CS107, Lecture 14
Assembly: The Runtime Stack

Reading: B&O 3.7
Learning Goals

• Learn what %rip represents and how it is updated.
• Learn how assembly calls functions and manages stack frames.
• Learn the rules of register use when calling functions.
Plan For Today

• The Instruction Pointer (%rip)
• Calling Functions
  • The Stack
  • Passing Control
  • Passing Data
  • Local Storage
• Break: Announcements
• Register Restrictions
• Pulling it all together: recursion example
Plan For Today

- The Instruction Pointer (%rip)
- Calling Functions
  - The Stack
  - Passing Control
  - Passing Data
  - Local Storage
- Break: Announcements
- Register Restrictions
- Pulling it all together: recursion example
• `%rip` is a special register that points to the instruction currently executing.
• Let’s dive deeper into how `%rip` works, and how jumps modify it.
%rip

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
%rip

000000000004004ed <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
000000000004004ed <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
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
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
%rip

000000000004004ed <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
%rip

000000000004004ed <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
Instructions Are Just Bytes!
Memory bus

Main memory

“hello, world\

hello code
Instructions Are Just Bytes!

Main Memory

Stack

Heap

Data

Text (code)

0x0
Instructions Are Just Just Bytes!

Parameters and local variables

Main Memory

- Stack
- Heap
- Data
- Text (code)

0x0
Instructions Are Just Bytes!

Allocated memory (malloc, calloc, realloc)

Main Memory

Stack

Heap

Data

Text (code)
Instructions Are Just Bytes!

Global variables (boo!), static variables and string literals

Main Memory

Stack

Heap

Data

Text (code)

0x0
Instructions Are Just Bytes!

Main Memory

Stack

Heap

Data

Text (code)

Machine code instructions

0x0
00000000000000004004ed <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
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
%rip

000000000004004ed <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
000000000004004ed <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  

%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  |
| 4004f0 | e5  |
| 4004ef | 89  |
| 4004ee | 48  |
| 4004ed | 55  |
%rip

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

%rip

0x4004f1
000000000004004ed <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

%rip

0x4004f8
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

0x4004fc
Special hardware is responsible for setting %rip’s value to the next instruction. It does %rip += size of current instruction (in bytes).
void loop() {
    int i = 0;
    while (i < 100) {
        i++;
    }
}
void loop() {
    int i = 0;
    while (i < 100) {
        i++;
    }
}

These are 0-based offsets in bytes for each instruction relative to the start of this function.
void loop() {
    int i = 0;
    while (i < 100) {
        i++;
    }
}

These are bytes for the machine code instructions. Instructions are variable length.
void loop() {
    int i = 0;
    while (i < 100) {
        i++;
    }
}

0x400570 <+0>: b8 00 00 00 00 mov $0x0,%eax
0x400575 <+5>: eb 03 jmp 0x40057a <loop+10>
0x400577 <+7>: 83 c0 01 add $0x1,%eax
0x40057a <+10>: 83 f8 63 cmp $0x63,%eax
0x40057d <+13>: 73 f8 jle 0x400577 <loop+7>
0x40057f <+15>: f3 c3 repz retq
```assembly
0x400570 <+0>: b8 00 00 00 00 mov $0x0,%eax
0x400575 <+5>: eb 03 jmp 0x40057a <loop+10>
0x400577 <+7>: 83 c0 01 add $0x1,%eax
0x40057a <+10>: 83 f8 63 cmp $0x63,%eax
0x40057d <+13>: 73 f8 jle 0x400577 <loop+7>
0x40057f <+15>: f3 c3 repz retq
```
%rip

0x400570 <+0>:  b8 00 00 00 00 00  mov $0x0,%eax
0x400575 <+5>:  eb 03  jmp 0x40057a <loop+10>
0x400577 <+7>:  83 c0 01  add $0x1,%eax
0x40057a <+10>: 83 f8 63  cmp $0x63,%eax
0x40057d <+13>: 73 f8  jle 0x400577 <loop+7>
0x40057f <+15>: f3 c3  repz retq

0xeb means jmp.
0x03 is the number of instruction bytes to jump relative to %rip.

