CS 107
Lecture 20:
Assembly Part IV

Monday, February 26, 2024

Computer Systems
Winter 2024
Stanford University
Computer Science Department

Reading: Course Reader: x86-64 Assembly Language, Textbook: Chapter 3.1-3.4

Lecturer: Chris Gregg
Today's Topics

- Reading: Chapter 3.6.7, 3.7
- Programs from class: /afs/ir/class/cs107/samples/lect20
- Comments on assembly writing
- More x86 Assembly Language
  - Procedures
    - The run-time stack
    - Control transfer (call/return)
    - Data transfer (arguments)
    - Local storage on the stack
    - Local storage in registers
    - Recursion
  - Arrays
  - Structures
  - Alignment
  - Function pointers
### The Run-Time Stack: Example

#### Disassembly of leaf(long y), y in %rdi:

```
00000000000400540 <leaf>:
L1  400540:  48 8d 47 02    lea 0x2(%rdi),%rax  // y+2
L2  400544:  c3            retq  // return
```

#### Disassembly of top(long x), x in %rdi:

```
00000000000400540 <top>:
T1  400545:  48 83 ef 05    sub $0x5,%rdi  // x-5
T2  400549:  e8 f2 ff ff ff  callq 400540 <leaf>  // Call leaf(x-5)
T3  40054e:  48 01 c0        add %rax,%rax  // Double result
T4  400551:  c3            retq  // Return
```

... Call to top from function main

```
M1  40055b:  e8 e5 ff ff ff  callq 400545, <top>  // Call top(100)
M2  400560:  48 89 c2        mov %rax,%rdx  // Resume
```

#### State values (at beginning)

<table>
<thead>
<tr>
<th>Label</th>
<th>PC</th>
<th>Instruction</th>
<th>%rdi</th>
<th>%rax</th>
<th>%rsp</th>
<th>*%rsp</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>M1</td>
<td>0x40055b</td>
<td>callq</td>
<td>100</td>
<td>–</td>
<td>0x820</td>
<td>–</td>
<td>Call top(100)</td>
</tr>
<tr>
<td>T1</td>
<td>0x400545</td>
<td>sub</td>
<td>100</td>
<td>–</td>
<td>0x818</td>
<td>0x400560</td>
<td>Entry of top</td>
</tr>
<tr>
<td>T2</td>
<td>0x400549</td>
<td>callq</td>
<td>95</td>
<td>–</td>
<td>0x818</td>
<td>0x400560</td>
<td>Call leaf(95)</td>
</tr>
<tr>
<td>L1</td>
<td>0x400540</td>
<td>lea</td>
<td>95</td>
<td>–</td>
<td>0x810</td>
<td>0x40054e</td>
<td>Entry of leaf</td>
</tr>
<tr>
<td>L2</td>
<td>0x400544</td>
<td>retq</td>
<td>–</td>
<td>97</td>
<td>0x810</td>
<td>0x40054e</td>
<td>Return 97 from leaf</td>
</tr>
<tr>
<td>T3</td>
<td>0x40054e</td>
<td>add</td>
<td>–</td>
<td>97</td>
<td>0x818</td>
<td>0x400560</td>
<td>Resume top</td>
</tr>
<tr>
<td>T4</td>
<td>0x400551</td>
<td>retq</td>
<td>–</td>
<td>194</td>
<td>0x818</td>
<td>0x400560</td>
<td>Return 194 from top</td>
</tr>
<tr>
<td>M2</td>
<td>0x400560</td>
<td>mov</td>
<td>–</td>
<td>194</td>
<td>0x820</td>
<td>–</td>
<td>Resume main</td>
</tr>
</tbody>
</table>
### The Run-Time Stack: Example

**Disassembly of leaf(long y), y in %rdi:**
```
00000000000400540 <leaf>:
L1  400540: 48 8d 47 02     lea 0x2(%rdi),%rax  // y+2
L2  400544: c3              retq  // return
```

**Disassembly of top(long x), x in %rdi:**
```
00000000000400540 <top>:
T1  400545: 48 83 ef 05     sub $0x5,%rdi      // x-5
T2  400549: e8 f2 ff ff ff  callq 400540 <leaf>  // Call leaf(x-5)
T3  40054e: 48 01 c0        add %rax,%rax        // Double result
T4  400551: c3              retq  // Return
```

---

```
... Call to top from function main

M1  40055b: e8 e5 ff ff ff  callq 400545, <top>  // Call top(100)
M2  400560: 48 89 c2        mov %rax,%rdx        // Resume
```

