40 Basic Practices in Assembly Language Programming

Zuoliu Ding

Rate me: 4.96/5 (132 votes)

29 Jan 2019 CPOL 49 min read

A discussion on some basic practices highly recommended in Assembly Language Programming.

Download Loop Test Project - 34.8 Kb

Download Calling ASM Proc In C Project - 2.9 KB

Download Test ADDR - 4.2 KB

Contents

Introduction

Assembly language is a low-level programming language for niche platforms such as IoTs, device drivers, and embedded systems. Usually, it’s the sort of language that Computer Science students should cover in their coursework and rarely use in their future jobs. From TIOBE Programming Community Index, assembly language has enjoyed a steady rise in the rankings of the most popular programming languages recently.

In the early days, when an application was written in assembly language, it had to fit in a small amount of memory and run as efficiently as possible on slow processors. When memory becomes plentiful and processor speed is dramatically increased, we mainly rely on high level languages with ready made structures and libraries in development. If necessary, assembly language can be used to optimize critical sections for speed or to directly access non-portable hardware. Today assembly language still plays an important role in embedded system design, where performance efficiency is still considered as an important requirement.

In this article, we’ll talk about some basic criteria and code skills specific to assembly language programming. Also, considerations would be emphasized on execution speed and memory consumption. I'll analyze some examples, related to the concepts of register, memory, and stack, operators and constants, loops and procedures, system calls, etc.. For simplicity, all samples are in 32-bit, but most ideas will be easily applied to 64-bit.

All the materials presented here came from my teaching [1] for years. Thus, to read this article, a general understanding of Intel x86-64 assembly language is necessary, and being familiar with Visual Studio 2010 or above is assumed. Preferred, having read Kip Irvine's textbook [2] and the MASM Programmer's Guide [3] are recommended. If you are taking an Assembly Language Programming class, this could be a supplemental reading for studies.

About instruction

The first two rules are general. If you can use less, don't use more.

1. Using less instructions

Suppose that we have a 32-bit `DWORD` variable:

ASM

```
.data
   var1 DWORD 123
```

The example is to add `var1` to `EAX`. This is correct with `MOV` and `ADD`:

ASM

```
mov ebx, var1
add eax, ebx
```

But as `ADD` can accept one memory operand, you can just

ASM

```
add eax, var1
```

2. Using an instruction with less bytes

Suppose that we have an array:

ASM

```
.data
   array DWORD 1,2,3
```

If want to rearrange the values to be 3,1,2, you could

ASM

```
mov eax,array             ;         eax =1
xchg eax,[array+4]        ; 1,1,3,  eax =2
xchg eax,[array+8]        ; 1,1,2,  eax =3
xchg array,eax            ; 3,1,2,  eax =1
```

But notice that the last instruction should be `MOV` instead of `XCHG`. Although both can assign `3` in `EAX` to the first array element, the other way around in exchange `XCHG` is logically unnecessary.

Be aware of code size, `MOV` takes 5-byte machine code but `XCHG` takes 6, as another reason to choose `MOV` here:

ASM

```
00000011  87 05 00000000 R      xchg array,eax
00000017  A3 00000000 R         mov array,eax
```

To check machine code, you can generate a Listing file in assembling or open the Disassembly window at runtime in Visual Studio. Also, you can look up from the Intel instruction manual.

About register and memory

In this section, we'll use a popular example, the nth Fibonacci number, to illustrate multiple solutions in assembly language. The C function would be like:

C++

```cpp
unsigned int Fibonacci(unsigned int n)
{
    unsigned int previous = 1, current = 1, next = 0;
    for (unsigned int i = 3; i <= n; ++i)
    {
        next = current + previous;
        previous = current;
        current = next;
    }
    return next;
}
```

3. Implementing with memory variables

At first, let's copy the same idea from above with two variables `previous` and `current` created here

ASM

```
.data
   previous DWORD ?
   current  DWORD ?
```

We can use EAX store the result without the next variable. Since MOV cannot move from memory to memory, a register like EDX must be involved for assignment previous = current. The following is the procedure FibonacciByMemory. It receives n from ECX and returns EAX as the nth Fibonacci number calculated:

ASM

```
;------------------------------------------------------------
FibonacciByMemory PROC
; Receives: ECX as input n
; Returns: EAX as nth Fibonacci number calculated
;------------------------------------------------------------
   mov   eax,1
   mov   previous,0
   mov   current,0
L1:
   add eax,previous       ; eax = current + previous
   mov edx, current       ; previous = current
   mov previous, edx
   mov current, eax
loop   L1
   ret
FibonacciByMemory ENDP
```

4. If you can use registers, don't use memory

A basic rule in assembly language programming is that if you can use a register, don't use a variable. The register operation is much faster than that of memory. The general purpose registers available in 32-bit are EAX, EBX, ECX, EDX, ESI, and EDI. Don't touch ESP and EBP that are for system use.

Now let EBX replace the previous variable and EDX replace current. The following is FibonacciByRegMOV, simply with three instructions needed in the loop:

ASM

```
;------------------------------------------------------------
FibonacciByRegMOV PROC
; Receives: ECX as input n
; Returns: EAX, nth Fibonacci number
;------------------------------------------------------------
   mov   eax,1
   xor   ebx,ebx
   xor   edx,edx
L1:
   add  eax,ebx      ; eax += ebx
   mov  ebx,edx
   mov  edx,eax
loop   L1
   ret
FibonacciByRegMOV ENDP
```

A further simplified version is to make use of `XCHG` which steps up the sequence without need of `EDX`. The following shows `FibonacciByRegXCHG` machine code in its Listing, where only two instructions of three machine-code bytes in the loop body:

ASM

```
             ;------------------------------------------------------------
000000DF      FibonacciByRegXCHG PROC
             ; Receives: ECX as input n
             ; Returns: EAX, nth Fibonacci number
             ;------------------------------------------------------------
000000DF  33 C0         xor   eax,eax
000000E1  BB 00000001   mov   ebx,1
000000E6             L1:
000000E6  93            xchg eax,ebx      ; step up the sequence
000000E7  03 C3         add  eax,ebx      ; eax += ebx
000000E9  E2 FB      loop   L1
000000EB  C3            ret
000000EC      FibonacciByRegXCHG ENDP
```

In concurrent programming

The x86-64 instruction set provides many atomic instructions with the ability to temporarily inhibit interrupts, ensuring that the currently running process cannot be context switched, and suffices on a uniprocessor. In someway, it also would avoid the race condition in multi-tasking. These instructions can be directly used by compiler and operating system writers.

5. Using atomic instructions

As seen above used `XCHG`, so called as atomic swap, is more powerful than some high level language with just one statement:

ASM

```
xchg  eax, var1
```

A classical way to swap a register with a memory `var1` could be

ASM

```
mov ebx, eax
mov eax, var1
mov var1, ebx
```

Moreover, if you use the Intel486 instruction set with the .486 directive or above, simply using the atomic `XADD` is more concise in the Fibonacci procedure. `XADD` exchanges the first operand (destination) with the second operand (source), then loads the sum of the two values into the destination operand. Thus we have

ASM

```
          ;-------------------------------------------------------------
000000EC    FibonacciByRegXADD PROC
          ; Receives: ECX as input n
          ; Returns: EAX, nth Fibonacci number
          ;-------------------------------------------------------------
000000EC  33 C0          xor    eax,eax
000000EE  BB 00000001    mov    ebx,1
000000F3              L1:
000000F3  0F C1 D8       xadd eax,ebx    ; first exchange and then add
000000F6  E2 FB       loop    L1
000000F8  C3             ret
000000F9    FibonacciByRegXADD ENDP
```

Two atomic move extensions are `MOVZX` and `MOVSX`. Another worth mentioning is bit test instructions, `BT`, `BTC`, `BTR`, and `BTS`. For the following example

ASM

```
.data
  Semaphore WORD 10001000b
.code
  btc Semaphore, 6  ; CF=0, Semaphore WORD 11001000b
```

Imagine the instruction set without `BTC`, one non-atomic implementation for the same logic would be

ASM

```
mov ax, Semaphore
shr ax, 7
xor Semaphore,01000000b
```

Little-endian

An x86 processor stores and retrieves data from memory using little-endian order (low to high). The least significant byte is stored at the first memory address allocated for the data. The remaining bytes are stored in the next consecutive memory positions.

6. Memory representations

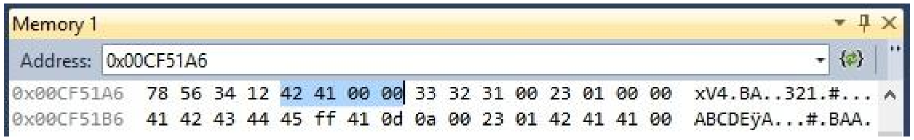
Consider the following data definitions:

ASM

```
.data
dw1 DWORD 12345678h
dw2 DWORD 'AB', '123', 123h
```

```
;dw3 DWORD 'ABCDE'  ; error A2084: constant value too large
by3 BYTE 'ABCDE', 0FFh, 'A', 0Dh, 0Ah, 0
w1 WORD 123h, 'AB', 'A'
```

For simplicity, the hexadecimal constants are used as initializer. The memory representation is as follows:



As for multiple-byte `DWORD` and `WORD` date, they are represented by the little-endian order. Based on this, the second `DWORD` initialized with `'AB'` should be `00004142h` and next `'123'` is `00313233h` in their original order. You can't initialize `dw3` as `'ABCDE'` that contains five bytes `4142434445h`, while you really can initialize `by3` in a byte memory since no little-endian for byte data. Similarly, see `w1` for a `WORD` memory.

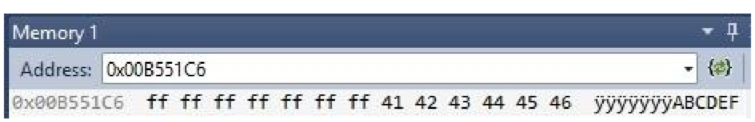
7. A code error hidden by little-endian

From the last section of using `XADD`, we try to fill in a byte array with first 7 Fibonacci numbers, as `01`, `01`, `02`, `03`, `05`, `08`, `0D`. The following is such a simple implementation but with a bug. The bug does not show up an error immediately because it has been hidden by little-endian.

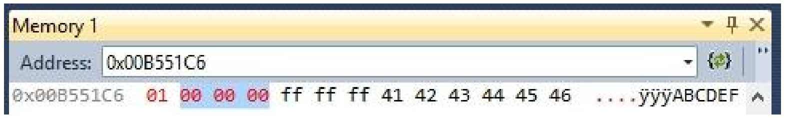
ASM

```
FibCount = 7
.data
FibArray BYTE FibCount DUP(0ffh)
BYTE 'ABCDEF'

.code
   mov  edi, OFFSET FibArray
   mov  eax,1
   xor  ebx,ebx
   mov  ecx, FibCount
 L1:
   mov  [edi], eax
   xadd eax, ebx
   inc  edi
 loop L1
```

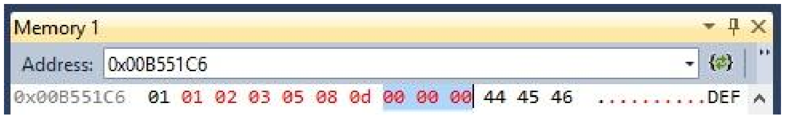
To debug, I purposely make a memory `'ABCDEF'` at the end of the byte array `FibArray` with seven `0ffh` initialized. The initial memory looks like this:



Let's set a breakpoint in the loop. When the first number `01` filled, it is followed by three zeros as this:



But OK, the second number `01` comes to fill the second byte to overwrite three zeros left by the first. So on and so forth, until the seventh `0D`, it just fits the last byte here:



All fine with an expected result in `FibArray` because of little-endian. Only when you define some memory immediately after this `FibArray`, your first three byte will be overwritten by zeros, as here `'ABCDEF'` becomes `'DEF'`. How to make an easy fix?

About runtime stack

The runtime stack is a memory array directly managed by the CPU, with the stack pointer register `ESP` holding a 32-bit offset on the stack. `ESP` is modified by instructions `CALL`, `RET`, `PUSH`, `POP`, etc.. When use `PUSH` and `POP` or alike, you explicitly change the stack contents. You should be very cautious without affecting other implicit use, like `CALL` and `RET`, because you programmer and the system share the same runtime stack.

8. Assignment with PUSH and POP is not efficient

In assembly code, you definitely can make use of the stack to do assignment `previous = current`, as in `FibonacciByMemory`. The following is `FibonacciByStack` where only difference is using `PUSH` and `POP` instead of two `MOV` instructions with `EDX`.

