Today’s Topics

Lecture:

› More assembly code!
  • Arithmetic and logic operations
  • If-else control
  • Loops (to be continued Friday)
### Basic addressing modes

(Think: assembly version of VARIABLES)

<table>
<thead>
<tr>
<th>Op</th>
<th>Source</th>
<th>Dest</th>
<th>Dest Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>movl</td>
<td>$0, %eax</td>
<td>Name of a register</td>
<td></td>
</tr>
<tr>
<td>movl</td>
<td>$0, 0x8f2713e0</td>
<td>Actual address literal (note address literals are different from other literals—don’t need $ in front)</td>
<td></td>
</tr>
<tr>
<td>movl</td>
<td>$0, (%rax)</td>
<td>Look in the register named, find an address there, and use it</td>
<td></td>
</tr>
</tbody>
</table>

Reminder: need to put $ in front of immediate values (constant literals)
### Basic addressing modes

(Think: assembly version of VARIABLES)

<table>
<thead>
<tr>
<th>Op</th>
<th>Source</th>
<th>Dest</th>
<th>Dest Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>movl</td>
<td>$0, %eax</td>
<td>%eax</td>
<td>Name of a register</td>
</tr>
<tr>
<td>movl</td>
<td>$0, 0x8f2713e0</td>
<td>0x8f2713e0</td>
<td>Actual address literal (note address literals are different from other literals—don’t need $ in front)</td>
</tr>
<tr>
<td>movl</td>
<td>$0, (%rax)</td>
<td>(%rax)</td>
<td>Look in the register named, find an address there, and use it</td>
</tr>
<tr>
<td>movl</td>
<td>$0, -24(%rbp)</td>
<td>-24(%rbp)</td>
<td>Add -24 to an address in the named register, and use that address</td>
</tr>
</tbody>
</table>

**Displacement must be a constant; to have a variable base and variable displacement, use two steps: add then mov.**

**Displacement can be positive or negative.**
## Basic addressing modes

(Think: assembly version of VARIABLES)

<table>
<thead>
<tr>
<th>Op</th>
<th>Source</th>
<th>Dest</th>
<th>Dest Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>movl</td>
<td>$0, %eax</td>
<td>moving $0</td>
<td>Name of a register</td>
</tr>
<tr>
<td>movl</td>
<td>$0, 0x8f2713e0</td>
<td>copying</td>
<td>Actual address literal (note address literals are different from other literals—don’t need $ in front)</td>
</tr>
<tr>
<td>movl</td>
<td>$0, (%rax)</td>
<td>memory</td>
<td>Look in the register named, find an address there, and use it</td>
</tr>
<tr>
<td>movl</td>
<td>$0, -24(%rbp)</td>
<td>memory</td>
<td>Add -24 to an address in the named register, and use that address</td>
</tr>
<tr>
<td>movl</td>
<td>$0, 8(%rbp, %eax, 2)</td>
<td>memory</td>
<td>Address to use = (8 + address in rbp) + (2 * index in eax)</td>
</tr>
</tbody>
</table>

- Any constant allowed
- Only 1, 2, 4, 8 allowed
## Basic addressing modes

(Think: assembly version of VARIABLES)

<table>
<thead>
<tr>
<th>Op</th>
<th>Source</th>
<th>Dest</th>
<th>Dest Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>movl</td>
<td>$0,</td>
<td>%eax</td>
<td>Name of a register</td>
</tr>
<tr>
<td>movl</td>
<td>$0,</td>
<td>0x8f2713e0</td>
<td>Actual address literal (note address literals are different from other literals—don’t need $ in front)</td>
</tr>
<tr>
<td>movl</td>
<td>$0,</td>
<td>(%rax)</td>
<td>Look in the register named, find an address there, and use it</td>
</tr>
<tr>
<td>movl</td>
<td>$0,</td>
<td>-24(%rbp)</td>
<td>Add -24 to an address in the named register, and use that address</td>
</tr>
<tr>
<td>movl</td>
<td>$0</td>
<td>8(%rbp, %eax, 2)</td>
<td>Address to use = (8 + address in rbp) + (2 * index in eax)</td>
</tr>
</tbody>
</table>
Matching exercise: Addressing modes use cases

