Assembly In One Step

RTK, last update: 23-Jul-97

A brief guide to programming the 6502 in assembly language. It will introduce the 6502 architecture, addressing modes, and instruction set. No prior assembly language programming is assumed, however it is assumed that you are somewhat familiar with hexadecimal numbers. Programming examples are given at the end. Much of this material comes from *6502 Software Design* by Leo Scanlon, Blacksburg, 1980.

The 6502 Architecture

The 6502 is an 8-bit microprocessor that follows the memory oriented design philosophy of the Motorola 6800. Several engineers left Motorola and formed MOS Technology which introduced the 6502 in 1975. The 6502 gained in popularity because of it's low price and became the heart of several early personal computers including the Apple II, Commodore 64, and Atari 400 and 800.

Simplicity is key

The 6502 handles data in its registers, each of which holds one byte (8-bits) of data. There are a total of three general use and two special purpose registers:

accumulator (A)	-	Handles all arithmetic and logic. The real heart of the system.
X and Y	-	General purpose registers with limited abilities.
S	-	Stack pointer.
P	-	Processor status. Holds the result of tests and flags.

Stack Pointer

When the microprocessor executes a JSR (Jump to SubRoutine) instruction it needs to know where to return when finished. The 6502 keeps this information in low memory from $0100 to $01FF and uses the stack pointer as an offset. The stack grows down from $01FF and makes it possible to nest subroutines up to 128 levels deep. Not a problem in most cases.

Processor Status

The processor status register is not directly accessible by any 6502 instruction. Instead, there exist numerous instructions that test the bits of the processor status register. The flags within the register are:

```
bit ->   7                               0
       +---+---+---+---+---+---+---+---+
       | N | V |   | B | D | I | Z | C |  <-- flag, 0/1 = reset/set
       +---+---+---+---+---+---+---+---+
```

N = NEGATIVE. Set if bit 7 of the accumulator is set.

V = OVERFLOW. Set if the addition of two like-signed numbers or the subtraction of two unlike-signed numbers produces a result greater than +127 or less than -128.

B = BRK COMMAND. Set if an interrupt caused by a BRK, reset if caused by an external interrupt.

D = DECIMAL MODE. Set if decimal mode active.

I = IRQ DISABLE. Set if maskable interrupts are disabled.

Z = ZERO. Set if the result of the last operation (load/inc/dec/add/sub) was zero.

C = CARRY. Set if the add produced a carry, or if the subtraction produced a borrow. Also holds bits after a logical shift.

Accumulator

The majority of the 6502's business makes use of the accumulator. All addition and subtraction is done in the accumulator. It also handles the majority of the logical comparisons (is A > B ?) and logical bit shifts.

X and Y

These are index registers often used to hold offsets to memory locations. They can also be used for holding needed values. Much of their use lies in supporting some of the addressing modes.

Addressing Modes

The 6502 has 13 addressing modes, or ways of accessing memory. The 65C02 introduces two additional modes.

They are:

mode	assembler format	
Immediate	#aa	
Absolute	aaaa	
Zero Page	aa	Note:

Implied		
Indirect Absolute	(aaaa)	aa = 2 hex digits
Absolute Indexed,X	aaaa,X	as $FF
Absolute Indexed,Y	aaaa,Y	
Zero Page Indexed,X	aa,X	aaaa = 4 hex
Zero Page Indexed,Y	aa,Y	digits as
Indexed Indirect	(aa,X)	$FFFF
Indirect Indexed	(aa),Y	
Relative	aaaa	Can also be
Accumulator	A	assembler labels

(Table 2-3. _6502 Software Design_, Scanlon, 1980)

Immediate Addressing

The value given is a number to be used immediately by the instruction. For example, LDA #$99 loads the value $99 into the accumulator.

Absolute Addressing

The value given is the address (16-bits) of a memory location that contains the 8-bit value to be used. For example, STA $3E32 stores the present value of the accumulator in memory location $3E32.

