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

Part of the compiler’s back end

EaC  Chapter 12
12.1 - 12.3
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.
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

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

The Precedence/Dependence Graph

All other dependences (output & anti) are covered, i.e., are satisfied through the dependencies shown
### Example latencies

#### Machine model (ISA) with different latencies/delays

<table>
<thead>
<tr>
<th>Operation</th>
<th>Cycles (latency/delay)</th>
</tr>
</thead>
<tbody>
<tr>
<td>load</td>
<td>3</td>
</tr>
<tr>
<td>loadI</td>
<td>1</td>
</tr>
<tr>
<td>loadAI</td>
<td>3</td>
</tr>
<tr>
<td>store</td>
<td>3</td>
</tr>
<tr>
<td>storeAI</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>outputAI</td>
<td>1</td>
</tr>
</tbody>
</table>
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

- \( \text{Cycle} \leftarrow 0 \)
- \( \text{Ready} \leftarrow \text{leaves of } P \)
- \( \text{Active} \leftarrow \emptyset \)

while (Ready \( \cup \) Active \( \neq \emptyset \))
  if (Ready \( \neq \emptyset \)) then
    remove an \( \text{op} \) from Ready
    \( S(\text{op}) \leftarrow \text{Cycle} \)
    \( \text{Active} \leftarrow \text{Active} \cup \text{op} \)
  \( \text{Cycle} \leftarrow \text{Cycle} + 1 \)
  for each \( \text{op} \in \text{Active} \)
    if (\( S(\text{op}) + \text{delay}(\text{op}) \leq \text{Cycle} \)) then
      remove \( \text{op} \) from Active
      for each successor \( s \) of \( \text{op} \) in \( P \)
        if (\( s \) is ready) then
          \( \text{Ready} \leftarrow \text{Ready} \cup s \)

Removal in priority order

op has completed execution

If successor’s operands are ready, put it on Ready
<table>
<thead>
<tr>
<th>Operation</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>load</td>
<td>3</td>
</tr>
<tr>
<td>loadI</td>
<td>1</td>
</tr>
<tr>
<td>loadAI</td>
<td>3</td>
</tr>
<tr>
<td>store</td>
<td>3</td>
</tr>
<tr>
<td>storeAI</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>outputAI</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):
- 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

⇒ 20 cycles

The Code

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

The Code

<table>
<thead>
<tr>
<th></th>
<th>Opcode</th>
<th>Source</th>
<th>Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>loadAI</td>
<td>r0,@w</td>
<td>r1</td>
</tr>
<tr>
<td>b</td>
<td>add</td>
<td>r1,r1</td>
<td>r1</td>
</tr>
<tr>
<td>c</td>
<td>loadAI</td>
<td>r0,@x</td>
<td>r2</td>
</tr>
<tr>
<td>d</td>
<td>mult</td>
<td>r1,r2</td>
<td>r1</td>
</tr>
<tr>
<td>e</td>
<td>loadAI</td>
<td>r0,@y</td>
<td>r3</td>
</tr>
<tr>
<td>f</td>
<td>mult</td>
<td>r1,r3</td>
<td>r1</td>
</tr>
<tr>
<td>g</td>
<td>loadAI</td>
<td>r0,@z</td>
<td>r2</td>
</tr>
<tr>
<td>h</td>
<td>mult</td>
<td>r1,r2</td>
<td>r1</td>
</tr>
<tr>
<td>i</td>
<td>storeAI</td>
<td>r1</td>
<td>r0,@w</td>
</tr>
</tbody>
</table>

The Dependence Graph
Scheduling Example

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

The Code

<table>
<thead>
<tr>
<th>Operation</th>
<th>Source</th>
<th>Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>a: loadAI</td>
<td>r0,@w</td>
<td>r1</td>
</tr>
<tr>
<td>b: add</td>
<td>r1,r1</td>
<td>r1</td>
</tr>
<tr>
<td>c: loadAI</td>
<td>r0,@x</td>
<td>r2</td>
</tr>
<tr>
<td>d: mult</td>
<td>r1,r2</td>
<td>r1</td>
</tr>
<tr>
<td>e: loadAI</td>
<td>r0,@y</td>
<td>r3</td>
</tr>
<tr>
<td>f: mult</td>
<td>r1,r3</td>
<td>r1</td>
</tr>
<tr>
<td>g: loadAI</td>
<td>r0,@z</td>
<td>r2</td>
</tr>
<tr>
<td>h: mult</td>
<td>r1,r2</td>
<td>r1</td>
</tr>
<tr>
<td>i: storeAI</td>
<td>r1</td>
<td>r0,@w</td>
</tr>
</tbody>
</table>

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 \implies r1
b: add r1,r1 \implies r1
c: loadAI r0,@x \implies r2
d: mult r1,r2 \implies r1
e: loadAI r0,@y \implies r3
f: mult r1,r3 \implies r1
g: loadAI r0,@z \implies r2
h: mult r1,r2 \implies r1
i: storeAI r1 \implies 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)

- **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 Code*

*The Dependence Graph*

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)

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

We assume full latency for anti-dependences here

The Dependence Graph
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 → **Scanner** → **tokens** → **Parser** → IR

Errors

Speed is an issue in scanning
⇒ use a specialized recognizer
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;