SECTION 6
R6500 PROGRAMMING CONCEPTS

This section provides an introduction to the programming of the R6502 microprocessor at the machine or assembly level. The R6500 Programming Manual and the references listed at the end of the section contain more detailed information. Our intention here is to show you how to perform common tasks with the R6502. You should use this section as a first step in R6502 programming or as a brief overview. The last part of the section discusses the overall problems of software development and lists some useful references.

6.1 PROGRAMMING TASKS

Let us first describe some of the tasks we would like the R6502 to perform. These include:

- Simple calculations or functions, such as addition, subtraction, logical AND, etc.

- Decision making, i.e., determining whether inputs or the results of calculations have certain values or are above or below certain levels. The R6502 should be able to choose a course of action depending on these decisions--after all, that is what intelligence is all about.

• Looping, i.e., repeating tasks a specified number of times or until some condition is satisfied.

• Array handling, i.e., processing groups of data items such as sensor readings, test inputs, control outputs, or strings of characters.

• Code conversion and manipulation, i.e., handling data that is or must be coded in some form such as BCD, ASCII, seven-segment, or Gray. Clearly this task is an essential part of obtaining data from input peripherals and sending data to output peripherals.

• Arithmetic, i.e., performing such functions as multiple precision addition and subtraction, multiplication, division, and square root.

Finally, we need some way of connecting the short programs that perform these tasks to our main program. Of course, this discussion will only touch on how to write and connect short programs--we will not touch here on how to define your problem, design the program, and debug, test, and document it. We will briefly discuss those topics later. Note also that Section 8 describes I/O programming with the R6522 Versatile Interface Adapter.

6.2 R6502 REGISTERS AND FLAGS

We have previously reviewed the R6502 registers and flags in Section 2.6. The registers are: (See Figure 6-1.)

Program counter (PC)

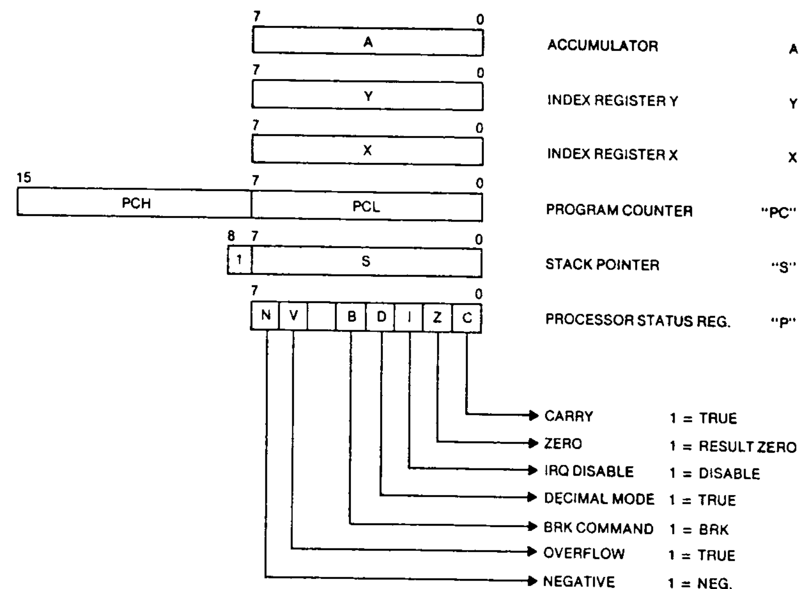

Figure 6-1. R6502 Registers and Flags

Accumulator (A)
Index Registers X and Y
Processor status (P)
Stack pointer (S)

e flags are:

Negative (N) or sign
Overflow (V)
Break command (B)
Decimal mode (D)
Interrupt disable (I)
Zero (Z)
Carry (C)

We may describe the uses of the various registers as follows:

PC contains the address of the next instruction to be executed. PC is incremented each time it is used; jump and branch instructions place a new value in this register.

A holds one operand (and the result) in arithmetic and logical operations. It is the center of processing activities.

