Code Shape III
Booleans, Relationals, & Control flow

EaC section 7.4 & 7.8

Copyright 2003, Keith D. Cooper, Ken Kennedy & Linda Torczon, all rights reserved.
Boolean & Relational Values

How should the compiler represent them?
• Answer depends on the target machine

Two classic approaches
• Numerical representation
• Positional (implicit) representation

Correct choice depends on both context and ISA
Boolean & Relational Values

Numerical representation
- Assign values to TRUE and FALSE
- Use hardware AND, OR, and NOT operations
- Use comparison to get a boolean from a relational expression

Examples

```
x < y  becomes  cmp_LT r_x, r_y → r_1

if (l < r)
   stmt_1
else
   stmt_2
```

conditional branch
Boolean & Relational Values

What if the ISA uses a condition code?
• Must use a conditional branch to interpret result of compare
• Necessitates branches in the evaluation

Example:

\[ x < y \quad \text{becomes} \quad \text{cmp } r_x, r_y \rightarrow cc_1 \]
\[ \text{cbr}_{LT} cc_1 \rightarrow L_T, L_F \]
\[ L_T: \quad \text{load} 1 \rightarrow r_2 \]
\[ \quad \text{br} \rightarrow L_E \]
\[ L_F: \quad \text{load} 0 \rightarrow r_2 \]
\[ L_E: \quad \text{... other stmts ...} \]

This “positional representation” is much more complex
Boolean & Relational Values

What if the ISA uses a condition code?
• Must use a conditional branch to interpret result of compare
• Necessitates branches in the evaluation

Example:

\[ x < y \] becomes
\[
\text{cmp } r_x, r_y \rightarrow cc_1 \\
cbr\text{ LT } cc_1 \rightarrow L_T, L_F \\
L_T: \quad \text{loadl } 1 \rightarrow r_2 \\
\quad \text{br } \rightarrow L_E \\
L_F: \quad \text{loadl } 0 \rightarrow r_2 \\
L_E: \quad ... \text{ other stmts } ...
\]

Condition codes
• are an architect’s hack
• allow ISA to avoid some comparisons
• complicates code for simple cases

This “positional representation” is much more complex
The last example actually encodes result in the PC. If result is used to control an operation, this may be enough.

<table>
<thead>
<tr>
<th>Example</th>
<th>Variations on the ILOC Branch Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>if (x &lt; y)</td>
<td>Straight Condition Codes</td>
</tr>
<tr>
<td></td>
<td>Boolean Compares</td>
</tr>
<tr>
<td>then a ← c + d</td>
<td>comp  rₓ,rᵧ⇒cc₁</td>
</tr>
<tr>
<td>else a ← e + f</td>
<td>cbr_LT cc₁ →L₁,L₂</td>
</tr>
<tr>
<td></td>
<td>L₁: add rₓ,rᵧ⇒ra</td>
</tr>
<tr>
<td></td>
<td>br →LOUT</td>
</tr>
<tr>
<td></td>
<td>L₂: add rₓ,rᵧ⇒ra</td>
</tr>
<tr>
<td></td>
<td>br →LOUT</td>
</tr>
<tr>
<td></td>
<td>LOUT: nop</td>
</tr>
<tr>
<td></td>
<td>LOUT: nop</td>
</tr>
</tbody>
</table>
Conditional move & predication both simplify this code.

<table>
<thead>
<tr>
<th>Example</th>
<th>Conditional Move</th>
<th>Predicated Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>if (x &lt; y) then a ← c + d else a ← e + f</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Other Architectural Variations

<table>
<thead>
<tr>
<th></th>
<th>Conditional Move</th>
<th>Predicated Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>comp</td>
<td>( r_x, r_y \Rightarrow cc_1 )</td>
<td>( \text{cmp}_{\text{LT}} \quad r_x, r_y \Rightarrow r_1 )</td>
</tr>
<tr>
<td>add</td>
<td>( r_c, r_d \Rightarrow r_1 )</td>
<td>( (r_1)? \quad \text{add} \quad r_c, r_d \Rightarrow r_a )</td>
</tr>
<tr>
<td>add</td>
<td>( r_e, r_f \Rightarrow r_2 )</td>
<td>( (\neg r_1)? \quad \text{add} \quad r_e, r_f \Rightarrow r_a )</td>
</tr>
<tr>
<td>i2i_(&lt;)</td>
<td>( cc_1, r_1, r_2 \Rightarrow r_a )</td>
<td></td>
</tr>
</tbody>
</table>

Both versions avoid the branches.
Both are shorter than CCs or Boolean-valued compare.
Are they better?
Boolean & Relational Values

Consider the assignment \( x \leftarrow a < b \land c < d \)

