CMSC389R

Binaries I

COMPUTER SCIENCE
UNIVERSITY OF MARYLAND

recap

HW6

--

Questions?

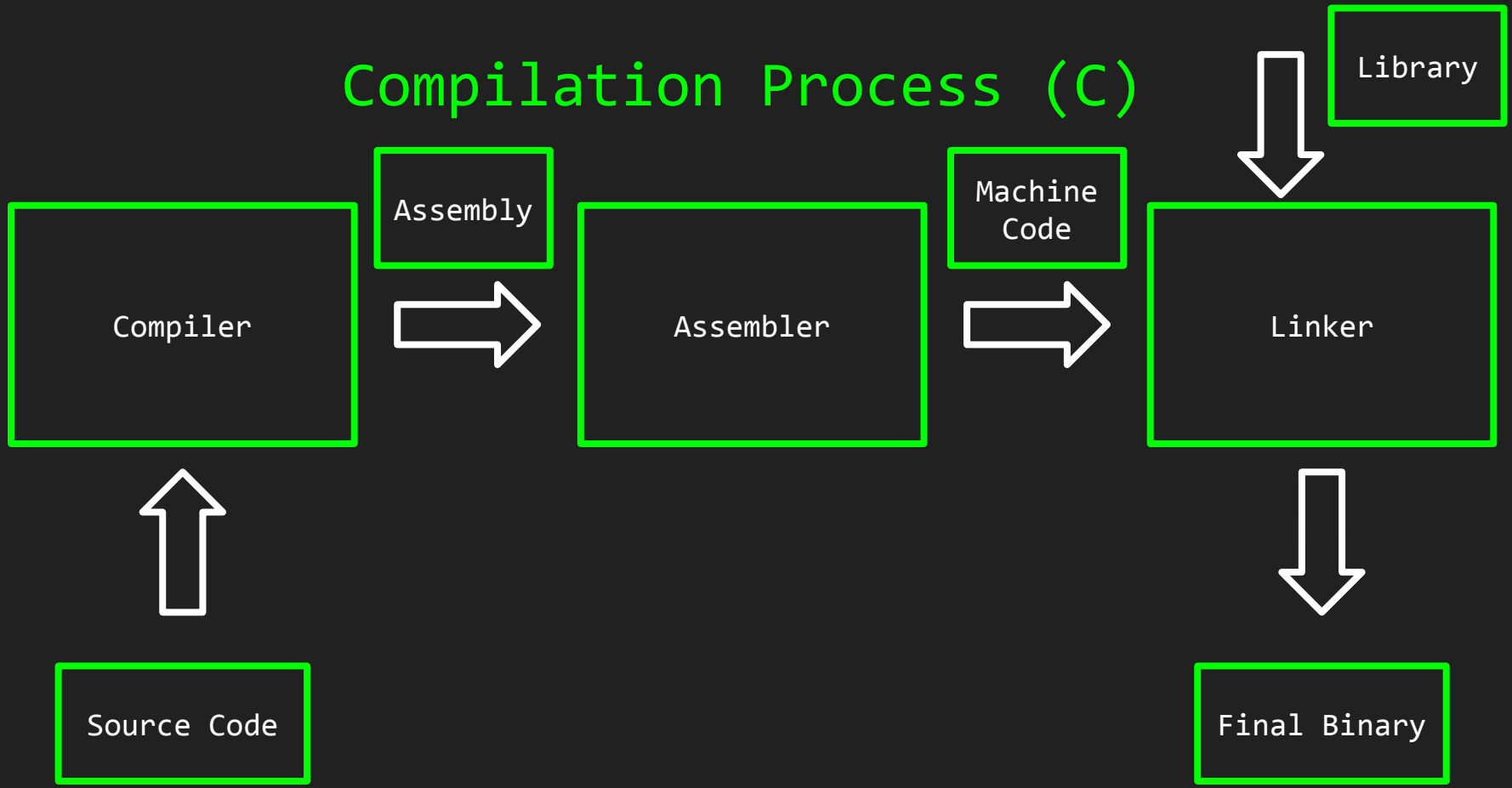
Itinerary

- How programs work
- Compilation process
- x86-64 Assembly
  - Language
  - Conventions
  - Writing/running assembly programs
- gdb review