ASM

```asm
;------------------------------------------------------------
FibonacciByStack
; Receives: ECX as input n
; Returns: EAX, nth Fibonacci number
;------------------------------------------------------------
   mov   eax,1
   mov   previous,0
   mov   current,0
L1:
   add  eax,previous      ; eax = current + previous
   push current           ; previous = current
   pop  previous
   mov  current, eax
loop   L1
```

```
   ret
FibonacciByStack ENDP
```

As you can imagine, the runtime stack built on memory is much slower than registers. If you create a test benchmark to compare above procedures in a long loop, you'll find that `FibonacciByStack` is the most inefficient. My suggestion is that if you can use a register or memory, don't use `PUSH` and `POP`.

9. Using INC to avoid PUSHFD and POPFD

When you use the instruction `ADC` or `SBB` to add or subtract an integer with the previous carry, you reasonably want to reserve the previous carry flag (`CF`) with `PUSHFD` and `POPFD`, since an address update with `ADD` will overwrite the `CF`. The following `Extended_Add` example borrowed from the textbook [2] is to calculate the sum of two extended long integers `BYTE` by `BYTE`:

ASM

Shrink ▲

```
;---------------------------------------------------------
Extended_Add PROC
; Receives: ESI and EDI point to the two long integers
;           EBX points to an address that will hold sum
;           ECX indicates the number of BYTEs to be added
; Returns:  EBX points to an address of the result sum
;---------------------------------------------------------
   clc                        ; clear the Carry flag
   L1:
      mov   al,[esi]          ; get the first integer
      adc   al,[edi]          ; add the second integer
      pushfd                  ; save the Carry flag

      mov   [ebx],al          ; store partial sum
      add   esi, 1            ; point to next byte
      add   edi, 1
      add   ebx, 1            ; point to next sum byte
      popfd                   ; restore the Carry flag
   loop   L1                  ; repeat the loop

   mov   dword ptr [ebx],0  ; clear high dword of sum
   adc   dword ptr [ebx],0  ; add any leftover carry
   ret
Extended_Add ENDP
```

As we know, the `INC` instruction makes an increment by 1 without affecting the `CF`. Obviously we can replace above `ADD` with `INC` to avoid `PUSHFD` and `POPFD`. Thus the loop is simplified like this:

ASM

```
L1:
   mov   al,[esi]        ; get the first integer
   adc   al,[edi]        ; add the second integer
```

```
   mov   [ebx],al         ; store partial sum
   inc   esi              ; add one without affecting CF
   inc   edi
   inc   ebx
loop   L1                 ; repeat the loop
```

Now you might ask what if to calculate the sum of two long integers `DWORD` by `DWORD` where each iteration must update the addresses by 4 bytes, as `TYPE DWORD`. We still can make use of `INC` to have such an implementation:

ASM

```
clc
xor   ebx, ebx

L1:
    mov eax, [esi +ebx*TYPE DWORD]
    adc eax, [edi +ebx*TYPE DWORD]
    mov [edx +ebx*TYPE DWORD], eax
    inc ebx
loop  L1
```

Applying a scaling factor here would be more general and preferred. Similarly, wherever necessary, you also can use the `DEC` instruction that makes a decrement by 1 without affecting the carry flag.

10. Another good reason to avoid PUSH and POP

Since you and the system share the same stack, you should be very careful without disturbing the system use. If you forget to make `PUSH` and `POP` in pair, an error could happen, especially in a conditional jump when the procedure returns.

The following `Search2DAry` searches a 2-dimensional array for a value passed in `EAX`. If it is found, simply jump to the `FOUND` label returning one in `EAX` as true, else set `EAX` zero as false.

ASM Shrink ▲

```
;------------------------------------------------------------
Search2DAry PROC
; Receives: EAX, a byte value to search a 2-dimensional array
;           ESI, an address to the 2-dimensional array
; Returns: EAX, 1 if found, 0 if not found
;------------------------------------------------------------
   mov  ecx,NUM_ROW        ; outer loop count

ROW:
   push ecx                ; save outer loop counter
   mov  ecx,NUM_COL        ; inner loop counter

   COL:
      cmp al, [esi+ecx-1]
      je FOUND
```

```
   loop COL

   add esi, NUM_COL
   pop  ecx                ; restore outer loop counter
loop ROW                   ; repeat outer loop

   mov eax, 0
   jmp QUIT
FOUND:
   mov eax, 1
QUIT:
   ret
Search2DAry ENDP
```

Let's call it in `main` by preparing the argument `ESI` pointing to the array address and the search value `EAX` to be `31h` or `30h` respectively for not-found or found test case:

ASM

```
.data
ary2D   BYTE  10h,  20h,  30h,  40h,  50h
        BYTE  60h,  70h,  80h,  90h,  0A0h
NUM_COL = 5
NUM_ROW = 2

.code
main PROC
   mov esi, OFFSET ary2D
   mov eax, 31h            ; crash if set 30h
   call Search2DAry
; See eax for search result
   exit
main ENDP
```

Unfortunately, it's only working in not-found for `31h`. A crash occurs for a successful searching like `30h`, because of the stack leftover from an outer loop counter pushed. Sadly enough, that leftover being popped by `RET` becomes a return address to the caller.

Therefore, it's better to use a register or variable to save the outer loop counter here. Although the logic error is still, a crash would not happen without interfering with the system. As a good exercise, you can try to fix.

Assembling time vs. runtime

I would like to talk more about this assembly language feature. Preferred, if you can do something at assembling time, don't do it at runtime. Organizing logic in assembling indicates doing a job at static (compilation) time, not consuming runtime. Differently from high level languages, all operators in assembly language are processed in assembling such as `+`, `-`, `*`, and `/`, while only instructions work at runtime like `ADD`, `SUB`, `MUL`, and `DIV`.

11. Implementing with plus (+) instead of ADD

Let's redo Fibonacci calculating to implement `eax = ebx + edx` in assembling with the plus operator by help of the `LEA` instruction. The following is `FibonacciByRegLEA` with only one line changed from `FibonacciByRegMOV`.

ASM

```
;------------------------------------------------------------
FibonacciByRegLEA
; Receives: ECX as input n
; Returns: EAX, nth Fibonacci number
;------------------------------------------------------------
   xor   eax,eax
   xor   ebx,ebx
   mov   edx,1
L1:
   lea  eax, DWORD PTR [ebx+edx]  ; eax = ebx + edx
   mov  edx,ebx
   mov  ebx,eax
loop   L1

   ret
FibonacciByRegLEA ENDP
```

This statement is encoded as three bytes implemented in machine code without an addition operation explicitly at runtime:

ASM

```
000000CE  8D 04 1A      lea eax, DWORD PTR [ebx+edx]  ; eax = ebx + edx
```

This example doesn't make too much performance difference, compared to `FibonacciByRegMOV`. But is enough as an implementation demo.

12. If you can use an operator, don't use an instruction

For an array defined as:

ASM

```
.data
   Ary1 DWORD 20 DUP(?)
```

If you want to traverse it from the second element to the middle one, you might think of this like in other language:

ASM

```
mov esi, OFFSET Ary1
add esi, TYPE DWORD    ; start at the second value
```

```
mov ecx LENGTHOF Ary1  ; total number of values
sub ecx, 1
div ecx, 2             ; set loop counter in half
L1:
   ; do traversing
Loop L1
```

Remember that ADD, SUB, and DIV are dynamic behavior at runtime. If you know values in advance, they are unnecessary to calculate at runtime, instead, apply operators in assembling:

ASM

```
mov esi, OFFSET Ary1 + TYPE DWORD   ; start at the second
mov ecx (LENGTHOF Ary1 -1)/2        ; set loop counter
L1:
   ; do traversing
Loop L1
```

This saves three instructions in the code segment at runtime. Next, let's save memory in the data segment.

13. If you can use a symbolic constant, don't use a variable

Like operators, all directives are processed at assembling time. A variable consumes memory and has to be accessed at runtime. As for the last Ary1, you may want to remember its size in byte and the number of elements like this:

ASM

```
.data
   Ary1 DWORD 20 DUP(?)
   arySizeInByte DWORD ($ - Ary1)  ; 80
   aryLength DWORD LENGTHOF Ary1   ; 20
```

It is correct but not preferred because of using two variables. Why not simply make them symbolic constants to save the memory of two DWORD?

ASM

```
.data
   Ary1 DWORD 20 DUP(?)
   arySizeInByte = ($ - Ary1)      ; 80
   aryLength EQU LENGTHOF Ary1     ; 20
```

Using either equal sign or EQU directive is fine. The constant is just a replacement during code preprocessing.

14. Generating the memory block in macro

For an amount of data to initialize, if you already know the logic how to create, you can use macro to generate memory blocks in assembling, instead of at runtime. The following macro creates all `47` Fibonacci numbers in a `DWORD` array named `FibArray`:

ASM

```
.data
val1 = 1
val2 = 1
val3 = val1 + val2

FibArray LABEL DWORD
DWORD val1                ; first two values
DWORD val2
WHILE val3 LT 0FFFFFFFFh  ; less than 4-billion, 32-bit
   DWORD val3             ; generate unnamed memory data
   val1 = val2
   val2 = val3
   val3 = val1 + val2
ENDM
```

As macro goes to the assembler to be processed statically, this saves considerable initializations at runtime, as opposed to `FibonacciByXXX` mentioned before.

For more about macro in MASM, see my article Something You May Not Know About the Macro in MASM [4]. I also made a reverse engineering for the `switch` statement in VC++ compiler implementation. Interestingly, under some condition the `switch` statement chooses the binary search but without exposing the prerequisite of a sort implementation at runtime. It's reasonable to think of the preprocessor that does the sorting with all known `case` values in compilation. The static sorting behavior (as opposed to dynamic behavior at runtime), could be implemented with a macro procedure, directives and operators. For details, please see Something You May Not Know About the Switch Statement in C/C++ [5].

About loop design

Almost every language provides an unconditional jump like `GOTO`, but most of us rarely use it based on software engineering principles. Instead, we use others like `break` and `continue`. While in assembly language, we rely more on jumps either conditional or unconditional to make control workflow more freely. In the following sections, I list some ill-coded patterns.

15. Encapsulating all loop logic in the loop body

To construct a loop, try to make all your loop contents in the loop body. Don't jump out to do something and then jump back into the loop. The example here is to traverse a one-dimensional integer array. If find an odd number, increment it, else do nothing.

Two unclear solutions with the correct result would be possibly like:

ASM

```asm
   mov ecx, LENGTHOF array
   xor esi, esi
L1:
   test array[esi], 1
   jnz ODD
PASS:
   add esi, TYPE DWORD
loop L1
   jmp DONE

ODD:
  inc array[esi]
jmp PASS
DONE:
```

ASM

```asm
   mov ecx, LENGTHOF array
   xor esi, esi
   jmp L1

ODD:
  inc array[esi]
jmp PASS

L1:
   test array[esi], 1
   jnz ODD
PASS:
   add esi, TYPE DWORD
loop L1
```

However, they both do incrementing outside and then jump back. They make a check in the loop but the left does incrementing after the loop and the right does before the loop. For a simple logic, you may not think like this; while for a complicated problem, assembly language could lead astray to produce such a spaghetti pattern. The following is a good one, which encapsulates all logic in the loop body, concise, readable, maintainable, and efficient.

ASM

```asm
   mov ecx, LENGTHOF array
   xor esi, esi
L1:
   test array[esi], 1
   jz PASS
   inc array[esi]
PASS:
   add esi, TYPE DWORD
loop L1
```

16. Loop entrance and exit

Usually preferred is a loop with one entrance and one exit. But if necessary, two or more conditional exits are fine as shown in `Search2DAry` with found and not-found results.

The following is a bad pattern of two-entrance, where one gets into `START` via initialization and another directly goes to `MIDDLE`. Such a code is pretty hard to understand. Need to reorganize or refactor the loop logic.

ASM

```asm
   ; do something
   je MIDDLE

   ; loop initialization
```

```
START:
   ; do something

MIDDLE:
   ; do something
loop START
```

The following is a bad pattern of two-loop ends, where some logic gets out of the first loop end while the other exits at the second. Such a code is quite confusing. Try to reconsider with a label jumping to maintain one loop end.

ASM

```
   ; loop initialization
START2:
   ; do something
   je NEXT
   ; do something
loop START2
   jmp DONE

NEXT:
   ; do something
loop START2
DONE:
```

17. Don't change ECX in the loop body

The register `ECX` acts as a loop counter and its value is implicitly decremented when using the `LOOP` instruction. You can read `ECX` and make use of its value in iteration. As see in `Search2DAry` in the previous section, we compare the indirect operand `[ESI+ECX-1]` with `AL`. But never try to change the loop counter within the loop body that makes code hard to understand and hard to debug. A good practice is to think of the loop counter `ECX` as read-only.

ASM

```
   ; do initialization
   mov ecx, 10
L1:
   ; do something
   mov eax, ecx                    ; fine
   mov ebx, [esi +ecx *TYPE DWORD]   ; fine
   mov ecx, edx                    ; not good
   inc ecx                         ; not good
   ; do something
loop L1
```

18. When jump backward...

Besides the LOOP instruction, assembly language programming can heavily rely on conditional or unconditional jumps to create a loop when the count is not determined before the loop. Theoretically, for a backward jump, the workflow might be considered as a loop. Assume that jx and jy are desired jump or LOOP instructions. The following backward jy L2 nested in the jx L1 is probably thought of as an inner loop.

