

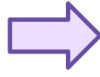


Assembly & Registers



The **compiler** converts our **C code** into low-level **assembly instructions**, which can then be converted into machine code (0s and 1s) run by your computer.

```
int gold = 10;
int treasure = 5;
gold += treasure;
```

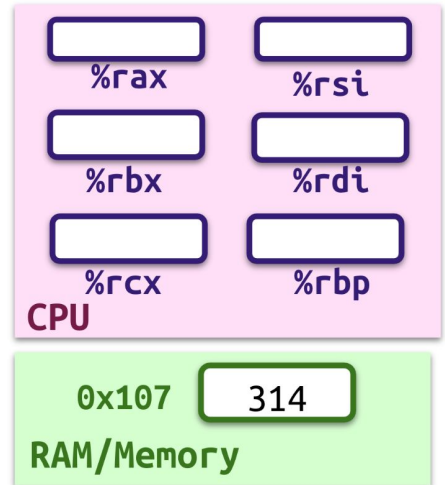


```
mov $10, %rax
mov $5, %rbx
addl %rbx, %rax
```

Think in terms of **registers**: variable boxes used as temporary storage / **scratch paper** for programs.

A register is an **very fast read/write slot** right on the CPU (not in memory), which can hold variable values

Move data in & out of registers to perform operations on it with the CPU, e.g., arithmetic operations, parameters into functions, return values



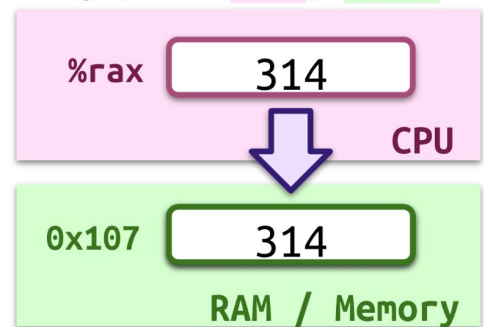
MOV Instruction



Key Instruction: mov src, dst

- Responsible for **copying data** around!
- src** can be an immediate (constant value), register, or memory location
- dst** can be a register or memory location
- Exception: src and dst cannot both be memory locations!** (must first temporarily copy the data into a register)

e.g., `mov %rax, 0x107`



Instructions like `mov` can **specify via a suffix the # of bytes** being moved, etc.

- `movb` for 1 byte being copied
- `movl` for "double word" (4 bytes)
- `movw` for "word" (2 bytes)
- `movq` for "quad word" (8 bytes)

Addressing Modes in Assembly



Immediate: `mov $0x5, dst`

A **dollar sign (\$)** indicates source is **constant value** to be moved into dst

Direct: `mov 0x106b, dst`

Fixed, static memory address

This reads from **Mem[0x106b]**, the memory at address 0x106b.

Indirect with Displacement:
`mov 8(%rax), dst`

Same as indirect, but has a **number prefix as bytes plus the address**, like pointer arithmetic!

The above reads from **Mem[\$rax + 8]**

Register: `mov $rax, dst`

Moves **value held in register** to dst.

Indirect: `mov ($rax), dst`

This reads from **Mem[\$rax]**, where the register \$rax holds an address, e.g., 0x106b

Note **parentheses** to contrast with above.

Indirect with Displacement + Scaled Index: `mov 8($rsp, $rcx, 4), dst`

More complex displacement: **offset + scaling factor** (e.g., 1,2,4,8), often to index into arrays.

Reads from **Mem[\$rsp + 8 + (4 * \$rcx)]**.



lea Instruction



Key Instruction: **lea** src, dest

- Similar to mov, **except that it doesn't dereference / look at memory**, i.e. it copies of the value of src itself to a register, not the data at the address src.
- It's really just pointer arithmetic! 🤔

%rax 0x100
%rcx 0x10
%rdx CPU

2002 2025
 0x100 0x140
 RAM/Memory

```
leaq (%rax, %rcx, 4), %rdx
```

%rdx 0x140 ← holds **%rax + 4 * %rcx**

lea doesn't dereference this address/look at memory!

```
movq (%rax, %rcx, 4), %rdx
```

%rdx 2025 ← holds **Mem[%rax + 4 * %rcx]**

Meanwhile, **mov** dereferences the address to get 2025!

Registers: Great Power = Great Responsibility

Many registers are given designated roles or responsibilities, e.g., for function calls:

- The first 4 arguments passed in are **\$rdi, \$rsi, \$rdx, \$rcx**, respectively
- The return value of a function (if any) is written to **\$rax** before it ends

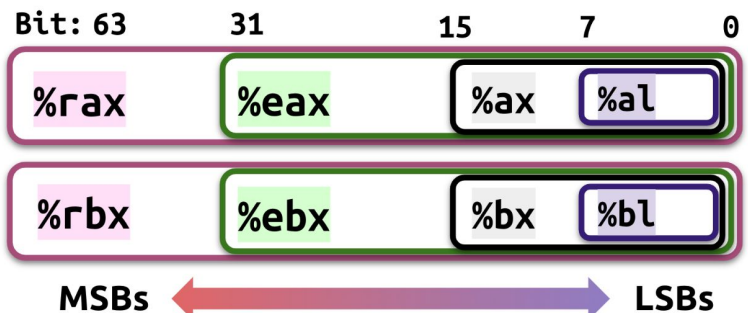


Registers Sizes & Subregisters

Registers can be **partially or fully referred to**. Example: we can access just **32 bits** of a register (e.g., **\$edi**) instead of all 64 bits (e.g., **\$rdi**), if we're operating on an **int**.

