CS415 Compilers
Instruction Scheduling
&
Lexical Analysis

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

• Second homework has been posted (bottom-up allocation & instruction scheduling); due on Friday, February 15, 11:59pm

• First project will be on list scheduling. Will be posted this week.
Instruction Scheduling

EaC  Chapter 12
12.1 - 12.3

Part of the compiler's back end

Instruction Selection → Register Allocation → Instruction Scheduling

Errors → Machine code

m register  k register
Dependences ⇒ defined on memory locations / registers

Statement/instruction b depends on statement/instruction a if there exists:

• **true** of flow dependence
  - a writes a location/register that b later reads  (RAW conflict)

• **anti** dependence
  - a reads a location/register that b later writes  (WAR conflict)

• **output** dependence
  - a writes a location/register that b later writes  (WAW conflict)

Dependences define ORDER CONSTRAINTS that need to be respected in order to generate correct code.

<table>
<thead>
<tr>
<th>true</th>
<th>anti</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a = )</td>
<td>( = a )</td>
<td>( a = )</td>
</tr>
<tr>
<td>( = a )</td>
<td>( a = )</td>
<td>( a = )</td>
</tr>
</tbody>
</table>
To capture properties of the code, build a precedence graph \( G \)

- Nodes \( n \in G \) are operations with \( \text{type}(n) \) and \( \text{delay}(n) \)
- An edge \( e = (n_1,n_2) \in G \) if \( n_2 \) depends on \( n_1 \)

The Code

\[
\begin{align*}
a &: \text{loadAl} & r0,@w & \Rightarrow r1 \\
b &: \text{add} & r1,r1 & \Rightarrow r1 \\
c &: \text{loadAl} & r0,@x & \Rightarrow r2 \\
d &: \text{mult} & r1,r2 & \Rightarrow r1 \\
e &: \text{loadAl} & r0,@y & \Rightarrow r3 \\
f &: \text{mult} & r1,r3 & \Rightarrow r1 \\
g &: \text{loadAl} & r0,@z & \Rightarrow r2 \\
h &: \text{mult} & r1,r2 & \Rightarrow r1 \\
i &: \text{storeAl} & r1 & \Rightarrow r0,@w
\end{align*}
\]

The Precedence/Dependence Graph

All other dependences (output & anti) are covered, i.e., are satisfied through the dependencies shown.
Operation | Cycles (latency/delay)
--- | ---
load | 3
loadI | 1
loadAI | 3
store | 3
storeAI | 3
add | 1
mult | 2
fadd | 1
fmult | 2
shift | 1
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.
Local (Forward) List Scheduling

Compile-time strategy implemented in back-end

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
### Scheduling Example

<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>
</tbody>
</table>

- **Loads & stores may or may not block**
  - Non-blocking \( \Rightarrow \) fill those issue slots

- **Branches typically have delay slots**
  - Fill slots with operations unrelated to branch condition evaluation
  - Percolates branch upward

- **Branch Prediction may hide branch latencies** (hardware feature)

---

**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):

<table>
<thead>
<tr>
<th></th>
<th>a: loadAI</th>
<th>r0, @w</th>
<th>r1</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>b: add</td>
<td>r1, r1</td>
<td>r1</td>
</tr>
<tr>
<td>4</td>
<td>c: loadAI</td>
<td>r0, @x</td>
<td>r2</td>
</tr>
<tr>
<td>7</td>
<td>d: mult</td>
<td>r1, r2</td>
<td>r1</td>
</tr>
<tr>
<td>8</td>
<td>e: loadAI</td>
<td>r0, @y</td>
<td>r3</td>
</tr>
<tr>
<td>11</td>
<td>f: mult</td>
<td>r1, r3</td>
<td>r1</td>
</tr>
<tr>
<td>12</td>
<td>g: loadAI</td>
<td>r0, @z</td>
<td>r2</td>
</tr>
<tr>
<td>15</td>
<td>h: mult</td>
<td>r1, r2</td>
<td>r1</td>
</tr>
<tr>
<td>17</td>
<td>i: storeAI</td>
<td>r1</td>
<td>r0, @w</td>
</tr>
</tbody>
</table>

The Code

⇒ 20 cycles

The Dependence Graph
Scheduling Example

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

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

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 \(\Rightarrow \) r3
f: mult r1,r3 \(\Rightarrow \) r1
g: loadAI r0,@z \(\Rightarrow \) r2
h: mult r1,r2 \(\Rightarrow \) r1
i: storeAI r1 \(\Rightarrow \) 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 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 r3
f: mult r1,r3 \Rightarrow r1
g: loadAI r0,@z \Rightarrow r2
h: mult r1,r2 \Rightarrow r1
i: storeAI r1 \Rightarrow 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
3. Perform list scheduling (forward)

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 r3
f: mult r1,r3 \rightarrow r1
g: loadAl r0,@z \rightarrow r2
h: mult r1,r2 \rightarrow r1
i: storeAI r1 \rightarrow r0,@w
```

We assume full latency for anti-dependences here
Scheduling Example

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

\[ \text{S(n)}: \]

\[
\begin{array}{c|l|l}
0 & a: \text{loadAl} & r0,@w \Rightarrow r1 \\
1 & c: \text{loadAl} & r0,@x \Rightarrow r2 \\
2 & e: \text{loadAl} & r0,@y \Rightarrow r3 \\
3 & b: \text{add} & r1,r1 \Rightarrow r1 \\
4 & d: \text{mult} & r1,r2 \Rightarrow r1 \\
6 & g: \text{loadAl} & r0,@z \Rightarrow r2 \\
7 & f: \text{mult} & r1,r3 \Rightarrow r1 \\
9 & h: \text{mult} & r1,r2 \Rightarrow r1 \\
11 & i: \text{storeAl} & r1 \Rightarrow r0,@w \\
14 & &
\end{array}
\]

We assume full latency for anti-dependences here

\[ \Rightarrow 14 \text{ cycles} \]
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
• ...

The purpose of the front end is to deal with the input language
• Perform a membership test: code ∈ source language?
• Is the program well-formed (semantically) ?
• Build an IR version of the code for the rest of the compiler

The front end is not monolithic
The Front End

Scanner

- Maps stream of characters into words/tokens
  - Basic unit of syntax
  - $x = x + y$ becomes $<\text{id}, x> <\text{eq}, => <\text{id}, x> <\text{pl}, +> <\text{id}, y> <\text{sc}, ;>$
- Characters that form a word/token are its lexeme
- Its part of speech (or syntactic category) is called its token type
- Scanner discards white space & (often) comments

Source code $\rightarrow$ Scanner $\rightarrow$ tokens $\rightarrow$ Parser $\rightarrow$ IR

Errors

Speed is an issue in scanning
$\Rightarrow$ use a specialized recognizer

Lecture 6
The Front End

Parser

- Checks stream of classified words (tokens) for grammatical correctness
- Determines if code is syntactically well-formed
- Guides checking at deeper levels than syntax (static semantics)
- Builds an IR representation of the code

We’ll get to parsing in the next lectures
Lexical Analysis

Read EaC: Chapters 2.1 - 2.5;