Zero Page Addressing

The first 256 memory locations ($0000-00FF) are called "zero page". The next 256 instructions ($0100-01FF) are page 1, etc. Instructions making use of the zero page save memory by not using an extra $00 to indicate the high part of the address. For example,

```
LDA $0023   -- works but uses an extra byte
LDA $23     -- the zero page address
```

Implied Addressing

Many instructions are only one byte in length and do not reference memory. These are said to be using implied addressing. For example,

```
CLC  -- Clear the carry flag
DEX  -- Decrement the X register by one
TYA  -- Transfer the Y register to the accumulator
```

Indirect Absolute Addressing

Only used by JMP (JuMP). It takes the given address and uses it as a pointer to the low part of a 16-bit address in memory, then jumps to that address. For example,

```
   JMP ($2345)   -- jump to the address in $2345 low and $2346 high

   So if $2345 contains $EA and $2346 contains $12 then the next
   instruction executed is the one stored at $12EA.  Remember, the
   6502 puts its addresses in low/high format.
```

Absolute Indexed Addressing

The final address is found by taking the given address as a base and adding the current value of the X or Y register to it as an offset. So,

```
   LDA $F453,X  where X contains 3
```

Load the accumulator with the contents of address $F453 + 3 = $F456.

Zero Page Indexed Addressing

Same as Absolute Indexed but the given address is in the zero page thereby saving a byte of memory.

Indexed Indirect Addressing

Find the 16-bit address starting at the given location plus the current X register. The value is the contents of that address. For example,

```
   LDA ($B4,X)  where X contains 6
```

gives an address of $B4 + 6 = $BA. If $BA and $BB contain $12 and $EE respectively, then the final address is $EE12. The value at location $EE12 is put in the accumulator.

Indirect Indexed Addressing

Find the 16-bit address contained in the given location (and the one following). Add to that address the contents of the Y register. Fetch the value stored at that address. For example,

```
   LDA ($B4),Y  where Y contains 6
```

If $B4 contains $EE and $B5 contains $12 then the value at memory location $12EE + Y (6) = $12F4 is fetched and put in the accumulator.

Relative Addressing

The 6502 branch instructions use relative addressing. The next byte is a signed offset from the current address, and the net sum is the address of the next instruction executed. For example,

```
   BNE $7F   (branch on zero flag reset)
```

will add 127 to the current program counter (address to execute) and start executing the instruction at that address. SImilarly,

```
BEQ $F9   (branch on zero flag set)
```

will add a -7 to the current program counter and start execution at the new program counter address.

Remember, if one treats the highest bit (bit 7) of a byte as a sign (0 = positive, 1 = negative) then it is possible to have numbers in the range -128 ($80) to +127 (7F). So, if the high bit is set, i.e. the number is > $7F, it is a negative branch. How far is the branch? If the value is < $80 (positive) it is simply that many bytes. If the value is > $7F (negative) then it is the 2's compliment of the given value in the negative direction.

2's compilment

The 2's compilment of a number is found by switching all the bits from 0 -> 1 and 1 -> 0, then adding 1. So,

```
$FF  =  1111 1111   <-- original
        0000 0000   <-- 1's compliment
     +          1
        ---------
        0000 0001   <-- 2's compliment, therefore $FF = -1
```

Note that QForth uses this for numbers greater than 32768 so that 65535 = -1 and 32768 = -32768.

In practice, the assembly language programmer uses a label and the assembler takes care of the actual computation. Note that branches can only be to addresses within -128 to +127 bytes from the present address. The 6502 does not allow branches to an absolute address.

Accumulator Addressing

Like implied addressing, the object of the instruction is the accumulator and need not be specified.

The 6502 Instruction Set

There are 56 instructions in the 6502, and more in the 65C02. Many instructions make use of more than one addressing mode and each instruction/addressing mode combination has a particular hexadecimal opcode that specifies it exactly. So,

```
A9  =  LDA #$aa   Immediate addressing mode load of accumulator
AD  =  LDA $aaaa  Absolute addressing mode load of accumulator
etc.
```

Some 6502 instructions make use of bitwise logic. This includes AND, OR, and EOR (Exclusive-OR). The tables below illustrate the effects of these operations:

```
   AND   1  1  ->  1    "both"
         1  0  ->  0
         0  1  ->  0
         0  0  ->  0

   OR    1  1  ->  1    "either one or both"
         1  0  ->  1
         0  1  ->  1
         0  0  ->  0

   EOR   1  1  ->  0    "one or the other but not both"
         1  0  ->  1
         0  1  ->  1
         0  0  ->  0
```

