x86 Assembly Using NASM

1. Binary Numbers

Binary:

binary is a base 2 numbering system, values are 0 and 1

we combine them to get value, **general formula** → $x \times 2^n + ... + y \times 2^1 + z \times 2^0$

Ex: Convert 1010 into decimal

(Binary value) ← **1010** = $((1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (0 \times 2^0)$ = $8 + 2$ = **10** → Decimal

Converting decimal to binary:

Ex: **37** convert into **Binary**

Division	Result	Remainder
37/2	18	1
18/2	9	0
9/2	4	1
4/2	2	0
2/2	1	0
1/2	0	1

37 → Binary = **100101** (Numbers are written from last to first)

---

2. Binary Basics 2

Basic add calculation cases in binary:

0	0	0
0	1	1
1	0	1

1	1	10

Ex: A = 0110, B = 0100, Add

1010 → 10

A	0 (1 carry)	1	1	0
B	0	1	0	0
result	1	0	1	0

Signed Binary Numbers:

binary nos are grouped in sets of 4, it basically it represents unit of memory, combining 8 of this bits we get a byte → divided into 2, 4-4 bit each or half of a byte

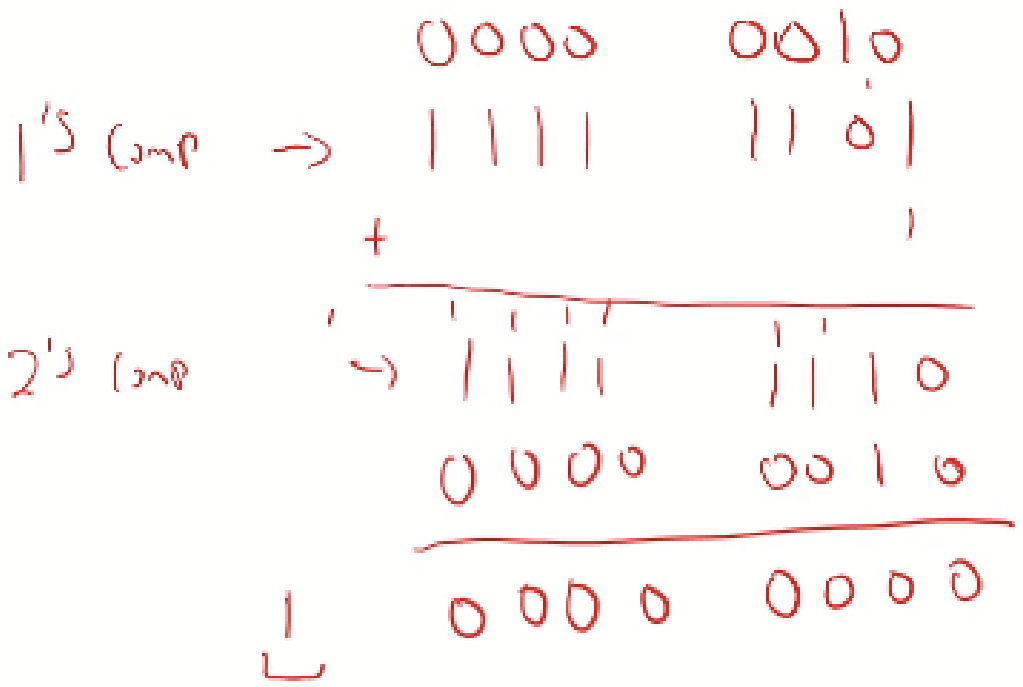
> Top bit indicates the sign and rest of the nos are binary numbers,
> top bit → sign, 0 =positive, 1 = negative
> Signed number is represented by **1**000 0010 → 1 in the start indicates that it's negative and rest is normal binary therefore **1**0000010 = -2

In normal arithmetic, A + (-A) = 0, A represent a number

10000010 + 00000010 = 10000100 → -4, therefore to keep this to our arithmetic, we use two's complement to represent these values

> 0000 0010 → 1111 1101 (one's complement)
> 1111 1101 + 1 (adding one to get two's complement)
> 1111 1110 → two's complement

now, 1111 1110 + 0000 0010 =**1** 0000 0000



This allows us to keep arithmetic properties and follow rules of mathematics without changing sign or value

---

3. Hexadecimal Numbers

rather then base 10 or base 2, we use base 16,
values in hexadecimal are **0-9, A-F** → A=10, B=11, ... , F = 15

Each 4 bit can be represented as 1 hexadecimal value, (highest value in 4 bit)
1111→ 1+2+4+8 = 15 → F in hexadecimal

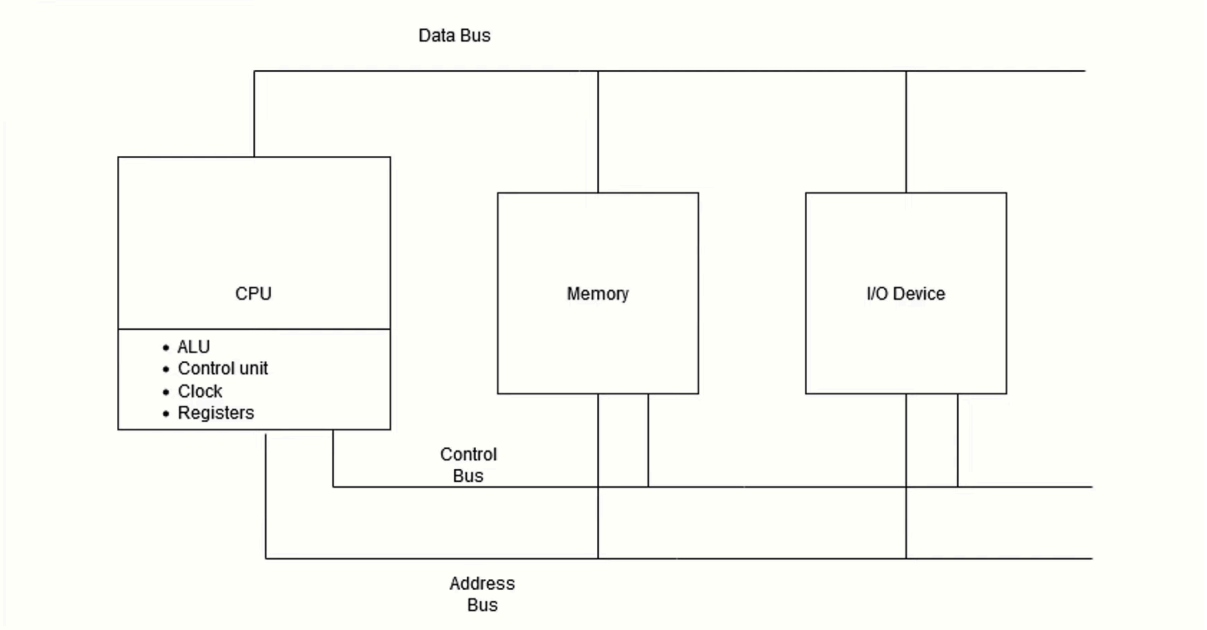
Example: 0100 1101 represent in hex

0100 → 4, 1101 → 13 = D
therefore 0100 1101 = 4D

---

4. Basics of x86 Architecture

x86 Processor Architecture:



Control Bus: Used to synchronize all action between devices attached to bus
Address Bus: Holds the address of instruction in data that are being transferred between cpu and memory or any other device, more like a pointer → where the data is moving
Data bus: Handles transfer of instruction and data between cpu and memory or any other device
I/O bus: takes care of all the inputs and outputs

CPU Components:

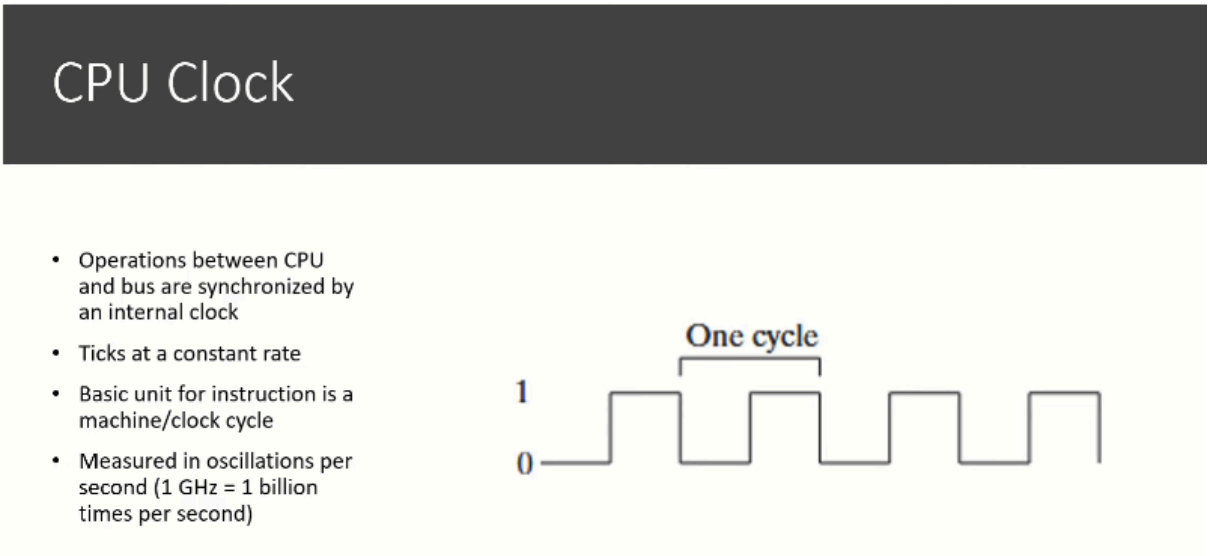
- **High frequency clock**
- **A control unit**
- **ALU (Arithmetic logical unit)**
- **Storage locations known as registers**

1. Arithmetic Logical Unit (ANU): Carries out logic and arithmetic, operations like ADD, OR, NOT, etc

2. Memory Registers: Type of computer memory close to the CPU, fastest way to access data

3. CPU Clock: Operation between CPU and Bus are synchronized by internal clock, Ticks at Constant rate, Basic

unit for instruction is machine/clock cycle, measured in oscillation per second (1Ghz = 1 billion oscillation / second)



4. Control Unit: Uses a binary decoder to convert coded instructions into timing and control signals, direct the operations of other units (memory, ALU, I/O)

Instruction Execution Cycle: CPU completes predefined set of steps to execute an instruction.

Steps:

1. **Fetch an instruction from instruction queue**
2. **Decode the instruction and check operand**
3. **If operands are involved, fetch it from memory / registers**
4. **Execute instruction and update status flags**
5. **Store the result if required**

Called a **Fetch Decode Execute** procedure

Reading From Memory:

Memory access is slower than register access due to following processes need to be followed:

1. Place the address of value we want to read on address bus

2. Changes the processor's RD pin (called assert)
3. Wait one clock cycle for memory to respond
4. Copy data from data bus to destination

Each steps take 1 clock cycle approx, Whereas registers access usually takes only one clock cycle

Caching: To reduce read/write time for memory, caches are used

In x86: Level-1 cache → stored on the CPU
Level-2 cache → stored outside and access by high-speed data bus

Constructed using static RAM, which does not need to be refreshed constantly making it more efficient, but it is more expensive.

---

5. Processor Modes and Registers

Modes of Operation:

- **Protected Mode:** The native processor state → default state, restricts process at their own section, can't interfere any other process
- **Read Address Mode:** Implements early programming enviroment with ability to switch modes → suitable and useful for hardware
- **System Management Mode:** Provides an OS with mechanisms for power management and security

Register Fundamentals:

x86 is 32-bit processor, each register = 32 bits in size

Example registers: EAX, EBX, ECX, EXD → E = Extended

- 16 bits can be accessed by dropping E → AX, BX, CX, DX
- 8-bit high can be accessed as AH, BH, CH, DH → leftmost (**0000** 0010)
- 8-bit low can be accessed as AL, BL, CL, DL → rightmost (0000 **0010**)

Registers:

Name	Description
eax	Extended accumulator, automatically used by multiplication and division instructions
ebx	General purpose
ecx	Used as a loop counter by the CPU
edx	General purpose
esi	High speed memory transfer
edi	High speed memory transfer
ebp	Used to reference function parameters and local variables on the stack
esp	A pointer to the current stack address

MOV (destination, source), Ex: MOV eax,1 → 1 is moved into register eax

stack → memory (RAM memory)

Special Purpose Registers:

- **EIP:** Instruction pointer, points to address of next instruction
- **EFLAGS:** Flags to denote the status of an operation:
  1. CF (Carry Flag)
  2. OF (Overflow Flag)
  3. SF (Sign Flag)
  4. ZF (Zero Flag)
  5. AC (Auxiliary Carry)
  6. PF (Parity Flag)

---

6. Making your First Program

Assembly language extensions can be `.asm` , `.s` or `.S` , `.nasm` , `.mlc`

Assembly code:

1. Open terminal and type the code $ `nano first.s` → to create a code in asm (first.s is file name)

code:

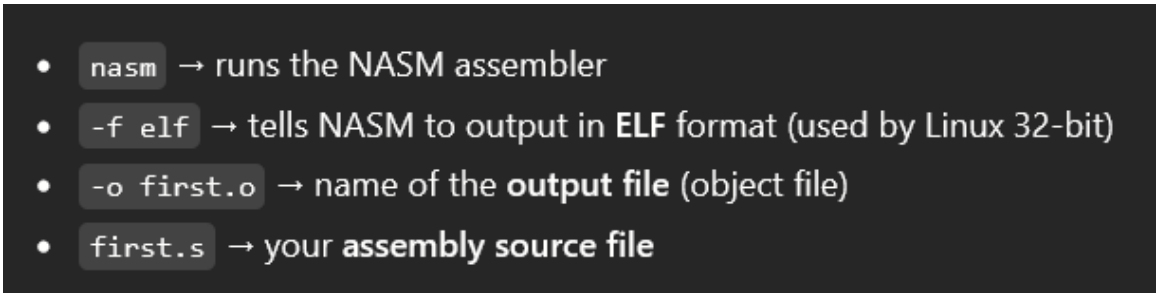
`section .data` → stores variables in memory when program runs

`section .text` → actual code of program

`global _start` → indicates start point (_start → label/segment)
`_start:` → everything below it executes

`MOV eax,1` → **tells OS what system call we want to do** when **INT 80h** is called, 1 indicates we run exit system call → ends program and sets exit status code → indicates if program was successful or not, status code is whatever we put in ebx, here status code = 1
`MOV ebx,1` → **output of status code**
`INT 80h` → **system interrupt, calls OS telling we need smth**, h = hex, 80h = 80 in hexadecimal

2. To compile the program, we need 2 instructions, `nasm -f elf -o first.o first.s` compiles **first.s i**nto object file (**first.o**)



3. Load the object file using `ld -m elf_i386 -o first first.o`

- `ld` → the linker (combines object files into an executable)
- `-m elf_i386` → tells it to use the **32-bit ELF format**
- `-o first` → name of the **output executable**
- `first.o` → your **input object file** (from NASM)

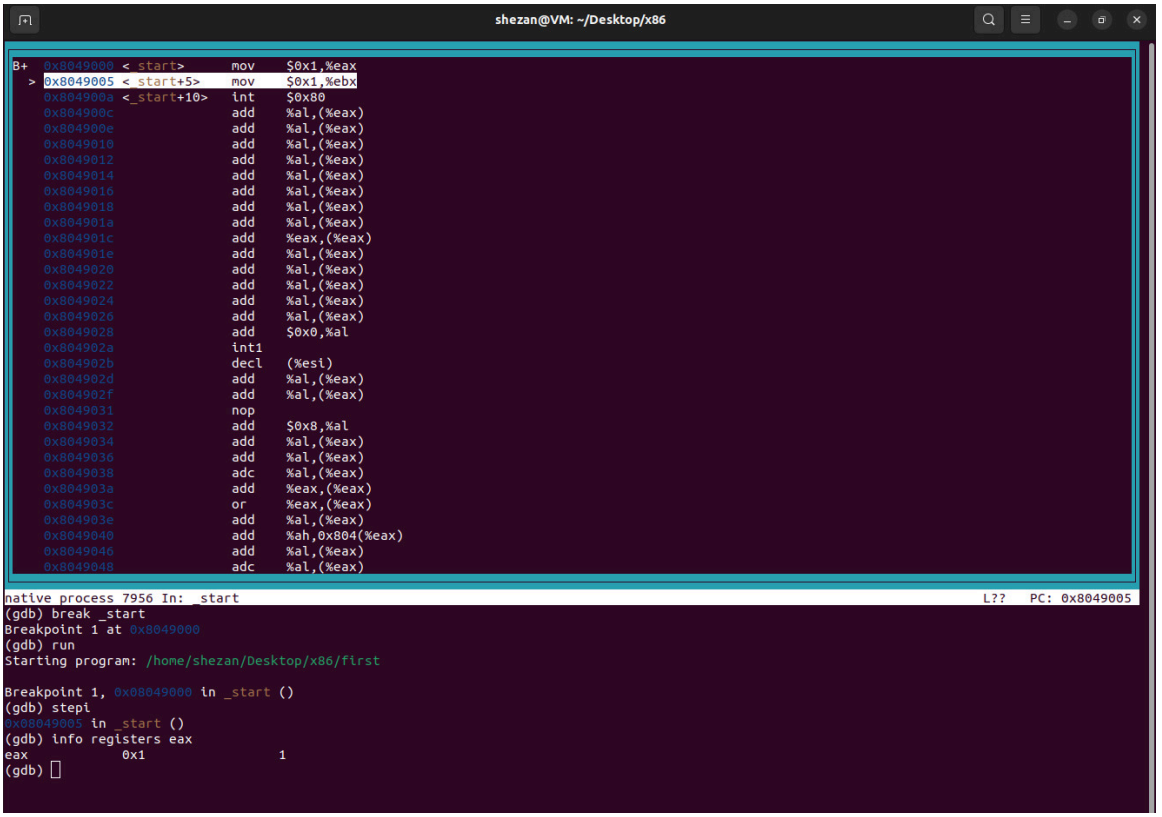
elf_i386 → compiling it specifically in x86 in 32 bit

4. `ls` to check our `first` file we created and run it by `./first` to see exit code, use `echo $?`

we got o/p = 1, because in **ebx** we put it as 1, if we change it the output changes

5. To use the debugger to debug the program use `gdb first` then if we type `layout asm` it puts us in assembly mode, showing all info related to assembly program

- put a break point at **_start** `break _start` → this tells debuggers to stop right as it hits that section
- type `run` to reach the breakpoint
- use `stepi` (an incremental step, moves 1 instruction forward),
- use `info registers` `eax` `(register name)` to see what value it has
- we can execute step by step and see what happens



7. Working with Data and Stack Memory

1. Create a new file (`nano data.s`)

```
  GNU nano 6.2                                    data.s
section .data
        num DD 5

section .text
global _start

_start:
        MOV eax,1
        MOV ebx,num
        INT 80h
```

`num` `DD` `5` → 32 bits are allocated to the value 5

```
shezan@VM:~/Desktop/x86/Storing Numeric Data$ echo $?
0
```

0 because we moved “num” so that stores the literal address not the value

```
B+  0x8049000 <_start>      mov    $0x1,%eax
    0x8049005 <_start+5>    mov    $0x804a000,%ebx
  > 0x804900a <_start+10>   int    $0x80
    0x804900c               add    %al,(%eax)
    0x804900e               add    %al,(%eax)
    0x8049010               add    %al,(%eax)
    0x8049012               add    %al,(%eax)
    0x8049014               add    %al,(%eax)
    0x8049016               add    %al,(%eax)
    0x8049018               add    %al,(%eax)
    0x804901a               add    %al,(%eax)
    0x804901c               add    %al,(%eax)
    0x804901e               add    %al,(%eax)
    0x8049020               add    %al,(%eax)
    0x8049022               add    %al,(%eax)
    0x8049024               add    %al,(%eax)
    0x8049026               add    %al,(%eax)
    0x8049028               add    %al,(%eax)
    0x804902a               add    %al,(%eax)
    0x804902c               add    %al,(%eax)
    0x804902e               add    %al,(%eax)
    0x8049030               add    %al,(%eax)
    0x8049032               add    %al,(%eax)
    0x8049034               add    %al,(%eax)
    0x8049036               add    %al,(%eax)
    0x8049038               add    %al,(%eax)
    0x804903a               add    %al,(%eax)
    0x804903c               add    %al,(%eax)
    0x804903e               add    %al,(%eax)
    0x8049040               add    %al,(%eax)
    0x8049042               add    %al,(%eax)
    0x8049044               add    %al,(%eax)
    0x8049046               add    %al,(%eax)
native process 4513 In: _start                                 L??
(gdb) break _start
Breakpoint 1 at 0x8049000
(gdb) run
Starting program: /home/shezan/Desktop/x86/Storing Numeric Data/data

Breakpoint 1, 0x08049000 in _start ()
(gdb) stepi
0x08049005 in _start ()
(gdb) stepi
0x0804900a in _start ()
(gdb) info register ebx
ebx            0x804a000           134520832
(gdb) x/x $ebx
0x804a000:      0x00000005
(gdb)
```

