1 ADD, ADC and EFLAGS

1.1 The ADD instruction

Consider the following program:

```
section .data

section .text
        global _start
        _start:
                MOV eax, 1
                MOV ebx, 2
                ADD eax, ebx
                INT 80h
```

The instruction **ADD eax, ebx** means the same as: **eax** = **eax** + **ebx**.

1.1.1 Debugging

Below are the values of the registers **eax** and **ebx** in before and after performing addition.

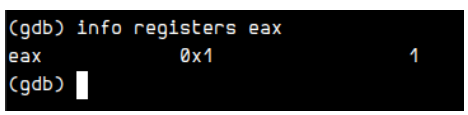

Figure 1: Value of **eax** after addition

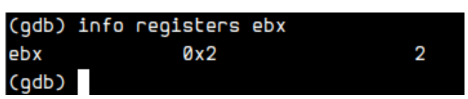

Figure 2: Value of **ebx** before addition

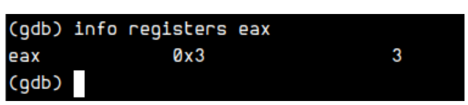

Figure 3: Value of **eax** after addition

1.2 EFLAGS

There is one more register called the **eflags** register. It is a 32-bit register used to indicate the **status/information** about the computations and control CPU operations. Each bit represents a specific flag.

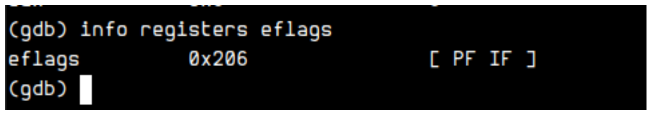

Figure 4: The **eflags** register.

Here **eflags** is giving information about the **ADD** operation and we see that two of its flags are set **PF** and **IF**.

PF is the **P**arity **F**lag. This flag is *set* if the value of an operation is **odd**. If the value of an operation is an even number then PF is set to 0. So, the **ADD** operation resulted into an odd value which is 3 and that's why PF was set.

IF is the **I**nterrupt **F**lag. This flag is *set* to 1 when we allow interrupts to be done on our system.

The figure below shows the the **eflags** register:

Figure 5: The **eflags** register.[Image source: GeeksForGeeks]

<u>NOTE:</u> In the assembly program above, include the line: **MOV eax, 1** before the interrupt statement otherwise the program won't exit and it will throw **segmentation fault**.

1.2.1 More flags of the EFLAGS register

Consider the following program:

```
section .data

section .text
        MOV al, 0b11111111
        MOV bl, 0b0001
        ADD al, bl
        MOV eax, 1
        INT 80h
```

The notation **0b** indicates a binary number. Therefore **0b11111111** indicates a binary number. Basically, this program adds two numbers. One is -1(in **al**) and the other is 1(in **bl**). We know that the result is 0 but we want to see how this is operation performed and if any carry is generated then where that carry is stored.

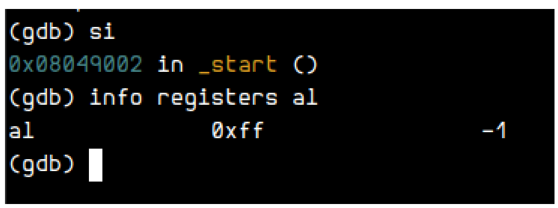

Figure 6: **al** stores -1

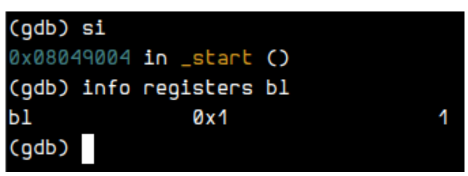

Figure 7: **bl** stores 1

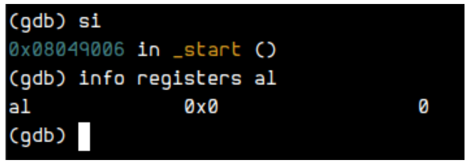

Figure 8: **al** stores 0

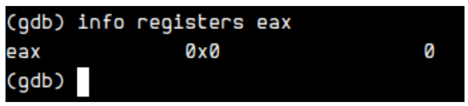

Figure 9: **eax** also stores 0

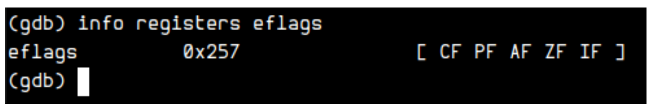

Figure 10: **eflags** has some set bits

The **ADD** operation looks like this:

11111111

00000001

00000000

This operation was shown for **al** and **bl**. So when we check the register **al**(Figure 8), we see that it stores 0. But interestingly enough, when we check the **eax** register(Figure 9), we see that it also stores 0.

The reason is that, the **ADD** operation was performed for only 1 byte registers(**al** and **bl** in this case). So, when the extra 1(carry) resulted from the addition, it was put into the **CF** flag of the **eflags** register and that is the reason why we see 0 as the value of **eax** as well.

CF is the **C**arry **F**lag. When there is a carry from the previous operation, then it's set to 1.

ZF is the **Z**ero **F**lag. It is set if the register that results from the operation is 0. In our case, **al** got set to 0.

1.3 The ADC instruction

Consider this assembly program:

```
section .data

section .text
        global _start
        _start:
                MOV al, 0b11111111
                MOV bl, 0b0001
                ADD al, bl  ; al = al + bl
                ADC ah, 0
                MOV eax, 1
                INT 80h

```

ADC instruction is called add with carry. It works similar to the **ADD** operation. It takes a destination and adds a source to it. It then adds the CF to the destination.

In this program we are using the **ah**(higher 8 bits) to store the carry. The instruction **ADC ah, 0** will basically add 0 to **ah** and then will also add the CF(the carry bit in CF) to it and store the result in **ah**.

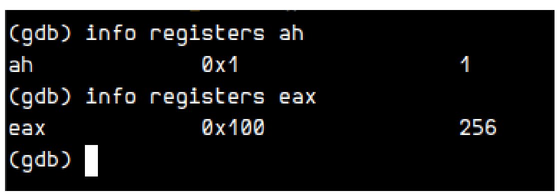

Figure 11: Value of **ah** and **eax**

We can see that **ah** stores 1(which is the CF) and **eax** stores 256.

Now the reason **eax** stores 256 is that it is a 32-bit register and **ah** is its higher 8-bits. So we know that the lower 8 bits(**al**) were all zeros(from addition) and the higher 8-bits were all 0 as well(from the previous addition). But now, we used the **ADC** instruction to put the carry into the higher 8 bits of **eax** which make it look like this:

00000000 00000000 00000001 00000000

which is 256 in decimal.