Local Instruction Scheduling
— A Primer for Lab 3 —

COMP 412
Fall 2005

What Makes Code Run Fast?

- Many operations have non-zero latencies
- Modern machines can issue several operations per cycle
- Execution time is *order-dependent (and has been since the 60's)*

Assumed latencies *(conservative)*

Operation	Cycles
load	3
store	3
loadI	1
add	1
mult	2
fadd	1
fmult	2
shift	1
branch	0 to 8

- Loads & stores may or may not block
  - > Non-blocking ⇒fill those issue slots
- Branch costs vary with path taken
- Branches typically have delay slots
  - > Fill slots with unrelated operations
  - > Percolates branch upward
- Scheduler should hide the latencies

Lab 3 will build a local scheduler

Example

w ← w * 2 * x * y * z

Simple schedule

1	loadAI	r0,@w	⇒ r1
4	add	r1,r1	⇒ r1
5	loadAI	r0,@x	⇒ r2
8	mult	r1,r2	⇒ r1
9	loadAI	r0,@y	⇒ r2
12	mult	r1,r2	⇒ r1
13	loadAI	r0,@z	⇒ r2
16	mult	r1,r2	⇒ r1
18	storeAI	r1	⇒ r0,@w
21	r1 is free		

2 registers, 20 cycles

Schedule loads early

1	loadAI	r0,@w	⇒ r1
2	loadAI	r0,@x	⇒ r2
3	loadAI	r0,@y	⇒ r3
4	add	r1,r1	⇒ r1
5	mult	r1,r2	⇒ r1
6	loadAI	r0,@z	⇒ r2
7	mult	r1,r3	⇒ r1
9	mult	r1,r2	⇒ r1
11	storeAI	r1	⇒ r0,@w
14	r1 is free		

3 registers, 13 cycles

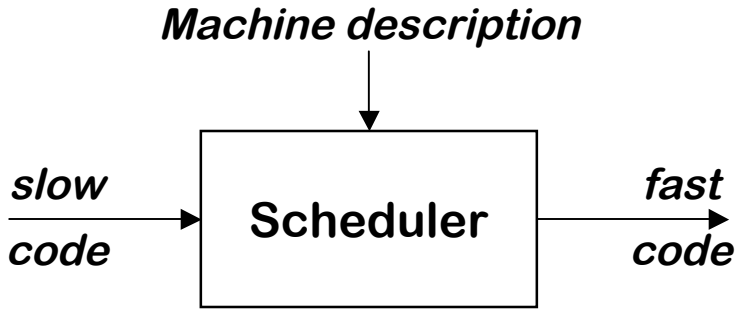
Reordering operations for speed is called *instruction scheduling*

Instruction Scheduling (Engineer's View)

The Problem

Given a code fragment for some target machine and the latencies for each individual operation, reorder the operations to minimize execution time

The Concept



The Task

- Produce correct code
- Minimize wasted cycles
- Avoid spilling registers
- Operate efficiently

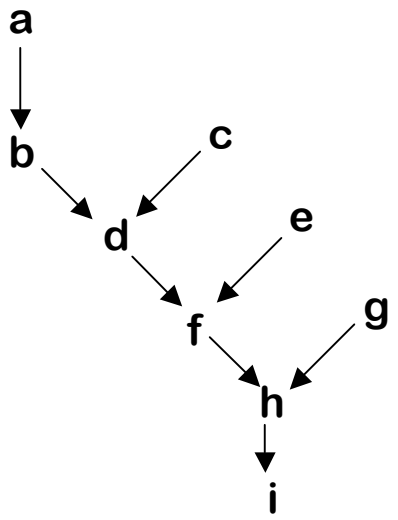
Instruction Scheduling (The Abstract View)

To capture properties of the code, build a precedence graph *G*

- Nodes $n \in G$ are operations with *type(n)* and *delay(n)*
- An edge $e = (n_1,n_2) \in G$ if & only if n_2 uses the result of n_1

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

The Code



The Precedence Graph

Instruction Scheduling (Definitions)

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

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

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

A schedule might also be optimal in terms of registers, power, or space….

