

1 Logical operators

1.1 The AND, OR and NOT operators

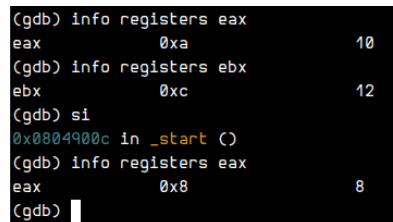
Assembly program:

```
1 section .data
2
3 section .text
4     global _start
5     _start:
6         MOV eax, 0b1010
7         MOV ebx, 0b1100
8
9         AND eax, ebx    ; AND operation
10
11        MOV eax, 0b1010
12        MOV ebx, 0b1100
13
14        OR  eax, ebx     ; OR operation
15
16        NOT eax         ; NOT operation
17
18        MOV eax, 1
19        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

| | | | | |
|------------|----------|----------|----------|----------|
| | 00000000 | 00000000 | 00000000 | 00001010 |
| <i>AND</i> | 00000000 | 00000000 | 00000000 | 00001100 |
| | <hr/> | | | |
| | 00000000 | 00000000 | 00000000 | 00001000 |

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



```
(gdb) info registers eax
eax             0xa             10
(gdb) info registers ebx
ebx             0xc             12
(gdb) si
0x0804900c in _start ()
(gdb) info registers eax
eax             0x8             8
(gdb)
```

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:

```

(gdb) info registers eax
eax          0xa          10
(gdb) si
0x08049016 in _start ()
(gdb) info registers ebx
ebx          0xc          12
(gdb) si
0x08049018 in _start ()
(gdb) info registers eax
eax          0xe          14
(gdb) 

```

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**.

```

(gdb) info registers ax
ax          0xffff1        -15
(gdb) info registers ah
ah          0xff          -1
(gdb) info registers al
al          0xf1          -15
(gdb) 

```

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.

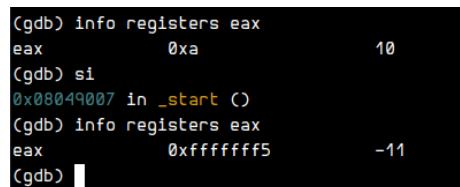
Program:

```
1 section .data
2
3 section .text
4     global _start
5
6     _start:
7         MOV eax, 0b1010
8         NOT eax
9
10        AND eax, 0x0000000f
11
12        MOV eax, 1
13        INT 80h
```

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

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

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



A screenshot of a GDB terminal window. The first command is '(gdb) info registers eax', showing 'eax' with value '0xa' and decimal '10'. The second command is '(gdb) si', showing the instruction '0x08049007 in _start ()'. The third command is '(gdb) info registers eax', showing 'eax' with value '0xffffffff5' and decimal '-11'. The prompt '(gdb) ' is visible at the bottom.

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

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.

So when we perform **AND** operation:

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

We get 5 as the result.

Image:

```
(gdb) info registers eax
eax      0x5      5
(gdb) info registers ax
ax       0x5      5
(gdb) info registers ah
ah       0x0      0
(gdb) info registers al
al       0x5      5
(gdb)
```

Figure 5: Keeping the last 4 bits unchanged

3 The XOR operation

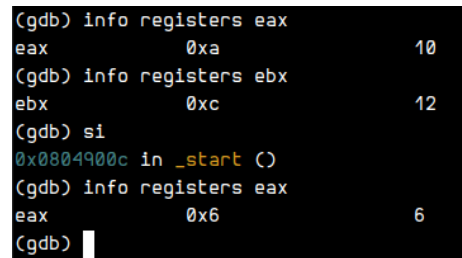
Program:

```
1 section .data
2
3 section .text
4     global _start
5     _start:
6         MOV eax, 0b1010
7         MOV ebx, 0b1100
8
9         XOR eax, ebx
10
11        MOV eax, 1
12        INT 80h
```

We are performing **XOR** operation:

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

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



```
(gdb) info registers eax
eax             0xa             10
(gdb) info registers ebx
ebx             0xc             12
(gdb) si
0x0804900c in _start ()
(gdb) info registers eax
eax             0x6             6
(gdb)
```

Figure 6: **XOR**