RUTGERS
THE STATE UNIVERSITY OF NEW JERSEY

CS415 Compilers

Instruction Scheduling (part 3)

Announcements

- First homework has been posted; due Wednesday, February 9
- Recitation slides are available on our website
- First project will be on instruction scheduling

Scheduling Example

Operation	Cycles
load	3
loadI	1
loadAI	3
store	3
storeAI	3
add	1
mult	2
fadd	1
fmult	2
shift	1
output	1
outputAI	1

Build a simple local scheduler (basic block)

- non-blocking loads & stores
- different latencies load/store vs. arith. etc. operations
- different heuristics
- forward / backward scheduling

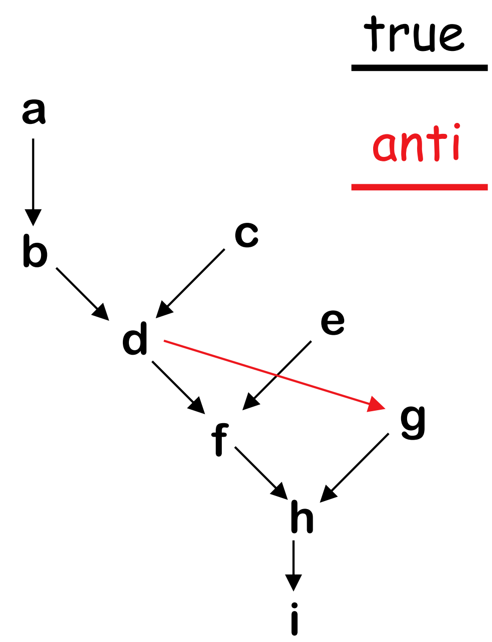
Scheduling Example

1. Build the dependence graph

S(n):				
0	a:	loadAI	r0,@w	⇒ r1
3	b:	add	r1,r1	⇒ r1
4	c:	loadAI	r0,@x	⇒ r2
7	d:	mult	r1,r2	⇒ r1
8	e:	loadAI	r0,@y	⇒ r3
11	f:	mult	r1,r3	⇒ r1
12	g:	loadAI	r0,@z	⇒ r2
15	h:	mult	r1,r2	⇒ r1
17	i:	storeAI	r1	⇒ r0,@w
20				

The Code

⇒ 20 cycles



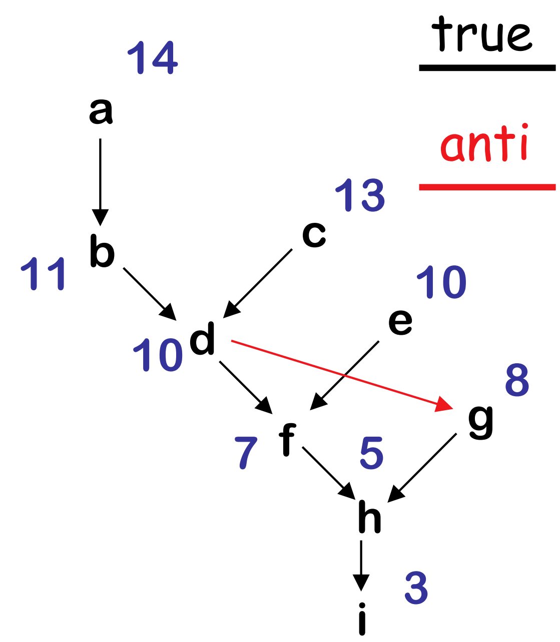
The Dependence Graph

Scheduling Example

1. Build the dependence graph
2. Determine priorities: longest latency-weighted path

```
a:  loadAI   r0,@w   ⇒ r1
b:  add      r1,r1   ⇒ r1
c:  loadAI   r0,@x   ⇒ r2
d:  mult     r1,r2   ⇒ r1
e:  loadAI   r0,@y   ⇒ r3
f:  mult     r1,r3   ⇒ r1
g:  loadAI   r0,@z   ⇒ r2
h:  mult     r1,r2   ⇒ r1
i:  storeAI  r1      ⇒ r0,@w
```

