Lamby's crash course in ASM for nerdy gamers

So WTH is this?

This is just a quick overview of using MASM32 to write ASM code. I have written it to go along with a collaborative project I am starting for the more nerdy types who play good old Warcraft II on the server at war2.ru The project is to create an editor for Warcraft II .PUD files in ASM, however the actual goal of the project is to teach people about PE files, and how programs look when they are decompiled and why, to increase the human resources we have to add new features to our beloved 20 year old game.

This is entirely based on my own knowledge and understanding so any or all of it may be completely wrong. I've just written it all down as I thought of it over a 2 day period in between doing normal life-related things, so I may have missed important facts or entire topics. It contains personal opinions and approaches that may be deemed inappropriate or completely messed up by the establishment. However, I'm the type of maniac who doesn't really care about being seen to be wrong, in fact if I am wrong about something I'd rather it be public so somebody will correct me than just continue being ignorant in a sealed box with my ego. The upside is that you all get the quick and easy version of how to do this stuff without all the bullshit attached, and if I'm wrong about something you can let me know.

I have posted the source code I have written as a starting point for the PUD editor project HERE. Its currently 400 lines, but about 4 fifths of that is comments mostly related to the internal structure of Windows GUI applications, there's actually not very much code there. At the moment it can be compiled and executed but it does nothing more than create a blank window with a menu that has 2 options. “About” will display a small about message, and “Exit” will close the window. Hardly the worlds most impressive app at this point lol, but it has the framework we will need for a Windows GUI application fairly well set out.

The second update is HERE. If has routines for loading/saving PUD files, an internal pud object container, a module framework, a template for creating new modules and a simple module to edit the map description.

hf you crazy people =D

So... ASM.. what's it all about?

At it's lowest level, ASM uses *mnemonics* to represent the actual opcodes that are being executed by the CPU. A "mnemonic" just means something that helps you remember something else, so for the opcode that *moves* a value from one place to another we use the mnemonic "**`mov`**". To put values onto the stack or get them back off again you use "**`push`**" and "**`pop`**", these are just handy short WORDs that are easy for us to associate with the actual mechanism of the various opcodes.

They are not "keywords", "directives" or language constructs like the syntax in most languages. Something like the **"`for`"** which declares a loop variable in many languages, is the trigger to do a number of different things; allocate counter storage, define the bounds of the loop, evaluate and translate the exit conditions into some kind of test the CPU can understand etc.

Depending on the language such a declaration can generate anywhere between maybe a dozen up to literally *hundreds* of instructions. But an ASM *mnemonic* it just an easy to remember place-holder for a single instruction, so we are for the mot part just working directly with the CPU.

So how does the CPU work?

WELL I'M SO GLAD YOU ASKED **;**D

The CPU has a number of internal storage containers called "*Registers*"
The CPU can get values from, or write them to, *memory*.
The CPU also has access to a special area of memory called "*the stack*".

Getting the hang of these 3 things and a handful of common instructions is pretty much all you need to get started with ASM.

The most complicated things we will be doing will all be related to interacting with Windows. There is, as always, a certain amount of inherent complication that goes with the territory when you are interacting with a multi-tasking OS, but this is no worse in ASM than any other serious language. Learning to effectively program with Windows is a lot harder than learning ASM

To access the OS we need to know how to call API functions from the system DLLs, which is easy as pie, however understanding how you are then expected to use the API functions you are calling can be a bit of a head-fuck at times, but we won't need to use anything too tricky for this project, and again this is a Windows issue, not an ASM issue.

But enough of that..... back to the CPU.

The Registers

The registers are the CPU's hands. Any value it needs to hold on to, or do anything clever with, it needs to have it in one of its registers.

The registers are physically located within the CPU. Doing things with numbers that are already in registers is much faster than values stored in memory, where they have to be fetched into the CPU and then returned afterwards.

x86 32bit processors have the following 8 registers:

A, B, C, D,
SI, DI,
also SP and BP

Plus there is IP and FLAGS but these two are not directly accessed by user code, so you don't need to worry about them to start off with, *beyond the basic idea that there are "**FLAGS**" somewhere that are set when certain things happen with numbers*.. i.e. when a calculation ends up equaling 0, then 'ZF' – the "zero flag" is set, but you don't really need to understand the whole mechanism at this point.

SP and BP are used for defining the stack, the stack frame and local variable allocation, but this all happens behind the scenes, so for now you can forget about those 2 as well. *People who want to learn about software reverse engineering will eventually have to get very chummy with these 2 but for getting started, just ignore them.*

It should be noted that for the purposes of this project:

a **BYTE** is an 8 bit value
a **WORD** is a 16 bit value (= 2 BYTEs)
a **DWORD** is a 32 bit value (= 4 BYTEs, = 2 WORDs)

The next 3 pages have a little introduction to registers thing (with pictures :) that explains the layout of the registers and how the various parts are accessed and why.