Therefore, $FF AND $0F = $0F since,

```
          1111 1111
     and  0000 1111
          ---------
          0000 1111  = $0F
```

AND is useful for masking bits. For example, to mask the high order bits of a value AND with $0F:

```
   $36 AND $0F  =  $06
```

OR is useful for setting a particular bit:

```
   $80 OR $08   =  $88

   since  1000 0000  ($80)
          0000 1000  ($08)
       or ---------
          1000 1000  ($88)
```

EOR is useful for flipping bits:

```
   $AA EOR $FF  =  $55

   since  1010 1010  ($AA)
          1111 1111  ($FF)
      eor ---------
          0101 0101  ($55)
```

Other 6502 instructions shift bits to the right or the left or rotate them right or left. Note that shifting to the left by one bit is the same as multipling by 2 and that shifting right by one bit is the same as dividing by 2.

The 6502 instructions fall naturally into 10 groups with two odd-ball instructions NOP and BRK:

- Load and Store Instructions
- Arithmetic Instructions
- Increment and Decrement Instructions
- Logical Instructions

Jump, Branch, Compare and Test Bits Instructions
Shift and Rotate Instructions
Transfer Instructions
Stack Instructions
Subroutine Instructions
Set/Reset Instructions
NOP/BRK Instructions

Load and Store Instructions

LDA - LoaD the Accumulator
LDX - LoaD the X register
LDY - LoaD the Y register

STA - STore the Accumulator
STX - STore the X register
STY - STore the Y register

Microprocessors spend much of their time moving stuff around in memory. Data from one location is loaded into a register and stored in another location, often with something added or subtracted in the process. Memory can be loaded directly into the A, X, and Y registers but as usual, the accumulator has more addressing modes available.

If the high bit (left most, bit 7) is set when loaded the N flag on the processor status register is set. If the loaded value is zero the Z flag is set.

Arithmetic Instructions

ADC - ADd to accumulator with Carry
SBC - SuBtract from accumulator with Carry

The 6502 has two arithmetic modes, binary and decimal. Both addition and subtraction implement the carry flag to track carries and borrows thereby making multibyte arithmetic simple. Note that in the case of subtraction it is necessary to SET the carry flag as it is the opposite of the carry that is subtracted.

Addition should follow this form:

```
CLC
ADC ...
.
.
ADC ...
.
.
.
```

Clear the carry flag, and perform all the additions. The carry between additions will be handled in the carry flag. Add from low byte to high byte. Symbolically, the net effect of an ADC instruction is:

A + M + C --> A

Subtraction follows the same format:

```
SEC
SBC ...
.
.
SBC ...
.
.
.
```

In this case set the carry flag first and then do the subtractions.
Symbolically,

A - M - ~C --> A , where ~C is the opposite of C

Ex.1

A 16-bit addition routine. $20,$21 + $22,$23 = $24,$25

```
   CLC         clear the carry
   LDA $20     get the low byte of the first number
   ADC $22     add to it the low byte of the second
   STA $24     store in the low byte of the result
   LDA $21     get the high byte of the first number
   ADC $23     add to it the high byte of the second, plus carry
   STA $25     store in high byte of the result

   ... on exit the carry will be set if the result could not be
       contained in 16-bit number.
```

Ex.2

A 16-bit subtraction routine. $20,$21 - $22,$23 = $24,$25

```
   SEC         clear the carry
   LDA $20     get the low byte of the first number
   SBC $22     add to it the low byte of the second
   STA $24     store in the low byte of the result
   LDA $21     get the high byte of the first number
   SBC $23     add to it the high byte of the second, plus carry
   STA $25     store in high byte of the result

   ... on exit the carry will be set if the result produced a
       borrow
```

Aside from the carry flag, arithmetic instructions also affect the N,
Z, and V flags as follows:

```
  Z = 1  if result was zero, 0 otherwise
  N = 1  if bit 7 of the result is 1, 0 otherwise
  V = 1  if bit 7 of the accumulator was changed, a sign change
```

Increment and Decrement Instructions

INC - INCrement memory by one

INX - INcrement X by one
INY - INcrement Y by one

DEC - DECrement memory by one
DEX - DEcrement X by one
DEY - DEcrement Y by one

The 6502 has instructions for incrementing/decrementing the index registers and memory. Note that it does not have instructions for incrementing/decrementing the accumulator. This oversight was rectified in the 65C02 which added INA and DEA instructions. The index register instructions are implied mode for obvious reasons while the INC and DEC instructions use a number of addressing modes.

