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

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
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 $t1 at address $t0
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 vs. 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]
struct LinkedList{
    int num;
    struct LinkedList& next;
}

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

0x1002F000
    num: 3
    next: 0x0

list.next.next

0x10040000
    num: 2
    next: 0x1002F000

list.next

list
    next: 0x10040000
Control-Flow Constructs

Function calls
Loops
If statements
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)

On return

- Tear down the actual parameters
- Retrieve and push the 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           # call f (via jump and link)
addu $sp $sp 8   # tear down actual parameters
sw $v0 0($sp)    # retrieve and push the result
subu $sp $sp 4   #
```
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 which block to execute next
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

Exit: li $v0 10
      syscall
Generating If-Then Statements

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 Statements

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

lw $t0 val
sw $t0 0($sp)  # evaluate condition LHS
subu $sp $sp 4  # push onto stack
li $t0 1
sw $t0 0($sp)  # evaluate condition RHS
subu $sp $sp 4  # push onto stack
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  # branch if condition false
li $t0 2
sw $t0 val
nop
L_0:
    ...  # end true branch

nop  # successor label
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

Exit: li $v0 10
    syscall
Generating If-Then-Else Statements

First, obtain names to use for the labels of the
- false branch
- successor

Generate code for the branch condition
- Can emit a jump to the (not-yet placed!) false-branch label

Generate code for the true branch
- Emit the code for the body of the true branch
- Emit a jump to the (not-yet placed!) successor label

Generate code for the false branch (similar to the true branch)
- Emit the false-branch label
- Emit the code for the body of the false branch

Emit the successor label
If-Then-Else Statements

... 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 #
il $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 # true branch
sw $t0 val
j L_0 # end true branch
L_1: # false branch
    ... # successor label
L_0: # successor label
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
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
lw $t0 4($sp)  # pop LHS into $t0
addu $sp $sp 4  #
lw $t0 4($sp)  #
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
An Alternative Approach to Conditionals

slt “set less than”

• \texttt{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 \texttt{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
QtSpim