ASM

```
; loop initialization
L1:
   ; do something
 L2:
   ; do something
 jy L2
   ; do something
jx L1
```

To have selection logic of if-then-else, it's reasonable to use a foreword jump like this as branching in the jx L1 iteration:

ASM

```
; loop initialization
L1:
   ; do something
 jy TrueLogic
   ; do something for false
   jmp DONE
 TrueLogic:
   ; do something for true
DONE:
   ; do something
jx L1
```

19. Implementing C/C++ FOR loop and WHILE loop

The high level language usually provides three types of loop constructs. A FOR loop is often used when a known number of iterations available in coding that allows to initiate a loop counter as a check condition, and to change the count variable each iteration. A WHILE loop may be used when a loop counter is unknown, e.g, it might be determined by the user input as an ending flag at runtime. A DO-WHILE loop executes the loop body first and then check the condition. However, the usage is not so strictly clear and limited, since one loop can be simply replaced (implemented) by the other programmatically.

Let's see how the assembly code implements three loop structures in high level language. The previously mentioned LOOP instruction should behave like the FOR loop, because you have to initialize a known loop counter in ECX. The "LOOP target" statement takes two actions:

- decrement ECX
- if ECX != 0, jump to target

To calculate the sum of `n+(n-1)+...+2+1`, we can have

ASM

```
   mov ecx, n
   xor eax, eax
L1:
   add eax, ecx
   loop L1
   mov sum, eax
```

This is the same as the `FOR` loop:

C++

```
int sum=0;
for (int i=n; i>0; i++)
   sum += i;
```

How about the following logic - for a `WHILE` loop to add any non-zero input numbers until a zero entered:

C++

```
int sum=0;
cin >> n;
while (n !=0)
{
   sum += n;
   cin >> n;
}
```

There is no meaning to use `LOOP` here, because you could not set or ignore any value in `ECX`. Instead, using a conditional jump to manually construct such a loop is required:

ASM

```
   xor ebx, ebx
   call ReadInt      ; Read an integer in EAX
L1:
   or eax, eax
   jz L2
   add ebx, eax
   call ReadInt      ; Read an integer in EAX
   jmp L1
L2:
   mov sum, ebx
```

Here the Irvine32 library procedure `ReadInt` is used to read an integer from the console into `EAX`. Using `OR` instead of `CMP` is just for efficiency, as `OR` doesn't affect `EAX` while affecting the zero flag for `JZ`. Next, considering the similar logic with `DO-WHILE` loop:

C++

```
int sum=0;
cin >> n;
do
{
   sum += n;
   cin >> n;
}
while (n !=0)
```

Still with a conditional jump to have a loop here, the code looks more straight, as it does loop body first and then check:

ASM

```
   xor ebx, ebx
   call ReadInt      ; Read an integer in EAX
L1:
   add ebx, eax
   call ReadInt      ; Read an integer in EAX
   or eax, eax
   jnz L1
   mov sum, ebx
```

20. Making your loop more efficient with a jump

Based on above understanding, we can now turn to the loop optimization in assembly code. For detailed instruction mechanisms, please see the Intel® 64 and IA-32 Architectures Optimization Reference Manual. Here, I only use an example of calculating the sum of `n+(n-1)+...+2+1` to illustrate the performance comparison between iteration implementations of `LOOP` and conditional jumps. As code in the last section, I create our first procedure named as `Using_LOOP`:

ASM

```
;---------------------------------------------------------
Using_LOOP PROC
; Receives: ECX, as n, an integer to calculate 1+2+...+n
; Returns:  EAX, the sum of 1+2+...+n
;---------------------------------------------------------
   xor eax, eax
L1:
   add eax, ecx
   loop L1
   ret
Using_LOOP ENDP
```

To manually simulate the `LOOP` instruction, I simply decrement `ECX` and if not zero, go back to the loop label. So I name the second one `Using_DEC_JNZ`:

ASM

```
;-----------------------------------------------------------
Using_DEC_JNZ PROC
; Receives: ECX, as n, an integer to calculate 1+2+...+n
; Returns:  EAX, the sum of 1+2+...+n
;-----------------------------------------------------------
   xor eax, eax
L1:
   add eax, ecx
 ; Two instructions here equivalent to LOOP L1
   dec ecx
   JNZ L1
   ret
Using_DEC_JNZ ENDP
```

A similar alternative could be a third procedure by using `JECXZ` below, naming it as `Using_DEC_JECXZ_JMP`:

ASM

```
;-----------------------------------------------------------
Using_DEC_JECXZ_JMP PROC
; Receives: ECX, as n, an integer to calculate 1+2+...+n
; Returns:  EAX, the sum of 1+2+...+n
;-----------------------------------------------------------
   xor eax, eax
L1:
   add eax, ecx
 ; Three instructions here quivalent to LOOP L1
   dec ecx
   JECXZ L2
   jmp L1
L2:
   ret
Using_DEC_JECXZ_JMP ENDP
```

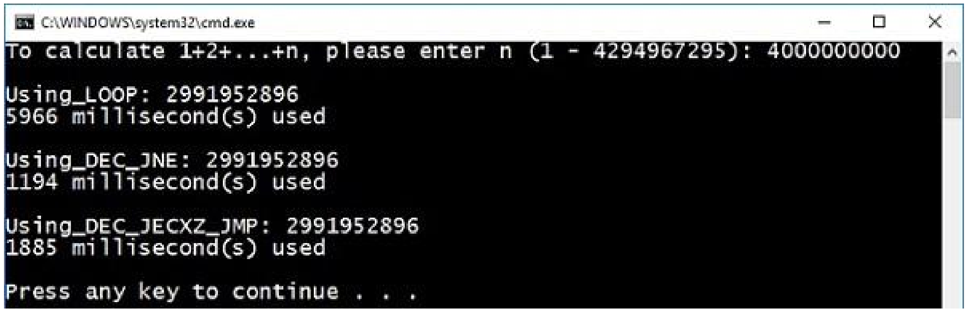
Now let's test three procedures by accepting a number `n` from the user input to save the loop counter, and then calling each procedure with a macro `mCallSumProc` (Here `Clrscr`, `ReadDec`, `Crlf`, and `mWrite` are from Irvine32 that will be mentioned shortly):

ASM

```
main PROC
   call Clrscr
   mWrite "To calculate 1+2+...+n, please enter n (1 - 4294967295): "
   call ReadDec                   ; read n from user into EAX
   mov  ecx, eax                  ; save n to the loop counter ECX
   call Crlf

   mCallSumProc Using_LOOP
   mCallSumProc Using_DEC_JNE
```

```
   mCallSumProc Using_DEC_JECXZ_JMP
   exit
main ENDP
```

To test, enter a large number like 4 billion. Although the sum is far beyond the 32-bit maximum `0FFFFFFFFh`, with only remainder left in `EAX` as (1+2+...+n) `MOD` 4294967295, it doesn't matter to our benchmark test. The following is the result from my Intel Core i7, 64-bit BootCamp:



Probably, the result will be slightly different on different systems. The test executable is available for try at LoopTest.EXE. Basically, using a conditional jump to construct your loop is more efficient than using the `LOOP` instruction directly. You can read "Intel® 64 and IA-32 Architectures Optimization Reference Manual" to find why. Also I would like to thank Mr. Daniel Pfeffer for his nice comments about optimizations that you can read in Comments and Discussions at the end.

Finally, I present above unmentioned macro as below. Again, it contains some Irvine32 library procedure calls. The source code in this section can be downloaded at Loop Test ASM Project. To understand further, please see the links in References

ASM

```
;-------------------------------------------------------
mCallSumProc MACRO SumProc:REQ
; Receives: SumProc, a summation procedure
;           ECX as n, to calculate 1+2+...+n
;-------------------------------------------------------
   push ecx
   call GetMseconds   ; get start time
   mov  esi,eax
   call SumProc
   mWrite "&SumProc: "
   call WriteDec
   call crlf

   call GetMseconds   ; get start time
   sub  eax,esi
   call WriteDec      ; display elapsed time
   mWrite <' millisecond(s) used', 0Dh,0Ah, 0Dh,0Ah >
   pop  ecx
ENDM
```

About procedure

Similar to functions in C/C++, we talk about some basics in assembly language's procedure.

21. Making a clear calling interface

When design a procedure, we hope to make it as reusable as possible. Make it perform only one task without others like I/O. The procedure's caller should take the responsibility to do input and putout. The caller should communicate with the procedure only by arguments and parameters. The procedure should only use parameters in its logic without referring outside definitions, without any:

- Global variable and array
- Global symbolic constant

Because implementing with such a definition makes your procedure un-reusable.

Recalling previous five `FibonacciByXXX` procedures, we use register `ECX` as both argument and parameter with the return value in `EAX` to make a clear calling interface:

ASM

```
;------------------------------------------------------------
FibonacciByXXX
; Receives: ECX as input n
; Returns: EAX, nth Fibonacci number
;------------------------------------------------------------
```

Now the caller can do like

ASM

```
; Read user's input n and save in ECX
call FibonacciByXXX
; Output or process the nth Fibonacci number in EAX
```

To illustrate as a second example, let's take a look again at calling `Search2DAry` in the previous section. The register arguments `ESI` and `EAX` are prepared so that the implementation of `Search2DAry` doesn't directly refer to the global array, `ary2D`.

ASM

```
... ...
NUM_COL = 5
NUM_ROW = 2

.code
main PROC
```

```
   mov esi, OFFSET ary2D
   mov eax, 31h
   call Search2DAry
; See eax for search result
   exit
main ENDP

;------------------------------------------------------------
Search2DAry PROC
; Receives: EAX, a byte value to search a 2-dimensional array
;           ESI, an address to the 2-dimensional array
; Returns: EAX, 1 if found, 0 if not found
;------------------------------------------------------------
   mov  ecx,NUM_ROW        ; outer loop count
... ...
   mov  ecx,NUM_COL        ; inner loop counter
... ...
```

Unfortunately, the weakness is its implementation still using two global constants `NUM_ROW` and `NUM_COL` that makes it not being called elsewhere. To improve, supplying other two register arguments would be an obvious way, or see the next section.

22. INVOKE vs. CALL

Besides the `CALL` instruction from Intel, MASM provides the 32-bit `INVOKE` directive to make a procedure call easier. For the `CALL` instruction, you only can use registers as argument/parameter pair in calling interface as shown above. The problem is that the number of registers is limited. All registers are global and you probably have to save registers before calling and restore after calling. The `INVOKE` directive gives the form of a procedure with a parameter-list, as you experienced in high level languages.

When consider `Search2DAry` with a parameter-list without referring the global constants `NUM_ROW` and `NUM_COL`, we can have its prototype like this

ASM

```
;---------------------------------------------------------------------
Search2DAry PROTO, pAry2D: PTR BYTE, val: BYTE, nRow: WORD, nCol: WORD
; Receives: pAry2D, an address to the 2-dimensional array
;           val, a byte value to search a 2-dimensional array
;           nRow, the number of rows
;           nCol, the number of columns
; Returns: EAX, 1 if found, 0 if not found
;---------------------------------------------------------------------
```

Again, as an exercise, you can try to implement this for a fix. Now you just do

ASM

```
INVOKE Search2DAry, ary2D, 31h, NUM_ROW, NUM_COL
; See eax for search result
```

Likewise, to construct a parameter-list procedure, you still need to follow the rule without referring global variables and constants. Besides, also attention to:

- The entire calling interface should only go through the parameter list without referring any register values set outside the procedure.

23. Call-by-Value vs. Call-by-Reference

Also be aware of that a parameter-list should not be too long. If so, use an object parameter instead. Suppose that you fully understood the function concept, call-by-value and call-by-reference in high level languages. By learning the stack frame in assembly language, you understand more about the low-level function calling mechanism. Usually for an object argument, we prefer passing a reference, an object address, rather than the whole object copied on the stack memory.

To demonstrate this, let's create a procedure to write month, day, and year from an object of the Win32 SYSTEMTIME structure.