Computer Programs

- Interpreted
  - Write source code (Python, Ruby, etc)
  - Run in interpreter
- Compiled
  - Write source code (Java, C, etc)
  - Compile (*javac*, *gcc*, *llvm*)
  - Run it

Compilation Process (C)



Instruction Set Architectures

- Complex Instruction Set Computer (CISC)
  - Single instructions are super powerful
  - Variable length instructions
- Reduced Instruction Set Computer (RISC)
  - Smaller set of instructions
  - Few instructions that deal with memory
  - Fixed-size instructions, usually 16/32-bits

Instruction Set Architecture

- Too many CPUs exist... many machine codes too
- x86: Intel CPUs, emulated by AMD
  - Desktop computers, servers
- ARM: IP licensed to companies who implement it
  - Raspberry Pi, Android phones, routers
- MIPS: Prevalent RISC arch we study today
  - Used in routers and old game consoles

Assembly Language

- We’ll be using x86 assembly in 64 bit mode
- Why still learn assembly?
  - Reverse Engineering (here)
  - OS development
  - Compiler writing
  - Computer architecture design
  - Legacy systems/devices

x86

- Registers
- Syntax
- Instructions
- Calling Conventions
- Tooling

Registers

- Original design made heavy use of an accumulator register
  - Many opcodes to do operations on just one register
- 8 “general purpose” registers
  - Some registers have specialized purposes
  - Naming convention is mostly historical
  - A lot more registers as well

General Purpose Registers

rax	“accumulator” register for math operations
rbx	“base” address register for memory calculations
rcx	“counter” register for repeated operations
rdx	auxiliary “data” register for math operations
rsi	“source index” for load/copy instructions
rdi	“destination index” for store/copy instructions
rbp	“base pointer” for stack frames
rsp	“stack pointer” for operations on the stack
r8, r9, ..., r15	new extra 64-bit registers only in x86-64

General Purpose Registers

access bits 63-0	bits 31-0	15-0	7-0
rax	eax	ax	al
rbx	ebx	bx	bl
rcx	ecx	cx	cl
rdx	edx	dx	dl
rsi	esi	si	sil
rdi	edi	di	dil
rbp	ebp	bp	bpl
rsp	esp	sp	spl
63	31	15	7 0

**bits in the range 15-8 can be addressed
with suffix ‘h’: ah, bh, sih, sph, etc**

Syntax

- x86 has two types of assembly syntax
- AT&T
  - Registers are marked with %
  - Immediates (number literals) marked with $
  - Memory addressing syntax uses () and is convoluted
  - Most instructions in format *<instr> <src>, <dst>*
- Intel
  - Registers and immediates don’t have marks
    - hex/binary immediates appended w/ h or b
    - If hex literal begins with abcdef, prepend 0
  - Memory addressing uses [] and is more intuitive
  - Most instructions in format *<instr> <dst>, <src>*

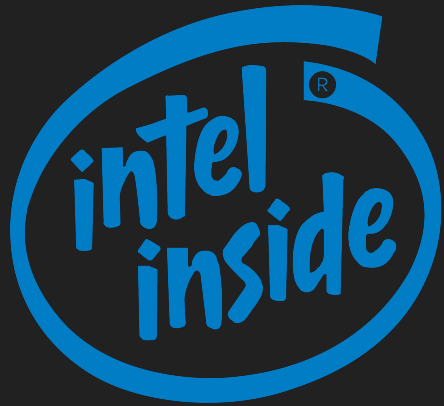
Syntax

Intel	C Equivalent	AT&T
`mov rax, 1`	`rax = 1;`	`movq $0x1, %rax`
`mov qword ptr [rax+rcx*8], 3`	`rax[rcx] = 3;`	`movq $3, (%rax, %rcx, 8)`

Syntax

Intel	C Equivalent	AT&T
`mov rax, 1`	`rax = 1;`	`movq $0x1, %rax`
`mov qword ptr [rax+rcx*8], 3`	`rax[rcx] = 3;`	`movq $3, (%rax, %rcx, 8)`

