x86-64 Assembly

Computers execute machine code, which is encoded as bytes, to carry out tasks on a computer. Since different computers have different processors, the machine code executed on these computers is specific to the processor. In this case, we'll be looking at the Intel x86-64 instruction set architecture which is most commonly found today. Machine code is usually represented by a more readable form of the code called assembly code. This machine is code is usually produced by a compiler, which takes the source code of a file, and after going through some intermediate stages, produces machine code that can be executed by a computer. Without going into too much detail, Intel first started out by building 16-bit instruction set, followed by 32 bit, after which they finally created 64 bit. All these instruction sets have been created for backward compatibility, so code compiled for 32 bit architecture will run on 64 bit machines. As mentioned earlier, before an executable file is produced, the source code is first compiled into assembly(.s files), after which the assembler converts it into an object program(.o files), and operations with a linker finally make it an executable.

The best way to actually start explaining assembly is by diving in. We'll be using radare2 to do this - radare2 is a framework for reverse engineering and analysing binaries. It can be used to disassemble binaries(translate machine code to assembly, which is actually readable) and debug said binaries(by allowing a user to step through the execution and view the state of the program). Download r2 from [here](#).

The first step is to execute the program intro by running
./file1

```
ashu@ashu-Inspiron-5379 ~/D/t/c/christmas-re> ./file1
the value of a is 4, the value of b is 5 and the value of c is 9
```

The above program shows that there are 3 variables(a, b, c) where c is the sum of a and b.

Time to see what's happening under the hood! Run the command
r2 -d ./file1

This will open the binary in debugging mode. Once the binary is open, one of the first things to do is ask r2 to analyze the program, and this can be done by typing in: aa

Which is the most common analysis command. It analyses all symbols and entry points in the executable.

The analysis in this case involves extracting function names, flow control information and much more! r2 instructions are usually based on a single character, so it is easy to get more information about the commands.

For general help, run:
?
For more specific information, for example, about analysis, run
a?

Once the analysis is complete, you would want to know where to start analysing from - most programs have an entry point defined as main. To find a list of the functions run:

afl

```
[0x00400a30]> afl | grep main

0x00400b4d    1 68           sym.main
0x00400e10  114 1657         sym.__libc_start_main
0x00403870  346 6038 -> 5941 sym._nl_find_domain
0x00415fe0    1 43           sym._IO_switch_to_main_get_area
0x0044cf00    1 8            sym._dl_get_dl_main_map
0x00470520    1 49           sym._IO_switch_to_main_wget_area
0x0048fae0    7 73   -> 69   sym._nl_finddomain_subfreeres
0x0048fb30   16 247  -> 237  sym._nl_unload_domain
```

Note that memory addresses may be different on your computer.

As seen here, there actually is a function at main. Let's examine the assembly code at main by running the command
pdf @main
Where pdf means print disassembly function. Doing so will give us the following view

```
[0x00400a30]> pdf @main
            ;-- main:
/ (fcn) sym.main 68
|   sym.main (int argc, char **argv, char **envp);
|           ; var int local_ch @ rbp-0xc
|           ; var int local_8h @ rbp-0x8
|           ; var int local_4h @ rbp-0x4
|           ; DATA XREF from entry0 (0x400a4d)
|           0x00400b4d      55             pushq %rbp
|           0x00400b4e      4889e5         movq %rsp, %rbp
|           0x00400b51      4883ec10       subq $0x10, %rsp
|           0x00400b55      c745f4040000.  movl $4, local_ch
|           0x00400b5c      c745f8050000.  movl $5, local_8h
|           0x00400b63      8b55f4         movl local_ch, %edx
|           0x00400b66      8b45f8         movl local_8h, %eax
|           0x00400b69      01d0           addl %edx, %eax
|           0x00400b6b      8945fc         movl %eax, local_4h
|           0x00400b6e      8b4dfc         movl local_4h, %ecx
|           0x00400b71      8b55f8         movl local_8h, %edx
|           0x00400b74      8b45f4         movl local_ch, %eax
|           0x00400b77      89c6           movl %eax, %esi
|           0x00400b79      488d3d881409.  leaq str.the_value_of_a_is__d__the_value_of_b_is__d_and_the_value_of_c_is__d, %rdi ; 0x492008 ; "the value of a is %d, the value of b is %d and the value of c is %d"
|           0x00400b80      b800000000     movl $0, %eax
|           0x00400b85      e8f6ea0000     callq sym.__printf
|           0x00400b8a      b800000000     movl $0, %eax
|           0x00400b8f      c9             leave
\           0x00400b90      c3             retq
```

