Implementing Control Flow Constructs

Comp 412

Copyright 2018, Keith D. Cooper & Linda Torczon, all rights reserved.
Students enrolled in Comp 412 at Rice University have explicit permission to make copies of these materials for their personal use.
Faculty from other educational institutions may use these materials for nonprofit educational purposes, provided this copyright notice is preserved.

Chapter 7 in EaC2e
# Lab 3 Schedule

<table>
<thead>
<tr>
<th>Sunday</th>
<th>Monday</th>
<th>Tuesday</th>
<th>Wednesday</th>
<th>Thursday</th>
<th>Friday</th>
<th>Saturday</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oct 21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td></td>
<td>26</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Start work</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Docs</td>
<td>Available Lab Lecture</td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Complete &amp; Correct Dependence Graph</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Correct Schedule</td>
<td>18</td>
<td>19</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Late day</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Lab 3 Due Date</td>
<td></td>
<td></td>
<td>Thanksgiving Break</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Not a late day</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Not a late day</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Not a late day</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The goal is here

You are here

Lab 3 Due Date

Thanksgiving Break

Late day

Not a late day

Not a late day

Not a late day
What is Left in Translation?

We have talked about the mechanism of translation. Next, we need to focus on the content, or product, of translation.

**Expressions**
- More operators in expressions
  - Boolean expressions, relational expressions, string operations
- Type-related issues in expressions
  - Mixed-type operations and value conversions.
- Assignment (including addressability)

**Control Flow**
- Code shape and SDT strategies

**Procedure and Function Calls**
- Functionality: what, precisely, does a procedure call do?
- Implementation: how does the compiler emit code to do that?
First, we need to add Boolean & relational expressions to the grammar

Boolean  →  Boolean ∨ AndTerm
          |  AndTerm
          |  AndTerm ∧ RelExpr
          |  RelExpr

AndTerm  →  AndTerm ∧ RelExpr
          |  RelExpr

RelExpr  →  RelExpr < Expr
          |  RelExpr ≤ Expr
          |  RelExpr = Expr
          |  RelExpr ≠ Expr
          |  RelExpr ≥ Expr
          |  RelExpr > Expr
          |  Expr

Expr     →  Expr + Term
          |  Expr - Term
          |  Term
          |  Term × Value
          |  Term ÷ Value
          |  Value
          |  ! Factor
          |  Factor
          |  ( Expr )
          |  number
          |  Reference

... where Reference derives a name, a subscripted name, a structure reference, a string reference, a function call, ...
Control-flow Constructs and Portability

Computer Scientists *naturally* want to isolate as much of the compiler as possible from the ISA.

- Enhances portability (conceptually)
- Leads to a better story

Implementation of control-flow is one place where reality intrudes

- From representation to comparison to branching, ISA matters
- Optimization cannot easily fix bad control-flow code-shape decisions
Boolean Values

The compiler needs to represent the values TRUE and FALSE

Numerical representation

• Assign values to TRUE and FALSE (design time)
  – Common choices are 1 / 0, -1 / 0, or non-zero / 0
  – Not a lot of difference between the choices

• Use hardware AND, OR, and NOT operations

⇒ Make a choice and use it uniformly

Implementation of Boolean values and relational comparisons depends heavily on features of the ISA.
Relational Values

Different instruction sets (ISAs) provide different kinds of support

<table>
<thead>
<tr>
<th>Comparison Produces</th>
<th>ISA Impact</th>
<th>Impact on Code Shape</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean condition code</td>
<td>multiple compare ops</td>
<td>⇒ complexity into expressions</td>
</tr>
<tr>
<td></td>
<td>multiple branch ops</td>
<td>⇒ complexity into branches</td>
</tr>
</tbody>
</table>

Branch consumes what compare produces

- Fundamental design issue: operations intended to work together
- Most ISAs settle on a single model

Implementation of Boolean values and relational comparisons depends heavily on features of the ISA.
Boolean & Relational Values

**Boolean-valued comparisons**
- Code can use the result of a relational comparison directly
- Presume that Boolean ops (e.g., AND, OR, NOT) work on result
- Leads to straightforward translation