---

**State values (at beginning):**

<table>
<thead>
<tr>
<th>Label</th>
<th>PC</th>
<th>Instruction</th>
<th>%rdi</th>
<th>%rax</th>
<th>%rsp</th>
<th>*%rsp</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>M1</td>
<td>0x40055b</td>
<td>callq</td>
<td>100</td>
<td>—</td>
<td>0x7fffffffe820</td>
<td>—</td>
<td>Call top(100)</td>
</tr>
<tr>
<td>T1</td>
<td>0x400545</td>
<td>sub</td>
<td>100</td>
<td>—</td>
<td>0x7fffffffe818</td>
<td>0x400560</td>
<td>Entry of top</td>
</tr>
<tr>
<td>T2</td>
<td>0x400549</td>
<td>callq</td>
<td>95</td>
<td>—</td>
<td>0x7fffffffe818</td>
<td>0x400560</td>
<td>Call leaf(95)</td>
</tr>
<tr>
<td>L1</td>
<td>0x400540</td>
<td>lea</td>
<td>95</td>
<td>—</td>
<td>0x7fffffffe810</td>
<td>0x40054e</td>
<td>Entry of leaf</td>
</tr>
<tr>
<td>L2</td>
<td>0x400544</td>
<td>retq</td>
<td>—</td>
<td>97</td>
<td>0x7fffffffe810</td>
<td>0x40054e</td>
<td>Return 97 from leaf</td>
</tr>
<tr>
<td>T3</td>
<td>0x40054e</td>
<td>add</td>
<td>—</td>
<td>97</td>
<td>0x7fffffffe818</td>
<td>0x400560</td>
<td>Resume top</td>
</tr>
<tr>
<td>T4</td>
<td>0x400551</td>
<td>retq</td>
<td>—</td>
<td>194</td>
<td>0x7fffffffe818</td>
<td>0x400560</td>
<td>Return 194 from top</td>
</tr>
<tr>
<td>M2</td>
<td>0x400560</td>
<td>mov</td>
<td>—</td>
<td>194</td>
<td>0x7fffffffe820</td>
<td>—</td>
<td>Resume main</td>
</tr>
</tbody>
</table>
The Run-Time Stack: Example

Disassembly of leaf(long y), y in %rdi:
00000000000400540 <leaf>:
L1 000540: 48 8d 47 02  lea 0x2(%rdi),%rax // y+2
L2 000544: c3     retq // return

Disassembly of top(long x), x in %rdi:
00000000000400540 <top>:
T1 000545: 48 83 ef 05  sub $0x5,%rdi // x-5
T2 000549: e8 f2 ff ff ff  callq 400540 <leaf> // Call leaf(x-5)
T3 00054e: 48 01 c0  add %rax,%rax // Double result
T4 000551: c3     retq // Return

... Call to top from function main

M1 00055b: e8 e5 ff ff ff  callq 400545, <top> // Call top(100)
M2 000560: 48 89 c2  mov %rax,%rdx // Resume

Instruction | State values (at beginning)
--- | ---
Label | PC | Instruction | %rdi | %rax | %rsp | *%rsp | Description
M1 | 0x40055b | callq | 100 | – | 0x7fffffff820 | – | Call top(100)
T1 | 0x400545 | sub | 100 | – | 0x7fffffff818 | 0x400560 | Entry of top
T2 | 0x400549 | callq | 95 | – | 0x7fffffff818 | 0x400560 | Cal leaf(95)
L1 | 0x400540 | lea | 95 | – | 0x7fffffff810 | 0x40054e | Entry of leaf
L2 | 0x400544 | retq | – | 97 | 0x7fffffff810 | 0x40054e | Return 97 from leaf
T3 | 0x40054e | add | – | 97 | 0x7fffffff818 | 0x400560 | Resume top
T4 | 0x400551 | retq | – | 194 | 0x7fffffff818 | 0x400560 | Return 194 from top
M2 | 0x400560 | mov | – | 194 | 0x7fffffff820 | – | Resume main

Stack "bottom"

| 0x7fffffff820 |
| 0x7fffffff818 |
| 0x400560 |

Stack "top"
The Run-Time Stack: Example

Disassembly of leaf(long y), y in %rdi:
0000000000400540 <leaf>:
L1 400540: 48 8d 47 02     lea 0x2(%rdi),%rax   // y+2
L2 400544: c3              retq   // return

Disassembly of top(long x), x in %rdi:
0000000000400540 <top>:
T1 400545: 48 83 ef 05     sub $0x5,%rdi        // x-5
T2 400549: e8 f2 ff ff ff  callq 400540 <leaf>  // Call leaf(x-5)
T3 40054e: 48 01 c0        add %rax,%rax        // Double result
T4 400551: c3              retq   // Return

... Call to top from function main

M1 40055b: e8 e5 ff ff ff  callq 400545, <top>  // Call top(100)
M2 400560: 48 89 c2        mov %rax,%rdx        // Resume

<table>
<thead>
<tr>
<th>Label</th>
<th>PC</th>
<th>Instruction</th>
<th>%rdi</th>
<th>%rax</th>
<th>%rsp</th>
<th>*%rsp</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>M1</td>
<td>0x40055b</td>
<td>callq</td>
<td>100</td>
<td>–</td>
<td>0x7fffffffe820</td>
<td>–</td>
<td>Call top(100)</td>
</tr>
<tr>
<td>T1</td>
<td>0x400545</td>
<td>sub</td>
<td>100</td>
<td>–</td>
<td>0x7fffffffe818</td>
<td>0x400560</td>
<td>Entry of top</td>
</tr>
<tr>
<td>T2</td>
<td>0x400549</td>
<td>callq</td>
<td>95</td>
<td>–</td>
<td>0x7fffffffe818</td>
<td>0x400560</td>
<td>Cal leaf(95)</td>
</tr>
<tr>
<td>L1</td>
<td>0x400540</td>
<td>lea</td>
<td>95</td>
<td>–</td>
<td>0x7fffffffe810</td>
<td>0x40054e</td>
<td>Entry of leaf</td>
</tr>
<tr>
<td>L2</td>
<td>0x400544</td>
<td>retq</td>
<td>–</td>
<td>97</td>
<td>0x7fffffffe810</td>
<td>0x40054e</td>
<td>Return 97 from leaf</td>
</tr>
<tr>
<td>T3</td>
<td>0x400545</td>
<td>retq</td>
<td>–</td>
<td>97</td>
<td>0x7fffffffe818</td>
<td>0x400560</td>
<td>Resume top</td>
</tr>
<tr>
<td>T4</td>
<td>0x400551</td>
<td>retq</td>
<td>–</td>
<td>194</td>
<td>0x7fffffffe818</td>
<td>0x400560</td>
<td>Return 194 from top</td>
</tr>
<tr>
<td>M2</td>
<td>0x400560</td>
<td>mov</td>
<td>–</td>
<td>194</td>
<td>0x7fffffffe820</td>
<td>–</td>
<td>Resume main</td>
</tr>
</tbody>
</table>