<table>
<thead>
<tr>
<th>Variations on the Iloc Branch Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Straight Condition Codes</strong></td>
</tr>
<tr>
<td>comp ( r_a, r_b \Rightarrow cc_1 )</td>
</tr>
<tr>
<td>cbr_LT ( cc_1 \rightarrow L_1, L_2 )</td>
</tr>
<tr>
<td>L_1: comp ( r_c, r_d \Rightarrow cc_2 )</td>
</tr>
<tr>
<td>cbr_LT ( cc_2 \rightarrow L_3, L_2 )</td>
</tr>
<tr>
<td>L_2: loadI ( 0 \Rightarrow r_x )</td>
</tr>
<tr>
<td>br ( \rightarrow L_{OUT} )</td>
</tr>
<tr>
<td>L_3: loadI ( 1 \Rightarrow r_x )</td>
</tr>
<tr>
<td>br ( \rightarrow L_{OUT} )</td>
</tr>
<tr>
<td>L_{OUT}: nop</td>
</tr>
</tbody>
</table>

Here, the boolean compare produces much better code
## Boolean & Relational Values

Conditional move & predication help here, too

\[ x \leftarrow a < b \land c < d \]

### Other Architectural Variations

<table>
<thead>
<tr>
<th>Conditional Move</th>
<th>Predicated Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>comp r_a, r_b \Rightarrow CC_1</td>
<td>cmp_LT r_a, r_b \Rightarrow r_1</td>
</tr>
<tr>
<td>i2i_&lt; cc_1, r_T, r_F \Rightarrow r_1</td>
<td></td>
</tr>
<tr>
<td>comp r_c, r_d \Rightarrow CC_2</td>
<td>cmp_LT r_c, r_d \Rightarrow r_2</td>
</tr>
<tr>
<td>and cc_2, r_T, r_F \Rightarrow r_2</td>
<td>and r_1, r_2 \Rightarrow r_x</td>
</tr>
<tr>
<td>i2i_&lt; r_1, r_2 \Rightarrow r_x</td>
<td></td>
</tr>
</tbody>
</table>

Conditional move is worse than Boolean compares
Predication is identical to Boolean compares

Context & hardware determine the appropriate choice
Control Flow

If-then-else
• Follow model for evaluating relationals & booleans with branches

Branching versus predication (e.g., IA-64)

• Frequency of execution
  → Uneven distribution ⇒ do what it takes to speed common case
• Amount of code in each case
  → Unequal amounts means predication may waste issue slots

• Control flow inside the construct
  → Any branching activity within the case base complicates the predicates and makes branches attractive
Control Flow

Loops
• Evaluate condition before loop (if needed)
• Evaluate condition after loop
• Branch back to the top (if needed)
Merges test with last block of loop body

while, for, do, & until all fit this basic model
Loop Implementation Code

for (i = 1; i < 100; i++) {
  \textit{body}
}
\textit{next statement}

\begin{itemize}
  \item \texttt{loadI 1 \Rightarrow r_1}
  \item \texttt{loadI 1 \Rightarrow r_2}
  \item \texttt{loadI 100 \Rightarrow r_3}
  \item \texttt{cmp\textunderscore GE r_1, r_3 \Rightarrow r_4}
  \item \texttt{cbr r_4 \Rightarrow L_2, L_1}
\end{itemize}

\textbf{L_1:} \textit{body}

\begin{itemize}
  \item \texttt{add r_1, r_2 \Rightarrow r_1}
  \item \texttt{cmp\textunderscore LT r_1, r_3 \Rightarrow r_5}
  \item \texttt{cbr r_5 \Rightarrow L_1, L_2}
\end{itemize}

\textbf{L_2:} \textit{next statement}

- \textit{Initialization}
- \textit{Pre-test}
- \textit{Post-test}
Break statements

Many modern programming languages include a break
• Exits from the innermost control-flow statement
  → Out of the innermost loop
  → Out of a case statement

Translates into a jump
• Targets statement outside control-flow construct
• Creates multiple-exit construct
• Skip in loop goes to next iteration

Only make sense if loop has > 1 block
Control Flow

Case Statements

1. Evaluate the controlling expression
2. Branch to the selected case
3. Execute the code for that case
4. Branch to the statement after the case

Parts 1, 3, & 4 are well understood, part 2 is the key
Control Flow

Case Statements
1. Evaluate the controlling expression
2. Branch to the selected case
3. Execute the code for that case
4. Branch to the statement after the case \( \text{use break} \)

Parts 1, 3, & 4 are well understood, part 2 is the key

Strategies
- Linear search (nested if-then-else constructs)
- Build a table of case expressions & binary search it
- Directly compute an address (requires dense case set)

Surprisingly many compilers do this for all cases!