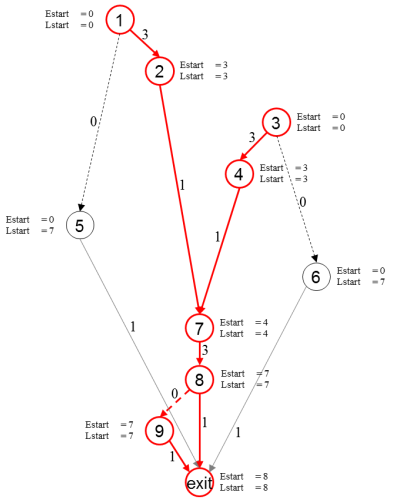
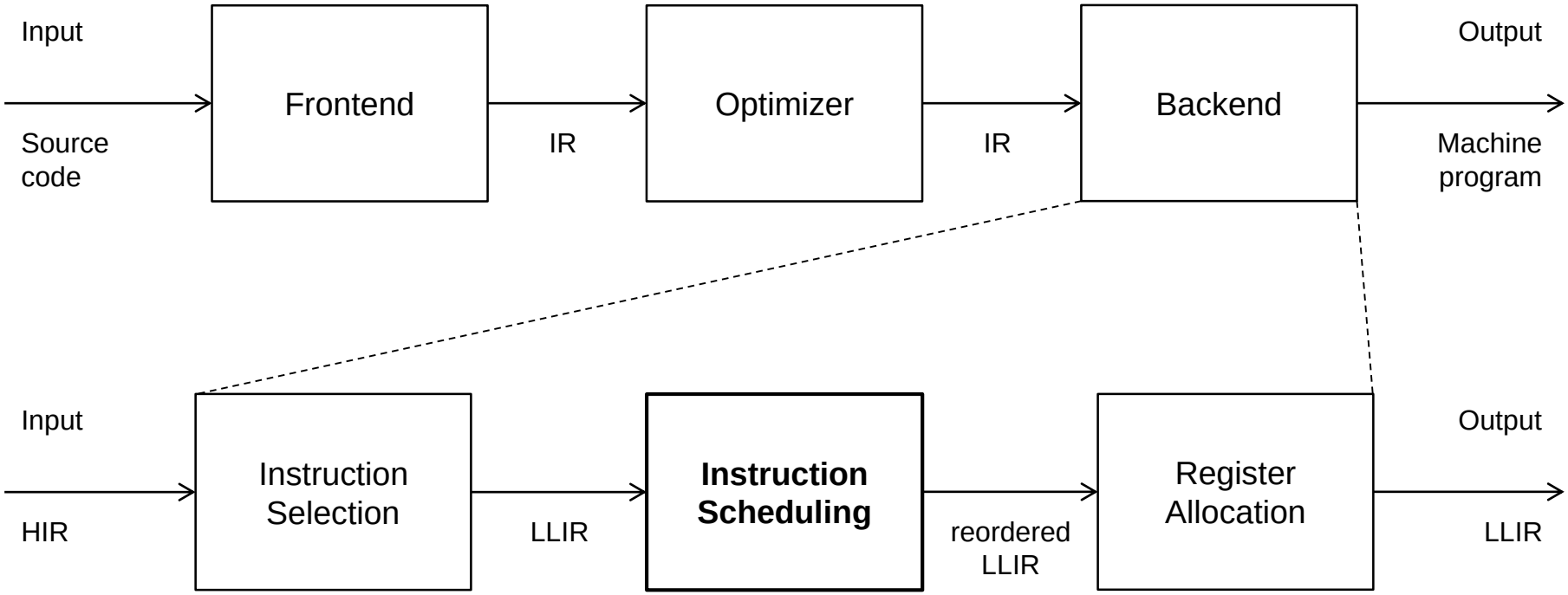
Instruction Scheduling

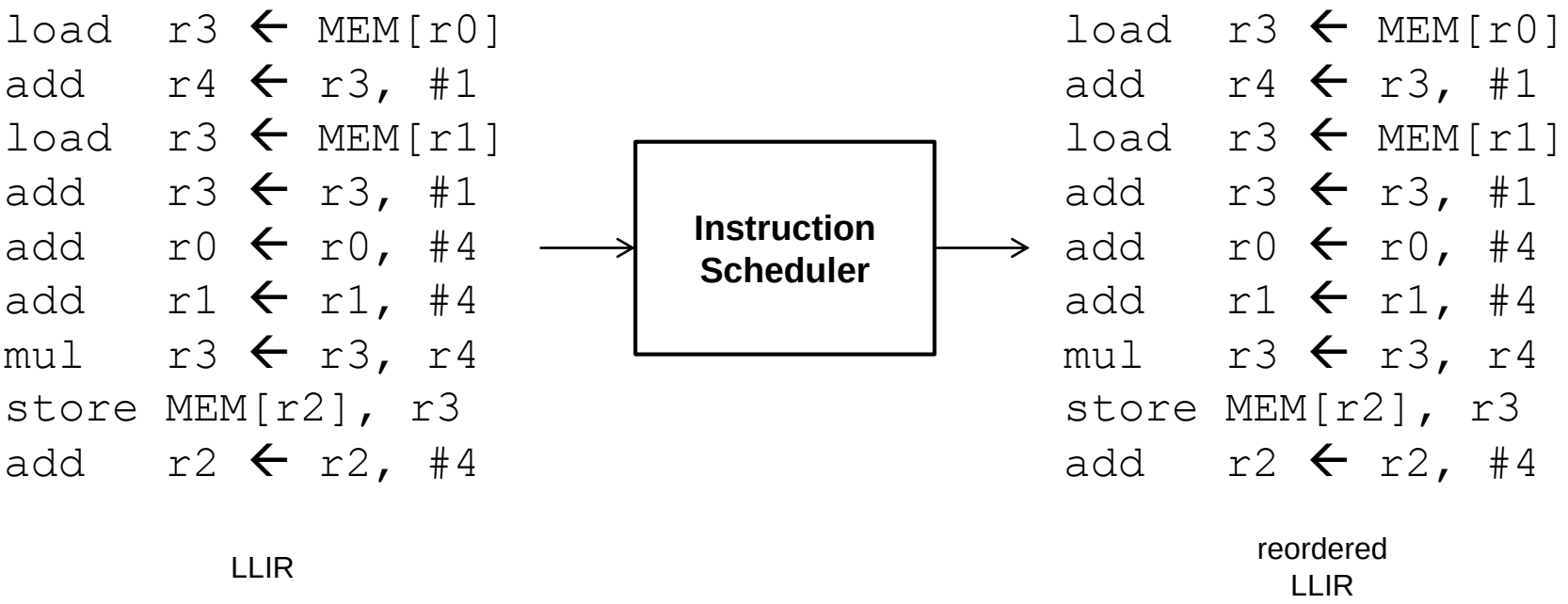


Instruction Scheduling



Motivation

- Or: why don't we just issue the code as-is?

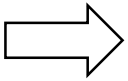


Motivation

- Assume a target machine with the following properties
  - pipelined with forwarding, single issue, in-order
  - operation latencies: add, sub: 1 cycle; mul, load: 3 cycles; store: 1 cycle

- Executing the code from before

```
load  r3 ← MEM[r0]
add   r4 ← r3, #1
load  r3 ← MEM[r1]
add   r3 ← r3, #1
add   r0 ← r0, #4
add   r1 ← r1, #4
mul   r3 ← r3, r4
store MEM[r2], r3
add   r2 ← r2, #4
```

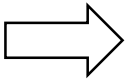


Motivation

- Assume a target machine with the following properties
  - pipelined with forwarding, single issue, in-order
  - operation latencies: add, sub: 1 cycle; mul, load: 3 cycles; store: 1 cycle

- Executing the code from before

```
load  r3 ← MEM[r0]
add   r4 ← r3, #1
load  r3 ← MEM[r1]
add   r3 ← r3, #1
add   r0 ← r0, #4
add   r1 ← r1, #4
mul   r3 ← r3, r4
store MEM[r2], r3
add   r2 ← r2, #4
```

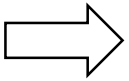


cycle	operation
1	load r3 ← MEM[r0]
2	
3	
4	add r4 ← r3, #1
5	load r3 ← MEM[r1]
6	
7	
8	add r3 ← r3, #1
9	add r0 ← r0, #4
10	add r1 ← r1, #4
11	mul r3 ← r3, r4
12	
13	
14	store MEM[r2], r3
15	add r2 ← r2, #4

