide Time: 02:10)



Instruction Scheduling - Precedence and Resource Constraints

MOS substeps (time)

	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	

Resource constraint

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

● At i=4, $c_{v_4}(1)+c_{v_3}(2)+c_{v_2}(3)+c_{v_1}(4)$ = 2+2+1+0 =5 ≤ R, satisfied

● At i=2, $c_{v_3}(0)+c_{v_2}(1)+c_{v_1}(2)$ = 3+3+2 =8 > R, NOT satis

Y.N. Srikant | Instruction Schedul

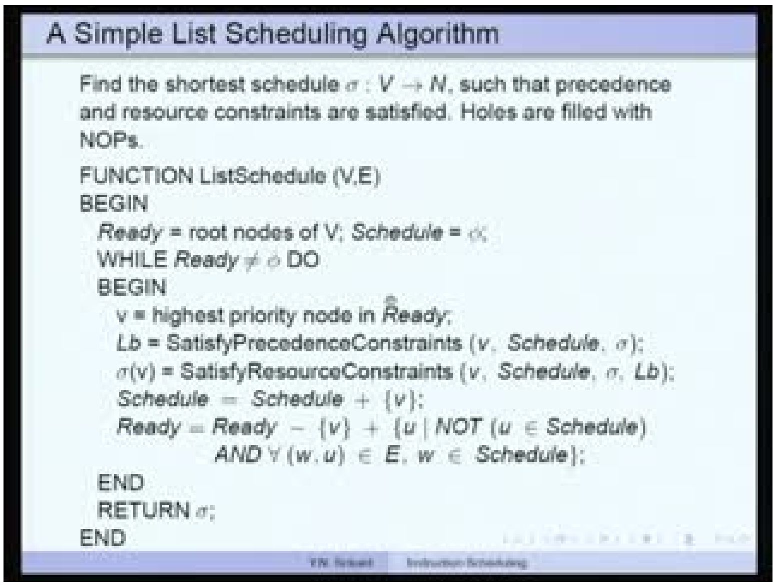
Quickly, a simple example to show the precedence and resource constraints, if u and v are related by an edge and the delay on that edge is d. Let us say, sigma u is the scheduled time of u and sigma v is the scheduled time of v. Then, the delay must be less than or equal to sigma v minus sigma u. So, in other words you cannot schedule sigma v d slots after sigma u you know earlier than d slots after sigma u.

Whereas, the resource constraints. Let us assume that v 1, v 2, v 3, v 4 are scheduled in 0, 1, 2, 3 time slots these are the sub steps of each MOS and the matrix contains numbers, which indicate the resource requirements of various sub steps.

So assume that we have 5 resources. At i equal to 4; that is, we start here 3; so this is 3, this is 4. So, look up the diagonal and we have 2 plus 2 plus 1 as the resource requirements of all the instructions which are still active. See this, sigma the v 3 is active; and v 2 in its second sub step and v 1 is in its third sub step and v 1 has finished.

So, we have to add up all these and make sure that it is less than or equal to 5, which is indeed true. The resource constraints are satisfied but if you look at i equal to 2; then, we sum up the resource requirements of all the three active instructions; v 3, v 2 and v 1 are all active. So 3 plus; 3 plus; 2; that is, 8, which is greater than 5. So the resource constraint is not satisfied that means this particular schedule the way we have put it here is not a legal schedule.

(Refer Slide Time: 04:10)



A Simple List Scheduling Algorithm

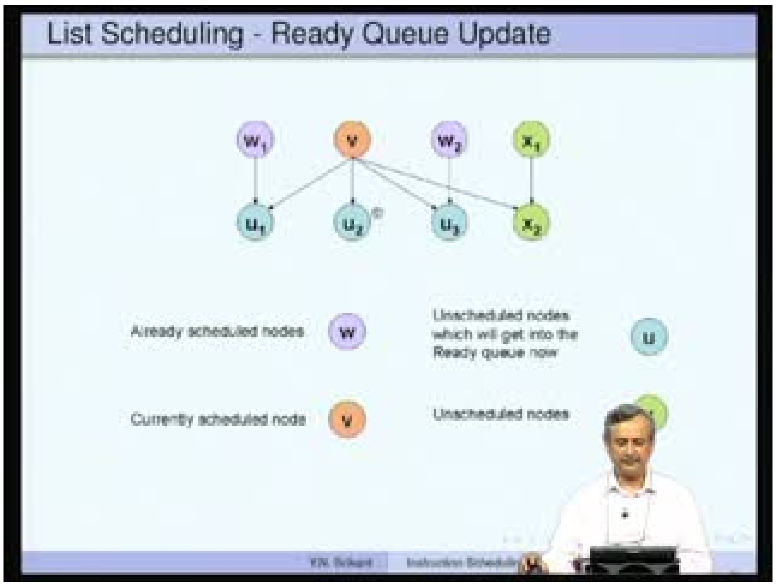
Find the shortest schedule $\sigma : V \to N$, such that precedence and resource constraints are satisfied. Holes are filled with NOPs.

```
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 | NOT (u $\in$ Schedule)
              AND $\forall$ (w, u) $\in$ E, w $\in$ Schedule};
  END
  RETURN $\sigma$;
END
```

Y.N. Srikant Instruction Scheduling

The simple list scheduling algorithm looks at a ready queue of nodes, which are ready to be scheduled. To begin with we have the root nodes of v and then, we pick the highest priority node in the ready. Then, check whether, it satisfies precedence constraints and find the lowest slot in which we can schedule it. We make sure that resource constraints are also satisfied and find the slot after l b, during at which v can be scheduled. Then, we update the ready queue and add this v 2 to the schedule.

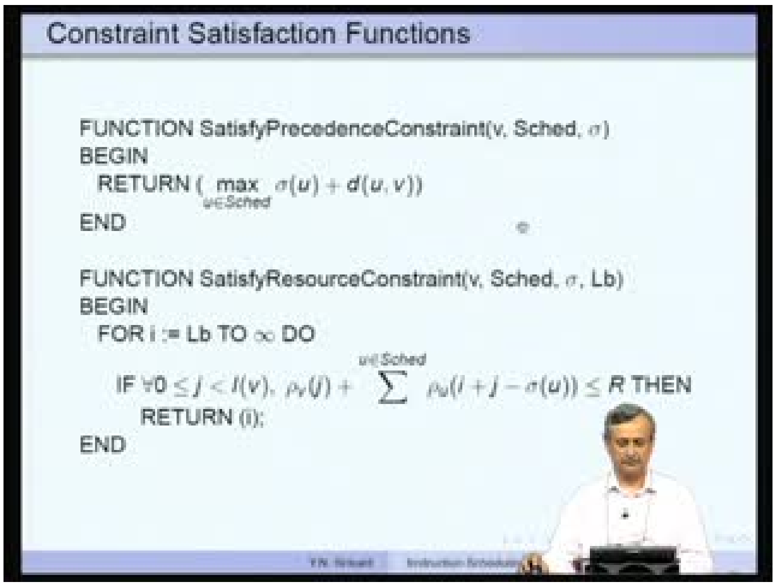
(Refer Slide Time: 04:45)



List Scheduling - Ready Queue Update

So this is the ready queue update. So v is the node which has been just now scheduled. So, it has many successors. Three of the successors have predecessors which are already scheduled; all the predecessors are scheduled. We can add the u 1; u 2; u 3 to the ready queue but x 2 has another predecessor x 1, which is not at scheduled so x 2 cannot be added to the ready queue.

(Refer Slide Time: 05:11)