Instruction Scheduling (What's so difficult?)

Critical Points

- All operands must be available
- Multiple operations can be *ready*
- Moving operations can lengthen register lifetimes
- Placing uses near definitions can shorten register lifetimes
- Operands can have multiple predecessors

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

Local scheduling is the simple case

- Restricted to straight-line code
- Consistent and predictable latencies

Instruction Scheduling: The Big Picture

1. Build a precedence graph, *P*
2. Compute a *priority function* over the nodes in *P*
3. Use list scheduling to construct a schedule, one cycle at a time
    a. Use a queue of operations that are ready
    b. At each cycle
        I. Choose the highest priority ready operation and schedule it
        II. Update the ready queue

Local list scheduling

- The dominant algorithm for twenty years
- A greedy, heuristic, local technique

Local List Scheduling

```
Cycle ← 1
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

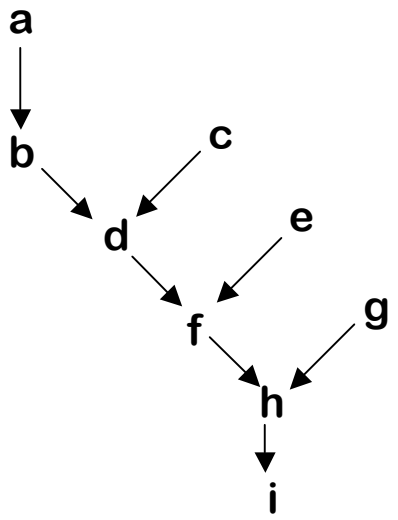
If successor's operands are ready, put it on Ready

Scheduling Example

1. Build the precedence graph

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

The Code



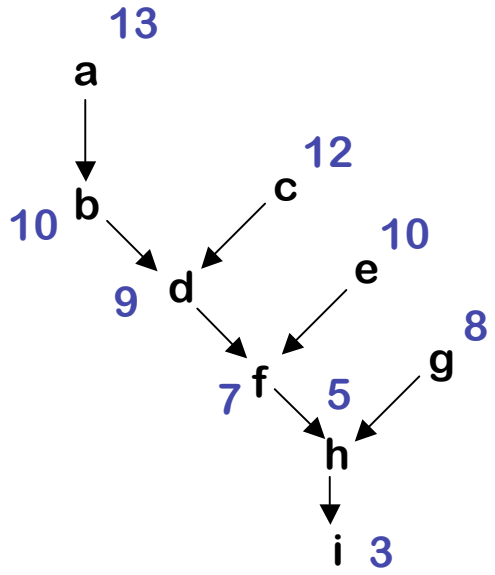
The Precedence Graph

Scheduling Example

1. Build the precedence 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   ⇒ r2
f:   mult      r1,r2   ⇒ r1
g:   loadAI    r0,@z   ⇒ r2
h:   mult      r1,r2   ⇒ r1
i:   storeAI   r1      ⇒ r0,@w
```

The Code



The Precedence Graph

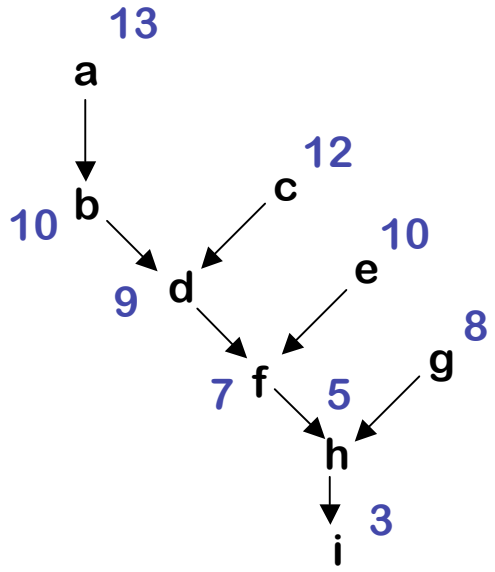
Scheduling Example

1. Build the precedence graph
2. Determine priorities: longest latency-weighted path
3. Perform list scheduling

New register name used

```
1)  a:  loadAI    r0,@w   ⇒ r1
2)  c:  loadAI    r0,@x   ⇒ r2
3)  e:  loadAI    r0,@y   ⇒ r3
4)  b:  add       r1,r1   ⇒ r1
5)  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
```