Motivation

- Can we do better?

```
load  r3 ← MEM[r0]
add   r4 ← r3, #1
load  r3 ← MEM[r1]
add   r3 ← r3, #1
add   r0 ← r0, #4
add   r1 ← r1, #4
mul   r3 ← r3, r4
store MEM[r2], r3
add   r2 ← r2, #4
```



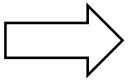
cycle	operation
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	
12	
13	
14	
15	

target machine properties
- pipelined with forwarding, single issue, in-order
- operation latencies: add, sub: 1 cycle; mul, load: 3 cycles; store: 1 cycle

Motivation

- Can we do better?

```
load  r3 ← MEM[r0]
add   r4 ← r3, #1
load  r3 ← MEM[r1]
add   r3 ← r3, #1
add   r0 ← r0, #4
add   r1 ← r1, #4
mul   r3 ← r3, r4
store MEM[r2], r3
add   r2 ← r2, #4
```



cycle	operation
1	load r4 ← MEM[r0]
2	load r3 ← MEM[r1]
3	add r0 ← r0, #4
4	add r4 ← r4, #1
5	add r3 ← r3, #1
6	mul r3 ← r3, r4
7	add r1 ← r1, #4
8	
9	store MEM[r2], r3
10	add r2 ← r2, #4

target machine properties
- pipelined with forwarding, single issue, in-order
- operation latencies: add, sub: 1 cycle; mul, load: 3 cycles; store: 1 cycle

Motivation

- Comparison

cycle	operation
1	load r3 ← MEM[r0]
2	
3	
4	add r4 ← r3, #1
5	load r3 ← MEM[r1]
6	
7	
8	add r3 ← r3, #1
9	add r0 ← r0, #4
10	add r1 ← r1, #4
11	mul r3 ← r3, r4
12	
13	
14	store MEM[r2], r3
15	add r2 ← r2, #4

versus

cycle	operation
1	load r4 ← MEM[r0]
2	load r3 ← MEM[r1]
3	add r0 ← r0, #4
4	add r4 ← r4, #1
5	add r3 ← r3, #1
6	mul r3 ← r3, r4
7	add r1 ← r1, #4
8	
9	store MEM[r2], r3
10	add r2 ← r2, #4

33% improvement

- Can we do even better? What is the optimal schedule?
- Should we do better? After all, we got out-of-order processors by now

Instruction Scheduling

- Solves the problem: how do we get a good schedule?
  i.e., defines an ordering of the operations in a partially ordered list of operations

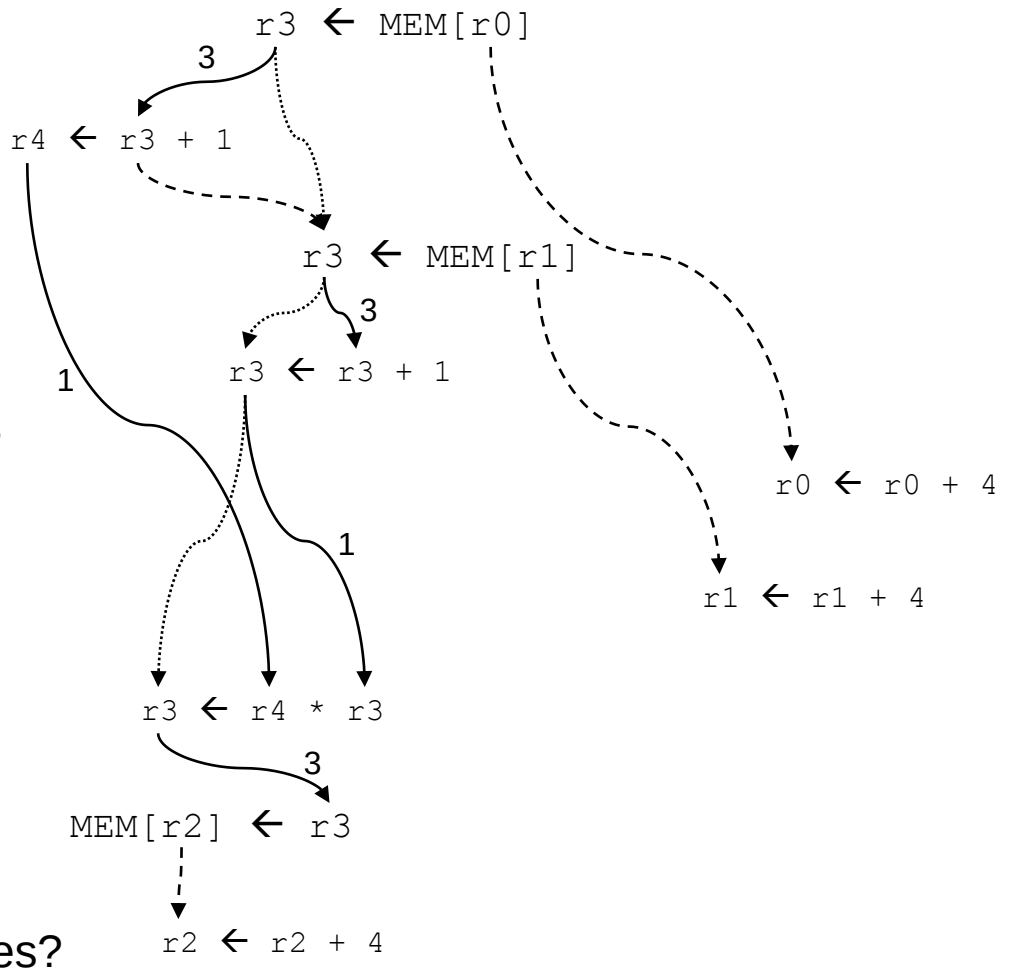
  - main constraint
    - preserve meaning of the code
      (control flow, data flow)

  - “good” schedule?
    - typically “shortest” in terms of execution time
    - under additional constraints (i.e., register pressure)

  - under consideration of H/W properties
    - operation latencies
    - processor pipeline
    - # of functional units (FU) available
    - memory hierarchy
    - …

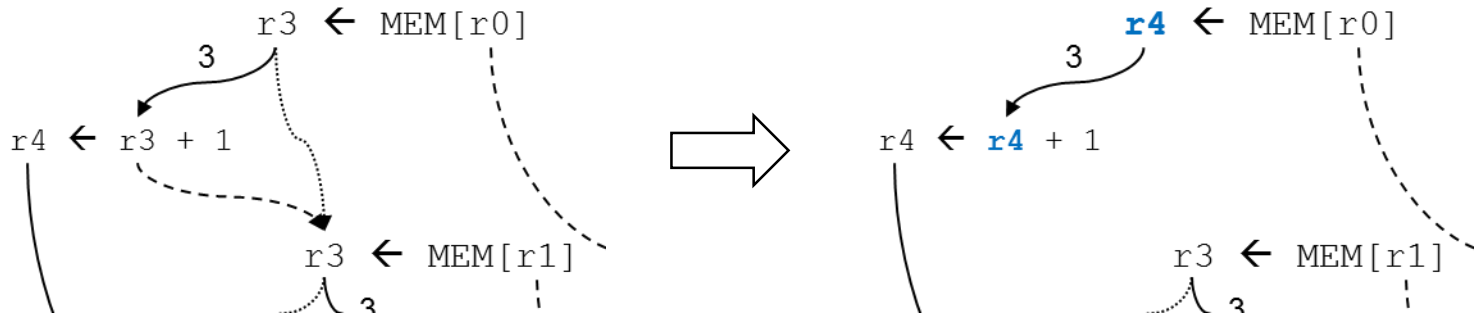
Data Dependence Graph

- $DDG = (V, E)$
  - nodes V representing each operation and augmented with
    - operation type
    - operation latency (delay)
  - edges E representing data dependences between operations
    - forward (def-use)
    - anti (use-def)
    - output (def-def)
  - root nodes = no successors
    leave nodes = no predecessors
  - latencies on nodes or edges?
  - latency of anti/output dependences?



Renaming

- Dealing with anti/output dependences
  - anti/output dependences are artificial dependences that constrain the scheduler
  - can be eliminated by renaming
    - effect on register pressure?
    - can we eliminate all anti/output dependences?



Instruction Schedule

- **$S(n)$:** $n \in V \rightarrow t \in \mathbb{N}^+$
  a mapping from an operation n to an non-negative integer t denoting the cycle in which the operation should be issued.