Constraint Satisfaction Functions

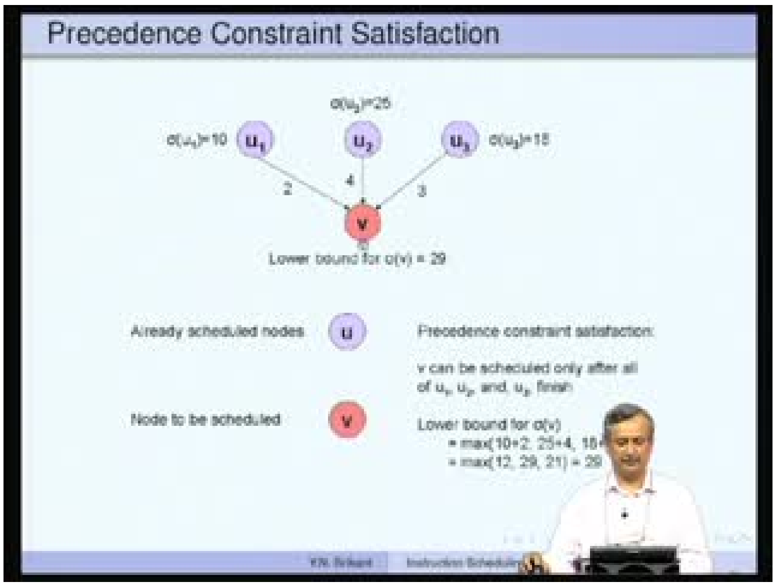
```
FUNCTION SatisfyPrecedenceConstraint(v, Sched, $\sigma$)
BEGIN
  RETURN ( $\max_{u \in Sched} \sigma(u) + d(u, v)$)
END

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

Y.N. Srikant | Instruction Scheduling

The precedence constraints and resource constraints are quiet straight forward. I already mentioned what they are so let us look at a picture.

(Refer Slide Time: 05:49)



Precedence Constraint Satisfaction

Precedence constraint satisfaction:

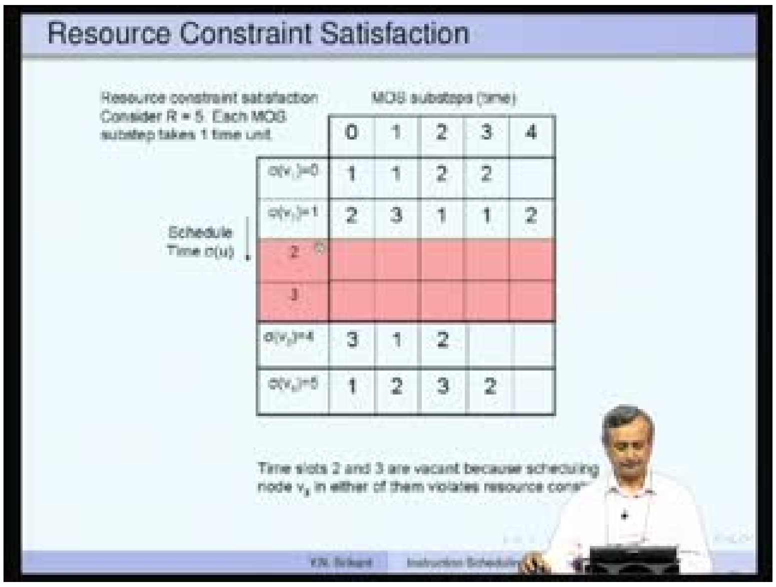
v can be scheduled only after all of u_1, u_2, and, u_3 finish

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

Y.N. Srikant | Instruction Scheduling

So v is the node and sigma u 1; sigma u 2; sigma u 3 are mentioned here as 10, 25 and 18. So, now what is the lower bound on sigma v, that is the slot in which can be scheduled; it will be the maximum of 10 plus 2; 25 plus 2 and 18 plus 3. So, that comes to 29. This is how the precedence constraint is satisfied and the lowest lower bound for sigma v is found.

(Refer Slide Time: 05:55)



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					
$\sigma(v_4)=4$	3	1	2		
$\sigma(v_5)=5$	1	2	3	2	

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

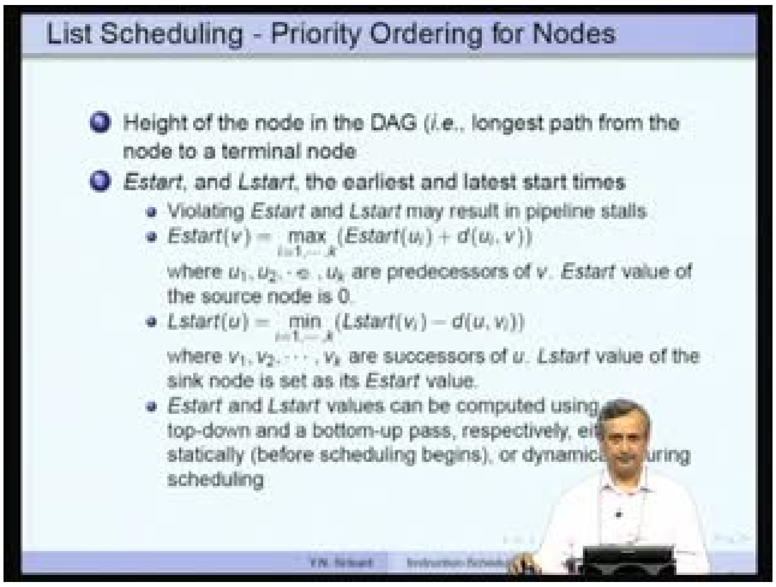
Y.N. Srikant | Instruction Schedulin

The resource constraint satisfaction at any slot that we want to schedule, we must make sure that the resources are available. So, as I showed you in the few minutes ago, scheduling v 3 here will cause a problem; there will be resource crunch so the same is true for this also.

For example, if we schedule sigma, we put sigma v 3 equal to 3. So, we would have 3 plus 1 plus 2, that is 6. So again we are violating the constraint.

So whereas, here it is 3 plus 1 4 and 1 plus 2 3 and this is 2. So no problem, thus a resource requirement will be satisfied., that is the reason why 2 and 3 have been left free.

(Refer Slide Time: 06:47)

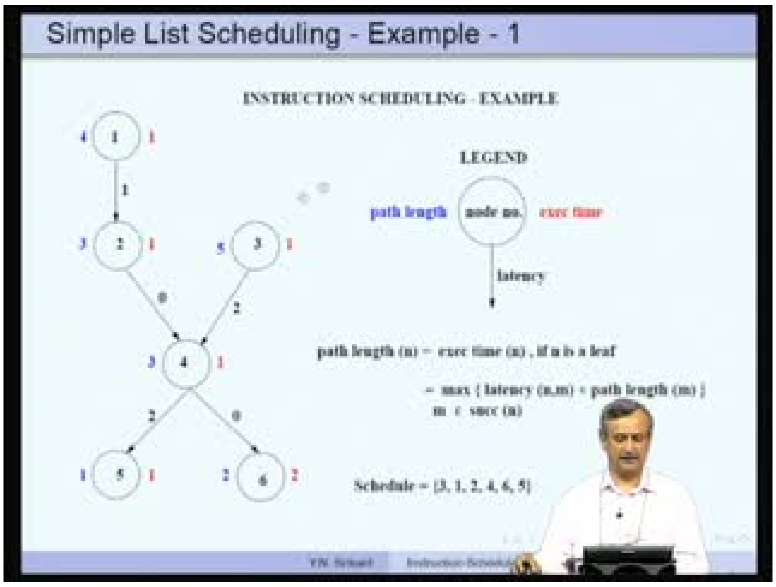


List Scheduling - Priority Ordering for Nodes

- 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
  - $Estart(v) = \max_{i=1,\cdots,k} (Estart(u_i) + d(u_i, v))$
    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

(Refer Slide Time: 07:04)



Simple List Scheduling - Example - 1

INSTRUCTION SCHEDULING - EXAMPLE

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

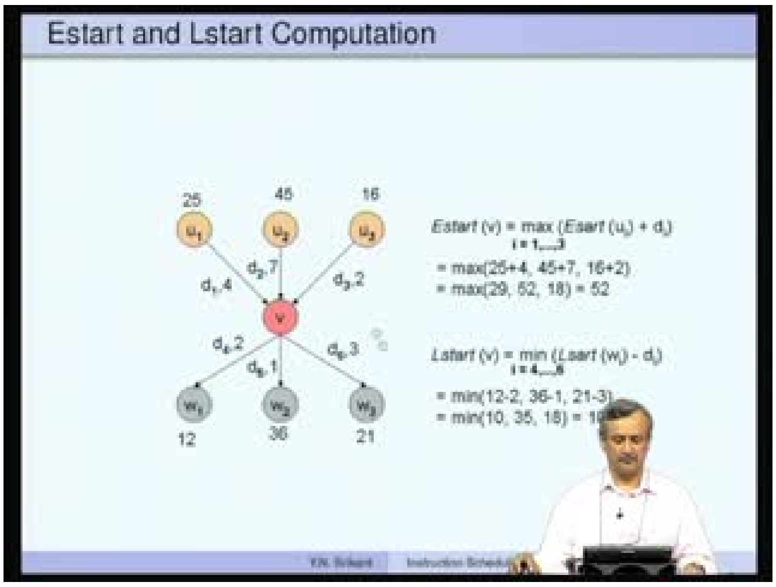
= max { latency (n,m) + path length (m) }
m ∈ succ (n)

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

Y.N. Srikant Instruction Scheduling

Now how do we pick the nodes from the ready queue? So, there is a priority ordering for nodes: One of the priorities that we use is the longest path from the node to a terminal node. So, that is something I can show here. So, you compute the path lengths starting from the bottom and travel to the top by adding the delays along the paths. You also have Estart and Lstart, the early start and latEstart deadlines for the nodes. So, let me show you, a very simple picture of how it can be computed.

(Refer Slide Time: 07:29)



Estart and Lstart Computation

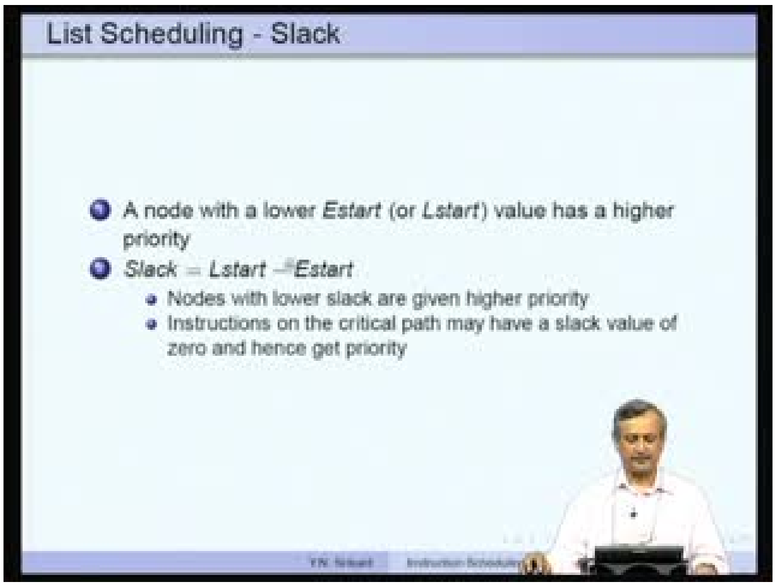
$$Estart\ (v) = \max_{i=1,\dots,3} (Esart\ (u_i) + d_i)$$
$$= \max(25+4,\ 45+7,\ 16+2)$$
$$= \max(29,\ 52,\ 18) = 52$$

$$Lstart\ (v) = \min_{i=4,\dots,6} (Lsart\ (w_i) - d_i)$$
$$= \min(12-2,\ 36-1,\ 21-3)$$
$$= \min(10,\ 35,\ 18) = 1$$

u 1, u 2, u 3 are the three nodes which are already scheduled the predecessors rather; the v is the node for which we want to compute the Estart u 1, u 2, u 3 have their Estart already computed not scheduled. So, 25, 45 and 16 are the Estart times for the u 1, u 2, u 3; the delays are 4, 7 and 2 on these arcs. So, again just like resource on the precedence constraint problem, we compute 25 plus 4, 45 plus 7 and 16 plus 2, find that 52 is the max of all this and that would be the earliest start time for this particular node v. So, that makes sense it is just like satisfying the precedence constraints.

For the latest start we consider from the bottom, the nodes w 1, w 2, w 3 which are successors of v 12, 36 and 21; subtract the delays and see which of these is minimum and that is the time the latest time at which we can start v, for example, this is minimum is 10 that corresponds to w 1 which is 12 minus 2 10; that means, if we schedule v at 10 then after two slots w 1 can be scheduled. If we schedule v earlier than that, then there is a precedence violation, and therefore we require a pipeline stall. If we schedule v later than that w 1 will have to wait for a slot because w 1 will be ready at 12, if we schedule this say at 12, then w 1 will have to wait for the 2 cycles that is still 14 in order to get schedule.

(Refer Slide Time: 09:26)



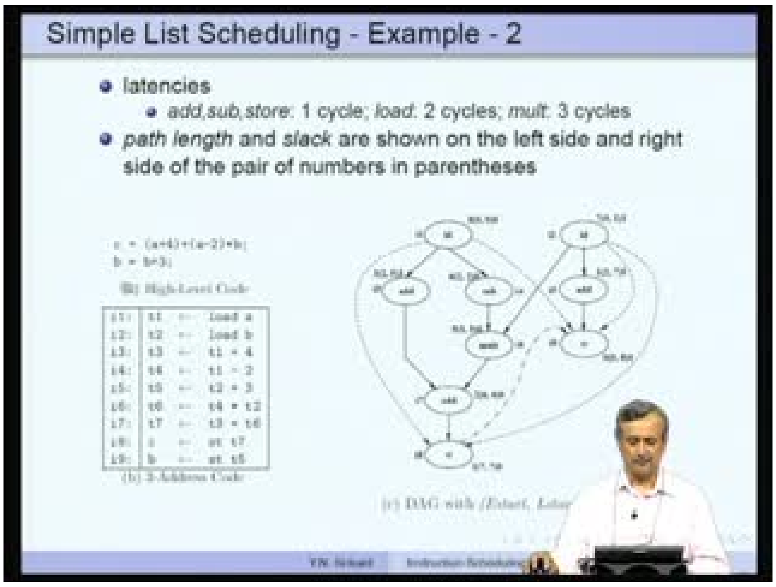
List Scheduling - Slack

- A node with a lower *Estart* (or *Lstart*) value has a higher priority
- *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

YN Srikant | Instruction Scheduling

So, either too early or too late will harm our heuristic will always make sure that the slack which is Lstart minus Estart is used as when it is used as the priority. We make sure that the lower slack nodes are given higher priority and it so happens on critical path possibly the slack would be 0 and they get the maximum priority. So, let us see how these Estart and Lstart values are computed for a nontribal example.

(Refer Slide Time: 09:50)



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, Lstar

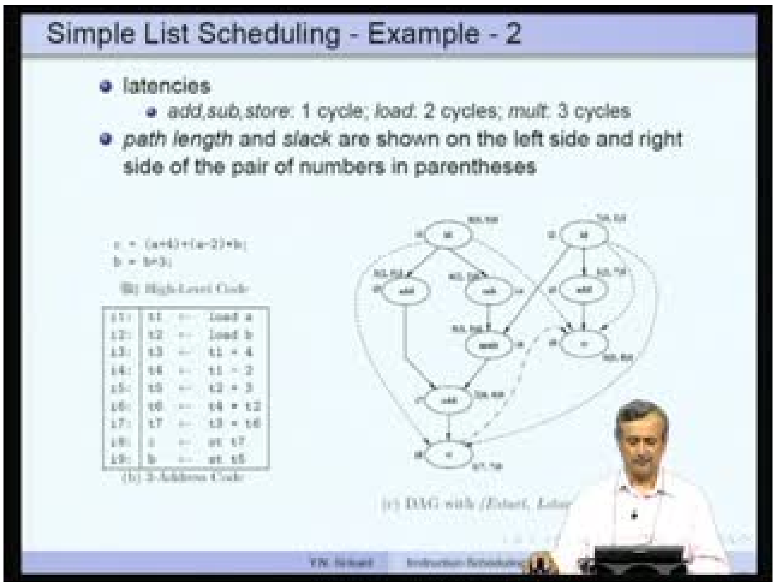
YN Srikant | Instruction Scheduling

We assume that, add sub and stores take one cycle each, load takes two cycles and multiply takes three cycles. So, path length is written on the left hand side of the parentheses and the slack is written on the right hand side of the parentheses. So, what is inside parentheses or the Estart and Lstart numbers.

So, if you look at it closely, we start the Estart computation from the top. So, the Estart number is 0 for i 1 and then as I told you in the last lecture, we compute the Estart values for i 3 and i 4 by adding the delay of load. So, 0 plus 2 becomes 2 or n 2 on the side as well. Here also because this is a root node we start the Estart value with 0. And from i 3 along the arc, from i 3 to i 7, we have a delay of 1; so, the value of Estart along this arc could be 2 plus 1, 3.

Whereas, if we consider this path the mult instruction will get 2 plus 1 3 and the add instruction will get 3 plus 3 6 because mult has a delay of 6 3. So, 6 is the max value and therefore, this gets 6. Similarly, Estart value of all the nodes are computed and when we reach here, this is 7. There is an output dependence from i 8 to i 9.So, this is forced to become 8 which is 7 plus 1.

(Refer Slide Time: 09:50)



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, Lstar

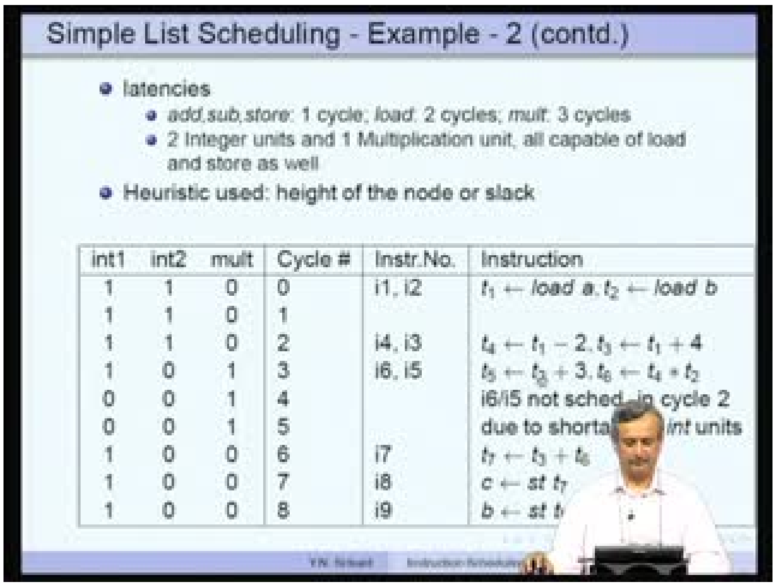
Y.N. Srikant | Instruction Scheduling

We start the Lstart values from here, it is assigned Lstart equal to Estart is the value assigned. So, we get a value of 8 for i 9. Now the predecessors are considered this has many predecessors. So, add has exactly one successor, when we consider add we get 8 minus 1 which is the delay of this as 7 for Lstart.

Whereas for this node this is the only successor so, again this gets 8 minus 1 as 7 and then adds gets 7 minus 1 as 6. Then we go on you know this add gets 6 minus 1 of 5 and this multiply gets 6 minus 3 as 3; this gets 3 minus 1 as 2 and from here this is 5 minus 2 that is 3; and this is 2 minus 2 that is 0 minimum is 0. So, this gets its Lstart value as 0 and from here we have two successors one is here and another is here. We need to consider these two and you know this has 3. So, 3 minus 2 is 1 that is the minimum value. Of course, when you look at this arc also, you get 8 minus 2 as 6, but 6 is much larger than 0.

This is the way the Lstart values are computed from the bottom towards the top. So, here is the program and we want to schedule this particular program by reordering the instructions. So, we have so far computed the path length and Estart Lstart values. So, Lstart minus Estart gives you this slack and they are all noted here. So, let us try scheduling these instructions:

(Refer Slide Time: 13:35)



Simple List Scheduling - Example - 2 (contd.)

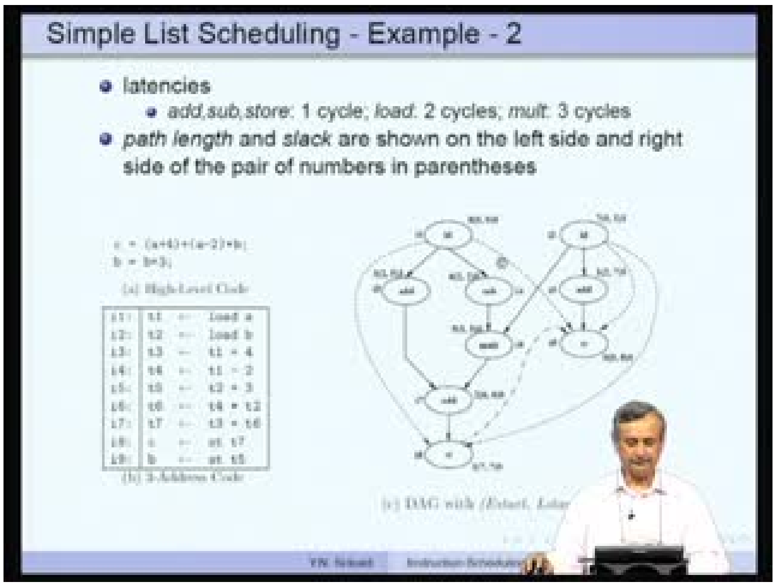
- latencies
  - *add,sub,store*: 1 cycle; *load*: 2 cycles; *mult*: 3 cycles
  - 2 Integer units and 1 Multiplication unit, all capable of load and store as well
- Heuristic used: height of the node or slack

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	$t_5 \leftarrow t_3 + 3, t_6 \leftarrow t_4 * t_2$
0	0	1	4		i6/i5 not sched. in cycle 2
0	0	1	5		due to shorta *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	$b \leftarrow st\ t$

As we said add sub store require one cycle, load requires two cycles and multiply requires three cycles. Let us assume that there are two integer units and one multiplication unit and all the three units are capable of load and store operations as well.

We can use the heuristic of either the height of the node or slack and the schedule will be identical. So, let us look at slack because we have already one example with height of the node.

(Refer Slide Time: 14:17)



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
```

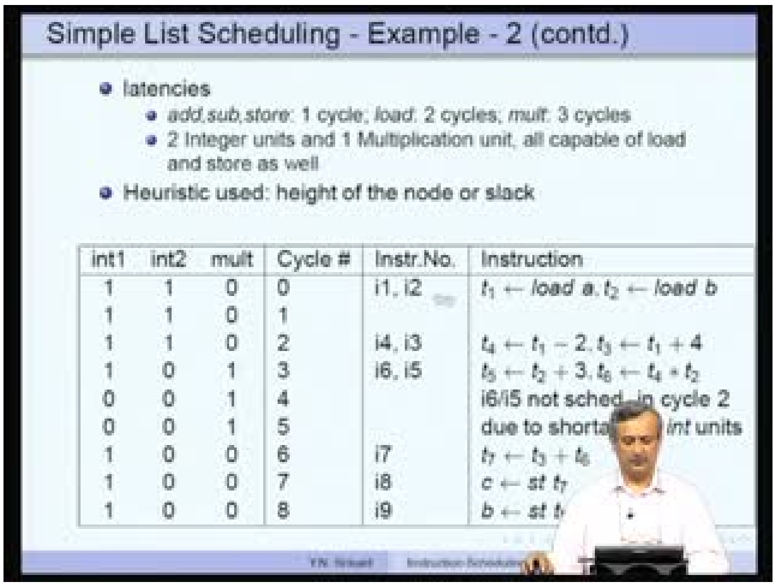
(b) 3-Address Code

(c) DAG with (Estart, Lstar

YN Srikant | Instruction Scheduling

So, to begin with these are the two nodes which are in the ready queue because both of them are the routes of are the dag.

