Chapter 8

Code Generation

Outline

- Code Generation Issues
- Target language Issues
- Addresses in Target Code
- Basic Blocks and Flow Graphs

Introduction

- The final phase of a compiler is code generator
- It receives an intermediate representation (IR) with supplementary information in symbol table
- Produces a semantically equivalent target program
- Code generator main tasks:
  - Instruction selection
  - Register allocation and assignment
  - Instruction ordering

Issues in the Design of Code Generator

- The most important criterion is that it produces correct code.

1. Input to the Code generator
2. The target Program
3. Instruction Selection
4. Register Allocation
5. Evaluation order

I. Input to the code generator

- IR + Symbol table + runtime addresses
- The many choices for the IR:
  - Three-address representations such as quadruples, triples, indirect triples
  - virtual machine representations such as byte codes and stack-machine code
  - Linear representations such as postfix notation
  - Graphical representations such as syntax trees and DAG's.
- We assume Syntactic and semantic errors have been already detected, type checking has taken place, and that type conversion operators have been inserted wherever necessary.

II. The target program

- Common target architectures are: RISC, CISC and Stack based machines
- We use a very simple RISC-like computer with addition of some CISC-like addressing modes

III. Instruction Selection

The code generator must map the IR program into a code sequence that can be executed by the target machine. The complexity of performing this mapping is determined by a factors such as

- the level of the IR
- the nature of the instruction-set architecture
- the desired quality of the generated code.

x=y+z

```
LD      R0, y
ADD     R0, R0, z
ST      x, R0
```

a=b+c
d=a+e

```
LD      R0, b
ADD     R0, R0, c
ST      a, R0
LD      R0, a
ADD     R0, R0, e
ST      d, R0
```

Consider an example

a=a+1

```
LD  R0, a          // R0 = a
ADD R0, R0, #1     // R0 = R0 + 1
ST  a, R0          // a = R0
```

IV. Register allocation

- Use of registers
  - Register allocation: selecting the set of variables that will reside in registers at each point in the program
  - Register assignment: selecting specific register that a variable reside in.

V. Evaluation Order

- Reducing number of loads and stores.

A simple target machine model

We assume the following kinds of instructions are available:

- Load operations: LD r, x and LD r1, r2
- Store operations: ST x,r
- Computation operations: OP dst, src1, src2
- Unconditional jumps: BR L
- Conditional jumps: Bcond r, L like BLTZ r, L

Addressing Modes

- In instructions, a location can be a variable name x referring to the memory location that is reserved for x (that is, the l-value of x).

Addressing modes(contd..)

- A location can also be an indexed address of the form a(r), where a is a variable and r is a register.
- The memory location denoted by a(r) is computed by taking the l-value of a and adding to it the value in register r.
- For example, the instruction LD R1, a(R2) has the effect of setting RI = contents (a + contents (R2)).
- This addressing mode is useful for accessing arrays, where a is the base address of the array (that is, the address of the first element), and r holds the number of bytes past that address we wish to go to reach one of the elements of array a.

Addressing modes(contd..)

- A memory location can be an integer indexed by a register.
- For ex- ample, LD R1, 100(R2) has the effect of setting R1 = contents(100+contents(R2)) - loading into R1 the value in the memory location obtained by adding 100 to the contents of register R2.
- This feature is useful for pointers,

Addressing modes(contd..)

- We also allow two indirect addressing modes: *r means the memory location found in the location represented by the contents of register r and *100(r) means the memory location found in the location obtained by adding 100 to the contents of r.
- For example, LD R1, * 100 (R2) has the effect of setting R1 = contents(contents(l00 + contents(R2))), that is, of loading into R1 the value in the memory location stored in the memory location obtained by adding 100 to the contents of register R2.

b = a [i]

LD R1, i **//R1 = i**

MUL R1, R1, 8 **//R1 = Rl * 8**

LD R2, a(R1) //R2=contents(a+contents(R1))

ST b, R2 **//b = R2**

a[j] = c

```
LD R1, c                    //R1 = c
LD R2, j                    // R2 = j
MUL R2, R2, 8               //R2 = R2 * 8
ST  a(R2), R1   //contents(a+contents(R2))=R1
```

x=*p

```
LD R1, p            //R1 = p
LD R2, 0(R1)        // R2 = contents(0+contents(R1))
ST  x, R2           // x=R2
```

Conditional-jump three-address instruction

If x<y goto L

```
LD R1, x            // R1 = x
LD R2, y            // R2 = y
SUB R1, R1, R2      // R1 = R1 - R2
BLTZ R1, M          // if R1 < 0 jump to M
```

costs associated with the addressing modes

- LD R0, R1 cost = 1
- LD R0, M cost = 2
- LD R1, *100(R2) cost = 2

Basic blocks and flow graphs

- A **basic block** is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end.
- The basic blocks become the nodes of a flow graph.
- A graph representation of three-address statements, called a **flow graph**.
- Nodes in the flow graph represent computations, and the edges represent the flow of control.

