Stack Operations

If we place 10 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 –Figure(1):

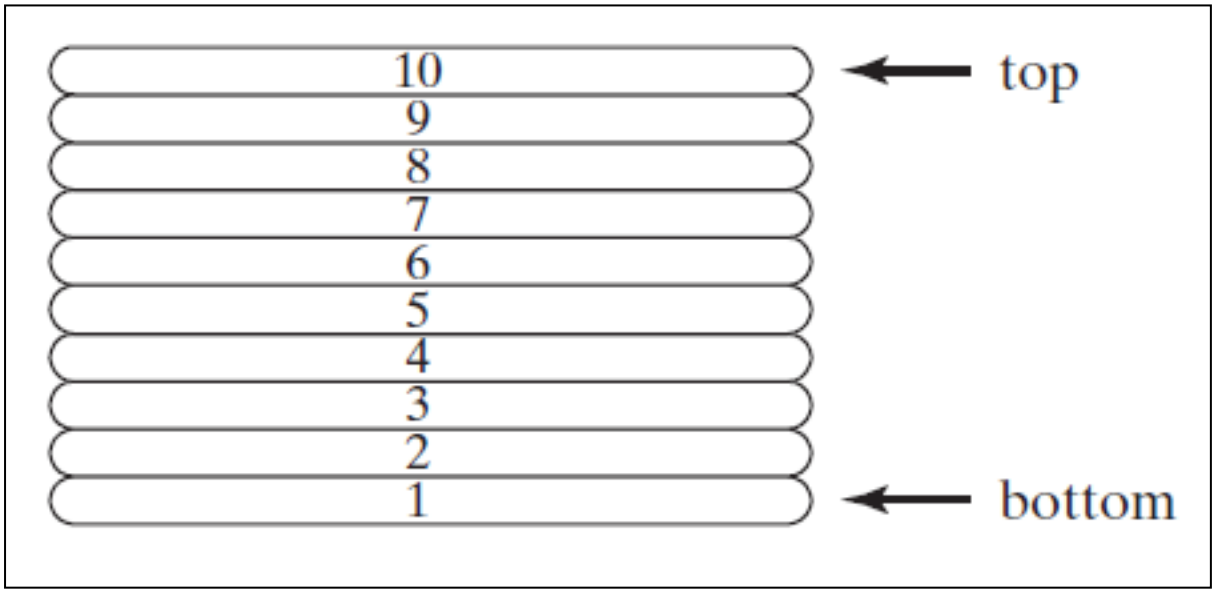

Figure (1) Stack of Plates.

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. 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.

Runtime Stack

The *runtime stack* is a memory array managed directly by the CPU, using the ESP register, known as the *stack pointer register*. The 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 Figure (2), the ESP (extended stack pointer) 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:

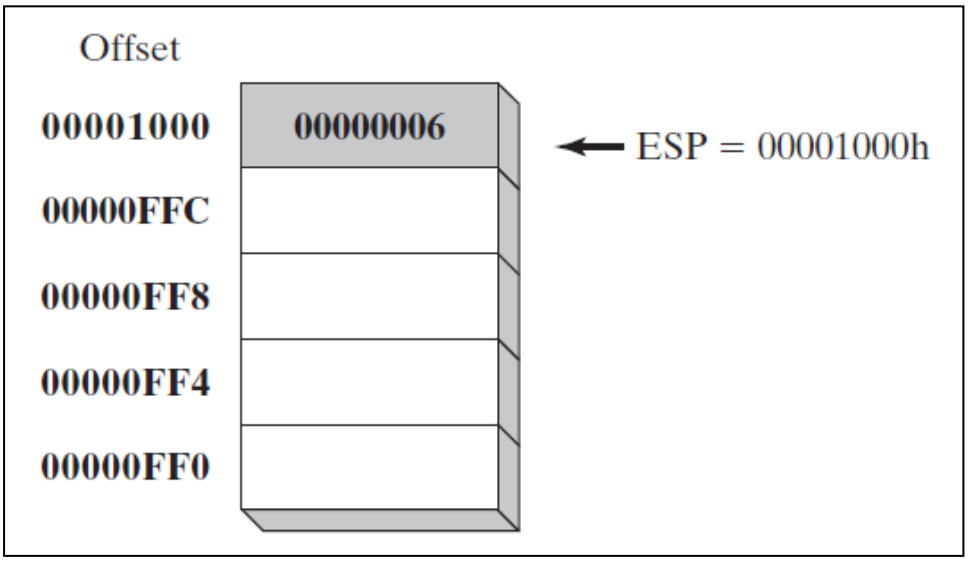

Figure (2) A Stack Containing a Single Value.

