Chapter 6.4
Outline

- Local flow of control
- Conditionals
- Switch
- Loops
## Machine Code Assumptions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>GOTO Label</td>
<td>Jump to Label</td>
</tr>
<tr>
<td>GOTO label register</td>
<td>Indirect jump</td>
</tr>
<tr>
<td>IF condition register then GOTO Label</td>
<td>Conditional Jump</td>
</tr>
<tr>
<td>IF not condition register then GOTO Label</td>
<td></td>
</tr>
</tbody>
</table>
Boolean Expressions

• In principle behave like arithmetic expressions
• But are treated specially
  – Different machine instructions
  – Used in control flow instructions
  – Shortcut computations
  – Negations can be performed at compile-time

if (a < b) goto l

Code for $a < b$ yielding a condition value
Conversion condition value into Boolean
Conversion from Boolean in condition value
Jump to l on condition value
Shortcut computations

• Languages such as C define shortcut computation rules for Boolean
• Incorrect translation of e₁ && e₂

Code to compute e₁ in loc₁
Code to compute e₂ in loc₂
Code for && operator on loc₁ and loc₂
Location Computation

• The result of a Boolean expression is pair of locations in the generated code
  – The **true** location corresponds to the target instruction when the condition holds
  – The **false** location corresponds to the target instruction when the condition does not hold
Code for e1 \&\& e2

Code for e1

if e1 then goto L11
goto L12

L11:

Code for \&\&

Code for e2

if e2 then goto L21
goto L22
Code for Booleans
(Location Computation)

• Top-Down tree traversal
• Generate code sequences instructions
• Jump to a designated ‘true’ label when the Boolean expression evaluates to 1
• Jump to a designated ‘false’ label when the Boolean expression evaluates to 0
• The true and the false labels are passed as parameters
Example

if ((a==0) && (b > 5))

x = ((7 * a) + b)
if

&

if

No label

Lt

Lt

Lf

Lf

Lf

Lt:

:=

Lt:

Lf:

:=

Lt:

Lt:

Lf:

:=

Cmp_Constant R0, 0

IF NOT EQ THEN GOTO Lf

Load_Local ‘a’, R0

Cmp_Constant R0, 5

IF GT THEN GOTO Lt

GOTO Lf

Cmp_Constant R0, 5

IF GT THEN GOTO Lt

GOTO Lf

Cmp_Constant R0, 5

IF GT THEN GOTO Lt

GOTO Lf

Load_Local ‘b’, R0

7

a

x

+ b

Code for :=
Procedure  Generate Code for Boolean control expression (Node, True label, False label) :

SELECT Node .type:
    CASE Comparison type:  // <, >, ==, etc. in C
        Generate code for comparison expression (Node .expr);
        // The comparison result is now in condition register
        IF True label /= No label:
            Emit(“If condition register THEN GOTO True label);
        IF False label /= No label:
            Emit(“GOTO False label);
        ELSE True label == No label:
            IF False label /= No label:
                Emit(“IF NOT condition register THEN GOTO”, False label);
    CASE Lazy and type:                                                  // && in C
        Generate code for Boolean control expression
        Node .left, No Label, False label);
        Generate code for Boolean control expression
        Node .right, True label, False label);
    CASE Negation type:                                                  // ! in C
        Generate code for Boolean control expression
        Node .arg, False label, True label);
    …
Code generation for IF

Allocate two new labels Lf, Lend

Generate code for Boolean(left, 0, Lf)

Code for Boolean with jumps to Lf

Code for Boolean with jumps to Lf

GOTO Lend

Lf:

Lend:

Boolean expression
true sequence
false sequence

Code for Boolean with jumps to Lf
Code for true sequence
Code for false sequence
Code generation for IF (no-else)

Allocate new label Lend

Generate code for Boolean(left, 0, Lend)

if

Boolean expression

true sequence

Lend:

Code for Boolean with jumps to Lend

Code for true sequence
Coercions into value computations

\[
\begin{align*}
\text{:=} & \quad \text{Generate new label Lf} \\
\text{Load\_Constant R0, 0;} & \quad \text{Generate code for Boolean(right, 0, Lf)} \\
\text{Load\_Local 'a', R1;} & \quad \text{CMP R1, 'b';} \\
\text{IF } <= \text{ GOTO Lf;} & \\
\text{Load\_Constant R0, 1} & \\
\text{Lf: Store\_Local R0, 'x'} &
\end{align*}
\]
Effects on performance