The Code



The Dependence Graph

List Scheduling Example

CYCLE = 0

The Code

READY - SET

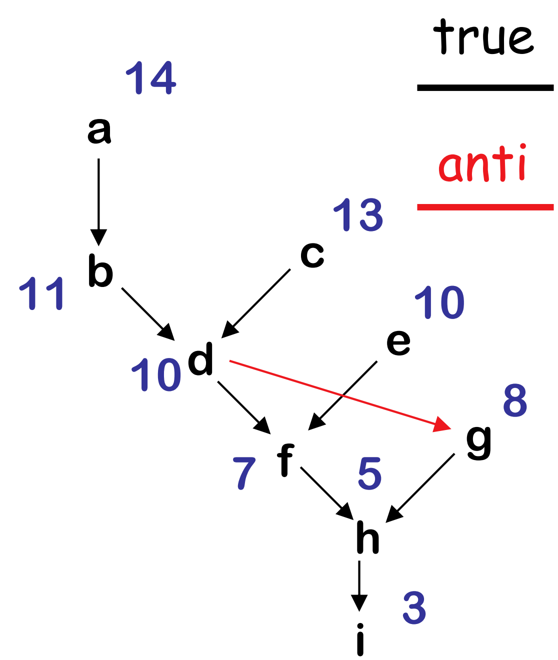
ACTIVE - SET

```
a: loadAI. r0, @w  => r1
b: add     r1, r1  => r1
c: loadAI  r0, @x  => r2
d: mult    r1, r2  => r1
e: loadAI  r0, @y  => r3
f: mult    r1, r3  => r1
g: loadAI  r0, @z  => r2
h: mult    r1, r2  => r1
i: storeAI r1  => r0,@w
```

S(n) =

The Generated Code

0
1
2
3
4
5
6
7
8
9
10
11
12



The Dependence Graph

(longest latency-weighted)

List Scheduling Example

The Code

CYCLE = 14

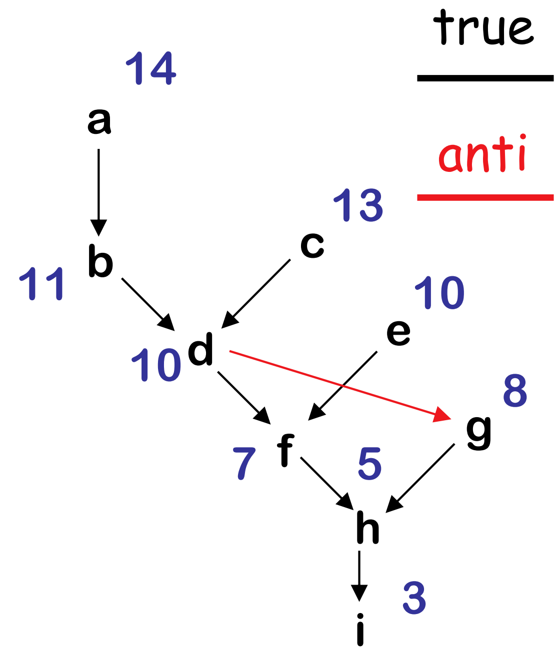
READY - SET

ACTIVE - SET

The Generated Code

0	a: loadAI. r0, @w => r1
1	c: loadAI r0, @x => r2
2	e: loadAI r0, @y => r3
3	b: add r1, r1 => r1
4	d: mult r1, r2 => r1
5	
6	g: loadAI r0, @z => r2
7	f: mult r1, r3 => r1
8	
9	h: mult r1, r2 => r1
10	
11	i: storeAI r1 => r0,@w
12	

S(n) =



The Dependence Graph

(longest latency-weighted)

Local (Forward) List Scheduling

```
Cycle ← 0
Ready ← leaves of P
Active ← Ø

while (Ready ∪ Active ≠ Ø)
   if (Ready ≠ Ø) then
      remove an op from Ready
      S(op) ← Cycle
      Active ← Active ∪ op

   Cycle ← Cycle + 1

   for each op ∈ Active
       if (S(op) + delay(op) ≤ Cycle) then
          remove op from Active
          for each successor s of op in P
              if (s is ready) then
                 Ready ← Ready ∪ s
```

Removal in priority order

op has completed execution

If successor's operands are ready, put it on Ready

Instruction Scheduling (Definitions)

A correct schedule S maps each $n \in N$ into a non-negative integer representing its cycle number such that

1. $S(n) \geq 0$, for all $n \in N$, obviously
2. If $(n_1, n_2) \in E$, $S(n_1) + delay(n_1) \leq S(n_2)$
3. For each type t, there are no more operations of type t in any cycle than the target machine can issue;

   (Note: we only use a single type here – single pipeline)

The length of a schedule S, denoted $L(S)$, is

$$L(S) = max_{n \in N} (S(n) + delay(n))$$

The goal is to find the shortest possible correct schedule.
S is time-optimal if $L(S) \leq L(S_1)$, for all other schedules S_1

Note: We are trying to minimize execution time here.

