HEATHKIT
CONTINUING
EDUCATION

Individual Learning Program

MICROPROCESSORS

UNIT 3

COMPUTER ARITHMETIC

EE-3401

HEATH COMPANY
BENTON HARBOR, MICHIGAN 49022

CONTENTS

Unit 3

COMPUTER ARITHMETIC

INTRODUCTION

In this Unit you will complete your study of the binary number system. Since microprocessors use binary numbers for data and control, it is important that you become familiar with them.

Computer arithmetic involves many forms of number manipulation. In the pages that follow you will be given the fundamentals of binary mathematics: addition, subtraction, multiplication, and division. Then you will learn to perform two's complement arithmetic using binary numbers. Finally, you will be shown how the microprocessor performs the four basic Boolean logic operations. These logical operations include AND, OR, exclusive OR, and invert.

UNIT OBJECTIVES

When you complete this Unit you will be able to:

1. Add two binary numbers.
2. Subtract one binary number from another.
3. Multiply one binary number by another.
4. Divide one binary number by another.
5. Derive the one's complement of a binary number.
6. Derive the two's complement of a binary number.
7. Add binary numbers using two's complement arithmetic.
8. Manipulate binary numbers using the AND operation.
9. Manipulate binary numbers using the OR operation.
10. Manipulate binary numbers using the exclusive OR operation.
11. Logically invert binary numbers.

UNIT ACTIVITY GUIDE

	Completion Time
☐ Read section on Binary Arithmetic.	________
☐ Answer Self-Test Review Questions 1-11.	________
☐ Read section on Two's Complement Arithmetic.	________
☐ Answer Self-Test Review Questions 12-21.	________
☐ Read section on Boolean Operations.	________
☐ Answer Self-Test Review Questions 22-30.	________
☐ Perform Experiment 4.	________
☐ Complete Unit Examination.	________
☐ Review Examination Answers.	________

BINARY ARITHMETIC

A number system can be used to perform two basic operations: addition and subtraction. But by using addition and subtraction, you can then perform multiplication, division, and any other numerical operation. In this section, binary arithmetic (addition, subtraction, multiplication, and division) will be examined, using decimal arithmetic as a guide.

Binary Addition

Binary addition is performed somewhat like decimal addition. If two decimal numbers, 56719 and 31863, are added together, the sum 88582 is obtained. You could analyze the details of this operation in the following manner.

NOTE: In the following explanations, the term "first column" refers to the first column of figures you work with in the problem — the column on the right (9, 3, and 2 in the following example). The term "second column" refers to the second column you work with, etc.

```
Carry:      00101
Addend:      56719
Augend:    + 31863
            ------
Sum:         88582
```

Adding the first column, decimal numbers 9 and 3, gives the sum of 12. This is expressed in the sum as the digit 2 with a carry of 1. The carry is then added to the next column. Adding the second column decimal numbers 1 and 6, and the carry from the first column, gives the sum of 8, with no carry. This process continues until all of the columns (including carries) have been added. The sum represents the numeric value of the addend and augend. (The **addend** is the number to be added to another number, while the **augend** is the number to which the addend is added.)

When you add two binary numbers, you perform the same operation. Figure 3-1 summarizes the four rules of addition with binary numbers.

1. 0 + 0 = 0
2. 0 + 1 = 1
3. 1 + 1 = 0 with a carry of 1.
4. 1 + 1 + 1 = 1 with a carry of 1.

Figure 3-1
Rules for binary addition.

To illustrate the process of binary addition, let's add 1101 to 1101.

```
Carry:     1101
Addend:     1101
Augend:   + 1101
Sum:       11010
```

In the first column, 1 plus 1 equals 0 with a carry of 1 to the second column. This agrees with rule 3. In the second column, 0 plus 0 equals 0 with no carry. To this sum, the carry from the first column is added. Thus, 0 plus 1 equals 1 with no carry. These two additions in the second column give a total sum of 1 with a carry of 0. Rules 1 and 2 were used to obtain the sum.

In column three, 1 plus 1 equals 0 with a carry of 1. To this sum, the second column carry is added. This yields a third column sum of 0 with a carry of 1 to column four. Rules 3 and 1 were used to obtain the sum.

In column four, 1 plus 1 equals 0 with a carry of 1. To this sum, the third column carry is added. This yields a fourth column sum of 1 with a carry to the fifth column. Rule 4 allows you to add three binary 1's and obtain 1 with a carry of 1.

In column five, there is no addend or augend. Therefore, you can assume rule 2 and add the carry to 0 to obtain the sum of 1. Thus, the sum of 1101_2 plus 1101_2 equals 11010_2. You can verify this by converting the binary numbers to decimal numbers.

Now study the following two examples of binary addition, where 10001111_2 is added to 10110101_2 and 111011_2 is added to 11001100_2.

```
Carry:     10111111
Addend:     10110101
Augend:   + 10001111
Sum:       101000100

Carry:     11111000
Addend:     11001100
Augend:   + 00111011
Sum:       100000111
```

When binary addition is performed with a microprocessor, 8-bit numbers are generally used. As shown in the last example, two zeros were added after the MSB of the augend to produce an 8-bit number. After addition, a 1 in the ninth bit is represented as the "carry" bit by the microprocessor. This will be explained in a later unit.

Binary Subtraction

Binary subtraction is performed exactly like decimal subtraction. Therefore, before binary subtraction can be attempted, decimal subtraction should be reexamined. You know that if decimal 5486 is subtracted from 8303, the difference 2817 is obtained.

```
Minuend after borrow:     7 12 9 13
Minuend:                  8  3 0  3
Subtrahend:              -5  4 8  6
Difference:               2  8 1  7
```

Because the digit 6 in the subtrahend is larger than the digit 3 in the minuend, a 1 is borrowed from the next higher-order digit in the minuend. If that digit is 0, as in this example, 1 is borrowed from the next higher-order digit that contains a number other than 0. That digit is reduced by 1 (from 3 to 2 in this example) and the digits skipped in the minuend are given the value 9. This is equivalent to removing 1 from 30 with the result of 29, as in this example. In the decimal system, the digit borrowed has the value of ten. Therefore, the minuend digit now has the value 13, and 6 from 13 equals 7.

