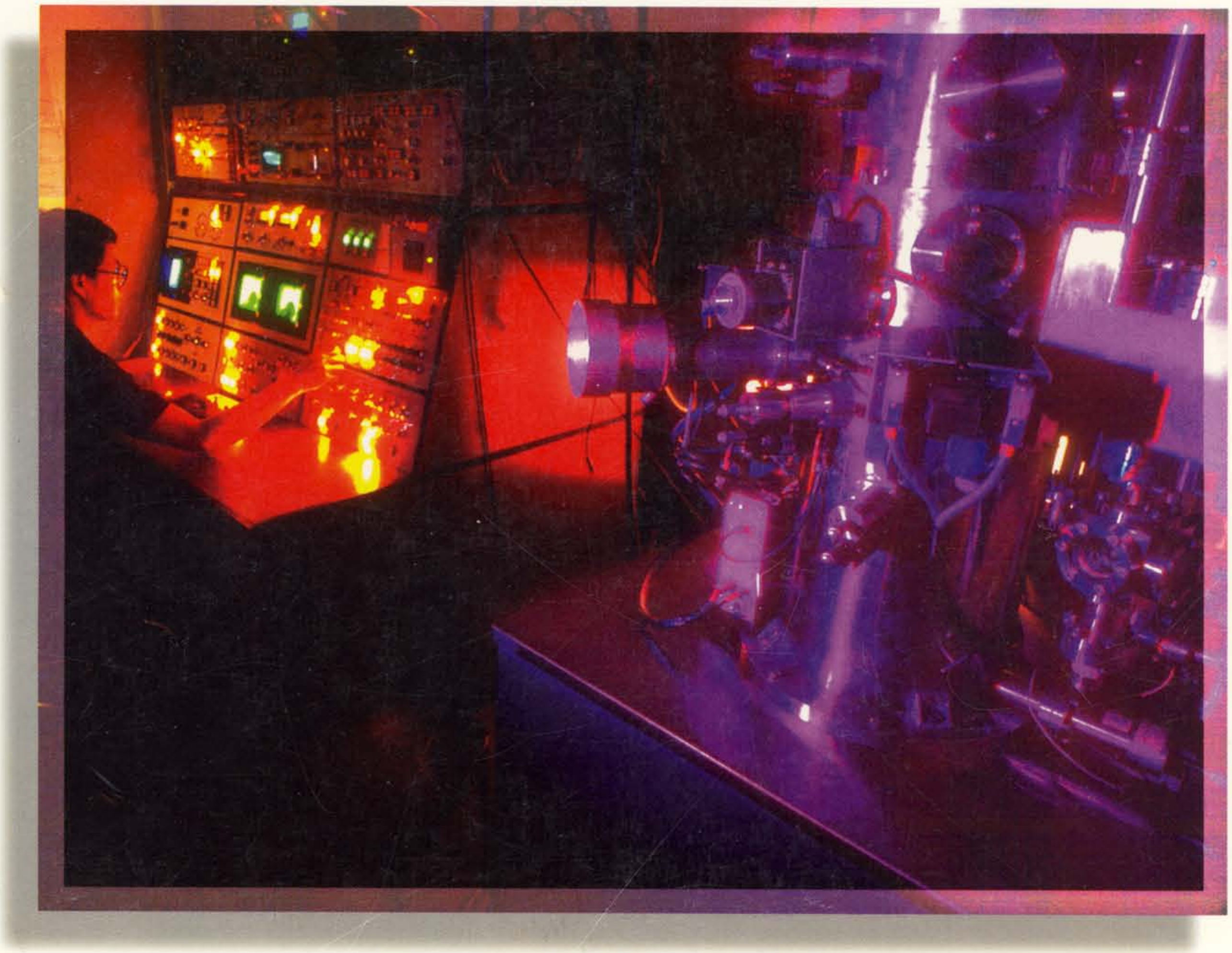
M I C R O P R O C E S S O R S

T E X T B O O K



MICROPROCESSORS

BOOK 1/BOOK 2

Heathkit®
EDUCATIONAL SYSTEMS

Microprocessors

Including Programming Experiments

HEATH COMPANY
BENTON HARBOR, MICHIGAN 49022

Model EB-6401A
595-3429-07

Library of Congress Cataloging in Publication Data
Main entry under title:

Microprocessors : including programming experiments.

Includes index.
1. Microprocessors. 2. Motorola 6800
(Microprocessor) 3. Microprocessors—Programming.
I. Heathkit/Zenith Educational Systems (Group)
QA76.5.M52245 1985 001.64 85-784
ISBN 0-87119-105-9

CONTENTS

Introduction VI
Course Objectives IX

UNIT ONE — NUMBER SYSTEMS AND CODES

Introduction 1-2
Unit Objectives 1-3
Decimal Number System 1-4
Binary Number System 1-9
Octal Number System 1-18
Hexadecimal Number System 1-27
Binary Codes 1-40
Unit 1 Summary 1-57
Positive Powers of 2 1-62
Negative Powers of 2 1-63
Positive Powers of 8 1-64
Positive Powers of 16 1-64
Negative Powers of 16 1-64

UNIT TWO — MICROCOMPUTER BASICS

Introduction 2-2
Unit Objectives 2-3
Terms and Conventions 2-4
An Elementary Microcomputer 2-12
Executing a Program 2-26
Addressing Modes 2-41
Unit 2 Summary 2-69

UNIT THREE — COMPUTER ARITHMETIC

Introduction 3-2
Unit Objectives 3-3
Binary Arithmetic 3-4

Two's Complement Arithmetic 3-24
Boolean Operations 3-33
Unit 3 Summary 3-43

UNIT FOUR — INTRODUCTION TO PROGRAMMING

Introduction 4-2
Unit Objectives 4-3
Branching 4-4
Conditional Branching 4-18
Microcomputer Programming 4-27
Algorithms 4-37
Additional Instructions 4-54
Unit 4 Summary 4-67

UNIT FIVE — THE 6800 MICROPROCESSOR — PART 1

Introduction 5-2
Unit Objectives 5-3
Architecture of the 6800 MPU 5-4
Instruction Set of the 6800 MPU 5-13
New Addressing Modes 5-34
Unit 5 Summary 5-51

UNIT SIX — THE 6800 MICROPROCESSOR — PART 2

Introduction 6-2
Unit Objectives 6-3
Stack Operations 6-4
Subroutines 6-15
Input — Output (I/O) Operations 6-25
Interrupts 6-35
Unit 6 Summary 6-49

PROGRAMMING EXPERIMENTS

Experiment 1 — Binary/Decimal Training Program 3
Experiment 2 — Hexadecimal/Decimal Training Program 13
Experiment 3 — Straight Line Programs 23
Experiment 4 — Arithmetic and Logic Instructions 41
Experiment 5 — Program Branches 53
Experiment 6 — Additional Instructions 89
Experiment 7 — New Addressing Modes 111
Experiment 8 — Arithmetic Operations 121
Experiment 9 — Stack Operations 133
Experiment 10 — Subroutines 141

EXAMINATIONS

Unit 1 Examination — Number Systems and Codes 165
Unit 2 Examination — Microcomputer Basics 169
Unit 3 Examination — Computer Arithmetic 173
Unit 4 Examination — Introduction to Programming 175
Unit 5 Examination — The 6800 Microprocessor — Part 1 ... 177
Unit 6 Examination — The 6800 Microprocessor — Part 2 ... 185

APPENDIX A — DEFINITION OF THE EXECUTABLE INSTRUCTIONS

INDEX

PROGRAMMING CARD — 6800/6808 INSTRUCTION SET
(BOOK 1 TEAR-OUT)

INTRODUCTION

The Microprocessors course is designed to teach you the fundamentals of the microprocessor unit (MPU), or as it is also known, the central processor unit (CPU). This is the decision maker, or brain, of the computer. Like the brain in your body, the MPU needs support from many peripheral devices to maintain the system. Because of that interrelationship, a course on microprocessors is not complete without a through discussion of how the MPU interfaces to its support devices and the outside world. To that end, we will show you how the MPU is structured, how you communicate with the MPU (its language), and how the MPU communicates with you and the system under its control.

The course is arranged into two broad topics or sections: microprocessor theory and programming, and microprocessor interfacing. The first topic is covered in the textbook set "Book 1" and "Book 2." In addition to the text material, these books contain related "programming experiments" and an appendix with a description of the complete 6800/6808 microprocessor instruction set. The second topic is covered in the "Student Workbook." In addition to interfacing theory, this textbook contains related "interfacing experiments" and an appendix with data sheets for all of the more complex IC's such as the 6800/6808 MPU.

