Assembly Language

**Introduction**

If you are reading these notes, you are most likely familiar with basic/intermediate programming. You understand hardware concepts to some degree as well as computer arithmetic in different base counting systems. You should probably have at least a basic understanding of the C programming language, as you will begin to see similarities between assembly and C. Now, I’m assuming you have an idea of what assembly language is and what it does considering you chose to read these notes, but perhaps it is best I repeat exactly what assembly language is, so we are on the same page. Assembly is indeed considered a programming language, though it is not a conventional one. There are two primary **styles** of syntax (Intel syntax and AT&T syntax), although the actual syntax limitations is dependent on the hardware. This document will explain Intel syntax and only Intel syntax because it is the most widely accepted one, is much cleaner and more human readable. The trade off is that not many systems actually use intel syntax, but if you can grasp the concepts here, you shouldn’t have much trouble reading AT&T. The language is extremely low level, in fact it is the lowest level **language** that we can write in, and it can be directly translated to machine code ie. the binary/hexadecimal, which your CPU reads, if you know what you are doing. It is important to remember that software is an illusion. There are only electrical signals, turning LEDs on or off, storing electrons in gates, and transmitting them back and forth. I remind you of this because assembly and hardware are closely tied. We are directly manipulating our hardware, unlike higher level languages which are just an abstraction of this to make things quicker. We will brush over all of the core concepts of assembly in this document, but we probably won’t get too advanced since assembly is really just something that takes a lot of practice via tinkering with circuits, and usually isn’t that practical in the real world. We will begin this topic with a review of counting systems in different bases (2, 8, 10, 16), since we will be doing a lot of conversions (especially in hexadecimal).

**Number Conversions Review**

You should be familiar with all the concepts that I am about to revisit, but I wanted to add them here as a refresher because if you don’t practice this stuff regularly, you tend to forget it pretty easily. We know that every counting system has a base. Bases are really just a convention that we make up to help keep our numbers short. I could write the number 10 as 10 lines ie. |||||||||| and so on for every consecutive number, but that becomes very difficult to count and write after a while. The base we are most familiar with is base 10, presumably because humans learned to count using their fingers. We use binary in computers because each number can be 1 of 2 states (0 or 1). When we exceed the available number of states, we always increment the following placeholder. For example, in binary we count 0000, then 0001, then we would suspect 0002 in base-10, but because we have exceeded the number of states (2), we instead increment the next placeholder by 1 ie. 0010 and flip all lower bits back to 0. The same concept is true for all other systems. Hexadecimal counts from 0 – 15, and then increments the next placeholder (1, 2, 3 ,4, 5, 6, 7, 8, 9, A, B, C, D, E, F, 10, 11, 12, etc.) The convenient thing about this is that these are all very easy to convert to decimal. If I want to convert the number $1F to decimal, I look at each individual place holder and multiply the digit by b^i where b is the number system’s base, and i is the index of our position, ie our placeholder. In the case of $1F, I look at F which is 15 in decimal, multiply it by 16^0 and get 15. I then add this to the result of the next digit (1 x 16^1) and get 31. And this makes sense because in order to have 1 in the second placeholder I must have already gone past 15 and then incremented by an additional F to arrive at 31. As another reminder, it is very easy to convert between binary and hexadecimal as well. Hexadecimal was created for the purpose of representing large binary numbers, so it makes sense that they convert between each other nicely. You can also think of it in terms of their bases. If binary is base 2 and hex is base 16, then really hexadecimal is just base 2^4 (2x2x2x2 = 16) which is why each hexadecimal placeholder is represented as a nibble in binary (4 bits). In other words, 4 bits provides an equal number of possible states as 1 number in hex (2^4 bits = 16^1 hex in terms of possible values).

