G22.2130-001
Compiler Construction
Lecture 12:
Code Generation I

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
Requirements

- Preserve semantic meaning of source program
- Make effective use of available resources of target machine
- Code generator itself must run efficiently

Challenges

- Problem of generating optimal target program is undecidable
- Many subproblems encountered in code generation are computationally intractable
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
Design Issues of a Code Generator

Input

– three-address presentations (quadruples, triples, ...)
– Virtual machine presentations (bytecode, stack-machine, ...)
– Linear presentation (postfix, ...)
– Graphical presentation (syntax trees, DAGs, ...)
Design Issues of a Code Generator

Target program
- Instruction set architecture (RISC, CISC)
- Producing absolute machine-language program
- Producing relocatable machine-language program
- Producing assembly language programs
Design Issues of a Code Generator

Instruction Selection

The complexity of mapping IR program into code-sequence for target machine depends on:

- Level of IR (high-level or low-level)
- Nature of instruction set (data type support)
- Desired quality of generated code (speed and size)
Design Issues of a Code Generator

Register Allocation

• Selecting the set of variables that will reside in registers at each point in the program

Register Assignment

• Picking the specific register that a variable will reside in
Design Issues of a Code Generator

Evaluation Order

- Selecting the order in which computations are performed
- Affects the efficiency of the target code
- Picking a best order is NP-complete
- Some orders require fewer registers than others
Simple Target-Machine

- Load/store operations
  - \( LD \) dst, addr
  - \( ST \) x, r
- Computation operations
  - \( OP \) dst, src1, src2
- Jump operations
  - \( BR \) L
- Conditional jumps
  - \( Bcond \) r, L
- Byte addressable
- n registers: RO, Rn-1
Simple Target-Machine

- **Addressing modes**
  - variable name
  - a(r) means contents(a + contents(r))
  - *a(r) means:
    contents(contents(a + contents(r)))
  - immediate: #constant (e.g. LD R1, #100)
Simple Target-Machine

Cost

- cost of an instruction = 1 + cost of operands
- cost of register operand = 0
- cost involving memory and constants = 1
- cost of a program = sum of instruction costs
Examples

\[ X = Y - Z \]

\[
\begin{align*}
LD & \quad R1, y & \quad // \quad R1 = y \\
LD & \quad R2, z & \quad // \quad R2 = z \\
SUB & \quad R1, R1, R2 & \quad // \quad R1 = R1 - R2 \\
ST & \quad x, R1 & \quad // \quad x = R1 \\
\end{align*}
\]

\[ b = a[i] \] (8-byte elements)

\[
\begin{align*}
LD & \quad R1, i & \quad // \quad R1 = i \\
MUL & \quad R1, R1, 8 & \quad // \quad R1 = R1 \times 8 \\
LD & \quad R2, a(R1) & \quad // \quad R2 = \text{contents}(a + \text{contents}(R1)) \\
ST & \quad b, R2 & \quad // \quad b = R2 \\
\end{align*}
\]

\[ x = *p \]

\[
\begin{align*}
LD & \quad R1, p & \quad // \quad R1 = p \\
LD & \quad R2, 0(R1) & \quad // \quad R2 = \text{contents}(0 + \text{contents}(R1)) \\
ST & \quad x, R2 & \quad // \quad x = R2 \\
\end{align*}
\]
More Examples

• \( a[j] = c \)
• \( *p = y \)
• if \( X < Y \) goto L
Generating Code for Handling the Stack

Size and layout of activation records are determined by the code generator using information from symbol table.

- Saves return address at beginning of activation record of callee.
- Constants giving address of beginning of activation record of callee.
- Transfers control to target code of procedure callee.

```
ST callee.staticArea, #here + 20
BR callee.codeArea
CALL callee
```

```
BR *callee.staticArea
RETURN
```
LD    SP, #stackStart  
     code for the first procedure
HALT

ADD   SP, SP, #caller.recordSize
ST    *SP, #here + 16
BR    callee.codeArea
SUB   SP, SP, #caller.recordSize
BR    *0(SP)
Basic Blocks and Flow Graphs

• Graph presentation of intermediate code
• Nodes of the graph are called basic blocks
• Edges indicate which block follows which other block.
• The graph is useful for doing better job in:
  – Register allocation
  – Instruction selection
Basic Blocks

• Definition: maximal sequence of consecutive instructions such that
  – Flow of control can only enter the basic block from the first instruction
  – Control leaves the block only at the last instruction

