R6500

MICROCOMPUTER SYSTEM

PROGRAMMING MANUAL

R6500

Rockwell International

$5.00
Document No. 29650 N30
Rev. 1, February 1979

NOTICE

Rockwell International reserves the right to change this product and its specifications at any time without notice to improve its design or performance.

No responsibility is assumed by Rockwell International for use of this information; nor for any infringement of patents or other rights of third parties which may result from the use of this information.

TABLE OF CONTENTS

APPENDICES

LIST OF EXAMPLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTORY REMARKS

1.0 MANUAL INTRODUCTION

Welcome to the R6500 Microcomputer System user family. This Programming Manual is designed to work in conjunction with the R6500 System Hardware Manual, which describes the basic hardware considerations when using the Rockwell International microcomputer family.

Before reading this Programming Manual, it is suggested that the reader acquaint himself with the Hardware Manual in order to understand the components available in the R6500 system and how these components are interconnected to form the R6500 system. This Programming Manual develops the concepts of microprocessor (CPU) internal architecture and how it is used, with attention given to input/output considerations. Familiarity with the hardware will facilitate easier understanding of these important concepts.

In order to best serve the total customer base, this manual is written in two levels. The first is a very basic introduction to the R6500 system, and the second level is for the user who has to refer to the manual on more than an occasional basis and who wants to rapidly scan and find specific sections. For the user who is quite familiar with programming and the R6500 instruction set, the appendices are the best references in the sense that they summarize, in a series of tables for convenience, all of the data which are discussed in detail in the manual.

It is recommended that even the user who is an experienced programmer and familiar with microprocessors still take the time to read through the manual in detail. Some of the architectural concepts are different from those found in second-generation machines, and this manual instructs the user how to optimize the utilization of the microprocessor, while providing an introduction of its basic concepts.

Criticism of this manual is welcomed at all times. Of particular interest are cases where the user was unable, by use of the index and appendix, to rapidly find the answer to a question developed in the course of designing a microprocessor system. Welcomed also are any comments which will enhance the content and format of the manual in future editions or addenda.

1.1 MICROPROCESSOR ARCHITECTURE

The R6502 through R6507 and R6512 through R6515 are all 8-bit microprocessors. That means that 8 bits of data are transferred or operated upon during each instruction cycle or operation cycle.

All devices in the R6500 family operate on data 8-bits-at-a-time, although some of the operations will appear to be serial or 16-bit-wide operations. A later section of this manual discusses the use of sequential operations on an 8-bit basis and how one can accomplish 16-bit effective operands and addressing.

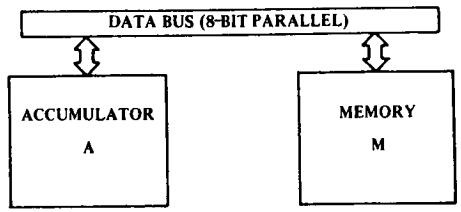
For some time, the computer industry has been designating 8-bit combinations of data by a term known as a "byte." In many large computers which operate simultaneously on multiple bytes of data, the number of bytes which are transferred and operated on by the machine in parallel are called a "word." Because the R6500 Microcomputer System's microprocessors are 8-bit microprocessors, the words and bytes are of equal length. Therefore, for convenience through the discussion of the basic 8-bit processors, "byte" and "word" will be used synonymously.

CHAPTER 2

THE DATA BUS, ACCUMULATOR AND ARITHMETIC UNIT

2.0 THE DATA BUS

Although most of the following discussion will consider how one operates with a general-purpose register called the accumulator, it must be understood that data has to transfer between the accumulator and outside sources by means of passing through the microprocessor to 8 lines called the data bus. The outside sources include the program which controls the microprocessor, the memory which will be used as interim storage for internal registers when they are to be used in a current operation, and the actual communications to the world through input/output (I/O) ports. Later in this document performance of transfers to and from each of these devices will be discussed. However, at present, discussion will center on the microprocessor itself.



Partial Block Diagram of a R6500 Microcomputer System Microprocessor

FIGURE 2.1

The only operation of the data bus is to transfer data between memory and the processor's internal registers such as the accumulator. Figure 2.1 displays the basic communication between the accumulator, A, and the memory, M, through the use of eight bidirectional data lines called the "data bus."

2.1 THE ACCUMULATOR

The accumulator is a register in which data are kept on which operations are performed. All operations between memory locations must be communicated through the accumulator or one of the auxiliary index registers. The accumulator is used as a temporary storage in moving data from one memory location to another. Therefore, the first use for the accumulator (A) is simply in the transfer of data from memory to the accumulator or from the accumulator to memory. One can bring data into the accumulator, perform operations such as AND/OR on the data, test the results of those operations, set new bits in the accumulator, or transfer the data back out to the outside world.

The accumulator serves as an interim storage for a series of operations such as adding two values together, with one of the values being loaded into the accumulator, the second added to it, and the results stored in the accumulator. The accumulator really serves two functions: 1) It is one of the primary storage points for the machine, and 2) it is the point at which intermediate results are normally stored.

2.1.1 LDA -- Load Accumulator with Memory

When instruction LDA is executed by the microprocessor, data are transferred from memory to the accumulator and stored in the accumulator.

Rather than continuing to give a word picture of the operation, introduced will be the symbolic representation M → A, where the arrow means "transfer to." Therefore, the LDA instruction symbolic representation is read "memory transferred to the accumulator."

LDA affects the contents of the accumulator, does not affect the carry or overflow flags; sets the zero flag if the accumulator is zero as a result of the LDA, otherwise resets the zero flag; sets the negative flag if bit 7 of the accumulator is a 1, otherwise resets the negative flag.

Although yet to be developed is the concept of addressing modes, for reference purposes LDA is a "Group One" instruction and has all of the major addressing modes of the machine available to it as

stated in Appendix A. These addressing modes include Immediate; Absolute; Zero Page; Absolute, X; Absolute, Y; Zero Page, X; Indexed Indirect; and Indirect Indexed.

2.1.2 STA -- Store Accumulator in Memory

This instruction transfers the contents of the accumulator to memory.

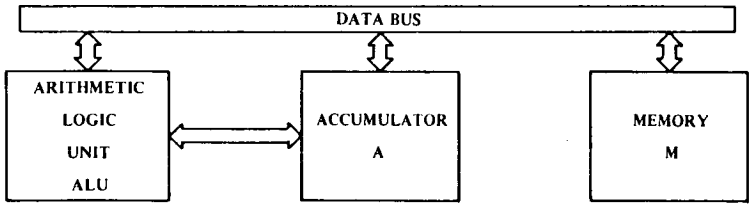
The symbolic representation for this instruction is A → M.