Microprocessors communicate with numbers. They use the simplest number system, binary, to process data. Then, depending on the computer, they use the octal, decimal, or hexadecimal number systems, and several of codes to communicate with man. The first unit in this course will describe the most common number systems and codes used by computers. Once you have built a foundation in number systems (microprocessor/computer communication), we will introduce you to the basic elements of a microprocessor in Unit 2. This is followed by a unit on computer arithmetic, to show you how a microprocessor manipulates a number system. The fourth unit introduces you to programming the computer. The last two units in the first section of the course expand on the theory of microprocessors and relate it specifically to the 6800 family of microprocessors.

The second section of the course contains two units on interfacing. The first describes interfacing fundamentals; including 3-state logic, all of the interface pins on the 6800 and 6808 microprocessor, and instruction timing. It also describes memory and display interfacing.

The second unit deals with interfacing with switches and the Peripheral Interface Adapter. Keep in mind that after you interface either the 6800 or 6808 MPU you, still have to program its operation. Therefore, you must complete the first section of the course on theory and programming before you attempt to interface the MPU.

There are two trainers that can be used with this course. One has the model number ET-3400, while the other has the model number ET-3400A. You can think of them as first and second generation trainers. The older ET-3400 trainer uses the 6800 MPU and has up to 512 bytes of memory. The ET-3400A uses the 6808 MPU and has 1024 bytes of memory. Both trainers operate in essentially the same manner. Any differences are fully explained in the experiments.

One final point about the 6800 MPU and the 6808 MPU. Both devices are essentially the same. The 6800 MPU uses an external clock while the 6808 MPU has an internal clock. Both microprocessors respond to the same instruction set, and operate in essentially the same manner. Any differences are hardware related and they are fully explained in the interfacing section. If any of the terms used in this introduction are unfamiliar, don't worry, each will be described in detail in the course.

A special programming card is included at the end of "Book 1." It has a perforated edge so that you can easily remove it from the textbook. The card contains all of the data needed to program a 6800 or 6808 MPU. You will find the card very helpful when you begin writing programs.

How to Use this Course

You will find two types of objectives in this course. Broad "Course Objectives" are listed after this Introduction. They give a general overview of the goals of the course. More specific "Unit Objectives" are listed near the front of each unit. They tell you exactly what you will be able to do as a result of studying that unit. This lets you maintain a constant check of your progress through the course.

Each unit is divided into sections. A "Self-Test Review" is included at the end of each section. This gives you an opportunity to evaluate your progress and pinpoint any areas that need review.

At the end of each unit, you will be directed to an experiment. The "Programming Experiments" are located after Unit 6, in "Book 2," while the "Interfacing Experiments" are located after Unit 8, in the "Student Workbook."

The "Unit Examinations" for Units 1 through 6 are located after the "Programming Experiments," while those for Units 7 and 8 are located after the "Interfacing Experiments." Your instructor has the answers.

COURSE OBJECTIVES

When you have completed this course, you will be able to:

1. Program a representative microprocessor.
2. Interface a representative microprocessor with the "outside world."
3. Describe the internal register structure of the 6800 and 6808 microprocessors.
4. Develop a program flowchart to define a problem.
5. Input and output data through a Peripheral Interface Adapter (PIA).

Unit 1

NUMBER SYSTEMS AND CODES

INTRODUCTION

The purpose of this first unit on microprocessors is to give you a firm foundation in number systems and codes. Binary numbers and codes are the basic language of all microprocessors. Octal and hexadecimal numbers allow easy manipulation of binary numbers and data. Thus, a good foundation in numbers and codes is essential to understanding microprocessors.

This unit will reacquaint you with the decimal number system, then expand the basic concept of numbers to the binary, octal, and hexadecimal systems. Understanding these systems fully will help you understand the many digital codes used with microprocessors. Although this unit can only give you a working knowledge of numbers and codes, you will become more proficient with them as you proceed through the units that follow.

A listing of number system tables has been provided at the end of this unit.

Examine the Unit Objectives listed in the next section to see what you will learn in this unit.

UNIT OBJECTIVES

When you complete this unit you will have the following knowledge and capabilities:

1. Given any decimal number, you will be able to convert it into its binary, octal, hexadecimal, and BCD equivalent.

2. Given any binary number, you will be able to convert it into its decimal, octal, hexadecimal, and BCD equivalent.

3. Given any octal number, you will be able to convert it into its decimal and binary equivalent.

4. Given any hexadecimal number, you will be able to convert it into its decimal and binary equivalent.

5. Given any BCD code, you will be able to convert it into its decimal and binary equivalent.

6. Given a list of popular digital codes, you will be able to read and identify them including pure binary, natural 8421 BCD, Gray, ASCII, and BAUDOT.

7. You will be able to convert a letter or number into its ASCII binary code, and convert an ASCII binary code into its letter or number equivalent.

8. You will be able to define the following terms:

Radix	BCD
Integer	Gray Code
Decimal	ASCII
Binary	BAUDOT
Octal	Most Significant Bit (MSB)
Hexadecimal	Least Significant Bit (LSB)
Bit	Most Significant Digit (MSD)
Parity	Least Significant Digit (LSD)

DECIMAL NUMBER SYSTEM

The number system we are all familiar with is the decimal number system. This system was originally devised by Hindu mathematicians in India about 400 A.D. The Arabs began to use the system about 800 A.D., where it became known as the Arabic Number System. After it was introduced to the European community about 1200 A.D., the system soon acquired the title "decimal number system."

A basic distinguishing feature of a number system is its **base** or **radix**. The base indicates the number of characters or digits used to represent quantities in that number system. The decimal number system has a base or radix of 10 because we use the ten digits 0 through 9 to represent quantities. When a number system is used where the base is not known, a subscript is used to show the base. For example, the number 4603_{10} is derived from a number system with a base of 10.

Positional Notation The decimal number system is positional or weighted. This means each digit position in a number carries a particular weight which determines the magnitude of that number. Each position has a weight determined by some power of the number system base, in this case 10. The positional weights are 10^0 (units)*, 10^1 (tens), 10^2 (hundreds), etc. Refer to Figure 1-1 for a condensed listing of powers of 10.

$10^0 = 1$
$10^1 = 10$
$10^2 = 100$
$10^3 = 1,000$
$10^4 = 10,000$
$10^5 = 100,000$
$10^6 = 1,000,000$
$10^7 = 10,000,000$
$10^8 = 100,000,000$
$10^9 = 1,000,000,000$

Figure 1-1
Condensed listing of powers of 10.

*Any number with an exponent of zero is equal to one.

We evaluate the total quantity of a number by considering the specific digits and the weights of their positions. For example, the decimal number 4603 is written in the shorthand notation with which we are all familiar. This number can also be expressed with positional notation.

$$\begin{aligned}&(4 \times 10^3) + (6 \times 10^2) + (0 \times 10^1) + (3 \times 10^0) = \\ &(4 \times 1000) + (6 \times 100) + (0 \times 10) + (3 \times 1) = \\ &4000 + 600 + 0 + 3 = 4603_{10}\end{aligned}$$

To determine the value of a number, multiply each digit by the weight of its position and add the results.

Fractional Numbers So far, only **integer** or whole numbers have been discussed. An integer is any of the natural numbers, the negatives of these numbers, or zero (that is, 0, 1, 4, 7, etc.). Thus, an integer represents a whole or complete number. But, it is often necessary to express quantities in terms of fractional parts of a whole number.

Decimal fractions are numbers whose positions have weights that are **negative powers of ten** such as $10^{-1} = \frac{1}{10} = 0.1$, $10^{-2} = \frac{1}{100} = 0.01$, etc.

Figure 1-2 provides a condensed listing of negative powers of 10 (decimal fractions).

$$10^{-1} = \frac{1}{10} = 0.1$$

$$10^{-2} = \frac{1}{100} = 0.01$$

$$10^{-3} = \frac{1}{1000} = 0.001$$

$$10^{-4} = \frac{1}{10{,}000} = 0.0001$$

$$10^{-5} = \frac{1}{100{,}000} = 0.00001$$

$$10^{-6} = \frac{1}{1{,}000{,}000} = 0.000001$$

Figure 1-2
Condensed listing of negative powers of 10.