I think it is also very important to quickly revise negative numbers or signed numbers as we refer to them in programming. A signed number is one which sacrifices a bit of data to be represented as negative. Signed and unsigned numbers have equal capacity for storage; in other words, 1 byte of signed data vs 1 byte of unsigned data contain different **ranges** of values but store the same **number** of possible values. The equation for unsigned values is 0 --> 2^n bits, and the equation for signed data is 2^(n-1) <-–-> (2^(n-1)) – 1 where n is the number of bits in binary. So for one byte (8 bits), the unsigned range would be 0 --> 2^n = 0 --> 255 and signed range is 2^7 <---> (2^7) – 1 = -128 <---> 127. Signed values can be represented in different ways. You are probably aware that computers tend to use two’s compliment for this, and this is because one’s compliment has two representations of 0 (-0 and 0) and we prefer to remove -0 since it is not a real number. Also keep in mind that 1’s compliment needs carry out values when an overflow occurs ie. no more placeholders are available, but in two’s compliment the carry out is omitted because once again, it only has one representation of 0 unlike one’s compliment. You may also remember a third way to represent unsigned values which is quite unconventional called signed magnitude, and this is where we actually change the MSB (most significant bit, or leftmost bit) to be either 1 (if negative), or 0 (if positive). This is not the case in one’s or two’s compliment which may express binary values as negative even if the first digit is a 0 because the CPU is able to keep track of what is meant to be negative and what is meant to be positive. So, for example, the one’s compliment of 7 or 0111 in binary is 1000 (remember, we just perform a bitwise AND with all 0s or in other words, simply flip the bits in one’s compliment) and 7 in two’s compliment is 1001. Both 1000 and 1001 are representations of -7 that we might mistake to be positive numbers, but that the CPU knows are negative numbers because it keeps track. Another example could be 12 or 1100 in binary. In one’s comp its 0011 and in two’s comp its 0100. In order to subtract -12 from -7, we will have to redo our conversion. This is because we know -7 -12 = -19 which occupies more than 4 bits. If we don’t convert 7 to -7 and 12 to -12 with a size large enough to store the result, we run into big issues. We perform one’s compliment on 7 and 12 again, this time using 8 bits, and find that 7 becomes 11111000 and 12 becomes 11110011. While we can technically do subtraction in binary, it is really just adding a negative number. But since both of our numbers are already negative, we can just add them (since -7 + -12 is equivalent to -7 -12). . Notice that there is a carry out bit. In two’s compliment, we ommit carry outs, but in one’s, we add it back into our result. . Knowing that this number should be -19, we can perform one’s compliment on it to check if we get 19. And indeed, the one’s compliment of 11101100 is 00010011 which is 19. To do this in two’s compliment would be the same, only that we would use the two’s compliment representation of -7 and -12 which would be 11111001 and 11110100 respectively. We then add: . We remove the carry to get 11101101. To check, we do two’s compliment on this and get 00010011.

**The Call Stack**

Alright, hopefully you have brushed up on your math, especially hex addition/subtraction which I didn’t go over, but that you should be familiar with. As you are aware, our program creates a structure in memory called the stack. The stack is allocated with a certain amount of memory, and when we exceed this memory, we receive a stack overflow. This is where our assembly code resides. In essence, we write our assembly code, and a program called the assembler outputs our executable object code in a computer readable format. The assembler is to assembly language as the compiler is to any other programming language. All that the assembler is doing however, is translating each instruction that we provide it into an operation code, or opcode for short. An opcode is a sequence of specific bits (generally written in hex) which represents a CPU instruction. This opcode is fetched, and then executed. This is also why I said that assembly language can directly be translated to machine/byte code (because if you know the opcodes for the instructions you are writing, you can simply substitute the instructions for opcodes). Processors vary in **architecture**. Architecture describes central attributes that differentiate families of processors. The two most common architectures as of 2021 are the 8086 architecture (more commonly referred to as x86), and it’s 64-bit big brother x86-64. x86 and x86-64 are generally just called 32 bit and 64 bit respectively. 32 bit and 64 bit refer primarily to the amount of bits used for important features such as the opcode table ie. the list of available instructions, as well as the number of bits that it can handle at once for performing arithmetic such as floating-point math. Each opcode must be at least one byte but can be multiple bytes as well. The upper nibble represents a row in the opcode table, while the lower nibble represents the column in the opcode table. Therefor, one byte = one instruction. We will look at the opcodes in more depth soon. For now, I just wanted to point out that a) our program runs in the stack, b) our processor has an architecture, and c) opcodes are what we use to fetch, decode, and execute instructions.

