1 First assembly program

```
section .data

section .text
        global _start

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

View the program here.

Breaking down the program:

- **.data** → This section is used for declaring and initializing variables.
- **.section** → This section contains the actual code of the program.
- **_start** → This is the entry point of the program.
- **MOV eax, 1** → Moves the immediate value 1 to the register **eax**.
- **MOV ebx, 1** → Moves the immediate value 1 to the register **ebx**.
- **INT 80h** → An interrupt with the value $80h$(h stands for *hexadecimal*).

• In x86 architecture, interrupt **0x80**(or **80h** is commonly used for making system calls in Linux.
• The value 1 in **eax** corresponds to the **sys_exit** system call in Linux.
• The value 1 in **ebx** is interpreted as the exit status code.
• **INT 80h** triggers an interrupt which transfers the control to the Linux kernel to execute the system call.

Simple explanation

This program is going to trigger an interrupt which will transfer the control to the kernel which will perform an action based on the value that is put into **eax**(in this case it's 1). So, basically **eax** tells the operating system what we want to do when the interrupt 80 is caught. So, in this case, we want the operating system to run the exit system call when the interrupt is caught. The exit system call will end the program and will set an exit status code. The status code is going to be whatever value we put into **ebx**(in this case, the status code will be 1).
So, in summary, register **eax** is telling us what system call we want to do. Register **ebx** tells what we want to output as our status code. **INT 80h** triggers an interrupt which tells the operating system to exit the program using **eax** set to 1.

There are some system calls mentioned along with their numbers here.

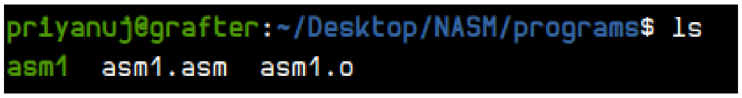
2 Compiling the assembly program

```
$ nasm -f elf -o asm1.o asm1.asm
```

Basically we are compiling our assembly program to an object file using the file format *elf*.
Next, we use the GNU linker to link the object file and create an executable:

```
$ ld -m elf_i386 -o asm1 asm1.o
```

• **-m elf_i386** → Indicates the target architecture. Here it is set to **elf_i386** which means the x86 architecture(32-bit).
After linking the object file, we have our executable file which is **asm1**.



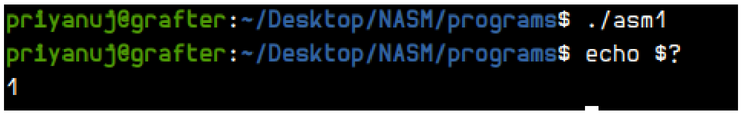
```
priyanuj@grafter:~/Desktop/NASM/programs$ ls
asm1   asm1.asm   asm1.o
```

Figure 1: The executable **asm1**

We can run this executable:

```
$ ./asm1
```

Since we set an exit status code in **ebx**, after running this executable, we can view that exit status code. In Linux, the **$?** stores the exit status code of any command.



```
priyanuj@grafter:~/Desktop/NASM/programs$ ./asm1
priyanuj@grafter:~/Desktop/NASM/programs$ echo $?
1
```

Figure 2: The exit status code in **$?**

3 Using GDB with our executable

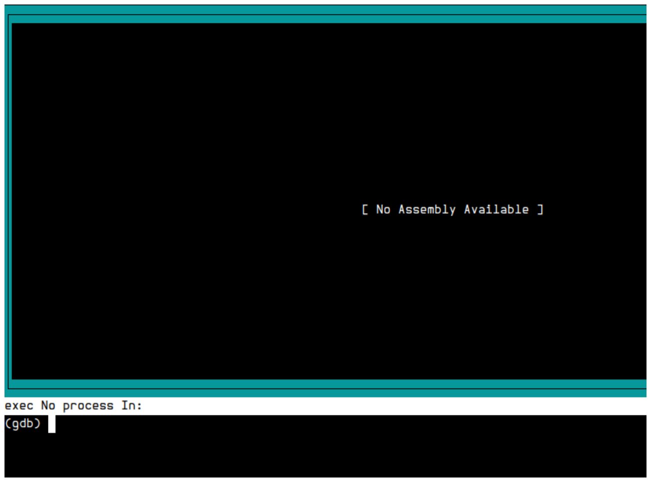
First, we will open GDB with our executable:

```
$ gdb ./asm1
```

3.1 Opening the assembly layout in GDB

```
(gdb) layout asm
```

This opens the assembly layout where we can view the assembly code of our currently executing function.



