x86-64 Hints

Svend Knudsen

Stuff to read

These are helpful to varying degrees.
x86_64 Quick Reference
x64 Cheat Sheet

Intel® 64 and IA-32 Architectures, Software Developer's Manual; Instruction Set Reference[1]
- This is as advanced as you get, but if you need to know everything about an instruction, this is the place to go. Keep in mind that this manual uses Intel syntax where we use GAS aka AT&T syntax where the operands are typically reversed.

Using Memory

For the things you will do in assembly you will generally have two methods of storing data: Registers and memory. You should already be familiar with registers, so I will go through how to use the stack and memory addresses to store data in and load from memory.

Memory Addresses

To access memory directly you need an address to a position in memory. If you have such an address you can use a regular **mov** instruction to move data between that position and a register. But the address itself is also a value, so if you just use **mov** regularly you will move the address. To instead access the data it points to, you put parentheses around it.
So if there is an address in %rax that references some data in memory, you can move the data into a register like this:

```
mov (%rax), %rbx
```

You can also move data from a register to memory:

```
mov %rbx, (%rax)
```

Allocating memory

So you might ask, where do we get these memory addresses from? (Because you can't just access any area of memory you want to.) There are two ways to go about this.
The first is to get the assembler to allocate a fixed amount of memory. To do this, you write a label under '.section .data':

```
.section .data

some_label_name:
  .space 1024
```

The '.space' directive tells the assembler “I want this much memory allocated when the program starts”, and then it handles the allocation. The label is then your way to access the address to the beginning of the allocated space. To move the address into a register you use '$':

```
mov $some_label_name, %rsi
```

You can also access the memory directly with parentheses like you do with an address in a register:

```
mov (some_label_name), %rax or
mov %rax, (some_label_name)
```

1 This is just the instruction set reference. You can find the entire manual on intel's website.

Also, when dealing with memory mov's, remember that you cannot mov data directly from a position in memory to another.

Under the .section .data you can also allocate space with something already in it. Specifically useful is the '.string' directive, which fills a space with a string.

```
label_name:
  .string “This string can for example be used with a write syscall\n”
```

Usually when you have allocated some space in the memory, you want to access more than just the beginning of it. To access another piece of the space you simply add a number to the address and that address then points to a position that many bytes further ahead in your space. So, if you have a space and you want to access a position 128 bytes after the beginning, all you do is:

```
mov $label, %r8
add $128, %r8
mov (%r8), %rax
```

You can also displace the address you access without changing the base address you are using. Write a constant outside the parentheses and that number is added to the value inside to get the actual address accessed:

```
mov 128(%r8), %rax
```

This method has more features and you can see those in the slides p. 63-64.

The other way to allocate memory is dynamically, usually through the sys_brk syscall, but you will probably get a function snippet for that purpose.

The Stack

There is also a stack available to your program. The primary ways to interact with this stack are the **push** and **pop** instructions.

`push %register` – moves data from a register to the top of the stack.
`pop %register` – moves data from the top of the stack to a register.

You can also access the stack directly with the *stack pointer register* %rsp. This register changes dynamically when you use push and pop, so generally don't mess with it. But sometimes it can be handy to access memory down the stack without popping the top.

Accessing less than 8 bytes

Sometimes you don't want to load 8 bytes from memory into a register. For example when working with strings, you have ASCII characters which are only 1 byte. So when you try to load a character into a register, you get the next 7 with it as well. Once again there are two ways to go about this.
movb only moves one byte at a time, but the destination register also has to be only one byte. You do this by accessing the least significant byte of a register with another name. See a list of these name in the link from above. For example:

```
mov $0, %rax
movb (%rsi), %al
```

Where %al is the least significant byte of %rax.

The other way to do it is with **movzx** (**mov**e **z**ero e**x**tended). This moves a byte and the adds zeros to it until it is a full quadword. Used like this:

```
movze (%rsi), %rax
```

This has the added effect of not needing to clean out the register beforehand.

System calls

List of x64 linux system calls

Sometimes you need to use OS functionality. In your cases it will most likely only be for memory handling and I/O. A system call is made by loading some values into specific registers and then using the `syscall` instruction. Here is an example of a *read* syscall:

```
mov $0, %rax
mov $0, %rdi
mov $buffer, %rsi
mov $20, %rdx
syscall
```

This system call makes the OS read 20 bytes of data from stdin and save it in memory starting at `$buffer`. Above you can see a list of syscalls, but the vast majority of those you will not need. Probably, the only ones you will use are read, write, open, fstat and brk. And the last two you will most likely get code snippets for. So that leaves the last 3:

Read
%rax: 0
%rdi: file descriptor[2]
%rsi: pointer to a buffer
%rdx: number of bytes to read

Copies data from the file to the given buffer.
Returns the number of bytes read in `%rax`.

Write
%rax: 1
%rdi: file descriptor
%rsi: pointer to a buffer
%rdx: number of bytes to write

Very similar to read, but the other way around, it writes *from* the buffer *to* the file.
Returns the number of bytes written in `%rax`.

Open
%rax: 2
%rdi: pointer to a string (the file name)
%rsi: flags[3]
%rdx: mode[4]

Returns a *file descriptor* for the open file in `%rax`.

Be advised: syscalls destroy the values in registers %r11 and %rcx, so be sure to save them if you need them.

2 A file descriptor is an integer that the OS uses to signify an open file for a process. *stdin*, *stdout* and *stderr* are also considered files, and their file descriptors are 0, 1 and 2 respectively.
3 There are a bunch of flags that you can use when opening a file, but we don't need to use any of them here, so just use 0.
4 There are 3 modes: read-only (0), write_only (1) and read-write (2). Usually you just use read-write.

Loops and jumps

Jumps are a very important part of assembly and are what you will use instead of *while*, *for* and *if* as you might be used to it. To use a jump, first make a `cmp` instruction of two values and then on the next line do the jump[5]. There are several types of jumps you can use:

`jmp <label>` An unconditional jump. Jumps to the label no matter what.
`je/jne <label>` Jumps to the label if the compared values are equal/not equal.
`jl/jg/jle/jge <label>` This is your full range of jump if <, >, <=, >=. WARNING: these jumps look at the compared values in the opposite order of what you would think. For example:

```
cmp %rax, %rbx
jl some_label
```

This will jump if *%rbx* is less then *%rax*, NOT the other way around.

There is also a **loop** instruction. It works like a jump but each time it jumps it decrements the value in `%rcx` and exits the loop when it reaches zero. So this:

```
loop_label:
do something
loop loop_label
```

is equivalent to

```
loop_label:
do something
dec %rcx
cmp $0, %rcx
je loop_label
```

5 Technically, most arithmetic instructions like **sub**, **add**, **dec** and **inc** set the same or some of the same flags as **cmp** does. But using **cmp** is probably easier to read/understand.