# Section 12
# x86 Basics

CS 107A, Autumn 2021
Andrew Benson (adbenson@)


https://twitter.com/programmersmeme/status/1108078617983156226

# Don’t forget to start recording

# Unix Tip Spotlight

- gdb conditional breakpoints
  - Breakpoints, but only if some condition about the code is true
  - `break myfile.c:46 if i == 35`

# Announcements

- Midterm???
- Midquarter Feedback Survey
  - https://forms.gle/B6zwdF77aWa7LyubA
  - Also on course homepage
- Section Feedback next Thursday
- More 1:1s soon

# You've come pretty far

- UNIX
- Using the terminal
- C integer types and 2's complement
- Bitwise operations
- Strings in C
- The C string library
- Pointers in C
- Weird C quirks with arrays
- Stack vs Heap vs data segment
- `malloc` / `free` / `calloc` / `realloc`
- `gdb` / `valgrind` / `make`
- Generic pointers
- Function Pointers

# There’s not much left to cover

- x86 Assembly
  - Data movement
  - Control flow
  - Stack discipline
- Internals of heap management
- Compiler optimizations
- Profiling with `callgrind`

# There's not much left to cover...but there's projects

- x86 Assembly
  - Data movement
  - Control flow
  - Stack discipline
- Internals of heap management
- Compiler optimizations
- Profiling with `callgrind`
- assign4
- Binary Bomb
- Heap Allocator

# Bird’s Eye View

| Day | Week 6 Wednesday | Thursday | Friday | Week 7 Monday | Tuesday | Wednesday | Thursday | Friday |
|---|---|---|---|---|---|---|---|---|
| CS 107A | | Section: x86 Basics | | | No Class - Democracy Day | | Section: x86 Control Flow | |
| CS 107 | Lab 5: Assembly | | Lecture: x86-64 Control Flow | Lecture: x86 Runtime Stack | | Lab: Runtime Stack | | No Lecture |
| CS 107 assignments | | | | | | assign4 due, assign5 released | | |

# Agenda

- C Compilation Process
- Inspecting x86 Assembly
- x86 Primitives
- Operand Forms
- `mov and lea`

# C Compilation Process

# C Compilation

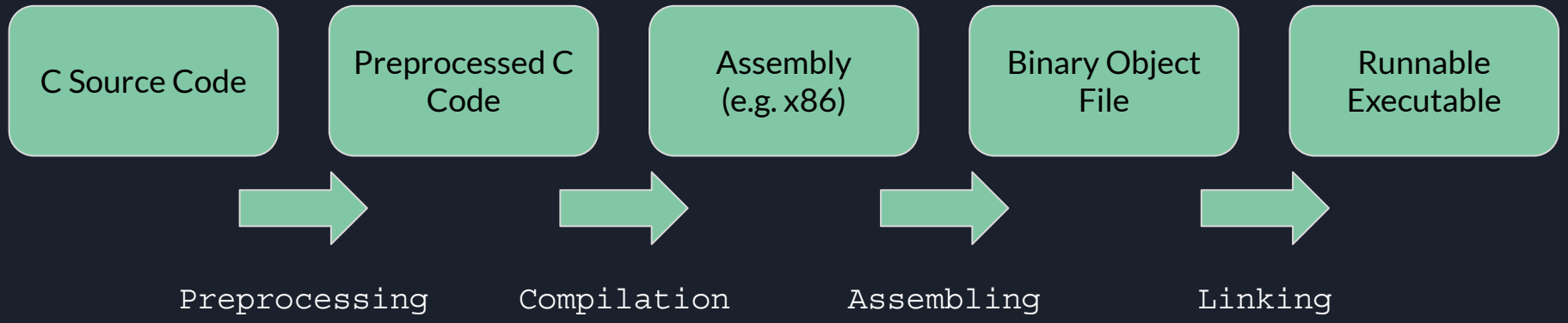
```
#define MAGIC 7

int x = 5;
x += MAGIC;
```

```
int x = 5;
x += 7;
```

```
movl $5, %rax
addl $7, %rax
```

```
010101010100101
101000101001010
010110101001101
        +
010101010100101
101000101001010
010110101001101
```

```
./myuniq
```



# Assembly

- Lots of different kinds, just like there's lot of programming languages
- Tied to hardware
- x86 (Intel, AMD): Most laptops, most desktops, most servers, some game consoles
    - Specifically x86-64, the 64-bit version
- ARM (Qualcomm, Apple, Samsung, NVIDIA): iPhones, most Androids, the new MacBooks, Internet of Things, Nintendo Switch
- RISC-V (SiFive): the new hotness, maybe?

