Code Generation for Control Flow Constructs
Roadmap

Last time:
– Got the basics of MIPS
– CodeGen for some AST node types

This time:
– Do the rest of the AST nodes
– Introduce control flow graphs
A Quick Warm-Up: MIPS for $id = 1 + 2$

General-Purpose 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 $t0$ at address $t1$

```
li $t0 1
sw $t0 0($sp)
subu $sp $sp 4
li $t0 2
sw $t0 0($sp)
subu $sp $sp 4
lw $t1 4($sp)
addu $sp $sp 4
lw $t0 4($sp)
addu $sp $sp 4
add $t0 $t0 $t1
sw $t0 0($sp)
subu $sp $sp 4
lw $t1 4($sp)
addu $sp $sp 4
lw $t0 4($sp)
addu $sp $sp 4
sw $t0 0($t1)
```
Same Example if id was Global

General-Purpose 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 $t0 at address $t1

<table>
<thead>
<tr>
<th>push 1</th>
<th>li</th>
<th>$t0 1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>sw</td>
<td>$t0 0($sp)</td>
</tr>
<tr>
<td></td>
<td>subu</td>
<td>$sp $sp 4</td>
</tr>
<tr>
<td>push 2</td>
<td>li</td>
<td>$t0 2</td>
</tr>
<tr>
<td></td>
<td>sw</td>
<td>$t0 0($sp)</td>
</tr>
<tr>
<td></td>
<td>subu</td>
<td>$sp $sp 4</td>
</tr>
<tr>
<td>pop opR</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>lw</td>
<td>$t1 4($sp)</td>
</tr>
<tr>
<td></td>
<td>addu</td>
<td>$sp $sp 4</td>
</tr>
<tr>
<td></td>
<td>lw</td>
<td>$t0 4($sp)</td>
</tr>
<tr>
<td></td>
<td>addu</td>
<td>$sp $sp 4</td>
</tr>
<tr>
<td></td>
<td>sw</td>
<td>$t0 0($sp)</td>
</tr>
<tr>
<td></td>
<td>subu</td>
<td>$sp $sp 4</td>
</tr>
<tr>
<td>pop LHS</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>la</td>
<td>$t0 -8($fp)</td>
</tr>
<tr>
<td>push RHS</td>
<td>id</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sw</td>
<td>$t0 0($sp)</td>
</tr>
<tr>
<td></td>
<td>subu</td>
<td>$sp $sp 4</td>
</tr>
<tr>
<td></td>
<td>lw</td>
<td>$t1 4($sp)</td>
</tr>
<tr>
<td></td>
<td>addu</td>
<td>$sp $sp 4</td>
</tr>
<tr>
<td>Do 1+2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>lw</td>
<td>$t0 4($sp)</td>
</tr>
<tr>
<td></td>
<td>addu</td>
<td>$sp $sp 4</td>
</tr>
<tr>
<td>push RHS</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>lw</td>
<td>$t0 4($sp)</td>
</tr>
<tr>
<td></td>
<td>addu</td>
<td>$sp $sp 4</td>
</tr>
<tr>
<td></td>
<td>sw</td>
<td>$t0 0($sp)</td>
</tr>
<tr>
<td></td>
<td>subu</td>
<td>$sp $sp 4</td>
</tr>
<tr>
<td>pop LHS</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>lw</td>
<td>$t0 4($sp)</td>
</tr>
<tr>
<td></td>
<td>addu</td>
<td>$sp $sp 4</td>
</tr>
<tr>
<td>pop RHS</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>lw</td>
<td>$t0 4($sp)</td>
</tr>
<tr>
<td></td>
<td>addu</td>
<td>$sp $sp 4</td>
</tr>
<tr>
<td>Do assign</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>sw</td>
<td>$t0 0($t1)</td>
</tr>
</tbody>
</table>

ctrl link (caller $fp)
ret address (caller $ra)
param 1
param 2
caller’s AR
Do We *Need* LHS computation?

