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

\[ \text{generate}(\text{opcode}, \ldots \ \text{args} \ldots) \]
e.g., \[ \text{generate}("add", "$t0", $t0", "$t1") \]
writes out add $t0, $t0, $t1
versions for fewer args as well

\[ \text{generateIndexed}(\text{opcode}, \text{arg1}, \text{arg2}, \text{offset}) \]
e.g., \[ \text{generateIndexed}("lw", "$t0", $t1", -12) \]
writes out lw $t0, -12($t1)

\[ \text{genPush}(\text{reg}) / \text{genPop}(\text{reg}) \]
\[ \text{nextLabel}() \] – returns a unique string to use as a label
\[ \text{genLabel}(L) \] – 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 = ...;
```
Kinds of control flow

In minim

  if exp {
    ...
  }

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

  while exp {
    ...
  }

What is needed at the assembly-code level

- branching
  - unconditional: b label
  - conditional: beq r1, src, label

- labels
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

```c
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?
genJumpCode methods

Recall: genJumpCode(Tlabel, Flabel)

TrueNode

IdNode

GreaterNode

OrNode
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
```

**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:

...

b doneIfLabel
```

```
doneOrLabel:

pop $t0
beq $t0, FALSE, falseLabel
.
.
.
b doneIfLabel
doneIfLabel:

falseLabel:
.
.
.
```

```
doneIfLabel:
```
Comparing the two approaches

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

Control-flow (jump-code) approach
+ fewer instructions generated/executed → 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