Lecture 8 Notes

Problem Set 1 and Announcements

- Discussion of progress on pset1--cumulative distribution graph
- pset is harder than anticipated--so 3 bonus late days (now 6 total for the semester)
- "wild applause"
- emails will be sent out w/info on git repos/submitting tonight
- How compare.pl works
- no need to denote final commit--but if you turn in late, let staff know.
- Test 26:
  - uses memcpy to copy allocated regions and metadata (1)
  - and uses memcpy to do the reverse copy (2)
  - and tries what should be an invalid free (3)
  - solution: check if pointers to b from adjacent links are valid

![Diagram of Test 26](image)

x86 Assembly code: how code is represented in memory

- focus on reading assembly code (nobody needs to do a ton of programming in assembly, but it’s good to be able to understand it.)
- x86 originally designed in the 1970s with 16-bit address spaces
- Alpha: extremely clean computer architecture--with 64 bit instructions
- size of instruction in x86: anywhere from 1 byte to 15 bytes
- example programs--Eddie has compiled them with gcc -o1 -s
  - -o1 means slightly optimized, -s means to output assembly code
- Assembly contains data and metadata
  - data: instructions and constants
  - metadata: debugging info and other metadata; indicated by a line that starts with a period

Example f00

```assembly
rep
ret
```

- f is a label (NOT CODE)
• rep and ret are instructions
• rep not necessary—see textbook
• ret means return from function
• f is a void function (can’t see any return values) that takes no arguments

Example f01

```
movl $0, %eax
ret
```

• this puts the value 0 in the eax register
• so 0 is returned, since eax register contains the return value

What is a register?
• the CPU can only directly interact with registers, not the rest of memory
• compiler maps variables onto registers
• x86 does arithmetic on registers, then moves results back into memory

What is a register?

```
xorl %eax, %eax
ret
```

• x86 general registers:
• %eax, %ebx, %ecx, %edx, %esi, %edi, %ebp, %esp, %eip
• %eip stores the address of executing instruction
• CPU is like a Turing machine, with %eip storing the current state
• naming of registers: x is for extended...and e is for extra extended (to eventually be 32 bits)

Example f02

General form of x86 instructions
• operator source, destination
• $ indicates a constant
• each instruction has an operator, a source, and a combined source/destination
in C, the program just consists of return 0;
but the xorl command is shorter than a movl to move 0 to %eax

BREAK

• note that any function that just returns zero can have return types of int, char, unsigned char, etc. (or if it returns NULL, char *, int *, etc.), but still corresponds to same assembly code.
• so the return type is lost—the processor doesn’t know about types, just knows how to treat numbers in different ways.
• so we lose performance info when moving back from assembly to C

Example f03

movl    a, %eax
subl    b, %eax
ret

• subtracts b from a
• here, a and b are globals
• can’t be heap addresses, since those are not known until runtime
• and can’t be locals, since functions could be called in multiple locations on the stack

Variables:
• $ indicates constant (ex. $0)
• names (ex. a, b) are references to globals
• and % indicates registers (e.g. %eax)

Example f04

movl    a, %eax
addl    b, %eax
ret

• adds b to a
• (unsigned and signed ints don’t make a difference with addition, so the assembly instructions are exactly the same)
• no such thing as declaring variables in assembly (except for globals)

Example f05

movl    x, %eax
movl    (%eax), %eax
ret
- indirect addressing
- in assembly, dereference an address by using ( ) → so (%eax) dereferences %eax

```c
extern int x;
int f(void) {
    return *x;
}
```

**Example f06**

```assembly
movl     x, %eax
movl     (%eax), %eax
ret
```

- the C source uses x as an array and returns x[0]...which is the same address as x
- So the assembly is the same, because the address is the same.

```c
extern int x[];
int f(void) {
    return x[0];
}
```

**Example f07**

```assembly
movzbl   x, %eax
ret
```

- in movzbl, z = zero, b = bytes
- l generally means 32 bits
- so movzbl means: move a byte with zero extension into 32-bit ("l") register; last byte of register is that byte, all others are zero
- (zero extension means to put zeroes into the remaining unused bytes of the register)
- so the C source contains an unsigned char x, casted to an int and returned
- if we make it a signed char, the assembly has movsbl instead of movzbl:

```assembly
movsbl   x, %eax
ret
```

- this means to move the byte and preserve sign by extending other bytes appropriately.

