This approach to assembly language is meant to give the reader a practical basic concept of what assembly language is and how it relates to the atari computer.

Chris Crawford is a self taught programmer that was interested in developing a computer wargame (s) simulation. Here he approaches learning assembly language in an informal light-hearted way with personal comments and suggestions, throughout the technical material.

Download Disk: CHRIS CRAWFORDS - ASSEMBLY LANGUAGE COURSE/ASSTUTO1.ATR

The material is covered in 8 sections consisting of these headings

---

Table of Contents

WHY LEARN ASSEMBLY LANGUAGE?#

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 that it 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-side give one byte of RAM. Eight adders make an eight-bit adder. We can thus create a RAM module by building many bytes 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 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 exactly the same thing. Unfortunately, the computer cannot read the assembly code, only the machine code. Therefore, 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 a 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 I=1 to 1,000,000,000...
EDIT PROGRAM
ASSEMBLE PROGRAM
DEBUG PROGRAM
NEXT I
```

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 outright value, such as $0600, or a symbolic reference, such as FISH, where 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 statement; 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 read data 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.

6502 ARITHMETIC 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 numbers 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 numerals 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 destroys the old value in the accumulator. 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 chequebook 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 until 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 or 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), the second two-byte value is called (LOFISH, HIFISH), and the two-byte result is (LOANSR, 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 wraparound (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 wraparound 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 subtraction, 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, 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 arithmetic. 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 ARITHMETIC#

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

BOOLEAN LOGIC#

A great deal of programming involves the use of Boolean logic. This is a standardized system for handling logical manipulations. It's sort of like algebra for logic. You must understand Boolean logic if you are to write assembly language programs, so let's get started.

Where algebra deals with numbers, Boolean logic deals with propositions. A proposition is just a statement such as "Fred eats worms". It can take only two possible values -- True or False. In our programs we seldom bother with broad and glorious propositions such as "Love is the universal language of truth" or "War is the extension of policy by other means". Instead, we normally deal with propositions such as "The joystick trigger has been pressed", or "There is a diskette in the disk drive".

When we use Boolean logic with a computer, we may think in terms of true and false, but the computer is actually working with 1's and 0's. We use the following convention: a 1 corresponds to a Boolean value of "true", while a 0 correspons to a Boolean "false". Using this system we can represent propositions inside the computer. However, programming requires more than the mere representation of data; we must also be able to manipulate that data. This brings us to the Boolean operators. There are four common Boolean operations necessary for most programming practices.

NOT

This is the simplest of Boolean operators. It takes a single Boolean value as an input and produces as its output the logical converse of the input. Thus, a true input yields output, while a false input generates a true input.

OR

This Boolean operator takes two Boolean values as its input and generates a single Boolean value as its output. The value of the output depends on the values of the inputs according to the following rule: If one input is true OR the other value is true, then the output is true. Otherwise, the output is false.

AND

This Boolean operator is just like the or-operator, except that it uses a different rule. Its rule is: If one input is true AND the other input is true, then the output is true; otherwise the output is false.

EXCLUSIVE-OR (XOR)#

This Boolean operator is just like the or-operator, except that its rule is: If one input is true, OR the other input is true, BUT not both are true, then the output is true; otherwise, the output is false.

When we use the 6502 for Boolean operations, you must remember that the operations are eight bits wide. Instead of working with one bit at a time, we use all eight bits of a word in parallel. The bits in a byte are independent and do not affect each other in any way -- at least as far as Boolean operations are concerned. The 6502 has three instructions for performing Boolean operations. These are AND, EOR, and ORA. The first performs an and-operation. For example, consider the following code:

```
LDA FISH
AND GOAT
```

This will first Load the accumulator with the value of FISH. It will then And the contents of the accumulator with the contents of GOAT. The result of the and-operation will be left in the accumulator.

The AND instruction can use an immediate operand if you desire, just as the ADC-instruction can. The EOR instruction provides the exclusive-or operator. It works just like the AND-instruction.

The ORA instruction provides the OR-operator in just the same way. If you wish to obtain the NOT operation, just use EOR #$FF; this will invert each bit in the accumulator. Because NOT is so easily reproduced with EOR, there is no special NOT instruction in the 6502.

APPLICATIONS OF BOOLEAN LOGIC

If you have any sense at all, you are probably asking, "What good is all this Boolean nonsense? What would I use it for?" Four applications are available:

PROGRAM LOGIC

Many times our programs encounter rather complex logical situations. The program must be able to load a file; if the FMS is in place and there is a diskette in the disk drive, and the diskette has the file we are looking for, or the file specification calls for a cassette load, then we will load the program. Many programming problems involve such Boolean operations, Keeping them straight is certainly a headache.

MASKING BITS

Sometimes we need to isolate particular bits in a byte. For example, in Eastern Front (1941) I used the character value to store the unit type. The color of the unit was encoded in the upper two bits of the byte, the type in the lower six bits. If I wanted to get only the unit type, I had to mask out the upper two bits. This I did with the following code:

```
LDA UNITCODE
AND #$3F
```

The AND-instruction eliminated the upper two bits, leaving me with just the unit type. Bit-masking like this is useful in many situations. We use it frequently when we pack bits into a byte to save memory. It is also handy with input handling. If you want to read the joystick port, you frequently mask out the bits in turn to see which is active. By the way, you mask out bits set to 1 with the AND-instruction. You mask out bits set to 0 with the ORA instruction. The logic is reversed.

SETTING AND CLEARING INDIVIDUAL BITS

We also use the AND and ORA instructions to set or clear individual bits within a byte. This is most often useful for handling arrays of flag bits. This little fragment of code will fold bytes together:

```
LDA FISH
EOR GOAT
AND MASK
EOR GOAT
STA ANSWER
```

This is a magical piece of code. See if you can figure out what it does. Experiment with two values of MASK: $OF and $FO.

SHIFT AND ROTATE INSTRUCTIONS

The 6502 also has instructions that allow you to shift the bits around inside a byte. The first of these are the shift instructions. One, ASL, shifts a byte to the left; the other, LSR, shifts a byte to the right. Thus, the byte %01101011, when shifted left, becomes %11010110. Each bit is shifted one position to the left. The leftmost bit is rudely pushed right out of the byte and falls away ("Aaaaaaaaarrrrrggggg!"). A zero is shifted into the rightmost bit. The LSR instruction does the same thing in the opposite direction. Note that ASL also doubles the value of the byte, while LSR halves it. Two ASL's multiply by four; three multiply by eight. This makes it easy to do simple multiplication, but be careful with round-off error here.

What happens if you try to multiply by 256? What do you get if you halve 3? A variation on the shift instructions are the rotate instructions. There are two: rotate left (ROL) and rotate right (ROR). These function just like the shift instructions, except that the bit that gets shoved into the bottom is not necessarily a zero; it is the contents of the Carry bit. The bit that gets pushed off the edge of the byte goes into the Carry bit, so it is not lost. Thus, if you rotate either way nine times, you'll be right back where you started.

Rotate instructions are a handy way to get a particular bit into the carry bit where you can work on it. Conversely, once you get your desired bit into the carry bit the way you want it, you can put it back into a byte with some rotate instructions.

INCREMENT AND DECREMENT INSTRUCTIONS

The last instructions I will cover are the increment and decrement instructions. These allow you to add one (increment) or subtract one (decrement) from a memory location. These are not considered to be arithmetic operations so they do not affect the Carry flag, nor are they affected by it. You cannot increment or decrement the accumulator, only RAM locations.

BRANCHING#

One of the most important ideas in computing is the concept of conditional execution. This is the ability of the program to execute different routines depending on conditions at the time of execution. The significance of this capability is best realized by considering how programs would operate in its absence. A program without conditional execution would not be able to change its program flow in response to conditions.

In other words, it would always execute exactly the same code in exactly the same order. Every run of the program would follow exactly the same sequence and perform exactly the same operations. Not very interesting, right? To get a grip on conditional execution, we need to look at it in its simplest expression. The simplest type of conditional execution is binary in nature. We have a chunk of code; the 6502 will either execute it or it will not execute it. The decision is made on the basis of a boolean value; a true value will tell us to execute the chunk, while a false value will tell the 6502 not to execute the chunk.

The basic mechanism for doing this is through an instruction that performs a transfer of control. This involves nothing more than altering the program counter. You may recall that the program counter is a register in the 6502 that points to the address of the currently executed instruction. When that instruction has been executed, the program counter is increased by the length of the instruction (1,2, or 3 bytes, depending on the instruction). It now points to the next instruction.

This little system allows the 6502 to step through a program in sequence. But there are also instructions that will alter the value of the program counter, alowing the 6502 to jump to another area of memory and another part of the program. The simplest of these is the JMP instruction. It takes the form JMP LABEL. This loads the value of the LABEL into the program counter. Its effect is to make the 6502 jump to the address of LABEL and continue execution from there.

It is directly analogous to a GOTO instruction in Basic. For conditional execution we need something more. We need the 6502 to have capability to make a binary decision based on a binary value. The solution used by the 6502 involves flags. These are single-bit Boolean values stored together in a single byte of the 6502 called the processor status register(SR). The status register is eight bits wide but stores only seven flags. These seven flags are labelled N,V, B, D, I, Z, and C. You have already encountered the C(Carry) flag and the D (Decimal) flag. In this chapter, we are concerned only with the N, V, Z, and C flags. The magic instruction that makes possible conditional execution can take many forms. Its general form is Bfv LABEL. The B stands for "branch".

The "f" stands for a flag, and the "v" stands for the value of the flag, either true or false. However, in this case, we do not use the terminology "true or false". Instead we use the terms "set" or "clear". "Set" means the same thing as "1" or "true", while "clear" means "0" or "false". The label is the address to which the 6502 should branch if the condition is satisfied. If the condition is not satisfied, then the 6502 will simply skip this branch instruction and go to the following instruction.

For example, suppose that we have the following instruction sequence:

```
LDA #0
BCS KAREKIA
LDA #5
```

```
KARELIA STA FISH
```

This will first load the accumulator with a zero. Then the 6502 encounters the BCS ("Branch on Carry Set") instruction. It looks at the Carry flag. If this flag is set then the 6502 will indeed branch to the label KARELIA. (For all you geography buffs, Karelia used to be in Finland). In other words, if the Carry flag is set, the 6502 will skip over the LDA #5 instruction.

Thus, a zero will be stored into FISH. However, if the Carry flag is clear, then the 6502 will not take the branch. It will instead continue executing the next instruction, which will load a 5 into the accumulator. Then it will come to the label KARELIA and store that 5 into FISH. Thus, the value of the Carry flag determines whether a zero or a five is stored into FISH.

The converse of BCS is BCC ("Branch on Carry Clear"). This will cause the 6502 to take the branch if the Carry flag is clear. There is also a pair of similar instructions for the V-flag. These are BVS and BVC. They will cause the 6502 to branch on the value of the V-flag. Now the situation gets unncessarily confusing. The istructions for the Z-flag should be BZS and BZC -- "Branch on Z Set" and "Branch on Z Clear". Unfortunately, the dumb designer of the 6502 thought he would get cute at this point, so instead he called these instructions BEQ and BNE, for "Branch on Equal" and "Branch on Not Equal". He never mentioned what he thought is supposed to be equal to what. (Remark from Editor: the 6502 designers were not dumb: in Assembler you often compare numbers by substracting them. That is also how the CPU internally works. if A=10 and B=10 and you do A - B the result is 0. So the BNE/BEQ react on the zero flag because this ment that the comparison was equal (Z flag set) or non-equal (Z-flag not set)

We're stuck with it, so make the best of it. Just remember what these instructions really mean BZS and BZC. If you think in terms of the Z-flag, it will work out just fine. If you try to think in terms of equal or not equal, your attention will be diverted from the real truth of the matter and you may make mistakes. So keep your eye on the ball and think in terms of Z! The next pair of branch instructions use the N-flag. These are even more insidious than the previous two. They are called BMI and BPL, meaning "Branch on Minus" and "Branch on Plus". At first glance, these appear to be reasonable substitutions for BNS and BNC. After all, if you load the accumulator with a signed number, and the number is negative, then the N-flag will be set, while if the number is positive, the N-flag will be clear. Thus, it would seem that BMI is truly equivalent to BNS and BPL is truly equivalent to BNC. This is the source of many a bug in beginner's programs. Consider the following fragment of code:

```
LDA FISH
SEC
SBC BOAT
BPL POSANSR
```

This code is supposed to branch to POSANSR if FISH is greater than GOAT. And indeed, if FISH is greater than GOAT, then when you subtract GOAT from FISH, you will get a positive result, right? Not necessarily! Suppose, for example, that the value in FISH is $C1 and the value in GOAT is 1. When the 6502 subtracts GOAT from FISH, it will get a result of $C0. Note that the highest bit of $C0 is set to 1. This is the value that will go into the N-flag. In other words, even though FISH is greater than GOAT, the 6502 will not take the branch, and this code will fail. The moral of his tale is, don't take those instructions literally. They are misleadingly named. When you see BPL, don't think "Branch on Plus", hink "Branch on N Clear". (Remark from editor: again, I would say Chris is wrong here. The plus and minus is coming from interpreting an 8bit value as an signed integer value (+128 to -127). All negative values have the high bit set and will be seen as negative)

Otherwise, you'll screw up someday. By the way, the correct branch to use in the above problem is BCS. Now for a catch with the branch instructions. A JMP instruction is a simple absolute jump -- you specify the target address and it goes there. The designers of the 6502 realized that the vast majority of branch instructions only go a short distance. They therefore decided to implement the branch instruction as a relative branch. The machine code doesn't specify the target of the branch,

it only specifies an offset. In other words, instead of saying, "jump there", it says, "jump so many bytes forward or backward". The allowable range is 126 bytes forward or backward. Thus, you can't branch anywhere you want, only to nearby locations. If you must branch further, reverse the logic of the branch and use the branch to skip over a JMP statement.

INDEX REGISTERS & LOOPING

We are now going to expand the model of the 6502 that you have been using. Until now, the 6502 I have described had nothing more than a status register, program counter, and accumulator. Now I am going to reveal the existence of two new registers in the 6502: the X- and Y-registers.

These two registers are eight-bit registers just like the accumulator. You can load numbers into them and store them out just as you can with the accumulator. You cannot do arithmetic or Boolean operations with them as you can with the accumulator. But you can do a number of very special things that greatly increase the power of the 6502.

Let's start with the simple move instructions. The first are LDX and LDY, which load the X- and Y- registers the same way that LDA loads the accumulator. Then there are STX and STY, which store the X- and Y-registers the same way that STA stores the accumulator. There are also four commands for transferring bytes between registers; these are TAX (transfer A to X), TAY (transfer A to Y), TXA (transfer X to A) , and TYA (transfer Y to A).

Then there are four special instructions that you will use very often. These are INX and INY, which increment (add one to) the X- and Y-registers, and DEX and DEY, which decrement (subtract one from) the X-and Y-registers. Finally, we have the CPX and CPY commands, which compare X or Y with the operand of the instruction. These two instructions operate in exactly the same way that the CMP instruction works, except that they use the X- and Y-registers instead of the accumulator.

What are these two registers used for? Well, they are sometimes used as temporary registers. If you are in the middle of a lengthy computation, and you need to save a value currently in the accumulator to make room for something else, the X- and Y-registers are a handy place to stuff values away for temporary storage. Programmers do this all the time.

However, temporary storage is not the real purpose and value arise from their utility as index registers. Index registers go hand in hand with loops; the best way to show you how they are used is to dump the whole schmeer at once and then explain it. So consider the following problem: your program has to deal with the possibility of user errors. Suppose you require the user to type in a file name for your program to read. What happens if this file is not on the disk? You have to put an error message on the screen that says, "FILE NOT ON DISK!" How do you print the message? Here's a sample bit of code that will do it:

```
      LDX #(ENDMSG-ERRMSG-1)
LOOP1 LDA ERRMSG,X
      STA SCREEN,X
      SEC
      SBC #$20
      DEX
      BPL LOOP1
      JMP ELSWHR
ERRMSG DB 'FILE NOT ON DISK1'
ENDMSG DS 1
```

Let's take apart this code and explain it step by step. First thing we do is load the X-register with the number of characters (minutes one) in the message. The expression (ENDMSG-ERRMSG-1) will calculate that length at assembly time. This turns out to be 17 characters long. If we were pedestrian

about it we could have just written LDX #16, but this way, if we decide to change the message we don't have to remember to go back and change the LDX command. Neat, huh?

OK, so now we have a 16 in the X-register. Now the 6502 comes to the next command -- LDA, ERRMSG, X. This command tells it to load the accumulator with the byte at (address ERRMSG, indexed by X). What this means is as follows: the 6502 will take the address ERRMSG and add the value of the x-register to that address. It will then go to the address so calculated and load the accumulator with the contents of that address. Since X contains a 16, the 6502 will go to the 16th byte after the first byte in the table ERRMSG. If you count characters, you will see that the 16th byte is the exclamation point. So the 6502 will load the ASCII code for an exclamation point into the accumulator.

The next two instructions (SEC, SBC #$20) are necessary to correct for the Atari's nonstandard handling of ASCII codes. They make sure that the exclamation will be printed on the screen as an exclamation point. The next instruction (STA SCREEN,X) stores the reslt indexed by X. The 6502 will add the contents of X (still 16) to the address SCREEN. It will then store the contents of the accumulator into that address. If that address is part of screen RAM, then you will see an exclamation point appear on the screen.

The next instruction that the 6502 encounters is the DEX instruction. This instruction subtracts one from the X-register, making it a 15. Next, the 6502 comes to the instruction BPL, LOOP 1. This will branch if the N-flag is clear. The vaue of the N-flag is affected by a DEX instruction. The value of bit D7 of the result is transferred to the N-flag. Bit Dd7 of 15 is a zero, hence the N-flag is clear, hence the 6502 will indeed take the branch. Note that it branches back up to LOOP 1.

Now it will repeat the process, only this time X contains a 15, not a 16. It will therefore grab the 15th character, an ASCII 'K', and store that to the screen position just before the exclamation point. Then it will subtract one form X to get a 14, and will continue the loop.

This process will continue, with the 6502 grabbing bytes in reverse order form the table and storing them onto the screen, until after the 6502 does the seroth byte. When X contains a zero, and the 6502 executes a DEX, it obtains the result $FF. This sets the N-flag. When the 6502 encounters the BPL command, it will NOT take the branch; instead, it will skip the branch and go on to the JMP statement. The loop is terminated. In this one fragment of code you have seen two major ideas: indexed addressing and looping. They are so closely related that it is hard to talk about one without talking about the other.

You can use indexed addressing with either the X-register or the Y-register. You most commonly use indexed addressing with the LDA and STA commands, but you can also use it with many of the other 6502 commands: ADC, SBC, CMP, AND, ORA, EOR, LSR, ROR, ADL, and ROL can all be used with indexed addressing. Indexed addressing allows you to work with tables or arrays of data. There is one ugly catch: all of your arrays must be less than 257 bytes long, because the index registers are only eight bits wide.

Most of the time this is not a serious problem. However, if you must address a larger table or array, you can use indirect addressing. To do this, you calculate the address that you desire to access, store that address in two contiguous bytes on page zero (low, then high) -- we call these two bytes a pointer -- and then refer to the pointer like so:

LDA (POINTER), Y This instruction will take the address out of pointer, add the value of Y to it, and load the accumulator with the contents of the address so calculated. If POINTER contains $4567 and Y contains a 2, then the 6502 will load the acumulator with the contents of address $4569. You are still restricted by the size of Y, but you can always go back and change the POINTER if you need to span larger arrays. In this case, you frequently just leave Y equal to zero and do all of your indexing directly with changes to POINTER.

The last topic I will take up is termination techniques. Every loop must somehow be terminated if you are to avoid the problem of the Sorcerer's Apprentice. You will note that the programming example I gave used a rather odd approach. I started at the end of the array and worked backwards. Why not start at the beginning and work forwards? It's slightly more efficient going backwards than forwards. When you go forwards, you have to terminate the loop with:

```
INX
CPX #17
BNE LOOP1
```

Whereas when you go backwards, you need only use:

```
DEX
BPL LOOP1
```

Going backwards you save one instruction. However, if this confuses you, feel free to count forward; that works, too, only it's a little less efficient.

There is also a problem on choosing whether to BNE or BPL. BPL restricts you to a range of only 127 bytes, but BNE allows you a range of 129 bytes by starting the branch two bytes after the branch test byte(a reverse branch starts at the same place but is a minimum of three bytes to jump over the branch code itself). Just remember to start the index from ERRMSG-1 and SCREEN-1 instead of ERRMSG and SCREEN.

There are lots of other sneaky ways to terminate loops, but they fall into advanced topics.

SUBROUTINES & THE STACK#

We now take up the first topic in this series that is not absolutely essential to writing programs: subroutines. The loops and indexed addressing discussed in the previous lecture are truly essential: it is hardly possible to write a useful program that has no loops.

Subroutines are a matter of convenience, not necessity. It is quite possible to write an entirely adequate program without using a single subroutine. However, you will find that the convenience of subroutines with large programs is so great that you would never want to write such a program without them.

The primary purpose of a subroutine is to perform some function that is frequently needed at many points in the program. Instead of having to repetitively insert the same code over and over again, we simply write it once, place it in a subroutine, and call that subroutine many times from the main program. The use of subroutines dramatically reduces the size of a program.

Subroutines are implemented on the 6502 in a fashion very similar to that used by BASIC. You may recall the two BASIC commands for subroutines: "GOSUB lineno" and "RETURN". The two corresponding 6502 commands are "JSR label" and RTS". The label in "JSR label" is the label of the beginning of the subroutines. Thus, writing and using subroutines in 6502 is trivially simple. First, you write the subroutine. You give it a name (say, "MYSUBR") and stick that label in front of the first instruction. You put an RTS command after the last normal command of the subroutine. To call the subroutine, you just put JSR MYSUBR. That's all it takes!

However, in order to understand how it works is not so easy. Here's the problem we must solve when the 6502 jumps to a subroutine, the JSR instruction tells it the destination address to which the 6502 must jump. But when the 6502 hits the RTS instruction, how does it know the address to which it must return? The RTS doesn't say, "Return to THIS address"; it says only "Return".

Moreover, how could the 6502 know where to return? If the subroutine can be called from, say, five different points in the program, how would the 6502 know which of those points to which it

must return? What if we gave the 6502 a special register for remembering return addresses? That is, whenever the 6502 encounters a JSR instruction, it stores the current address into its return address register. Then when it encounters an RTS instruction, it simply takes the address out of the return address register. There is only one problem with this: what if we use nested subroutines (one subroutine calls another)? The second subroutine call will erase the return address for the first subroutine call. Trouble!

The solution to all this is called a stack. A stack is a chunk of RAM allocated for certain special operations such as subroutines. The 6502 stack is stored on page one -- that is, addresses $0100 to $01FF.

The stack operates like 128 return address registers arranged in sequence (remember: two bytes per address). The 6502 keeps a stack pointer register to keep track of which byte in the stack is currently being used. I will now trace through the operation of the stack in a subroutine. We start with the stack pointer set equal to $FF. That means that the stack is empty; the stack pointer is at the very top of the stack. The 6502 encounters a JSR instruction. It takes the current value of the program counter and breaks it into two bytes. It pushes the first byte onto the stack. This means that it stores the first byte at $01FF, then decrements the stack pointer. Now the stack pointer is $FE. The 6502 then pushes the second byte of the return address onto the stack, storing that byte at $01FE and decrementing the stack pointer to $FD. Then the 6502 jumps to the subroutine. When it encounters the RTS instruction, it pulls the two address bytes off of the stack (increments stack pointers and loads byte at address $0100,SP). Those two bytes go directly into the program counter, returning the 6502 to the original entry point.

The advantage of this approach is that it allows very deep nesting of subroutines. If one subroutine calls another, the 6502 simply stores more values onto the stack. The addresses won't be confused because you always exit subroutines in exactly the reverse of the order that you entered them. You can use the stack yourself, if you wish. You have six instructions that allow you to play with the stack: PHA, PLA, PHP, PLP, TSX, and TXS. The PHA instruction pushes the value of the accumulator onto the stack and decrements the stack pointer. The PLA instruction increments the stack pointer and pulls the current stack value into the accumulator. These two instructions allow you to store and retrieve values onto the stack. They must be exactly balanced, though, or you will generate that most feared of bugs, the stack crashes.