Stack "bottom"
0x7fffffffe820
Stack "top"
0x7fffffffe818
0x400560
The Run-Time Stack: Example

Disassembly of leaf(long y), y in %rdi:

L1  400540: 48 8d 47 02  lea 0x2(%rdi),%rax  // y+2
L2  400544: c3              retq                 // return

Disassembly of top(long x), x in %rdi:

T1  400545: 48 83 ef 05     sub $0x5,%rdi        // x-5
T2  400549: e8 f2 ff ff ff  callq 400540 <leaf>  // Call leaf(x-5)
T3  40054e: 48 01 c0        add %rax,%rax        // Double result
T4  400551: c3              retq                 // Return

...Call to top from function main

M1  40055b: e8 e5 ff ff ff  callq 400545, <top>  // Call top(100)
M2  400560: 48 89 c2        mov %rax,%rdx        // Resume

Instruction	State values (at beginning)

%rdi  %rax  %rsp  *%rsp  Description

M1  0x40055b  callq  100  —  0x7fffffffe820  —  Call top(100)
T1  0x400545  sub    100  —  0x7fffffffe818 0x400560  Entry of top
T2  0x400549  callq  95   —  0x7fffffffe818 0x400560  Cal leaf(95)
L1  0x400540  lea    95   —  0x7fffffffe810 0x40054e  Entry of leaf
L2  0x400544  retq   —  97   0x7fffffffe810 0x40054e  Return 97 from leaf
T3  0x40054e  add    —  97   0x7fffffffe818 0x400560  Resume top
T4  0x400551  retq   —  194  0x7fffffffe818 0x400560  Return 194 from top
M2  0x400560  mov    —  194  0x7fffffffe820  —  Resume main
### The Run-Time Stack: Example

#### Disassembly of `leaf(long y)`, y in `%rdi`:
```
0000000000400540 <leaf>:
L1 400540: 48 8d 47 02  lea 0x2(%rdi),%rax // y+2
L2 400544: c3        retq  // return
```

#### Disassembly of `top(long x)`, x in `%rdi`:
```
0000000000400540 <top>:
T1 400545: 48 83 ef 05  sub $0x5,%rdi        // x-5
T2 400549: e8 f2 ff ff ff  callq 400540 <leaf> // Call leaf(x-5)
T3 40054e: 48 01 c0        add %rax,%rax        // Double result
T4 400551: c3        retq  // Return
```

#### Call to `top` from function `main`:
```
M1 40055b: e8 e5 ff ff ff  callq 400545, <top> // Call top(100)
M2 400560: 48 89 c2        mov %rax,%rdx        // Resume
```

#### Stack values (at beginning):

<table>
<thead>
<tr>
<th>Instruction</th>
<th>State values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi %rax %rsp</td>
<td>*%rsp</td>
<td></td>
</tr>
<tr>
<td>M1 0x40055b</td>
<td>100 0x7fffffff820</td>
<td>Call <code>top(100)</code></td>
</tr>
<tr>
<td>T1 0x400545</td>
<td>100 0x7fffffff818 0x400560</td>
<td>Entry of <code>top</code></td>
</tr>
<tr>
<td>T2 0x400549</td>
<td>95 0x7fffffff818 0x400560</td>
<td>Call <code>leaf(95)</code></td>
</tr>
<tr>
<td>L1 0x400540</td>
<td>95 0x7fffffff810 0x40054e</td>
<td>Entry of <code>leaf</code></td>
</tr>
<tr>
<td>L2 0x400544</td>
<td>- 97 0x7fffffff810 0x40054e</td>
<td>Return 97 from <code>leaf</code></td>
</tr>
<tr>
<td>T3 0x40054e</td>
<td>- 97 0x7fffffff818 0x400560</td>
<td>Resume <code>top</code></td>
</tr>
<tr>
<td>T4 0x400551</td>
<td>- 194 0x7fffffff818 0x400560</td>
<td>Return 194 from <code>top</code></td>
</tr>
<tr>
<td>M2 0x400560</td>
<td>- 194 0x7fffffff820</td>
<td>Resume <code>main</code></td>
</tr>
</tbody>
</table>
### The Run-Time Stack: Example

Disassembly of leaf(long y), y in %rdi:
```
00000000000400540 <leaf>:
L1 400540: 48 8d 47 02  lea 0x2(%rdi),%rax     // y+2
L2 400544: c3                              retq     // return
```

Disassembly of top(long x), x in %rdi:
```
00000000000400540 <top>:
T1 400545: 48 83 ef 05  sub $0x5,%rdi        // x-5
T2 400549: e8 f2 ff ff ff  callq 400540 <leaf> // Call leaf(x-5)
T3 40054e: 48 01 c0  add %rax,%rax         // Double result
T4 400551: c3                              retq     // Return
```

### Stack "bottom"

- 0x7fffffffffe820
- 0x7fffffffffe818
- 0x400560

### Stack "top"

... Call to top from function main

M1 40055b: e8 e5 ff ff ff  callq 400545, <top> // Call top(100)
M2 400560: 48 89 c2  mov %rax,%rdx      // Resume