This instruction affects none of the flags in the processor status register and does not affect the accumulator.

It is a "Group One" instruction and has the following addressing modes available to it: Absolute; Zero Page; Absolute, X: Absolute, Y; Zero Page, X; Indexed Indirect; and Indirect Indexed.

2.2 THE ARITHMETIC UNIT

One of the functions to be expected from any computer is the ability to compute or perform arithmetic operations. Even in a simple control problem, one often finds it useful to add two numbers in order to determine that a value has been reached, or to subtract two numbers in order to calculate a new value which must be obtained. In addition, many problems involve some rudimentary form of decimal or binary arithmetic; certainly, many applications will involve both. The R6500 has an 8-bit arithmetic unit which interfaces to the accumulator as shown in Figure 2.2.



Partial Block Diagram Including Arithmetic Logic Unit of a R6500 Computer System Microprocessor
FIGURE 2.2

The arithmetic unit is composed of several major parts. The most important of these is the circuitry necessary to perform a two's complement add of 8-bit parallel values and to generate an 8-parallel-bit binary result plus a carry. A review of binary and binary-coded decimal (BCD) arithmetic is presented in Appendix H. However, a quick review of the concept of "carry" is in order. The largest range that can be represented in an 8-bit number is 256, with values ranging between 0 and 255. If we add any two numbers which result in a sum which is greater than 255, we represent the result with a ninth bit plus the 8 bits of the excess over 255. The ninth bit is called "carry."

2.2.1 ADC -- Add Memory to Accumulator with Carry

This instruction adds the value of memory and carry from the previous operation to the value of the accumulator and stores the result in the accumulator.

The symbolic representation for this instruction is $A + M + C \rightarrow A$.

This instruction affects the accumulator. It sets the carry flag when the sum of a binary add exceeds 255 or when the sum of a decimal add exceeds 99; otherwise carry is reset. The overflow flag is set when the sign or bit 7 is changed due to the result exceeding +127 or -128; otherwise overflow is reset. The negative flag is set if the accumulator result contains bit 7 on; otherwise the negative flag is reset. The zero flag is set if the accumulator result is 0; otherwise the zero flag is reset.

It is a "Group One" instruction and has the following addressing modes: Immediate; Absolute; Zero Page: Absolute, X; Absolute, Y; Zero Page, X; Indexed Indirect; and Indirect Indexed.

The ninth bit of the result is stored in the carry flag and the remaining 8 bits reside in the accumulator. The carry flag can be thought of as a flag bit which is remote from the accumulator itself, but which is directly affected by accumulator operations as though it were a ninth bit in the accumulator. The primary reason for not viewing the carry bit as merely a ninth bit in the accumulator is that one has program control over its state by being able to set (to "1") or clear (to "0") the bit and, of course, it is not part of the 8-bit accumulator in data transfer operations. Examples employing the Add with Carry operation follow.

Example 2.1: Add 2 Numbers with Carry: No Carry Generation

	0000 1101	13	= (A)*
	1101 0011	211	= (M)*
	1	1	= CARRY
Carry = $\boxed{0}$	1110 0001	225	= (A)

*(A) and (M) refer to the "contents" of the accumulator and "contents" of memory respectively.

Example 2.2: Add 2 Numbers with Carry; Carry Generation

```
                1111   1110    254 = (A)
                0000   0110      6 = (M)
                       1         1 = CARRY
Carry = /1/     0000   0101      5 = (A)
```

While the accumulator contains "5," the carry flag signals the user that the result exceeded 255 and, therefore, the result can be properly interpreted as 256 + 5 = 261.

2.2.1.1 Multiple-Precision Addition

To perform the addition of two numbers, one issues to the microprocessor an ADC instruction which adds the memory and the accumulator, and stores the results in the accumulator with the carry bit being set if the results exceed 255.

To add numbers with significantly higher value than 255, it would be necessary to represent these numbers by a series of serial 8-bit numbers. With the 16 bits in two serial 8-bit numbers, it is possible to represent binary numbers of greater than 65,000 in value. In order to add two 16-bit numbers together and thus accomplish double-precision addition, one first loads the lowest byte of one number into the accumulator, clears the carry flag and then adds the second number to the first number in the accumulator using the ADC command. One then stores this result into another memory location using the STA command. The carry flag now represents the carry from the lowest byte to the highest byte. One can then load the high-order byte of the first number, add with carry again to the high value of the second number, and store the result in the high-order byte of the result. Thus, it can be seen that the carry allows us to perform as much precision arithmetic as is necessary. The example listing below displays the commands used to execute the addition of two 16-bit numbers.

Example 2.3: Adding Two 16-Bit Numbers

	High-Order-Byte	Low-Order Byte
First Number	H1	L1
Second Number	H2	L2
Result of Addition	H3	L3

```
LDA     L1      Load low-order byte, first number
CLC             Clear carry flag (carry = 0)
ADC     L2      Add L1 to low-order byte, second number
STA     L3      Store result in memory, carry flag is
                still set if set in ADC operation
LDA     H1      Load high-order byte, first number
ADC     H2      Add H1 and carry value from first ADC
                operation to high-order byte, second
                number
STA     H3      Store result in memory
```

In this example it was necessary to clear the carry flag before starting the add instruction. This, of course, means that commands exist that set and clear the carry flag allowing for addition without values generated from the prior operation. One could also, at the end of the program, check to see if the result exceeded 16 bits by testing the carry flag. Exactly how one alters and tests flags will be discussed in the Flag and Branches Sections. The examples below display the concept of carry from the addition of the low order bytes.

Example 2.4: Add Two 16-Bit Numbers, No Carry from Low-Order Add

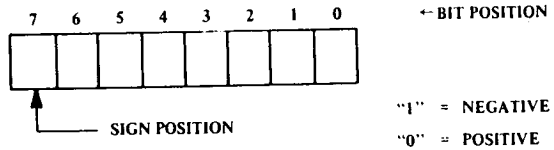
```
          0000   0001   0000   0010    258
          0001   0000   0001   0000   4112

          Add low-order bytes:  (clear carry)

               0000   0010   (A)
               0001   0000   (M)
Carry = /0/    0001   0010   (A)

          Add high-order bytes (carry = 0):

               0000   0001   (A)
               0001   0000   (M)
                         0   CARRY
Carry = /0/    0001   0001   (A)

     Result = 0001   0001   0001   0010 = 4370
```

Example 2.5: Add Two 16-Bit Numbers, with Carry from Low-Order Add

```
          0000   0001   1000   0000    384
          0000   0000   1000   0000    128

          Add low-order bytes:  (clear carry)

               1000   0000   (A)
               1000   0000   (M)
Carry = /1/    0000   0000   (A)

          Add high-order bytes:  (carry = 1)

               0000   0001   (A)
               0000   0000   (M)
                         1   CARRY
Carry = /0/    0000   0010   (A)

     Result = 0000   0010   0000   0000 = 512
```