In the second column, 8 from 9 equals 1. Since the subtrahend is larger than the minuend in the third column, 1 is borrowed from the next higher-order digit. This raises the minuend value from 2 to 12, and 4 from 12 equals 8. In the fourth column, the minuend was reduced from 8 to 7 because of the previous borrow, and 5 from 7 equals 2.

Whenever 1 is borrowed from a higher-order digit, the borrow is equal in value to the radix or base of the number system. Therefore, a borrow in the decimal number system equals ten, while a borrow in the binary number system equals two.

When you subtract one binary number from another, you use the same method described for decimal subtraction. Figure 3-2 summarizes the four rules for binary subtraction.

1. $0 - 0 = 0$
2. $1 - 1 = 0$
3. $1 - 0 = 1$
4. $0 - 1 = 1$ with a borrow of 1.

Figure 3-2
Rules for binary subtraction.

To illustrate the process of binary subtraction, let's subtract 1101 from 11011.

```
Minuend after borrow:     0 10 10 1 1
Minuend:                  1  1  0 1 1
Subtrahend:           -      1  1 0 1
                      ---------------
Difference:                  1  1 1 0
```

The "minuend after borrow" now shows the value of each minuend digit after a borrow occurs. Remember that binary 10 equals decimal 2.

In the first column, 1 from 1 equals 0 (rule 2). Then, 0 from 1 in the second column equals 1 (rule 3). In the third column, 1 from 0 requires a borrow from the fourth column. Thus, 1 from 10_2 equals 1 (rule 4). The minuend in the fourth column is now 0, from the previous borrow. Therefore, a borrow is required from the fifth column, so that 1 from 10_2 in the fourth column equals 1 (rule 4). Because of the previous borrow, the minuend in the fifth column is now 0 and the subtrahend is 0 (nonexistant), so that 0 from 0 equals 0 (rule 1). The 0 in the fifth column is not shown in the difference because it is not a significant bit. Thus, the difference between 11011_2 and 1101_2 is 1110_2. You can verify this by converting the binary numbers to decimal numbers.

As a further example of binary subtraction, subtract 00100101_2 from 11000100_2, as shown below. Then proceed to the next example and subtract 10111010_2 from 11101110_2.

Minuend after borrow:	1 0 1 1 1 10 1 10
Minuend:	1 1 0 0 0 1 0 0
Subtrahend:	−0 0 1 0 0 1 0 1
Difference:	1 0 0 1 1 1 1 1

Minuend after borrow:	0 0 10 10 1 1 1 0
Minuend:	1 1 1 0 1 1 1 0
Subtrahend:	−1 0 1 1 1 0 1 0
Difference:	0 0 1 1 0 1 0 0

When a borrow is required in the minuend, 1 is obtained from the next high-order bit that contains a 1. That bit then becomes 0, and all bits skipped (0 value bits) are given the value 1. This is equivalent to removing 1 from 1000_2 with the result of 0111_2.

As with binary addition, microprocessors generally perform subtraction on 8-bit number groups. In the previous example, the answer contained only six significant bits, but two 0 bits were added to maintain the 8-bit grouping. This would also be true for the minuend and subtrahend.

Subtraction of a large number from a smaller number will be described in a later section of this Unit.

Binary Multiplication

Multiplication is a short method of adding a number to itself as many times as it is specified by the multiplier. However, if you were to multiply 324_{10} by 223_{10}, you would probably use the following method.

```
Multiplicand:                324
Multiplier:                × 223
First partial product:       972
Second partial product:     648
Third partial product:     648
Carry:                    0121
Final product:             72252
```

Using this short form of multiplication, you multiply the multiplicand by each digit of the multiplier and then sum the partial products to obtain the final product. Note that, for convenience, the additive carries are set-down under the partial products rather than over them as in normal addition.

Binary multiplication follows the same general principles as decimal multiplication. However, with only two possible multiplier bits (1 or 0), binary multiplication is a much simpler process. Figure 3-3 lists the rules of binary multiplication. These rules are used to multiply 1111_2 by 1101_2 on the next page.

1. $0 \times 0 = 0$
2. $0 \times 1 = 0$
3. $1 \times 0 = 0$
4. $1 \times 1 = 1$

Figure 3-3
Rules for binary multiplication.

Multiplicand:	1111
Multiplier:	×1101
First partial product:	1111
Second partial product:	0000
Carry:	0000
Sum of partial products:	1111
Third partial product:	1111
Carry:	111100
Sum of partial products:	1001011
Fourth partial product:	1111
Carry:	1111000
Final product:	11000011

As with decimal multiplication, you multiply the multiplicand by each bit in the multiplier and add the partial sums. First you multiply 1111_2 by the least significant multiplier bit (1) and set down the partial product so the least significant bit (LSB) is under the multiplier bit. Then you multiply the multiplicand by the next multiplier bit (0) and set down the partial product so the LSB is under the multiplier bit. Now that there are two partial products, they should be added. Although it is possible to add more than two binary numbers, keeping track of the multiple carries may become confusing. Therefore, for these examples, add only two partial products at a time.

Notice that the first partial product is identical to the multiplicand. The second partial product is all zeros. Since the binary number system contains only ones and zeros, the partial product will always equal either the multiplicand or zero. Because of this, you can obtain the third partial product by copying the multiplicand. Begin with the LSB under the third multiplier bit. Add this value to the previous partial sum. Now obtain the fourth partial product by copying the multiplicand. Begin with the LSB under the fourth multiplier bit. Add this value to the previous partial sum. This is the final product. You can verify the result by converting the binary numbers to decimal.

Reexamine the illustration for the previous multiplication example. Notice that binary multiplication is a process of shift and add. For each 1 bit in the multiplier you copy down the multiplicand, beginning with the LSB under the bit. You can ignore any zeros in the multiplier. But do not make the mistake of setting down the multiplicand under the 0 bit.

To make sure you fully understand binary multiplication, multiply 1001_2 by 1100_2 and then multiply 1101_2 by 1111_2.