**Registers**

Registers are something that might confuse you if you’ve been programming for a bit and aren’t aware of what they are. This is because registers are an exclusive concept in assembly language. They are also semantically confusing as well. Based on a processor’s architecture, it will have a certain number of “primary” registers. Registers are simply specific locations reserved in memory for holding temporary data. The “primary” registers often just referred to as “registers” are located on a special chip within the processor and are called using their appropriately assigned names. It is not technically incorrect to have other registers that exist elsewhere, such as RAM, and be only accessible through their memory address. It is simply that when we say “register” we are usually talking about the ones reserved on the special memory chip in the processor. With that semantic mess out of the way, let me re-iterate. Registers hold temporary data that we know we will need to use. Unlike regular programming, where data is generally stored in variables, in assembly we are generally relying on registers to store our data. Variables sort of exist, although they are typically called “labels” because we simply label a memory location for simplicity (same as how regular programming languages work under the hood). Typical registers include A, B, C, D, X, Y, IP/PC, and SP. IP/PC stand for Instruction Pointer and Program Counter respectively, but refer to the same thing, and SP stands for Stack Pointer. The others stand for nothing, and are just simple letters to remember.

**Memory Mapping**

One thing that is not crucial to understand off the bat, but is very helpful, are memory maps. Any piece of complex computer hardware will likely have a memory map. When the manufacturers decide upon the chips to use in their product, they need to efficiently map specific memory addresses to specific chips/tasks. This is sort of difficult to understand without having an idea of how computer hardware works, but in essence, we can enable and disable certain chips based on the memory address that we attempt to access. For example, if I try to access memory location $0000-$2000, that might be mapped to the ROM chip. Then $2001-$4000 might be for the RAM chip, and then $4001-$4100 might be for our I/O ports. Memory maps are pretty much just documented to help the manufacturers remember which memory locations serve which purpose, otherwise the entire thing becomes messy and inefficient.

**Other**

This is the final section before we go into some syntax. One big difference between the intel flavor syntax, and AT&T/Motorola syntax is the endianness. Most intel-based chips are little-endian, meaning that the lower order nibble gets swapped with the high order nibble. AT&T/Motorola have big-endianness, which is how we usually read hexadecimal. The main argument as to why these needed to differ is efficiency, but I digress. Along with big-endian vs little-endian, the way we represent binary numbers are hexadecimal numbers differs between the two syntaxes as well. Intel style generally places a B after its binary number and an H after it’s hex number, whereas AT&T/Motorola flavors generally represent binary numbers with a % in front of the number, and 0x in front of hexadecimal numbers. The final thing I wanted to go over is what a word refers to. A word is sort of like byte, short, int, long datatypes, as it is able to store a specific amount of data, however, this amount of data is based on the processor’s architecture. Therefor a word on an x86 machine would mean 32 bits, whereas a word on an x86-64 machine would hold 64 bits.

**Environment**

I am for the most part going to be summarizing this guide: [https://www.tutorialspoint.com/assembly\_programming/assembly\_environment\_setup.htm](https://www.tutorialspoint.com/assembly_programming/assembly_environment_setup.html). They will show you how to install NASM which is the assembler that we will be using. We will also be working with Intel-32 processors such as the Pentium processor, which was a very popular family of processors from 1993 ‘till present, although the present day Pentiums don’t share much in common with the earlier models. Just as a note, the guide I provided states that you will need Linux, although there are alternatives to Linux such as Windows Subsystem for Linux (WLS), or lightweight VM like Cygwin or MINGW. You could also install Oracle’s Virtual Box VM manager for free if you’re comfortable torrenting a Linux ISO, and setting it up.

