title: Write-up for Assembly Crash Course

Write-up for Assembly Crash Course

1. set-register:

> In this level, you will work with registers! Please set the following:
> rdi = 0x1337

Use mov to assign 0x1337 to rdi:

```
mov rdi, 0x1337
```

2. set-multiple-registers:

> In this level, you will work with multiple registers. Please set the following:
> rax = 0x1337
> r12 = 0xCAFED00D1337BEEF
> rsp = 0x31337

Similarly, use MOV to assign values to registers:

```
mov rax, 0x1337
mov r12, 0xCAFED00D1337BEEF
mov rsp, 0x31337
```

3. add-to-register:

> Do the following:
> add 0x331337 to rdi

Here, we need to add 0x331337 to rdi, so we should use add instead of mov rdi, 0x331337 (which only changes the value without adding).

```
add rdi, 0x331337
```

4. linear-equation-registers:

> Using your new knowledge, please compute the following:
>
> f(x) = mx + b, where:
> m = rdi
> x = rsi
> b = rdx
> Place the result into rax.

In this challenge, we need to perform the operation: mx + b, where m = `rdi` , x = `rsi` , and b = `rdx` .

- First, we compute m * x (rdi *rsi) using `imul` :

```
imul rdi, rsi
```

- Then, we store this value in rax using:

```
mov rax, rdi
```

- Finally, we add rdx to rax with:

```
add rax, rdx
```

5. integer-division:

> Please compute the following:
>
> speed = distance / time, where:
> distance = rdi
> time = rsi
> speed = rax
> Note that distance will be at most a 64-bit value, so rdx should be 0 when dividing.

In this challenge, we need to divide `rdi` by `rsi` , and store the result in `rax` . We use `div` to perform the division.

Since when using `div`, only `rax` is used as the dividend, and the quotient is also stored in `rax` while the remainder is stored in `rdx`, we first move `rdi` into `rax` and set `rdx` to 0 to ensure an accurate result.

```
mov rdx, 0
mov rax, rdi
div rsi
```

6. modulo-operation:

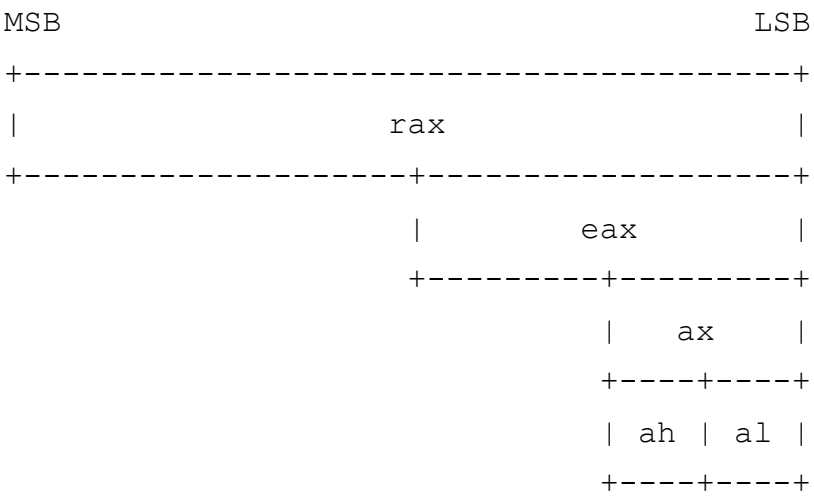
> Please compute the following: rdi % rsi
>
> Place the value in rax.

In the previous challenge, we performed division to get the quotient. In this case, we are dividing to get the remainder, which is stored in `rdx`. We will divide as before and then move the value of `rdx` into `rax`.

```
mov rdx, 0
mov rax, rdi
div rsi
mov rax, rdx
```

7. set-upper-byte:

> For example, the lower 32 bits of rax can be accessed using eax, the lower 16 bits using ax, and the lower 8 bits using al.



Lower register bytes access is applicable to almost all registers.

> Using only one move instruction, please set the upper 8 bits of the ax register to 0x42.