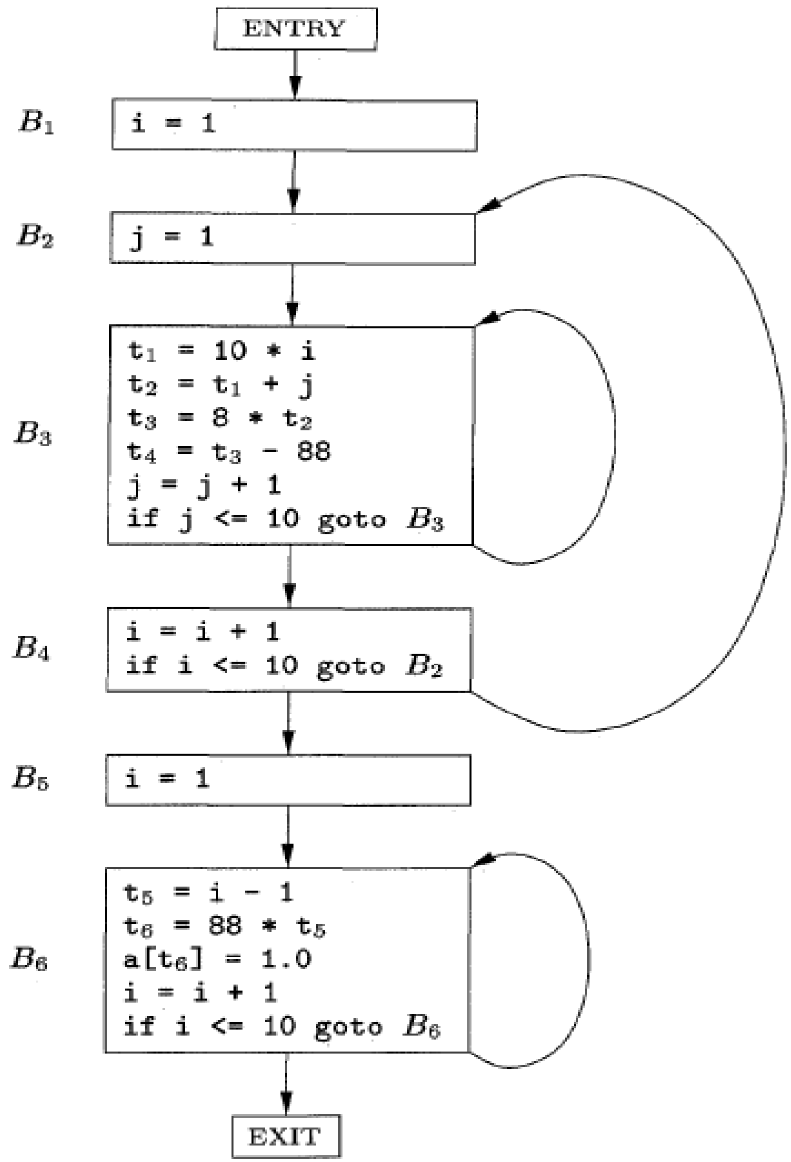
Rules for finding leaders

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

Consider an example

```
1)   i = 1
2)   j = 1
3)   t1 = 10 * i
4)   t2 = t1 + j
5)   t3 = 8 * t2
6)   t4 = t3 - 88
7)   a[t4] = 0.0
8)   j = j + 1
9)   if j <= 10 goto (3)
10)  i = i + 1
11)  if i <= 10 goto (2)
12)  i = 1
13)  t5 = i - 1
14)  t6 = 88 * t5
15)  a[t6] = 1.0
16)  i = i + 1
17)  if i <= 10 goto (13)
```

Flow graph based on Basic Blocks



Liveness and next-use information

- We wish to determine for each three address statement x=y+z what the next uses of x, y and z are.
- Algorithm:
  - Attach to statement i the information currently found in the symbol table regarding the next use and liveness of x, y, and z.
  - In the symbol table, set x to "not live" and "no next use.“
  - In the symbol table, set y and z to "live" and the next uses of y and z to i.