Code Generation 2

Roadmap (Where are we?)

Last lecture
- Code generation
  - Code shape
  - Expressions
  - Assignments

This lecture
- Code generation
  - Arrays
  - Boolean and relational values
  - Control flow
How does the compiler handle $A[i,j]$?

First, must agree on a storage scheme

**Row-major order** (most languages)
- Lay out as a sequence of consecutive rows
- Rightmost subscript varies fastest
- $A[1,1], A[1,2], A[1,3], A[2,1], A[2,2], A[2,3]$

**Column-major order** (Fortran)
- Lay out as a sequence of columns
- Leftmost subscript varies fastest
- $A[1,1], A[2,1], A[1,2], A[2,2], A[1,3], A[2,3]$

**Indirection vectors** (Java)
- Vector of pointers to pointers to ... to values
- Takes much more space, trades indirection for arithmetic
- Not amenable to analysis

---

Laying Out Arrays

The Concept

<table>
<thead>
<tr>
<th>$A$</th>
<th>1,1</th>
<th>1,2</th>
<th>1,3</th>
<th>1,4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2,1</td>
<td>2,2</td>
<td>2,3</td>
<td>2,4</td>
</tr>
</tbody>
</table>

**Row-major order**

| $A$ | 1,1 | 1,2 | 1,3 | 1,4 | 2,1 | 2,2 | 2,3 | 2,4 |

**Column-major order**

| $A$ | 1,1 | 2,1 | 1,2 | 2,2 | 1,3 | 2,3 | 1,4 | 2,4 |

**Indirection vectors**

| $A$ | 1,1 | 1,2 | 1,3 | 1,4 |
|-----|-----|-----|-----|
|     | 2,1 | 2,2 | 2,3 | 2,4 |

These have distinct & different cache behavior
Computing an Array Address

A[i]
• @A + (i - low) x sizeof(A[1])
• In general: base(A) + (i - low) x sizeof(A[1])

int A[1:10] ⇒ low is 1
Make low 0 for faster access (saves a – )

Almost always a power of 2, known at compile-time ⇒ use a shift for speed

What about A[i1, i2]?

Row-major order, two dimensions
@A + ((i1 - low1) x (high2 - low2 + 1) + i2 - low2) x sizeof(A[1])

Column-major order, two dimensions
@A + ((i2 - low2) x (high1 - low1 + 1) + i1 - low1) x sizeof(A[1])

Indirection vectors, two dimensions
*(A[i1])[i2] — where A[i1] is, itself, a 1-d array reference

This stuff looks expensive!
Lots of implicit +, -, x ops
Optimizing Address Calculation for $A[i,j]$

In row-major order, where $w = \text{sizeof}(A[1,1])$

$$@A + (i - \text{low}_1)(\text{high}_2 - \text{low}_2 + 1) \times w + (j - \text{low}_2) \times w$$

Which can be factored into

$$@A + i \times (\text{high}_2 - \text{low}_2 + 1) \times w + j \times w$$

$$- (\text{low}_1 \times (\text{high}_2 - \text{low}_2 + 1) \times w) + (\text{low}_2 \times w)$$

If $\text{low}_i$, $\text{high}_i$, and $w$ are known, the last term is a constant

Define $@A_0$ as

$$@A - (\text{low}_1 \times (\text{high}_2 - \text{low}_2 + 1) \times w + \text{low}_2 \times w$$

And $\text{len}_2$ as $(\text{high}_2 - \text{low}_2 + 1)$

Then, the address expression becomes

$$@A_0 + (i \times \text{len}_2 + j) \times w$$

Array References

What about arrays as actual parameters?

Whole arrays, as call-by-reference parameters

- Need dimension information $\Rightarrow$ build a dope vector
- Store the values in the calling sequence
- Pass the address of the dope vector in the parameter slot
- Generate complete address polynomial at each reference

Some improvement is possible

- Save $\text{len}_i$ and $\text{low}_i$ rather than $\text{low}_i$ and $\text{high}_i$
- Pre-compute the fixed terms in prologue sequence

What about call-by-value?

- Most c-b-v languages pass arrays by reference
- This is a language design issue
Array References

What about A[12] as an actual parameter?