  Constraints:

  - S(n) ≥ 1 (and at least one operation o with S(o) = 1)
  - if $(n_1, n_2) \in E$ then S(n_1) + delay(n_1) ≤ S(n_2)
  - for each t, there are no more operations with S(n) = t than the H/W can support

- Length of the schedule
  $L(S) = \max_{n \in V} (S(n) + \text{delay}(n))$

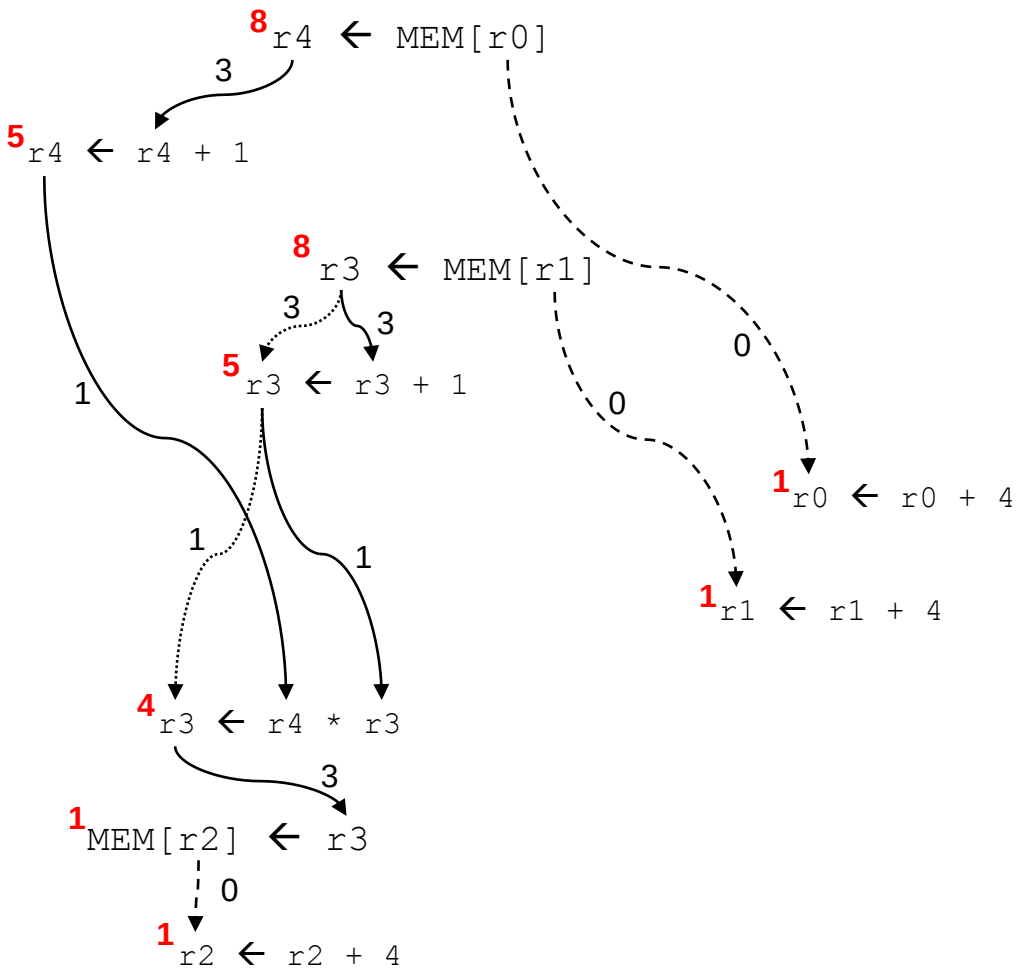
S

cycle t	operation n
1	load r4 ← MEM[r0]
2	load r3 ← MEM[r1]
3	add r0 ← r0, #4
4	add r4 ← r4, #1
5	add r3 ← r3, #1
6	mul r3 ← r3, r4
7	add r1 ← r1, #4
8	
9	store MEM[r2], r3
10	add r2 ← r2, #4

$L(S) = 11$

Instruction Schedule

- Path length
  starting at the root, annotate each node with its accumulated delay
- the **critical path** is the longest path over all paths in the data dependence graph
- the minimal schedule cannot be shorter than the critical path



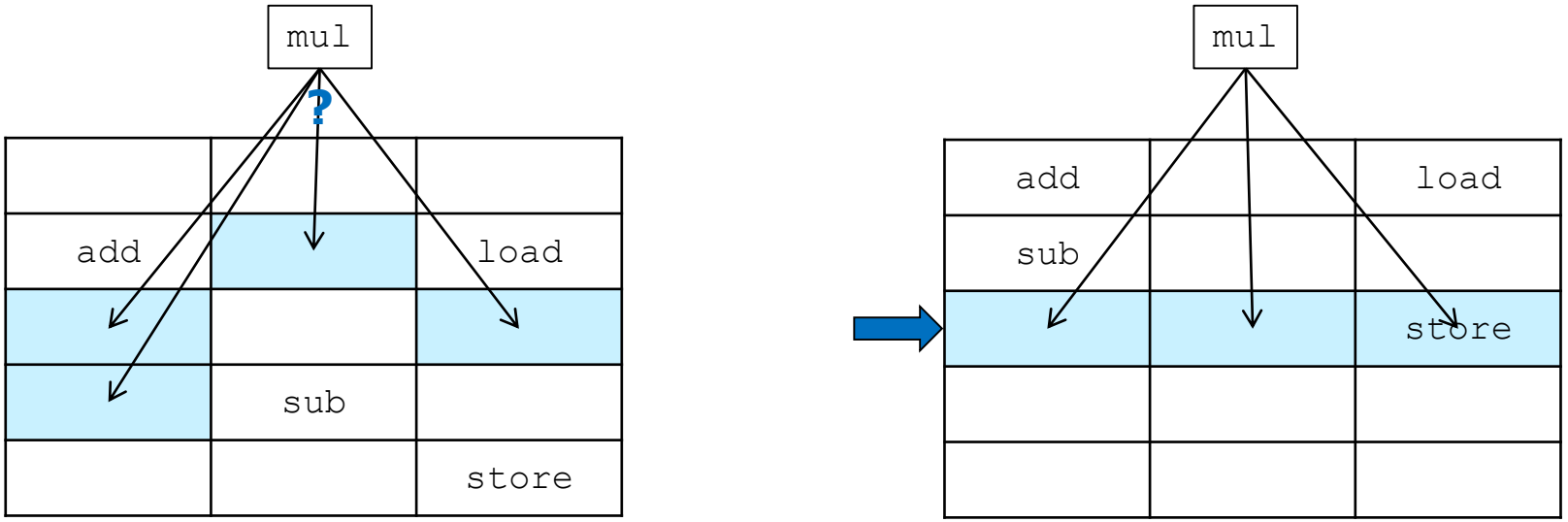
Instruction Scheduling Techniques

- Local instruction scheduling is an NP-complete problem (scheduling → job shop scheduling → TSP)

- Classification of techniques

scheduling	cycle	
	operation	
search	greedy	
	backtrack	
flow analysis	linear	
	graph	
scheduling unit	acyclic	basic block
		trace
		DAG
	cyclic	

Cycle vs. Operation Scheduling



- operation scheduling more powerful than cycle-based scheduling in the presence of long-latency operations
  - however, much more complicated to implement

Linear vs. Graph-Based Techniques

- Linear techniques
  - runtime O(n)
  - produces the schedule by one or more passes over the input LLIR
  - most common technique: critical-path scheduling
    - three passes: ASAP, ALAP, non-critical operations
  - limitation: unable to consider global properties of operations

- Graph-based techniques
  - runtime: O(n^2) for DAG creation plus scheduling
  - prevalent technique: list scheduling (O($n\log n$))
    - greedy: select one operation and schedules it

List Scheduling

- The prevalent scheduling heuristics are based on **list scheduling**

- Method
  - rename (optional)
  - build data dependence graph
  - assign priorities to operations
  - iteratively select and schedule an operation

List Scheduling

- The list scheduling algorithm

```
t := 1
ready := { leaves of DDG }
active := {}
while (ready ∪ active ≠ {})  do
  for each operation o in active do
    if (S(o) + delay(o) < t) then
      active := active \ {o}
      for each successor s of o in DDG do
        if (s is ready) then
          ready := ready ∪ {s}

  if (ready ≠ {}) then
    o := pick the operation from ready
         with the highest priority
    if (o can be scheduled on the H/W units) then
      ready := ready \ {o}
      active := active ∪ {o}
      S(op) := t

  t := t + 1
end
```

< or ≤?

List Scheduling

- Picking an operation from `ready`
  - if `ready` never contains more than one operation, the generated schedule is optimal
  - if more than one operations are ready, the choice of the next-to-be-scheduled operation is critical to the performance of the algorithm
    - pick the operation with the highest priority
    - most algorithms use several priorities to break ties

- How do we compute these priorities?