A radix point (decimal point for base 10 numbers) **separates** the **integer** and **fractional** parts of a number. The integer or whole portion is to the left of the decimal point and has positional weights of units, tens, hundreds, etc. The fractional part of the number is to the right of the decimal point and has positional weights of tenths, hundredths, thousandths, etc. To illustrate this, the decimal number 278.94 can be written with positional notation as shown below.

$$(2 \times 10^2) + (7 \times 10^1) + (8 \times 10^0) + (9 \times 10^{-1}) + (4 \times 10^{-2}) =$$
$$(2 \times 100) + (7 \times 10) + (8 \times 1) + (9 \times 1/10) + (4 \times 1/100) =$$
$$200 + 70 + 8 + 0.9 + 0.04 = 278.94_{10}$$

In this example, the left-most digit (2×10^2) is the **most significant digit** or MSD because it carries the greatest weight in determining the value of the number. The right-most digit, called the **least significant digit** or LSD, has the lowest weight in determining the value of the number. Therefore, as the term implies, the MSD is the digit that will affect the greatest change when its value is altered. The LSD has the smallest effect on the complete number value.

Self-Test Review

1. The __________ indicates the number of characters or digits in a number system.

2. In the decimal number system, the base or radix is ________________.

3. Write the following numbers using positional notation.

   A. 4563_{10}
   B. 26.32_{10}
   C. 536.9_{10}

4. In the decimal number system, the radix point is called the __________ __________.

5. Convert the following positional notations into their shorthand decimal form.

   A. $(5 \times 10^1) + (2 \times 10^0) + (3 \times 10^{-1}) + (8 \times 10^{-2})$
   B. $(4 \times 10^{-1}) + (6 \times 10^{-2}) + (2 \times 10^{-3})$
   C. $(3 \times 10^3) + (7 \times 10^2) + (1 \times 10^1) + (0 \times 10^0)$

6. The radix point separates the __________ and __________ parts of a number.

Answers

1. base or radix.

2. 10.

3. A. $4563_{10} = 4000 + 500 + 60 + 3.$
   $= (4 \times 10^3) + (5 \times 10^2) + (6 \times 10^1) + (3 \times 10^0)$
   B. $(2 \times 10^1) + (6 \times 10^0) + (3 \times 10^{-1}) + (2 \times 10^{-2})$
   C. $(5 \times 10^2) + (3 \times 10^1) + (6 \times 10^0) + (9 \times 10^{-1})$

4. decimal point.

5. A. $(5 \times 10^1) + (2 \times 10^0) + (3 \times 10^{-1}) + (8 \times 10^{-2}) =$
   $(5 \times 10) + (2 \times 1) + (3 \times 1/10) + (8 \times 1/100) =$
   $50 + 2 + 0.3 + 0.08 = 52.38_{10}$
   B. 0.462_{10}
   C. 3710_{10}

6. integer or whole, fractional.

BINARY NUMBER SYSTEM

The simplest number system that uses positional notation is the binary number system. As the name implies, a **binary** system contains only two elements or states. In a number system this is expressed as a base of 2, using the digits 0 and 1. These two digits have the same basic value as 0 and 1 in the decimal number system.

Because of its simplicity, microprocessors use the binary number system to manipulate data. Binary data is represented by binary digits called **bits**. The term bit is derived from the contraction of **b**inary dig**it**. Microprocessors operate on groups of bits which are referred to as words. The binary number 11101101 contains eight bits.

Positional Notation

As with the decimal number system, each bit (digit) position of a binary number carries a particular weight which determines the magnitude of that number. The weight of each position is determined by some power of the number system base (in this example 2). To evaluate the total quantity of a number, consider the specific bits and the weights of their positions. (Refer to Figure 1-3 for a condensed listing of powers of 2.) For example, the binary number 110101 can be written with positional notation as follows:

$$(1 \times 2^5) + (1 \times 2^4) + (0 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0)$$

To determine the decimal value of the binary number 110101, multiply each bit by its positional weight and add the results.

$$(1 \times 32) + (1 \times 16) + (0 \times 8) + (1 \times 4) + (0 \times 2) + (1 \times 1) =$$
$$32 + 16 + 0 + 4 + 0 + 1 = 53_{10}$$

$2^0 = 1_{10}$	$2^6 = 64_{10}$
$2^1 = 2_{10}$	$2^7 = 128_{10}$
$2^2 = 4_{10}$	$2^8 = 256_{10}$
$2^3 = 8_{10}$	$2^9 = 512_{10}$
$2^4 = 16_{10}$	$2^{10} = 1024_{10}$
$2^5 = 32_{10}$	$2^{11} = 2048_{10}$

Figure 1-3
Condensed listing of powers of 2.

Fractional binary numbers are expressed as negative powers of 2. Figure 1-4 provides a condensed listing of negative powers of 2. In positional notation, the binary number 0.1101 can be expressed as follows:

$$(1 \times 2^{-1}) + (1 \times 2^{-2}) + (0 \times 2^{-3}) + (1 \times 2^{-4})$$

To determine the decimal value of the binary number 0.1101, multiply each bit by its positional weight and add the results.

$$(1 \times 1/2) + (1 \times 1/4) + (0 \times 1/8) + (1 \times 1/16) =$$
$$0.5 + 0.25 + 0 + 0.0625 = 0.8125_{10}$$

In the binary number system, the radix point is called the binary point.

$$2^{-1} = \frac{1}{2} = 0.5_{10}$$

$$2^{-2} = \frac{1}{4} = 0.25_{10}$$

$$2^{-3} = \frac{1}{8} = 0.125_{10}$$

$$2^{-4} = \frac{1}{16} = 0.0625_{10}$$

$$2^{-5} = \frac{1}{32} = 0.03125_{10}$$

$$2^{-6} = \frac{1}{64} = 0.015625_{10}$$

$$2^{-7} = \frac{1}{128} = 0.0078125_{10}$$

$$2^{-8} = \frac{1}{256} = 0.00390625_{10}$$

Figure 1-4
Condensed listing of negative powers of 2.

Converting Between the Binary and Decimal Number Systems

In working with microprocessors, you will often need to determine the decimal value of binary numbers. In addition, you will find it necessary to convert a specific decimal number into its binary equivalent. The following information shows how such conversions are accomplished.

Binary to Decimal To convert a binary number into its decimal equivalent, add together the weights of the positions in the number where binary 1's occur. The weights of the integer and fractional positions are indicated below.

INTEGER									FRACTIONAL		
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	●	2^{-1}	2^{-2}	2^{-3}
128	64	32	16	8	4	2	1		.5	.25	.125

Binary Point

As an example, convert the binary number 1010 into its decimal equivalent. Since no binary point is shown, the number is assumed to be an integer number, where the binary point is to the right of the number. The right-most bit, called the **least significant bit** or LSB, has the lowest integer weight of $2^0 = 1$. The left-most bit is the **most significant bit** (MSB) because it carries the greatest weight in determining the value of the number. In this example, it has a weight of $2^3 = 8$. To evaluate the number, add together the weights of the positions where binary 1's appear. In this example, 1's occur in the 2^3 and 2^1 positions. The decimal equivalent is ten.

Binary Number	1		0		1		0		
Position Weights	2^3		2^2		2^1		2^0		
Decimal Equivalent									
	8	+	0	+	2	+	0	=	10_{10}

To further illustrate this process, convert the binary number 101101.11 into its decimal equivalent.

Binary Number	1		0		1		1		0		1		.1		1		
Position Weights	2^5		2^4		2^3		2^2		2^1		2^0		2^{-1}		2^{-2}		
Decimal Equivalent																	
	32	+	0	+	8	+	4	+	0	+	1	+	.5	+	.25	=	45.75_{10}

Decimal to Binary A decimal integer number can be converted to a different base or radix through successive divisions by the desired base. To convert a decimal integer number to its binary equivalent, successively divide the number by 2 and note the remainders. When you divide by 2, the remainder will always be 1 or 0.

The remainders form the equivalent binary number.

As an example, the decimal number 25 is converted into its binary equivalent.

$25 \div 2 = 12$ with remainder	1	← LSB
$12 \div 2 = 6$	0	
$6 \div 2 = 3$	0	
$3 \div 2 = 1$	1	
$1 \div 2 = 0$	1	← MSB

Divide the decimal number by 2 and note the remainder. Then divide the quotient by 2 and again note the remainder. Then divide the quotient by 2 and again note the remainder. Continue this division process until 0 results. Then collect remainders beginning with the last or most significant bit (MSB) and proceed to the first or least significant bit (LSB). The number $11001_2 = 25_{10}$. Notice that the remainders are collected in the reverse order. That is, the first remainder becomes the least significant bit, while the last remainder becomes the most significant bit.