X and Y generally hold counters for looping or indexes for handling tables or arrays.

S holds the address (on page 1) of the top of the RAM stack used for saving subroutine return and addresses and the previous contents of registers and flags.

The uses of the common flags are described as follows. Note that we generally use these flags by testing their values with conditional jump instructions.

N is used to check status bits and sign values.

Z is used to check for equality (after a subtraction), for a zero value in a counter, and for a zero value in a bit (after a logical AND).

C is used to save carries or borrows in arithmetic operations, to determine the result of numerical comparisons, and to determine the value of a bit (after a shift operation).

6.3 SIMPLE OPERATIONS

Most simple operations take place in the R6502 accumulator. The instructions available include:.

- addition (ADC)
- subtraction (SBC)
- logical AND (AND)
- logical OR (ORA)
- logical EXCLUSIVE OR (EOR)

All of these instructions assume that one operand is in the accumulator and the other is in memory. The result is saved in the accumulator.

Three steps are necessary for the CPU to perform an arithmetic or logical operation on two numbers. Each of these steps corresponds to a separate R6502 instruction:

1. Load the accumulator with one operand.

2. Perform the operation on the accumulator and the other operand.

3. Save the result (from the accumulator) in memory.

Thus, the R6502 works just like a simple hand calculator in which you enter one operand, perform the operation with the other operand, and save the result for later use.

NOTE

All arithmetic and logical operations are performed on two 8-bit numbers at a time. Operations involving more operands or longer operands require a series of instructions. The CARRY bit must be used to transfer carries or borrows between successive additions and subtractions.

Examples:

1. Logically AND the contents of memory locations 40 and 41 place the result in 42.

(R6500 assembly format)

```
LDA     $40   GET FIRST OPERAND
AND     $41   LOGICAL AND
STA     $42   STORE RESULT
BRK
```


Examples (cont.):

(AIM 65 mnemonic format)

```
LDA     40
AND     41
STA     42
BRK
```

Here we use the R6500 Assembler notation in which $ means "hexadecimal". (The $ is not used when you enter instructions with the AIM 65 I command.)

2. Clear the four most significant bits of memory location 40 and place the result in 41.

(R6500 assembly format)

```
LDA     $40    GET OPERAND
AND     #$0F   CLEAR 4 MSB'S
STA     $41    STORE RESULT
BRK
```

Here we use the R6500 Assembler notation in which # means "immediate", i.e., the number that follows is data rather than a memory address.

(AIM 65 mnemonic format)

```
LDA     40
AND     #0F
STA     41
BRK
```

Examples (Cont.)

3. Set the most significant bit of memory location 40 and place the result in 41.

(R6500 assembly format)

```
LDA     $40     GET OPERAND
ORA     #$80    SET MSB
STA     $41
BRK
```

Remember that anything logically ORed with a '1' produces a result of 1.

(AIM 65 mnemonic format)

```
LDA     40
ORA     #80
STA     41
BRK
```

Note that a few simple operations can be performed directly on memory. These operatons are:

Increment (add 1)
Decrement (subtract 1)
Arithmetic shift left (move all bits left by one position and clear LSB)
Logical shift right (move all bits right by one position and clear MSB)

Rotate left or right (move all bits including the CARRY by one position as if the ends of the number were connected through the CARRY).

There are also shift and rotate instructions for the accumulator.

6.4 MAKING DECISIONS

In the simple programs of the previous discussion, the R6502 acts like an inexpensive hand calculator. The conditional branch instructions make the R6502 into an intelligent controller, capable of performing different actions based on input data or the results of calculations. Table 6-1 summarizes these instructions.

The R6502 executes conditional branch instruction as follows:

- If the condition is met, a new value is placed in the program counter and the R6502 executes the instruction at that address next.
- If the condition is not met, the program counter is not changed and the R6502 continues executing instructions sequentially.

Figure 6-2 is a flowchart of the conditional branch.

