How to use this manual

This manual is primarily meant to cover some of the material that is not covered in the slides. The slides are meant to be more digestible than these docs. Each header is the same as its place in the table of contents and will go into more detail as to the contents of each section.

Before you start

This document is meant to be used with the slides/pdf. It is recommended that you look at the Before you start slides (slides 4 and 5).

The slides mention the **Programmer's Manual**. In the table of contents all of section 1.1 is mostly about using fasm from the command line. You don't have to use the command line when using the fasm ide that is already there. Section 1.2 contains information regarding very basic references to assembly syntax and the sizing of the individual registers, data directives, and arithmetic and logical operation priority list. Section 2.1 is a much more in depth dive into the syntax of fasm specifically including the simd syntax of mmx, sse, and avx. Section 2.2 mainly revolves around using keywords to customize how the code is assembled. Section 2.3 regards the preprocessor. It is basic compared to the fasm g tutorial which is also mentioned in the fasm g Manual. Section 2.4 is mostly about how fasm is formatted for which operating system you use.

The slides also mention **Understanding the Flat Assembler** manual. The manual is suggested mostly for advanced users. It mostly reveals some aspects of how the optimization and utility of the assembling process is handled by fasm. It also goes in some detail regarding macroinstructions and preprocessing.

Finally slide 4 mentions the **Window's Programming** manual. This manual goes into specific utilities in the windows part of fasm such as structures and procedures. Structures are a common programming utility and their functionality is copied into fasm. Procedures are a specific type of windows macro used for winapi and other similar things. The manual also goes into imports and exports where the syntax is explained in more detail. The manual also goes into COM and resources which are specific windows utilities in fasm.

In slide 5 the slides refer to the **Design Principles**. The small blog refers to the experience and design philosophies into some of the design decisions that built up fasm.

In the folder for fasm (in my case fasmw17332) there are 4 major folders the fasmw.ini configuration file, a fasm pdf which is the manual and the Window's programming manual combined, the fasmw ide executable and a fasm executable. One of the folders is called EXAMPLES. In examples you will see a bunch of example fasm files that will help you when

learning to code in fasm. Another is called INCLUDE. The INCLUDE file stores win32a.inc, win32ax.inc, win32axp.inc and the wide character and 64 bit versions of these files. These files are responsible for handling Win32/Win64 functions with ascii and wide characters(unicode characters). **The API folder includes the default win32 headers and should be used when importing Win32 functions with include 'win32a.inc' or another win32/win64 includes.** The EQUATES file has both the 32 and 64 versions of the INCLUDE files contents. The pcount holds the files that are used with the win32axp.inc and win64axp.inc files. The MACROS folder is used for macros and other extensions to the fasm assembler such as if statements and structs. The ENCODING file has text encoding files for windows including utf8 and others.

Assembly Overview

In slide 6 machine code is mentioned. Assembly code is the regular code that you write in assembly while machine code is an interpretation of what the computer sees. What that means is your computer generates an opcode and the operand. The opcode is the number assigned to whatever directive you want to use. You can think of numbering everything that you can do so that you can tell the computer which directive you want to use. Operands are the arguments you give after the opcode. For more information go to https://www.felixcloutier.com/x86/. When looking at something like an .exe you read machine code.

In slide 9 at&t syntax and intel syntax are mentioned. They are essentially different ways of writing the same thing. Intel syntax is the syntax used by fasm. At&t syntax is commonly used by the gnu corporation which means that GAS uses at&t syntax. The main difference between the two is that intel syntax uses a <destination>, <src> syntax which means that mov rax, 1 will move 1 into rax while at&t syntax uses a <src>, <destination>. Intel syntax however uses implicit suffixes where mov automatically recognizes that the src and destination you are adding are their select size. At&t syntax however requires that you add a b (byte), w (word), l (long) or q (quad) to the end of your mov. This means that mov turns into movl. Registers also have a % in front of them and () are put around the variable instead of []. Intel syntax also formats a complex move with something like mov rax, rbx * 8 + 3. The rbx is multiplied by 8 and then 3 is added. At&t syntax adds constants outside of the () and multiplication is added on by a comma with movq 3(%rbx , 8), %rax.

Binary Formats

The main purpose of an assembler is to generate machine code as stated before in the slides. However certain formats are OS specific. The formats in the slides are examples however you can also make flat binaries in the fasm. Flat binaries are binaries that do not have any structure set by one of these formats.

