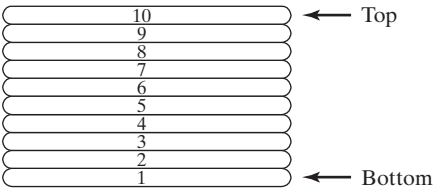
Lecture No.11

Lecture Outlines

5.1 Stack Operations

If we place ten plates on each other as in the following diagram, the result can be called a stack. While it might be possible to remove a dish from the middle of the stack, it is much more common to remove from the top. New plates can be added to the top of the stack, but never to the bottom or middle (Fig. 5–1):

Figure 5–1 Stack of plates



A *stack data structure* follows the same principle as a stack of plates: New values are added to the top of the stack, and existing values are removed from the top. Stacks in general are useful structures for a variety of programming applications, and they can easily be implemented using object-oriented programming methods. If you have taken a programming course that used data structures, you have worked with the *stack abstract data type*. A stack is also called a LIFO structure (*Last-In, First-Out*) because the last value put into the stack is always the first value taken out.

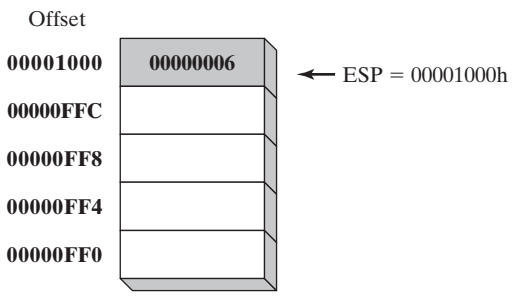
In this chapter, we concentrate specifically on the *runtime stack*. It is supported directly by hardware in the CPU, and it is an essential part of the mechanism for calling and returning from procedures. Most of the time, we just call it the stack.

5.1.1 Runtime Stack (32-Bit Mode)

The *runtime stack* is a memory array managed directly by the CPU, using the ESP (extended stack pointer) register, known as the *stack pointer register*. In 32-bit mode, ESP register holds a 32-bit offset into some location on the stack. We rarely manipulate ESP directly; instead, it is indirectly modified by instructions such as CALL, RET, PUSH, and POP.

ESP always points to the last value to be added to, or *pushed* on, the top of stack. To demonstrate, let's begin with a stack containing one value. In Fig. 5-2, the ESP contains hexadecimal 00001000, the offset of the most recently pushed value (00000006). In our diagrams, the top of the stack moves downward when the stack pointer decreases in value:

Figure 5–2 A stack containing a single value



Each stack location in this figure contains 32 bits, which is the case when a program is running in 32-bit mode.

> The runtime stack discussed here is not the same as the *stack abstract data type* (ADT) discussed in data structures courses. The runtime stack works at the system level to handle subroutine calls. The stack ADT is a programming construct typically written in a high-level programming language such as C++ or Java. It is used when implementing algorithms that depend on last-in, first-out operations.

Push Operation

A 32-bit *push operation* decrements the stack pointer by 4 and copies a value into the location in the stack pointed to by the stack pointer. Figure 5-3 shows the effect of pushing 000000A5 on a stack that already contains one value (00000006). Notice that the ESP register always points to the last item pushed on the stack. The figure shows the stack ordering opposite to that of the stack of plates we saw earlier, because the runtime stack grows downward in memory, from higher addresses to lower addresses. Before the push, ESP = 00001000h; after the push, ESP = 00000FFCh. Figure 5-4 shows the same stack after pushing a total of four integers.

Figure 5–3 Pushing integers on the stack.

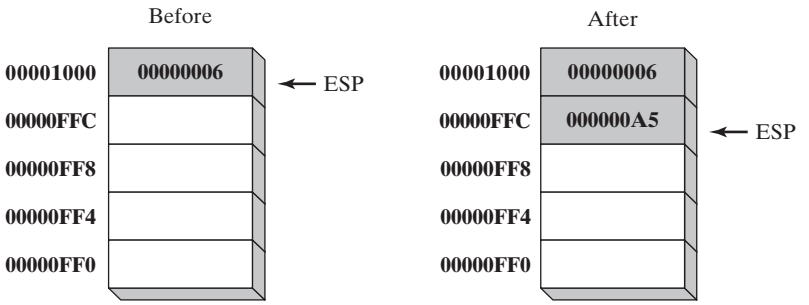
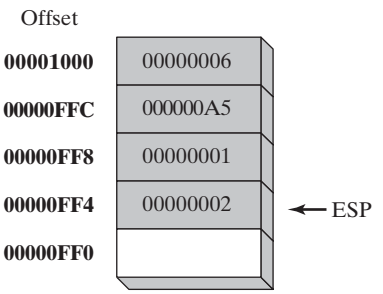