`x/x $ebx` , `x` → “examine memory” command, `/x` → format specifier: display in hex, `$ebx` → the address to look at (the value inside the EBX register)

here at `ebx` , the address is stored for the variable, to store the value we need to use `MOV ebx,[num]`

```
  GNU nano 6.2                                                  data.s
section .data
        num DD 5

section .text
global _start

_start:
        MOV eax,1
        MOV ebx,[num]
        INT 80h
```

now, `ebx` stores the value 5

```
shezan@VM:~/Desktop/x86/Storing Numeric Data$ ./data
shezan@VM:~/Desktop/x86/Storing Numeric Data$ echo $?
5
shezan@VM:~/Desktop/x86/Storing Numeric Data$
```

8. Working with Byte and Word Data

```
  GNU nano 6.2                                                  data.s *
section .data
        num1 DB 1
        num2 DB 2

section .text
global _start

_start:
        MOV ebx,[num1]
        MOV ecx,[num2]
        MOV eax,1
        INT 80h
```

num1 and num2 are stored next to each other in memory (after 1 byte)

```
B+  0x8049000 <_start>      mov    0x804a000,%ebx
    0x8049006 <_start+6>    mov    0x804a001,%ecx
  > 0x804900c <_start+12>   mov    $0x1,%eax
    0x8049011 <_start+17>   int    $0x80
    0x8049013               add    %al,(%eax)
    0x8049015               add    %al,(%eax)
    0x8049017               add    %al,(%eax)
    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)
    0x8049037               add    %al,(%eax)
    0x8049039               add    %al,(%eax)
    0x804903b               add    %al,(%eax)
    0x804903d               add    %al,(%eax)
    0x804903f               add    %al,(%eax)
    0x8049041               add    %al,(%eax)
    0x8049043               add    %al,(%eax)
    0x8049045               add    %al,(%eax)
    0x8049047               add    %al,(%eax)
    0x8049049               add    %al,(%eax)
    0x804904b               add    %al,(%eax)
native process 9608 In: _start                                          L??
(gdb) info register ebx
ebx            0x201               513
(gdb) x/x 0x804a000
0x804a000:      0x00000201
(gdb) stepi
0x0804900c in _start ()
(gdb) info register ecx
ecx            0x2                 2
```

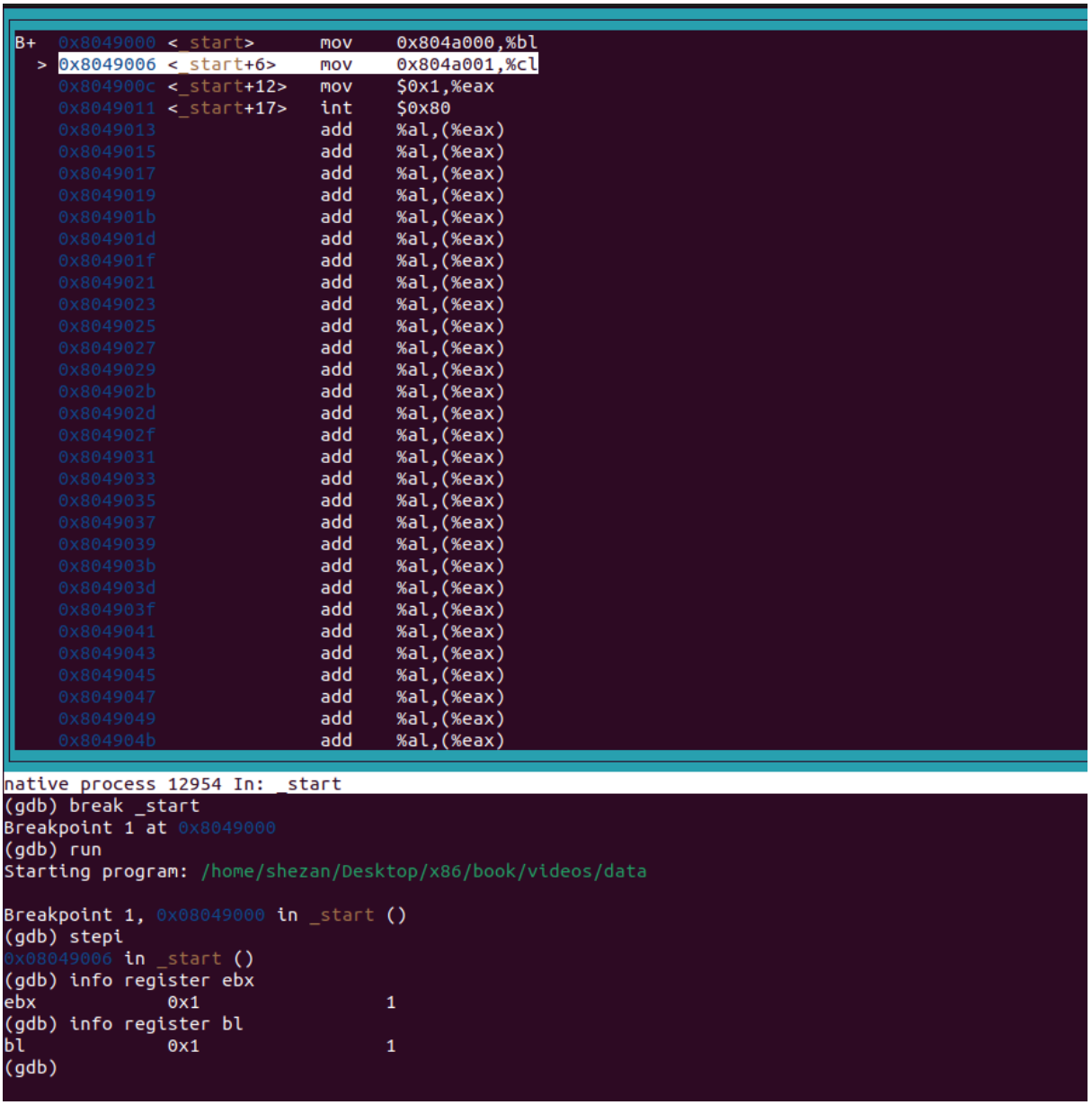
we get value of `ebx` as **513** because `ebx` and `ecx` are 32-bit registers and since `num1` and `num2` are stored a bit apart, the memory is `[01] [02] [00] [00] [00]...` → `[00]` = 1 byte and 32 bits = 4 bytes therefore, the `ebx` fetches the values of 4 bytes or

32 bits, making it's values from `num1` memory address to 3 more bits, therefore it is `0x00000201` and **201** in decimal is **513** therefore we get **513**
This does not happen with `ecx` because, `ecx` starts from `num2` memory address and therefore it gets `0x00000002` and we get **2**

In order to fix this we need to use low registers [`al` , `bl` , `cl` , `dl`]

```
  GNU nano 6.2                                                  data.s *
section .data
        num1 DB 1
        num2 DB 2

section .text
global _start

_start:
        MOV bl,[num1]
        MOV cl,[num2]
        MOV eax,1
        INT 80h
```

`bl` , `cl` uses low 8 bits (00000000 **<u>00000000</u>**) hence this solves the problem of over looking the memory values and sticks to just 8 bits, so num1 = `0x00000001` and since bl will go to 8 bits, it'll go till `[01]` solving the overlooking problem or operand size mismatch



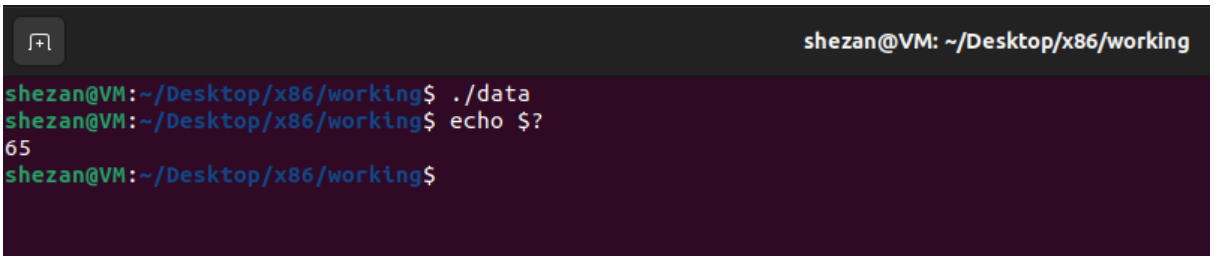
Here we can see that this fixed out issue, if we use high register, the value changes as the low bit goes from $2^0 \dots 2^7$ and high bit goes from $2^8 \dots 2^{15}$ so the value “1” at high bit = $1 \times 2^8 = 256$

```
B+  0x8049000 <_start>      mov    0x804a000,%bh
  > 0x8049006 <_start+6>    mov    0x804a001,%ch
    0x804900c <_start+12>   mov    $0x1,%eax
    0x8049011 <_start+17>   int    $0x80
    0x8049013               add    %al,(%eax)
    0x8049015               add    %al,(%eax)
    0x8049017               add    %al,(%eax)
    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)
    0x8049037               add    %al,(%eax)
    0x8049039               add    %al,(%eax)
    0x804903b               add    %al,(%eax)
    0x804903d               add    %al,(%eax)
    0x804903f               add    %al,(%eax)
    0x8049041               add    %al,(%eax)
    0x8049043               add    %al,(%eax)
    0x8049045               add    %al,(%eax)
    0x8049047               add    %al,(%eax)
    0x8049049               add    %al,(%eax)
    0x804904b               add    %al,(%eax)
native process 13616 In: _start
(gdb) break _start
Breakpoint 1 at 0x8049000
(gdb) run
Starting program: /home/shezan/Desktop/x86/book/videos/data

Breakpoint 1, 0x08049000 in _start ()
(gdb) stepi
0x08049006 in _start ()
(gdb) info registers bl
bl             0x0                0
(gdb) info registers bh
bh             0x1                1
(gdb) info registers ebx
ebx            0x100              256
```

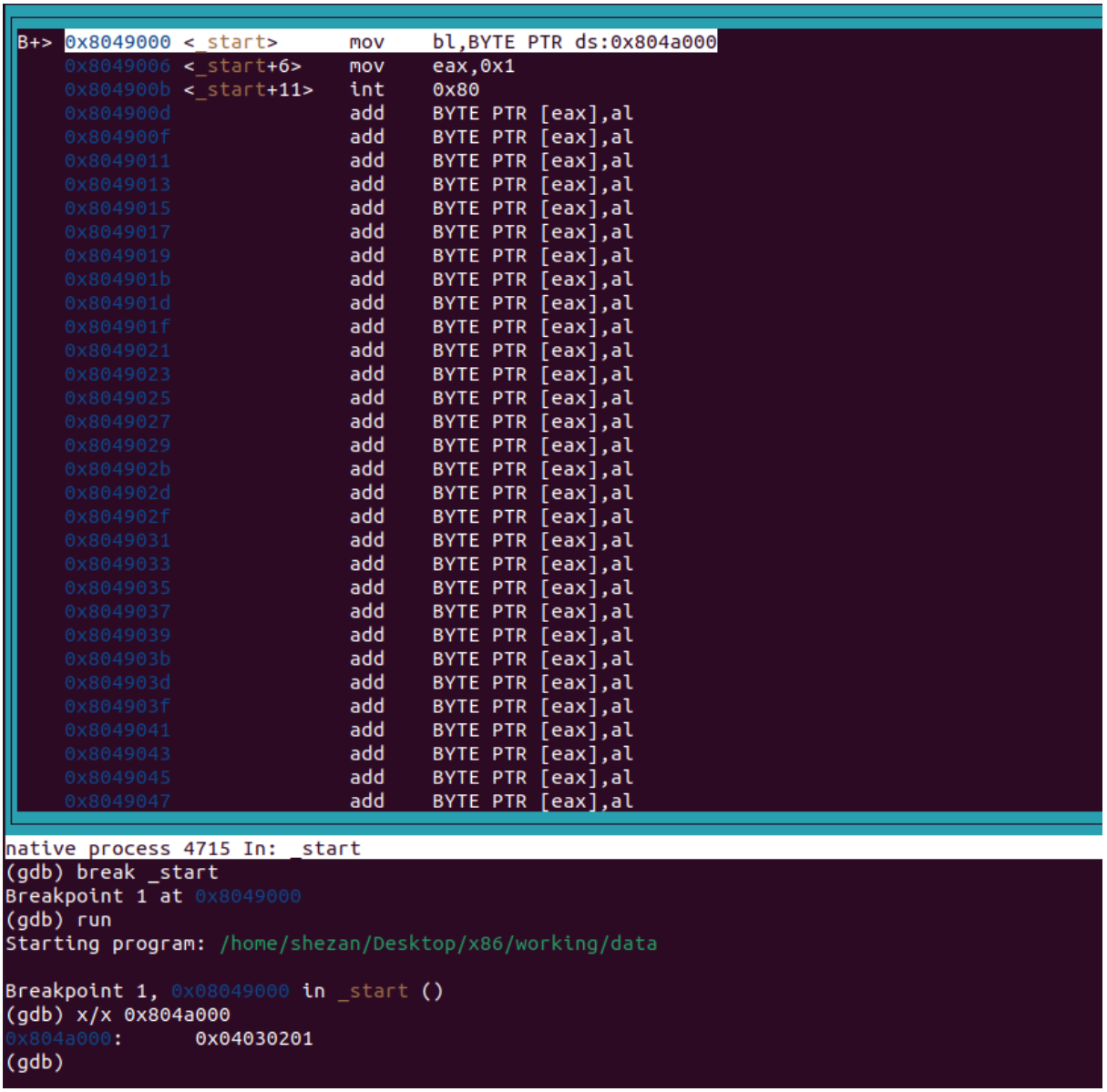
9. Characters, Strings and Lists

```
  GNU nano 6.2                                                  data.s
section .data
        char DB 'A'

section .text
global _start

_start:
        MOV bl,[char]
        MOV eax,1
        INT 80h
```

Here, in this code we are storing ‘A’ in `char` variable with it’s size as `defined byte` or `1 byte`, when we run this code and `echo $?` its output, it’ll print the ASCII encoding value of ‘A’ (i.e **65**)



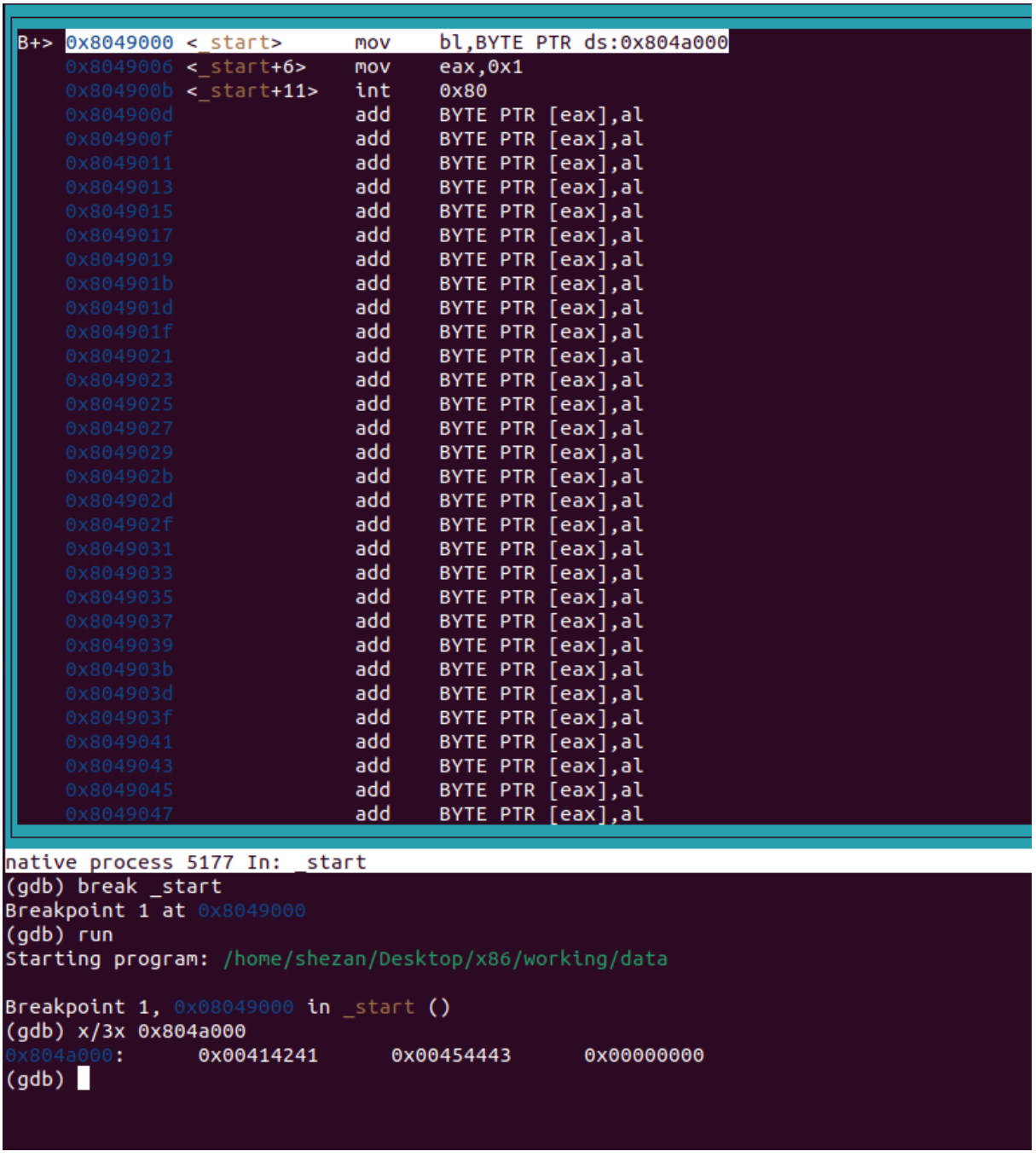
The `list DB 1,2,3,4` stores all these nos in 8-bit sequence each so, the memory segment becomes `[01] [02] [03] [04] [00]...[nn]`, Therefore the value at the pointer of list becomes 0x04030201



memory around this will be mostly initialized to `[00]`, now we don't know how long our list is (end?) actually which is a major issue in x86, to overcome this, we need to add something that indicates the end of list, for instance, if we are dealing with positive values, we can add `-1` to indicate end of list. We can put `0` if we know there are **no 0(s)** in the list, this is called `null terminator`.

```
  GNU nano 6.2                                                  data.s
section .data
        string1 DB "ABA",0
        string2 DB "CDE",0

section .text
global _start

_start:
        MOV bl,[string1]
        MOV eax,1
        INT 80h
```

Here we add `0` as our null terminator to indicate that the string ends at ‘A’ and ‘E’



Here we can see how adding 0 after the list affected in showing that string ended, the output should have been `0x45414241` but since we added 0 as null terminator, the output becomes `0x00414241` and same for `0x00454443` (These are the HEX of ASCII), because 0 as null terminator makes the memory segment like `[41] [42] [41] [00] [43] ... [45]` instead of `[41] [42] [41] [43] ... [45]`, this helps us understand that the string ended at `[41]`

10. **Declaring Uninitialized Data**

`section .bss` : **Block Started by Symbol**, **memory section** in an executable that **reserves space for uninitialized variables**