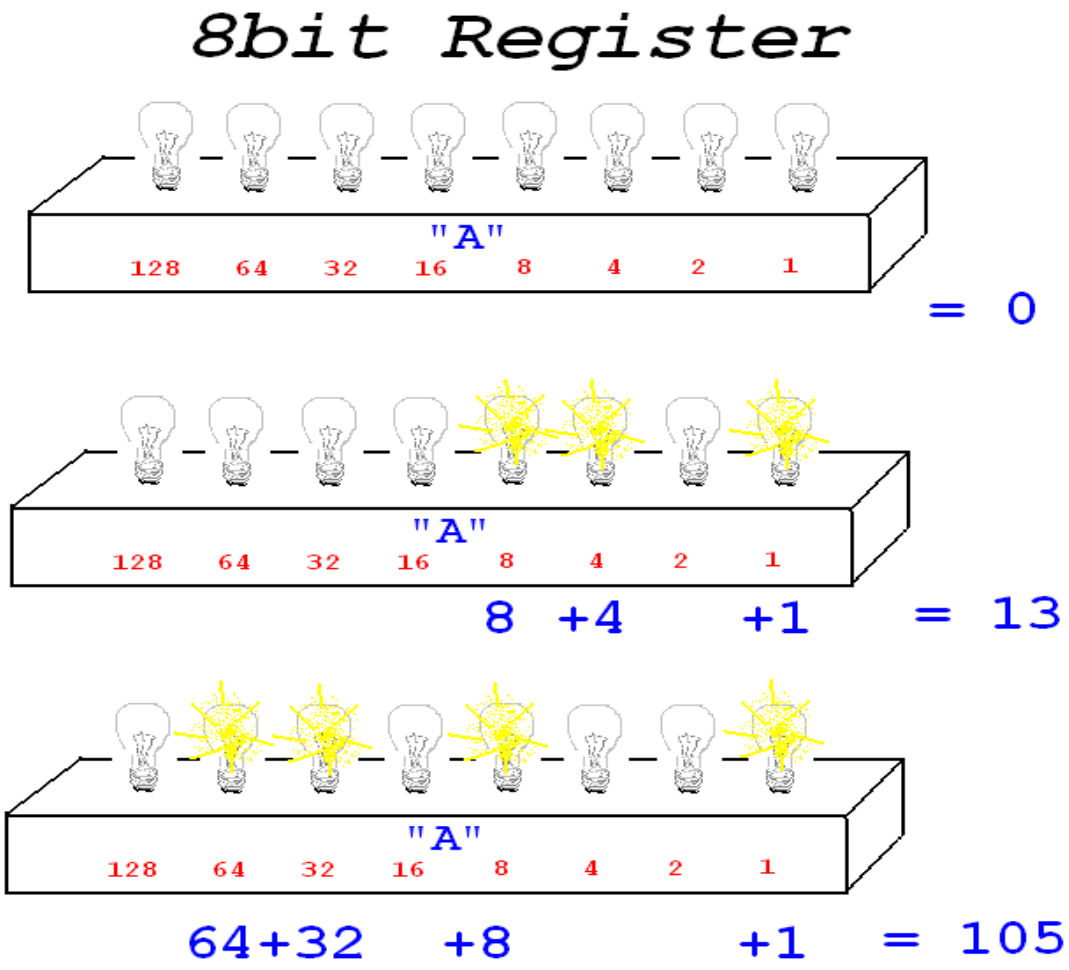
| *Just have a read through and don't worry if you don't instantly pick up on everything there. You don't need to fully understand or remember it all just get the basic idea and you can come back to it later when you start using them....*

....but take note that there will be absolutely NO making fun of Lamby's crappy illustrations, and anyone found to be sniggering behind their hand will be sent to the naughty chair IMMEDIATELY!

Introduction to registers

Lets take a quick trip back to the '70s when 8bit processors were the norm, and if you were lucky you had 4 registers to play with; A, B, C, and D.

Here's the A register. It is a series of 8 transistors that can each be switched ON or OFF... like a line of 8 light bulbs. These are the bits that represent a number between 0 and 255.



For the really new: its a base2 number; it has two digits '0' and '1'. … or 'Off' and 'On'
… or "False' and 'True'
call them what you will, it all adds up the same...

People usually use base10, which has 10 digits 0,1,2,3,4,5,6,7,8 and 9

Programmers often use base16 (aka hexadecimal)
it has 16 digits; 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E and F

