

1 Working with byte and word data

We are declaring some bytes in the assembly program below:

```
1 section .data
2     num DB 1
3     num2 DB 2
4     num3 DB 3
5
6 section .text
7     global _start
8     _start:
9         MOV ebx, [num]
10        MOV ecx, [num2]
11        MOV edx, [num3]
12        MOV eax, 1
13        INT 80h
```

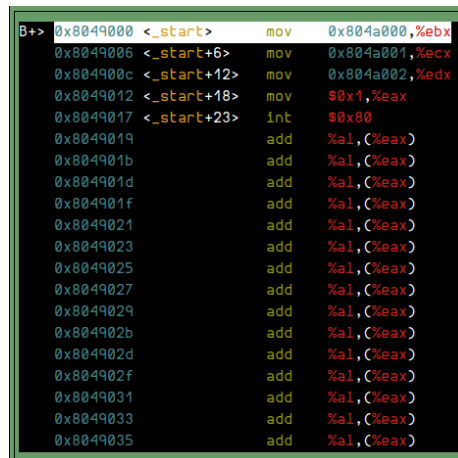
So, everything looks normal(for now). We are just moving the bytes into the registers. Note that these registers are 32-bit registers i.e. 4 byte registers and the variables are just 1 byte each.

So, how will 1 byte be stored in 4 bytes?? Let's find out using GDB.

2 Using GDB to find out how 1 byte is stored in 4 bytes

Everything is just the same as the previous examples. Let's focus on the main portion.

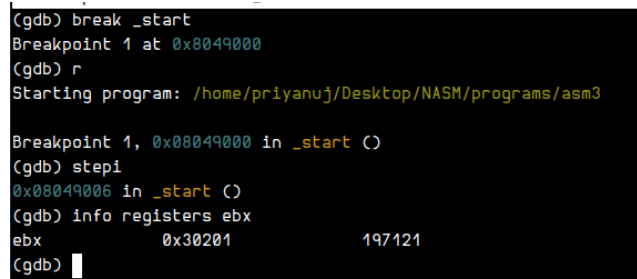
The assembly layout:



```
0x8049000 <_start> mov 0x804a000,%ebx
0x8049006 <_start+6> mov 0x804a001,%ecx
0x804900c <_start+12> mov 0x804a002,%edx
0x8049012 <_start+18> mov $0x1,%eax
0x8049017 <_start+23> int $0x80
0x8049019 add %al,%eax
0x804901b add %al,%eax
0x804901d add %al,%eax
0x804901f add %al,%eax
0x8049021 add %al,%eax
0x8049023 add %al,%eax
0x8049025 add %al,%eax
0x8049027 add %al,%eax
0x8049029 add %al,%eax
0x804902b add %al,%eax
0x804902d add %al,%eax
0x804902f add %al,%eax
0x8049031 add %al,%eax
0x8049033 add %al,%eax
0x8049035 add %al,%eax
```

Figure 1: Starting with GDB

Notice the value in **ebx** in the image below:



```
(gdb) break _start
Breakpoint 1 at 0x8049000
(gdb) r
Starting program: /home/priyanuj/Desktop/NASM/programs/asm3

Breakpoint 1, 0x8049000 in _start ()
(gdb) stepi
0x8049006 in _start ()
(gdb) info registers ebx
ebx             0x30201             197121
(gdb)
```

Figure 2: Value in **ebx**

We can see that via the **info registers** command that the value in **ebx** is not 1. Rather it's a very large number.

How??

If we use the **x/x** command then we will get the following:

```
(gdb) x/x 0x804a000
0x804a000:      0x00030201
(gdb) █
```

Figure 3: What does **0x804a000** store in it??

So, this tells us that in the address **0x804a000** which is a 32-bit address the value stored in it is **0x00030201**. What is this value?!

If we look at the binary equivalent of this hexadecimal equivalent number:

$$(00030201)_{16} = (00000000\ 00000011\ 00000010\ 00000001)_2$$

We can clearly see that $(00000011)_2$ is $(3)_{10}$ in decimal, $(00000010)_2$ is $(2)_{10}$ in decimal and $(00000001)_2$ is $(1)_{10}$ in decimal.

So, bytes, in a 32-bit register or are stored next to each other.