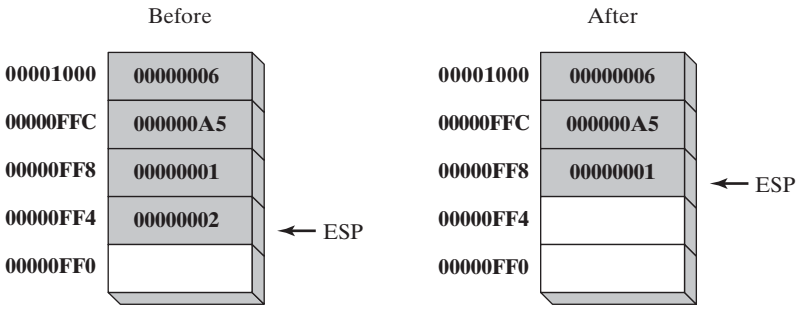
Figure 5–4 Stack, after pushing 00000001 and 00000002.



Pop Operation

A *pop operation* removes a value from the stack. After the value is popped from the stack, the stack pointer is incremented (by the stack element size) to point to the next-highest location in the stack. Figure 5-5 shows the stack before and after the value 00000002 is popped.

Figure 5–5 Popping a value from the runtime stack.



The area of the stack below ESP is logically empty, and will be overwritten the next time the current program executes any instruction that pushes a value on the stack.

Stack Applications

There are several important uses of runtime stacks in programs:

- A stack makes a convenient temporary save area for registers when they are used for more than one purpose. After they are modified, they can be restored to their original values.
- When the CALL instruction executes, the CPU saves the current subroutine's return address on the stack.
- When calling a subroutine, you pass input values called *arguments* by pushing them on the stack.
- The stack provides temporary storage for local variables inside subroutines.

5.1.2 PUSH and POP Instructions

PUSH Instruction

The PUSH instruction first decrements ESP and then copies a source operand into the stack. A 16-bit operand causes ESP to be decremented by 2. A 32-bit operand causes ESP to be decremented by 4. There are three instruction formats:

```
PUSH reg/mem16
PUSH reg/mem32
PUSH imm32
```

POP Instruction

The POP instruction first copies the contents of the stack element pointed to by ESP into a 16- or 32-bit destination operand and then increments ESP. If the operand is 16 bits, ESP is incremented by 2; if the operand is 32 bits, ESP is incremented by 4:

```
POP reg/mem16
POP reg/mem32
```

PUSHFD and POPFD Instructions

The PUSHFD instruction pushes the 32-bit EFLAGS register on the stack, and POPFD pops the stack into EFLAGS:

```
pushfd
popfd
```

The MOV instruction cannot be used to copy the flags to a variable, so PUSHFD may be the best way to save the flags. There are times when it is useful to make a backup copy of the flags so you can restore them to their former values later. Often, we enclose a block of code within PUSHFD and POPFD:

```
pushfd                          ; save the flags
;
; any sequence of statements here...
;
popfd                           ; restore the flags
```

When using pushes and pops of this type, be sure the program's execution path does not skip over the POPFD instruction. When a program is modified over time, it can be tricky to remember where all the pushes and pops are located. The need for precise documentation is critical!

A less error-prone way to save and restore the flags is to push them on the stack and immediately pop them into a variable:

```
.data
saveFlags DWORD ?
.code
pushfd                          ; push flags on stack
pop  saveFlags                  ; copy into a variable
```

The following statements restore the flags from the same variable:

```
push saveFlags                  ; push saved flag values
popfd                           ; copy into the flags
```

PUSHAD, PUSHA, POPAD, and POPA

The PUSHAD instruction pushes all of the 32-bit general-purpose registers on the stack in the following order: EAX, ECX, EDX, EBX, ESP (value before executing PUSHAD), EBP, ESI, and EDI. The POPAD instruction pops the same registers off the stack in reverse order. Similarly, the PUSHA instruction, pushes the 16-bit general-purpose registers (AX, CX, DX, BX, SP, BP, SI, DI) on the stack in the order listed. The POPA instruction pops the same registers in reverse. You should only use PUSHA and POPA when programming in 16-bit mode. We cover 16-bit programming in Chapters 14–17.

