6.035 Infosession 3

Spring 2016

(Part of Slides from previous years)
Code Generation at a Glance

• Translate all the instructions in the intermediate representation to assembly language
• Handle expressions
• Allocate space
  • Local variables
  • Global variables
  • Arrays
• Adhere to function calling conventions
• Short circuiting of conditionals
• Runtime checks
Design Intermediate Representation

• Expressive enough to be able to perform analysis and transformation

• Concrete enough to be able to easily generate machine code

What to use as intermediate representation?
• Same as high IR? (with semantic restrictions)
• Assembly Language?
Design #1: From AST to Assembly

• Will have a compiler immediately

• But it will make your life difficult when doing most of the optimizations

• Ideally, want to provide framework for performing code transformations easily
Design Intermediate Representation

• Static Single Assignment

• Infinite register machine

• Stack-based machine
## Expression Evaluation Alternatives

<table>
<thead>
<tr>
<th>High Level</th>
<th>Temporaries</th>
<th>In Place</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a = 1 \times 2 + 3 \times 4)</td>
<td>(t1 = 1 \times 2)</td>
<td>(t1 = 1)</td>
</tr>
<tr>
<td>(b = a \times a + 1)</td>
<td>(t2 = 3 \times 4)</td>
<td>(t1 *= 2)</td>
</tr>
<tr>
<td></td>
<td>(a = t1 + t2)</td>
<td>(t2 = 3)</td>
</tr>
<tr>
<td></td>
<td>(t3 = a \times a)</td>
<td>(t2 = t2 \times 4)</td>
</tr>
<tr>
<td></td>
<td>(b = t3 + 1)</td>
<td>(t1 += t2)</td>
</tr>
</tbody>
</table>

Code Generation at a Glance

• Translate all the instructions in the intermediate representation to assembly language

• Allocate space
  • Local variables.
  • Global variables
  • Arrays

• Adhere to function calling conventions

• Short circuiting of conditionals

• Runtime checks
Variables

Start from Names (Source code) and Descriptors (high IR)

Intermediate allocation

• Everything on the stack?
  • Later optimize by moving to registers
• Everything in a register?
  • “Spill” excess to the stack
• Other techniques...

• Final allocation (fixed registers + stack)
• Register allocation is hard! (so start simple!)
Conditionals

Must eventually become labels and jumps

if (a) { foo } else { bar }

Becomes:

cmp $0, a
jne l1
  bar
  jmp l2
l1:foo
l2: //...
Target: x86-64

- Stack values are 64-bit (8-byte)
- Values in decaf are 64-bit (integer) or 1-bit (boolean)
- For this phase, we are not optimizing for space
- Use 64-bits (quadword) for ints and bools.
- Use instructions that operate on 64-bit values for stack and memory operations, e.g. mov
- Same for arithmetic operations
## Registers (Linux Calling Convention)

<table>
<thead>
<tr>
<th>Register</th>
<th>Purpose</th>
<th>Saved across calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>temp register; return value</td>
<td>No</td>
</tr>
<tr>
<td>%rbx</td>
<td>callee-saved register</td>
<td>Yes</td>
</tr>
<tr>
<td>%rcx</td>
<td>used to pass 4th argument to functions</td>
<td>No</td>
</tr>
<tr>
<td>%rdx</td>
<td>used to pass 3rd argument to functions</td>
<td>No</td>
</tr>
<tr>
<td>%rsp</td>
<td>stack pointer</td>
<td>Yes</td>
</tr>
<tr>
<td>%rbp</td>
<td>callee-saved; base pointer</td>
<td>Yes</td>
</tr>
<tr>
<td>%rsi</td>
<td>used to pass 2nd argument to functions</td>
<td>No</td>
</tr>
<tr>
<td>%rdi</td>
<td>used to pass 1st argument to functions</td>
<td>No</td>
</tr>
<tr>
<td>%r8</td>
<td>used to pass 5th argument to functions</td>
<td>No</td>
</tr>
<tr>
<td>%r9</td>
<td>used to pass 6th argument to functions</td>
<td>No</td>
</tr>
<tr>
<td>%r10-r11</td>
<td>temporary</td>
<td>No</td>
</tr>
<tr>
<td>%r12-r15</td>
<td>callee-saved registers</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Assembly Instructions