2.2.1.2 Signed Arithmetic

It is possible to look at the add operation and the way data are represented in memory in a different way. If, in the 16-bit problem (Examples 2.4 and 2.5), one were working with 15 bits of precision (in other words, 15 bits of valid data) plus 1 bit of sign (0 for positive and 1 for negative), it would be possible to perform signed binary arithmetic without changing the adder, but by merely changing the way the results are interpreted. In order to facilitate this concept, the microprocessor has the ability to represent positive or negative numbers by means of a negative flag which will be discussed length in Section 3.7. In the R6500 microprocessor family, bit 7 is the sign position bit. This means that the highest-order byte in a series of bytes should have the sign in the eighth position. If, for simplicity, one talks about signed 8-bit numbers, it would mean that one was allowed only 128 combinations of each sign because that is the most that can be represented in 7 bits, with the eighth bit or the highest bit reserved for the sign position.



Byte Orientation with Sign Position

FIGURE 2.3

In the following examples of signed arithmetic it should be noted that operations are occurring on a 7-bit field of numbers and that any carry generated out of that field will reside in the eighth bit -- not in the carry flag discussed during the add operations. The generation of a carry out of the field is the same as when adding two 8-bit numbers, except for the fact that the normal carry flag does not correctly represent the fact that the field has been exceeded. This is because the true carry from adding the two 7-bit numbers resides in the sign bit position. Therefore, the carry flag has no real meaning. Instead, there is a separate flag, the overflow flag, which is used to indicate when a carry from 7 bits has occurred and allows the user to write correction programs.

In each example, the negative numbers are in two's complement form. Also included in each result will be the status of the carry and overflow flags. The overflow flag is set whenever the sign bit (bit 7) is changed as a result of the operation.

Example 2.6: Add Two Positive Numbers with No Overflow

```
          0000   0101   +5   (A)
          0000   0111   +7   (M)
Carry = /0/    0000   1100   +12  (A)
Overflow = /0/
```

"0" in bit 7 indicates positive result. Note that both the carry and overflow flag remain cleared.

Example 2.7: Add Two Positive Numbers with Overflow

	0111	1111	+127	(A)
	0000	0010	+ 2	(M)
Carry = /0/	1000	0001	"-127"	(A)

Overflow = /1/ "1" in bit 7 indicates negative result and the two's complement of the result is 127; however, the overflow flag is set indicating the allowable range was exceeded in the addition.

Therefore, examination of the overflow indicated that the result was in fact not negative but that the bit 7 position represented an overflow beyond the value of 127. Hence the user is flagged of an incorrect result and a correction routine (program) must follow.

Example 2.8: Add Positive and Negative Number with Positive Result

	0000	0101	+5	(A)
	1111	1101	-3	(M)
Carry = /1/	0000	0010	+2	(A)

Overflow = /0/ "0" in bit 7 indicates positive result. (Recall that though the carry flag is set, it has no meaning in signed operations.)

Example 2.9: Add Positive and Negative Number with Negative Result

	0000	0101	+5	(A)
	1111	1001	-7	(M)
Carry = /0/	1111	1110	-2	(A)

Overflow = /0/ "1" in bit 7 indicates negative result.

Example 2.10: Add Two Negative Numbers without Overflow

	1111	1011	-5	(A)
	1111	1001	-7	(M)
Carry = /1/	1111	0100	-12	(A)

Overflow = /0/ "1" in bit 7 indicates negative result.

Example 2.11: Add Two Negative Numbers with Overflow

	1011	1110	-66	(A)
	1011	1111	-65	(M)
Carry = /1/	0111	1101	"+125"	(A)

Overflow = /1/ "0" indicates positive result, but the overflow flag is set indicating that the allowable range was exceeded in the operation. Without the overflow indication, the result would be interpreted as +125. The overflow, however, indicated that the result was negative and exceeded the value -128. Hence the user is flagged of an incorrect result, indicating the need for a correction routine.

2.2.1.3 Decimal Addition

There is a way for the user to organize data for decimal operations. The R6500 System's microprocessors have a modified adder which allows the user to represent his numbers as two 4-bit binary coded decimal (BCD) numbers packed into a single byte. This is a unique feature of the R6500 microprocessor family in that the operation in the following example can be performed.

Example 2.12: Decimal Addition

```
CLC                    Clear Carry Flag
SED                    Set Decimal Mode
LDA   0111   1001      79
ADC   0001   0100     +14
STA   1001   0011      93
```

The microprocessor adder has the unique capability of performing real time correction to the normal expected binary result without any direct interference from the programmer. Other popular microprocessors require a separate instruction (Decimal Adjust) which corrects the direct binary result of the arithmetic unit to obtain the same final results as are available on this microprocessor directly.

In order to make the same arithmetic unit perform either as a binary adder or as a decimal adder, the user chooses the mode in which he is going to operate (either decimal or binary) by setting

another flip-flop in the microprocessor called the "decimal flag". As shown in this example, one not only initializes the adder by clearing the carry flag, but also puts the processor into decimal mode with the SED instruction. Even though this also requires one instruction, it is possible to put the machine in decimal mode once and perform many long strings of decimal numbers without further user intervention. The "Decimal Adjust" feature on other microprocessors requires programming subsequent to each binary operation. The CLD instruction returns the arithmetic unit to the binary adder mode.

2.2.1.4 Add Summary

In summary, the basic arithmetic unit is a binary adder which, under control of the ADC command, performs binary arithmetic on the accumulator and data, storing the result in the accumulator. Depending on the way the user looks at the data which are presented to the adder and the results which are obtained, the user can determine whether or not the result exceeds 255 binary or 99 decimal; he can perform precision arithmetic by use of the ninth bit or carry flag; he can control whether or not the microprocessor is a decimal adder by setting the decimal mode; and he can represent his numbers as signed binary numbers by analyzing other flags that are set in the machine.

2.2.2 SBC Subtract Memory from Accumulator with Borrow

This instruction subtracts the value of memory and borrow from the value of the accumulator, using two's complement arithmetic, and stores the result in the accumulator. Borrow is defined as the carry flag complemented; therefore, a resultant carry flag indicates that a borrow has not occurred.

The symbolic representation for this instruction is

$A - M - \overline{C} \rightarrow A$.

This instruction affects the accumulator. The carry flag is set if the result is greater than or equal to 0. The carry flag is reset when the result is less than 0, indicating a borrow. The overflow flag is set when the result exceeds +127 or -128; otherwise, it

is reset. The negative flag is set if the result in the accumulator has bit 7 on, otherwise it is reset. The Z flag is set if the result in the accumulator is 0; otherwise, it is reset.