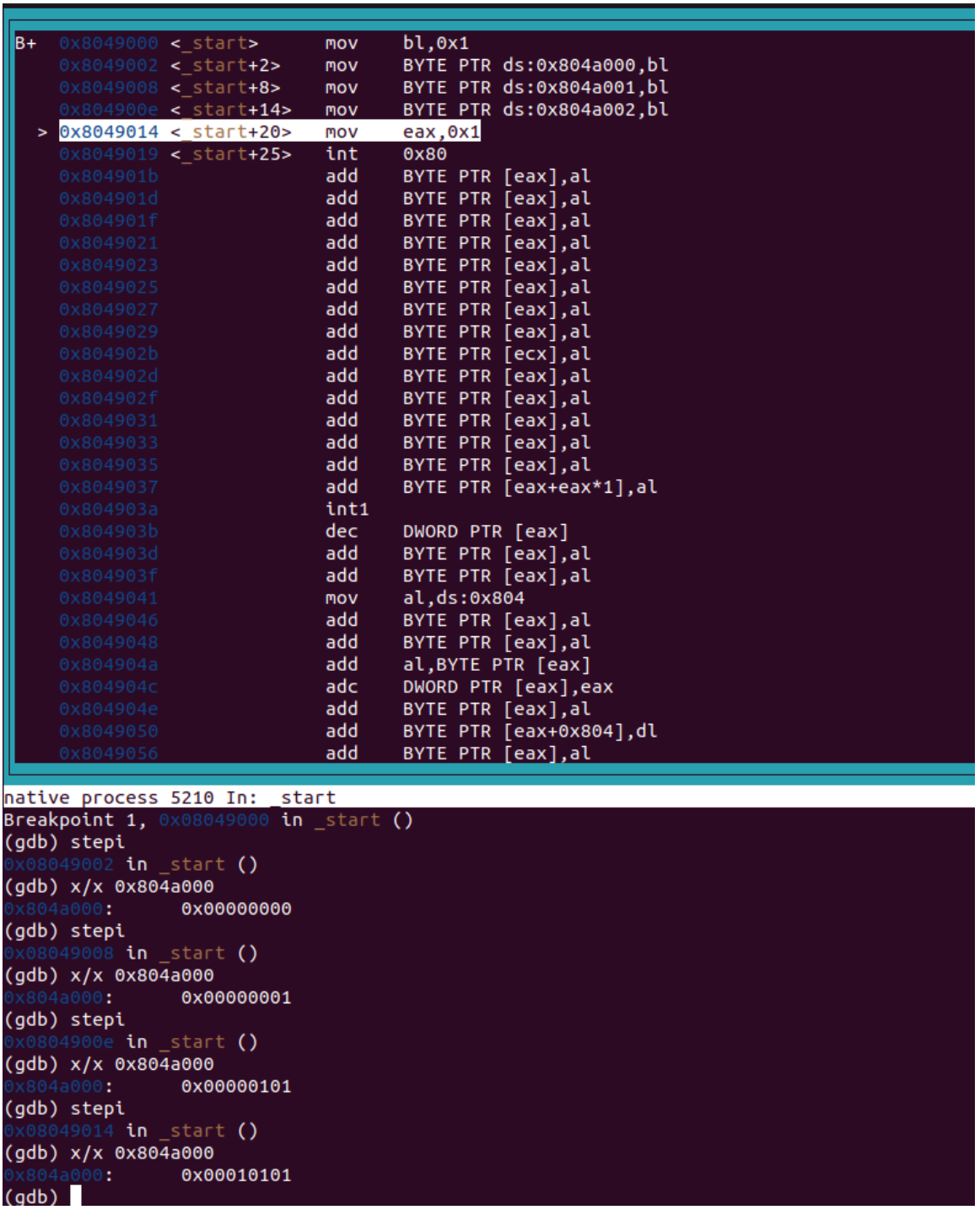
`num RESB 3` = Reserve Bytes

```
 GNU nano 6.2                                                  unit.s *
section .bss
        num RESB 3

section .text
global _start

_start:
        MOV bl,1
        MOV [num],bl
```

we can't directly place value into memory slot just like we can't remove data from register coz x86 doesn't have understanding of how big is all that, so move the data we 1st move value into bl and then move bl into [num], because bl is byte of register B, it understands that we are working with byte sized data

```
 GNU nano 6.2                                                  unit.s *
section .bss
        num RESB 3

section .text
global _start

_start:
        MOV bl,1
        MOV [num],bl
        MOV [num+1],bl
        MOV [num+2],bl

        MOV eax
        INT 80h
```

[num] points to 1st memory location, to get to second we use [num+1] and so on



Here we can see that [num] executes and gives us the output as `0x000000 01` this happens because 1 memory segment = [00] therefore here it was [01] after that we moved the value of bl into num+1 that gives us the output as `0x00000101` since it moves 1 segment ahead and same goes for [num+2]

```
 GNU nano 6.2                                                    unit.s
section .bss
        num RESB 3

section .data
        num2 DB 3 DUP(2)

section .text

global _start

_start:
        MOV bl,1
        MOV bl,[num2]
```

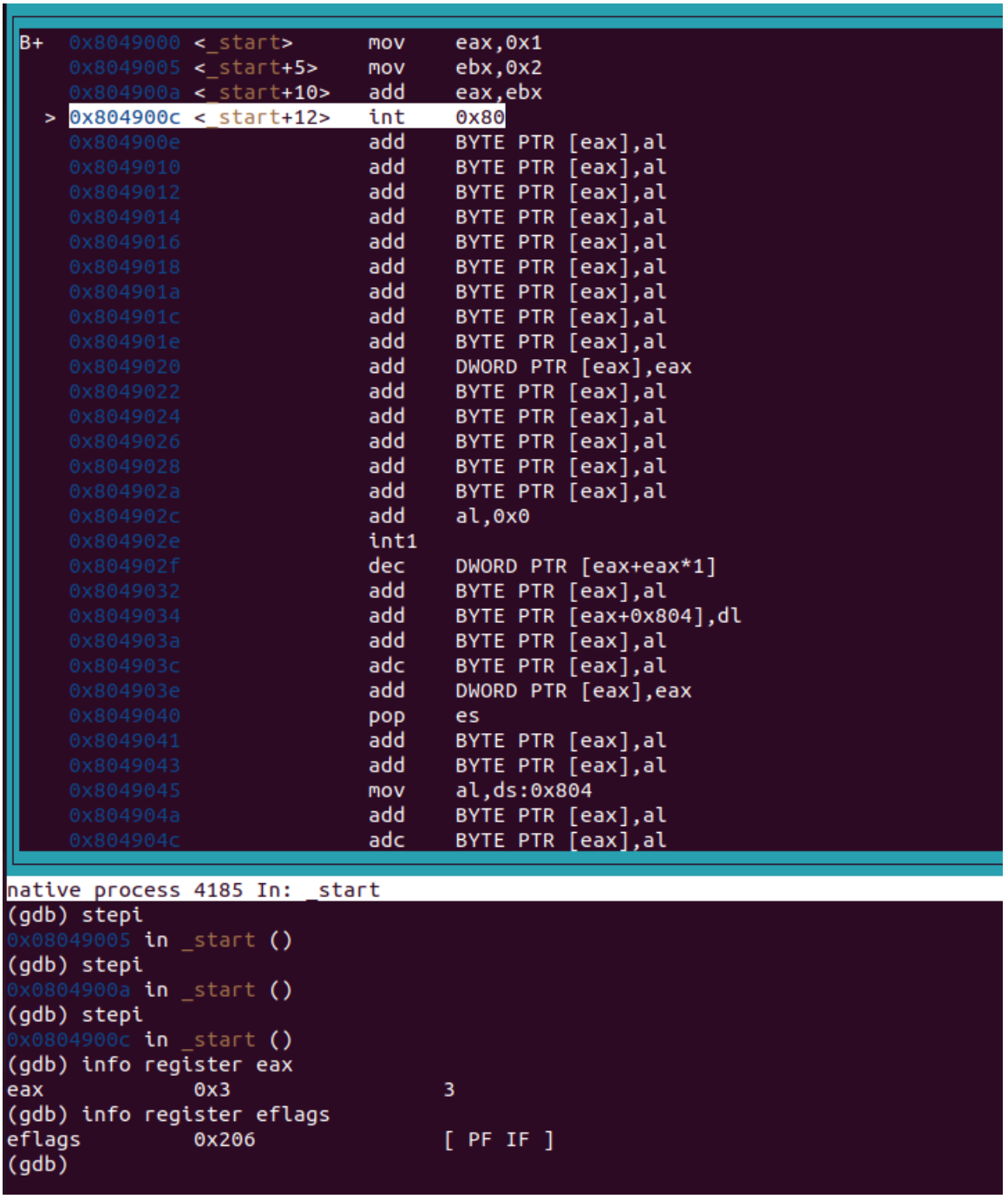
`num2 DB 3 DUP (2)` → defines 3 bytes, each initialized with the value 2 (i.e., 02 02 02). DUP → Duplicate.
It expands into `num2 DB 2, 2, 2`

```
B+> 0x8049000 <_start>       mov    bl,0x1
    0x8049002 <_start+2>     mov    bl,BYTE PTR ds:0x804a000
    0x8049008 <_start+8>     mov    BYTE PTR ds:0x804a004,bl
    0x804900e <_start+14>    mov    BYTE PTR ds:0x804a005,bl
    0x8049014 <_start+20>    mov    BYTE PTR ds:0x804a006,bl
    0x804901a <_start+26>    mov    eax,0x1
    0x804901f <_start+31>    int    0x80
    0x8049021                add    BYTE PTR [eax],al
    0x8049023                add    BYTE PTR [eax],al
    0x8049025                add    BYTE PTR [eax],al
    0x8049027                add    BYTE PTR [eax],al
    0x8049029                add    BYTE PTR [eax],al
    0x804902b                add    BYTE PTR [eax],al
    0x804902d                add    BYTE PTR [eax],al
    0x804902f                add    BYTE PTR [eax],al
    0x8049031                add    BYTE PTR [eax],al
    0x8049033                add    BYTE PTR [eax],al
    0x8049035                add    BYTE PTR [eax],al
    0x8049037                add    BYTE PTR [eax],al
    0x8049039                add    BYTE PTR [eax],al
    0x804903b                add    BYTE PTR [eax],al
    0x804903d                add    BYTE PTR [eax],al
    0x804903f                add    BYTE PTR [eax],al
    0x8049041                add    BYTE PTR [eax],al
    0x8049043                add    BYTE PTR [eax],al
    0x8049045                add    BYTE PTR [eax],al
    0x8049047                add    BYTE PTR [eax],al
    0x8049049                add    BYTE PTR [eax],al
    0x804904b                add    BYTE PTR [eax],al
    0x804904d                add    BYTE PTR [eax],al
    0x804904f                add    BYTE PTR [eax],al
    0x8049051                add    BYTE PTR [eax],al
    0x8049053                add    BYTE PTR [eax],al
native process 5937 In: _start
(gdb) x/x 0x804a000
0x804a000:      0x00020202
```

11. ADD, ADC, and EFLAGS

```
 GNU nano 6.2                                                    add.s
section .data


section .text
global _start

_start:
        MOV eax,1
        MOV ebx,2
        ADD eax,ebx
        INT 80h
```

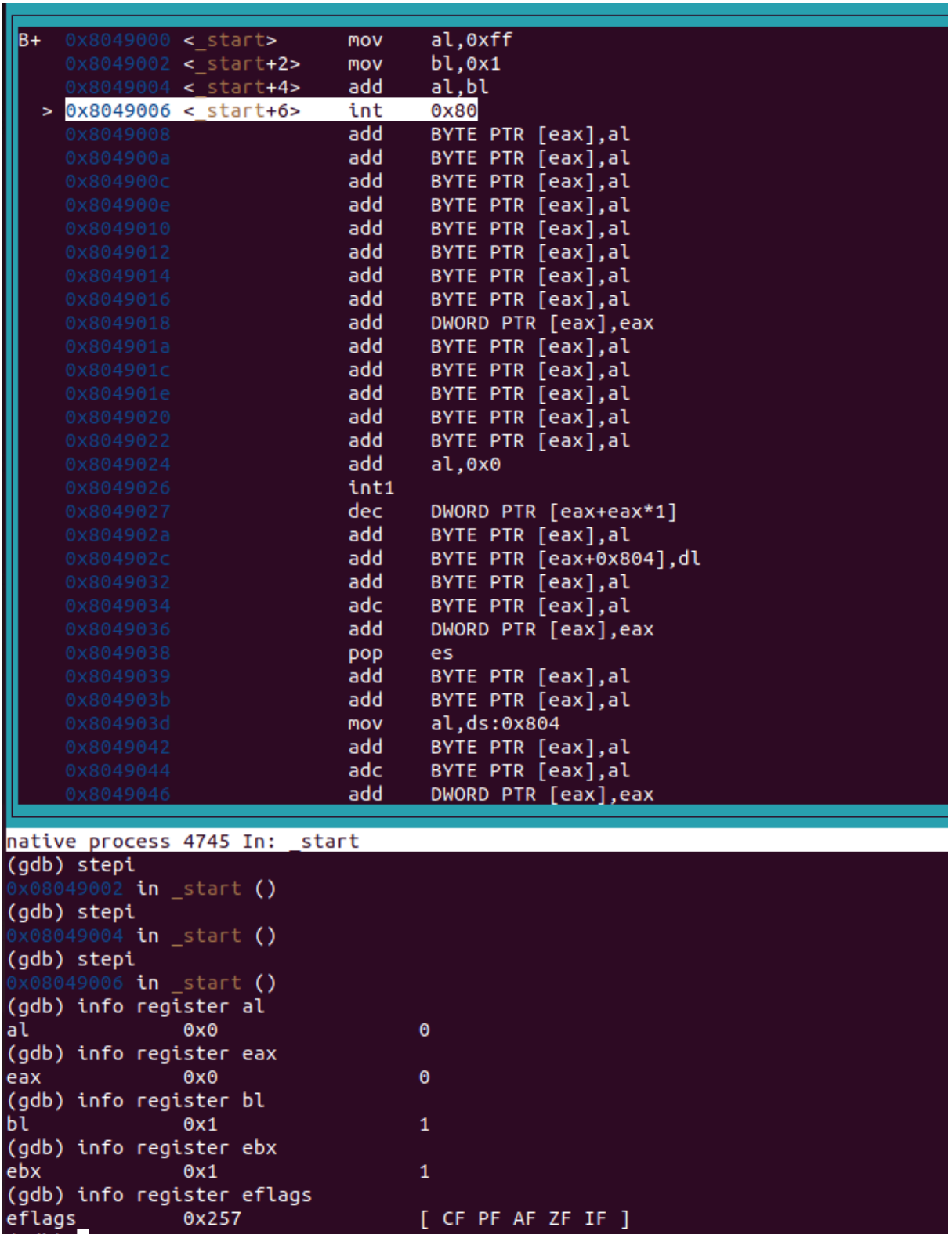
Here, we are adding 2 values, we initialize the value of a register as 1 and b as 2, then we add the values 1 & 2 and then store it in a register (`ADD eax,ebx`) this will change the eax value to 3 and hence if we try and run it using `./add` it will show **segmentation fault (core dumped)** error, to fix this, we need to add `MOV eax,1` before calling `INT 80h`



Here the added value 3 is successfully stored in `eax` and we can also see EFLAGS in use, here `eflags` shows us information about the operation just ran, the `[PF IF]` shows what flags are set on, we can also see this using `0x206` converting this into binary and checking, ***more info***

```
 GNU nano 6.2                                                    add.s
section .data


section .text
global _start

_start:
        MOV al,0b11111111
        MOV bl,0b0001
        ADD al,bl
        INT 80h
```

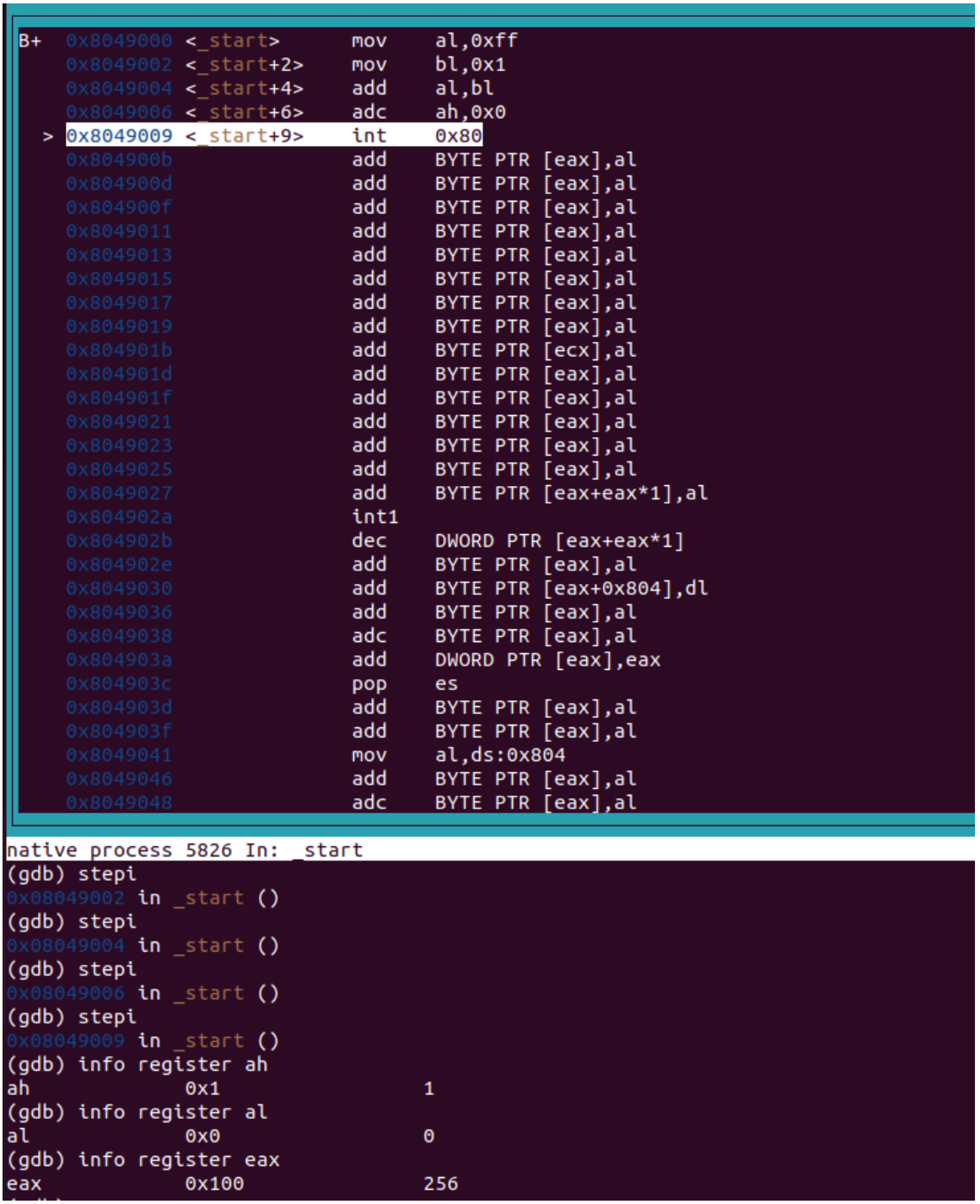
Here, `0b` represents that the number is in binary format `9b11111111` → 255, `0b0001` → 1



when we add `11111111 + 00000001` we get answer as **1**0000 0000, therefore carry flag is set `[CF PF AF ZF IF]`

```
 GNU nano 6.2                                                    add.s
section .data


section .text
global _start

_start:
        MOV al,0b11111111
        MOV bl,0b0001
        ADD al,bl
        ADC ah,0
        INT 80h
```

`ADC` or **Add Carry Register** is used to add the value from source to it's destination along with the value of `CF` or **Carry flag** (more like +1)



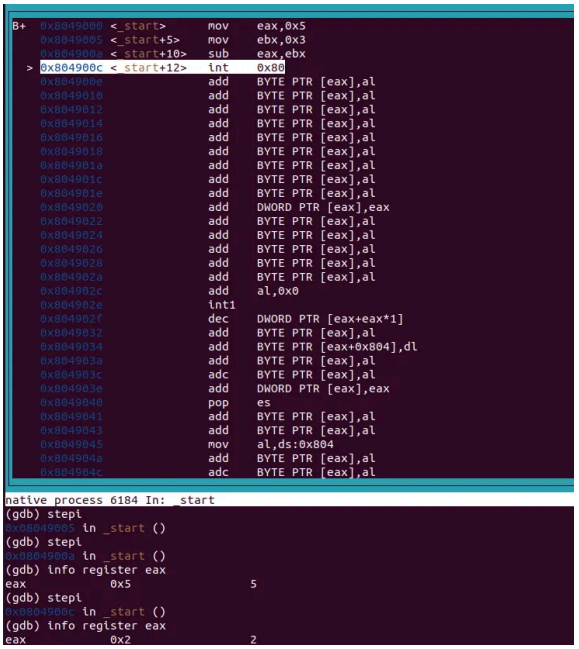
Here, value of `eax` becomes 256, because we added `255 + 1` to `al` , making it's value 0 but adding 0+1 (carry flag) to the `ah` makes the value 1, that is 2^8 (i.e **256**), therefore we get **256** in `eax`