(Refer Slide Time: 14:29)



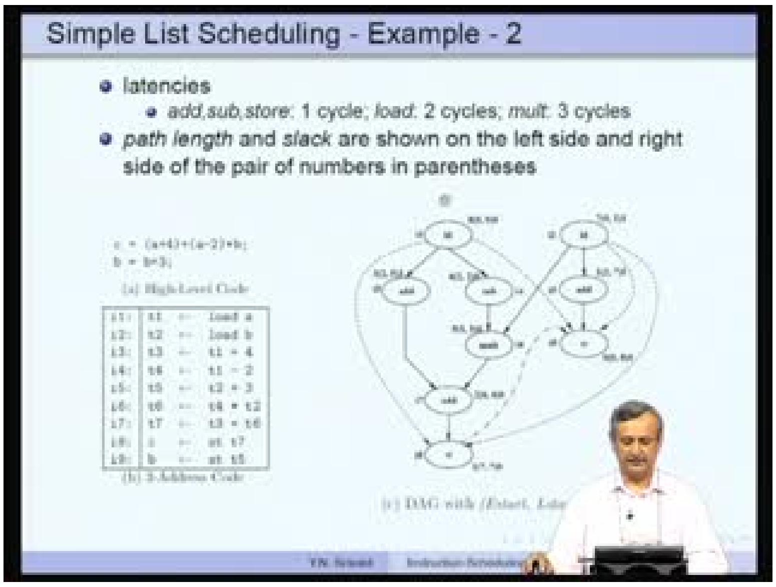
Simple List Scheduling - Example - 2 (contd.)

- 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		i6/i5 not sched. in cycle 2
0	0	1	5		due to shorta *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	$b \leftarrow st\ t$

YN Srikant | Instruction Scheduling

(Refer Slide Time: 14:44)



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

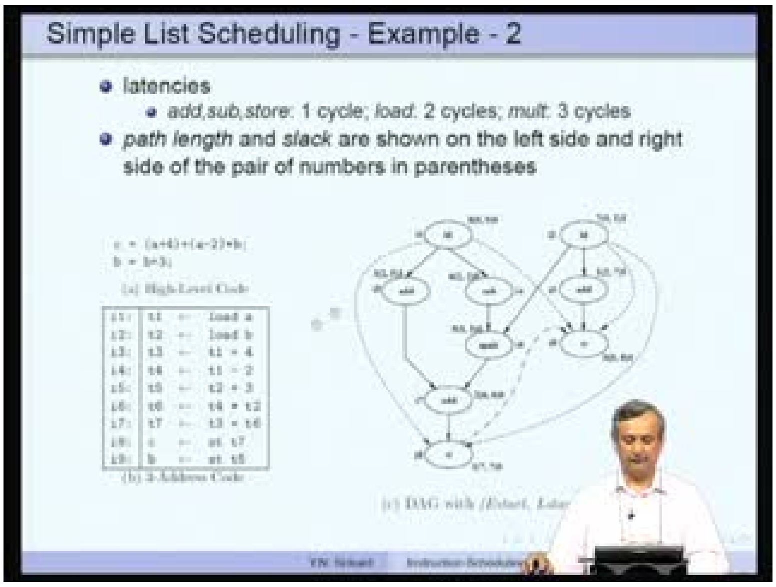
(b) 3-Address Code

(c) DAG with (Estart, Lsta

Y.N. Srikant Instruction Scheduling

So, in cycle number 0, all the resources become just before the cycle number 0, beginning of it all the resources are available. There are two instructions which need to be scheduled both are load operations so, load and load. We can schedule both of them on the two integer units which are available and the multiply unit is not used. So, here a 1 indicates that the integer unit being used and 0 indicates that unit is not being used. So, 1 1 0 indicates integer unit 1 and 2 are be used, but the multiply unit is not being used. So, two unit two instructions are scheduled both are load. Now we cannot schedule any instruction in this particular cycle number 1, for the simple reason that load requires two cycles to compute and complete. Before the load is complete, neither of these two instructions nor this particular instructions can be enabled. So, we just keep cycle number 1 vacant and during this period also int 1 and int 2 are busy mult 0 because load requires 2 cycles to complete

(Refer Slide Time: 15:48)



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

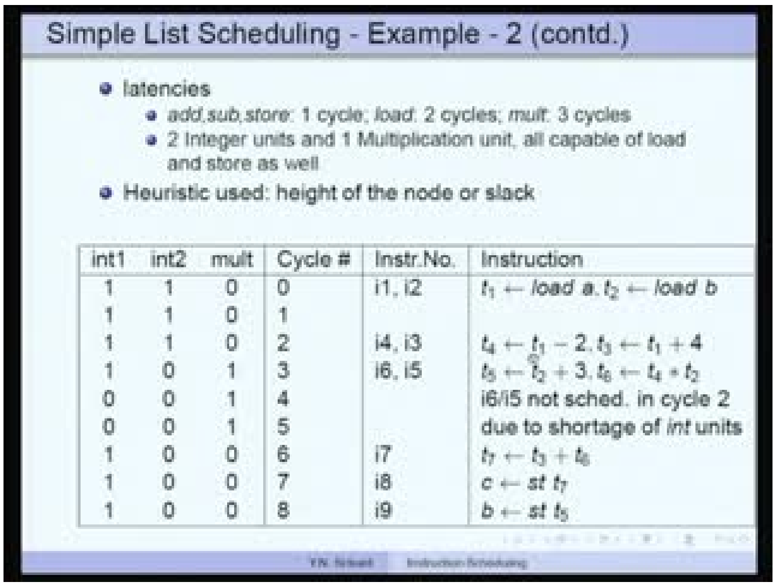
i1:	t1	←	load a
i2:	t2	←	load b
i3:	t3	←	t1 + 4
i4:	t4	←	t1 - 2
i5:	t5	←	t2 + 3
i6:	t6	←	t4 * t2
i7:	t7	←	t3 + t6
i8:	c	←	st t7
i9:	b	←	st t5

(b) 3-Address Code

(c) DAG with (Estart, Lstar

YN Srikant Instruction Scheduling

(Refer Slide Time: 16:08)



Simple List Scheduling - Example - 2 (contd.)

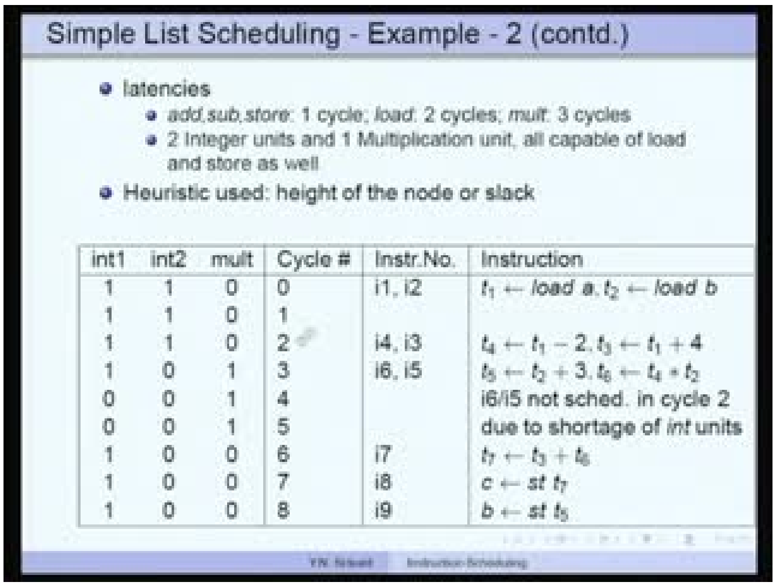
- latencies
  - *add,sub,store*: 1 cycle; *load*: 2 cycles; *mult*: 3 cycles
  - 2 Integer units and 1 Multiplication unit, all capable of load and store as well
- Heuristic used: height of the node or slack

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		i6/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

YN Srikant Instruction Scheduling

In cycle number 2, now we have all 3 of them you know enabled right so, this i 3 then i 4 and i 5. All the three instructions are now ready to be executed, the load has completed, but obviously we cannot schedule all three of them. There is a shortage of the units so, even though there are three instructions, we cannot really schedule all three of them. So, we need to pick one or more of these instructions then scheduled them. How to pick? That choice is done made actually based on the slack. The slacks are 3, 0 and 5 so it is obvious that the highest slack corresponds to instruction i 5. Then we need to actually look at the lowest slack not the highest slack. The lowest slack instruction corresponds to this i 4 so, i 4 is scheduled first. We still have another integer unit available, therefore the next lowest slack corresponds to i 3. So, i 3 can be scheduled but i 5 cannot be scheduled because there is no other integer unit available to us.

(Refer Slide Time: 17:02)



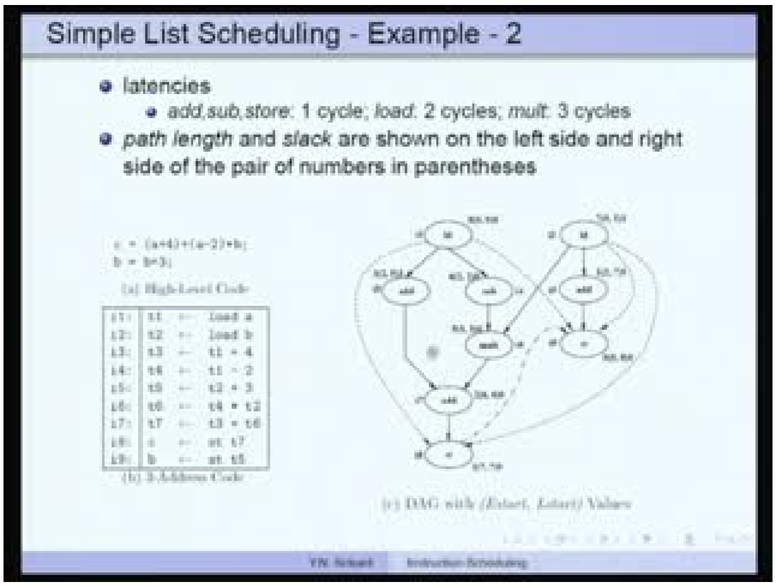
Simple List Scheduling - Example - 2 (contd.)

- 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		i6/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

(Refer Slide Time: 17:14)