It is a "Group One" instruction. It has addressing modes Immediate; Absolute; Zero Page; Absolute, X; Absolute, Y; Zero Page, X; Indexed Indirect; and Indirect Indexed.

In a binary machine, the classical way to perform arithmetic is by using two's complement notation. In using two's complement notation, any subtraction operation becomes a sequence of bit complementations and additions. This reduces the complexity of the circuits required to perform a subtraction.

When the SBC instruction is employed in single-precision subtraction, there will normally be no borrow; therefore, the programmer must set the carry flag, by using the SEC (Set carry to 1) instruction, before using the SBC instruction. The microprocessor adds the carry flag to the complemented memory data, resulting in a true two's complement form of the memory value with its sign inverted.

Example 2.13: Subtract Two Numbers with Borrow; Positive Result

Assume a single precision subtraction where A contains 5 and M contains 3. The carry flag must be set to a 1 using the SEC instruction, thereby representing the no-borrow condition.

The adder changes the sign of M by taking the two's complement of M. This involves complementing M and adding the carry bit.

M = 3	0000	0011
Complemented M	1111	1100
Add C = 1		1
-M = -3	1111	1101

The adder adds A and the two's complement -M together. This operation occurs simultaneously with the complement operation.

A = 5	0000	0101
Add -M = -3	1111	1101
Carry = /1/	0000	0010 = +2

The presence of the carry flag after this operation indicates that No Borrow was required, therefore the result is +2.

Example 2.14: Subtract Two Numbers with Borrow; Negative Result

Assume a single-precision subtraction where A contains 5 and M contains 6. Set the carry flag to a 1 with SEC to indicate No Borrow.

```
          M = 6     0000   0110
Complemented M      1111   1001
     Add C = 1                 1
        -M = -6     1111   1010

          A = 5     0000   0101
    Add -M = -6     1111   1010
 Carry = /0/        1111   1111 = -1
```

The absence of the carry flag after this operation indicates that a borrow was required, therefore the result is a -1 in two's complement form. The absolute (unsigned) result in straight binary could be obtained by taking the two's complement of this number.

2.2.2.1 Multiple-Precision Subtraction

Double-precision subtraction is implemented in a fashion similar to addition. An example for subtracting a 16-bit number and storing the result follows:

Example 2.15: Subtracting Two 16-Bit Numbers

	High-Order Byte	Low-Order Byte
First Number	H1	L1
Second Number	H2	L2
Result of Subtraction	H3	L3

```
SEC          Set Carry
LDA   L1     Load-Low Order Byte, First Number
SBC   L2     Subtract with Borrow, Low-Order Byte of Second
             Number from L1
STA   L3     Store Result in Memory
LDA   H1     Load High-Order Byte, First Number
SBC   H2     Subtract with Borrow, High-Order Byte of Second
             Number from H1
STA   H3     Store Result in Memory
```


Example 2.16: Subtract in Double-Precision Format; Positive Result

Assume a double-precision subtraction where 255 is to be subtracted from 512 for an example. Since there has been no borrow coming into this subtraction operation, the carry flag must be set.

Following are the 2 numbers in binary form:

	High-Order Byte		Low-Order Byte	
A field = 512	0000	0010	0000	0000
M field = 255	0000	0000	1111	1111

Since the adder can only operate on single byte numbers, the programmer must operate on the low-order bytes first.

```
          M = 1111   1111
Complemented M = 0000   0000
     Add C = 1                 1
        -M      0000   0001

          A = 0000   0000
    Add -M = 0000   0001
 Carry = /0/ 0000   0001
```

The carry is brought over to the subtract operation on the high-order bytes.

```
          M = 0000   0000
Complemented M = 1111   1111
     Add C = 0                 0
        -M      1111   1111

          A = 0000   0010
    Add -M = 1111   1111
 Carry = /1/ 0000   0001
```

The result in binary form follows:

```
Carry = /1/     0000   0001   0000   0001 = +257
```

The presence of the carry flag after the highest-order byte subtraction indicates that the entire number required No Borrow, therefore it is a positive number in straight binary form.

Example 2.17: Subtract in Double Precision Format; Negative Result

Now assume a double-precision subtraction where 512 is to be subtracted from 255. Again, since there has been no borrow coming into this subtraction operation, the carry flag must be set.

Following are the two numbers in binary form:

	High-Order Byte		Low-Order Byte	
A field = 255	0000	0000	1111	1111
M field = 512	0000	0010	0000	0000

Operating on the low-order byte:

```
          M = 0000  0000
          M̄ = 1111  1111
Add C = 1           1
Carry = /1/ 0000  0000  = -M

          A = 1111  1111
Add -M = /1/ 0000  0000
Carry = /1/ 1111  1111
```

The presence of the carry = 1 indicates no borrow.

The carry is now brought over to the high-order byte subtract operation:

```
          M = 0000  0010
          M̄ = 1111  1101
Add C = 1           1
            1111  1110

          A = 0000  0000
      M̄ + C = 1111  1110
Carry = /0/ 1111  1110
```

The result in binary form is:

Carry = /0/ 1111 1110 1111 1111 = -257

Carry = /0/ indicates the presence of a borrow, therefore the number is negative and is in two's complement form.

2.2.2.2 Signed Arithmetic

Signed numbers can be subtracted, using the SBC instruction, just as easily as they can be added. The microprocessor converts the numbers from memory to its two's complemented form and then adds it to the value of the accumulator just as it does in an unsigned

subtract described in Section 2.2.2. The addition operation is identical to that described, and to the examples given in Section 2.2.1.2

It should be remembered that before using the SBC instruction, either signed or unsigned, the carry flag must be set to a 1 in order to indicate a no borrow condition. The resultant carry flag has no meaning after a signed arithmetic operation.

2.2.2.3 Decimal Subtract

As indicated in Section 2.2.1.3, it is possible to represent numbers as packed 4-bit BCD numbers. In this case, which is again unique to this microprocessor, it is possible to make the adder act as though it were a decimal adder. In this case, the function of the machine is one of correcting for the subtraction of positive numbers by complementing the number, setting the carry and performing binary arithmetic with an automatic correction at the time the result is stored in the accumulator. The unique capabilities of this adder give the results as shown in the next example.

Example 2.18: Decimal Subtraction

```
SED                       Set Decimal Mode
SEC                       Set Carry Flag
LDA   0100  0100    44
SBC   0010  1001    29
STA   0001  0101    15
```

By setting the decimal mode and setting the carry flag, one can subtract number 29 from number 44 with the results in the accumulator automatically being 15.