If corresponding parameter is a scalar, it’s easy
• Pass the address or value, as needed
• Must know about both formal & actual parameter
• Language definition must force this interpretation

What is corresponding parameter is an array?
• Must know about both formal & actual parameter
• Meaning must be well-defined and understood
• Cross-procedural checking of conformability

⇒ Again, this is a language design issue

Example: Array Address Calculations in a Loop

DO J = 1, N
END DO

• Naive: Perform the address calculation twice

DO J = 1, N
    R1 = A0 + (J x lenI + I) x floatsize
    R2 = B0 + (J x lenI + I) x floatsize
    MEM(R1) = MEM(R1) + MEM(R2)
END DO
Example: Array Address Calculations in a Loop

\[
\text{DO } J = 1, N \\
\text{END DO}
\]

- **Sophisticated:** Move common calculations out of loop

\[
\begin{align*}
R1 &= I \times \text{floatsize} \\
C &= \text{len} \times \text{floatsize} \quad ! \text{Compile-time constant} \\
R2 &= \text{@A_0} + R1 \\
R3 &= \text{@B_0} + R1 \\
\text{DO } J = 1, N \\
\quad a &= J \times c \\
\quad R4 &= R2 + a \\
\quad R5 &= R3 + a \\
\quad \text{MEM}(R4) &= \text{MEM}(R4) + \text{MEM}(R5) \\
\text{END DO}
\end{align*}
\]

- **Very sophisticated:** Convert multiply to add (Operator Strength Reduction)

\[
\begin{align*}
R1 &= I \times \text{floatsize} \\
C &= \text{len} \times \text{floatsize} \quad ! \text{Compile-time constant} \\
R2 &= \text{@A_0} + R1 \\
R3 &= \text{@B_0} + R1 \\
\text{DO } J = 1, N \\
\quad R2 &= R2 + c \\
\quad R3 &= R3 + c \\
\quad \text{MEM}(R2) &= \text{MEM}(R2) + \text{MEM}(R3) \\
\text{END DO}
\end{align*}
\]
Boolean & Relational Values

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

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

Choice depends
• context
• instruction set architecture (ISA)

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

\[ x < y \quad \text{becomes} \quad \text{cmp}_{LT} \ r_x, r_y \Rightarrow r_1 \]

if \ (x < y) \nthen stmt_1 \quad \text{becomes} \quad \text{cmp}_{LT} \ r_x, r_y \Rightarrow r_1 
else stmt_2 \quad \text{becomes} \quad \text{cbr} \quad r_1 \rightarrow \_stmt_1, \_stmt_2 \]
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:

\[
\begin{align*}
\text{cmp} & \quad r_x, r_y \Rightarrow cc_1 \\
\text{cbr}_{\text{LT}} & \quad cc_1 \rightarrow L_T, L_F \\
x < y & \text{ becomes } \begin{cases} 
L_T: & \text{loadl } 1 \Rightarrow r_2 \\
\text{br} & \rightarrow L_E \\
L_F: & \text{loadl } 0 \Rightarrow r_2 \\
L_E: & \text{...other stmts...}
\end{cases}
\end{align*}
\]

This "positional representation" is much more complex

---

Boolean & Relational Values

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><strong>VARIATIONS ON THE ILBR STRUCTURE</strong></th>
<th><strong>Straight Condition Codes</strong></th>
<th><strong>Boolean Compares</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Example</strong></td>
<td>( \text{comp} ) ( r_x, r_y \Rightarrow cc_1 )</td>
<td>( \text{cmp}_{\text{LT}} ) ( r_x, r_y \Rightarrow r_1 )</td>
</tr>
<tr>
<td><strong>Example</strong></td>
<td>( \text{cbr}_{\text{LT}} ) ( cc_1 \rightarrow L_T, L_F )</td>
<td>( \text{cbr} ) ( r_1 \rightarrow L_T, L_F )</td>
</tr>
<tr>
<td><strong>Example</strong></td>
<td>( L_T: ) ( \text{add} ) ( r_x, r_y \Rightarrow r_a )</td>
<td>( L_T: ) ( \text{add} ) ( r_o, r_d \Rightarrow r_a )</td>
</tr>
<tr>
<td><strong>Example</strong></td>
<td>( \text{br} ) ( \rightarrow L_{\text{OUT}} )</td>
<td>( \text{br} ) ( \rightarrow L_{\text{OUT}} )</td>
</tr>
<tr>
<td><strong>Example</strong></td>
<td>( L_F: ) ( \text{add} ) ( r_x, r_y \Rightarrow r_a )</td>
<td>( L_F: ) ( \text{add} ) ( r_o, r_d \Rightarrow r_a )</td>
</tr>
<tr>
<td><strong>Example</strong></td>
<td>( \text{br} ) ( \rightarrow L_{\text{OUT}} )</td>
<td>( \text{br} ) ( \rightarrow L_{\text{OUT}} )</td>
</tr>
<tr>
<td><strong>Example</strong></td>
<td>( L_{\text{OUT}}: ) ( \text{nop} )</td>
<td>( L_{\text{OUT}}: ) ( \text{nop} )</td>
</tr>
</tbody>
</table>