---

<table>
<thead>
<tr>
<th>Label</th>
<th>PC</th>
<th>Instruction</th>
<th>%rdi</th>
<th>%rax</th>
<th>%rsp</th>
<th>*%rsp</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>M1</td>
<td>0x40055b</td>
<td>callq</td>
<td>100</td>
<td>–</td>
<td>0x7fffffffffe820</td>
<td>–</td>
<td>Call top(100)</td>
</tr>
<tr>
<td>T1</td>
<td>0x400545</td>
<td>sub</td>
<td>100</td>
<td>–</td>
<td>0x7fffffffffe818</td>
<td>0x400560</td>
<td>Entry of top</td>
</tr>
<tr>
<td>T2</td>
<td>0x400549</td>
<td>callq</td>
<td>95</td>
<td>–</td>
<td>0x7fffffffffe818</td>
<td>0x400560</td>
<td>Call leaf(95)</td>
</tr>
<tr>
<td>L1</td>
<td>0x400540</td>
<td>lea</td>
<td>95</td>
<td>–</td>
<td>0x7fffffffffe810</td>
<td>0x40054e</td>
<td>Entry of leaf</td>
</tr>
<tr>
<td>L2</td>
<td>0x400544</td>
<td>retq</td>
<td>–</td>
<td>97</td>
<td>0x7fffffffffe810</td>
<td>0x40054e</td>
<td>Return 97 from leaf</td>
</tr>
<tr>
<td>T3</td>
<td>0x40054e</td>
<td>add</td>
<td>–</td>
<td>97</td>
<td>0x7fffffffffe818</td>
<td>0x400560</td>
<td>Resume top</td>
</tr>
<tr>
<td>T4</td>
<td>0x400551</td>
<td>retq</td>
<td>–</td>
<td>194</td>
<td>0x7fffffffffe818</td>
<td>0x400560</td>
<td>Return 194 from top</td>
</tr>
<tr>
<td>M2</td>
<td>0x400560</td>
<td>mov</td>
<td>–</td>
<td>194</td>
<td>0x7fffffffffe820</td>
<td>–</td>
<td>Resume main</td>
</tr>
</tbody>
</table>
### The Run-Time Stack: Example

**Disassembly of leaf(long y), y in %rdi:**

```
00000000000400540 <leaf>:
L1  400540: 48 8d 47 02     lea 0x2(%rdi),%rax // y+2
L2  400544: c3              retq // return
```

**Disassembly of top(long x), x in %rdi:**

```
00000000000400540 <top>:
T1  400545: 48 83 ef 05     sub $0x5,%rdi // x-5
T2  400549: e8 f2 ff ff ff  callq 400540 <leaf> // Call leaf(x-5)
T3  40054e: 48 01 c0        add %rax,%rax // Double result
T4  400551: c3              retq // Return
```

... Call to top from function main

```
M1  40055b: e8 e5 ff ff ff  callq 400545, <top> // Call top(100)
M2  400560: 48 89 c2        mov %rax,%rdx // Resume
```

**State values (at beginning)**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>PC</th>
<th>%rdi</th>
<th>%rax</th>
<th>%rsp</th>
<th>%*rsp</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>M1</td>
<td>0x40055b</td>
<td>100</td>
<td>-</td>
<td>0x7fffffff</td>
<td>0x7fffffff</td>
<td>Call top(100)</td>
</tr>
<tr>
<td>T1</td>
<td>0x400545</td>
<td>100</td>
<td>-</td>
<td>0x7fffffff</td>
<td>0x7fffffff</td>
<td>Entry of top</td>
</tr>
<tr>
<td>T2</td>
<td>0x400549</td>
<td>95</td>
<td>-</td>
<td>0x7fffffff</td>
<td>0x7fffffff</td>
<td>Cal leaf(95)</td>
</tr>
<tr>
<td>L1</td>
<td>0x400540</td>
<td>95</td>
<td>-</td>
<td>0x7fffffff</td>
<td>0x7fffffff</td>
<td>Entry of leaf</td>
</tr>
<tr>
<td>L2</td>
<td>0x400544</td>
<td>-</td>
<td>97</td>
<td>0x7fffffff</td>
<td>0x7fffffff</td>
<td>Return 97 from leaf</td>
</tr>
<tr>
<td>T3</td>
<td>0x40054e</td>
<td>-</td>
<td>97</td>
<td>0x7fffffff</td>
<td>0x7fffffff</td>
<td>Resume top</td>
</tr>
<tr>
<td>T4</td>
<td>0x400551</td>
<td>-</td>
<td>194</td>
<td>0x7fffffff</td>
<td>0x7fffffff</td>
<td>Return 194 from top</td>
</tr>
<tr>
<td>M2</td>
<td>0x400560</td>
<td>-</td>
<td>194</td>
<td>0x7fffffff</td>
<td>0x7fffffff</td>
<td>Resume main</td>
</tr>
</tbody>
</table>
The Run-Time Stack: Example

Disassembly of leaf(long y), y in %rdi:

00000000000400540 <leaf>:
L1 400540: 48 8d 47 02 lea 0x2(%rdi),%rax // y+2
L2 400544: c3 retq // return

Disassembly of top(long x), x in %rdi:

00000000000400540 <top>:
T1 400545: 48 83 ef 05 sub $0x5,%rdi // x-5
T2 400549: e8 f2 ff ff ff callq 400540 <leaf> // Call leaf(x-5)
T3 40054e: 48 01 c0 add %rax,%rax // Double result
T4 400551: c3 retq // Return

... Call to top from function main

M1 40055b: e8 e5 ff ff ff callq 400545, <top> // Call top(100)
M2 400560: 48 89 c2 mov %rax,%rdx // Resume