Multiplicand:	1001
Multiplier:	×1100
First partial product:	0000
Second partial product:	0000
Carry:	0000
Sum of partial products:	00000
Third partial product:	1001
Carry:	00000
Sum of partial products:	100100
Fourth partial product:	1001
Carry:	000000
Final product:	1101100

Multiplicand:	1101
Multiplier:	×1111
First partial product:	1101
Second partial product:	1101
Carry:	11000
Sum of partial products:	100111
Third partial product:	1101
Carry:	100100
Sum of partial products:	1011011
Fourth partial product:	1101
Carry:	1111000
Final product:	11000011

In the first of these last two examples, the two zeros in the multiplier were included in the multiplication process. This was to insure that the multiplicand was copied down under the proper multiplier bits. The multiplication process could have been represented in this manner:

Multiplicand:	1001
Multiplier:	×1100
Third partial product:	100100
Fourth partial product:	1001
Carry:	000000
Final product:	1101100

Remember, just as in decimal multiplication, you must keep track of any zeros by setting a zero in the product under the 0 bit in the multiplier. This is very important when the zero occupies the LSB.

Binary Division

Division is the reverse of multiplication. Therefore, it is a procedure for determining how many times one number can be subtracted from another. The process you are probably familiar with is called "long" division. If you were to divide decimal 181 by 45, you would obtain the quotient, 4-1/45, as follows:

```
                    004        Quotient
Divisor      45 ) 181          Dividend
                  180
                    1          Remainder
```

Using long division, you would examine the most significant digit in the dividend and determine if the divisor was smaller in value. In this example the divisor is larger, so the quotient is zero. Next, you examine the two most significant digits. Again the divisor is larger, so the quotient is again zero. Finally, you examine the whole dividend and discover it is approximately four times the divisor in value. Therefore, you give the quotient a value of 4. Next, you subtract the product of 45 and 4 (180) from the dividend. The difference of one represents a fraction of the divisor. This fraction is added to the quotient to produce the correct answer of 4-1/45.

Binary division is performed in a similar manner. However, binary division is a simpler process since the number base is two rather than ten. First, let's divide 100011_2 by 101_2.

```
                    000111     Quotient
Divisor:     101 ) 100011      Dividend
                     101
                      111      Remainder
                      101
                       101     Remainder
                       101
                         0     Remainder
```

Using long division, you examine the dividend beginning with the MSB and determine the number of bits required to exceed the value of the divisor. When you find this value, place a one in the quotient and subtract the divisor from the selected dividend value. Then carry the next least significant bit in the dividend down to the remainder. If you can subtract the divisor from the new remainder, place a one in the quotient. Then subtract the divisor from the remainder and carry the next least significant bit in the dividend (LSB in this example) down to the remainder. If the divisor can be subtracted from the new remainder, place a one in the quotient and subtract the divisor from the remainder. Continue the process until all of the dividend bits have been carried down. Then express any remainder as a fraction of the divisor in the quotient. Thus, 100011_2 divided by 101_2 equals 111_2. You can verify the answer by converting the binary numbers to decimal.

To make sure you fully understand binary division, work out the following examples of long division. Divide 101000_2 by 1000_2 and then divide 100111_2 by 110_2.

```
                     000101     Quotient
Divisor       1000 ) 101000     Dividend
                     1000↓↓
                       1000     Remainder
                       1000
                          0     Remainder

                     000110.1   Quotient
Divisor:       110 ) 100111.0   Dividend
                      110↓ |
                       111 |    Remainder
                       110 ↓ ↓
                         11 0   Remainder
                         11 0
                            0   Remainder
```

In the second example, the quotient was not a whole number, but rather a whole number plus a fraction (remainder divided by the divisor). The answer 110-11/110 is correct. You could have left the answer in this form or, as in the example, continue the division process until the remainder was zero. This is made possible by adding a sufficient number of zeros after the binary point to permit division by the divisor. In the previous example, only one zero was added after the binary point. As you learned in Unit 1, adding zeros after the binary point will not affect the value of the number. Note that some numbers cannot be solved in this manner (e.g., decimal 1/3).

Representing Negative Numbers

Until now, we have been examining binary arithmetic using unsigned numbers. However, when you perform some arithmetic operations with a microprocessor, you must be able to express both positive and negative (signed) numbers. Over the years three methods have been developed for representing signed numbers. Of these, only one method has survived. The two older methods will be examined first, and then the system that is used today.

SIGN AND MAGNITUDE. Using this system, a binary number contained both the sign (+ or −) and the value of the number. Therefore, positive and negative values were expressed as follows:

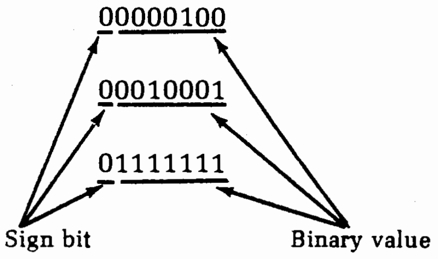
$$\begin{tikzcd} +45_{10} & = & \underline{0} & \underline{0101101}_2 \\ & & \text{SIGN} \arrow[u] \arrow[d] & \text{MAGNITUDE} \arrow[u] \arrow[d] \\ -45_{10} & = & \overline{1} & \overline{0101101}_2 \end{tikzcd}$$

The MSB of the binary number indicated the sign, while the remaining bits contained the value of the number. As you can see, a zero sign bit indicated a positive value, while a one sign bit indicated a negative value.

While this method of representing negative numbers may seem logical, its popularity was short-lived. Because it required complex and slow arithmetic circuitry, it was abandoned long before microprocessors were invented.

ONE'S COMPLEMENT. Another method of representing negative numbers became popular in the early days of computers. It was called the one's complement method. Using this system, positive numbers were represented in the same way as in the sign-magnitude system. That is, the MSB in any number was considered to be a sign bit. A sign bit of 0 represented positive. Using 8-bit numbers, positive values were represented like this:

$$+4_{10} = \underline{0}\,\underline{0000100}$$
$$+17_{10} = \underline{0}\,\underline{0010001}$$
$$+127_{10} = \underline{0}\,\underline{1111111}$$



Negative numbers were represented by the **one's complement** of the positive value. The one's complement of a number is formed by changing all 0's to 1's and all 1's to 0's. As shown above, $+4_{10}$ is represented as $\underline{0}\,\underline{0000100}_2$. By changing all 0's to 1's and all 1's to 0's, the representation for -4_{10} was formed. In this case:

$$-4_{10} = \underline{1}\,\underline{1111011}_2$$

Notice that all the bits, including the sign bit, were inverted. In the same way:

$$-17_{10} = \underline{1}\,\underline{1101110}_2$$

$$-127_{10} = \underline{1}\,\underline{0000000}_2$$

The one's complement method is not used for representing signed numbers in microprocessors. However, as you will see later, you may still be called upon to find the one's complement of a number. Remember, you do this by simply changing all 0's to 1's and all 1's to 0's.

