CSCI-GA.2130-001
Compiler Construction
Lecture 13:
Code Generation II

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
Main Tasks of Code Generator

- **Instruction selection**: choosing appropriate target-machine instructions to implement the IR statements
- **Registers allocation and assignment**: deciding what values to keep in which registers
- **Instruction ordering**: deciding in what order to schedule the execution of instructions
Principal Uses of Registers

- In many machines, operands of an instruction must be in registers.
- Registers make good temporaries.
- Registers are used to hold global values, generated in basic block and used in another.
- Registers help with run-time storage management.
Simple Code Generator

• For basic blocks
• Assume we have one choice of machine instructions
• Quick summary
  – Consider each three-address instruction in turn
  – Decide what loads are necessary to get needed operands in registers
  – Generate the loads
  – Generate the instruction itself
  – Generate store if needed

• LD reg, mem
• ST mem, reg
• OP reg, reg, reg
Simple Code Generator

• We need a data structure that tells us:
  – What program variables have their value in registers, and which registers if so.
  – Whether the memory location associated with the variable has the latest value.

• So we need two structures
  – For each available register: a register descriptor keeps track of variable names whose current value is in that register
  – For each program variable: an address descriptor keeps track of the location(s) where the current value can be found
Simple Code Generator

• Assume there are enough registers
• getReg(I) function
  – input: three-address code instruction I
  – Output: Selects register for each memory location associated with I
  – Has access to all register and variable descriptors
Simple Code Generator

For a three-address instruction such as $x = y + z$, do the following:

1. Use $getReg(x = y + z)$ to select registers for $x$, $y$, and $z$. Call these $R_x$, $R_y$, and $R_z$.

2. If $y$ is not in $R_y$ (according to the register descriptor for $R_y$), then issue an instruction $LD R_y, y'$, where $y'$ is one of the memory locations for $y$ (according to the address descriptor for $y$).

3. Similarly, if $z$ is not in $R_z$, issue and instruction $LD R_z, z'$, where $z'$ is a location for $z$.

4. Issue the instruction $ADD R_x, R_y, R_z$.

**SPECIAL CASE:** For copy instructions in the form of $x = y$ we assume $getReg$ will always choose the same register for $x$ and $y$

**Ending the basic block:** For each variable whose memory location is not up to date generate $ST x, R$ (R is the register where $x$ exists at end of the block)
Managing Register and Address Descriptors

For the instruction LD $R, x$

(a) Change the register descriptor for register $R$ so it holds only $x$.
(b) Change the address descriptor for $x$ by adding register $R$ as an additional location.

For the instruction ST $x, R$, change the address descriptor for $x$ to include its own memory location.

For an operation such as ADD $R_x, R_y, R_z$ implementing a three-address instruction $x = y + z$

(a) Change the register descriptor for $R_x$ so that it holds only $x$.
(b) Change the address descriptor for $x$ so that its only location is $R_x$. Note that the memory location for $x$ is not now in the address descriptor for $x$.
(c) Remove $R_x$ from the address descriptor of any variable other than $x$. 
Managing Register and Address Descriptors

When we process a copy statement $x = y$, after generating the load for $y$ into register $R_y$, if needed, and after managing descriptors as for all load statements

(a) Add $x$ to the register descriptor for $R_y$.
(b) Change the address descriptor for $x$ so that its only location is $R_y$. 
Example

\[ t = a - b \]
\[ u = a - c \]
\[ v = t + u \]
\[ a = d \]
\[ d = v + u \]

\[ \begin{array}{ccc}
R1 & R2 & R3 \\
\hline
\end{array} \quad \begin{array}{ccccccc}
a & b & c & d & t & u & v \\
\hline
\end{array} \]

\[ \begin{array}{c}
a \quad t \\
\hline
a, R1 & b & c & d & R2 \\
\end{array} \]

\[ t = a - b \]
LD R1, a
LD R2, b
SUB R2, R1, R2

For the instruction LD \( R, x \)

(a) Change the register descriptor for register \( R \) so it holds only \( x \).

(b) Change the address descriptor for \( x \) by adding register \( R \) as an additional location.

For an operation such as ADD \( R_x, R_y, R_z \) implementing a three-address instruction \( x = y + z \)

(a) Change the register descriptor for \( R_x \) so that it holds only \( x \).

(b) Change the address descriptor for \( x \) so that its only location is \( R_x \).
   Note that the memory location for \( x \) is not now in the address descriptor for \( x \).

(c) Remove \( R_x \) from the address descriptor of any variable other than \( x \).
Example

\[ t = a - b \]
\[ u = a - c \]
\[ v = t + u \]
\[ a = d \]
\[ d = v + u \]

\[
\begin{array}{cccccccc}
R1 & R2 & R3 & a & b & c & d & t & u & v \\
\hline
a & t & & a, R1 & b & c & d & R2 & & \\
& & & u & t & c & & & a & b & c, R3 & d & R2 & R1 \\
\end{array}
\]

\[
u = a - c
\]

LD R3, c
SUB R1, R1, R3

For the instruction LD \( R, x \)

(a) Change the register descriptor for register \( R \) so it holds only \( x \).