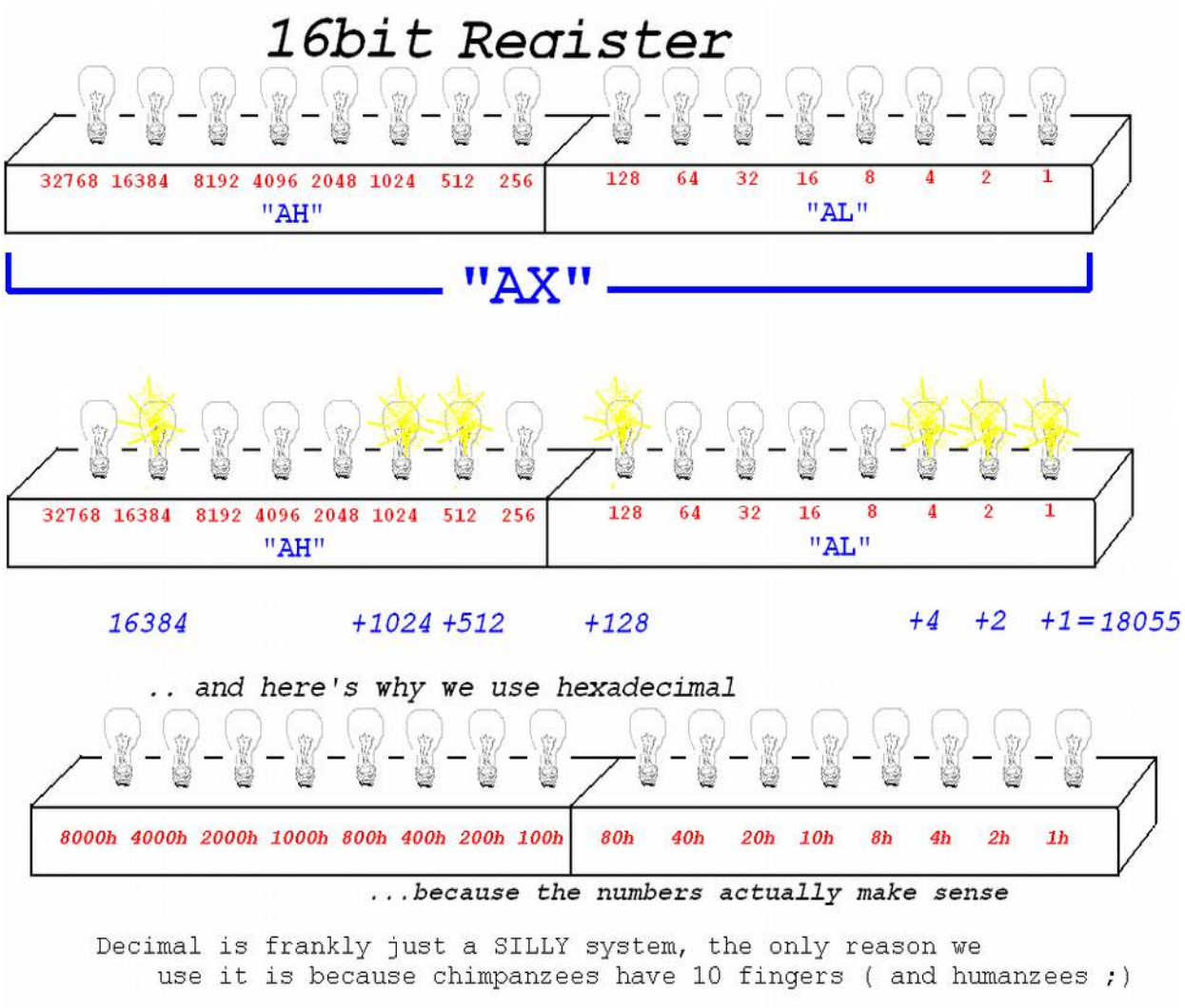
So wind forward a decade or so and OMW look what they have now!

... 16 BIT PROCESSORS!!!

Now everything had 16 bits, including our '**A**' register.

However there were many, many things that still needed to be stored/processed 1 BYTE (8bits) at a time, so it made little sense to force the user to load 16 bits into the register every time, whether they needed them or not.

Because of this, the new 16bit register, called "**AX**", was also split into two 8bit registers ('**AL**' for the low BYTE and '**AH**' for the high BYTE) which could also be accessed independently.



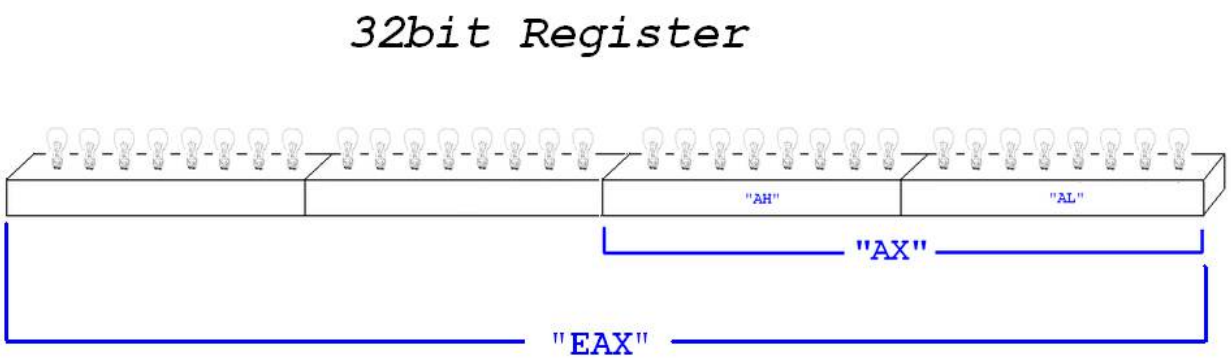
Also around this time, the boffins making the chips decided that a couple more registers would be very handy to do things with; so we got **SI** and **DI**. The *source index* and *destination index* registers were designed mainly just to hold memory offsets, so they weren't split into high and low BYTE registers, and could only be accessed 16bits at a time, which was fine considering their purpose, and that we still had 8 BYTE sized registers to use, being: **AL, AH, BL, BH, CL, CH, DL** and **DH**.... So NP there =D

A 16bit number can hold any value from 0 thru 65535

But wait! There's more! … who would have thought of it? **32 bit processors!**
What will they think of next?

The old 8bit '**A**' register got an extra BYTE attached to it, and became '**AX**', So now the 16bit '**AX**' register got extended to become the 32bit register '**EAX**'

You can still access the low WORD of EAX as AX, and you can still access the low and high BYTEs of AX as AL and AH. There is however, no way to directly access the high WORD in EAX, but in practice, most times this really isn't necessary. If you really want to access that WORD individually there are ways to do it, but usually if you are working with a WORD sized value you would have it in AX anyway (or BX or whatever).




```
EAX =  Extended register - 32 bits
 AX =  Low  Word of EAX  - 16 bits
 AH =   High Byte of AX  -  8 bits
 AL =   Low  Byte of AX  -  8 bits
```

So now the first four registers can be accessed in 4 ways each (low BYTE, high BYTE, low WORD or full DWORD), and we also have these SI and DI registers which were 16 bit only, but have now been extended to become then 32bit registers ESI and EDI. The low WORD of these can still be accessed as SI and DI, but not individual BYTEs.

Similarly, the SP and BP registers have become ESP and EBP, their low WORDs can still be accessed via SP and BP however in practice they rarely, if ever are. Of course the FLAGS register is now 32 bit, and the instruction pointer has become EIP, but you are unlikely to ever see one of those in captivity.