Figure 3-4 shows an interesting relationship. In the first column, 8-bit patterns of 0's and 1's are shown. The second column shows the decimal number that each pattern represents if you consider the pattern to be an unsigned binary number. Notice that an 8-bit pattern can represent unsigned numbers between 0 and 255_{10}.

The third column shows the decimal number that each pattern represents if you consider the pattern to be a one's complement binary number. Notice that the range of numbers is from -127_{10} to $+127_{10}$. Notice also that there are two representations of zero. The pattern $0000\ 0000_2$ represents +0 while its one's complement ($1111\ 1111_2$) represents −0.

TWO'S COMPLEMENT. The method used to represent signed numbers in microprocessors is called two's complement. In this system, positive numbers are represented just as they were with the sign-and-magnitude method and the one's complement method. That is, it uses the same bit pattern for all positive values up to $+127_{10}$. However, negative numbers are represented as the two's complement of positive numbers.

The two's complement of a number is formed by taking the one's complement and then adding 1. For example if you work with 8-bit numbers and use the two's complement system, $+4_{10}$ is represented by 00000100_2. To find -4_{10}, you must take the two's complement of this number. You do this by first taking the one's complement, which is 11111011_2. Next, add 1 to form the two's complement:

$$\begin{array}{r} 11111011_2 \\ +\quad\quad\quad 1 \\ \hline 11111100_2 \end{array}$$

Thus, the two's complement representation of -4_{10} is 11111100_2.

To be sure you have the idea, look at a second example: how do you express -17_{10} as an 8-bit two's complement number? Start with the two's complement representation of $+17_{10}$, which is 00010001_2. Take the one's complement by changing all 0's to 1's and 1's to 0's. Thus, the one's complement of $+17_{10}$ is 11101110_2. Next, find the two's complement by adding 1:

$$\begin{array}{r} 11101110_2 \\ +\quad\quad\quad 1 \\ \hline 11101111_2 \end{array}$$

BIT PATTERN	UNSIGNED BINARY	1's COMPLEMENT
00000000	0	+0
00000001	1	+1
00000010	2	+2
00000011	3	+3
•	•	•
•	•	•
•	•	•
•	•	•
01111100	124	+124
01111101	125	+125
01111110	126	+126
01111111	127	+127
10000000	128	−127
10000001	129	−126
10000010	130	−125
10000011	131	−124
•	•	•
•	•	•
•	•	•
•	•	•
11111100	252	−3
11111101	253	−2
11111110	254	−1
11111111	255	−0

Figure 3-4
Table of bit pattern values for unsigned binary numbers and 1's complement numbers.

Figure 3-5 compares unsigned, two's complement, and one's complement numbers. Several 8-bit patterns are shown on the left. The other three columns show the decimal number represented by these patterns.

Notice that the range of 8-bit two's complement numbers is from -128_{10} to $+127_{10}$. Notice also that there is only one representation for 0.

If this table included all 256_{10} possible 8-bit patterns, you could look up any pattern to see what number it represents. The patterns which have 0 as their MSB are easy to determine without a table. The pattern represents the binary number directly. But what decimal number is represented by the two's complement number 11110011? You should know that this represents some negative number because the MSB is a 1.

Actually, you can determine the value very easily by simply taking the two's complement to find the equivalent positive number. Remember, you find the two's complement, by taking the one's complement and adding 1. The one's complement is 00001100_2. Thus, the two's complement is:

$$\begin{array}{rl} & 00001100_2 \\ + & 1 \\ \hline & 00001101_2 \quad \text{or } +13_{10} \end{array}$$

Since the two's complement of 11110011_2 represents $+13_{10}$, then 11110011_2 must equal -13_{10}.

BIT PATTERN	UNSIGNED BINARY	2's COMPLEMENT	1's COMPLEMENT
00000000	0	0	+0
00000001	1	+1	+1
00000010	2	+2	+2
00000011	3	+3	+3
•	•	•	•
•	•	•	•
•	•	•	•
•	•	•	•
01111100	124	+124	+124
01111101	125	+125	+125
01111110	126	+126	+126
01111111	127	+127	+127
10000000	128	−128	−127
10000001	129	−127	−126
10000010	130	−126	−125
10000011	131	−125	−124
•	•	•	•
•	•	•	•
•	•	•	•
•	•	•	•
11111100	252	−4	−3
11111101	253	−3	−2
11111110	254	−2	−1
11111111	255	−1	−0

Figure 3-5
Table of bit pattern values for unsigned binary, 2's complement and 1's complement numbers.

Self-Test Review

1. ______________ and ______________ are the two basic operations that can be performed with a number system.

2. Add the following binary numbers.

A. $\begin{array}{r} 10011011 \\ \underline{+00010111} \end{array}$ B. $\begin{array}{r} 11000110 \\ \underline{+00110001} \end{array}$ C. $\begin{array}{r} 10000110 \\ \underline{+00110110} \end{array}$

3. Subtract the following binary numbers.

A. $\begin{array}{r} 11011011 \\ \underline{-10110010} \end{array}$ B. $\begin{array}{r} 10001011 \\ \underline{-10000001} \end{array}$ C. $\begin{array}{r} 11011001 \\ \underline{-00111011} \end{array}$

4. Multiply the following binary numbers.

A. $\begin{array}{r} 1011 \\ \underline{\times 1101} \end{array}$ B. $\begin{array}{r} 1101 \\ \underline{\times 1001} \end{array}$ C. $\begin{array}{r} 1100 \\ \underline{\times 1100} \end{array}$

5. Solve for the quotient in the following groups.

A. $101\,\overline{)1001011}$ B. $11\,\overline{)111001}$ C. $1101\,\overline{)11110111}$

6. 10001111_2 represents decimal__________ in sign/magnitude notation.

