CS 536 Announcements for Wednesday, April 19, 2023

Last Time
• variable access at runtime
  • local vs global variables
  • static vs dynamic scopes

Today
• start looking at details of MIPS
  • code generation

Next Time
• continue code generation

Compiler Big Picture

Scanner

Parser

Static-Semantic Analysis

IR Codegen

Optimizer

MC Codegen
Compiler Back End: Design Decisions

When do we generate?
- directly from AST
- during SDT

How many passes?
- fewer passes
- more passes

What do we generate?
- machine code
- intermediate representation (IR)

Possible IRs
- CFG (control-flow graph)
- 3AC (three-address code)
  - instruction set for a fictional machine
  - every operator has at most 3 operands
  - provides illusion of infinitely many registers
  - "flatten out" expressions
3AC Example

3AC instruction set

**Assignment**
- $x = y \text{ op } z$
- $x = \text{ op } y$
- $x = y$

**Jumps**
- if ( $x \text{ op } y$) goto $L$

**Indirection**
- $x = y[z]$
- $y[z] = x$
- $x = \&y$
- $x = *y$
- $*y = x$

**Call/Return**
- param $x,k$
- retval $x$
- call $p$
- enter $p$
- leave $p$
- return
- retrieve $x$

**Type Conversion**
- $x = \text{ AtoB } y$

**Labeling**
- label $L$

**Basic Math**
- times, plus, etc.

Example

**source code**

```c
if (x + y * z > x * y + z)
    a = 0;

b = 2;
```

**3AC code**

```3AC
tmp1 = y * z
tmp2 = x + tmp1
tmp3 = x * y
tmp4 = tmp3 + z
if (tmp2 <= tmp4) goto L
    a = 0
L: b = 2
```

3AC representation

- each instruction represented using a structure called a “quad”
  - space for the operator
  - space for each operand
  - pointer to auxiliary info (label, successor quad, etc.)
- chain of quads sent to an architecture-specific machine-code-generation phase

---

Week 12 (W)  Page 3
Code Generation

For brevis
- skip building a separate IR
- generate code by traversing the AST
  - add codeGen methods to AST nodes
  - directly emit corresponding code into file

Two high-level goals
- generate correct code
- generate efficient code

Simplified strategy
Make sure we don't have to worry about running out of registers
- for each operation, put all arguments on the stack
- make use of the stack for computation
- only use two registers for computation

Different AST nodes have different responsibilities
Many nodes simply "direct traffic"
- ProgramNode.codeGen
- List-node types
- DeclNode
  - RecordDeclNode
  - FnDeclNode
  - VarDeclNode
Code Generation for Global Variable Declarations

Source code:

```plaintext
integer name;
record MyRecord instance;
```

In AST: VarDeclNode

Generate:

```
.data
.align 2   # align on word boundaries
_name: .space N   # N is the size of variable
```

Size of variable

- for scalars, well-defined: integer, boolean are 4 bytes
- for records: 4*size of records

Code Generation for Function Declarations

Need to generate

- preamble
- prologue
- body
- epilogue
MIPS Crash Course

Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>$sp</td>
<td>stack pointer</td>
</tr>
<tr>
<td>$fp</td>
<td>frame pointer</td>
</tr>
<tr>
<td>$ra</td>
<td>return address</td>
</tr>
<tr>
<td>$v0</td>
<td>used for system calls and to return int values from function calls, including the syscall that reads an int</td>
</tr>
<tr>
<td>$f0</td>
<td>used to return double values from function calls, including the syscall that reads a double</td>
</tr>
<tr>
<td>$a0</td>
<td>used for output of int and string values</td>
</tr>
<tr>
<td>$f12</td>
<td>used for output of double values</td>
</tr>
<tr>
<td>$t0 - $t7</td>
<td>temporaries for ints</td>
</tr>
<tr>
<td>$f0 - $f30</td>
<td>registers for doubles (used in pairs; i.e., use $f0 for the pair $f0, $f1)</td>
</tr>
</tbody>
</table>

Program structure

Data
- label: .data
- variable names & size; heap storage

Code
- label: .text
- program instructions
- starting location: main
MIPS Crash Course (cont.)

Data

<table>
<thead>
<tr>
<th>name</th>
<th>type</th>
<th>value(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>vl</td>
<td>.word</td>
<td>10</td>
</tr>
<tr>
<td>a1</td>
<td>.byte</td>
<td>'a', 'b'</td>
</tr>
<tr>
<td>a2</td>
<td>.space</td>
<td>40</td>
</tr>
</tbody>
</table>

40 here is allocated space – no value is initialized

Memory instructions

**lw**  
`register_destination, RAM_source`
- copy word (4 bytes) at source RAM location to destination register.

**lb**  
`register_destination, RAM_source`
- copy byte at source RAM location to low-order byte of destination register

**li**  
`register_destination, value`
- load immediate value into destination register

**sw**  
`register_source, RAM_dest`
- store word in source register into RAM destination

**sb**  
`register_source, RAM_dest`
- store byte in source register into RAM destination

Arithmetic instructions

**add**  
`$t0,$t1,$t2`

**sub**  
`$t2,$t3,$t4`

**addi**  
`$t2,$t3, 5`

**addu**  
`$t1,$t6,$t7`

**subu**  
`$t1,$t6,$t7`

**mult**  
`$t3,$t4`

**div**  
`$t5,$t6`

**mfhi**  
`$t0`

**mflo**  
`$t1`
MIPS Crash Course (cont.)

Control instructions

```
b  target
beq  $t0,$t1,target
blt  $t0,$t1,target
ble  $t0,$t1,target
bgt  $t0,$t1,target
bge  $t0,$t1,target
bne  $t0,$t1,target

j  target
jr  $t3

jal  sub_label    # "jump and link"
```

Check out: MIPS tutorial

https://minnie.tuhs.org/CompArch/Resources/mips_quick_tutorial.html