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
```

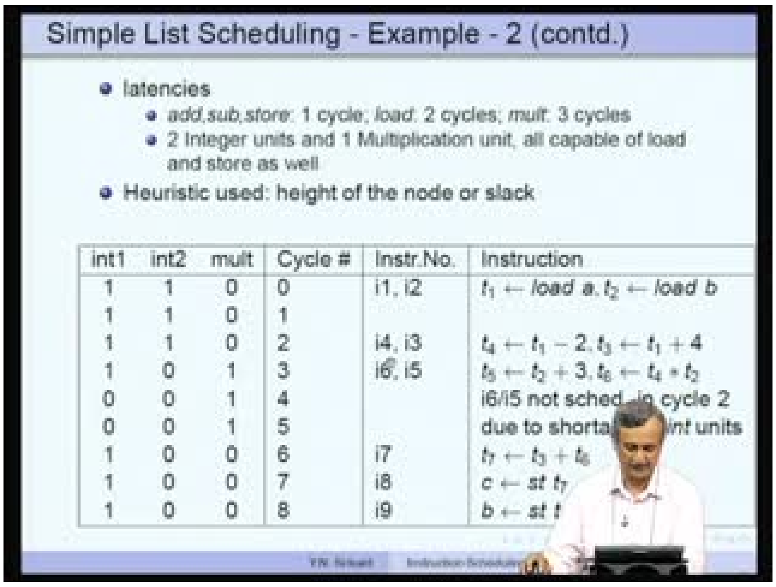
(b) 3-Address Code

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

Y.N. Srikant Instruction Scheduling

So, we keep i 5 in the ready queue. Now once i 4 completes i 6 is enabled and it can be added to the ready queue. But remember we cannot add i 7 to the ready queue because even though i 5 has completed i 6 has not yet completed. So, in the ready queue we now have i 6 and i 5 one of them requires mult and the other one requires add. So, obviously int and mult are available at this point both are free. We take that services and schedule both instructions in the mult and add units.

(Refer Slide Time: 17:38)



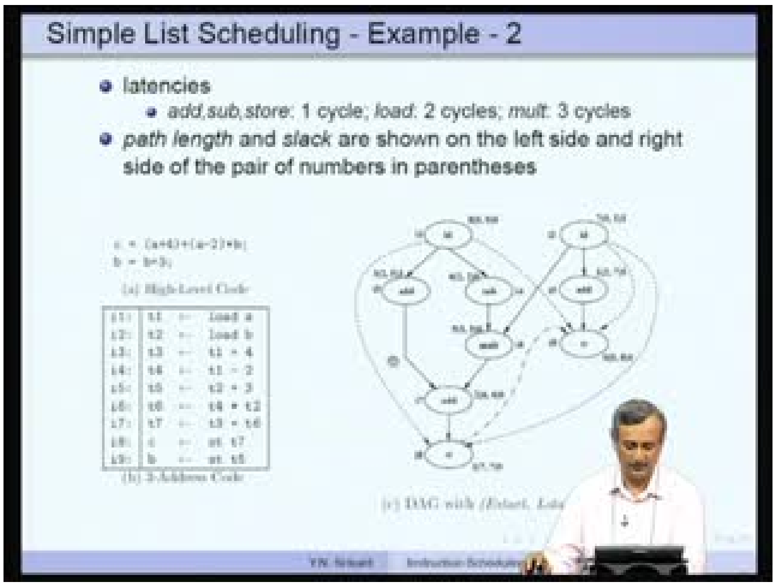
Simple List Scheduling - Example - 2 (contd.)

- 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		i6/i5 not sched [illegible] cycle 2
0	0	1	5		due to shorta[illegible] *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	$b \leftarrow st\ t$[illegible]

Y.N. Srikant Instruction Scheduling

(Refer Slide Time: 18:12)



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

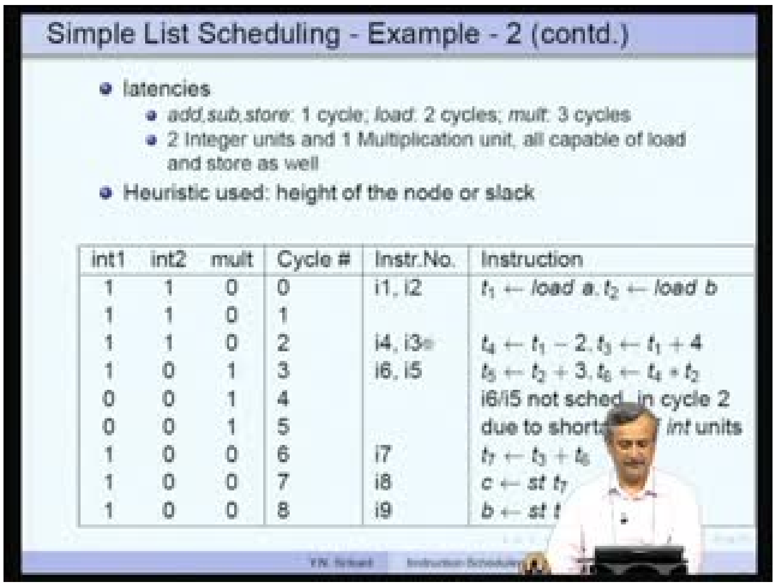
(b) 3-Address Code

(c) DAG with (Estart, Lsta

Y.N. Srikant Instruction Scheduling

Now the next step, we obviously cannot schedule in i 6 and i 5 etcetera in cycle number 2 because of shortage of units i 5 is not scheduled in; there is no int available and i 6 cannot be scheduled, until we have completed an instruction. So, until i 4 completes we cannot schedule i 6. So, that is the major difficulty here and we are using slack so, we cannot go to i 6 even though it is a multiply unit and it is free we cannot schedule i 6 a head of another instruction i 5 because its slack is you know it is 0 and that is 5. We need to take care of these slacks also. We cannot violate the rules of the slack. So, what happens is we end up scheduling i 6 first because it has lowest slack and then i 5 in this order. So, 2 cannot take anything because there are no units available.

(Refer Slide Time: 18:45)



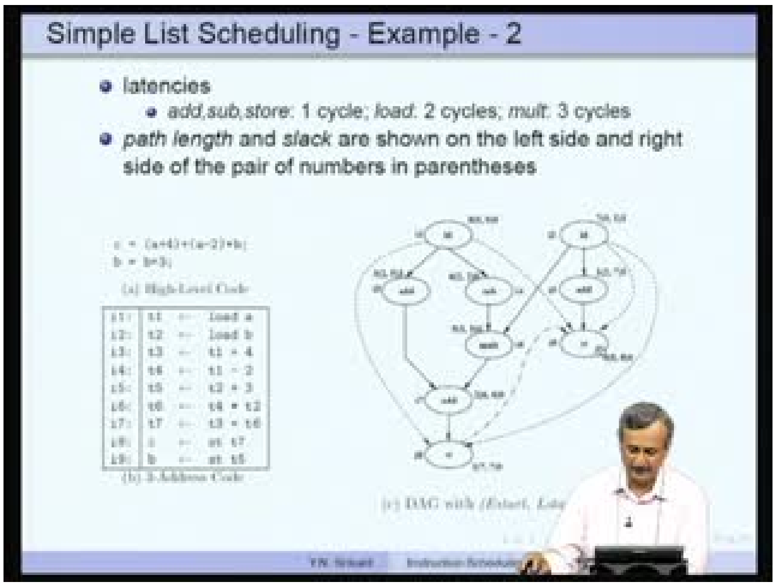
Simple List Scheduling - Example - 2 (contd.)

- 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		i6/i5 not sched. in cycle 2
0	0	1	5		due to short [illegible] *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	$b \leftarrow st\ t$ [illegible]

Y.N. Srikant Instruction Schedu

(Refer Slide Time: 19:11)



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

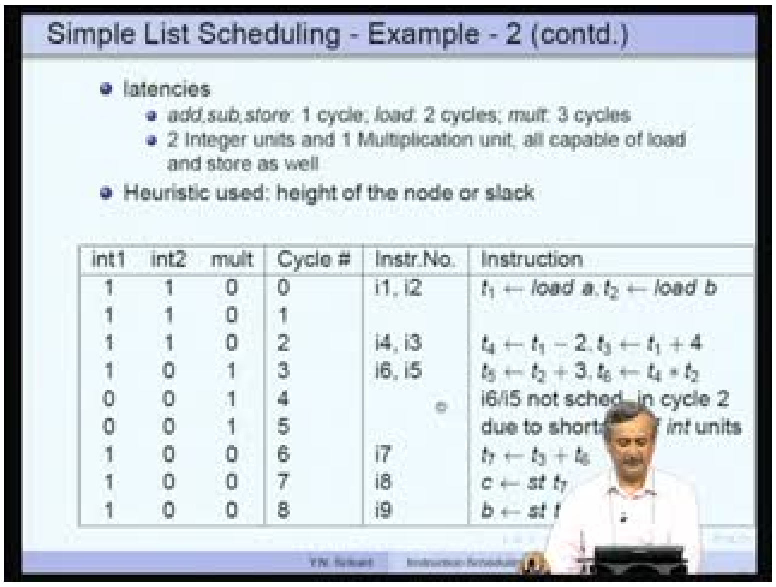
(b) 3-Address Code

(c) DAG with (Estart, Lstart

Y.N. Srikant Instruction Schedu

Now the next one to be scheduled will only be instruction i 7. We are going to wait for a long time because multiply has not completed so, please look at the picture again. This instruction even though i 5 has completed and this is the next instruction; this cannot be executed until i 8 is completed. So, that is the major problem, we cannot execute this. So, we have complete i 6 that means, we will require three cycles. So, we wait and then schedule i 7.

(Refer Slide Time: 19:32)



Simple List Scheduling - Example - 2 (contd.)

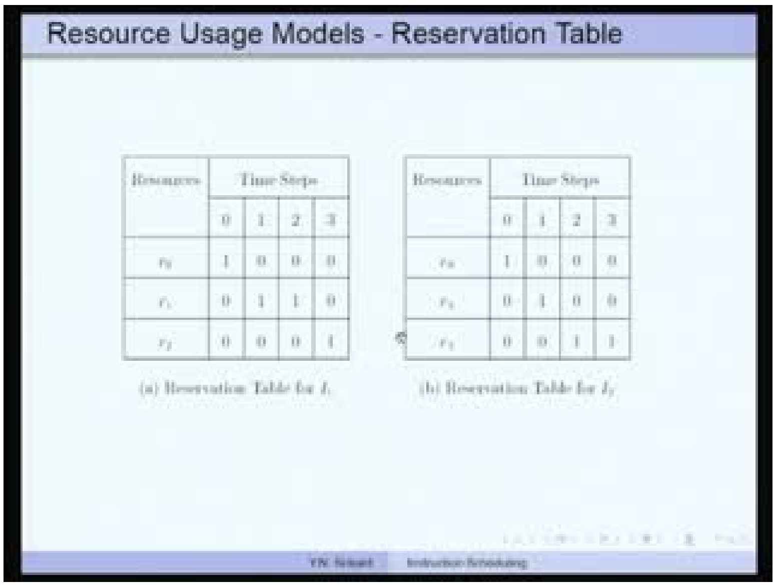
- latencies
  - *add,sub,store*: 1 cycle; *load*: 2 cycles; *mult*: 3 cycles
  - 2 Integer units and 1 Multiplication unit, all capable of load and store as well
- Heuristic used: height of the node or slack

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		i6/i5 not sched. in cycle 2
0	0	1	5		due to short [illegible] *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	$b \leftarrow st$ [illegible]

YN Srikant   Instruction Scheduling

So i 7 is completed then only one integer unit i 8 is in the next cycle and i 9 in the last cycle. So, this how we actually schedule the instructions according to their slack and that is about it. So this is the example for scheduling with slack.

(Refer Slide Time: 19:54)



Resource Usage Models - Reservation Table

Resources	Time Steps			
	0	1	2	3
r_0	1	0	0	0
r_1	0	1	1	0
r_2	0	0	0	1

(a) Reservation Table for I_1

Resources	Time Steps			
	0	1	2	3
r_0	1	0	0	0
r_1	0	1	0	0
r_2	0	0	1	1

(b) Reservation Table for I_2

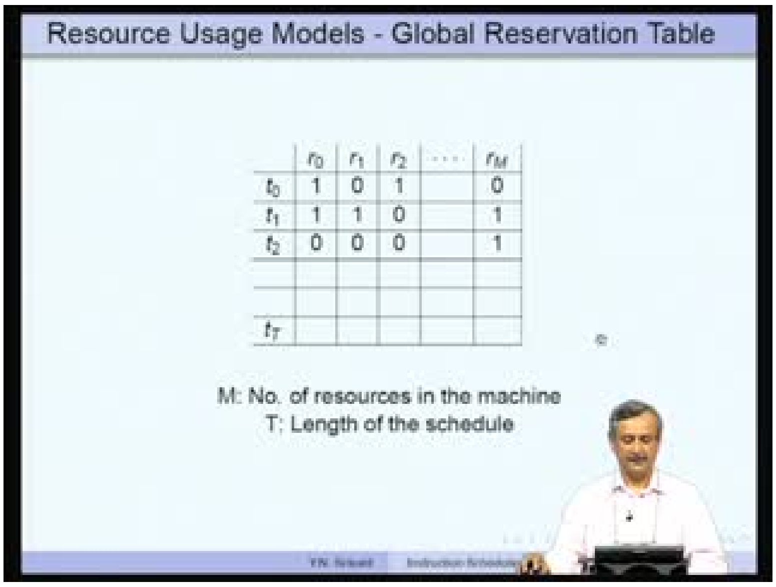
YN Srikant   Instruction Scheduling

Then we still have not formally considered how to represent the resource availability. I showed you, an example here this is the resource reservation table that we are going to talk about very soon. It shows all the resources and their availability in the various cycles.

So, we are going to have a resource reservation table for each instruction. So, for i 1 a class of instructions rather each class of instructions will have a resources reservation table.

This shows for the various time steps needed to execute any instruction in that particular class. What are the various resources used in the various cycles? So, for the two instructions i 1 and i 2, we have shown the two reservation tables. So, for example, r 0 is used in cycle 0, but not anymore; r 1 is used in cycles 1 and 2; r 3 is used in cycle 3. Similarly, r 1, r 2, r 3 in the is used in the cycle 1 cycle 2 and cycle 3, 2 and 3.

(Refer Slide Time: 21:09)



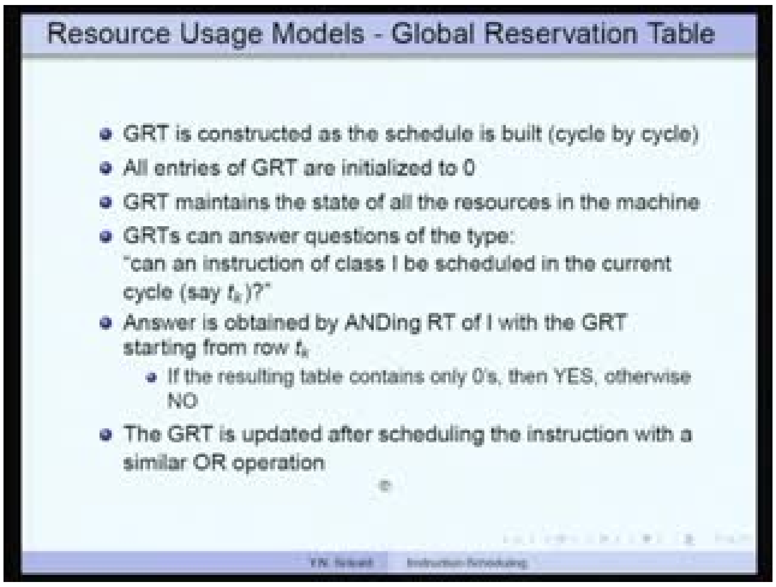
Resource Usage Models - Global Reservation Table

	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

So, from this we consider the construct the global reservation table, which I have already shown you. There are times slots here and resources here and it shows availability of various resources and in the various timeslots.

(Refer Slide Time: 21:22)



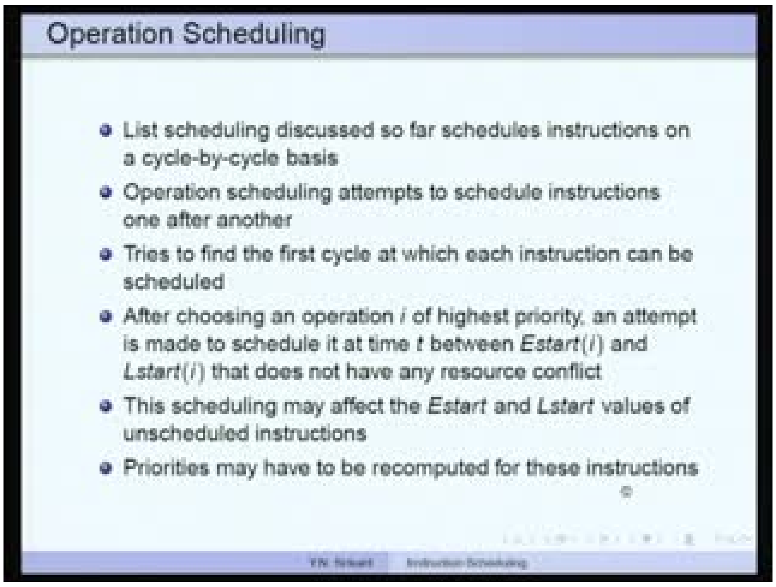
Resource Usage Models - Global Reservation Table

- 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, the global reservation table is constructed as the schedule is built, which I have shown you just now. All the entries of GRT are initialized to 0 and 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. So, this answer is obtained very simply by ANDing the reservation table of i with the GRT starting from row t k. So, if the resulting table contains only 0s, then obviously resources available but if it contains once along that particular column for that resource, then it is not available. Then the GRT is updated after scheduling the instruction with a similar OR operation. So, the AND operation is done in order to make sure that, you know, availability is checked and the OR operation is actually going to be the one which updates the ric GRT.

(Refer Slide Time: 22:27)



Operation Scheduling

- List scheduling discussed so far schedules instructions on a cycle-by-cycle basis
- Operation scheduling attempts to schedule instructions one after another
- Tries to find the first cycle at which each instruction can be scheduled
- After choosing an operation *i* of highest priority, an attempt is made to schedule it at time *t* between *Estart*(*i*) and *Lstart*(*i*) that does not have any resource conflict
- This scheduling may affect the *Estart* and *Lstart* values of unscheduled instructions
- Priorities may have to be recomputed for these instructions

Y.N. Srikant | Instruction Scheduling

Then what we have seen so far is one method of scheduling called cycle by cycle scheduling. In other words what we do is we look at the cycle and see how many instructions can be scheduled in that particular cycle and go ahead. So, it is not as if we pick an instruction and see where that particular instruction fits in the sequence of timeslots available to us. So, obviously, this second method which I just now mention is also one of the scheduling methodology and that is called operation scheduling.

So, list scheduling discussed so far schedules instructions on a cycle by cycle basis. Operation scheduling, attempts to schedule instructions one after another. Just pick the instructions which have been ordered and schedule them one after another. It tries to find the first cycle at which each instruction can be scheduled. So, it is not the cycle which is picked and then the instructions which can be scheduled there, but we pick an instruction and say in which cycle it can be scheduled.

Obviously, we need to make sure that precedence and resource constraints are satisfied for that particular instruction in that cycle.

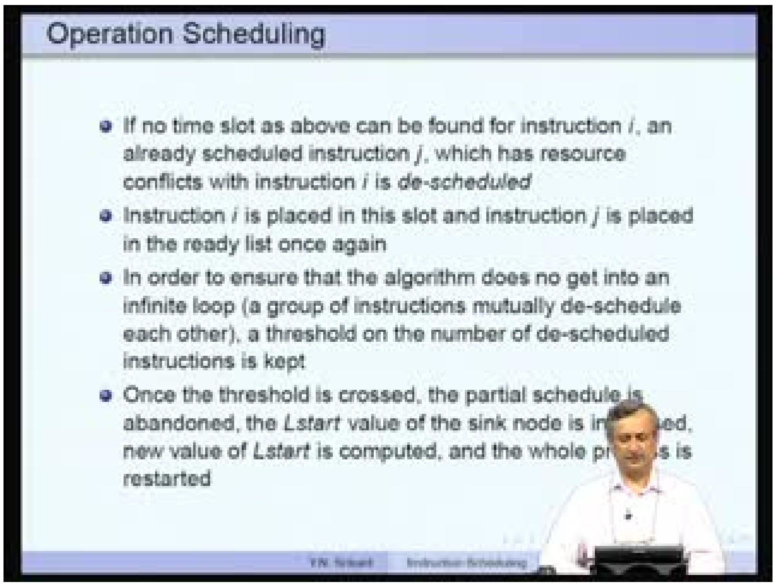
After choosing an operation i of highest priority, we make an attempt to schedule that instruction in the slack period, that is between Estart and Lstart; make sure that there are no resource conflicts in when the schedule that particular instruction in this slack period.

So, what may happen, this scheduling may not be at the Estart value right? So, we may schedule an instruction at Estart plus 1, e or Estart plus 2 etcetera, no just before Lstart at Lstart, or even after Lstart.

Once we if we have scheduled it exactly t start, then the Estart and Lstart values of the other instructions remain as they are. Whereas if you schedule it in any slot other than Estart, then the Estart and Lstart values of the successors; the other unscheduled instruction etcetera will definitely change.

For Estart the successor values will change, for Lstart the predecessors value will change. So, we may have to compute the Estart and Lstart of the unscheduled instructions again, and then compute the slack as a priority for the instructions and continue with the com, scheduling of the instruction.

(Refer Slide Time: 25:31)



Operation Scheduling

- If no time slot as above can be found for instruction *i*, an already scheduled instruction *j*, which has resource conflicts with instruction *i* is *de-scheduled*
- Instruction *i* is placed in this slot and instruction *j* is placed in the ready list once again
- In order to ensure that the algorithm does no get into an infinite loop (a group of instructions mutually de-schedule each other), a threshold on the number of de-scheduled instructions is kept
- Once the threshold is crossed, the partial schedule is abandoned, the *Lstart* value of the sink node is in[illegible]ed, new value of *Lstart* is computed, and the whole pr[illegible]s is restarted

Y.N. Srikant Instruction Scheduling

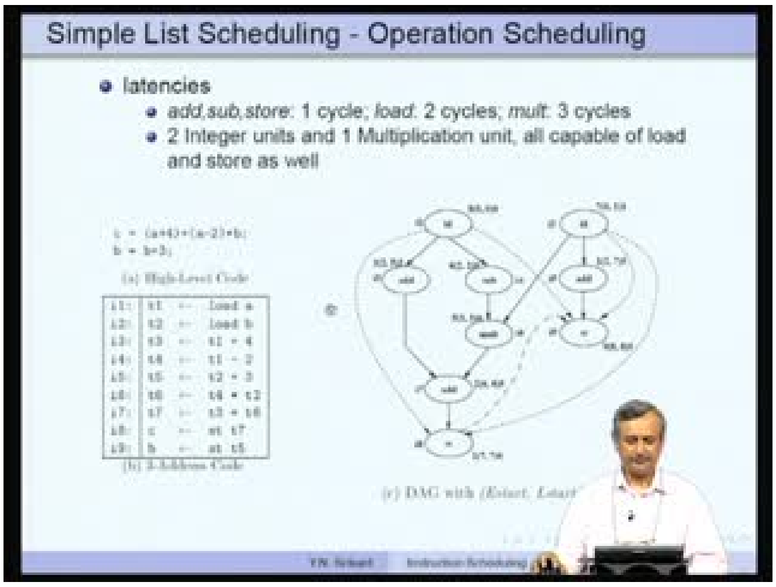
Suppose no timeslot can be found for an instruction i, then that is we do not find any slot between Estart and Lstart for several reasons. It is possible that there are resource conflicts during these slots are it is also possible that there are other instructions sitting in the timeslots, and therefore, the resources are not available.

If this happens, an already scheduled instruction may have to be thrown out. So, that is, we call this as de-scheduling of instructions. Which instruction? The one which is using the resources that we want for our instruction is de-scheduled which is removed and the instruction i is placed in this slot. The instruction j is again put back into the ready list and ordered according to its slack priority.

Now this may lead to a problem, I may actually place an instruction and then, when we pick another instruction my instruction may go out. So, this and then next time I kick the instruction which kick me. So, we want to make sure that this does not get into a loop, right. So, to make sure that the algorithm does not get into an infinite loop, a group of instructions mutually de-scheduling each other. We place a threshold on the number of de-scheduled instructions. So, let us say, we do not worry about the particular instructions but just the number of instructions which have been de-scheduled.

If this crosses a threshold value, then you know the Lstart value of this sink node is increased by 1. Then we throw the partial schedule out; increase the Lstart value by 1; recompute the Lstart values and again restart the process.

(Refer Slide Time: 27:41)



Simple List Scheduling - Operation Scheduling

- 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