7. The 1's complement of 00010110_2 is ______________.

8. The 2's complement of 00010110_2 is ______________.

9. The 2's complement number 11100110 represents the decimal number ______________.

10. Find the signed decimal equivalents of the following two's complement numbers.

Two's Complement Number	Decimal Number
00000111	
10000111	
11111111	
01110000	
10000000	

11. Find the two's complement representation for the following signed decimal numbers.

Decimal Number	Two's Complement Number
+32	
−32	
+73	
− 7	
−120	

Answers

1. Addition, subtraction.

2. A.

Carry:	00011111
Addend:	10011011
Augend:	+ 00010111
Sum:	10110010

B. 11110111.

C. 10111100.

3. A.

Minuend after borrow:	1 0 10 1 1 0 1 1
Minuend:	1 1 0 1 1 0 1 1
Subtrahend:	− 1 0 1 1 0 0 1 0
Difference:	1 0 1 0 0 1

B. 1010.

C. 10011110.

4. A.

Multiplicand:	1011
Multiplier:	× 1101
First partial product:	1011
Second partial product:	00000
Carry:	0000
Sum of partial products:	01011
Third partial product:	101100
Carry:	01000
Sum of partial products:	110111
Fourth partial product:	1011000
Carry:	1110000
Final product:	10001111

B. 1110101.

C. 10010000

5. A. Divisor:

```
            0001111     Quotient
     101 ) 1001011      Dividend
            101
            1000        Remainder
             101
              111       Remainder
              101
               101      Remainder
               101
                 0      Remainder
```

B. 10011.

C. 10011.

6. −15.

7. 11101001_2.

8. 11101010_2.

9. First, find the two's complement of 11100110 by changing 1's to 0's; 0's to 1's; and adding 1:

```
00011001
       1
--------
00011010
```

Since this number represents $+26_{10}$, the original number must have represented -26_{10}.

10.

Two's Complement Number	Decimal Number
00000111	+7
10000111	−121
11111111	−1
01110000	+112
10000000	−128

11.

Decimal Number	Two's Complement Number
+32	00100000
−32	11100000
+73	01001001
−7	11111001
−120	10001000

TWO'S COMPLEMENT ARITHMETIC

In the previous section, you saw that signed numbers are represented in microprocessors in two's complement form. In this section you will see why.

In digital electronic devices such as computers, simple circuits cost less and operate faster than more complex ones. Two's complement numbers are used with arithmetic because they allow the simplest, cheapest, and fastest circuits.

A characteristic of the two's complement system is that both signed and unsigned numbers can be added by the same circuit. For example, suppose you wish to add the **unsigned** numbers 132_{10} and 14_{10}. The addition looks like this:

Addend:	10000100_2	132_{10}
Augend:	00001110_2	$+\ 14_{10}$
Sum:	10010010_2	146_{10}

As you saw in the previous unit, the microprocessor has an ALU circuit that can add unsigned binary numbers in this way. The adder in the ALU is designed so that when the bit pattern 10000100 appears at one input and 00001110 appears at the other, the bit pattern 10010010 appears at the output.

The question arises, "How does the ALU know that the bit patterns at the inputs represent unsigned numbers and not two's complement numbers?" The answer is "it doesn't." The ALU always adds as if the inputs were unsigned binary numbers. Nevertheless, it still produces the correct sum even if the inputs are signed two's complement numbers.

Look at the example given above. If you assume that the inputs are two's complement signed numbers, then the addend, augend, and sum are:

Addend:	10000100_2	-124_{10}
Augend:	00001110_2	$+\ 14_{10}$
Sum:	10010010_2	-110_{10}

Notice that the bit patterns are the same. Only the meaning of the bit patterns has changed. In the first example, we assumed that the bit patterns represented unsigned numbers and the adder produced the proper unsigned result. In the second example, we assumed that the bit patterns represented signed numbers. Again, the adder produced the proper signed result.

This proves a very important point. The adder in the ALU always adds bit patterns as if they are unsigned binary numbers. It is our interpretation of these bit patterns that decides if unsigned or signed numbers are indicated. The beauty of two's complement is that the bit patterns can be interpreted either way. This allows us to work with either signed or unsigned numbers without requiring different circuits for each.

Two's complement arithmetic also simplifies the arithmetic logic unit in another way. All microprocessors have a subtract instruction. Thus, the ALU must be able to subtract one number from another. However, if this required a separate subtraction circuit, the complexity and cost of the ALU would be increased. Fortunately, two's complement arithmetic allows the ALU to perform a subtract operation using an adder circuit. That is, the MPU uses the same circuit for both addition and subtraction.

The MPU performs subtraction by a binary addition process. To see why this works, it may be helpful to look at a similar process with the decimal number system. The decimal equivalent of two's complement is called ten's complement. Since you are more familiar with the decimal number system, briefly examine ten's complement arithmetic.

Ten's Complement Arithmetic

An easy way to illustrate ten's complement is to consider an analogy. Visualize the odometer or mileage indicator on your car. Generally, this is a six-digit device that indicates mileage between 00,000.0 and 99,999.9 miles. Let's ignore the tenths digit and concentrate on the other five.

In an automobile, the register generally operates in only one direction (forward). However, consider what happens if it is turned backwards instead. Starting at +3 miles, the count proceeds backwards as follows:

```
00,003
00,002
00,001
00,000
99,999
99,998
99,997
 etc.
```

It is easy to visualize that 99,999 represents −1 mile. Also, 99,998 represents −2 miles; 99,997 represents −3 miles; etc. This is how signed numbers are represented in ten's complement form.

Once you accept this system for representing positive and negative numbers, you can perform arithmetic with these signed numbers. For example, if you add +3 and −2, the result should be +1. Using the system developed above, +3 is represented by 00003 while −2 is represented by 99,998. Thus, the addition looks like this:

$$\begin{array}{r@{\qquad}r} 00003 & +3 \\ +99998 & -2 \\ \hline 100001 & +1 \end{array}$$

Discard final carry.

If you now discard the final carry on the left in the sum, the answer is 000 01, the representation of +1. You can also find the ten's complement of a digit by subtracting the digit from ten. For example, the ten's complement of 6 is 4 since 10−6 = 4. To complement a number containing more than one digit, raise ten to a power equal to the total number of digits, then subtract the number from it. For example, to obtain the ten's complement of 654_{10}, first raise ten to the third power since there are three digits in the number. Then, subtract 654 from the result.

$$\begin{array}{rr} 10^3 = & 1000 \\ & -654 \\ \hline & 346 \end{array}$$

Thus, the ten's complement of 654_{10} is 346_{10}.