- Match up which use cases make the most sense for which addressing modes (some guesswork expected)

<table>
<thead>
<tr>
<th>Op</th>
<th>Src</th>
<th>Dest</th>
<th>Use case?</th>
</tr>
</thead>
<tbody>
<tr>
<td>movl</td>
<td>$0</td>
<td>8(%rbp, %eax, 2)</td>
<td></td>
</tr>
<tr>
<td>movl</td>
<td>$0,</td>
<td>%eax</td>
<td></td>
</tr>
<tr>
<td>movl</td>
<td>$0,</td>
<td>0x8f2713e0</td>
<td></td>
</tr>
<tr>
<td>movl</td>
<td>$0,</td>
<td>4(%rax)</td>
<td></td>
</tr>
</tbody>
</table>

Use cases

(a) Prepare to use 0 in a calculation
(b) Zero out a field of a struct
(c) Zero out a given array index
(d) Zero out a global variable
Arithmetic and bitwise operations
Arithmetic and bitwise operations template

- op1 and op2 can be any of the addressing modes we’ve seen
  - though, again, at most one of op1 and op2 can touch memory
  - note we can get a sneaky 2-in-1 memory access if we use a memory location for op2, because it is a source and destination

<table>
<thead>
<tr>
<th>Op</th>
<th>Source1</th>
<th>Source2/Dest</th>
<th>Dest Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>op1</td>
<td>op2</td>
<td>op2 += op1</td>
</tr>
</tbody>
</table>

Stanford University
Arithmetic and bitwise operations

add src, dst  # dst += src
sub src, dst  # dst -= src
imul src, dst # dst *= src
inc dst       # dst += 1
dec dst       # dst -= 1
neg dst       # dst = -dst

and src, dst  # dst &= src
or src, dst   # dst |= src
xor src, dst  # dst ^= src
not dst       # dst = ~dst

shl count, dst # dst <<= count (left shift)
sar count, dst # dst >>= count (arithmetic right shift)
shr count, dst # dst >>= count (logical right shift)
Reference material: How to view assembly on myth

Handy tips
**Objdump: makes readable assembly from executable**

- Gives you output like below, for all functions in executable
- Doesn’t execute or debug code, just provides this output for you to read

```
000000000040055d <sum_array>:
  40055d:   ba 00 00 00 00 00     mov   $0x0,%edx
  400562:   b8 00 00 00 00 00     mov   $0x0,%eax
  400567:   eb 09                 jmp   400572
  400569:   48 63 ca              movslq %edx,%rcx
  40056c:   03 04 8f             add   (%rdi,%rcx,4),%eax
  40056f:   83 c2 01             add   $0x1,%edx
  400572:   39 f2               cmp   %esi,%edx
  400574:   7c f3               jl    400569
  400576:   f3 c3               repz retq
```

```
myth5> objdump -d sum         // replace "sum" with your executable
```

```
```
Gdb: debug in C or assembly *(or both at the same time!)*

- In addition to all its other wonderful features, gdb lets you see assembly version of code you are debugging
  - You can step through code either line-by-line in C code, or instruction-by-instruction in assembly code
  - ...Or see both at the same time!

```
myth5> gdb sum       // replace "sum" with your executable
Reading symbols from sum...done.
(gdb) break main
(gdb) run
(gdb) layout split
```

```
int main(int argc, char *argv[])

0x400578 <main>       sub    $0x28,%rsp
0x40057c <main+4>     movl    $0x58,(%rsp)
0x400583 <main+11>    "Sum movl    $0x5e,0x4(%rsp)res, nscores));
0x40058b <main+19>    movl    $0x46,0x8(%rsp)
0x400593 <main+27>    movl    $0x5c,0xc(%rsp)
0x40059b <main+35>    movl    $0x53,0x10(%rsp)bly Available ]
0x4005a3 <main+43>    movl    $0x5c,0x14(%rsp)
0x4005ab <main+51>    movl    $0x1,0x18(%rsp)
0x4005b3 <main+59>    movl    $0x5c,0x1c(%rsp)
0x4005bb <main+67>    mov    $0x8,%esi
0x4005c0 <main+72>    mov    %rsp,%rdi
```

Child process 22274 In: main
Examples: Let’s read code!