```
[ No Assembly Available ]
exec No process In:
(gdb)
```

Figure 3: **layout asm**

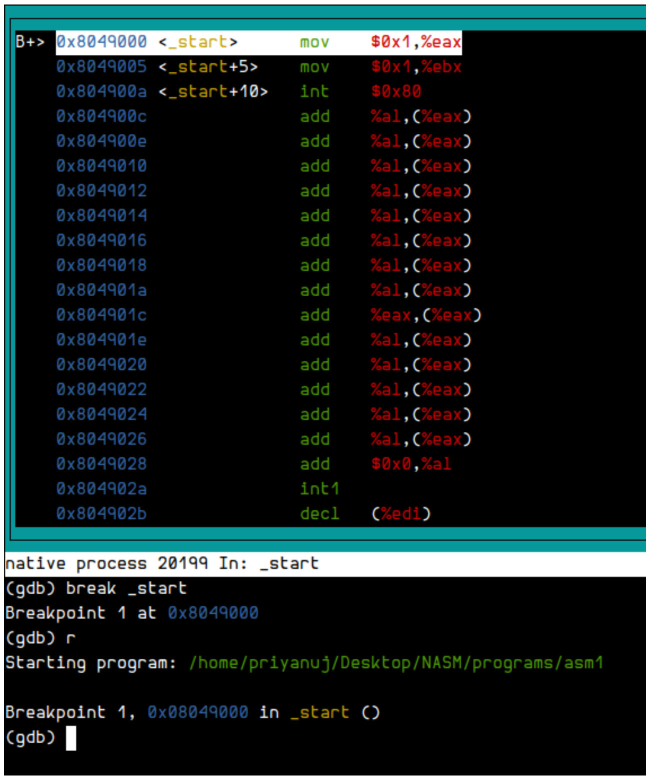
3.2 Setting breakpoints

Now, we will set a break point at **_start** label.

```
(gdb) break _start
```

Next, we will run the program:

```
(gdb) r
```



```
B+> 0x8049000 <_start>     mov    $0x1,%eax
    0x8049005 <_start+5>   mov    $0x1,%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    %eax,(%eax)
    0x804901e              add    %al,(%eax)
    0x8049020              add    %al,(%eax)
    0x8049022              add    %al,(%eax)
    0x8049024              add    %al,(%eax)
    0x8049026              add    %al,(%eax)
    0x8049028              add    $0x0,%al
    0x804902a              int1
    0x804902b              decl   (%edi)
native process 20199 In: _start
(gdb) break _start
Breakpoint 1 at 0x8049000
(gdb) r
Starting program: /home/priyanuj/Desktop/NASM/programs/asm1