Priority Functions

- Priority function in list scheduling

```
procedure ListScheduling(…);
begin
  …
     o := pick one operation from ready
          using some priority function;
  …
end;
```

  - common priority functions:
    - height: distance from exit node
      gives priority to amount of work left
    - slackness: inverse of slack
      gives priority to operations on the critical path
    - register use: number of source operands
      reduces the number of live registers
    - uncover: fanout (number of children)
      free up nodes quickly
    - original instruction order

List Scheduling

- Priorities based on the DDG
  - *Estart*: earliest start time (ASAP – as soon as possible)

$$Estart(op) = \begin{cases} 0 & if\ op\ has\ no\ predecessors \\ \max\limits_{p \in pred(op)} Estart(p) + latency(p) & otherwise \end{cases}$$

  - *Lstart*: latest start time (ALAP – as late as possible)

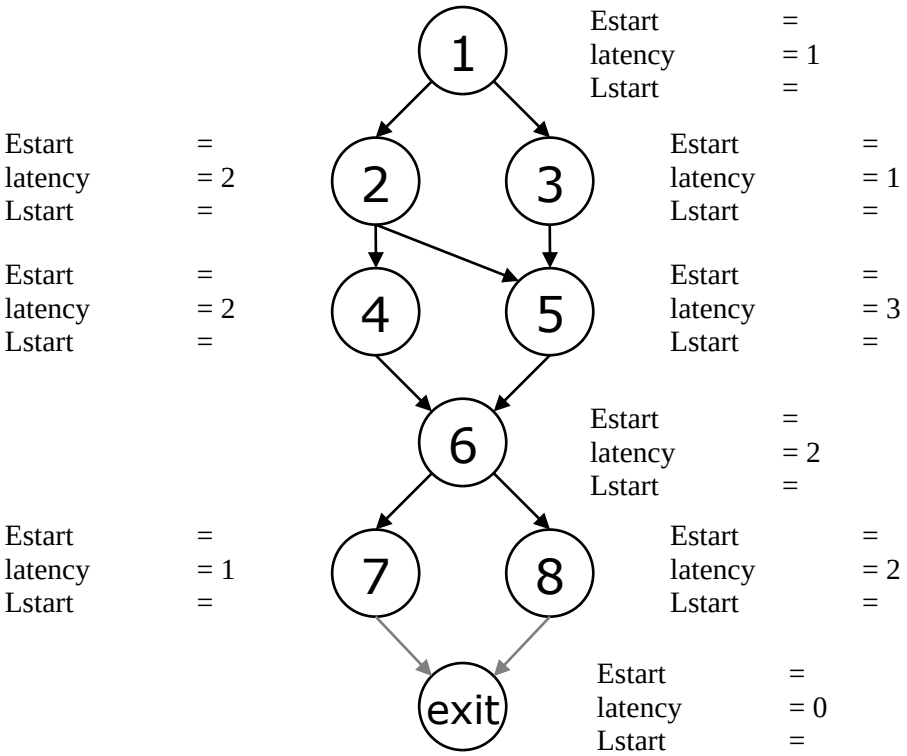
$$Lstart(op) = \begin{cases} Estart(op) & if\ op\ has\ no\ successors \\ \min\limits_{s \in succ(op)} Lstart(s) - latency(op) & otherwise \end{cases}$$

  - *slack*: scheduling freedom

$$slack(op) = Lstart(op) - Estart(op)$$

List Scheduling

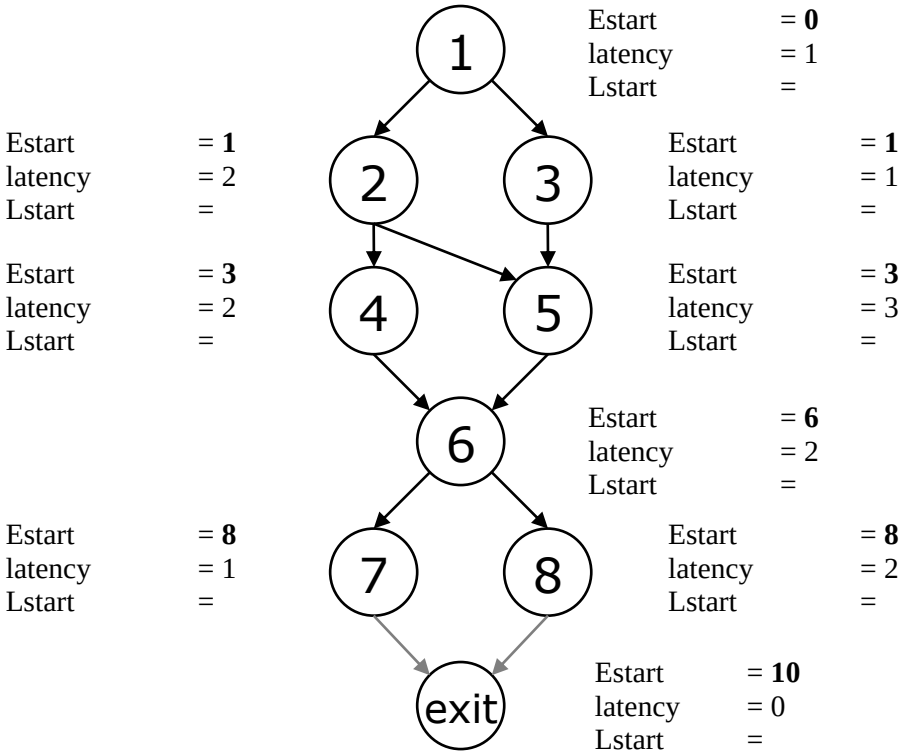
- Computing *Estart*, *Lstart*, *slack*



List Scheduling

- Computing ***Estart***, *Lstart*, *slack*

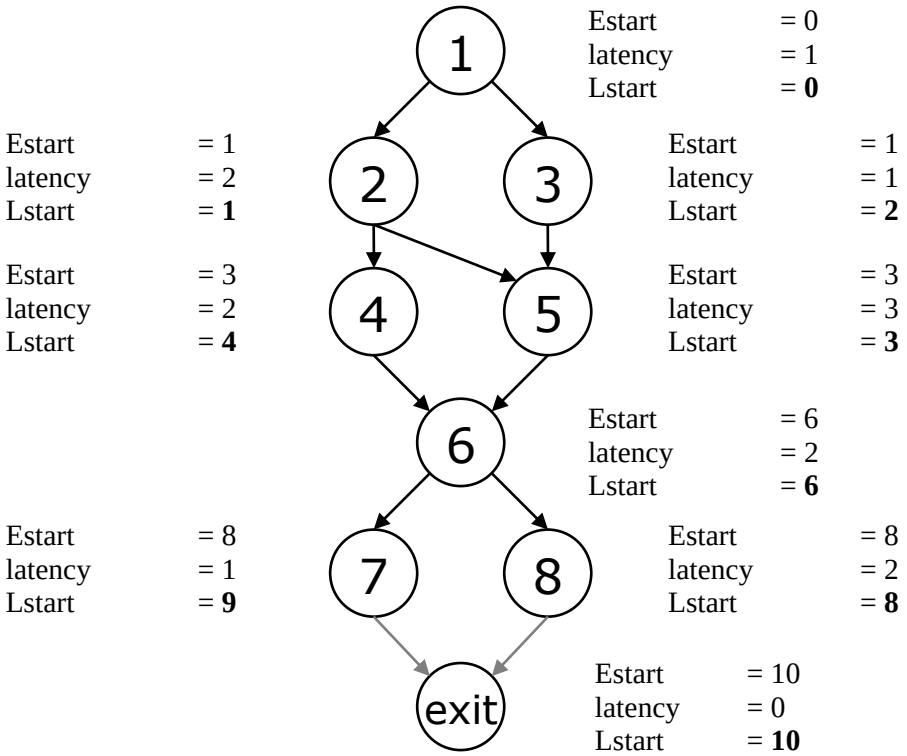
$$Estart(op) = \begin{cases} 0 & if\ op\ has\ no\ predecessors \\ \max\limits_{p \in pred(op)} Estart(p) + latency(p) & otherwise \end{cases}$$



