Activation records store:

- return values (sometimes)
- return addresses (always)
- parameters (sometimes)
- local variables (sometimes)
- caller-saved registers (sometimes)
Registers: %rax, %rbx, %rcx, %rdx, ...

In the ABI, registers are either caller or callee saved:

Callee: %rsp\(^1\), %rbp, %rbx, %r12 through %r15

Caller: All others

\(^1\)Somewhat by default.
How are activation records stored?

- That's the job of \%rbp.
- \%rbp is the ‘base pointer’, i.e. the beginning of the activation record for the current function.
What can we do with these registers?
There are many instructions (documentation spans 2,200+ pages)

Most of these we don’t care about

Ones we do fall generally into two categories:
  - Arithmetic and logic
  - Control flow

Ones we don’t care about: SIMD, I/O, privilege, memory, floating-point . . .
**Arithmetic and logic**

$imm = $immediate, $r = $register, $r/m = $either register or memory

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Instruction</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>movq $r/m, $r</td>
<td>movq $r, $r/m</td>
<td>movq $imm, $r/m</td>
</tr>
<tr>
<td>addq $r, $r/m</td>
<td>addq $r/m, $r</td>
<td>addq $imm, $r/m</td>
</tr>
<tr>
<td>subq $r, $r/m</td>
<td>subq $r/m, $r</td>
<td>subq $imm, $r/m</td>
</tr>
<tr>
<td>imulq $r/m, $r</td>
<td>idivq $r/m</td>
<td>negq $r/m</td>
</tr>
<tr>
<td>orq $r, $r/m</td>
<td>orq $r/m, $r</td>
<td>orq $imm, $r/m</td>
</tr>
<tr>
<td>andq $r, $r/m</td>
<td>andq $r/m, $r</td>
<td>andq $imm, $r/m</td>
</tr>
<tr>
<td>xorq $r, $r/m</td>
<td>xorq $r/m, $r</td>
<td>xorq $imm, $r/m</td>
</tr>
<tr>
<td>notq $r/m</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Examples:**

- `movq $32, %rax` | Copies 32 into %rax
- `imulq %rbx, %rax` | Computes %rax = %rax * %rbx
Control flow

\[ \text{imm} = \text{immediate}, \ r = \text{register}, \ r/m = \text{either register or memory} \]

<table>
<thead>
<tr>
<th>cmpq r/m, r</th>
<th>cmpq r, r/m</th>
<th>cmpq imm, r/m</th>
</tr>
</thead>
<tbody>
<tr>
<td>jmpq imm</td>
<td>jCCq imm</td>
<td>callq imm</td>
</tr>
<tr>
<td>ret</td>
<td>cmovCCq</td>
<td></td>
</tr>
</tbody>
</table>

Useful condition codes: \( e, \ ne, \ l, \ le, \ ge, \ g \). Where \( e = \text{equal}, \ n = \text{not}, \ l = \text{less-than}, \ g = \text{greater-than} \)

Examples:

- \( \text{cmpq } \$32, \ %\text{rcx} \): Compares \%rcx against 32
- \( \text{cmoveq } \$64, \ %\text{rcx} \): If last compare was equal, copy 64 into \%rcx
- \( \text{je label} \): If last compare was equal, jump to label. Otherwise, continue
And also ... stack manipulation

\[ \text{imm} = \text{immediate}, \ r = \text{register}, \ r/m = \text{either register or memory} \]

| pushq r/m | pushq imm | popq r/m |

Examples:

pushq %rcx    \quad \text{Pushes} \ %rcx \ \text{onto the stack;} \ %rcx \ \text{is untouched}
popq %rdx    \quad \text{Copies the top of stack into} \ %rdx, \ \text{and removes} \ %rdx \ \text{from the stack}
addq $8, %rsp \quad \text{Pop quiz!}
A note on notation

- You’ll often see instructions written as `mov $32, %rax`
  - The q is a size suffix, can usually be inferred
- You’ll also see instructions written as `mov rax, 32`
  - Two different syntaxes for x86 assembly
  - We’re using GNU syntax
Memory accesses

Can access memory...

At a constant address:

movq $10, 32
Address: 32

Using a register:

movq $11, (%rdi)
Address: %rdi

Using a register and offset:

movq $12, 32(%rdi)
Address: r%di + 32

Using a base and index register:

movq $13, (%rdi, %rdx, 8)
Address: %rdi + %rdx*8

...and an offset:

movq $14, 1024(%rdi, %rdx, 8)
Address: %rdi + %rdx*8 + 1024