The common uses of the conditional branch instructions are as follows:

```
BCC     (BRANCH IF CARRY CLEAR)
BCS     (BRANCH IF CARRY SET)
```

Table 6-1. R6502 Conditional Branch Instructions

INSTRUCTION	FLAG USED	VALUE ON WHICH BRANCH OCCURS
BCC	CARRY	0
BCS	CARRY	1
BNE	ZERO	0
BEQ	ZERO	1
BPL	NEGATIVE	1
BMI	NEGATIVE	1
BVC	OVERFLOW	0
BVS	OVERFLOW	1

If the specified flag does not have the indicated value, no branch occurs.

1. After a comparison to determine if one value is greater than or less than another. Typical sequences and their effects are as follows:

```
CMP      OTHER
BCC      LARGE
```

causes a branch to address LARGE if (A) < (OTHER).

```
CMP      #THRSH
BCS      ABOVE
```


causes a branch to address ABOVE if (A) ≥ THRSH.

Note that after a CMP instruction, C = 0 if the number in A is less than the other number (unsigned). Table 6-2 shows the states of the major flags (N, Z, C) after a CMP, CPX, or CPY instruction.

Table 6-2. Compare Instruction Results

CONDITION	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

*N is valid only for two's complement

2. After an addition or subtraction to determine if a carry or borrow was produced. Typical sequences are:

```
ADC      NEXT
BCS      RIPPLE
```

causes a branch to address RIPPLE if the addition results in a carry.

```
SBC      FACTOR
BCS      DONE
```

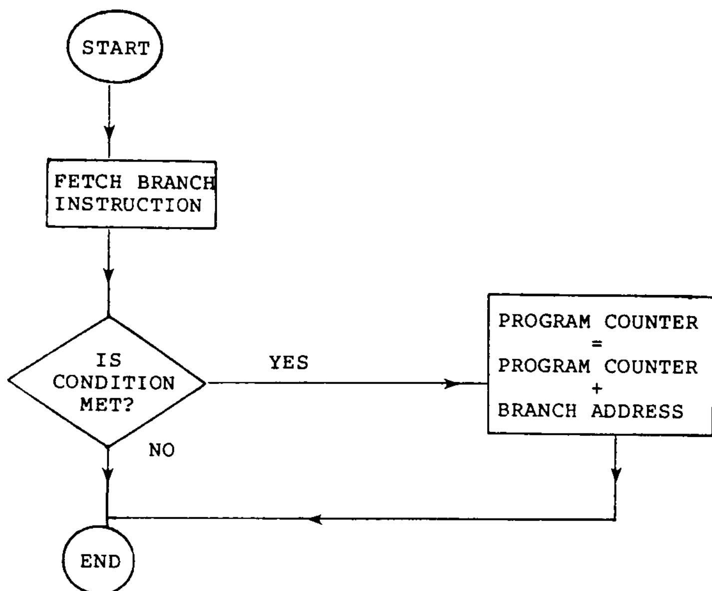

Figure 6-2. Flowchart of a Conditional Branch

causes a branch to address DONE if the subtraction does not require a borrow. Note that a borrow is needed if no CARRY is generated. (The logic for borrows is the opposite of the logic for carries.)

3. After a shift to determine the value of a particular bit. Typical sequences and their effects are:

```
ASL     MULT
BCC     NOADD
```

causes a branch to address NOADD if the most significant bit of the contents of MULT is zero.

```
ROR     INPUT
BCS     LAST
```

causes a branch to address LAST if the least significant bit of the contents of INPUT is one.

```
BEQ     (BRANCH IF Z = 1)
BNE     (BRANCH IF Z = 0)
```

NOTE

Remember, Z = 1 if the previous result was zero and Z = 0 if the previous result was not zero.

1. After a comparison, to determine if two values are equal. Typical sequences are:

```
CMP     OTHER
BNE     DIFF
```

causes a branch to address DIFF if $(A) \neq$ (OTHER).

```
CMP    #KEY
BEQ    FOUND
```

causes a branch to address FOUND if (A) = KEY. Note again the difference between ordinary (or direct) addressing and immediate addressing.