---

12. Subtraction and Sign Flags

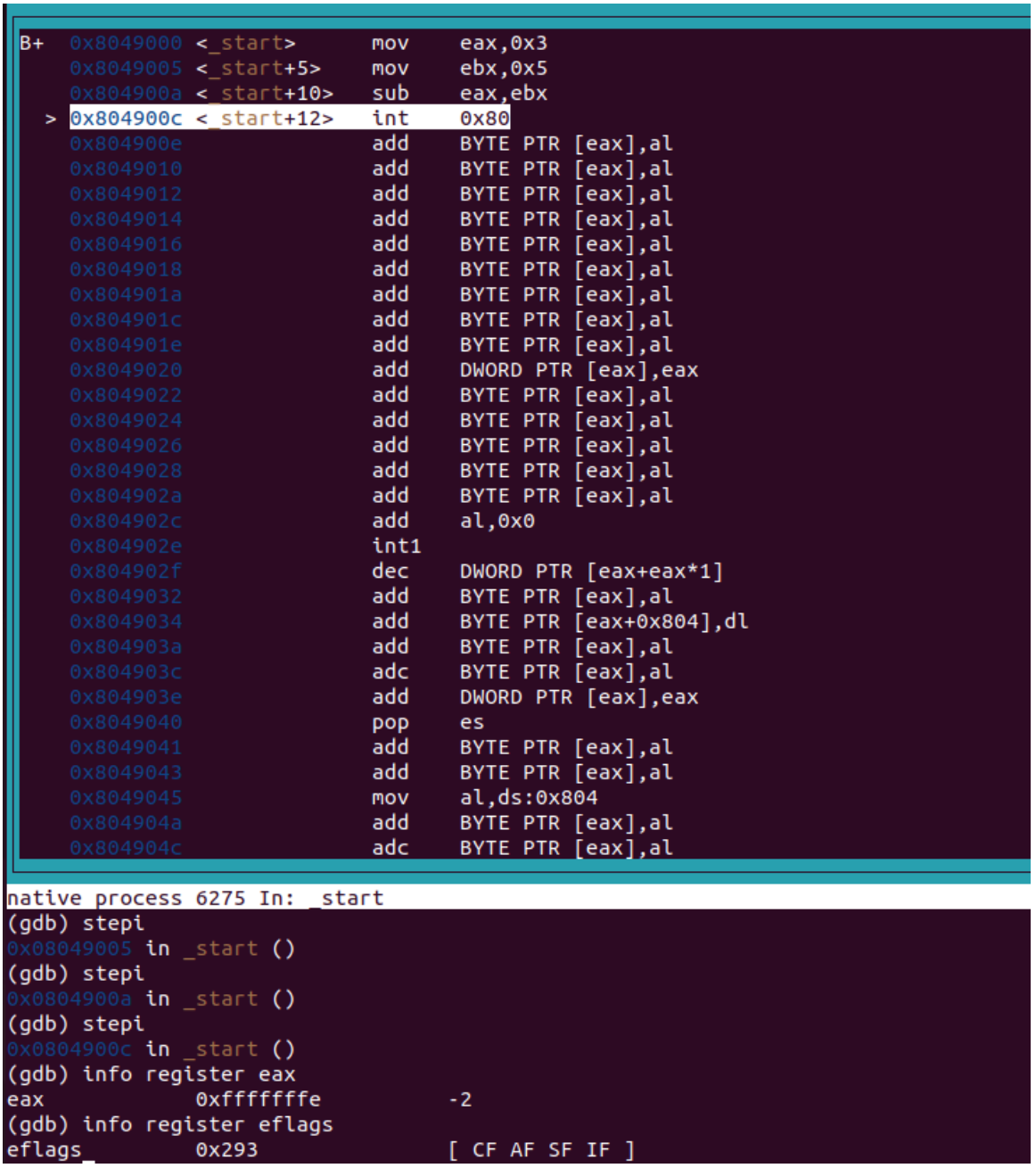
```
 GNU nano 6.2                                                    sub.s
section .data

section .text
global _start

_start:
        MOV eax,5
        MOV ebx,3
        SUB eax,ebx
        INT 80h
```

`SUB eax,ebx` → `eax - ebx`, and stores in `eax`




```
 GNU nano 6.2                                                    sub.s
section .data

section .text
global _start

_start:
        MOV eax,3
        MOV ebx,5
        SUB eax,ebx
        INT 80h
```

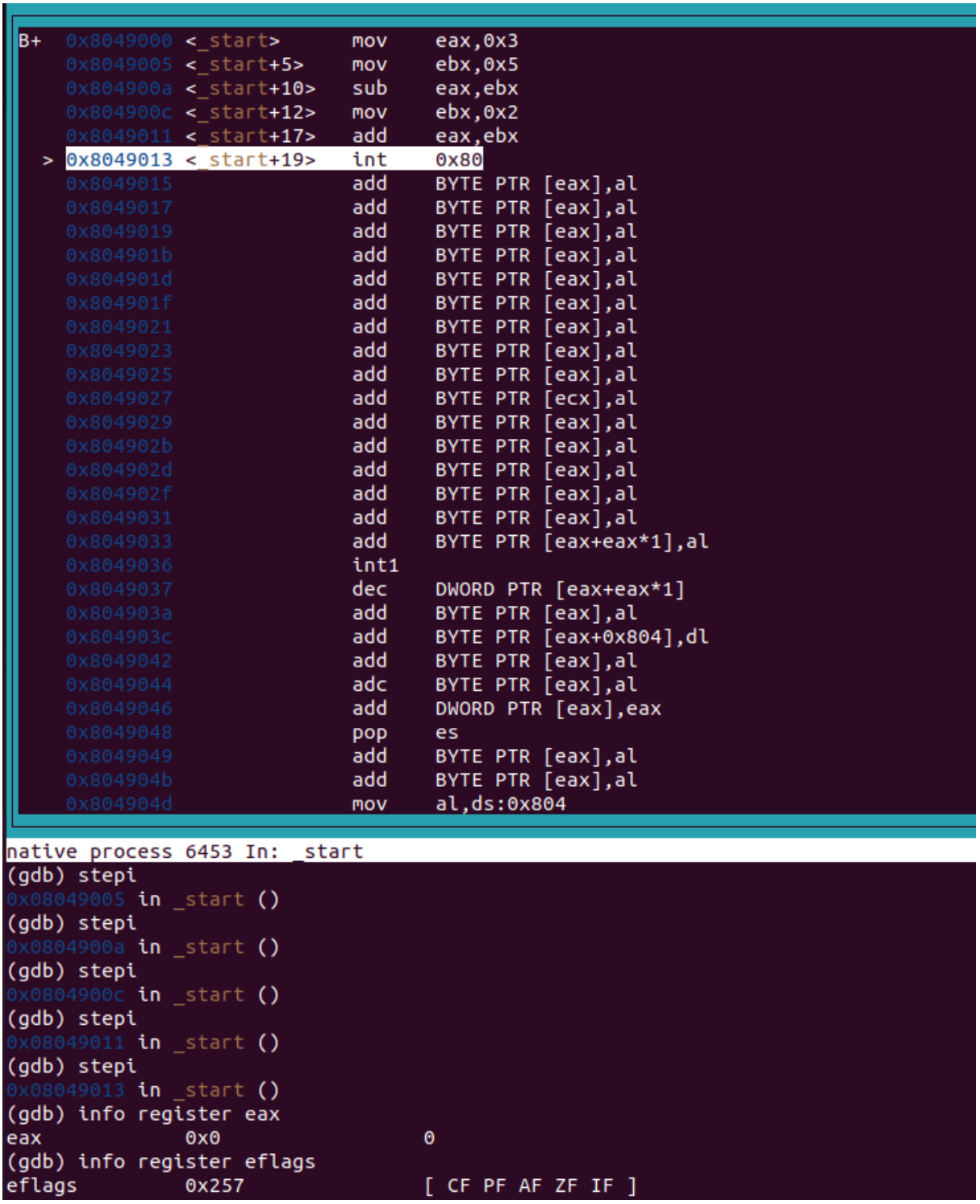
If we interchange the value and force it to have a negative number on subtraction, the output changes



How does the machine know that the output is -2, it can be a really big number, therefore `CF` or **Carry Flag** steps in, it serves 2 function, representing a carry and borrow in subtraction, we need to keep borrowing causing `0x fffffff e` now more imp is `SF` or **Sign flag**, this indicates that the operation produced a negative output (here, **-2**), that's how it knows that it's a negative number.

```
 GNU nano 6.2                                                    sub.s
section .data

section .text
global _start

_start:
        MOV eax,3
        MOV ebx,5
        SUB eax,ebx
        MOV ebx,2
        ADD eax,ebx
        INT 80h
```

This works, exactly like normal arithmetic, here we will be adding `ebx,2` to `eax` which is equal to -2



Here, we can see the output is 0 and it did not trigger the `SF` or **Sign Flag.**

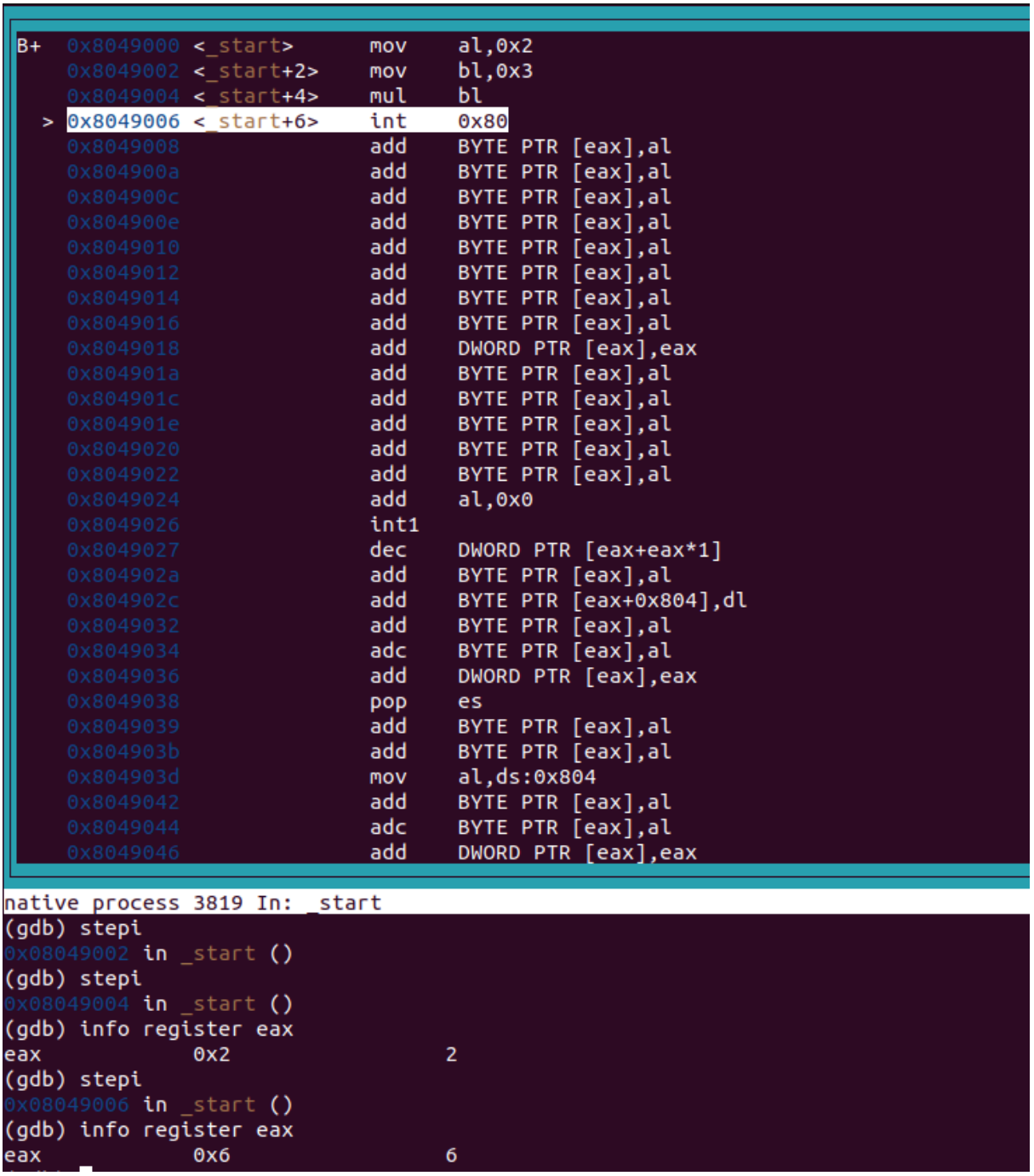
13. **Multiplying numbers with MUL and IMUL**

```
 GNU nano 6.2                                                    mul.s
section .text
global _start

_start:
        MOV al,2
        MOV bl,3
        MUL bl
        INT 80h
```

Here, we mention only one operand at `MUL`, `MUL` always uses **AL**, **AX**, or **EAX** as the default operand. Since they are the accumulator register, they are actually meant for arithmetic operation, hence they are used by default in **multiplication and division**

```
B+  0x8049000 <_start>      mov    al,0x2
    0x8049002 <_start+2>    mov    bl,0x3
    0x8049004 <_start+4>    mul    bl
  > 0x8049006 <_start+6>    int    0x80
    0x8049008               add    BYTE PTR [eax],al
    0x804900a               add    BYTE PTR [eax],al
    0x804900c               add    BYTE PTR [eax],al
    0x804900e               add    BYTE PTR [eax],al
    0x8049010               add    BYTE PTR [eax],al
    0x8049012               add    BYTE PTR [eax],al
    0x8049014               add    BYTE PTR [eax],al
    0x8049016               add    BYTE PTR [eax],al
    0x8049018               add    DWORD PTR [eax],eax
    0x804901a               add    BYTE PTR [eax],al
    0x804901c               add    BYTE PTR [eax],al
    0x804901e               add    BYTE PTR [eax],al
    0x8049020               add    BYTE PTR [eax],al
    0x8049022               add    BYTE PTR [eax],al
    0x8049024               add    al,0x0
    0x8049026               int1
    0x8049027               dec    DWORD PTR [eax+eax*1]
    0x804902a               add    BYTE PTR [eax],al
    0x804902c               add    BYTE PTR [eax+0x804],dl
    0x8049032               add    BYTE PTR [eax],al
    0x8049034               adc    BYTE PTR [eax],al
    0x8049036               add    DWORD PTR [eax],eax
    0x8049038               pop    es
    0x8049039               add    BYTE PTR [eax],al
    0x804903b               add    BYTE PTR [eax],al
    0x804903d               mov    al,ds:0x804
    0x8049042               add    BYTE PTR [eax],al
    0x8049044               adc    BYTE PTR [eax],al
    0x8049046               add    DWORD PTR [eax],eax
native process 3819 In:  _start
(gdb) stepi
0x08049002 in _start ()
(gdb) stepi
0x08049004 in _start ()
(gdb) info register eax
eax            0x2                 2
(gdb) stepi
0x08049006 in _start ()
(gdb) info register eax
eax            0x6                 6
```

Here we can see the value we store in `bl` was multiplied by value in `al` by default giving us the value



Here we can see the value we store in `bl` was multiplied by value in `al` by default giving us the value `0x6 / 6`

```
  GNU nano 6.2                                                    mul.s
section .text
global _start

_start:
        MOV al,0xFF
        MOV bl,2
        MUL bl
```

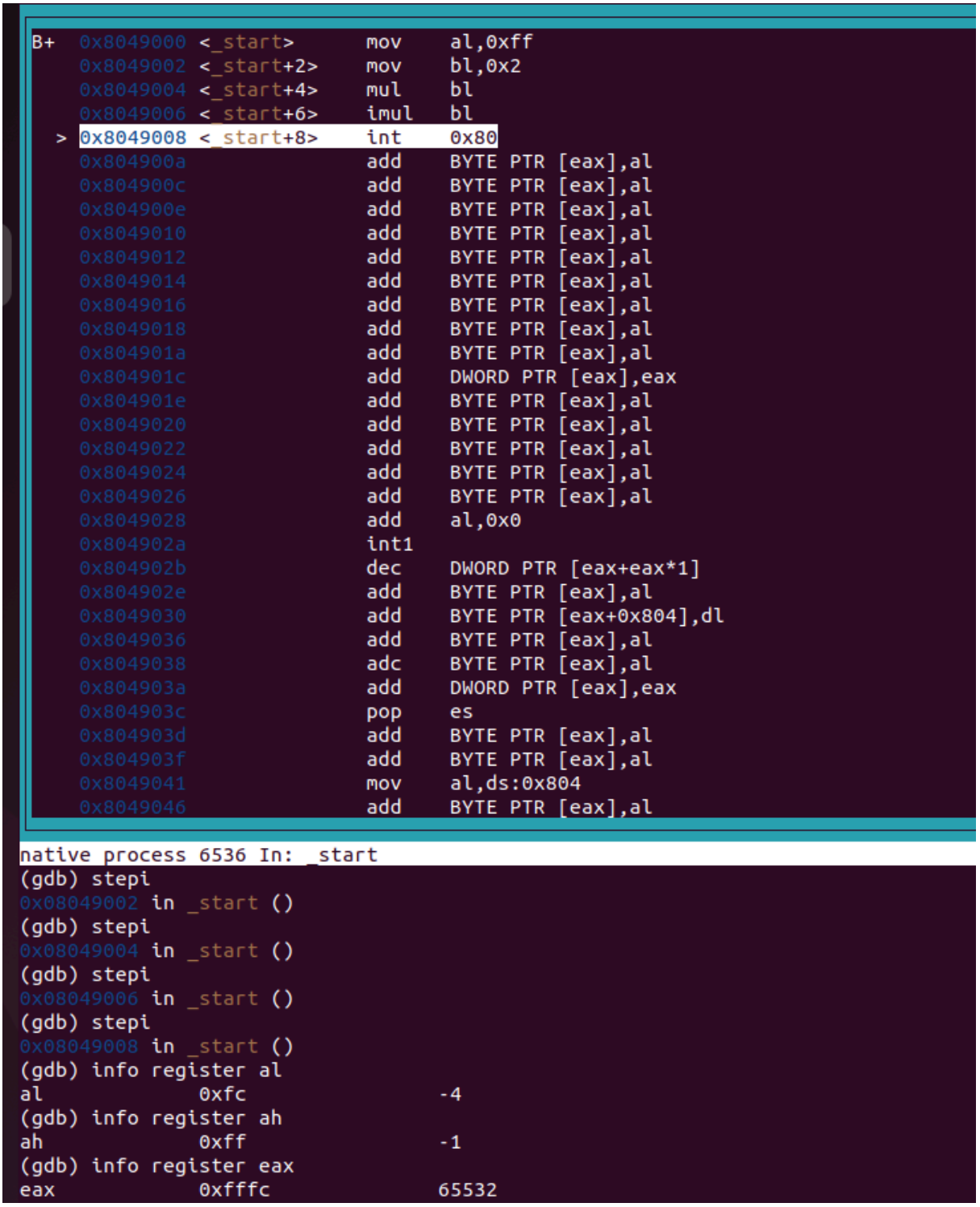
What happens if we multiply a bigger number that can exceed the register limit, `0xFF → 255`

```
B+  0x8049000 <_start>      mov    al,0xff
    0x8049002 <_start+2>    mov    bl,0x2
    0x8049004 <_start+4>    mul    bl
  > 0x8049006               add    BYTE PTR [eax],al
    0x8049008               add    BYTE PTR [eax],al
    0x804900a               add    BYTE PTR [eax],al
    0x804900c               add    BYTE PTR [eax],al
    0x804900e               add    BYTE PTR [eax],al
    0x8049010               add    BYTE PTR [eax],al
    0x8049012               add    BYTE PTR [eax],al
    0x8049014               add    BYTE PTR [eax],al
    0x8049016               add    BYTE PTR [eax],al
    0x8049018               add    DWORD PTR [eax],eax
    0x804901a               add    BYTE PTR [eax],al
    0x804901c               add    BYTE PTR [eax],al
    0x804901e               add    BYTE PTR [eax],al
    0x8049020               add    BYTE PTR [eax],al
    0x8049022               add    BYTE PTR [eax],al
    0x8049024               add    al,0x0
    0x8049026               int1
    0x8049027               dec    DWORD PTR [eax+eax*1]
    0x804902a               add    BYTE PTR [eax],al
    0x804902c               add    BYTE PTR [eax+0x804],dl
    0x8049032               add    BYTE PTR [eax],al
    0x8049034               adc    BYTE PTR [eax],al
    0x8049036               add    DWORD PTR [eax],eax
    0x8049038               pop    es
    0x8049039               add    BYTE PTR [eax],al
    0x804903b               add    BYTE PTR [eax],al
    0x804903d               mov    al,ds:0x804
    0x8049042               add    BYTE PTR [eax],al
    0x8049044               adc    BYTE PTR [eax],al
    0x8049046               add    DWORD PTR [eax],eax
native process 4019 In:
(gdb) stepi
0x08049002 in _start ()
(gdb) stepi
0x08049004 in _start ()
(gdb) stepi
0x08049006 in ?? ()
(gdb) info register al
al             0xfe               -2
(gdb) info register ah
ah             0x1                1
(gdb) info register eax
eax            0x1fe              510
```