Of course the x86-64 architecture has not only again doubled the size or the registers making them RAX, RBX etc., but also doubled the number of usable registers from 8 to 16, but that is another story, we are only using the 32 bit instruction set here.

So we have ended up with these registers:

NAME	*Access full **32bits** as:*	*Access low **16bits** as:*	*Access 2nd **8bits** as:*	*Access low **8bits** as:*
Accumulator	`eax`	`ax`	`ah`	`al`
Data	`ebx`	`bx`	`bh`	`bl`
Counter	`ecx`	`cx`	`ch`	`cl`
I/O	`edx`	`dx`	`dh`	`dl`
Source Index	`esi`	`si`	-	-
Destination Index	`edi`	`di`	-	-
Stack Pointer	`esp`	`sp`	-	-
Base Pointer	`ebp`	`bp`	-	-

***I have been presenting registers in ALL CAPS, just to make the description clearer in amoungst piles of text, but when coding I prefer them in lowercase as they are in this table, the same applies to mnemonics.**

All the registers hold 32bits of data each, and (with the exception of FLAGS and IP) they can all be used to store a 32bit number at any given time (best not to mess with ESP or EBP either), however they each also have their own individual tasks that they are designed for.

The 'A' register or 'EAX' as we now know it, is also called ***"The Accumulator"***. It is the right arm of the CPU and does the majority of the heavy lifting. Some instructions only operate on the accumulator, such as 'MUL' (multiply) or 'DIV' (divide). Also, by convention, the EAX register is used to store the return value from any function or procedure that returns one.

The 'C' register is the "counter", and is specialized to perform tasks such as loop-counting.

B is designated "data" and D is "I/O" however, in practice these two are mainly used as general purpose workhorses and aren't particularly specialised as per those names. The only notable thing about these two is that D becomes relevant for MUL and DIV overflow/remainder.

SI and DI are "source index" and "destination index". These are used specifically (along with the counter) if any of the 'string' instructions are used with a repeat prefix, but more on that later. Even if not being used for one of the specific instructions that require them in these roles, ESI and EDI are conveniently named to make code nice and readable if they are used for those purposes when carrying out other tasks.

SP and BP define the stack, they are the "stack pointer" and the "base pointer".

ACCESSING MEMORY

So now we know what our registers are and how they are named we can start using them, and we need to use them because one of the basic rules of x86 programming is: ***You can't move a value directly from one memory location to another***, you have to first move the value to a register, and then move it from the register to the destination location. So if we are accessing memory, we are using a register.

**The exception to this rule is you can push the contents of a memory location onto the stack or pop a value off the stack to a location. As the stack is a region of memory this is an exception, however stack memory access is extremely specialised. More on this later.*

OK, first lets try out a register, how about good old EAX?

```
mov eax, 3
```

Wow, now EAX contains the value 3! I'm excited.
Lets get a valid address in there..... hmm, say perhaps 0x00401010

WHY THIS ADDRESS?
All standard .exe files load at 0x00400000 This is the image base, its taken from the IMAGE_OPTIONAL_HEADER structure in the PE file. Most other addresses in the PE headers are virtual addresses (VA) and are relative to this address. Two relevant VAs are BaseOfCode and BaseOfData, these are added to the image base to find the addresses of the executable code and the image resident data. So for instance, WC2 has these at 0x00001000 and 0x00090000 so the code starts from address 0x401000 and the data from 0x490000 ....anyway 0x00401010 is a safe bet for now

```
mov eax, 401010h
```
EAX now holds an *address*

So what's at that address?

```
mov eax,[eax]
```
eax now holds the 4 bytes (DWORD) at 0x00401010, whatever they are.

This is called ***dereferencing.*** Taking a pointer and replacing it with the value at the address it points to. The square brackets "[ ]" indicate that their contents should be *dereferenced.*

The same could be achieved with:
```
mov eax, [401010h]
```

The x86 dereferencing mechanism is quite robust; it can take two registers, an immediate value and a multiplier.
So : **`mov ebx, [eax+200h+edi*4]`** is valid and the CPU will calculate (edi*4)+eax+0x200 when that instruction is executed, then put the DWORD *at that address* in EBX.

'200h' is an ***immediate value.*** This just means you are giving the processor a number, not telling it to get the number from a register or from memory, its just a number.

The multiplier ('*4' here) is limited to one of 2, 4, 8 (and maybe 16 … can't remember lol)

THE STACK

The stack is a special region of memory that is used by the CPU to temporarily store numbers. In its normal usage it is a simple FILO buffer (first in last out). The simplest and most common way to access the stack is with the PUSH and POP instructions. PUSH puts a value on the top of the stack, and POP takes the top value off the stack.

```
mov  eax, 3       ; eax now holds the value '3'
push eax          ; '3' is now on the top of the stack
pop  ebx          ; ebx now also holds the value '3'
                  ;        (and the '3' has gone from the top of the stack)
```

So what can we do with the stack?

