Condition Codes

• processor state register ($psr$)

<table>
<thead>
<tr>
<th>$impl$</th>
<th>$ver$</th>
<th>$icc$</th>
<th>$EC$</th>
<th>$F$</th>
<th>$S$</th>
<th>$P$</th>
<th>$T$</th>
<th>$CWP$</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>27</td>
<td>23</td>
<td>19</td>
<td>13</td>
<td>12</td>
<td>11</td>
<td>7</td>
<td>6</td>
</tr>
</tbody>
</table>

• integer condition codes — the $icc$ field — holds 4 bits

- $N$ set if the last ALU result was negative
- $Z$ set if the last ALU result was zero
- $V$ set if the last ALU result overflowed
- $C$ set if the last ALU instruction that modified $icc$ caused a carry out of, or a borrow into, bit 31

- $cc$ versions of the integer arithmetic instructions set all the codes
- $cc$ versions of the logical instructions set only $N$ and $Z$
- tests on the condition codes implement conditionals and loops
- carry and overflow are used to implement multiple-precision arithmetic
- see §4.8 in Paul

Compare and Test

• test and compare synthetic instructions set condition codes

• to test a single value

  $\text{test reg}$

  $\text{orcc reg, %g0, %g0}$

• compare two values

  $\text{cmp src}_1, src_2$

  $\text{subcc src}_1, src_2, %g0$

  $\text{cmp src, value}$

  $\text{subcc src, value, %g0}$

• using $%g0$ as a destination discards the result
Carry and Overflow

- if the carry bit (C) is set
  - the last addition resulted in a carry
  - or the last subtraction resulted in a borrow
- carry is needed to implement arithmetic using numbers represented in several words, e.g. multiple-precision addition
  - \( \text{addcc} \ %g3, %g5, %g7 \)
  - \( \text{addxcc} \ %g2, %g4, %g6 \)
  - \( (%g6, %g7) = (%g2, %g3) + (%g4, %g5) \)
  - the most-significant word is in the even register;
  - the least-significant word is in the odd register
- overflow (V) indicates that the result of signed addition or subtraction doesn’t fit

Branches

- branch instructions transfer control based on \(icc\)

\[ \begin{array}{c|c|c|c|c|c|} \hline
\text{010} & a & \text{cond} & 00 & 01 & \text{disp22} \\
\hline
0 & 31 & 29 & 28 & 24 & 21 \end{array} \]

- target is a \(PC\)-relative address and is \(PC + 4 \times \text{disp22}\), where \(PC\) is the address of the branch instruction
- unconditional branches

<table>
<thead>
<tr>
<th>branch</th>
<th>condition</th>
<th>synthetic synonym</th>
</tr>
</thead>
<tbody>
<tr>
<td>ba</td>
<td>branch always</td>
<td>jmp</td>
</tr>
<tr>
<td>bn</td>
<td>branch never</td>
<td>nop</td>
</tr>
</tbody>
</table>
Branches, cont’d

- raw condition-code branches

<table>
<thead>
<tr>
<th>branch</th>
<th>condition</th>
<th>synthetic synonym</th>
</tr>
</thead>
<tbody>
<tr>
<td>bnz</td>
<td>$!Z$</td>
<td></td>
</tr>
<tr>
<td>bz</td>
<td>$Z$</td>
<td></td>
</tr>
<tr>
<td>bpos</td>
<td>$!N$</td>
<td></td>
</tr>
<tr>
<td>bneg</td>
<td>$N$</td>
<td></td>
</tr>
<tr>
<td>bcc</td>
<td>$!C$</td>
<td>bgeu</td>
</tr>
<tr>
<td>bcs</td>
<td>$C$</td>
<td>blu</td>
</tr>
<tr>
<td>bvc</td>
<td>$!V$</td>
<td></td>
</tr>
<tr>
<td>bvs</td>
<td>$V$</td>
<td></td>
</tr>
</tbody>
</table>