• Check out the x86-64 Architecture guide.
  • On course’s Resources page
• We are using AT&T assembler syntax (gcc)
• Instructions have the form:
  • `operator op1 op2`, which is equivalent to
    `op2 = op1 operator op2`
• `$x` denotes immediate integer (base 10) value `x`
• `%r??` is a register
• You can use names of global variables directly
Allocating Read Only Data

All Read-Only data in the text segment

Integers
- use immediates

Strings
- use the .string macro

```
.section .rodata
.msg:
   .string "Five: %d\n"

.section .text
.globl main
main:
   enter $0, $0
   mov $.msg, %rdi
   mov $5, %rsi
   mov $0, %rax
   call printf
   leave
   ret
```
Allocating Global Variables

- Allocation: Use the assembler's .comm directive
- Use name or
- Use PC relative addressing
- %rip is the current instruction address
- X(%rip) will add the offset from the current instruction location to the space for x in the data segment to %rip
- Creates easily relocatable binaries

...  
.section .text
.globl main

main:
enter$0, $0
mov $.msg, %rdi
mov x, %rsi
mov $0, %rax
call printf
leave
ret

.comm x, 8, 8
Allocating Global Variables

- Allocation: Use the assembler's .comm directive
- Use name or
- Use PC relative addressing
- %rip is the current instruction address
- X(%rip) will add the offset from the current instruction location to the space for x in the data segment to %rip
- X is a constant offset
- Creates easily relocatable binaries

```assembly
.section .text
.globl main
main:
    enter$0, $0
    mov $.msg, %rdi
    mov X(%rip), %rsi
    mov $0, %rax
    call printf
    leave
    ret

.comm x, 8, 8
```
Addressing Modes

• (%reg) is the memory location pointed to by the value in %reg

• movq $5, -8(%rbp)
Arrays

- What code would you write for?

...  
mov $5, %r10
mov $4, %r11
???...

.comma a, 8 * 10, 8

The data segment grows toward larger addresses.

How to access an array element?

We want something like
- `base + offset * type_size`

AT&T Asm Syntax:
- `offset(base, index, scale) = offset + base + (index * scale)`
Arrays

• What code would you write for?

... 
mov $5, %r10
mov $4, %r11
mov %r10, a(, %r11, 8)

.comm a, 8 * 10, 8
Runtime Checks

• Array bounds:
  • For every read and write for a[idx]:

    if (idx < 0 || idx >= length_a) { exit(-1); }

• Program returns
  • If a function returns a value, the execution must not fall off without returning a value (i.e., check that you always assign a value to the return register %rax)
  • Error handling: error(-2)
Procedure Abstraction

• Stack Frames
• Calling Convention

• What to do with live registers across a procedure call?
  • Callee Saved (belong to the caller)
  • %rsp, %rbp, %r12-15
  • The caller must assume that all other registers will be used by the callee
Generated Code

• Your code for this stage **should** be inefficient!
• Stack locations for all temporary values and variables
• For an expression, load operand value(s) into register(s) then perform operation and write to location in stack
• Use regs %r10 and %r11 for temporaries
Design a Low-Level IR

• Don’t worry about machine portability
  • flat low-level IRs.
  • 2 address code: operand\textsuperscript{1} op = operand\textsuperscript{2}
  • 3 address code: result = operand\textsuperscript{1} op operand\textsuperscript{2}

• Close to ASM language (linear list of instructions)
  • binops, labels, jumps, calls, names, locations

• Make it flexible -- operands can be names or machines locations
  • First generate low-level IR with names, then a later pass resolves names to locations
Compiler Flow

Template approach

• break/continue and short-circuiting
• Translate from AST to low IR

Then have multiple passes to “lower” IR to machine level

• resolve names to locations on stack
• activation frame sizes for stack size calculations
• pass arguments to methods for a call
Compilation

• Compile C file to assembly:
  
  gcc -O0 -S -fverbose-asn foo.c -o foo.s

• Building generated assembly code (gcc is in thise case a front-end for “as”):
  
  gcc -c foo.s -o foo.o

• From object file to assembly: objdump -d foo.o

• Compile to executable: gcc foo.s -o foo
• **Due:** March 18th!!!

• Worth **25%** of the grade

• Documentation 20%, Testing 80%

• **Start early!**