Once you find the ten's complement, you can subtract one number from another by an indirect method using only addition. Since childhood you have subtracted like this:

$$\begin{array}{l@{\qquad}r} \text{Minuend:} & 973 \\ \text{Subtrahend:} & -654 \\ \cline{2-2} \text{Difference:} & 319 \end{array}$$

However, you can arrive at the same answer by using the ten's complement of the subtrahend and adding. Recall that the ten's complement of 654_{10} is 346_{10}. Let's compare these two methods of subtraction:

STANDARD METHOD		TEN'S COMPLEMENT METHOD	
Minuend	973	973	Minuend
Subtrahend	−654	+346	Ten's complement of subtrahend
Difference	319	1319	Difference

Discard final carry

Notice that when you use the ten's complement method, the answer is too large by 1000_{10}. However, you can still arrive at the correct answer by simply discarding the final carry.

While the ten's complement method of subtraction works, it is not used because it is more complex than the standard method. In fact, it does not eliminate subtraction entirely since the ten's complement itself is found by subtraction.

The binary equivalent of ten's complement is two's complement. It overcomes the disadvantage of ten's complement in that the two's complement can be formed without any subtraction at all. Recall that you can form the two's complement of a binary number by changing all 0's to 1's, all 1's to 0's and then adding 1. Let's examine two's complement arithmetic in more detail.

Two's Complement Subtraction

As in ten's complement arithmetic, you can form the two's complement by subtracting from a power of the base (two). However, because the MPU cannot subtract directly, it uses the method given earlier for finding the two's complement. Once the two's complement is formed, the MPU can perform subtraction indirectly by adding the two's complement of the subtrahend to the minuend.

To illustrate this point, look at the following two ways of subtracting 26_{10} from 69_{10}. The two numbers are expressed as they would appear to an 8-bit microprocessor. The standard method of subtraction looks like this:

Minuend:	01000101_2	69
Subtrahend:	-00011010_2	−26
Difference:	00101011_2	43

While this method works fine on paper, it's of little use to the microprocessor since the MPU has no subtract circuitry. However, the MPU can still perform subtraction by the indirect method of adding the two's complement of the subtrahend to the minuend:

```
                       Minuend:        01000101
Two's complement of    Subtrahend:    +11100110
                       Difference:    100101011
                                      ↑
                                      └─ Discard final carry
```

This illustrates a major reason for using the two's complement system to represent signed numbers. It allows the MPU to perform subtraction and addition with the same circuit.

The method that the MPU uses to perform subtraction is of little importance to the user of microprocessors. Most microprocessors have a subtract instruction. This instruction is used like any other without regard for how the operation is implemented internally. When the subtract instruction is implemented, the MPU automatically takes care of operations like complementing the subtrahend, adding, and discarding the carry. The procedure has been explained here so you can appreciate the importance of two's complement arithmetic.

Arithmetic With Signed Numbers

There are many applications in which the microprocessor must work with signed numbers. In these cases, signed numbers are represented in two's complement form. While this greatly simplifies the circuitry of the MPU, it places an extra burden on the user. The programmer must ensure that all signed numbers are entered into the microprocessor in two's complement form. Also, the resulting data produced by the MPU may be in two's complement form. Here's how an 8-bit MPU handles signed numbers.

Adding Positive Numbers. Assume that the MPU is to add the two positive numbers +7 and +3. Since an 8-bit MPU is assumed, the arithmetic operation looks like this:

```
 00000111    + 7
+00000011    + 3
 00001010    +10
```

The sign bits are underlined. Remember, when representing signed numbers, that the MSB is the sign bit. A 0 represents "+" and a 1 represents "−." In this example, you added +7 and +3 to form a sum of $+10_{10}$. You know that all three numbers are positive since the MSB's are all 0's.

While this operation seems straightforward enough, it is easy for the unwary to make an error when adding positive numbers. Remember, the highest 8-bit positive number you can represent in two's complement form is $+127_{10}$. If the sum exceeds this value, an error occurs. For example, suppose you attempt to add $+65_{10}$ to $+67_{10}$. The MPU adds the numbers as if they are unsigned binary:

$$\begin{array}{r} \underline{0}1000001 \\ \underline{0}1000011 \\ \hline \underline{1}0000100 \end{array}$$

If the answer is interpreted as a two's complement number, an error has occurred. You have added two positive numbers and yet the answer appears to be negative since the MSB of the sum is 1. This is called a two's complement overflow. It occurs when the sum exceeds $+127_{10}$. Many microprocessors have a way of detecting this condition. We will discuss this in more detail in a future unit.

Adding Positive and Negative Numbers. The real beauty of the two's complement system is illustrated when you add numbers with unlike signs. For example, assume that an 8-bit microprocessor is to add +7 and −3. Remember, since these are signed numbers, they must be represented in two's complement form. That is, +7 is represented as 00000111_2 while −3 is represented as 11111101_2. If these two numbers are added, the sum will be:

$$\begin{array}{lrr} \text{Addend:} & 00000111 & (+7) \\ \text{Augend:} & +11111101 & +(-3) \\ \hline \text{Sum:} & \underset{\uparrow\ \text{Discard final carry}}{1}00000100 & (+4) \end{array}$$

Notice that the sum is correct if you ignore the final carry bit. Keep in mind that the MPU adds the two numbers as if they were unsigned binary numbers. It is merely our interpretation of the answer that makes the system work for signed numbers.

The system also works when the negative number is larger. For example, when −9 is added to +8 the result should be −1. Remember, the signed numbers must be represented in two's complement form:

```
Addend:    11110111    (−9)
Augend:    00001000   +(+8)
           --------   -----
Sum:       11111111     −1
```

Notice that the sum is the two's complement representation for −1.

Adding Negative Numbers. The final case involves two negative numbers. If both numbers are negative, then the sum should also be negative.

For example, suppose the MPU is to add −3 to −4. Obviously, the result should be −7. The two signed numbers must be represented in two's complement form. That is, −3 must be represented as 11111101_2 while −4 must be represented as 11111100_2. The MPU adds these two bit patterns as if they were unsigned binary numbers. Thus the result is:

```
Addend:     11111101    (−3)
Augend:    +11111100   +(−4)
           ---------   -----
Sum:       111111001    (−7)
           ↑
           └─Discard final carry
```

Once again, the answer is correct if you ignore the final carry bit.