(b) Change the address descriptor for \( x \) by adding register \( R \) as an additional location.

For an operation such as ADD \( R_x, R_y, R_z \) implementing a three-address instruction \( x = y + z \)

(a) Change the register descriptor for \( R_x \) so that it holds only \( x \).

(b) Change the address descriptor for \( x \) so that its only location is \( R_x \). Note that the memory location for \( x \) is not now in the address descriptor for \( x \).

(c) Remove \( R_x \) from the address descriptor of any variable other than \( x \).
Example

\[
t = a - b \\
u = a - c \\
v = t + u \\
a = d \\
d = v + u
\]

**Example**

\[
\begin{array}{ccccccccc}
R1 & R2 & R3 & a & b & c & d & t & u & v \\
\hline
u & t & c & a & b & c, R3 & d & R2 & R1 & \\
\hline
u & t & v & a & b & c & d & R2 & R1 & R3 \\
\end{array}
\]

\[
v = t + u \\
ADD R3, R2, R1
\]

For an operation such as `ADD R_x, R_y, R_z` implementing a three-address instruction \( x = y + z \)

(a) Change the register descriptor for \( R_x \) so that it holds only \( x \).

(b) Change the address descriptor for \( x \) so that its only location is \( R_x \).
   Note that the memory location for \( x \) is *not* now in the address descriptor for \( x \).

(c) Remove \( R_x \) from the address descriptor of any variable other than \( x \).
Example

\[ t = a - b \]
\[ u = a - c \]
\[ v = t + u \]
\[ a = d \]
\[ d = v + u \]

<table>
<thead>
<tr>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>t</th>
<th>u</th>
<th>v</th>
</tr>
</thead>
<tbody>
<tr>
<td>u</td>
<td>t</td>
<td>v</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>R2</td>
<td>R1</td>
<td>R3</td>
</tr>
<tr>
<td>u</td>
<td>a</td>
<td>d</td>
<td>v</td>
<td>R2</td>
<td>b</td>
<td>c</td>
<td>d, R2</td>
<td>R1</td>
<td>R3</td>
</tr>
</tbody>
</table>

\[ a = d \]
LD R2, d

For the instruction LD R, x

(a) Change the register descriptor for register R so it holds only x.
(b) Change the address descriptor for x by adding register R as an additional location.

When we process a copy statement \( x = y \), after generating the load for \( y \) into register \( R_y \), if needed, and after managing descriptors as for all load statements (per rule 1):

(a) Add \( x \) to the register descriptor for \( R_y \).
(b) Change the address descriptor for \( x \) so that its only location is \( R_y \).
Example

t = a - b
u = a - c
v = t + u
a = d
d = v + u

\[ d = v + u \]
ADD R1, R3, R1

For an operation such as \( \text{ADD } R_x, R_y, R_z \) implementing a three-address instruction \( x = y + z \):

(a) Change the register descriptor for \( R_x \) so that it holds only \( x \).

(b) Change the address descriptor for \( x \) so that its only location is \( R_x \).
   Note that the memory location for \( x \) is not now in the address descriptor for \( x \).

(c) Remove \( R_x \) from the address descriptor of any variable other than \( x \).
Example

\[ t = a - b \]
\[ u = a - c \]
\[ v = t + u \]
\[ a = d \]
\[ d = v + u \]

Ending the basic block: For each variable whose memory location is not up to date
generate \( ST \ x, R \) (\( R \) is the register where \( x \) exists at end of the block)

For the instruction \( ST \ x, R \), change the address descriptor for \( x \) to include
its own memory location.
How `getReg` works?

(Example: \( x = y + z \))

If \( y \) is currently in a register, pick a register already containing \( y \) as \( R_y \). Do not issue a machine instruction to load this register, as none is needed.

If \( y \) is not in a register, but there is a register that is currently empty, pick one such register as \( R_y \).

What if neither of the above cases are feasible?
How `getReg` works?

(Example: \( x = y + z \))

Let \( R \) be a candidate register and \( v \) is one of the variables stored in \( R \).

If the address descriptor for \( v \) says that \( v \) is somewhere besides \( R \), then we are OK.

If \( v \) is \( x \), the value being computed by instruction \( I \), and \( x \) is not also one of the other operands of instruction \( I \) (\( z \) in this example), then we are OK. The reason is that in this case, we know this value of \( x \) is never again going to be used, so we are free to ignore it.

Otherwise, if \( v \) is not used later (that is, after the instruction \( I \), there are no further uses of \( v \), and if \( v \) is live on exit from the block, then \( v \) is recomputed within the block), then we are OK.

