**The mov instruction**

The mov instruction is the most important instruction used in assembly in Linux, and we used it in all the previous examples.

The mov instruction is used to move data between registers, and between registers and memory.

Let's try to move data between registers and memory:

global \_start

section .text

\_start:

mov al, [mem1]

mov bx, [mem2]

mov ecx, [mem3]

mov rdx, [mem4]

mov rax, 60

mov rdi, 0

syscall

section .data

mem1: db 0x12

mem2: dw 0x1234

mem3: dd 0x12345678

mem4: dq 0x1234567891234567

*In mov al, [mem1], the brackets mean move the contents of mem1 to al. If we use mov*

*al, mem1 without brackets, it will move the pointer of mem1 to al.*

Now, let's talk about moving data from register to memory. Take a look at the following code:

global \_start

section .text

\_start:

mov al, 0x34

mov bx, 0x5678

mov byte [mem1], al

mov word [mem2], bx

mov rax, 60

mov rdi, 0

syscall

section .data

mem1: db 0x12

mem2: dw 0x1234

mem3: dd 0x12345678

mem4: dq 0x1234567891234567

Now, let's talk about moving data from register to memory. Take a look at the following code:

global \_start

section .text

\_start:

mov al, 0x34

mov bx, 0x5678

mov byte [mem1], al

mov word [mem2], bx

mov rax, 60

mov rdi, 0

syscall

section .data

mem1: db 0x12

mem2: dw 0x1234

mem3: dd 0x12345678

mem4: dq 0x1234567891234567

**Data swapping**

global\_start

section .text

\_start:

mov rax, 0x1234

mov rbx, 0x5678

xchg rax, rbx

mov rcx, 0x9878

xchg rcx, [mem1]

mov rax, 60

mov rdi, 0

syscall

section .data

mem1: dw 0x1234

**Load effective address**

global\_start

section .text

\_start:

lea rax, [mem1]

lea rbx, [rax]

mov rax, 60

mov rdi, 0

syscall

section .data

mem1: dw 0x1234

Both are now pointing at mem1, which contains 0x1234.

**Arithmetic operations**

global\_start

section .text

\_strat:

mov rax,0x1

add rax, 0x2

mov rbx,0x3

add bl, byte [mem1]

mov rcx, 0x9

sub rcx, 0x1

mov dl, 0x5

sub byte [mem2],dl

mov rax, 60

mov rdi, 0

syscall

section .data

mem1: db 0x2

mem2: db 0x9

Addicion with carry flag and subtraction with borrow

global\_start

section .text

\_start:

mov rax, 0x5

stc

add rax, 0x1

mov rbx, 0x5

stc

sub rbx, 0x1

mov rax,60

mov rdi, 0

syscall

section .data

**Increment and decrement operations**

global\_start

section .text

\_start:

mov rax, 0x5

inc rax

inc rax

mov rbx, 0x6

dec rbx

dec rbx

mov rax, 60

mov rdi, 0

syscall

section .data

**Loops**

Now, we are going to talk about loops in assembly. Like in any other high-level language (Python, Java, and so on), we can use loops for iteration using the RCX register as a counter, then the loop keyword.

global\_start

section .text

\_start:

mov rcx, 0x5

mov rbx, 0x1

increment:

inc rbx

loop increment

mov rax, 60

mov rdi, 0

section .data

In the previous code, we wanted to increment the contents of RBX five times, so we moved 0x5 to the rcx register, then moved 0x1 to the rbx register

Then, we called loop increment, which will decrement the contents of the RCX register and then go to start again from the increment tag:

So, we have to find a way to save the RCX register, such as saving it in the stack.

global\_start

section .text

\_start:

mov rcx, 0x5

increment:

push rcx

mov rax, 1

mov rdi, 1

mov rsi, hello

mov rdx, length

syscall

pop rcx

loop increment

mov rax, 60

mov rdi, 0

syscall

section .data

hello: db ‘Hello There!’,0xa

length: equ $-hello

**Controlling the flow**

Now, let's start with the unconditional jump:

global\_start

section .text

\_start:

jmp exit\_ten

mov rax, 60

mov rdi, 12

syscall

mov rax, 60

mov rdi, 0

syscall

exit\_ten:

mov rax,60

mov rdi, 10

syscall

mov rax,60

mov rdi,1

syscall

section .data

Let's modify the previous example, but using the jb instruction, as follows:

The jump if below (jb) instruction means it will execute the jump if a **carry flag** (**CF**) is set (CF is equal to 1).

