Code Generation

Chapter 8

Code Generation

- Severe requirements
  - Target program preserve the semantics
  - Effective use of available resources
  - Itself must be efficient
- Primary tasks
  - Instruction selection
  - Register allocation and assignment
  - Instruction ordering

Input to the Code Generator

- IR produced by the front end
  - Three address code: quadruples, triples
  - Virtual machine: bytecodes, stack-machine code
  - Linear representations: postfix notation
  - Graphical representations: DAG, syntax tree
- Symbol table
  - Determine the run-time addresses of the names in IR

The Target Program

- RISC (reduced instruction set computer)
  - Many registers, three-address instructions, simple addressing modes, simple instruction set arch
- CISC (complex instruction set computer)
  - few registers, two-address instructions, various addressing modes, variable-length instruction set
- Stack-based Machine
  - Pushing operands onto a stack
  - Almost disappeared, then revived with Java Virtual Machine (JVM)

The Target Program

- Absolute machine-language programs
  - Can be placed in a fixed location in memory and immediately executed
- Relocatable machine-language programs
  - Subprograms can be complied separately, then linked together and loaded for execution by linking loader
- Assembly-language program
  - Code generation easier, need assembly step

Instruction Selection

- Map the IR into a code sequence that can be executed by the target machine

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
```

Register Allocation

- Register allocation: select the set of vars that will reside in registers
- Register assignment: pick the specific register that a var will reside in

```
t = a + b
t = t * c
t = t / d
```

```
L  R1, a
A  R1, b
M  R0, c
D  R0, d
ST  R1, t
```

Evaluation Order

- The order can affect the efficiency
  - Some orders require fewer registers
  - Picking best order is NP-complete

The Target Language

- Assume following kinds of instructions
  - Load: LD dst, addr (LD r, x)
  - Store: ST x, r
  - Computation: OP dst, src1, src2 (unary operators do not have src2)
  - Unconditional jumps: BR L
  - Conditional jumps: Bcond r, L (BLTZ r, L)

The Target Language

- Addressing mode
  - Variable name x
  - Indexed address a(r)
    - LD R1, a(R2) is R1 = contents(a+contents(R2))
  - An integer indexed by a register
    - LD R1, 100(R2) is R1 = contents(100+contents(R2))
  - Indirect addressing modes
    - LD R1, *100(R2) is R1=contents(contents(100+contents(R2))
  - Immediate constant address mode
    - LD R1, #100

The Target Language Example

x=y-z

```
LD  R1, y
LD  R2, z
SUB R1, R1, R2
ST  x, R1
```

b=a[i]

```
LD  R1, i
MUL  R1, R1, 8
LD R2, a(R1)
ST  b, R2
```

x=*p

```
LD  R1, p
LD R2, 0(R1)
ST  x, R2
```

if x < y goto L

```
LD  R1, x
LD  R2, y
SUB  R1, R1, R2
BLTZ  R1, L
```

Program and Instruction Costs

- Determining the actual cost of compiling and running a program is complex
  - Cost of an instruction is one plus the costs associated with the addressing modes
    - LD R0, R1: cost 1
    - LD R0, M: cost 2
    - LD R1, *100(R2): cost 3

TEST YOURSELF #1

- Generate code for the following three address sequences

```
x = a[i]
y = b[i]
Z = x * y
```

```
y = *q
q = q + 4
*p = y
p = p + 4
```

Basic Blocks and Flow Graphs

- Partition IR into basic blocks, which are maximal sequences of consecutive three-address instructions that
  - The flow of control can only enter the basic block through the first instruction
  - Control leave the block without halting/branching, except at the last instruction
- The basic blocks become the nodes of a flow graph, whose edges indicate which blocks can follow which other blocks

Create Basic Blocks

- An instruction is a leader if
  - The first instruction in the IR
  - The target of a jump
  - Immediately follows a jump
- For each leader, its basic block consists of itself and all the instructions upto (but not including) the next leader, or the end of the IR

TEST YOURSELF #2

- Construct basic blocks of the following IR

```
for i from 1 to 10 do
        for j from 1 to 10 do
                a[i,j]=0.0
for i from 1 to 10 do
        a[i,i]=1.0
```