**The Basics**

Pretty much all assembly language flavors will have you split the code into two sections: One for constant data, such as variables, and then a section for the program code. There is a third section that generally goes untouched by the programmer called the stack section that keeps track of return addresses when calling a subroutine among other data. I’m not particularly fond of how Intel decides to do this, but I won’t complain here. In the Intel syntax we’re looking at, each section requires a header to differentiate itself from other sections. The data section starts with section .data. The stack section, called bss, starts with section .bss. Finally, the section for our code starts with section .text. The code section is unique because it requires us to tell the kernel where the actual code starts. That means that the full declaration for this section actually looks more like this:

section.text

global \_start ; kernel starts reading code here

\_start:

Notice that I placed a semi colon after global \_start. This is not used to end lines unlike most programming languages. Confusingly enough, this is actually how we make comments, and is usually universal for all assembly syntaxes. As I stated earlier, we have opcodes which are bytes that represent an instruction within the opcode table. Opcodes have aliases/mnemonics to make our lives easier, and these mnemonics get translated back to their original hexadecimal notation upon assembly to machine code. For example, **INC** is the label for 40H (40 hexadecimal) and it is a CPU instruction which increments a variable. In the syntax we’re looking at, each statement is laid out in the following order: [label] mnemonic [operands] [;comment] where the fields in square brackets are optional. Here are some more examples to demonstrate this order:

INC COUNT ; Increment the memory variable COUNT

MOV TOTAL, 48 ; Transfer the value 48 in the

; memory variable TOTAL

ADD AH, BH ; Add the content of the

; BH register into the AH register

AND MASK1, 128 ; Perform AND operation on the

; variable MASK1 and 128

ADD MARKS, 10 ; Add 10 to the variable MARKS

MOV AL, 10 ; Transfer the value 10 to the AL register

**Hello World**

Now that we’ve looked at the 3 sections our assembly program will use, along with the structure for our instructions, we can code a hello world program. Here is the final result and then we will look through it:

section .text

global \_start ; must be declared for linker (ld)

\_start: ; tells linker entry point

mov edx, len ; message length

mov ecx, msg ; message to write

mov ebx, 1 ; file descriptor (stdout)

mov eax, 4 ; system call number (sys\_write)

int 0x80 ; call kernel

mov eax, 1 ; system call number (sys\_exit)

int 0x80 ; call kernel

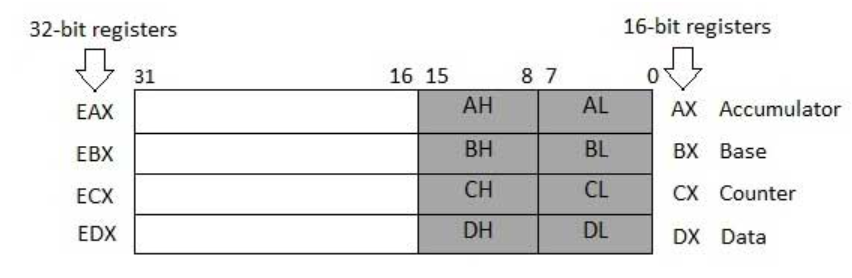
section .data

msg db “Hello, world!”, 0xa ; string to be printed

len equ $ - msg ; length of the string

Alright, let’s look through this. Right after \_start, you will probably be immediately confused. For the most part, we are using the MOV instruction. What this does is move the right operand into the left operand (which must be a register in this case). So imagine the EDX register as a temporary variable, and we are setting it equal to len. The problem is, where’s len defined? Well it’s actually down below under the section.data header. We can see that len equals $ - of message. The comment helps us decipher this further, telling us that $ - means “length of”. Note that the 0xA in hexadecimal is a line feed on Linux based systems ie. it’s equivalent to ‘\n’. The length of msg, including this line feed is 14 bytes. So we place 14 into the EDX register, then we move the actual string, msg, into the ECX register. Once again, the comment on the third line helps us out by telling us that the 1 will be used later to tell the program that msg should get written to the stdout file stream. The Sys-call named sys\_write is a kernel function of Linux which will will look into later. Once the appropriate values are stored in their appropriate registers, we call the INT instruction which is short for hardware interrupt. Essentially, it tells the CPU, stop what you’re doing and pass the code to the appropriate interrupt handler. 0x80 happens to be the system kernel, so the kernel will execute the code written thus far. With all of these values in place, the kernel knows it’s performing a sys\_write to stdout with the value msg and a length of 14 ie. read 14 bytes starting from the memory address of msg. The final two operations load EAX with 1 which is the sys-call for sys\_exit, and then the kernel once again executes this terminating the program. Note that in order to assemble this, assuming NASM is installed, we run the command nasm -f elf hello.asm. The file extension .asm is what we use for assembly language programs. -f elf means format the object file in the Executable and Linkable Format (ELF), which is a common object file format. Note that we’ve only created an object file which converted our .asm file to machine code. In order to create an executable, we need to link the appropriate libraries as well. The command to do this is ld -m elf\_i386 -s -o hello hello.o. I will let you read the man pages if you’re curious as to what all the options mean, but if you’re familiar with C, I think you will understand most of it.

