Code Generation, Continued
How to be a MIPS Master

• It’s really easy to get confused with assembly
  – Try writing a program by hand before having the compiler generate it
  – Draw lots of pictures of program flow
  – Have your compiler output detailed comments

• Get help
  – Post on piazza
Roadmap

• Last time:
  – Talked about compiler backend design points
  – Decided to go with direct to machine code design for our language

• This time:
  – Run through what the actual codegen pass will look like
Review: Global Variables

• Showed you one way to do declaration last time:
  
  .data
  .align 2
  _name: .space 4

• Simpler form for primitives:
  
  .data
  _name: .word <value>
Review: Functions

– Preamble
  • Sort of like the function signature

– Prologue
  • Set up the function

– Body
  • Do the thing

– Epilogue
  • Tear down the function
Function Preambles

```
int f(int a, int b){
    int c = a + b;
    int d = c - 7;
    return c;
}
```

This label gives us something to jump to

```
jal f
```

.text
f:
#... Function body ...

Function Prologue

• Recall our view of the Activation Record
  1. save the return address
  2. save the frame pointer
  3. make space for locals
  4. update the frame ptr

- low mem

- high mem

- static size

- space for local 2
- space for local 1
- ctrl link (caller $fp)
- ret address (caller $ra)
- param 1
- param 2
- caller’s AR
Function Prologue: MIPS

• Recall our view of the Activation Record
  1. save the return address
  2. save the frame pointer
  3. make space for locals
  4. update the frame ptr

```asm
.text
f:
  sw $ra 0($sp)       #call lnk
  subu $sp $sp 4      # (push)
  sw $fp 0($sp)       #ctrl lnk
  subu $sp $sp 4      # (push)
  subu $sp $sp 8      #locals
  addu $fp $sp 16     #update fp
```
Function Epilogue

• Restore Caller AR
  1. restore return address
  2. restore frame pointer
  3. restore stack pointer
  4. return control

$ra: (old $ra)
Function Epilogue: MIPS

• Restore Caller AR
  1. restore return address
  2. restore frame pointer
  3. restore stack pointer
  4. return control

.text
f:
  sw $ra 0($sp)
  subu $sp $sp 4
  sw $fp 0($sp)
  subu $sp $sp 4
  subu $sp $sp 8
  addu $fp $sp 16
  #... Function body ...
  lw $ra, 0($fp)
  move $t0, $fp
  lw $fp, -4($fp)
  move $sp, $t0
  jr $ra
Function Body

• Obviously, quite different based on content
  – Higher-level data constructs
    • Loading parameters, setting return
    • Evaluating expressions
  – Higher-level control constructs
    • Performing a call
    • Loops
    • Ifs
Function Locals

<table>
<thead>
<tr>
<th>sp</th>
<th>.text</th>
</tr>
</thead>
<tbody>
<tr>
<td>space for local 2</td>
<td>f:</td>
</tr>
<tr>
<td>space for local 1</td>
<td># ... prologue ... #</td>
</tr>
<tr>
<td>ctrl link (caller $fp)</td>
<td>lw $t0, -8($fp)</td>
</tr>
<tr>
<td>ret address (caller $ra)</td>
<td>lw $t1, -12($fp)</td>
</tr>
<tr>
<td>param 1</td>
<td># ... epilogue ... #</td>
</tr>
<tr>
<td>param 2</td>
<td></td>
</tr>
<tr>
<td>caller’s AR</td>
<td></td>
</tr>
</tbody>
</table>
Function Returns

Return Stack

- caller’s AR
- param 2
- param 1
- ret address (caller $ra)
- ctrl link (caller $fp)
- space for local 2
- space for local 1

.push

.text
f:
    # ... prologue ... #
    lw $t0, -8($fp)
    lw $t1, -12($fp)
    lw $v0, -8($fp)
    j f_exit
f_exit:
    # ... epilogue ... #
Function Body: Expressions

- **Goal**
  - Serialize ("flatten") an expression tree
- **Use the same insight as the parser**
  - Use a work stack and a post-order traversal

```
+  1
  |
*  2
  |
id
```

```
Id:
1  2  *
+  1
```
Serialized Pseudocode

• Key insight
  – Use the stack pointer location as “scratch space”
  – At operands: push value onto the stack
  – At operators: pop source values from stack, push result

\[
\begin{align*}
\text{id} & \quad \text{id} \\
2 & \quad \text{2} \\
\ast & \quad \ast \\
\text{id} & \quad \text{id} \\
2 & \quad 2
\end{align*}
\]
Serialized MIPS

L1: push 2
L2: push id
L3: pop id into t1
L4: pop 2 into t0
L5: mult t0 * t1 into t0
L6: push t0

L1: li $t0 2
    sw $t0 0($sp)
    subu $sp $sp 4
L2: lw $t0 id
    sw $t0 0($sp)
    subu $sp $sp 4
L3: lw $t1 4($sp)
    addu $sp $sp 4
L4: lw $t0 4($sp)
    addu $sp $sp 4
L5: mult $t0 $t0 $t1
L6: sw $t0 0($sp)
    subu $sp $sp 4
Stmts

• By the end of the expression, our stack isn’t exactly as we left it
  – Contains the result of the expression
  – This is by design