If you write a procedure that modifies a number of 32-bit registers, use PUSHAD at the beginning of the procedure and POPAD at the end to save and restore the registers. The following code fragment is an example:

```
MySub PROC
     pushad                     ; save general-purpose registers
     .
     .
     mov eax,...
     mov edx,...
     mov ecx,...
     .
     .
     popad                      ; restore general-purpose registers
     ret
MySub ENDP
```

An important exception to the foregoing example must be pointed out; procedures returning results in one or more registers should not use PUSHA and PUSHAD. Suppose the following **ReadValue** procedure returns an integer in EAX; the call to POPAD overwrites the return value from EAX:

```
ReadValue PROC
     pushad                      ; save general-purpose registers
     .
     .
     mov   eax,return_value
     .
     .
     popad                       ; overwrites EAX!
     ret
ReadValue ENDP
```

Example: Reversing a String

Let's look at a program named *RevStr* that loops through a string and pushes each character on the stack. It then pops the letters from the stack (in reverse order) and stores them back into the same string variable. Because the stack is a LIFO (*last-in, first-out*) structure, the letters in the string are reversed:

```
; Reversing a String           (RevStr.asm)

.386
.model flat,stdcall
.stack 4096
ExitProcess PROTO,dwExitCode:DWORD

.data
aName BYTE "Abraham Lincoln",0
nameSize = ($ - aName) - 1

.code
main PROC
; Push the name on the stack.
     mov   ecx,nameSize
     mov   esi,0

L1:  movzx eax,aName[esi]              ; get character
     push  eax                         ; push on stack
     inc   esi
     loop  L1

; Pop the name from the stack, in reverse,
; and store in the aName array.
     mov   ecx,nameSize
     mov   esi,0

L2:  pop   eax                         ; get character
     mov   aName[esi],al               ; store in string
     inc   esi

     loop  L2
```

```
    INVOKE ExitProcess,0
main ENDP
END main
```

5.2 Defining and Using Procedures

If you've already studied a high-level programming language, you know how useful it can be to divide programs into *subroutines*. A complicated problem is usually divided into separate tasks before it can be understood, implemented, and tested effectively. In assembly language, we typically use the term *procedure* to mean a subroutine. In other languages, subroutines are called methods or functions.

In terms of object-oriented programming, the functions or methods in a single class are roughly equivalent to the collection of procedures and data encapsulated in an assembly language module. Assembly language was created long before object-oriented programming, so it doesn't have the formal structure found in object-oriented languages. Assembly programmers must impose their own formal structure on programs.

5.2.1 PROC Directive

Defining a Procedure

Informally, we can define a *procedure* as a named block of statements that ends in a return statement. A procedure is declared using the PROC and ENDP directives. It must be assigned a name (a valid identifier). Each program we've written so far contains a procedure named **main**, for example,

```
main PROC
.
.
main ENDP
```

When you create a procedure other than your program's startup procedure, end it with a RET instruction. RET forces the CPU to return to the location from where the procedure was called:

```
sample PROC
    .
    .
    ret
sample ENDP
```

Labels in Procedures

By default, labels are visible only within the procedure in which they are declared. This rule often affects jump and loop instructions. In the following example, the label named *Destination* must be located in the same procedure as the JMP instruction:

```
jmp Destination
```

It is possible to work around this limitation by declaring a *global label*, identified by a double colon (::) after its name:

```
Destination::
```

In terms of program design, it's not a good idea to jump or loop outside of the current procedure. Procedures have an automated way of returning and adjusting the runtime stack. If you directly transfer out of a procedure, the runtime stack can easily become corrupted. For more information about the runtime stack.