NOTE: Do not attempt to use a calculator to perform this conversion. It would only supply you with confusing results.

To further illustrate this, the decimal number 175 is converted into its binary equivalent.

$175 \div 2 = 87$ with remainder	1	← LSB
$87 \div 2 = 43$	1	
$43 \div 2 = 21$	1	
$21 \div 2 = 10$	1	
$10 \div 2 = 5$	0	
$5 \div 2 = 2$	1	
$2 \div 2 = 1$	0	
$1 \div 2 = 0$	1	← MSB

The division process continues until 0 results. The remainders are collected to produce the number $10101111_2 = 175_{10}$.

To convert a decimal fraction to a different base or radix, multiply the fraction successively by the desired base and record any integers produced by the multiplication as an overflow. For example, to convert the decimal fraction 0.3125 into its binary equivalent, multiply repeatedly by 2.

$0.3125 \times 2 = 0.625 = 0.625$	with overflow	0	← MSB
$0.6250 \times 2 = 1.250 = 0.250$		1	
$0.2500 \times 2 = 0.500 = 0.500$		0	
$0.5000 \times 2 = 1.000 = 0$		1	← LSB

These multiplications will result in numbers with a 1 or 0 in the units position (the position to the left of the decimal point). By recording the value of the units position, you can construct the equivalent binary fraction. This units position value is called the "overflow." Therefore, when 0.3125 is multiplied by 2, the overflow is 0. This becomes the most significant bit (MSB) of the binary equivalent fraction. Then 0.625 is multiplied by 2. Since the product is 1.25, the overflow is 1. When there is an overflow of 1, it is effectively subtracted from the product when the value is recorded. Therefore, only 0.25 is multiplied by 2 in the next multiplication process. This method continues until an overflow with no fraction results. It is important to note that you can not always obtain 0 when you multiply by 2. Therefore, you should only continue the conversion process to the accuracy or precision you desire. Collect the conversion overflows beginning at the radix (binary) point with the MSB and proceed to the LSB. This is the same order in which the overflows were produced. The number $0.0101_2 = 0.3125_{10}$.

To further illustrate this process, the decimal fraction 0.90625 is converted into its binary equivalent.

$0.90625 \times 2 = 1.8125 = 0.8125$	with overflow	1	← MSB
$0.81250 \times 2 = 1.6250 = 0.6250$		1	
$0.62500 \times 2 = 1.2500 = 0.2500$		1	
$0.25000 \times 2 = 0.5000 = 0.5000$		0	
$0.50000 \times 2 = 1.0000 = 0$		1	← LSB

The multiplication process continues until either 0 or the desired precision is obtained. The overflows are then collected beginning with the MSB at the binary (radix) point and proceeding to the LSB. The number $0.11101_2 = 0.90625_{10}$.

If the decimal number contains both an integer and fraction, you must separate the integer and fraction using the decimal point as the break point. Then perform the appropriate conversion process on each number portion. After you convert the binary integer and binary fraction, recombine them. For example, the decimal number 14.375 is converted into its binary equivalent.

$14.375_{10} = 14_{10} + 0.375_{10}$

$14 \div 2 = 7$	with remainder	0	←	LSB
$7 \div 2 = 3$		1		
$3 \div 2 = 1$		1		
$1 \div 2 = 0$		1	←	MSB

$\boxed{14_{10} = 1110_2}$

$0.375 \times 2 = 0.75 = 0.75$	with overflow	0	←	MSB
$0.750 \times 2 = 1.50 = 0.50$		1		
$0.500 \times 2 = 1.00 = 0$		1	←	LSB

$\boxed{0.375_{10} = 0.011_2}$

$14.375_{10} = 14_{10} + 0.375_{10} = 1110_2 + 0.011_2 = 1110.011_2$.

Self-Test Review

7. The base or radix of the binary number system is ____________.

8. A binary digit is called a ____________.

9. Convert the following binary integers to decimal.

   A. 101101
   B. 1001
   C. 1101100

10. Convert the following binary fractions to decimal.

    A. 0.011
    B. 0.01101
    C. 0.1001

11. Convert the following decimal integers to binary.

    A. 63
    B. 12
    C. 132

12. Convert the following decimal fractions to binary.

    A. 0.4375
    B. 0.96875
    C. 0.625

13. Convert 13.125_{10} to binary.

Answers

7. 2

8. bit

9. A. $101101_2 =$
$(1 \times 2^5) + (0 \times 2^4) + (1 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) =$
$32 + 0 + 8 + 4 + 0 + 1 = 45_{10}$
B. $1001_2 = 9_{10}$
C. $1101100_2 = 108_{10}$

10. A. $0.011_2 = (0 \times 2^{-1}) + (1 \times 2^{-2}) + (1 \times 2^{-3}) =$
$0 + \frac{1}{4} + \frac{1}{8} = 0 + 0.25 + 0.125 = 0.375_{10}$
B. $0.01101_2 = 0.40625_{10}$
C. $0.1001_2 = 0.5625_{10}$

11. A.

$63 \div 2 = 31$	with remainder	1	← LSB
$31 \div 2 = 15$		1	
$15 \div 2 = 7$		1	
$7 \div 2 = 3$		1	
$3 \div 2 = 1$		1	
$1 \div 2 = 0$		1	← MSB

$\boxed{63_{10} = 111111_2}$

B. $12_{10} = 1100_2$
C. $132_{10} = 10000100_2$

12. A.

```
0.4375 × 2 = 0.875 = 0.875   with overflow  0 ← MSB
0.8750 × 2 = 1.750 = 0.750                  1
0.7500 × 2 = 1.500 = 0.500                  1
0.5000 × 2 = 1.000 = 0                      1 ← LSB
```

$\boxed{0.4375_{10} = 0.0111_2}$

B. $0.96875_{10} = 0.11111_2$

C. $0.625_{10} = 0.101_2$

13. $13.125_{10} = 13_{10} + 0.125_{10}$

```
13 ÷ 2 = 6   with remainder  1 ← LSB
 6 ÷ 2 = 3                   0
 3 ÷ 2 = 1                   1
  1÷ 2 = 0                   1 ← MSB
```

$\boxed{13_{10} = 1101_2}$

```
0.125 × 2 = 0.25 = 0.25   with overflow  0 ← MSB
0.250 × 2 = 0.50 = 0.50                  0
0.500 × 2 = 1.00 = 0                     1 ← LSB
```

$\boxed{0.125_{10} = 0.001_2}$

$13.125_{10} = 13_{10} + 0.125_{10} = 1101_2 + 0.001_2 = 1101.001_2$

OCTAL NUMBER SYSTEM

Octal is another number system that is often used with microprocessors. It has a base (radix) of 8, and uses the digits 0 through 7. These eight digits have the same basic value as the digits 0—7 in the decimal number system.

As with the binary number system, each digit position of an octal number carries a positional weight which determines the magnitude of that number. The weight of each position is determined by some power of the number system base (in this example, 8). To evaluate the total quantity of a number, consider the specific digits and the weights of their positions. Refer to Figure 1-5 for a condensed listing of powers of 8. For example, the octal number 372.01 can be written with positional notation as follows:

$$(3 \times 8^2) + (7 \times 8^1) + (2 \times 8^0) + (0 \times 8^{-1}) + (1 \times 8^{-2})$$

The decimal value of the octal number 372.01 is determined by multiplying each digit by its positional weight and adding the results. As with decimal and binary numbers, the radix (octal) point separates the integer from the fractional part of the number.

$$(3 \times 64) + (7 \times 8) + (2 \times 1) + (0 \times 0.125) + (1 \times 0.015625) =$$
$$192 + 56 + 2 + 0 + 0.015625 = 250.015625_{10}$$

$$8^{-4} = \frac{1}{4096} = 0.000244140625_{10}$$
$$8^{-3} = \frac{1}{512} = 0.001953125_{10}$$
$$8^{-2} = \frac{1}{64} = 0.015625_{10}$$
$$8^{-1} = \frac{1}{8} = 0.125_{10}$$

$$1_{10} = 8^0$$
$$8_{10} = 8^1$$
$$64_{10} = 8^2$$
$$512_{10} = 8^3$$
$$4096_{10} = 8^4$$
$$32768_{10} = 8^5$$
$$262144_{10} = 8^6$$

Figure 1-5
Condensed listing of powers of 8.

