6.035
Project 3: Unoptimized Code Generation

Jason Ansel
MIT - CSAIL
Quiz Monday

- 50 minute quiz Monday
- Covers everything up to yesterday's lecture
  - Lexical Analysis (REs, DFAs, NFAs)
  - Syntax Analysis (CFGs, top-down parsing, bottom-up parsing)
  - Semantics Analysis (type checking, type systems, attribute grammars)
- Questions similar to miniquizzes, but a bit harder
Project 3 Roadmap

- **Design and Checkpoint**
  - Due Monday March 10\textsuperscript{th}
  - Checkpoint
    - Document of your proposed design (email me)
    - Create a tarball of what you have
    - If you get codegen to work, no effect
    - If you have problems at end, we will be very harsh if you haven’t done much work by the checkpoint

- **Group meeting**
  - not mandatory, meet with me if you want.

- **Final Implementation and Report**
  - Due on March 16\textsuperscript{th}
Course Machines

- Meet tyner.csail.mit.edu and silver.csail.mit.edu!
- Two AMD64 machines
  - dual processor
  - dual core per processor
  - 8 gigs of RAM
- Use them for running your compiled assembly code.
  - User: le0X, password in le0X-pass in group dir
- Can access files over ssh:
  - git clone athena.dialup.mit.edu:/mit/6.035/group/...
Unoptimized Code Generation

- Translate all the instructions in the intermediate representation to assembly language
  - Allocate space for the variables.
    - Globals
    - Arrays
  - Adhere to calling conventions
  - Short circuiting
  - Runtime checks
Low-level IR design choices

- Classes to use
  - Same has high IR? (with restrictions)
  - Newly added classes?
  - Mix?

- Level of the low IR
  - How close to assembly?

- Alternate representations?
  - Single Static Assignment
  - Infinite register machine (DirectX, etc)
  - Stack-based machine (Java bytecode, etc)
Math ops

- High Level:
  a = 1 \times 2 + 3 \times 4
  b = a \times a + 1

- Temporaries:
  t1 = 1 \times 2
  t2 = 3 \times 4
  a = t1 + t2
  t3 = a \times a
  b = t3 + 1

- In place:
  t1 = 2 \quad (movq)
  t1* = 1 \quad (imul)
  t2 = 4 \quad (movq)
  t2* = 3 \quad (imul)
  a = t2 \quad (mov)
  a += t1 \quad (add)
  t3 = a \quad (mov)
  t3* = a \quad (imul)
  b = 1 \quad (movq)
  b += t3 \quad (add)
Variables / Temporaries

- Names (input) become...
- 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!
  - Start simple
Control flow

- Must eventually become labels and jumps
  
  if (a) { foo } else { bar }

- Becomes:
  
  cmp $0, a
  jne l1
  bar
  jmp l2
  l1:
  foo
  l2:
x86-64 (AMD64)

- Stack values are 64-bit (8-byte)
- Values in decaf are 32-bit or 1-bit
- 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 mem operations, e.g. mov
- Arithmetic instructions have 32-bit operands, add, sub, etc
## Registers

<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</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 function</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 function</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>
ASM Instructions

- Check out the x86-64 Architecture guide.
- Remember that we are using AT&T assembler syntax (gcc)
- Usually, operator op1 op2
  - op2 = op1 operator op2
- $x$ denotes immediate integer (base 10) value $x$
- %r?? is a register
- You can use names of globals directly
ASM Instructions

• Some caveats:
  – mov instructions sometimes need a suffix if the assembler cannot resolve the data size
  – For example when you move an immediate into memory: `movq $1, -8(%rbp)`
Registers

- Instructions allow only limited memory operations
  - add -4(%rbp), -8(%rbp)
  - mov -4(%rbp), %r10
  add %r10, -8(%rbp)

- Important for performance
  - limited in number

- Special registers
  - %rbp base pointer
  - %rsp stack pointer
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
```
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 re-locatable binaries

