x86 Assembly Tutorial

COS 318: Fall 2017

Project 1 Schedule

- Design Review: Monday 9/25
  - Sign up for 10-min slot from 3:00pm to 7:00pm
  - Complete set up and answer posted questions
- (Official) Precept: Monday 9/25, 7:30pm
- Due: Sunday, 10/01, 11:55pm

Overview

- Assembly Language Overview
  - Registers, Flags, Memory Addressing, Instructions, Stack / Calling Convention
- BIOS
- Quick kernel debugging tutorial

Registers

General Purpose Register: 8,16,32 bits

31	15	7	0	
	AH	AL	AX = AH | AL	EAX
	BH	BL	BX = BH | BL	EBX
	CH	CL	CX = CH | CL	ECX
	DH	DL	DX = DH | DL	EDX
	BP			EBP
	SI			ESI
	DI			EDI
	SP			ESP

Segment Registers (16bits)

CS
DS
SS
ES
FS
GS

Instruction Pointer: EIP (32bits)
Flags(32bits): EFLAGS

Flags

- Function of flags
  - Control the behavior of CPU
  - Save the status of last instruction
  - Details: https://en.wikipedia.org/wiki/FLAGS_register

Flags

- Important flags:
  - CF: carry flag
  - ZF: zero flag
  - SF: sign flag
  - IF: interrupt (sti, cli)
  - DF: direction (std, cld)

AT&T syntax

- Prefix register names with % (e.g. %ax)
- Instruction format: **instr src,dest**
  - movw %ax,%bx
- Prefix constants (immediate values) with $
  - movw $1,%ax
- Suffix instructions with size of data
  - b for byte,w for word(16bits), l for long(32 bits)

Memory Addressing (Real Mode)

- 1MB memory
  - Valid address range: 0x00000 ~ 0xFFFFF
- See full 1MB with 20-bit addresses
- 16-bit segments and 16-bit offsets

Memory Addressing (Real Mode)

- Format (AT&T syntax):
  - **segment:displacement(base,index)**
- Offset = Base + Index + Displacement
- Address = (Segment * 16) + offset
- Displacement: Constant
- Base: %bx, %bp
- Index: %si, %di
- Segment: %cs, %ds, %ss, %es, %fs, %gs

Memory Addressing (Real Mode)

- **segment:displacement(base,index)**
- Components are optional
  - Default segment: %bp: %ss;   %bx, %si, %di : %ds;
- You can override: %es:(%bx)
- Examples:
  - (%si) = %ds: (%si)   memory address: %ds * 16 + %si
  - +4(%bp) = %ss + 4(%bp) memory address: %ss * 16 + 4 + %bp
  - 100 = %ds:100

Instructions: Arithmetic & Logic

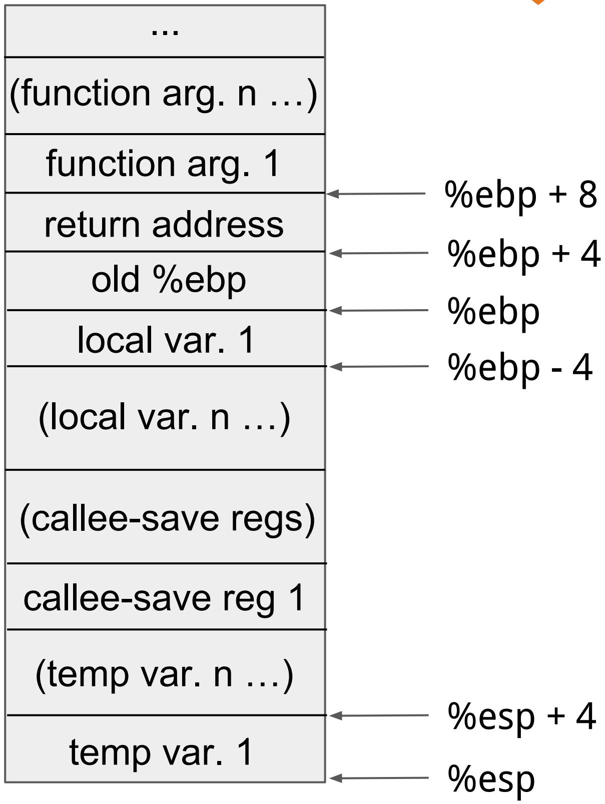
- **add/sub{l,w,b} source,dest**
- **inc/dec/neg{l,w,b} dest**
- **cmp{l,w,b} source,dest**
- **and/or/xor{l,w,b} source,dest ...**
- Restrictions
  - No more than one memory operand