```
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)  T 6 = 88 * t5
15)  A[t6] = 1.0
16)  I = i+1
17)  I i<= 10 goto (13)
```

Next-Use Information

- Knowing when the value of a var will be used next is essential for good code generation
- j uses the value of x computed at i if
  - statement i assigns to x
  - If j has x as an operand, and control can reach j from i without an intervening assignment to x

Next-Use Algorithm

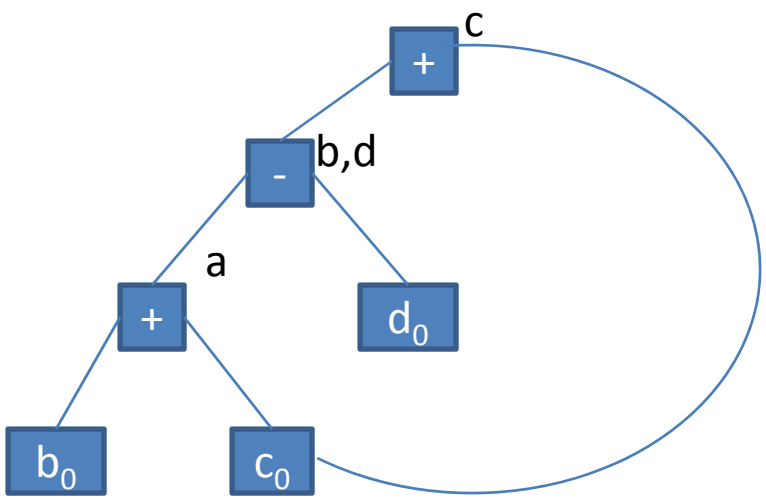
- Input: a basic block B
- Output: attach to i: x=y+z the next-use info
- Method: start at the last statement B
  - Attach to i the information currently found in symbol table
  - set x to “not live” and “no next use”
  - Set y/z to “live” and the next uses of y/z to I
  - Note the steps of 2 and 3 may not be interchanged

Optimization of Basic Blocks

- Substantial improvement can be gained by performing local optimization
- DAG Representation of basic blocks
  - A node for each initial values of the vars
  - Node for each statement
    - Children are the statements of the last definitions
    - Labeled by operators and vars for which it is the last definition in the block
  - Some nodes are "output nodes" whose vars are live on exit
    - Part of global flow analysis, discussed later

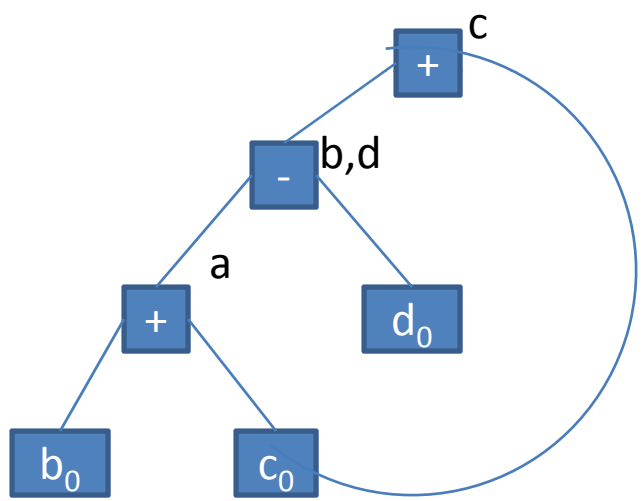
DAGs for Basic Blocks

a = b + c
b = a – d
c = b + c
d = a - d



Finding Local Common Subexpressions

a = b + c
b = a – d
c = b + c
d = a - d

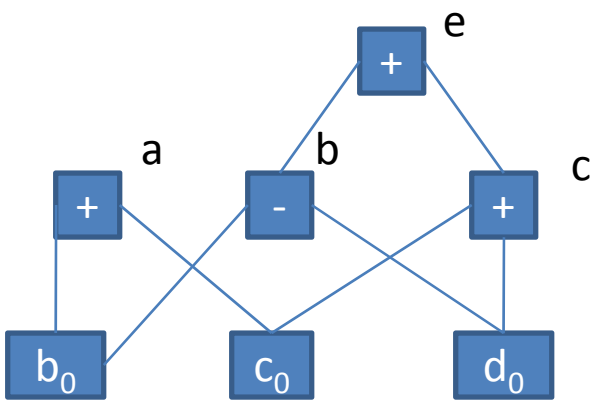


a = b + c
d = a – d
c = d + c

Dead Code Elimination

- Delete any root that has no live variables, and repeat the procedure

a = b + c
b = b – d
c = c + d
e = b + c



Use of Algebraic Laws

- Identities
  - x+0=0+x=x; x*1=1*x=x; x-0=x; x/1=x;
- Reduction in strenth
  - x^2=x*x; 2*x = x+x; x/2=x*0.5
- Commutativity
  - x*y=y*x
- Associativity
  - a = b+c; e=c+d+b;
  - a = b+c; t=c+d; e = t+b
  - a = b + c; e = a+d

A Simple Code Generator

- Generate code for a single basic block
- How to use registers?
  - In most machine architectures, some or all of the operands must be in registers
  - Registers make good temporaries
  - Hold values that are computed in one basic block and used in other blocks
  - Often used with run-time storage management

Register and Address Descriptors

- For each available register, a register descriptor (RD) keeps track of the vars whose current value in that register
  - Initially empty
- For each var, an address descriptor (AD) keeps track of the locations where the current value of the var can be found
  - Location can be a register, a memory address, etc.

Code-generation Algorithm

- For a three-address instruction, e.g. x=y+z
  - Use *getReg(x=y+z)* to select registers R_x, R_y, R_z for x, y , z
  - If y is not in R_y, issue an instruction LD R_y, y'
    - Similarly for x
  - Issue the instruction ADD R_x, R_y, R_z
- Copy statement x=y
  - If y is not already in register, generate LD R_y, y'
  - Adjust RD for R_y so it includes x
  - Change AD for x so its only location is R_y
- Ending the basic block
  - If x is used at other blocks, issue ST x, R_x

Managing Register and Address Descriptors

- For LD R, x
  - Change RD for R so it holds only x
  - Change AD for x by adding R as an additional location
- For ST x, R
  - Change AD for x to include its own memory location
- For ADD R_x, R_y, R_z
  - Change RD for R_x so it holds only x
  - Change AD for x so its only location is R_x
  - Remove R_x from the AD of any var other than x

Example

