x86 Assembly Tutorial

COS 318: Fall 2020
Project 1 Schedule

Mon, 9/7
Tue, 9/8
Fri, 9/11

x86 Assembly Tutorial
Deadline to register teams or sign-up for random team assignment

Design Review Sessions
(15 min per team)

Precept 1
(Design details and Q&A)

Use Piazza and OHs.

Project 1 Due
Fri, 9/11
Mon, 9/14
Tue, 9/15
3-7pm ET
7:30-8:20 pm ET

Deadline to register teams or sign-up for random team assignment

Mon, 9/7
Tue, 9/8
Fri, 9/11

Mon, 9/14
Tue, 9/15

Use Piazza and OHs.

Precept 1
(Design details and Q&A)

Fri, 9/11
Mon, 9/14
Tue, 9/15

Sun, 9/20, 11:55 pm ET

Project 1: Bootloader

Option 1: if picking own teammates
Option 2: if prefer random assignment
Monday 9/7 & Tuesday 9/8, 7:30pm - 8:20pm
Monday 9/14 & Tuesday 9/15, 3:00pm - 7:00pm
Monday 9/14 & Tuesday 9/15, 7:30pm - 8:20pm
Sunday 9/20, 11:55pm

Team Registration Form
Random Team Assignment Request Form
x86 Assembly Tutorial
Design Review
Precept 1
Project 1 due

The link will open on 9/12 or 9/13.
Overview

● x86 (*technically* IA-32) Assembly Overview
  ○ Registers, Flags, Memory Addressing, Instructions, Stack, Calling Conventions, Directives, Segments

● BIOS (Basic Input/Output System) + GDB (GNU Debugger)

● Design Review
# Registers

General Purpose Registers: 8, 16, 32 bits

<table>
<thead>
<tr>
<th>31</th>
<th>15</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>AH</td>
<td>AL</td>
<td>AX = AH</td>
<td>AL</td>
</tr>
<tr>
<td>BH</td>
<td>BL</td>
<td>BX = BH</td>
<td>BL</td>
</tr>
<tr>
<td>CH</td>
<td>CL</td>
<td>CX = CH</td>
<td>CL</td>
</tr>
<tr>
<td>DH</td>
<td>DL</td>
<td>DX = DH</td>
<td>DL</td>
</tr>
<tr>
<td>BP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SI</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DI</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SP</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Segment Registers: 16 bits (hold 16-bit segment selectors to identify memory segment)

|    |    |    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|----|
| EAX | EBX | ECX | EDX | EBP | ESI |EDI |ESP |
| CS  | DS  | SS  | ES  | FS  | GS  |

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

- code segment
- data segment
- stack segment
- data segment
- data segment
- data segment
Flags

The 32-bit EFLAGS register:

Flags types:
- Status
- Control
- System
● Important Flags
  ○ CF: Carry flag
  ○ ZF: Zero flag
  ○ SF: Sign flag
  ○ IF: Interrupt flag (sti, cli)
  ○ DF: Direction flag (std, cld)
<table>
<thead>
<tr>
<th><strong>Instruction Syntax Conventions</strong></th>
<th><strong>Gnu Syntax (AT&amp;T)</strong></th>
<th><strong>Intel</strong></th>
</tr>
</thead>
</table>
| Immediate operands              | Preceded by "$"  
  e.g.: push $4  
  movl $0xd00a, %eax              | Undelimited  
  e.g.: push 4  
  mov ebx, d00ah                                                                                     |
| Register operands               | Preceded by "%"  
  e.g.: %eax                | Undelimited  
  e.g.: eax                                                                                         |
| Argument order (e.g. adds the address of C variable "foo" to register EAX) | source1, [source2,] dest  
  e.g.: addl $ _foo, %eax | dest, source1 [, source2]  
  e.g.: add eax, _foo                                                                                   |
| Single-size operands            | Explicit with operand sizes  
  opcode{b,w,l}  
  e.g.: movb foo, %al | Implicit with register name, **byte ptr**  
  **word ptr**, or **dword ptr**  
  e.g.: mov al, foo                                                                                   |
| Address a C variable "foo"      | _foo | [ _foo]                          |
| Address memory pointed by a register (e.g. EAX) | (%eax) | [eax]                              |
| Address a variable offset by a value in the register | _foo(%eax) | [eax + _foo]                       |
| Address a value in an array "foo" of 32-bit integers | _foo(,%eax,4) | [eax*4+foo]                        |
| Equivalent to C code *(p+1)      | 1(%eax) | If EAX holds the value of p, then  
  [eax+1]                                                                                           |