2. After a decrement or increment to determine if a counter has been decremented or incremented to zero. Typical sequences are:

```
DEX
BNE    LOOP
```

causes a branch to address LOOP if register X has not been decremented to zero.

```
INC    LSIG
BEQ    RIPPL
```

causes a branch to address RIPPL if the result of incrementing LSIG was zero. Note that INC does not affect the C (or CARRY) bit.

3. After a logical AND to determine the value of a particular bit. Typical sequences are as follows:

```
AND    #$40
BEQ    NOT1
```

causes a branch to address NOT1 if bit 6 of the accumulator is zero.

```
AND    #$08
BNE    BITON
```

causes a branch to address BITON if bit 3 of the accumulator is one.

```
BMI    (BRANCH ON N = 1)
BPL    (BRANCH ON N = 0)
```

4. To check the value of the most significant bit. Typical sequences are as follows:

```
LDA    FLAGS
BMI    FOUND
```

causes a branch to address FOUND if the most significant bit of FLAGS is 1.

```
LDA    SWCHS
BPL    CLSD
```

causes a branch to address CLSD if the most significant bit of SWCHS is 0.

The availability of the N bit makes bit position 7 special even if it is not being used for a sign. Since its value can be determined directly (without a shift or logical instruction), bit position 7 is often used for serial I/O and for the most frequently interrogated switches and user flags. Bit positions 0 and 6 are also somewhat special because of the shift instructions.

Let us now look at a few simple examples of decisions:

1. Find the larger value of memory locations 40 and 41, and store the value in 42.

(R6500 assembly format)

```
        LDA     $40   GET FIRST VALUE
        CMP     $41   IS SECOND VALUE LARGER?
        BCS     DONE
        LDA     $41   YES, GET SECOND VALUE
DONE    STA     $42   STORE LARGER VALUE
        BRK
```

(AIM 65 mnemonic format)

```
LDA     40
CMP     41
BCS     02
LDA     41
STA     42
BRK
```


The branch is taken if (40) $\geq$ (41). Note that in comparisons and subtractions the R6502 sets C if no borrow is necessary and clears C if a borrow is required.

2. Set memory location 42 to 1 if the contents of 40 and 41 are equal, and to 0 otherwise.

(R6500 assembly format)

```
        LDX     #0    MARK = ZERO
        LDA     $40   GET FIRST VALUE
        CMP     $41   IS SECOND VALUE THE SAME?
        BNE     DONE
        INX           YES, MARK = ONE
DONE    STX     $42   STORE MARK
        BRK
```

The branch is taken if (40) $\neq$ (41).

(AIM 65 mnemonic format)

```
LDX     #00
LDA     40
CMP     41
BNE     01
INX
STX     42
BRK
```

3. Set memory location 41 to 1 if bit 5 of memory location 40 is 1 and to 0 otherwise.

(R6500 assembly format)

```
        LDX     #0      MARK = ZERO
        LDA     $40     IS BIT 5 OF DATA 1?
        AND     #$20
        BEQ     DONE
        INX             YES, MARK = ONE
DONE    STX     $41     STORE MARK
        BRK
```

(AIM 65 mnemonic format)

```
        LDX     #00
        LDA     40
        AND     #20
        BEQ     01
        INX
        STX     41
        BRK
```

The branch is taken if bit 5 of memory location 40 is zero.

4. Set memory location 41 to 1 if bit 7 of memory location 40 is 1 and to 0 otherwise.

(R6500 Assembler format)

```
        LDX     #0
        LDA     $40
        BPL     DONE
        INX
```



```
DONE    STX     $41
        BRK
```

The branch is taken if bit 7 of memory location 40 is zero.