<table>
<thead>
<tr>
<th>Instruction</th>
<th>State values (at beginning)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>%rax</td>
</tr>
<tr>
<td>M1 0x40055b</td>
<td>callq</td>
</tr>
<tr>
<td>T1 0x400545</td>
<td>sub</td>
</tr>
<tr>
<td>T2 0x400549</td>
<td>callq</td>
</tr>
<tr>
<td>L1 0x400540</td>
<td>lea</td>
</tr>
<tr>
<td>L2 0x400544</td>
<td>retq</td>
</tr>
<tr>
<td>T3 0x40054e</td>
<td>add</td>
</tr>
<tr>
<td>T4 0x400551</td>
<td>retq</td>
</tr>
<tr>
<td>M2 0x400560</td>
<td>mov</td>
</tr>
</tbody>
</table>
Data Transfer

- Procedure calls can involve passing data as arguments, and can return a value to the calling function, as well.
- Most often, the arguments can fit into registers, so we don't need to involve the stack. If there are more than six integer arguments, we put them onto the stack.
- When one function calls another, the calling function needs to copy the arguments into the proper registers (%rdi, %rsi, %rdx, %rcx, %r8, %r9, in that order)
- If there are more than 6 arguments, the calling function allocates space on the stack in 8-byte chunks (even if smaller values are passed).
- The return value (if an integer) is returned in the %rax register.
Data Transfer

- Example:

```c
void proc(long a1, long *a1p,
    int a2, int *a2p,
    short a3, short *a3p,
    char a4, char *a4p)
{
    *a1p += a1;
    *a2p += a2;
    *a3p += a3;
    *a4p += a4;
}
```

Arguments passed as follows:

- `a1` in `%rdi` (64 bits)
- `a1p` in `%rsi` (64 bits)
- `a2` in `%edx` (32 bits)
- `a2p` in `%rcx` (64 bits)
- `a3` in `%r8w` (16 bits)
- `a3p` in `%r9` (64 bits)
- `a4` at `%rsp+8` (8 bits)
- `a4p` at `%rsp+16` (64 bits)

```
int main()
{
    long a = 1;
    int b = 2;
    short c = 3;
    char d = 4;

    proc(a,&a,b,&b,c,&c,d,&d);
    printf("a:%ld, b:%d, c:%d, d:%d\n",a,b,c,d);
    return 0;
}
```

```
int main()
{
    long a = 1;
    int b = 2;
    short c = 3;
    char d = 4;

    proc(a,&a,b,&b,c,&c,d,&d);
    printf("a:%ld, b:%d, c:%d, d:%d\n",a,b,c,d);
    return 0;
}
```
void proc(long a1, long *a1p,  
        int a2, int *a2p,  
        short a3, short *a3p,  
        char a4, char *a4p)  
{  
    *a1p += a1;  
    *a2p += a2;  
    *a3p += a3;  
    *a4p += a4;  
}  

Arguments passed as follows:

a1 in %rdi    (64 bits)
a1p in %rsi   (64 bits)
a2 in %edx    (32 bits)
a2p in %rcx   (64 bits)
a3 in %r8w   (16 bits)
a3p in %r9    (64 bits)
a4 at %rsp+8   ( 8 bits)
a4p at %rsp+16 (64 bits)

Why movl and not movb?!
Example:

```c
void proc(long a1, long *a1p,
    int a2, int *a2p,
    short a3, short *a3p,
    char a4, char *a4p)
{
    *a1p += a1;
    *a2p += a2;
    *a3p += a3;
    *a4p += a4;
}
```

Arguments passed as follows:

```plaintext
a1  in %rdi    (64 bits)
a1p in %rsi    (64 bits)
a2  in %edx    (32 bits)
a2p in %rcx    (64 bits)
a3  in %r8w    (16 bits)
a3p in %r9     (64 bits)
a4  at %rsp+8  ( 8 bits)
a4p at %rsp+16 (64 bits)
```

Why `movl` and not `movb`?!

Under the hood, a mov instruction will still fetch an entire 64-bits, so this is actually faster than retrieving 64-bits and masking out the upper bits (not required information for you to know!)
We haven't seen much use of local storage on the stack, but there are times when it is necessary:

- When there are not enough registers to hold the local data
- When the address operator '\&' is applied to a local variable, and we have to be able to generate an address for it.
- Some of the local variables are arrays or structs, and must be accessed by array or structure references.
- The typical way to allocate space on the stack frame is to decrement the stack pointer.
- Remember, a function must return the stack pointer to the proper value (such that the top of the stack is the return address) before it returns.
Local Stack Storage

• Example:

```c
long swap_add(long *xp, long *yp)
{
    long x = *xp;
    long y = *yp;
    *xp = y;
    *yp = x;
    return x + y;
}
```

```c
long caller()
{
    long arg1 = 534;
    long arg2 = 1057;
    long sum = swap_add(&arg1, &arg2);
    long diff = arg1 - arg2;
    return sum * diff;
}
```

caller:

```
subq $16, %rsp      // allocate 16 bytes for stack frame
movq $534, (%rsp)   // store 534 in arg1
movq $1057, 8(%rsp) // store 1057 in arg2
lea 8(%rsp), %rsi   // compute &arg2 as second argument
movq %rsp, %rdi     // compute &arg1 as first argument
call swap_add       // call swap_add(&arg1, &arg2)
movq (%rsp),%rdx    // get arg1
subq 8(%rsp), %rdx  // compute diff = arg1 - arg2
imulq %rdx, %rax    // compute sum * diff
addq $16, %rsp      // deallocate stack frame
ret
```

• The caller must allocate a stack frame due to the presence of address operators.
Local Storage in Registers

• You may not have noticed, but none of the examples in the book so far have used `%rbx` for anything, and gcc often uses `%rax`, `%rcx`, `%rdx`, etc., but skips right over `%rbx`.

