1 Logical operators

1.1 The AND, OR and NOT operators

Assembly program:

```
section .data

section .text
        global _start
        _start:
                MOV eax, 0b1010
                MOV ebx, 0b1100

                AND eax, ebx   ; AND operation

                MOV eax, 0b1010
                MOV ebx, 0b1100

                OR eax, ebx    ; OR operation

                NOT eax        ; NOT operation

                MOV eax, 1
                INT 80h
```

The value in **eax** is 10 and in **ebx** is 12. The first operation is the **AND** operation. So, when we perform **AND** between 10 and 12 i.e

$$\begin{array}{r} 00000000\ 00000000\ 00000000\ 00001010 \\ AND\ 00000000\ 00000000\ 00000000\ 00001100 \\ \hline 00000000\ 00000000\ 00000000\ 00001000 \end{array}$$

which results in 8. This result is stored in **eax** as shown in the image below:

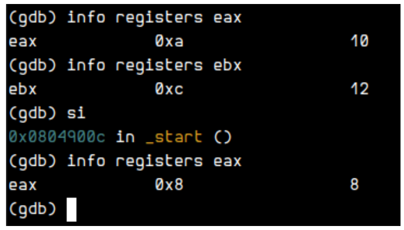

Figure 1: **AND** operation

The next operation is **OR** and we re-assign the values 10 and 12 to **eax** and **ebx** respectively.

So, we perform the **OR** operation between **eax** and **ebx** i.e

$$\begin{array}{r} 00000000\ 00000000\ 00000000\ 00001010 \\ OR\ 00000000\ 00000000\ 00000000\ 00001100 \\ \hline 00000000\ 00000000\ 00000000\ 00001110 \end{array}$$

which results in 14 and again stored in **eax** as shown below:

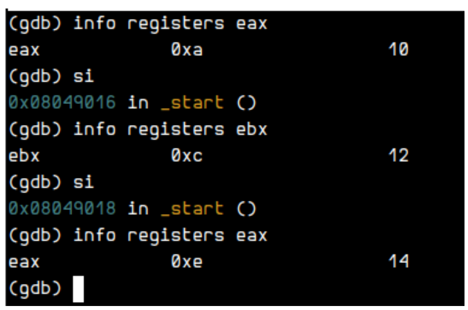

Figure 2: **OR** operation

The final operation is the **NOT** operation. Here we basically invert all the set bits and the unset bits. The value in **eax** is 14(from the previous **OR** operation).

$$\begin{array}{r} NOT\ 00000000\ 00000000\ 00000000\ 00001110 \\ \hline 11111111\ 11111111\ 11111111\ 11110001 \end{array}$$

This value is -15 which gets stored in **eax**.

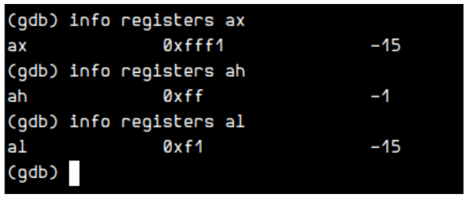

Figure 3: **NOT** operation

2 Masking

Let's say, we have a value 10 in **eax** and after performing a logical operation, we want only the last 4-bits to be affected by the operation. Then we would use a mask. We need to **filter** those bits that we need.

Program:

```
section .data

section .text
        global _start

        _start:
                MOV eax, 0b1010
                NOT eax

                AND eax, 0x0000000f

                MOV eax, 1
                INT 80h
```

If we perform a **NOT** operation on **eax** which stores 10 then this is what we will get:

$$\begin{array}{r} NOT\ 00000000\ 00000000\ 00000000\ 00001010 \\ \hline 11111111\ 11111111\ 11111111\ 11110101 \end{array}$$

is -11 which is stored in **eax** as shown below:

We only wanted the higher 24-bits to change. We will use a mask that will only change the higher 24-bits and will keep the lower 4-bits unchanged.

0x0000000f is a 32-bit mask whose last 4-bits are set. First we performed the **NOT** operation on **eax** and then we performed the **AND** operation between **eax** and the mask.

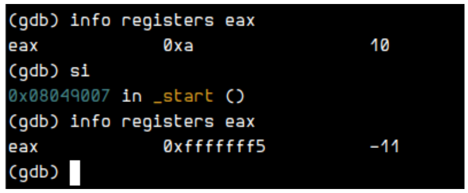

Figure 4: -11 stored in **eax**

So when we perform **AND** operation:

$$\begin{array}{r}
11111111\ 11111111\ 11111111\ 11110101 \\
AND\ 00000000\ 00000000\ 00000000\ 00001111 \\
\hline
00000000\ 00000000\ 00000000\ 00000101
\end{array}$$

We get 5 as the result.

Image:

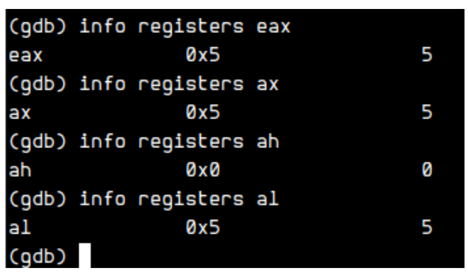

Figure 5: Keeping the last 4 bits unchanged

3 The XOR operation

Program:

```
section .data

section .text
        global _start
        _start:
                MOV eax, 0b1010
                MOV ebx, 0b1100

                XOR eax, ebx

                MOV eax, 1
                INT 80h
```

We are performing **XOR** operation:

$$\begin{array}{r} 00000000\ 00000000\ 00000000\ 00001010 \\ XOR\ 00000000\ 00000000\ 00000000\ 00001100 \\ \hline 00000000\ 00000000\ 00000000\ 00000110 \end{array}$$

We get 6 which is stored in **eax** as shown below:

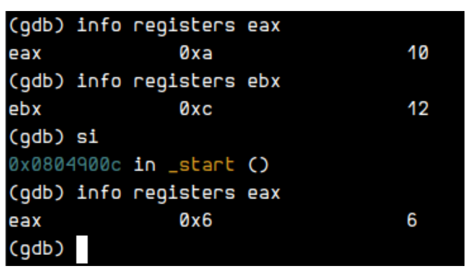

Figure 6: **XOR**