Supplementary Notes By: M.J.Malone

Starting to Program in 6502 Assembly Code

Quotes of Murphy's Kin:

1) It is impossible to make anything fool-proof because fools are so ingenious.

2) If you explain something so thoroughly and clearly so that everyone will understand, someone will not understand.

Introduction to the 6502

Accumulator and Assembler Basics

The 6502 has several 'registers'. Registers are special memory locations that are internal to the processor and are directly involved in processor instruction codes. The accumulator (.ACC) is one such register. The code fragment that would transfer data from one memory location (A_loc) to another (B_loc) would be coded as: (Note that the numbers $0400 and $0800 are hexidecimal numbers which do have decimal equivalents. The conversion between hex, decimal and binary will be explained later.)

```
;
A_loc = $0400
B_loc = $0800
;
  LDA A_loc
  STA B_loc
;
```

Where LDA is the mnemonic for LoaD Accumulator and STA is the mnemonic for STore Accumulator. In this example the two assignments 'A_loc=' and 'B_loc=' give values to the labels A_loc and B_loc. Note that A_loc and B_loc are NOT variables but labels or constants used to replace numerical values and make the code more readable. The locations described by A_loc and B_loc will contain data important to the running of the program, the variables. Now A_loc and B_loc are not very descriptive but 'Abs_Value' and 'Buffer_ptr' are also legal labels. The assembler sees the above code fragment as:

```
;
  LDA $0400
  STA $0800
;
```

with the labels substituted into the code. This fragment loads the value from address $0400 into the accumulator and then stores the

value into memory address $0800.

You may think of labels 'Buffer_ptr' as a variable as in a high level language when you go 'LDA Buffer_ptr' to get the value of the variable into the accumulator. Remember that the assembler sees Buffer_ptr as an address of a location in memory, it is the programmer who decides that that location has a special meaning or purpose and gives it a meaningful name.

The .X and .Y Registers and Indexing

The .X and .Y registers are similar to the .ACC in most respects, data can be loaded into them and stored into memory from them. The above example could as easily be programmed with LDX and STX as it was with LDA and STA as follows:

```
;
A_loc = $0400
B_loc = $0800
;
  LDX A_loc
  STX B_loc
;
```

The same code could also be written for the .Y but there is a special purpose reserved for the .X and .Y registers. In the previous section we saw that labels were used to name important memory locations and simulate the function of variables for the programmer. The assignment 'b=a' in a higher level language may be coded using the example above. If the programmer wanted to use an array and say assign 'b~[3]=a' it may be done as follows:

```
;
A_loc         = $0400
B_array_start = $0800
;
  LDA A_loc
  LDX #3
  STA B_array_start,X
;
```

As before, the value of memory location 'A_loc' is loaded into memory. The value of the array index '3' is loaded into the X register. The statement 'STA B_array_start,X' takes the value in the memory and stores it into the location (B_array_start+.X). Here B_array_start is the first address for the array and .X acts as the offset INDEX into the memory space that follows. This is called simple indexing and can be used to store and recall data that is organized into tables.

The 6502 as an 8 bit Processor

For programmers accustomed to high level languages often the assembly language of processors seems very limiting. So far the implementation of higher level language assignment statements has seemed to be as simple as a few substitutions, a few LDA and STA to

get the job done. Unfortunately this is about the easiest high level language concept to transfer to assembly and then only in very restricted cases.

The 6502 is an 8 bit processor and all memory fetches (LDA etc) read values from 8 bit memory locations. All numbers stored in the accumulator are in the range of 0-255 or $00-$FF. The .X and .Y registers are also 8 bit as is the .SP the stack pointer for the 6502. This limits the stack to 256 memory locations (located between addresses $0100-$01FF). The program counter is a 16 bit number which varies between $0000 and $FFFF. The address space is defined as the range of memory locations that can be pointed to by the program counter. In the case of the 6502, 16 bits, $0000-$FFFF, 0-65535 represents an address space of 64K, where 1K is defined as 1024 for binary applications.