Conversion From Decimal to Octal

Decimal to octal conversion is accomplished in the same manner as decimal to binary, with one exception; the base number is now 8 rather than 2. As an example, the decimal number 194 is converted into its octal equivalent.

$$\begin{array}{lll}
194 \div 8 = 24 \text{ with remainder } & 2 & \leftarrow \text{LSD} \\
24 \div 8 = 3 & 0 & \\
3 \div 8 = 0 & 3 & \leftarrow \text{MSD}
\end{array}$$

Divide the decimal number by 8 and note the remainder. (The remainder can be any number from 0 to 7.)

Then divide the quotient by 8 and again note the remainder. Continue dividing until 0 results. Finally, collect the remainders beginning with the last or most significant digit (MSD) and proceed to the first or least significant digit (LSD). The number $302_8 = 194_{10}$. Figure 1-6 illustrates the relationship between the first several decimal, octal, and binary integers.

DECIMAL	OCTAL	BINARY
0	0	0
1	1	1
2	2	10
3	3	11
4	4	100
5	5	101
6	6	110
7	7	111
8	10	1000
9	11	1001
10	12	1010
11	13	1011
12	14	1100
13	15	1101
14	16	1110
15	17	1111
16	20	10000
17	21	10001
18	22	10010
19	23	10011
20	24	10100

Figure 1-6
Sample comparison of decimal, octal, and binary integers.

To further illustrate this process, the decimal number 175 is converted into its octal equivalent.

$$\begin{array}{rll} 175 \div 8 = 21 \text{ with remainder } & 7 & \leftarrow \text{LSD} \\ 21 \div 8 = 2 & 5 & \\ 2 \div 8 = 0 & 2 & \leftarrow \text{MSD} \end{array}$$

The division process continues until a quotient of 0 results. The remainders are collected, producing the number $257_8 = 175_{10}$.

To convert a decimal fraction to an octal fraction, multiply the fraction successively by 8 (octal base). As an example, the decimal fraction 0.46875 is converted into its octal equivalent.

$$\begin{array}{lll} 0.46875 \times 8 = 3.75 = 0.75 \text{ with overflow} & 3 & \leftarrow \text{MSD} \\ 0.75000 \times 8 = 6.00 = 0 & 6 & \leftarrow \text{LSD} \end{array}$$

Multiply the decimal number by 8. If the product exceeds one, subtract the integer (overflow) from the product. Then multiply the product fraction by 8 and again note any "overflow." Continue multiplying until an overflow, with 0 for a fraction, results. Remember, you can not always obtain 0 when you multiply by 8. Therefore, you should only continue this conversion process to the accuracy or precision you desire. Collect the conversion overflows beginning at the radix (octal point) with the MSD and proceed to the LSD. The number $0.36_8 = 0.46875_{10}$. Figure 1-7 illustrates the relationship between decimal, octal, and binary fractions.

Now, the decimal fraction 0.136 will be converted into its octal equivalent with four-place precision.

$$\begin{array}{lll} 0.136 \times 8 = 1.088 = 0.088 \text{ with overflow} & 1 & \leftarrow \text{MSD} \\ 0.088 \times 8 = 0.704 = 0.704 & 0 & \\ 0.704 \times 8 = 5.632 = 0.632 & 5 & \\ 0.632 \times 8 = 5.056 = 0.056 & 5 & \leftarrow \text{LSD} \end{array}$$

$$0.136_{10} \cong 0.1055_8$$

The number 0.1055_8 approximately equals 0.136_{10}. If you convert 0.1055_8 back to decimal (using positional notation), you will find $0.1055_8 = 0.135986328125_{10}$. This example shows that extending the precision of your conversion is of little value unless extreme accuracy is required.

DECIMAL	OCTAL	BINARY
0.015625	0.01	0.000001
0.03125	0.02	0.00001
0.046875	0.03	0.000011
0.0625	0.04	0.0001
0.078125	0.05	0.000101
0.09375	0.06	0.00011
0.109375	0.07	0.000111
0.125	0.1	0.001
0.140625	0.11	0.001001
0.15625	0.12	0.00101
0.171875	0.13	0.001011
0.1875	0.14	0.0011
0.203125	0.15	0.001101
0.21875	0.16	0.00111
0.234375	0.17	0.001111
0.25	0.2	0.01
0.265625	0.21	0.010001
0.28125	0.22	0.01001
0.296875	0.23	0.010011
0.3125	0.24	0.0101

Figure 1-7
Sample comparison of decimal, octal, and binary fractions.

As with decimal to binary conversion of a number that contains both an integer and fraction, decimal to octal conversion requires two operations. You must separate the integer from the fraction, then perform the appropriate conversion on each number. After you convert them, you must recombine the octal integer and octal fraction. For example, convert the decimal number 124.78125 into its octal equivalent.

$124.78125_{10} = 124_{10} + 0.78125_{10}$

$124 \div 8 = 15$ with remainder $4 \leftarrow$ LSD

$15 \div 8 = 1$ $\quad 7$

$1 \div 8 = 0$ $\quad 1 \leftarrow$ MSD

$\boxed{124_{10} = 174_8}$

$0.78125 \times 8 = 6.25 = 0.25$ with overflow $6 \leftarrow$ MSD

$0.25000 \times 8 = 2.00 = 0$ $\quad 2 \leftarrow$ LSD

$\boxed{0.78125_{10} = 0.62_8}$

$124.78125_{10} = 124_{10} + 0.78125_{10} = 174_8 + 0.62_8 = 174.62_8$

Converting Between the Octal and Binary Number Systems

Microprocessors manipulate data using the binary number system. However, when larger quantities are involved, the binary number system can become cumbersome. Therefore, other number systems are frequently used as a form of binary shorthand to speed-up and simplify data entry and display. The octal number system is one of the systems that is used in this manner. It is similar to the decimal number system, which makes it easier to understand numerical values. In addition, conversion between binary and octal is readily accomplished because of the value structure of octal. Figures 1-6 and 1-7 illustrate the relationship between octal and binary integers and fractions.

As you know, three bits of a binary number exactly equal eight value combinations. Therefore, you can represent a 3-bit binary number with a 1-digit octal number.

$$101_2 = (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) = 4 + 0 + 1 = 5_8$$

Because of this relationship, converting binary to octal is simple and straight forward. For example, binary number 101001 is converted into its octal equivalent.

101001_2
REWRITE AS
MSB ↘101 001↙ LSB
YIELDS
51_8

To convert a binary number to octal, first separate the number into groups containing three bits, beginning with the least significant bit. Then convert each 3-bit group into its octal equivalent. This gives you an octal number equal in value to the binary number.

Binary fractions can also be converted to their octal equivalents using the same process, with one exception. The binary bits must be separated into groups of three beginning with the most significant bit. For example, the binary fraction 0.011101 is converted into its octal equivalent.

$$\begin{array}{c} 0.011101_2 \\ \text{REWRITE AS} \\ \text{MSB} \searrow \qquad \swarrow \text{LSB} \\ 0.011 \quad 101 \\ \text{YIELDS} \\ 0.35_8 \end{array}$$

Again, you must first separate the binary number into groups of three beginning at the radix (binary) point. Then convert each 3-bit group into its octal equivalent.

To separate binary numbers into 3-bit groups when the number does not contain the necessary bits, add zeros to the number until the number can be separated into 3-bit groups. For example, binary number 10010101.1011 is converted into its octal equivalent.

$$\begin{array}{c} 10010101.1011_2 \\ \text{REWRITE AS} \\ \text{MSB} \searrow \qquad\qquad\qquad \swarrow \text{LSB} \\ \underline{0}10 \quad 010 \quad 101.101 \quad 1\underline{00} \\ \text{YIELDS} \\ 225.54_8 \end{array}$$

As before, the integer part of the number is separated into 3-bit groups, beginning at the radix (binary) point. Note that the third group contains only two bits. However, a zero can be added to the group without changing the value of the binary number. Next, the fractional part of the number is separated into 3-bit groups, beginning at the radix (binary) point. Note that the second group contains only one bit. By adding two zeros to the group, the group is complete with no change in the value of the binary number.

NOTE: Whenever you add zeros to a **binary integer**, always place them to the **left** of the most significant bit. When you add zeros to a **binary fraction**, always place them to the **right** of the least significant bit.

After you have formed the 3-bit groups, convert each group into its octal equivalent. This gives you an octal number equal in value to the binary number. Now convert binary number 1101110.01 into its octal equivalent.

$$\begin{array}{c} 1101110.01_2 \\ \text{REWRITE AS} \\ \text{MSB} \searrow \qquad\qquad \swarrow \text{LSB} \\ \underline{00}1 \quad 101 \quad 110.01\underline{0} \\ \text{YIELDS} \\ 156.2_8 \end{array}$$

Separate the integer and fraction into 3-bit groups, adding zeros as necessary. Then convert each 3-bit group to octal. **Never** shift the radix (binary) point in order to form 3-bit groups.

