

# CSE 582 – Compilers

## Instruction Scheduling

Hal Perkins  
Autumn 2002

11/21/2002 © 2002 Hal Perkins & UW CSE O-1

# Agenda

- Instruction scheduling issues – latencies
- List scheduling

Credits: Adapted from slides by Keith Cooper, Rice University

11/21/2002 © 2002 Hal Perkins & UW CSE O-2

# Issues

- Many operations have non-zero latencies
- Modern machines can issue several operations per cycle
- Loads & Stores may or may not block
  - ∴ may be slots after load/store for other work
- Branch costs vary
- Branches on modern processors typically have delay slots
- GOAL: Scheduler should reorder instructions to hide latencies and take advantage of delay slots

11/21/2002 © 2002 Hal Perkins & UW CSE O-3

# Some Idealized Latencies

Operation	Cycles
LOAD	3
STORE	3
ADD	1
MULT	2
SHIFT	1
BRANCH	0 TO 8

11/21/2002 © 2002 Hal Perkins & UW CSE O-4

# Example: $w = w * 2 * x * y * z;$

<ul style="list-style-type: none"> <li>Simple schedule</li> </ul> <pre> 1 LOAD r1 &lt;- w 4 ADD r1 &lt;- r1,r1 5 LOAD r2 &lt;- x 8 MULT r1 &lt;- r1,r2 9 LOAD r2 &lt;- y 12 MULT r1 &lt;- r1,r2 13 LOAD r2 &lt;- z 16 MULT r1 &lt;- r1,r2 18 STORE w &lt;- r1 21 r1 free           </pre> <p>2 registers, 20 cycles</p>	<ul style="list-style-type: none"> <li>Loads early</li> </ul> <pre> 1 LOAD r1 &lt;- w 2 LOAD r2 &lt;- x 3 LOAD r3 &lt;- y 4 ADD r1 &lt;- r1,r1 5 MULT r1 &lt;- r1,r2 6 LOAD r2 &lt;- z 7 MULT r1 &lt;- r1,r3 9 MULT r1 &lt;- r1,r2 11 STORE w &lt;- r1 14 r1 is free           </pre> <p>3 registers, 13 cycles</p>
---	---

11/21/2002 © 2002 Hal Perkins & UW CSE O-5

# Instruction Scheduling

- Problem
  - Given a code fragment for some machine and latencies for each operation, reorder to minimize execution time
- Constraints
  - Produce correct code
  - Minimize wasted cycles
  - Avoid spilling registers
  - Do this efficiently

11/21/2002 © 2002 Hal Perkins & UW CSE O-6

## Precedence Graph

- Nodes  $n$  are operations
- Attributes of each node
  - type – kind of operation
  - delay – latency
- If node  $n_2$  uses the result of node  $n_1$ , there is an edge  $e = (n_1, n_2)$  in the graph

11/21/2002 © 2002 Hal Perkins & UW CSE O-7

## Example Graph

- Code
  - a LOAD  $r1 \leftarrow w$
  - b ADD  $r1 \leftarrow r1, r1$
  - c LOAD  $r2 \leftarrow x$
  - d MULT  $r1 \leftarrow r1, r2$
  - e LOAD  $r2 \leftarrow y$
  - f MULT  $r1 \leftarrow r1, r2$
  - g LOAD  $r2 \leftarrow z$
  - h MULT  $r1 \leftarrow r1, r2$
  - i STORE  $w \leftarrow r1$

11/21/2002 © 2002 Hal Perkins & UW CSE O-8

## Schedules (1)

- A correct schedule  $S$  maps each node  $n$  into a non-negative integer representing its cycle number, and
  - $S(n) \geq 0$  for all nodes  $n$  (obvious)
  - If  $(n_1, n_2)$  is an edge, then  $S(n_1) + \text{delay}(n_1) \leq S(n_2)$
  - For each type  $t$  there are no more operations of type  $t$  in any cycle than the target machine can issue

11/21/2002 © 2002 Hal Perkins & UW CSE O-9

## Schedules (2)

- The *length* of a schedule  $S$ , denoted  $L(S)$  is
 
$$L(S) = \max_n (S(n) + \text{delay}(n))$$
- The goal is to find the shortest possible correct schedule
  - Other possible goals: minimize use of registers, power, space, ...

11/21/2002 © 2002 Hal Perkins & UW CSE O-10

## Constraints

- Main points
  - All operands must be available
  - Multiple operations can be ready at any given point
  - Moving operations can lengthen register lifetimes
  - Moving uses near definitions can shorten register lifetimes
  - Operations can have multiple predecessors
- Collectively this makes scheduling NP-complete
- Local scheduling is the simpler case
  - Straight-line code
  - Consistent, predictable latencies

11/21/2002 © 2002 Hal Perkins & UW CSE O-11

## Algorithm Overview

- Build a precedence graph  $P$
- Compute a *priority function* over the nodes in  $P$  (typical: longest latency-weighted path)
- Use list scheduling to construct a schedule, one cycle at a time
  - Use queue of operations that are ready
  - At each cycle
    - Chose a ready operation and schedule it
    - Update ready queue
- Rename registers to avoid false dependencies and conflicts

11/21/2002 © 2002 Hal Perkins & UW CSE O-12

## List Scheduling Algorithm

```
Cycle = 1; Ready = leaves of P; Active = empty;
while (Ready and/or Active are not empty)
  if (Ready is not empty)
    remove an op from Ready;
    S(op) = Cycle;
    Active = Active + op;
  Cycle++;
  for each op in Active
    if (S(op) + delay(op) <= Cycle)
      remove op from Active;
      for each successor s of op in P
        if (s is ready - i.e., all operands available)
          add s to Ready
```

11/21/2002

© 2002 Hal Perkins & UW CSE

O-13

## Example

```
n Code
a LOAD r1 <- w
b ADD r1 <- r1,r1
c LOAD r2 <- x
d MULT r1 <- r1,r2
e LOAD r2 <- y
f MULT r1 <- r1,r2
g LOAD r2 <- z
h MULT r1 <- r1,r2
i STORE w <- r1
```

11/21/2002

© 2002 Hal Perkins & UW CSE

O-14

## Variations

- n Backward list scheduling
  - n Work from the root to the leaves
  - n Schedules instructions from end to beginning of the block
- n In practice, try both and pick the result that minimizes costs
  - n Little extra expense since the precedence graph and other information can be reused
- n Global scheduling and loop scheduling
  - n Extend basic idea in more aggressive compilers

11/21/2002

© 2002 Hal Perkins & UW CSE

O-15