The core of assembly language involves using registers to do the following:

- Transfer data between memory and register, and vice versa
- Perform arithmetic operations on registers and data
- Transfer control to other parts of the program

Since the architecture is x86-64, the registers are 64 bit and Intel has a list of 16 registers:

64 bit	32 bit
%rax	%eax
%rbx	%ebx
%rcx	%ecx

%rdx	%edx
%rsi	%esi
%rdi	%edi
%rsp	%esp
%rbp	%ebp
%r8	%r8d
%r9	%r9d
%r10	%r10d
%r11	%r11d
%r12	%r12d
%r13	%r13d
%r14	%r14d
%r15	%r15d

Even though the registers are 64 bit, meaning they can hold up to 64 bits of data, other parts of the registers can also be referenced. In this case, registers can also be referenced as 32 bit values as shown. What isn't shown is that registers can be referenced as 16 bit and 8 bit(higher 4 bit and lower 4 bit).

The first 6 registers are known as general purpose registers while %rsp and %rbp are special purpose and their meaning will be explained later on. To move data using registers, the following instruction is used:
movq source, destination
This involves:

- Transferring constants(which are prefixed using the *$* operator) e.g. *movq $3 rax* would move the constant 3 to the register
- Transferring values from a register e.g. *movq %rax %rbx* which involves moving value from rax to rbx
- Transferring values from memory which is shown by putting registers inside brackets e.g. *movq %rax (%rbx)* which means move value stored in %rax to memory location represented by %rbx.

The last letter of the mov instruction represents the size of the data:

Intel Data Type	Suffix	Size(bytes)
Byte	b	1
Word	w	2
Double Word	l	4
Quad Word	q	8

Quad Word	q	8
Single Precision	s	4
Double Precision	l	8

When dealing with memory manipulation using registers, there are other cases to be considered:

- (Rb, Ri) = MemoryLocation[Rb + Ri]
- D(Rb, Ri) = MemoryLocation[Rb + Ri + D]
- (Rb, Ri, S) = MemoryLocation(Rb + S * Ri]
- D(Rb, Ri, S) = MemoryLocation[Rb + S * Ri + D]

Some other important instructions are:

- *leaq source, destination:* this instruction sets destination to the address denoted by the expression in source
- *addq source, destination:* destination = destination + source
- *subq* source, destination: destination = destination - source
- *imulq source, destination:* destination = destination * source
- *salq source, destination:* destination = destination << source where << is the left bit shifting operator
- *sarq source, destination*: destination = destination >> source where >> is the right bit shifting operator
- *xorq source, destination*: destination = destination XOR source
- *andq source, destination*: destination = destination & source
- *orq source, destination*: destination = destination | source

Now let's actually walk through the assembly code to see what the instructions mean when combined.

```
|   sym.main (int argc, char **argv, char **envp);
|           ; var int local_ch @ rbp-0xc
|           ; var int local_8h @ rbp-0x8
|           ; var int local_4h @ rbp-0x4
|           ; DATA XREF from entry0 (0x400a4d)
|           0x00400b4d      55             pushq %rbp
|           0x00400b4e      4889e5         movq %rsp, %rbp
|           0x00400b51      4883ec10       subq $0x10, %rsp
|           0x00400b55      c745f4040000.  movl $4, local_ch
|           0x00400b5c      c745f8050000.  movl $5, local_8h
|           0x00400b63      8b55f4         movl local_ch, %edx
|           0x00400b66      8b45f8         movl local_8h, %eax
|           0x00400b69      01d0           addl %edx, %eax
|           0x00400b6b      8945fc         movl %eax, local_4h
|           0x00400b6e      8b4dfc         movl local_4h, %ecx
|           0x00400b71      8b55f8         movl local_8h, %edx
|           0x00400b74      8b45f4         movl local_ch, %eax
|           0x00400b77      89c6           movl %eax, %esi
|           0x00400b79      488d3d881409.  leaq str.the_value_of_a_is
|           0x00400b80      b800000000     movl $0, %eax
|           0x00400b85      e8f6ea0000     callq sym.__printf
|           0x00400b8a      b800000000     movl $0, %eax
|           0x00400b8f      c9             leave
\           0x00400b90      c3             retq
```