Consider: you are in a subroutine. You push a value onto the stack, but forget to pull it off. When the 6502 attempts to return to its original location, it pulls two address bytes off the stack -- but they're the wrong two bytes. One of them is the value you pushed but didn't pull. Result: the 6502 return to the wrong address. Your program goes haywire and the computer crashes. This is called a stack crash. This type of crash tends to the particularly difficult to recover from.

Prevention is the best medicine here. The rule for preventing stack crashes is simple and absolute: each and every push onto the stack must be balanced by one pull from the stack. Violate this rule and you will certainly experience a stack crash.

The next pair of stack manipulation instructions are PHP and PLP. These push and pull the process status register from the stack. They are useful for two purposes. First, you may wish to save the values of the various flags before performing some operation, then restore them so that you can branch on a previously created condition. Second, it is sometimes handy to PHP, than PLA to get the processor status register into the accumulator where you can more directly manipulate it. Again, each push must be balanced by one pull. The third stack manipulation pair of commands do not modify the stack. They are TSX and TXS. These transfer the stack pointer to and from the x-register.

Once in the x-register, you can change the value of the stack value and then TXS to jump over sections of the stack. This can be a very handy way to pass parameters to subroutines, but it is also very tricky. If you make a mistake, you will generate a stack crash. So be careful with this one. I have always avoided these commands like the plague. They are very dangerous and never essential.

INTERRUPTS#

We now approach one of the most difficult topics in the world of assembly: interrupts. This is such a messy topic that very few high-level languages make any provision for interrupts. Moreover, interrupts are one of the best ways around to crash your program hard. Programmers using interrupts must be very careful.

The standard way to handle this problem is with a technique called polling. Your program runs out every now and then to check whether the high-priority situation has arisen. If it has, then the program responds to it. If not, it returns to its original work.

The problem with polling arises from the choice of polling interval. If you choose a long(infrequent) polling interval, then you may not respond to a demand quickly enough. If you choose a short (frequent) polling interval, then you will respond quickly to the demand, but you will never have any time for your regular computations.

You may think this type if situation is infrequent, but I can list quite a few situations where this is fairly common. Most I/O operations involve short bursts of computation at infrequent intervals, but they must be attended to on a tight schedule. For example, talking to a cassette deck involves very little real work from the CPU, but they must be done according to a precise schedule. Even a disk drive is very slow by the standards of a 6502. Or how about keyboard response? When the human operator presses a button, he wants to see a response NOW, not two or three seconds from now. Yet he could press on that button at any time. So should your program sit on its hands waiting for a keypress, or should it ignore the human operator?

The solution to all of these problems is the interrupt. An interrupt is rather like a subroutine that can be called by a hardware action. There is a wire going into the 6502 called IRQ(Interrupt Request). That wire is normally quiet. But when something important happens, like a keypress, the computers' hardware puts a signal on that wire to interrupt the 6502.

Here's what happens next: The 6502 is busy running a program, but when it gets the interrupt signal it first checks the 1-bit (internal) in the processor status register. If the 1-bit is set, it decides to ignore the interrupt, but if it is clear, it proceeds to the next step. It saves the process status register and the current value of the program counter onto the stack.

Then it loads the program counter with the address stored a special place in ROM -- it's either $FFFC or $FFFE, I can never get it straight. It thus jumps to the address specified in ROM. It expects to find an interrupt service routine there, which presumably will deal with the keypress in the appropriate manner.

This routine will probably start by pushing the A, X, and Y registers onto the stack to preserve them. When done, the routine will the pull them off the stack and execute an RTI instruction, which causes the 6502 to pull the processor status register off the stack, and then pull the program counter off and resume operating.

The important thing about the rather complex sequence is that is allows the 6502 to drop whatever its doing, service the interrupt, and the return to its earlier functioning without skipping a beat. The overriding goal of all this is to be absolutely certain that, when the 6502 returns from the interrupt, it returns in EXACTLY the same state that it was in when the interrupt hit. Otherwise, all sorts of horrible, untraceable bugs would result(and the computer would never work).

Imagine -- you are in the middle of some huge computation when a interrupt strikes. It subtly changes some very tiny parameter, just enough to insure that when the computation resumes, it will be slightly incorrect. When you try to find the bug, you discover that sometimes the code works perfectly and sometimes it fouls up, and you can't figure out why it should do that. Very bad business! Moral: interrupts must follow a very tight discipline if they are to be of any utility.