When you add two negative numbers, you must remember the capacity of the MPU. The largest negative number that can be represented by eight bits is -128_{10}. If the sum exceeds this value, the sum will appear to be in error. For example, suppose you add -120_{10} to -18_{10}.

```
                  10001000    (−120)
                  11101110   +(− 18)
                 ---------   -------
                 101110110
                ↗ ↑
Ignore carry      │
                  Sign bit
```

Notice that the sign bit in the sum is 0, representing a positive number. Thus, the MPU has added two negative numbers and has produced a positive result. This apparent error is caused by exceeding the 8-bit capacity. This is another example of two's complement overflow.

Self-Test Review

12. In microprocessors, signed numbers are represented in ________ ____________ form.

13. The ALU adds bit patterns as if they represent ____________ binary numbers.

14. When a microprocessor executes a subtract instruction, what operations are actually performed inside the MPU?

15. What is the largest 8-bit positive number that can be represented in two's complement form?

16. When you are adding two positive numbers, what is meant by two's complement overflow?

17. If $+19_{10}$ and -21_{10} are added by an 8-bit microprocessor, the two's complement result will be ____________.

18. Can two's complement overflow occur when two negative numbers are added?

19. A microprocessor adds 10001110_2 to 00010001_2. If these are unsigned binary numbers, the resulting bit pattern will be ____________. If these are two's complement numbers, the resulting bit pattern will be ____________. .

20. If the bit patterns in question 19 represent unsigned numbers, the resulting bit pattern represents decimal ____________.

21. If the bit patterns in question 19 represent two's complement numbers, the resulting bit pattern represents decimal ____________.

Answers

12. Two's complement.

13. Unsigned.

14. The following operations occur:

    1. The MPU complements the subtrahend by changing 0's to 1's and 1's to 0's.

    2. One is added to the complemented subtrahend to form the two's complement.

    3. The two's complement of the subtrahend is added to the minuend.

15. 01111111_2 or $+127_{10}$.

16. When you add positive numbers, two's complement overflow occurs when the sum exceeds $+127_{10}$.

17. 11111110_2 or -2_{10}.

18. Yes. When you add negative numbers, two's complement overflow occurs when the sum exceeds -128_{10}.

19. In either case, the resulting bit pattern will be 10011111.

20. 159_{10}.

21. -97_{10}.

BOOLEAN OPERATIONS

Along with the basic mathematical processes examined earlier, the microprocessor can manipulate binary numbers logically. This system was conceived using the theorems developed by the mathematician George Boole. As a result, this branch of binary mathematics is given the name Boolean Algebra. In this section, the Boolean operations performed by the microprocessor will be examined. A more detailed description of Boolean Algebra is provided in the Heathkit Continuing Education Series course titled "Digital Techniques."

AND Operation

The AND function produces the logical product of two or more logic variables. That is, the logical product of an AND operation is logic 1 if all of the variable inputs are logic 1. If any of the input variables are logic 0, the logical product is 0. This process can be represented by the formula $A \cdot B = C$, where A and B represent input variables (logic 1 or 0) and C represents the output or logical product of the AND operation. The AND function is designated by a dot between the variables. Do not confuse it with the mathematical multiplication sign.

Figure 3-6 is a "truth" table for a two-variable AND function. The 1's and 0's represent all of the possible logic combinations. Thus, you can see that the AND function is a sort of "all or nothing" operation. Unless all the input variables are logic 1, the output cannot be logic 1.

INPUTS		OUTPUT
A	B	C
0	0	0
0	1	0
1	0	0
1	1	1

Figure 3-6
Truth Table for an AND function.

When the microprocessor implements the logic AND operation, one 8-bit binary number is ANDed with a second 8-bit binary number. Refer to Figure 3-7 for an illustration of this process.

	8-BIT NUMBER		8-BIT NUMBER		RESULT OF AND OPERATION	
MSB	1	•	1	=	1	MSB
	0	•	0	=	0	
	0	•	1	=	0	
	1	•	0	=	0	
	1	•	1	=	1	
	0	•	1	=	0	
	1	•	0	=	0	
LSB	0	•	0	=	0	LSB

Figure 3-7
8-bit logic AND operation.

Although more than two logic variables can be ANDed together, the microprocessor operates on only two variables at a time. Now try one more example of the AND operation. AND 10011101 with 11000110.

```
1 • 1 = 1   MSB
0 • 1 = 0
0 • 0 = 0
1 • 0 = 0
1 • 0 = 0
1 • 1 = 1
0 • 1 = 0
1 • 0 = 0   LSB
```

OR Operation

The OR (sometimes known as inclusive OR) function produces the logical sum of two or more logic variables. That is, the logical sum of an OR operation is logic 1 if either input is logic 1. The logical sum is 0 if **all** of the input variables are logic 0. This process can be represented by the formula A + B = C, where A and B represent input variables and C represents the output or logical sum of the OR operation. The OR function is designated by a plus sign (or a circled dot ⊙) between the variables. Do not confuse the plus sign with the mathematical add sign.

Figure 3-8 is a "truth" table for a two-variable OR function. The 1's and 0's represent all of the possible logic combinations. Thus, you can see that the OR function is a sort of "either or both" operation. If either or both input variables are logic 1, the output must be logic 1.

INPUTS		OUTPUT
A	B	C
0	0	0
1	0	1
0	1	1
1	1	1

Figure 3-8
Truth Table for an OR function.

When the microprocessor implements the logic OR operation, one 8-bit binary number is ORed with a second 8-bit binary number. Refer to Figure 3-9 for an illustration of this process.

	8-BIT NUMBER		8-BIT NUMBER		RESULT OF OR OPERATION	
MSB	1	+	1	=	1	MSB
	0	+	0	=	0	
	0	+	1	=	1	
	1	+	0	=	1	
	1	+	1	=	1	
	0	+	1	=	1	
	1	+	0	=	1	
LSB	0	+	0	=	0	LSB

Figure 3-9
8-bit logic OR operation.

As with the AND function, two or more logic variables can be ORed together. However, the microprocessor operates on only two variables at a time. Now try one more example of the OR operation. OR 10011101 with 11000101.

$1 + 1 = 1$	MSB
$0 + 1 = 1$	
$0 + 0 = 0$	
$1 + 0 = 1$	
$1 + 0 = 1$	
$1 + 1 = 1$	
$0 + 0 = 0$	
$1 + 1 = 1$	LSB

Exclusive OR Operation

The Exclusive OR (EOR or XOR) function performs a logical test for "equalness" between two logic variables. That is, if two variable inputs are equal (both logic 1 or 0), the output or result of the EOR operation is logic 0. If the inputs are not equal (one is logic 1, the other logic 0) the output is logic 1. This can be represented by the formula $A \oplus B = C$, where A and B represent input variables and C represents the output or result. The EOR function is designated by a circled plus sign between the variables.