# Inspecting x86 Assembly

# Two Methods

- 1) `objdump -d <executable>`
- 2) `gdb <executable>`
  - `layout asm`

# x86 Example

```
int main()
{
  int n = 0;
  for (int i = 0; i < 5; i++) {
    n += i;
  }
  return n;
}
```

```
00000000004004d6 <main>:
  4004d6:    55                       push %rbp
  4004d7:    48 89 e5                 mov  %rsp,%rbp
  4004da:    c7 45 f8 00 00 00 00     movl $0x0,-0x8(%rbp)
  4004e1:    c7 45 fc 00 00 00 00     movl $0x0,-0x4(%rbp)
  4004e8:    eb 0a                    jmp  4004f4
<main+0x1e>
  4004ea:    8b 45 fc                 mov  -0x4(%rbp),%eax
  4004ed:    01 45 f8                 add  %eax,-0x8(%rbp)
  4004f0:    83 45 fc 01              addl $0x1,-0x4(%rbp)
  4004f4:    83 7d fc 04              cmpl $0x4,-0x4(%rbp)
  4004f8:    7e f0                    jle  4004ea
<main+0x14>
  4004fa:    8b 45 f8                 mov  -0x8(%rbp),%eax
  4004fd:    5d                       pop  %rbp
  4004fe:    c3                       retq
  4004ff:    90                       nop
```

# x86 Primitives

# Instructions and Arguments

```
int main()
{
  int n = 0;
  for (int i = 0; i < 5; i++) {
    n += i;
  }
  return n;
}
```

```
00000000004004d6 <main>:
  4004d6:    55                       push %rbp
  4004d7:    48 89 e5                 mov  %rsp,%rbp
  4004da:    c7 45 f8 00 00 00 00     movl $0x0,-0x8(%rbp)
  4004e1:    c7 45 fc 00 00 00 00     movl $0x0,-0x4(%rbp)
  4004e8:    eb 0a                    jmp  4004f4
<main+0x1e>
  4004ea:    8b 45 fc                 mov  -0x4(%rbp),%eax
  4004ed:    01 45 f8                 add  %eax,-0x8(%rbp)
  4004f0:    83 45 fc 01              addl $0x1,-0x4(%rbp)
  4004f4:    83 7d fc 04              cmpl $0x4,-0x4(%rbp)
  4004f8:    7e f0                    jle  4004ea
<main+0x14>
  4004fa:    8b 45 f8                 mov  -0x8(%rbp),%eax
  4004fd:    5d                       pop  %rbp
  4004fe:    c3                       retq
  4004ff:    90                       nop
```

2 arguments

1 argument

0 arguments

# Instruction Syntax

```
[INSTRUCTION][optional SUFFIX] [OPERANDS]

ex) addl $0x1,-0x4(%rbp)

ex) add  %eax,-0x8(%rbp)
```

# Instructions

- Use the CS 107 x86 Reference.