We'll be using the Intel syntax for this course



Memory Instructions

- mov *dst*, *src* --> dst = src;
- Different ways to express *dst* and *src*
  - *src* can be an “immediate” value
    - mov rax, 123 --> rax = 123;
  - *dst* and *src* can be registers
    - mov rcx, rdx --> rcx = rdx;
  - *dst* or *src* can utilize “register indirection”
    - mov rdi, byte ptr [rsi] or mov byte ptr [rdi], sil
    - rdi = *rsi; or *rdi = sil;
    - [brackets] similar to memory dereference * in C
- Variants of mov: movsx, movzx, movabs, etc

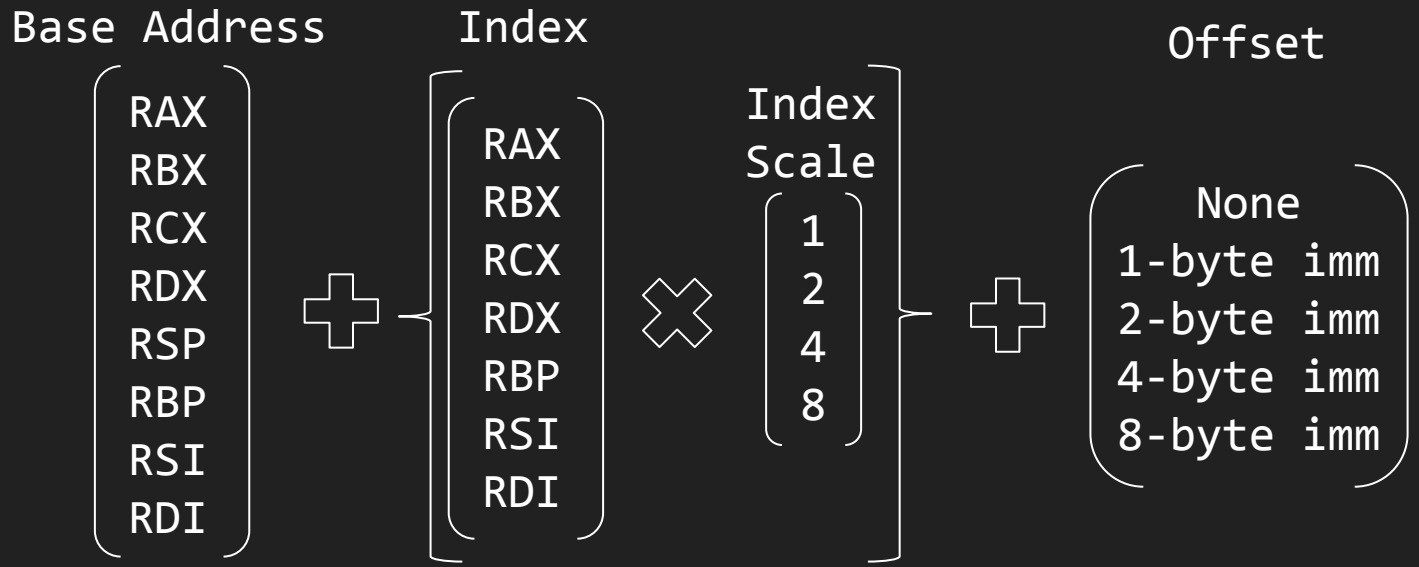
Memory Instructions

- Using mov to/from memory can confuse assembler -- how much memory do we want to actually move?
- Size of memory is either explicit in instruction or implicit by register size

`mov rax, byte [rbx]`	move 1 byte into rax
`mov rax, word [rbx]`	move 2 bytes into rax
`mov rax, dword [rbx]`	move 4 bytes into rax
`mov rax, qword [rbx]`	move 8 bytes into rax

