CS415 Compilers

Instruction Scheduling
(part 3)

These slides are based on slides copyrighted by Keith Cooper, Ken Kennedy & Linda Torczon at Rice University
Announcements

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

<table>
<thead>
<tr>
<th>Operation</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>load</td>
<td>3</td>
</tr>
<tr>
<td>loadl</td>
<td>1</td>
</tr>
<tr>
<td>loadAl</td>
<td>3</td>
</tr>
<tr>
<td>store</td>
<td>3</td>
</tr>
<tr>
<td>storeAl</td>
<td>3</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>output</td>
<td>1</td>
</tr>
<tr>
<td>outputAl</td>
<td>1</td>
</tr>
</tbody>
</table>

#### Build a simple local scheduler (basic block)

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

S(n):

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

\(\Rightarrow 20\) cycles

The Code

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

The Code

```plaintext
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 Dependence Graph
The Code

\[ S(n) = \]

- 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 Dependence Graph (longest latency-weighted)
**List Scheduling Example**

**The Code**

```
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
```

**READY - SET**

**ACTIVE - SET**

**The Generated Code**

```
b: add  r1, r1 => r1
d: mult r1, r2 => r1
g: loadAI r0, @z => r2
f: mult r1, r3 => r1
h: mult r1, r2 => r1
i: storeAI r1 => r0, @w
```

**The Dependence Graph**

(longest latency-weighted)
Local (Forward) List Scheduling

\[
\begin{align*}
\text{Cycle} & \leftarrow 0 \\
\text{Ready} & \leftarrow \text{leaves of } P \\
\text{Active} & \leftarrow \emptyset \\
\text{while } (\text{Ready} \cup \text{Active} \neq \emptyset) \\
\quad \text{if } (\text{Ready} \neq \emptyset) \text{ then} \\
\quad \quad \text{remove an } op \text{ from Ready} \\
\quad \quad S(op) & \leftarrow \text{Cycle} \\
\quad \quad \text{Active} & \leftarrow \text{Active} \cup op \\
\quad \text{Cycle} & \leftarrow \text{Cycle} + 1 \\
\quad \text{for each } op \in \text{Active} \\
\quad \quad \text{if } (S(op) + \text{delay}(op) \leq \text{Cycle}) \text{ then} \\
\quad \quad \quad \text{remove } op \text{ from Active} \\
\quad \quad \quad \text{for each successor } s \text{ of } op \text{ in } P \\
\quad \quad \quad \quad \text{if } (s \text{ is ready}) \text{ then} \\
\quad \quad \quad \quad \text{Ready} & \leftarrow \text{Ready} \cup s \\
\end{align*}
\]
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 $(n_1, n_2) \in E$, $S(n_1) + \text{delay}(n_1) \leq S(n_2)$
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

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

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

The Code

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

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

### The Code

- **a:** loadAI $r_0,@w \rightarrow r_1$
- **b:** add $r_1,r_1 \rightarrow r_1$
- **c:** loadAI $r_0,@x \rightarrow r_2$
- **d:** mult $r_1,r_2 \rightarrow r_1$
- **e:** loadAI $r_0,@y \rightarrow r_3$
- **f:** mult $r_1,r_3 \rightarrow r_1$
- **g:** loadAI $r_0,@z \rightarrow r_2$
- **h:** mult $r_1,r_2 \rightarrow r_1$
- **i:** storeAI $r_1 \rightarrow r_0,@w$

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

The Code

```
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 Dependence Graph

We assume full latency for anti-dependences here
1. Build the dependence graph
2. Determine priorities: longest latency-weighted path
3. Perform list scheduling (forward)

S(n):

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

The Code

⇒ 14 cycles

The Dependence Graph

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