As has been indicated, one can perform both addition and subtraction when the machine is set in decimal mode, treating the bytes to be added as unsigned, positive, binary-coded digits. In addition the carry flag represents the case when the result in the number exceeded 99, and in subtraction the absence of the carry flag represents a true borrow situation.

2.2.3 Carry and Overflow During Arithmetic Operations

It is necessary to set or reset the carry flag prior to the beginning of any arithmetic instruction. Because the carry flag is set or reset as a result of the arithmetic operation at the end of the loop, one can test the flag to determine whether or not a carry or a borrow occurred in the operation. By proper use of the overflow flag one can treat the high-order bit of any set of bytes as a sign bit as long as the results of the negative numbers are carried in two's complement form. The microprocessor also sets the overflow flip-flop to indicate when a result larger than can be stored in a 7-bit field has occurred and when the resultant sign is incorrect. In binary arithmetic the carry flag set indicates results in excess of 256, and in decimal arithmetic indicates results in excess of 99. Although the input carry is very important to these operations, a simple rule is: Set the carry flag prior to subtract; clear the carry flag prior to add.

2.2.4 Logical Operands

In implementing a parallel binary adder there are several useful logic functions which are subsets of a binary add operation. In the R6500 System's microprocessor family, these subsets are employed t implement the logical operands "AND," "OR," and "EOR" (Exclusive Or). These operations are used to test and control bit manipulations.

2.2.4.1 AND -- Memory with Accumulator

The AND instruction transfers the accumulator and memory to the adder which performs a bit-by-bit AND operation and stores the result back in the accumulator.

This instruction affects the accumulator; sets the zero flag if the result in the accumulator is 0, otherwise resets the zero flag; sets the negative flag if the result in the accumulator has bit 7 on, otherwise resets the negative flag.

This is symbolically represented by A $\wedge$ M $\rightarrow$ A.

AND is a "Group One" instruction having addressing modes of Immediate; Absolute; Zero Page; Absolute, X; Absolute, Y; Zero Page, X; Indexed Indirect; and Indirect Indexed.

One of the uses for the AND operation is that of resetting a bit in memory. In the example below,

Example 2.19: Clearing a Bit with AND

```
LDA    1100   X111, where X is 0 or 1
AND    1111   0111
STA    1100   0111
```

a byte is loaded into the accumulator and the AND instruction resets the accumulator bit 3 to 0. The accumulator is then stored back into memory, thereby resetting the bit.

2.2.4.2 ORA "OR" Memory with Accumulator

The ORA instruction transfers the memory and the accumulator to the adder which performs a binary "OR" on a bit-by-bit basis and stores the result in the accumulator.

This is indicated symbolically by A $\vee$ M $\rightarrow$ A.

This instruction affects the accumulator; sets the zero flag if the result in the accumulator is 0, otherwise resets the zero flag; sets the negative flag if the result in the accumulator has bit 7 on, otherwise resets the negative flag. ORA is a "Group One" instruction. It has the addressing modes Immediate; Absolute; Zero Page; Absolute, X; Absolute, Y; Zero Page, X; Indexed Indirect; and Indirect Indexed.

To set a bit, the OR instruction is used as shown below:

Example 2.20: Setting a Bit with OR

```
LDA    1110   X111, where X is 0 or 1
ORA    0000   1000
STA    1110   1111
```

2.2.4.3 EOR -- "Exclusive OR" Memory with Accumulator

The EOR instruction transfers the memory and the accumulator to the adder which performs a binary "EXCLUSIVE OR" on a bit-by-bit basis and stores the result in the accumulator.

This is indicated symbolically by A $\forall$ M → A.

This instruction affects the accumulator; sets the zero flag if the result in the accumulator is 0, otherwise resets the zero flag; sets the negative flag if the result in the accumulator has bit 7 on, otherwise resets the negative flag.

EOR is a "Group One" instruction having addressing modes of Immediate; Absolute; Zero Page; Absolute, X; Absolute, Y; Zero Page, X; Indexed Indirect; and Indirect Indexed.

One of the uses of the EOR instruction is in complementing bytes. This is accomplished below by exclusive ORA-ing the byte with all 1's.

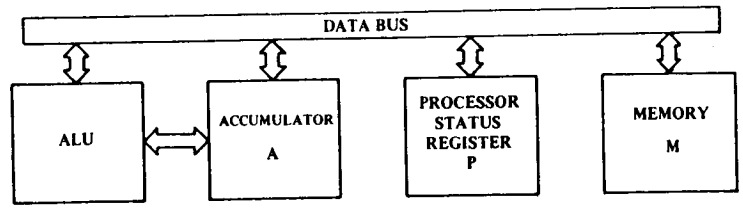
Example 2.21: Complementing a Byte with EOR

```
LDA   1010   1111
EOR   1111   1111
STA   0101   0000
```


CHAPTER 3

CONCEPTS OF FLAGS AND STATUS REGISTER

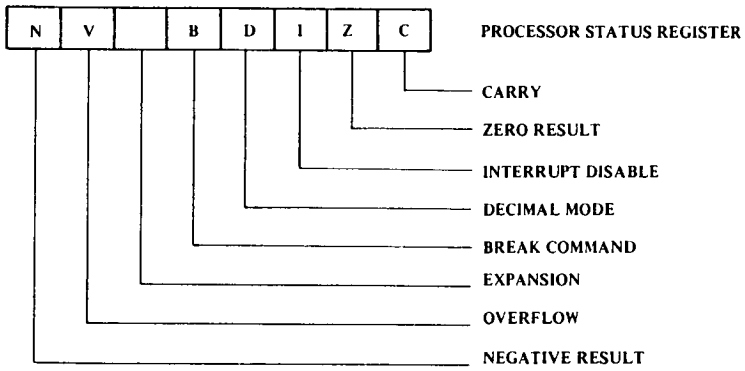
One can view each of the individual flags or status bits in the machine as individual flip-flops. The carry flag can be considered the ninth bit of an arithmetic operation. The decimal mode flag is set and cleared by the user and used by the microprocessor to select either binary or decimal mode. For programming convenience the microprocessor treats all of the flags or status bits as component bits of a single 8-bit register. In Figure 3.1 the processor status register (or "P" register) is added to the block diagram.



Partial Block Diagram of a R6500 Microcomputer System Microprocessor Including P Registers

FIGURE 3.1

Each of the individual flags or bits has its own particular meaning in the microprocessor as defined in Figure 3.2



Processor Status Register "P"

FIGURE 3.2

3.0 CARRY FLAG (C)

The carry bit which is modified as a result of specific arithmetic operations or by a set or clear carry command has been discussed previously. In the case of shift and rotate instructions, the carry bit is used as a ninth bit as it is in the binary arithmetic operation. The carry flag can be set or reset by the programmer. A SEC instruction will set and a CLC instruction will reset the carry flag. Operations which affect the carry are ADC, ASL, CLC, CMP, CPX, CPY, LSR, PLP, ROL, ROR, RTI, SBC, and SEC.

