

## **An Introduction to Assembly**

Assembly language is specific to a processor's architecture - for example, a SPARC processor will use a different set of assembly instructions than a CPU using the x86 architecture, which will differ from the assembly instructions used when programming a PIC micro-controller. Since the most common architecture is x86, that is the instruction set we will be dealing with here. Before delving into the actual assembly instructions however, let's take a look at the CPU registers and process memory.

### **CPU Registers**

A processor takes data and instructions that are stored in memory and performs whatever calculations are required, then writes the output back into memory as applicable. However, the CPU needs a place to store the data it retrieves from memory while it calculates; this is where the registers come in. Registers are small segments of memory inside the CPU that are used for temporarily storing data; some have specific functions, others are just used for general data storage. In a 32-bit processor, each register can hold 32 bits of data; in a 64-bit processor, the registers can hold 64 bits of data. This paper will assume the classic 32-bit registers are being used, but even if you have a 64-bit CPU, as long as it is backwards compatible with 32-bit applications, all of the following information is still applicable.

There are many registers used by a processor, but we are concerned primarily with a group of registers called the general purpose registers. The general purpose registers are composed of:

EAX  
EBX  
ECX  
EDX  
ESI  
EDI  
ESP  
EBP  
EIP

The EAX register is called the accumulator, and is commonly used to hold the results of a calculation. If a function returns a value, this value will be placed in the EAX register so that the code that called the function can access the return value.

EBX is a pointer to the data segment, and ECX is normally used to count the number of iterations in a loop; EDX is used as an I/O pointer. It is important to note that while these are the suggested functions of the EAX, EBX, ECX and EDX registers, they are not restricted to these uses, with a few exceptions. For example, EAX can be used to hold data regardless of whether or not that data is the result of some calculation; however, if a function returns a value, that value will always be stored in the EAX register.

ESI and EDI are used to specify source and destination addresses respectively; they are most often used when copying strings from one memory address to another.

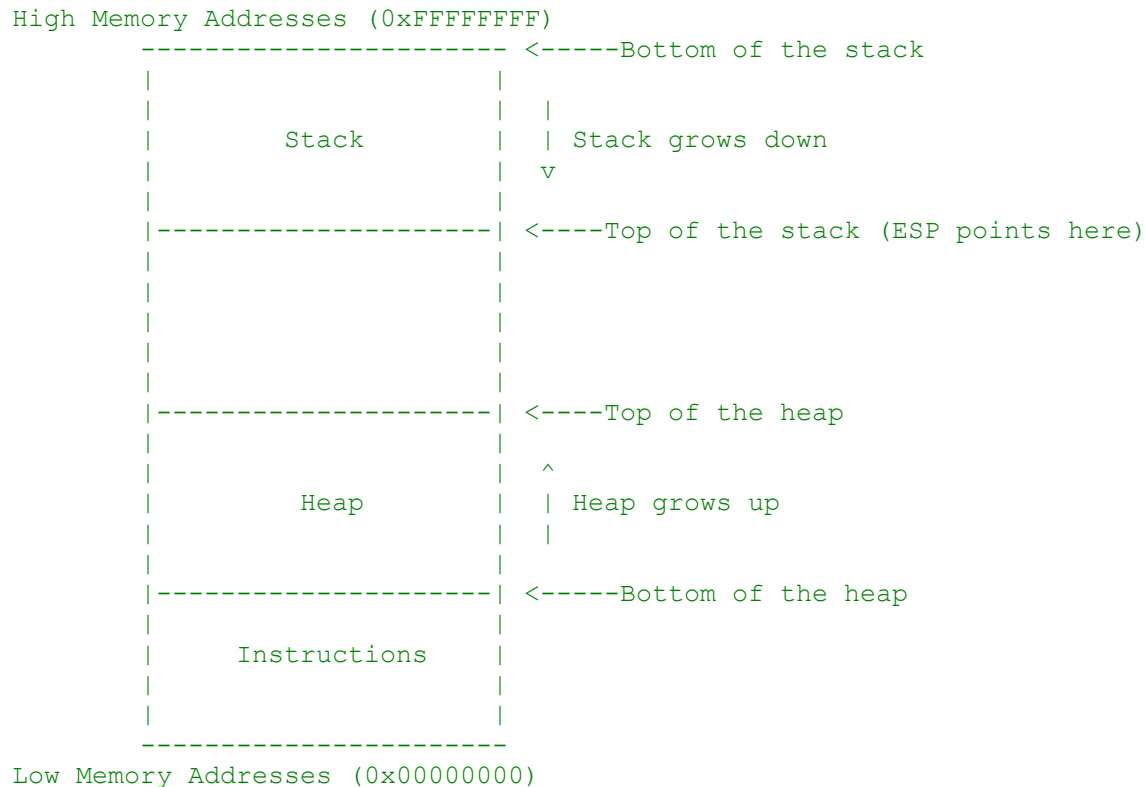
ESP is a stack register, called a stack pointer, that points to the top of the stack; EBP is also a stack register (called the base pointer), used to reference local variables and function arguments on the stack. The exact purpose and usage of the ESP and EBP registers will be clarified in the following sections.

