Code Generation Part I

Chapter 9

Position of a Code Generator in the Compiler Model

Code Generation

• Code produced by compiler must be correct
  – Source-to-target program transformation should be \textit{semantics preserving}
• Code produced by compiler should be of high quality
  – Effective use of target machine resources
  – Heuristic techniques should be used to generate good but suboptimal code, because generating optimal code is undecidable
Target Program Code

- The back-end code generator of a compiler may generate different forms of code, depending on the requirements:
  - Absolute machine code (executable code)
  - Relocatable machine code (object files for linker)
  - Assembly language (facilitates debugging)
  - Byte code forms for interpreters (e.g. JVM)

The Target Machine

- Implementing code generation requires thorough understanding of the target machine architecture and its instruction set

  - Our (hypothetical) machine:
    - Byte-addressable (word = 4 bytes)
    - Has $n$ general purpose registers $R_0, R_1, \ldots, R_{n-1}$
    - Two-address instructions of the form $\text{op source, destination}$

The Target Machine: Op-codes and Address Modes

- Op-codes ($\text{op}$), for example
  - $\text{MOV}$ (move content of source to destination)
  - $\text{ADD}$ (add content of source to destination)
  - $\text{SUB}$ (subtract content of source from dest.)

- Address modes

<table>
<thead>
<tr>
<th>Mode</th>
<th>Form</th>
<th>Address</th>
<th>Added Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Absolute</td>
<td>H</td>
<td>R</td>
<td>1</td>
</tr>
<tr>
<td>Register</td>
<td>R</td>
<td>R</td>
<td>0</td>
</tr>
<tr>
<td>Indexed</td>
<td>c(R)</td>
<td>c+content(R)</td>
<td>1</td>
</tr>
<tr>
<td>Indirect register</td>
<td>*c</td>
<td>contents(R)</td>
<td>0</td>
</tr>
<tr>
<td>Indirect indexed</td>
<td>*c(R)</td>
<td>contents(c+contents(R))</td>
<td>1</td>
</tr>
<tr>
<td>Literal</td>
<td>#c</td>
<td>N/A</td>
<td>1</td>
</tr>
</tbody>
</table>
Instruction Costs

- Machine is a simple, non-super-scalar processor with fixed instruction costs
- Realistic machines have deep pipelines, I-cache, D-cache, etc.
- Define the cost of instruction
  \[ \text{Cost} = 1 + \text{cost(source-mode)} + \text{cost(destination-mode)} \]

Examples

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operation</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV R0, R1</td>
<td>Store content(R0) into register R1</td>
<td>1</td>
</tr>
<tr>
<td>MOV R0, M</td>
<td>Store content(R0) into memory location M</td>
<td>2</td>
</tr>
<tr>
<td>MOV R1, R0</td>
<td>Store content(R1) into register R0</td>
<td>2</td>
</tr>
<tr>
<td>MOV 4(R0), M</td>
<td>Store contents(4+content(R0)) into M</td>
<td>3</td>
</tr>
<tr>
<td>MOV *4(R0), M</td>
<td>Store contents(content(4+content(R0))) into M</td>
<td>3</td>
</tr>
<tr>
<td>MOV #1, R0</td>
<td>Store 1 into R0</td>
<td>2</td>
</tr>
<tr>
<td>ADD 4(R0), *12(R1)</td>
<td>Add content(4+content(R0)) to content(12+content(R1))</td>
<td>3</td>
</tr>
</tbody>
</table>

Instruction Selection

- Instruction selection is important to obtain efficient code
- Suppose we translate three-address code

```plaintext
x := y + z
```

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operation</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV y, R0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD y, R0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MOV R0, x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a := a + 1</td>
<td>MOV a, R0</td>
<td></td>
</tr>
<tr>
<td>ADD #1, R0</td>
<td>MOV R0, a</td>
<td></td>
</tr>
<tr>
<td>INC a</td>
<td>Cost = 6</td>
<td></td>
</tr>
<tr>
<td>Cost = 3</td>
<td>Cost = 2</td>
<td></td>
</tr>
</tbody>
</table>