Figure 3-10 is a "truth" table for the EOR function. The 1's and 0's represent all of the possible logic combinations. You can see that the EOR function is a sort of "either but not both" operation. That is, either input can be logic 1 or 0, but not both for a logic 1 output.

INPUTS		OUTPUT
A	B	C
0	0	0
0	1	1
1	0	1
1	1	0

Figure 3-10
Truth Table for an EOR function.

When the microprocessor implements the logic EOR operation, one 8-bit binary number is exclusively ORed with a second 8-bit number. Refer to Figure 3-11 for an illustration of this process.

	8-BIT NUMBER		8-BIT NUMBER		RESULT OF EOR OPERATION	
MSB	1	$\oplus$	1	=	0	MSB
	0	$\oplus$	0	=	0	
	0	$\oplus$	1	=	1	
	1	$\oplus$	0	=	1	
	1	$\oplus$	1	=	0	
	0	$\oplus$	1	=	1	
	1	$\oplus$	0	=	1	
LSB	0	$\oplus$	0	=	0	LSB

Figure 3-11
8-bit logic EOR operation.

Now try one more example of the EOR operation. EOR 10011101_2 with 11000101_2.

1	$\oplus$	1	=	0	MSB
0	$\oplus$	1	=	1	
0	$\oplus$	0	=	0	
1	$\oplus$	0	=	1	
1	$\oplus$	0	=	1	
1	$\oplus$	1	=	0	
0	$\oplus$	0	=	0	
1	$\oplus$	1	=	0	LSB

Invert Operation

The invert operation performs a direct complement of a single input variable. That is, a logic 1 input will produce a logic 0 output. This process can be represented by the truth table in Figure 3-12.

INPUT	OUTPUT
A	$\overline{A}$
1	0
0	1

Figure 3-12
Truth Table for an invert function.

Note that the complement of A is $\overline{A}$. The bar above the A indicates that A has been inverted, and is read "not A." Therefore, the complement of A is "not A" ($\overline{A}$).

When the microprocessor implements the logic invert operation, the 8-bit binary number is complemented. This operation is also known as 1's complement. Thus, the complement of 11010110_2 is 00101001_2. As with the previous logic operations, the invert function operates on each individual bit of the 8-bit number.

Self-Test Review

22. The result of an AND operation is binary 1 when:

    A. All inputs are binary 0.

    B. Any one input is binary 0.

    C. All inputs are binary 1.

    D. Any one input is binary 1.

23. Perform the AND operation on the following 8-bit number pairs.

    A. 11010110 and 10000111.

    B. 00110011 and 11110000.

    C. 10101010 and 11011011.

24. The result of an OR operation is binary 0 when:

    A. All inputs are binary 1.

    B. All inputs are binary 0.

    C. Any one input is binary 1.

    D. Any one input is binary 0.

25. Perform the OR operation on the following 8-bit number pairs.

    A. 11010110 and 10000111.

    B. 00110011 and 11110000.

    C. 10101010 and 11011011.

26. The result of an XOR operation is binary 0 if the inputs are:

    A. Equal.

    B. Not equal.

27. The symbol for the EOR operation is:

    A. $\bullet$

    B. $+$

    C. $\oplus$

    D. $\times$

28. Perform the EOR operation on the following 8-bit number pairs.

    A. 11010110 and 10000111.

    B. 00110011 and 11110000.

    C. 10101010 and 11011011.

29. $\overline{\mathrm{A}}$ represents the ______________ of **A**.

    A. Sum.

    B. Product.

    C. Complement.

    D. Supplement.

30. Perform the invert operation on the following 8-bit numbers.

    A. 11010110.

    B. 00110011.

    C. 10101010.

Answers

22. C. All inputs are binary 1.

23. A.
$1 \cdot 1 = 1$
$1 \cdot 0 = 0$
$0 \cdot 0 = 0$
$1 \cdot 0 = 0$
$0 \cdot 0 = 0$
$1 \cdot 1 = 1$
$1 \cdot 1 = 1$
$0 \cdot 1 = 0$

B. 00110000.

C. 10001010.

24. B. All inputs are binary 0.

25. A.
$1 + 1 = 1$
$1 + 0 = 1$
$0 + 0 = 0$
$1 + 0 = 1$
$0 + 0 = 0$
$1 + 1 = 1$
$1 + 1 = 1$
$0 + 1 = 1$

B. 11110011.

C. 11111011.

26. A. Equal.

27. C. $\oplus$

28. A. $1 \oplus 1 = 0$
$1 \oplus 0 = 1$
$0 \oplus 0 = 0$
$1 \oplus 0 = 1$
$0 \oplus 0 = 0$
$1 \oplus 1 = 0$
$1 \oplus 1 = 0$
$0 \oplus 1 = 1$

B. 11000011.

C. 01110001.

29. C. Complement.

30. A. 00101001.

B. 11001100.

C. 01010101.

EXPERIMENT 4

Perform Experiment 4 in Unit 9 of this course. After you finish the experiment, return to this Unit and complete the Unit Examination.

UNIT EXAMINATION

1. Add 10010110_2 to 1101_2.

2. Subtract 1011_2 from 10110110_2.

3. Multiply 1001_2 by 1100_2.

4. Divide 100111_2 with 110_2.

5. The 1's complement of 00110110_2 is ______________.

6. The 2's complement of 00110110_2 is ______________.

7. Using 2's complement arithmetic, add $+75_{10}$ to -6_{10}.

8. Using 2's complement arithmetic, add -35_{10} to -75_{10}.

9. Using 2's complement arithmetic, subtract -15_{10} from -85_{10}.

10. The truth table Figure 3-13represents the logical ______________ function.

INPUT		OUTPUT
A	B	C
0	0	0
1	0	1
0	1	1
1	1	0

Figure 3-13
Truth Table for Exam Question 10.

11. Logically AND 11011010 with 10010110.

12. Logically OR 11011010 with 10010110.

13. Logically EOR 11011010 with 10010110.

14. Logically invert 11011010.