The following is the version of call-by-value, where we use the dot operator to retrieve individual `WORD` field members from the `DateTime` object and extend their 16-bit values to 32-bit `EAX`:

ASM

```
;---------------------------------------------------------
WriteDateByVal PROC, DateTime:SYSTEMTIME
; Receives: DateTime, an object of SYSTEMTIME
;---------------------------------------------------------
   movzx eax, DateTime.wMonth
   ; output eax as month
   ; output a separator like '/'
   movzx eax, DateTime.wDay
   ; output eax as day
   ; output a separator like '/'
   movzx eax, DateTime.wYear
   ; output eax as year
   ; make a newline
   ret
WriteDateByVal ENDP
```

The version of call-by-reference is not so straight with an object address received. Not like the arrow ->, pointer operator in C/C++, we have to save the pointer (address) value in a 32-bit register like `ESI`. By using `ESI` as an indirect operand, we must cast its memory back to the `SYSTEMTIME` type. Then we can get the object members with the dot:

ASM

```
;---------------------------------------------------------
WriteDateByRef PROC, datetimePtr: PTR SYSTEMTIME
; Receives: DateTime, an address of SYSTEMTIME object
;---------------------------------------------------------
   mov esi, datetimePtr
```

```
    movzx eax, (SYSTEMTIME PTR [esi]).wMonth
    ; output eax as month
    ; output a separator like '/'
    movzx eax, (SYSTEMTIME PTR [esi]).wDay
    ; output eax as day
    ; output a separator like '/'
    movzx eax, (SYSTEMTIME PTR [esi]).wYear
    ; output eax as year
    ; make a newline
    ret
WriteDateByRef ENDP
```

You can watch the stack frame of argument passed for two versions at runtime. For `WriteDateByVal`, eight `WORD` members are copied on the stack and consume sixteen bytes, while for `WriteDateByRef`, only need four bytes as a 32-bit address. It will make a big difference for a big structure object, though.

24. Avoid multiple RET

To construct a procedure, it's ideal to make all your logics within the procedure body. Preferred is a procedure with one entrance and one exit. Since in assembly language programming, a procedure name is directly represented by a memory address, as well as any labels. Thus directly jumping to a label or a procedure without using `CALL` or `INVOKE` would be possible. Since such an abnormal entry would be quite rare, I am not to going to mention here.

Although multiple returns are sometimes used in other language examples, I don't encourage such a pattern in assembly code. Multiple `RET` instructions could make your logic not easy to understand and debug. The following code on the left is such an example in branching. Instead, on the right, we have a label `QUIT` at the end and jump there making a single exit, where probably do common chaos to avoid repeated code.

ASM

```
MultiRetEx PROC
   ; do something
   jx NEXTx
   ; do something
   ret

NEXTx:
   ; do something
   jy NEXTy
   ; do something
   ret

NEXTy:
   ; do something
   ret
MultiRetEx ENDP
```

ASM

```
SingleRetEx PROC
   ; do something
   jx NEXTx
   ; do something
   jmp QUIT
NEXTx:
   ; do something
   jy NEXTy
   ; do something
   jmp QUIT
NEXTy:
   ; do something
QUIT:
   ; do common things
   ret
SingleRetEx ENDP
```

Object data members

Similar to above `SYSTEMTIME` structure, we can also create our own type or a nested:

```
Rectangle STRUCT
   UpperLeft COORD <>
   LowerRight COORD <>
Rectangle ENDS

.data
rect Rectangle { {10,20}, {30,50} }
```

The `Rectangle` type contains two COORD members, `UpperLeft` and `LowerRight`. The Win32 `COORD` contains two `WORD` (`SHORT`), `X` and `Y`. Obviously, we can access the object `rect`'s data members with the dot operator from either direct or indirect operand like this

ASM

```
; directly access
mov rect.UpperLeft.X, 11

; cast indirect operand to access
mov esi,OFFSET rect
mov (Rectangle PTR [esi]).UpperLeft.Y, 22

; use the OFFSET operator for embedded members
mov esi,OFFSET rect.LowerRight
mov (COORD PTR [esi]).X, 33
mov esi,OFFSET rect.LowerRight.Y
mov WORD PTR [esi], 55
```

By using the `OFFSET` operator, we access different data member values with different type casts. Recall that any operator is processed in assembling at static time. What if we want to retrieve a data member's address (not value) at runtime?

25. Indirect operand and LEA

For an indirect operand pointing to an object, you can't use the `OFFSET` operator to get the member's address, because `OFFSET` only can take an address of a variable defined in the data segment.

There could be a scenario that we have to pass an object reference argument to a procedure like `WriteDateByRef` in the previous section, but want to retrieve its member's address (not value). Still use the above `rect` object for an example. The following second use of `OFFSET` is not valid in assembling:

ASM

```
mov esi,OFFSET rect
mov edi, OFFSET (Rectangle PTR [esi]).LowerRight
```

Let's ask for help from the `LEA` instruction that you have seen in `FibonacciByRegLEA` in the previous section. The `LEA` instruction calculates and loads the effective address of a memory operand. Similar to the `OFFSET` operator, except that only `LEA` can obtain an address calculated at runtime:

ASM

```
mov esi,OFFSET rect
lea edi, (Rectangle PTR [esi]).LowerRight
mov ebx, OFFSET rect.LowerRight

lea edi, (Rectangle PTR [esi]).UpperLeft.Y
mov ebx, OFFSET rect.UpperLeft.Y

mov esi,OFFSET rect.UpperLeft
lea edi, (COORD PTR [esi]).Y
```

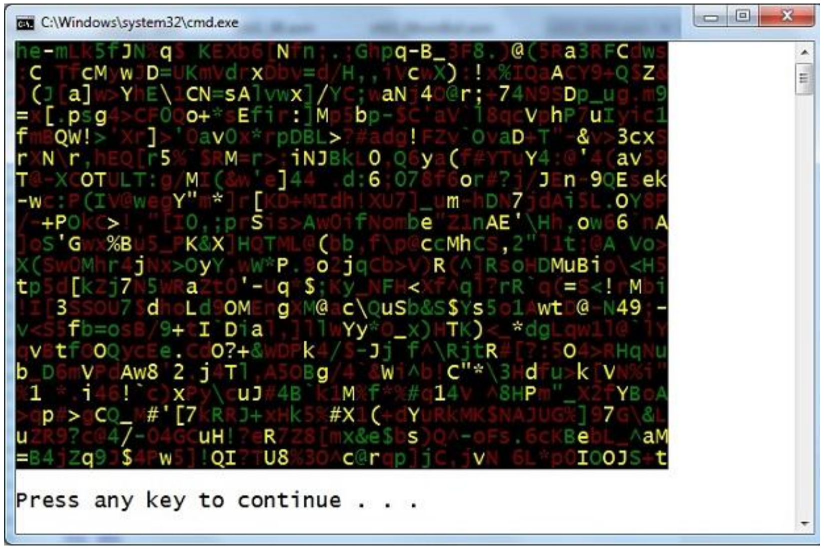
I purposely have `EBX` here to get an address statically and you can verify the same address in `EDI` that is loaded dynamically from the indirect operand `ESI` at runtime.

About system I/O

From Computer Memory Basics, we know that I/O operations from the operating system are quite slow. Input and output are usually in the measurement of milliseconds, compared with register and memory in nanoseconds or microseconds. To be more efficient, trying to reduce system API calls is a nice consideration. Here I mean Win32 API call. For details about the Win32 functions mentioned in the following, please refer to MSDN to understand.

26. Reducing system I/O API calls

An example is to output `20` lines of `50` random characters with random colors as below:



We definitely can generate one character to output a time, by using SetConsoleTextAttribute and WriteConsole. Simply set its color by

ASM

```
INVOKE SetConsoleTextAttribute, consoleOutHandle, wAttributes
```

Then write that character by

ASM

```
INVOKE WriteConsole,
   consoleOutHandle,     ; console output handle
   OFFSET buffer,        ; points to string
   1,                    ; string length
   OFFSET bytesWritten, ; returns number of bytes written
   0
```

When write `50` characters, make a new line. So we can create a nested iteration, the outer loop for `20` rows and the inner loop for `50` columns. As `50` by `20`, we call these two console output functions 1000 times.

However, another pair of API functions can be more efficient, by writing `50` characters in a row and setting their colors once a time. They are WriteConsoleOutputAttribute and WriteConsoleOutputCharacter. To make use of them, let's create two procedures:

ASM

```
;------------------------------------------------------------------------
ChooseColor PROC
```

```
; Selects a color with 50% probability of red, 25% green and 25% yellow
; Receives: nothing
; Returns:  AX = randomly selected color

;----------------------------------------------------------------------
ChooseCharacter PROC
; Randomly selects an ASCII character, from ASCII code 20h to 07Ah
; Receives: nothing
; Returns:  AL = randomly selected character
```

We call them in a loop to prepare a `WORD` array `bufColor` and a BYTE array `bufChar` for all `50` characters selected. Now we can write the `50` random characters per line with two calls here:

ASM

```
INVOKE WriteConsoleOutputAttribute,
      outHandle,
      ADDR bufColor,
      MAXCOL,
      xyPos,
      ADDR cellsWritten

INVOKE WriteConsoleOutputCharacter,
      outHandle,
      ADDR bufChar,
      MAXCOL,
      xyPos,
      ADDR cellsWritten
```

Besides `bufColor` and `bufChar`, we define `MAXCOL = 50` and the `COORD` type `xyPos` so that `xyPos.y` is incremented each row in a single loop of `20` rows. Totally we only call these two APIs 20 times.

About PTR operator

MASM provides the operator `PTR` that is similar to the pointer `*` used in C/C++. The following is the `PTR` specification:

- **type PTR** expression
  Forces the expression to be treated as having the specified type.
- [[distance]] **PTR type**
  Specifies a pointer to type.

This means that two usages are available, such as `BYTE PTR` or `PTR BYTE`. Let's discuss how to use them.

27. Defining a pointer, cast and dereference

The following C/C++ code demonstrates which type of Endian is used in your system, little endian or big endian? As an integer type takes four bytes, it makes a pointer type cast from the array name `fourBytes`, a `char` address, to an `unsigned int` address. Then it displays the integer result by dereferencing the `unsigned int` pointer.

C++

```cpp
int main()
{
   unsigned char fourBytes[] = { 0x12, 0x34, 0x56, 0x78 };
   // Cast the memory pointed by the array name fourBytes, to unsigned int address
   unsigned int *ptr = (unsigned int *)fourBytes;
   printf("1. Directly Cast: n is %Xh\n", *ptr);
   return 0;
}
```

As expected in x86 Intel based system, this verifies the little endian by showing `78563412` in hexadecimal. We can do the same thing in assembly language with `DWORD PTR`, which is just similar to an address casting to 4-byte `DWORD`, the `unsigned int` type.

ASM

```asm
.data
fourBytes BYTE 12h,34h,56h,78h

.code
mov eax, DWORD PTR fourBytes        ; EAX = 78563412h
```

There is no explicit dereference here, since `DWORD PTR` combines four bytes into a `DWORD` memory and lets `MOV` retrieve it as a direct operand to `EAX`. This could be considered equivalent to the (`unsigned int *`) cast.

Now let's do another way by using `PTR DWORD`. Again, with the same logic above, this time we define a `DWORD` pointer type first with `TYPEDEF`:

ASM

```asm
DWORD_POINTER TYPEDEF PTR DWORD
```

This could be considered equivalent to defining the pointer type as `unsigned int *`. Then in the following data segment, the address variable `dwPtr` takes over the `fourBytes` memory. Finally in code, `EBX` holds this address as an indirect operand and makes an explicit dereference here to get its `DWORD` value to `EAX`.

ASM

```asm
.data
fourBytes BYTE 12h,34h,56h,78h
dwPtr DWORD_POINTER fourBytes

.code
```

```
mov ebx, dwPtr       ; Get DWORD address
mov eax, [ebx]       ; Dereference, EAX = 78563412h
```

To summarize, `PTR DWORD` indicates a `DWORD` address type to define(declare) a variable like a pointer type. While `DWORD PTR` indicates the memory pointed by a `DWORD` address like a type cast.

28. Using PTR in a procedure

To define a procedure with a parameter list, you might want to use `PTR` in both ways. The following is such an example to increment each element in a `DWORD` array:

ASM

```
;---------------------------------------------------------
IncrementArray PROC, pAry:PTR DWORD, count:DWORD
; Receives: pAry  - pointer to a DWORD array
;           count - the array count
; Returns:  pAry, every vlues in pAry incremented
;---------------------------------------------------------
   mov edi,pAry
   mov ecx,count

 L1:
   inc DWORD PTR [edi]
   add edi, TYPE DWORD
 loop L1
   ret
IncrementArray ENDP
```

As the first parameter `pAry` is a `DWORD` address, so `PTR DWORD` is used as a parameter type. In the procedure, when incrementing a value pointed by the indirect operand `EDI`, you must tell the system what the type(size) of that memory is by using `DWORD PTR`.

Another example is the earlier mentioned `WriteDateByRef`, where `SYSTEMTIME` is a Windows defined structure type.

ASM

```
;--------------------------------------------------------
WriteDateByRef PROC, datetimePtr: PTR SYSTEMTIME
; Receives: DateTime, an address of SYSTEMTIME object
;--------------------------------------------------------
   mov esi, datetimePtr
   movzx eax, (SYSTEMTIME PTR [esi]).wMonth
  ... ...
   ret
WriteDateByRef ENDP
```

Likewise, we use `PTR SYSTEMTIME` as the parameter type to define `datetimePtr`. When `ESI` receives an address from `datetimePtr`, it has no knowledge about the memory type just like a `void` pointer in C/C++. We have to cast it as a `SYSTEMTIME` memory, so as to retrieve its data members.