```
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
```

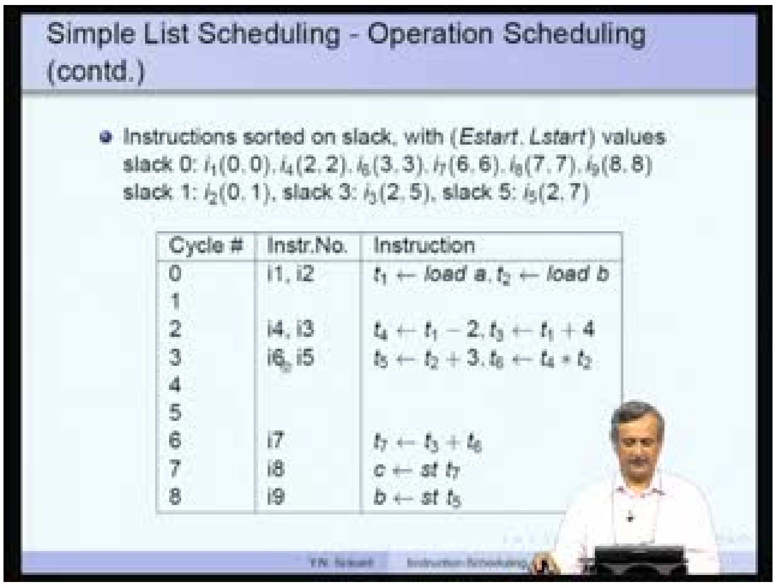
(b) 3-Address Code

(c) DAG with (Estart, Lstart)

Y.N. Srikant Instruction Scheduling

So, let us consider operation scheduling on the same example; this is the example that we have: With 2 integer units 1 the multiplication unit all capable of load and store as well. So, here again we do not have a ready queue in the way we had it before, we are just going to keep all the instructions ordered according to their priority and that is our ready queue.

(Refer Slide Time: 28:09)



Simple List Scheduling - Operation Scheduling (contd.)

- Instructions sorted on slack, with (*Estart*, *Lstart*) values
  slack 0: $i_1(0,0)$, $i_4(2,2)$, $i_6(3,3)$, $i_7(6,6)$, $i_8(7,7)$, $i_9(8,8)$
  slack 1: $i_2(0,1)$, slack 3: $i_3(2,5)$, slack 5: $i_5(2,7)$

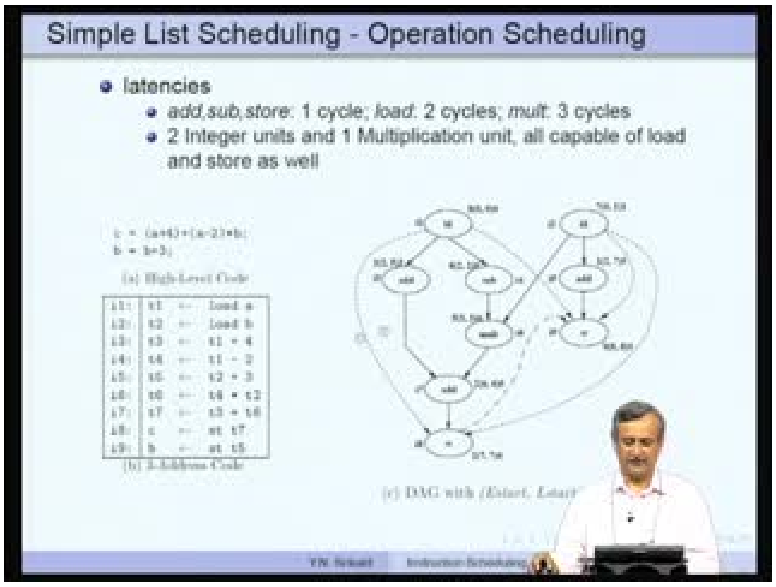
Cycle #	Instr.No.	Instruction
0	i1, i2	$t_1 \leftarrow \textit{load } a, t_2 \leftarrow \textit{load } b$
1		
2	i4, i3	$t_4 \leftarrow t_1 - 2, t_3 \leftarrow t_1 + 4$
3	i6, i5	$t_5 \leftarrow t_2 + 3, t_6 \leftarrow t_4 * t_2$
4		
5		
6	i7	$t_7 \leftarrow t_3 + t_6$
7	i8	$c \leftarrow st\ t_7$
8	i9	$b \leftarrow st\ t_5$

So, instruction sorted on slack with Estart and Lstart values. So, slack 0 we have i 1, i 4, i 6, i 7, i 8, i 9. So, these are ordered according to the Estart value 0, 2, 3, 6, 7, 8.

Slack 1 there is only one instruction i 2 with Estart and Lstart at 0 and 1, slack 3 is i 3 is 2, 5 and slack 5 is i 5 with 2 comma 7. So, this is our sequence of instruction. We are simply going to pick the instruction one by one, see whether we can place it. For example, pick i 1 it should be put in slot 0, as far as possible. All the resources are obviously available put it here.

Next I have this is the final schedule that i shown not the partial one. So, i 2 is not yet placed. So, i 1 has been placed in slot 0. Next is i 4 place it in slot 2. So how are we sure that the precedence requirements are take care of? That is taken care of by the Estart computation. So, we are not going to place any instruction pry prior to its Estart value; that is why precedence constraints are automatically satisfied. We need to check the resources constraints.

(Refer Slide Time: 29:46)



Simple List Scheduling - Operation Scheduling

- 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

```
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

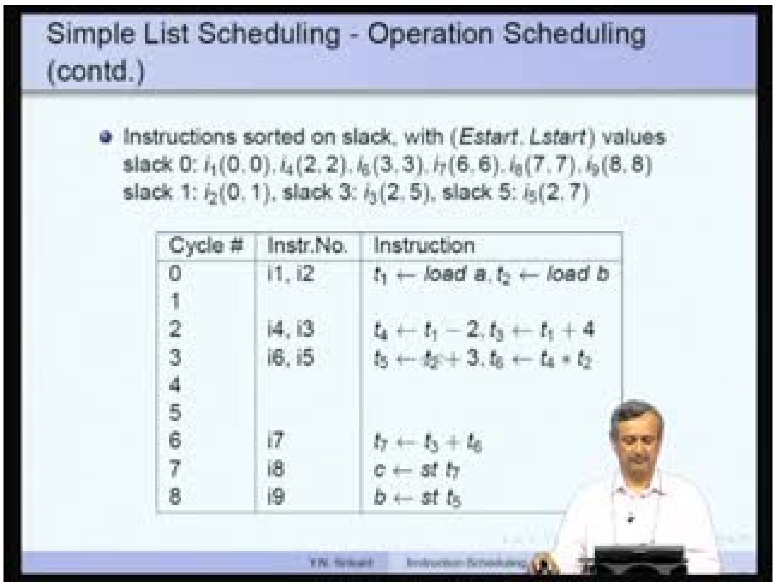
(b) 3-Address Code

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

Y.N. Srikant Instruction Scheduling

In nine you know slot number 2, i 4 can be indeed place because resources that is the into unit is available. Then comes i 6 in slot 3 which is fine no problem. So, what is instruction i 6? i 6 is a mult instruction.

(Refer Slide Time: 29:49)



Simple List Scheduling - Operation Scheduling (contd.)

- Instructions sorted on slack, with (*Estart*, *Lstart*) values
  slack 0: $i_1(0,0)$, $i_4(2,2)$, $i_6(3,3)$, $i_7(6,6)$, $i_8(7,7)$, $i_9(8,8)$
  slack 1: $i_2(0,1)$, slack 3: $i_3(2,5)$, slack 5: $i_5(2,7)$

Cycle #	Instr.No.	Instruction
0	i1, i2	$t_1 \leftarrow$ *load a*, $t_2 \leftarrow$ *load b*
1		
2	i4, i3	$t_4 \leftarrow t_1 - 2$, $t_3 \leftarrow t_1 + 4$
3	i6, i5	$t_5 \leftarrow t_2 + 3$, $t_6 \leftarrow t_4 * t_2$
4		
5		
6	i7	$t_7 \leftarrow t_3 + t_6$
7	i8	$c \leftarrow st\ t_7$
8	i9	$b \leftarrow st\ t_5$

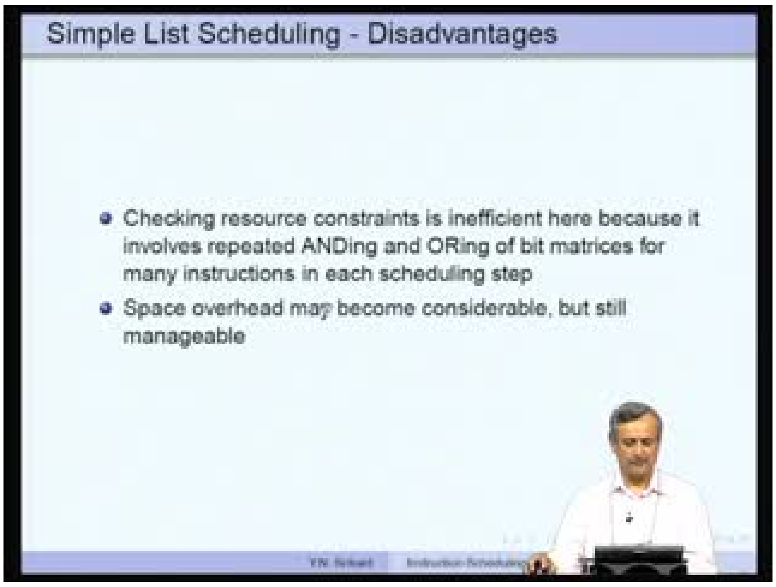
Y.N. Srikant Instruction Scheduling

Then comes i 7 in slot 6 so that is again no problem resources are available. Then i 8 in slot 7, i 9 in slot 8 and then we have i 2 in slot 0. That is possible because we have used only 1 integer unit in slot 0; the other one is still available to us. i 3 in slot 2, the same story another integer unit is still available so, we can use it. And i 5 in slot 2 so, i 6 is a mult, wherever i 5 is a an add. So, integer unit is definitely available so, we can place it in slot 3.

So far we did not face any problem because there were enough resources in these slots available, but if you did not have 2 integer units for example, we could not have placed both the instructions; here i 2 would not have been scheduled, here i 2 possibly i 2 for example, has 0 and 1 as its Estart and Lstart. So, we would have actually placed it here without violation of any other precedence and so on and so forth. Because there is only 1 integer unit i 2 will have do you placed here.

The problem is more safe here because i 3 for example, it has 2, 5 it cannot be placed here. We would have placed actually i 3 in slot number 4. So, slot number 3, i 5 comes later. So, we would have placed this in slot number 3 and i 5 automatically would have been push to slot number 4. So, i 5 is can be between 2 and 7 so, it could have been placed here. So, that would have what you a schedule using the operation scheduling, but we would not have placed it to the exactly short value, but there is a slack so, there would not have been any pipelines stalls either. So, there are the length of the schedule would have remained as nine as in this particular case. So, that is an examples to show how operation scheduling can be done for basic blocks using slack as the priority.

(Refer Slide Time: 32:10)



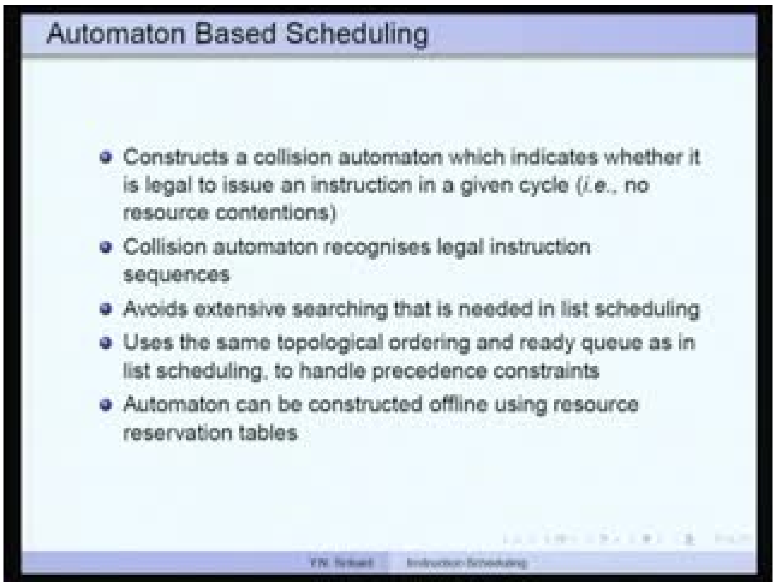
Simple List Scheduling - Disadvantages

- 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

What are the problems with this simple list scheduling, either the cycle by cycle method or the operation scheduling method. The checking of resource constraints is a bit inefficient, why? It involves repeated ANDing and OR-ing of bit matrices for many instructions in each scheduling step.