We have more to see because the number of bytes stored in 4 bytes changes in each of the registers.

```
(gdb) stepi
0x0804900c in _start ()
(gdb) info registers ecx
ecx      0x302      770
(gdb) █
```

Figure 4: Value in **ecx**(Notice the change)

As we move towards the next instruction using the **stepi** command, we can see that **MOV ecx, [num2]** gets executed.

Notice that **ecx** stores a value(look at the hexadecimal value).

We again use the **x/x** command on the next address which is **0x804a001**(also, notice that the addresses **0x804a000**, **0x804a001** and **0x804a002**, all these are 1 byte apart from each other).

```
(gdb) x/x 0x804a001
0x804a001:      0x00000302
(gdb) █
```

Figure 5: Value of **0x804a001**

We can see that the binary equivalent of **0x00000302** is:

$$(00000302)_{16} = (00000000\ 00000000\ 00000011\ 00000010)_2$$

Now, we again step to the next instruction: **INT 80h** which means that **MOV edx, [num3]** has completed execution. Let's see what **edx** has in store:

```
(gdb) stepi
0x08049012 in _start ()
(gdb) info registers edx
edx                0x3                3
(gdb) █
```

Figure 6: Value in **edx**

So, **edx** stores only 3. Let's see what is in **0x804a002**

```
(gdb) x/x 0x804a002
0x804a002:          0x00000003
(gdb) █
```

Figure 7: Value in **0x804a002**

$(00000003)_{16}$ in hexadecimal is $(00000000\ 00000000\ 00000000\ 00000011)_2$ in binary.

Why were these numbers stored in the registers like this???

Let's look at the data section again:

```
section .data
    num DB 1
    num2 DB 2
    num3 DB 3
```

If we look at, say, [Figure 5](#) we can see the value(in hex) **0x00000302**. As we know that one hex digit represents 4-bits. So, two hex digits will represent 8-bits or 1 byte. So from the figure we can say that **0x02** represents 1 byte, **0x03** represents 1 byte, **0x00** represents 1 byte and the last(left-most) **0x00** also represents 1 byte. So all total there 4 bytes i.e 32-bits.

3 IMPORTANT

How actually is 1 byte stored in the 32-bit address?

- Let's know a basic concept first: **The memory is like a big array.**

We know that **0x804a000** is a 32-bit long **address**. That doesn't mean that the memory block is 32-bits long. It's like the memory blocks shown in figure **Figure 8**. So, the length of the address has nothing to do with the size of the memory block to which that address is pointing to.

Now, coming to the command **x/x**. Let's understand the meaning of this command:

x/x → This command is used to examine memory at a specified address and display the contents as a **single hexadecimal integer**. The **/x** modifier specifies the **display format** as hexadecimal.

For example:

```
(gdb) x/x 0x804a000
```

This command will basically display the contents of the memory pointed to by this address(**0x804a000**) as a 32-bit hexadecimal integer. This 32 bit hexadecimal integer is not only taken from **0x804a000** but this 32-bit hexadecimal is made up of 3 more memory blocks pointed to by **0x804a001**, **0x804a002** and **0x804a003** each address contributing 1 byte(2 hex digits). In simple words this command displays the contents in the memory starting from **0x804a000** and going all the way till **0x804a003**.

And that's the reason why we see **0x00030201**. Now if we look at this hexadecimal integer, **0x00** is the MSB and **0x01** is LSB. So the byte ordering is *Little Endian* because the LSB is stored in **0x804a000** which is the first address.

Watch this [video](#).

Consider this command:

```
(gdb) x/1xb 0x804a000
```

This command displays 1 byte in hexadecimal format. So, in this case only the value in the memory block pointed to by **0x804a000** will be displayed as a hexadecimal byte. See figure **Figure 9**.

Now, for **x/3xb**

```
(gdb) x/3xb 0x804a000
```

This command displays 3 consecutive bytes in hexadecimal format. The three consecutive bytes will come from **0x804a000**, **0x804a001** and **0x804a002**.

0x804a000 stores 1(hex: **0x01**), **0x804a001** stores 2(hex: **0x02**) and **0x804a002** stores 3(hex: **0x03**). See figure **Figure 10**.

Note that this command(s) will work in similar way for any address not just for **0x804a000**.