• Number of executed instructions
• Unconditional vs. conditional branches
• Instruction cache
• Branch prediction
• Target look-ahead
Code for case statements

- Three possibilities
  - Sequence of IFs
    - $O(n)$ comparisons
  - Jump table
    - $O(1)$ comparisons
  - Balanced binary tree
    - $O(\log n)$ comparisons

- Performance depends on $n$
- Need to handle runtime errors
Simple Translation

tmp_case_value := case expression;
IF tmp_case_value = l_1 THEN GOTO label_1;
IF tmp_case_value = l_2 THEN GOTO label_2;
...
IF tmp_case_value = l_n THEN GOTO label_n;
GOTO label_else; // or insert the code at label else

label 1:
    Code for statement sequence_1
    GOTO label_next;

label 2:
    Code for statement sequence_2
    GOTO label_next;
...

label n:
    Code for statement sequence_n
    GOTO label_next;

label else:
    Code for else-statement sequence
tmp_case_value := case expression;
IF tmp_case_value = 1 THEN GOTO label_1;
...
IF tmp_case_value = n THEN GOTO label_n;
GOTO label_else; // or insert the code at label_else

label_1:
  code for statement sequence_1
GOTO label_next;
...

label_n:
  code for statement sequence_n
GOTO label_next;

label_else:
  code for else-statement sequence

label_next:
Jump Table

• Generate a table of \( L_{\text{high}} - L_{\text{low}} + 1 \) entries
  – Filled at ?time

• Each entry contains the start location of the corresponding case or a special label

• Generated code

\[
\begin{align*}
\text{tmp\_case\_value:= case expression;} \\
\text{if tmp\_case\_value < } L_{\text{low}} \text{ GOTO label\_else;} \\
\text{if tmp\_case\_value > } L_{\text{high}} \text{ GOTO label\_else;} \\
\text{GOTO table[tmp\_case\_value − } L_{\text{low}} ];
\end{align*}
\]
Balanced trees

- The jump table may be inefficient
  - Space consumption
  - Cache performance
- Organize the case labels in a balanced tree
  - Left subtrees smaller labels
  - Right subtrees larger labels
- Code generated for node_k

  label_k: IF tmp_case_value < l_k THEN
          GOTO label of left branch;
  IF tmp_case_value >l_k THEN
          GOTO label of right branch;
  code for statement sequence;
  GOTO label_next;
Repetition Statements (loops)

- Similar to IFs
- Preserve language semantics
- Performance can be affected by different instruction orderings
- Some work can be shifted to compile-time
  - Loop invariant
  - Strength reduction
  - Loop unrolling
while statements

Generate new labels test_label, $L_{\text{end}}$

test_label: while Boolean expression

Generate code for Boolean(left, 0, $L_{\text{end}}$)

GOTO test_label;

$L_{\text{end}}$: statement

Sequence

Code for Boolean with jumps to $L_{\text{end}}$

statement sequence
while statements(2)

Generate labels test_label, Ls

GOTO test_label: while

Generate code for Boolean(left, Ls, 0)

Ls:

Code for Boolean with jumps to $L_S$

test_label:

Boolean expression

Code for statement sequence

statement

Sequence
For-Statements

• Special case of while
• Tricky semantics
  – Number of executions
  – Effect on induction variables
  – Overflow
Simple-minded translation

FOR i in lower bound .. upper bound DO
    statement sequence
END for

\[\downarrow\]

i := lower_bound;
tmp_ub := upper_bound;
WHILE i <= tmp_ub DO
    code for statement sequence
    i := i + 1;
END WHILE
Correct Translation

FOR i in lower bound .. upper bound DO
  statement sequence
END for

↓

i := lower_bound;

tmp_ub := upper_bound;
IF i > tmp_ub THEN GOTO end_label;

loop_label:
  code for statement sequence
  if (i==tmp_ub) GOTO end_label;

i := i + 1;

GOTO loop_label;

end_label:
for (exp1; exp2; exp3) {
    body;
}

exp1;
while (exp2) {
    body;
    exp3;
}
Summary

• Handling control flow statements is usually simple

• Complicated aspects
  – Routine invocation
  – Non local gotos

• Runtime profiling can help