All inc/dec instructions have alter the processor status flags in the following way:

  Z = 1 if the result is zero, 0 otherwise
  N = 1 if bit 7 is 1, 0 otherwise

Logical Instructions

AND - AND memory with accumulator
ORA - OR memory with Accumulator
EOR - Exclusive-OR memory with Accumulator

These instructions perform a bitwise binary operation according to the tables given above. They set the Z flag if the net result is zero and set the N flag if bit 7 of the result is set.

Jump, Branch, Compare, and Test Bits

JMP - JuMP to another location (GOTO)

BCC - Branch on Carry Clear,   C = 0
BCS - Branch on Carry Set,   C = 1
BEQ - Branch on EQual to zero,   Z = 1
BNE - Branch on Not Equal to zero, Z = 0
BMI - Branch on MInus,   N = 1
BPL - Branch on PLus,   N = 0
BVS - Branch on oVerflow Set,   V = 1
BVC - Branch on oVerflow Clear,   V = 0

CMP - CoMPare memory and accumulator
CPX - ComPare memory and X
CPY - ComPare memory and Y

BIT - test BITs

This large group includes all instructions that alter the flow of the program or perform a comparison of values or bits.

JMP simply sets the program counter (PC) to the address given. Execution proceeds from the new address. The branch instructions are relative jumps. They cause a branch to a new address that is either

127 bytes beyond the current PC or 128 bytes before the current PC. Code that only uses branch instructions is relocatable and can be run anywhere in memory.

The three compare instructions are used to set processor status bits. After the comparison one frequently branches to a new place in the program based on the settings of the status register. The relationship between the compared values and the status bits is,

	N	Z	C
A, X, or Y < Memory	1	0	0
A, X, or Y = Memory	0	1	1
A, X, or Y > Memory	0	0	1

The BIT instruction tests bits in memory with the accumulator but changes neither. Only processor status flags are set. The contents of the specified memory location are logically ANDed with the accumulator, then the status bits are set such that,

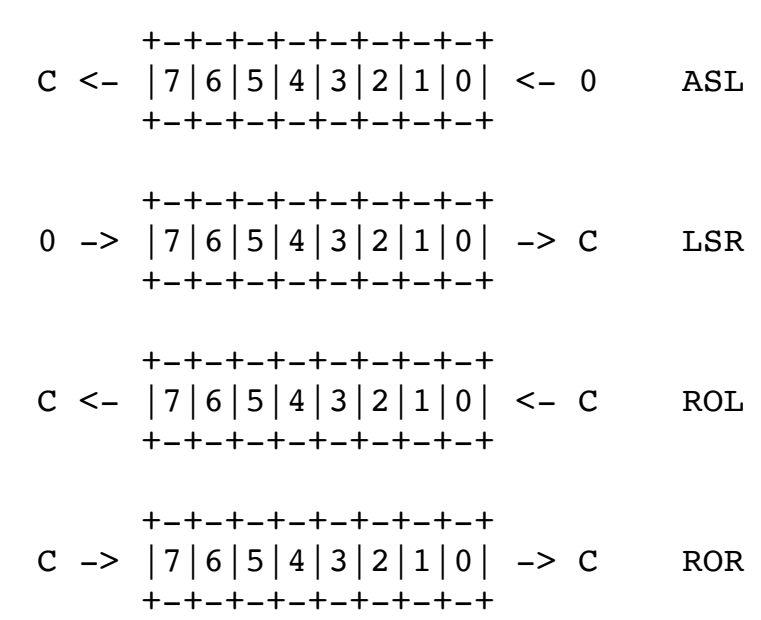
* N receives the initial, un-ANDed value of memory bit 7.
* V receives the initial, un-ANDed value of memory bit 6.
* Z is set if the result of the AND is zero, otherwise reset.

So, if $23 contained $7F and the accumulator contained $80 a BIT $23 instruction would result in the V and Z flags being set and N reset since bit 7 of $7F is 0, bit 6 of $7F is 1, and $7F AND $80 = 0.