Each stack location in this figure contains 32 bits, which is the case when a program is running in 32-bit mode. In 16-bit real-address mode, the SP register points to the most recently pushed value and stack entries are typically 16 bits long.

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 (3) shows the effect of pushing 000000A5 on a

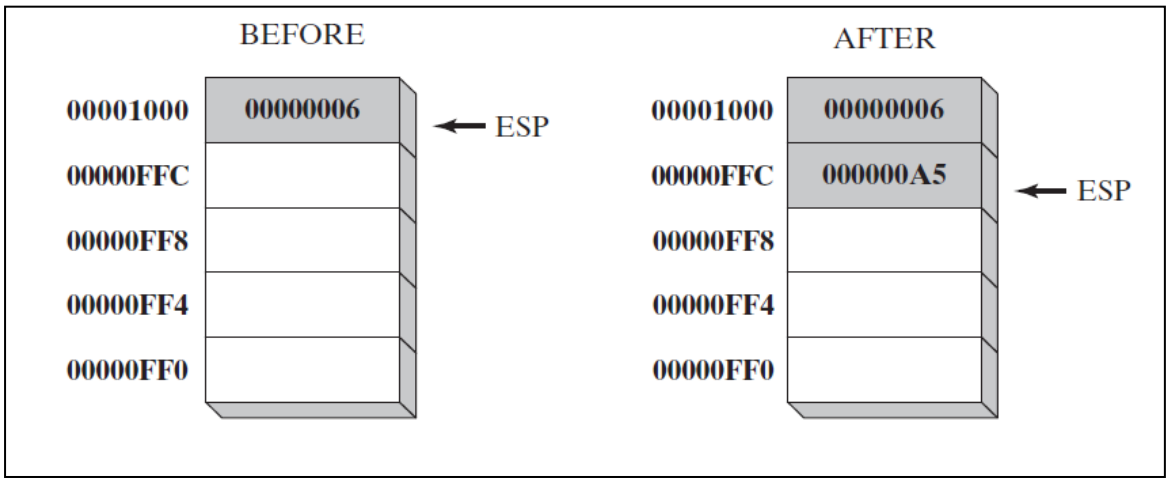

Figure (3) Pushing Integers on the Stack

stack that already contains one value (00000006). Notice that the ESP register always points to the top of 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 (4) shows the same stack after pushing a total of four integers.