```
t = a – b
u = a – c
v = t + u
a = d
d = v + u
```

- t, u, v are temp vars, while a, b, c, d are global
- Assume registers are enough
  - Reuse registers whenever possible

Example

R1	R2	R3

a	b	c	d	t	u	v
a	b	c	d			

t = a - b

LD R1, a; LD R2, b; SUB R2, R1, R2

R1	R2	R3
a	t	

a	b	c	d	t	u	v
a,R1	b	c	d	R2		

u = a - c

LD R3, c; SUB R1, R1, R3

R1	R2	R3
u	t	c

a	b	c	d	t	u	v
a	b	c,R3	d	R2	R1	

v = t + u

ADD R3, R2, R1

R1	R2	R3
u	t	v

a	b	c	d	t	u	v
a	b	c	d	R2	R1	R3

a = d

LD R2, d

R1	R2	R3
u	a,d	v

a	b	c	d	t	u	v
R2	b	c	d,R2		R1	R3

d = v + u

ADD R1, R3, R1

R1	R2	R3
d	a	v

a	b	c	d	t	u	v
R2	b	c	R1			R3

ST a, R2; ST d, R1

R1	R2	R3
d	a	v

a	b	c	d	t	u	v
a,R2	b	c	d,R1			R3

Function getReg

- Consider picking R_y for y in x = y + z
  - If y in a register, do nothing
  - If y not in a register and there is a empty one, choose it as R_y
  - Let v be one of the var in R
    - We are OK if v is somewhere besides R
    - We are OK if v is x
    - We are OK if v is not used later
    - Spill: ST v, R

Peephole Optimization

- Exam a sliding window and replace instruction sequence with a shorter or faster sequence
  - Redundant-instruction elimination
  - Flow-of-control optimization
  - Algebraic simplifications
  - Use a machine idioms

Eliminating Redundant Loads and Stores