**Examples**

\[
\begin{align*}
\text{if} (x < y) & \quad \text{then } \text{stmt}_1 \quad \text{becomes} \quad \text{cmp} \_\text{LT} \quad r_x, r_y \quad \Rightarrow \quad r_1 \\
\text{else } \text{stmt}_2 & \quad \text{becomes} \quad \text{cmp} \_\text{LT} \quad r_x, r_y \quad \Rightarrow \quad r_1 \\
& \quad \text{cbr} \quad r_1 \quad \rightarrow \quad \text{L} \_\text{stmt}_1, \text{L} \_\text{stmt}_2
\end{align*}
\]
Boolean & Relational Values

When compare produces a condition code

- Must use a branch to interpret result of compare
- Necessitates branches in the evaluation

Example

\[ x < y \text{ becomes } \]

\[
\text{cmp } r_x, r_y \Rightarrow CC_1
\]
\[
cbr_{LT} \quad CC_1 \quad \Rightarrow \quad L_T, L_F
\]
\[
L_T: \quad \text{load} I \quad \text{TRUE} \quad \Rightarrow \quad r_2
\]
\[
\quad \text{br} \quad \quad \quad \Rightarrow \quad L_E
\]
\[
L_F: \quad \text{load} I \quad \text{FALSE} \quad \Rightarrow \quad r_2
\]
\[
\quad \text{br} \quad \quad \quad \Rightarrow \quad L_E
\]
\[
L_E: \quad \ldots \text{other statements} \ldots
\]

If \textbf{cmp} sets a condition-code register, rather than returning a Boolean value, the implementation of \( x < y \) becomes much more complex.
Boolean & Relational Values

When compare produces a condition code

- Must use a branch to interpret result of compare
- Necessitates branches in the evaluation

Example

if \( (x < y) \)
then \( stmt_1 \) becomes
else \( stmt_2 \)

\[
\begin{align*}
\text{if } (x < y) & \quad \text{cmp } r_x, r_y \Rightarrow CC_1 \\
\text{then } stmt_1 & \quad \text{cbr}_{\text{LT}} CC_1 \rightarrow L_T, L_F \\
\text{else } stmt_2 & \quad \\
L_T: & \quad stmt_1 \\
\text{br} & \quad \rightarrow L_E \\
L_F: & \quad stmt_2 \\
\text{br} & \quad \rightarrow L_E \\
L_E: & \quad \ldots \text{other statements} \ldots
\end{align*}
\]

If the sole use of the relational expression’s value is to control an if-then-else construct, then we can fold the statements into the evaluation.
Boolean & Relational Values

When compare produces a condition code:

- Must use a branch to interpret result of compare
- Necessitates branches in the evaluation

Example

If the sole use of the relational expression’s value is to control an if-then-else construct, then we can fold the statements into the evaluation.

```
if (x < y)
  then stmt₁
  becomes
  cmp rₓ, rᵧ  ⇒  CC₁
  cbr_LT CC₁  →  L_T, L_F

L_T:  stmt₁
  br    →  L_E

L_F:  stmt₂
  br    →  L_E

L_E:  ... other statements ...
```

Here, x < y is true
Here, it is false

This code represents the result of the comparison implicitly, in the PC

This idea works on most ISAs and in many circumstances.
Other Ops That Interpret A Comparison

ISAs sometimes include other ops that can interpret a comparison

**Conditional Move Operation**
- Chooses its argument based on result of the comparison
- Can envision either a Boolean or a CC-based version
  Avoids a disruptive branch

**Predicated Instructions**
- Allows a Boolean value to control execution of a single operation
- Typically, controls whether result is assigned or discarded
  Avoids a disruptive branch

Branch on a compare may be hard to predict
Implementing **IF-THEN-ELSE**

**Example**

```plaintext
if (x < y)
then a ← c + d
else a ← e + f
```

Both avoid branches and lead to shorter code. Note, however, that the path lengths through the code are about the same.