Registers

Do note that moving a register that is smaller than a qword will allow you to keep the value in the smaller register in the lower however many bits in the smaller register.

```
mov rax, 70000
mov eax, 32
; The value is now 70032
```

The fasm docs section 1.2.1 shows all registers that are not in long mode (they are in 32 bits). The fasm docs 2.1.19 go into the 64 bit general, sse, and avx registers.

Entry

The entry keyword is only used in windows.

Simple Instructions

In slide 19 the mov instruction is mentioned however the slides do not mention the full syntax of the mov instruction with memory. In fasm the you can specify the number of a memory location in the place of a variable. When this happens you have to specify the size of the instruction you want to mov into the memory location.

Ex.

```
mov byte ptr 7, 3
mov [7],3 ; Effectively the same as the thing above.
```

The syntax for moving variables is also customizable in certain ways. You can customize the size of the initialized variable to be smaller than the size of the initialized size. If it is bigger than the initial size then it will find the memory location and extract whatever size you specified after the memory location.

Ex.

```
mov eax, dword ptr num ; This moves the value of num and 3 bytes after that.
mov eax, dword [num] ; This is the same as the line above.
```

Ex.

```
mov eax, dword num
mov eax, num ; This is the same as the line above.
```

For more information go to 1.2.1 of the Programmer's Manual

You can add an n into a conditional jump to make it a not of itself. For example you can make je (jump equal) into jne (jump not equal)

The ZF (zero flag) eflag is used to see when a value evaluates to zero. One example is when checking if two values are equal because subtracting a number by itself will equal 0.

You can when jump when ZF is 1 with the following example

```
        mov al, 0
        cmp al, 0
        jz label_1 ; Jump is taken
label_2:
        mov al, 1
label_1:
        mov al, 2 ; This is run.
```

The CF (carry flag) eflag is used when a value is too big to be stored by your destination. This means that if you try to store a value greater than 65536 (2^16) into an ax register then ax would be unable to store it. In this scenario if you assume ax is 0 and try to add 65537 then the value will wrap around to 0 again.

The OF (overflow flag) flag is used when a value is too big to be stored by your destination. The OF flag applies to signed arithmetic whereas the CF flag applies to unsigned arithmetic.

The SF (sign flag) eflag is used to determine if the number you are evaluating is positive or negative. If the value is positive the flag is set to 0 otherwise when the value is negative the flag is set to 1.

You can jump when SF is 1 with the following example.

```
        mov al, -3
        cmp al, bl ; -3 (al) - 0 (bl) = -3 which is negative.
        js label_1 ; The jump is taken.
label_2:
        mov rbx, 1
label_1 : ; This executes
        mov rax, 1
```

The PF (parity flag) eflag evaluates the bottom byte of a value. If the binary value has an even number of 1s then the PF is set to 1. Otherwise it is set to 0.

You can jump when PF is 1 with the following example.

```
        mov al, 3
        cmp al, bl ; 3 (al) - 0 (bl) = 3 which is odd.
        jp label_1 ; The jump is taken.
label_2:
        mov rbx, 1
label_1 : ; This executes
        mov rax, 1
```

The Aux carry eflag is used with BCD (binary-coded decimal). BCD represents decimal digits (0–9) in 4 bits (a nibble), so the CPU needs to know if an operation on one decimal digit (the lower nibble) caused a carry into the next decimal digit (the upper nibble).
That's exactly what the AF flag tells it.

Go to the intel ia-32 vol1 Section 3.4.3.1 for more information.

Test is similar to cmp however both arguments in the test are anded instead of subtracted.

You can evaluate an arithmetic expression or logical expression with a conditional jump the same way you would a cmp.

Division with rounding ex.

```
mov ecx, 1111001000011000b ; This is the divisor in decimal(31000)
mov eax, 0100100111110000b ; This is binary.
mov edx, 10b ; You have to add rax and rdx. They make 150000
div ecx ; dividend/divisor = quotient. rax is 4 and rdx is 26.
; rax is the actual answer and rdx is the remainder (simplified).
```

Most times when using cmp multiple flags will be set at the same time. Every time cmp is used the flags that do not qualify are reset.

For more information on improving basic instructions like add and sub go to 2.1.3 of the Programmer's Manual.