The most frequent use of the stack is for calling suroutines (procedures), it is also commonly used to creat local variables, but more on that stuff later. Otherwise, one very common use for the stack is just to temporarily save the value in registers. For the most part, you ony have 6 registers to use, and if you need to use more than 6 numbers for whatever you are doing it is easy just to push some of them onto the stack for a while. i.e.:

```
; im half way through doing something, but I need to
; calculate something else before I continue...

push ebx   ; put the value in ebx on top of the stack
push ecx   ; put the value in ecx on top of the stack (the ebx value is now 2nd)

; now I can use EBX and ECX to do something else
; ….....
; …( other code )....
; ........

pop ecx       ; get the ecx value back from the stack
pop ebx       ; get the ebx value back from the stack
              ;(note they are popped in the reverse order to how they were pushed)

;now I have the original values I was using back again.
```

The stack is also handy for moving values from one memory location to another:

```
mov esi, my_source_address
mov edi, my_destination_address

push [esi]
pop  [edi]
```

This is not the most efficiant way to move large amounts of data, but is handy for retrieving individual values. Note that EDI and ESI contain pointers so they are *dereferenced* with the [ ] square brackets. Just putting "`push esi`" here would push the address in ESI onto the stack, not the value that is in memory at that address.

SUBROUTINES

*Suroutines are sections of code that perform a specific task. They can be "called" many times from multiple different places in a program, whenever you need to perform that specific task. Functions imported from the Windows API or other dll files are subroutines, as are procedures you define with the "**proc**" directive or any other section of code that you access with a "**call**" instruction and ends with a "**ret**" (return) instruction.*

Lets make a simple function that adds two numbers together, we'll call it "MyAdd" :

```
MyAdd proc first_number :DWORD, second_number :DWORD

    mov eax, first_number
    mov ebx, second_number
    add eax,ebx
    ret

MyAdd endp
```

We give this function two numbers and it adds them together. The numbers we give it are called ***arguments.*** When it has finished the result will be in EAX, this is called the ***return value***. Supplying arguments to a function is called ***passing.*** So you can say, "you ***pass*** the MyAdd function 2 integers and it ***returns*** the sum of those integers"

Arguments are passed to functions by pushing them onto the stack

We can use this function like this:

```
mov ebx, 3
mov ecx, 5

  push ecx          ; push second argument
  push ebx          ; push first argument
  call MyAdd        ; call "MyAdd"

; da-da!      Eax now holds 8
```

*NOTE: Arguments are pushed onto the stack in reverse order, so the first argument, which is pushed *last*, ends up on the top of the stack.

So EAX has our answer '8', *but what is in EBX and ECX?* 3 and 5? No. Actually now EBX and ECX both hold the value 5. When we call MyAdd it puts the argument "**second_number**" in EBX which is the value that was passed by the **push ecx.** So our MyAdd function is a bit naughty, it doesn't leave everything as it found it. It can be said that this function is *destructive* for EBX, because it overwrites the value in EBX.

If we wanted to keep the initial values intact, then knowing that we are calling a function that is destructive to EBX we could take precautions like this:

```
  push ebx    ; save ebx

    push ecx      ; push second argument
    push ebx      ; push first argument
    call MyAdd    ; call "MyAdd"

  pop ebx     ; restore ebx
```

This will preserve the value in EBX by temporarily storing it on the stack during the call to MyAdd. The two arguments that are pushed onto the stack are removed when the the function returns, leaving our saved EBX value at the top.

As we are calling our own function, another option here would be to give our MyAdd function better manners, and have it not send EBX back all messed up, like this:

```
MyAdd proc first_number :DWORD, second_number :DWORD

  push ebx

    mov eax, first_number
    mov ebx, second_number
    add eax,ebx

  pop ebx

  ret
MyAdd endp
```

MyAdd also changes the value in EAX, however this is to be expected as EAX is the accepted location for returning any value from a function. *Any time you call anything you should not expect to get EAX back unscathed.*

*** When writing procs DON'T FORGET to put a **`ret`** instruction at the end.***
If you use ASM enough you will end up doing this then searching for the bug for ages before you realise and slap yourself on the forehead. It's the price you pay for not using high level baby-sitting. Don't say I didn't warn you ;)

The **`call`** instruction actually **`push`**es the address of the next instruction following it onto the stack, then jumps to the target location. The **`ret`** instruction **`pop`**s the top value off the stack then jumps to that location, hence it is very important to have the stack balanced before attempting to return from a call

#ForTheRecord

If you really wanted to use the MyAdd proc it would be much better written as:

```
MyAdd proc first_number :DWORD, second_number :DWORD
  mov eax, first_number
  add eax, second_number
  ret
MyAdd endp
```

Its just been presented as a device to demonstrate a couple of points.

API calls

Windows API functions operate in exactly the same way as MASM procs. In fact many of them were built using MASM procs so you can call them just like we called "MyAdd".

Lets allocate some memory for a buffer using **GlobalAlloc** from kernel32.dll.

From MSDN:

```
HGLOBAL WINAPI GlobalAlloc(
  _In_ UINT   uFlags,
  _In_ SIZE_T dwBytes
);
```

Anybody who knows a bit of C will understand this, but for those who may not; it describes the usage of the **GlobalAlloc** API function. It says that you pass it two arguments; "**uFlags**", and "**dwBytes**" and it returns a type "**HGLOBAL**", which is a global handle. Often a handle is just a pointer to something. Maybe a pointer to a structure that defines an object, in this case its just a pointer to the memory that the OS has allocated for us. In a 32bit environment pointers are DWORDs. The two arguments are said to be of types "**UINT**" and "**SIZE_T**". These, like almost every single other type def in the entire 32bit API just mean "DWORD" and "DWORD".

Types are mainly a form of error checking that high level languages use to restrict what the programmer can do with certain variables: a form of baby-sitting. When you get down to the nitty-gritty there are very few types that actually matter beyond defining the physical number of bits that are used to store a value. A notable exception would be floating point values destined for the FPU but the rest are basically shit (the C programmers start a riot, Lamby tells them all to STFU learn something). As long as you are passing the right values to a function it does not matter a rats ass what type some esoteric C header somewhere declared it as (Welcome to ASM woot!). "but what about signs?"A signed integer is just another DWORD, if the function you are calling is using bit31 as a sign bit then send it an appropriate DWORD value, which in most cases is exactly the same DWORD value as an unsigned integer. Sure we have to keep an eye out for these, but we're all grown up now; we can handle it!

