Condition Codes

- processor state register ($psr$)

<table>
<thead>
<tr>
<th>impl</th>
<th>ver</th>
<th>icc</th>
<th>EC</th>
<th>EF</th>
<th>S</th>
<th>P</th>
<th>E</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>
<td>5</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>23</td>
<td>22</td>
<td>21</td>
<td>20</td>
</tr>
</tbody>
</table>

- $N$ 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 page 28 in the SPARC Architecture Manual, §4.8 in Paul
Compare and Test

- test and compare *synthetic* instructions set condition codes

- to test a single value
  
  ```
  tst reg
  orcc reg,%g0,%g0
  ```

- compare two values
  
  ```
  cmp src_1, src_2
  subcc src_1, src_2, %g0
  cmp src, value
  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

```
addcc  %g3,%g5,%g7
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 $icc$

\[
\begin{bmatrix}
\text{a} \\
\text{n} \\
\vdots \\
\text{vs} \\
\end{bmatrix} \quad \{a\} \\
\text{label}
\]

- “annul” bit; more later

Branches are format 2 instructions

<table>
<thead>
<tr>
<th>00</th>
<th>a</th>
<th>cond</th>
<th>010</th>
<th>disp22</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>29</td>
<td>28</td>
<td>24</td>
<td>21</td>
</tr>
</tbody>
</table>

- 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 ^V]</td>
<td>![C ^Z]</td>
</tr>
<tr>
<td>bg</td>
<td>![Z</td>
<td>![N ^V)]</td>
<td>![C</td>
</tr>
<tr>
<td>ble</td>
<td>![Z</td>
<td>![N ^V)]</td>
<td>![C</td>
</tr>
<tr>
<td>bge</td>
<td>![N ^V]</td>
<td>![C]</td>
<td></td>
</tr>
<tr>
<td>bl</td>
<td>![N ^V]</td>
<td>![C]</td>
<td></td>
</tr>
</tbody>
</table>
Control Transfer

- normally, instructions are fetched and executed from sequential memory locations
- program counter, $\text{PC}$, is address of the current instruction, and the program counter, $n\text{PC}$, is address of the next instruction: $n\text{PC} = \text{PC} + 4$
- branches, control-transfer instructions change $n\text{PC}$ to something else
- control-transfer instructions

<table>
<thead>
<tr>
<th>instruction</th>
<th>type</th>
<th>addressing mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{bicc}$</td>
<td>conditional branches</td>
<td>$\text{PC}$-relative</td>
</tr>
<tr>
<td>$\text{fbfcc}$</td>
<td>floating point</td>
<td>$\text{PC}$-relative</td>
</tr>
<tr>
<td>$\text{cbcCC}$</td>
<td>coprocessor</td>
<td>$\text{PC}$-relative</td>
</tr>
<tr>
<td>$\text{jmpl}$</td>
<td>jump and link</td>
<td>register indirect</td>
</tr>
<tr>
<td>$\text{reott}$</td>
<td>return from trap</td>
<td>register indirect</td>
</tr>
<tr>
<td>$\text{call}$</td>
<td>procedure call</td>
<td>$\text{PC}$-relative</td>
</tr>
<tr>
<td>$\text{ticc}$</td>
<td>traps</td>
<td>register-indirect vectored</td>
</tr>
</tbody>
</table>

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

• branches

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

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

• calls

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

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

#define a %l0
#define b %l1
#define c %l3

cmp a,b
ble L1; nop
mov a,c
ba L2; nop
L1: mov b,c
L2: ...

• loops

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

becomes

#define i %l0
#define n %l1

clr i
L1: cmp i,n
  bge L2; nop
  ...
  inc i
  ba L1; nop
L2: ...

• lcc generates

clr i
ba L5; nop
L2: ...
  inc i
L5: cmp i,n
  bl L2; nop