ASSEMBLY - NUMBERS

Numerical data is generally represented in binary system. Arithmetic instructions operate on binary data. When numbers are displayed on screen or entered from keyboard, they are in ASCII form.

So far, we have converted this input data in ASCII form to binary for arithmetic calculations and converted the result back to binary. The following code shows this −

```
section .text
   global _start        ;must be declared for using gcc

_start:                 ;tell linker entry point
   mov eax,'3'
   sub     eax, '0'

   mov  ebx, '4'
   sub     ebx, '0'
   add  eax, ebx
   add eax, '0'

   mov  [sum], eax
   mov ecx,msg
   mov edx, len
   mov ebx,1           ;file descriptor (stdout)
   mov eax,4           ;system call number (sys_write)
   int 0x80           ;call kernel

   mov ecx,sum
   mov edx, 1
   mov ebx,1           ;file descriptor (stdout)
   mov eax,4           ;system call number (sys_write)
   int 0x80           ;call kernel

   mov eax,1           ;system call number (sys_exit)
   int 0x80           ;call kernel

section .data
msg db "The sum is:", 0xA,0xD
len equ $ - msg
segment .bss
sum resb 1
```

When the above code is compiled and executed, it produces the following result −

```
The sum is:
7
```

Such conversions, however, have an overhead, and assembly language programming allows processing numbers in a more efficient way, in the binary form. Decimal numbers can be represented in two forms −

- ASCII form
- BCD or Binary Coded Decimal form

ASCII Representation

In ASCII representation, decimal numbers are stored as string of ASCII characters. For example, the decimal value 1234 is stored as −

```
31 32 33 34H
```

Where, 31H is ASCII value for 1, 32H is ASCII value for 2, and so on. There are four instructions for processing numbers in ASCII representation −

- **AAA** − ASCII Adjust After Addition
- **AAS** − ASCII Adjust After Subtraction
- **AAM** − ASCII Adjust After Multiplication
- **AAD** − ASCII Adjust Before Division

These instructions do not take any operands and assume the required operand to be in the AL register.

The following example uses the AAS instruction to demonstrate the concept −

```
section .text
   global _start         ;must be declared for using gcc

_start:                  ;tell linker entry point
   sub     ah, ah
   mov     al, '9'
   sub     al, '3'
   aas
   or      al, 30h
   mov     [res], ax

   mov edx,len          ;message length
   mov ecx,msg          ;message to write
   mov ebx,1          ;file descriptor (stdout)
   mov eax,4          ;system call number (sys_write)
   int 0x80          ;call kernel

   mov edx,1          ;message length
   mov ecx,res          ;message to write
   mov ebx,1          ;file descriptor (stdout)
   mov eax,4          ;system call number (sys_write)
   int 0x80          ;call kernel

   mov eax,1          ;system call number (sys_exit)
   int 0x80          ;call kernel

section .data
msg db 'The Result is:',0xa
len equ $ - msg
section .bss
res resb 1
```

When the above code is compiled and executed, it produces the following result−

```
The Result is:
6
```

BCD Representation

There are two types of BCD representation −

- Unpacked BCD representation
- Packed BCD representation

In unpacked BCD representation, each byte stores the binary equivalent of a decimal digit. For example, the number 1234 is stored as −

```
01 02 03 04H
```

There are two instructions for processing these numbers −

- AAM - ASCII Adjust After Multiplication
- AAD - ASCII Adjust Before Division

The four ASCII adjust instructions, AAA, AAS, AAM, and AAD, can also be used with unpacked BCD representation. In packed BCD representation, each digit is stored using four bits. Two decimal digits are packed into a byte. For example, the number 1234 is stored as −

```
12 34H
```

There are two instructions for processing these numbers −

- DAA - Decimal Adjust After Addition
- DAS - decimal Adjust After Subtraction

There is no support for multiplication and division in packed BCD representation.

Example

The following program adds up two 5-digit decimal numbers and displays the sum. It uses the above concepts −

```
section .text
   global _start        ;must be declared for using gcc

_start:                 ;tell linker entry point

   mov     esi, 4       ;pointing to the rightmost digit
   mov     ecx, 5       ;num of digits
   clc
add_loop:
   mov  al, [num1 + esi]
   adc  al, [num2 + esi]
   aaa
   pushf
   or  al, 30h
   popf

   mov [sum + esi], al
   dec esi
   loop add_loop

   mov edx,len          ;message length
   mov ecx,msg          ;message to write
   mov ebx,1            ;file descriptor (stdout)
   mov eax,4            ;system call number (sys_write)
   int 0x80             ;call kernel

   mov edx,5            ;message length
   mov ecx,sum          ;message to write
   mov ebx,1            ;file descriptor (stdout)
   mov eax,4            ;system call number (sys_write)
   int 0x80             ;call kernel

   mov eax,1            ;system call number (sys_exit)
   int 0x80             ;call kernel

section .data
msg db 'The Sum is:',0xa
len equ $ - msg
num1 db '12345'
num2 db '23456'
sum db '     '
```

When the above code is compiled and executed, it produces the following result −

```
The Sum is:
35801
```