Converting octal to binary is just the opposite of the previous process. You simply convert each octal number into its 3-bit binary equivalent. For example, convert the octal number 75.3 into its binary equivalent.

$$\begin{array}{c} 75.3_8 \\ \text{YIELDS} \\ \text{MSB} \searrow \qquad\qquad \swarrow \text{LSB} \\ 111 \qquad 101.011 \\ \\ \text{REWRITE AS} \\ 111101.011_2 \end{array}$$

The above example is a simple conversion. Now a more complex octal number (1752.714) will be converted to a binary number.

$$\begin{array}{c} 1752.714_8 \\ \text{YIELDS} \\ \text{MSB} \searrow \qquad\qquad\qquad\qquad \swarrow \text{LSB} \\ \underline{00}1 \quad 111 \quad 101 \quad 010.111 \quad 001 \quad 1\underline{00} \\ \text{REWRITE AS} \\ 1111101010.1110011_2 \end{array}$$

Again, each octal digit is converted into its 3-bit binary equivalent. However, in this example, there are two insignificant zeros in front of the MSB and after the LSB. Since these zeros have no value, they should be removed from the final result.

Self-Test Review

14. The base or radix of the octal number system is ________________.

15. Convert the following decimal integers to octal.

    A. 156
    B. 32
    C. 1785

16. Convert the following decimal fractions to octal. Do not use greater than 4-place precision.

    A. 0.1432
    B. 0.8125
    C. 0.6832

17. Convert 735.984375_{10} to octal.

18. Convert the following binary numbers to octal.

    A. 10000111.01101
    B. 11101.0101
    C. 1001101.000001

19. Convert the following octal numbers to binary.

    A. 372.61
    B. 11.001
    C. 3251.034

Answers

14. 8

15. A. $156 \div 8 = 19$ with remainder $4 \leftarrow$ LSD
$19 \div 8 = 2$ 3
$2 \div 8 = 0$ $2 \leftarrow$ MSD

$$\boxed{156_{10} = 234_8}$$

B. $32_{10} = 40_8$
C. $1785_{10} = 3371_8$

16. A. $0.1432 \times 8 = 1.1456 = 0.1456$ overflow $1 \leftarrow$ MSD
$0.1456 \times 8 = 1.1648 = 0.1648$ 1
$0.1648 \times 8 = 1.3184 = 0.3184$ 1
$0.3184 \times 8 = 2.5472 = 0.5472$ $2 \leftarrow$ LSD

$$\boxed{0.1432_{10} = 0.1112_8}$$

B. $0.8125_{10} = 0.64_8$
C. $0.6832_{10} = 0.5356_8$

17. $735.984375_{10} = 735_{10} + 0.984375_{10}$
$= 735 \div 8 = 91$ with remainder $7 \leftarrow$ LSD
$91 \div 8 = 11$ 3
$11 \div 8 = 1$ 3
$1 \div 8 = 0$ $1 \leftarrow$ MSD

$$\boxed{735_{10} = 1337_8}$$

$0.984375 \times 8 = 7.875 = 0.875$ overflow
$7 \leftarrow$ MSD
$0.875000 \times 8 = 7.00 = 0$ $7 \leftarrow$ LSD

$$\boxed{0.984375_{10} = 0.77_8}$$

$735.984375_{10} = 735_{10} + 0.984375_{10} = 1337_8 + 0.77_8 = 1337.77_8$

18. A. $10000111.01101_2 = 010\ 000\ 111.011\ 010_2$
$= 207.32_8$
B. $11101.0101_2 = 35.24_8$
C. $1001101.000001_2 = 115.01_8$

19. A. $372.61_8 = 011\ 111\ 010.110\ 001_2 = 11111010.110001_2$.
B. $11.001_8 = 1001.000000001_2$
C. $3251.034_8 = 11010101001.0000111_2$

HEXADECIMAL NUMBER SYSTEM

Hexadecimal is another number system that is often used with microprocessors. It is similar in value structure to the octal number system, and thus allows easy conversion with the binary number system. Because of this feature and the fact that hexadecimal simplifies data entry and display to a greater degree than octal, you will use hexadecimal more often than any other number system in this course. As the name implies, hexadecimal has a base (radix) of 16_{10}. It uses the digits 0 through 9 and the letters A through F.

The letters are used because it is necessary to represent 16_{10} different values with a single digit for each value. Therefore, the letters A through F are used to represent the number values 10_{10} through 15_{10}. The following discussion will compare the decimal number system with the hexadecimal number system.

All of the numbers are of equal value between systems ($0_{10} = 0_{16}$, $3_{10} = 3_{16}$, $9_{10} = 9_{16}$, etc.). For numbers greater than 9, this relationship exists: $10_{10} = A_{16}$, $11_{10} = B_{16}$, $12_{10} = C_{16}$, $13_{10} = D_{16}$, $14_{10} = E_{16}$, and $15_{10} = F_{16}$. Using letters in counting may appear awkward until you become familiar with the system. Figure 1-8 illustrates the relationship between decimal and hexadecimal integers, while Figure 1-9 illustrates the relationship between decimal and hexadecimal fractions.

DECIMAL	HEXADECIMAL	BINARY
0	0	0
1	1	1
2	2	10
3	3	11
4	4	100
5	5	101
6	6	110
7	7	111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111
16	10	10000
17	11	10001
18	12	10010
19	13	10011
20	14	10100
21	15	10101
22	16	10110
23	17	10111
24	18	11000
25	19	11001
26	1A	11010
27	1B	11011
28	1C	11100
29	1D	11101
30	1E	11110
31	1F	11111
32	20	100000
33	21	100001
34	22	100010
35	23	100011

Figure 1-8
Sample comparison of decimal, hexadecimal, and binary integers.

DECIMAL	HEXADECIMAL	BINARY
0.00390625	0.01	0.00000001
0.0078125	0.02	0.0000001
0.01171875	0.03	0.00000011
0.015625	0.04	0.000001
0.01953125	0.05	0.00000101
0.0234375	0.06	0.0000011
0.02734375	0.07	0.00000111
0.03125	0.08	0.00001
0.03515625	0.09	0.00001001
0.0390625	0.0A	0.0000101
0.04296875	0.0B	0.00001011
0.046875	0.0C	0.00011
0.05078125	0.0D	0.00001101
0.0546875	0.0E	0.0000111
0.05859375	0.0F	0.00001111
0.0625	0.1	0.0001
0.06640625	0.11	0.00010001
0.0703125	0.12	0.0001001
0.07421875	0.13	0.00010011
0.078125	0.14	0.000101
0.08203125	0.15	0.00010101
0.0859375	0.16	0.0001011
0.08984375	0.17	0.00010111
0.09375	0.18	0.00011
0.09765625	0.19	0.00011001
0.1015625	0.1A	0.0001101
0.10546875	0.1B	0.00011011
0.109375	0.1C	0.000111
0.11328125	0.1D	0.00011101
0.1171875	0.1E	0.0001111
0.12109375	0.1F	0.00011111
0.125	0.2	0.001

Figure 1-9
Sample comparison of decimal, hexadecimal, and binary fractions.

As with the previous number systems, each digit position of a hexadecimal number carries a positional weight which determines the magnitude of that number. The weight of each position is determined by some power of the number system base (in this example, 16_{10}). The total quantity of a number can be evaluated by considering the specific digits and the weights of their positions. (Refer to Figure 1-10 for a condensed listing of powers of 16_{10}.) For example, the hexadecimal number E5D7.A3 can be written with positional notation as follows:

$$(E \times 16^3) + (5 \times 16^2) + (D \times 16^1) + (7 \times 16^0) + (A \times 16^{-1}) + (3 \times 16^{-2})$$

The decimal value of the hexadecimal number E5D7.A3 is determined by multiplying each digit by its positional weight and adding the results. As with the previous number systems, the radix (hexadecimal) point separates the integer from the fractional part of the number.

$$(14 \times 4096) + (5 \times 256) + (13 \times 16) + (7 \times 1) + (10 \times 1/16) + (3 \times 1/256) =$$
$$57344 + 1280 + 208 + 7 + 0.625 + 0.01171875 =$$
$$58839.63671875_{10}$$

$$16^{-4} = \frac{1}{65536} = 0.0000152587890625_{10}$$
$$16^{-3} = \frac{1}{4096} = 0.000244140625_{10}$$
$$16^{-2} = \frac{1}{256} = 0.00390625_{10}$$
$$16^{-1} = \frac{1}{16} = 0.0625_{10}$$

$$1_{10} = 16^0$$
$$16_{10} = 16^1$$
$$256_{10} = 16^2$$
$$4096_{10} = 16^3$$
$$65536_{10} = 16^4$$
$$1048576_{10} = 16^5$$
$$16777216_{10} = 16^6$$

Figure 1-10
Condensed listing of powers of 16.