Since the highest 8-bit of the `ax` register can be accessed through `al` and `ah`, we can set the value of `ah` to `0x42` using:

```
mov ah, 0x42
```

8. efficient-modulo:

> If we have x % y, and y is a power of 2, such as 2^n, the result will be the lower n bits of x.
> Please compute the following:
> rax = rdi % 256
> rbx = rsi % 65536

For the operation `rax = rdi % 256` and `rbx = rsi % 65536`, since 256 and 65536 are 2^8^ and 2^16^ respectively, the remainder when dividing by 256 and 65536 will be the last 8-bit and the last 16-bit of the registers holding the dividend.
For `rax`, the lowest 8-bit are `al`, and for `rdi`, the lowest 8-bit are `dil`.
For `rbx`, the lowest 16-bit are `bx`, and for `rsi`, the lowest 16-bit are `si`.

Therefore, the result of the `mod` operation for:

- `rdi % 256` will be stored in `dil`.
- `rsi % 65536` will be stored in `si`.

```
mov al, dil
mov bx, si
```

9. efficient-modulo:

> Using only the following instructions:
>
> mov, shr, shl
> Please perform the following: Set rax to the 5th least significant byte of rdi.
>
> For example:
>
> rdi = | B7 | B6 | B5 | B4 | B3 | B2 | B1 | B0 |
> Set rax to the value of B4

We will shift `rdi` to the right by 32-bit, so the value B4 is moved to the lower part of the register. After that, we will move the value of `dil` into `al`.

```
shr rdi, 32
mov al, dil
```

10. bitwise-and:

> Without using the following instructions: mov, xchg, please perform the following:
>
> Set rax to the value of (rdi AND rsi)

We have:

a xor a = 0
a xor b = b (when a = 0)

First, use `xor` to set `rax` = `rdi`, since `rax xor rax = 0`, which leaves `rdi` in `rax`.
Then, use and to perform a bitwise and between `rax` and `rsi`.

```
xor rax, rax
xor rax, rdi
and rax, rsi
```

11. check-even:

> Using only the following instructions:
>
> and, or, xor
>
> Implement the following logic:
>
> if x is even then y = 1
> else y = 0
> Where:
>
> x = rdi
> y = rax

The parity (even or odd) of a number is determined by its 2^0^ bit.

- If this bit is `1`, the number is odd.
- If this bit is `0`, the number is even.

To implement this logic:

- If `rdi` is even, `rax = 1`.
- If `rdi` is odd, `rax = 0`.

We using and 1 to extract the least significant bit of rdi is 1 or 0. We use `and 1` to check whether the 2^0^ bit of `rdi` is `1` or `0`. Then, we use `xor 1` to invert the bit (`1 → 0`, `0 → 1`).

```
and rdi, 1
xor rax, rax
xor rax, rdi
xor rax, 1
```

12. memory-read:

> Please perform the following: Place the value stored at 0x404000 into rax. Make sure the value in rax is the original value stored at 0x404000.

Use the following instruction to move the value at memory address `0x404000` into `rax`:

```
mov rax, [0x404000]
```

13. memory-write:

> Please perform the following: Place the value stored in rax to 0x404000.

Using:

```
mov [0x404000], rax
```

14. memory-increment:

> Please perform the following:
> Place the value stored at 0x404000 into rax.
> Increment the value stored at the address 0x404000 by 0x1337.

> Make sure the value in rax is the original value stored at 0x404000 and make sure that [0x404000] now has the incremented value.

Using:

```
mov rax, [0x404000]
mov rbx, 0x1337
add [0x404000], rbx
```

15. byte-access:

> Please perform the following: Set rax to the byte at 0x404000.

We have:
Byte = 1 byte = 8 bits

```
mov al, [0x404000]
```

16. memory-size-access:

> Please perform the following:
>
> Set rax to the byte at 0x404000
> Set rbx to the word at 0x404000
> Set rcx to the double word at 0x404000
> Set rdx to the quad word at 0x404000

The breakdown of the names of memory sizes:
Quad Word = 8 Bytes = 64 bits
Double Word = 4 bytes = 32 bits
Word = 2 bytes = 16 bits
Byte = 1 byte = 8 bits

```
mov al, [0x404000]
mov bx, [0x404000]
mov ecx, [0x404000]
mov rdx, [0x404000]
```

17. little-endian-write:

> For this challenge we will give you two addresses created dynamically each run.
>
> The first address will be placed in rdi.
> The second will be placed in rsi.
>
> Using the earlier mentioned info, perform the following:
> Set [rdi] = 0xdeadbeef00001337
> Set [rsi] = 0xc0ffee0000
>
> Hint: it may require some tricks to assign a big constant to a dereferenced register.
> Try setting a register to the constant value then assigning that register to the dereferenced register.