If we are not OK by one of the first two cases, then we need to generate the store instruction \( ST v, R \) to place a copy of \( v \) in its own memory location. This operation is called a *spill*.

Pick the register with the fewest number of spilled values.
Peephole Optimization

• Improvement of running time or space requirement of target program
• Can be applied to intermediate code or target code
• Peephole: is a small sliding window on a program
• Replace instructions in the peephole by faster/shorter sequence whenever possible
• May require repeated passes for best results
Peephole Optimization: Eliminating Redundant Loads/Stores

Optimization is obvious

BUT

Store instruction must not have a label (why?)
-> the load and store must be in the same basic block
Peephole Optimization: Eliminating Unreachable Code

- Unlabeled instruction immediately following an unconditional jump
- Eliminate jumps over jumps

```plaintext
if debug == 1 goto L1
goto L2
L1: print debugging information
L2:
```
```plaintext
if debug != 1 goto L2
print debugging information
L2:
```
Peephole Optimization:
Flow-of-Control Optimizations

\[ \text{goto L1} \]
\[ \ldots \]
\[ \text{L1: goto L2} \]

\[ \text{if } a < b \text{ goto L1} \]
\[ \ldots \]
\[ \text{L1: goto L2} \]

\[ \text{goto L1} \]
\[ \ldots \]
\[ \text{L1: if } a < b \text{ goto L2} \]
\[ \text{L3:} \]

\[ \text{goto L2} \]
\[ \ldots \]
\[ \text{L1: goto L2} \]

\[ \text{if } a < b \text{ goto L2} \]
\[ \ldots \]
\[ \text{L1: goto L2} \]

\[ \text{goto L1} \]
\[ \ldots \]
\[ \text{L1: if } a < b \text{ goto L2} \]
\[ \text{L3:} \]

\[ \text{if } a < b \text{ goto L2} \]
\[ \text{goto L3} \]
\[ \ldots \]
\[ \text{L3:} \]
Peephole Optimization:  
Algebraic Simplification and Reduction in Strength

• Get rid of expressions like $X = X + 0$ or  
  $X = X \times 1$

• Reduction in strength: replace  
  expensive operations with cheaper ones
  – $x^2 \rightarrow x \times x$
  – fixed point instead of floating point
  – Some multiplications with left shifts
  – …
Tree-Translation Scheme

- Method of code-generation
- Intermediate code is in the form or tree
  - replacement← template {action}
- Tree matching
- Stops when tree is reduced to one node, or no more matching can be done
<table>
<thead>
<tr>
<th></th>
<th>Action</th>
<th>Template</th>
<th>Replacement</th>
</tr>
</thead>
<tbody>
<tr>
<td>1)</td>
<td>( R_i \leftarrow C_a )</td>
<td>( { \text{LD } R_i, #a } )</td>
<td></td>
</tr>
<tr>
<td>2)</td>
<td>( R_i \leftarrow M_x )</td>
<td>( { \text{LD } R_i, x } )</td>
<td></td>
</tr>
<tr>
<td>3)</td>
<td>( M \leftarrow \frac{M_x}{R_i} )</td>
<td>( { \text{ST } x, R_i } )</td>
<td></td>
</tr>
<tr>
<td>4)</td>
<td>( M \leftarrow \frac{\text{ind}}{R_j} )</td>
<td>( { \text{ST } R_i, R_j } )</td>
<td></td>
</tr>
<tr>
<td>5)</td>
<td>( R_i \leftarrow \text{ind} )</td>
<td>( { \text{LD } R_i, a(R_j) } )</td>
<td></td>
</tr>
<tr>
<td>6)</td>
<td>( R_i \leftarrow \text{ind} )</td>
<td>( { \text{ADD } R_i, R_i, a(R_j) } )</td>
<td></td>
</tr>
<tr>
<td>7)</td>
<td>( R_i \leftarrow R_j )</td>
<td>( { \text{ADD } R_i, R_i, R_j } )</td>
<td></td>
</tr>
<tr>
<td>8)</td>
<td>( R_i \leftarrow C_1 )</td>
<td>( { \text{INC } R_i } )</td>
<td></td>
</tr>
</tbody>
</table>
\[ a[i] = b + 1 \]
a[i] = b + 1

ADD R0, R0, i(SP).
$a[i] = b + 1$

\[
\text{ind} \quad R_0 \quad M_b \quad C_1 \\
\]

\[
R_i \leftarrow M_x \quad \{ \text{LD R1, b} \} \\
\]

\[
R_1 \quad C_1 \quad \{ \text{INC R1} \} \\
\]
\[ a[i] = b + 1 \]

\[ \text{ST } *R_0, R_1 \]
So

- skim: 8.8, 8.9.3, 8.9.4, 8.9.5, 8.10, 8.11
- Read: rest of 8.6->8.9