Preparation for your binary bomb assignment
Preparing for binary bomb: reading assembly code

00000000004005ac <sum_example1>:
   4005bd: 8b 45 e8 mov %esi,%eax
   4005c3: 01 d0 add %edi,%eax
   4005cc: c3 retq

Which of these C functions do you think is most likely to have generated the above assembly?

// (A)
void sum_example1() {
    int x, y;
    int sum = x + y;
}

// (B)
void sum_example1(int x, int y) {
    int sum = x + y;
}

// (C)
int sum_example1() {
    int x, y;
    return x + y;
}

// (D)
int sum_example1(int x, int y) {
    return x + y;
}
Preparing for binary bomb: reading assembly code

```
000000000004005ac <sum_example1>:
4005bd: 8b 45 e8
        mov    %esi,%eax
4005c3: 01 d0                   add    %edi,%eax
4005cc: c3
        retq
```

Which of these C functions do you think is most likely to have generated the above assembly?

// (A)  
void sum_example1() {
    int x, y;
    int sum = x + y;
}

// (B)  
void sum_example1(int x, int y) {
    int sum = x + y;
}

// (C)  
int sum_example1() {
    int x, y;
    return x + y;
}

// (D)  
int sum_example1(int x, int y) {
    return x + y;
}

To the untrained eye, it looks like we add two uninitialized variables together and then return from the function without returning a value. Because we never write to %esi/%edi before using them, and the return instruction takes no argument.

That would suggest choice (A). But…

Because this mov copies %esi to %eax, the add is essentially adding %esi and %edi
Register special uses
(includes a few of the most common—for more complete list see reference on course website)

<table>
<thead>
<tr>
<th>Register</th>
<th>Conventional use</th>
<th>Low 32-bits</th>
<th>Low 16-bits</th>
<th>Low 8-bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>Return value</td>
<td>%eax</td>
<td>%ax</td>
<td>%al</td>
</tr>
<tr>
<td>%rdi</td>
<td>1st argument</td>
<td>%edi</td>
<td>%di</td>
<td>%dil</td>
</tr>
<tr>
<td>%rsi</td>
<td>2nd argument</td>
<td>%esi</td>
<td>%si</td>
<td>%sil</td>
</tr>
<tr>
<td>%rdx</td>
<td>3rd argument</td>
<td>%edx</td>
<td>%dx</td>
<td>%dl</td>
</tr>
<tr>
<td>%r10</td>
<td>Scratch/temporary</td>
<td>%r10d</td>
<td>%r10w</td>
<td>%r10b</td>
</tr>
<tr>
<td>%r11</td>
<td>Scratch/temporary</td>
<td>%r11d</td>
<td>%r11w</td>
<td>%r11b</td>
</tr>
<tr>
<td>%rip</td>
<td>Instruction pointer</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%rflags</td>
<td>Status/condition code bits</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Preparing for binary bomb: reading assembly code

000000000004005ac <sum_example1>:

4005bd: 8b 45 e8
mov %esi,%eax

4005c3: 01 d0                   add    %edi,%eax

4005cc: c3 retq

Which of these C functions do you think is most likely to have generated the above assembly?

// (A)
void sum_example1() {
    int x, y;
    int sum = x + y;
}

// (B)
void sum_example1(int x, int y) {
    int sum = x + y;
}

// (C)
int sum_example1() {
    int x, y;
    return x + y;
}

// (D)
int sum_example1(int x, int y) {
    return x + y;
}

Now that we know that the first and second parameters of a function are stored in %esi and %edi, we know that the sum is adding two input parameters.

And now that we know that a function’s return value is stored in %eax, we know that the function returns the sum.

So the answer is (D).

Because this mov copies %esi to %eax, the add is essentially adding %esi and %edi.

Stanford University
More assembly code reading examples

These are meant to help you with the binary bomb assignment by giving examples of how to step-by-step bite off little pieces of understanding of a passage of assembly code.
Preparing for binary bomb: reading assembly code

00000000000400578 <sum_example2>:

400578:       8b 47 0c
            mov 0xc(%rdi),%eax

40057b:       03 07                   add (%rdi),%eax

40057d:       2b 47 18                sub 0x18(%rdi),%eax

400580:       c3
            retq

int sum_example2(int arr[]) {
    int sum = 0;
    sum += arr[0];
    sum += arr[3];
    sum -= arr[6];
    return sum;
}

Which of these represents the C code’s sum variable?