```
LD a, R0
ST R0, a
```

Eliminating Unreachable Code

```
if debug==1 goto L1
goto L2
L1: print debugging info
L2:
```

```
if debug!=1 goto L2
L1: print debugging info
L2:
```

```
L2:
```

Flow-of-Control Optimizations

```
goto L1
...
L1: goto L2
```

```
goto L2
...
L1: goto L2
```

```
if a < b goto L1
...
L1: goto L2
```

```
if a < b goto L2
...
L1: goto L2
```

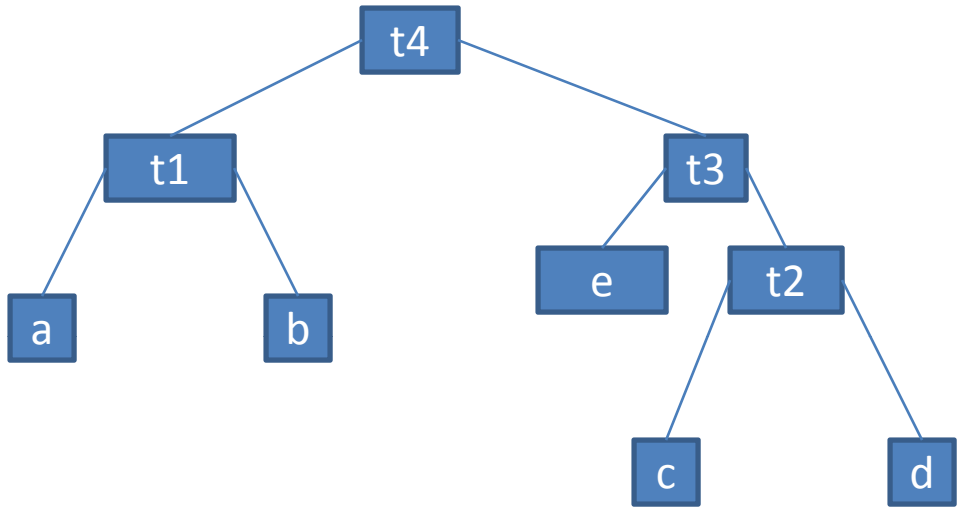
Optimal Code Generation for Expressions

- We can choose registers optimally
  - If a basic block consists of a single expression, or
  - It is sufficient to generate code for a block one expression at a time

Ershov Numbers

- Assign the nodes of an expression tree a number that tells how many registers needed
  - Label leaf 1
  - The label of an interior node with one child is the label of its child
  - The label of an interior node with two children
    - the larger one if the labels are different
    - One plus the label if the labels are the same

Ershov Numbers



(a-b)+e*(c+d)

t1 = a – b
t2 = c + d
t3 = e * t2
t4 = t1 + t3

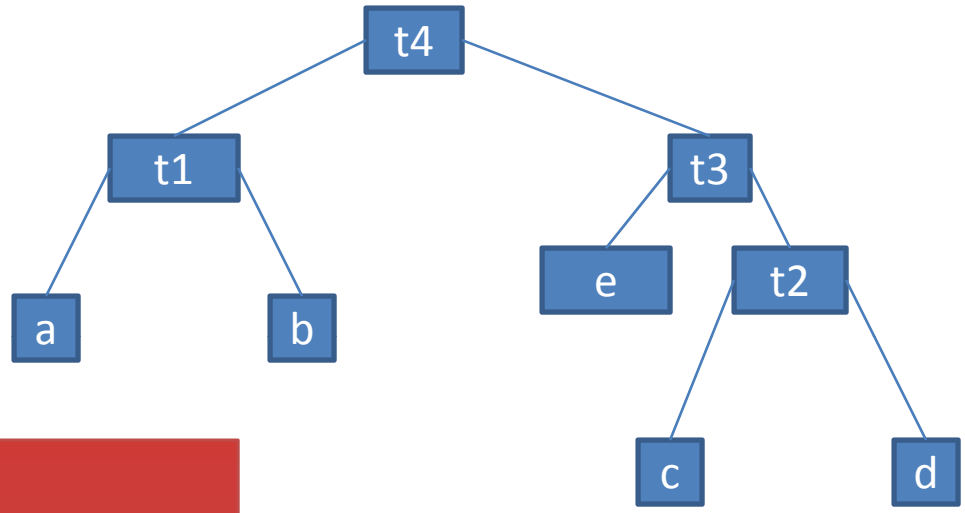
Generating Code From Labeled Expression Tree

- Recursive algorithm starting at the root
  - Label k means k registers will be used
  - R_b, R_{b+1}, …, R_{b+k-1} , where b>=1 is a base
- To generate machine code for a node with label k and two children with equal labels
  - Recursively generate code for the right child, using base b+1: R_{b+1}, R_{b+2}, …, Result in R_{b+k-1}