It expands the size of the destination to make it large enough to store the value, here, `al` gets expanded into `eax` to store the value **510**, this is a major difference between multiplication and addition, in `add` we need to keep track of our output and assign accordingly, `mul` does that for us.

```
  GNU nano 6.2                                                  mul.s
section .text
global _start

_start:
        MOV al,0xFF
        MOV bl,2
        MUL bl
        IMUL bl
        INT 80h
```

Now let's use signed multiplication `IMUL` , first we multiply `-1 or 255` with 2 in bl, we get `0x1fe` as we saw above, since all this is using `al` , it can only store the low bits, therefore it stores the `0x01` in `ah` and `0xfe` in `al`



Here, we got `0x1fe` then it was `IMUL` by `bl` again, so `0x1fe * 2 = 0x3fc` here, `0xfc` is stored in `al` and `0x03` is discarded and instead `0xff` is stored in `ah` this is called sign extension

14. Dividing numbers with DIV and IDIV

```
 GNU nano 6.2                                                    div.s
section .text
global _start

_start:
        MOV eax,11
        MOV ecx,2
        DIV ecx
        INT 80h
```

The `DIV` works the same as the `MUL` operand, here, the result of the operation is stored in the `eax` register and the remainder is stored in the `edx` register

```
B+  0x8049000 <_start>      mov    eax,0xb
    0x8049005 <_start+5>    mov    ecx,0x2
    0x804900a <_start+10>   div    ecx
  > 0x804900c <_start+12>   int    0x80
    0x804900e               add    BYTE PTR [eax],al
    0x8049010               add    BYTE PTR [eax],al
    0x8049012               add    BYTE PTR [eax],al
    0x8049014               add    BYTE PTR [eax],al
    0x8049016               add    BYTE PTR [eax],al
    0x8049018               add    BYTE PTR [eax],al
    0x804901a               add    BYTE PTR [eax],al
    0x804901c               add    BYTE PTR [eax],al
    0x804901e               add    BYTE PTR [eax],al
    0x8049020               add    DWORD PTR [eax],eax
    0x8049022               add    BYTE PTR [eax],al
    0x8049024               add    BYTE PTR [eax],al
    0x8049026               add    BYTE PTR [eax],al
    0x8049028               add    BYTE PTR [eax],al
    0x804902a               add    BYTE PTR [eax],al
    0x804902c               add    al,0x0
    0x804902e               int1
    0x804902f               dec    DWORD PTR [eax+eax*1]
    0x8049032               add    BYTE PTR [eax],al
    0x8049034               add    BYTE PTR [eax+0x804],dl
    0x804903a               add    BYTE PTR [eax],al
    0x804903c               adc    BYTE PTR [eax],al
    0x804903e               add    DWORD PTR [eax],eax
    0x8049040               pop    es
    0x8049041               add    BYTE PTR [eax],al
    0x8049043               add    BYTE PTR [eax],al
    0x8049045               mov    al,ds:0x804
    0x804904a               add    BYTE PTR [eax],al
    0x804904c               adc    BYTE PTR [eax],al
native process 7713 In:  _start
(gdb) stepi
0x08049005 in _start ()
(gdb) stepi
0x0804900a in _start ()
(gdb) stepi
0x0804900c in _start ()
(gdb) info register eax
eax            0x5                 5
(gdb) info register edx
edx            0x1                 1
```

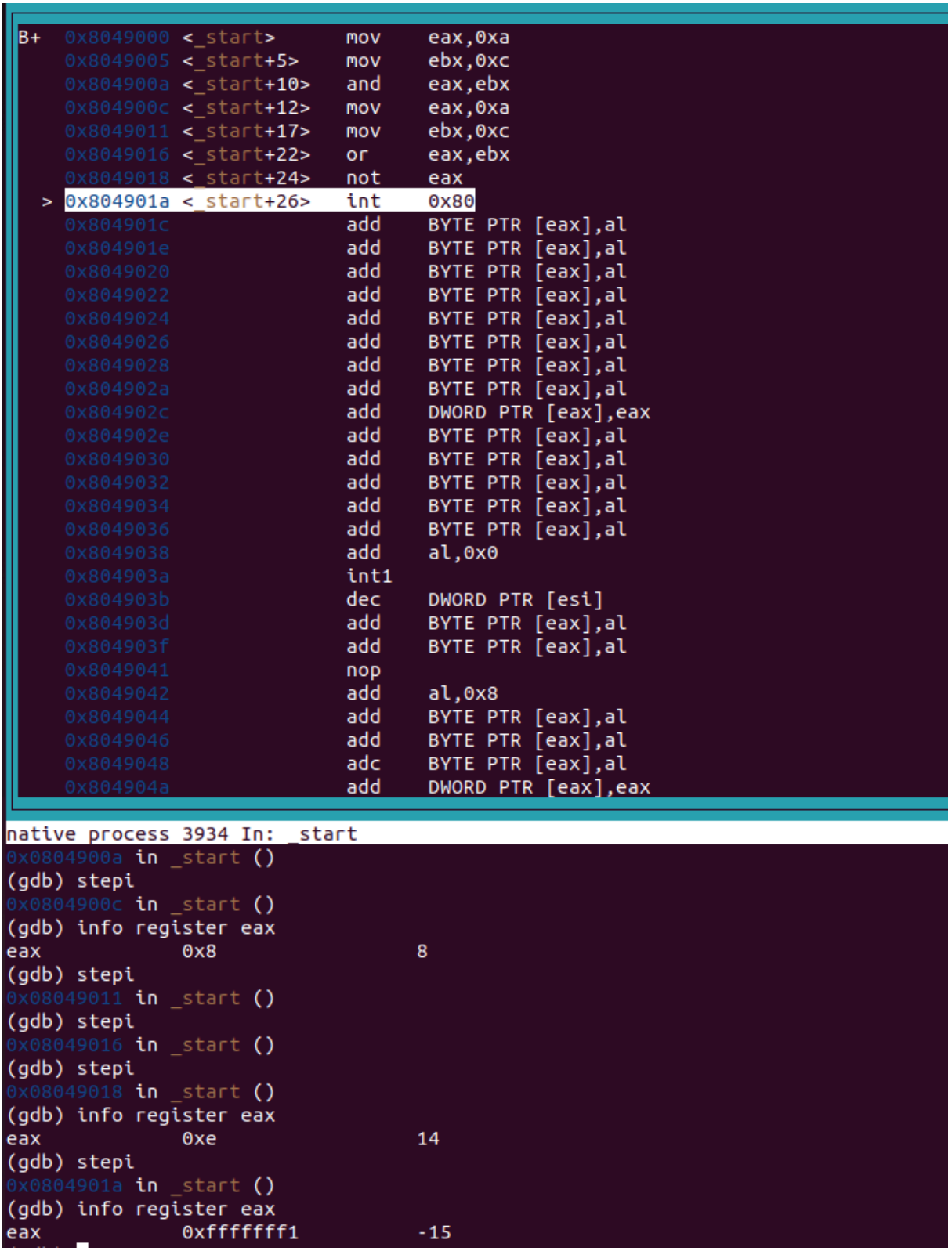
As we see, the result is stored in `eax` and the remainder is stored in `edx`

`IDIV` works just like `IMUL` too, we consider everything as signed and then we deal with it, we can take the example of `MOV eax,0xff` from last time

15. Logical Operators

```
 GNU nano 6.2                                                    logic.s

section .text
global _start

_start:
        MOV eax,0b1010
        MOV ebx,0b1100
        AND eax,ebx

        MOV eax,0b1010
        MOV ebx,0b1100
        OR eax,ebx

        NOT eax

        INT 80h
```

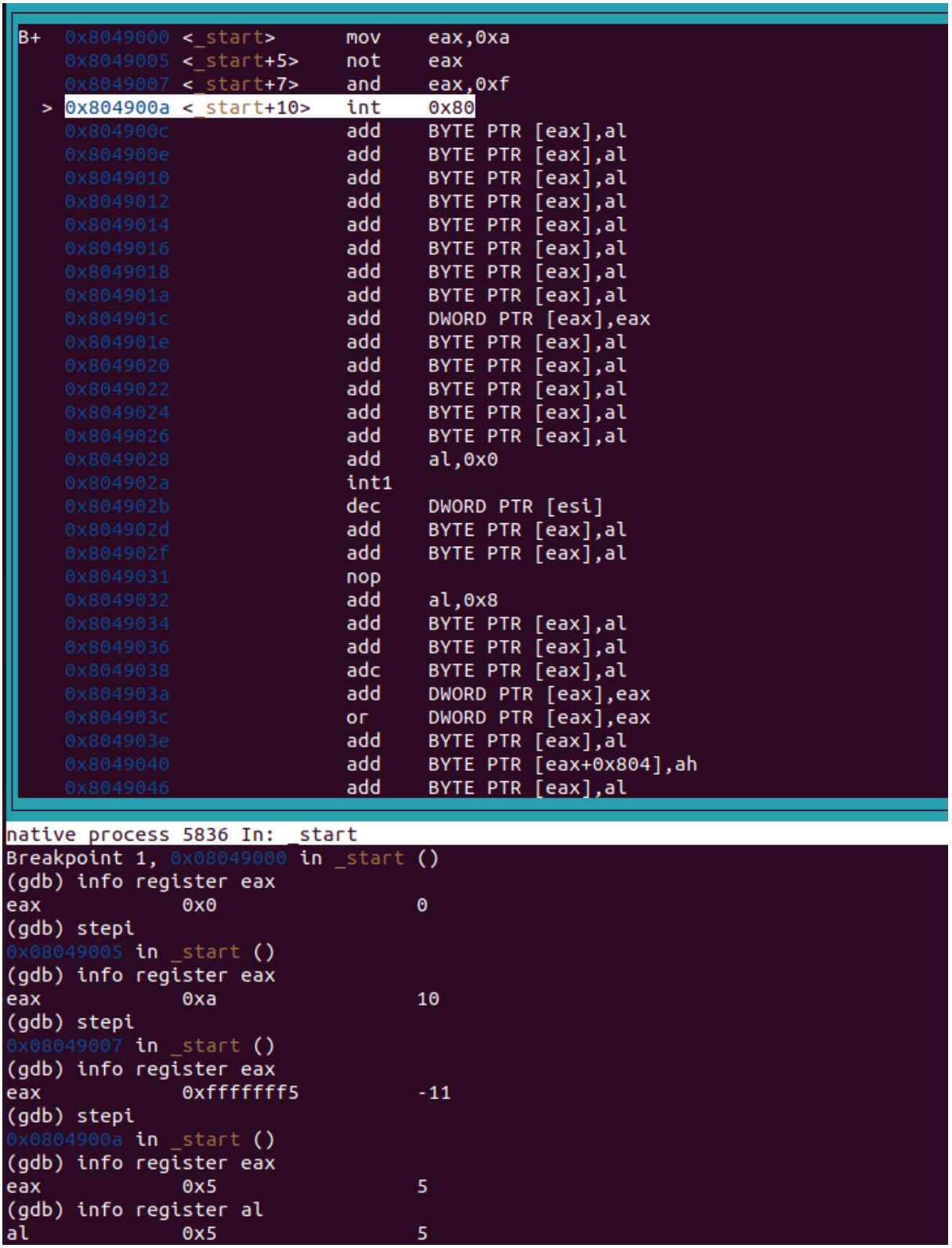
Here we are using `AND`, `OR`, and `NOT` operators, same as they work in real world, but `NOT` is a bit different



We had `0x1110` and we expect `0x0001` instead we got `0xfffffff1` in `al` we get `0xf1` this happens because the `NOT` operator flips every single bit in the register even if that was not the part of the number

```
 GNU nano 6.2                                                    logic.s *

section .text
global _start

_start:
        MOV eax,0b1010
        NOT eax
        AND eax,0x0000000F              ; 0xF
        INT 80h
```

To fix the above issue we use a concept called masking, we use this to make sure our output does not change and the actual input value remains rest all are reset to normal, here we use `AND 0xf` this fill the last 4 bits (1111) to ensure that the last bit value does not change and rest all bits that were not a part of the program are resettled back to normal



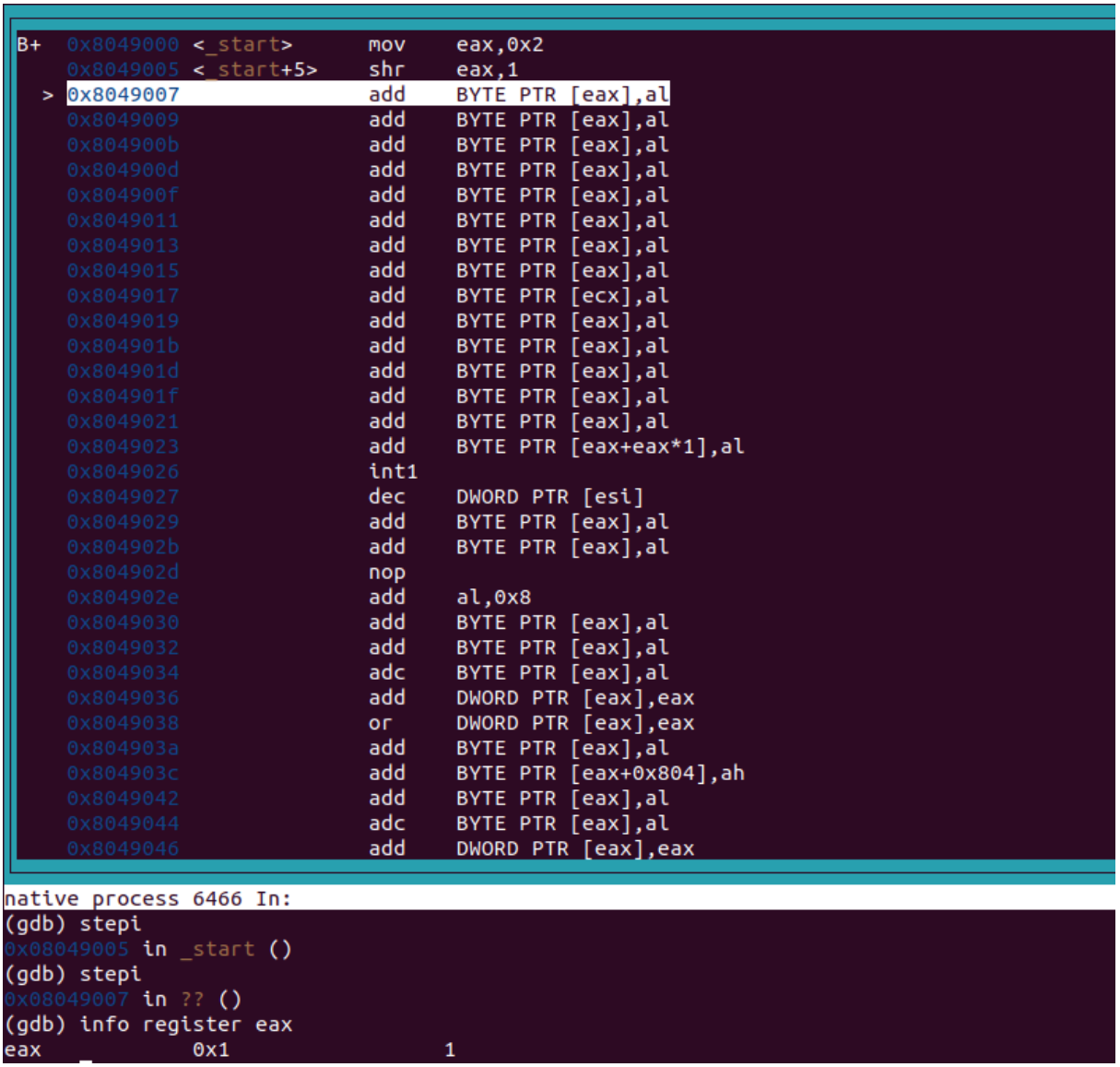
Since we `AND eax,0xf` what this does is here so out actual output here does not change and stays the same and the bits that were changed without any involvement got back to normal

We also have `XOR` that works the same as `AND / OR` operator, it sets value when there's only one, 1 value

16. **Shifts**

```
 GNU nano 6.2                                                shift.s

section .text
global _start

_start:
        MOV eax,2
        SHR eax,1
```

Using `SHR` we are going to shift bits to right side, 2 in binary is `0010` since we shift it to right by **1**, the output we get should be `0001` and the last value is always sent to carry flag

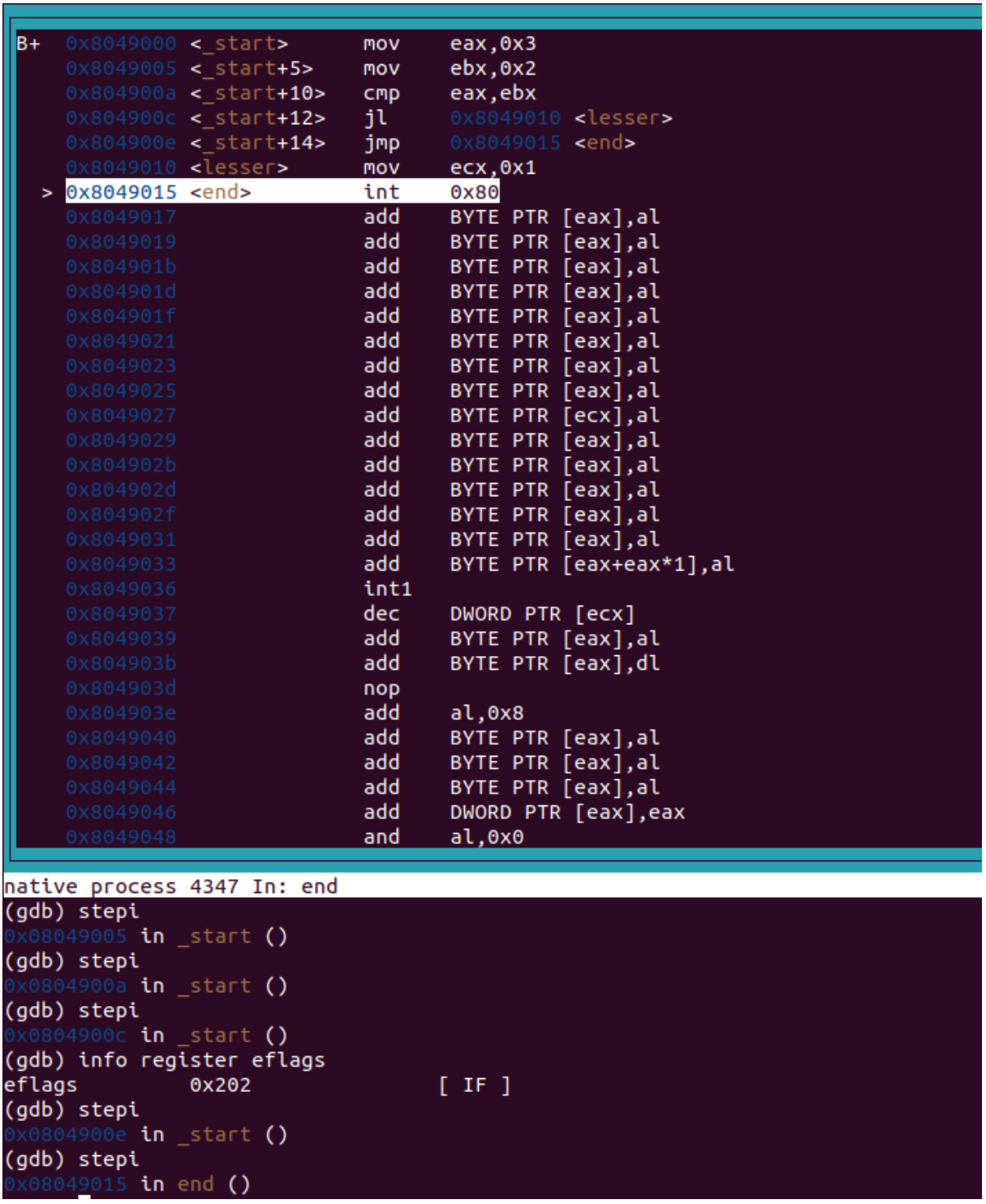


Shifting to right can also be seen as dividing value by two, (Ex: 0100 → 0010), quicker to use for dividing it by two

`SHL` is the same as `SHR` but we get $x/2$, in `SHL` we get $x * 2$, there are more **Shift** operator, `SAL & SAR` shift arithmetic left and shift arithmetic right, mostly to deal with signed

17. Comparison and Jump Instructions

`CMP` subtracts 2 registers and discards the result but uses it to set the `EFLAGS`, tells if an register had a higher, negative and equal values, the `JL` and `JMP` are jump instructions



Here, we can see that we assigned the value of `eax > ebx` therefore the `CMP` function didn't set any flag, also our `JL` did not jump to the `lesser:` section since we set it condition to **if less**. However, the `JMP` flag is always executed as it has no condition to follow whatsoever.