For allocating a chunk of normal memory the uFlags argument should be GMEM_FIXED which just equals 0. The other argument is just the amount of memory you want allocated, so to have the API allocate 0x1000 bytes of memory you would just do this:

```
push 1000h              ; dwBytes arg
push 0                  ; uFlags arg
call GlobalAlloc        ; call the function
```

The return value is, of course, in EAX. Anybody who thinks that this is somehow easier in C or delphi or whatever needs their head read, or more likely just doesn't know how to program in assembly. Actually you can also use the invoke syntax which looks even neater and will compile to the same thing: "`invoke GlobalAlloc,0,100h`" however I am not using invoke for this project because I want people to understand what they will see when reverse engineering things. Invoke is also restrictive because sometimes its handy to push arguments to functions onto the stack some time before you actually call them.

If you are using API calls you will also have to include the appropriate .inc and .lib files, so here as we are using Kernel32.dll we would need:

```
include kernel32.inc
includelib kernel32.lib
```

LABELS

Labels just define a place in the .exe file, which is an address. When compiled labels are converted into the address of the location they define.

There are a number of ways to define labels. We have already dealt with one of them, **`proc`** names are just labels. When your code is compiled they will just be converted to the address of that chunk of code. You can also define a label at any time by typing a name with a colon '**`:`**' at the end like this:

```
MyLabel:
```

then somewhere else you can put:

```
call MyLabel
```

or

```
jmp MyLabel
```

or even

```
mov eax, OFFSET MyLabel
```

Which will put the address of the place in your code where you put the label into EAX
OFFSET tells the compiler that the instruction wants the actual address of a label. Normally when supplied with a label the compiler will generate an instruction to dereference the the location, except for program flow instructions (call, jmp etc..)

Labels can also be defined when you declare data, usually in the .data or .data? sections...

```
hWindow  dd 0
```

"**`hWindow`**" may seem like a variable name, and in the end you often end up using them like that, but it is actually a label that represents an address, in this case the location where that particular DWORD is stored.

MASM also has a handy syntax for declaring trivial labels that are only needed for minor flow control purposes. This label is a special case:

```
@@:
```

You can use the **`@@:`** label as often as you like, and access it using either **`@B`** or **`@F`**.

`jmp @B` means jump back to the previous **`@@:`** label.
`jmp @F` means jump forward to the next **`@@:`** label

Comparison and Flow Control

*One of the central concepts of programming is **conditional branching**, which most people would be familiar with as looking something like:*

"IF (some condition) THEN (do something) ELSE (other things)"

The CPU actually performs conditional branching using one of the set of "conditional jump" instructions. Any instruction starting with 'j' is a jump instruction. The basic instruction is "jmp" which is an unconditional jump, it just means *"go HERE and keep executing code"*. The other jump instructions are all conditional jumps, they mean "*IF (some condition) THEN go HERE and keep executing code"*. The condition is the state of one or more of 4 flags, being the Zero, Parity, Sign and Overflow flags. It is not necessary to start off with to understand the exact conditions and triggers for each flag. Usually immediately before the conditional jump instruction is an instruction that evaluates a particular value, so the program will either jump or not based on that evaluation. Often the evaluation will be either a "test" or "cmp" (compare) instruction. EXAMPLE:

```
cmp eax, 0FFh
ja @F
  ;Do Something With a Byte
  jmp the_end
@@:
; Do Something Else
the_end:
```

First it compares the value in EAX with the value 0xFF (255), the cmp instruction will set whatever flags are appropriate for this comparison depending on the value in EAX.

Then we have a **`ja`** (*jump if above*) instruction. What this means is the the program will jump to the target location if EAX > 0xFF. When you are starting it is not necessary to know that the conditions for the **`ja`** conditional jump are that both the CARRY and ZERO flags are unset or why this is the case. You just need to understand that you are comparing two values and jumping if Val1 > Val2. The logic of this jump could be reversed by using a **`jbe`** instruction (*jump if below or equal*). Most of these instructions have multiple aliases i.e. a **`jbe`** instruction is the same as a **`jna`** (*jump if not above*), they are just two different names for the same thing.

Common Conditional Jump Mnemonics

```
ja      jump if above
jae     jump if above or equal
jb      jump if below
jbe     jump if below or equal
je      jump if equal
jne     jump if not equal
jg      jump if greater           (signed)
jge     jump if greater or equal  (signed)
jl      jump if less              (signed)
jle     jump if less or equal     (signed)
jz      jump if zero
jnz     jump if not zero
```

(You can find a more complete list in opcodes.chm in your masm32\help directory)

Data Declaration

In ASM you can declare the actual values you want in your file anywhere you want them. This is usually done in the **.data** *section, but you can actually do it in the* **.code** *section also.*

Most commonly you will see data declared using "**db**", "**dw**" or "**dd**". Which stand for "declare BYTE", "declare WORD" and "declare DWORD".