When working with asm you can use your loops to modularize your code. When writing labels it is a good idea to indent the code written under a label. This makes it a lot easier to read your code because you easily distinguish the start and end of a label.

Eflags

There are more eflags than the status flags in slide 22.

Bitwise/Logical Operations

Bitwise operators work on the individual bits in a number. Logical operators work on boolean values. Since booleans are represented in binary they can easily be confused.

The shr or sar command has a syntax of shr <arg1>, <arg2> where arg2 is either a number or the cl register. Arg1 can be a register of any size or a memory element. An example of how this works is that if arg1 was 0111 and you shifted it by 1 (shl [arg1], 1) then the result would be 1110. Shl or sal does the same thing to the left. Shifting registers is commonly used for exponentiating and squaring values. For example if you shift a value right by 1 then you are squaring it and if you shift a value right by 2 then you are exponentiating it by 4.

For more information go to Section 2.1.5 in the Programmer's Manual.

All of the gates below work with registers or memory values when you add a byte size in front of them. The bitwise operator goes individually through each bit in its arguments and evaluates based on the individual bits. For example if I had a value 1111 a bitwise not operation would evaluate the left most bit and see it is a zero and so would change it to a 0. Then it would go the second most left bit and see it is a 1 and so would change it to a 0. It would go through every bit and then the result would be 0000.

The syntax for an and gate is and <arg1>, <arg2>. The and gate only sets the bit to 1 when arg1 and arg2 are 1. Otherwise the value is set to 0.
The syntax for an or gate is or <arg1>, <arg2>. The or gate only sets the bit to 0 when arg1 and arg2 are 0. Otherwise the value is set to 1.
The syntax for a not gate is not <arg1>. The not gate sets the bit to 1 when the bit is 0 and 1 when the bit is 0.
The syntax for an xor gate is xor <arg1>, <arg2>. The xor gate sets the bit to 1 when there is only one bit that is one in arg1 and arg2. Otherwise the bit is set to 0.

Strings

When using the destination(rdi/edi) and source(rsi/esi) registers you first have to set the dflag. You can set the dflag to 0 with cld and set it to 1 with std. If you want to save all of your eflags you use pushf and popf to push and pop them to the stack. When dflag is 0 then using lodsb shifts the pointer to the right. What that means is that if the h in hello is at location 0 then using lodsb will shift rsi/esi to memory location 1 and al will have h stored at it. Using lodsw will shift rsi/esi 2 and ax will be used to store the value of the word at memory location 0. Using lodsd (lods dword) and lodsq (lods qword) do the same things accordingly. When looking at a large word you can combine the use of different sizes of lods such as lodsd and lodsq to go through a string more efficiently.

```
format PE64
entry start
section '.data' readable writeable
        variable1 db "This is a variable", 0
        variable2 rb 200
section '.code' readable executable
        start:
                sub rsp, 8
                mov rsi, variable1 ; Store the first index of variable1 in rsi.
                mov rdi, variable2 ; Store the first index of variable1 in rdi.
                lodsb ; Loads the index at rsi in the al register and then increments it  by 1 byte.
                stosb ; Stores the index at the al register in rdi and then increments it by 1 byte.
                lodsw ; Loads the index at rsi in the ax register and then increments it  by 2 bytes.
                stosw ; Stores the index at the ax register in rdi and then increments it by 2 bytes.
                lodsd ; Loads the index at rsi in the eax register and then increments it  by 4 bytes.
                stosd ; Stores the index at the eax register in rdi and then increments it by 4 bytes.
                lodsq ; Loads the index at rsi in the rax register and then increments it by 8 bytes.
                stosq ; Stores the index at the rax register in rdi and then increments it by 8 bytes.
```

Stack

In most cases when you force the stack to be 64 bit aligned by doing something like formatting in PE64. You can force a push and pop of dwords and words with pushw and pushd. You can also push and pop all 8 registers (rax, rbx, rcx, rdx, rsi, rdi, rbp, and rsp) with pusha and popa.

You can find more information at 2.1.19 in the Programmer's Manual.

Calls

There are different types of returns with calls. You can use near returns and far returns. The only difference is that retn (near returns) is within the same file while retf (far returns) is outside of the library file such as in a different asm file. The call initially pushes the instruction pointer to the stack and the ret pops it from the stack.