global\_start

section .text

\_start:

mov rax,1

mov rdi,1

mov rsi, hello\_one

mov rdx, length\_one

syscall

stc

jb print\_three

mov rax,1

mov rdi,1

mov rsi, hello\_two

mov rdix, length\_two

syscall

print\_three:

mov rax,1

mov rdi,1

mov rsi, hello\_three

mov rdx, length\_three

syscall

mov rax, 60

mov rdi, 11

syscall

section .data

hello\_one: db ‘hello one ‘, 0xa

length\_one: equ $-hello\_one

hello\_two: db ‘hello two’,0xa

length\_two: equ $-hello\_two

hello\_three: db ‘hello three’,0xa

length\_three: equ $-hello\_three

Now, let's look at a different method, that is, the jump if below or equal (jbe) instruction, which means CF is equal to 1 or **zero flag (ZF**) is equal to 1. The previous example will work too, but let's try something else to set ZF is equal to 1:

global\_start

section .text

\_start:

mov al, 0x1

sub al , 0x1

jbe exit\_ten

mov rax, 60

mov rdi, 0

syscall

exit\_tem:

mov rax, 60

mov rdi, 10

syscall

section .data

In the previous code, the subtraction operation will set ZF and then we will use the jbe instruction to test whether CF is equal to 1 or ZF is equal to 1; if true, then it will jump to execute exit\_ten.

Another type is jump if not sign (jns), which means SF is equal to 0:

global \_start

section .text

\_start:

mov al, 0x1

sub al, 0x3

jns exit\_ten

mov rax, 60

mov rdi, 0

syscall

exit\_ten:

mov rax, 60

mov rdi, 10

syscall

section .data

In the previous code, the subtraction operation will set the **sign flag** (**SF**) equal to 1. After that, we will test whether SF is equal to 0, which will fail, and it won't jump to execute exit\_ten and will continue with the normal exit with exit status 0.

**Procedures**

Procedures in assembly can act as functions in high-level language, which means that you can

write a block of code, then you can call it to execute.

For example, we can build a procedure that can take two numbers and add them. Also, we can

use it many times during execution using the call instruction.

Building procedures is easy. First, define your procedure before \_start, then add your

instructions and end your procedure with the ret instruction.

Let's try to build a procedure that can take two numbers and add them:

global \_start

section .text

addition:

add bl,al

ret

\_start:

mov al, 0x1

mov bl, 0x3

call addition

mov r8,0x4

mov r9, 0x2

call addition

mov rax, 60

mov rdi, 1

syscall

section .data

**Logical operations**

**Bitwise operations**

**AND:**

global \_start

section .text

\_start:

mov rax,0x10111011

mov rbx,0x11010110

and rax, rbx

mov rax, 60

mov rdi, 10

syscall

section .data

**OR:**

global \_start

section .text

\_start:

mov rax,0x10111011

mov rbx,0x11001010

or rax,rbx

mov rax, 60

mov rdi, 10

syscall

section .data

**XOR:**

global \_start

section .text

\_start:

mov rax,0x10111011

mov rbx,0x11001010

xor rax,rbx

mov rax,60

mov rdi,10

syscall

section .data

**Tip**: *You can use the XOR instruction on a register with itself to clear the content of*

*that register. For instance, xor rax and rax will fill the RAX register with zeros.*

**NOT**

global \_start

section .text

\_start:

mov al,0x00

not al

mov rax,60

mov rdi,10

syscall

section .data

**Bit-shifting operarions**

**Arithmetic shift operations**

**SAL:**

global \_start

section .text

\_start:

mov rax,0x0fffffffffffffff

sal rax,4

sal rax,4

mov rax,60

mov rdi,0

syscall

section .data

**SAR:**

global \_start

section .text

\_start:

mov rax,0x0ffffffffffffff

sar rax,4

mov rax,60

mov rdi,0

syscall

section .data

**Tip:** *The most significant bit is used as an indication for the sign,* ***0*** *for the positive*

*number and* ***1*** *for the negative number.*

**Logical shift**

global \_start

section .text

\_start:

mov rax, 0x0fffffffffffffff

shl rax, 4

shl rax, 4

mov rax, 60

mov rdi, 0

syscall

section .data

global \_start

section .text

\_start:

mov rax, 0xffffffffffffffff

shr rax, 32

mov rax, 60

mov rdi, 0

syscall

section .data

**Rotate operation**

global \_start

section .text

\_start:

mov rax, 0xffffffff00000000

ror rax, 32

mov rax, 60

mov rdi, 0

syscall

section .data



global \_start

section .text

\_start:

mov rax, 0xffffffff00000000

rol rax, 32

mov rax, 60

mov rdi, 0

syscall

section .data