List Scheduling

- Computing *Estart*, ***Lstart***, *slack*

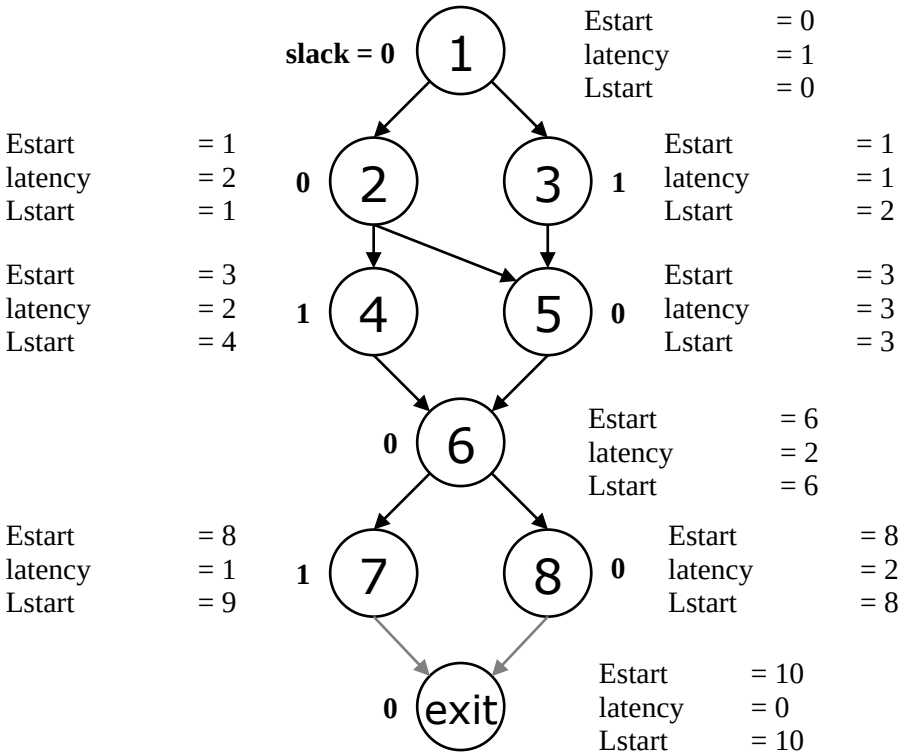
$$Lstart(op) = \begin{cases} Estart(op) & if\ op\ has\ no\ successors \\ \min_{s \in succ(op)} Lstart(s) - latency(op) & otherwise \end{cases}$$



List Scheduling

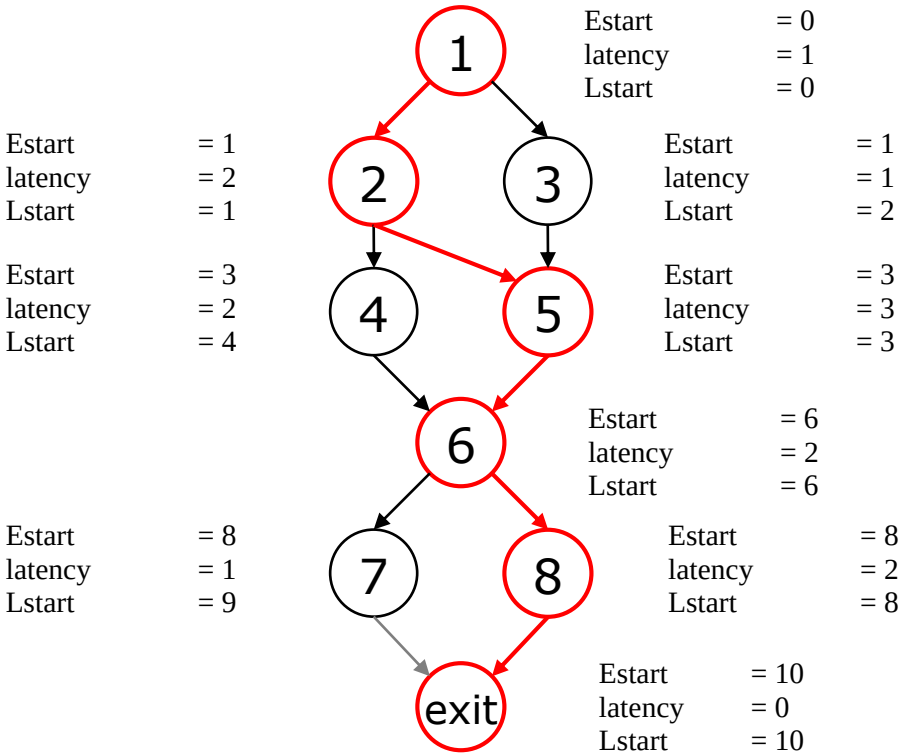
- Computing *Estart*, *Lstart*, ***slack***

$$slack(op) = Lstart(op) - Estart(op)$$



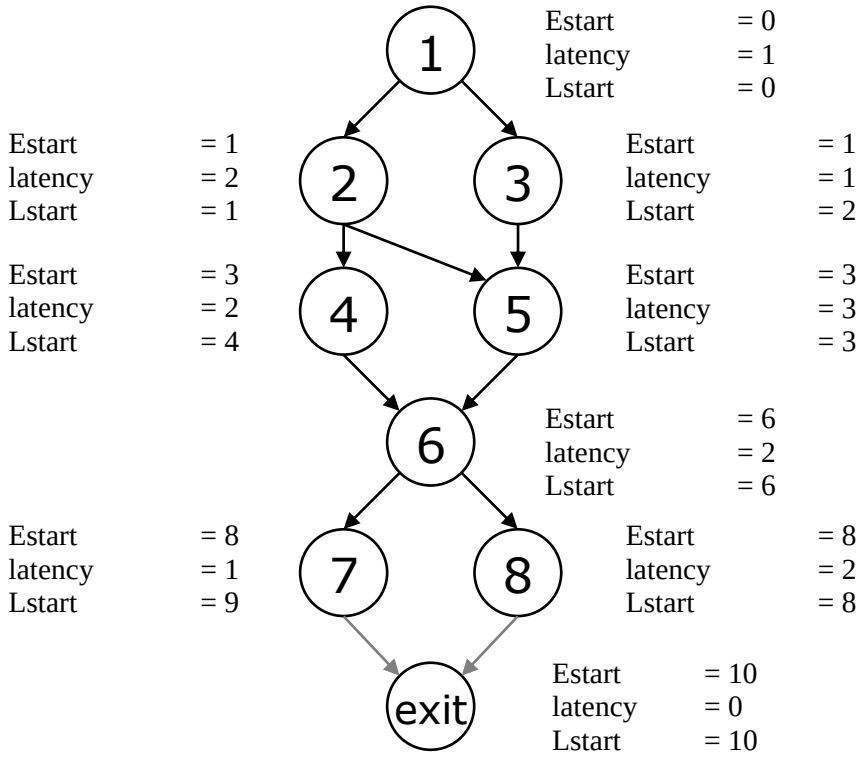
List Scheduling

- Another way to look at the critical path
  - sequence of critical operations
  - critical operation: *slack*(*op*) = 0



Priority Function: Height-Based

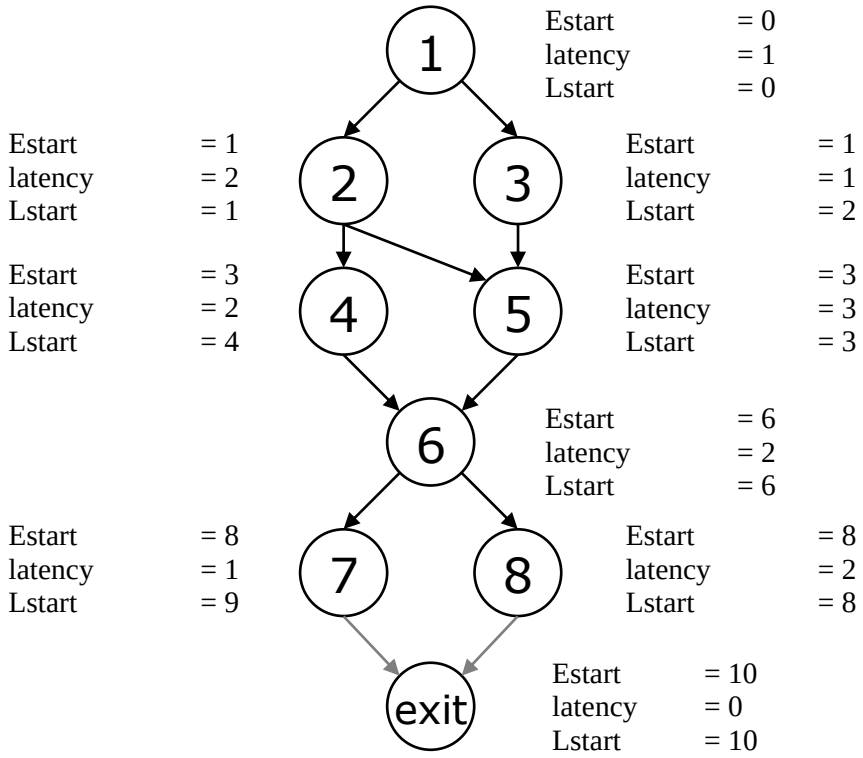
- Height-based priority function
  - gives priority to amount of work left
  - $priority(op) = Lstart(exit) - Lstart(op) + 1$



op	priority
1	
2	
3	
4	
5	
6	
7	
8	
exit	

Priority Function: Height-Based

- Height-based priority function
  - gives priority to amount of work left
  - $priority(op) = Lstart(exit) - Lstart(op) + 1$