AIM 65 mnemonic format)

```
        LDX     #00
        LDA     40
        BPL     01
        INX
        STX     41
        BRK
```

NOTE

In actual machine language programs, all conditional branches use relative addresses that specify how much to add to the program counter, if the branch is taken. The relative address is an 8-bit two's complement number that allows branches of -128 to +127 locations from the end of the branch instruction. You can get longer branches by using the unconditional JMP instruction and a little inverted logic.

```
        BEQ     NEXT
        JMP     FAR
NEXT
```

causes a branch to memory address FAR if $Z \neq 0$.

6.5 LOOPS

The simplest way to have the R6502 microprocessor repeat a section of a program a specified number of times is:

1. Before entering the section to be repeated, initialize a counter to a specified value.

2. After completing the section, decrement the counter.

3. If the result of Step 2 is not zero, return to the start of the section.

Clearly, the branch instruction will be BNE. You can place the counter in register X, in register Y, or in a memory location. Figure 6-3 is a flowchart of the logic for a program loop.

The basic loop structure then is (using register X):

```
        LDX     #COUNT
LOOP    .
        .
        .
        .
        .
        DEX
        BNE     LOOP
```


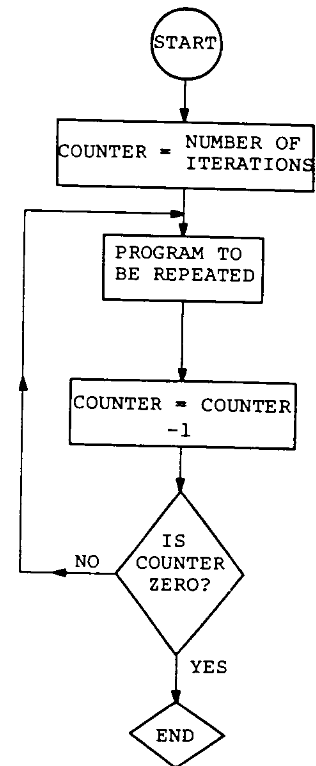

Figure 6-3. Flowchart of a Program Loop

An alternative approach is to count up instead of down, i.e.,

```
        LDX     #0
LOOP    .
        .
        .
        .
        INX
        CPX     #COUNT
        BNE     LOOP
```

One extra instruction (CPX) is necessary, but this method may be more convenient within the program itself.

Note that the looping method is essentially the same as marking off iterations by hand. You must be careful that you only initialize the counter once but that you increment it or decrement it each time through the loop. Of course, you can place loops within loops (called nesting) as long as you keep the counters separate and handle the branches properly.

6.6 DATA ARRAYS

The key to processing data arrays with the R6502 microprocessor is to use an index register to hold the element number. Remember that we need two things to describe an element in any array.

1. The starting or base address of the entire array.

2. The element number or index.

Remember that we can characterize an element in an array as A_i where A is the name of the array and i is the element number.

We can use the R6502's indexed addressing to process arrays. In this addressing mode, the R6502 adds the contents of an index register (X or Y) to the address in the instruction to get the actual data address.

Example:

If (X) = 06, the instruction AND $2000, X results in

(A) = (A) · (2000 +(X))
    = (A) · (2006)

Note that X and Y are 8-bit registers so that the offset can never be larger than 256.

An important point is that we can change X or Y (with INX, INY, DEX, or DEY) and then repeat the instruction. Each iteration gets the data from a different address in memory. The actual address of the data is called the effective address. Note that we can change the effective address even if the instruction is in a read-only memory.

Let us now look at some simple examples:

1. Form a checksum in memory location 40 by EXCLUSIVE ORing the contents of memory locations 41 through 48.

(R6500 assembly format)

```
        LDA     #0      CHECKSUM = 0
        LDX     #0      COUNT = 0
CHSUM   EOR     $41,X   EXCLUSIVE OR AN ELEMENT
        INX
        CPX     #8      OUT OF ELEMENTS?
        BNE     CHSUM   NO, KEEP FORMING CHECKSUM
        STA     $40     YES, SAVE CHECKSUM
        BRK
```

(AIM 65 mnemonic format)

```
        LDA     #00
        LDX     #00
        EOR     41,X
        INX
        CPX     #08
        BNE     F9
        STA     40
        BRK
```

Of course, we could count down just as easily as up. Which do you think is better?