The conditional `jump` instruction allows us to jump at any section based on any condition, we still need to be careful while we use this branching with

conditional as we need an else condition else it might crash the code.

18. Creating a Loop

Finally we will be seeing an actual code that will be achieving a certain task and do it properly.
We will be creating an **array of [1,2,3,4]** and we will be storing the sum of this array in `cl` register and we will use 2 sections, i.e `loop:` and `end:`.

```
  GNU nano 6.2                                                first.s
section .data
        list DB 1,2,3,4

section .text
global _start

_start:
        MOV eax,0
        MOV cl,0

loop:
        MOV bl,[list + eax]
        ADD cl,bl
        INC eax
        CMP eax,4
        JE end
        JMP loop

end:
        MOV eax,1
        MOV ebx,1
        INT 80h
```

1. Declare a variable name list with an array of [1,2,3,4] 1 byte each.
2. Initialize the index position of the array to `eax` (i.e 0) and initialize value of `cl` to 0.
3. `loop:` section is used to move the values of `list` into `bl` register one by one and than add value of `eax` to it, which will be incremented by 1 as we go further.
4. Add the value of `bl` into `cl` to get the sum of all the numbers in array.
5. Using `INC` instruction to increment the value of `eax` by 1 and then we use `CMP` to check if the value of `eax` reaches the final index position of the list (i.e 4).

1. If it does, jump to the `end:` section and end the program, if not continue to execute it using `JMP loop` till we achieve it.

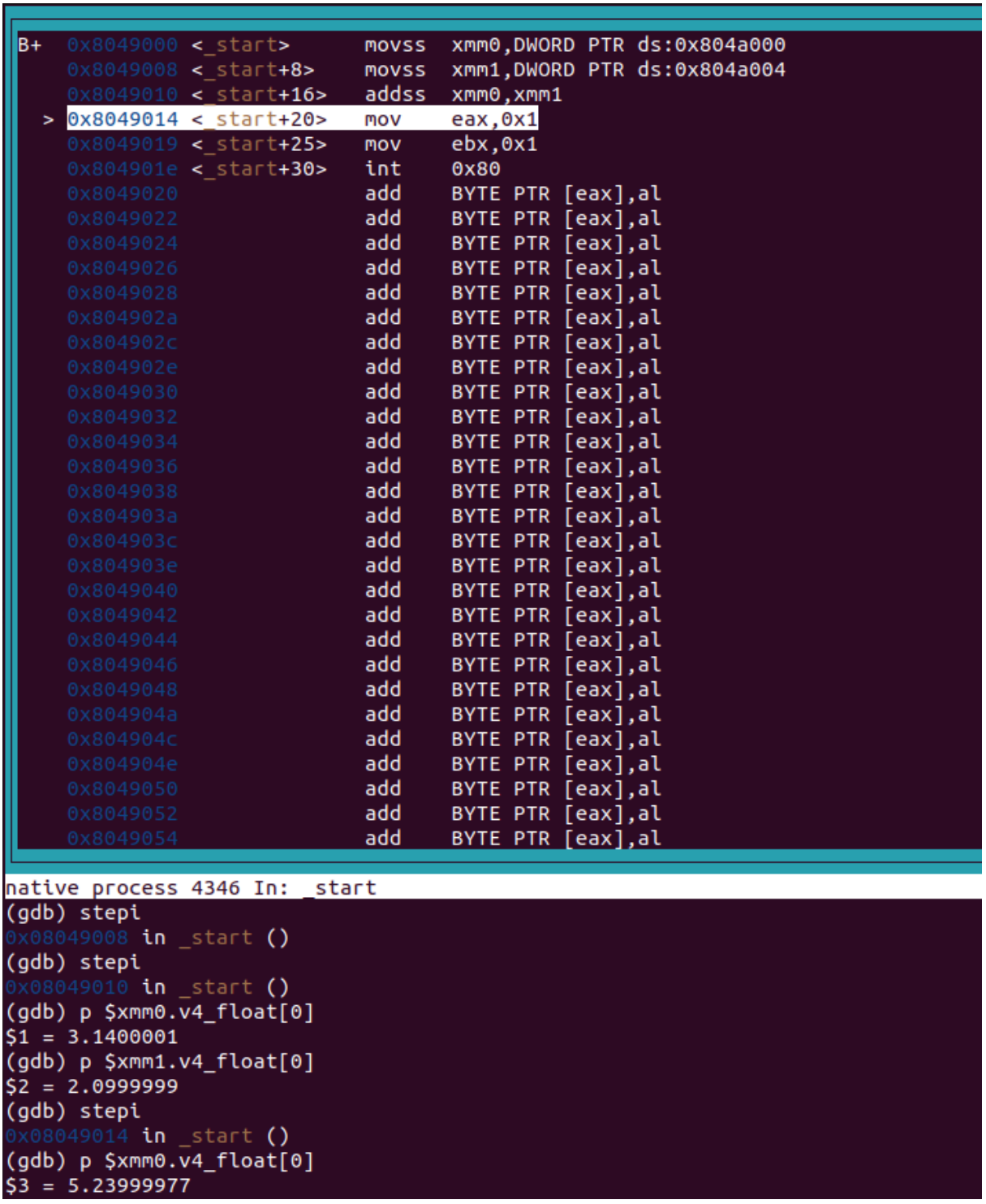
19. Floating Point Numbers

Just like how we initialize and move the values to one variable, floating values work the same (Ex. `x DD 3.14`) we move this using `MOVSS` instead of `MOV` **ss → scalar single precision**, the register we move this into can work with packed as well as scalar data.

We have special registers for this called `xmm` register

```
 GNU nano 6.2                                                first.s
section .data
        x DD 3.14
        y DD 2.1

section .text
global _start

_start:
        MOVSS xmm0,[x]
        MOVSS xmm1,[y]
        ADDSS xmm0,xmm1

        MOV eax,1
        MOV ebx,1
        INT 80h
```

For addition of 2 floating numbers, we use `ADDSS` and this has same syntax, (`ADDSS destination,source`)



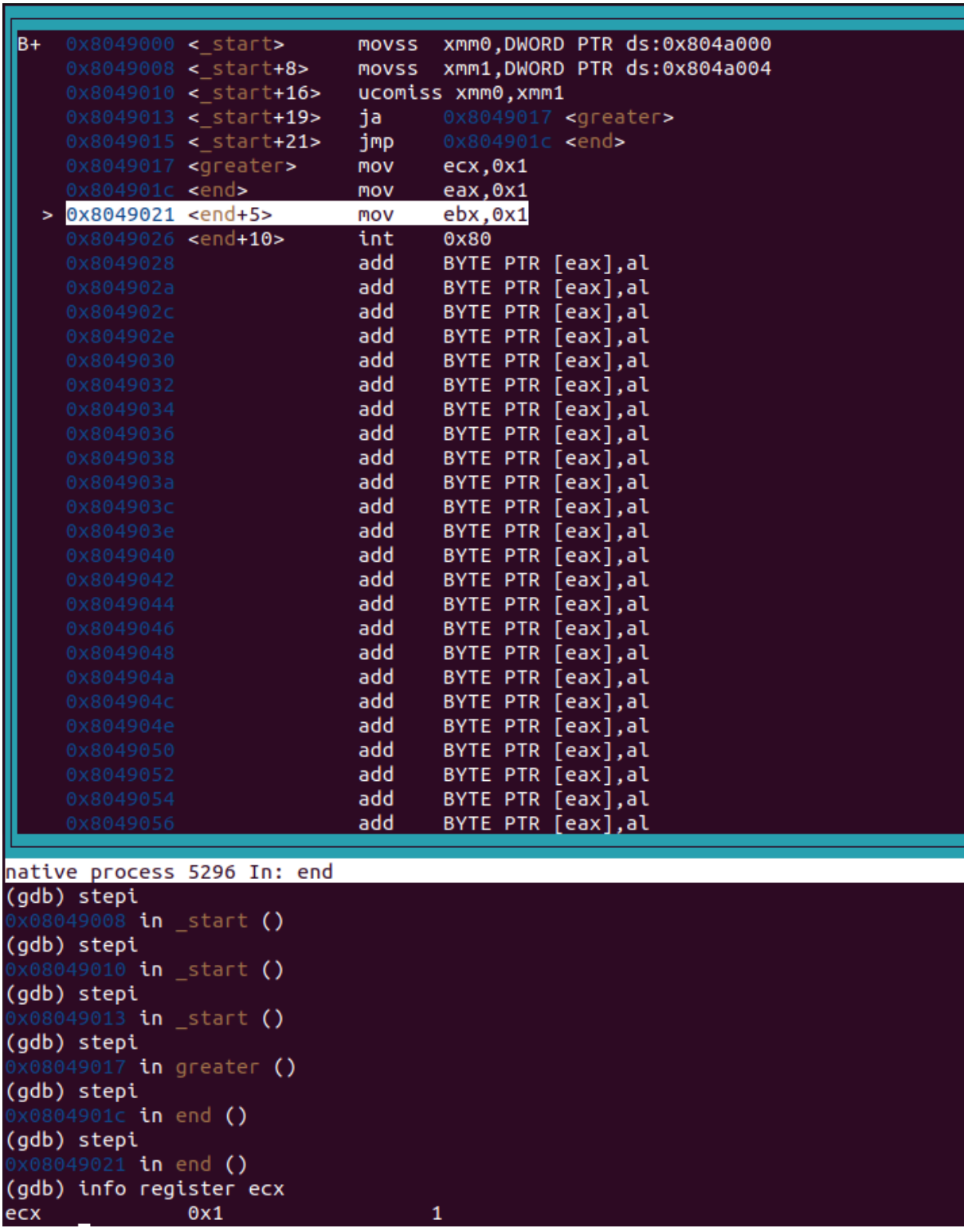
Using `p $xmm0.v4_float[0]` to check the value at `xmm0` register at `[0]` index in `float`

You can also notice that the values that we actually stored we slightly different, as we store **2.1 and 3.14** but the values stored are **3.1400001** and **2.0999999** this also affects the arithmetic operations of the floating numbers

20. **Comparing Floating Point Numbers**

we use `UCOMISS` to compare the floating values, some jumps will be different too
(JB/JBE → Jump below/ or equal, JA/JAE → Jump above/or equal, Just like JG and JL

```
  GNU nano 6.2                                                first.s
section .data
        x DD 3.14
        y DD 2.1

section .text
global _start

_start:
        MOVSS xmm0,[x]
        MOVSS xmm1,[y]
        UCOMISS xmm0,xmm1
        JA greater
        JMP end

greater:
        MOV ecx,1

end:
        MOV eax,1
        MOV ebx,1
        INT 80h
```



Since value of `xmm0` was greater than `xmm1` the code jumps to `greater:` section to and finishes as per the `JMP` and `UCOMISS` instructions

21. Calling Standard C Functions

We will be running our **object (<filename>.o)** file that will be will be made with **nasm** through **gcc.**

> All the external functions should be mentioned above the code, `extern <functionn_name>`

```
 GNU nano 6.2                                                first.s
extern printf
extern exit

section .data
        msg1 DD "Hello World!",0
        msg2 DD "this is second string"
        fmt DB "Output is: %s %s",10,0

section .text
global main

main:
        PUSH msg2
        PUSH msg1
        PUSH fmt
        CALL printf
        PUSH 10
        CALL exit
```

`extern printf & exit` are the external functions that tell the assembler that this is a `C` function, these are external functions that will be linked in later, through GCC.

`msg1 & msg2` are the Defined Double word (4 byte) and `fmt` is Defined byte (1 byte) with `10` is the new line character and `0` as the null terminator, `%s` is just the format specifier.

`global main` & `main:` is the main function for `C` program.

We need to put this data somewhere where it is accessible to `printf` so we will put this data on the stack using `PUSH` instruction.

`printf(format, value)`, therefore since stack is **LIFO,** we push `msg2 & msg1` first then the `fmt` and we call our `printf` function, exit functions takes one argument that is the exit code hence we push `"10"`, this is also called exit code and then we call the `exit` function.

```
shezan@VM:~/Desktop/x86/Calling standard C functions$ nano first.s
shezan@VM:~/Desktop/x86/Calling standard C functions$ nasm -f elf -o first.o first.s
shezan@VM:~/Desktop/x86/Calling standard C functions$ gcc -no-pie -m32 first.o -o first
shezan@VM:~/Desktop/x86/Calling standard C functions$ ./first
Output is: Hello World! this is second string
```

We compile the code by using `nasm -f elf -o <output.o> <input.s>` , then we compile it using GCC, `gcc -no-pie -m32 <input.o> -o <input>` .

We will run the output file, here `first` and the output is displayed, if we call the `echo $?` it will display the `exit code` .

---

22. Calling C Functions

The steps and everything is exact same, we just need to add a `<name>.c` file of our own having a `C` function in it and then we use `extern <func_name>` and while compiling, in `gcc -no-pie -m32 <input.o>` **`<input>.c`** `-o <output>`. Here we add our `C` file that has our function that we are calling in `asm` code.

```
  GNU nano 6.2                                                    test.c
#include <stdio.h>

extern int test(int,int);

int test(int a, int b){
        printf("Here!\n");
        return a+b;
}
```

test.c file

```
  GNU nano 6.2                                                   first.s
extern test
extern exit

section .data

section .text
global main

main:
        PUSH 1
        PUSH 2
        CALL test
        PUSH eax
        CALL exit
```

first.s file

We push `eax` before calling our exit because the `return a+b` is stored in the `eax` register by default so to get out sum as our **exit status or code** we `PUSH eax` before calling the standard `exit` function

```
shezan@VM:~/Desktop/x86/Calling C functions$ nano test.c
shezan@VM:~/Desktop/x86/Calling C functions$ nano first.s
shezan@VM:~/Desktop/x86/Calling C functions$ nasm -f elf -o first.o first.s
shezan@VM:~/Desktop/x86/Calling C functions$ gcc -no-pie -m32 first.o test.c -o first
shezan@VM:~/Desktop/x86/Calling C functions$ ./first
Here!
shezan@VM:~/Desktop/x86/Calling C functions$ echo $?
3
```

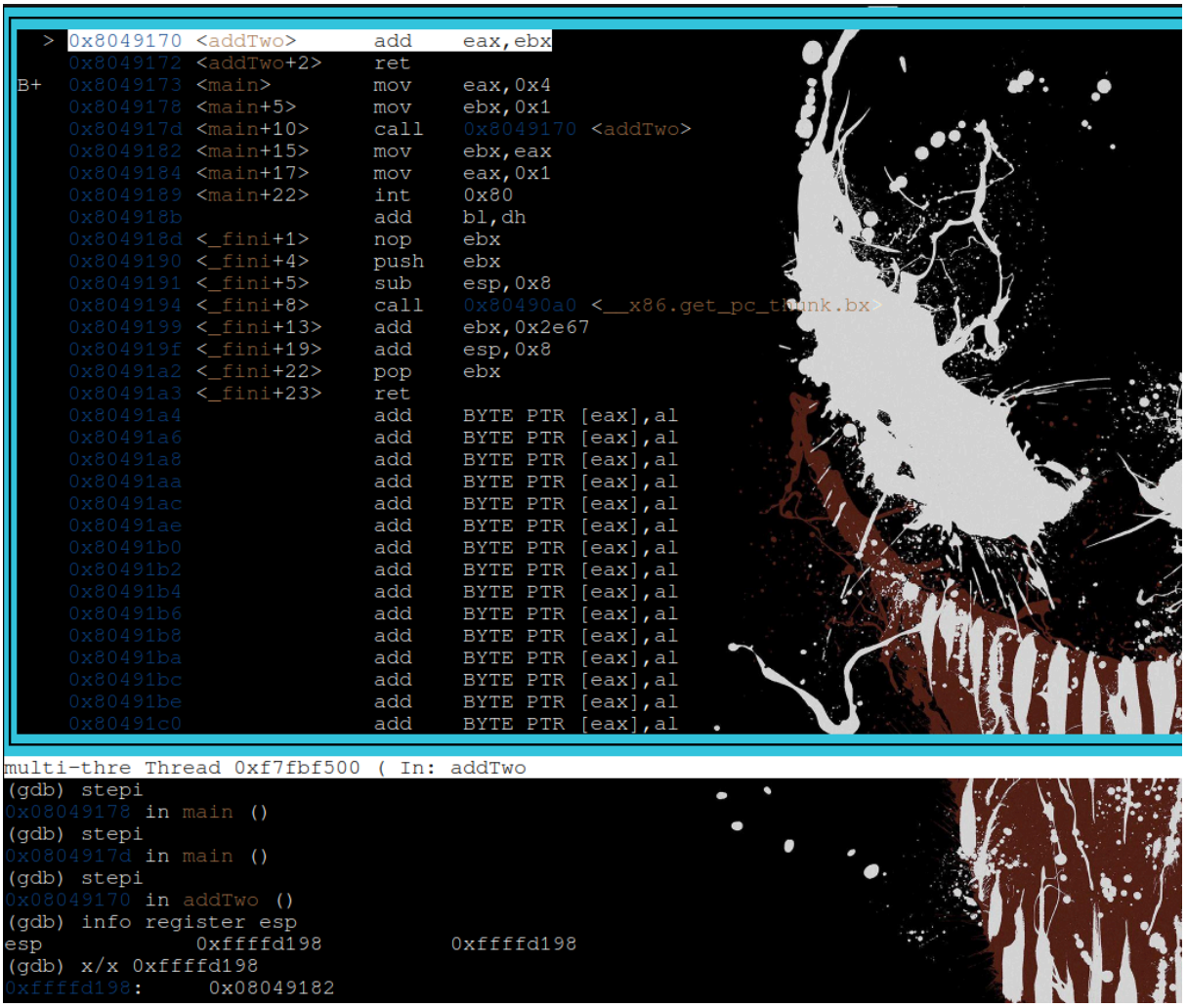
Output

23. **Basics of x86 functions/procedures**

Procedures are like functions in x86, more like representation of a function, just as we call call functions through see, now we will create functions in x86, how they are structured and how they are stored in a stack.