So, there are many matrices here. And if there are many resources as well the overhead may become considerable; but it is still not too bad; but there is a lot of work.

(Refer Slide Time: 32:58)

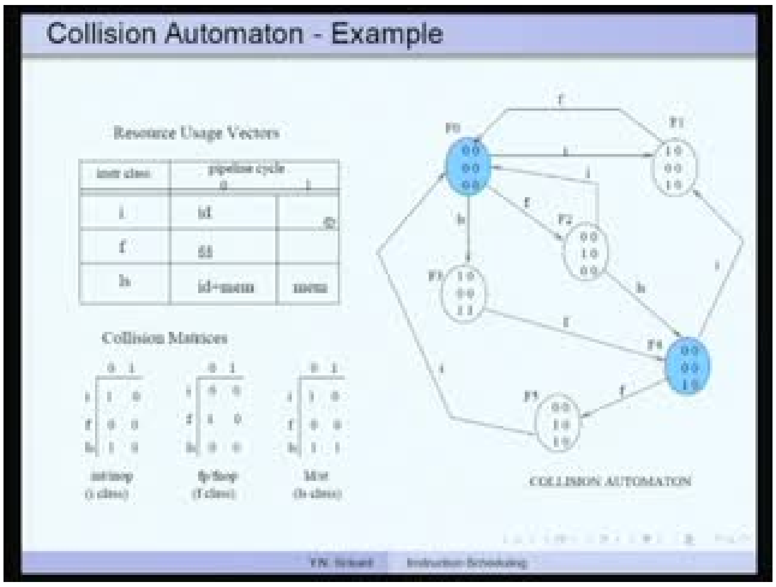


Automaton Based Scheduling

- Constructs a collision automaton which indicates whether it is legal to issue an instruction in a given cycle (*i.e.*, no resource contentions)
- Collision automaton recognises legal instruction sequences
- Avoids extensive searching that is needed in list scheduling
- Uses the same topological ordering and ready queue as in list scheduling, to handle precedence constraints
- Automaton can be constructed offline using resource reservation tables

Y.N. Srikant | Instruction Scheduling

(Refer Slide Time: 33:40)



Collision Automaton - Example

Resource Usage Vectors

instr class	pipeline cycle 0	1
i	id	
f	fd	
ls	id+mem	mem

Collision Matrices

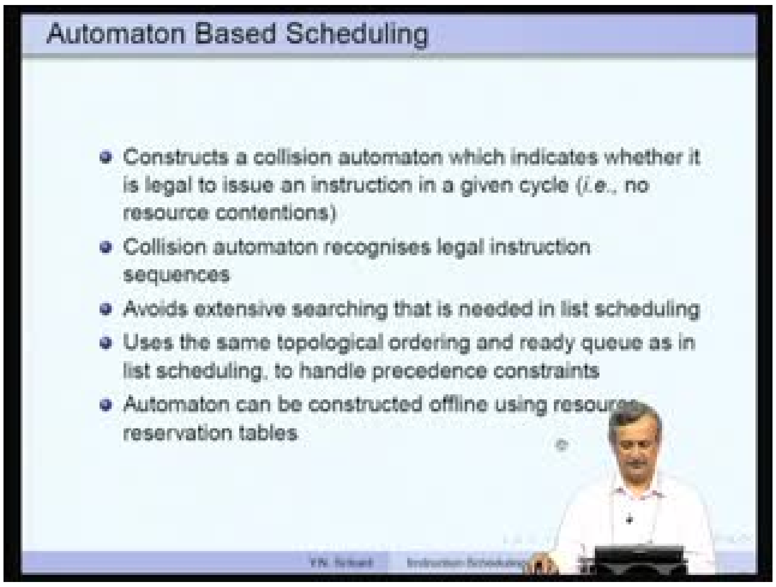
$$\begin{array}{c|cc} & 0 & 1 \\ \hline i & 1 & 0 \\ f & 0 & 0 \\ ls & 1 & 0 \end{array} \qquad \begin{array}{c|cc} & 0 & 1 \\ \hline i & 0 & 0 \\ f & 1 & 0 \\ ls & 0 & 0 \end{array} \qquad \begin{array}{c|cc} & 0 & 1 \\ \hline i & 1 & 0 \\ f & 0 & 0 \\ ls & 1 & 1 \end{array}$$

int/mop (i class) fp/fsop (f class) ld/st (ls class)

Y.N. Srikant | Instruction Scheduling

How do we avoid such overheads? So, in other words we want to actually get rid of the overheads involved in resource constraint checking. How do we do that? See that can be done by constructing what is known as a collision automaton. So, what is a collision automaton. So, collision automaton indicates whether it is legal to issue an instruction in a particular cycle that is what a collision automaton is all about. That is that me show you an example, here is the collision automaton for a simple example. These are the various states of the collision automaton. Each one of the states would have a matrix and there are many edges going out of this particular each state. So, this is a finite state automaton.

(Refer Slide Time: 34:10)

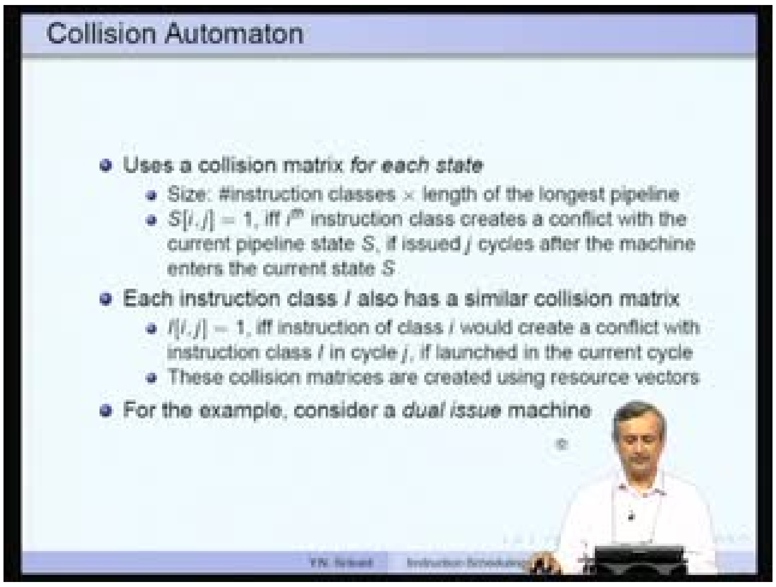


Automaton Based Scheduling

- Constructs a collision automaton which indicates whether it is legal to issue an instruction in a given cycle (*i.e.*, no resource contentions)
- Collision automaton recognises legal instruction sequences
- Avoids extensive searching that is needed in list scheduling
- Uses the same topological ordering and ready queue as in list scheduling, to handle precedence constraints
- Automaton can be constructed offline using resource reservation tables

So each of these edges is labeled with n instruction of class; let us go back so, the collision automaton what does it do it? Actually recognizes a legal instruction sequence. So, in other words, if we pick instructions from the ready queue, see if it is legal to be issued from a particular state; issue it; go to the next state; again issue whatever is possible and so on and so forth. We actually get a sequence of instructions which is legal. Because in every state we make sure that there are no resource constraints and the precedence part is taken care of by the ready queue. It avoids extensive searching that is needed in list scheduling and it uses the same topological ordering ready queue as in list scheduling to handle precedence constraints.

(Refer Slide Time: 35:13)

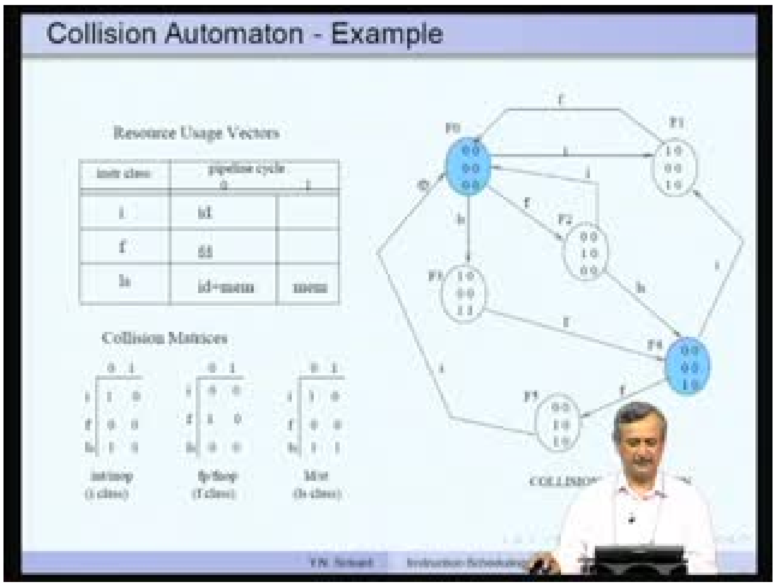


Collision Automaton

- Uses a collision matrix *for each state*
  - Size: #instruction classes × length of the longest pipeline
  - $S[i, j] = 1$, iff i^{th} instruction class creates a conflict with the current pipeline state S, if issued j cycles after the machine enters the current state S
- Each instruction class I also has a similar collision matrix
  - $I[i, j] = 1$, iff instruction of class i would create a conflict with instruction class I in cycle j, if launched in the current cycle
  - These collision matrices are created using resource vectors
- For the example, consider a *dual issue* machine

Y.N. Srikant Instruction Scheduling

(Refer Slide Time: 35:21)



Collision Automaton - Example

Resource Usage Vectors

instr class	pipeline cycle 0	1
i	id	
f	fd	
ls	id+mem	mem

Collision Matrices

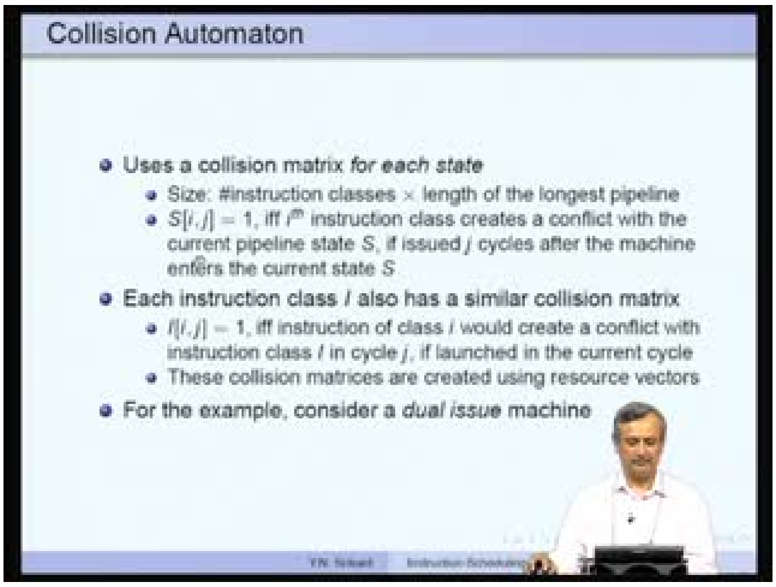
$$\begin{array}{c|cc} & 0 & 1 \\ \hline i & 1 & 0 \\ f & 0 & 0 \\ ls & 1 & 0 \end{array} \qquad \begin{array}{c|cc} & 0 & 1 \\ \hline i & 0 & 0 \\ f & 1 & 0 \\ ls & 0 & 0 \end{array} \qquad \begin{array}{c|cc} & 0 & 1 \\ \hline i & 1 & 0 \\ f & 0 & 0 \\ ls & 1 & 1 \end{array}$$

int/inop (i class) fp/flop (f class) M/st (ls class)

Y.N. Srikant Instruction Scheduling

The most important part is automaton can be constructed offline using the resource reservation tables. What are these tables? For it for example, it uses a collision matrix for each state. So, I showed you a collision matrix for example, this is the collision matrix for i class; this is the collision matrix for f class and this is the collision matrix for the load store class. And here is the resource usage vector for the instruction i class, that is a integer class. The pipeline requires, the integer d code in cycle 0 and nothing in cycle 1 for the floating point it requires the floating point d code and nothing the next one. For the load store it requires integer d code, then the memory and again it requires memory in the next cycle as well. So, this is the resource requirement not that this is a great pipeline machine, but it is it is suffices to show our example.

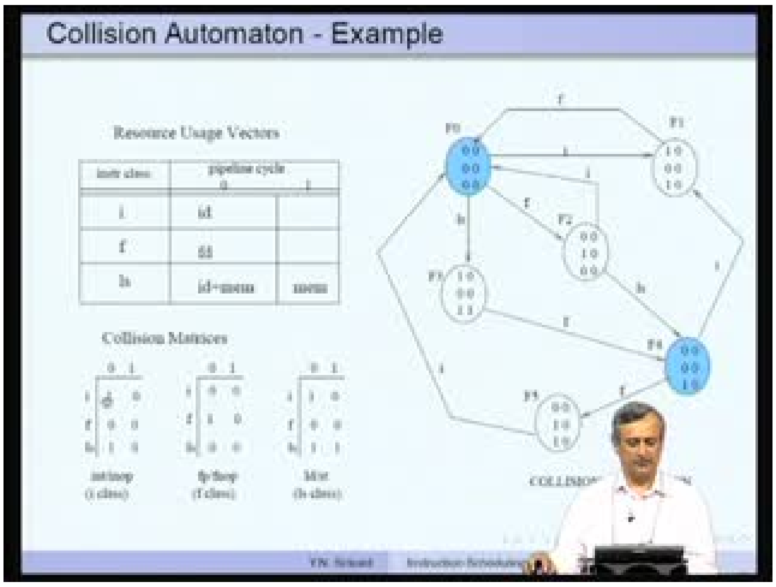
(Refer Slide Time: 36:28)



Collision Automaton

- Uses a collision matrix *for each state*
  - Size: #instruction classes × length of the longest pipeline
  - $S[i,j] = 1$, iff i^{th} instruction class creates a conflict with the current pipeline state S, if issued j cycles after the machine enters the current state S
- Each instruction class I also has a similar collision matrix
  - $I[i,j] = 1$, iff instruction of class i would create a conflict with instruction class I in cycle j, if launched in the current cycle
  - These collision matrices are created using resource vectors
- For the example, consider a *dual issue* machine

(Refer Slide Time: 36:37)



Collision Automaton - Example

Resource Usage Vectors

instr class	pipeline cycle 0	1
i	id	
f	fd	
ls	id+mem	mem

Collision Matrices

$$\begin{array}{c|cc} & 0 & 1 \\ \hline i & 1 & 0 \\ f & 0 & 0 \\ ls & 1 & 0 \end{array} \qquad \begin{array}{c|cc} & 0 & 1 \\ \hline i & 0 & 0 \\ f & 1 & 0 \\ ls & 0 & 0 \end{array} \qquad \begin{array}{c|cc} & 0 & 1 \\ \hline i & 1 & 0 \\ f & 0 & 0 \\ ls & 1 & 1 \end{array}$$

int/inop (i class) fp/fpop (f class) ld/st (ls class)

So, this is the reservation table showing for each instruction class what are the various resources used in the various cycles, and in the collision matrix for example, the size of the collision matrix is number of instructions into the length of the longest pipeline. So, here for example, these this is the length of the pipeline that is 2. So, we have as many rows as the number of instruction classes and as many columns as the length of the pipeline itself, longest pipeline.