• Each instruction is assigned to exactly on basic block
1) \( i = 1 \)
2) \( j = 1 \)
3) \( t1 = 10 \times i \)
4) \( t2 = t1 + j \)
5) \( t3 = 8 \times t2 \)
6) \( t4 = t3 - 88 \)
7) \( a[t4] = 0.0 \)
8) \( j = j + 1 \)
9) \( \text{if } j \leq 10 \text{ goto (3)} \)
10) \( i = i + 1 \)
11) \( \text{if } i \leq 10 \text{ goto (2)} \)
12) \( i = 1 \)
13) \( t5 = i - 1 \)
14) \( t6 = 88 \times t5 \)
15) \( a[t6] = 1.0 \)
16) \( i = i + 1 \)
17) \( \text{if } i \leq 10 \text{ goto (13)} \)
First we determine leader instructions:

1. The first three-address instruction in the intermediate code is a leader.

2. Any instruction that is the target of a conditional or unconditional jump is a leader.

3. Any instruction that immediately follows a conditional or unconditional jump is a leader.

1) \( i = 1 \)
2) \( j = 1 \)
3) \( t1 = 10 \times i \)
4) \( t2 = t1 + j \)
5) \( t3 = 8 \times t2 \)
6) \( t4 = t3 - 88 \)
7) \( a[t4] = 0.0 \)
8) \( j = j + 1 \)
9) \( \text{if } j \leq 10 \text{ goto (3)} \)
10) \( i = i + 1 \)
11) \( \text{if } i \leq 10 \text{ goto (2)} \)
12) \( i = 1 \)
13) \( t5 = i - 1 \)
14) \( t6 = 88 \times t5 \)
15) \( a[t6] = 1.0 \)
16) \( i = i + 1 \)
17) \( \text{if } i \leq 10 \text{ goto (13)} \)
Fist we determine *leader* instructions:

1. The first three-address instruction in the intermediate code is a leader.
   
   1) \( i = 1 \)
   2) \( j = 1 \)

2. Any instruction that is the target of a conditional or unconditional jump is a leader.

   3) \( t_1 = 10 \times i \)
   4) \( t_2 = t_1 + j \)
   5) \( t_3 = 8 \times t_2 \)
   6) \( t_4 = t_3 - 88 \)
   7) \( a[t_4] = 0.0 \)
   8) \( j = j + 1 \)
   9) if \( j \leq 10 \) goto (3)

3. Any instruction that immediately follows a conditional or unconditional jump is a leader.

   10) \( i = i + 1 \)
   11) if \( i \leq 10 \) goto (2)
   12) \( i = 1 \)
   13) \( t_5 = i - 1 \)
   14) \( t_6 = 88 \times t_5 \)
   15) \( a[t_6] = 1.0 \)
   16) \( i = i + 1 \)
   17) if \( i \leq 10 \) goto (13)
First we determine *leader* instructions:

1. The first three-address instruction in the intermediate code is a leader.
2. Any instruction that is the target of a conditional or unconditional jump is a leader.
3. Any instruction that immediately follows a conditional or unconditional jump is a leader.

Basic block starts with a leader instruction and stops before the following leader instruction.
ENTRY

$B_1$

\[ i = 1 \]

$B_2$

\[ j = 1 \]

$B_3$

\[ t_1 = 10 \times i \]
\[ t_2 = t_1 + j \]
\[ t_3 = 8 \times t_2 \]
\[ t_4 = t_3 - 88 \]
\[ j = j + 1 \]
\[ \text{if } j \leq 10 \text{ goto } B_3 \]

$B_4$

\[ i = i + 1 \]
\[ \text{if } i \leq 10 \text{ goto } B_2 \]

$B_5$

\[ i = 1 \]

$B_6$

\[ t_5 = i - 1 \]
\[ t_6 = 88 \times t_5 \]
\[ a[t_6] = 1.0 \]
\[ i = i + 1 \]
\[ \text{if } i \leq 10 \text{ goto } B_6 \]

EXIT
DAG Representation of Basic Blocks

• Leaves for initial values of variables (we may not know the values so we use a0, b0, ...)
• Node for each expression
• Node label is the expression operation
• Next to the node we put the variable(s) for which the node produced last definition
• Children of a node consists of nodes produce last definition of operands
Finding Local Common Subexpressions

\[
\begin{align*}
    a &= b + c \\
    b &= a - d \\
    c &= b + c \\
    d &= a - d
\end{align*}
\]

```
> a = b + c
    b = a - d
    c = b + c
    d = a - d
```

```
> a = b + c
    d = a - d
    c = d + c
```
Construct the DAG for the basic block

\[
\begin{align*}
d &= b \times c \\
e &= a + b \\
b &= b \times c \\
a &= e - d
\end{align*}
\]
Dead Code Elimination

• From the basic block DAG:
• Remove any root node that has no live variables
• Repeat until no nodes can be removed
\[a = b + c;\]
\[b = b - d\]
\[c = c + d\]
\[e = b + c\]
So

- Skim: 8.3.3, 8.5.4, 8.5.5, 8.5.6, and 8.5.7
- Read: 8.1 -> 8.5