Now let's get into some of the technical gore involving interrupts. First, there are two interrupts on the 6502. They are called IRQ (Interrupt Request) and NMI(Non- Maskable Interrupt). The idea is that the IRQ can be masked out by setting the 1-bit with the SEI instruction. Then you use the CLI instruction to clear the 1-bit.

Thus, IRQ is used for interrupts that have second priority. NMI is reserved for first-priority interrupts, it is not maskable. However, the designers of the Atari computers routed IRQ and NMI through the POKEY and ANTIC chips respectively. And they put mask registers into these chips. Thus, the NMI can be masked out after all -- but only on Atari computers. Other 6502-based computers don't allow that.

The NMI and IRQ interrupts have separate interrupt vectors in ROM, so they can be treated differently. These vectors route the interrupts to the OS, but the OS is smart enough to route interrupt flow thorugh some RAM locations. This means that you can intercept these two interrupts by altering the contents of the RAM-vectors. (I won't list them here, there are a number of them for different situations). You must be careful, though, when altering such a RAM vector. What happens if an interrupt strikes after you have changed one byte of the address and before you have changed the other byte?

The 6502 will fly off into never-never land and you have crashed. Sure, it's unlikely, but good programmers don't count on luck to make their programs work. You have to guarantee that the interrupt won't occur before you mess with the vector. Use SETVBV from the OS.

The two primary applications of interrupts with the Atari computers are for VBIs (Vertical Blank Interrupts) and DLIs (Display List Interrupts). These are very involved topics covered quite thoroughly in the book 'De Re Atari'. VBIs are most often used for animation control, input handling, and other time-critical operations. For example, the entire player I/O of my game Eastern Front (1941) is handled by VBIs. The scrolling, giving of orders, identifying units, and so forth is all done by VBIs. The mainline routine meanwhile figures the artificial intelligence. DLIs are used to enhance the graphics on the screen. You can get more colors, more use out of players, more scrolling, and more character sets with proper use of DLIs. Again, consult 'De Re Atari' for a full treatment of this complex subject.

Interrupts are extremely difficult to debug because they tend to crash the system when they fail. You must exercise the strictest discipline in writing interrupt code. Timing problems, seldom of concern in mainline programming, can become critical with interrupts. What happens, for example, if your interrupt service routine takes so much execution time that more interrupts arrive than you can service? Bad things, I assure you.

You must always ask yourself, what happens if an interrupt strikes here? Or there? You must assume that an interrupt will strike at the worst possible time, and write your code to deal with that possibility.

The most important discipline to follow in writing interrupt service routines is this: keep your interrupt database separate from your mainline database. If the ISR can freely write to variables used by the mainline, you will certainly have problems when the mainline attempts to work with variables whose values change in unpredictable ways. You must set up ironclad rules about when the ISR can mess with variables used by the mainline, what it can do to them, and how it notifies the mainline routines that it has indeed altered them.

Approach interrupts with extreme caution. They are very powerful, but every programmer can tell you horror stories about debugging interrupt routines.

ADVANCED TOPICS#

We have covered all of the traditional material associated with 6502 assembly language PROGRAMMING. However, there remain a number of topics that should be addressed before we finish. They are not closely associated with each other, so I will take them in random order.

The first topic is perhaps the most difficult one for a beginning assembly-language programmer: Where do I begin? How do I put together an entire assembly language project?

The problem here is seldom a technical one. Most beginners are stopped by their own lack of goals rather than any lack of technical expertise. One does not just write an assembly language program because one knows assembly language -- that is putting the cart before the horse. One starts with goals and then considers means.

A story from my early days with micros will illustrate this point. I did not have anybody to teach me assembly language. I decided in 1976 that I wanted to do wargames on computers. Accordingly I bought a KIM-1, an early 6502-based single-board computer. I received it in January 1977. I studied the manuals and taught myself 6502 machine language. I had my first wargame up and running in six weeks. That means that I not only taught myself 6502 in six weeks, but I wrote and debugged a program at the same time.

Now, the point of this story is NOT "Wow, isn't Chris Crawford the smartest programmer who ever lived!" The point of this story is that goal-oriented learning is far more effective than goal-less learning. Had I sat in on some technical course on 6502, I would have taken months and months to learn the material. Because I had a clear goal, I learned very quickly.