Specifically, a reference to the **e-named register** accesses the **lower 32-bit subregister**

%edi is subregister of **%rdi**
%eax is subregister of **%rax**



New GDB Commands for Assembly!



disassemble: prints the assembly for the currently executing function (can write *(gdb) disassemble func_name* to disassemble another function)

b *address: Places a breakpoint on a specific assembly instruction, via its address

stepi, nexti: Assembly versions of step, next, going one line of assembly at a time

info reg: Prints out the values in all integer registers

print \$register: Prints out a specific register's contents, e.g., `print $rax`

Note: Can print using a typecast, e.g., `(gdb) print (char*) ptr`, or in hex, via `print /x`

```
(gdb) p (char *)$rax
```

```
$3 = 0x400720 "Hello, world!\n"
```

```
(gdb) p *(long *)$rax
```

```
(gdb) p *(long *)((char *)$rsp + 8)
```

set \$register = value: manually set a register's contents, useful for testing behavior

In Terminal / myth, can run **objdump -d executable_name** to list all assembly instructions



Ben's 'Waterfall Method' of Reading Assembly

Ben Reads Assembly for Pleasure (totally)



General Guidelines:

(may have to adapt to specific assembly task!)

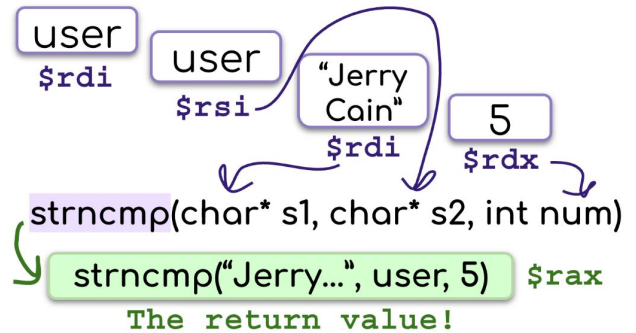
- ❑ Write each register value (or corresponding variable) in a separate, labeled box.
- ❑ Assembly is poetry, flowing top to bottom – for each line, write a new box lower than the previous one and indented to the right, with the updated value (don't cross out old boxes!).
- ❑ Treat function calls as a stanza break – write each function call explicitly, and draw arrows from the registers being passed in. Write a box for return `$rax` after, and reset indentation.

```

🐻 is_not_poohbear(user):
<+0> mov $rdi, $rsi
<+4> mov $0x106a106b, $rdi
<+8> mov $0x5, $rdx
<+12> call strncmp
<+16> ret

```

Small toy example! Assume `$0x106a106b` corresponds to the string "Jerry Cain".



Tips for Assembly & C Reverse-Engineering



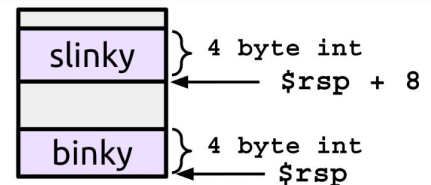
Try to understand what the assembly is doing certainly, but be more concerned with mapping your understanding to **equivalent C code**, and not necessarily the original.

Look for registers that alternate between being updated and being read (**think very active, lively registers**), because they typically correspond to key **local variables**.

"If it looks like a `char*`, swims like a `char*`, and quacks like a `char*`, it's probably a `char*`"

👉 Assembly operates directly on raw data/bits, but **context can help in determining type!**

📖 **Keep a record / sketch of the stack frame.** And when portions of it are written to or updated (e.g., a 4-byte sliver at address `$rsp` to store an integer variable there), **update your sketch accordingly!**



🧩 **Decomposition!** If you encounter a long af stream of assembly, use function calls to **segment the instructions into smaller, more manageable sections**. And when you see a loop 🌀, first aim to divide it into 3 or so parts: a **pre-loop, within-loop, and post-loop**.

Additional Tips: Loops

You can usually ignore all push and pop operations when reading, as they're almost always boilerplate in place to save & ultimately restore caller-owner register values.