1) Compute RHS expr on stack
2) Compute LHS *location* on stack
3) Pop LHS into $t1$
4) Pop RHS into $t0$
5) Store value $t0$ at address $t1$
Simple Assign, You Try

- Generate stack-machine style MIPS code for
  \[ id = 1 + 2; \]

**Algorithm**
1) Compute RHS expr on stack
2) Compute LHS *location* on stack
3) Pop LHS into $t1$
4) Pop RHS into $t0$
5) Store value $t1$ at address $t0$
Dot Access

• Fortunately, we know the offset from the base of a struct to a certain field statically
  – The compiler can do the math for the slot address
  – This isn’t true for languages with pointers!

```c
struct Demo inst;
struct Demo inst2;
inst.b.c = inst2.b.c + 1;
```

load this address  load this value
void v() {  
    struct Inner{  
        bool hi;  
        int there;  
        int c;  
    };  

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

    struct Demo inst;  
    inst.b.c = inst.b.c;  
}

**inst is based at -8($fp )**

**field b.c is -8 off the base**

---

**LHS**

```
subu $t0 $fp 16
sw $t0 0($sp)
```

**RHS**

```
lw $t0 -16($fp)
sw $t0 0($sp)
subu $sp $sp 4
```
Control Flow Constructs

• Function Calls
• Loops
• Ifs
Function Call

• Two tasks:
  – Put argument values on the stack (pass-by-value semantics)
  – Jump to the callee preamble label
  – Bonus 3rd task: save live registers
    • (We don’t have any in a stack machine)
  – Semi-bonus 4th task: retrieve result value
Function Call Example

```c
int f(int arg1, int arg2){
    return 2;
}

int main(){
    int a;
    a = f(a,4);
}
```

```assembly
li $t0 4     # push arg 2
sw $t0 0($sp) #
subu $sp $sp 4 #
lw $t0 -8($fp) # push arg 1
sw $t0 0($sp) #
subu $sp $sp 4 #
jal f          # goto f
addu $sp $sp 8 # tear down params
sw $v0 -8($fp) # retrieve result
Generating If-then Stmts

• First, get names for the true and false, and successor labels
• Generate the head of the loop
  – Make calls to the (not-yet placed!) true and false labels
• Generate the true branch
  – Place the true label
  – Write the body of the branch
  – Jump to the (not-yet placed!) successor label
• Generate the false branch (just like the true branch)
• Place the successor label
If-then Stmts

...  
if (val == 1){
  val = 2;
}
 ...

  lw  $t0  val   # evaluate condition LHS
  sw  $t0  0($sp) # push onto stack
  subu $sp $sp 4 #
  li   $t0 1  # evaluate condition RHS
  sw  $t0  0($sp) # push onto stack
  subu $sp $sp 4 #
  lw  $t1 4($sp) # pop RHS into $t1
  addu $sp $sp 4 #
  lw  $t0 4($sp) # pop LHS into $t0
  addu $sp $sp 4 #
  bne  $t0  $t1  L_0  # skip if condition false
  li   $t0 2  # Loop true branch
  sw  $t0  val  # end true branch
  j  L_0  # branch successor
  L_0:         # branch successor
...
If-then-elseStmts

... if (val == 1){
    val = 2;
} else {
    val = 3;
}
...

lw $t0 val       # evaluate condition LHS
sw $t0 0($sp)    # push onto stack
subu $sp $sp 4   #
li $t0 1         # evaluate condition RHS
sw $t0 0($sp)    # push onto stack
subu $sp $sp 4   #
lw $t1 4($sp)    # pop RHS into $t1
addu $sp $sp 4   #
lw $t0 4($sp)    # pop LHS into $t0
addu $sp $sp 4   #
bne $t0 $t1 L_1  # branch if condition
false
li $t0 2         # Loop true branch
sw $t0 val       # end true branch
j L_0            #
L_1:               # false branch
    ...          #
L_0:               # branch successor
Generating While Loops

• Very similar to if-then stmts
  – Generate a bunch of labels
  – Label for the head of the loop
  – Label for the successor of the loop

• At the end of the loop body
  – Unconditionally jump back to the head
While Loop

while (val == 1){
    val = 2;
}

L_0:
lw $t0 val          # evaluate condition LHS
sw $t0 0($sp)       # push onto stack
subu $sp $sp 4      #
li $t0 1            # evaluate condition RHS
sw $t0 0($sp)       # push onto stack
subu $sp $sp 4      #
lw $t1 4($sp)       # pop RHS into $t1
addu $sp $sp 4      #
lw $t0 4($sp)       # pop LHS into $t0
addu $sp $sp 4      #
bne $t0 $t1 L_1     # branch loop end
li $t0 2            # Loop body
sw $t0 val          #
j L_0                # jump to loop head
L_1:                  # Loop successor
...
P6 Helper Functions

• Generate (opcode, ...args...)
  – Generate(“add”, “T0”, “T0”, “T1”)
    • writes out add $t0, $t0, $t1
  – Versions for fewer args as well

• Generate indexed (opcode, “Reg1”, “Reg2”, offset)

• GenPush(reg) / GenPop(reg)

• NextLabel() – Gets you a unique label

• GenLabel(L) – Places a label
QtSpim