Shift and Rotate Instructions

ASL - Accumulator Shift Left
LSR - Logical Shift Right
ROL - ROtate Left
ROR - ROtate Right

Use these instructions to move things around in the accumulator or memory. The net effects are (where C is the carry flag):



Z is set if the result it zero. N is set if bit 7 is 1. It is always reset on LSR. Remember that ASL A is equal to multiplying by two and that LSR is equal to dividing by two.

Transfer Instructions

```
TAX  - Transfer Accumulator to X
TAY  - Transfer Accumulator to Y
TXA  - Transfer X to accumulator
TYA  - Transfer Y to Accumulator
```

Transfer instructions move values between the 6502 registers. The N and Z flags are set if the value being moved warrants it, i.e.

```
LDA #$80
TAX
```

causes the N flag to be set since bit 7 of the value moved is 1, while

```
LDX #$00
TXA
```

causes the Z flag to be set since the value is zero.

Stack Instructions

```
TSX  - Transfer Stack pointer to X
TXS  - Transfer X to Stack pointer

PHA  - PusH Accumulator on stack
PHP  - PusH Processor status on stack
PLA  - PulL Accumulator from stack
PLP  - PulL Processor status from stack
```

TSX and TXS make manipulating the stack possible. The push and pull instructions are useful for saving register values and status flags. Their operation is straightforward.

Subroutine Instructions

```
JSR  - Jump to SubRoutine
RTS  - ReTurn from Subroutine
RTI  - ReTurn from Interrupt
```

Like JMP, JSR causes the program to start execution of the next instruction at the given address. Unlike JMP, JSR pushes the address of the next instruction after itself on the stack. When an RTS instruction is executed the address pushed on the stack is pulled off the stack and the program resumes at that address. For example,

```
   LDA #$C1   ; load the character 'A'
   JSR print  ; print the character and it's hex code
   LDA #$C2   ; load 'B'
   JSR print  ; and print it
   .
   .
   .
print JSR $FDED  ; print the letter
      JSR $FDDA  ; and its ASCII code
      RTS        ; return to the caller
```

RTI is analagous to RTS and should be used to end an interrupt routine.

Set and Reset (Clear) Instructions

CLC - CLear Carry flag
CLD - CLear Decimal mode
CLI - CLear Interrupt disable
CLV - CLear oVerflow flag

SEC - SEt Carry
SED - SEt Decimal mode
SEI - SEt Interrupt disable

These are one byte instructions to specify processor status flag settings.

CLC and SEC are of particular use in addition and subtraction respectively. Before any addition (ADC) use CLC to clear the carry or the result may be one greater than you expect. For subtraction (SBC) use SEC to ensure that the carry is set as its compliment is subtracted from the answer. In multi-byte additions or subtractions only clear or set the carry flag before the initial operation. For example, to add one to a 16-bit number in $23 and $24 you would write:

```
LDA $23     ; get the low byte
CLC         ; clear the carry
ADC #$02    ; add a constant 2, carry will be set if result > 255
STA $23     ; save the low byte
LDA $24     ; get the high byte
ADC #$00    ; add zero to add any carry that might have been set above
STA $24     ; save the high byte
RTS         ; if carry set now the result was > 65535
```

Similarly for subtraction,

```
LDA $23     ; get the low byte
SEC         ; set the carry
SBC #$02    ; subtract 2
STA $23     ; save the low byte
LDA $24     ; get the high byte
SBC #$00    ; subtract 0 and any borrow generated above
STA $24     ; save the high byte
RTS         ; if the carry is not set the result was < 0
```

Other Instructions

==========

NOP - No OPeration (or is it NO oPeration ? :)
BRK - BReaK

NOP is just that, no operation. Useful for deleting old instructions, reserving room for future instructions or for use in careful timing loops as it uses 2 microprocessor cycles.

BRK causes a forced break to occur and the processor will immediately start execution of the routine whose address is in $FFFE and $FFFF. This address is often the start of a system monitor program.

Some simple programming examples

A few simple programming examples are given here. They serve to illustrate some techniques commonly used in assembly programming. There are doubtless dozens more and I make no claim at being a proficient assembly language programmer. For examples of addition and subtraction see above on CLC and SEC.

