Easy x86-64

A tutorial on programming with the x86-64 in Assembly

View on GitHub Download .zip Download .tar.gz

"I'm the operator with my pocket calculator." -Kraftwerk

There has been much interest in assembly lately (whether the real 6502, or the fictional DCPU–16; I even created my own virtual 8-bit CPU called i808 in 2007), but none of this attention focuses on the architecture that is most popular in today's computers. If you are reading this on a desktop, laptop, or server then your computer is most likely using x86–64 (or x86). x86–64 is the 64-bit superset of the 32-bit x86 architecture and any modern CPU from AMD or Intel supports it. This document will focus on the most used parts of x86–64.

Assembly language is the lowest level of abstraction in computers that is still easily readable by humans. Assembly language translates directly to the bytes that are executed by your computer's processor.

Learning assembly is a useful exercise and will give you a deeper understanding of what takes place 'under the hood'. While the vast majority of programming is done via high-level languages such as C, C++, Java, etc., it is sometimes advantageous to write partial segments of code in assembly if execution speed is a high priority. For instance, code segments with heavy math calculations for 3D games or scientific processes stand to benefit significantly from the speedup that can be achieved with assembly.

In this document we will be using 'Intel' syntax instead of 'AT&T'. Therefore, opcodes that use multiple arguments work in the following form:

```
opcode destination, source
```

Any numbers with the prefix '0x' in x86–64 assembly language (and by extension, in this document) are in hexadecimal (hex) format. If you're not familiar with hex numbers, I recommend you read the Wikipedia article before beginning.

Registers

Registers are probably the most complicated part of the x86–64 architecture and the complications that arise from them are mainly due to the carry-over from the legacy 32-bit and 16-bit x86 architectures. x86–64 has 16 64-bit general purpose registers named R0 - R15. These registers can be broken down into separate parts by bit size and can also be referenced by their legacy x86 names. More information on register names and breakdowns can be found here.

For instance, R0 is a 64-bit register (also known as a quad word). If you only want to use 32 bits, then that section can be referenced by R0D (a double word), 16 bits by R0W (a word), or 8 bits by R0B (a byte).

These D, W, and B refereces are examples of carry-over from the 16-bit word days:

- 8 bits = 1 byte or 'halfword'
- 16 bits = 2 bytes = 1 word
- 32 bits = 4 bytes = 2 words = 1 double word
- 64 bits = 8 bytes = 4 words = 1 quad word

Further complications present themselves with certain opcodes depending on specific registers. This will be explored in more detail in the Multiplication and Division section.

Basic Operations

The most basic operations are assigning a value to a register or moving a value between two registers. In x86–64 this is called a move or **mov**. This terminology is misleading, as nothing is moved; it is merely copied or stored.

```
mov rax, 15                                     ; Store the value 15 in rax
mov rcx, rax                                ; Copy the value in rax to rcx
mov rbx, 18446744073709551615   ; Store the largest possible 64-bit number in rbx
```

Addition and Subtraction

We can **add** specific registers together:

```
mov rax, 11                             ; Store the value 11 in rax
mov rcx, 500                    ; Store the value 500 in rcx
add rax, rcx                    ; Add the value in rcx to rax
```

We can also **add** a value to a register:

```
mov rax, 25                             ; Store the value 25 in rax
add rax, 12                             ; Add 12 to rax; rax now contains 37
```

We can **sub**tract the value of one register from another:

```
mov r15, 1337                   ; Store the value 1337 in r15
mov r12, 55                             ; Store the value 55 in r12
sub r15, r12                    ; Subtract the value in r12 from r15
```

We can also **sub**tract a value from a register:

```
mov rcx, 123                    ; Store the value 123 in rax
sub rcx, 24                             ; Subtract 24 from rcx; rcx now contains 99
```

Additions and subtractions can be used with any of the available registers.