EXAMPLES:

```
.data
db  1
```

Puts a 0x01 BYTE at the start of the **.data** section.

```
.data
dd 1
```

Puts a DWORD 0x00000001 at the start of the **.data** section.

```
.data
db 1,2,3,0
```

Puts the BYTEs 0x01, 0x02, 0x03, 0x00 at the start of the **.data** section.

```
.data
dd 30201h
```

Puts a DWORD 0x00030201 at the start of the **.data** section.
**absolutely identical results to the previous example.*

```
.data
dw 201h, 3h
```

Puts the WORDs 0x0201, 0x0003 at the start of the **.data** section.
**absolutely identical results to the previous 2 examples.*

```
.data
db "hello",0
```

Puts the BYTEs 0x68, 0x65, 0x6C, 0x6C, 0x6F, 0x00 at the start of the **.data** section.

```
.data
hWindow dd 0
```

Puts DWORD 0 at the start of the **.data** section., and labels this location as "hWindow"

The **DUP** directive:

DUP just tells the compiler to *duplicate* the declaration the specified number of times.

```
.data
MyBuffer db 100h dup (0)
```

Puts 256 zero BYTEs at the start of the **.data** section., and labels the location of the first BYTE as "MyBuffer"

Uninitialised Data

In addition to the **.data** *section there is also a* **.data?** *section. The* **.data?** *section is a virtual section, it does not exist in the .exe file and it cannot contain initialised data.*

Its size and location are defined in the .exe there is no definition of what this section contains. The .data? section is created in the process' address space when the application is mapped by the system loader. In effect it is just a big pile of zeros, but its contents are technically undefined until the app writes something to it, i.e. you can't initially assume that any location in the .data? section is zero (or any other value) until you explicitly write that value to the location.

You allocate space in the **.data?** section the same way you do anywhere else, except that everything in the **.data?** Section must be initialised to '**?**' (undefined).

EXAMPLES:

```
.data?
db ?
```

Allocates 1 byte of space at the start of the **.data?** section.

```
.data?
hFile dd ?
```

Allocates 4 bytes of space at the start of the **.data?** section and labels the location "hFile".

The **.data?** section is a way of getting the system loader to allocate memory for you. So what's the point?

```
.data?
MyBuffer dd 1000000h dup (?)
```

Causes the system loader to create a 64MB byte buffer for you in VRAM when your app starts.

```
.data
MyBuffer dd 1000000h dup (0)
```

Causes the linker to write over 67 million 0x00 bytes to your .exe file and functions in exactly the same way to the **.data?** section version.

Basically, if you need to specify a starting value for something, put it in the **.data** section, otherwise put it in the **.data?** section.

Bonus points: WTH is going on here?

```
.code
push 0
call @F
db "world",0
@@:
call @F
db "hello",0
@@:
push 0
call MessageBoxA
```

Local Variables

Local variables are temporary storage containers that are valid only inside the proc they were declared in and only for the duration of a single call to that proc.

Locals are created using space in stack memory. MASM uses a bit of smoke and mirrors to make this process transparant when you are coding, so to start you dont need to know how it works. When you get to reverse engineering it is extremely relevant, but that is a subject for a later time.

EXAMPLES:

```
ByteSwap proc ptrNo1:DWORD, ptrNo2:DWORD

  ; *** swap the byte values at two addresses ***

LOCAL myTemp:BYTE

    mov esi,ptrNo1
    mov edi,ptrNo2


    mov al,[esi]
    mov myTemp,al

    mov al,[edi]
    mov [esi],al

    mov al,myTemp
    mov [edi],al

    ret
ByteSwap endp

StupidMultiply proc myNumber:DWORD

  ; *** multiply myNumber by 4 in the stupidest way possible ***

LOCAL Local1:DWORD
LOCAL another_local:DWORD
LOCAL some_crap:DWORD

    mov eax, myNumber

    mov Local1, eax
    mov another_local, eax
    mov some_crap, eax

    add eax, Local1
    add eax, another_local
    add eax, some_crap

    ret
ByteSwap endp
```

Common Instructions

In addition to the ones we have already discussed here's a few more common instructions you might encounter when starting.

`xor` (unsurprisingly) performs a logical XOR of the two operands, however one of its most common uses is for ***zeroing a register***. When you XOR a number with itself, the result is always 0, and this also just happens to be the fastest and most compact way to achieve this. So when you see:

```
xor eax, eax
```

it is setting the EAX register to 0

`test` This is type of comparison similar to **`cmp`** It tests if a certain bit is set in the value you are testing by performing a logical AND with another value. In practice it is often used just to test if something is equal to zero or not. i.e.:

```
push 1000h
push 0
call GlobalAlloc

test eax,eax
jz MyErrorHandler

mov MyBuffer, eax
```

Here we are testing EAX against itself, so the zero flag will be set only if no bits in EAX are set (=0) So we are checking if **GlobalAlloc** has returned NULL, which means it failed to allocate our buffer, so we then jump to an error handling routine that might display a message about system memory and exit the app. Otherwise EAX contains the address of the buffer, so we store this in an appropriate data section container and continue.

`shl` (shift left) This causes the bits in a register to be shifted the requested number of places to the left. Because the bits in a register represent a BASE2 number, this causes its value to be multiplied by 2 once for every shift:

```
shl eax,3
```

This multiplys the value in EAX by 8, provided that none of the top 3 bits in EAX are initially set. (i.e. EAX < 536,870,912)

`shr` (shift right) This causes the bits in a register to be shifted the requested number of places to the right. Because the bits in a register represent a BASE2 number, this causes its value to be divided by 2 once for every shift:

```
shr eax,2
```

