Assembly Language Course (from Z*Magazine)#

Table of Contents

Z*Magazine Archive: http://www.atariarchives.org/cfn/12/05/02/index.php

CHRIS CRAWFORD - ASSEMBLY LANGUAGE COURSE FOR WORLDWIDE USERS NETWORK

Assembly language is the great barrier that divides the professional programmer from the amateur. It is the most powerful language available for a microcomputer.

There are four reasons for learning to program in assembly language. First, the speed of execution of assembly language is very high -- about ten time higher than BASIC on the average, perhaps a thousand times faster on certain operations.

Even ACTION, the fastest high-level language, is only about half as fast as assembly language. Second, assembly language tends to be more compact than many languages. Again, ACTION! provides a good comparison. Code produced by ACTION! is about twice as large as equivalent assembly language.

The third reason to program in assembly language is that assembly gives you access to features of the machine that simply are not available in high-level languages. Interrupts are the most notable examples.

Finally, the most important reason for learning to program in assembly language is thait will help you to understand the machine better. And that is a very good place to begin, for you cannot learn assembly language unless you know a little bit about computers.

HOW COMPUTERS WORK#

I am now going to describe how computers work, in very rough terms. Computers operate on a hierarchy of concepts that spans a great range, rather like the hierarchy that starts with protons and electrons, moves through atoms, molecules, cells, people to civilizations.

A civilization is composed of protons and electrons, but to understand how it is so composed one must know a great deal about the intermediate steps. So too is a computer composed of transistors. There are four intermediate steps between the transistor and the computer.

A transistor is an electrically operated switch. We can assemble transistors into gates that will turn circuits on or off depending on the states of other circuits. There are a variety of gates reflecting the various Boolean operations: AND, OR, NOT, NAND, NOR and EOR.

Gates can be assembled into latches, decoders, and adders. A latch is the simplest memory element: it remembers one bit of information. A decoder translates a number encoded in binary form on a few wires into a selection of one of many wires. An adder will add two one-bit values, with a carry, and generate a carry of its own.

We can next broaden each of these devices into an eight-bit device by simply slinging the devices side by side. Eight one-bit latches slung side-by-the side give one byte of RAM. Eight adders make an eight-bit adder.

We can thus create a RAM module by building many butes of RAM. We access this RAM module with three buses: a data bus, an address bus, and a control bus. The data bus carries information between the central processing unit and the RAM module.

The address bus is sixteen bits wide; a decoder inhe RAM module. The address bus is sixteen bits wide; a decoder in the RAM module takes the numeric value on the address bus and decodes it to select the single byte of RAM that is indicated by the address. The control bus establishes the direction of the data flow on the data bus and the timing of data transfer.

The central processing unit (CPU) represents the highest intellectual level of the computer. It is composed of four parts: the Arithmetic and Logic Unit (ALU), the registers, the address bus controller, and the instruction decoder. The ALU is composed of adders and gate arrays that crunch numbers. The particular device to use is selected with a decoder.

The registers are simply on-board RAM. The address bus controller is a device that puts the desired RAM address onto the address bus. The real heart of the CPU is the instruction decoder, a very complex decoder that takes the program instructions out of RAM and translates them into action. It does this by feeding the instructions (which are numbers) into decoder circuits that activate the desired gateways in the CPU.

PROGRAMMING A MICROPROCESSOR#

Machine code is nothing more than a bunch of numbers that mean something to the CPU. It's hard to work with pure numbers, so we use a little code that makes it easier for us to understand the codes that the computer uses. This programmer-friendlier code is called assembly language, It is a direct, one-to- one translation of machine code. Here is an example of the two side by side:

Machine Code	Assembly Language	
A9 05	LDA #FINGERS	
133 $9C	STA COUNT	

The code on the right may not look very readable, but you must agree, it's far more readable than the code on the left. And they both mean exactlthe same thing.

Unfortunately, the computer cannot read the assembly code, only the machine code. Thefore, we need a translator program that will translate the easier-to-understand code on the right into the impossible-to-understand code on the left. This translator program is called an assembler.

A program that goes in the reverse direction, translating machine code to assembly, is called disassembler. It may seem like a bother to go through all the hassle of using an assembler, but it is actually much easier.

Assembly language is not only more readable than machine code, but it is also assembly-time relocatable; this means you can move it around in RAM freely before you start the assembly process. A good assembler also offers a number of extra features that make it easier to keep track of your program or modify it quickly.