The line starting with *sym.main* indicates we're looking at the main function. The next 3 lines are used to represent the variables stored in the function. The second column indicates that they are integers(*int),* the 3rd column specifies the name that r2 uses to reference them and the 4th column shows the actual memory location.

The first 3 instructions are used to allocate space on that stack(ensure that there's enough room for variables to be allocated and more). We'll start looking at the program from the 4th instruction(*movl $4*). We want to analyse the program while it runs and the best way to do this is using breakpoints. A breakpoint specifies where the program should stop executing. This is useful as it allows us to look at the state of the program at that particular point. So let's set a breakpoint using the command
db address
in this case, it would be
db 0x00400b55

To ensure the breakpoint is set, we run the *pdf @main* command again and see a little *b* next to the instruction we want to stop at

```
[0x00400a30]> pdf @main
            ;-- main:
/ (fcn) sym.main 68
|   sym.main (int argc, char **argv, char **envp);
|           ; var int local_ch @ rbp-0xc
|           ; var int local_8h @ rbp-0x8
|           ; var int local_4h @ rbp-0x4
|           ; DATA XREF from entry0 (0x400a4d)
|           0x00400b4d      55             pushq %rbp
|           0x00400b4e      4889e5         movq %rsp, %rbp
|           0x00400b51      4883ec10       subq $0x10, %rsp
|           0x00400b55 b    c745f4040000.  movl $4, local_ch
```

Now that we've set a breakpoint, let's run the program using
dc

```
[0x00400a30]> dc
hit breakpoint at: 400b55
[0x00400b55]> pdf
            ;-- main:
            ;-- rax:
/ (fcn) sym.main 68
|   sym.main (int argc, char **argv, char **envp);
|           ; var int local_ch @ rbp-0xc
|           ; var int local_8h @ rbp-0x8
|           ; var int local_4h @ rbp-0x4
|           ; DATA XREF from entry0 (0x400a4d)
|           0x00400b4d      55             pushq %rbp
|           0x00400b4e      4889e5         movq %rsp, %rbp
|           0x00400b51      4883ec10       subq $0x10, %rsp
|           ;-- rip:
|           0x00400b55 b    c745f4040000.  movl $4, local_ch
```

Running *dc* will execute the program until we hit the breakpoint. Once we hit the breakpoint and print out the main function, the *rip* which is the current instruction shows where execution has stopped. From the notes above, we know that the *mov* instruction is used to transfer values. This statement is transferring the value 4 into the *local_ch* variable. To view the contents of the *local_ch* variable, we use the following instruction
px @memory-address