Example: SumOf Three Integers

Let's create a procedure named **SumOf** that calculates the sum of three 32-bit integers. We will assume that relevant integers are assigned to EAX, EBX, and ECX before the procedure is called. The procedure returns the sum in EAX:

```
SumOf PROC
    add  eax,ebx
    add  eax,ecx
    ret
SumOf ENDP
```

Documenting Procedures

A good habit to cultivate is that of adding clear and readable documentation to your programs. The following are a few suggestions for information that you can put at the beginning of each procedure:

- A description of all tasks accomplished by the procedure.
- A list of input parameters and their usage, labeled by a word such as **Receives**. If any input parameters have specific requirements for their input values, list them here.
- A description of any values returned by the procedure, labeled by a word such as **Returns**.
- A list of any special requirements, called *preconditions*, that must be satisfied before the procedure is called. These can be labeled by the word **Requires**. For example, for a procedure that draws a graphics line, a useful precondition would be that the video display adapter must already be in graphics mode.

> The descriptive labels we've chosen, such as Receives, Returns, and Requires, are not absolutes; other useful names are often used.

With these ideas in mind, let's add appropriate documentation to the **SumOf** procedure:

```
;-----------------------------------------------------
; sumof
;
; Calculates and returns the sum of three 32-bit integers.
; Receives: EAX, EBX, ECX, the three integers. May be
;           signed or unsigned.
; Returns:  EAX = sum
-----------------------------------------------------
SumOf PROC

        add   eax,ebx
        add   eax,ecx
        ret
    SumOf ENDP
```

Functions written in high-level languages like C and C++ typically return 8-bit values in AL, 16-bit values in AX, and 32-bit values in EAX.

5.2.2 CALL and RET Instructions

The CALL instruction calls a procedure by directing the processor to begin execution at a new memory location. The procedure uses a RET (return from procedure) instruction to bring the processor back to the point in the program where the procedure was called. Mechanically speaking, the CALL instruction pushes its return address on the stack and copies the called procedure's address into the instruction pointer. When the procedure is ready to return, its RET instruction pops the return address from the stack into the instruction pointer. In 32-bit mode, the CPU executes the instruction in memory pointed to by EIP (instruction pointer register). In 16-bit mode, IP points to the instruction.

Call and Return Example

Suppose that in **main**, a CALL statement is located at offset 00000020. Typically, this instruction requires 5 bytes of machine code, so the next statement (a MOV in this case) is located at offset 00000025:

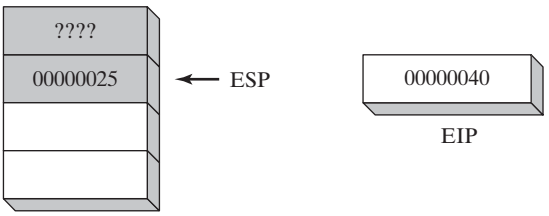
```
            main PROC
00000020      call MySub
00000025      mov  eax,ebx
```

Next, suppose that the first executable instruction in **MySub** is located at offset 00000040:

```
            MySub PROC
00000040     mov eax,edx
               .
               .
             ret
            MySub ENDP
```

When the CALL instruction executes (Fig. 5-6), the address following the call (00000025) is pushed on the stack and the address of **MySub** is loaded into EIP. All instructions in **MySub** execute up to its RET instruction. When the RET instruction executes, the value in the stack pointed to by ESP is popped into EIP (step 1 in Fig. 5-7). In step 2, ESP is incremented so it points to the previous value on the stack (step 2).

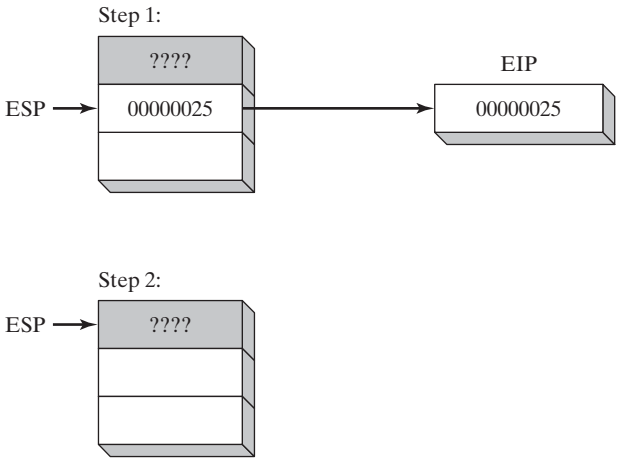
FIGURE 5–6 Executing a CALL instruction.