Number Conversions

The conversions between hex and decimal representation can be done however students should be aware that since the 6502 is an 8 bit machine, everything is much more convenient in hex. For instance the address $8000 is immediately recognizable as the first address in the upper half of memory (because it begins $80), at the beginning of a memory page (because it ends in 00). The equivalent number in decimal 32768, unless memorized for what it is, cannot be recognized as easily. Similarly it is necessary when dealing with I/O ports to control individual bits of a memory location. In this case the binary representation of %1010000 is immediately recognizable as bits 7 and 5 of the byte set whereas the meaning of the equivalent decimal number 160 is not nearly so clear.

It is advisable to use hexidecimal numbers for addresses, binary numbers where bit on/off states are important. Decimal numbers should be used only when 'magic' numbers are such as 26 letters in the alphabet, 24 hours in a day, 30 days in a month etc that we are accustomed to seeing as decimal are needed. Note that numbers 9 or smaller are the same in hex as decimal so there is no problem for small constants.

The conversion between hex, binary and decimal will most likely be done on students hand calculators however recalling what you learned in grade 3: There are 10 symbols for digits in the decimal number system 0-9. All numbers are stated with these 10 symbols (note the number 10 is expressed in decimal). Numbers larger than 9 are stated using ten's and unit's digits:

98 = 9*10 + 8

or more generally:

34256 = 3*10^4 + 4*10^3 + 2*10^2 + 5*10^1 + 6*10^0

Hexidecimal numbers are represented with 16 symbols ($10 = 16 in hex), the symbols 0-9 and the letters A-F. As units digits, 0-9 correspond to the decimal numbers 0-9, ie: 5 = $5 etc. As units digits, A-F correspond to the decimal numbers 10-15. A hexidecimal number can be expressed:

$7A3F = $7*$10^3 + $A*$10^2 + $3*$10^1 + $F*$10^0

So far this is parallel to the representation of a decimal number as above. To perform a conversion you would simply convert the hexidecimal numbers in the expansion to decimal.

$7A3F = 7*16^3 + 10*16^2 + 3*16^1 + 15*16^0 = 31295 decimal

A similar operation can be done in binary but once the meaning of units, ten's and hundreds digits is understood then the expansions of binary numbers are as easily calculated as for hexidecimal numbers.
Often conversions between hexidecimal and binary or the reverse are necessary. Similar expansions as above could be performed but there is a short cut. Noting that 2^4 = 16 and 2 is the base of the binary system and 16 is base of the hexidecimal system, groups of four binary digits must somehow correspond to hexidecimal digits. Observing that:

```
        %0000 = $0        %1000 = $8
        %0001 = $1        %1001 = $9
        %0010 = $2        %1010 = $A
        %0011 = $3        %1011 = $B
        %0100 = $4        %1100 = $C
        %0101 = $5        %1101 = $D
        %0110 = $6        %1110 = $E
  %0111 = $7        %1111 = $F
```

The conversion of multidigit hexidecimal numbers becomes straight forward.

$7A3F = $7,A,3,F = %0111,1010,0011,1111 = %0111101000111111

This is very useful when hardware tests have to be done. If the program counter of the 6502 were pointing at address $7A3F then

the binary representation would be useful for understanding the voltage values on the address pins of the 6502 chip. In digital circuits, 5 volts corresponds to logic 1 or binary 1; 0 volts corresponds to logic 0 or binary 0. The binary representation of the address %0111101000111111 is also the logic levels for the address lines Adr15-Adr0 and hence the voltages on the address lines would read (counting from Adr15 to 0): 0V, 5V, 5V, 5V, 5V, 0V, 5V, 0V, 0V, 0V, 5V, 5V, 5V, 5V, 5V and 5V. Summary of 6502 Basic Structure

By now you should be realizing that operation of digital computers are really not so mysterious at all. True the internal workings of a 6502 are very complex, it is not hard to imagine it being made up of a great number of smaller blocks that interpret voltages as logic levels and perform logical operations. The acculumator is in fact eight circuits that can each hold a voltage of 0 or 5 volts, organized into a unit that is routed out of the processor to memory by the STA instruction for instance. Since the actual operations, when reduced to voltages and switching transistors are very simple, they are very fast. Even the 6502, a relatively slow processor can perform a STA operation in as little as 3 microseconds or perform 333,333 such STA's in one second of

operation in the slowest 1 MHz implementation.
We have been introduced to the accumulator and the .X and .Y index registers. As mentioned before the system 8 bit stack pointer .SP points into an area of memory from $0100-$01FF. For now suffice it to say that the stack is very important to the operation of the processor and the actual instructions that involve the stack will be discussed later. The program counter is a 16 bit register that points to the instruction in memory that is currently being executed. The processor status register which has not been discussed until now, is a group of flags that represent the current state of the processor and the results of the last calculation performed.