Signed and Unsigned

In assembly language programming, you can define an integer variable as either signed as `SBYTE`, `SWORD`, and `SDWORD`, or unsigned as `BYTE`, `WORD`, and `DWORD`. The data ranges, for example of 8-bit, are

- `BYTE`: 0 to 255 (`00h` to `FFh`), totally 256 numbers
- `SBYTE`: half negatives, -128 to -1 (`80h` to `FFh`), half positives, 0 to 127 (`00h` to `7Fh`)

Based on the hardware point of view, all CPU instructions operate exactly the same on signed and unsigned integers, because the CPU cannot distinguish between signed and unsigned. For example, when define

ASM

```
.data
   bVal   BYTE   255
   sbVal  SBYTR  -1
```

Both of them have the 8-bit binary `FFh` saved in memory or moved to a register. You, as a programmer, are solely responsible for using the correct data type with an instruction and are able to explain a results from the flags affected:

- The carry flag `CF` for unsigned integers
- The overflow flag `OF` for signed integers

The following are usually several tricks or pitfalls.

29. Comparison with conditional jumps

Let's check the following code to see which label it jumps:

ASM

```
mov   eax, -1
cmp   eax, 1
ja    L1
jmp   L2
```

As we know, `CMP` follows the same logic as `SUB` while non-destructive to the destination operand. Using `JA` means considering unsigned comparison, where the destination `EAX` is `FFh`, i.e. `255`, while the source is `1`. Certainly `255` is bigger than `1`, so that makes it jump to `L1`. Thus, any unsigned comparisons such as `JA`, `JB`, `JAE`, `JNA`, etc. can be remembered as A(Above) or B(Below). An unsigned comparison is determined by `CF` and the zero flag `ZF` as shown in the following examples:

CMP if	Destination	Source	ZF(ZR)	CF(CY)
Destination<Source	1	2	0	1
Destination>Source	2	1	0	0
Destination=Source	1	1	1	0

Now let's take a look at signed comparison with the following code to see where it jumps:

ASM

```asm
mov   eax, -1
cmp   eax, 1
jg    L1
jmp   L2
```

Only difference is `JG` here instead of `JA`. Using `JG` means considering signed comparison, where the destination `EAX` is `FFh`, i.e. `-1`, while the source is `1`. Certainly `-1` is smaller than `1`, so that makes `JMP` to `L2`. Likewise, any signed comparisons such as `JG`, `JL`, `JGE`, `JNG`, etc. can be thought of as G(Greater) or L(Less). A signed comparison is determined by `OF` and the sign flag `SF` as shown in the following examples:

CMP if	Destination	Source	SF(PL)	OF(OV)
Destination<Source: (***SF !=***	-2	127	0	1
OF)	-2	1	1	0
Destination>Source: (***SF ==***	127	1	0	0
OF)	127	-1	1	1
Destination = Source	1	1	***ZF=1***	

30. When CBW, CWD, or CDQ mistakenly meets DIV...

As we know, the `DIV` instruction is for unsigned to perform 8-bit, 16-bit, or 32-bit integer division with the dividend `AX`, `DX:AX`, or `EDX:EAX` respectively. As for unsigned, you have to clear the upper half by zeroing `AH`, `DX`, or `EDX` before using `DIV`. But when perform signed division with `IDIV`, the sign extension `CBW`, `CWD`, and `CDQ` are provided to extend the upper half before using `IDIV`.

For a positive integer, if its highest bit (sign bit) is zero, there is no difference to manually clear the upper part of a dividend or mistakenly use a sign extension as shown in the following example:

ASM

```asm
mov eax,1002h
cdq
mov ebx,10h
div ebx  ; Quotient EAX = 00000100h, Remainder EDX = 2
```

This is fine because `1000h` is a small positive and `CDQ` makes `EDX` zero, the same as directly clearing `EDX`. So if your value is positive and its highest bit is zero, using `CDQ` and

ASM

```
XOR EDX, EDX
```

are exactly the same.

However, it doesn't mean that you can always use `CDQ`/`CWD`/`CBW` with `DIV` when perform a positive division. For an example of 8-bit, `129/2`, expecting quotient `64` and remainder `1`. But, if you make this

ASM

```
mov  al, 129
cbw             ; Extend AL to AH as negative AX = FF81h
mov  bl,2
div  bl         ; Unsigned DIV, Quotient should be 7FC0 over size of AL
```

Try above in debug to see how integer division overflow happens as a result. If really want to make it correct as unsigned `DIV`, you must:

ASM

```
mov  al, 129
XOR  ah, ah     ; extend AL to AH as positive
mov  bl,2
div  bl         ; Quotient AL = 40h,  Remainder AH = 1
```

On the other side, if really want to use `CBW`, it means that you perform a signed division. Then you must use `IDIV`:

ASM

```
mov  al, 129    ; 81h (-127d)
cbw             ; Extend AL to AH as negative AX = FF81h
mov  bl,2
idiv bl         ; Quotient AL = C1h (-63d), Remainder AH = FFh (-1)
```

As seen here, `81h` in signed byte is decimal `-127` so that signed `IDIV` gives the correct quotient and remainder as above

31. Why 255-1 and 255+(-1) affect CF differently?

To talk about the carry flag `CF`, let's take the following two arithmetic calculations:

ASM

```
mov al, 255
sub al, 1      ; AL = FE  CF = 0

mov bl, 255
add bl, -1     ; BL = FE  CF = 1
```

From a human being's point of view, they do exactly the same operation, `255` minus `1` with the result 254 (`FEh`). Likewise, based on the hardware point, for either calculation, the CPU does the same operation by representing `-1` as a two's complement `FFh` and then add it to `255`. Now `255` is `FFh` and the binary format of `-1` is also `FFh`. This is how it has been calculated:

```
   1111 1111
+  1111 1111
-------------
   1111 1110
```

Remember? A CPU operates exactly the same on signed and unsigned because it cannot distinguish them. A programmer should be able to explain the behavior by the flag affected. Since we talk about the `CF`, it means we consider two calculations as unsigned. The key information is that `-1` is `FFh` and then `255` in decimal. So the logic interpretation of `CF` is

- For `sub al, 1`, it means `255` minus `1` to result in `254`, without need of a borrow, so `CF` = `0`
- For `add bl, -1`, it seems that `255` plus `255` is resulted in `510`, but with a carry `1,0000,0000b` (`256`) out, `254` is a remainder left in byte, so `CF` = `1`

From hardware implementation, `CF` depends on which instruction used, `ADD` or `SUB`. Here **MSB** (Most Significant Bit) is the highest bit.

- For `ADD` instruction, `add bl, -1`, directly use the carry out of the MSB, so `CF` = `1`
- For `SUB` instruction, `sub al, 1`, must **INVERT** the carry out of the MSB, so `CF` = `0`

32. How to determine OF?

Now let's see the overflow flag `OF`, still with above two arithmetic calculations as this:

ASM

```
mov al, 255
sub al, 1      ; AL = FE  OF = 0

mov bl, 255
add bl, -1     ; BL = FE  OF = 0
```

Both of them are not overflow, so `OF` = `0`. We can have two ways to determine `OF`, the logic rule and hardware implementation.

Logic viewpoint: The overflow flag is only set, `OF` = `1`, when

- Two positive operands are added and their sum is negative
- Two negative operands are added and their sum is positive

For signed, `255` is `-1` (`FFh`). The flag `OF` doesn't care about `ADD` or `SUB`. Our two examples just do `-1` plus `-1` with the result `-2`. Thus, two negatives are added with the sum still negative, so `OF` = `0`.

Hardware implementation: For non-zero operands,

- `OF` = (carry out of the MSB) `XOR` (carry into the MSB)

As seen our calculation again:

```
   1111 1111
+  1111 1111
-------------
   1111 1110
```

The carry out of the MSB is `1` and the carry into the MSB is also `1`. Then `OF` = (`1 XOR 1`) = `0`

To practice more, the following table enumerates different test cases for your understanding:

Overflow Flag	*0 XOR 0 = 0*	*0 XOR 1 = 1*	*1 XOR 0 = 1*	*1 XOR 1 = 0*
Both values either positive or negative	$1_d + 1_d = 2_d$ 0000 0001 + 0000 0001 0000 0010	$64_d + 64_d = -128_d$ 0100 0000 + 0100 0000 1000 0000	$-127_d + -128_d = 1_d$ 1000 0001 + 1000 0000 0000 0001	$-1_d + -1_d = -2_d$ 1111 1111 + 1111 1111 1111 1110
One value positive the other negative	$-128_d + 127_d = -1_d$ 1000 0000 + 0111 1111 1111 1111	N/A	N/A	$-1_d + 127_d = 126_d$ 1111 1111 + 0111 1111 0111 1110

Ambiguous "LOCAL" directive

As mentioned previously, the `PTR` operator has two usages such as `DWORD PTR` and `PTR DWORD`. But MASM provides another confused directive `LOCAL`, that is ambiguous depending on the context, where to use with exactly the same reserved word. The following is the specification from MSDN:

> ***LOCAL localname [[, localname]]...***
> ***LOCAL label [[[count]]] [[:type]] [[, label [[[count]]] [[type]]]]...***

- *In the first directive, within a macro, `LOCAL` defines labels that are unique to each instance of the macro.*
- *In the second directive, within a procedure definition (PROC), `LOCAL` creates stack-based variables that exist for the duration of the procedure. The label may be a simple variable or an array containing count elements.*

This specification is not clear enough to understand. In this section, I'll expose the essential difference in between and show two example using the `LOCAL` directive, one in a procedure and the other in a macro. As for your familiarity, both examples calculate the nth Fibonacci number as early `FibonacciByMemory`. The main point delivered here is:

- The variables **declared** by `LOCAL` in a macro are NOT local to the macro. They are system generated global variables on the data segment to resolve redefinition.
- The variables **created** by `LOCAL` in a procedure are really local variables allocated on the stack frame with the lifecycle only during the procedure.

For the basic concepts and implementations of data segment and stack frame, please take a look at some textbook or MASM manual that could be worthy of several chapters without being talked here.

33. When LOCAL used in a procedure

The following is a procedure with a parameter `n` to calculate nth Fibonacci number returned in `EAX`. I let the loop counter `ECX` take over the parameter `n`. Please compare it with `FibonacciByMemory`. The logic is the same with only difference of using the local variables `pre` and `cur` here, instead of global variables `previous` and `current` in `FibonacciByMemory`.

ASM

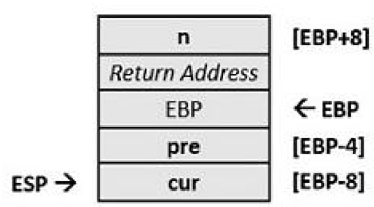
```
;------------------------------------------------------------
FibonacciByLocalVariable PROC USES ecx edx, n:DWORD
; Receives: Input n
; Returns: EAX, nth Fibonacci number
;------------------------------------------------------------
LOCAL pre, cur :DWORD

   mov   ecx,n
   mov   eax,1
   mov   pre,0
   mov   cur,0
L1:
   add eax, pre      ; eax = current + previous
   mov edx, cur
   mov pre, edx
   mov cur, eax
 loop   L1

   ret
FibonacciByLocalVariable ENDP
```

The following is the code generated from the VS Disassembly window at runtime. As you can see, each line of assembly source is translated into machine code with the parameter `n` and two local variables created on the stack frame, referenced by `EBP`:

ASM

```
   231: ;------------------------------------------------------------
   232: FibonacciByLocalVariable PROC USES ecx edx, n:DWORD
011713F4 55                   push        ebp
011713F5 8B EC                mov         ebp,esp
011713F7 83 C4 F8             add         esp,0FFFFFFF8h
011713FA 51                   push        ecx
011713FB 52                   push        edx
   233: ; Receives: Input n
   234: ; Returns: EAX, nth Fibonacci number
   235: ;------------------------------------------------------------
   236: LOCAL pre, cur :DWORD
   237:
   238:    mov   ecx,n
011713FC 8B 4D 08             mov         ecx,dword ptr [ebp+8]
   239:    mov   eax,1
011713FF B8 01 00 00 00       mov         eax,1
   240:    mov   pre,0
01171404 C7 45 FC 00 00 00 00 mov         dword ptr [ebp-4],0
   241:    mov   cur,0
0117140B C7 45 F8 00 00 00 00 mov         dword ptr [ebp-8],0
   242: L1:
   243:    add eax,pre      ; eax = current + previous
01171412 03 45 FC             add         eax,dword ptr [ebp-4]
   244:    mov EDX, cur
01171415 8B 55 F8             mov         edx,dword ptr [ebp-8]
   245:    mov pre, EDX
01171418 89 55 FC             mov         dword ptr [ebp-4],edx
   246:    mov cur, eax
0117141B 89 45 F8             mov         dword ptr [ebp-8],eax
   247:    loop   L1
0117141E E2 F2                loop        01171412
   248:
   249:    ret
01171420 5A                   pop         edx
01171421 59                   pop         ecx
01171422 C9                   leave
01171423 C2 04 00             ret         4
   250: FibonacciByLocalVariable ENDP
```