**Registers**

So, I think now would be an appropriate time to cover the general purposes of each register. It’s not like you are obligated to use every register for it’s intended purpose, but it is good to stick to the recommended purpose as best as possible. First we’ll look at EAX, EBX, ECX, and EDX. 

Notice how each of these is 32 bits long. Omitting the ‘E’ at the beginning limits us to the lower 16-bit registers: AX, BX, CX, and DX. AH, BH, CH, and DH refer to the high byte of the lower 16-bit registers, and AL-DL refer to the lower byte of the lower 16-bit registers. We use AX as the input and output for most of our arithmetic operations. For example, if we want to multiply, we use EAX, AX, or AL according to the size of the operand. BX is the base register. It is usually used for index addressing in an array. CX is the counter register. It stores the loop count in iterative operations. Finally, DX is the data register and is used in conjunction with AX to perform larger scale arithmetic operations.

Lets look at pointer registers EIP, ESP, and EBP along with their lower 16-bit counterparts IP, SP, and BP. These three registers keep track of our location in the stack when we are executing our code. IP is the instruction pointer. You can think of it as an arrow, always pointing at the next instruction to execute. SP points to the start of our stack. IP really just stores an offset based on how many instructions have been executed so far and subtracts that from SP. We subtract instead of add simply because our stack gets read top to bottom. Finally, BP helps reference parameter variables passed to sub-routines (assembly’s name for functions).

Next we have the 32-bit index registers, ESI and EDI and their lower 16-bit counterparts SI and DI. SI means source index ie. the first index of a string, whereas DI means destination index ie. the final index of a given string.

Now lets look at the control registers. Pretty much all processors, if not all, have a control register that keeps track of the status of certain operation attributes. Each status is called a flag, and each flag is represented as one bit in the control register. EIP in combination with another register known as the flags register makeup the control registers. Here’s a description of each flag:



* Overflow Flag (OF) − It indicates the overflow of a high-order bit (leftmost bit) of data after a signed arithmetic operation.
* Direction Flag (DF) − It determines left or right direction for moving or comparing string data. When the DF value is 0, the string operation takes left-to-right direction and when the value is set to 1, the string operation takes right-to-left direction.
* Interrupt Flag (IF) − It determines whether the external interrupts like keyboard entry, etc., are to be ignored or processed. It disables the external interrupt when the value is 0 and enables interrupts when set to 1.
* Trap Flag (TF) − It allows setting the operation of the processor in single-step mode. The DEBUG program we used sets the trap flag, so we could step through the execution one instruction at a time.
* Sign Flag (SF) − It shows the sign of the result of an arithmetic operation. This flag is set according to the sign of a data item following the arithmetic operation. The sign is indicated by the high-order of leftmost bit. A positive result clears the value of SF to 0 and negative result sets it to 1.
* Zero Flag (ZF) − It indicates the result of an arithmetic or comparison operation. A nonzero result clears the zero flag to 0, and a zero result sets it to 1.
* Auxiliary Carry Flag (AF) − It contains the carry from bit 3 to bit 4 following an arithmetic operation; used for specialized arithmetic. The AF is set when a 1-byte arithmetic operation causes a carry from bit 3 into bit 4.
* Parity Flag (PF) − It indicates the total number of 1-bits in the result obtained from an arithmetic operation. An even number of 1-bits clears the parity flag to 0 and an odd number of 1-bits sets the parity flag to 1.
* Carry Flag (CF) − It contains the carry of 0 or 1 from a high-order bit (leftmost) after an arithmetic operation. It also stores the contents of last bit of a shift or rotate operation.

**Syscalls**

Let’s look at how the seemingly random movement of values into seemingly random registers made the kernel print our message. To view