`addTwo` is our function that we will create to add the values in `eax` and `ebx`

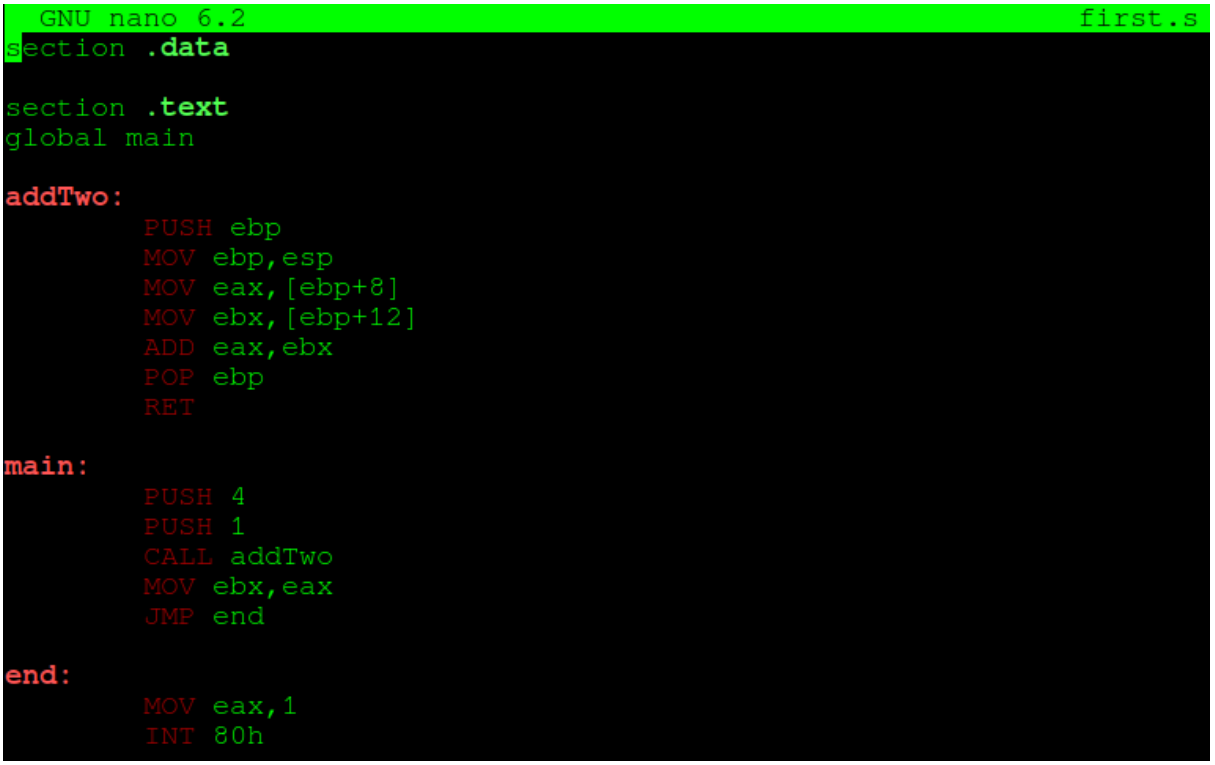
`RET` is return function and how it works is, it stores the value of the next instruction after hitting the `CALL addTwo` function, we will see that in gdb



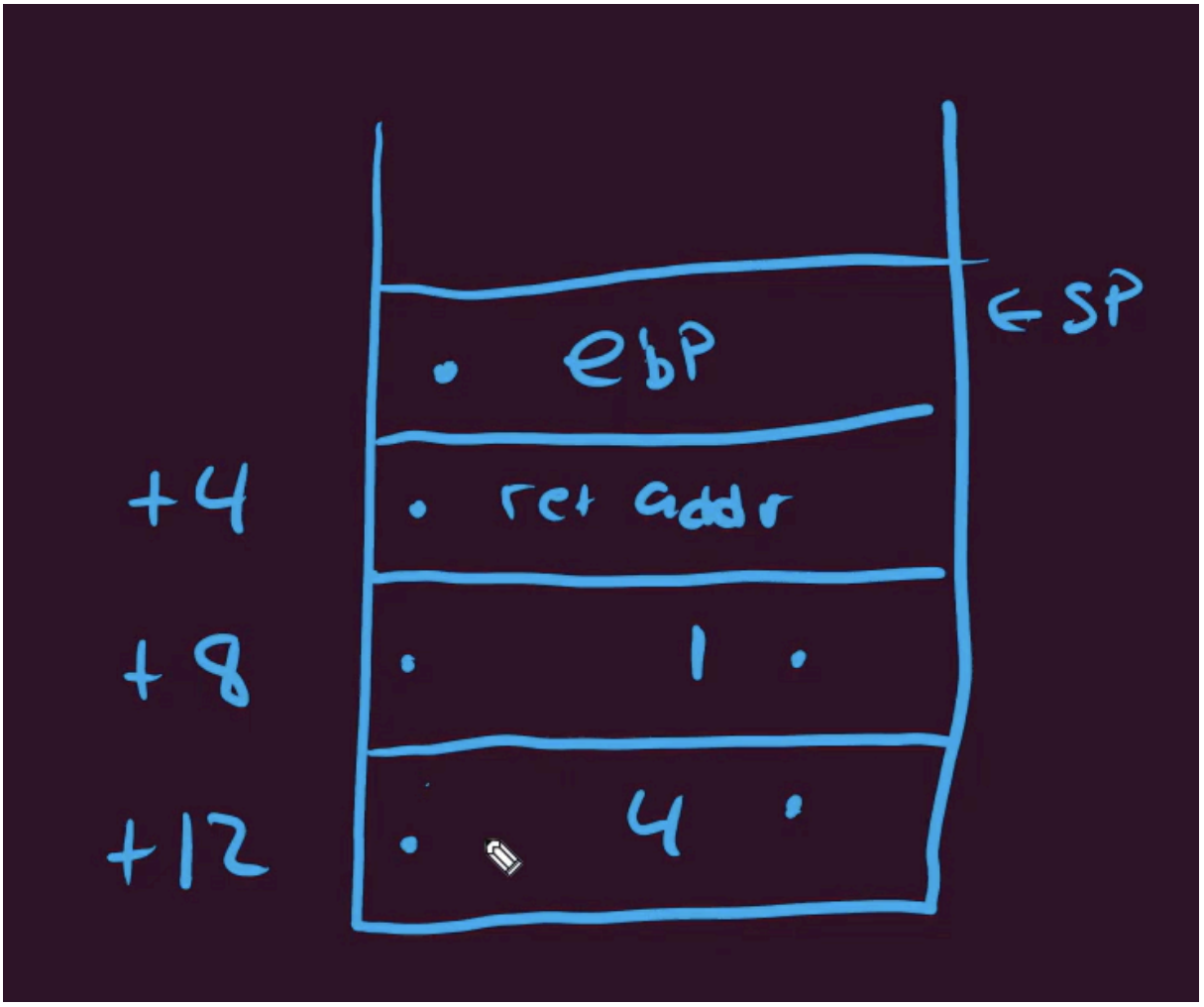
Here we see that after call, `0x08049182` is the address for next instruction and it stores it `esp` and when it hit return, it will go back the the last instruction it was at and then move ahead completing the code

---

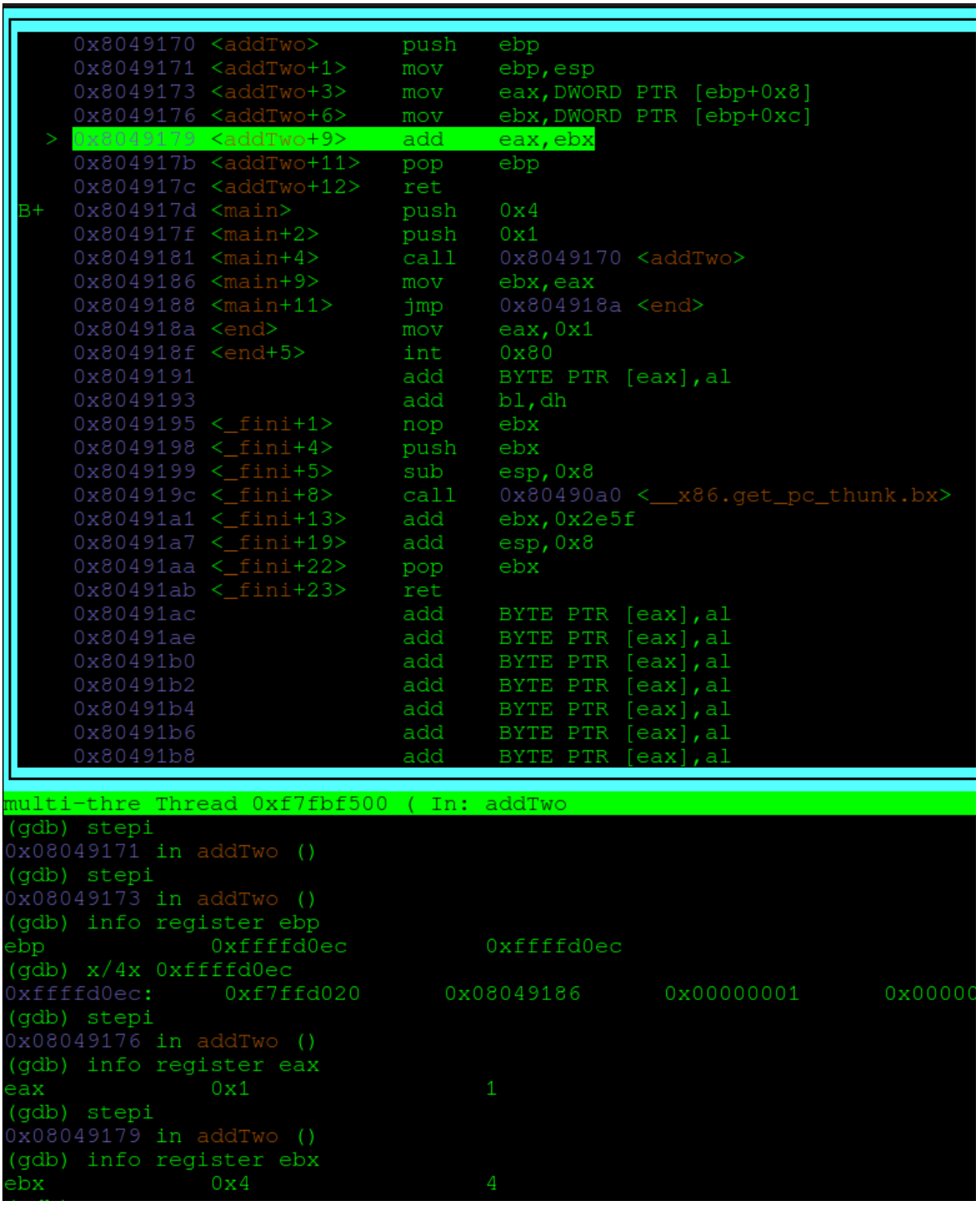
We push our data through the stack because registers are our precious resources, we can't always overwrite register to pass them on functions, and sometimes the data is too big for the register



`ebp` is a base pointer, acts as a divider, anything above it is something else, everything below it is part of function, we move `esp to ebp` to make sure they both return same address. Why we used `MOV eax,[ebp+8]` and `MOV ebx,[ebp+12]` is explained below, We `POP ebp` because the stack pointer (`esp`) points to `ebp` this will cause our code to fail and get stuck



This is a rough diagram of stack of our above code, each memory segment is of 4, therefore to get to our instruction we used `MOV eax,[ebp+8]` and `MOV ebx,[ebp+12]` to get to our instructions



as we can see here after pushing `ebp` and `MOV ebp,esp` when we check the `info register ebp` it shows us the address in `esp` and when we examine it using `x/4x 0xffffd0ec` we see the 4 memory segments , as follows:

1. Address of the **Stack pointer (esp)**
2. Address of the next instruction to continue after `RET`

3. Value of `eax`
4. Value of `ebx`

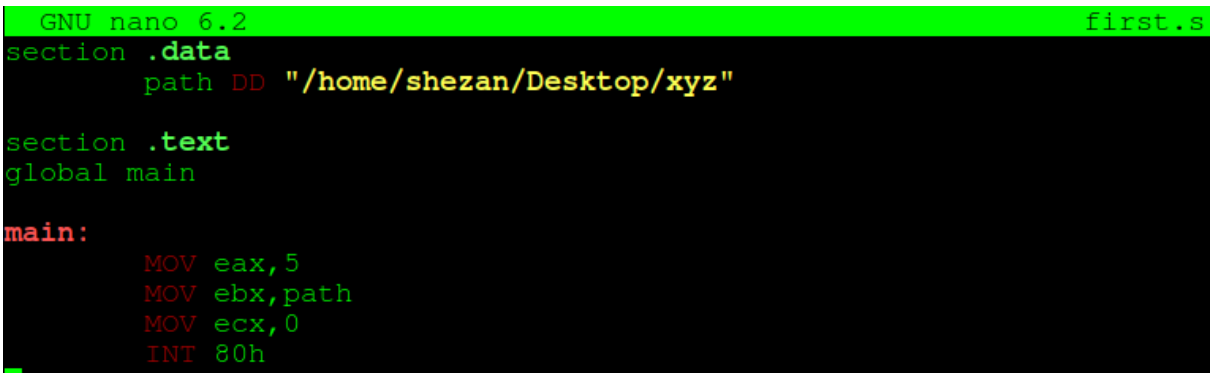
25. Opening and Reading Files

To open and read files we need system calls, it is used to interact with Operating system itself,

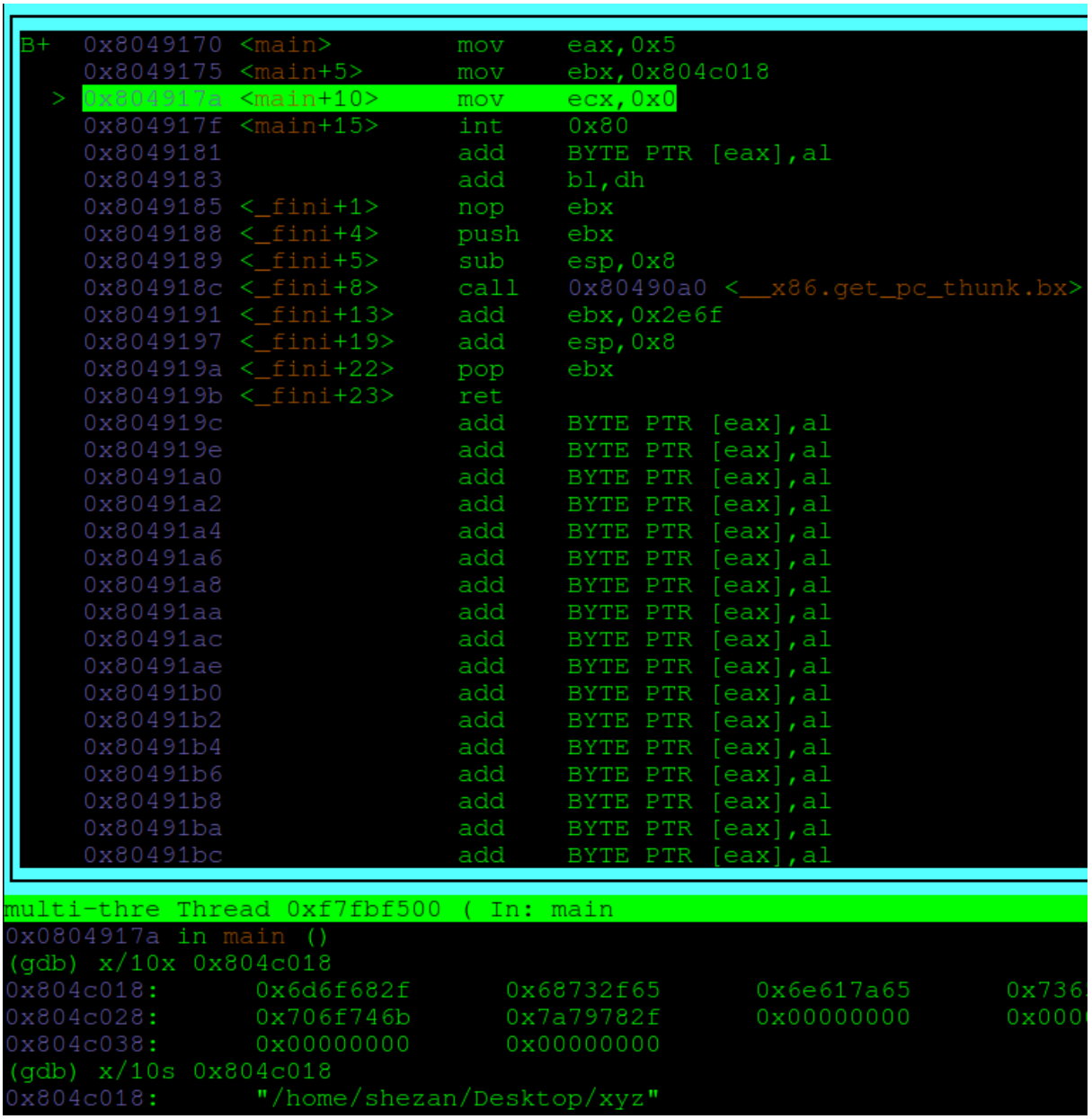
How we can understand system calls for opening and reading files? Click on the below button to get a tabular format of the System calls in Linux

Click on the following button to go to the manual page of open system call

To get the file mode (read only / write only, etc) we need the binary code as our asm needs INT code as it mentioned, click the button to check the int value.



We `MOV eax,5` as from the system call table we can see the `eax = 5` is the code for `sys_open` call, we move path to `ebx` as it's mentioned in linux sys call table, we also need the mode of access therefore we move it's value of role into `ecx` , here '0'

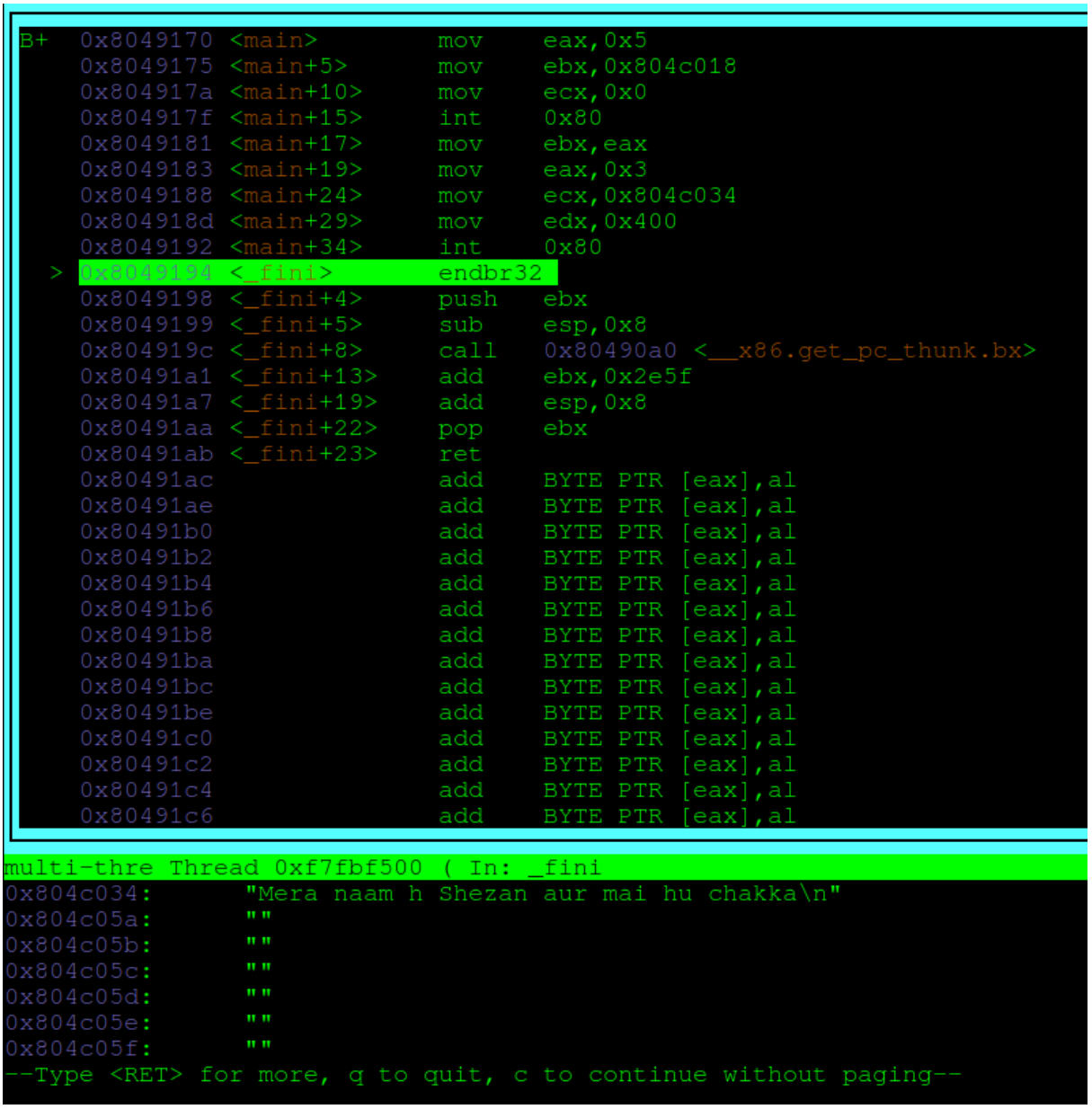


When we examine the value of `ebx`, the `0x804c018` in strings, displays the actual value or the path that we placed inside the `ebx`.