With no jump, %rip would advance to the next line. This `jmp` says to then go 3 bytes further!
%rip

0x400570 <+0>:  b8 00 00 00 00  mov $0x0,%eax
0x400575 <+5>:  eb 03            jmp 0x40057a <loop+10>
0x400577 <+7>:  83 c0 01        add $0x1,%eax
0x40057a <+10>: 83 f8 63       cmp $0x63,%eax
0x40057d <+13>: 73 f8          jle 0x400577 <loop+7>
0x40057f <+15>: f3 c3          repz retq

0x03 is the number of instruction bytes to jump relative to %rip.

With no jump, %rip would advance to the next line. This jmp says to then go 3 bytes further!
%rip

0x400570 <+0>:  b8 00 00 00 00 mov $0x0,%eax
0x400575 <+5>:  eb 03       jmp 0x40057a <loop+10>
0x400577 <+7>:  83 c0 01       add $0x1,%eax
0x40057a <+10>: 83 f8 63       cmp $0x63,%eax
0x40057d <+13>: 73 f8       jle 0x400577 <loop+7>
0x40057f <+15>: f3 c3       repz retq

0x73 means jle.
0x400570 <+0>:  b8 00 00 00 00  mov  $0x0,%eax
0x400575 <+5>:  eb 03           jmp  0x40057a <loop+10>
0x400577 <+7>:  83 c0 01       add  $0x1,%eax
0x40057a <+10>: 83 f8 63       cmp  $0x63,%eax
0x40057d <+13>: 73 f8           jle  0x400577 <loop+7>
0x40057f <+15>: f3 c3           repz retq

0xf8 is the number of instruction bytes to jump relative to %rip. This is -8 (in two’s complement!).

With no jump, %rip would advance to the next line. This jmp says to then go 8 bytes back!
%rip

0x400570 <+0>:  b8 00 00 00 00  mov $0x0,%eax
0x400575 <+5>:  eb 03          jmp 0x40057a <loop+10>
0x400577 <+7>:  83 c0 01       add $0x1,%eax
0x40057a <+10>: 83 f8 63      cmp $0x63,%eax
0x40057d <+13>: 73 f8          jle 0x400577 <loop+7>
0x40057f <+15>: f3 c3         repz retq

0xf8 is the number of instruction bytes to jump relative to %rip. This is -8 (in two’s complement!).

With no jump, %rip would advance to the next line. This jmp says to then go 8 bytes back!
Summary: Instruction Pointer

• Machine code instructions live in main memory, just like stack and heap data.
• %rip is a register that stores a number (an address) of the currently executing instruction. It marks where we currently are in the program’s instructions.
• To advance to the next instruction, special hardware adds the size of the current instruction in bytes.
• jmp instructions work by adjusting %rip by a specified amount.
Plan For Today

- The Instruction Pointer (%rip)
- Calling Functions
  - The Stack
  - Passing Control
  - Passing Data
  - Local Storage
- Break: Announcements
- Register Restrictions
- Pulling it all together: recursion example
How do we call functions in assembly?
To call a function in assembly, we must do a few things:

• **Pass Control** – %rip must be adjusted to execute the callee’s instructions, and then resume the caller’s instructions afterwards.

• **Pass Data** – we must pass any parameters and receive any return value.

• **Manage Memory** – we must handle any space needs of the callee on the stack.

Terminology: **caller** function calls the **callee** function.
Plan For Today

• The Instruction Pointer (%rip)

• Calling Functions
  • The Stack
  • Passing Control
  • Passing Data
  • Local Storage

• **Break:** Announcements

• Register Restrictions

• Pulling it all together: recursion example
• `%rsp` is a special register that stores the address of the current “top” of the stack (the bottom in our diagrams, since the stack grows downwards).
• `%rsp` is a special register that stores the address of the current “top” of the stack (the bottom in our diagrams, since the stack grows downwards).
• `%rsp` is a special register that stores the address of the current “top” of the stack (the bottom in our diagrams, since the stack grows downwards).
• **%rsp** is a special register that stores the address of the current “top” of the stack (the bottom in our diagrams, since the stack grows downwards).
%\texttt{rip}\textsuperscript{\%} is a special register that stores the address of the current “top” of the stack (the bottom in our diagrams, since the stack grows downwards).