**Bottom line:** your mileage may vary. The best implementation depends on specific details of the ISA.

<table>
<thead>
<tr>
<th>Conditional Move</th>
<th>Predicated Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>comp</code></td>
<td><code>cmp_LT</code></td>
</tr>
<tr>
<td><code>r_x, r_y</code></td>
<td><code>r_x, r_y</code></td>
</tr>
<tr>
<td><code>⇒ CC_1</code></td>
<td><code>⇒ r_1</code></td>
</tr>
<tr>
<td><code>add</code></td>
<td><code>(r_1)</code></td>
</tr>
<tr>
<td><code>r_c, r_d</code></td>
<td><code>add</code></td>
</tr>
<tr>
<td><code>⇒ r_1</code></td>
<td><code>r_c, r_d</code></td>
</tr>
<tr>
<td><code>add</code></td>
<td><code>(r_1)</code></td>
</tr>
<tr>
<td><code>r_e, r_f</code></td>
<td><code>add</code></td>
</tr>
<tr>
<td><code>⇒ r_2</code></td>
<td><code>r_e, r_f</code></td>
</tr>
<tr>
<td><code>i2i_LT</code></td>
<td><code>(r_1)</code></td>
</tr>
<tr>
<td><code>CC_1, r_1, r_2</code></td>
<td><code>add</code></td>
</tr>
<tr>
<td><code>⇒ r_a</code></td>
<td><code>r_a</code></td>
</tr>
</tbody>
</table>

**Encodes x < y in CC_1, but never creates a Boolean value**

**Creates an Boolean value for x < y**

COMP 412, Fall 2018
Generating a branching **IF-THEN-ELSE** with **SDT**

To generate branching code with **SDT**, need to create & track the labels

1. Create labels for “then part”, “else part”, and “exit”
   - Generate three labels and push them onto the stack

2. Emit branch to appropriate part (then or else) after **Expr**
   - Emit the branch followed by the label for **then part** (labelled **nop**)

3. At end of **WithElse**, emit branch to the exit
   - Emit a jump to the exit label followed by the label for the **else part** (labelled **nop**)

4. At end of second **WithElse**, emit branch to the exit
   - Emit a jump to the exit label followed by the label for the **exit** statement
Generating a predicated **IF-THEN-ELSE** with **SDT**

The predicated code is easier to generate with **SDT**

1. Evaluate the controlling *Expr* into a register
   - *Processor may use one or more dedicated predicate registers*
   - *May require both the predicate and its logical complement*

2. Generate a dense stream of ops for the **then** clause
   - *Predicate each instruction with the Expr’s value*

3. Generate a dense stream of ops for the **else** clause
   - *Predicate each instruction with the logical complement of the Expr’s value*

4. Count on the instruction scheduler to interleave them.
   - *Requires the scheduler to understand the anti-dependences created by the use of a predicate and its logical complement*
   - *Adds complication to the scheduler, but it comes from predication, not from this particular code-generation scheme*

*Your performance may vary.*
Control Flow

Loops
• Compiler should follow a uniform template
• 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

General Schema for Loops
Generating a **FOR** Loop

A typical for loop might be specified as follows:

\[
\text{Stmt} \rightarrow \text{FOR NAME} = \text{Expr}_1 \text{ TO } \text{Expr}_2 \text{ BY } \text{Expr}_3 \{ \text{Stmts} \}
\]

With the usual meaning

⇒ **NAME** is an index variable that starts with \(\text{Expr}_1\) and runs to \(\text{Expr}_2\) by increments of \(\text{Expr}_3\)

- The rules for the individual statements in \(\text{Stmts}\) will emit code for the loop body
- The rule for the loop must emit code for the control structure

**while**, **until**, **repeatuntil**, & **repeatwhile** all fit the same basic template
Generating a **FOR** Loop

**How should the code look? Two obvious choices ...**

---

```plaintext
for i = 1 to 100 by 1
  { ... }
```

---

```
  i ← 1
  loop: if (i > 100) goto end
       ... loop body ...
       i ← i + 1
       goto loop
  end:   ... next statement ...
```

