

Assembly Programming II

Objectives.

- Implement simple assembly programs covering the following topics:
 - Defining different types of variables.
 - Performing integer arithmetic: addition, subtraction, multiplication and division.

Aim

The aim of this session is to learn how to write simple assembly programs

Registers

x86 processors have eight 32-bit general purpose registers (top of figure in Figure 1). The first letter of their name is always the letter 'e'. Their names are historical. The EAX, EBX, ECX, and EDX registers, can be treated as 16-bit and 8-bit registers too. This is because older processors used to support shorter registers. The least significant 2 bytes of EAX for example, can be treated as a 16-bit register called AX. The least significant byte of AX can be used as a single 8-bit register called AL, while the most significant byte of AX can be used as a single 8-bit register called AH. These 8-bit registers refer to the same physical 16-bit register. When a two-byte quantity is placed into AX, the update affects the value of AH, AL, and EAX.

Note that there are special purpose registers too, e.g., instruction pointer (EIP). The developer cannot amend the values of those registers.

X64 processors have sixteen 64-bit general purpose registers (bottom of figure in Figure 1), eight of which extend the 32-bit registers. In this module, we will focus on the x86 architecture.

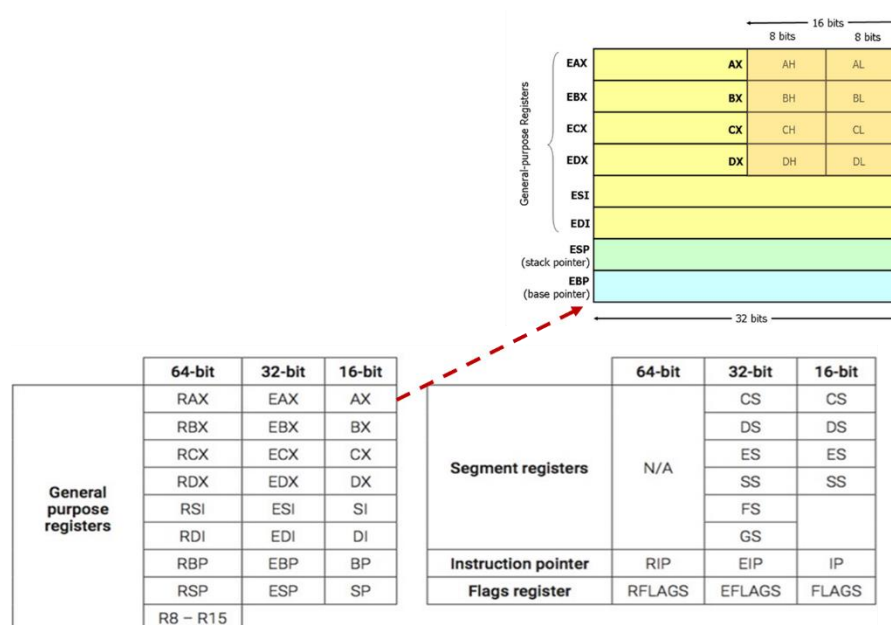


Fig.1 x86-64 architecture registers

Example #1 - Multiplication (MUL instruction)

The multiplication (mul instruction) in x86 is more complicated than the addition and subtraction. The mul instruction has only one operand, that of the multiplier. The multiplicand is always the eax/ax/al register when applying 32/16/8 bit multiplication, respectively (Fig.2).

| | | $2 \times 3 = 6$ |
|------------|--------------|------------------|
| Multiplier | Multiplicand | Product |
| M8/R8 | <i>al</i> | <i>ax</i> |
| M16/R16 | <i>ax</i> | <i>dx:ax</i> |
| M32/R32 | <i>eax</i> | <i>edx:eax</i> |
| M64/R64 | <i>rax</i> | <i>rdx:rax</i> |

Fig.2 MUL instruction

When applying multiplication, the product might need up to twice the number of the input bits. Consider the case where $99 \times 99 = 9801$ in decimal. The input operands have two digits while the product has 4 digits. In binary, when multiplying two 8bit numbers the product needs 16bit, when multiplying two 16bit numbers, the product needs 32bit etc. Consider the following 2-bit multiplicand 3_{10} (11_2) and 2-bit multiplier 3_{10} (11_2). The product is 9_{10} (1001_2), and it cannot be contained in 2-bits; it requires 4-bits. At most we require double the size of the multiplier or the multiplicand.