When you suspect there's an **if test** or an **if/else test**:

- Code branches off, so look for where/how it branches, i.e. any **conditional jumps** and **unconditional jumps** to higher addresses
- Use that to demarcate what is part of the **if test**, the **if body**, and **potentially the body of a corresponding else branch**.

```

check condition
Jump to [IfBody] if condition true
[Else]:
  <If false statements>
  Jump to [EndIf]
[IfBody]:
  <If true statements>
[EndIf]

```

🧩 **More on for/while loops:** When you see one, **decompose it!**

- Look for the **conditional jump forward** and the **unconditional jump backwards** (can be the reverse! i.e. conditional jump back)
- Above case: **backwards jump keeps us in loop, forward jump exits**
- Use this to demarcate which instructions are **within the body of the loop**, which ones precede it, and **which ones come after it**.

```

[Initialize] (e.g., int i = 0)
[Test]:
  Check OPPOSITE of loop condition
  Jump to [LoopEnd] if true
[LoopBody]:
  <statements>
  <Update> (e.g., i++)
  Jump to [Test]
[LoopEnd]:
  everything else

```

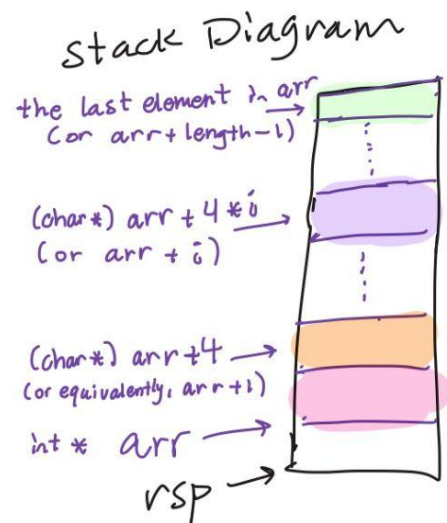
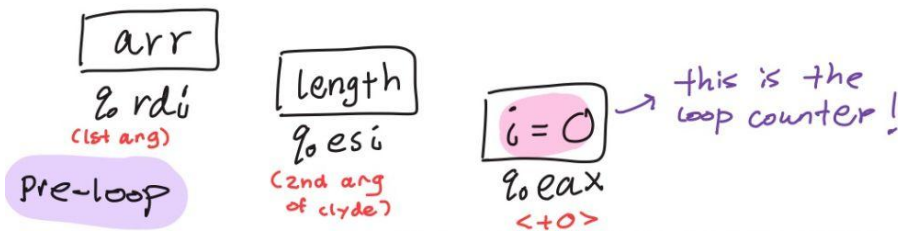


Sample waterfall trace on a loop that increments each element in an array by 1. Each box is labeled with its corresponding line of assembly, e.g., `<+0>`, and note pointer arithmetic in assembly works in terms of bytes offsets, hence the `(char*)` casts.

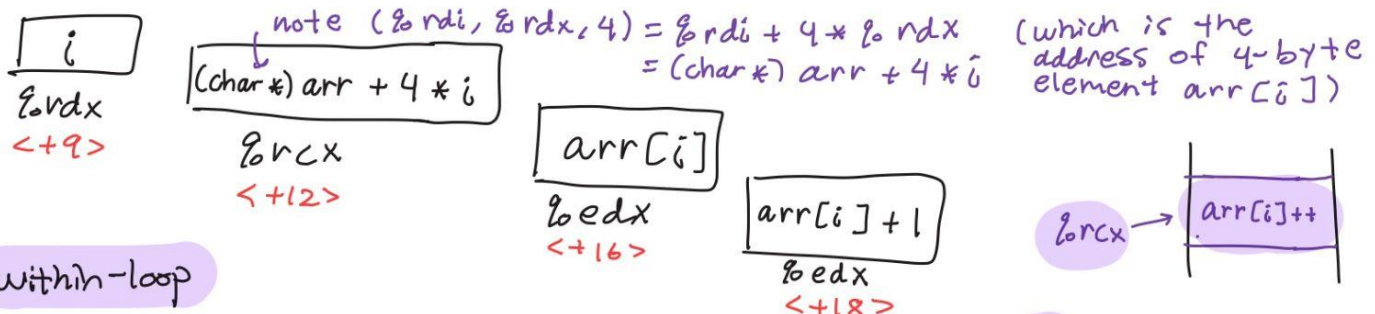
```

<+0>:  mov    $0x0,%eax
<+5>:  cmp     %esi,%eax           // task: fill in the blanks! :)
<+7>:  jge     0x401154 <clyde+28> int clyde(int arr[], int length) {
<+9>:  movslq  %eax,%rdx           for (int i = 0; i < length; i++) {
<+12>: lea     (%rdi,%rdx,4),%rcx   arr[i]++;
<+16>: mov     (%rcx),%edx
<+18>: add     $0x1,%edx
<+21>: mov     %edx,(%rcx)
<+23>: add     $0x1,%eax
<+26>: jmp     0x40113d <clyde+5>    }
<+28>: movslq  %esi,%rsi
<+31>: mov     -0x4(%rdi,%rsi,4),%eax
<+35>: ret

```



<+5> Compare `%esi` (`length`) with `%eax` (`i`) and with **<+7>** `jge`, if `i >= length`, jump to $\bullet$, otherwise if `i < length`, proceed from here.



<+21> The next line moves `%edx`, or `arr[i] + 1`, into `%rcx`, or `&arr[i]`

(increments loop counter!) `i + 1` $\rightarrow$ `%eax` `<+23>`

<+26> now, jump back to loop check $\blacksquare$