Instructions: Data Transfer

- **mov{l,w,b} source, dest**
- **xchg{l,w,b} dest**
- movsb/movsw
  - %es:(%di) ← %ds:(%si)
  - Often used with %cx to move a number of bytes
    - movw $0x10,%cx
    - rep movsw
- Segment registers can only appear with registers

Stack Layout

- Grows from high to low
  - Lowest address = “top” of stack
- %esp points to top of the stack
  - Used to reference temporary variables
- %ebp points to bottom of stack frame
  - Used for local vars + function args.



...

(function arg. n …)

function arg. 1

%ebp + 8

return address

%ebp + 4

old %ebp

%ebp

local var. 1

%ebp - 4

(local var. n …)

(callee-save regs)

callee-save reg 1

(temp var. n …)

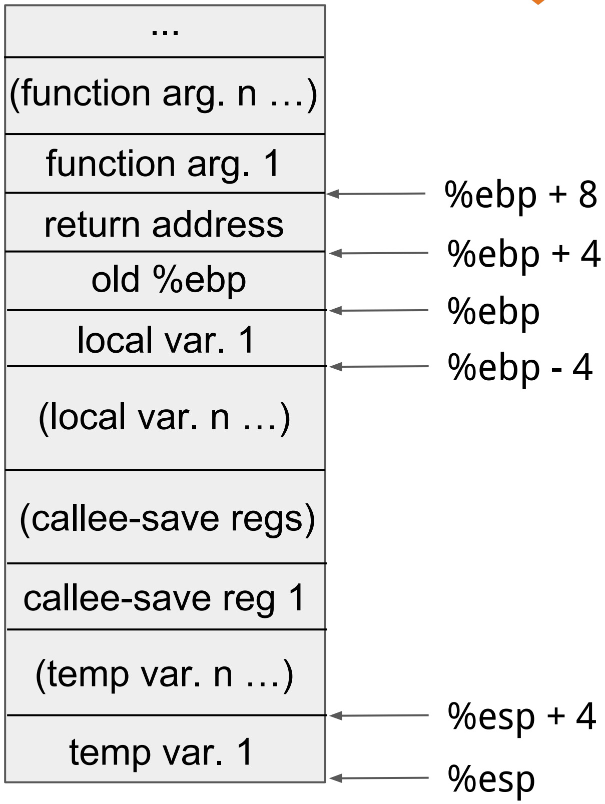
%esp + 4

temp var. 1

%esp

Calling Convention

- When calling a function:
  - 1. Push caller-save regs onto stack
  - 2. Push function args onto stack
  - 3. Push return address + branch
- In subroutine:
  - 1. Push old %ebp + set %ebp = %esp
  - 2. Allocate space for local variables
  - 3. Push callee-save regs if necessary



...