3.0.1 SEC -- Set Carry Flag

This instruction initializes the carry flag to a 1. This operation should normally precede a SBC loop.

This instruction affects no registers in the microprocessor and no flags other than the carry flag which is set.

3.0.2 CLC -- Clear Carry Flag

This instruction initializes the carry flag to a 0. This operation should normally precede an ADC loop.

This instruction affects no registers in the microprocessor and no flags other than the carry flag which is reset.

3.1 ZERO FLAG (Z)

This flag is automatically set by the microprocessor during any data movement or calculation operation when the 8 bits of results of the operation are 0. Therefore, the bit is on ("1") when the results are 0, and off ("0") when the results are not equal to 0. The feature of the machine is similar to that of the PDP11 in the sense that operations which are decrementing (or incrementing) index registers or memory locations have a built-in test for 0 as a result of decrementing (or incrementing) to the 0 condition. It is also possible to test for 0 condition immediately following load and other logical operations, as opposed to processors which have to do a test and branch instruction. The Z flag is not directly settable or resettable by an instruction but is affected by the following instructions: ADC, AND, ASL, BIT, CMP, CPY, CPX, DEC, DEX, DEY, EOR, INC, INX, INY, LDA, LDX, LDY, LSR, ORA, PLA, PLP, ROL, ROR, RTI, SBC, TAX, TAY, TXA, TSX AND TYA. The Z flag is not updated after a resulting decimal addition or subtraction (ADC, SBC).

3.2 INTERRUPT DISABLE (I)

The interrupt disable is a flip-flop made use of by the programmer and by the microprocessor to control the operations of the interrupt request pin. A more detailed discussion of the effects of the interrupt disable are given in the discussion under interrupt control. However, the purpose of the interrupt disable is to disable the effects of the interrupt request pin ($\overline{IRQ}$). The interrupt disable, I, is set by the microprocessor during reset and interrupt commands. The I bit is reset by the CLI instruction or the PLP instruction, or at a return from interrupt in which the interrupt disable was reset prior to the interrupt. The interrupt flag may be set by the programmer using a SEI instruction and is cleared by the

programmer with a CLI instruction. Instructions which affect the interrupt disable are BRK, CLI, PLP, RTI and SEI.

3.2.1 SEI -- Set Interrrupt Disable

This instruction initializes the interrupt disable to a 1. It is used to mask interrupt requests during system reset operations and during interrupt commands.

It affects no registers in the microprocessor and no flags other than the interrupt disable which is set.

SEI is a single-byte instruction and its addressing mode is Implied.

3.2.2 CLI -- Clear Interrupt Disable

This instruction initializes the interrupt disable to a 0. This allows the microprocessor to receive interrupts.

It affects no registers in the microprocessor and no flags other than the interrupt disable which is cleared.

CLI is a single-byte instruction and its addressing mode is Implied.

3.3 DECIMAL MODE FLAG (D)

As discussed, the purpose of the decimal mode flag is to control whether or not the adder operates as a straight binary adder for add and subtract instructions, or as a decimal adder for add and subtract instructions. The SED instruction sets the flag and the CLD instruction resets it. The only instructions which affect the decimal mode flag are CLD, PLP, RTI and SED.

3.3.1 SED -- Set Decimal Mode

This instruction sets the decimal mode flag D to a 1. This makes all subsequent ADC and SBC instructions operate as a decimal arithmetic operation.

SED affects no registers in the microprocessor and no flags other than the decimal mode which is set to a 1.

3.3.2 CLD -- Clear Decimal Mode

This instruction sets the decimal mode flag to a 0. This causes all subsequent ADC and SBC instructions to operate as simple binary operations.

CLD affects no registers in the microprocessor and no flags other than the decimal mode flag which is set to a 0.

3.4 BREAK COMMAND (B)

The break command flag is set only by the microprocessor and is used to determine during an interrupt service sequence whether or not the interrupt was caused by BRK command or by a real interrupt. A more detailed discussion of BRK is in the interrupt section. This bit should be considered to have meaning only during an analysis of a normal interrupt sequence. There are no instructions which can set or which reset this bit.

3.5 EXPANSION BIT

The next bit in the flag register is an unused bit. It is most likely that this bit will appear to be on when one is analyzing the bit pattern in the processor status register; however, no guarantee as to its state is made, as this bit will be used in expanded versions of the microprocessor.

3.6 OVERFLOW (V)

As discussed in the section on arithmetic operations, if one is to look at the binary arithmetic operations as signed binary operations, there needs to be some indication of the fact that the result of the arithmetic operation has a greater value than could be contained in the 7 bits of the result. This bit is the overflow bit and during ADC and SBC instructions represents a status of an overflow into the sign position. The user who is not using signed arithmetic can totally ignore this flag during his programming; however, this flag has the same meaning as the carry to the user who is using signed binary numbers. It indicates that a sign correction routine must be used if this bit is on after an add or subtract using signed numbers.

In addition to its use for monitoring the validity of the sign bit in ADC and SBC instructions, the overflow flag is dramatically different from the overflow flags from PDP11 and the MC6800. In each of those systems the overflow flag was very carefully controlled so as to allow certain signed branches for analysis of signed numbers. These branches have been deleted from the R6500 System's microprocessor series because of confusion and difficulty often associated with using them, and, accordingly, the overflow flag is applicable only to the operation of ADC and SBC, and then only when signed numbers are being used.

However, in order to maximize the effectiveness of this testable flag the BIT instruction which may be used to sample interface devices, allows the overflow flag to reflect the condition of bit 6 in the sampled field. During a BIT instruction the overflow flag is set equal to the content of the bit 6 on the data tested with BIT instruction. When used in this mode, the overflow has nothing to do with signed arithmetic but is just another sense bit for the microprocessor. Instructions which affect the V flag are ADC, BIT, CLV, PLP, RTI and SBC. On certain versions of the microprocessor the V bit will also be available for stimulus from the outside world.

3.6.1 CLV -- Clear Overflow Flag

This instruction clears the overflow flag to a 0. This command is utilized in conjunction with the set overflow pin which can change the state of the overflow flag with an external signal.

CLV affects no registers in the microprocessor and no flags other than the overflow flag which is set to a 0.