source: [http://flint.cs.yale.edu/cs422/doc/pc-arch.html](http://flint.cs.yale.edu/cs422/doc/pc-arch.html)
Memory Addressing

- Memory addressing modes:
  1. Real address (unprotected)  2. Protected  3. System Management

- Real address mode:
  - 1MB of memory (20-bit addresses)
  - Valid address range: 0x00000 ~ 0xFFFFF
  - 16-bit segment add. [times 16, i.e., +4 bits] + 16-bit offset add.
Memory Addressing (Real Mode)

- Format (AT&T syntax):
  - segment:displacement(base,index,scale)

- Offset = Base + Index * Scale + Displacement

- Address = (Segment * 16) + Offset

- Displacement: Constant

- Base: %bx, %bp

- Index: %si, %di

- Segment: %cs, %ds, %ss, %es, %fs, %gs
## Data Types

<table>
<thead>
<tr>
<th>Name</th>
<th>Size (bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>8</td>
</tr>
<tr>
<td>word</td>
<td>16</td>
</tr>
<tr>
<td>double-word</td>
<td>32</td>
</tr>
<tr>
<td>(long in gnu assembler)</td>
<td></td>
</tr>
<tr>
<td>quad-word</td>
<td>64</td>
</tr>
</tbody>
</table>
Instructions: Arithmetic & Logic

- Arithmetic, such as:
  - `add/sub{l,w,b} source,dest`
  - `inc/dec/neg{l,w,b} dest`
  - `cmp{l,w,b} source,dest`

- Logic, such as:
  - `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} source, dest` (exchange)
- `movsb/movsw` (move byte/word)
  - `%es:(%di) ← %ds:(%si)`
  - Often used with `%cx` to move a number of bytes
    - `movw $0x10,%cx`
    - `rep movsw` (repeat)
Stack Layout

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

- **When calling a function:**
  1. Push caller-save regs onto stack
  2. Push function args on to stack
  3. Push return address + branch

- **In subroutine:**
  1. Push old %ebp + set %bp = %sp
  2. Allocate space for local variables
  3. Push callee-save regs if necessary

<table>
<thead>
<tr>
<th></th>
<th>%bp - 4</th>
<th>%bp</th>
<th>%bp + 4</th>
<th>%bp + 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(function arg. n …)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>function arg. 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>return address</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>old %ebp</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>local var. 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(local var. n …)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(callee-save regs)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>callee-save reg 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(temp var. n …)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>temp var. 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%sp + 4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%sp</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Instructions: Stack Access

● pushl source
  ○ %sp ← %sp - 4
  ○ %ss:(%sp) ← source

● popl dest
  ○ dest ← %ss:(%sp)
  ○ %sp ← %sp + 4
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{e,ne,l,le,g,ge}
  - Jump to label if {=,!=,<,<=,>,>=}
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

[Oracle Documentation](https://docs.oracle.com/cd/E26502_01/html/E28388/eoiyg.html)
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
Basic Input/Output System (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

- sending a character to the display at the current cursor position
  - `ah = 0x0e, indicating this is function 0x0e`
  - `al = holding the character to write`
  - `bh = active page number (Use 0x00)`
  - `bl = foreground color (graphics mode only) (Use 0x02)`
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

- Write print_char and print_string assembly functions
- Be ready to describe:
  - How to move the kernel from disk to memory
  - How to create disk image
  - (More specific guidelines are provided on the project page)