(a) 0xc(%rdi)
(b) %rdi
(c) (%rdi)
(d) 0x18(%rdi)
(e) %eax
Preparing for binary bomb: reading assembly code

00000000000400578 <sum_example2>:

400578:  8b 47 0c                   mov 0xc(%rdi),%eax
40057b:  03 07                   add (%rdi),%eax
40057d:  2b 47 18                sub 0x18(%rdi),%eax
400580:  c3                   retq

int sum_example2(int arr[]) {
  int sum = 0;
  sum += arr[0];
  sum += arr[3];
  sum -= arr[6];
  return sum;
}

Which of these represents the C code’s 6 (as in arr[6])?

(a) 0xc
(b) %rdi
(c) (%rdi)
(d) 0x18
(e) %eax
What does it mean for a program to execute?

How do we move from one instruction to the next? How do computers “do stuff”?
Data storage vs. “doing stuff” on a computer

- Data sits in memory (and, we now know, registers)
- We understand that there are instructions that control movement of the data and operations on it
- ...
- But **who controls the instructions**? How do we know what to do now? …or do next?
Progress through a function

00000000004004ed <loop>:

4004ed: 55 push %rbp
4004ee: 48 89 e5 mov %rsp,%rbp
4004f1: c7 45 fc 00 00 00 00 movl $0x0,-0x4(%rbp)
4004f8: 83 45 fc 01 addl $0x1,-0x4(%rbp)
4004fc: eb fa jmp 4004f8 <loop+0xb>
Progress through a function

00000000004004ed <loop>:

4004ed: 55 push %rbp
4004ee: 48 89 e5 mov %rsp,%rbp
4004f1: c7 45 fc 00 00 00 00 movl $0x0,-0x4(%rbp)
4004f8: 83 45 fc 01 addl $0x1,-0x4(%rbp)
4004fc: eb fa jmp 4004f8 <loop+0xb>
Progress through a function

4004ed: 55  
push %rbp

4004ee: 48 89 e5  
mov %rsp,%rbp

4004f1: c7 45 fc 00 00 00 00  
movl $0x0,-0x4(%rbp)

4004f8: 83 45 fc 01  
addl $0x1,-0x4(%rbp)

4004fc: eb fa  
jmp 4004f8 <loop+0xb>
Progress through a function

000000000004004ed <loop>:

4004ed:  55 push %rbp
4004ee:  48 89 e5 mov %rsp,%rbp
4004f1:  c7 45 fc 00 00 00 00 movl $0x0,-0x4(%rbp)
4004f8:  83 45 fc 01 addl $0x1,-0x4(%rbp)
4004fc:  eb fa jmp 4004f8 <loop+0xb>
Progress through a function

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4004ed:</td>
<td>55</td>
<td>push %rbp</td>
</tr>
<tr>
<td>4004ee:</td>
<td>48 89 e5</td>
<td>mov %rsp,%rbp</td>
</tr>
<tr>
<td>4004f1:</td>
<td>c7 45 fc 00 00 00 00</td>
<td>movl $0x0,-0x4(%rbp)</td>
</tr>
<tr>
<td>4004f8:</td>
<td>83 45 fc 01</td>
<td>addl $0x1,-0x4(%rbp)</td>
</tr>
<tr>
<td>4004fc:</td>
<td>eb fa</td>
<td>jmp 4004f8 &lt;loop+0xb&gt;</td>
</tr>
</tbody>
</table>
Progress through a function

000000000004004ed <loop>:

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4004ed:</td>
<td>55</td>
<td>push</td>
</tr>
<tr>
<td>4004ee:</td>
<td>48 89 e5</td>
<td>mov</td>
</tr>
<tr>
<td>4004f1:</td>
<td>c7 45 fc 00 00 00 00</td>
<td>movl</td>
</tr>
<tr>
<td>4004f8:</td>
<td>83 45 fc 01</td>
<td>addl</td>
</tr>
<tr>
<td>4004fc:</td>
<td>eb fa</td>
<td>jmp</td>
</tr>
</tbody>
</table>
Progress through a function