2. Count all the zeroes occurring in memory locations 41 through 48. Place the result in memory location 40.

(R6500 Assembler format)

```
        LDX     #0      ELEMENT COUNT = 0
        LDY     #0      ZERO COUNT = 0
```



```
CHZRO   LDA     $41,X   IS NEXT ELEMENT ZERO?
        BNE     CTELM
        INY             YES, INCREMENT ZERO COUNT
CTELM   INX
        CPX     #8
        BNE     CHZRO
        STY     $40
        BRK
```

(AIM 65 mnemonic format)

```
        LDX     #00
        LDY     #00
        LDA     41,X
        BNE     01
        INY
        INX
        CPX     #08
        BNE     F6
        STY     40
        BRK
```

3. Clear all the memory locations starting with 41 and ending with 48.

(R6500 assembly format)

```
        LDA     #0      GET ZERO
        LDX     #0      COUNT = ZERO
CLR1    STA     $41,X   CLEAR A LOCATION
        INX
        CPX     #8
```

```
        BNE         CLR1
        BRK
```

(AIM 65 mnemonic format)

```
        LDA         #00
        LDX         #00
        STA          41,X
        INX
        CPX         #08
        BNE          F9
        BRK
```

Note the use of INX (or DEX) and CPX. Remember that INX and DEX can only increment or decrement X by 1. For larger steps you must either repeat the INX or DEX instruction or perform the required arithmetic in the accumulator. There are no instructions for performing arithmetic in X or Y.

6.7 CODE CONVERSION AND MANIPULATION

You can use the R6502 microprocessor to handle data in any code. The processor itself only handles numerical values. It does not care what the meaning of the data is to peripheral devices or to an operator. So the processor handles control characters, letters, digits, etc. all in the same way. The programmer must provide the intelligence that interprets specific values properly.

Examples:

1. Set memory location 41 to 1 if memory location 40 contains a valid ASCII digit and to 0 otherwise. The ASCII code represents the decimal digits as 30 through 39 hex.

(R6500 assembly format)

```
        LDX     #0
        LDA     $40     GET DIGIT
        CMP     #$30    IS IT BELOW ASCII ZERO?
        BCC     DONE    YES, NOT VALID
        CMP     #$3A    IS IT ABOVE ASCII NINE?
        BCS     DONE    YES, NOT VALID
        INX             VALID IF IN-BETWEEN
DONE    STX     $41
        BRK
```

(AIM 65 mnemonic format)

```
        LDX     #00
        LDA     40
        CMP     #30
        BCC     05
        CMP     #3A
        BCS     01
        INX
        STX     41
        BRK
```

Remember that the logic for borrows is the inverse of the logic for carries.

2. Count the number of ASCII E's (45 hex) in memory locations 41 through 48. Place the count in memory location 40.

(R6500 assembly format)

```
        LDX     #0       ELEMENT COUNT = 0
        LDY     #0       E COUNT = 0
        LDA     #$45     GET ASCII E
CHKE    CMP     $41,X    IS NEXT ELEMENT AN E?
        BNE     CTELM
        INY              YES, INCREMENT E COUNT
CTELM   INX
        CPX     $8
        BNE     CHKE
        STY     $40
        BRK
```

(AIM 65 mnemonic format)

```
        LDX     #00
        LDY     #00
        LDA     #45
        CMP     41, X
        BNE     01
        INY
        INX
        CPX     #08
        BNE     F6
        STY     40
        BRK
```


As for code conversions, some (like ASCII to decimal or decimal to ASCII) can be performed with simple arithmetic instructions. More complex conversions involving seven-segment code, Gray codes, etc. can be handled with look-up tables. These tables are just data arrays that are stored in memory in a convenient order. For example, the ith element in the array could simply contain the code corresponding to element i. If the table starts at address CDTAB, the program to convert an element in A into its coded equivalent is:

```
    TAX
    LDA         CDTAB,X
```

Note that the lookup table procedure does not depend at all on what is in the table. In fact, many of the entries could be the same if you are willing to waste some memory. If the table does not change, you could even make it part of the program or store it in a read-only memory. Lookup tables are discussed more completely in J.B. Peatman, Microcomputer-based Design, McGraw-Hill, 1977, Chapter 7 and in L.A. Leventhal, "Cut Your Processor's Computation Time", Electronic Design, August 16, 1977, pp. 82-88.