My advice to you, the beginning assembly programmer, is this: You have acquainted yourself with the rudiments of 6502 programming. If you have some project you would like to pursue, some goal you would like to achieve, then do it. If not, don't waste your time trying to use a tool for its own sake.

Assuming you pass this first test, there remains the broad problem of organizng your assembly language program. I suggest that you break your program up into six modules, each forming a separate source code file. These six modules would be:

EQUATES file: this file defines all of the equates used by the program: the data areas, the page zero and page six usage, and perhaps some of the large graphics and screen structures.

DATA file: this file contains all of the static tables used by the program. This would include all the text messages that would be printed onto the screen, bitmaps of graphics images, graphics character set definitions, and so forth.

INITIALIZATION code: this file contains the routines that initialize the program when it first fires up. They set up the screen, clear out all the special graphics and sound registers, zero out all the arrays that need to be cleared, and do all the other legwork associated with clearing the decks for a program.

INTERRUPT code: this module contains the code associated with any interrupts used by your program. This would most commonly involve vertical blank interrupts and display list interrupts. In as much as your interrupts should be well-separated from your other code, you might as well keep the code in a separate file.

MAINLINE code. This includes the main program loop that controls the primary behavior of the program. If you have problems imagining this, think of it as nothing more than a series of subroutine calls arranged in a loop, with each subroutine handling one chunk of the overall process.

SUBROUTINE code: After a while you build up a collection of subroutines for handling standard processes in the program. Keep them here.

The second topic I would like to talk about is the place of the 6502 in the larger world of microprocessors. The 6502 is undoubtedly the most successful of microprocessors to date, having been installed in more systems than any other microprocessor. It is also a very old microprocessor, having first appeared in 1976. That makes it quite old.

A very simple way to approach the world of microprocessors is to group them into two sets -- the Sixes and the Eights. The Eights represent the earliest group of microprocessors, they trace their lineage all the way back to the 4004, the first microprocessor. The 4004 was followed by the 8008, the first eight-bit microprocessor. The 8008 was superseded by the 8080, which was in turn followed by the Z-80.

The Z-80 was the most advanced eight-bit processor in the Eights line. The next step was to go to 16 bits with the 8088 and 8086. These were followed by the more powerful 80186, 80286, and 80386.

The fundamental philosophy of all the Eights can be expressed in two words: features and compatibility. The designers of the Eights were always adding new features to the microprocessors with each successive generation. The goal seemed to be to pack as many bells and whistles in as would fit. The second goal, compatibility, meant compatibility with the previous microprocessor in the series. This insured that software developed for previous versions would still run on the newer versions.

The result of this design philosophy was a series of powerful microprocessors that were quite complex in layout and rather difficult to learn. The features were piled up on each other in a bewildering array. Once you learn the system, it seems natural enough. But it is something of a mess.

The Sixes include the 6800, the 6502, the 6809, and the 68000. The two key words guiding the design of the Sixes are cleanliness and speed. The idea was to make the instruction sets clean, powerful, and fast. The hope was that the processors would be so easy to learn that compatibility would not be a problem. The design approach was to use just a few simple instructions, but give them variations that greatly extend their power. Thus, the 6502 has a LDA instruction that can be used with a great many addressing modes.

The 68000 is the 32-bit entry into the Sixes line. It carries the idea of cleanliness even further than the 6502. The 68000 uses a single instruction with different modes to replace the 6502 instructions LDA, LDX, LDV, STA, STX, STY, TXA, TAX, TYA, PHA and PLA. That's quite a simplification!

The 68000 also boasts sixteen registers, each 32 bits wide. That's a total of 512 bits of register space, the 6502 has 32 bits of equivalent register space. Those sixteen registers eliminate many of the data-shuffling problems so common with the 6502.

The 68000 has a linear address space 24 bits wide -- that's sixteen megabytes! Thus, a 68000 can directly address 16 megabytes of RAM and ROM. The 6502, by contrast, can only address 64K directly -- it must use paging systems that slow it down to address more memory.

Finally, the 68000 has a number of advanced capabilities that make possible a number of special capabilities. I will describe just one stack frames. The 68000 makes it easy to set up a local, temporary stack when you enter a subroutine. Thus, subroutines can have their own local variables stored on the stack, accessed via a special stack pointer register. The 68000 will manage all the housekeeping necessary to keep such a system straight.

Did I mention that 68000 has hardware multiply/divide?