- Single copy of test

---

```
  i ← 1
  if (i > 100) goto end
  loop: ... loop body ...
         i ← i + 1
         if (i < 101) goto loop
         end:   ... next statement ...
```

- Two copies of test

---

- One copy version may be easier to generate
  - Can generate initialization & test before body, increment and jump after

- Two copy version may optimize better
  - Single block loop body will incorporate increment and test in one block

Converting one to the other is more complex than you might think.
See Dave Whalley’s paper in 1991 **PLDI**.
Generating a **FOR** Loop with **SDT**

### Generating the single test version of the loop

<table>
<thead>
<tr>
<th>(1)</th>
<th>(2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Stmt \rightarrow \text{FOR NAME} = \text{Expr}_1 \text{ TO } \text{Expr}_2 \text{ BY } \text{Expr}_3 { Stmts }$</td>
<td></td>
</tr>
</tbody>
</table>

(1) Use **bison**’s mid-production action to insert code after $\text{Expr}_2$

- Initialize **NAME** to $\text{Expr}_1$
  - Let **SDT** generate code to evaluate $\text{Expr}_1$ into a register
- Test **NAME** against $\text{Expr}_2$
  - Let **SDT** generate code to evaluate $\text{Expr}_2$ into a register
  - Compare against **NAME** and branch accordingly
  - Emit a label before the loop body

(2) On the reduction, emit the end-of-loop code

- Increment **NAME** by $\text{Expr}_3$
  - Let **SDT** generate code to evaluate $\text{Expr}_2$ into a register
- Jump to **loop**
Generating a **FOR** Loop with **SDT**

**Generating the two test version of the loop**

\[
Stmt \rightarrow \text{FOR } NAME = Expr_1 \text{ TO } Expr_2 \text{ BY } Expr_3 \{ \text{Stmts} \}
\]

(1) Use **bison**’s mid-production action to insert code after \(Expr_2\)

- Initialize **NAME** to \(Expr_1\)
  - Let **SDT** generate code to evaluate \(Expr_1\) into a register
- Test **NAME** against \(Expr_2\)
  - Let **SDT** generate code to evaluate \(Expr_2\) into a register
  - Compare against **NAME** and branch accordingly
  - Emit a label before the loop body

(2) On the reduction, emit the end-of-loop code

- Increment **NAME** by \(Expr_3\)
  - Let **SDT** generate code to evaluate \(Expr_2\) into a register
- Generate the other test and branch
  - This process involves evaluating some of the \(Expr\)’s multiple times
Generating a **FOR** Loop with **SDT**

**Generating the two test version of the loop**

\[ Stmt \rightarrow \text{FOR NAME} = \text{Expr}_1 \text{ TO } \text{Expr}_2 \text{ BY } \text{Expr}_3 \{ \text{Stmts} \} \]

1. **Use bison**’s mid-production action to insert code after \( \text{Expr}_2 \)
   - Initialize **NAME** to \( \text{Expr}_1 \)
     - Let **SDT** generate code to evaluate \( \text{Expr}_1 \) into a register
   - Test **NAME** against \( \text{Expr}_2 \)
     - Let **SDT** generate code to evaluate \( \text{Expr}_2 \) into a register
     - Compare against **NAME** and branch accordingly
     - Emit a label before the loop body

2. **On the reduction, emit the end-of-loop code**
   - Increment **NAME** by \( \text{Expr}_3 \)
     - Let **SDT** generate code to evaluate \( \text{Expr}_2 \) into a register
   - Generate the other test and branch
     - This process involves evaluating some of the \( \text{Expr} \)’s multiple times

In general, the compiler needs a data structure to hold the information for the loop. One way to handle the second test is to include a buffer of code in that “loop structure”. The compiler generates the test into the buffer, then copies it into each place where it is needed.
Code For a Two-Test FOR Loop

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

The two-test template produces a code shape for loops that works well for optimization. ([271] in EaC2e)

Initialization

Pre-test

Post-test

Text shows examples of other kinds of loops translated into the basic template.
Control Flow

Loops
• Compiler should follow a uniform template
• 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