6.8 ARITHMETIC

The only explicit arithmetic instructions available on the R6502 are:

ADC - Add with Carry, i.e., (A) = (A) + (M) + C

SBC - Subtract with Borrow, i.e., (A) = (A) - M - 1+C

More complex arithmetic operations, such as multiplication, division, or square root, must be implemented either:

1. As a series of additions, subtractions, and shifts, some algorithms are in

   T.C. Chen, "Automatic Computation of Exponentials, Logarithms, Ratios, and Square Roots", IBM Journal of Research and Development, July 1972, pp. 380-388.

   A.M. Despain, "Fourier Transform Computers Using CORDIC Iterations", IEEE Transactions on Computers, October 1974, pp. 993-1001.

   H.C. Schmid, Decimal Computation, Wiley, New York, 1974.1.

2. With lookup tables. For example, see:

   T.A. Seim, "Numerical Interpolation for Microprocessor-based Systems", Computer Design, February 1978, pp. 111-116.

3. In special hardware as discussed in S. Waser, "State-of-the-art in High-Speed Arithmetic Integrated Circuits", Computer Design, July 1978, pp 67-75.

Note the following features of R6500 arithmetic instructions:

1. The CARRY is always included. If you do not want its value to affect the operation, you must explicitly clear it with CLC before an addition or set it with SEC before a subtraction.

2. All additions and subtractions are decimal (BCD) if D = 1. But remember to clear D (CLD) when you are through with the decimal arithmetic.

3. You can use ASL A to add the accumulator to itself.

4. You can use ADC #0 or SBC #0 to add C to the accumulator or subtract 1 minus C from the accumulator.

5. Both ADC and SBC are 8-bit operations in which the result ends up in A.

6. The V bit tells if two's complement overflow has occurred, i.e., if the magnitude of the result has affected its sign bit.

Some simple examples are:

1. Add the contents of memory locations 40 and 41, and place the result in 42.

(R6500 assembly format)

```
CLC
LDA     $40     GET FIRST OPERAND
ADC     $41     ADD SECOND OPERAND
STA     $42     STORE SUM
BRK
```

(AIM 65 mnemonic format)

```
CLC
LDA        40
ADC        41
STA        42
BRK
```

2. Add the 16-bit number in memory locations 40 and 41 (MSB's in 41) to the 16-bit number in memory locations 42 and 43 (MSB's in 43), and place the result in 44 and 45 (MSB's in 45).

(R6500 assembly format)

```
CLC
LDA        $40     ADD LSB'S
ADC        $42
STA        $44
LDA        $41     ADD MSB'S
ADC        $43
STA        $45
BRK
```

(AIM 65 mnemonic format)

```
CLC
LDA        40
ADC        42
STA        44
LDA        41
ADC        43
STA        45
BRK
```


Note that we need the carry from the least significant bits to include in the addition of the most significant bits.

3. Add the 64-bit number starting in memory location 40 (LSB's first) to the 64-bit number starting in memory location 50 (LSB's first). Save the sum in memory locations 60 on, starting with the least significant bits.

(R6500 assembly format)

```
      CLC                   CLEAR CARRY TO START
      LDY        #08        BYTE COUNTER = 8
      LDX        #00        INDEX = 0
ADDW  LDA        $40,X      ADD 8 BITS
      ADC        $50,X
      STA        $60,X      STORE 8 BITS
      INX
      DEY
      BNE        ADDW
      BRK
```

(AIM 65 mnemonic format)

```
CLC
LDY        #08
LDX        #00
LDA        40,X
ADC        50,X
STA        60,X
INX
DEY
```

BNE F8

BRK

Note that INX and DEY do not affect the CARRY but CPX or CPY would.