USING AN ASSEMBLER#

There are three steps involved in writing an assembly language program: editing, assembling, and debugging. Editing is the process of typing in your assembly language statements. Assembling is the invocation of the assembler. Debugging is the process of running your program and analyzing why it doesn't work. Thus, the entire process of writing an assembly-language process can be described by a fictitious BASIC program:

```
FOR 1= 1 to 1,000,000,000...
EDIT PROGRAM
ASSEMBLE PROGRAM
DEBUG PROGRAM
NEXT 1
```

THE 6502 MICROPROCESSOR#

The first item in the 6502 that I will describe is the accumulator. This is a single one-byte register in the 6502. It is the central workbench of the microprocessor; almost everything happens in the accumulator. Your first three instructions on the 6502 are:

LDA address (Load the Accumulator with the contents of address)

This instruction loads the accumulator with the contents of the memory location specified by the value of address. The address can be specified by either an out-right value, such as $0600, or a symbolic reference, such as FISH, ere the value of FISH has been previously declared by, say, an ORG statement or an equate statement.

LDA #value (Load the Accumulator with value)

This is much like the earlier statment; it loads the accumulator with a number, only the number loaded is specified immediately rather than stored in a memory location. Thus, the command LDA # 9 will put a 9 into the accumulator.

STA address (Store the Accumulator into address)

This command will store the contents of the accumulator into the RAM location whose address is specified in the command. It is just like the first command, except that the direction of data motion is reversed. The LDA command is like a read, which the STA is like a write.

You are now equipped to move data around inside the computer. These commands will allow you to readata from one area of memory and store it into another. LDA and STA are the two most common instructions used in any 6502 program.

Exercise: Write a program that will read the contents of address $FE00 and store the result into address $680. Your biggest problem here will be just getting your assembler to work. Therefore, I will give the answer away:

```
ROMADD ORG $FE00
RAMADD ORG $680
        ORG $600
        LDA ROMADD
        STA RAMADD
        BRK
        END
```

That's the program. Try to get it running with your assembler.

NUMBER SYSTEMS#

In this lecture I will take up the problem of arithmetic on the 6502. I choose this topic only because it is fairly simple to do on the 6502. There are a couple of nerve-jangling problems associated with 6502 arithmetic, but I will breeze over those in a very cavalier fashion.

Before we can do arithmetic, though, you must know a little bit about number systems. There are three that you must know: decimal, binary, and hexadecimal.

Decimal is the standard numbeat you have used since grade school. You count 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, and then you reach 0 again, so you put down a 1 in the tens place and resume counting from 0.

Binary works the same way, except that there are only two digits, not ten. The two digits are 0 and 1. You count 0, then 1, then you reach 0 again, so you put down a 1 in the twos place and resume counting from 0. Thus, counting from 0 to ten in binary like this:

Decimal	*Binary*	
0	0	
1	1	
2	10	
3	11	
4	100	
5	101	
6	110	
7	111	
8	1000	
9	1001	
10	1010	

In binary, instead of having ones, tens, and hundreds places, we have ones, twos, fours and eights places. It takes a lot more digits to express a number in binary, but then again, we have only the two numberals 0 and 1 to work with, so what does one expect?

The hexadecimal number system is a base-16 system. In this system, you count from 0 to 16 like so 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E ,F,10. The 10 in hexadecimal really means 16 in decimal. So 10 is 16, right? Black is white, truth is likes....stay with assembly language long enough and you'll believe anything.

Actually, it's easy to avoid confusion. We use little prefixes to tell you and the computer whether a number is expressed in decimal, binary, or hexadecimal. No prefix means decimal. A $ prefix means hexadecimal; a % means binary. Thus %10 means 2 while $10 means 16, but 10 means just plain old 10. Hexadecimal is not hard to learn at all; if you go into any store you will see that they use hexadecimal on all their signs.

ADDITION AND SUBTRACTION#

Addition with the 6502 is very simple; it uses the ADC instruction. This instruction means "Add with Carry"; I'll get to the Carry part in just a moment.For now,let me explain the instruction. The ADC instruction has an operand, normally a location in memory. When the instruction is executed, it takes the contents of that memory location and adds that value to the value in the accumulator.