(function arg. n …)

function arg. 1

return address

old %ebp

local var. 1

(local var. n …)

(callee-save regs)

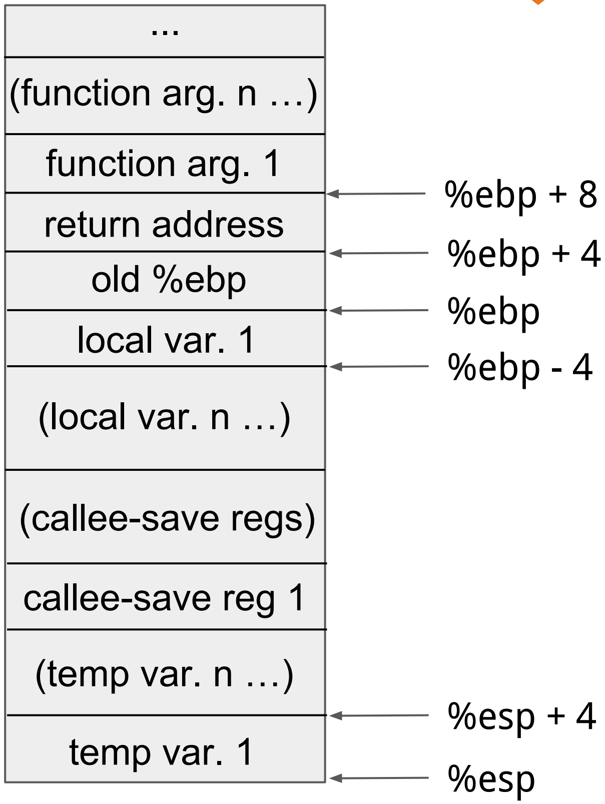
callee-save reg 1

(temp var. n …)

temp var. 1

Instructions: Stack Access

- **pushl source**
  - %esp ← %esp - 4
  - %ss:(%esp) ← source
- **popl dest**
  - dest ← %ss:(%esp)
  - %esp ← %esp + 4



...

(function arg. n ...)

function arg. 1

return address

old %ebp

local var. 1

(local var. n ...)

(callee-save regs)

callee-save reg 1

(temp var. n ...)

temp var. 1

Instructions: Control Flow

- **jmp label**
  - %eip ← label
- **ljmp NEW_CS, offset**
  - %cs ← NEW_CS
  - %eip ← offset

- **call label**
  - push %eip
  - %eip ← label
- **ret**
  - pop %eip

Instructions: Conditional Jump

- Relies on %eflags bits
  - Most arithmetic operations change %eflags
- **j* label**
  - Jump to label if * flag is 1
- **jn* label**
  - Jump to label if * flag is 0

Assembler Directives

- Commands that speak directly to the assembler
  - Are not instructions
- Examples:
  - .globl - defines a list of symbols as global
  - .equ - defines a constant (like #define)
  - .bytes, .word, .asciz - reserve space in memory

Assembler Segments

- Organize memory by data properties
  - .text - holds executable instructions
  - .bss - holds zero-initialized data (e.g. static int i;)
  - .data - holds initialized data (e.g. char c = ‘a’;)
  - .rodata - holds read-only data
- Stack / Heap - Set up by linker / loader / programmer

BIOS Services

- Use BIOS services through int instruction
  - Must store parameters in specified registers
  - Triggers a software interrupt
- **int INT_NUM**
  - int $0x10 - video services
  - int $0x13 - disk services
  - int $0x16 - keyboard services

Kernel testing / debugging

- We provide some test cases with each project
  - Run ‘make TEST=1’ where appropriate
- For debugging: use qemu-gdb
  - Run ‘make qemu-gdb’
  - In another terminal, run ‘gdb’ from same directory

Useful GDB Commands

- r - show register values
- sreg - show segment registers
- s - step into instruction
- n - next instruction
- c - continue
- u <start> <stop> - disassembles C code into assembly

- b - set a breakpoint
- d <n> - delete a breakpoint
- bpd / bpe <n> - disable / enable a breakpoint
- x/Nx addr - display hex dump of N words, starting at addr
- x/Ni addr - display N instructions, starting at addr

Design Review

- Be ready to answer the following questions:
  - At what point does the processor start executing 32-bit code? What exactly causes the switch from 16 to 32-bit mode?
  - What is the last instruction of the boot loader executed, and what is the first instruction of the kernel it loads?
  - Where is the first instruction of the kernel?
  - How does the boot loader decide how many sectors it must read in order to fetch the entire kernel from disk? Where does it get this information?