This is a bit much when the LHS is a variable

- We end up doing a single load to find the address, then a store, then a load
- We know a lot of the computation at compile time
Static v Dynamic Computation

Static
– Perform the computation at compile-time

Dynamic
– Perform the computation at runtime

As applied to memory addresses…
– Global variable location
– Local variable
– Field offset
More Complex LHS addresses

Chain of dereferences

java: a.b.c.d

Array cell address

arr[1]
arr[c]
arr[1][c]
arr[c][1]
Dereference Computation

```c
struct LinkedList{
    int num;
    struct LinkedList& next;
}
```

```c
list.next.next.num = list.next.num
```

multi-step code to load this address

multi-step code to load this value

- Get base addr of list
- Get offset to next field
- Load value in next field
- Get offset to next field
- Load value in next field
- Get offset to num field
- Load that address
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
```
We Need a New Tool

Control Flow Graph

- Important representation for program optimization
- Helpful way to visualize source code
Control Flow Graphs: the Other CFG

Think of a CFG like a flowchart
- Each block is a set of instructions
- Execute the block, decide on next block
Basic Blocks

Nodes in the CFG
Largest run of instructions that will always be executed in sequence

Line1: li $t0 4
Line2: li $t1 3
Line3: add $t0 $t0 $t1
Line4: sw $t0 val
Line5: j Line2
Line6: sw $t0 0($sp)
Line7: subu $sp $sp 4
Conditional Blocks

Branch instructions cause a node to have multiple out-edges.

Entry: li $t0 3
lw $t1 0($sp)
beq $t0 $t1 Exit

True: sw $t2 val
nop

false

Entry: li $t0 3
lw $t1 0($sp)
beq $t0 $t1 Exit

Exit: li $v0 10
syscall
Generating If-then Stmts

First, get label for the exit
Generate the head of the if
  – Make jumps to the (not-yet placed!) exit label
Generate the true branch
  – Write the body of the true node
Place the exit 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          # If true branch
sw $t0 val        # end true branch
nop                # branch successor
L_0:               ...
...
Conditional Blocks

Entry: li $t0 3
lw $t1 0($sp)
beq $t0 $t1 False
True: sw $t2 val
j Exit
False: sw $t2 val2
nop
Exit: li $v0 10
syscall

Entry: li $t0 3
lw $t1 0($sp)
beq $t0 $t1 False
True: sw $t2 val
j Exit
False: sw $t2 val2
nop
Jump
Exit: li $v0 10
syscall
Generating If-then-Else Stmts

First, get name for the false and exit labels
Generate the head of the if
– Make jumps to the (not-yet placed!) exit and false labels

Generate the true branch
– Write the body of the true node
– Jump to the (not-yet placed!) exit label

Generate the false branch
– Place the false label
– Write the body of the false node

Place the exit label
If-then-Else Stmts

... 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
j L_0            # end true branch
L_1:
...
L_0:               # false branch
# branch successor
While Loops CFG

Entry: li $t0 3
lw $t1 -8($fp)
slt $t2 $t0 $t1
bne $t2 $0 Exit

Body: lw $t0 -8($fp)
li $t1 1
sub $t0 $t0 $t1
j Entry

Exit: li $v0 10
syscall

Unconditional jump

Fallthrough

Jump on false
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
  li $t0 2         #
  sw $t0 val       #
  j L_0            # jump to loop head
L_1:               # Loop successor
  ...
```
A Note on Conditionals

We lack instructions for branching on most relations
- No “branch if reg1 < reg2”
- Instead we use the slt “set less than”
  - `slt $t2 $t1 $t0`
    - $t2$ is 1 when $t1 < t0$
    - $t2$ otherwise set to 0
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
Questions?

Looking forward
– More uses of the CFG
– Program analysis
– Optimization

Homework: see QtSpim resources
QtSpim