• One reason is that `%rbx` is designated, by convention, to be a "caller owned" register:

<table>
<thead>
<tr>
<th>%rax</th>
<th>%eax</th>
<th>%ax</th>
<th>%al</th>
</tr>
</thead>
<tbody>
<tr>
<td>return value</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%rbx</td>
<td>%ebx</td>
<td>%bx</td>
<td>%bl</td>
</tr>
<tr>
<td>caller owned</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

• What that means is that if a function uses `%rbx`, it guarantees that it will restore `%rbx` to its original value when the function returns.

• The full list of caller owned registers are `%rbx`, `%rbp`, and `%r12-%r15.

If a function uses any of those registers, it must save them on the stack to restore.
Local Storage in Registers

- The other registers are "callee owned", meaning that if function P calls function Q, function P must save the values of those registers on the stack (or in caller owned registers!) if it wants to retain the data after function Q is called.
- Example:

```c
long P(long x, long y) {
    long u = Q(y);
    long v = Q(x);
    return u + v;
}
```

- The first time Q is called, x must be saved for later, and the second time Q is called, u must be saved for later.
Recursion!

- The conventions we have been discussing allow for functions to call themselves. Each procedure call has its own private space on the stack, and the local variables from all of the function calls do not interfere with each other.
- The only thing a program needs to worry about is a stack overflow, because the stack is a limited resource for a program.
- Example:

```c
long rfact(long n)
{
    long result;
    if (n <= 1)
        result = 1;
    else
        result = n * rfact(n-1);
    return result;
}
```

```assembly
rfact:
    pushq %rbx
    save %rbx
    movq %rdi,%rbx
    store n in caller-owned reg
    movl $1,%eax
    set return value = 1
    cmpq $1,%rdi
    compare n:1
    jle .L35
    if <=, jump to done
    leaq -1(%rdi),%rdi
    compute n-1
    call rfact
    recursively call rfact(n-1)
    imulq %rbx,%rax
    multiply result by n
    .L35
    done:
    pop %rbx
    restore %rbx
    retq
    return
```
Array Allocation and Access

- Arrays in C map in a fairly straightforward way to X86 assembly code, thanks to the addressing modes available in instructions.
- When we perform pointer arithmetic, the assembly code that is produced will have address computations built into them.
- Optimizing compilers are very good at simplifying the address computations (in lab you will see another optimizing compiler benefit in the form of division — if the compiler can avoid dividing, it will!). Because of the transformations, compiler-generated assembly for arrays often doesn't look like what you are expecting.
- Consider the following form of a data type $T$ and integer constant $N$:

$$ T \ A[N] $$

- The starting location is designated as $x_A$
- The declaration allocates $N \times \text{sizeof}(T)$ bytes, and gives us an identifier that we can use as a pointer (but it isn't a pointer!), with a value of $x_A$. 
Array Allocation and Access

- Example:

<table>
<thead>
<tr>
<th>Array</th>
<th>Element Size</th>
<th>Total Size</th>
<th>Start address</th>
<th>Element i</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>A[12];</td>
<td>1</td>
<td>12</td>
<td>$X_A + i$</td>
</tr>
<tr>
<td>char</td>
<td>*B[8];</td>
<td>8</td>
<td>64</td>
<td>$X_B + 8i$</td>
</tr>
<tr>
<td>int</td>
<td>C[6];</td>
<td>4</td>
<td>24</td>
<td>$X_C + 4i$</td>
</tr>
<tr>
<td>double</td>
<td>*D[5]</td>
<td>8</td>
<td>40</td>
<td>$X_D + 8i$</td>
</tr>
</tbody>
</table>

- The memory referencing operations in x86-64 are designed to simplify array access. Suppose we wanted to access $C[3]$ above. If the address of $C$ is in register `%rdx`, and 3 is in register `%rcx`.
- The following copies $C[3]$ into `%eax`,

```
    movl (%rdx,%rcx,4), %eax
```
Pointer Arithmetic

- C allows arithmetic on pointers, where the computed value is calculated according to the size of the data type referenced by the pointer.
- The array reference \texttt{A[i]} is identical to \texttt{*(A+i)}
- Example: if the address of array \texttt{E} is in \texttt{%rdx}, and the integer index, \texttt{i}, is in \texttt{%rcx}, the following are some expressions involving \texttt{E}:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Type</th>
<th>Value</th>
<th>Assembly Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{E}</td>
<td>int *</td>
<td>\texttt{x_E}</td>
<td>\texttt{movq %rdx, %rax}</td>
</tr>
<tr>
<td>\texttt{E[0]}</td>
<td>int</td>
<td>\texttt{M[x_E]}</td>
<td>\texttt{movl (%rdx), %eax}</td>
</tr>
<tr>
<td>\texttt{E[i]}</td>
<td>int</td>
<td>\texttt{M[x_E+4i]}</td>
<td>\texttt{movl (%rdx,%rcx,4) %eax}</td>
</tr>
<tr>
<td>\texttt{&amp;E[2]}</td>
<td>int *</td>
<td>\texttt{x_E+8}</td>
<td>\texttt{leaq 8(%rdx), %rax}</td>
</tr>
<tr>
<td>\texttt{E+i-1}</td>
<td>int *</td>
<td>\texttt{x_E+4i-4}</td>
<td>\texttt{leaq -4(%rdx,%rcx,4), %rax}</td>
</tr>
<tr>
<td>\texttt{*(E+i-3)}</td>
<td>int</td>
<td>\texttt{M[x_E+4i-12]}</td>
<td>\texttt{movl -12(%rdx,%rcx,4) %eax}</td>
</tr>
<tr>
<td>\texttt{&amp;E[i]-E}</td>
<td>long</td>
<td>\texttt{i}</td>
<td>\texttt{movq %rcx,%rax}</td>
</tr>
</tbody>
</table>
Pointer Arithmetic