Use the `mov` instruction to store values into memory at specific addresses. Then, we will use `mov` instructions to store values from registers into memory:
`0xdeadbeef00001337` is a 64-bit value in binary, stored using a 64-bit register.
`0xc0ffee0000` is a 40-bit value in binary, stored using a 64-bit register.

```
mov rax, 0xdeadbeef00001337
mov rbx, 0xc0ffee0000
mov [rdi], rax
mov [rsi], rbx
```

18. memory-sum:

> Perform the following:
> Load two consecutive quad words from the address stored in rdi.
> Calculate the sum of the previous steps' quad words.
> Store the sum at the address in rsi.

Two consecutive quad words from the address stored in rdi are rdi and rdi+8. Use add and mov to store a value in rax and then save the value in rax to the memory address stored in rsi:

```
mov rax, [rdi]
add rax, [rdi+8]
mov [rsi], rax
```

19. stack-subtraction:

> On x86, the pop instruction will take the value from the top of the stack and put it into a register.
>
> Similarly, the push instruction will take the value in a register and push it onto the top of the stack.
>
> Using these instructions, take the top value of the stack, subtract rdi from it, then put it back.

Retrieve the value at the top of the stack using `pop rax`.
Subtract the value of `rdi` from `rax`.
Push the result back onto the stack using `push rax`.

```
pop rax
sub rax, rdi
push rax
```

20. swap-stack-values:

> Using only the following instructions:
>
> push
> pop
> Swap values in rdi and rsi.
>
> Example:
>
> If to start rdi = 2 and rsi = 5
> Then to end rdi = 5 and rsi = 2

Use `push` to sequentially store the values of `rdi` and `rsi` into the register, then use `pop` to retrieve them in reverse order, `rsi` first, then `rdi`.

```
push rdi
push rsi
pop rdi
pop rsi
```

21. average-stack-values:

> Without using pop, please calculate the average of 4 consecutive quad words stored on the stack. Push the average on the stack.
>
> Hint:
>
> RSP+0x?? Quad Word A
> RSP+0x?? Quad Word B
> RSP+0x?? Quad Word C
> RSP Quad Word D

`rsp` always stores the memory address of the top of the stack so that the quad words are [rsp+24], [rsp+16], [rsp+8], and [rsp], respectively. We will use rax to store the sum of these values and then divide by rbx (4 is loaded into rbx).

```
mov rax, [rsp]
add rax, [rsp+8]
add rax, [rsp+16]
add rax, [rsp+24]
mov rbx, 4
div rbx
push rax
```

22. absolute-jump:

> For all jumps, there are three types:
>
> Relative jumps: jump + or - the next instruction.
> Absolute jumps: jump to a specific address.
> Indirect jumps: jump to the memory address specified in a register.
>
> In this level, we will ask you to do an absolute jump. Perform the following: Jump to the absolute address 0x403000.

In x86, absolute jumps (jump to a specific address) are accomplished by first putting the target address in a register reg, then doing jmp reg.

```
mov rax, 0x403000
jmp rax
```

23. relative-jump:

> In this level, we will ask you to do a relative jump. You will need to fill space in your code with something to make this relative jump possible. We suggest using the nop instruction. It's 1 byte long and very predictable.
>
> In fact, the assembler that we're using has a handy .rept directive that you can use to repeat assembly instructions some number of times: GNU Assembler Manual
>
> Useful instructions for this level:
>
> jmp (reg1 | addr | offset)
> nop
> Hint: For the relative jump, look up how to use labels in x86.
>
> Using the above knowledge, perform the following:
>
> Make the first instruction in your code a jmp.
> Make that jmp a relative jump to 0x51 bytes from the current position.
> At the code location where the relative jump will redirect control flow, set rax to 0x1.