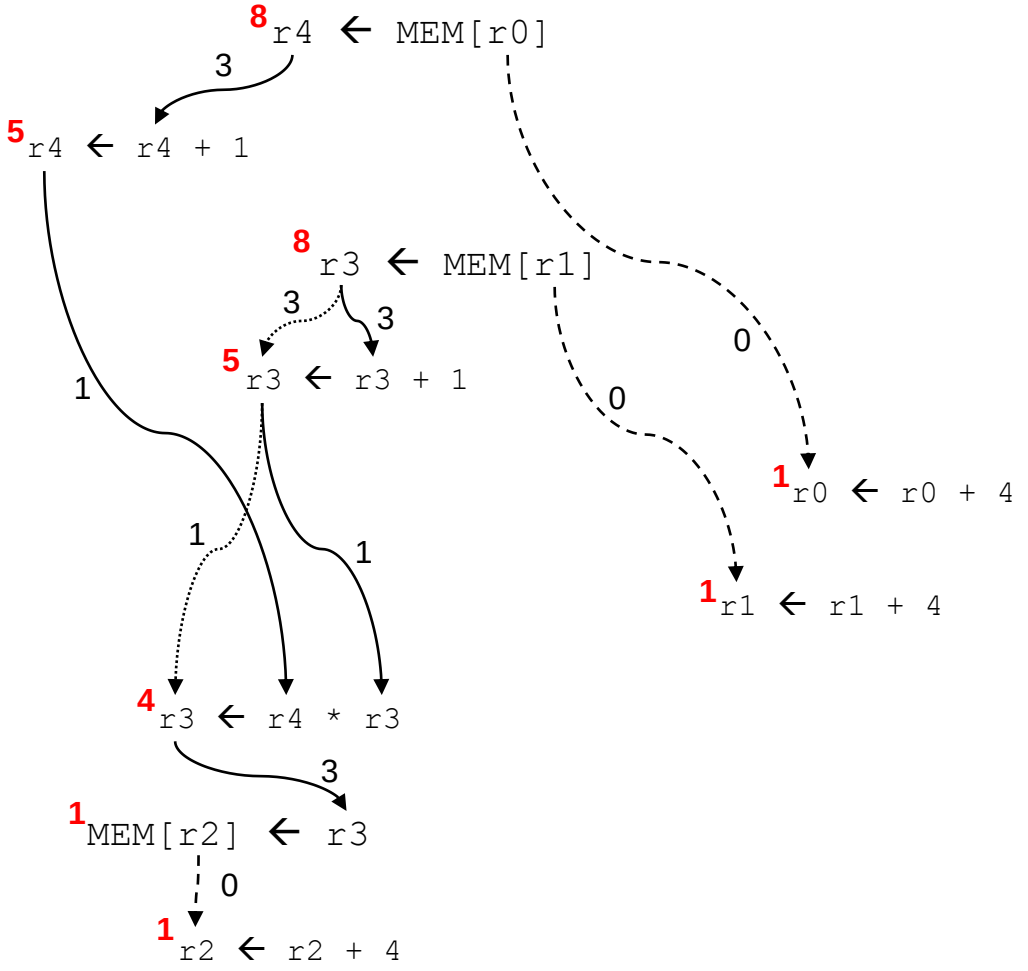


op	priority
1	11
2	10
3	9
4	7
5	8
6	5
7	2
8	3
exit	1

Example

- Applying height-based list scheduling to the previous example

```
1 load   r4 ← MEM[r0]
2 add    r4 ← r4, #1
3 load   r3 ← MEM[r1]
4 add    r3 ← r3, #1
5 add    r0 ← r0, #4
6 add    r1 ← r1, #4
7 mul    r3 ← r3, r4
8 store MEM[r2], r3
9 add    r2 ← r2, #4
```

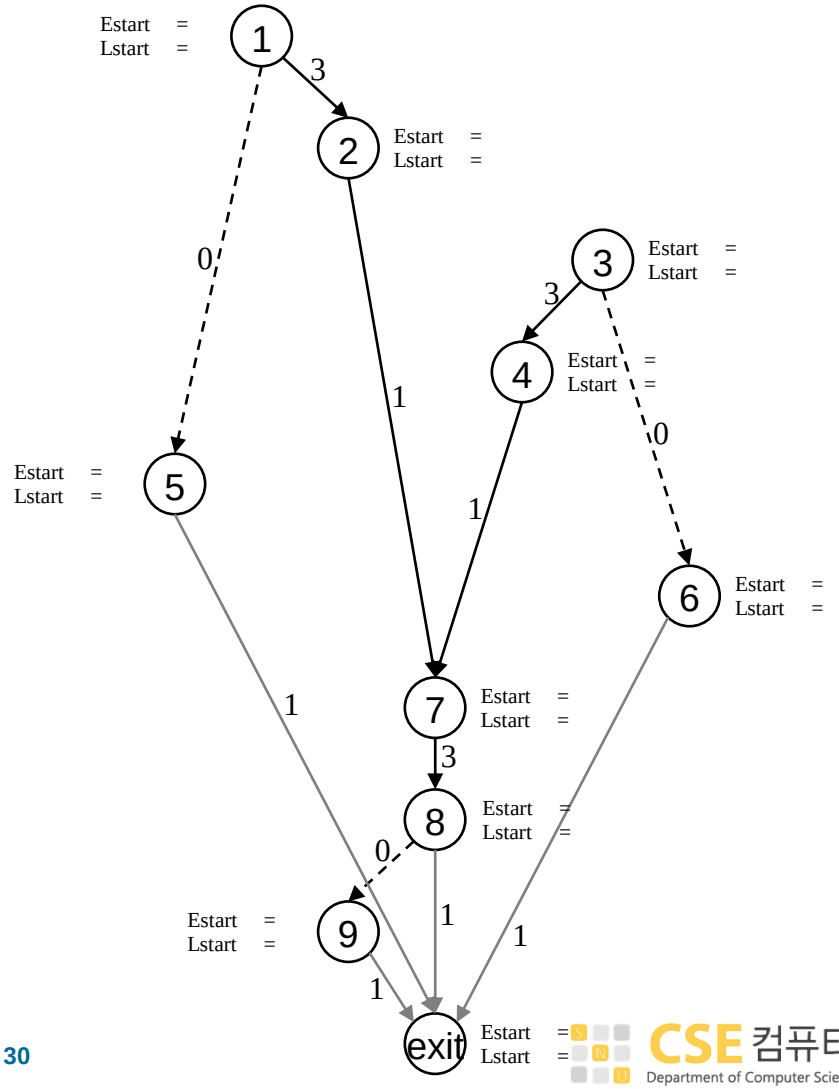


target machine properties
- pipelined with forwarding, single issue, in-order
- operation latencies: add, sub: 1 cycle; mul, load: 3 cycles; store: 1 cycle