The Code



The Precedence Graph

Detailed Scheduling Algorithm I

Idea: Keep a collection of worklists W[c], one per cycle

- We need MaxC = max delay + 1 such worklists

Code:

```
for each n ∈ N do begin count[n] := 0; earliest[n] = 0 end
for each (n1,n2) ∈ E do begin
    count[n2] := count[n2] + 1;
    successors[n1] := successors[n1] ∪ {n2};
end
for i := 0 to MaxC – 1 do W[i] := ∅;
Wcount := 0;
for each n ∈ N do
    if count[n] = 0 then begin
        W[0] := W[0] ∪ {n}; Wcount := Wcount + 1;
    end
c := 0;  // c is the cycle number
cW := 0;// cW is the number of the worklist for cycle c
instr[c] := ∅;
```

Detailed Scheduling Algorithm II

```
while Wcount > 0 do begin
    while W[cW] = ∅ do begin
        c := c + 1; instr[c] := ∅; cW := mod(cW+1,MaxC);
    end
    nextc := mod(c+1,MaxC);
    while W[cW] ≠ ∅ do begin
Priority → select and remove an arbitrary instruction x from W[cW];
        if ∃ free issue units of type(x) on cycle c then begin
            instr[c] := instr[c] ∪ {x}; Wcount := Wcount - 1;
            for each y ∈ successors[x] do begin
                count[y] := count[y] – 1;
                earliest[y] := max(earliest[y], c+delay(x));
                if count[y] = 0 then begin
                    loc := mod(earliest[y],MaxC);
                    W[loc] := W[loc] ∪ {y}; Wcount := Wcount + 1;
                end
            end
        else W[nextc] := W[nextc] ∪ {x};
    end
end
```

More List Scheduling

List scheduling breaks down into two distinct classes

Forward list scheduling	Backward list scheduling
• Start with available operations	• Start with no successors
• Work forward in time	• Work backward in time
• Ready ⇒ all operands available	• Ready ⇒ latency covers uses

Variations on list scheduling

- Prioritize critical path(s)
- Schedule last use as soon as possible
- Depth first in precedence graph (minimize registers)
- Breadth first in precedence graph (minimize interlocks)
- Prefer operation with most successors

Lab 3

- Implement two schedulers for basic blocks in ILOC
- One must be list scheduling with specified priority
- Other can be a second priority, or a different algorithm
- Same ILOC subset as in lab 1 (plus NOP)
- Different execution model
  - Two asymmetric functional units
  - Latencies different from lab 1
  - Simulator different from lab 1

Lab 3 — Specific questions

1. *Are we in over our heads?* No. The hard part of this lab should be trying to get good code. The programming is not bad; you can reuse some stuff from lab 1. You have all the specifications & tools you need. Jump in and start programming.

2. *How many registers can we use?* Assume that you have as many registers as you need. If the simulator runs out, we'll get more. Rename registers to avoid false dependences & conflicts.

3. *What about a store followed by a load?* If you can show that the two operations must refer to different memory locations, the scheduler can overlap their execution. Otherwise, the store must complete before the load issues.

4. *What about test files?* We will put some test files online on OwlNet. We will test your lab on files you do not see. We have had problems (in the past) with people optimizing their labs for the test data. (Not that you would do that!)

Lab 3 - Hints

- Begin by renaming registers to eliminate false dependencies
- Pay attention to loads and stores
  - When can they be reordered?
- Understand the simulator
- Inserting NOPs may be necessary to get correct code
- Tiebreakers can make a difference