COMS W4115
Programming Languages and Translators
Lecture 20: Code Generation Algorithms
April 10, 2013

1. Target Machine

- *n* general-purpose registers
- Instructions: load, store, compute, jump, conditional jump
- Various addressing modes:
  - indexed address
  - integer indexed by a register
  - indirect addressing
  - immediate constant
- Example 1: for `x = y + z` we can generate

```
LD  R1, y       // R1 = y
LD  R2, z       // R2 = z
ADD R1, R1, R2  // R1 = R1 + R2
ST  x, R1       // x  = R1
```

- Example 2: for `b = a[i]` where a is an array of integers we can generate

```
LD  R1, i       // R1 = i
MUL R1, R1, #4  // R1 = R1 * 4
LD  R2, a(R1)   // R2 = contents(a + contents(R1))
ST  b, R2       // b  = R2
```

- Example 3: for `a[j] = c` where a is an array of integers we can generate

```
LD  R1, c       // R1 = c
LD  R2, j       // R2 = j
MUL R2, R2, #4  // R2 = R2 * 4
ST  a(R2), R1   // contents(a + contents(R2)) = R1
```

- Example 4: for `x = *p` we can generate

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

- Example 5: for `*p = y` we can generate

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

- Example 6: for `if x < y goto L` we can generate

```
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
```

Here `M` is the label of the first machine instruction generated from the three-address instruction that has label `L`

- Instruction cost: 1 + cost associated with the addressing modes of the operands

2. Names to Addresses

- Addresses in the target code are in the runtime address space.
- Names in the IR need to be converted into addresses in the target code.
- Example: managing the addresses in the runtime stack
  - The code for the first procedure initializes the runtime stack by setting the stack pointer SP to the start of the stack.
  - A procedure call increments SP, saves the return address, and transfers control to the called procedure.
  - In the return sequence
    - the called procedure transfers control to the return address using the instruction `BR *0(SP)`
    - the caller decrements SP to its previous value

3. Computing Next-Use Information

- Knowing when the value of a variable will be used next is essential for generating good code.
- If there is a three-address instruction sequence of the form

```
i:  x = y + z

.
. no assignments to x between instructions i and j
.

j:  a = x + b
```

then we say statement `j` *uses* the value of `x` computed at `i`.

- We also say that variable `x` is *live* at statement `i`.
- A simple way to find next uses is to scan backward from the end of a basic block keeping track for each name `x` whether `x` has a next use in the block and if not whether `x` is live on exit from that block. See Alg. 8.7, p. 528.

4. A Simple Code Generator

- Here we describe an algorithm for generating code for a basic block that keeps track of what values are in registers so it can avoid generating unnecessary loads and stores.
- It uses a register descriptor to keep track of what variable values are in each available register.
- It uses an address descriptor to keep track of the location or locations where the current value of each variable can be found.
- For the instruction x = y + z it generates code as follows:
  - It calls a function getReg(x = y + z) to select registers Rx, Ry, and Rz for variables x, y, and z.
  - If y is not in Ry, it issues the load instruction `LD Ry, My` where My is one of the memory locations for y in the address descriptor.
  - Similarly, if z is not in Rz, it issues a load instruction `LD Rz, Mz`.
  - It then issues the instruction `ADD Rx, Ry, Rz`.
- For the instruction x = y it generates code as follows:
  - It calls a function getReg(x = y) to select a register Ry for both x and y. We assume retReg will always choose the same register for both x and y.
  - If y is not in Ry, issue the load instruction `LD Ry, My` where My is one of the memory locations for y in the address descriptor.
  - If y is already in Ry, we issue no instruction.
- At the end of the basic block, it issues a store instruction `ST x, R` for every variable x that is live on exit from the block and whose current value resides only in a register R.
- The register and address descriptors are updated appropriately as each machine instruction is issued.
- If there are no empty registers and a register is needed, the function getReg generates a store instruction `ST v, R` to store the value of the variable v in some occupied register R. Such a store is called a *spill*. There are a number of heuristics to choose the register to spill.

5. Optimal Code Generation for Expression Trees

- In this section we assume we are using a *k*-register machine with instructions of the form
  - `LD reg, mem`
  - `ST mem, reg`
  - `OP reg, reg, reg`

  to evaluate expressions.
- Ershov numbers
  - An expression tree is a syntax tree for an expression.
  - Numbers, called Ershov numbers, can be assigned to label the nodes of an expression tree. The Ershov number at a node gives the minimum number of registers needed to evaluate on a register machine the expression generated by that node with no spills.

- Algorithm to label the nodes of an expression tree with Ershov numbers
  1. Label all leaves 1.
  2. The label of an interior node with one child is the label of its child.
  3. The label of an interior node with two children is the larger of the labels of its children if these labels are different; otherwise, it is one plus the label of the left child.
- Sethi-Ullman algorithm generates register machine code that minimizes the number of spills to evaluate an expression tree. Ershov numbers guide the evaluation order.
  - Input: an expression tree labeled with Ershov and a *k*-register machine.
  - Output: an optimal sequence of register machine instructions to evaluate the root of the tree into a register.
  - The details of the algorithm are in Section 8.10 of ALSU, pp. 567-573.

6. Practice Problems

1. ALSU, Exercise 8.2.5 (p. 517).
2. ALSU, Exercise 8.10.2 (p. 573).

7. Reading

- ALSU, Sections 8.2-8.4, 8.6, 8.10

---

aho@cs.columbia.edu