In this case, the corresponding memory address for *local_ch* will be *rbp-0xc*(from the first few lines of @pdf main)
This instruction prints the values of memory in hex:

```
[0x00400b55]> px @ rbp-0xc
- offset -       0 1  2 3  4 5  6 7  8 9  A B  C D  E F  0123456789ABCDEF
0x7ffc914f7bc4  0000 0000 1890 6b00 0000 0000 7018 4000  ......k.....p.@.
0x7ffc914f7bd4  0000 0000 1911 4000 0000 0000 0000 0000  ......@.........
0x7ffc914f7be4  0000 0000 0000 0000 0100 0000 f87c 4f91  .............|O.
0x7ffc914f7bf4  fc7f 0000 4d0b 4000 0000 0000 0000 0000  ....M.@.........
0x7ffc914f7c04  0000 0000 0600 0000 8e00 0000 8000 0000  ................
0x7ffc914f7c14  0a00 0000 0000 0000 0000 0000 0000 0000  ................
0x7ffc914f7c24  0000 0000 0000 0000 0000 0000 0000 0000  ................
0x7ffc914f7c34  0000 0000 0000 0000 0000 0000 0004 4000  ..............@.
0x7ffc914f7c44  0000 0000 52db fe41 3933 915f 1019 4000  ....R..A93._..@.
0x7ffc914f7c54  0000 0000 0000 0000 0000 0000 1890 6b00  ..............k.
0x7ffc914f7c64  0000 0000 0000 0000 0000 0000 52db de86  ............R...
0x7ffc914f7c74  2711 68a0 52db 8a50 3933 915f 0000 0000  '.h.R..P93._....
0x7ffc914f7c84  0000 0000 0000 0000 0000 0000 0000 0000  ................
0x7ffc914f7c94  0000 0000 0000 0000 0000 0000 0000 0000  ................
0x7ffc914f7ca4  0000 0000 0000 0000 0000 0000 0000 0000  ................
0x7ffc914f7cb4  0000 0000 0000 0000 0000 0000 0000 0000  ................
```

This shows that the variable currently doesn't have anything stored in it(it's just *0000)*. Let's execute this instruction and go to the next one using the following command(which only goes to the next instruction)
ds
If we view the memory location after running this command, we get the following:

```
[0x00400b55]> px @ rbp-0xc
- offset -       0 1  2 3  4 5  6 7  8 9  A B  C D  E F  0123456789ABCDEF
0x7ffc914f7bc4  0400 0000 1890 6b00 0000 0000 7018 4000  ......k.....p.@.
0x7ffc914f7bd4  0000 0000 1911 4000 0000 0000 0000 0000  ......@.........
0x7ffc914f7be4  0000 0000 0000 0000 0100 0000 f87c 4f91  .............|O.
0x7ffc914f7bf4  fc7f 0000 4d0b 4000 0000 0000 0000 0000  ....M.@.........
0x7ffc914f7c04  0000 0000 0600 0000 8e00 0000 8000 0000  ................
0x7ffc914f7c14  0a00 0000 0000 0000 0000 0000 0000 0000  ................
0x7ffc914f7c24  0000 0000 0000 0000 0000 0000 0000 0000  ................
0x7ffc914f7c34  0000 0000 0000 0000 0000 0000 0004 4000  ..............@.
0x7ffc914f7c44  0000 0000 52db fe41 3933 915f 1019 4000  ....R..A93._..@.
0x7ffc914f7c54  0000 0000 0000 0000 0000 0000 1890 6b00  ..............k.
0x7ffc914f7c64  0000 0000 0000 0000 0000 0000 52db de86  ............R...
0x7ffc914f7c74  2711 68a0 52db 8a50 3933 915f 0000 0000  '.h.R..P93._....
0x7ffc914f7c84  0000 0000 0000 0000 0000 0000 0000 0000  ................
0x7ffc914f7c94  0000 0000 0000 0000 0000 0000 0000 0000  ................
0x7ffc914f7ca4  0000 0000 0000 0000 0000 0000 0000 0000  ................
0x7ffc914f7cb4  0000 0000 0000 0000 0000 0000 0000 0000  ................
[0x00400b55]>
```