Memory Instructions

- Can compute intricate addresses by using expressions in []



address = base + (index * scale) + offset

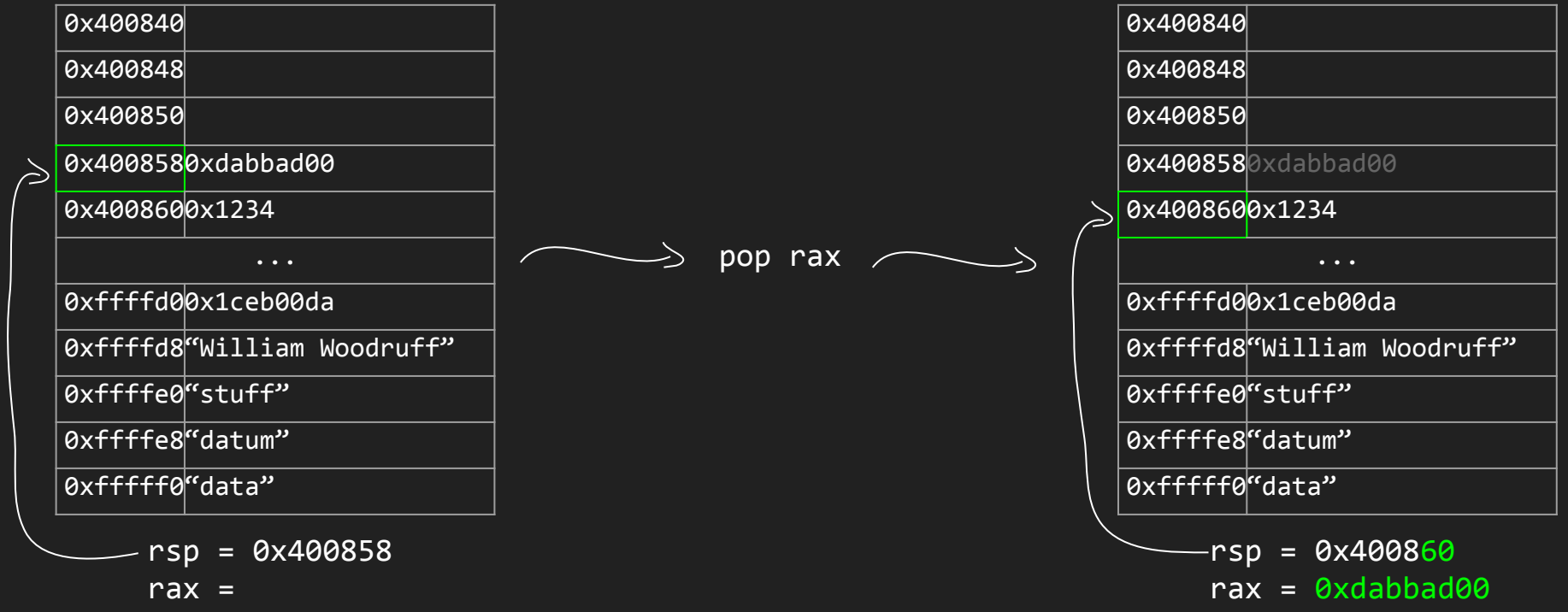
Immediate values

- Immediate values can be represented differently
  - Decimal: *12345*
  - Hex: *0x1234* or *1234h*
    - If using suffix *h*, and the imm leads with A-F, you need to add a “0” as a prefix
    - e.g. *0BEEFh* rather than *BEEFh*
  - Octal: *176o*
  - Binary: *10110110b*