RUTGERS

Instruction Scheduling (What's so difficult?)

Critical Points

- All operands must be available
- Multiple operations can be *ready*
- Operands can have multiple predecessors

Together, these issues make scheduling *hard* (NP-Complete)

Local scheduling is the simple case

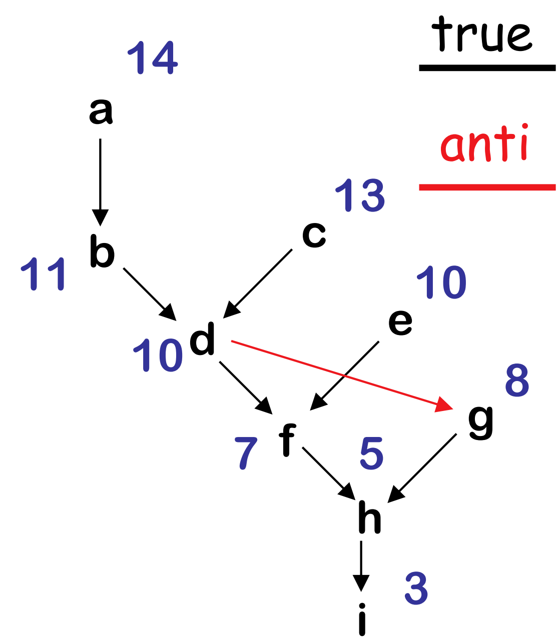
- Restricted to straight-line code (single basic block)
- Consistent and predictable latencies

Scheduling Example

1. Build the dependence graph
2. Determine priorities: longest latency-weighted path

a:	loadAI	r0,@w	⇒ r1
b:	add	r1,r1	⇒ r1
c:	loadAI	r0,@x	⇒ r2
d:	mult	r1,r2	⇒ r1
e:	loadAI	r0,@y	⇒ r3
f:	mult	r1,r3	⇒ r1
g:	loadAI	r0,@z	⇒ r2
h:	mult	r1,r2	⇒ r1
i:	storeAI	r1	⇒ r0,@w

The Code



The Dependence Graph

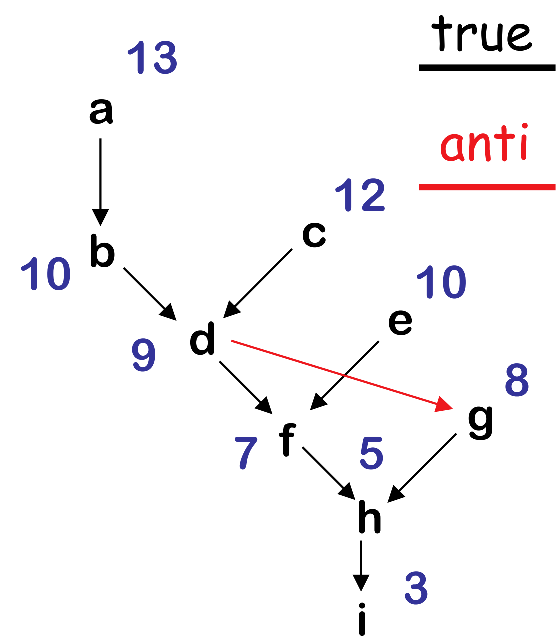
Note: Here we assume that an operation has to finish to satisfy an anti dependence.
Our ILOC simulator **takes only one cycle to satisfy an anti dependence** since read-stage is executed before write stage (EaC).

Scheduling Example

1. Build the dependence graph
2. Determine priorities: longest latency-weighted path

a:	loadAI	r0,@w	⇒ r1
b:	add	r1,r1	⇒ r1
c:	loadAI	r0,@x	⇒ r2
d:	mult	r1,r2	⇒ r1
e:	loadAI	r0,@y	⇒ r3
f:	mult	r1,r3	⇒ r1
g:	loadAI	r0,@z	⇒ r2
h:	mult	r1,r2	⇒ r1
i:	storeAI	r1	⇒ r0,@w