It leaves the sum of the two numbers in the accumulator. This of course destorys the old value in the acucmulator. You can use the immediate mode of addressing with the ADC instruction, in which case it adds the value itself. Thus, "ADC # 9" will add a 9 to the contents of the accumulator, while "ADC FISH" will add the contents of address FISH to the accumulator.

Subtraction is just like addition. The instruction to use is SBC, which means "Subtract Borrowing Carry". Again, I'll tell you about the Carry part in a moment. This instruction subtracts the operand from the contents of the accumulator, leaving the result in the accumulator, It also can be done in either immediate mode (e.g. SBC#5) or absolute mode (e.g., SBC GOAT).

WORD LENGTH PROBLEMS#

If that were all there were to arithmetic with the 6502, programmers would be paid a lot less. The first killer problem is that the 6502 uses 8-bit words; that is, the numbers that the 6502 stores and works with are only 8 bits wide. This means that the biggest number the 6502 can comprehend is 255. Uh-oh! What happens if you want to have a checkbook balancing program and you have more than $255? What happens if you get more than 255 points in your "Decapitate the Orphans" game? In fact, what happens if you just ignore the limit and add, say, 10 to 250?

Well, believe it or not,the 6502 will give you an answer of 4. Why? The number system that the 6502 uses is like a wheel, with 0 at the top, counting clockwise 1,2,3,... all the way up to 255, which lies right next to the 0. If you go up from 255 you just wrap around past the 0 and start all over. Similarly if you subtract 2 from 0, you'll get 254.

The solution to all this is provided by the Carry bit, discussion of which I've been putting off unitl now. The Carry bit is a flag that the 6502 uses to remember when it has done arithmetic that carried it over the boundary between 0 and 255. By using it properly, you can solve your arithmetic problems.

The first trick to using the Carry bit is to use multi-byte words. This means that, instead of using a single byte to store a number, you use several. For example, if you use two bytes to remember a number, you can store a number as large as 65,535. three bytes lets you to to 16,777,215. Four bytes lets you go to 4,294, 967,295. big enough for you?

To use multi-byte arithmetic, you set up a series of additions of subtractions. Suppose, for example, that you want to add two two-byte words. The program fragment to do this would look like this:

```
LDA     LOFISH
CLC
ADC     LOGOAT
STA     LOANSR
LDA     HIFISH
ADC     HIGOAT
STA     HIANSR
```

This little fragment of code assumes that the first two-byte value is called (LOGOAT, HIGOAT), and that the , HIANSR). The new instruction, CLC, stands for "Clear Carry" and it means that the Carry bit should be set to 0. It should always be used with all additions except chained additions like this one.

The code does the following: first it adds the two low values. If the addition resulted in a wrap-around (result greater than 255), then the Carry bit was set; otherwise, it was cleared. Then it performed the second addition, adding in the value of the Carry bit (That's why we call it "Add with Carry"). Thus, if a wrap- around occurred, an additional one was added into the high sum. This system insures that multi-byte addition works properly.

For subtraction, you use the SEC instruction ("Set Carry"). Otherwise, you handle subtraction the same way that you handle addition. In both addition and subraction, though, the low bytes must be handled first, then the higher bytes in the proper order (lower to higher).

DECIMAL & SIGNED ARITHMETIC#

There are two variations on standard 6502 arithmetic. Both are so rarely used that I will not treat them here. The first is decimal arithmetic using the Decimal flag. This allows you to set up an automatic decimal adjust mode. This is useful in certain types of arithmetic, decimal adjust mode. This is useful in certain types of arithmetic, primarily BCD arithmetic.

If you don't know what this is, don't bother with the Decimal flag. Your program should always begin with the instruction CLD, which means "Clear Decimal Flag". I will tell you this just once: failure to clear the decimal flag is the source of the most frustrating and impossible-to-trace bug in the 6502. Every single program should start with the instruction CLD.

The second arcane bit of 6502 arithmetic is signed arithemetic. It uses the V flag ("oVerflow"). Signed arithmetic is always confusing and seldom useful. In 7 years of working with the 6502, I have never had need of it. Don't bother.

LIMITATIONS ON 6502 ARITHEMETIC#

There are quite a few limitations on 6502 arithmetic. There is no facility for multiplication and division; you have to wirte subroutines to do that. You must design your programs to make do with 8-bit words; failing in that, you must use multi-byte arithemetic, with its consequent price in speed and TAM. All in all, arithmetic is a real pain on the 6502. This is the major reason why most 6502 programs do so little arithmetic.