Condition code version does not directly produce \((x < y)\)

Boolean version does

Still, there is no significant difference in the code produced
Conditional move & predication both simplify this code

Both versions avoid the branches
Both are shorter than CCs or Boolean-valued compare
Are they better? What about power?

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

### Other Architectural Variations

<table>
<thead>
<tr>
<th>Conditional Move</th>
<th>Predicated Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>comp r_x r_y := CC_1</td>
<td>cmp_LT r_x r_y := r_1</td>
</tr>
<tr>
<td>add r_c r_d := r_1</td>
<td>(r_1)? add r_c r_d := r_a</td>
</tr>
<tr>
<td>add r_a r_2 := r_2</td>
<td>(¬r_1)? add r_a r_1 := r_a</td>
</tr>
<tr>
<td>i2i_&lt;</td>
<td>cc_1, r_1, r_2 := r_a</td>
</tr>
</tbody>
</table>

### Variations on the ILOC Branch Structure

<table>
<thead>
<tr>
<th>Straight Condition Codes</th>
<th>Boolean Compare</th>
</tr>
</thead>
<tbody>
<tr>
<td>comp r_x r_y := CC_1</td>
<td>cmp_LT r_x r_y := r_1</td>
</tr>
<tr>
<td>cbr_LT cc_1 → L_1, L_2</td>
<td>cmp_LT r_c r_d := r_2</td>
</tr>
<tr>
<td>L_1: comp r_c r_d := CC_2</td>
<td>and r_1, r_2 := r_a</td>
</tr>
<tr>
<td>cbr_LT cc_2 → L_3, L_2</td>
<td></td>
</tr>
<tr>
<td>L_2: loadI 0 := r_x</td>
<td></td>
</tr>
<tr>
<td>br → L_OUT</td>
<td></td>
</tr>
<tr>
<td>L_3: loadI 1 := r_x</td>
<td></td>
</tr>
<tr>
<td>br → L_OUT</td>
<td></td>
</tr>
<tr>
<td>L_OUT: nop</td>
<td></td>
</tr>
</tbody>
</table>

Potential Problem: Optimized boolean expression

Here, the boolean compare produces much better code
Short Circuiting

- Function of programming language semantics
  - Is evaluating all terms of boolean expression required?
  - If not, can stop evaluation once value is established
    - Eliminates unnecessary work
  - Examples
    - (false AND ...) is false
    - (true OR ...) is true

- May be required by some programming languages
  - Java Example
    - if ( (x != null) && x.isGood() ) { ... }
  - Error if x is null and expression evaluation not short circuited

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++) {

next statement

Initialization

Pre-test

Post-test

next statement
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

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
   (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)
Generating Stack Code

• Boolean expressions
  → Leave 1 on stack if true
  → Leave 0 on stack if false

E₁ == E₂
{ E₁
  E₂
  if_icmpeq L₁
  iconstant 0
  goto L₂
  L₁:
  iconstant 1
  L₂:
}

E₁ &amp; E₂
{ E₁
  dup
  ifeq L₁
  pop
  E₂
  L₁:
}

E₁ || E₂
{ E₁
  dup
  ifne L₁
  pop
  E₂
  L₁:
}

E₁ < E₂
{ E₁
  E₂
  if_icmplt L₁
  iconstant 0
  goto L₂
  L₁:
  iconstant 1
  L₂:
}

Generating Stack Code

• Control structures
  → Look for 1 or 0 on top of stack

x = E
{ E
  istore addr(x)
}

if ( E ) S
{ E
  ifeq L
  S
  L:
}

while ( E ) S
{ L1:
  E
  ifeq L2
  S
  goto L1
  L2:
}