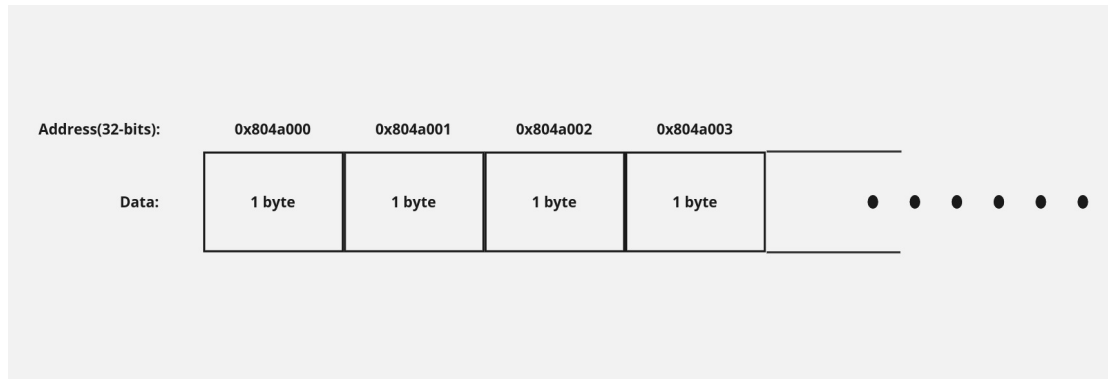


Figure 8: Memory makeup

```
(gdb) x/1xb 0x804a000
0x804a000: 0x01
```

Figure 9: Hexadecimal byte

```
(gdb) x/3xb 0x804a000
0x804a000: 0x01 0x02 0x03
```

Figure 10: 3 hexadecimal bytes

How are registers storing the values?

Now, the registers **ebx**, **ecx** and **edx** are 32-bit registers(see [Figure 11](#)).

Now, if we want to see the contents inside of register **ebx**, then we will use the **info registers** command(see [Figure 2](#)). We can see that **ebx** stores the value **0x30201**. Why?? Because **ebx** is a 32-bit register. When we execute **MOV ebx, [num]** then since **ebx** is a 32-bit register so the contents of the memory block starting from address **0x804a000** are stored into the register. We know **0x804a000** points a memory block of 1 byte data but **ebx** is 4 bytes long, so we need 3 more bytes to store into **ebx**. So, the memory blocks pointed to by the addresses **0x804a001**, **0x804a002** and **0x804a003** are also stored into the register **ebx**.

Similarly when we execute **MOV ecx, [num2]** then again 4 bytes are stored into register **ecx**. So the contents of the memory block starting from the address **0x804a001** to **0x804a004** are stored into the register. That's why when we run the **info registers** command on **ecx** we get the output that we see in [Figure 4](#) because the address **0x804a001** points to a memory block that contains the byte 2(hex: **0x02**), **0x804a002** points to a memory block that contains the byte 3(hex: **0x03**), **0x804a003** points to a memory block that is uninitialized and so is the case with **0x804a004**.

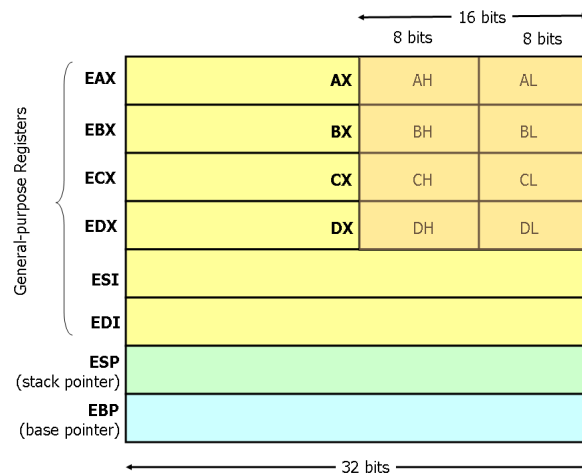


Figure 11: x86 Registers(Image source: [Google](#))

- I found these links helpful:
 - <https://cs.stackexchange.com/questions/157633/why-does-a-32-bit-address-only-contain-1-byte-when-32-bits-4-bytes>
 - <https://www.quora.com/How-does-a-32-bit-address-represent-1-byte-of-memory>

4 The solution to the problem

Before starting, this image is important:

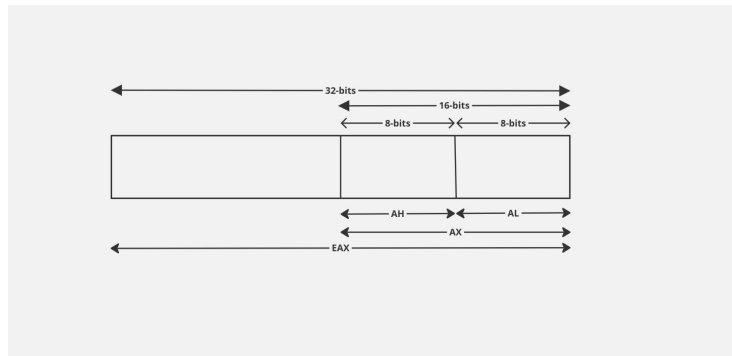


Figure 12: Register sizes

We modified our program:

```

1 section .data
2     num DB 1
3     num2 DB 2
4     num3 DB 3
5
6 section .text
7     global _start
8     _start:
9         MOV bl, [num]
10        MOV cl, [num2]
11        MOV dl, [num3]
12        MOV eax, 1
13        INT 80h

```

And again starting debugging using GDB.

NOTE:

Now, from **Figure 12**, we can say that **EAX** is a 32-bit register, **AX** is a sub-register that is 16-bits, **AH**(high order byte) is another sub-register that is 8-bits in size and **AL**(low order byte) is yet another sub-register whose size equals the size of **AH**. Similarly, for registers **EBX**, **ECX** and **EDX**, we have similar sub-registers as well(letter-wise i.e for **EBX** there are **BX**, **BH**, **BL**).

Now, when we use the command:

```
(gdb) info registers bl
```

This is what we get:

```
(gdb) info registers bl
bl          0x1          1
(gdb) █
```

Figure 13: Finally! Some normal output

So, 1 is stored into the lower bits. Register **bl** is 1 byte long and only 1 byte can be stored in it.

But surprisingly, when we type:

```
(gdb) info registers ebx
```

We get the following output:

```
(gdb) info registers ebx
ebx        0x1          1
(gdb) █
```

Figure 14: **ebx** stores only 1 now!!

Why did that happen?? Because we know that **bl** is a sub-register of **ebx** and it is 1 byte long and also it is the lower byte of **ebx**. So when we stored the value of **num** into **bl**. Only the lower 8-bits(1 byte) of **ebx** got initialized and the rest 24 bits were uninitialized(0). And the whole register kind of looked like this:

00000000 00000000 00000000 00000001

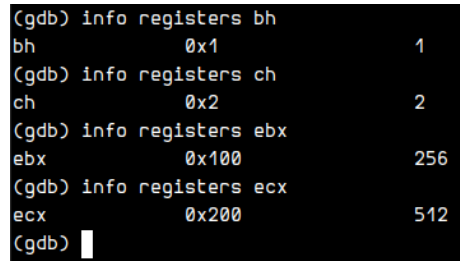
That is the reason why we see 1 in **ebx**.

5 What will happen if we store the bytes in the higher byte of the registers?

Now, we again modified our program:

```
1 section .data
2     num DB 1
3     num2 DB 2
4     num3 DB 3
5
6 section .text
7     global _start
8     _start:
9         MOV bh, [num]
10        MOV ch, [num2]
11        MOV dh, [num3]
12        MOV eax, 1
13        INT 80h
```

We are storing the bytes in the higher bytes of the registers. Let's see what happens then.



```
(gdb) info registers bh
bh          0x1          1
(gdb) info registers ch
ch          0x2          2
(gdb) info registers ebx
ebx         0x100        256
(gdb) info registers ecx
ecx         0x200        512
(gdb)
```

Figure 15: Storing in the higher bytes

One thing is very clear that when we check the values in the registers **bh** and **ch** we can see that the correct values are stored(just like the previously modified code) because these registers are 8 bits in size.

But one thing is different, the value in the 32-bit registers **ebx** and **ecx** are not 1 and 2, they are something else. HOW??

Let's see. In this program we stored the bytes in the high bytes of the registers i.e 1 is stored in the higher bytes of the register **ebx**(see [Figure 12](#)).

So for that reason the **ebx** looks kind of like this:

00000000 00000000 00000001 00000000

This binary representation is actually of the decimal number 256(i.e 2^8 if you look at the position of 1).

Similarly, the register **ecx** looks kind of like this:

00000000 00000000 00000010 00000000

This binary representation is of the decimal number 512(i.e 2^9 , look at the position of 1).

And that's the reason why we see those values while checking the values in the registers **ebx** and **ecx**.