# Optional Instruction Suffixes

| Name | Suffix | Width |
|---|---|---|
| byte | `b` | `1 byte` |
| word | `w` | `2 bytes` |
| double word | `l` | `4 bytes` |
| quad word | `q` | `8 bytes` |

- `movb`
- `movw`
- `movl`
- `movq`
- `addl`
- `subq`
- `shlw`
- `(If there's no suffix, one is inferred)`

# Operands: 3 Options

- Immediates
  - Literally hardcoded hex numbers, prefixed with a `$`
    - `$0x3`
    - `$0x1cfd`
- Registers
  - Prefixed with a `%`
- Memory Addresses in “Operand Form”

# General-Purpose Registers

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 64-bit | RAX | RBX | RCX | RDX | RSI | RDI | RBP | RSP | R8 | R9 | R10 | R11 | R12 | R13 | R14 | R15 |
| 32-bit | EAX | EBX | ECX | EDX | ESI | EDI | EBP | ESP | R8D | R9D | R10D | R11 | R12D | R13D | R14D | R15D |
| 16-bit | AX | BX | CX | DX | SI | DI | BP | SP | R8W | R9W | R10W | R11W | R12W | R13W | R14W | R15W |
| 8-bit | AL | ZBL | CL | DL | SIL | DIL | BPL | SPL | R8B | R9B | R10B | R11B | R12B | R13B | R14B | R15B |

# Some Registers with Special Meanings (memorize these)

| | |
|---|---|
| %rax | Return value of a function |
| %rdi | First argument to a function |
| %rsi | Second argument to a function |
| %rdx | Third argument to a function |
| %rip | Address of next instruction to execute |
| %rsp | Address of the top of the current stack |

# Operand Forms

# Operand Forms

- General form:
  - `b(reg1, reg2, a)` (where `a` is `1`, `2`, `4`, or `8`)
  - Calculated as: `reg1 + a * reg2 + b`
  - Convenient way to do a calculation without explicitly doing `iadd`, `imul`, etc (faster since supported by hardware)
  - Also often used to calculate a memory address within an array or struct
  - Some parts can be omitted – assume its value in the formula above is 0 (or 1 for a) – but you have to keep the parentheses, or else it's just a register, not an operand form
  - Example: `17(%rdx, %rcx, 4)`

# Calculate the value of the operand form expression

- For reference: `b(reg1, reg2, a) => reg1 + a * reg2 + b`
- Assume the following:
  - `%rax` contains `0x10`
  - `%rcx` contains `0x2`

1) `(%rax)`

2) `4(%rax)`

3) `(%rax, %rcx)`

4) `4(%rax, %rcx)`

# Calculate the value of the operand form expression

- For reference: `b(reg1, reg2, a) => reg1 + a * reg2 + b`
- Assume the following:
  - `%rax` contains `0x10`
  - `%rcx` contains `0x2`

5) `(, %rcx, 8)`

6) `(%rax, %rcx, 2)`

7) `0x0(%rax, %rcx, 2)`

8) `5(%rax, %rcx)`

# mov and lea

# lea

- ALWAYS of the form:
  - `lea <operand form> <register>`

    which are source and destination respectively
- Calculates the value of the source operand form, then sticks it into the destination register
- No “dereferencing” occurs!
- Example: `lea 5(%rax, %rcx), %rbx`
  - Calculates the value `%rax + %rcx + 5`
  - Puts it into `%rbx`

# mov

- Has many variants (move immediate to register, register to register, immediate to operand form, etc)
- In particular, sometimes of the form:
  - `mov <operand form> <register>`

    which are source and destination respectively

- Calculates the value of the source operand form, dereferences it to read a value, then sticks it into the destination register
  - So that operand form better calculate a memory address! (lea doesn't have to)
- Example: `mov 5(%rax, %rcx), %rbx`
  - Calculates the value `%rax + %rcx + 5`
  - Dereferences this value and reads the appropriate number of bytes (here, 8, since this is implied `movq`)
  - Puts it into `%rbx`

# lea / mov exercises

- `mov $1, %rbx`
- `mov %rsp, %rbx`
- `movb $6, %al`
- `movs $6, %ax`
- `movl $6, %eax`
- `movq $6, %rax`
- `movsbl %al, %ebx`
- `movzbl %al, $ebx`

# `lea` / `mov` exercises

- `lea (%rax, %rcx), %rax`
- `mov (%rax, %rcx), %rax`
- `lea (, %rcx, 8), %rax`
- `mov (, %rcx, 8), %rax`

# Section 12 Extra Code Exercises

```
git clone /afs/ir/class/cs107a/WWW/git/section12
```