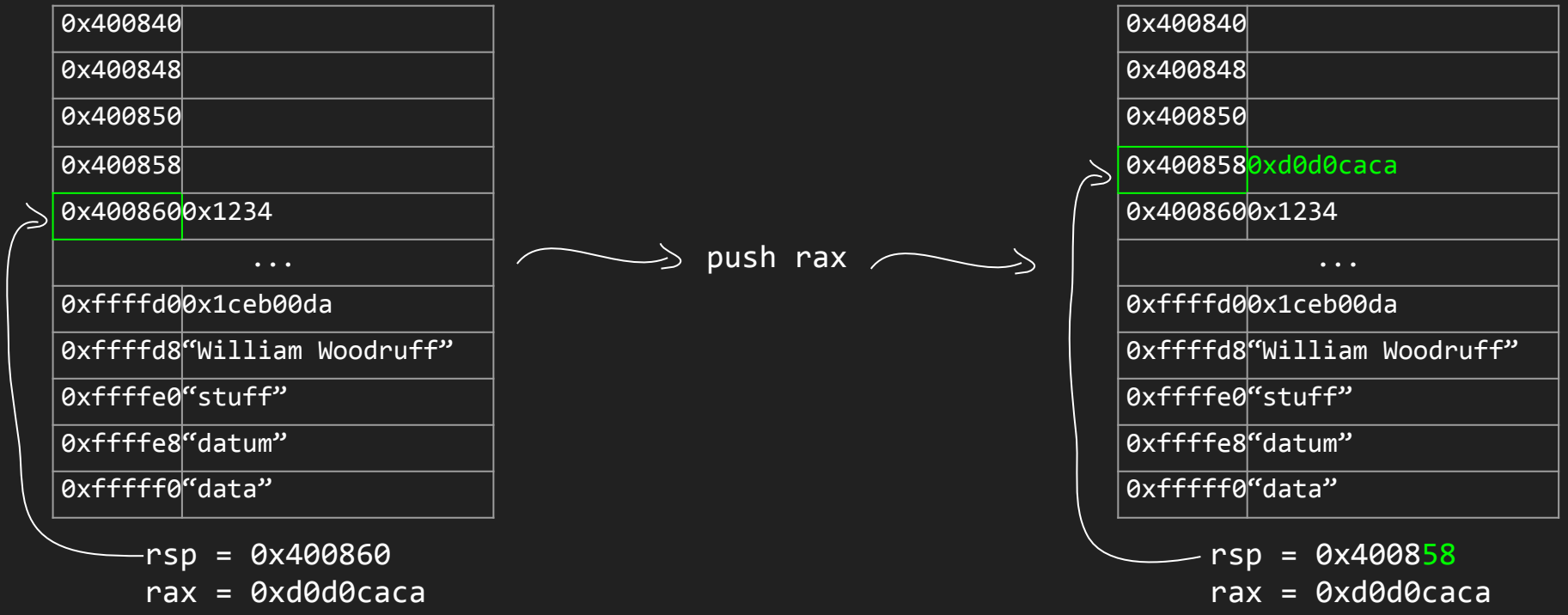
Stack Instructions

- The stack is a way to visualize memory as a first-in, last-out data structure
- Starts at a high address, grows “downward” to lower address
- Stack pointer *rsp* keeps track of the top element of the stack
- *push op*: rsp -= 8, then pushes operand *op* onto the stack
  - *op* can be a register, immediate, or data in memory
- *pop dst*: pops top element off the stack, stores into *dst*, rsp += 8
  - *dst* can be a register or a location in memory

Stack Instructions



Stack Instructions



Math Instructions

- Common forms (with # as the operation)
  - *opcode dst, src*: --> dst #= src;
  - *opcode src*: --> rax #= src;
- *add dst, src*: dst += src;
- *sub dst, src*: dst -= src;
- *imul dst, src*: dst *= src; //truncates to fit in 64-bit
- *or dst, src*: dst |= src;
- *xor dst, src*: dst ^= src;
- *shl dst, src*: dst <<= src;
- etc. etc.

Comparison Instructions

- rflags register
  - Special register that keeps state of last instruction
  - Overflow, zero, carry, signedness, and more

- *cmp reg1, reg2*: does reg1 - reg2 and update rflags
  - Is reg1 > reg2? Are they equal? Etc

- *test reg1, reg2*: does reg1 & reg2 and update rflags
  - Is reg1 = reg2? Is the parity (# of bits) odd/even?

Control Flow Instructions

- Use rflags to decide the execution...
- Jumps
  - *j## func*: sets rip = *func* based on condition code ##
  - JL or JLE: less than (or equal to)
  - JE or JNE: equal or not equal
  - JZ or JNZ: zero or not zero (i.e. is zero flag set?)
  - *Many more...*
- A few other instructions depend on rflags conditions

Control Flow Instructions

- ...or hop around instructions unconditionally
- *call func*
  - Pushes return address (instruction after this *call*)
  - Jumps to address at *func* (either a label or imm)
- *ret*
  - Pops top value on stack into *rip*
  - ***CAREFUL***: make sure return address is at top of stack!