Conversion From Decimal to Hexadecimal

Decimal to hexadecimal conversion is accomplished in the same manner as decimal to binary or octal, but with a base number of 16_{10}. As an example, the decimal number 156 is converted into its hexadecimal equivalent.

```
156 ÷ 16 = 9   with remainder 12 = C  ←  LSD
  9 ÷ 16 = 0                   9 = 9  ←  MSD
```

Divide the decimal number by 16_{10} and note the remainder. If the remainder exceeds 9, convert the 2-digit number to its hexadecimal equivalent (12_{10} = C in this example). Then divide the quotient by 16 and again note the remainder. Continue dividing until a quotient of 0 results. Then collect the remainders beginning with the last or most significant digit (MSD) and proceed to the first or least significant digit (LSD). The number $9C_{16} = 156_{10}$. NOTE: The letter H after a number is sometimes used to indicate hexadecimal. However, this course will always use the subscript 16.

To further illustrate this, the decimal number 47632 is converted into its hexadecimal equivalent.

```
47632 ÷ 16 = 2977   with remainder  0 = 0  ←  LSD
 2977 ÷ 16 = 186                    1 = 1
  186 ÷ 16 = 11                    10 = A
   11 ÷ 16 = 0                     11 = B  ←  MSD
```

The division process continues until a quotient of 0 results. The remainders are collected, producing the number $BA10_{16} = 47632_{10}$. Remember, any remainder that exceeds the digit 9 must be converted to its letter equivalent. (In this example, 10 = A, and 11 = B.)

To convert a decimal fraction to a hexadecimal fraction, multiply the fraction successively by 16_{10} (hexadecimal base). As an example the decimal fraction 0.78125 is converted into its hexadecimal equivalent.

```
0.78125 × 16 = 12.5 = 0.5  with overflow  12 = C  ←  MSD
0.50000 × 16 =  8.0 = 0                    8 = 8  ←  LSD
```

Multiply the decimal by 16_{10}. If the product exceeds one, subtract the integer (overflow) from the product. If the "overflow" exceeds 9, convert the 2-digit number to its hexadecimal equivalent. Then multiply the product fraction by 16_{10} and again note any overflow. Continue multiplying until an overflow, with 0 for a fraction, results. Remember, you can not always obtain 0 when you multiply by 16. Therefore, you should only continue the conversion to the accuracy or precision you desire. Collect the conversion overflows beginning at the radix point with the MSD and proceed to the LSD.The number $0.C8_{16} = 0.78125_{10}$.

Now the decimal fraction 0.136 will be converted into its hexadecimal equivalent with five-place precision.

0.136 × 16 = 2.176 = 0.176	overflow	2 = 2	→ MSD
0.176 × 16 = 2.816 = 0.816		2 = 2	
0.816 × 16 = 13.056 = 0.056		13 = D	
0.056 × 16 = 0.896 = 0.896		0 = 0	
0.896 × 16 = 14.336 = 0.336		14 = E	→ LSD

The number $0.22D0E_{16}$ approximately equals 0.136_{10}. If you convert $0.22D0E_{16}$ back to decimal (using positional notation), you will find $0.22D0E_{16} = 0.1359996795654296875_{16}$. This example shows that extending the precision of your conversion is of little value unless extreme accuracy is required.

As shown in this section, conversion of an integer from decimal to hexadecimal requires a different technique than for conversion of a fraction. Therefore, when you convert a hexadecimal number composed of an integer and a fraction, you must separate the integer and fraction, then perform the appropriate operation on each. After you convert them, you must recombine the integer and fraction. For example, the decimal number 124.78125 is converted into its hexadecimal equivalent.

$124.78125_{10} = 124_{10} + 0.78125_{10}$

$124 \div 16 = 7$ with remainder $12 = C \leftarrow$ LSD

$7 \div 16 = 0$ $\quad 7 = 7 \leftarrow$ MSD

$\boxed{124_{10} = 7C_{16}}$

$0.78125 \times 16 = 12.5 = 0.5$ overflow $12 = C \leftarrow$ MSD

$0.50000 \times 16 = 8.0 = 0$ $\quad 8 = 8 \leftarrow$ LSD

$\boxed{0.78125_{10} = 0.C8_{16}}$

$124.78125_{10} = 124_{10} + 0.78125_{10} = 7C_{16} + 0.C8_{16} = 7C.C8_{16}$

First separate the decimal integer and fraction. Then convert the integer and fraction to hexadecimal.

Finally, recombine the integer and fraction.

Converting Between the Hexadecimal and Binary Number Systems

Previously, the octal number system was described as an excellent shorthand form to express large binary quantities. This method is very useful with many microprocessors. The trainer used with this course uses the hexadecimal number system to represent binary quantities. As a result, frequent conversions from binary-to-hexadecimal are necessary. Figures 1-8 and 1-9 illustrate the relationship between hexadecimal and binary integers and fractions.

As you know, four bits of a binary number exactly equal 16_{10} value combinations. Therefore, you can represent a 4-bit binary number with a 1-digit hexadecimal number:

$$1101_2 = (1 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) = 8 + 4 + 0 + 1 = 13_{10} = D_{16}$$

Because of this relationship, converting binary to hexadecimal is simple and straightforward. For example, binary number 10110110 is converted into its hexadecimal equivalent.

10110110_2

REWRITE AS

MSB ↘ 1011 0110 ↙ LSB

YIELDS

$B6_{16}$

To convert a binary number to hexadecimal, first separate the number into groups containing four bits, beginning with the least significant bit. Then convert each 4-bit group into its hexadecimal equivalent. Don't forget to use letter digits as required. This gives you a hexadecimal number equal in value to the binary number.

Now convert a larger binary number (10101101101) into its hexadecimal equivalent.

10101101101_2

REWRITE AS

MSB ↘ $\underline{0}101$ 0110 1101 ↙ LSB

YIELDS

$56D_{16}$

Again, the binary number is separated into 4-bit groups beginning with the LSB. However, the third group contains only three bits. Since each group must contain four bits, a zero must be added after the MSB. The third group will then have four bits with no change in the value of the binary number. Now each 4-bit group can be converted into its hexadecimal equivalent. **Whenever you add zeros to a binary integer, always place them to the left of the most significant bit.**

Binary fractions can also be converted to their hexadecimal equivalents using the same process, with one exception; the binary bits are separated into groups of four, beginning with the most significant bit (at the radix point). For example, the binary fraction 0.01011011 is converted into its hexadecimal equivalent.

$$\begin{tikzcd}[row sep=small, column sep=small]
 & 0.01011011_2 & \\
 & \text{REWRITE AS} & \\
\text{MSB} \arrow[dr] & & \text{LSB} \arrow[dl] \\
 & 0.0101 \quad 1011 & \\
 & \text{YIELDS} & \\
 & 0.5B_{16} &
\end{tikzcd}$$

Again, you must separate the binary number into groups of four, beginning with the radix point. Then convert each 4-bit group into its hexadecimal equivalent. This gives you a hexadecimal number equal in value to the binary number.

Now convert a larger binary fraction (0.1101001101) into its hexadecimal equivalent.

$$\begin{tikzcd}[row sep=small, column sep=small]
 & 0.1101001101_2 & \\
 & \text{REWRITE AS} & \\
\text{MSB} \arrow[dr] & & \text{LSB} \arrow[dl] \\
 & 0.1101 \quad 0011 \quad 01\underline{00} & \\
 & \text{YIELDS} & \\
 & 0.D34_{16} &
\end{tikzcd}$$

Separate the binary number into 4-bit groups, beginning at the radix (binary) point (MSB). Note that the third group contains only two bits. Since each group must contain four bits, two zeros must be added after the LSB. The third group will then have four bits with no change in the value of the binary number. Now, each 4-bit group can be converted into its hexadecimal equivalent. **Whenever you add zeros to a binary fraction, always place them to the right of the least significant bit.**

Now, a binary number containing both an integer and a fraction (110110101.01110111) will be converted into its hexadecimal equivalent.