  - Recursively generate code for the right child, using base b: R_b, R_{b+1}, …, Result in R_{b+k-2}
  - Generate instruction OP R_{b+k-1} , R_{b+k-2} ,R_{b+k-1}

Generating Code From Labeled Expression Tree

- To generate machine code for a node with label k and two children with unequal labels
  - Recursively generate code for the child with label k, using base b: R_b, R_{b+1}, …, Result in R_{b+k-1}
  - Recursively generate code for the child with label m, using base b: R_b, R_{b+1}, …, Result in R_{b+m-1}
  - Generate instruction OP R_{b+k-1} , R_{b+m-1} ,R_{b+k-1}
- For a leaf x, if base is b generate LD R_b, x

Ershov Numbers



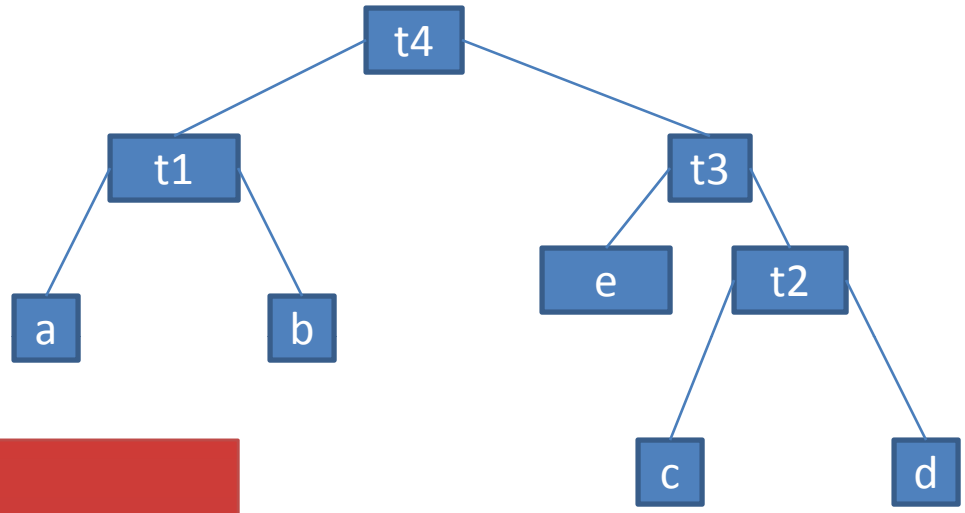

```
LD R3, d
LD R2, c
ADD R3, R2, R3
LD R2, e
MUL R3, R2, R3
```

```
LD R2, b
LD R1, a
SUB R2, R1, R2
ADD R3, R2, R3
```

Insufficient Supply of Registers

- Input: a labeled tree and a number r of registers
- For a node N with at least one child labeled r or greater
  - recursively generate code for the big child with b=1. The result will appear in R_r
  - Generate machine instruction ST t_k, R_r
  - If the little child has label r or greater, b=1. If the label is j<r, then b=r-j. The result in R_r
  - Generate the instruction LD R_{r-1}, t_k
  - Generate OP R_r, R_r, R_{r-1} or OP R_r, R_{r-1}, R_r

Insufficient Supply of Registers




```
LD R2, d
LD R1, c
ADD R2, R1, R2
LD R1, e
MUL R2, R1, R2
ST t3, R2
```

```
LD R2, b
LD R1, a
SUB R2, R1, R2
LD R1, t3
ADD R2, R2, R1
```