Load Effective Address
Part I

Written By: Vandad Nahavandi Pour
Email: AlexiLaiho.cob@GMail.com
Web-site: http://www.ASMTrauma.com

1 Introduction

One of the instructions that is well known to Assembly programmers, specially those who know about its advantages and disadvantages, is the LEA[1] instruction. The LEA Instruction can be very useful because it can *compute* an effective address with a factor, an index a base and a displacement. These are all the addressing modes available in IA-32 Architecture of CPUs and you can combine all these in one instruction. Despite what a lot of people think, the LEA instruction is *not* used to work with a memory address but to perform a rather mathematical-related operation. In this article, we are going to discuss how this instruction can and should(n't) be used.

2 Overview of the LEA instruction

Before we delve into the world of Effective Addresses and ways of computing them with the LEA instruction, it would be of no harm to learn the general form of the instruction first:

```
LEA    r16, m
LEA    r32, m
LEA    r64, m
```

As you can see, this instruction is also available for 16 bit and 64 bit modes of the CPU. The operand to the right side of this instruction is called the source operand and to the left (r16, r32, r64) we have the destination. Our destination can be a 16bit, a 32-bit or a 64 bit CPU's General Purpose Register such as the accumulator (EAX), the Base Index (EBX), the Source Index (ESI) and etc.

The source operand of the LEA instruction can only be a memory address (offset). For example, if you have declared a 32-bit variable called *Integer1*, you can retrieve its effective address in this way:

```
LEA    EAX , Integer1
```

Note that different assemblers have different syntaxes. For example, the above code snippet should be coded as shown below, if you are coding in NASM:

```
LEA    EAX , [Integer1]
```

[1] Load Effective Address

The important thing to note about this instruction is that it does *not* put the value of the variable *Integer1* into EAX. However, it does move the effective address of this variable into the accumulator.

In the next section, we will be talking about the advantages and disadvantages of the LEA instruction that will help you optimize your programs for size and speed.

3 Optimization with LEA

The LEA instruction can help you optimize your code in different ways. Some of these are listed below:

1. Add a value to itself
2. Add two General Purpose Registers and put the result in another
3. Multiply a General Purpose Register by 2, 4 or 8
4. Add a General Purpose Register to another multiplied by 2, 4 or 8
5. Add a General Purpose Register to another multiplied by 2, 4 or 8 plus a 32-bit value.

The above list enumerates the advantages brought to your code from the instruction's point of view but you can certainly list a lot of other advantages of using the *LEA* instruction such as vector operations, single and multidimensional array access and etc.

The *LEA* instruction or any other instruction that uses effective addresses, can be made of 4 parts as shown below:

$$\text{SEGMENT} + \text{BASE} + (\text{INDEX} * \text{SCALE}) + \text{DISPLACEMENT}$$

Now let us describe each of these elements shown above:

- **SEGMENT:** This element specifies the segment in which the address should be located. This can be any of the valid segments either in RM[2] or in PM[3], such as *CS, DS, ES, SS, GS, FS.*
- **BASE:** The base element specifies the offset into the segment. For example, if you specify the Code Segment (CS) as the Segment element described below and set the base to 0x0000FFFF, the address will be *[CS:0x0000FFFF]*. The BASE element can be any of the general purpose registers in IA-32, such as *EAX, EBX, ECX, EDX, ESI, EDI.*
- **INDEX*SCALE** The index multiplied by an scale allows a factor to be multiplied by the scale of either 2, 4 or 8. For example, you want to navigate inside a single dimensional array with the length of 50 DWORD[4]. Now if you set a counter to count from 0 to 25 (50/2) and multiply this counter by 4 each time, you will get 0, 4, 8, 12, 16, etc. This will give you a perfect opportunity for filling the array as you will see in the examples provided below.

[2]Real Mode
[3]Protected Mode
[4]DWORD = Double Word = 4 Bytes

- **DISPLACEMENT** Last but not least, is the displacement element in calculating an effective address. The displacement is simply a 32-bit value in IA-32 that allows a constant to be added to the calculated address.

We talked about the advantages of the *LEA* instruction but we never explained them in details. How about doing it right now?

3.1 Add a value to itself

You can provide the BASE and the INDEX element to calculate a value multiplied by 2. For example, if you want to calculate *EAX*2*, then you can use the below combination of addressing modes in the *LEA* instruction:

```
LEA    EAX , [EAX+EAX]
```

This will effectively add the EAX Register to itself. You can do the same thing with other general purpose registers, too. Note that there are other ways for calculating what the above *LEA* instruction does such as:

```
ADD    EAX , EAX
```

Or

```
SHL    EAX , 01
```

These three instruction run at the speed of 1 Clock cycle on a PIII 800 MHZ machine with 512 MB SD RAM.

3.2 Add two General Purpose Registers and put the result in another

As you saw in the previous sub-section, you could add a general purpose register to itself in order to calculate its value multiplied by 2. In addition to that, you can use another general purpose register as the destination and also another for the index. For example, if you want to calculate *EAX = ECX+EDX*, you can use the *LEA* instruction in this way:

```
LEA    EAX , [ECX+EDX]
```

Note that the above syntax is also compatible with what NASM uses. If you like to specify the size of the source operand (the side to the left of the *LEA* Instruction), you can use the *DWORD/WORD/BYTE* size specifiers.