Example

- Applying height-based list scheduling to the previous example

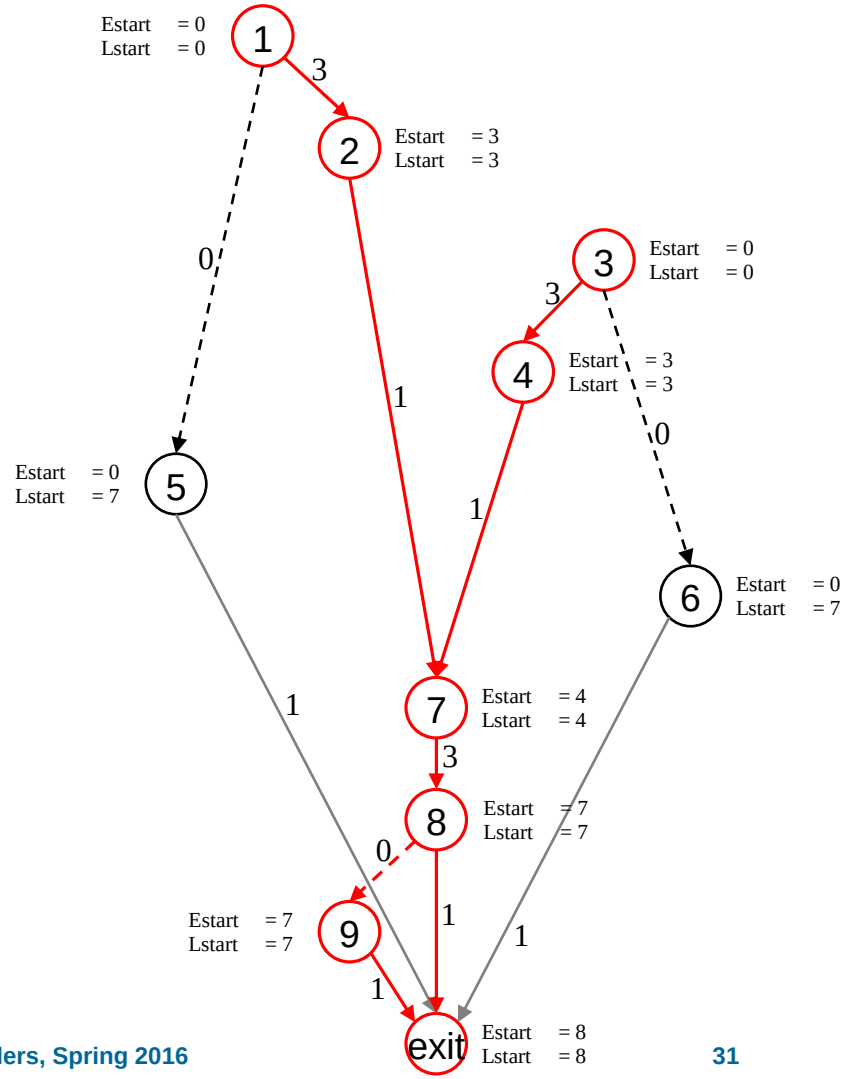
```
1 load  r4 ← MEM[r0]
2 add   r4 ← r4, #1
3 load  r3 ← MEM[r1]
4 add   r3 ← r3, #1
5 add   r0 ← r0, #4
6 add   r1 ← r1, #4
7 mul   r3 ← r3, r4
8 store MEM[r2], r3
9 add   r2 ← r2, #4
```



target machine properties
- pipelined with forwarding, single issue, in-order
- operation latencies: add, sub: 1 cycle;
  mul, load: 3 cycles; store: 1 cycle

Example

- Applying height-based list scheduling to the previous example



op	priority
1	9
3	9
2	6
4	6
7	5
5	2
6	2
8	2
9	2
exit	1

Example

op	priority
1	9
3	9
2	6
4	6
7	5
5	2
6	2
8	2
9	2
exit	1

- Applying height-based list scheduling to the previous example

Initialization

```
t := 1
ready := { 1, 3, 5, 6 }
active := {}
```

Iteration:

t = 1, ready = {1,3,5,6}, active = {}

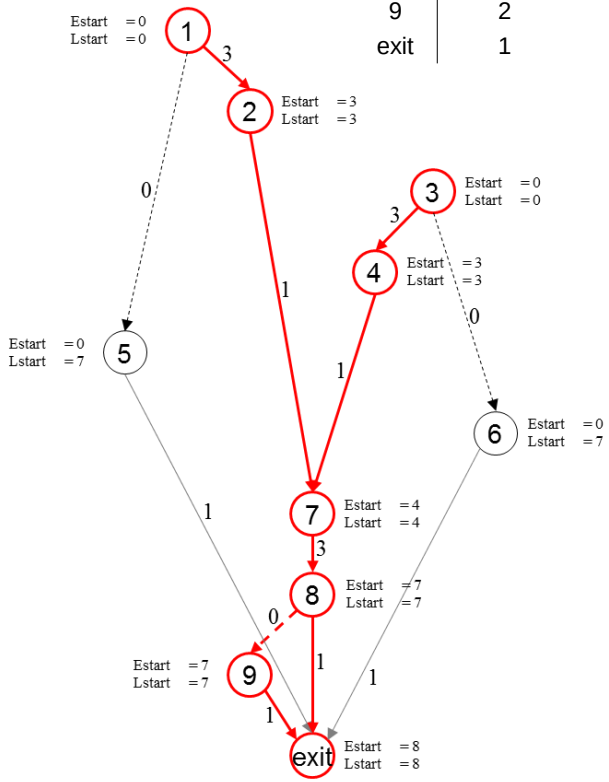
```
active is empty → pass

ready is not empty
  o := 1 (with priority 9)
  S(1) = 1
```

S	
t	op
1	1

t = 2, ready = {3,5,6}, active = {1}

```
active is not empty
  o = 1: S(1)+delay(1)≤t? no

ready is not empty
  o := 3 (with priority 9)
  ready := {5,6}, active = {1,3}
  S(3) = 2
```

S	
t	op
1	1
2	3



Example

op	priority
1	9
3	9
2	6
4	6
7	5
5	2
6	2
8	2
9	2
exit	1

- Applying height-based list scheduling to the previous example

Iteration:

t = 3, ready = {5,6}, active = {1,3}

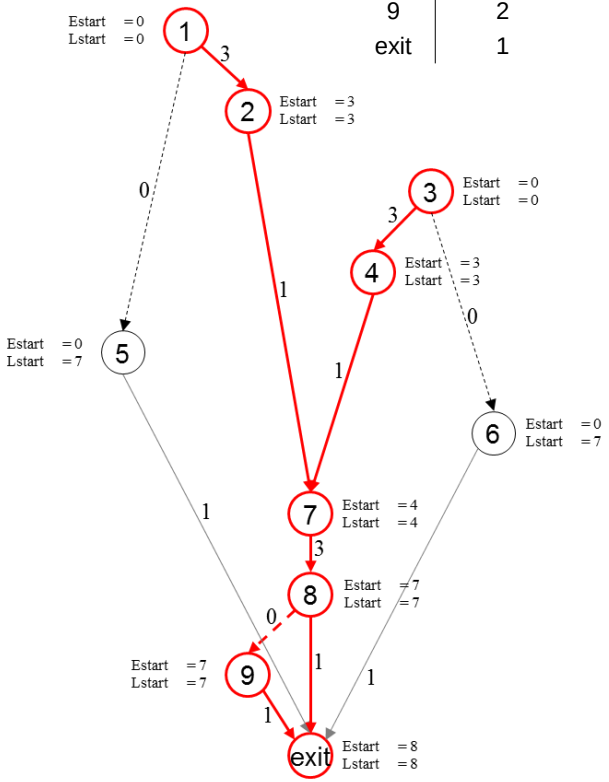
```
active is not empty
  o = 1: S(1)+delay(1)≤3? no
  o = 3: S(3)+delay(3)≤3? No

ready is not empty
  o := 5 (with priority 2)
  ready := {6}, active = {1,3,5}
  S(5) = 3
```

S	
t	***op***
1	1
2	3
3	5

t = 4, ready = {6}, active = {1,3,5}

```
active is not empty
  o = 1: S(1)+delay(1)≤4? yes
    active := active \ {1}
    ready := ready ∪ {2}
  o = 3: S(3)+delay(3)≤4? No

ready is not empty
  o := 2 (with priority 6)
  ready := {6}, active = {3,5,2}
  S(2) = 4
```

S	
t	***op***
1	1
2	3
3	5
4	2



Example

op	priority
1	9
3	9
2	6
4	6
7	5
5	2
6	2
8	2
9	2
exit	1

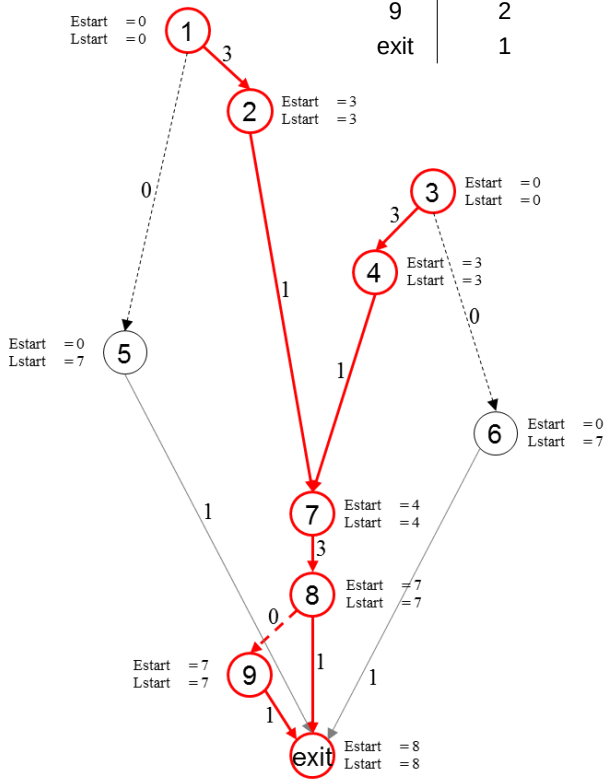
- Applying height-based list scheduling to the previous example