When `FibonacciByLocalVariable` running, the stack frame can be seen as below:



Obviously, the parameter `n` is at `EBP+8`. This

ASM

```
add esp, 0FFFFFFF8h
```

just means

ASM

```
sub esp, 08h
```

moving the stack pointer `ESP` down eight bytes for two `DWORD` creation of `pre` and `cur`. Finally the `LEAVE` instruction implicitly does

ASM

```
mov esp, ebp
pop ebp
```

that moves `EBP` back to `ESP` releasing the local variables `pre` and `cur`. And this releases `n`, at `EBP+8`, for STD calling convention:

ASM

```
ret 4
```

34. When LOCAL used in a macro

To have a macro implementation, I almost copy the same code from `FibonacciByLocalVariable`. Since no `USES` for a macro, I manually use `PUSH`/`POP` for `ECX` and `EDX`. Also without a stack frame, I have to create **global** variables `mPre` and `mCur` on the data segment. The `mFibonacciByMacro` can be like this:

ASM Shrink ▲

```
;------------------------------------------------------------
mFibonacciByMacro MACRO n
; Receives: Input n
; Returns: EAX, nth Fibonacci number
;------------------------------------------------------------
LOCAL mPre, mCur, mL
.data
   mPre DWORD ?
   mCur DWORD ?

.code
   push ecx
   push edx

   mov   ecx,n
   mov   eax,1
```

```
   mov   mPre,0
   mov   mCur,0
mL:
   add  eax, mPre      ; eax = current + previous
   mov  edx, mCur
   mov  mPre, edx
   mov  mCur, eax
   loop   mL

   pop edx
   pop ecx
ENDM
```

If you just want to call `mFibonacciByMacro` once, for example

ASM

```
mFibonacciByMacro 12
```

You don't need `LOCAL` here. Let's simply comment it out:

ASM

```
; LOCAL mPre, mCur, mL
```

`mFibonacciByMacro` accepts the argument `12` and replace `n` with `12`. This works fine with the following Listing MASM generated:

ASM

```
              mFibonacciByMacro 12
0000018C           1   .data
0000018C 00000000        1      mPre DWORD ?
00000190 00000000        1      mCur DWORD ?
00000000           1   .code
00000000  51           1      push ecx
00000001  52           1      push edx
00000002  B9 0000000C       1      mov   ecx,12
00000007  B8 00000001       1      mov   eax,1
0000000C  C7 05 0000018C R  1      mov   mPre,0
     00000000
00000016  C7 05 00000190 R  1      mov   mCur,0
     00000000
00000020           1   mL:
00000020  03 05 0000018C R  1      add  eax,mPre      ; eax = current + previous
00000026  8B 15 00000190 R  1      mov edx, mCur
0000002C  89 15 0000018C R  1      mov mPre, edx
00000032  A3 00000190 R     1      mov mCur, eax
00000037  E2 E7        1      loop   mL
00000039  5A           1      pop edx
0000003A  59           1      pop ecx
```

Nothing changed from the original code with just a substitution of 12. The variables mPre and mCur are visible explicitly. Now let's call it twice, like

ASM

```
mFibonacciByMacro 12
mFibonacciByMacro 13
```

This is still fine for the first mFibonacciByMacro 12 but secondly, causes three redefinitions in preprocessing mFibonacciByMacro 13. Not only are data labels, i.e., variables mPre and mCur, but also complained is the code label mL. This is because in assembly code, each label is actually a memory address and the second label of any mPre, mCur, or mL should take another memory, rather than defining an already created one:

ASM

Shrink ▲

```
               mFibonacciByMacro 12
 0000018C          1   .data
 0000018C 00000000       1      mPre DWORD ?
 00000190 00000000       1      mCur DWORD ?
 00000000          1   .code
 00000000  51          1      push ecx
 00000001  52          1      push edx
 00000002  B9 0000000C      1      mov   ecx,12
 00000007  B8 00000001      1      mov   eax,1
 0000000C  C7 05 0000018C R  1      mov   mPre,0
      00000000
 00000016  C7 05 00000190 R  1      mov   mCur,0
      00000000
 00000020          1   mL:
 00000020  03 05 0000018C R  1      add  eax,mPre      ; eax = current + previous
 00000026  8B 15 00000190 R  1      mov edx, mCur
 0000002C  89 15 0000018C R  1      mov mPre, edx
 00000032  A3 00000190 R     1      mov mCur, eax
 00000037  E2 E7         1      loop   mL
 00000039  5A            1      pop edx
 0000003A  59            1      pop ecx

               mFibonacciByMacro 13
 00000194          1   .data
             1      mPre DWORD ?
FibTest.32.asm(83) : error A2005:symbol redefinition : mPre
 mFibonacciByMacro(6): Macro Called From
  FibTest.32.asm(83): Main Line Code
             1      mCur DWORD ?
FibTest.32.asm(83) : error A2005:symbol redefinition : mCur
 mFibonacciByMacro(7): Macro Called From
  FibTest.32.asm(83): Main Line Code
 0000003B          1   .code
 0000003B  51          1      push ecx
 0000003C  52          1      push edx
```

```
 0000003D  B9 0000000D       1      mov   ecx,13
 00000042  B8 00000001       1      mov   eax,1
 00000047  C7 05 0000018C R  1      mov   mPre,0
      00000000
 00000051  C7 05 00000190 R  1      mov   mCur,0
      00000000
              1   mL:
FibTest.32.asm(83) : error A2005:symbol redefinition : mL
 mFibonacciByMacro(17): Macro Called From
  FibTest.32.asm(83): Main Line Code
 0000005B  03 05 0000018C R  1      add  eax,mPre      ; eax = current + previous
 00000061  8B 15 00000190 R  1      mov edx, mCur
 00000067  89 15 0000018C R  1      mov mPre, edx
 0000006D  A3 00000190 R     1      mov mCur, eax
 00000072  E2 AC        1      loop   mL
 00000074  5A           1      pop edx
 00000075  59           1      pop ecx
```

To rescue, let's turn on this:

ASM

```
LOCAL mPre, mCur, mL
```

Again, running `mFibonacciByMacro` twice with `12` and `13`, fine this time, we have:

ASM

Shrink ▲

```
              mFibonacciByMacro 12
0000018C          1   .data
0000018C 00000000       1      ??0000 DWORD ?
00000190 00000000       1      ??0001 DWORD ?
00000000          1   .code
00000000  51          1      push ecx
00000001  52          1      push edx
00000002  B9 0000000C       1      mov   ecx,12
00000007  B8 00000001       1      mov   eax,1
0000000C  C7 05 0000018C R  1      mov   ??0000,0
     00000000
00000016  C7 05 00000190 R  1      mov   ??0001,0
     00000000
00000020          1   ??0002:
00000020  03 05 0000018C R  1      add  eax,??0000      ; eax = current + previous
00000026  8B 15 00000190 R  1      mov edx, ??0001
0000002C  89 15 0000018C R  1      mov ??0000, edx
00000032  A3 00000190 R     1      mov ??0001, eax
00000037  E2 E7       1      loop   ??0002
00000039  5A          1      pop edx
0000003A  59          1      pop ecx

              mFibonacciByMacro 13
00000194          1   .data
```

```
00000194 00000000        1      ??0003 DWORD ?
00000198 00000000        1      ??0004 DWORD ?
0000003B           1   .code
0000003B  51           1      push ecx
0000003C  52           1      push edx
0000003D  B9 0000000D       1      mov   ecx,13
00000042  B8 00000001       1      mov   eax,1
00000047  C7 05 00000194 R  1      mov   ??0003,0
     00000000
00000051  C7 05 00000198 R  1      mov   ??0004,0
     00000000
0000005B           1   ??0005:
0000005B  03 05 00000194 R  1      add  eax,??0003      ; eax = current + previous
00000061  8B 15 00000198 R  1      mov edx, ??0004
00000067  89 15 00000194 R  1      mov ??0003, edx
0000006D  A3 00000198 R     1      mov ??0004, eax
00000072  E2 E7        1      loop   ??0005
00000074  5A           1      pop edx
00000075  59           1      pop ecx
```

Now the label names, mPre, mCur, and mL, are not visible. Instead, running the first of mFibonacciByMacro 12, the preprocessor generates three system labels ??0000, ??0001, and ??0002 for mPre, mCur, and mL. And for the second mFibonacciByMacro 13, we can find another three system generated labels ??0003, ??0004, and ??0005 for mPre, mCur, and mL. In this way, MASM resolves the redefinition issue in multiple macro executions. You must declare your labels with the LOCAL directive in a macro.

However, by the name LOCAL, the directive sounds misleading, because the system generated ??0000, ??0001, etc. are not limited to a macro's context. They are really global in scope. To verify, I purposely initialize mPre and mCur as 2 and 3:

ASM

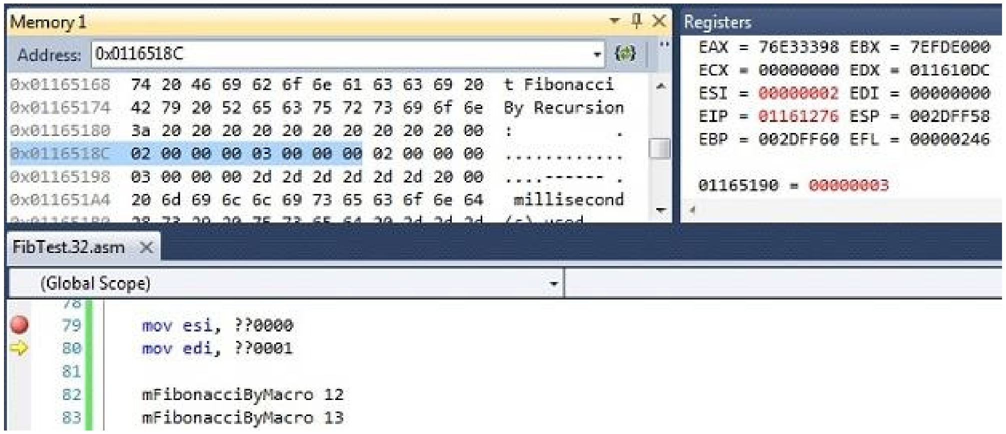
```
LOCAL mPre, mCur, mL
.data
   mPre DWORD 2
   mCur DWORD 3
```

Then simply try to retrieve the values from ??0000 and ??0001 even before calling two mFibonacciByMacro in code

ASM

```
mov esi, ??0000
mov edi, ??0001

mFibonacciByMacro 12
mFibonacciByMacro 13
```

To your surprise probably, when set a breakpoint, you can enter **&**??0000 into the VS debug Address box as a normal variable. As we can see here, the ??0000 memory address is 0x0116518C with DWORD values 2, 3, and so on. Such a ??0000 is allocated on the data segment together with other properly named variables, as shown string ASCII beside:



To summarize, the `LOCAL` directive declared in a macro is to prevent data/code labels from being globally redefined.

Further, as an interesting test question, think of the following multiple running of `mFibonacciByMacro` which is working fine without need of a `LOCAL` directive in `mFibonacciByMacro`. Why?

ASM

```
mov ecx, 2
L1:
   mFibonacciByMacro 12
loop L1
```

Calling an assembly procedure in C/C++ and vice versa

Most assembly programming courses should mention an interesting topic of mixed language programming, e.g., how C/C++ code calls an assembly procedure and how assembly code calls a C/C++ function. But probably, not too much would be involved, especially for manual stack frame manipulation and name decoration. Here in first two sections, I'll give a simple example of C/C++ code calling an assembly procedure. I'll show `C` and `STD` calling conventions, using procedures either with advanced parameter lists or directly dealing with stack frame and name mangling.

The logic just calculates `x-y`, like `10-3` to show `7` resulted:

C++

```
int someFunction(int x, int y)
{
   return x-y;
}

cout << "Call someFunction: 10-3 = " << someFunction(10, 3) << endl;
```

When calling an assembly procedure from a C/C++ function, both must be consistent to use the same calling and naming conventions, so that a linker can resolve references to the caller and its callee. As for Visual C/C++ functions, `C` calling convention can be designated by the keyword __cdecl that should be default in a C/C++ module. And `STD` calling convention can be designated by __stdcall. While on the assembly language side, MASM also provides reserved words `C` and `stdcall` correspondingly. In an assembly language module, you can simply use the `.model` directive to declare all procedures follow `C` calling convention like this:

ASM

```
.model flat, C
```

But you also can override this global declaration by indicating an individual procedure as a different calling convention like:

ASM

```
ProcSTD_CallWithParameterList PROC stdcall, x:DWORD, y:DWORD
```

The following sections suppose that you have basic knowledge and understanding about above.

