CSE 401 – Compilers

Lecture 19: x86-64, GNU Assembler, and Project Code Generation, Part I

Michael Ringenburg

Winter 2013

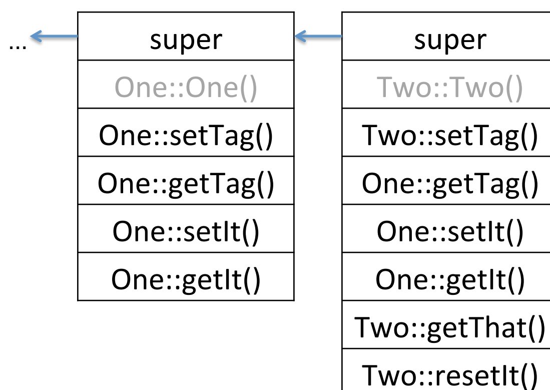
Reminders/ Announcements

- Midterms are graded
  - If you didn't pick yours up on Friday, stop by my office this afternoon before 3, or during my office hours Wednesday.
- Project part 3 due this Friday
- Part 4 will be due on Friday, March 15 (last day of class). I will put the assignment out this week, once we finish the necessary material.

Finishing Up From Friday: Quick Review

- Recall, every *class* has a method dispatch table.
  - All objects contain a pointer (typically their first word) to the method table of their *runtime* class.
- Method dispatch table of derived classes:
  - If A extends B, and B has *n* methods,
  - First *n* (non-default-constructor) method entries of A's method table will be for the same methods, and in the same order, as the *n* entries of B's table.
  - A's method table will also contain a pointer to B's method table – typically first word.
    - Default constructor is typically the second word, if present in language.

Finishing Up From Friday: Quick Review




```
class One {
  int tag;
  int it;
  void setTag()      { … }
  int getTag()       { … }
  void setIt(int it) { … }
  int getIt()        { … }
}

class Two extends One {
  int it;
  void setTag()      { … }
  int getThat()      { … }
  void resetIt()     { … }
}
```

Finishing Up From Friday: Runtime Type Checking

- Why the super class pointer, if we already have code pointers for every method?
  - Runtime type checking
- Use the method table for the class as a “runtime representation” of the class
- The test for “o instanceof C” is
  - Is o’s method table pointer == &C$$? If so, result is `true`.
  - Otherwise, recursively get pointer to superclass method table from the method table and check that
  - Stop when you reach Object (or a null pointer, depending on how you represent things)
    - If no match by the top of the chain, result is “false”
- Same test as part of check for legal downcast

o instanceof C pseudocode

```
clsTable = o->methodTable; // first word of o
while (clsTable != null) {
   if (clsTable == &C$$) {
      return true;
   }
   clsTable = clsTable[0];  // check super class
}
return false;
```

- Many compilers have a “lowering” phase, prior to code gen (and often prior to optimization) that converts complex language features to lower-level code like this.
  - E.g., EDG front end has option to lower C++ to C (plus library calls)

Agenda for the Next Two Lectures

- What you need for your project code generation
  - You'll follow the basic recipes described in last week's lectures, with a few differences:
    - GNU (AT&T) assembler instead of Intel/Microsoft
    - x86-64 instead of IA-32
  - The bootstrap program
    - A simple C program that sets up the stack/heap and calls your generated assembly – saves you a lot of work
    - Also implements the system interface (e.g. I/O, memory)
    - Essentially, a "mini runtime library".
  - A very basic code generation strategy (and examples)

Motivation: Project Compiler Target

- Compiler output is an assembly-language file that is linked to the "real" main program written in C (described later)
  - Lets the C library set up the stack, heap; handle I/O, etc.
- Target code is Linux x86-64 gcc asm
  - Examples on these slides use this notation

Intel vs. GNU Assembler

- The GNU assembler uses AT&T syntax. Main differences:

	Intel/Microsoft	AT&T/GNU as
Operand order: op a,b	a = a op b (dst first)	b = a op b (dst last)
Memory address	[baseregister+offset]	offset(baseregister)
Instruction mnemonics	mov, add, push, …	movl, addl, pushl [operand size/ type is added to end – l is 32 bit integer, q if 64 bit int]
Register names	eax, ebx, ebp, esp, …	%eax, %ebx, %ebp, %esp, …
Constants	17, 42	$17, $42
Comments	; to end of line	# to end of line or /* … */
Data location size directives	dd, dq	.int, **.quad (64 bit)**

Code Sample (64 bit): Intel vs. Gnu

```
; Intel/Microsoft prologue
;
push    rbp
mov     rbp,rsp
sub     rsp,16

; Store rdi to frame ptr - 8
movq    [rbp – 8],rdi
```

```
# GNU/AT&T prologue
#
pushq   %rbp
movq    %rsp,%rbp
subq    $16,%rsp

# Store rdi to frame ptr -8
movq    %rdi,-8(%rbp)
```