We can see that the first 2 bytes have the value 4! If we do the same process for the next instruction, we'll see that the variable *local_8h* has the value 5.

```
[0x00400b55]> px @ rbp-0x8
- offset -       0 1  2 3  4 5  6 7  8 9  A B  C D  E F  0123456789ABCDEF
0x7ffc914f7bc8  0500 0000 0000 0000 7018 4000 0000 0000  ........p.@.....
0x7ffc914f7bd8  1911 4000 0000 0000 0000 0000 0000 0000  ..@.............
0x7ffc914f7be8  0000 0000 0100 0000 f87c 4f91 fc7f 0000  .........|O.....
0x7ffc914f7bf8  4d0b 4000 0000 0000 0000 0000 0000 0000  M.@.............
0x7ffc914f7c08  0600 0000 8e00 0000 8000 0000 0a00 0000  ................
0x7ffc914f7c18  0000 0000 0000 0000 0000 0000 0000 0000  ................
0x7ffc914f7c28  0000 0000 0000 0000 0000 0000 0000 0000  ................
0x7ffc914f7c38  0000 0000 0000 0000 0004 4000 0000 0000  ..........@.....
0x7ffc914f7c48  52db fe41 3933 915f 1019 4000 0000 0000  R..A93._..@.....
0x7ffc914f7c58  0000 0000 0000 0000 1890 6b00 0000 0000  ..........k.....
0x7ffc914f7c68  0000 0000 0000 0000 52db de86 2711 68a0  ........R...'.h.
0x7ffc914f7c78  52db 8a50 3933 915f 0000 0000 0000 0000  R..P93._........
0x7ffc914f7c88  0000 0000 0000 0000 0000 0000 0000 0000  ................
0x7ffc914f7c98  0000 0000 0000 0000 0000 0000 0000 0000  ................
0x7ffc914f7ca8  0000 0000 0000 0000 0000 0000 0000 0000  ................
0x7ffc914f7cb8  0000 0000 0000 0000 0000 0000 0000 0000  ................
```

If we go to the instruction *movl local_8h, %eax,* we know from the notes that this moves the value from local_8h to the %eax register. To see the value of the %eax register, we can use the command: *dr*

```
[0x00400b55]> dr
rax = 0x00400b4d
rbx = 0x00400400
rcx = 0x0044ba90
rdx = 0x00000004
r8 = 0x00000000
r9 = 0x00000007
r10 = 0x00000002
r11 = 0x00000001
r12 = 0x00401910
r13 = 0x00000000
r14 = 0x006b9018
r15 = 0x00000000
rsi = 0x7ffc914f7cf8
rdi = 0x00000001
rsp = 0x7ffc914f7bc0
rbp = 0x7ffc914f7bd0
rip = 0x00400b66
rflags = 0x00000206
orax = 0xffffffffffffffff
```

If we execute the instruction and run the *dr* command again, we get:

```
[0x00400b55]> dr
rax = 0x00000005
rbx = 0x00400400
rcx = 0x0044ba90
rdx = 0x00000004
r8 = 0x00000000
r9 = 0x00000007
```

This technically skips the previous instruction *movl local_ch, %edx* but the same process can be applied with it.

Showing the value of rax(the 64 bit version) to be 5. We can do the same for similar instructions and view the values of the registers changing.

When we come to the *addl %edx, %eax,* we know that this will add the values in edx and eax and store them in eax. Running *dr* shows us the rax contains 5 and rdx contains 4, so we'd expect rax to contain 9 after the instruction is executed

```
[0x00400b55]> dr
rax = 0x00000005
rbx = 0x00400400
rcx = 0x0044ba90
rdx = 0x00000004
```

Executing *ds* to move to the next instruction then executing *dr* to view register variable shows us the we are correct

```
[0x00400b55]> dr
rax = 0x00000009
rbx = 0x00400400
```

The next few instructions involve moving the values in registers to the variables and vice versa

```
;-- rip:
0x00400b6b      8945fc        movl %eax, local_4h
0x00400b6e      8b4dfc        movl local_4h, %ecx
0x00400b71      8b55f8        movl local_8h, %edx
0x00400b74      8b45f4        movl local_ch, %eax
0x00400b77      89c6          movl %eax, %esi
```

```
0x00400b79    488d3d881409.   leaq str.the_value_of_a_is__d__the_value_of_b_is__d_and_the_value_of_c_is__d, %r
0x00400b80    b800000000      movl $0, %eax
0x00400b85    e8f6ea0000      callq sym.__printf
0x00400b8a    b800000000      movl $0, %eax
0x00400b8f    c9              leave
0x00400b90    c3              retq
```

After that, a string(which is the output is loaded into a register and the *printf* function is called in the 3rd line. The second line clears the value of *eax* as *eax* is sometimes used to store results from functions. The 4th line clears the value of *eax.* The 5th and 6th line are used to exit the main function.

The general formula for working through something like this is:

- set appropriate break points
- use *ds* to move through instructions and check the values of register and memory
- if you make a mistake, you can always reload the program using the *ood* command