What does a 1 or 0 indicate in this collision matrix table so, for example, it says for the instruction class i in cycle 0 if we issue another instruction of i class this is the table for i class. So, all these are i class considerations. So, this says the whether it is legal to issue another i class instruction in cycle 0 or cycle 1. And in cycle 0 if we issue that another i instruction 1 is already in progress, we trying to issue another one, there will be a conflict that is, what this one indicates. How did this happen its simple each i class instructions requires the i d unit. So, we cannot issue to i class instructions in cycle 0, there would be a conflict of interest.

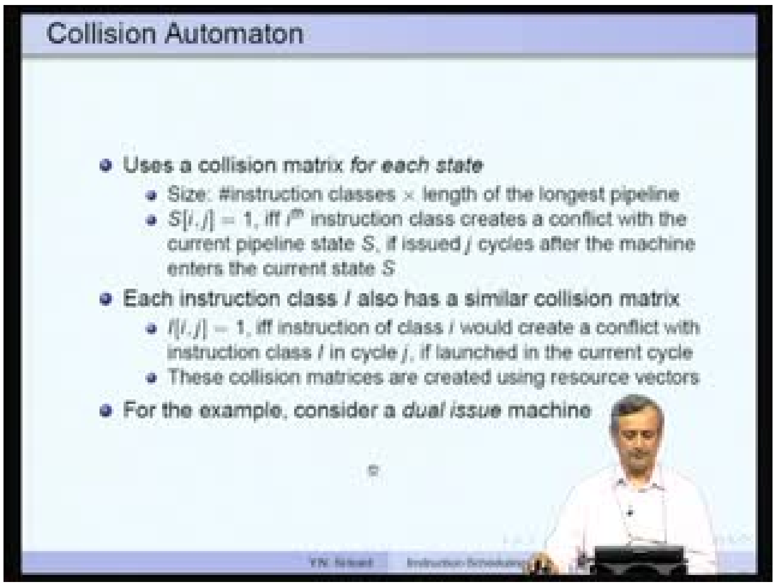
Whereas, if five in issue, the first i class instruction in cycle 0 and the second i class instruction in cycle 1, there is no conflict because cycle one does not require idea again.

There is obviously no conflict between i class and f class instructions. So, both cycle 0 and cycle 1 have 0s here. So, it is legal to actually issue an f class instruction along with an i class instruction either in cycle 0 or in cycle 1 after the i class instruction is initiated. There is a problem with the l s also, because l s also requires i d unit in cycle 0. So, we cannot issue an i class and l s class instruction together. So, there is a conflict in cycle 0 but in cycle 1 l s requires only mem so there is no conflict.

Let us look at the f. No conflict between i class so both are 0, conflict in cycle 0 of f class so 1 is here and 0 is here, that is what this is and there is no conflict with l s because l s does not requires the f d.

How about the last one l s class? There is a conflict with i class i cannot issue l s and i in the same cycle. So, this is 1 and this is 0. I can issue f with l s no problem, because it does not require any i d unit. I cannot issue another l s instruction together with this l s instruction, in either cycle 0 or cycle 1. So, both require mem so there will be conflict, that is what this is saying.

(Refer Slide Time: 39:29)



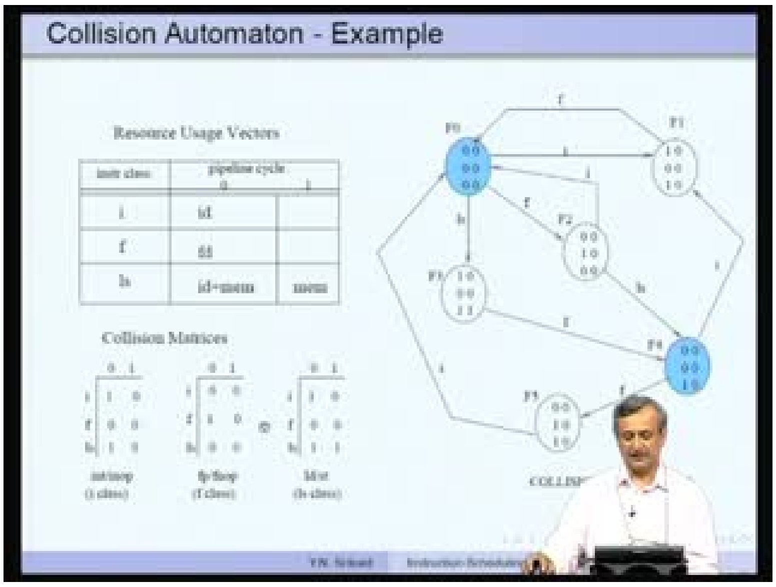
Collision Automaton

- Uses a collision matrix *for each state*
  - Size: #instruction classes × length of the longest pipeline
  - $S[i,j] = 1$, iff i^{th} instruction class creates a conflict with the current pipeline state S, if issued j cycles after the machine enters the current state S
- Each instruction class I also has a similar collision matrix
  - $I[i,j] = 1$, iff instruction of class i would create a conflict with instruction class I in cycle j, if launched in the current cycle
  - These collision matrices are created using resource vectors
- For the example, consider a *dual issue* machine

Y.N. Srikant | Instruction Scheduling

S i j equal to 1 if and only if ith instruction class creates a conflict with the current pipeline state S if issued j cycles after the machine enters the current state S. So, that is why, there are many state.

(Refer Slide Time: 40:17)



Collision Automaton - Example

Resource Usage Vectors

instr class	pipeline cycle 0	1
i	id	
f	fd	
ls	id+mem	mem

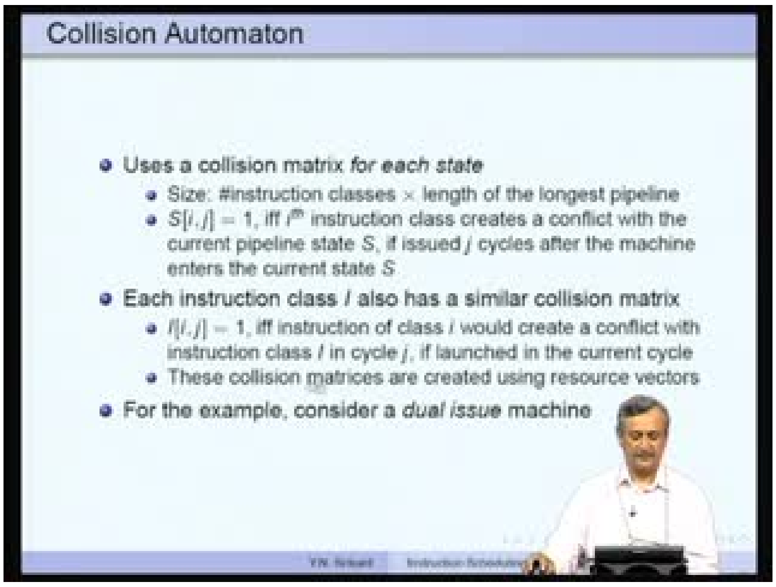
Collision Matrices

$$\begin{array}{c|cc} & 0 & 1 \\ \hline i & 1 & 0 \\ f & 0 & 0 \\ ls & 1 & 0 \end{array} \qquad \begin{array}{c|cc} & 0 & 1 \\ \hline i & 0 & 0 \\ f & 1 & 0 \\ ls & 0 & 0 \end{array} \qquad \begin{array}{c|cc} & 0 & 1 \\ \hline i & 1 & 0 \\ f & 0 & 0 \\ ls & 1 & 1 \end{array}$$

int/nop (i class) fp-fnop (f class) ld/st (ls class)

Y.N. Srikant | Instruction Scheduling

(Refer Slide Time: 40:21)



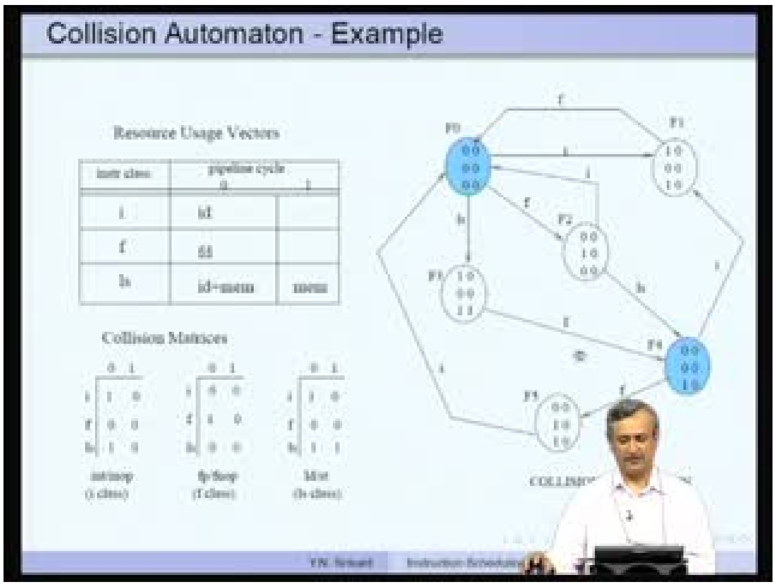
Collision Automaton

- Uses a collision matrix *for each state*
  - Size: #instruction classes × length of the longest pipeline
  - $S[i,j] = 1$, iff i^{th} instruction class creates a conflict with the current pipeline state S, if issued j cycles after the machine enters the current state S
- Each instruction class I also has a similar collision matrix
  - $I[i,j] = 1$, iff instruction of class i would create a conflict with instruction class I in cycle j, if launched in the current cycle
  - These collision matrices are created using resource vectors
- For the example, consider a *dual issue* machine

Y.N. Srikant | Instruction Scheduling

Now once you know this is the collision matrix for each state, but in each instruction class i also has a similar collision matrix. So, i a i j equal to 1, if and only if instruction class i would create a conflict with instruction class i; in cycle j; if launched in the current cycle that is what I showed here. These are the instruction class collision matrices. These collision matrices are created using the resource vectors. Now, we also saw the matrix in each of these states, what is that about? That is what this is.

(Refer Slide Time: 40:46)



Collision Automaton - Example

Resource Usage Vectors

instr class	pipeline cycle 0	1
i	id	
f	fd	
ls	id+mem	mem

Collision Matrices

$$\begin{array}{c|cc} & 0 & 1 \\ \hline i & 1 & 0 \\ f & 0 & 0 \\ ls & 1 & 0 \end{array} \qquad \begin{array}{c|cc} & 0 & 1 \\ \hline i & 0 & 0 \\ f & 1 & 0 \\ ls & 0 & 0 \end{array} \qquad \begin{array}{c|cc} & 0 & 1 \\ \hline i & 1 & 0 \\ f & 0 & 0 \\ ls & 1 & 1 \end{array}$$

int/mop (i class)   fp-fiop (f class)   ld/st (ls class)

Y.N. Srikant | Instruction Scheduling

So, in the collision matrix, for the state S i j equal to 1 if and only, if the ith instruction class creates a conflict with the current pipeline state S, if issued j cycles after the machine enters the current state S. So in other words, the first column corresponds to the current cycle, the next column and corresponds to the cycle after the current cycles. So current cycle plus 1, and if you had a third column it would have been current cycles plus 2 and so on. How many columns do we have? That is straight forward, you know, as much as the length of the pipeline - longest pipeline. So, that is what we have to here. And how many rows do we have? As many as number of instruction classes.

So, that is 3 here. How does the whole thing work? To begin with, we are in straight 0 so, forget the blue color for the present. The collision matrix has all 0s, because all resources are available no instructions have been issued so far so, there are no conflicts. Therefore, it is legal to issue any instruction from the state f 0 the Estart state. So, let us say, we issue either i or f or l s, we go to a different state. How do we compute the new state here? Just take this particular sate collision matrix, or it with the collision matrix of that particular instruction class which you are considering. When we branch on i class take the i class collision matrix, which is 1 0 0 0 1 0, or it with this so we get 1 0 0 0 1 0.

Similarly, if we issue an f instruction, we get this or with the collision matrix of the f class which is 0 0 1 0 0, then for the l s we get 1 0 0 0 1 1.

Now the situation becomes slightly more difficult. Which instructions are legal from these three states? So, to check that what we really do is, we have remember launched an instruction, but we have not advanced any cycle counters. We are still in the same cycle as we were in this particular state.

Let us say we consider state f 1, look at the first column which corresponds to the current cycle. If there are any 0s in the first column, then the instruction corresponding to that particular row or the instructions in the class corresponding to that particular row are all legal to be issued from this particular state without any collision.

So, in this case the f class instructions are legal to be issued from the state f 1 but there is a 1 in the other 2 elements of this particular vector in the column, i class and l s class. So, those are illegal to be issued from the state f 1 y that is straight forward there are already 1s here so, in the same cycle I cannot issue another i class and l class l s class, whereas f class is legal.

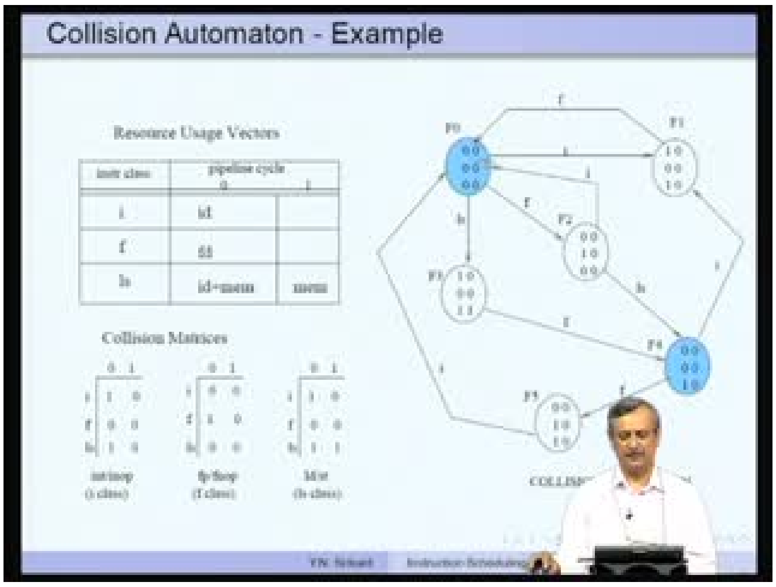
So, it is legal to issue another f class instruction from state f 1 let us do that. What do we get? You the new state that we get, obviously is not this. We take 1 0 0 0 1 0 or it with this particular f class collision matrix so, there is a 1 here and that comes here so, we get 1 1 1 0 0 0.

Now there are no more 0s in the first column of the state f 1 prime that we just now computed. which means we have only three classes of instructions in our example; that means there are no instructions belonging to any class that can be issued legally from this particular state f 1 prime, because it has all 1s in the first column. There would only be collisions if we issues them.

In such a case, what we do is, we have to now advance the cycle y 1. How do we capture that effect in the state diagram? Simply left shift, so we now have f 1 prime has 1 1 1 0 0 0, simply left shift by one column the new state matrix that we got. So, if we do that this 1 1 1 0 0 0 becomes 0 0 0 0 0 0 because 0 shift from the right end.

That state is nothing but the Estart state. This indicates that now again all the resources are available. If you have any 1s here, you know 0 1 0 or something like that in the second column, then 0 1 0 would have become the first column, we would have go on to new state not this particular state.

