CS 536 Announcements for Wednesday, April 17, 2024

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

Today
- wrap up code generation
  - tuple access
  - control-flow constructs
- introduce control flow graphs

Next Time
- optimization

P6 : Codegen class

Constants for registers and logical constants
  - e.g., \( FP, SP, T0, T1 \)

Methods to help automatically generate code
  - \( \text{generate}(\text{opcode, ... args ... }) \)
    - e.g., \( \text{generate}("add", "$t0", "$t0", "$t1") \)
    - writes out \( \text{add }$t0, $t0, $t1\)
    - versions for fewer args as well

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

  - \( \text{genPush}(\text{reg}) / \text{genPop}(\text{reg}) \)

  - \( \text{nextLabel()} \) – returns a unique string to use as a label

  - \( \text{genLabel}(\text{L}) \) – places a label

\( \text{L} \) is of the form \( .LX \) of the form \( .LX \) of the form \( .LX \) of the form \( .LX \) of the form \( .LX \)
Code Generation for Tuple Access

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

**Example**

```plaintext
tuple Inner {
    logical hi.
    integer there.
    integer c.
}

tuple Demo {
    tuple Inner b.
    integer val.
}

void f() {
    tuple Demo inst.

    ... = inst:b:c.
    inst:b:c = ... .
```

RHS - put *value* on stack

1w $t0, -16($fp) # $t0 = value stored at $fp - 16

LHS - put *address* on stack

subu $t0, $fp, 16 # $t0 = $fp - 16

push $t0
Control flow graphs

Kinds of control flow
- function calls (e.g., `jal`, `jr`)
- selection (e.g., `if`, `if-else`, `if-elseif`, `switch`)
- repetition (e.g., `while`, `do-while`, `repeat until`, `for`)
- short-circuited operators (`&&`, `||`)

Control flow graph (CFG)
- important representation for program optimization
- helpful way to visualize source code

Example

```assembly
Line1: li $t0, 4
Line2: li $t1, 3
Line3: add $t0, $t0, $t1
Line4: sw $t0, val
Line5: b Line2
Line6: sw $t0, 0($sp)
Line7: subu $sp, $sp, 4
```

![Control flow graph example](image_url)
Kinds of control flow in base

if exp [ ...

... ]

if exp [ ...

... ] else [ ...

... ]

while exp [ ...

... ]

What is needed at the assembly-code level

- branching
  - unconditional
  - conditional

\[ \text{MIPS} \]

\[
\begin{align*}
\text{b} & \quad \text{label} \\
\text{beq} \ r1, src, \ text{label} \\
\text{Also:} & \quad \text{bne, bgt, bge, blt, ble}
\end{align*}
\]

use branch in if/while control structures (rather than jump)

Register or immediate value
Code generation for if statements

base code example:

```python
if a == b :
    $ body of if
```

Need to linearize - output a sequence of instructions

Code generation steps:

• get a label for end of construct
• generate code for expression
• generate conditional branch
• generate body of if
• place end-of-construct label

Code generation for if-else statements

base code example:

```python
if a > b :
    $ body of if
else :
    $ body of else
```

Need these labels to be unique, i.e., generated by Codegen.nextLabel()
Code generation for if-else statements (cont.)

base code:

```java
if a > b [
    $ body of if
]
else [
    $ body of else
]
```

MIPS code outline:

```
lw $t0, addr_a
push $t0
lw $t0, addr_b
push $t0
pop $t1
pop $t0
sgt $t0, $t0, $t1
push $t0
beq $t0, FALSE, falseLabel
```

Use Codegen.java
- has push/pop methods
to generate MIPS code

```
sgt R2, R0, R1
```
sets R2 to 1 if R0 > R1
to 0 otherwise

Also have: sge, slt, sles, seq, sne

Note: only ended up using
```
beq & b branching instrs
```

CodeGen
- on It(a)
Code generation for if-else statements (cont.)

Revisiting the CFG

```assembly
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, falseLabel
```

1. # code for true branch
   2. b doneIfLabel

falseLabel:

3. # code for false branch

```assembly
doneIfLabel:
```

Code generation for while statements

Base code example:

```python
while a == b [
    $ body of while
]
```
MIPS tips

It’s really easy to get confused with assembly

Some suggestions

- **start simple:** main procedure that prints the value 1
  - get procedure `main` to compile and run
    - function prologue and epilogue
  - trivial case of expressions: evaluating the constant 1, which pushes a 1 on the stack
    - printing: `write << 1`.
  - then grow your compiler incrementally
    1. expressions
    2. control constructs
    3. 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