```assembly
... .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 .comm name, size, alignment
```

The .comm directive allocates storage in the data section. The storage is referenced by the identifier name. Size is measured in bytes and must be a positive integer. Name cannot be predefined. Alignment is optional. If alignment is specified, the address of name is aligned to a multiple of alignment.
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 re-locatable binaries

```
...  
.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  

.comm name, size, alignment  
The .comm directive allocates storage in the data section. The storage is referenced by the identifier name. Size is measured in bytes and must be a positive integer. Name cannot be predefined. Alignment is optional. If alignment is specified, the address of name is aligned to a multiple of alignment
```
Addressing Modes

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

- movq $5, -8(%rbp)
What about Arrays

- What code would you write for?
  

  ```
  ...
  mov $5, %r10
  mov $4, %r11
  ???
  ???
  ...
  .comm a, 8 * 10, 8
  ```
Array Addressing

- 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)
What about Arrays

- What code would you write for?

```asm
...  
mov $5, %r10
mov $4, %r11
???
...
.comma a, 8 * 10, 8
```
What about Arrays

- What code would you write for?


... 
mov $5, %r10
mov $4, %r11
mov %r10, a(, %r11, 8)
...
.comm a, 8 * 10, 8
Procedure Abstraction

- Stack frames (activation records)
- Calling convention
Registers

• 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
Your 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
  - Why?
Example

if (x == 20) { x = 0; } else { x = 5; }

mov -24(%rbp), %r10
mov $20, %r11
cmp %r10, %r11
mov $0, %r11
mov $1, %r10
cmove %r10, %r11
mov %r11, -32(%rbp)

movq $1, %r11
cmp %r10, %r11
je .true_block
mov $5, %r10
jmp .done

.true_block:
mov $5, %r10
mov %r10, -24(%rbp)

.done:
Reusing Temporaries

- You can allocate a temporary for each expression
- You can reuse temporaries very simply
- Ex:
  - eval E1 into T1
  - eval E2 into T2
  - T3 = T1 + T2
- After T3 is assigned, do we need T1 and T2?
Reusing Temporaries

• Simple stack algorithm:
  – Keep a count for temporaries $c$ (init to 0)
    • create a temporary location named $T_c$
    • each $T_c$ is a different location on the stack
    • $T_c$ is reused!
  – While traversing IR
    • Whenever a temporary name is used as an operand, decrement $c$ by 1
    • Whenever a temporary name is generated use $T_c$ and increase $c$ by 1
Reusing Temporaries Example

\[ x = 1 \times 2 + 3 \times 4 - 5 \times 6 \]

<table>
<thead>
<tr>
<th>Statement</th>
<th>Value of T after statement (0 at start)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T0 = 1 \times 2 )</td>
<td>1</td>
</tr>
<tr>
<td>( T1 = 3 \times 4 )</td>
<td>2</td>
</tr>
<tr>
<td>( T0 = T0 + T1 )</td>
<td>1</td>
</tr>
<tr>
<td>( T1 = 5 \times 6 )</td>
<td>2</td>
</tr>
<tr>
<td>( T0 = T0 - T1 )</td>
<td>1</td>
</tr>
<tr>
<td>( x = T0 )</td>
<td>0</td>
</tr>
</tbody>
</table>
Another Intermediate Representation

- You could translate your AST directly into ASM code
- But for the next stages you will be optimizing your code
  - These optimizations are defined to operate at a low level
    - EX, register allocation after locations have been assigned to all temps and vars
Design a Low IR

- Don’t worry about machine portability
  - flat low-level IRs.
    - 2 address code looks nice+
      - operand\(_1\) operation= operand\(_2\)
  - Close to ASM language (linear list)
    - binops, labels, jumps, calls, names, locations
- Make it flexible
  - operands can be names or machines locations
  - first generate lowIR with names, then a later pass resolves names to locations
Possible Compiler Flow

- I recommend the template approach
  - break/continue and short-circuiting are not hard
- Use the template approach to translate AST to low IR
- Then have multiple passes to “lower” it to machine level
  - resolve names to locations on stack
  - activation frame sizes for stack size calculations
  - pass arguments to methods for a call