(Refer Slide Time: 40:46)



Collision Automaton - Example

Resource Usage Vectors

instr class	pipeline cycle 0	1
i	id	
f	fil	
ls	id+mem	mem

Collision Matrices

int/mop (i class)

$$\begin{array}{c|cc} & 0 & 1 \\ \hline i & 1 & 0 \\ f & 0 & 0 \\ ls & 1 & 0 \end{array}$$

fp/flop (f class)

$$\begin{array}{c|cc} & 0 & 1 \\ \hline i & 0 & 0 \\ f & 1 & 0 \\ ls & 0 & 0 \end{array}$$

ld/st (ls class)

$$\begin{array}{c|cc} & 0 & 1 \\ \hline i & 1 & 0 \\ f & 0 & 0 \\ ls & 1 & 1 \end{array}$$

YN Srikant | Instruction Scheduling

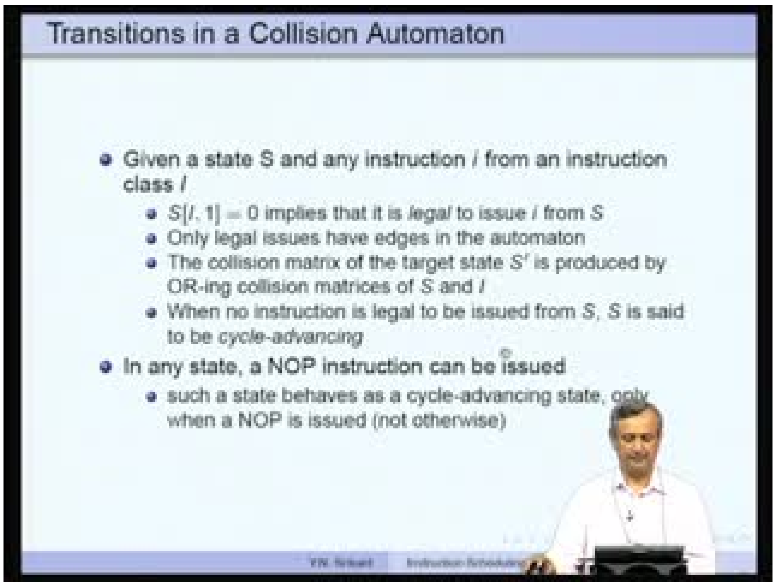
So, from here on issue of f, we go back to this particular state f 0 and we have also advanced the cycle. So, that is precisely what we want to show here. This blue indicates that it is a cycle-advancing state. So, once we jump to that state, we find that it is cycle-advancing state after the left shift of the column. We jump to f 0 and then advance the cycle.

So let us look at, this state f 2. This has only 1 in the first column. So, it is legal to issue i and l s class instructions from f 2. So, there are 2 arcs, with one with i, another with l s. With i, if you arc it with the i class collision matrix, we get all 1s in the first column so similar to f 1 we left shift and then go back to f 0. Whereas, if we had issued in l s class instruction that was a 1 0 1 and there was a 1 in the second column also. So, this matrix becomes 1 1 1 0 0 1 and when we left shift it becomes 0 0 1 and 0 0 0, that is what we have shown here. So, again we come to this new state and that is a cycle-advancing state because we have done a left shift.

So from here again, we find that there are two 0s so i class and l s f class are legal to be issued; so we issue that i and f. So, we go to this particular state on i and f on this particular state; again from here when we issue an i, we go to this state so on and so forth.

This is how the collision automaton works. How do we use it in scheduling? Let us see so I hope this is understood the salient points are when we have a starts state all resources are available, so any instructions can be issued from the Estart state. Once we issue an instruction they are to compute the new collision matrix. We are the state matrix with the collision matrix of that particular instruction class, we get a new state. We check the first column of that particular state, to see if it has any 0s; if it has then the column the instructions corresponding to that class are legal to be issued. Other if there are only ones, then we left s hift the matrix by one column and then if it is a new state, we go to that new state and make it cycle-advancing, so that is a summary of the collision automaton and its construction working etcetera.

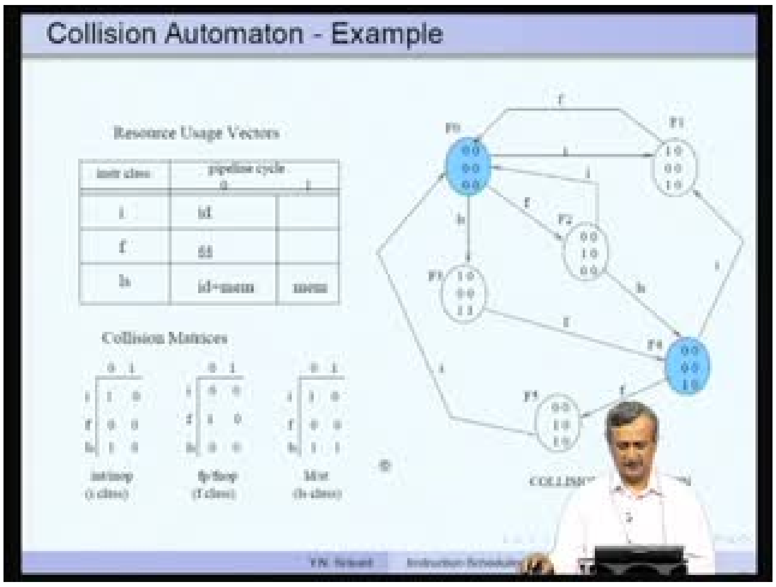
(Refer Slide Time: 48:46)



Transitions in a Collision Automaton

- Given a state S and any instruction *I* from an instruction class *I*
  - *S*[*I*, 1] = 0 implies that it is *legal* to issue *I* from *S*
  - Only legal issues have edges in the automaton
  - The collision matrix of the target state *S′* is produced by OR-ing collision matrices of *S* and *I*
  - When no instruction is legal to be issued from *S*, *S* is said to be *cycle-advancing*
- In any state, a NOP instruction can be issued
  - such a state behaves as a cycle-advancing state, only when a NOP is issued (not otherwise)

(Refer Slide Time: 49:31)



Collision Automaton - Example

Resource Usage Vectors

instr class	pipeline cycle 0	1
i	id	
f	fd	
ls	id+mem	mem

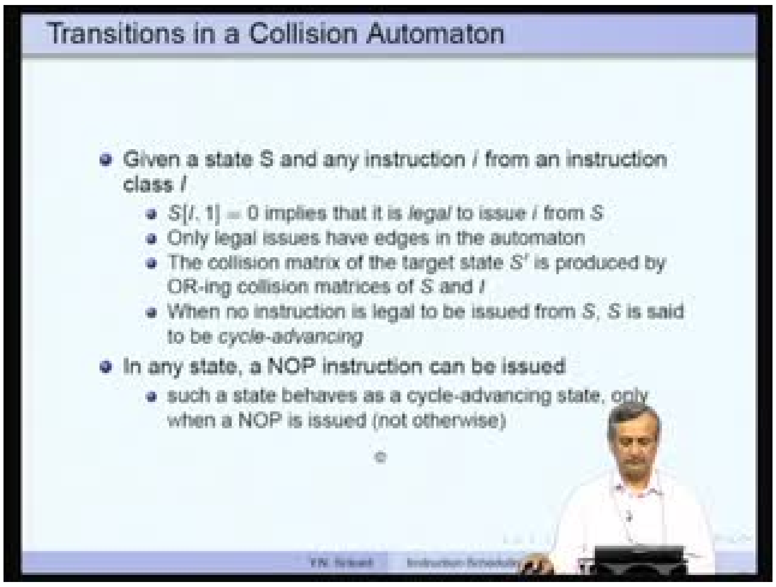
Collision Matrices

$$\begin{array}{c|cc} & 0 & 1 \\ \hline i & 1 & 0 \\ f & 0 & 0 \\ ls & 1 & 0 \end{array} \qquad \begin{array}{c|cc} & 0 & 1 \\ \hline i & 0 & 0 \\ f & 1 & 0 \\ ls & 0 & 0 \end{array} \qquad \begin{array}{c|cc} & 0 & 1 \\ \hline i & 1 & 0 \\ f & 0 & 0 \\ ls & 1 & 1 \end{array}$$

int/inop (i class) fp/flop (f class) ld/st (ls class)

So what are the transitions in the collision automaton? So, whatever I said is formally written here, given a state S and any instruction i from an instruction class i; s of i comma 1 is equal to 0 implies that it is legal to issue an instruction i from s, that is what I said. Only legal issues have edges in the automaton. The collision matrix of the target state S prime is produced by OR-ing the matrices of s and i. When no instruction is legal to be issued from S S is said to be cycle-advancing. In any state this is important a NOP instruction can be issued. It is always legal to issue a NOP instruction from any of these we have not written the collision matrix for NOP class but there are going to be no collisions with any NOP instruction, because NOP does not require any resources.

(Refer Slide Time: 49:45)

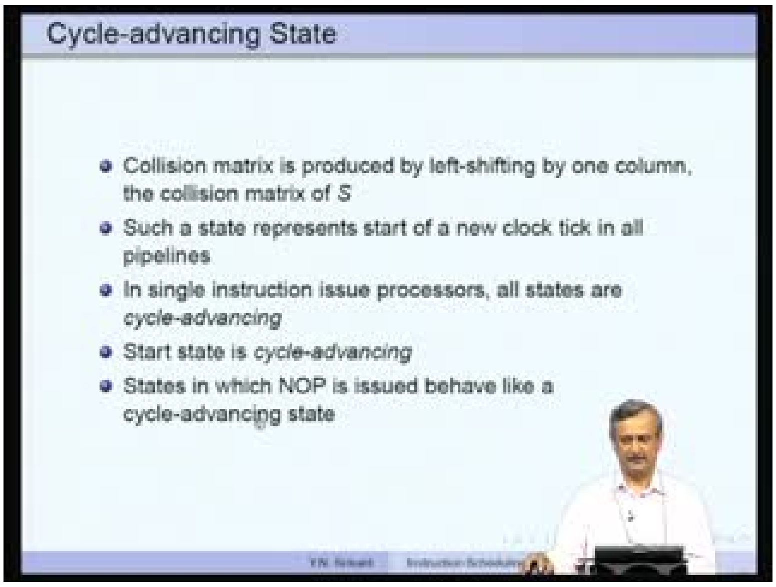


Transitions in a Collision Automaton

- Given a state S and any instruction *I* from an instruction class *I*
  - *S*[*I*, 1] = 0 implies that it is *legal* to issue *I* from *S*
  - Only legal issues have edges in the automaton
  - The collision matrix of the target state *S'* is produced by OR-ing collision matrices of *S* and *I*
  - When no instruction is legal to be issued from *S*, *S* is said to be *cycle-advancing*
- In any state, a NOP instruction can be issued
  - such a state behaves as a cycle-advancing state, only when a NOP is issued (not otherwise)

Such a state behaves as a cycle-advancing state only when, a NOP, is issued otherwise, it is not going to behave as a cycle-advancing state.

(Refer Slide Time: 50:00)

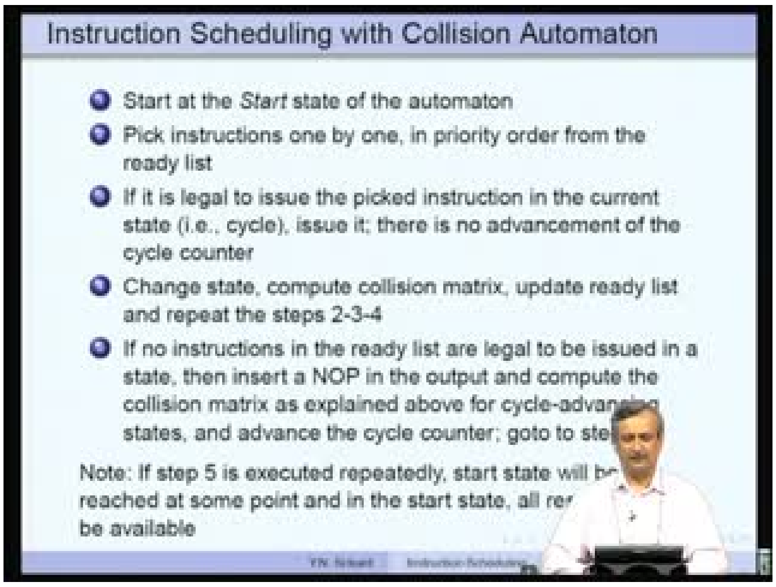


Cycle-advancing State

- Collision matrix is produced by left-shifting by one column, the collision matrix of *S*
- Such a state represents start of a new clock tick in all pipelines
- In single instruction issue processors, all states are *cycle-advancing*
- Start state is *cycle-advancing*
- States in which NOP is issued behave like a cycle-advancing state

What is a cycle-advancing state? Collision matrix is produced by left shifting one column- the collision matrix of S. So, such a state represents start of a new clock tick in all pipeline in single instruction issue processors all states are cycle-advancing. Start state is cycle-advancing and states in which NOP is issued behave like a cycle-advancing state.

(Refer Slide Time: 50:32)



Instruction Scheduling with Collision Automaton

- Start at the *Start* state of the automaton
- Pick instructions one by one, in priority order from the ready list
- If it is legal to issue the picked instruction in the current state (i.e., cycle), issue it; there is no advancement of the cycle counter
- Change state, compute collision matrix, update ready list and repeat the steps 2-3-4
- If no instructions in the ready list are legal to be issued in a state, then insert a NOP in the output and compute the collision matrix as explained above for cycle-advan states, and advance the cycle counter; goto to ste

Note: If step 5 is executed repeatedly, start state will b reached at some point and in the start state, all re be available

Y.N. Srikant Instruction Scheduling

Now how do we do instruction scheduling with the collision automaton. Fairly straight forward, you begin at the Estart state of the automaton. Remember, we have the same ready list that was available to us in cycle by cycle scheduling. So pick instructions one by one in the priority order from the ready list.

So if it is legal to issue the picked instruction in the current state that is you have a 0 corresponding to that instruction class in the state matrix; issue it. There is no advancement of the cycle counter now. But there is a change of state on that particular instruction class takes that arc; go to the new state. Compute the collision matrix by OR-ing the state matrix with the collision matrix of the instruction class. Now, you must update the ready list because once this instruction is issued its successes become eligible to be issued and repeat the steps 2-3-4; so, this goes on. But if no instructions in the ready list are legal to be issued in a state, then, we insert a NOP in the output and compute the collision matrix as we explained for the cycle-advancing states that are left shifting of the column. Now we also advance the cycle counter and go back to step 2

The reason is there are no instructions which are legal to be issued. Now, we insert a NOP in the output that means, a NOP, will be executed when nothing is being possible now new instruction it can be executed. So, once we do that we have actually skipped one cycle, so that means it is a cycle-advancing state.

Now, because we have skipped 1 cycle it is possible that more resources are now available. So, that is why the left shifting of the column takes place. So, once we do that more resources may be available once the cycle has been skipped; new instructions may be schedulable in the current cycle that we get. If step 5 is executed repeatedly, the Estart state will be reached at some point and in the Estart state obviously all the resources will be available. Why? Look at this way each one of the instructions requires a finite amount of resource and it will definitely complete in a certain finite amount of time. So, if we go on skipping cycles and insert NOPs, when all the resources are busy eventually, in a couple of cycles all the resources will become available again. In other words, if we executes step 5 of cycle-advancing state, repeatedly in a few steps after all the resources become available we will be reaching the Estart state. So, again we can start re issuing instructions from the Estart state onwards.

So this is how the collision automaton works. The advantage of the collision automaton is that we do not have to do any checking of the resources using reservation table etcetera, that is, all inbuilt in the collision automaton itself. But, the problem with collision automaton is its size can become very large for a very large instruction said. People actually have looked at this issue they have considered factoring the automaton, making smaller automaton and so on and so forth.

So, these are some of the advanced topics that are outside the scope of this particular lecture. This is the end of the lecture today and in the next lecture, we will look at optimal interior linear programming based instructions scheduling. Thank you.