You can find more information at 2.1.6 in the Programmer's Manual.

Floats

Do note that floats are imprecise due to how computers remember floats. Representing float point numbers is hard to do in binary so computers have to make trade offs of precision for range. The IEEE754 outlines the way that floats are handled.

Floats are signed and so one bit is reserved for the sign. A computer handles a float by multiplying a decimal value by 2 to an exponent. In single precision(32 bit) floats the exponent is 8 bits long. The exponent is the part of the float that is multiplied to the mantissa, which is 23 bits long in single precision floats, and allows for floats to have their range. The way a float works is that the mantissa is exponentiated by however much the exponent is. For example 6.25 would be represented as 0 10000001 10010000000000000000000. The leftmost 0 is the

sign (in this case positive). The middle bits are the exponent. The exponents have a base added to them (in this case 127) so <explain adding the base>. Our exponent is 2 in our case as the normalized value of 101.01 (6.25 in binary where 101 in binary is the integer value 6 and .01 in binary is equal to 1/4) is equal to (1 + 0.1001) (2^2) in binary. The final part is the mantissa which has the aforementioned .1001 in binary.

Ex of storing 3.5 in binary
Convert 3.5 to binary

- 3.5 in decimal -> 11.1 in binary (11 is 3 and .1 is 1/2)

Normalize the binary to a 1.<something> (2^exponent)

- 11.1 -> 1.11 (2^1)

Add a base to the exponent

- 127 + 1 = 128 (128 is the exponent)

Extract the mantissa

- 1.11 -> 11 (11 is the mantissa)

Add it all together

- 0 10000000 11000000000000000000000 (0 is the positive sign, 10000000 is equal to 128 and is the exponent, and 11000000000000000000000 is the mantissa)

Winapi

The windows api is a notoriously difficult api. There are certain ways that you have to handle the windows api because it is not always consistent. When looking at the windows functions when searching on google you should know that the types next to some of the parameters are not type enforced in assembly.

Fasm was made a long time ago and so has some legacy code that it works with. One easy example is the windows ide. The fasm windows ide uses some legacy code that is not commonly used in modern gui coding. Winapi also only contains the 6 main dlls. Some functionality that winapi implements are not in those dlls such as the Ole32.dll.

Windows Heap

You should go to https://learn.microsoft.com/en-us/windows/win32/api/heapapi/ for information regarding heaps and heap functions in windows. The library used for windows functions is heapapi.h in kernel32.dll.

In windows a default heap is made with process. You call it with the GetProcessHeap api. You can allocate a specific number of free bytes with HeapAlloc and HeapReAlloc for a specific use. You mark those allocated bytes as unused with HeapFree. You can create a private heap that stays with the process whenever it is closed and opened with HeapCreate. You can then destroy it with HeapDestroy.

A HANDLE points to a resource such as the start of a heap. The normal way to write a variable that stores a handle is to type an h in front of whatever you want to name the handle.

GetProcessHeap returns a HANDLE. A handle is a specific type of pointer to a memory location. In the windows documentation the function is written as

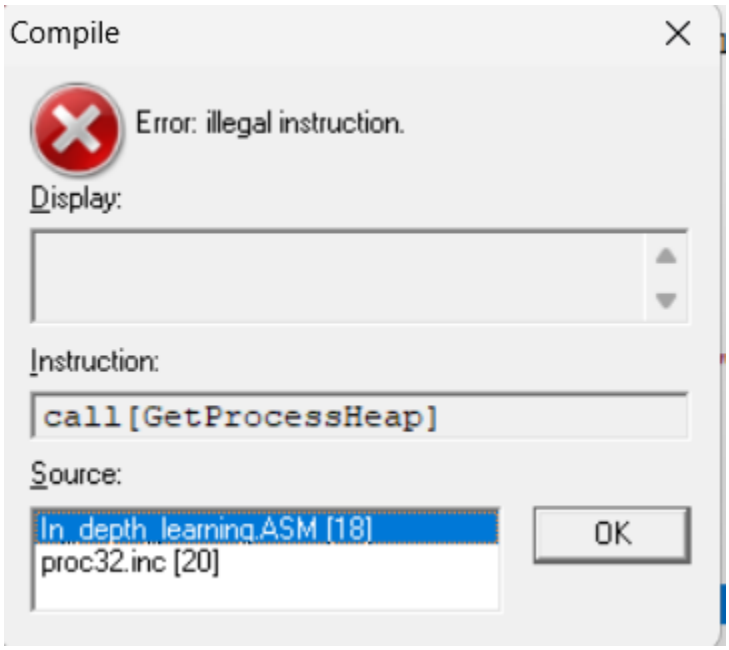
HANDLE GetProcessHeap();

When using asm in 64 bit mode it is essential that the first line be sub rsp, 8. This is meant to align the stack. If you don't do this then there will be problems with winapi reading the parameters you give it.