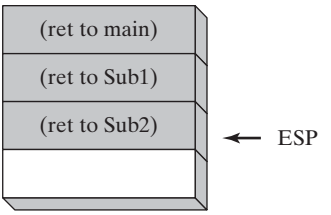
5.2.3 Nested Procedure Calls

A *nested procedure call* occurs when a called procedure calls another procedure before the first procedure returns. Suppose that **main** calls a procedure named **Sub1**. While **Sub1** is executing, it calls the **Sub2** procedure. While **Sub2** is executing, it calls the **Sub3** procedure. The process is shown in Fig. 5-8.

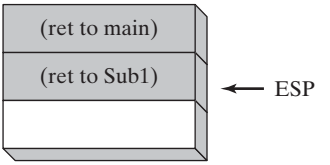
Figure 5–7 Executing the RET instruction.



When the RET instruction at the end of **Sub3** executes, it pops the value at stack[ESP] into the instruction pointer. This causes execution to resume at the instruction following the call **Sub3** instruction. The following diagram shows the stack just before the return from **Sub3** is executed:



After the return, ESP points to the next-highest stack entry. When the RET instruction at the end of **Sub2** is about to execute, the stack appears as follows:



Finally, when **Sub1** returns, stack[ESP] is popped into the instruction pointer, and execution resumes in **main**:

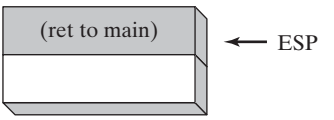
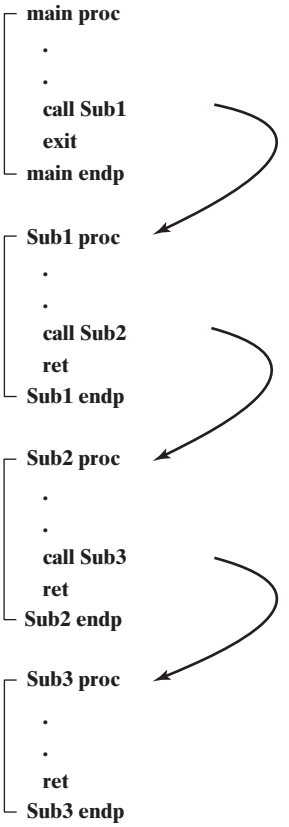


Figure 5–8 Nested procedure calls.



Clearly, the stack proves itself a useful device for remembering information, including nested procedure calls. Stack structures, in general, are used in situations where programs must retrace their steps in a specific order.

5.2.4 Passing Register Arguments to Procedures

If you write a procedure that performs some standard operation such as calculating the sum of an integer array, it's not a good idea to include references to specific variable names inside the procedure. If you did, the procedure could only be used with one array. A better approach is to pass the offset of an array to the procedure and pass an integer specifying the number of array elements. We call these *arguments* (or *input parameters)*. In assembly language, it is common to pass arguments inside general-purpose registers.

In the preceding section we created a simple procedure named **SumOf** that added the integers in the EAX, EBX, and ECX registers. In **main**, before calling **SumOf**, we assign values to EAX, EBX, and ECX:

```
.data
theSum  DWORD  ?
.code
main PROC
```

```
mov   eax,10000h                ; argument
mov   ebx,20000h                ; argument
mov   ecx,30000h                ; argument
call  Sumof                     ; EAX = (EAX + EBX + ECX)
mov   theSum,eax                ; save the sum
```

After the CALL statement, we have the option of copying the sum in EAX to a variable.

5.2.5 Example: Summing an Integer Array

A very common type of loop that you may have already coded in C++ or Java is one that calculates the sum of an integer array. This is very easy to implement in assembly language, and it can be coded in such a way that it will run as fast as possible. For example, one can use registers rather than variables inside a loop.

Let's create a procedure named **ArraySum** that receives two parameters from a calling program: a pointer to an array of 32-bit integers, and a count of the number of array values. It calculates and returns the sum of the array in EAX:

```
;-----------------------------------------------------
; ArraySum
;
; Calculates the sum of an array of 32-bit integers.
; Receives: ESI = the array offset
;           ECX = number of elements in the array
; Returns:  EAX = sum of the array elements
;-----------------------------------------------------
ArraySum PROC
    push  esi                  ; save ESI, ECX
    push  ecx
    mov   eax,0                ; set the sum to zero

L1: add   eax,[esi]            ; add each integer to sum
    add   esi,TYPE DWORD       ; point to next integer

    loop  L1                   ; repeat for array size

    pop   ecx                  ; restore ECX, ESI
    pop   esi
    ret                        ; sum is in EAX
ArraySum ENDP
```

Nothing in this procedure is specific to a certain array name or array size. It could be used in any program that needs to sum an array of 32-bit integers. Whenever possible, you should also create procedures that are flexible and adaptable.