Key idea: \%\texttt{rsp} must point to the same place before and after a function is called, since stack frames go away when a function finishes.
• The **push** instruction pushes the data at the specified source onto the top of the stack, adjusting %rsp accordingly.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>pushq S</td>
<td>R[%rsp] ← R[%rsp] – 8; M[R[%rsp]] ← S</td>
</tr>
</tbody>
</table>
The **push** instruction pushes the data at the specified source onto the top of the stack, adjusting %rsp accordingly.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
</tr>
</thead>
</table>
| pushq S     | $R[\%rsp] \leftarrow R[\%rsp] - 8;$  
$M[R[\%rsp]] \leftarrow S$ |
• The **push** instruction pushes the data at the specified source onto the top of the stack, adjusting `%rsp` accordingly.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pushq S</code></td>
<td><code>R[%rsp] ← R[%rsp] - 8; M[R[%rsp]] ← S</code></td>
</tr>
</tbody>
</table>
The **push** instruction pushes the data at the specified source onto the top of the stack, adjusting `%rsp` accordingly.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>pushq S</td>
<td>( R[%rsp] \leftarrow R[%rsp] - 8; )</td>
</tr>
<tr>
<td></td>
<td>( M[R[%rsp]] \leftarrow S )</td>
</tr>
</tbody>
</table>

This behavior is equivalent to the following, but `pushq` is a shorter instruction:

```
subq $8, %rsp
movq S, (%rsp)
```

Sometimes, you’ll see instructions just explicitly decrement the stack pointer to make room for future data.
• The **pop** instruction pops the topmost data from the stack and stores it in the specified destination, adjusting `%rsp` accordingly.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
</tr>
</thead>
</table>
| popq D      | D ← M[R[%rsp]]  
R[%rsp] ← R[%rsp] + 8; |

• **Note**: this *does not* remove/clear out the data! It just increments `%rsp` to indicate the next push can overwrite that location.
The **pop** instruction pops the topmost data from the stack and stores it in the specified destination, adjusting `%rsp` accordingly.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
</tr>
</thead>
</table>
| popq D     | \( D \leftarrow M[R[\%rsp]] \)  
\( R[\%rsp] \leftarrow R[\%rsp] + 8; \) |

This behavior is equivalent to the following, but popq is a shorter instruction:

```
movq (%rsp), D  
addq $8, %rsp
```

Sometimes, you’ll see instructions just explicitly increment the stack pointer to pop data.
Pushing onto the Stack

- Example:

<table>
<thead>
<tr>
<th>Initially</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>0x123</td>
</tr>
<tr>
<td>%rdx</td>
<td>0</td>
</tr>
<tr>
<td>%rsp</td>
<td>0x108</td>
</tr>
</tbody>
</table>

Stack "bottom"

Increasing address

0x108

Stack "top"
Pushing onto the Stack

- Example:

<table>
<thead>
<tr>
<th>Initially</th>
<th>pushq %rax</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax 0x123</td>
<td>%rax 0x123</td>
</tr>
<tr>
<td>%rdx 0</td>
<td>%rdx 0</td>
</tr>
<tr>
<td>%rsp 0x108</td>
<td>%rsp 0x100</td>
</tr>
</tbody>
</table>

Stack "bottom"

Increasing address

Stack "top"
Pushing onto the Stack

- Example:

<table>
<thead>
<tr>
<th>Initially</th>
<th>pushq %rax</th>
<th>popq %rdx</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax: 0x123</td>
<td>%rax: 0x123</td>
<td>%rax: 0x123</td>
</tr>
<tr>
<td>%rdx: 0</td>
<td>%rdx: 0</td>
<td>%rdx: 0x123</td>
</tr>
<tr>
<td>%rsp: 0x108</td>
<td>%rsp: 0x100</td>
<td>%rsp: 0x108</td>
</tr>
</tbody>
</table>

Stack "bottom"

- Increasing address
- 0x108

Stack "top"
To call a function in assembly, we must do a few things:

- **Pass Control** – `%rip` must be adjusted to execute the callee’s instructions, and then resume the caller’s instructions afterwards.