3.3 Multiply a General Purpose Register by 2, 4 or 8

To be able to multiply a general purpose register by 2, 4 or 8, you can use the LEA instruction with INDEX*SCALE as shown below:

```
LEA    EAX , [ECX+04]
```

The above example calculates *EAX = ECX*4*. You can also specify the destination operand (the one to the left) to be the same as the source operand. For example:

```
LEA    EAX , [EAX+04]
```

And the above example calculates *EAX = EAX*4*. Note that you can do this operation with the *SHL* instruction, too.

3.4 Add a General Purpose Register to another multiplied by 2, 4 or 8

This way of using the *LEA* instruction starts to show you one of the advantages it has over normal addressing modes mostly used in programs and generated by a log of compilers. For example, you need to calculate *EAX = ECX + (EDX*02)*. For this, you can use the LEA instruction with:

- **EAX** as the destination
- **ECX** as the BASE
- **EDX** as the INDEX
- **02** as the SCALE

Here is an example:

```
LEA    EAX , [ECX + EDX*04]
```

Note that the SCALE is preferred to be mentioned to the left side of the INDEX as in:

```
LEA    EAX , [ECX + 04*EDX]
```

This way you will avoid confusion when a DISPLACEMENT is also used in the effective addressing mode, as explained next.

3.5 Add a General Purpose Register to another multiplied by 2, 4 or 8 plus a 32-bit value.

Among 4 different addressing modes explained above, this mode is the most complete, taking advantage of all possible factors involved in the calculation of an effective address. In this mode, you can for example calculate *EAX = ECX + (EDX*08) + 25* in which case:

- **EAX** is the destination
- **ECX** is the BASE
- **EDX** is the INDEX
- **08** is the SCALE
- **25** is the DISPLACEMENT

Note that the DISPLACEMENT is a constant value and can not be a register. Now let us calculate the address shown above:

```
LEA    EAX , [ECX + 08*EDX + 25]
```

After reading these instructions, you should not be able to proceed to the next section which gives an example on how you can optimize your program for both speed and code size, using the *LEA* instruction.

4 Putting it all to work

Now let us consider an example: we want to calculate the below value using first normal integer instructions such as SHL and ADD and etc:

ECX = (EAX * 2) + (EDX * 8) + 100

A simple way for doing this is using the *SHR* and/or the *ADD* instruction as shown below:

```
MOV EAX , Integer1
MOV EDX , Integer2
SHL EAX , 01
SHL EDX , 03
ADD EDX , 100
ADD EDX , EAX
MOV ECX , EDX
```

As you can see, the above code snippet, although working fine, has a dependency chain on the EDX Register. At the 7^{th} line of the code, you can see that the *ECX* register should wait for the result of the *EDX* register in the previous line to be calculated which itself depends on the previous line and that line depends on its previous. This is a long dependence chain. However, the above code runs at the speed of 10 clock cycles on a PIII 800 MHZ with 512 MB SDRAM with the code aligned on a DWORD boundary, if both *Integer1* and *Integer2* are declared in the stack.

Now let us consider improving the above code by first moving one of the values to the *ECX* register to move the *EDX* out of the whole code:

```
MOV EAX , Integer1
MOV ECX , Integer2
SHL EAX , 1
SHL ECX , 3
ADD ECX , EAX
ADD ECX , 100
```

You might be surprised to hear that the above code runs slower than the previous one, at the speed of 11 clock cycles on the same CPU with the same conditions. You can try to speed this code up by breaking the dependency chain at the 4^{th}, 5^{th} and the 6^{th} steps of the code.

Now let us take advantage of the *LEA* instruction to see what will happen and whether or not we will gain any improvements over the speed of execution:

```
MOV EAX , Integer1
MOV EDX , Integer2
ADD EAX , EAX
LEA ECX , DWORD PTR [EAX+08*EDX+100]
```

The above code does the expected job for us but at the speed of 8 clock cycles on the same CPU and with the same conditions for the code segment and computer memory. This is compared to 11 and 10 clock cycles achieved by having used the previous methods of pure arithmetic instructions. Note that if

you change the *ADD* instruction to *SHL* in order to multiply the accumulator[5] by 2, you will get the same speed in the execution which is 8 clock cycles.

5 Conclusion

Using the *LEA* instruction can be very advantageous specially if you break dependency chains between different lines of code in your program. The LEA Instruction pairs on both the *u* and the *v* pipe only if the two adjacent executions of this instruction do not depend on the result of each other[6].

You can also use the *LEA* instruction to move constant values to general purpose registers, for example:

```
LEA    EAX , [0]
```

This can be used as a very effective way of breaking dependency chains. There is much more to the *LEA* instruction that I will explain in the next article so that you will further understand the advantages that can be gained by using this instruction.

[5] EAX

[6] To be more precise, the second instruction should not depend on the result of the first