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 convential one. There are two primary versions of syntax (Intel syntax and AT&T syntax), although there are additional sudo assembly syntaxes that you may stumble accross. This document will explain Intel syntax and only Intel syntax because it is the most widely accepted one, is much more clean and human readable. 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 on all 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, 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 of 2 states (0 or 1). When we exceed the available amount of states, we always increment the following placeholder. For example, in binary we count 0000, then 0001, then 0002, but because we have exceeded the amount of states (2), we instead increment the next placeholder by 1 ie. 0010. 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 amount of possible states as 1 number in hex (2^4 or 16^1 possible states).

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 sacrafices 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 amount of numbers. 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 may mistake to be positive numbers, but that the CPU knows are negative numbers. Another example could be 12 or 1100 in binary. In one’s its 0011 and in two’s 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 equivellant 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 shoudld 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 recieve 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 excecutable object code. 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 op-code is a sequence of specific bits (or more commonly, some hexadecimal value) which represents a CPU instruction. This op-code is fetched, and then excecuted. This is also why I said that assembly language can directly be translated to machine/byte code (because if you know the op-codes for the instructions you are writing, you can simply substitute the instructions for op-codes).