3.6.2 Determination of Overflow

To briefly recap the concept of overflow detection, one must understand that the machine signals an overflow based on the data entered to the operation and the final result. Since, with signed arithmetic, the range of numbers that can be represented is +127 to -12 the overflow flag will never be set when numbers of opposite sign are added, since their result will never exceed that range. The machine deals with this by recognizing that for any two positive numbers, the "bit 7" of each is a "0," and that for any arithmetic operation

yielding a result less than or equal to +127, the resultant "bit 7" must be a "0." If it is a 1, the overflow flag is set.

Similarly, when two negative numbers are added, the "bit 7" of each is a "1" and for any result yielding a value less than or equal to -128, the resultant "bit" must be a "1." If it is a 0, the overflow flag is set.

Therefore, the machine recognizes by knowledge of the "bit 7" of each of the numbers to be added what the resultant "bit 7" must be in a non-overflow situation. If these conditions are not met, the overflow flag goes set.

3.7 NEGATIVE FLAG (N)

As already discussed, one of the uses of the microprocessor is to perform arithmetic operations on signed numbers. To allow the user to readily sample the status of the sign bit (bit 7), the N flag is set equal to bit 7 of the resulting value in all data movement and data arithmetic. This means, for instance, after a signed add one can determine the sign of the result by sampling the N flag directly rather than finding a way to isolate bit 7. Although signs were the primary purpose for which the N flag was intended, its usefulness far exceeds that of strictly a sign bit. Because of every operation including simple moves and add operations the N bit is equal to the status of bit 7 as a result of the operation; its primary use becomes that of an easily testable bit. Almost all single-bit instructions, all interrupts and all I/O status flags use bit 7 as a sense bit. This allows the user to perform some type of memory access operation such as Load A followed by immediate conditional branch based on the status of bit 7 as reflected in the N flag. Like the Z bit, this flag is not settable or controllable by the programmer and represents the status of the last data movement operation. Instructions which affect the negative flag are ADC, AND, ASL, BIT, CMP, CPY, CPX, DEC, DEX, DEY, EOR, INC, INX, INY, LDA, LDX, LDY, LSR, ORA, PLA, PLP, ROL, ROR, RTI, SBC, TAX, TAY, TSX, TXA and TYA.

3.8 FLAG SUMMARY

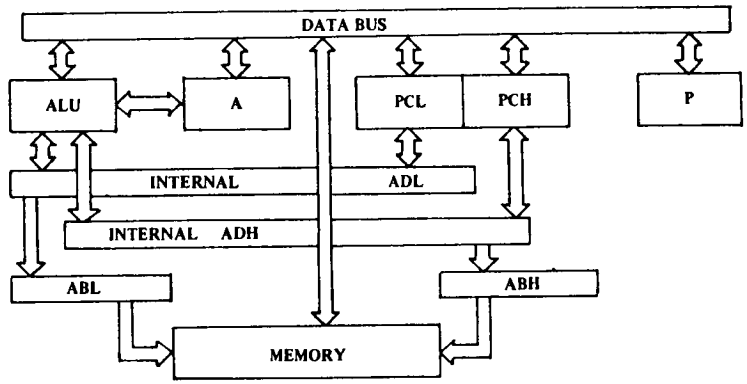
To summarize, the microprocessor treats a series of flags or status bits as a single register called the "P" or "Program Status" register. Some of these flags are controllable only by the programmer (such as the D flag); others are controllable by both the user program and microprocessor (such as the interrupt disable flag). Some of them are set and reset by almost every processor operation, such as the N and Z flags. Each of these flags has its own meaning to the programmer at a particular point in time. When combined with the concept of conditional branches, they represent a powerful test and jump capability not normally found in a machine of this magnitude. Except, perhaps, for the carry flag which is used as part of the arithmetic instructions, the flags by themselves have relatively little meaning unless one has the ability to test them. For this purpose a series of conditional branch instructions is designed into the machine.

CHAPTER 4

TEST, BRANCH AND JUMP INSTRUCTIONS

4.0 CONCEPTS OF PROGRAM SEQUENCE

To this point, this manual has presented little discussion of how the microprocessor understands the instructions used to perform various arithmetic and accumulator manipulations. However, it is appropriate that the concept of a program and how the microprocessor determines each instruction be developed. More registers are required in the machine as shown in the figure below.



Partial Block Diagram of a R6500 Microcomputer System Microprocessor Including Program Counter and Internal Address Bus

FIGURE 4.1

Although two 8-bit registers have been added, they are the only registers in the machine that act as though they are one 16-bit register. They implement a concept known as "program count" or "program sequence", and subsequently their value will be referred to as "PC" or "program count". In certain operations it may be convenient to talk about how one affects the "program count low" (PCL) which will be the lower 8-bit register or the "program count high" (PCH) which will be the higher 8-bit register. The reason for this register's being 16 bits in length is that if it had only 8 bits it would be able to reference only 256 locations. Since it is through the address bus that one accesses memory, the program counter which defines the addressable location should be as "wide" a word as possible.

The accessing of a memory location is called "addressing". It is the selection of a particular eight-bit data word (byte) out of the 65,536 possibilities for memory data locations. This selection is transmitted to the memory through the 16 address lines (ADH, ADL) of the microprocessor.

For a more detailed discussion of how an individual memory byte is selected by the address lines, the reader is referred to Chapter 1 of the Hardware Manual.

If the program counter were only 1 byte and if the bit pattern which allows the microprocessor to choose which instruction it wants to act on next, such as "LDA" as opposed to an "AND", were contained in one byte of data, we could have only 256 program steps. Although the machine of this length might make an interesting toy, it would have no real practical value. Therefore, almost all of the competitive 8-bit microprocessors are designed with a double-length program counter. Even though some of the microprocessors of the R6500 Microcomputer System do not have all of the output address lines necessary to allow the user to address 65K bytes of program (due to package pinout constraints), in all cases the program counter (PC) is capable of addressing a full 65K by virtue of its 16-bit length.

4.0.1 Use of Program Counter to Fetch an Instruction

The microprocessor contains an internal timing and state control counter. This counter, along with a decode matrix, governs the operation of the microprocessor on each clock cycle. When the state of the microprocessor indicates that a new instruction is needed, the program counter (program address pointer) is used to choose (address) the next memory location and the value which the memory sends back is decoded in order to determine what operation the microprocessor is going to perform next.

To utilize the program counter to perform this operation correctly, it must always be addressing the operation the user wants to perform next. This operation may be an instruction, or it may be data on which the instruction will operate.