- Practice: $x_S$ is the address of a `short` integer array, $S$, stored in `%rdx`, and a long integer index, $i$, is stored in register `%rcx`.
- For each of the following expressions, give its type, a formula for its value, and an assembly-code implementation. The result should be stored in `%rax` if it is a pointer, and the result should be in register `%ax` if it has a data type `short`.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Type</th>
<th>Value</th>
<th>Assembly Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S+1$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$S[3]$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&amp;$S[i]$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$S[4*i+1]$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$S+i−5$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Pointer Arithmetic

- Practice: $x_S$ is the address of a **short** integer array, $S$, stored in `%rdx`, and a long integer index, $i$, is stored in register `%rcx`.
- For each of the following expressions, give its type, a formula for its value, and an assembly-code implementation. The result should be stored in `%rax` if it is a pointer, and the result should be in register `%ax` if it has a data type **short**.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Type</th>
<th>Value</th>
<th>Assembly Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S+1$</td>
<td>short *</td>
<td>$x_S + 2$</td>
<td><code>leaq 2(%rdx),%rax</code></td>
</tr>
<tr>
<td>$S[3]$</td>
<td>short</td>
<td>$M[x_S + 6]$</td>
<td><code>movw 6(%rdx),%ax</code></td>
</tr>
<tr>
<td>&amp;$S[i]$</td>
<td>short *</td>
<td>$x_S + 2i$</td>
<td><code>leaq (%rdx,%rcx,2),%rax</code></td>
</tr>
<tr>
<td>$S[4*i+1]$</td>
<td>short</td>
<td>$M[x_S + 8i + 2]$</td>
<td><code>movw 2(%rdx,%rcx,8),%ax</code></td>
</tr>
<tr>
<td>$S+i-5$</td>
<td>short *</td>
<td>$x_S + 2i - 10$</td>
<td><code>leaq -10(%rdx,%rcx,2),%rax</code></td>
</tr>
</tbody>
</table>
Structures

- The C `struct` declaration is used to group objects of different types into a single unit.
- Each "field" is referenced by a name, and can be accessed using dot (.) or (if there is a pointer to the struct) arrow (→) notation.
- Structures are kept in contiguous memory.
- A pointer to a struct is to its first byte, and the compiler maintains the byte offset information for each field.
- In assembly, the references to the fields are via the byte offsets.
• Example:

```c
struct rec {
    int i;
    int j;
    int a[2];
    int *p;
};
```

• This structure has four fields: two 4-byte values of type `int`, a two-element array of type `int`, and an 8-byte `int` pointer, for a total of 24 bytes:

<table>
<thead>
<tr>
<th>Offset</th>
<th>0</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>24</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contents</td>
<td>i</td>
<td>j</td>
<td>a[0]</td>
<td>a[1]</td>
<td>p</td>
</tr>
</tbody>
</table>

• The numbers along the top of the diagram are the byte offsets of the fields from the beginning of the structure.
• Note that the array is embedded in the structure.
• To access the fields, the compiler generates code that adds the field offset to the address of the structure.
Structures

- Example:

```c
struct rec {
    int i;
    int j;
    int a[2];
    int *p;
};
```

- This structure has four fields: two 4-byte values of type `int`, a two-element array of type `int`, and an 8-byte `int` pointer, for a total of 24 bytes:

```
<table>
<thead>
<tr>
<th>Offset</th>
<th>0</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>24</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contents</td>
<td>i</td>
<td>j</td>
<td>a[0]</td>
<td>a[1]</td>
<td>p</td>
</tr>
</tbody>
</table>
```

- Example: Variable `r` of type `struct rec *` is in register `%rdi`. The following copies element `r->i` to element `r->j`:

```asm
movl (%rdi), %eax  // get r->i
movl %eax, 4(%rdi)  // store in r->j
```

- The offset of `i` is 0, so `i`'s field is `%rdi`. The offset of `j` is 4, so the offset of 4 is added to the address of `%rdi` to store into `j`. 
Structures

• Example:

```c
define struct rec {
    int i;
    int j;
    int a[2];
    int *p;
};
```

• This structure has four fields: two 4-byte values of type `int`, a two-element array of type `int`, and an 8-byte `int` pointer, for a total of 24 bytes:

<table>
<thead>
<tr>
<th>Offset</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>i</td>
</tr>
<tr>
<td>4</td>
<td>j</td>
</tr>
<tr>
<td>8</td>
<td>a[0]</td>
</tr>
<tr>
<td>16</td>
<td>a[1]</td>
</tr>
<tr>
<td>24</td>
<td>p</td>
</tr>
</tbody>
</table>

• We can generate a pointer to a field by adding the field's offset to the struct address. To generate `&(r->a[1])` we add offset `8 + 4 = 12`. For a pointer `r` in register `%rdi` and long `int` variable `i` in `%rsi`, we can generate the pointer value `&(r->a[i])` with one instruction:

```
leaq 8(%rdi,%rsi,4), %rax // set %rax to &r->a[i]
```
• Example:

```
struct rec {
    int i;
    int j;
    int a[2];
    int *p;
};
```

