Load Effective Address
Part II

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

1 Introduction

In the previous manual, we discussed some general usage examples of the LEA[1] instruction. In this article, we are going to solve a simple equation with different methods each of which is given its time of execution measured by clock cycles. You will also see how useful some instructions are when optimizing for speed and/or for size of the code.

2 An example for the LEA instruction

In this section, we are given a problem that we have to solve. Suppose that we have declared two local variables namely, *Integer1* and *Integer2* in these locations

```
Integer1 = DWORD [ESP - 0x04]
Integer2 = DWORD [ESP - 0x08]
```

What we are asked to compute is:

```
EAX = ((Integer1 * 10) + (Integer2 * 6) + 100) / 4
```

I have provided 6 solutions for solving this problem. Below sub-sections present these solutions one by one, with the slowest code in execution provided first and the fastest provided at last.

2.1 Solution 1

The above problem can be solved by a beginner assembly programmer by performing the following calculations:

1. Multiply Integer1 by 10
2. Multiply Integer2 by 6
3. Add the result of the first step to the second or vice versa
4. Add 100 to the result of the previous step
5. Divide the result of the previous step by 4

[1] Load Effective Address

We take the 5^{th} step a little farther and instead of dividing, we shift the result to the right 2 bits. A sample program that demonstrates these steps is given below:

```
MOV EAX , Integer1
MOV EDX , 10
MUL EDX
MOV ECX , EAX
MOV EAX , Integer2
MOV EDX , 06
MUL EDX
ADD EAX , ECX
ADD EAX , 100
SHR EAX , 2
```

The above code snippet is executed at the speed of 25 clock cycles on a PIII 800MHZ CPU with 512 MB SDRAM while the code and the stack segments are aligned on a DWORD boundary. You can actually see the dependency chains in this code snipped. We can speed this process up by breaking these dependency chains but we would only be writing a code that runs 1 to 2 clock cycles faster than the one provided above.

2.2 Solution 2

Using the *MUL* instruction was one of the reasons that the code provided in the previous sub-section was running slowly. How about using the *IMUL* instruction in this form:

```
IMUL    r32 , imm32
```

Where **r32** is a 32-bit general purpose register which is multiplied by a Double Word[2] immediate value at **imm32**. A sample program that takes advantage of this form of the *IMUL* instruction is given below which calculates the same value as did the previous example:

```
MOV EAX , Integer1
IMUL EAX , 10
MOV ECX , Integer2
IMUL ECX , 6
ADD EAX , ECX
ADD EAX , 100
SHR EAX , 2
```

The above code snippet has certainly improved compared to the one mentioned in **Solution 1**, however, it is not what we want. This code snippet executes at the speed of 22 clock cycles on the same CPU with the same condition of the stack and the code segment with the same memory.

[2]DWORD

2.3 Solution 3

In the 5^{th} and the 6^{th} lines of the code provided in the previous sub-section, we added the value of the *ECX* register to the accumulator[3] and then added the value 100 to the result, also in the accumulator. As described in the previous article, we can eliminate these steps and combine them into one single step: the LEA instruction like this:

```
LEA    EAX , DWORD PTR [EAX+ECX+100]
```

Therefore, the complete solution would be:

```
MOV EAX , Integer1
IMUL EAX , 10
MOV ECX , Integer2
IMUL ECX , 6
LEA EAX , DWORD PTR [EAX+ECX+100]
SHR EAX , 2
```

The above code runs at the speed of 21 clock cycles on the same CPU with the same conditions as stated above. This is better but not good enough. Read on...

2.4 Solution 4

You must know that multiplication and division are very time consuming on any processor including *AMD*. What if we could eliminate two multiplications that we are using in our solution?

To be able to multiply a value by 10 (as in our first multiplication), you can perform the following calculations:

1. Add the value to itself in order to multiply it by 2.
2. Multiply the result of the above step by 4.
3. Add the result of the first and the second steps.

For example, if you want to multiply 6 by 10 then you can do these:

1. Add the value to itself: $6 + 6 = 12$
2. Multiply the above result by 4: $12 \times 4 = 48$
3. Add the two results together: $48 + 12 = 60$

The result of the last operation is 60 which is equal to 6×10. We can do these programmatically with the *LEA* instruction, too:

```
ADD EAX , EAX LEA EAX , DWORD PTR [EAX + 0x04 *
EAX]
```

[3] EAX

Now that we know how to multiply a value by 10, we are only left to solve the same puzzle for multiplication by 6. This is also easy! All you have to do is to perform the following calculations on a value in order to multiply that value by 6:

1. Add the value to itself in order to multiply it by 2.
2. Multiply the result of the above step by 2.
3. Add the results of the two above steps.