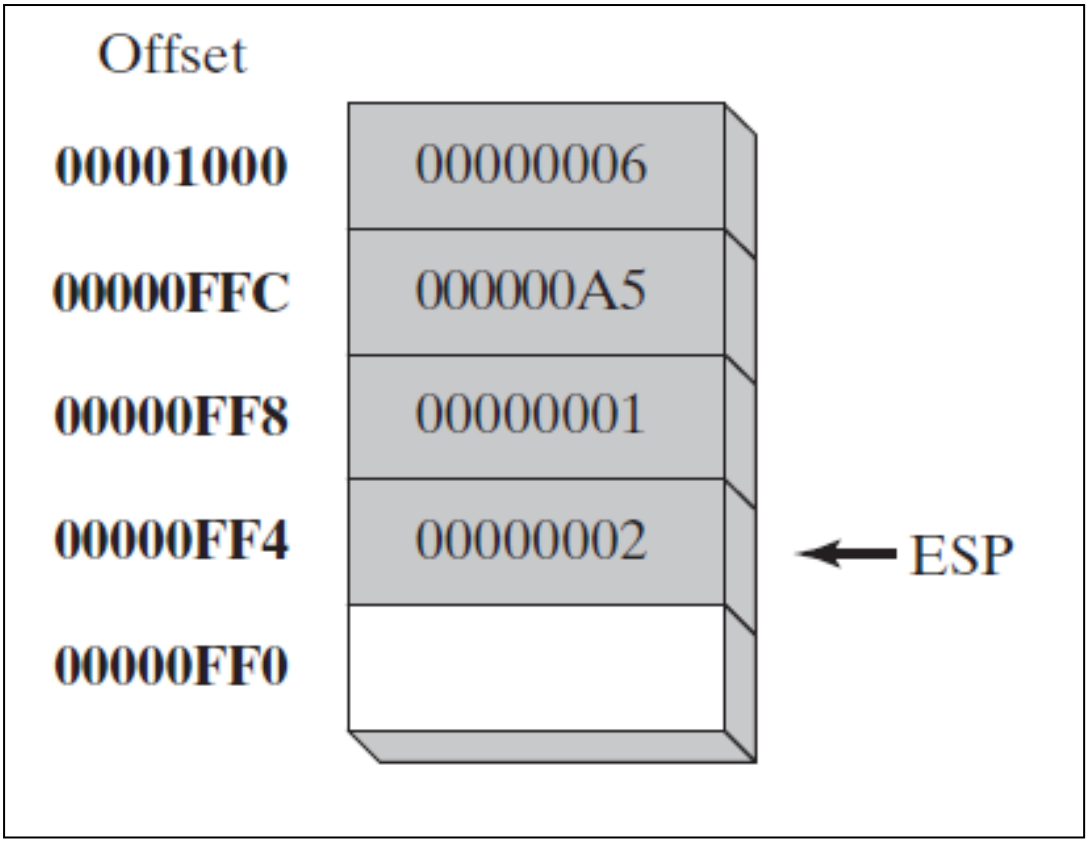

Figure (4) Stack, after Pushing 00000001 and 00000002.

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) shows the stack before and after the value 00000002 is popped.

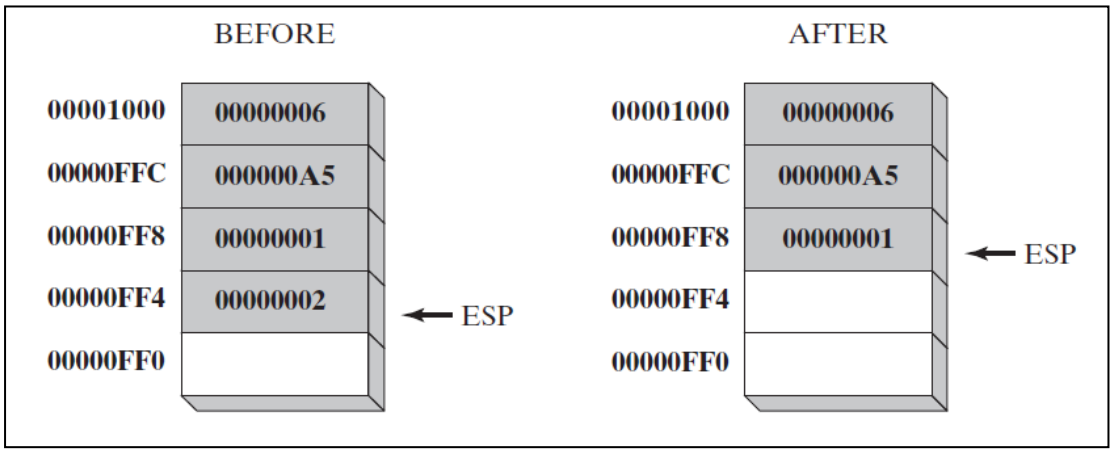

Figure (5) Popping a Value from the Runtime Stack.

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.

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*

Immediate values are always 32 bits in 32-bit mode. In real-address mode, immediate values default to 16 bits, unless the .386 processor (or higher) directive is used.

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*

Windows API Functions

With the release of Windows 95 by Microsoft in 1995, a new set of system calls was introduced. This set was call the Windows 32 bit Application Program Interface (Win32 API). The set consists of 32 bit system functions usable by any Win32 application.

The application programming interface (API) is a set of functions available from the operating system to support the programmer for building applications that can communicate with the operating system. The API is also available to ease and speed up the programming process. For example, several functions are available to create, read, and delete a file. The programmer is not concerned of how these functions operate; instead he or she can concentrate on tasks such as what data to write to the file, how to represent the data inside the file, when to query the file and when to delete it.

The purpose of the API was to provide a set of optimized system level operations allowing applications to run faster. The set is currently supported by all Windows platforms. All the API functions are stored in the following dynamic link libraries: Kernel32.dll, User32.dll, Gui32.dll and Advapi.dll. When an process is labeled a Win32 process it indicates that process uses the Win32 API. When a Win32 process is first executed it is analyzed by the operating system and the memory address of each Win32 API system functions that it may call is exported from a DLL and placed in an import address table (IAT). Each Win32 process has its own IAT and when the process makes an API system call, it looks up the function's address in the IAT and passes to that address any necessary parameters and the function proceeds with execution. When an system call is made it is usually from a process running in User mode, the called function is filtered through the operating system to its equivalent function in the Kernel of the operating system. Once in the kernel a service is usually requested to carry out the operation and the result filters back up the user application that originally made the call.