- **Pass Data** – we must pass any parameters and receive any return value.

- **Manage Memory** – we must handle any space needs of the callee on the stack.

Terminology: **caller** function calls the **callee** function.
Plan For Today

• The Instruction Pointer (%rip)

• Calling Functions
  • The Stack
  • Passing Control
  • Passing Data
  • Local Storage

• Break: Announcements
• Register Restrictions
• Pulling it all together: recursion example
• **Problem:** %rip points to the currently executing instruction. To call a function, we must remember the next caller instruction to resume at after.

• **Solution:** push the current value of %rip onto the stack. Then call the function. When it is finished, put this value back into %rip and continue executing.

E.g. `main()` calls `foo()`:

```
E.g. main() calls foo():

Stack

main()

...%rip 0x3021
%rsp 0xff20
```

E.g. `main()` calls `foo()`:

```
E.g. main() calls foo():

Stack

main()

...%rip 0x3021
%rsp 0xff20
```
Remembering Where We Left Off

**Problem:** %rip points to the currently executing instruction. To call a function, we must remember the next caller instruction to resume at after.

**Solution:** push the current value of %rip onto the stack. Then call the function. When it is finished, put this value back into %rip and continue executing.

E.g. `main()` calls `foo()`:

```
Stack
...
main()
...
```

```
%rip 0x3021
%rsp 0xff18
```

```
0x3026
```
Remembering Where We Left Off

• **Problem:** %rip points to the currently executing instruction. To call a function, we must remember the next caller instruction to resume at after.

• **Solution:** push the current value of %rip onto the stack. Then call the function. When it is finished, put this value back into %rip and continue executing.

*E.g. main() calls foo:*

---

```
Stack
...
0x3026
...
foo()
...
main()
...

%rip 0x4058
%rsp 0xff08
```

E.g. main() calls foo():

```
... 0x3026 ...
foo() ...
main() ...
```

---

```
... 0x3026 ...
foo() ...
main() ...
```
• **Problem:** %rip points to the currently executing instruction. To call a function, we must **remember** the next caller instruction to resume at after.

• **Solution:** push the current value of %rip onto the stack. Then call the function. When it is finished, put this value back into %rip and continue executing.

E.g. `main()` calls `foo()`:

Stack

```
%rip  0x3026
%.rsp 0xff18
main() ...
```

...
• **Problem:** %rip points to the currently executing instruction. To call a function, we must remember the next caller instruction to resume at after.

• **Solution:** push the current value of %rip onto the stack. Then call the function. When it is finished, put this value back into %rip and continue executing.

*E.g. main() calls foo(*):

```
Stack

main()

%rip 0x3026

%rsp 0xff18
```

…
The **call** instruction pushes the value of %rip onto the stack and sets %rip to point to the beginning of the specified function’s instructions.

```assembly
call Label
call *Operand
```

The **ret** instruction pops the value of %rip from the stack and stores it in %rip.

```assembly
ret
```

The stored %rip value for a function is called its **return address**. It is the address of the instruction at which to resume the function’s execution. (not to be confused with **return value**, which is the value returned from a function).
To call a function in assembly, we must do a few things:

- **Pass Control** – `%rip` must be adjusted to execute the function being called and then resume the caller function afterwards.
- **Pass Data** – we must pass any parameters and receive any return value.
- **Manage Memory** – we must handle any space needs of the callee on the stack.