Let us write the assembly code for multiplying two 32bit numbers, **1001 x 999**. The 999 must be stored to *eax* register. The 1001 can be stored into any 32bit register. The result is stored into two 32bit registers (*edx* and *eax*); *eax* is used for the less significant part while *edx* is used for the most significant part. $1001 \times 999 = 999,999$ and therefore the result needs just 32bits. In this case, *edx*=0 and *eax*=999,999. The code follows

```
.data ; data segment
; define your variables here
Lower_sum DWORD 0 ; define a variable sum (32-bit) with initial value 0
higher_sum DWORD 0 ; define a variable sum (32-bit) with initial value 0
a DWORD 1001 ; multiplier
b DWORD 999 ; multiplicand

.code ; code segment

main PROC ; main procedure
; write your assembly code here
mov eax, a ; move multiplicand to eax
mul b ; do a x b and put the result into edx:eax
mov Lower_sum, eax ; move eax value to sum
mov higher_sum, edx ;
```

What if $a=2500000$ and $b=3000000$? In this case the result is $7.500.000.000.000$ and needs more than 32bits. Therefore, part of the result will be stored into *edx* and part into *eax*. In this case *eax*=987101184 and *edx*=1746. This is further explained below

$$987101184_{10} = 0011\ 1010\ 1101\ 0101\ 1111\ 1000\ 0000\ 0000_2$$

$$1746_{10} = 110\ 1101\ 0010_2$$

If we combine the binary sequences of the two numbers above, the answer will be $7.500.000.000.000$ as

$$7.500.000.000.000_{10} = 110\ 1101\ 0010\ 0011\ 1010\ 1101\ 0101\ 1111\ 1000\ 0000\ 0000_2$$

Example #2 - Division.

The power rating P is determined by the current rating I and the voltage rating V . The relationship is given by: $P = VI$. Given $P=2200$ watts and $V=220$ volts, write a program that would compute the current I .

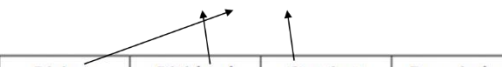
Solution

Dr. Vasilios Kelefouras, University of Plymouth, School of Engineering, Computing and Mathematics

When applying multiplication, the product might need up to twice the number of the input bits. For the same reason, in division, the dividend might contain up to twice the number of the divisor and quotient bits. For example, in $P = V \times I$, P needs more bits than V and I . Thus, if we divide this expression by V , then $I = P/V$ and P might have more bits.

First, we must define the aforementioned variables. To do so, 16-bits are enough to store the maximum value, i.e., 2200. Thus we will define the variables using the 'WORD' data type. Our program needs to perform the following operation $I = P/V$. The multiplication and the division instructions are not that straightforward (see lecture slides and Fig.3). In order to divide two 16-bit values, we must store the dividend into 'dx' and 'ax' 16-bit registers. Therefore, the dividend consists of two 16-bit values and thus it is 32-bit. This is because in the general case, if the quotient is 16bits, the dividend can be 32bits. We will ignore this and we will divide two 16-bit numbers and we will store the result into another 16-bit register (Fig.3). Thus, we will clear the high part of the dividend (mov dx, 0). To sum up, to perform $I = P/V$, we have to store the 'P' value into the 'ax' register and the 'V' value into any other 16-bit register we want (let's say 'cx'). Then, we will call the 'DIV' instruction and the quotient will be stored into 'ax' while the remainder into 'dx' registers.

40 : 8 = 5



| Divisor | Dividend | Quotient | Remainder |
|---------|----------|----------|-----------|
| M8/R8 | ax | al | ah |
| M16/R16 | dx:ax | ax | dx |
| M32/R32 | edx:eax | eax | edx |
| M64/R64 | rdx:rax | rax | rdx |

Fig.3 DIV instruction

The code follows

```
.data
    power WORD 2200
    volt WORD 220
    zero WORD 0
    quotient WORD 0
    remainder WORD 0

.code
main PROC
    mov dx, zero
    mov ax, power
    mov cx, volt
    div cx
    mov quotient, ax
    mov remainder, dx
```

The above task can be implemented by using 8bit division too. This is because the dividend is stored into two 8bit registers and the divisor needs 8bits. Can you write the assembly program?

Example #3 - Multiplication and Division.

Consider that a module has two pieces of assessments: a coursework and a test; each component carries 50% of the total module marks. Below we give the marks obtained by a student.

- a. Coursework: 35 out of 60.
- b. Test: 45 out of 55.

Write a program to compute the student's overall percentage in the module.

Hint: Given two components a and b with equal weighting, if a student gets x out of c in a, and y out of d in b, then the overall percentage can be calculated with the following formula:

$$[(x/c)*(50/100)] + [(y/d)*(50/100)]*100$$

You may need to rearrange this to get an appropriate result. Why? This is because integer division in the current form would lead to wrong results (we will not study floating point division here). Rearranging the formula becomes:

$$(50xd + 50yc)/cd$$

Try to apply integer division in x/c and y/d , the result will be zero as the quotient's range is always $0 \leq \text{quotient} < 1$. However, using the above formula this problem is eliminated as the quotient's value is a percentage (not $0 \leq \text{quotient} < 1$).

Thus, the above problem is simplified into writing an assembly program that does $(50xd + 50yc)/cd$. You already know how to do the division. See the two multiplication examples on the lecture slides and try to write this program on your own. The solution is provided on github.

Further Reading

1. Chapter 1 and Chapter 2 in 'Modern X86 Assembly Language Programming' , available at <https://www.pdfdrive.com/download.pdf?id=185772000&h=3dfb070c1742f50b500f07a63a30c86a&u=cache&ext=pdf>
2. x86 and amd64 instruction reference Available at All the x86 instructions can be found here <https://www.felixcloutier.com/x86/> .