Other loops follow the same basic pattern
• While loops have test on entry (two-test loop)
• Until loops have no test on entry (single test at bottom)

Optimize the translation for the case of the single-block loop

General Schema for Loops
Break statements

Many programming languages include a break statement

Exits from the innermost control-flow statement
• Out of the innermost loop
• Out of a case statement

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

break only makes sense if loop has > 1 block
skip or continue becomes branch to post-test 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

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)

Case statements are a place where attention to code shape pays off.
Case Statement Implementation

Using Linear Search

```
switch( expr ) {
    case 0:    block_0;
               break;
    case 17:   block_17;
               break;
    case 23:   block_23;
               break;
    default:   block_d;
               break;
}
```

```
Temp ← expr;
If (temp == 0)                           
then block_0;
else if (temp == 17)                     
then block_17;
else if (temp == 23)                     
then block_23;
else block_d;
```

Case Statement Implementation

- Cost of execution depends on frequency of values for `expr`
- Clear benefit to reordering cases
Case Statement Implementation

Using Binary Search

```plaintext
switch( expr ) {
    case 0:   block_0;  break;
    case 17:  block_17; break;
    ...    ...         ...  
    case 99:  block_99; break;
    default: block_d;  break;
}
```

<table>
<thead>
<tr>
<th>Value</th>
<th>Label</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LB_0</td>
</tr>
<tr>
<td>17</td>
<td>LB_17</td>
</tr>
<tr>
<td>99</td>
<td>LB_99</td>
</tr>
</tbody>
</table>

Case Statement Implementation

- Cost of execution depends on $\log_2$ of number of cases
- Separate label for default case
Case Statement Implementation

Using a Jump Table

```c
switch(expr) {
    case 0: block0; break;
    case 1: block1; break;
    case 2: block2; break;
    ...
    case 9: block9; break;
    default: block_d; break;
}
```

```
temp ← expr;

If (0 ≤ temp ≤ 9) then {
    addr ← @LT + temp * 4
    target ← Mem[addr]
    jump to target
}
else jump to LB_d
```

Cost is \(O(1)\)

Can optimize code further

If “expr” is one byte, use a full table and avoid the if-then-else
Case Statement Implementation

Using a Jump Table

temp ← expr;

If (0 ≤ temp ≤ 9) then {
    addr ← @LT + temp * 4
    target ← Mem[addr]
    jump to target
}
else jump to LB_d

Original Code

temp ← expr * 4

if (0 ≤ temp ≤ 36) then{
    offset ← @LT + temp
    addr ← Mem[ARP + offset]
    jump to addr
}
else jump to LB_d

Mapping into low-level operations (e.g., ILOC)

• We need the label, @LT, in a register
  – If case statement is in a loop, load @LT before the loop
  – Could put @LT in the AR, but that requires initializations

• Code maps cleanly into a few operations (say 3?)

Use lshift rather than mult
Fold the constants
Use an addI
Use a loadAI
Simple jump
Simple jump

COMP 412, Fall 2018
Generating a **CASE** Statement With **SDT**

The grammar for a case statement might resemble:

```
CaseStmt  →  SWITCH ( Expr ) INTO { CaseList } ;
CaseList  →  Caselist OneCase
           |       OneCase
OneCase   →  CASE LABEL : Stmt ; BREAK ;
```

The difficulty is deciding how to implement the “switch”

- Need to see the cases before making the decision
- The reductions occur in the “wrong” order
Generating a **CASE** Statement With **SDT**

1. At (1), create and preserve an **EVAL** label and an **EXIT** label, then emit a jump to the **EVAL** label

2. In each **OneCase**, create a code label, preserve it in a list of `<case label, code label>` pairs, emit the code label, and generate the **Stmt**

3. After **Stmt**, emit a jump to the **EXIT** label

4. At (2), analyze the list of labels, choose an evaluation strategy, emit the **EVAL** label, then emit the code to evaluate the **Expr** and jump to the selected case label

5. Emit the **EXIT** label

---

**CaseStmt** \(\rightarrow\) **SWITCH** (**Expr**) **INTO** \{ **CaseList** \}

**CaseList** \(\rightarrow\) **Caselist** **OneCase**

\[\text{1}\]

