Understanding x86 Assembly Code

To view the assembly code generated for your program in Visual Studio you need to set a break point at the location you want to examine, then run the application in debug-mode. When it stops at your break point, press Ctrl-F11.

Below is an example of what you see when you call function `foo()`. One breakpoint was set just before the function call and another at the start of the function implementation. The assembly code window shows you the source code and then the assembly code it generated:

```
  int varint = 5;
0097BAA9  mov          dword ptr [varint],5
  int test = foo(varint,3);
0097BAB0  push         3
0097BAB2  lea          eax,[varint]
0097BAB5  push         eax
0097BAB6  call         foo (096648Dh)
0097BABB  add          esp,8
0097BABE  mov          dword ptr [test],eax

int foo(int & a, int b) {
00F0A940  push         ebp
00F0A941  mov          ebp,esp
00F0A943  sub          esp,44h
00F0A946  push         ebx
00F0A947  push         esi
00F0A948  push         edi
  int orig = a;
00F0A949  mov          eax,dword ptr [a]
00F0A94C  mov          ecx,dword ptr [eax]
00F0A94E  mov          dword ptr [orig],ecx
  a = a + 1;
00F0A951  mov          eax,dword ptr [a]
00F0A954  mov          ecx,dword ptr [eax]
00F0A956  add          ecx,1
00F0A959  mov          edx,dword ptr [a]
00F0A95C  mov          dword ptr [edx],ecx
  return orig * b;
00F0A95E  mov          eax,dword ptr [orig]
00F0A961  imul         eax,dword ptr [b]
}
00F0A965  pop          edi
00F0A966  pop          esi
00F0A967  pop          ebx
00F0A968  mov          esp,ebp
```

```
00F0A96A  pop         ebp
00F0A96B  ret
```

Instructions are shown in AT&T syntax: mnemonic source, destination.

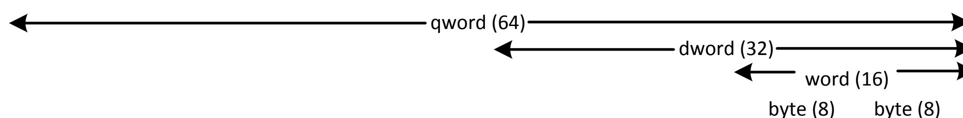
Source and destination are operands and can be immediate values, registers, memory addresses, or labels. Immediate values are constants, and on some systems will be prefixed by a $. For instance, $0x5 represents the number 5 in hexadecimal. Register names may be prefixed by a %.

Memory addresses are referenced by `name`. For any `name`, `name` is the address of the memory and `[name]` is the contents of the memory.

Values have a specified size. A `byte` is 8-bits, a `word` is 16 bits, a dword is 32 bits. In assembler these sizes are specified by `byte ptr`, `word ptr` and `dword ptr`.

Registers are data storage locations within the CPU. The *width* of a CPU's registers is defined by its architecture. So if you have a 64-bit CPU, your registers will be 64 bits wide.

The x86 family has 8 registers whose prefix depends on the architecture as shown below:



RAX	**EAX**	**Return Value AX**	AH	AL
RBX	**EBX**	*	BH	BL
RCX	**ECX**	**CX**	CH	CL
RDX	**EDX**	**DX**	DH	DL
RSI	**ESI**	*		
RDI	**EDI**	*		
RSP	**ESP**	**Stack Pointer**		
RBP	**EBP**	* **Base Pointer**		
R8-R15				

Note: the * registers must be restored on return from a function call. See below.

In addition to these registers there will be instruction address registers and status registers.

For backwards compatibility, any register can be accessed as a narrower type by using a different register prefix.

The registers of particular interest are:

~`ax` – The return value from a function is placed here.

~`bp` - Base Pointer for the current function from which arguments and local variables are offset.

~`sp` - Stack Pointer – on entry to a function, the base-address for the function.

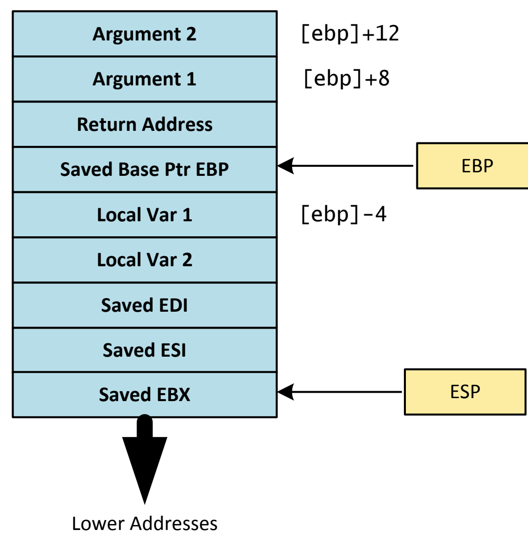
On return from a function call ~`bp` must be restored to its previous state so that the base-address is not lost for the calling context. In addition, the C/C++ compiler expects any function to return with ~`bx`, ~`si` and ~`di` in the same state that it got them. It can use them internally, but must restore

their values before returning. These registers, marked with * in the diagram above, are known as "untouchables".