EIP is the instruction pointer register - it controls program execution by pointing to the address of the next instruction to be executed. For example, if your program calls a function that is located at the address of 0x08ffff1d, the value stored in EIP will be changed to that address so that the CPU knows where to go in order to execute the first instruction of that function. Note that there is no way to *directly* control the value stored in EIP.

The 'E' at the beginning of each register name stands for Extended. When a register is referred to by its extended name, it indicates that all 32 bits of the register are being addressed. An interesting thing about registers is that they can be broken down into smaller subsets of themselves; the first sixteen bits of each register can be referenced by simply removing the 'E' from the name. For instance, if you wanted to only manipulate the first sixteen bits of the EAX register, you would refer to it as the AX register. Additionally, registers AX through DX can be further broken down into two eight bit parts. So, if you wanted to manipulate only the first eight bits (bits 0-7) of the AX register, you would refer to the register as AL; if you wanted to manipulate the last eight bits (bits 8-15) of the AX register, you would refer to the register as AH ('L' standing for Low and 'H' standing for High).

## Process Memory and the Stack

Often, a process will need to deal with more data than there are available registers. To remedy this, each process running in memory has what is referred to as a stack. The stack is simply an area of memory which the process uses to store data such as local variables, command line/function arguments, and return addresses. Before examining the stack in detail, let's take a look at how a process is generally arranged in memory:



As you can see, there are three main sections of memory:

1. Stack Section - Where the stack is located, stores local variables and function arguments.
2. Data Section - Where the heap is located, stores static and dynamic variables.
3. Code Section - Where the actual program instructions are located.

The stack section starts at the high memory addresses and grows downwards, towards the lower memory addresses; conversely, the data section (heap) starts at the lower memory addresses and grows upwards, towards the high memory addresses. Therefore, the stack and the heap grow towards each other as more variables are placed in each of those sections.

## Essential Assembly Instructions

Instruction	Example	Explanation
push	push eax	Pushes the value stored in EAX onto the stack
pop	pop eax	Pops a value off of the stack and stores it in EAX
call	call 0x08ffff01	Calls a function located at 0x08ffff01
mov	mov eax,0x1	Moves the value of 1 into the EAX register
sub	sub eax,0x1	Subtracts 1 from the value in the EAX register
add	add eax,0x1	Adds 1 to the value in the EAX register
inc	inc eax	Increases the value stored in EAX by one
dec	dec eax	Decreases the value stored in EAX by one
cmp	cmp eax,edx	Compare values in EAX and EDX; if equal set the zero flag* to 1
test	test eax,edx	Performs an AND operation on the values in EAX and EDX; if the result is zero, sets the zero flag to 1
jmp	jmp 0x08ffff01	Jump to the instruction located at 0x08ffff01
jnz	jnz 0x08ffff01	Jump if the zero flag is set to 1
jne	jne 0x08ffff01	Jump to 0x08ffff01 if a comparison is not equal
and	and eax,ebx	Performs a bitwise AND operation on the values stored in EAX and EBX; the result is saved in EAX
or	or eax,ebx	Performs a bitwise OR operation on the values stored in EAX and EBX; the result is saved in EAX
xor	xor eax,ebx	Performs a bitwise XOR operation on the values stored in EAX and EBX; the result is saved in EAX
leave	leave	Remove data from the stack before returning
ret	ret	Return to a parent function
nop	nop	No operation (a 'do nothing' instruction)

\*The zero flag (ZF) is a 1 bit indicator which records the result of a cmp or test instruction.

Each instruction performs one specific task, and can deal directly with registers, memory addresses, and the contents thereof. It is easiest to understand exactly what these functions are used for when seen in the context of a simple hello world program, which we will do a little bit later.

### **Assembly syntax**

There are two types of syntax used in assembly code: Intel and AT&T. Each display the same instructions, just a little bit differently (in the above examples I have used Intel syntax). The primary difference is that the source and destination operands are flip-flopped. Look at the differences in how the syntaxes display the instruction to move the number 1 into the EAX register :

Intel Syntax: *mov eax, 0x1*

AT&T Syntax: *mov \$0x1,%eax*

Besides the source (the number 1) and the destination (the EAX register) being reversed, the AT&T syntax also adds a percent sign in front of all register names and a dollar sign in front of hexadecimal numbers. Regardless of syntax however, it is still the same instruction.

You should be familiar with both syntaxes, as different disassemblers may use either one or the other syntax when disassembling a program. For my following examples I will be using the Intel syntax since it is a little easier to understand; however, the GNU debugger (gdb), which we will be using later in this paper, uses AT&T syntax. As such, I will be supplying both the AT&T and Intel versions of the sample programs in order to give exposure to both syntaxes. For more information on the differences between AT&T and Intel syntaxes, see the [gnu.org](http://gnu.org) link in the references section at the end of this paper.