**Terminology:** caller function calls the callee function.
Plan For Today

• The Instruction Pointer (%rip)
• Calling Functions
  • The Stack
  • Passing Control
  • Passing Data
  • Local Storage
• Break: Announcements
• Register Restrictions
• Pulling it all together: recursion example
Parameters and Return

• There are special registers that store parameters and the return value.
• To call a function, we must put any parameters we are passing into the correct registers. (%rdi, %rsi, %rdx, %rcx, %r8, %r9, in that order)
• Parameters beyond the first 6 are put on the stack.
• If the caller expects a return value, it looks in %rax after the callee completes.
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4,
                      i1, i2, i3, i4);
    ...
}

int func(int *p1, int *p2, int *p3, int *p4,
          int v1, int v2, int v3, int v4) {
    ...
}
Parameters and Return

```c
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4,
                      i1, i2, i3, i4);
    ...
}

int func(int *p1, int *p2, int *p3, int *p4,
          int v1, int v2, int v3, int v4) {
    ...
}
```

```
0x40054f <+0>:    sub    $0x18,%rsp
0x400553 <+4>:    movl    $0x1,%c(%rsp)
0x40055b <+12>:   movl    $0x2,%8(%rsp)
0x400563 <+20>:   movl    $0x3,%4(%rsp)
0x40056b <+28>:   movl    $0x4,(%rsp)
```
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4, i1, i2, i3, i4);
    
}

int func(int *p1, int *p2, int *p3, int *p4, int v1, int v2, int v3, int v4) {
    
}
Parameters and Return

```c
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4, 
                      i1, i2, i3, i4);
    ...
}

int func(int *p1, int *p2, int *p3, int *p4, 
          int v1, int v2, int v3, int v4) {
    ...
}
```

Machine code:
```
0x40054f <+0>:    sub    $0x18,%rsp
0x400553 <+4>:    movl    $0x1,0xc(%rsp)
0x40055b <+12>:   movl    $0x2,0x8(%rsp)
0x400563 <+20>:   movl    $0x3,(%rsp)
0x40056b <+28>:   movl    $0x4,(%rsp)
0x400572 <+35>:   pushq   $0x4
```
Parameters and Return

```c
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4,
                        i1, i2, i3, i4);
    ...
}

int func(int *p1, int *p2, int *p3, int *p4,
          int v1, int v2, int v3, int v4) {
    ...
}
```

```
sub $0x18,%rsp
movl $0x1,0xc(%rsp)
movl $0x2,0x8(%rsp)
movl $0x3,0x4(%rsp)
movl $0x4(%rsp)
```
Parameters and Return

```c
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4,
                       i1, i2, i3, i4);
    ...
}

int func(int *p1, int *p2, int *p3, int *p4,
          int v1, int v2, int v3, int v4) {
    ...
}
```

```
int main(
    0xffe9fc 1
    0xffe9f8 2
    0xffe9f4 3
    0xffe9f0
)
```
main() {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4, i1, i2, i3, i4);
    ...
}

int func(int *p1, int *p2, int *p3, int *p4, int v1, int v2, int v3, int v4) {
    ...
}

0x40055b <+12>:    movl $0x2,0x8(%rsp)
0x400563 <+20>:    movl $0x3,0x4(%rsp)
0x40056b <+28>:    movl $0x4,(%rsp)
0x400572 <+35>:    pushq $0x4
0x400572 <+37>:    pushq &av2
Parameters and Return

```c
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4, i1, i2, i3, i4);
    ...
}

int func(int *p1, int *p2, int *p3, int *p4, int v1, int v2, int v3, int v4) {
    ...
}
```

```
0x00000000 <+20>:  movl  $0x3,0x4(%rsp)
0x00000008 <+28>:  movl  $0x4,(%rsp)
0x0000000c <+35>:  pushq $0x4
0x0000000e <+37>:  pushq $0x3
0x00000010 <+39>:  movq  $0x2,%r9d
0x00000014 <+45>:  movq  $0x1,%r8d
```
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4, i1, i2, i3, i4);
    ...
}

int func(int *p1, int *p2, int *p3, int *p4, int v1, int v2, int v3, int v4) {
    ...
}
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4, i1, i2, i3, i4);
    ...
}

int func(int *p1, int *p2, int *p3, int *p4, int v1, int v2, int v3, int v4) {
    ...
}
Parameters and Return

```c
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;

    int result = func(&i1, &i2, &i3, &i4, i1, i2, i3, i4);

    ...
}

int func(int *p1, int *p2, int *p3, int *p4, int v1, int v2, int v3, int v4) {
    ...
}
```