**Example f10**

```assembly
movl     x, %eax
movzbl   (%eax), %eax
ret
```
- returns an unsigned char x, dereferenced.

```c
extern unsigned char *x
int f() {
    return *x;
}
```

- if you use short instead of unsigned char, need movzwl
- where w=word (16-bit)
- also q = quad word (64-bit)

---

**Example f11**

```assembly
movl x , %eax
movzbl 1(%eax), %eax
ret
```

- 1(%eax) is the dereference of %eax + 1 \(\rightarrow \) *(%eax + 1)
- so it corresponds to returning x[1]

```c
extern unsigned char *x
int f() {
    return x[1];
}
```

**Example f12**

```assembly
movl 4(%esp), %eax
ret
```

- %esp is on the stack (s indicates stack)
- it's the identity function
- that's all folks

```c
unsigned f(unsigned i) {
    return i;
}
```
<table>
<thead>
<tr>
<th><strong>Instruction</strong></th>
<th><strong>What it does</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>general pattern:</td>
<td></td>
</tr>
<tr>
<td>OP SRC DST</td>
<td></td>
</tr>
<tr>
<td>SRC ← SRC OP DST</td>
<td></td>
</tr>
<tr>
<td>(like += operators in C)</td>
<td></td>
</tr>
<tr>
<td>$[stuff]$</td>
<td>stuff is a constant value</td>
</tr>
<tr>
<td>%[stuff]</td>
<td>stuff is an address</td>
</tr>
<tr>
<td>stuff</td>
<td>stuff is a reference to a global variable</td>
</tr>
<tr>
<td>(%[stuff])</td>
<td>dereference (%[stuff]) to get *stuff</td>
</tr>
<tr>
<td>N(%[stuff])</td>
<td>dereference (%[stuff] + N) = *(%[stuff] +N), where N = 1,2,3...</td>
</tr>
<tr>
<td>rep</td>
<td>not really necessary, spaceholder that makes function have &gt;1 byte</td>
</tr>
<tr>
<td>ret</td>
<td>return from function</td>
</tr>
<tr>
<td>movl [A], [B]</td>
<td>moves [A] into [B]</td>
</tr>
<tr>
<td>xorl</td>
<td>XOR</td>
</tr>
<tr>
<td>subl [A], [B]</td>
<td>[B] - [A]</td>
</tr>
<tr>
<td>addl [A], [B]</td>
<td>[A]+[B]</td>
</tr>
<tr>
<td>movzbl [A], [B]</td>
<td>z=0, b= byte, l = 32bits move byte w/ zero extension into a 32bit register</td>
</tr>
<tr>
<td></td>
<td>%eax</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>movsbl [A], [B]</td>
<td>like movzbl, except s = sign extension, so will fill other bits with top byte of [A]</td>
</tr>
<tr>
<td>movzw [A],[B]</td>
<td>w = 2 bytes/16 bits</td>
</tr>
<tr>
<td>Symbol/Register</td>
<td>What it does</td>
</tr>
<tr>
<td>-----------------</td>
<td>--------------</td>
</tr>
<tr>
<td><code>%eax</code></td>
<td>register that holds return value of function</td>
</tr>
</tbody>
</table>
| `%ebx, ecx, edx, esi, edi, ebp, esp` | general purpose registers commonly seen in code  
x - symbolizes extended register  
e - 32bit version of register  
s - stack |
| `%eip`          | address of currently executing instruction (the current state of the “turing machine” x86 processor) |