Multiplication and Division

In this section we will be using the **mul** and **div** opcodes. These operations are more complicated and highlight the unique purposes of several registers.

```
mov rax, 50                             ; Store the value 50 in rax
mov rcx, 12                             ; Store the value 12 in rcx
mul rcx                                 ; Multiply rax by rcx. In this case rax will be set to 600
```

The initial number must be stored in rax. rax can be multiplied by a value in any of the other registers. The result will be stored in rdx:rax.

```
mov rax, 800                    ; Store the value 800 in rax
mov rdx, 0                              ; Clear rdx to 0
mov rbx, 100                    ; Store the value 100 in rbx
div rbx                                 ; Divide rdx:rax by rbx. In this case rdx will be set to 0, and rax will
```

Registers rdx:rax must hold the dividend, while any other register can hold the divisor. After the div opcode executes, the quotient is stored in rax and the remainder in rdx.

Branching

Branching allows us to redirect the program flow based on certain conditions. These conditions can be checked using comparisons.

Comparisons allow us to compare the content of two registers and the system flags will be set depending on the result of the comparison. We can then change the code execution based on these system flags.

Let's try something like a simple C 'for' loop.

```
        mov rax, 0                      ; Set rax to 0
increment_loop:
        add rax, 1                      ; Add 1 to rax
        cmp rax, 10                     ; Compare the value in rax to 10
        jne increment_loop      ; If they are not equal then jump to increment_loop
```

The above code will loop 10 times. jne refers to 'Jump if Not Equal'. This means the execution will jump back to 'increment_loop' if rax does not contain the value 10. There are many other jump commands:

- jmp - JuMP - A direct jump without looking at the system flags
- je - Jump if Equal
- jne - Jump if Not Equal
- jl - Jump if Less
- jle - Jump if Less or Equal
- jg - Jump if Greater
- jge - Jump if Greater or Equal

Another kind of branch is a function call. A function call allows us to jump to a specific section of code that will return us to where we left off when the fuction call is completed.

```
        mov rax, 14                                     ; Set rax to 14
        mov rcx, 23                                     ; Set rcx to 23
        call add_and_subtract_one       ; Call the function
        cmp rax, 5
        je test_function_sucess         ; If rax == 5 then jump, if not then continue to next line

        ...

add_and_subtract_one:                   ; Function to add rcx to rax and then subtract 1
        add rax, rcx
        sub rax, 1
        ret
```

Accessing Memory

The registers can be used to read from and write to system memory. The mov opcode is used in a similar manner as we have seen earlier. Instead of providing a literal value we can use a memory address that is encapsulated in [square brackets].

```
mov rax, [0x200000]             ; Copy a 64-bit value from memory address 0x200000 to rax
mov [0x402000], rbx             ; Copy a 64-bit value from rbx to memory address 0x402000
```

The Stack

The stack is an area of memory used for storing temporary information. A stack is a last in, first out (LIFO) data structure. The push operation adds to the top of the list and the pop operation removes an item from the top of the list. If you were to push the numbers 5, 7, and 15 onto the stack, you would pop them out as 15 first, then 7, and lastly 5. In assembly, you can push registers onto the stack and pop them out later - this ability is useful when you want to save the value of a register while utilizing that register for another purpose.

```
mov rax, 25                             ; Store the value 25 in rax
push rax                                ; Push the value in rax to the stack
mov rax, 12                             ; Store the value 12 in rax
pop rax                                 ; Pop the first value in the stack to rax. In this case rax is set to 25
```

There is no requirement to push and pop to/from the same register. For instance, both of these segments have the same result:

```
mov rcx, rax                    ; Copy the value in rax to rcx

push rax                                ; Push the value in rax to the stack
pop rcx                                 ; Pop the first value in the stack to rcx
```

Further Reading

This document only scrapes the surface of the opcodes and functionality that is available with the x86-64 architecture.

Intel Software Developers Manuals can be found on their website. AMD manuals can be found on their website.

Shameless plug Usage of x86-64 assembly can be seen in BareMetal OS (Source Code), which was written by myself entirely in assembly.

// EOF