CS 536 Announcements for Thursday, April 28, 2022

Last Time
- continue code generation
- function declaration, call, and return
- expressions
- literals
- assignment
- I/O

Today
- wrap up code generation
- dot-access
- control-flow constructs
  - numeric approach
  - control-flow approach

Next Time
- optimization

P6: Codegen class

Constants for registers and boolean constant
e.g., FP, SP, T0, T1

Methods to help automatically generate code

generate(opcode, ... args ... )
e.g., generate("add", "$t0", "$t0", "$t1")
writes out add $t0, $t0, $t1
versions for fewer args as well
generateIndexed(opcode, arg1, arg2, offset)
e.g., generateIndexed("lw", "$t0", "$t1", -12)
writes out lw $t0, -12($t1)
genPush(reg) / genPop(reg)

nextLabel() – returns a unique string to use as a label
genLabel(L) – places a label

Code Generation for Dot-access

Offset from base of struct to certain field is known **statically**
- compiler can do the math for the slot address
- not true for languages with pointers!

Example

```c
struct Inner {
    bool hi;
    int there;
    int c;
};

struct Demo {
    struct Inner b;
    int val;
};

void f() {
    struct Demo inst;
    ...
    = inst.b.c;
    inst.b.c = ...
}
```

**RHS** - put value on stack
    `lw $t0, -16($fp)  # $t0 = value stored at $fp-16`
    `push $t0`

**LHS** - put address on stack
    `subu $t0, $fp, 16  # $t0 = $fp-16`
    `push $t0`

Kinds of control flow

In minim

if exp {
    ...
} else {
    ...
}

What is needed at the assembly-code level

- branching
  - unconditional
  - conditional
- labels

MIPS

b label
beq rl, src, label
Also: bne, bgt, bge, blt, ble

use branch in if/while control structure (rather than jump)
register or immediate value
Code generation for control flow

minim code example:

```c
if (a > b) || c {
    /* body of if */
} else {
    /* body of else */
}
```

Two approaches:
- numeric approach
- control-flow approach

Numeric approach:
- leave codeGen as is for condition of if (i.e., put value onto stack)
- branch if value from top of stack has a particular value

Generated code (outline) for if-then-else

```
codeGen() on boolean expression
pop $t0
beq $t0, FALSE, falseLabel
```

```
codeGen() on stmts in true branch
b endIfLabel
```

```
falseLabel:
```

```
codeGen() on stmts in false branch
```

```
endIfLabel:
```
Code generation for control flow (cont.)

Control-flow approach:
- push the branching into the code generation of the expression in the if condition
  - avoid push and pop of value
  - make use of MIPS other branching options

Generated code (outline) for if-then-else

generate code to evaluate expression and then branch:
  go to trueLabel if true, falseLabel if false

trueLabel:
  codeGen() on stmts in true branch
  b endIfLabel

falseLabel:
  codeGen() on stmts in false branch

endIfLabel:

To make this work, we need methods in ExpNodes that generate the branching
- create genJumpCode methods
  - inputs: true label, false label
  - output: generates code to jump (branch) to
    true label if the expression evaluates to true,
    false label if the expression evaluates to false

- which ExpNodes need genJumpCode methods?

And, Or, Not
Greater, Less, GreaterEq, LessEq
Equals, NotEquals
True, False
Id
Fn Call Exp
Assign Exp
Dot Access (not in P6, though)

any nodes that can evaluate to boolean value (they could show up as
exp node in condition of while or if)
**genJumpCode methods**

Recall: `genJumpCode(Tlabel, Flabel)`

**TrueNode**

\[
\text{b Tlabel}
\]

**IdNode**

\[
\begin{align*}
\text{lw} & \text{ $t0$, address of var} \\
\text{beg} & \text{ $t0$, FALSE, Flabel} \\
\text{b} & \text{ Tlabel}
\end{align*}
\]

**GreaterNode**

\[
\text{leftExp. code} \text{ (Gen(} \text{)} \\
\text{rightExp. code} \text{ (Gen(} \text{)} \\
\text{genPop(T1) } \# \text{ pop R operand} \\
\text{genPop(T0) } \# \text{ pop L operand} \\
\text{bge T0, T1, Tlabel } \# \text{ branch to Tlabel if T0} \geq \text{T1} \\
\text{b Flabel}
\]

**OrNode**

\[
\begin{align*}
\text{leftExp. code} \text{ (Gen(} \text{)} \\
\text{genPop(T0)} \\
\text{beg T0, TRUE, Tlabel} \\
\text{rightExp. code} \text{ (Gen(} \text{)} \\
\text{genPop(T0)} \\
\text{beg T0, TRUE, Tlabel} \\
\text{b Flabel}
\end{align*}
\]

Looks like what we wanted to avoid

Better implementation

\[
\begin{align*}
\text{leftExp. genJumpCode(Tlabel, NewLabel)} \\
\text{genLabel(NewLabel)} \\
\text{rightExp. genJumpCode(Tlabel, Flabel)}
\end{align*}
\]
Example

Numeric Approach

```
lw $t0, addr_a
push $t0

lw $t0, addr_b
push $t0

pop $t1
pop $t0

sgt $t0, $t0, $t1
push $t0

pop $t0
beq $t0, FALSE, continueLabel
li $t1, TRUE
push $t1
b doneOrLabel
```

```
continueLabel:  
lw $t0, addr_c
push $t0

doneOrLabel:    
pop $t0
beq $t0, FALSE, falseLabel

falseLabel:

doneIfLabel:
```

Control-flow Approach

```
lw $t0, addr_a
push $t0

lw $t0, addr_b
push $t0

pop $t1
pop $t0

bgt $t0, $t1, trueLabel
b newLabel
```

```
newLabel:  
lw $t0, addr_c
beq $t0, FALSE, falseLabel
b trueLabel
```

```
trueLabel:  
```

```
falseLabel:

doneIfLabel:
```

IfElif calls

OrNode.genJumpCode(trueLabel, falseLabel)

OrNode
Greater.genJumpCode(trueLabel, newLabel)

IdNode.genJumpCode(trueLabel, falseLabel)
Comparing the two approaches

**Numeric approach**
- + no extra methods to write
- – more instructions generated/executed

**Control-flow (jump-code) approach**
- + fewer instructions generated/executed $\rightarrow$ smaller, faster code
- – extra methods to write (and debug)

You may implement either approach in Programming Assignment 6

**MIPS tips**

It’s really easy to get confused with assembly

Some suggestions
- start simple: main procedure with “print(1);”
  - get procedure `main` to compile and run
  - function prologue and epilog
  - trivial case of expressions: evaluating the constant 1, which pushes a 1 on the stack
  - printing: `print(1);`
- then grow your compiler incrementally
  - expressions
  - control constructs
  - call/return

Create super simple test cases
- main procedure: print the value of some expression
- create more and more complicated expressions

Regression suite
- rerun **all** test cases to check whether you introduced a bug
- more suggestions
  - try writing desired assembly code by hand before having the compiler generate it
  - draw pictures of program flow
  - have your compiler put in detailed comments in the assembly code it emits