64-bit Method Tables: Intel vs. Gnu

```
        .data
One$$:  dq  0        ; no superclass
        dq  One$One
        dq  One$setTag
        dq  One$getTag
        dq  One$setIt
        dq  One$getIt

Two$$:  dq  One$$    ; parent
        dq  Two$Two
        dq  Two$setTag
        dq  One$getTag
        dq  One$setIt
        dq  One$getIt
        dq  Two$getThat
        dq  Two$resetIt
```

```
        .data
One$$:  .quad  0      # no superclass
        .quad  One$One
        .quad  One$setTag
        .quad  One$getTag
        .quad  One$setIt
        .quad  One$getIt

Two$$:  .quad  One$$  # parent
        .quad  Two$Two
        .quad  Two$setTag
        .quad  One$getTag
        .quad  One$setIt
        .quad  One$getIt
        .quad  Two$getThat
        .quad  Two$resetIt
```

x86-64

- Designed by AMD and announced in 1999-2000. First processors in 2003. (AMD called it AMD64.)
- Intel bet on Itanium for 64-bit processors, but just in case had a not-so-secret project to add AMD64 to the Pentium 4
  - General consensus: Itanium was too radical of a departure from IA-32, hard to get compiler and OS buy-in. AMD64 was much closer to IA-32, easier to adopt.
  - Announced in 2004 (first called IA-32e, then EM64T, finally Intel 64)
- Generic term is now x86-64

Some x86-64 References

(Links on project web)

- x86-64 Machine-Level Programming
  - Earlier version of sec. 3.13 of *Computer Systems: A Programmer's Perspective* 2nd ed. by Bryant & O'Hallaron (CSE 351 textbook)
- x86-64 Instructions and ABI
  - Handout for University of Chicago CMSC 22620, Spring 2009, by John Reppy
- ~~From www.x86-64.org:~~
  - ~~System V Application Binary Interface AMD64 Architecture Processor Supplement~~
  - ~~Gentle Introduction to x86-64 Assembly~~
  - These links were provided for past instances of this course, but the x86-64.org site appears dead – if you can find copies of this information elsewhere, please post to the discussion board.

x86-64 Main features

- 16 64-bit general registers; 64-bit integers (but int typically defaults to 32 bits in most compilers; long is 64 bits)
- 64-bit address space; pointers are 8 bytes
- 8 additional SSE registers (total 16)
  - More efficient than x87 floating point, used by default for fp
- Register-based function call conventions
- Additional addressing modes (pc relative)
  - More efficient position independent code (shared libraries)
- 32-bit legacy mode
- Some pruning of old features

x86-64 registers

- 16 64-bit general registers
  - 64 bit version of original registers: %rax, %rbx, %rcx, %rdx, %rsi, %rdi, %rbp, %rsp,
  - Extended registers: %r8-%r15
- Original registers (rax-rsp) can be used as 64-bit ints or pointers, or 32-bit ints (upper half set to 0 automatically)
  - 32 bit version of %rax is %eax, etc.
  - Also possible to reference low-order 16- and 8-bit chunks

x86-64 Function Calls

- First 6 arguments in registers, rest on the stack
- Result returned in %rax
- Stack frame should be 16-byte aligned when call instruction is executed
  - Allows callee to assume stack frame is 16-byte aligned initially. Some instructions perform better or require 16-byte aligned data (SSE instructions).
- We'll use %rbp as frame pointer, but compilers often adjust %rsp once on function entry and reference locals relative to %rsp using a fixed-size stack frame
  - The latter requires a bit of bookkeeping by the compiler, but frees up %rbp

x86-Register Usage

- %rax – function result
- Arguments 1-6 passed in these registers
  - %rdi, %rsi, %rdx, %rcx, %r8, %r9
  - OO languages: “this” pointer is first argument, in %rdi
- %rsp – stack pointer; value must always be 8-byte aligned, and 16-byte aligned when calling a function
  - So callee can assume initial 16-byte alignment. Makes it easier to ensure locals are 16-byte aligned if needed.
  - Your functions won’t assume this, but may call external functions (e.g., I/O, allocation) that do.
- %rbp – frame pointer (optional use)
  - We’ll use it (makes life easier).

x86-64 Register Save Conventions

- A called function must preserve these registers (or save/restore them if it wants to use them)
  - %rbx, %rbp, %r12-%r15
- %rsp isn’t on the “callee save list”, but needs to be properly restored for return
  - I.e., pop your stack frame!
- All other registers can change across a function call – so if you’ll need them after a call, save them somewhere!

x86-64 Function Call

- Caller places up to 6 arguments in registers, rest on stack, then executes call instruction (which pushes 8-byte return address)
- On entry, called function prologue is like the 32-bit version:

```
pushq   %rbp              # store old frame pointer
movq    %rsp,%rbp         # set new frame pointer
subq    $framesize,%rsp   # allocate stack frame
```

- Note that 8 byte return address plus 8 byte frame pointer gets you back to 16 byte alignment (assuming you were when call was made).

x86-64 Function Return

- Called function puts result in %rax (if any) and restores any callee-save registers if needed
- Called function returns with:

```
movq    %rbp,%rsp    # Pop stack frame
popq    %rbp         # Restore old frame pointer
ret                  # Pops and jumps to return address
```

  – Same logic as 32-bit
  – Can replace movq/popq with "leave" instruction, as was suggested last week.
- If caller allocated space for arguments it deallocates as needed
  – And stores result from %rax if needed.