We know that `.rept` helps us repeat a certain instruction, and now we need to use `jmp` to jump to an instruction located 0x51 bytes (or 81 bytes) ahead. Since the `nop` instruction takes up 1 byte and does nothing, we will apply that here.

```
jmp jump
.rept 81
nop
.endr
jump:
mov rax, 0x1
```

24. jump-trampoline:

> Create a two jump trampoline:
> Make the first instruction in your code a jmp.
> Make that jmp a relative jump to 0x51 bytes from its current position.
> At 0x51, write the following code:
> Place the top value on the stack into register rdi.
> jmp to the absolute address 0x403000.

We simply combine the two previous tasks:

```
jmp jump
.rept 81
nop
.endr
jump:
pop rdi
mov rax, 0x403000
jmp rax
```

25. conditional-jump:

> Using the above knowledge, implement the following:
>
> if [x] is 0x7f454c46:
> y = [x+4] + [x+8] + [x+12]
> else if [x] is 0x00005A4D:
> y = [x+4] - [x+8] - [x+12]
> else:
> y = [x+4] * [x+8] * [x+12]
> Where:
>
> x = rdi, y = rax.
> Assume each dereferenced value is a signed dword. This means the values can start as a negative value at each memory position.
>
> A valid solution will use the following at least once:
>
> jmp (any variant), cmp

```
    mov eax, dword ptr [rdi]        ; đọc [x] vào eax
    cmp eax, 0x7f454c46
    jne else_if

    mov eax, dword ptr [rdi+4]      ; eax = [x+4]
    add eax, dword ptr [rdi+8]      ; eax += [x+8]
    add eax, dword ptr [rdi+12]     ; eax += [x+12]
    jmp end

else_if:
    cmp eax, 0x00005A4D
    jne else
```

```
    mov eax, dword ptr [rdi+4]      ; eax = [x+4]
    sub eax, dword ptr [rdi+8]      ; eax -= [x+8]
    sub eax, dword ptr [rdi+12]     ; eax -= [x+12]
    jmp end

else:
    mov eax, dword ptr [rdi+4]      ; eax = [x+4]
    imul eax, dword ptr [rdi+8]     ; eax *= [x+8]
    imul eax, dword ptr [rdi+12]    ; eax *= [x+12]

end:
    nop
```

26. indirect-jump:

> Using the above knowledge, implement the following logic:
>
> if rdi is 0:
> jmp 0x40301e
> else if rdi is 1:
> jmp 0x4030da
> else if rdi is 2:
> jmp 0x4031d5
> else if rdi is 3:
> jmp 0x403268
> else:
> jmp 0x40332c
> Please do the above with the following constraints:
>
> Assume rdi will NOT be negative.
> Use no more than 1 cmp instruction.
> Use no more than 3 jumps (of any variant).
> We will provide you with the number to 'switch' on in rdi.
> We will provide you with a jump table base address in rsi.
> Here is an example table:
>
> [0x40427c] = 0x40301e (addrs will change)
> [0x404284] = 0x4030da
> [0x40428c] = 0x4031d5
> [0x404294] = 0x403268
> [0x40429c] = 0x40332c

Use `cmp` to compare the value of `rdi` with 3. If it is greater than 3, jump to `default_case`, which jump to default address `rsi + 0x20` (use `ja` for unsigned comparison).

In the cases of 0, 1, 2, or 3, jump to the address `[rsi + rdi * 8]` following the jump table.

```
cmp rdi, 3
ja default_case
jmp [rsi + rdi*8]

default_case:
jmp [rsi + 0x20]
```

27. average-loop:

> In most programming languages, a structure exists called the for-loop, which allows you to execute a set of instructions for a bounded amount of times. The bounded amount can be either known before or during the program's run, with "during" meaning the value is given to you dynamically.
>
> As an example, a for-loop can be used to compute the sum of the numbers 1 to n:
>
> sum = 0
> i = 1
> while i <= n:
> sum += i
> i += 1
> Please compute the average of n consecutive quad words, where:
>
> rdi = memory address of the 1st quad word
> rsi = n (amount to loop for)
> rax = average computed

Similar to computing the average of 4 integer quad words, the average of n integer quad words can be calculated using a loop and jump instructions.
We use rbx to store the index (from 1 to n) to control the loop.

```
mov rax, [rdi]
mov rbx, 0x01
loop:
cmp rbx, rsi
je end
add rax, [rdi+rbx*8]
```

```
add rbx, 0x01
jmp loop
end:
div rsi
```