After the system call the result is stored in the `eax` it can also be called as file descriptor. we can use it to read data with it.

```
  GNU nano 6.2                                                first.s
section .data
        path DD "/home/shezan/Desktop/xyz"

section .bss
        buffer: resb 1024

section .text
global main

main:
        MOV eax,5
        MOV ebx,path
        MOV ecx,0
        INT 80h

        MOV ebx,eax
        MOV eax,3
        MOV ecx,buffer
        MOV edx,1024
        INT 80h
```

we move `3` in `eax` to read the data from the file all those are available in above buttons, we reserve `1024` bytes so that our bytes are not short for the data, we move `buffer` in `ecx` and size of the bytes in `edx` because



data is stored in the `buffer:` where we reserved our `1024` bytes to read data, and we can see it using `x/10s 0x804c034` (buffer address), and the data in file is printed

26. Using Lseek with Files

We can use Lseek to skip through some characters to read directly specific lines

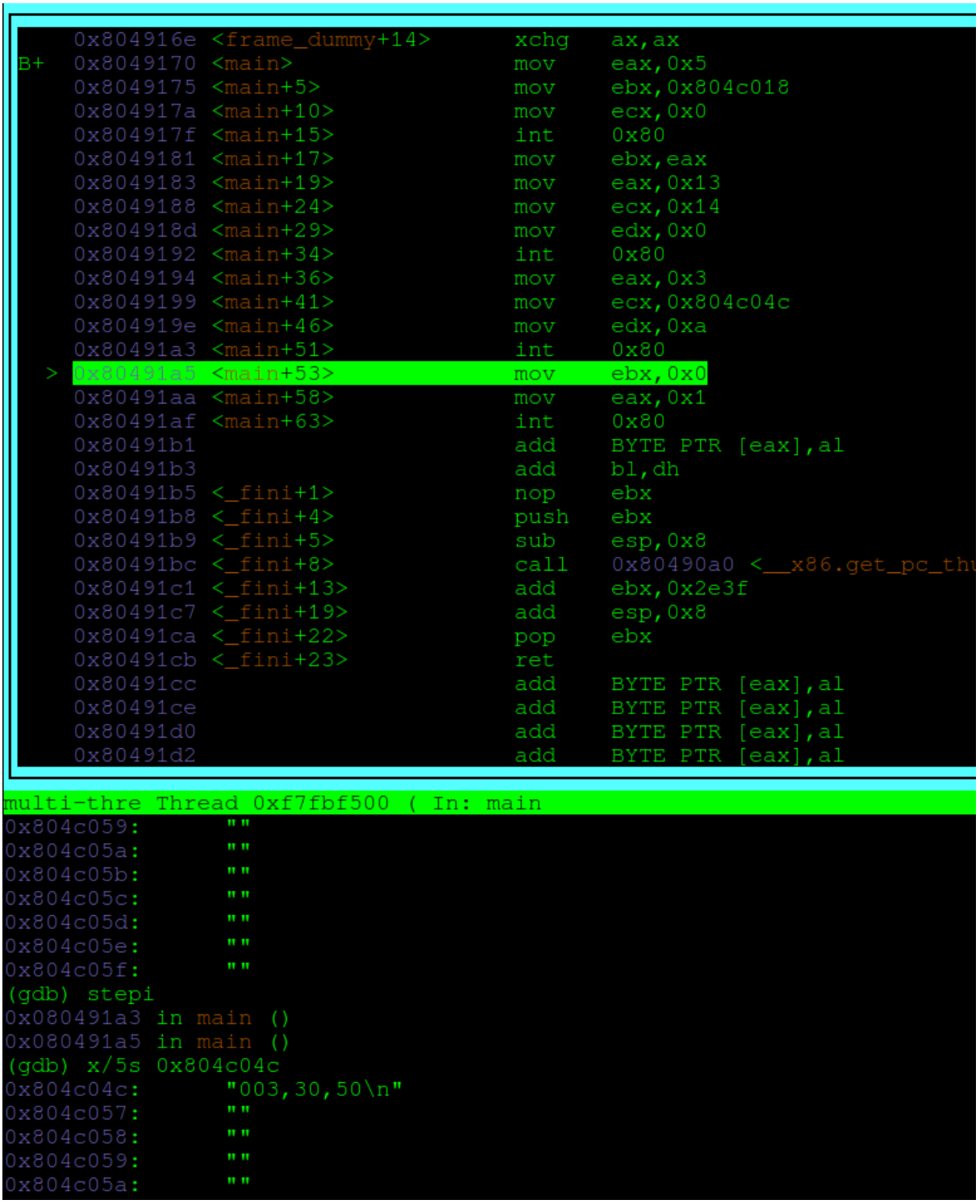
```
  GNU nano 6.2                                                first.s
section .data
        pathname DD "/home/shezan/x86/Using Lseek with files/test.txt"

section .bss
        buffer resb 10

section .text
global main

main:
        MOV eax,5
        MOV ebx,pathname
        MOV ecx,0
        INT 80h

        MOV ebx,eax
        MOV eax,19
        MOV ecx,20
        MOV edx,0
        INT 80h

        MOV eax,3
        MOV ecx,buffer
        MOV edx,10
        INT 80h

        MOV ebx,0
        MOV eax,1
        INT 80h
```

First.s (asm code)

`MOV eax,5` to open the file, moving `pathname` to `ebx` for the file, `MOV ecx,0` for read only mode.

`MOV ebx,eax` to move file descriptor that is stored in `eax` to `ebx`, `MOV eax,19` for the **sys_call**, `MOV ecx,20` to skip 20 characters of the file `MOV edx,0` to start skipping from the start of the file.

`MOV eax,3` to read the file, `MOV ecx,buffer` to reserve 10 byte of data, `MOV edx,10` to move 10 byte of data into the `ecx`



Here we can see that we skipped starting 20 characters (actual 9 chars and 1 “\n” char), and only the 10 bytes are stored in the buffer, this is how Lseek works and helps us to get to the data we want without a fuss.

27. **Creating and Writing Files**

We can combine 2 parameters using `bitwise OR` operator we can do it in 2 ways, each flag individually or all in head or paper and place it in register. `101o` denotes that this is octal value

```
  GNU nano 6.2                                                first.s
section .data
        pathname DD "/home/shezan/x86/Creating and writing files/test.txt",0
        toWrite DD "Hello World!",0AH,0DH,"$"

section .text
global main

main:
        MOV eax,5
        MOV ebx,pathname
        MOV ecx,101o
        MOV edx,700o
        INT 80h

        MOV ebx,eax
        MOV eax,4
        MOV ecx,toWrite
        MOV edx,15
        INT 80h

        MOV eax,1
        MOV ebx,0
        INT 80h
```

First.s (asm code)

We got `101o` and `700o` by using `OR` operation on various different permissions all together,

`0AH, 0DH` , `MOV edx,15` is the count of characters we are moving

`101o` → create file and writing

`700o` → Read, write and execute permissions

```
shezan@VM:~/x86/Creating and writing files$ nano first.s
shezan@VM:~/x86/Creating and writing files$ ./first
shezan@VM:~/x86/Creating and writing files$ ls
first  first.o  first.s  test.txt
shezan@VM:~/x86/Creating and writing files$ cat test.txt
Hello World!
```

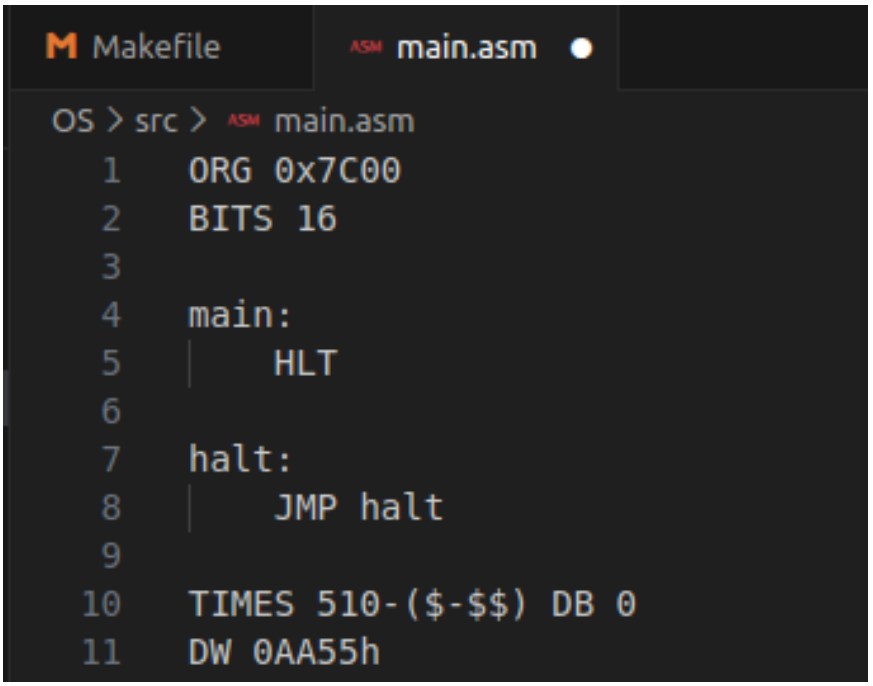
Output

28. Building a Simple Bootloader

A bootloader is a program responsible for booting a computer, when a computer 1st boots, it enters BIOS (Basic Input Output System). it contains various utility and tools that allows OS to perform Hardware initialization,

In legacy booting format, the BIOS searches for OS in the disk and start the booting process so it takes each of bootable device and loads it into memory at a location which is `0x7C00` and this memory location will load the first segment of disk into it, and it looks for a special signature that is `0xAA55`, if it sees this signature, it will start executing from the start of the data segment and start running the code it has there.

in order to get our code running as a bootloader, we need to write `0xAA55` into `0x7C00` location.



main.asm

`ORG 0x7C00` is used to tell the assembler to do our addressing to this address.

`BITS 16` to set **16 bits** when a computer boots, it starts in 16-bit mode, this is for backward compatibility reason.

`HLT` (Halt) is used to pause the CPU, till it reaches a particular interrupt.

`halt:` for infinite loop

`TIMES 510-($-$$) DB 0`, pads the file with “0” until it reaches 510 bytes

`DW 0AA55h`, These are the *final two bytes* (byte 510 and 511) of the 512-byte boot sector. BIOS checks for this signature to decide:

- ✔️ Disk is bootable
- ❌ Disk is not bootable → BIOS ignores it

x86 systems store multi-byte values in **little-endian** order.

Every **block of data** (also called a *sector*) on a bootable disk is **512 bytes**. This matches the behavior of a classic **1.44MB floppy disk**, where **each sector = 512 bytes**.

Makefile main.asm

OS > Makefile

```
ASM=nasm

SRC_DIR = src
BUILD_DIR = build

$(BUILD_DIR)/main.img: $(BUILD_DIR)/main.bin
    cp $(BUILD_DIR)/main.bin $(BUILD_DIR)/main.img
    truncate -s 1440k $(BUILD_DIR)/main.img

$(BUILD_DIR)/main.bin: $(SRC_DIR)/main.asm
    $(ASM) $(SRC_DIR)/main.asm -f bin -o $(BUILD_DIR)/main.bin
```

Makefile

```
shezan@VM:~/Desktop/x86/Building a simple bootloader/OS$ make
make: 'build/main.img' is up to date.
shezan@VM:~/Desktop/x86/Building a simple bootloader/OS$ qemu-system-i386 -fda build/main.img
WARNING: Image format was not specified for 'build/main.img' and probing guessed raw.
         Automatically detecting the format is dangerous for raw images, write operations on b
 be restricted.
         Specify the 'raw' format explicitly to remove the restrictions.
```

QEMU - Press Ctrl+Alt+G to release grab

Machine View

```
SeaBIOS (version 1.15.0-1)

iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+07F8B590+07ECB590 CA00


Booting from Hard Disk...
Boot failed: could not read the boot disk

Booting from Floppy...
_
```

output

29. Printing A Message in BIOS

main.asm

```asm
ORG 0x7C00
BITS 16

main:
    MOV ax,0
    MOV ds,ax
    MOV es,ax
    MOV ss,ax

    MOV sp,0x7C00

    MOV si,os_boot_msg
    CALL print

    HLT

halt:
    JMP halt

print:
    PUSH si
    PUSH ax
    PUSH bx

print_loop:
    LODSB
    OR al,al
    JZ done_print

    MOV ah, 0x0E
    MOV bh,0
    INT 0x10

    JMP print_loop

done_print:
    POP bx
    POP ax
    POP si
    RET

os_boot_msg: DB 'Mat kar lala Mat kar',0x0D,0x0A,0


TIMES 510-($-$$) DB 0
DW 0AA55h
```

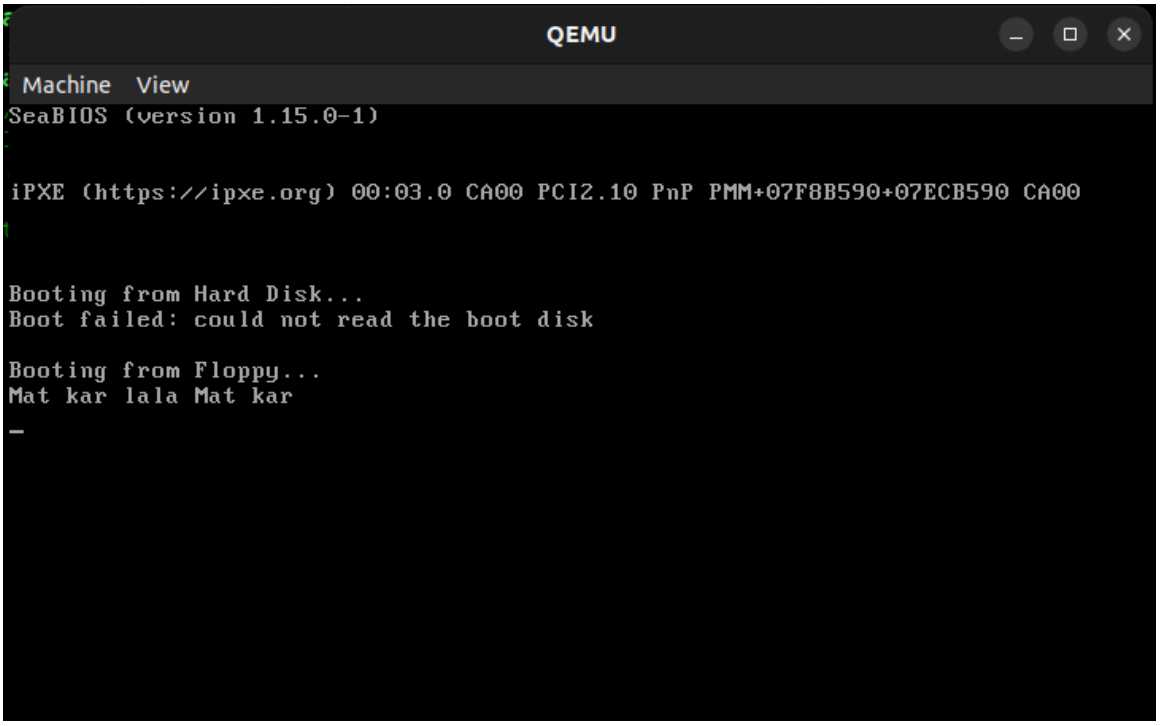
main.asm

Initialing our environment for working, origin from `0x7C00` memory address and `BITS 16` to set it on 16-bit mode
We move **“0”** in all the segments, Data segment, Extra segment and Stack segment so that we have more accurate result, We move `0x7C00` into stack pointer to set it at start of bootloader.

we push registers in our `print` section, `LODSB` to load the msgs 1 by 1 into al, we use `OR al, al` to check if our al has set any zero flag, this is done to ensure that the string is completely added, including the null terminator as null terminator will set the 0 flag in `OR` operation, we compare it using `JZ`.
Once the `JZ` is hit, it'll pop all the registers in stack and return to the Loop, and start executing further, `MOV ah,0x0E` is for:

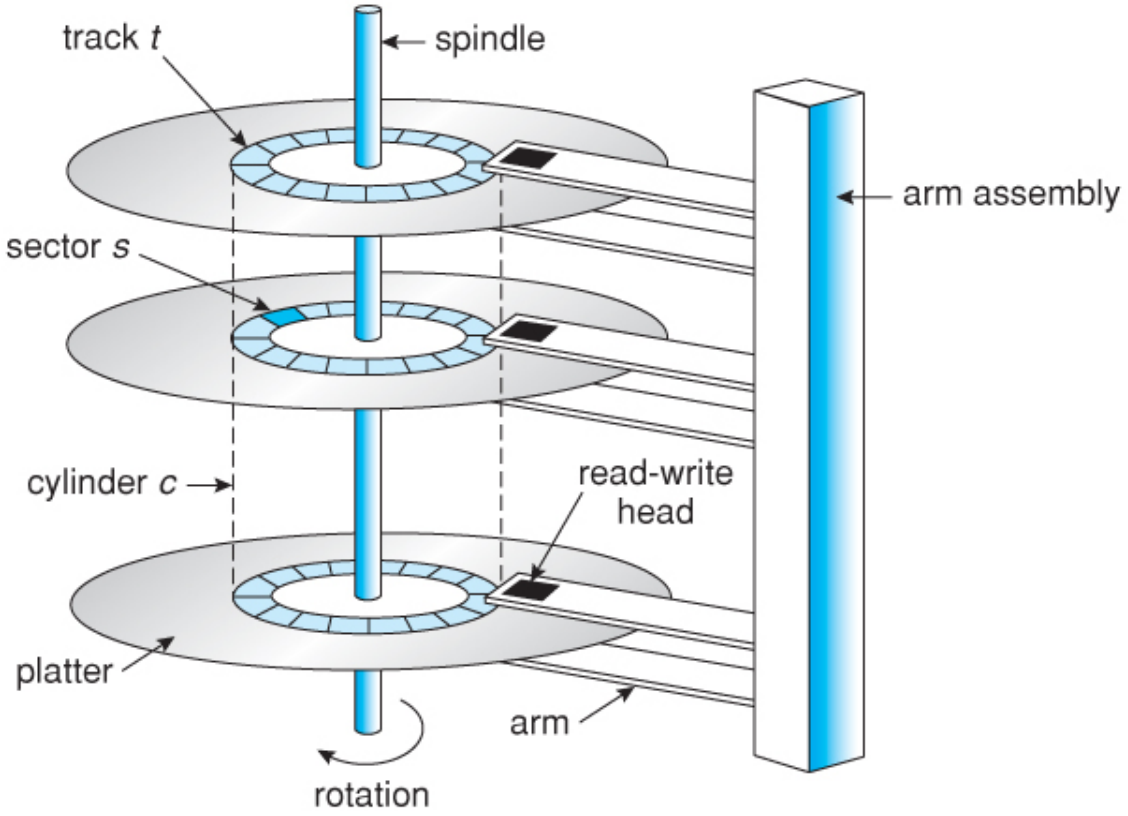
- `AH, 0x0E`: This sets the BIOS interrupt 0x10 function to "Teletype output," which prints a character to the screen.
- `BH, 0`: Specifies the page number for the display (usually 0 for the main screen).
- `INT 0x10`: Calls BIOS video services. With `AH = 0x0E`, it prints a character stored in `AL` to the screen.

all this is done again and again till our loop condition is satisfied



Output window

30. **Disk Storage Structure**



Hard Disk Structure

Each disk is called a platter, it has 2 heads top and bottom, both sides have data and can be read and write from. We divide everything into little segments called tracks and we divide tracks into sectors.

Sector size is fixed on the disk (commonly 512 bytes or 4096 bytes), it does not depend on the file format.

Tracks directly above each other is called a cylinder, we locate data by looking at the cylinder and then select the head (surface) instead of the platter and then the actual segment that tells where our data is.

Cylinder-head-sector (CHS)

- Defines a tuple of the cylinder, head, and sector of data
- LBA is preferred, as it translated to over devices better
- Used for partitioning and older devices

Cylinder-head-sector (CHS)

Logical Block Addressing (LBA)

- A linear addressing scheme where blocks are located by integer index
- First LBA block is 0, second is 1, and so on

Logical Block Addressing (LBA)

Converting CHS to LBA

- $LBA = (C * TH * TS) + (H * TS) + (S - 1)$
  - C -> Sector cylinder number
  - TH -> Total headers on disk
  - TS -> Total sections on disk
  - H -> Sector head number
  - S -> Sector's number

Converting CHS to LBA

Converting LBA to CHS

- t = LBA/sectors per track
- s = (LBA % sectors per track) + 1
- h = (t % number of heads)
- c = (t/number of headers)

Converting LBA to CHS

---

31. Creating a FAT12 Disk

We have 2 sections of actucal OS bootloader and kernal,

bootloader sets it into expected state and then loads kernal into memory and kernal does everything, manages resources, handles system call