In the R6500 System's microprocessor family, the program counter is set with the value of the address of an instruction. The microprocessor then puts the value of the program counter onto the address bus, transferring the 8 bits of data at that memory address into the instruction decode. The program counter then automatically increments by one, and the microprocessor fetches further data for address operation necessary to complete the instruction. In the simple example below,

Example 4.1: Accessing Instructions with the P-Counter Value

P Counter*	Location Contents	
0100**	LDA	*Program Counter
0101	ADC	**Hexadecimal
0102	STA	Notation

one can see how the program counter is used to access the instruction sequence load A, add with carry, and store the result. In this example, the program counter would start out containing 0100. The microprocessor would read location 0100 by using the program counter to access memory, and would then interpret and implement the LDA instruction as previously described. The program counter will automatically increment by 1 on each instruction fetch, stepping to 0101. After performing the LDA, the microprocessor would fetch the

next instruction addressing memory with the program counter. This would pick up the ADC instruction, the add would then be performed, the program counter which has been incremented to 0102 would be used to address the next instruction, STA. The P counter incrementing once with each instruction is an oversimplified view of what actually transpires within the microprocessor.

The R6500 series of microprocessors (CPUs) usually require more than one byte to correctly interpret an instruction. The first byte of an instruction is called the OP CODE, and it is coded to contain the basic operation such as LDA (load accumulator with memory) and also the data necessary to allow the microprocessor to interpret the address of the data on which the operation will occur. In most cases, this address will appear in memory right after the OP CODE byte. This allows the microprocessor to use the program counter to access the address as well as the OP CODE.

The following example shows how the program counter picks up the instruction and the address of data located at address 5155.

Example 4.2: Accessing Data Address With P-Counter Value

P Counter	Location Contents
0100	LDA
0101	55
0102	51
0103	Next Instruction

The OP CODE appears in Location Address 0100. The code for the 55 would appear next in Location Address 0101 and the 51 would appear in Location Address 0102, and the OP CODE for the next instruction appears in Location Address 0103. In this example, we see that the program counter is used not only to pick up the operation code, LDA, but also to pick up the address of the memory location from which the LDA is going to obtain its data. In this case, the program counter automatically is incremented three times to pick up the full instruction with the microprocessor interpreting each of the individual fetches as the appropriate data. In other words, the first

fetch is used to pick up the OP CODE, LDA, the second fetch is used to pick up the low-order address byte of the data, and the third fetch is used to pick up the high-order address byte of the data. This is the form in which many of the microprocessor instructions will appear, as it is the most simple form of addressing in the machine and allows referencing to any memory location.

Assuming that the microprocessor has the ability to start the program counter at a known instruction, it should be fairly obvious that the program counter would then continue to advance from that location up to the maximum memory location, roll over to the least memory location and continue incrementing through the memory, fetching instructions and addresses as it went. This would give us an interesting sequential program but one which lacked one tremendously powerful concept. The program would have no ability to perform tests or implement various options based on the results of those tests.

In the previous section, the concept of flags which are set as a result of the microprocessor operations was developed.

To use these flags, the program should be able to test them and then change the sequence of operations which are being performed depending on the result of the test. The program counter is going to continually put out an address, the microprocessor is going to fetch the instruction stored at that address and perform operations based on that instruction. In order to change a sequence of performed instructions by the microprocessor, the programmer must change the value in the program counter. Therefore, test instructions are incorporated which may result in a change of program count sequence as a result of performing one of the tests. The simplest way to change program sequence is to substitute a new value into the program counter location. In the R6500 System's series of microprocessors the simplest way to change the program count sequence is with a JMP instruction.

4.0.2 JMP -- Jump to New location

In this instruction, the data from the memory location located in the program sequence after the OP CODE is loaded into the low-order byte of the program counter (PCL) and the data from the next memory location after that is loaded into the high-order byte of the program counter (PCH).

The symbolic notation for jump is (PC + 1)→PCL, (PC + 2)→PCH. As stated earlier, the "( )" means "contents of" a memory location. PC indicates the contents of the program counter at the time the OP CODE is fetched. Therefore (PC + 2)→PCH reads, "the contents of the program counter two locations beyond the OP CODE fetch location are transferred to the new PC high order byte."

The addressing modes are Absolute and Absolute Indirect.

The JMP instruction affects no flags and only PCL and PCH.

The JMP instruction allows use of the program counter to access the new program counter value as illustrated by the following example:

Example 4.3: Use of JMP Instruction (Absolute Addressing Mode)

Address	Data	Comments
0100	JMP	Jump to Location 3625
0101	25	(New PCL byte)
0102	36	(New PCH byte)
3625	OP CODE	Next Instruction

The program counter in the example starts out at location 100. The microprocessor loads a jump instruction. The program counter automatically increments to 101 where the microprocessor picks up and temporarily stores the 25. The program counter automatically increments to 102 where the microprocessor picks up the 36.

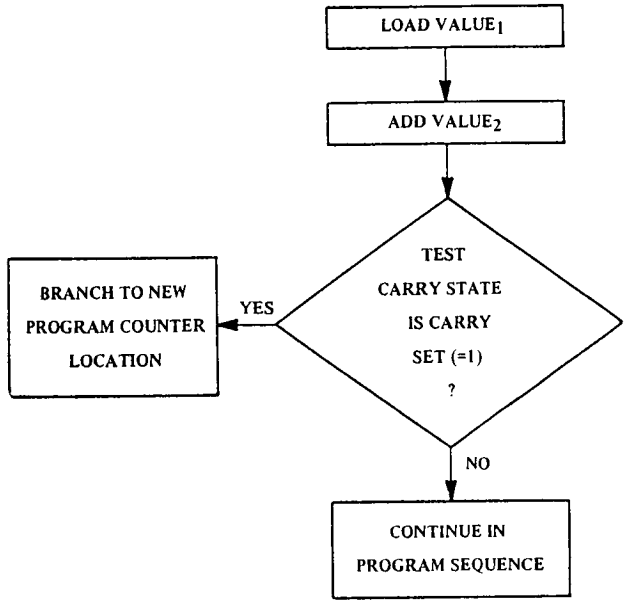
The 3625 is substituted into the program counter and is used to address the next instruction. Therefore, the JMP instruction contains within its address the new program counter location.

Although the jump allows the change of program sequence, it does so without performing any test. So it is a JMP instruction that is employed when it is desired to change the program counter no matter what conditions have occurred.

Another JMP addressing Mode is the Indirect Addressing Mode. Before this technique can be understood, the basis of indirect addressing found in Chapter 6 must be reviewed. The JMP Indirect instruction is detailed in Chapter 9.

4.1 BRANCHING

To allow for conditional program sequence change, eight conditional branch instructions are available for testing and performing optional changes of the program counter based on the status of the flags. To perform a conditional change of sequence, the microprocessor must interpret the instruction, test the value of a flag, and then change the P counter if the value agrees with the instruction. If the condition is not met, the program counter continues to increment in its normal fashion. Figure 4.2 illustrates how a conditional test might be used.



Use of Conditional Test
FIGURE 4.2