Disassembly:

```
0x400572 <+35>:  pushq  $0x4
0x400574 <+37>:  pushq  $0x3
0x400576 <+39>:  mov   $0x2,%r9d
0x40057c <+45>:  mov   $0x1,%r8d
```

Stack frame:

- `%rsp`
- `%rip`

 Addresses:

- 0x400572
- 0x400574
- 0x400576
- 0x40057c
- 0xffe9fc
- 0xffe9f8
- 0xffe9f4
- 0xffe9f0
- 0xffe9e8
- 0xffe9e0
- 0xffe9e0
- 0x400576
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4, i1, i2, i3, i4);
    ...
}

int func(int *p1, int *p2, int *p3, int *p4, int v1, int v2, int v3, int v4) {
    ...
}
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4,
                        i1, i2, i3, i4);
    ...
}

int func(int *p1, int *p2, int *p3, int *p4,
          int v1, int v2, int v3, int v4) {
    ...
}

Parameters and Return
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4, i1, i2, i3, i4);
    ...
}

int func(int *p1, int *p2, int *p3, int *p4, int v1, int v2, int v3, int v4) {
    ...
}
Parameters and Return

```c
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4,
        i1, i2, i3, i4);
    ...
}

int func(int *p1, int *p2, int *p3, int *p4,
    int v1, int v2, int v3, int v4) {
    ...
}
```

```
 mov $0x1,%r8d
 lea 0x10(%rsp),%rcx
 lea 0x14(%rsp),%rdx
 lea 0x18(%rsp),%rsi
```

```
0x00007c <+45>: mov $0x1,%r8d
0x000082 <+51>: lea 0x10(%rsp),%rcx
0x000087 <+56>: lea 0x14(%rsp),%rdx
0x00008c <+61>: lea 0x18(%rsp),%rsi
```
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4, i1, i2, i3, i4);
    ...
}

int func(int *p1, int *p2, int *p3, int *p4, int v1, int v2, int v3, int v4) {
    ...
}
Parameters and Return

```
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4, i1, i2, i3, i4);
    ...
}

int func(int *p1, int *p2, int *p3, int *p4, int v1, int v2, int v3, int v4) {
    ...
}
```

```
0x400587 <+56>:   lea    0x14(%rsp),%rdx
0x40058c <+61>:   lea    0x18(%rsp),%rsi
0x400591 <+66>:   lea    0x1c(%rsp),%rdi
0x400596 <+71>:   callq  0x400546 <func>
                  ...
```

%rip 0x400591
%rsp 0xffe9e0
%r9d 1
%r8d 2
%rcx 0xffe9f0
%rdx 0xffe9f4
%rdi 0xffe9fc
%rsi 0xffe9f8
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4, i1, i2, i3, i4);
    ...
}

int func(int *p1, int *p2, int *p3, int *p4,
         int v1, int v2, int v3, int v4) {
    ...
}

0x40058c <+61>:   lea    0x18(%%rsp),%%rsi
0x400591 <+66>:   lea    0x1c(%%rsp),%%rdi
0x400596 <+71>:   callq  0x400546 <func>
0x40059b <+76>:   add    $0x10,%%rsp
...
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4, i1, i2, i3, i4);
    ...}

int func(int *p1, int *p2, int *p3, int *p4, int v1, int v2, int v3, int v4) {
    ...
}
```c
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4, i1, i2, i3, i4);
    ...
}

int func(int *p1, int *p2, int *p3, int *p4, int v1, int v2, int v3, int v4) {
    ...
}
```

```assembly
0x40058c <+61>:   lea    0x18(%rsp),%rsi
0x400591 <+66>:   lea    0x1c(%rsp),%rdi
0x400596 <+71>:   callq  0x400546 <func>
0x40059b <+76>:   add    $0x10,%rsp
```
To call a function in assembly, we must do a few things:

• **Pass Control** – %rip must be adjusted to execute the function being called and then resume the caller function afterwards.

• **Pass Data** – we must pass any parameters and receive any return value.

• **Manage Memory** – we must handle any space needs of the callee on the stack.