**OneCase** \(\rightarrow\) **CASE** **LABEL** : **Stmt** ; **BREAK** ;

\[\text{2}\]
Generating a **CASE** Statement With **SDT**

The resulting code has the following shape:

<table>
<thead>
<tr>
<th></th>
<th>Step (1)</th>
<th>Step (2)</th>
<th>Step (3)</th>
<th>Step (2)</th>
<th>Step (3)</th>
<th>Step (2)</th>
<th>Step (3)</th>
<th>Step (4)</th>
<th>Step (5)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CL₁:</strong></td>
<td>jump EVAL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>jump EXIT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>CL₂:</strong></td>
<td>code for first case</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>jump EXIT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>CLₙ:</strong></td>
<td>code for second case</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>jump EXIT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>EVAL:</strong></td>
<td>code to evaluate <em>Expr</em> to a label <strong>CLᵢ</strong> and jump to <strong>CLᵢ</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>EXIT:</strong></td>
<td>nop</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

It is trivial for the optimizer to straighten out this confused control flow. See, for example, the **CLEAN** algorithm in § 10.2.2 of **EaC2e**.
Implementing the Switch (Step 4)

When Step (4) executes in SDT, all the information is available

• Compiler has a list of the case labels and their code labels
• Compiler has access to the result of evaluating $Expr$
  – Step (4) happens on reduction by $CaseStmt$ production (at vertical line 2)
• Compiler can implement any of the three schemes for switch

Choosing the method

• Compiler can walk the list and do some analysis
  – Look at number of labels & density of labels
• Choose among if-then-else, binary search, or a jump table
Implementing the Switch (Step 4)

When Step (4) executes in SDT, all the information is available
• Compiler has a list of the case labels and their code labels
• Compiler has access to the result of evaluating $Expr$
  – Step (4) happens on reduction by $CaseStmt$ production (at vertical line 2)
• Compiler can implement any of the three schemes for switch

Using an if-then-else structure
• Compiler walks the list of <case label, code label> pairs
  – For each pair, generate a test on $Expr = $ case label, with branch to code label
  – Use the taken case for the code label & the fall through for the next case
    → Reduces average penalty
  – Handles default as the last else clause
• Creates the expected code
• An extra transfer of control per case, beyond other two methods
Implementing the Switch (Step 4)

When Step (4) executes in SDT, all the information is available
- Compiler has a list of the case labels and their code labels
- Compiler has access to the result of evaluating $Expr$
  - Step (4) happens on reduction by $CaseStmt$ production (at vertical line 2)
- Compiler can implement any of the three schemes for switch

Using a binary search
- Compiler walks the list of <case label, code label> pairs
  - Generates a sorted table of the pairs, which it statically initializes
    → Assembler directive
- Inserts either “canned” inline code or a call to library routine
  - Followed by a jump to the matching code label
- One extra transfer of control per case as price of SDT
Implementing the Switch (Step 4)

When Step (4) executes in SDT, all the information is available

- Compiler has a list of the case labels and their code labels
- Compiler has access to the result of evaluating $Expr$
  - Step (4) happens on reduction by $CaseStmt$ production (at vertical line 2)
- Compiler can implement any of the three schemes for switch

Using a jump table

- Compiler walks the list of <case label, code label> pairs
  - Generates an ordered table of code labels
- Inserts index on $Expr$ into the table, followed by jump
- One extra transfer of control per case as price of SDT
Generating a **CASE** Statement With **SDT**

**The Bottom Line**
- With some careful planning, you can generate good code for a case statement using **SDT**
- The resulting code will have extraneous **jump** operations
  - Optimization can eliminate them (branch straightening)
  - The code for the if-then-else implementation is particularly ugly

**The alternative**
- Compiler could use **SDT** to build a small **AST**
- Compiler could walk the **AST** multiple times to analyze, and then generate the right code for the case statement
The material on Boolean values and relational comparisons is drawn from § 7.4 in EaC2e. The text presents the material in much greater depth, with more examples.
You should read that section, along with the rest of Chapter 7. Lecture will not cover all of the material in Chapter 7, but you are responsible for it on the final exam.
Boolean Values