• This structure has four fields: two 4-byte values of type `int`, a two-element array of type `int`, and an 8-byte `int` pointer, for a total of 24 bytes:

<table>
<thead>
<tr>
<th>Offset</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>i</td>
</tr>
<tr>
<td>4</td>
<td>j</td>
</tr>
<tr>
<td>8</td>
<td>a[0]</td>
</tr>
<tr>
<td>16</td>
<td>a[1]</td>
</tr>
<tr>
<td>24</td>
<td>p</td>
</tr>
</tbody>
</table>

• The following code implements `r->p = &r->a[r->i + r->j];`

```
// r in %rdi
movl 4(%rdi),%eax         // get r->j
addl (%rdi),%eax          // add r->i
cltq                      // extend %eax to 8 bytes, %rax
leaq 8(%rdi,%rax,4), %rax // compute &r->a[r->i + r->j]
movq %rax, 16(%rdi)        // store in r->p
```
Structures

• Example:

```c
struct rec {
    int i;
    int j;
    int a[2];
    int *p;
};
```

• This structure has four fields: two 4-byte values of type `int`, a two-element array of type `int`, and an 8-byte `int` pointer, for a total of 24 bytes:

<table>
<thead>
<tr>
<th>Offset</th>
<th>0</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>24</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contents</td>
<td>i</td>
<td>j</td>
<td>a[0]</td>
<td>a[1]</td>
<td>p</td>
</tr>
</tbody>
</table>

• Notice that all struct manipulation is handled at compile time, and the machine code doesn't contain any information about the field declarations or the names of the fields.

• The compiler does all the work, keeping track as it produces the assembly code.

• BTW, if you're curious about how the compiler actually does the transformation from C to assembly, take a compilers class, e.g., CS143.
Data Alignment

- Computer systems often put restrictions on the allowable addresses for primitive data types, requiring that the address for some objects must be a multiple of some value $K$ (normally 2, 4, or 8).
- These alignment restrictions simplify the design of the hardware.
- For example, suppose that a processor always fetches 8 bytes from the memory system, and an address must be a multiple of 8. If we can guarantee that any double will be aligned to have its address as a multiple of 8, then we can read or write the values with a single memory access.
- For x86-64, Intel recommends the following alignments for best performance:

<table>
<thead>
<tr>
<th>$K$</th>
<th>Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>char</td>
</tr>
<tr>
<td>2</td>
<td>short</td>
</tr>
<tr>
<td>4</td>
<td>int, float</td>
</tr>
<tr>
<td>8</td>
<td>long, double, char *</td>
</tr>
</tbody>
</table>
• The compiler enforces alignment by making sure that every data type is organized in such a way that every field within the struct satisfies the alignment restrictions.

• For example, let's look at the following struct:

  ```c
  struct S1 {
      int i;
      char c;
      int j;
  };
  ```

• If the compiler used a minimal allocation:

• This would make it impossible to align fields `i` (offset 0) and `j` (offset 5). Instead, the compiler inserts a 3-byte gap between fields `c` and `j`:

• So, don't be surprised if your structs have a `sizeof()` that is larger than you expect!
Let's look at the following code:

```c
void *gfind_max(void *arr, int n, size_t elemsz,
                 int (*compar)(const void *, const void *))
{
    void *pmax = arr;
    for (int i = 1; i < n; i++) {
        void *ith = (char *)arr + i*elemsz;
        if (compar(ith, pmax) > 0)
            pmax = ith;
    }
    return pmax;
}

int cmp_alpha(const void *p, const void *q)
{
    const char *first = *(const char **)p;
    const char *second = *(const char **)q;
    return strcmp(first, second);
}

int main(int argc, char *argv[])
{
    char **pmax = gfind_max(argv+1, argc-1, sizeof(argv[0]), cmp_alpha);
    printf("max = %s\n", *pmax);
    return 0;
}
```
Let's look at the following code:

```c
void *gfind_max(void *arr, int n, size_t elemsz,
    int (*compar)(const void *, const void *))
{
    void *pmax = arr;
    for (int i = 1; i < n; i++) {
        void *ith = (char *)arr + i*elemsz;
        if (compar(ith, pmax) > 0)
            pmax = ith;
    }
    return pmax;
}

int cmp_alpha(const void *p, const void *q)
{
    const char *first = *(const char **)p;
    const char *second = *(const char **)q;
    return strcmp(first, second);
}

int main(int argc, char *argv[])
{
    char **pmax = gfind_max(argv+1, argc-1, sizeof(argv[0]), cmp_alpha);
    printf("max = %s\n", *pmax);
    return 0;
}
```

Because `compar` is a function pointer, the compiler calls the function via the address that is in the compar variable.

Let's take a look at this in gdb.
References and Advanced Reading

• References:
  • Stanford guide to x86-64: https://web.stanford.edu/class/cs107/guide/x86-64.html
  • CS107 one-page of x86-64: https://web.stanford.edu/class/cs107/resources/onepage_x86-64.pdf
  • gdbtui: https://beej.us/guide/bggdb/
  • More gdbtui: https://sourceware.org/gdb/onlinedocs/gdb/TUI.html
  • Compiler explorer: https://gcc.godbolt.org

• Advanced Reading:
  • Stack frame layout on x86-64: https://eli.thegreenplace.net/2011/09/06/stack-frame-layout-on-x86-64
  • history of x86 instructions: https://en.wikipedia.org/wiki/X86_instruction_listings
  • x86-64 Wikipedia: https://en.wikipedia.org/wiki/X86-64