Branching

CS 217

Condition Codes

• Processor State Register (PSR)

<table>
<thead>
<tr>
<th></th>
<th>icc</th>
<th></th>
</tr>
</thead>
</table>

• Integer condition codes (icc)
  - 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 was overflowed
  - C set if the last ALU instruction that modified the icc caused a carry out of, or a borrow into, bit 31

Condition Codes (cont)

• cc versions of the integer arithmetic instructions set all the codes
• ccc versions of the logical instructions set only N and Z bits
• Tests on the condition codes implement conditional branches and loops
• Carry and overflow are used to implement multiple-precision arithmetic
Compare and Test

- Synthetic instructions set condition codes
  \[ \text{tst reg} \quad \text{orcc reg, } \%g0, \%g0 \]
  \[ \text{cmp src1, src2} \quad \text{subcc src1, src2, } \%g0 \]
  \[ \text{cmp src, value} \quad \text{subcc src, value, } \%g0 \]
- Using \%g0 as the destination discards the result

---

Carry and Overflow

- If the carry bit is set
  the last addition resulted in a carry, or
  the last subtraction resulted in a borrow
- Used for multi-word addition
  \[ \text{addcc } \%g3, \%g5, \%g7 \]
  \[ \text{addcc } \%g2, \%g4, \%g6 \]
  \( (\%g6, \%g7) = (\%g2, \%g3) + (\%g4, \%g5) \)
  the most significant word is in the even register
- Overflow indicates result of subtraction
  (or signed-addition) doesn’t fit

---

Branch Instructions

- Transfer control based on icc
  \[
  \begin{array}{c}
  b \\
  \downarrow \\
  \text{cond} \\
  \hline
  00 \quad 010 \quad \text{disp22}
  \end{array}
  \]
  target is a PC-relative address: PC + 4 x disp22
  where PC is the address of the branch instruction
Branch Instructions (cont)

• Unconditional branches (and synonyms)
  
<table>
<thead>
<tr>
<th>Instruction</th>
<th>signed</th>
<th>unsigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>ba</td>
<td>Z</td>
<td>Z</td>
</tr>
<tr>
<td>bn</td>
<td>Z</td>
<td>IZ</td>
</tr>
<tr>
<td>br</td>
<td>N</td>
<td>IC</td>
</tr>
<tr>
<td>bcs</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>bvc</td>
<td>V</td>
<td></td>
</tr>
</tbody>
</table>

• Raw condition-code branches
  
<table>
<thead>
<tr>
<th>Instruction</th>
<th>signed</th>
<th>unsigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>bns</td>
<td>IZ</td>
<td></td>
</tr>
<tr>
<td>bzb</td>
<td>Z</td>
<td></td>
</tr>
<tr>
<td>bps</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>bneq</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>bcc</td>
<td>IC</td>
<td></td>
</tr>
<tr>
<td>bcs</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>bvc</td>
<td>V</td>
<td></td>
</tr>
<tr>
<td>bvs</td>
<td>V</td>
<td></td>
</tr>
</tbody>
</table>

Branching Instructions (cont)

• Comparisons
  
<table>
<thead>
<tr>
<th>Instruction</th>
<th>signed</th>
<th>unsigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>be</td>
<td>Z</td>
<td>Z</td>
</tr>
<tr>
<td>bne</td>
<td>IZ</td>
<td>IZ</td>
</tr>
<tr>
<td>bg</td>
<td>![Z</td>
<td>(N^V)]</td>
</tr>
<tr>
<td>ble</td>
<td>![Z</td>
<td>(N^V)]</td>
</tr>
<tr>
<td>bge</td>
<td>![N^V)]</td>
<td>IC</td>
</tr>
<tr>
<td>bl</td>
<td>N^V</td>
<td>C</td>
</tr>
</tbody>
</table>

Control Transfer

• Instructions normally fetched and executed from sequential memory locations
• PC is the address of the current instruction, and nPC is the address of the next instruction (nPC = PC + 4)
• Branches and control transfer instructions change nPC to something else
Control Transfer (cont)

- Control transfer instructions

\[
\begin{array}{|c|c|c|c|}
\hline
\text{instruction} & \text{type} & \text{addressing mode} \\
\hline
\text{bicc} & \text{conditional branch} & \text{PC-relative} \\
\text{jmp}l & \text{jump and link} & \text{register indirect} \\
\text{return} & \text{return from trap} & \text{register indirect} \\
\text{call} & \text{procedure call} & \text{PC-relative} \\
\text{traps} & \text{traps} & \text{register indirect (vectored)} \\
\hline
\end{array}
\]

PC-relative addressing is like register displacement addressing that uses the PC as the base register.

Control Transfer (cont)

- Branch instructions

\[
\begin{align*}
\text{op} & \text{ a } \text{ cond } \text{ op2 } \text{ disp22} \\
\text{nPC} & = \text{PC} + 4 \times \text{signextend(disp22)}
\end{align*}
\]

- Calls

\[
\begin{align*}
\text{op} & \text{ disp30} \\
\text{nPC} & = \text{PC} + \text{zeroextend(disp30)}
\end{align*}
\]

position-independent code does not depend on where it’s loaded; uses PC-relative addressing.