The Code



The Dependence Graph

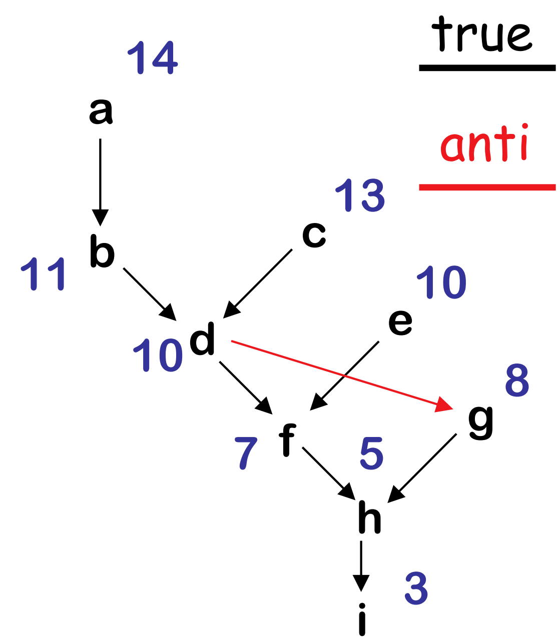
Note: Here we assume that an operation has to finish to satisfy an anti dependence.
Our ILOC simulator **takes only one cycle to satisfy an anti dependence** since read-stage is executed before write stage (EaC).

Scheduling Example

1. Build the dependence graph
2. Determine priorities: longest latency-weighted path
3. Perform list scheduling (forward)

a:	loadAI	r0,@w	⇒ r1
b:	add	r1,r1	⇒ r1
c:	loadAI	r0,@x	⇒ r2
d:	mult	r1,r2	⇒ r1
e:	loadAI	r0,@y	⇒ r3
f:	mult	r1,r3	⇒ r1
g:	loadAI	r0,@z	⇒ r2
h:	mult	r1,r2	⇒ r1
i:	storeAI	r1	⇒ r0,@w

The Code



The Dependence Graph

We assume full latency for anti-dependences here

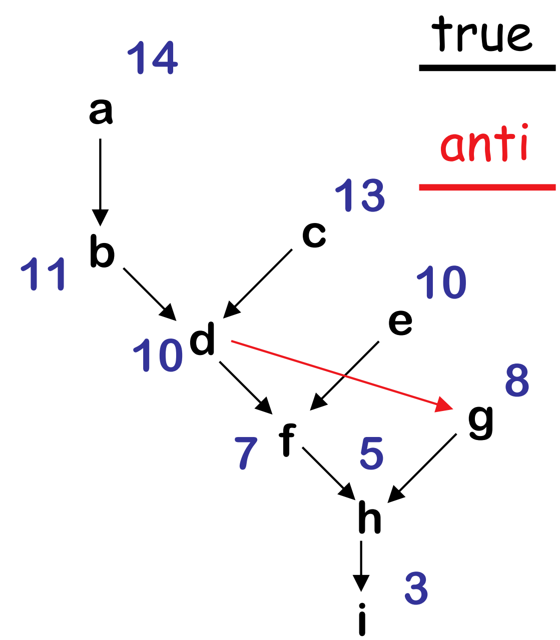
RUTGERS

Scheduling Example

1. Build the dependence graph
2. Determine priorities: longest latency-weighted path
3. Perform list scheduling (forward)

S(n):				
0	a:	loadAI	r0,@w	⇒ r1
1	c:	loadAI	r0,@x	⇒ r2
2	e:	loadAI	r0,@y	⇒ r3
3	b:	add	r1,r1	⇒ r1
4	d:	mult	r1,r2	⇒ r1
6	g:	loadAI	r0,@z	⇒ r2
7	f:	mult	r1,r3	⇒ r1
9	h:	mult	r1,r2	⇒ r1
11	i:	storeAI	r1	⇒ r0,@w
14				

The Code



The Dependence Graph

⇒ **14 cycles**

We assume full latency for anti-dependences here

More on Scheduling

Forward list scheduling
- start with available ops
- work forward
- ready ⇒ all operands available

Backward list scheduling
- start with no successors
- work backward
- ready ⇒ latency covers operands

Different heuristics (forward) based on Dependence Graph
1. Longest latency weighted path to root (⇒ critical path)
2. Highest latency instructions (⇒ more overlap)
3. Most immediate successors (⇒ create more candidates)
4. Most descendents (⇒ create more candidates)
5. …

Interactions with register allocation (Note: we are not doing this)
- perform dynamic register renaming (⇒ may require spill code)
- move life ranges around (⇒ may remove or require spill code)
- …

Next topic

Register Allocation EaC 13.1 – 13.3
(Top-down and Bottom-Up Allocation)