Testing the ArraySum Procedure

The following program tests the ArraySum procedure by calling it and passing the offset and length of an array of 32-bit integers. After calling ArraySum, it saves the procedure's return value in a variable named theSum.

```
; Testing the ArraySum procedure (TestArraySum.asm)

.386
.model flat, stdcall
.stack 4096
ExitProcess PROTO, dwExitCode:DWORD

.data
array DWORD 10000h,20000h,30000h,40000h,50000h
theSum DWORD ?

.code
main PROC
     mov   esi,OFFSET array     ; ESI points to array
     mov   ecx,LENGTHOF array   ; ECX = array count
     call  ArraySum             ; calculate the sum
     mov   theSum,eax           ; returned in EAX

     INVOKE ExitProcess,0
main ENDP

;-----------------------------------------------------
; ArraySum
; Calculates the sum of an array of 32-bit integers.
; Receives: ESI = the array offset
; ECX = number of elements in the array
; Returns: EAX = sum of the array elements
;-----------------------------------------------------

ArraySum PROC
     push  esi                  ; save ESI, ECX
     push  ecx
     mov   eax,0                ; set the sum to zero

L1:
     add   eax,[esi]            ; add each integer to sum
     add   esi,TYPE DWORD       ; point to next integer
     loop  L1                   ; repeat for array size

       pop   ecx                  ; restore ECX, ESI
       pop   esi
       ret                        ; sum is in EAX
  ArraySum ENDP

  END main
```

5.2.6 Saving and Restoring Registers

In the **ArraySum** example, ECX and ESI were pushed on the stack at the beginning of the procedure and popped at the end. This action is typical of most procedures that modify registers. Always save and restore registers that are modified by a procedure so the calling program can be sure that none of its own register values will be overwritten. The exception to this rule pertains to registers used as return values, usually EAX. Do not push and pop them.

USES Operator

The USES operator, coupled with the PROC directive, lets you list the names of all registers modified within a procedure. USES tells the assembler to do two things: First, generate PUSH instructions that save the registers on the stack at the beginning of the procedure. Second, generate POP instructions that restore the register values at the end of the procedure. The USES operator immediately follows PROC, and is itself followed by a list of registers on the same line separated by spaces or tabs (not commas).

The ArraySum procedure from Section 5.2.5 used PUSH and POP instructions to save and restore ESI and ECX. The USES operator can more easily do the same:

```
ArraySum PROC USES esi ecx
     mov   eax,0                ; set the sum to zero
L1:
     add   eax,[esi]            ; add each integer to sum
     add   esi,TYPE DWORD       ; point to next integer
     loop  L1                   ; repeat for array size

     ret                        ; sum is in EAX
ArraySum ENDP
```

The corresponding code generated by the assembler shows the effect of USES:

```
ArraySum PROC
     push  esi
     push  ecx
     mov   eax,0                ; set the sum to zero

L1:
     add   eax,[esi]            ; add each integer to sum
     add   esi,TYPE DWORD       ; point to next integer
     loop  L1                   ; repeat for array size

     pop   ecx
     pop   esi
     ret
ArraySum ENDP
```

> *Debugging Tip:* When using the Microsoft Visual Studio debugger, you can view the hidden machine instructions generated by MASM's advanced operators and directives. Right-click in the Debugging window and select *Go to Disassembly*. This window displays your program's source code along with hidden machine instructions generated by the assembler.

Exception There is an important exception to our standing rule about saving registers that applies when a procedure returns a value in a register (usually EAX). In this case, the return register should not be pushed and popped. For example, in the SumOf procedure in the following example, it pushes and pops EAX, causing the procedure's return value to be lost:

```
SumOf PROC                      ; sum of three integers
     push  eax                  ; save EAX
     add   eax,ebx              ; calculate the sum
     add   eax,ecx              ; of EAX, EBX, ECX
     pop   eax                  ; lost the sum!
     ret
SumOf ENDP
```