35. Using C calling convention in two ways

Let's first see a high level procedure with a parameter list easily from the following. I purposely leave blank for the calling convention attribute field in the `.model` directive, but I have `PROC C` to define it as `C` calling convention:

ASM

```
.386P
.model flat  ; No any convention declared

.code
;-----------------------------------------------------------------------
ProcC_CallWithParameterList PROC C, x:DWORD, y:DWORD
; Explicitly declared, C calling convention with Parameter list
; Receives: x and y as unsigned integers
; Returns:  EAX, the result x-y
;-----------------------------------------------------------------------
    mov    eax, x      ; first argument
    sub    eax, y      ; second argument
    ret
ProcC_CallWithParameterList endp
```

The procedure `ProcC_CallWithParameterList` simply does subtraction `x-y` and returns the difference in `EAX`. In order to call it from a function in a `.CPP` file, I must have an equivalent `C` prototype declared in the `.CPP` file accordingly, where `__cdecl` is default:

C++

```
extern "C" int ProcC_CallWithParameterList(int, int);
```

Then call it in `main()` like

C++

```
cout << "C-Call With Parameters:  10-3 = " << ProcC_CallWithParameterList(10, 3) << endl;
```

Using the language attribute `C` to declare `ProcC_CallWithParameterList` makes a lot hidden behind the scene. Please recall what happens to the `C` calling convention __cdecl. The main point I want show here is

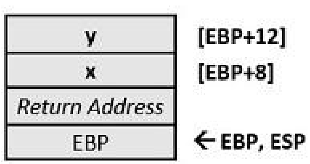
Convention	Implementation required
Argument passing	*From right to left*
Stack maintenance	*Caller pops arguments from the stack*
Name decoration	*Underscore character (_) prefixed to the function name*

Based on these specifications, I can manually create this procedure to fit `C` calling convention:

ASM

```
;-----------------------------------------------------------------------
_ProcC_CallWithStackFrame PROC near
; For __cdecl, manually making C calling convention with Stack Frame
; Receives: x and y on the Stack Frame
; Returns:  EAX, the result x-y
;-----------------------------------------------------------------------
    push    ebp
    mov     ebp,esp

    mov     eax,[ebp+8]      ; first argument x
    sub     eax,[ebp+12]     ; second argument y

    pop     ebp
    ret
_ProcC_CallWithStackFrame endp
```

As seen here, an underscore is prepended as `_ProcC_CallWithStackFrame` and two arguments `x` and `y` passed in reverse order with the stack frame looks like this:



Now let's verify that two procedures work exactly the same by C++ calls

C++

```cpp
extern "C" {
   int ProcC_CallWithParameterList(int, int);
   int ProcC_CallWithStackFrame(int, int);
}

int main()
{
   cout << "C-Call With Parameters:  10-3 = " << ProcC_CallWithParameterList(10, 3) << endl;
   cout << "C-Call With Stack Frame: 10-3 = " << ProcC_CallWithStackFrame(10, 3) << endl;
   // ... ...
}
```

36. Using STD calling convention in two ways

Now we can take a look at `STD` call in the similar way. The following is simply a parameter list procedure with the language attribute `stdcall` defined for `PROC`:

ASM

```asm
;-----------------------------------------------------------------------
ProcSTD_CallWithParameterList PROC stdcall, x:DWORD, y:DWORD
; Explicitly declared, C calling convention with Parameter list
; Receives: x and y as unsigned integers
; Returns:  EAX, the result x-y
;-----------------------------------------------------------------------
    mov    eax, x      ; first argument
    sub    eax, y      ; second argument
    ret
ProcSTD_CallWithParameterList endp
```

Except for the calling conventions, no difference between `ProcSTD_CallWithParameterList` and `ProcC_CallWithParameterList`. In order to call `ProcSTD_CallWithParameterList` from a `C` function, the prototype should be like this:

C++

```cpp
extern "C" int __stdcall ProcSTD_CallWithParameterList(int, int);
```

Notice that `__stdcall` is a must to declare this time. Likewise, using `stdcall` to declare `ProcSTD_CallWithParameterList` also hides a lot details. Please recall what happens to the `STD` calling convention __stdcall. The main point to talk is

Convention	**Implementation required**
Argument passing	*From right to left*
Stack	*Called function itself pops arguments from the stack*

maintenance	
Name decoration	*Underscore character (_) prefixed to the function name. The name is followed by the at sign (@) and the byte count in decimal of the argument list*

Based on these specifications, I can manually create this procedure to fit `STD` calling convention.

ASM

```
;-----------------------------------------------------------------------
_ProcSTD_CallWithStackFrame@8 PROC near
; For __stdcall, manually making STD calling convention with Stack Frame
; Receives: x and y on the Stack Frame
; Returns:  EAX, the result x-y
;-----------------------------------------------------------------------
    push   ebp
    mov    ebp,esp

    mov    eax,[ebp+8]      ; first argument x
    sub    eax,[ebp+12]     ; second argument y

    pop    ebp
    ret 8
_ProcSTD_CallWithStackFrame@8 endp
```

Although the stack frame is the same with two arguments `x` and `y` passed in reverse order, one difference is `_ProcSTD_CallWithStackFrame@8` suffixed by the number eight, 8 bytes of two int type arguments. Another is `ret 8` that is for this procedure itself to release the stack argument memory.

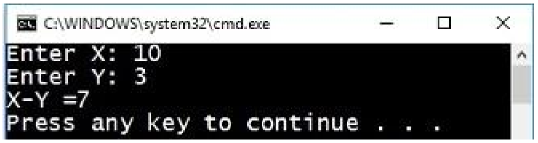
Now put all together, we can verify four procedures getting called by C++ with the same results:

C++

```
extern "C" {
   int ProcC_CallWithParameterList(int, int);
   int ProcC_CallWithStackFrame(int, int);
   int __stdcall ProcSTD_CallWithParameterList(int, int);
   int __stdcall ProcSTD_CallWithStackFrame(int, int);
}

int main()
{
   cout << "C-Call With Parameters:  10-3 = " << ProcC_CallWithParameterList(10, 3) << endl;
   cout << "C-Call With Stack Frame: 10-3 = " << ProcC_CallWithStackFrame(10, 3) << endl;
   cout << "STD-Call With Parameters:  10-3 = " << ProcSTD_CallWithParameterList(10, 3) << endl;
   cout << "STD-Call With Stack Frame: 10-3 = " << ProcSTD_CallWithStackFrame(10, 3) << endl;
}
```

37. Calling cin/cout in an assembly procedure

This section will answer an opposite question, how to call C/C++ functions from an assembly procedure. We really need such a technique to make use of ready-made high level language subroutines for I/O, floating point data, and math function processing. Here I simply want to perform a subtraction task in an assembly procedure, together with input and output by calling `cin` and `cout` like this:



I use `C` calling convention for both calls and in order to do this, let's make three `C` prototypes:

C++

```
extern "C" {
   // A C function to be called in DoSubtraction, passing 'X' or 'Y' as an input prompt
   int ReadFromConsole(unsigned char);
   // A C function to be called in DoSubtraction, to show expression text and integer result
   void DisplayToConsole(char*, int);
   // An assembly procedure to be called in C++ main()
   void DoSubtraction();
}
```

It's trivial defining first two functions to be called in `DoSubtraction`, while `DoSubtraction` is supposed to call in `main()`:

C++

```
int ReadFromConsole(unsigned char by)
{
   cout << "Enter " << by <<": ";
   int i;
   cin >> i;
   return i;
}

void DisplayToConsole(char* s, int n)
{
   cout << s << n <<endl <<endl;
}

int main()
{
   DoSubtraction();
   // ... ...
}
```

Now is time to implement the assembly procedure `DoSubtraction`. Since `DoSubtraction` will call two C++ functions for I/O, I have to make their equivalent prototypes acceptable and recognized by `DoSubtraction`:

ASM

```
ReadFromConsole PROTO C, by:BYTE
DisplayToConsole PROTO C, s:PTR BYTE, n:DWORD
```

Next, simply fill the logic to make it work by invoking `ReadFromConsole` and `DisplayToConsole`:

ASM

```
;----------------------------------------------------------------------------
DoSubtraction PROC C
; Call C++ ReadFromConsole to read X, Y and DisplayToConsole show X-Y
;----------------------------------------------------------------------------
.data
   text2Disp BYTE 'X-Y =', 0
   diff DWORD ?
.code
   INVOKE ReadFromConsole, 'X'
   mov diff, eax
   INVOKE ReadFromConsole, 'Y'
   sub diff, eax
   INVOKE DisplayToConsole, OFFSET text2Disp, diff
   ret
DoSubtraction endp
```

Finally, all source code in above three sections is available for download at CallingAsmProcInC, with `main.cpp`, `subProcs.asm`, and VS project.

About ADDR operator

In 32-bit mode, the `INVOKE`, `PROC`, and `PROTO` directives provide powerful ways for defining and calling procedures. Along with these directives, the `ADDR` operator is an essential helper for defining procedure parameters. By using `INVOKE`, you can make a procedure call almost the same as a function call in high-level programming languages, without caring about the underlying mechanism of the runtime stack.

Unfortunately, the `ADDR` operator is not well explained or documented. The MASM simply said it as an address expression (an expression preceded by ADDR). The textbook [1], mentioned a little more here:

The `ADDR` operator, also available in 32-bit mode, can be used to pass a pointer argument when calling a procedure using `INVOKE`. The following `INVOKE` statement, for example, passes the address of `myArray` to the `FillArray` procedure:

ASM

```
INVOKE FillArray, ADDR myArray
```

The argument passed to `ADDR` must be an assembly time constant. The following is an error:

ASM

```
INVOKE mySub, ADDR [ebp+12] ; error
```

The ADDR operator can only be used in conjunction with INVOKE. The following is an error:

ASM

```
mov esi, ADDR myArray ; error
```

All these sound fine, but are not very clear or accurate, and even not conceptually understandable in programming. `ADDR` not only can be used at assembly time with a global variable like `myArray` to replace `OFFSET`, it also can be placed before a stack memory, such as a local variable or a procedure parameter. The following is actually possible without causing an assembly error:

ASM

```
INVOKE mySub, ADDR [ebp+12]
```

Don't do this, just because unnecessary and somewhat meaningless. The `INVOKE` directive automatically generates the prologue and epilogue code for you with `EBP` and pushes arguments in the format of `EBP` offset. The following sections show you how smart is the `ADDR` operator, with different interpretations at assembly time and at runtime.

38. With global variables defined in data segment

Let's first create a procedure to perform subtraction `C=A-B`, with all three address parameters (call-by-reference). Obviously, we have to use indirect operand `ESI` and dereference it to receive two values from `parA` and `parB`. The out parameter `parC` saves the result back to the caller:

ASM

```
;-------------------------------------------------------------------
SubWithADDR PROC, parA:PTR BYTE, parB:PTR BYTE, parC:PTR BYTE
;
; The task to perform subtraction C=A-B.
; Receives: Pointer parameters parA, parB, parC to three BYTE memory
; Returns:  The result A-B in parC
;-------------------------------------------------------------------

   mov esi, parA
   mov al, [esi]
   mov esi, parB
   sub al, [esi]
   mov esi, parC
   mov [esi], al
   ret
SubWithADDR ENDP
```

And define three global variables in the `DATA` segment:

ASM

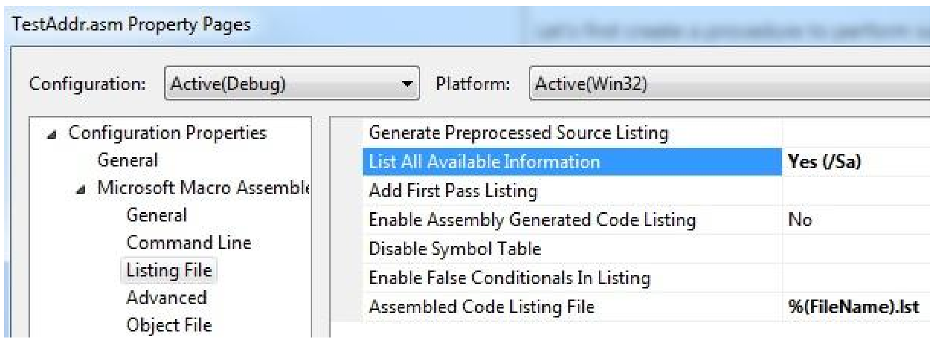
```
.data
valA BYTE 7
valB BYTE 3
valC BYTE 0
```

Then directly pass these global variables to `SubWithADDR` with `ADDR` as three addresses:

ASM

```
; Test 1:
INVOKE SubWithADDR, ADDR valA, ADDR valB, ADDR valC
mov bl, valC
```