GetProcessHeap ex.

```
invoke GetProcessHeap
or rax, rax
jz error
mov [hHeap], rax
```

If you include win32a, win32ax, or win32axp while you have set your format to PE64 then every time you invoke a winapi the assembler will give an illegal instruction error.



ExitProcess is the api used to kill the process that is running.

The way you write to the heap is the same way you write with a string.

```
format PE64
entry start
include 'win64a.inc'
section '.idata' import data readable writeable
library user32, 'user32.dll', \
        kernel32, 'kernel32.dll'

        include 'api/user32.inc'
        include 'api/kernel32.inc'
section '.data' readable writeable
        variable1 db "This is a variable", 0
        hHeap dq 0
section '.code' readable executable
        start:
                sub rsp, 8
                invoke GetProcessHeap
                or rax, rax
                jz error
                mov [hHeap], rax
                mov rdi, [hHeap]
                mov rsi, variable1
                lodsb ; Gets the first letter of variable1.
                ; and puts it into al.
                stosb ; Takes the value in al and puts it
                ; into rdi.

        error:
                invoke ExitProcess
```

HeapAlloc ex.

```
invoke HeapAlloc, [hHeap], HEAP_ZERO_MEMORY, 3 ; Allocates 3 bytes.
or rax, rax
jz error
mov [hSpecificHeap], rax
```

HeapReAlloc ex.

```
invoke HeapReAlloc, [hHeap], HEAP_ZERO_MEMORY, [hSpecificHeap], 5
; Reallocates hSpecificHeap to 5 bytes.
or rax, rax
jz error
mov rdi, [hSpecificHeap]
```

And all together

```
format PE64
entry start
include 'win64a.inc'
section '.idata' import data readable writeable
library user32, 'user32.dll', \
        kernel32, 'kernel32.dll'

        include 'api/user32.inc'
        include 'api/kernel32.inc'
section '.data' readable writeable
        variable1 db "This is a variable", 0
        hHeap dq 0
        hSpecificHeap dq 0
section '.code' readable executable
        start:
                sub rsp, 8
                invoke GetProcessHeap
                or rax, rax
                jz error
                mov [hHeap], rax
                invoke HeapAlloc, [hHeap], HEAP_ZERO_MEMORY, 3 ; Allocates 3 bytes.
                or rax, rax
                jz error
                mov [hSpecificHeap], rax
                invoke HeapReAlloc, [hHeap], HEAP_ZERO_MEMORY, [hSpecificHeap], 5
                ; Reallocates hSpecificHeap to 5 bytes.
                or rax, rax
                jz error
                mov rdi, [hSpecificHeap]
                mov rsi, variable1
                lodsb ; Gets the first letter of variable1 and puts it into al.
                stosb ; Takes the value in al and puts it into rdi._
        error:
                invoke ExitProcess
```

Segment Registers

Computers cannot represent the physical memory address with just a 64 bit register. For that reason segment registers are used to add an upper 16 bits to whatever they register they are being added to. There are 6 16 bit segment registers. They are the cs, ds, es, fs, gs and ss registers. The cs register contains the offset for the instruction pointer, the ds register contains the offset for the data section, the es register contains the offset for string operations and other data storage, the ss contains the offset for the stack and the fs and gs contain offsets for whatever the operating system determines.

The syntax for appending on the segment registers is <size> ptr <segment register>:<variable>. It is usually fitted inside of a command such as the example below.

Ex.

```
add [num], 2 ; The assembled code is below
; add byte ptr ds:[401033], 2_(401033 is the memory location for num)
```

Suggested Additions

If you find anything in this document is unclear or you become familiar with linux fasm, vector registers or another topic then you are welcome to send a pull request and suggest changes or additions. For more information you can go to the readme in the Individual Projects folder in the github.