Instruction Modifiers

- *loop func*: sets rip = *func* so long as rcx != 0, decrements rcx

```
looper
 2          mov rax, 1
 1          mov rcx, 5
3
 1  myloop: add rax, rax
 2          loop myloop
 3          ret
~
looper          3,0-1     All
```

What might this snippet here do?

What could go wrong with using the *loop* instruction?

Compiler Explorer

- https://godbolt.org/
- Explore how different compilers convert source code into assembly

Writing x86-64

- We’ll be using the System V x64 ABI
  - Dictates assembly calling conventions, object file formats (.o files), and executable file formats
  - Default ABI for GNU/Linux systems

Writing x86-64

- Most important aspects:
  - Call/return from functions with ***call*** and ***ret***
  - Functions preserve ***rbx***, ***rsp***, ***rbp***, ***r12-15***
  - Functions may clobber ***rax***, ***rdi***, ***rsi***, ***rdx***, ***rcx***, ***r8-11***
  - Parameters are passed into ***rdi***, ***rsi***, ***rdx***, ***rcx***, ***r8***, ***r9***
    - Additional parameters are pushed onto stack
  - Return values are stored in *rax* before calling *ret*

Writing x86-64

- What does this mean?
  - If my assembly function clobbers a preserved register, save it first, then restore it after!
  - *call func, push rbx, ..., pop rbx, ret*

  - If I don’t want my registers to get clobbered, save them first, then restore!
  - *push rax, call func, ..., ret, pop rax*

Register	Usage	Preserved across function calls
%rax	temporary register; with variable arguments passes information about the number of vector registers used; 1st return register	No
%rbx	callee-saved register; optionally used as base pointer	Yes
%rcx	used to pass 4th integer argument to functions	No
%rdx	used to pass 3rd argument to functions; 2nd return register	No
%rsp	stack pointer	Yes
%rbp	callee-saved register; optionally used as frame pointer	Yes
%rsi	used to pass 2nd argument to functions	No
%rdi	used to pass 1st argument to functions	No
%r8	used to pass 5th argument to functions	No
%r9	used to pass 6th argument to functions	No
%r10	temporary register, used for passing a function's static chain pointer	No
%r11	temporary register	No
%r12–r15	callee-saved registers	Yes
%xmm0–%xmm1	used to pass and return floating point arguments	No
%xmm2–%xmm7	used to pass floating point arguments	No
%xmm8–%xmm15	temporary registers	No
%mmx0–%mmx7	temporary registers	No
%st0,%st1	temporary registers; used to return long double arguments	No
%st2–%st7	temporary registers	No
%fs	Reserved for system (as thread specific data register)	No
mxcsr	SSE2 control and status word	partial
x87 SW	x87 status word	No
x87 CW	x87 control word	Yes

If your register is preserved across function calls, then save inside your function and restore before calling *ret*

If your register is NOT preserved across function calls, then save before using *call* and restore after returning

gdb review

- gdb - the GNU debugger
- (*r*)*un*: run the program until a breakpoint or finish
- (*b*)*reak*: set a breakpoint at an address
  - e.g. *b *0x400573*
- (*c*)*ontinue*: continues execution until another breakpoint or finish
- *stepi* or *si*: single step into an instruction
- (*i*)*nfo*: show information
  - e.g. *info reg rax*

gdb review

- *backtrace* or *ba*: show function call stack trace
- x/<format> <addr or reg>: print memory contents
  - e.g. *x/s 0x608010* --> prints like a C string
  - e.g. *x/5bx 0x60802c* --> prints 5 bytes as hex
  - e.g. *x/f 0x608132* --> prints as a float?? lol
- (*q*)*uit*: quits gdb
- *help*: we all need it sometimes
- To save you pain: https://github.com/hugsy/gef

homework #7

will be posted soon.