28. count-non-zero:

> A second loop structure exists called the while-loop to fill this demand. In the while-loop, you iterate until a condition is met.
>
> As an example, say we had a location in memory with adjacent numbers and we wanted to get the average of all the numbers until we find one bigger or equal to 0xff:
>
> average = 0
> i = 0
> while x[i] < 0xff:
> average += x[i]
> i += 1
> average /= i
> Using the above knowledge, please perform the following:
>
> Count the consecutive non-zero bytes in a contiguous region of memory, where:
>
> rdi = memory address of the 1st byte
> rax = number of consecutive non-zero bytes
> Additionally, if rdi = 0, then set rax = 0 (we will check)!
>
> An example test-case, let:
>
> rdi = 0x1000
> [0x1000] = 0x41
> [0x1001] = 0x42
> [0x1002] = 0x43
> [0x1003] = 0x00
> Then: rax = 3 should be set.

```
mov rax, 0                      ; rax = 0, initialize counter
cmp rdi, 0          ; if rdi == 0
je end              ; → rax remains 0, exit

loop:
mov bl, [rdi]
```

```
; read 1 byte from address rdi into bl
cmp bl, 0               ; compare with 0
je end                  ; if equal → end loop

inc rax                 ; increment counter
inc rdi                 ; move to the next byte
jmp loop

end:
nop
```

29. string-lower:

> In this level, you will be provided with a contiguous region of memory again and will loop over each performing a conditional operation till a zero byte is reached. All of which will be contained in a function!
>
> A function is a callable segment of code that does not destroy control flow.
>
> Functions use the instructions "call" and "ret".
>
> The "call" instruction pushes the memory address of the next instruction onto the stack and then jumps to the value stored in the first argument.
>
> Let's use the following instructions as an example:
>
> 0x1021 mov rax, 0x400000
> 0x1028 call rax
> 0x102a mov [rsi], rax
> call pushes 0x102a, the address of the next instruction, onto the stack.
> call jumps to 0x400000, the value stored in rax.
> The "ret" instruction is the opposite of "call".
>
> ret pops the top value off of the stack and jumps to it.
>
> Let's use the following instructions and stack as an example:
>
> Stack ADDR VALUE
> 0x103f mov rax, rdx RSP + 0x8 0xdeadbeef
> 0x1042 ret RSP + 0x0 0x0000102a
> Here, ret will jump to 0x102a.
>
> Please implement the following logic:

> str_lower(src_addr):
> i = 0
> if src_addr != 0:
> while [src_addr] != 0x00:
> if [src_addr] <= 0x5a:
> [src_addr] = foo([src_addr])
> i += 1
> src_addr += 1
> return i
> foo is provided at 0x403000. foo takes a single argument as a value and returns a value.
>
> All functions (foo and str_lower) must follow the Linux amd64 calling convention (also known as System V AMD64 ABI): System V AMD64 ABI
>
> Therefore, your function str_lower should look for src_addr in rdi and place the function return in rax.
>
> An important note is that src_addr is an address in memory (where the string is located) and [src_addr] refers to the byte that exists at src_addr.
>
> Therefore, the function foo accepts a byte as its first argument and returns a byte.

Solve:

```
mov rbx, 0              ; Initialize counter i = 0
cmp rdi, 0              ; Check if src_addr is null
je end
cmp rsi, 0              ; Check if string length = 0
je end
while:
    mov al, byte ptr [rdi]  ; Load current character
    cmp al, 0               ; Check for null terminator
    je end
    cmp al, 0x5a            ; Compare with 'Z'
    ja next
    inc rbx                 ; Increment counter i
    push rdi                ; Save string pointer
    mov dil, al             ; Pass character to foo
    call 0x403000           ; Call foo
    pop rdi                 ; Restore pointer
    mov byte ptr [rdi], al  ; Store result
next:
    inc rdi                 ; Advance string pointer
    dec rsi                 ; Decrement remaining length
```

```
    jz end                  ; Exit if string ends
    jmp while
end:
    mov rax, rbx            ; Return counter i
    ret
```

30. most-common-byte:

A function stack frame is a set of pointers and values pushed onto the stack to save things for later use and allocate space on the stack for function variables.