- comparisons

<table>
<thead>
<tr>
<th>branches</th>
<th>signed</th>
<th>unsigned</th>
<th>synthetic synonym</th>
</tr>
</thead>
<tbody>
<tr>
<td>be</td>
<td>$Z$</td>
<td>$Z$</td>
<td>bz</td>
</tr>
<tr>
<td>bne</td>
<td>$!Z$</td>
<td>$!Z$</td>
<td>bnz</td>
</tr>
<tr>
<td>bg</td>
<td>$!Z$</td>
<td>$(N\wedge V)$</td>
<td>$(C \wedge Z)$</td>
</tr>
<tr>
<td>ble</td>
<td>$Z$</td>
<td>$(N\wedge V)$</td>
<td>$C$</td>
</tr>
<tr>
<td>bge</td>
<td>$!Z$</td>
<td>$(N\wedge V)$</td>
<td>$!C$</td>
</tr>
<tr>
<td>bl</td>
<td>$N\wedge V$</td>
<td>$!C$</td>
<td></td>
</tr>
</tbody>
</table>

Control Transfer

- normally, instructions are fetched and executed from sequential memory locations

- program counter, $PC$, is address of the current instruction, and the program counter, $nPC$, is address of the next instruction: $nPC = PC + 4$

- branches, control-transfer instructions change $nPC$ to something else

- control-transfer instructions

<table>
<thead>
<tr>
<th>instruction</th>
<th>type</th>
<th>addressing mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>bicc</td>
<td>conditional branches</td>
<td>PC-relative</td>
</tr>
<tr>
<td>f6fccc</td>
<td>floating point</td>
<td>PC-relative</td>
</tr>
<tr>
<td>cbcccc</td>
<td>coprocessor</td>
<td>PC-relative</td>
</tr>
<tr>
<td>jmp1</td>
<td>jump and link</td>
<td>register indirect</td>
</tr>
<tr>
<td>rett</td>
<td>return from trap</td>
<td>register indirect</td>
</tr>
<tr>
<td>call</td>
<td>procedure call</td>
<td>PC-relative</td>
</tr>
<tr>
<td>ticc</td>
<td>traps</td>
<td>register-indirect vectored</td>
</tr>
</tbody>
</table>

- $PC$-relative addressing is like register displacement addressing that uses $PC$ as the base register
Control Transfer, cont’d

• branches

\[
\begin{array}{c|c|c|c|c}
00 & a & \text{cond} & 010 & \text{disp22} \\
\hline
31 & 29 & 28 & 24 & 21
\end{array}
\]

\[n_{\text{PC}} = PC + 4 \times \text{signextend}(\text{disp22})\]

jumping to an arbitrary location may require two branches, but branches are used to build conditionals and loops in “small” code blocks

• calls

\[
\begin{array}{c|c}
01 & \text{disp30} \\
\hline
31 & 29
\end{array}
\]

\[n_{\text{PC}} = PC + 4 \times \text{zeroextend}(\text{disp30})\]

is multiplied by 4 because all instructions are word aligned

• position-independent code is code whose correct execution does not depend on where it is loaded, i.e., all instructions use PC-relative addressing

Branching Examples

• if-then-else

if (a > b)
c = a;
else
c = b;
becomes

\[
\begin{align*}
\text{#define } a & \ %l0 \\
\text{#define } b & \ %l1 \\
\text{#define } c & \ %l3 \\
\text{cmp } a, & b \\
\text{ble } & L1; \ \text{nop} \\
\text{mov } & a,c \\
\text{ba } & L2; \ \text{nop} \\
\text{L1: } & \text{mov } b,c \\
\text{L2: } & ...
\end{align*}
\]

• loops

for (i = 0; i < n; i++)
...
becomes

\[
\begin{align*}
\text{#define } i & \ %l0 \\
\text{#define } n & \ %l1 \\
\text{clr } i \\
\text{L1: } & \text{cmp } i,n \\
& \text{bge } L2; \ \text{nop} \\
& \text{...} \\
& \text{inc } i \\
& \text{ba } L1; \ \text{nop} \\
\text{L2: } & ...
\end{align*}
\]

• lcc generates

\[
\begin{align*}
\text{clr } i \\
\text{ba } & L5; \ \text{nop} \\
\text{L2: } & \text{...} \\
& \text{inc } i \\
\text{L5: } & \text{cmp } i,n \\
& \text{bl } L2; \ \text{nop}
\end{align*}
\]