This divides the value in EAX by 4, but will round the result to a whole integer. (i.e. the information in the bottom 2 bits is lost)

inc (increase) adds 1 to a value

```
xor eax,eax     ;   eax =0
inc eax         ;   eax =1
inc eax         ;   eax =2 … etc.
```

dec (decrease) subtracts 1 from a value

```
mov eax,5       ;   eax =5
dec eax         ;   eax =4
dec eax         ;   eax =3 … etc.
```

add adds a given amount to a value

```
mov eax,5       ;   eax =5
add eax,3       ;   eax =8
add eax,eax     ;   eax =16
add eax,[esi]   ;   … etc.
```

sub subtracts a given amount from a value

```
mov eax,5       ;   eax =5
sub eax,2       ;   eax =3
shl eax,3       ;   eax =24
sub eax,20      ;   eax =4
sub eax,ecx     ;   … etc.
```

loop decreases ECX by one and jump to the target if ECX != 0

```
mov ecx,5       ;   loop count is 5

xor eax,eax     ;   eax =0
@@:
  inc eax
  inc eax
loop @B
dec eax         ;   eax =9
```

m2m (memory to memory) This is NOT and instruction, but a widely used and handy macro that will insert instructions to move a value straight from one memory location to another.

```
m2m MyLocal,[401020]
```

will expand to:

```
push [401020]
pop MyLocal
```

But in practice you can just use it like a **mov** instruction.

MASM type checking

*What? You mean after that annoying rant this thing actually **does** do type checking?*

hehe, Yes, and I might direct you to the line that says "*beyond defining the physical number of bits that are used to store a value*". MASM is very good at keeping track of the *SIZE* of all the various elements in your code. This is the difference between labels that are defined during data declarations and labels that are defined elsewhere. Many other assemblers require you to specifically define the data size of almost everything you do, however if you have declared:

```
.data?
MyDD dd ?
MyDB db ?
```

then MASM knows when you tell it to `mov MyDD,10h` that it should generate an instruction to put a DWORD 0x00000010 at that address, and when you `mov MyDB,10h` that it should generate an instruction to put a BYTE 0x10 at that address.

Similarly if you say `mov al,[esi]` it knows that AL is a BYTE register so it generates an instruction to read a BYTE from the address pointed to by ESI.

If you try to `mov MyDD, al` the compiler will generate an error, even though "MyDD" is just a location and it is perfectly valid to move the byte in AL to that location, however MASM does do a little bit of baby-sitting and says "MyDD" is a DWORD sized location and AL is BYTE sized … can't do it. There are occasions where you want to do this sort of thing, but if you do you must do something like this:

```
mov edi, OFFSET MyDD
mov [edi], al
```

Occasionally there are cases where a size is not specified by the source so you must explicitly define it with a type declaration and the ptr directive i.e.: `mov [edi], 0AFh` Will generate an error, because 0AFh is an immediate value, so a type is not implied, it could be a BYTE 0xAF or a WORD 0x00AF etc. and although EDI is obviously a 32bit register, it is being dereferenced so it is a pointer to a location and that location does not have a defined size, in this case we must specifically tell the compiler what data size we want to use i.e.:

```
mov BYTE ptr [edi], 0AFh
```

or

```
mov DWORD ptr [edi],0AFh
```

It should also be noted that if not otherwise declared, and if its own personal morals don't demand that it generates an error, MASM32 assumes everything is a DWORD
i.e.:

```
push 0
```

pushes a DWORD zero onto the stack

```
LOCAL my_local
```

creates a DWORD sized local, even though no type has been specified.

BLAH BLAH BLAH

Some less relevant stuff I wrote while producing this document. Read at your own peril.

We will be (I think) exclusively programming the CPU. All modern PC CPUs also have what is called a co-processor attached. Co-processors mainly handle the more complex mathematical tasks. Your CPU can add, subtract, multiply, and to a point divide numbers, but they only work with integers, or "whole numbers", they don't do fractions or decimal points, let alone anything like a square root; that stuff is all co-processor territory, but for most tasks this stuff is not needed. i.e. I have never seen any code in the wc2 executable that uses the co-processor. Co-processors handle "floating point" (FP) numbers hence they are sometimes referred to as the "FPU", they are good at that stuff, but much slower than your CPU, plus there are there are time overheads for getting values from the CPU to the FPU and back again, so its better to just avoid your FPU unless you are doing something that you really need it for. As well as the FPU, there are now a whole new set of instructions and registers that operate on the SSE processor standard extensions and that's "*a whole nother kettle of fish"* we wont be messing with either :P

There's lots of other neat stuff built into the PE format that can be exploited to it's fullest using ASM, and there's also some higher level syntax built into many assemblers. MASM in particular has a whole raft of features that have been added in over 30 years of development, but all of that is optional. At its core MASM operates as a 1:1 assembler. (BTW: IMHO it is the most efficient, succinct and bug-free thing that Microsoft has ever produced, probably because it is the tool that their best programmers developed first and foremost for *their own* use.)

With MASM you can use syntax like .if / .then / .else / .while etc. and these will be converted into a fairly simple sequence of opcodes that produces the program flow you would expect from these familiar terms, however *I am not going to be using any of this* for the PUD editor project, for the simple reason that this is NOT what you see when you disassemble an executable, they are just another way to make source code look friendlier on paper, but in the end they just obscure what is really going on. Once you are used to looking at the actual instructions they are no harder to read than the higher level syntax.

About the only higher-level constructs that I will be using are procedure definitions – the "**proc**" / "**endp**" procedure definitions, and local stack variables created with "**LOCAL**" This is for two reasons; firstly because any compiled language (particularly C) that uses procedure calls and local variables, creates similar code structures and uses comparable methods to create procedures in their compiled code, so you will need to understand how these are created to understand them in a disassembly, and second because it will allow for a better understanding of the calling conventions (mostly "*stdcall*") that are used to access these procedures. This is relevant, in no small part because this is not just how you call internal routines, it is also how everything in the OS is accessed, via procedure calls to procs in the system DLLs.

Have **YOU** *got the chops?*