Coding in Win32 environment

As you may know Windows runs in protected mode and so our code will do so as well. Windows provides a virtual address space of theoretically 4GB of memory for every process. The use of this virtual memory allows the system to use the hard disk for swapping when the physical memory isn't enough. When you code, you code in a so called "flat" memory model. This means you don't need to care for the segment registers anymore and that makes the ASM coding a hell easier. You only need DWORD offsets when you address memory in Win32. In contrast to 16-bit systems like DOS and Win 3.1, 32-bit systems use DWORDs as offsets. Do not modify the segment registers or your program will destroyed with a chance of 99,99%. You will use the 32-bit registers much more than before (if you haven't used them already before). Let's take the LOOP instruction for example: Now the whole ECX will decrement and not only CX. Remember that! In protected mode (as the name suggests) the memory can be protected. So you may have read/write access, read only access or no access at all.

Many people tried to use interrupts in Win32 inline ASM code. But this doesn't work because you don't call REAL MODE interrupts. You would call the protected mode INTs and the good old DOS INTs aren't available anymore. Instead of INTs you need to use the Windows API. For a complete documentation take a look at Microsoft's MSDN (http://www.msdn.microsoft.com). At last you must remind that you will

code CASE SENSITIVE from now on! It's just like in C++. This is really important and so write MessageBoxA and not mESSAGEboXa for example!

Hello World! in Win32 ASM
==========

```
.386
.model flat

        extrn ExitProcess:proc
        extrn MessageBoxA:proc

.data

        msg_title   DB "MessageBox title",0
        msg_message DB "Hello World!",0

.code

start:
        push 0
        push offset msg_title
        push offset msg_message
        push 0
        call MessageBoxA

        push 0
        call ExitProcess

end start
```

And now the explanations.

- .386
- .model flat

This is obvious. The processor directive MUST be before the memory model and it must be at least a 386. The model directive says we use a flat memory model.

- extrn ExitProcess:proc
- extrn MessageBoxA:proc

Here we import 2 APIs from Kernel32.dll. Do not forget the :proc after the API names! The linker will give you no error, but your program will definitively destroyed!

- msg_title DB "MessageBox title",0

Note that almost every string in Windows is zero terminated.

- push 0

- push offset msg_title
- push offset msg_message
- push 0
- call MessageBoxA

At this time we call an API, the MessageBoxA API to be exactly.
See below for more info.

- push 0
- call ExitProcess

Yes, no INTs anymore. We use the ExitProcess API to quit. In this code example we used 0 as exit code.

Something more about APIs
==========

The MessageBoxA call might look a little strange to you. Let's see what the MSDN tells us about this API:

```
int MessageBox(HWND  hwndOwner,       // handle of owner window
               LPCTSTR  lpszText,     // address of text in message box
               LPCTSTR  lpszTitle,    // address of title of message box
               UINT  fuStyle          // style of message box
               );
```

In Win32, parameters aren't passed in registers anymore. Instead they are pushed on the stack. You really can assume that every parameter is DWORD size. If you code 'push 0' this instruction will push a DWORD 0 on the stack, not a WORD.
If you take a closer look you will notice that the parameters are pushed on the stack in the reverse order. So you have to push the last parameter as the first one and the first parameter as the last one.
Then simply call the API. The return value will always be in EAX.
Do not forget to save register values which you need before you call an API. In good old DOS times you knew exactly which registers will be destroyed by an INT call, but in the case of APIs you never know. So this is especially important in loops because ECX can be anything after the API call.

How to compile and link a Win32 program?
==========

For our 'hello world' program (hello.asm) we would compile it as the following:

```
tasm32 /ml hello.asm
tlink32 /Tpe /aa /c hello.obj,,,import32.lib
```

As you can see you need to use tasm32.exe and tlink32.exe and not the DOS versions .Let's discuss the parameters briefly:

/ml - compile case sensitive

/Tpe - set's output to PE (Portable EXE), /Tpd would be DLL

/aa - uses Windows API

/c - case sensitive linking

import32.lib - Normally, you specify only the import32.lib file for the linker. This is the standard file and it's used by the linker for our API references. Import32.lib contains all APIs from kernel32.dll, user32.dll and gdi32.dll (may be more, but at least these ones).