Warm-Up Problem

- How would we encode switch statements in our translation of WLP4?

CS 241 Lecture 18

Code Generation Continued Again!

With thanks to Brad Lushman, Troy Vasiga, Kevin Lanctot, and Carmen Bruni

Let’s Relocate!

- Let’s load some code on the board

Back to Codegen: What's Left

- Most of our statements have been completed except for if and while statements (and delete but that comes much later…)
- For conditionals, we need to handle boolean tests.
- Convention: Store 1 inside \$11. (Now we have true and false stored somewhere)
- Aside: It may be useful to store print somewhere (say, \$10)

Code Structure Thus Far

```
; Prologue
.import print
lis $4
.word 4
lis $10
.word print
lis $11
.word 1
sub $29, $30, $4
; end  Prologue and begin body
; space for variables
; translated WLP4 code
; end body and begin epilogue
add $30, $29, $4
jr $31
```

Boolean Tests

- What would the code for the rule test $\rightarrow$ $expr_A$ < $expr_B$?
- code(test $\rightarrow$ $expr_A$ < $expr_B$) = code($expr_A$)
  \+ push($3)
  \+ code($expr_B$)
  \+ pop($5)
  \+ `slt $3, $5, $3`
- What should we do for test $\rightarrow$ exprA > exprB?

Boolean Tests

- Just change slt $3, $5, $3 to slt $3, $3, $5!

More Boolean Tests

Try to translate the rule $test \rightarrow exprA\ != exprB$.

More Boolean Tests

Try to translate the rule `test` → `exprA != exprB`.

```
code(test)= code(exprA)
          + push($3)
          + code(exprB)
          + pop($5)
          ; maybe store $6 and $7 if used
          + slt $6, $3, $5
          + slt $7, $5, $3
          ; Note 0 <= $6 + $7 <= 1
          + add $3, $6, $7
```

What should we do for `test` → `exprA == exprB`?

More Boolean Tests

- What should we do for
  test $\longrightarrow$ $expr_A$ == $expr_B$?
- Key idea is a == b is the same as !(a! = b).
- But we also don't have ! (not)...
- Sure we do! Add the line sub \$3, \$11, \$3 to flip 0 to 1 and vice versa!

Last Two Tests

- How do we do test $\rightarrow$ $expr_A$ <= $expr_B$ or test $\rightarrow$ $expr_A$ >= $expr_B$?
- Use the fact that a <= b is the same as !(a > b) and similarly for >=.
- This leaves us with our final if and while statements.

if Statements

Rule: statement → IF (test) {stmts1} ELSE {stmts2} .

Translation (Hint: Use Labels!):

```
code(statement) = code(test)
                + beq $3, $0, else
                + code(stmts1)
                + beq $0, $0, endif
                + else: code(stmts2)
                + endif:
```

Caution: Be wary of multiple labels! How do we fix this?

Multiple if Statements

- With the above, if we have multiple if statements (anywhere in the program!), their label names will conflict
- We need a way of inventing *totally unique label names*
- This sounds like a hard problem…

Simple if Counter Idea

- … but it’s not.
- Keep track of how many if statements you have. Have a counter of these; say ifcounter.
- Each time you have an if statement, increment the counter.
- Use label names like else# and endif# where# corresponds to the ifcounter.
- Note: This isn’t some silly solution for WLP4. gcc uses the imaginative label names “L1”, “L2”, “L3”, etc. for this.

while Statements

Rule: statement → WHILE (test) {statements}.

Translation (Hint: Use Labels!):

```
code(statement) = loop: code(test)
                + beq $3, $0, endWhile
                + code(stmts)
                + beq $0, $0, loop
                + endWhile:
```

Again, be sure to have a while loop counter variable like with i f statements!

(You can just use the same counter, too)

An Extremely Important Final Tip

- Since you are generating MIPS code; note that you can also generate **comments** with your MIPS code!
- We recommend that you also output some comments which will make it easier to decipher what you were doing when you see your final MIPS code.
- Debugging code generators is *hard*. gdb won't help you if the problem is in your output code, not your own code.

Recap

Before we continue, a moment to recap our conventions:

- \$0 is always 0.
- \$1 and \$2 are for arguments 1 and 2 in wain.
- \$3 is always for output (and possibly intermediate computations).
- \$4 is always 4.
- \$5 is also for intermediate computations.
- \$6 and \$7 may be for intermediate computations, depending on how you implement some cases.
- \$10 will store `print` (if you want)
- \$11 is reserved for 1.
- \$29 is our frame pointer (fp).
- \$30 is our stack pointer (sp).
- \$31 is our return address (ra).

Prologue

At the beginning of our code, we must:

- Load 4 into $4 and 1 into $11.
- Import print. Store in $10
- Store the return address on the stack.
- Initialize the frame pointer hence creating a stack frame
- Store registers 1 and 2

Body

- Need to store local variables in the stack frame
- Contain MIPS code corresponding to the WLP4 program

Epilogue

- Need to pop the stack frame
- Also, need to restore the previous variables.

Pointers

At last, we have reached pointers. We need to support all the following:

- NULL
- Allocating and deallocating heap memory
- Dereferencing
- Address-of
- Pointer arithmetic
- Pointer comparisons
- Pointer assignments and pointer access

Here we go!

NULL

- The first and obvious choice for us for the value of NULL is 0x0. Why is this a problem for us?

NULL

- On our system, 0x0 is a valid memory address! C was designed assuming that the operating system would protect 0, but we don't have that (you'll learn about how this is done in CS350)
- We would like our NULL to crash if attempting to dereference. Crashing is good!

NULL

- We would like our NULL to crash if attempting to dereference. Crashing is good!
- We can force this by picking a value for NULL that is *not word-aligned*.
- Word-aligned for us means that the address is a multiple of 4. A valid value for NULL for us, then, is 0x1. That's a good enough value, so we'll use it.

Code

- code(factor → NULL) =
  `add $3, $0, $11`
- Note that attempts to use NULL with either lw or sw will result in a crash, since MIPS is expecting a word-aligned address.

Dereferencing

- What about dereferencing? Consider $factor_1 \rightarrow$ STAR $factor_2$
- The value in $factor_2$ is a pointer (otherwise you should have a type error!). What we want is to access the value at $factor_2$ and load it somewhere.
- As always, we load into \$3. Since $factor_2$ is a memory address, we want to load the value in the memory address at \$3 and store in \$3.
- code($factor_1 \rightarrow$ STAR $factor_2$) = code($factor_2$) + lw \$3, 0(\$3)

Mind Your Types!

- Be wary of the difference between lvalues and pointers
- While it may be useful to implement code(lvalue) to generate a pointer, if the type of the lvalue is a pointer, it'll be a pointer to a pointer!