Code Generation
Part I

Chapter 8
(1st ed. Ch.9)
Position of a Code Generator in the Compiler Model

Source program → Front-End → Code Optimizer → Code Generator → Target program

Intermediate code

Lexical error
Syntax error
Semantic error

Symbol Table
Code Generation

• Code produced by compiler must be correct
  – Source-to-target program transformation should be 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 $R0, R1, \ldots, R_{n-1}$
  - Two-address instructions of the form

\[ \text{op source, destination} \]
The Target Machine: Op-codes and Address Modes

• Op-codes (*op*), for example
  
  **MOV** (move content of *source* to *destination*)
  **ADD** (add content of *source* to *destination*)
  **SUB** (subtract content of *source* from *dest.*)

• Address modes

<table>
<thead>
<tr>
<th>Mode</th>
<th>Form</th>
<th>Memory Address</th>
<th>Added Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Absolute</td>
<td>M</td>
<td>M</td>
<td>1</td>
</tr>
<tr>
<td>Register</td>
<td>R</td>
<td>N/A</td>
<td>0</td>
</tr>
<tr>
<td>Indexed</td>
<td>c(R)</td>
<td>c+contents(R)</td>
<td>1</td>
</tr>
<tr>
<td>Indirect register</td>
<td>*R</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
  \[ = 1 + \text{cost}(\text{source-mode}) + \text{cost}(\text{destination-mode}) \]
## Examples

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operation</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV R0,R1</td>
<td>Store $\text{content}(\text{R0})$ into register R1</td>
<td>1</td>
</tr>
<tr>
<td>MOV R0,M</td>
<td>Store $\text{content}(\text{R0})$ into memory location M</td>
<td>2</td>
</tr>
<tr>
<td>MOV M,R0</td>
<td>Store $\text{content}(\text{M})$ into register R0</td>
<td>2</td>
</tr>
<tr>
<td>MOV 4(R0),M</td>
<td>Store contents(4+contents(R0)) into M</td>
<td>3</td>
</tr>
<tr>
<td>MOV *4(R0),M</td>
<td>Store contents(contents(4+contents(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 contents(4+contents(R0)) to value at location contents(12+contents(R1))</td>
<td>3</td>
</tr>
</tbody>
</table>
Instruction Selection

• Instruction selection is important to obtain efficient code

• Suppose we translate three-address code

\[ x := y + z \]

to:

\begin{align*}
\text{MOV } y, R0 \\
\text{ADD } z, R0 \\
\text{MOV } R0, x
\end{align*}

\[ a := a + 1 \]

\begin{align*}
\text{MOV } a, R0 \\
\text{ADD } #1, R0 \\
\text{MOV } R0, a
\end{align*}

Cost = 6

\begin{align*}
\text{ADD } #1, a \\
\text{INC } a
\end{align*}

Cost = 3

Cost = 2

Better

Best
Instruction Selection: Utilizing Addressing Modes

• Suppose we translate \( a := b + c \) into

\[
\begin{align*}
\text{MOV} & \quad \text{b}, R0 \\
\text{ADD} & \quad \text{c}, R0 \\
\text{MOV} & \quad \text{R0}, a
\end{align*}
\]

• Assuming addresses of \( a, b, \) and \( c \) are stored in \( R0, R1, \) and \( R2 \)

\[
\begin{align*}
\text{MOV} & \quad \text{*R1}, \text{*R0} \\
\text{ADD} & \quad \text{*R2}, \text{*R0}
\end{align*}
\]

• Assuming \( R1 \) and \( R2 \) contain values of \( b \) and \( c \)

\[
\begin{align*}
\text{ADD} & \quad \text{R2}, R1 \\
\text{MOV} & \quad \text{R1}, a
\end{align*}
\]


Need for Global Machine-Specific Code Optimizations

- Suppose we translate three-address code
  \[ x := y + z \]
  to:
  \[
  \begin{align*}
  &\text{MOV } y, R0 \\
  &\text{ADD } z, R0 \\
  &\text{MOV } R0, x
  \end{align*}
  \]

- Then, we translate
  \[
  \begin{align*}
  &a := b + c \\
  &d := a + e
  \end{align*}
  \]
  to:
  \[
  \begin{align*}
  &\text{MOV } b, R0 \\
  &\text{ADD } c, R0 \\
  &\text{MOV } R0, a \\
  &\text{MOV } a, R0 \\
  &\text{MOV } d, R0 \\
  &\text{ADD } e, R0 \\
  &\text{ADD } e, R0 \\
  &\text{MOV } R0, d
  \end{align*}
  \]

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

\[
\begin{align*}
t &:= a \times b \\
t &:= t + a \\
t &:= t / d
\end{align*}
\]

\[
\begin{align*}
\text{MOV} &\ a, R1 \\
\text{MUL} &\ b, R1 \\
\text{ADD} &\ a, R1 \\
\text{DIV} &\ d, R1 \\
\text{MOV} &\ R1, t
\end{align*}
\]

\[
\begin{align*}
\text{MOV} &\ a, R0 \\
\text{MOV} &\ R0, R1 \\
\text{MUL} &\ b, R1 \\
\text{ADD} &\ R0, R1 \\
\text{DIV} &\ d, R1 \\
\text{MOV} &\ R1, t
\end{align*}
\]

\[
\begin{align*}
\{ R1 = t \} \\
\{ R0 = a, R1 = t \}
\end{align*}
\]
Choice of Evaluation Order

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

```
a+b-(c+d)*e
```

```
t1:=a+b
t2:=c+d
t3:=e*t2
t4:=t1-t3
```

```
MOV a,R0
ADD b,R0
MOV R0,t1
MOV c,R1
ADD d,R1
MOV e,R0
MUL R1,R0
MOV t1,R1
SUB R0,R1
MOV R1,t4
```

```
t2:=c+d
t3:=e*t2
t1:=a+b
t4:=t1-t3
```

```
MOV c,R0
ADD d,R0
MOV e,R1
MUL R0,R1
MOV a,R0
ADD b,R0
SUB R1,R0
MOV R0,t4
```
Generating Code for Stack Allocation of Activation Records

t1 := a + b
param t1
param c
t2 := call foo,2
...

func foo
...
return t1

100: ADD #16,SP Push frame
108: MOV a,R0 Store a+b
116: ADD b,R0 Store c
124: MOV R0,4(SP) Store a+b
132: MOV c,8(SP) Store c
140: MOV #156,*SP Store return address
148: GOTO 500 Jump to foo
156: MOV 12(SP),R0 Get return value
164: SUB #16,SP Remove frame
172: ...
500: ...
564: MOV R0,12(SP) Store return value
572: GOTO *SP Return to caller

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