Better

Add #1, a

Best

INC a

Cost = 2
Instruction Selection: Utilizing Addressing Modes

- Suppose we translate $a := b + c$ into
  
  MOV b, R0  
  ADD c, R0  
  MOV R0, a

- Assuming addresses of $a$, $b$, and $c$ are stored in $R0$, $R1$, and $R2$
  
  MOV *R1, *R0  
  ADD *R2, *R0

- Assuming $R1$ and $R2$ contain values of $b$ and $c$
  
  ADD R2, R1  
  MOV R1, a

Need for Global Machine-Specific Code Optimizations

- Suppose we translate three-address code
  
  $x := y + z$  
  
  to:  
  
  MOV y, R0  
  ADD z, R0  
  MOV R0, x

- Then, we translate
  
  $a := b + c$  
  $d := a + e$  
  
  to:  
  
  MOV a, R0  
  ADD b, R0  
  MOV R0, a  
  MOV a, R0  
  ADD w, R0  
  MOV R0, d

  Redundant

Register Allocation and Assignment

- Efficient utilization of the limited set of registers is important to generate good code

- Registers are assigned by
  
  - *Register allocation* to select the set of variables that will reside in registers at a point in the code
  
  - *Register assignment* to pick the specific register that a variable will reside in

- Finding an optimal register assignment in general is NP-complete
Example

\[
\begin{align*}
  t &= a \times b \\
  t &= t + a \\
  t &= t / d \\
  (R1 = t) &\quad \quad (R0 = a, R1 = t)
\end{align*}
\]

\[
\begin{array}{l}
  \text{MOV} \ a, R1 \\
  \text{MUL} \ b, R1 \\
  \text{ADD} \ a, R1 \\
  \text{DIV} \ d, R1 \\
  \text{MOV} \ R1, t
\end{array}
\]

Choice of Evaluation Order

- When instructions are independent, their evaluation order can be changed

\[
\begin{align*}
  t1 &= a + b \\
  t2 &= c + d \\
  t3 &= e \times t2 \\
  t4 &= t1 - t3
\end{align*}
\]

\[
\begin{array}{l}
  \text{MOV} \ a, R0 \\
  \text{ADD} \ b, R0 \\
  \text{MOV} \ R0, t1 \\
  \text{MOV} \ c, R1 \\
  \text{ADD} \ d, R1 \\
  \text{MOV} \ e, R0 \\
  \text{MUL} \ R1, R0 \\
  \text{SUB} \ R1, R0 \\
  \text{MOV} \ R0, t4
\end{array}
\]

Generating Code for Stack Allocation of Activation Records

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1 := a + b</td>
<td>100: ADD #16, SP</td>
<td>Push frame</td>
</tr>
<tr>
<td>param t1</td>
<td>108: MOV a, R0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>116: ADD b, R0</td>
<td>Store a+b</td>
</tr>
<tr>
<td>t2 := call foo, 2</td>
<td>124: MOV @R0,4(SP)</td>
<td>Store call foo address</td>
</tr>
<tr>
<td></td>
<td>132: MOV c,8(SP)</td>
<td>Store c</td>
</tr>
<tr>
<td></td>
<td>140: MOV #156, *SP</td>
<td>Store return address</td>
</tr>
<tr>
<td></td>
<td>148: GOTO 500</td>
<td>Jump to foo</td>
</tr>
<tr>
<td>func foo</td>
<td>156: MOV 12(SP), R0</td>
<td>Get return value</td>
</tr>
<tr>
<td></td>
<td>164: SUB #16, SP</td>
<td></td>
</tr>
<tr>
<td>return t1</td>
<td>172: -</td>
<td>Remove frame</td>
</tr>
<tr>
<td></td>
<td>500: MOV R0,12(SP)</td>
<td>Store return value</td>
</tr>
<tr>
<td></td>
<td>572: GOTO *SP</td>
<td>Return to caller</td>
</tr>
</tbody>
</table>

Note: Language and machine dependent
Here we assume C-like implementation with SP and no FP