

A Simple Code Generator

Registers, Descriptors and the Code-Generation Algorithm

K.R. Chowdhary, Professor of CS

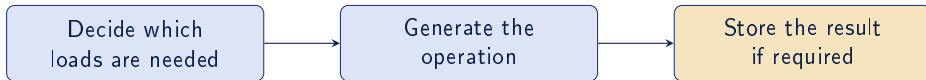
7CS5A Compiler Construction (Code Generation) | Lecture 16

16.1 A Simple Code Generator

Goal

Generate code for a **single basic block**. Take each three-address instruction in turn and track which values are in which registers, so that unnecessary loads and stores are avoided.

Pipeline for each instruction



The algorithm needs bookkeeping: what each register holds, and where each variable's current value lives. These are the *register* and *address descriptors* (slide 5).

Four Principal Uses of Registers

1. Operands

In most architectures, some or all operands of an operation must be in registers.

2. Temporaries

Hold the result of a subexpression while a larger expression is evaluated, or a variable used only within one basic block.

3. Global values

Values computed in one block and used in others, e.g. a loop index incremented around the loop and used several times within it.

4. Run-time storage management

Managing the run-time stack: stack pointers and possibly the top elements of the stack itself.

Competing needs

The number of registers is limited, so these uses compete.

Assumptions and Machine Instructions

Assumptions

- ▶ A set of registers is available for values used *within* the block; it excludes registers reserved for global variables and the stack
- ▶ The block is already a preferred sequence of three-address instructions (e.g. common subexpressions combined)
- ▶ Each operator has exactly one machine instruction taking operands in registers and leaving the result in a register
- ▶ After freeing all registers by storing their values, enough registers remain for any three-address operation

Instruction forms

- ▶ LD *reg, mem*
- ▶ ST *mem, reg*
- ▶ OP *reg, reg, reg*

16.1.1 Register and Address Descriptors

Register descriptor

For each *available register*: the variable names whose current value is in that register.

- ▶ Initially all are empty (only block-local registers are used)
- ▶ As generation proceeds, a register holds the value of zero or more names

Address descriptor

For each *program variable*: the location(s) where its current value can be found.

- ▶ A register, a memory address, a stack location, or several of these
- ▶ Can be stored in the symbol-table entry

Why both?

We must know if a variable's value is in a register, and whether its memory location is still correct: a new value may have been computed in a register and not yet stored.

16.1.2 The Code-Generation Algorithm

getReg(I)

Selects a register for each memory location associated with three-address instruction I .

- ▶ Has access to the register and address descriptors for the block
- ▶ May also use data-flow information, e.g. variables live on exit

Commutativity

For $x = y + z$, treat $+$ as a generic operator with machine instruction ADD. Then y must go in the *second* register of the ADD, never the third.

Improvement: for commutative $+$, generate code for both $x = y + z$ and $x = z + y$ and keep the better sequence.

The details of `getReg` are discussed after the basic algorithm on the next slide.

Machine Instructions for Operations

For $x = y + z$

1. Call `getReg( $x = y + z$ )` to pick registers R_x , R_y , R_z
2. If y is not in R_y (per the register descriptor), issue `LD  $R_y, y'$` , where y' is a memory location of y from its address descriptor
3. Likewise, if z is not in R_z , issue `LD  $R_z, z'$`
4. Issue the operation `ADD  $R_x, R_y, R_z$`

Resulting code when neither operand is in a register

```
LD  Ry, y'  
LD  Rz, z'  
ADD Rx, Ry, Rz
```

Next: 16.2 Peephole Optimization

Improving the generated code with local transformations. (This section appears only as a heading in these notes.)