Now let's generate the code Listing by use the option "**Listing All Available Information**" as below:



The Listing simply shows three `ADDR` operators replaced by `OFFSET`:

ASM

```
              ; Test 1:
              INVOKE SubWithADDR, ADDR valA, ADDR valB, ADDR valC
0000005B  68 00000002 R   *       push   OFFSET valC
00000060  68 00000001 R   *       push   OFFSET valB
00000065  68 00000000 R   *       push   OFFSET valA
0000006A  E8 FFFFFF91     *       call   SubWithADDR
0000006F  8A 1D 00000002 R      mov bl, valC
```

This is logically reasonable, since `valA`, `valB`, and `valC` are created statically at assembly time and the `OFFSET` operator must be applied at assembly time accordingly. In such a case, where we can use `ADDR`, we also can use `OFFSET` instead. Let's try

ASM

```
INVOKE SubWithADDR, ADDR valA, OFFSET valB, OFFSET valC
```

and regenerate the Listing here to see actually no essential differences:

ASM

```
              ; Test 1:
              INVOKE SubWithADDR, ADDR valA, OFFSET valB, OFFSET valC
0000005B  68 00000002 R   *       push   dword  ptr OFFSET FLAT: valC
00000060  68 00000001 R   *       push   dword  ptr OFFSET FLAT: valB
00000065  68 00000000 R   *       push   OFFSET valA
0000006A  E8 FFFFFF91     *       call   SubWithADDR
0000006F  8A 1D 00000002 R      mov bl, valC
```

39. With local variables created in a procedure

In order to test `ADDR` applied to a local variable, we have to create another procedure where three local variables are defined:

ASM

```
;---------------------------------------------------------
WithLocalVariable PROC
LOCAL locA, locB, locC: BYTE
;
; INVOKE SubWithADDR with three local variable addresses
; Receives: None
; Returns:  The result A-B in CL via locC
;---------------------------------------------------------

   mov locA, 8
   mov locB, 2
   INVOKE SubWithADDR, ADDR locA, ADDR locB, ADDR locC
   mov cl, locC
   ret
WithLocalVariable ENDP
```

Notice that `locA`, `locB`, and `locC` are the memory of `BYTE` type. To reuse `SubWithADDR` by `INVOKE`, I need to prepare values like `8` and `2` to the input arguments `locA` and `locB`, and let `locC` to get back the result. I have to apply `ADDR` to three of them to satisfy the calling interface of `SubWithADDR` prototype. Now simply do the second test:

ASM

```
; Test 2:
call WithLocalVariable
```

At this moment, the local variables are created on the stack frame. This is the memory dynamically created at runtime. Obviously, the assembly time operator `OFFSET` cannot be assumed by `ADDR`. As you might think, the instruction `LEA` should be coming on duty (`LEA` mentioned already: **11**. Implementing with plus (+) instead of ADD and **21**. Making a clear calling interface).

Wow exactly, the operator `ADDR` is now cleaver enough to choose `LEA` this time. To be readable, I want to avoid using Listing to see 2s complement offset to `EBP`. Instead, check the Disassembly intuitive display at runtime here. The code shows three `ADDR` operators replaced by three `LEA` instructions, working with `EBP` on the stack as follows:

ASM

```
    43: WithLocalVariable PROC
00401046 55                   push        ebp
00401047 8B EC                mov         ebp,esp
00401049 83 C4 F4             add         esp,0FFFFFFF4h
    44: LOCAL locA, locB, locC: BYTE
    45: ;
    46: ; INVOKE SubWithADDR with three local variable addresses
    47: ; Receives: None
    48: ; Returns:  The result A-B in CL via locC
    49: ;--------------------------------------------------------
    50:
    51:    mov locA, 8
0040104C C7 45 FC 08 00 00 00 mov         dword ptr [ebp-4],8
    52:    mov locB, 2
00401053 C7 45 F8 02 00 00 00 mov         dword ptr [ebp-8],2
    53:    INVOKE SubWithADDR, ADDR locA, ADDR locB, ADDR locC
0040105A 8D 45 F7             lea         eax,[ebp-9]
0040105D 50                   push        eax
0040105E 8D 45 F8             lea         eax,[ebp-8]
00401061 50                   push        eax
00401062 8D 45 FC             lea         eax,[ebp-4]
00401065 50                   push        eax
00401066 E8 C5 FF FF FF       call        00401030
    54:    mov cl, locC
0040106B 8A 4D F7             mov         cl,byte ptr [ebp-9]
    55:    ret
0040106E C9                   leave
0040106F C3                   ret
    56: WithLocalVariable ENDP
```

where the hexadecimal `00401030` is `SubWithADDR`'s address. Because of the `LOCAL` directive, MASM automatically generates the prologue and epilogue with `EBP` representations. To view `EBP` offset instead of variable names like `locA`, `locB`, and `locC`, just uncheck the Option: **Show symbol names**:

40. With arguments received from within a procedure

The third test is to make ADDR apply to arguments. I create a procedure WithArgumentPassed and call it like:

ASM

```
; Test3:
INVOKE WithArgumentPassed, 9, 1, OFFSET valC
```

Reuse the global valC here with OFFSET, since I hope to get the result 8 back. It's interesting to see how to push three values in the Listing:

ASM

```
              ; Test3:
              INVOKE WithArgumentPassed, 9, 1, OFFSET valC
0000007B  68 00000002 R   *       push   dword  ptr OFFSET FLAT: valC
00000080  6A 01      *       push    +000000001h
00000082  6A 09      *       push    +000000009h
00000084  E8 FFFFFFB7      *       call   WithArgumentPassed
```

The implementation of WithArgumentPassed is quite straight and simply reuse SubWithADDR by passing arguments argA and argB prefixed with ADDR to be addresses, while ptrC already a pointer without ADDR:

ASM

```
;-----------------------------------------------------------------
WithArgumentPassed PROC argA: BYTE, argB: BYTE, ptrC: PTR BYTE
;
; INVOKE SubWithADDR with three argument addresses
; Receives: Parameters argA, argB in BYTE and ptrC as PTR BYTE
; Returns:  The result A-B in DL via ptrC
;-----------------------------------------------------------------

   INVOKE SubWithADDR, ADDR argA, ADDR argB, ptrC
   mov esi, ptrC
   mov dl, [esi]
   ret
WithArgumentPassed ENDP
```

If you are familiar with the concepts of stack frame, imagine the behavior of ADDR that must be very similar to the local variables, since arguments are also dynamically created memory on the stack at runtime. The following is the generated Listing with two ADDR operators replaced by LEA. Only difference is the positive offset to EBP here:

ASM

```
           ;-----------------------------------------------------------
00000040   WithArgumentPassed PROC argA: BYTE, argB: BYTE, ptrC: PTR BYTE
           ;
```

```
          ; INVOKE SubWithADDR with three argument addresses
          ; Receives: Parameters argA, argB in BYTE and ptrC as PTR BYTE
          ; Returns:  The result A-B in DL via ptrC
          ;----------------------------------------------------------

00000040  55         *       push   ebp
00000041  8B EC      *       mov    ebp, esp
             INVOKE SubWithADDR, ADDR argA, ADDR argB, ptrC
00000043  FF 75 10      *       push   dword  ptr ss:[ebp]+000000010h
00000046  8D 45 0C      *       lea    eax, byte  ptr ss:[ebp]+00Ch
00000049  50         *       push   eax
0000004A  8D 45 08      *       lea    eax, byte  ptr ss:[ebp]+008h
0000004D  50         *       push   eax
0000004E  E8 FFFFFFAD      *       call   SubWithADDR
00000053  8B 75 10         mov esi, ptrC
00000056  8A 16         mov dl, [esi]
             ret
00000058  C9         *       leave
00000059  C2 000C      *       ret    0000Ch
0000005C        WithArgumentPassed ENDP
```

Because of `WithArgumentPassed PROC` with a parameter-list, MASM also generates the prologue and epilogue with `EBP` representations automatically. Three address arguments pushed in the reverse order are `EBP` plus `16` (`ptrC`), plus `12` (`argB`), and plus `8` (`argA`).

Finally, all source code in above three sections available to download at TestADDR, with `TestADDR.asm`, `TestADDR.lst`, and `TestADDR.vcxproj`.

Summary

I talked so much about miscellaneous features in assembly language programming. Most of them are from our class teaching and assignment discussion [1]. The basic practices are presented here with short code snippets for better understanding without irrelevant details involved. The main purpose is to show assembly language specific ideas and methods with more strength than other languages.

As noticed, I haven't given a complete test code that requires a programming environment with input and output. For an easy try, you can go [2] to download the Irvine32 library and setup your MASM programming environment with Visual Studio, while you have to learn a lot in advance to prepare yourself first. For example, the statement `exit` mentioned here in `main` is not an element in assembly language, but is defined as `INVOKE ExitProcess,0` there.

Assembly language is notable for its one-to-one correspondence between an instruction and its machine code as shown in several Listings here. Via assembly code, you can get closer to the heart of the machine, such as registers and memory. Assembly language programming often plays an important role in both academic study and industry development. I hope this article could serve as an useful reference for students and professionals as well.

References

1. CSCI 241, Assembly Language Programming class site
2. Kip Irvine, Assembly Language for x86 Processors, 7th edition
3. MASM Programmer's Guide, MASM 6.1 Documentation
4. Zuoliu Ding, Something You May Not Know About the Macro in MASM
5. Zuoliu Ding, Something You May Not Know About the Switch Statement in C/C++

History

- January 28, 2019 -- Added: About ADDR operator, three sections
- January 22, 2017 -- Added: Calling an assembly procedure in C/C++ and vice versa, three sections
- January 11, 2017 -- Added: FOR/WHILE loop and Making loop more efficient, two sections
- December 20, 2016 -- Added: Ambiguous "LOCAL" directive, two sections
- November 28, 2016 -- Added: Signed and Unsigned, four sections
- October 30, 2016 -- Added: About PTR operator, two sections
- October 16, 2016 -- Added: Little-endian, two sections
- October 11, 2016 -- Added: the section, Using INC to avoid PUSHFD and POPFD
- October 2, 2016 -- Added: the section, Using atomic instructions
- August 1, 2016 -- Original version posted

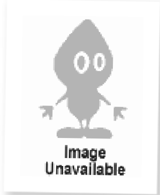
License

Written By

Zuoliu Ding

United States

Adjunct Professor in Computer Science



Comments and Discussions

You must **Sign In** to use this message board.

Search Comments

Spacing Relaxed Layout Normal Per page 25 Update

First Prev Next

assembly language x86 processor	**Member 14784282**	**30-Apr-20 20:07**
the UNWISE use of XCHG	**Member 14691692**	**16-Dec-19 14:45**
Assembly Code	**Member 14170884**	**4-Mar-19 10:13**
Re: Assembly Code	Member 14784282	30-Apr-20 20:10
Show Intentions	**G3ZHX**	**31-Jan-19 8:25**
Re: Show Intentions	Zuoliu Ding	2-Feb-19 7:54
Very good article, 5 points. I suggest a second part using x86-64	**Armando A Bouza**	**30-Jan-19 7:02**
Re: Very good article, 5 points. I suggest a second part using x86-64	Zuoliu Ding	30-Jan-19 10:52
Where is the article?	**SMD111**	**27-Mar-17 6:33**
Vote 5+	**danzar101**	**27-Jan-17 14:28**
My vote of 5	**Farhad Reza**	**21-Dec-16 10:07**
Re: My vote of 5	Zuoliu Ding	24-Jan-17 7:00
Thank you	**Hugh Wood**	**21-Dec-16 8:01**
Avoiding PUSHFD and POPFD	**Tom Spink**	**21-Dec-16 2:18**
Re: Avoiding PUSHFD and POPFD	Zuoliu Ding	21-Dec-16 6:13
My vote of 5	**csharpbd**	**20-Dec-16 19:56**

Minor improvement	**Nelek**	**20-Dec-16 13:02**
Re: Minor improvement	Zuoliu Ding	20-Dec-16 18:40
Re: Minor improvement	Nelek	21-Dec-16 1:23
My vote of 5!	**jediYL**	**3-Dec-16 20:37**
Nice article, within limits.	**Daniel Pfeffer**	**30-Nov-16 22:25**
Re: Nice article, within limits.	Zuoliu Ding	1-Dec-16 6:54
Re: Nice article, within limits.	David A McKelvie	23-Jan-17 23:22
Re: Nice article, within limits.	Zuoliu Ding	26-Mar-21 16:35
What compiler are you using for this assembly language?	**marshal craft**	**29-Nov-16 9:00**

Last Visit: 31-Dec-99 18:00 Last Update: 22-Sep-22 5:26 Refresh 1 2 Next ▷

General News Suggestion Question Bug Answer Joke Praise Rant Admin

Use Ctrl+Left/Right to switch messages, Ctrl+Up/Down to switch threads, Ctrl+Shift+Left/Right to switch pages.