Breakpoint 1, 0x08049000 in _start ()
(gdb)
```

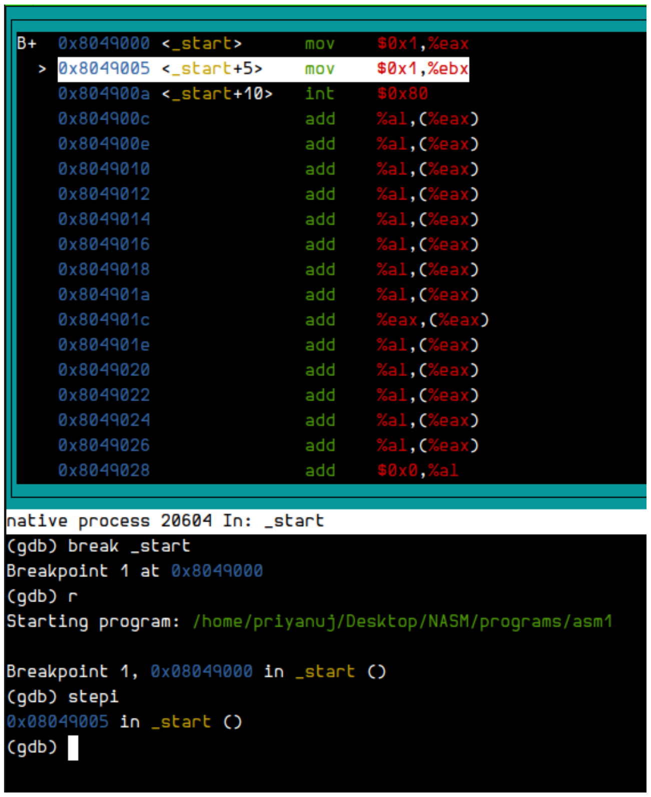
Figure 4: Break points set

Execution stopped at the **_start** label.

3.3 Executing one instruction at a time

Now, we will use the **stepi** command of GDB to execute each instruction, one at a time.

```
(gdb) stepi
```



```
B+  0x8049000 <_start>       mov    $0x1,%eax
  > 0x8049005 <_start+5>     mov    $0x1,%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    %eax,(%eax)
    0x804901e                add    %al,(%eax)
    0x8049020                add    %al,(%eax)
    0x8049022                add    %al,(%eax)
    0x8049024                add    %al,(%eax)
    0x8049026                add    %al,(%eax)
    0x8049028                add    $0x0,%al

native process 20604 In: _start
(gdb) break _start
Breakpoint 1 at 0x8049000
(gdb) r
Starting program: /home/priyanuj/Desktop/NASM/programs/asm1

Breakpoint 1, 0x08049000 in _start ()
(gdb) stepi
0x08049005 in _start ()
(gdb)
```

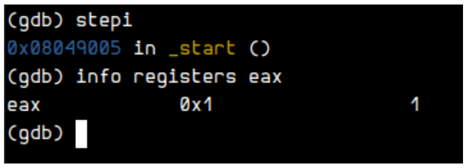
Figure 5: Executing one instruction at a time

3.4 Viewing the value stored in a register

The instruction **MOV eax, 1** got executed and execution stopped at the instruction after it. To prove that 1 indeed got stored into **eax**, we will use the following command:

```
(gdb) info registers eax
```

This gives us the following output:

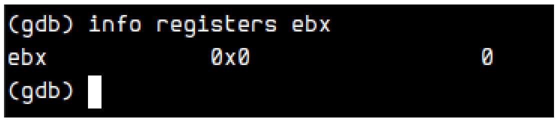


```
(gdb) stepi
0x08049005 in _start ()
(gdb) info registers eax
eax            0x1                 1
(gdb) 
```

Figure 6: Checking if **eax** actually stores 1 or not

So, **eax** stores the value 1.
We can also see what is stored in the **ebx** register.
For that, we use the same command:

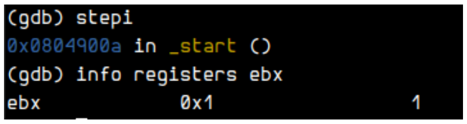
```
(gdb) info registers ebx
```



```
(gdb) info registers ebx
ebx            0x0                 0
(gdb) 
```

Figure 7: Checking the value of the **ebx** register

Now, we again use the **stepi** command to execute the instruction **MOV ebx, 1** and then run the **info registers** command again:



```
(gdb) stepi
0x0804900a in _start ()
(gdb) info registers ebx
ebx            0x1                 1
```

Figure 8: Updated value of **ebx**

4 *stepi* command

This command is used to execute a single machine instruction and then stop, allowing the user to step through a program at the assembly instruction level.

This command is a part of GDB's low-level debugging features.

It's useful for understanding the execution flow of a program at the machine code level.