0000000000004004ed <loop>:

4004ed: 55 push
4004ee: 48 89 e5 mov
4004f1: c7 45 fc 00 00 00 00 movl
4004f8: 83 45 fc 01 addl
4004fc: eb fa jmp
Progress through a function

0000000000004004ed <loop>:

4004ed:      55                push
4004ee:     48 89 e5           mov
4004f1:   c7 45 fc 00 00 00 00 movl
4004f8:    83 45 fc 01          addl
4004fc:    eb fa                jmp
Progress through a function

0000000000004004ed <loop>:

4004ed:      55      push
4004ee:     48 89 e5    mov
4004f1:    c7 45 fc 00 00 00 00    movl
4004f8:    83 45 fc 01    addl
4004fc:    eb fa    jmp
Progress through a function

00000000004004ed <loop>:

- 4004ed: 55  push
- 4004ee: 48 89 e5  mov
- 4004f1: c7 45 fc 00 00 00 00  movl
- 4004f8: 83 45 fc 01  addl
- 4004fc: eb fa  jmp

This special register has a name:

%rip

Special hardware is responsible for setting its value to the next instruction

- doing %rip += size in bytes of current instruction
“Interfering” with `%rip

If `%rip always advances to next instruction, how do we “skip” instructions in an if-else, or repeat instructions in a loop?
The jmp instruction

000000000004004ed <loop>:

4004ed: push %rbp
4004ee: mov %rsp,%rbp
4004f1: movl $0x0,-0x4(%rbp)
4004f8: addl $0x1,-0x4(%rbp)
4004fc: jmp 4004f8 <loop+0xb>
The jmp instruction

000000000004004ed <loop>:

4004ed: push %rbp
4004ee: mov %rsp,%rbp
4004f1: movl $0x0,-0x4(%rbp)
4004f8: addl $0x1,-0x4(%rbp)
4004fc: jmp 4004f8 <loop+0xb>
The jmp instruction

40000000004004ed <loop>:
    4004ed: push %rbp
    4004ee: mov %rsp,%rbp
    4004f1: movl $0x0,-0x4(%rbp)
    4004f8: addl $0x1,-0x4(%rbp)
    4004fc: jmp 4004f8 <loop+0xb>
The jmp instruction

4004ed:

jmp

4004f1:
mov

4004ee:
push %rbp

4004fc:
addl $0x1, -0x4(%rbp)

4004f8 <loop+0xb>

4004f9

4004fa

4004fb

4004fc

4004fd

55

48

89
c7

45

fc

00

00

00

83

45

fc

01

eb

fa
The jmp instruction

000000000004004ed <loop>:
  4004ed: push %rbp
  4004ee: mov %rsp,%rbp
  4004f1: movl $0x0,-0x4(%rbp)
  4004f8: addl $0x1,-0x4(%rbp)
  4004fc: jmp 4004f8 <loop+0xb>

%rip
The jmp instruction

000000000004004ed <loop>:

4004ed: push %rbp
4004ee: mov %rsp,%rbp
4004f1: movl $0x0,-0x4(%rbp)
4004f8: addl $0x1,-0x4(%rbp)
4004fc: jmp 4004f8 <loop+0xb>
The jmp instruction

0000000000004004ed <loop>:
  4004ed:  push  %rbp
  4004ee:  mov   %rsp,%rbp
  4004f1:  movl  $0x0,-0x4(%rbp)
  4004f8:  addl  $0x1,-0x4(%rbp)
  4004fc:  jmp   4004f8 <loop+0xb>
The jmp instruction

00000000004004ed <loop>:

4004ed: push %rbp
4004ee: mov %rsp,%rbp
4004f1: movl $0x0,-0x4(%rbp)
4004f8: addl $0x1,-0x4(%rbp)
4004fc: jmp 4004f8 <loop+0xb>

%rip

4004fd fa
4004fc eb
4004fb 01
4004fa fc
4004f9 45
4004f8 83
4004f7 00
4004f6 00
4004f5 00
4004f4 00
4004f3 fc
4004f2 45
4004f1 c7
e5
4004ef 89
4004ee 48
4004ed 55
void loop()
{
    int i = 0;
    again:
        ++i;
    goto again;
}