An example is when you need to multiply 7 by 6:

1. Add the value to itself: $7 + 7 = 14$
2. Multiply the above result by 2: $14 \times 2 = 28$
3. Add the two above results together: $28 + 14 = 42$

You can see that 42 is equal to 6×7. We can do the same programmatically using the LEA instruction, of course:

```
ADD EAX , EAX
LEA EAX , DWORD PTR [EAX + 0x02 * EAX]
```

Okay, so let us start writing the final code snippet:

```
MOV EDX , Integer2
MOV EAX , Integer1
ADD EAX , EAX
LEA EAX , DWORD PTR [EAX+04h*EAX+100]
ADD EDX , EDX
LEA EDX , DWORD PTR [EDX+02*EDX]
ADD EAX , EDX
SHR EAX , 02
```

The above code runs at the speed of 12 clock cycles in comparison to the 3 previous codes that run at over 20 clock cycles. You can see that the fourth step of the code also eliminates the need for an extra *ADD* instruction to add the value of 100 to the result. It does it using the 32-bit immediate displacement that can be used in the *LEA* instruction. To find more about displacement and how it can and should be used, read the first article of these series available for download at the link provided at top of this article.

Note that the pairability of the instructions used in the above code snippet is really important in how fast the whole group of instructions can be executed on a particular CPU[4]. For this reason, I will provide some of the general forms of the instructions used in the above example with their pairabilities and number of clock cycles they take to execute:

r32 = Register 32 bit, m32 = Memory Operand 32 bit, u = Pairable in u-pipe, v = Pairable in v-pipe

[4]Central Processing Unit

```
MOV     r32 , m32       1 clock cycle   uv
ADD     r32 , r32       1 clock cycle   uv
LEA     r32 , m         1 clock cycle   uv
NOP                     1 clock cycle   uv
SHR     r32 , imm8      1 clock cycle   u
```

There is one instruction in the above table that has not been used in the previous example. I will use this instruction in the last solution to our problem to show you how you can use pairability in order to gain speed.

2.5 Solution 5

As shown in the previous solution, two *MOV* instructions can be paired in both the *u* and the *v* pipes. This means that using two *MOV* instructions is beneficial if you know the rest of the instructions are paired correctly, too. This does not mean that a *MOV* and an *ADD* instruction can not be paired because they both pair on the *u* and the *v* pipes. Now look at the below code which has more dependency chains than the previous example:

```
MOV EAX , Integer1
ADD EAX , EAX
LEA EAX , DWORD PTR [EAX+04h*EAX+100]
MOV EDX , Integer2
ADD EDX , EDX
LEA EDX , DWORD PTR [EDX+02*EDX]
ADD EAX , EDX
SHR EAX , 02
```

Surprisingly, the above code runs 1 clock cycle faster than the previous (runs at 11 clock cycles). The next solution which is the last, will provide another way of solving this problem which executes faster than all the rest of the solutions provided above.

2.6 Solution 6

It is time that we take advantage of the *NOP* instruction to speed up our code snippet again. In this example, I will give you the exact amount of clock cycles that the CPU needs to execute instructions up to a certain point. You will then be able to see how pairing comes handy:

```
MOV EDX , Integer2 ; 8
MOV EAX , Integer1 ; 7
ADD EAX , EAX ; 8
ADD EDX , EDX ; 7
LEA EAX , DWORD PTR [EAX+04h*EAX+100] ; 9
LEA EDX , DWORD PTR [EDX+02*EDX] ; 9
NOP ; 10
ADD EAX , EDX ; 14
SHR EAX , 02 ; 10
```

The above code snippet runs at the speed of 10 clock cycles together with the stack frame creation and its destruction. The test is done on a PIII 800MHZ CPU with 512 MB SDRAM and with both the code and the stack segments aligned on a DWORD boundary.

The numbers mentioned after each line of the code denotes the number of clock cycles that the CPU would take to execute the instructions up to that point. For example, the CPU would take 9 clock cycles to execute instructions from the first *MOV* to the last *LEA* or it would take 14 clock cycles to execute instructions between the first *MOV* to the last *ADD*.

One thing you need to bear in mind is that all the timings mentioned in this article includes the clock cycles that the CPU would take for creating and destroying the stack frame (PUSH EBP, etc).

You can see that the CPU needs 7 clock cycles to execute the code up to the second *MOV* and also 7 clock cycles to execute up to the second *ADD*, but how is that possible? It is because the first two *MOV* instructions do not use the *ALU* but the two *ADD* instructions do which is why each group of 2 instructions can be executed in parallel.

Also note our little *NOP* that comes very handy for the sake of pairability. Up to the *NOP* instruction, inclusive, the CPU would need 10 clock cycles to execute the code but the same amount of clock cycles is needed for the code including the last two instructions. The reason is that the last three instructions are executed by the CPU all at once because the *SHR* insrtuction, as stated before, can only pair on the *u*-pipe and the CPU would suffer a stall after the *ADD* instruction which pairs with the *NOP*.

3 Conclusion

In this article, you saw how important the use of the *LEA* instruction is and how you can take advantage of pairing in order to speed up your code. Testing the code for pairability is a great programming practice but this can not be done for every single line of the code that you write. If it takes 2 days to write a program, it would take a month to perform such optimization tricks on it, therefore, no matter how important the speed of execution is, you should also take into accound how many precious moments you have to put in making the code execute faster. All in all, if you have a general idea of what you are doing with instructions and how they can and should be used, go ahead and take your time optimizing your program.