The Nice Thing About Standards…

- The above is the System V/AMD64 ABI convention (used by Linux, OS X)
- Microsoft's x64 calling conventions are slightly different (sigh…)
  - First four parameters in registers %rcx, %rdx, %r8, %r9; rest on the stack
  - Caller must allocate stack frame space for callee to save four values passed in parameter registers, even if fewer than four are passed. Additional parameters (5+) placed above these on stack.
    - They say it simplifies (their implementation of) variable argument and unprototyped functions.
- Not relevant for us (linux), but worth being aware of it

Where are we?

- This covers the basics of x86-64 and GNU assembler syntax.
  - See the references mentioned earlier for more details.
- Next up: How do we apply all of this to our MiniJava compilers?

Running MiniJava Programs

- To run a MiniJava program
  - Space needs to be allocated for a stack and a heap
  - %rsp and other registers need to have sensible initial values
  - We need some way to allocate storage (new) and communicate with the outside world

Bootstraping from C

- Idea: take advantage of the existing C runtime library to set all of this up, and provide I/O and allocation functionality.
- Use a small C main program to call the MiniJava main method as if it were a C function
- C's standard library provides the execution environment and we can call C functions from compiled code for I/O, malloc, etc.

boot.c

- We will provide a small bootstrap named boot.c with the part 4 assignment.
  - A tiny C program that calls your compiled code as if it were an ordinary C function (assumes your main label is asm_main).
- It also contains some functions that compiled code can call as needed
  - This is a mini “runtime library”
    - Real programming environments often have very sophisticated runtime libraries, to handle things like memory management, system interface, parallelism, certain language features, program startup, etc.
    - We are basically leveraging gcc’s C runtime for most of this, with a tiny MiniJava interface layer on top.
  - Add to this if you like
    - Sometimes simpler to generate a call to a newly written library routine instead of generating in-line code – implementer tradeoff

Bootstrap Program Sketch

```
#include <stdio.h>
extern void asm_main();  /* label for your compiled code */
/* execute compiled program */
void main() { asm_main(); }
/* return next integer from standard input */
long get() { … }
/* write x to standard output */
void put(long x) { … }
/* return a pointer to a block of memory at least nBytes large (or null
   if insufficient memory available) */
char* minijavaalloc(long nBytes) { return malloc(nBytes); }
```

Main Program Label

- Compiler needs special handling for the static main method label
  - Identical label must be declared extern in the C bootstrap program and declared .globl in the .s asm file
  - asm_main used above, and in demo .s file
    - Could be changed, but probably no point
    - Why not "main"? (Hint: what is/where is the *real* main function?)
      - Aside: some environments (e.g, ARM) actually start execution with a runtime library function __main that calls the user main after some low-level initialization. Others (e.g., gcc) insert a call to an initialization function at the beginning of user main.

Interfacing to "Library" code

- Trivial to call "library" functions
- Evaluate parameters using the regular calling conventions
- Make sure %rsp is 16 byte aligned
  - **Danger, Will Robinson!!**
- Generate a call instruction using the function label
  - E.g., "call put"
  - Linker will hook everything up

System.out.println(exp)

- For example, you could implement MiniJava's "print" statement as follows:

```
<compile exp; result in %rax>
movq    %rax,%rdi    ; load argument register
call put             ; call external put routine
```

- If the stack is not kept 16-byte aligned, calls to external C or library code are the most likely place for a runtime error
  - The code you generate is unlikely to assume/care about 16-byte alignment

Note: External Names

- In a Linux environment, an external symbol is used as-is (xyzzy)
- In Windows and OS X, an external symbol xyzzy is written in asm code as _xyzzy (leading underscore)
  - So it would have been "call _put" on the last slide
- Adapt to whatever environment you're using – but what you turn in should run on attu using the Linux conventions

Assembler File Format

- GNU syntax is roughly this (sample code will be provided with part 4 of the project)

```
    .text                    # code segment
    .globl asm_main          # label for main program

    ;; generated code        # repeat .code/.data as needed
asm_main:                    # start of compiled "main"
    ...

    .data
    ;; generated method tables  # repeat .text/.data as needed
    ...
    end
```

Let's Take A Look at the Bootstrap, and a sample .s file

Next Time

- A code generation strategy for your project.
- Some examples of applying the strategy.
- Discussion of how to deal with x86-64's register-based calling convention.