64-bit Intel processors add another 8 64-bit registers, R8-R15, sixteen 128-bit registers, XMM0-XMM15 and eight 80-bit floating-point registers FPR0-FPR7

A Function Call

When a function call is encountered, the assembler inserts instructions to push the arguments onto the stack (in reverse order) and then the return address for the calling instruction. The stack pointer now points at the next available stack location. These operations are wrapped in the assembler `call` instruction. It is then the function's responsibility to save the current base-pointer (ebp) and set the base pointer to the current stack pointer, as the base for this function. It must also save the other untouchable registers (edi,esi,ebx) by pushing these onto the stack. But before pushing the untouchable registers it will decrement the stack pointer to leave room for the local variables. All these will be accessed with offsets to the base-pointer.



Stack use at a function call

Understanding the code

```
    int varint = 5;
  0097B AA9  mov         dword ptr [varint],5
```

`mov` assigns the value 5, as a `dword`, to the address pointed to by the name `varint`.

```
  0097BAB0  push        3
  0097BAB2  lea         eax,[varint]
  0097BAB5  push        eax
  0097BAB6  call        foo (096648Dh)
```

The arguments for the function call are pushed onto the stack in reverse order. The first argument is a reference, so the destination address is first obtained by `lea` (Load Effective Address) which is placed in `eax`. That address is then pushed onto the stack. Finally the function is called.

```
int foo(int & a, int b) {
00F0A940  push        ebp
00F0A941  mov         ebp,esp
00F0A943  sub         esp,44h
00F0A946  push        ebx
00F0A947  push        esi
00F0A948  push        edi
```

Upon entering a function, `esp` points to the base address for our function. The arguments and local variables are at known offsets from this pointer. It becomes the base-pointer inside our function. Before we copy the `esp` into `esb` we must preserve the old `esb` on the stack for later reinstating.

Upon entering a function, the value of the "untouchable" registers must be stored. The `push` instructions save these registers to the stack.

`push` decrements the stack pointer, `esp`, and writes its operand to the stack.

`mov` copies the second argument into the first, here the stack pointer (`esp`) is copied to the base pointer register. `ebp` now points to the base of the function's stack frame.

`sub` (subtract) adjusts the stack pointer to make room for pushing the other "untouchable" registers and local variables onto the stack.

```
    int orig = a;
00F0A949  mov         eax,dword ptr [a]
00F0A94C  mov         ecx,dword ptr [eax]
00F0A94E  mov         dword ptr [orig],ecx
```

The argument is now moved to register `eax`, specifying its size as `dword ptr`.

```
dword ptr [a]  says "'a' is a dword pointer; give me the value it
points to"
```

But because the argument was passed by reference, we need to follow the argument to its address and get the value there. The value is placed in `ecx`.

Finally, the value is copied onto the stack at the location reserved for orig.

```
  a = a + 1;
00F0A951  mov         eax,dword ptr [a]
00F0A954  mov         ecx,dword ptr [eax]
00F0A956  add         ecx,1
00F0A959  mov         edx,dword ptr [a]
00F0A95C  mov         dword ptr [edx],ecx
```

Now the argument is dereferenced again and placed in `ecx`, then 1 is added to the register.

```
00F0A959  mov         edx,dword ptr [a]
00F0A95C  mov         dword ptr [edx],ecx
```

Here the result is written back to the argument address because the argument was a referece. `edx` is used to hold the address in the first `mov`. The second `mov` writes the result from `ecx` to the address held in `edx`.

```
  return orig * b;
00F0A95E  mov         eax,dword ptr [orig]
00F0A961  imul        eax,dword ptr [b]
}
```

At the return point the local variable on the stack at `orig` is moved into `eax`. It is then integer-multiplied (`imul`) by argument `b`. The caller of the function recovers the return result from register `eax`.

Now the final clean-up (epilogue):

```
00F0A965  pop         edi
00F0A966  pop         esi
00F0A967  pop         ebx
00F0A968  mov         esp,ebp
00F0A96A  pop         ebp
00F0A96B  ret
```

The untouchable registers are restored from the stack and the local variables discarded by moving the stack pointer back to the base with `mov esp,ebp`