6.9 SUBROUTINES

The simplest way to make a routine commonly available from any point in other programs is to make it into a subroutine. This requires:

1. A JSR (Jump To Subroutine) instruction in the main program to transfer control to the subroutine.
2. An RTS (Return From Subroutine) instruction at the end of the subroutine to transfer control back to the main program.

NOTE

Only one copy of the subroutine has to be in memory. Note also that control is automatically transferred back to the instruction immediately following the JSR which sent the processor to the subroutine.

How does the processor know where to return? The answer is that it uses the RAM stack and the stack pointer. JSR and RTS work as follows:

JSR stores the 8 most significant bits of the program counter at the address given by the stack pointer, decrements the stack pointer, similarly stores the 8 least significant bits of the program counter, and decrements the stack pointer again.

Symbolically, what happens is described by:

```
(0100 + (S))  = (PCH)
        (S)   = (S) - 1
(0100 + (S))  = (PCL)
        (S)   = (S) - 1
```

RTS performs the opposite function. It increments the stack pointer and fetches the program counter from those locations, i.e.:

```
  (S)  =  (S) + 1
(PCL)  =  (0100 + (S))
  (S)  =  (S) + 1
(PCH)  =  (0100 + (S))
```

Note the following features of JSR and RTS:

1. The stack is always on page 1. That is, the 8 most significant bits of the address used are always 01 hex, while the 8 least significant bits are given by the contents of register S.
2. The stack grows down in memory, i.e., from higher addresses to lower addresses. Note that the stack area is just ordinary read/write memory; the only thing that moves is the value in the stack pointer.

3. The overall program must initialize the stack pointer and manage the stack.

4. Subroutines can use other subroutines, since the old return addresses are saved in the stack. Note that the first address entered is the last one retrieved.

5. Register contents can also be saved in the stack using PHA and PHP (plus TXA and TYA for X and Y) and restored using PLA and PLP.

6.10 THE TASKS OF PROGRAM WRITING

Of course, developing programs involves far more than just writing short sequences of instructions. The software designer also must:

- Define the problem.
- Design the program.
- Debug, test, and document the program.

These activities typically take far longer than the writing of instructions. A common rule of thumb is that project time is spent as follows:

- 40% definition and design
- 20% writing instruction (or coding)
- 40% debugging testing, and documentation

We do not have the space here to do justice to any of these stages, let alone all of them. If you are serious about programming, we suggest that you consult the following sources:

Chapin, N., Flowcharts, Auerbach, Princeton, N.J., 1971

Hughes, J. K. and J. I. Michtom, A Structured Approach to Programming, Prentice-Hall, Englewood Cliffs, N.J., 1977

Kerninghan, B. W. and P. J. Plauger, The Elements of Programming Style, McGraw-Hill, New York, 1974.

Leventhal, L. A., Introduction to Microprocessors: Software, Hardware, Programming, Prentice-Hall, Englewood Cliffs, N.J., 1978, Chapter 6

McGowan, C. L. and J. R. Kelly, Top-Down Structured Programming Techniques, Petrocelli/Charter, New York, 1975

Schneider, J. M. et. al., Introduction to Programming and Problem-Solving with Pascal, Wiley, New York, 1978

Yourdon, E., Techniques of Program Structure and Design, Prentice-Hall, Englewood Cliffs, N.J., 1976

6.11 REFERENCES ON R6502 PROGRAMMING

Butterfield, J. et. al., The First Book of KIM, Hayden, Rochelle Park, N.J., 1978

Foster, C. C., Programming a Microcomputer: 6502,
Addison-Wesley, Reading, Ma., 1978

Zaks, R., Microcomputer Programming: 6502, Sybex,
Berkeley, Ca., 1978