The compiler needs to represent the values TRUE and FALSE

Numerical representation

• Assign values to **TRUE** and **FALSE** (design time)
  – Common choices are
• Use hardware **AND**, **OR**, and **NOT** operations
• Use comparison to get a Boolean from a relational expression

Different instruction sets (ISA) provide different kinds of support

• Comparison returns Boolean & branch uses Boolean
• Comparison sets a condition code & branch reads a condition code
• Comparison returns Boolean & operations are predicated
• **ISA** provides conditional move operation (comparison + predicate?)

Implementation of Boolean values and relational comparisons depends heavily on features of the **ISA**.
Implementing **IF-THEN-ELSE**

**Example**

if (x < y)
then a ← c + d
else a ← e + f

No significant difference in the quality of the code — same basic set of operations

<table>
<thead>
<tr>
<th>Condition Codes</th>
<th>Boolean Comparisons</th>
</tr>
</thead>
<tbody>
<tr>
<td>comp r_x, r_y</td>
<td>cmp_LT r_x, r_y</td>
</tr>
<tr>
<td>cbr_LT CC_1</td>
<td>cbr</td>
</tr>
<tr>
<td>L_1: add r_c, r_d</td>
<td>L_1: add r_c, r_d</td>
</tr>
<tr>
<td>L_2: add r_e, r_f</td>
<td>L_2: add r_e, r_f</td>
</tr>
<tr>
<td>L_OUT: nop</td>
<td>L_OUT: nop</td>
</tr>
</tbody>
</table>

**Encodes x < y in CC_1, but never creates a Boolean value**

**Creates an Boolean value for x < y**
IF-THEN-ELSE Statements

Predicate creates an alternative strategy

- Evaluate the conditional and predicate the “then” & “else” blocks
- Depending on (1) the number & distribution of ops under L1 and L2, and (2) the branch latency, one strategy or the other will be better

Assume two functional units

<table>
<thead>
<tr>
<th>if  Expr</th>
<th>Unit 1</th>
<th>Unit 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>then Stmt₁</td>
<td></td>
<td></td>
</tr>
<tr>
<td>else Stmt₂</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L1: op1 op2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>jump to L3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L2: op3 op4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>jump to L3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L3: nop</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Unit 1</th>
<th>Unit 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>compare ⇒ r₁</td>
<td></td>
</tr>
<tr>
<td>(r₁)</td>
<td>op1</td>
</tr>
<tr>
<td>(!r₁)</td>
<td>op3</td>
</tr>
<tr>
<td>L3:</td>
<td>nop</td>
</tr>
</tbody>
</table>
### IF-THEN-ELSE Statements

#### The tradeoffs
- Using control-flow makes better use of the functional units (issued ops are used)
- Using control-flow takes a potential hit on the branch & jump latency
- Using predication avoids the branch latency
- Using predication leads to operations that are issued but not used

#### Your mileage will vary
- Best choice depends on number & balance between ops in then and else parts
- At some point, there are enough ops to cover the branch & jump latencies

```
<table>
<thead>
<tr>
<th>Unit 1</th>
<th>Unit 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>compare &amp; branch</td>
</tr>
<tr>
<td>L1:</td>
<td>op1</td>
</tr>
<tr>
<td>jump to L3</td>
<td></td>
</tr>
<tr>
<td>L2:</td>
<td>op3</td>
</tr>
<tr>
<td>jump to L3</td>
<td></td>
</tr>
<tr>
<td>L3:</td>
<td>nop</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Unit 1</th>
<th>Unit 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>compare ⇒ r1</td>
</tr>
<tr>
<td>(r1)</td>
<td>op1</td>
</tr>
<tr>
<td>(!r1)</td>
<td>op3</td>
</tr>
<tr>
<td>L3:</td>
<td>nop</td>
</tr>
</tbody>
</table>
```

If `Expr` then `Stmt_1` else `Stmt_2`