Branching Examples

- if-then-else

\[
\begin{align*}
\text{if } (a > b) & \text{ #define a } \%10 \\
 & \text{define b } \%11 \\
 & \text{define c } \%12 \\
 & \text{cmp a,b} \\
 & \text{ble L1; nop} \\
 & \text{mov a,c} \\
 & \text{ba L2; nop} \\
 & \text{L1: mov b,c} \\
 & \text{L2: ...}
\end{align*}
\]
Branching Examples (cont)

• Loops
  
  ```
  for (i=0; i<n; i++)
  . .
  define i %10
  define n %11
  clr i
  L1: cmp i, n
  bge L2; nop
  . .
  inc i
  ba L1; nop
  L2:
  ```

• Alternative implementation
  
  ```
  for (i=0; i<n; i++)
  . .
  define i %10
  define n %11
  clr i
  L2: cmp i, n
  b1 L1; nop
  L1:
  . .
  inc i
  ```

Instruction Pipelining

• Pipeline

  ```
  Fetch Decode Opcode Execute Store
  Fetch Decode Opcode Execute Store
  Fetch Decode Opcode Execute Store
  . .
  ```

• PC is incremented by 4 at the Fetch stage to retrieve the next instruction
Pipelining (cont)

- A delay slot is caused by a jmp instruction

```
P C    nPC  instruction
  8    12    add    add    add
 12    16    jmp  40    jmp    jmp
 16    40    delay  delay
     ... Delay  Delay
  40    44    sub    sub
```

Delay Slots

- One option: use nop in all delay slots
- Optimizing compilers try to fill delay slots

```
if (a>b) c=a; else c=b;
    cmp a, b    cmp a, b
    ble L1;    ble L1
    nop;    mov b, c
    mov a, c    mov a, c
    ba L2;    L1: ...
    nop
    L1: mov b, c
    L2: ...
```

Annul Bit

- Controls the execution of the delay-slot instruction

```
bg, a    L1
mov a, c
```

the ,a causes the mov instruction to be executed if the branch is taken, and not executed if the branch is not taken

- Exception

   ba, a L does not execute the delay-slot instruction
Annul Bit (cont)

* Optimized for \( i=0; i<n; i++ \) 1;2;...;n

\[
\begin{align*}
&\text{clr} \ i &\text{clr} \ i \\
&\text{ba} \ L2 &\text{ba},a \ L2 \\
&L1: 1 &L1: 2 \\
&2 &\ldots \\
&\ldots &n \\
&\text{inc} \ i &\text{inc} \ i \\
&L2: \text{cmp} \ i,n &L2: \text{cmp} \ i,n \\
&\text{bl} \ L1 &\text{bl},a \ L1 \\
&\text{nop} &1 \\
\end{align*}
\]

While-Loop Example

\[
\begin{align*}
\text{while (\ldots) } &\begin{align*}
\text{test: cmp } \ldots \\
&\text{b: done} \\
&\text{nop} \\
&\text{stmt}_1 \\
&\vdots \\
&\text{stmt}_n \\
\end{align*} \\
&\text{3 instr} \\
\vline \\
\begin{align*}
&\text{stmt}_1 \\
&\vdots \\
&\text{stmt}_n \\
&\text{test: cmp } \ldots \\
&\text{b: loop} \\
&\text{nop} \\
&\text{done: } \ldots \\
\end{align*} \\
&\text{2 instr}
\end{align*}
\]

While-Loop (cont)

* Move test to end of loop
* Eliminate first test

\[
\begin{align*}
\text{test: cmp } \ldots &\begin{align*}
\text{ba test} \\
&\text{b: done} \\
&\text{nop} \\
&\text{loop: stmt}_1 \\
&\vdots \\
&\text{stmt}_n \\
&\text{cmp } \ldots \\
&\text{b: loop} \\
&\text{nop} \\
&\text{done: } \ldots \\
\end{align*} \\
&\text{ba test} \\
&\text{b: done} \\
&\text{nop} \\
&\text{loop: stmt}_1 \\
&\vdots \\
&\text{stmt}_n \\
&\text{test: cmp } \ldots \\
&\text{b: loop} \\
&\text{nop} \\
&\text{done: } \ldots \\
\end{align*}
\]
While-Loop (cont)

- Eliminate the `nop` in the loop

```c
ba test
nop
loop: stmt1
    stmt_n
test: cmp ...
bnz, a loop
stmt1...
```

now 2 overhead instructions per loop

---

If-Then-Else Example

```c
if (...) {
    t-stmt1...
    t-stmt_n
} else {
    e-stmt1...
    e-stmt_n
} else {
    e-stmt1...
    e-stmt_n
} else {
    next: ...
```