A count down loop

```
        ;
        ; An 8-bit count down loop
        ;
start LDX #$FF    ; load X with $FF = 255
loop  DEX         ; X = X - 1
      BNE loop    ; if X not zero then goto loop
      RTS         ; return
```

How does the BNE instruction know that X is zero? It doesn't, all it knows is that the Z flag is set or reset. The DEX instruction will set the Z flag when X is zero.

```
        ;
        ; A 16-bit count down loop
        ;
start LDY #$FF    ; load Y with $FF
loop1 LDX #$FF    ; load X with $FF
loop2 DEX         ; X = X - 1
      BNE loop2   ; if X not zero goto loop2
      DEY         ; Y = Y - 1
      BNE loop1   ; if Y not zero goto loop1
      RTS         ; return
```

There are two loops here, X will be set to 255 and count to zero for each time Y is decremented. The net result is to count the 16-bit number Y (high) and X (low) down from $FFFF = 65535 to zero.

Other examples

** Note: All of the following examples are lifted nearly verbatim from
the book "6502 Software Design", whose reference is above.

```
; Example 4-2.  Deleting an entry from an unordered list
;
; Delete the contents of $2F from a list whose starting
; address is in $30 and $31.  The first byte of the list
; is its length.
;

deluel  LDY #$00     ; fetch element count
        LDA ($30),Y
        TAX          ; transfer length to X
        LDA $2F      ; item to delete
nextel  INY          ; index to next element
        CMP ($30),Y  ; do entry and element match?
        BEQ delete   ; yes. delete element
        DEX          ; no. decrement element count
        BNE nextel   ; any more elements to compare?
        RTS          ; no. element not in list. done

; delete an element by moving the ones below it up one location

delete  DEX          ; decrement element count
        BEQ deccnt   ; end of list?
        INY          ; no. move next element up
        LDA ($30),Y
        DEY
        STA ($30),Y
        INY
        JMP delete
deccnt  LDA ($30,X)  ; update element count of list
        SBC #$01
        STA ($30,X)
        RTS



; Example 5-6.  16-bit by 16-bit unsigned multiply
;
; Multiply $22 (low) and $23 (high) by $20 (low) and
; $21 (high) producing a 32-bit result in $24 (low) to $27 (high)
;

mlt16   LDA #$00     ; clear p2 and p3 of product
        STA $26
        STA $27
        LDX #$16     ; multiplier bit count = 16
nxtbt   LSR $21      ; shift two-byte multiplier right
        ROR $20
        BCC align    ; multiplier = 1?
        LDA $26      ; yes. fetch p2
        CLC
        ADC $22      ; and add m0 to it
        STA $26      ; store new p2
        LDA $27      ; fetch p3
        ADC $23      ; and add m1 to it
```

```
align   ROR A         ; rotate four-byte product right
        STA $27       ; store new p3
        ROR $26
        ROR $25
        ROR $24
        DEX           ; decrement bit count
        BNE nxtbt     ; loop until 16 bits are done
        RTS



; Example 5-14.  Simple 16-bit square root.
;
; Returns the 8-bit square root in $20 of the
; 16-bit number in $20 (low) and $21 (high). The
; remainder is in location $21.

sqrt16  LDY #$01      ; lsby of first odd number = 1
        STY $22
        DEY
        STY $23       ; msby of first odd number (sqrt = 0)
again   SEC
        LDA $20       ; save remainder in X register
        TAX           ; subtract odd lo from integer lo
        SBC $22
        STA $20
        LDA $21       ; subtract odd hi from integer hi
        SBC $23
        STA $21       ; is subtract result negative?
        BCC nomore    ; no. increment square root
        INY
        LDA $22       ; calculate next odd number
        ADC #$01
        STA $22
        BCC again
        INC $23
        JMP again
 nomore STY $20       ; all done, store square root
        STX $21       ; and remainder
        RTS
```

This is based on the observation that the square root of an integer is equal to the number of times an increasing odd number can be subtracted from the original number and remain positive. For example,

```
        25
      -  1          1
        --
        24
      -  3          2
        --
        21
      -  5          3
        --
        16
      -  7          4
        --
         9
      -  9          5 = square root of 25
```

```
                    --
                     0
```

If you are truly interested in learning more, go to your public library and seek out an Apple machine language programming book. If your public library is like mine, there will still be plenty of early 80s computer books on the shelves. :)

Last update: 30-Jan-00

Back