Instruction Scheduling

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)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>load</td>
<td>3</td>
</tr>
<tr>
<td>store</td>
<td>3</td>
</tr>
<tr>
<td>loadI</td>
<td>1</td>
</tr>
<tr>
<td>add</td>
<td>1</td>
</tr>
<tr>
<td>mult</td>
<td>2</td>
</tr>
<tr>
<td>fadd</td>
<td>1</td>
</tr>
<tr>
<td>fmult</td>
<td>2</td>
</tr>
<tr>
<td>shift</td>
<td>1</td>
</tr>
<tr>
<td>branch</td>
<td>0 to 8</td>
</tr>
</tbody>
</table>

• 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

\[ w \leftarrow w \ast 2 \ast x \ast y \ast z \]

1. loadAI r0, @w
2. store r1, @w
3. add r1, r1
4. loadAI r0, @x
5. mult r1, r1
6. loadAI r0, @y
7. mult r1, r1
8. loadAI r0, @z
9. mult r1, r1
10. storeAI r1, @w
11. r1 is free
12. loadAI r0, @w
13. loadAI r0, @x
14. loadAI r0, @y
15. add r1, r1
16. mult r1, r2
17. loadAI r0, @z
18. mult r1, r2
19. storeAI r1, @w
20. r1 is free
21. r1 is free

2 registers, 20 cycles
3 registers, 13 cycles

Reordering operations for speed is called instruction scheduling

Instruction Scheduling

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
Scheduler
  slow
  fast
Machine description
  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 ∈ G are operations with type(n) and delay(n)
- An edge e = (n1, n2) ∈ G if & only if n2 uses the result of n1

The Code

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

The Precedence Graph

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

Instruction Scheduling

(Definitions)
A correct schedule S maps each n ∈ N into a non-negative integer representing its cycle number, and:
1. S(n) ≥ 0, for all n ∈ N obviously
2. If (n1, n2) ∈ E, S(n1) + delay(n1) ≥ S(n2)
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) ≤ L(Si), for all other schedules Si.

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

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

Scheduling Example

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

The Code
The Precedence Graph

New register name used
Detailed Scheduling Algorithm I

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

\[ \text{Code:} \]

\[
\begin{array}{l}
\text{for each } n \in N \text{ do begin count}[n] := 0; \text{ earliest}[n] = 0 \text{ end} \\
\text{for each } (n_1, n_2) \in E \text{ do begin} \\
\text{ count}[n_2] := \text{ count}[n_2] + 1; \\
\text{ successors}[n_1] := \text{ successors}[n_1] \cup \{n_2\}; \\
\text{ end} \\
\text{ for } i := 0 \text{ to } \text{ MaxC} - 1 \text{ do begin} \\
\text{ W}[i] := \emptyset; \\
\text{ Wcount := 0; } \\
\text{ for each } n \in N \text{ do begin} \\
\text{ if count}[n] = 0 \text{ then begin} \\
\text{ W}[0] := W[0] \cup \{n\}; \text{ Wcount := Wcount + 1; } \\
\text{ end} \\
\text{ c := 0; } \quad \text{ // c is the cycle number} \\
\text{ cW := 0; } \\
\text{ end} \\
\text{ end} \end{array}
\]

Detailed Scheduling Algorithm II

\[ \text{Code:} \]

\[
\begin{array}{l}
\text{while } W\text{count} > 0 \text{ do begin} \\
\text{ while } W[cW] = \emptyset \text{ do begin} \\
\text{ c := c + 1; instr[c] := \emptyset; cW := \text{ mod}(cW+1, \text{ MaxC}); } \\
\text{ end} \\
\text{ nextc := \text{ mod}(c+1, \text{ MaxC}); } \\
\text{ while } W[nextc] \neq \emptyset \text{ do begin} \\
\text{ select and remove an arbitrary instruction } x \text{ from } W[nextc]; \\
\text{ if } \exists \text{ free issue units of type } (x) \text{ on cycle } c \text{ then begin} \\
\text{ instr[c] := instr[c] \cup \{x\}; Wcount := Wcount - 1; } \\
\text{ for each } y \in \text{ successors}[x] \text{ do begin} \\
\text{ count}[y] := \text{ count}[y] - 1; \\
\text{ earliest[y] := max(earliest[y], c+delay(x)); } \\
\text{ if count}[y] = 0 \text{ then begin} \\
\text{ loc := mod(earliest[y], \text{ MaxC}); } \\
\text{ W[loc] := W[loc] \cup \{y\}; Wcount := Wcount + 1; } \\
\text{ end} \\
\text{ end} \\
\text{ end} \\
\text{ end} \end{array}
\]

More List Scheduling

List scheduling breaks down into two distinct classes

<table>
<thead>
<tr>
<th>Forward list scheduling</th>
<th>Backward list scheduling</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Start with available operations</td>
<td>• Start with no successors</td>
</tr>
<tr>
<td>• Work forward in time</td>
<td>• Work backward in time</td>
</tr>
<tr>
<td>• Ready ⇒ all operands available</td>
<td>• Ready ⇒ latency covers uses</td>
</tr>
</tbody>
</table>

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