Terminology: **caller** function calls the **callee** function.
Plan For Today

• The Instruction Pointer (%rip)
• Calling Functions
  • The Stack
  • Passing Control
  • Passing Data
  • Local Storage
• Break: Announcements
• Register Restrictions
• Pulling it all together: recursion example
Local Storage

• So far, we’ve often seen local variables stored directly in registers, rather than on the stack as we’d expect. This is for optimization reasons.

• There are **three** common reasons that local data must be in memory:
  • We’ve run out of registers
  • The ‘&’ operator is used on it, so we must generate an address for it
  • They are arrays or structs (need to use address arithmetic)
long swap_add(long *xp, long *yp) {
    long x = *xp;
    long y = *yp;
    *xp = y;
    *yp = x;
    return x + y;
}

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

• The Instruction Pointer (%rip)
• Calling Functions
  • The Stack
  • Passing Control
  • Passing Data
  • Local Storage

• Break: Announcements
• Register Restrictions
• Pulling it all together: recursion example
Plan For Today

• No lecture Friday
• None! Let’s take a breather 😊
Plan For Today

• The Instruction Pointer (%rip)
• Calling Functions
  • The Stack
  • Passing Control
  • Passing Data
  • Local Storage
• Break: Announcements
• Register Restrictions
• Pulling it all together: recursion example
Register Restrictions

• There is only one copy of registers for all programs and instructions.
• Therefore, there are some rules that callers and callees must follow when using registers so they do not interfere with one another.
• There are two types of registers: **caller-owned** and **callee-owned**
Register Restrictions

Caller Owned

• If a callee wants to use this, they must save the existing value and restore it back into the register after they are done.

• If a caller wants to use this, they can store values there during function calls and can assume that the values will be preserved.

Callee-Owned

• If a callee wants to use this, they can do so without worrying about overwriting data. They do not need to save the existing value.

• If a caller wants to use this, they must be careful when calling a function, as that function may overwrite the contents! They may want to save the existing value and restore it back into the register after the function call.
long P(long x, long y) {
    long u = Q(y);
    long v = Q(x);
    return u + v;
}

long P(long x, long y), x in %rdi, y in %rsi:
    push %rbp
    push %rbx
    mov %rdi,%rbp
    mov %rsi,%rdi
    callq 40056d <Q(long)>
    mov %rax,%rbx
    mov %rbp,%rdi
    callq 40056d <Q(long)>
    add %rbx,%rax
    pop %rbx
    pop %rbp
    retq
long P(long x, long y) {
    long u = Q(y);
    long v = Q(x);
    return u + v;
}

long P(long x, long y), x in %rdi, y in %rsi:
    push %rbp
    push %rbx
    mov %rdi,%rbp
    mov %rsi,%rdi
    callq 40056d <Q(long)>
    mov %rax,%rbx
    mov %rbp,%rdi
    callq 40056d <Q(long)>
    add %rbx,%rax
    pop %rbx
    pop %rbp
    retq
long P(long x, long y) {
    long u = Q(y);
    long v = Q(x);
    return u + v;
}

long P(long x, long y), x in %rdi, y in %rsi:

    push %rbp
    push %rbx
    mov %rdi,%rbp
    mov %rsi,%rdi
    callq 40056d <Q(long)>
    mov %rax,%rbx
    mov %rbp,%rdi
    callq 40056d <Q(long)>
    add %rbx,%rax
    pop %rbx
    pop %rbp
    retq
Plan For Today

• The Instruction Pointer (%rip)
• Calling Functions
  • The Stack
  • Passing Control
  • Passing Data
  • Local Storage
• Break: Announcements
• Register Restrictions
• Pulling it all together: recursion example
Example: Recursion

• Let’s take a look at an example of recursion at the assembly level.
• We’ll put to use everything we’ve learned about registers, the stack, function calls, parameters, and assembly instructions!
• We’ll also see how helpful GDB can be when tracing through assembly.
Recap

• The Instruction Pointer (%rip)
• Calling Functions
  • The Stack
  • Passing Control
  • Passing Data
  • Local Storage
• Break: Announcements
• Register Restrictions
• Pulling it all together: recursion example

That’s it for assembly! **Next time:** managing the heap