Iteration:

t = 5, ready = {6}, active = {3,5,2}

```
active is not empty
  o = 3: S(3)+delay(3)≤5? yes
    active := active \ {3}
    ready := ready ∪ {4}
  o = 5: S(5)+delay(5)≤4? yes
    active := active \ {5}
    no data dependent successors
  o = 2: S(2)+delay(2)≤4? yes
    active := active \ {2}
    successor (7) not ready due to 4

ready is not empty
  o := 4 (with priority 6)
  ready := {6}, active = {4}
  S(4) = 5
```

S	
t	***op***
1	1
2	3
3	5
4	2
5	4



Example

op	priority
1	9
3	9
2	6
4	6
7	5
5	2
6	2
8	2
9	2
exit	1

- Applying height-based list scheduling to the previous example

Iteration:

t = 6, ready = {6}, active = {4}

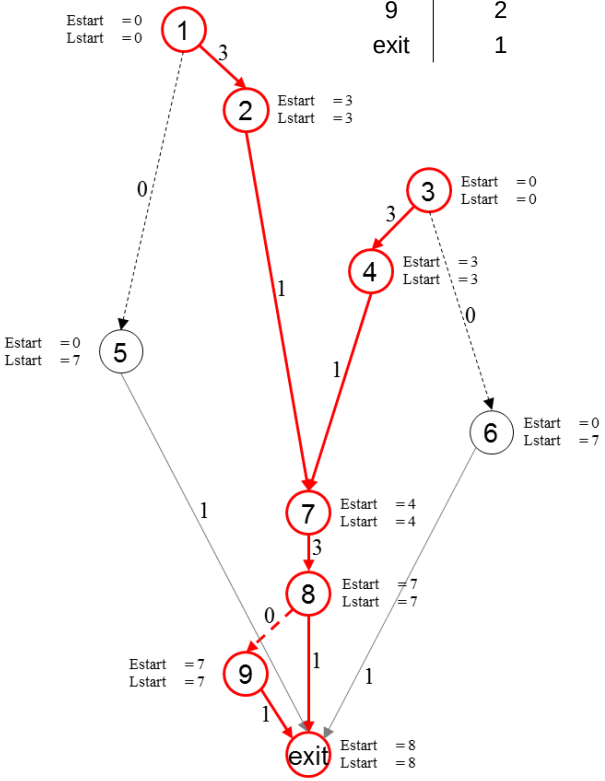
```
active is not empty
  o = 4: S(4)+delay(4)≤6? ne
    active := active \ {4}
    ready := ready ∪ {7}

ready is not empty
  o := 7 (with priority 5)
  ready := {6}, active = {7}
  S(7) = 6
```

S	
t	***op***
1	1
2	3
3	5
4	2
5	4
6	7

t = 7, ready = {6}, active = {7}

```
active is not empty
  o = 7: S(7)+delay(7)≤7? No

ready is not empty
  o := 6 (with priority 2)
  ready := {}, active = {6,7}
  S(6) = 7
```

S	
t	***op***
1	1
2	3
3	5
4	2
5	4
6	7
7	6



Example

op	priority
1	9
3	9
2	6
4	6
7	5
5	2
6	2
8	2
9	2
exit	1

- Applying height-based list scheduling to the previous example

Iteration:

t = 8, ready = {}, active = {6,7}

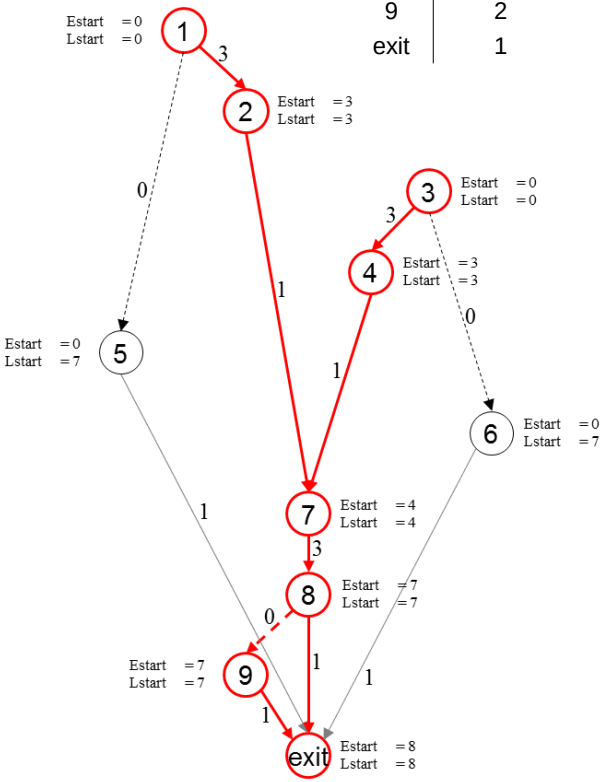
```
active is not empty
  o = 6: S(6)+delay(6)≤8? yes
    active := active \ {6}
    no data dependent successors
  o = 7: S(7)+delay(7)≤8? no

ready is empty
```

S	
t	*op*
1	1
2	3
3	5
4	2
5	4
6	7
7	6
8	

t = 9, ready = {}, active = {7}

```
active is not empty
  o = 7: S(7)+delay(7)≤9? yes
    active := active \ {7}
    ready := ready ∪ {8,9}

ready is not empty
  o := 8 (with priority 2)
  ready := {9}, active = {8}
  S(8) = 9
```

S	
t	*op*
1	1
2	3
3	5
4	2
5	4
6	7
7	6
8	
9	8



Example

- Applying height-based list scheduling to the previous example

Iteration:

t = 10, ready = {9}, active = {8}

```
  active is not empty
    o = 8: S(8)+delay(8)≤10? yes
      active := active \ {8}
      no data dependent successors

  ready is not empty
    o := 9 (with priority 2)
    ready := {}, active = {9}
    S(9) = 10
```

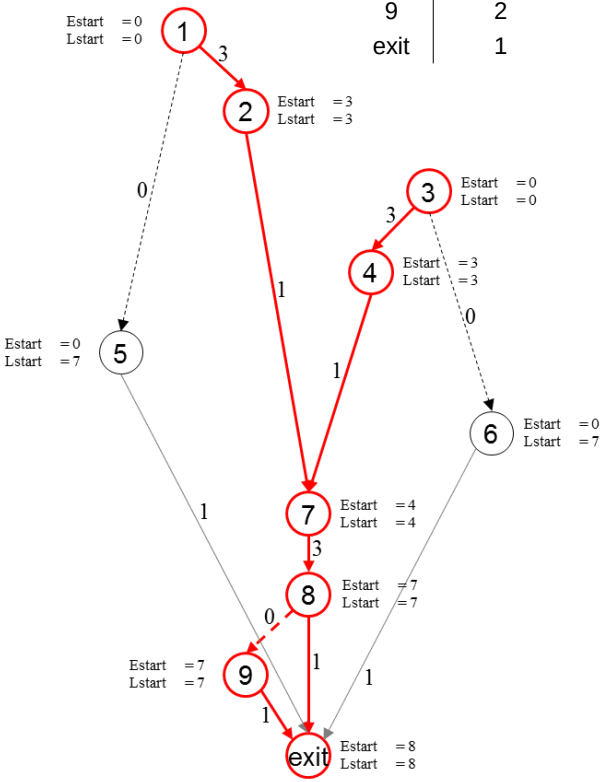
t = 11, ready = {}, active = {9}

```
  active is not empty
    o = 9: S(9)+delay(9)≤11? yes
      active := active \ {9}
      no data dependent successors

  ready is empty

done.
```

op	priority
1	9
3	9
2	6
4	6
7	5
5	2
6	2
8	2
9	2
exit	1

S	
t	***op***
1	1
2	3
3	5
4	2
5	4
6	7
7	6
8	
9	8
10	9

S	
t	***op***
1	1
2	3
3	5
4	2
5	4
6	7
7	6
8	
9	8
10	9



Instruction Scheduling & Register Allocation

- Phase ordering between instruction scheduling and register allocation
  - effects of the scheduler on the RA
    - the scheduler can use renaming to get rid of anti dependences to obtain more freedom in scheduling
    - the resulting overlap of previously constrained operations may increase register pressure
    - which, in turn, may force the register allocator to spill one more variable
  - and vice versa (the RA constrains the scheduler in a RA-first compiler)
- Combining scheduling and register allocation
  - has the potential to produce better solutions
  - typically not done due to complexity