Classes of 6502 Instructions

The reader will be introduced to several classes of 6502 instructions. Some instructions require arguments and some do not and this is indicated where appropriate.

Data Transfer Instructions

There are several instructions in the family of data transfer instructions but basically they fall into two categories, those which move data between memory and registers and those that move data between registers.

```
 LDA, LDX, LDY  arg    Load 'arg' into Accumulator, X or Y registers
 STA, STX, STY  arg    Store the Accumulator, X or Y registers to memory 'arg'
 STZ            arg   *Store the value zero into a memory location 'arg'

     TAX, TAY           Transfer .A to .X, .A to .Y
     TXA, TYA           Transfer .X to .A, .Y to .A
     TXS, TSX           Transfer .X to .SP, .SP to .X

     *65C02 Only
```

These instructions are very straight forward, they move an 8 bit value ($00-$FF) from the one place to another. After the instruction, the data is in two places, where it was and where it was moved to.

Go to Instructions

The go to instructions are as follows:

```
JMP  dest          Jump, Set the Program Counter = 'dest'

JSR  dest          Jump to a Subroutine, saving the return address
RTS                Return from subroutine
```

The JMP statement is a pure 'go to' which simply sets the program counter to a new address and continues the execution of the code at that point. The JSR statement first takes that current program counter position and pushes it into the stack and then jumps

to the new address. When the RTS instruction is encountered at the end of the subroutine, the previous program location is pulled from the stack and the processor continues after the subroutine call.

Compare Instructions

So far we have examined ways of moving data and unconditionally changing the path of the program. We will now look at ways to conditionally change the path of the program. In machine language this is divided into two operations, a comparison and a branch. There are three compare statements:

```
CMP arg        Compare the .A
CPX arg                   .X
CPY arg                   .Y  to  'arg'
```

The results of these comparisons are recorded in the system flags N-negative, Z-zero and C-carry for later use in branch instructions.

Branch Instructions

Branch instructions test the state of one of the flags and either branch or not. Branches are relative jumps up to 128 bytes forward or back in the code determined by the offset argument. The branch instructions are:

```
BNE offset    Branch on result not equal    , result not zero: Z=0
BEQ offset    Branch on result equal        , result zero    : Z=1
BMI offset    Branch on result greater      , result <0      : N=1
BPL offset    Branch on result less or equal, result =>0     : N=0
BCC offset    Branch on carry clear                          : C=0
BCS offset    Branch on carry set                            : C=1
BVC offset    Branch on overflow clear                       : V=0
BVS offset    Branch on overflow set                         : V=1
BRA offset   *Branch always
```

*65C02 only

Arithmetic and Logical Instructions

We have examined data moving and comparing instructions, conditional branches and unconditional changes to the program flow. The last class of instructions we will examine are the arithmetic and logical instructions where actual computations occur. All of these instructions involve the accumulator as one of the arguments.

```
 ADC arg   Add with carry        .A + 'arg' + C  ==> .A  (C,V,N,Z)
 SBC arg   Subtract with borrow  .A - 'arg' - !C ==> .A  (C,V,N,Z)
 AND arg   Logical bitwise And   .A and 'arg'    ==> .A    (N,Z)
 ORA arg   Logical bitwise Or    .A  or 'arg'    ==> .A    (N,Z)
 EOR arg   Logical bitwise XOr   .A xor 'arg'    ==> .A    (N,Z)
```

The ADC and SBC use the carry to allow multibyte additions to be done using the C flag to carry to or borrow from the next higher

byte of the operation. The V flag is the overflow flag. It is set whenever a computation exceeds $FF or goes under $00. Note that the overflow flag can also be set using the SO set overflow pin on the 6502 allowing an additional input line to the processor.

Instruction Summary

There are other instructions for the 6502 but they are less commonly used then those introduced above. It would be most useful now to attempt to use the instructions learned in a few code fragments to see how instructions interrelate.