First, let's talk about the special register rbp, the Stack Base Pointer.

The rbp register is used to tell where our stack frame first started. As an example, say we want to construct some list (a contiguous space of memory) that is only used in our function. The list is 5 elements long, and each element is a dword. A list of 5 elements would already take 5 registers, so instead, we can make space on the stack!

The assembly would look like:

; setup the base of the stack as the current top
mov rbp, rsp
; move the stack 0x14 bytes (5 * 4) down
; acts as an allocation
sub rsp, 0x14
; assign list[2] = 1337
mov eax, 1337
mov [rbp-0xc], eax
; do more operations on the list ...
; restore the allocated space
mov rsp, rbp
ret
Notice how rbp is always used to restore the stack to where it originally was. If we don't restore the stack after use, we will eventually run out. In addition, notice how we subtracted from rsp, because the stack grows down. To make the stack have more space, we subtract the space we need. The ret and call still work the same.

Consider the fact that to assign a value to list[2] we subtract 12 bytes (3 dwords). That is because stack grows down and when we moved rsp our stack contains addresses <rsp, rbp).

Once again, please make function(s) that implement the following:

> most_common_byte(src_addr, size):
> i = 0
> while i <= size-1:
> curr_byte = [src_addr + i]
> [stack_base - curr_byte * 2] += 1
> i += 1
>
> b = 1
> max_freq = 0
> max_freq_byte = 0
> while b <= 0x100:
> if [stack_base - b * 2] > max_freq:
> max_freq = [stack_base - b * 2]
> max_freq_byte = b
> b += 1
>
> return max_freq_byte
> Assumptions:
>
> There will never be more than 0xffff of any byte
> The size will never be longer than 0xffff
> The list will have at least one element
> Constraints:
>
> You must put the "counting list" on the stack
> You must restore the stack like in a normal function
> You cannot modify the data at src_addr

I had some dificulties with this challenge, so that I analyzed it a bit more carefully (also as a way to review the previous challs)

Set up the function frame:

```
most_common_byte:
    push rbp
    mov rbp, rsp
    sub rsp, 0x200
```

- Allocate 512 bytes = 256 words (2 bytes per entry) on the stack to create a frequency array for counting occurrences of bytes (0–255).

Initialize the count array to 0:

```
mov rcx, 0x100
lea r8, [rbp - 0x200]
xor eax, eax
.init_loop:
    mov [r8 + rcx*2 - 2], ax
    loop .init_loop
```

- `rcx = 256` — number of entries (words).
- `r8` is the base address of the count array.
- The loop writes `ax = 0` to each element at `[r8 + rcx*2 - 2]` (since each entry is a word = 2 bytes).

Count the frequency of each byte:

```
xor ecx, ecx
.count_loop:
    cmp ecx, esi
    jge .count_done
    movzx eax, byte [rdi + ecx]
    inc word ptr [r8 + rax*2]
    inc ecx
    jmp .count_loop
```

- `ecx` is used as the loop index `i`.
- Loop continues until `i >= size` (`esi`).
- Reads the byte at `[rdi + i]` and stores it in `eax` (value from 0–255).
- Increments the corresponding count at `[r8 + eax*2]`.

Find the byte with the highest frequency:

```
.count_done:
    mov ecx, 0
    xor edx, edx
    xor eax, eax
```

- `ecx` iterates through each byte value (0–255).
- `edx` stores `max_freq`.
- `eax` stores `max_freq_byte`.

```
.find_max_loop:
    cmp ecx, 0xff
    jg .find_max_done
    movzx ebx, word ptr [r8 + ecx*2]
    cmp ebx, edx
    jle .not_greater
    mov edx, ebx
    mov eax, ecx
.not_greater:
    inc ecx
    jmp .find_max_loop
```

- Compares the frequency of each byte (`ecx`) with the current `max_freq` (`edx`).
- If greater, updates `max_freq = ebx` and `max_freq_byte = ecx` .

End the function:

```
.find_max_done:
    mov rsp, rbp
    pop rbp
    ret
```

- Restore the stack to its original state.
- Return the result (most common byte) in `rax` .