110110101.01110111_2

REWRITE AS

MSB LSB

$\underline{000}1$ 1011 0101.0111 0111

YIELDS

$1B5.77_{16}$

The integer part of the number is separated into groups of four, **beginning** at the radix point. Note that three zeros were added to the third group to complete the group. The fractional part of the number is separated into groups of four, **beginning** at the radix point. (No zeros were needed to complete the fractional groups.) The integer and fractional 4-bit groups are then converted to hexadecimal. The number $110110101.01110111_2 = 1B5.77_{16}$. **Never** shift the radix point in order to form 4-bit groups.

Converting hexadecimal to binary is just the opposite of the previous process; simply convert each hexadecimal number into its 4-bit binary equivalent. For example, convert the hexadecimal number 8F.41 into its binary equivalent.

$8F.41_{16}$

YIELDS

MSB LSB

1000 1111.0100 0001

REWRITE AS

10001111.01000001_2

Convert each hexadecimal digit into a 4-bit binary number. Then condense the 4-bit groups to form the binary value equal to the hexadecimal value. The number $8F.41_{16} = 10001111.01000001_2$.

Now, the hexadecimal number 175.4E will be converted into its binary equivalent.

$$175.4E_{16}$$

YIELDS

$$\underline{000}\overset{\overset{\text{MSB}}{\swarrow}}{1}\ 0111\ 0101.0100\ 11\overset{\overset{\text{LSB}}{\searrow}}{1}\underline{0}$$

REWRITE AS

$$101110101.0100111_2$$

Again, each hexadecimal digit is converted into its 4-bit binary equivalent. However, in this example there are three insignificant zeros in front of the MSB and one after the LSB. Since these zeros have no value, they should be removed from the final result.

Self-Test Review

20. The base or radix of the hexadecimal number system is _______________.

21. Convert the following decimal integers to hexadecimal.

    A. 783
    B. 5372
    C. 957

22. Convert the following decimal fractions to hexadecimal. Do not use greater than four-place precision.

    A. 0.653
    B. 0.109375
    C. 0.4567

23. Convert 1573.125_{10} to its hexadecimal equivalent.

24. Convert the following binary numbers to hexadecimal

    A. 100001101.01011
    B. 11111011001.01
    C. 110001101.00010010101

25. Convert the following hexadecimal numbers to binary.

    A. AE7.D2
    B. 2C5.21F8
    C. 1B6.64E

Answers

20. 16_{10}.

21. A. $783 \div 16 = 48$ with remainder $15 = F \leftarrow$ LSD
$48 \div 16 = 3$ $0 = 0$
$3 \div 16 = 0$ $3 = 3 \leftarrow$ MSD

$\boxed{783_{10} = 30F_{16}}$

B. $5372_{10} = 14FC_{16}$
C. $957_{10} = 3BD_{16}$

22. A. $0.653 \times 16 = 10.448 = 0.448$ with overflow $10 = A \leftarrow$ MSD
$0.448 \times 16 = 7.168 = 0.168$ $7 = 7$
$0.168 \times 16 = 2.688 = 0.688$ $2 = 2$
$0.688 \times 16 = 11.008 = 0.008$ $11 = B \leftarrow$ LSD

$\boxed{0.653_{10} = 0.A72B_{16}}$

B. $0.109375_{10} = 0.1C_{16}$
C. $0.4567_{10} = 0.74EA_{16}$

23. A. $1573.125_{10} = 1573_{10} + 0.125_{10}$
$1573 \div 16 = 98$ with remainder $5 = 5 \leftarrow$ LSD
$98 \div 16 = 6$ $2 = 2$
$6 \div 16 = 0$ $6 = 6 \leftarrow$ MSD

$\boxed{1573_{10} = 625_{16}}$

$0.125 \times 16 = 2.00 = 0$ with overflow $2 = 2 \leftarrow$ MSD $\leftarrow$ LSD

$\boxed{0.125_{10} = 0.2_{16}}$

$1573.125_{10} = 1573_{10} + 0.125_{10} = 625_{16} + 0.2_{16} = 625.2_{16}$

24. A. $100001101.01011_2 = 0001\ 0000\ 1101.0101\ 1000_2$
$= 10D.58_{16}$

B. $11111011001.01_2 = 7D9.4_{16}$
C. $110001101.00010010101_2 = 18D.12A_{16}$

25. A. $AE7.D2_{16} = 1010\ 1110\ 0111.1101\ 0010_2$
$= 101011100111.1101001_2$
B. $2C5.21F8_{16} = 1011000101.0010000111111_2$
C. $1B6.64E_{16} = 110110110.01100100111_2$

BINARY CODES

Converting a decimal number into its binary equivalent is called "coding." A decimal number is expressed as a binary code or binary number. The **binary number system**, as discussed, is known as the pure binary code. This name distinguishes it from other types of binary codes. This section will discuss some of the other types of binary codes used in computers.

Binary Coded Decimal

The decimal number system is easy to use because it is so familiar. The binary number system is less convenient to use because it is less familiar. It is difficult to quickly glance at a binary number and recognize its decimal equivalent. For example, the binary number 1010011 represents the decimal number 83. It is difficult to tell immediately by looking at the number what its decimal value is. However, within a few minutes, using the procedures described earlier, you could readily calculate its decimal value. The amount of time it takes to convert or recognize a binary number quantity is a distinct disadvantage in working with this code despite the numerous hardware advantages. Engineers recognized this problem early and developed a special form of binary code that was more compatible with the decimal system. Because so many digital devices, instruments and equipment use decimal input and output, this special code has become very widely used and accepted. This special compromise code is known as binary coded decimal (BCD). The BCD code combines some of the characteristics of both the binary and decimal number systems.

8421 BCD Code The BCD code is a system of representing the decimal digits 0 through 9 with a four-bit binary code. This BCD code uses the standard 8421 position **weighting system** of the pure binary code. The standard 8421 BCD code and the decimal equivalents are shown in Figure 1-11, along with a special Gray code that will be described later. As with the pure binary code, you can convert the BCD numbers into their decimal equivalents by simply adding together the weights of the bit positions whereby the binary 1's occur. Note, however, that there are only ten possible valid 4-bit code arrangements. The 4-bit binary numbers representing the decimal numbers 10 through 15 are invalid in the BCD system.

DECIMAL	8421 BCD	GRAY	BINARY
0	0000	0000	0000
1	0001	0001	0001
2	0010	0011	0010
3	0011	0010	0011
4	0100	0110	0100
5	0101	0111	0101
6	0110	0101	0110
7	0111	0100	0111
8	1000	1100	1000
9	1001	1101	1001
10	0001 0000	1111	1010
11	0001 0001	1110	1011
12	0001 0010	1010	1100
13	0001 0011	1011	1101
14	0001 0100	1001	1110
15	0001 0101	1000	1111

Figure 1-11
Codes.

To represent a decimal number in BCD notation, substitute the appropriate 4-bit code for each decimal digit. For example, the decimal integer 834 in BCD would be 1000 0011 0100. Each decimal digit is represented by its equivalent 8421 4-bit code. A space is left between each 4-bit group to avoid confusing the BCD format with the pure binary code. This method of representation also applies to decimal fractions. For example, the decimal fraction 0.764 would be 0.0111 0110 0100 in BCD. Again, each decimal digit is represented by its equivalent 8421 4-bit code, with a space between each group.

An advantage of the BCD code is that the ten BCD code combinations are easy to remember. Once you begin to work with binary numbers regularly, the BCD numbers may come to you as quickly and automatically as decimal numbers. For that reason, by simply glancing at the BCD representation of a decimal number you can make the conversion almost as quickly as if it were already in decimal form. As an example, convert a BCD number into its decimal equivalent.

$$0110\ 0010\ 1000.1001\ 0101\ 0100 = 628.954_{10}$$

The BCD code simplifies the man-machine interface but it is less efficient than the pure binary code. It takes more bits to represent a given decimal number in BCD than it does with pure binary notation. For example, the decimal number 83 in pure binary form is 1010011. In BCD code the decimal number 83 is written as 1000 0011. In the pure binary code, it takes only seven bits to represent the number 83. In BCD form, it takes eight bits. It is inefficient because, for each bit in a data word, there is usually some digital circuitry associated with it. The extra circuitry associated with the BCD code costs more, increases equipment complexity, and consumes more power. Arithmetic operations with BCD numbers are also more time consuming and complex than those with pure binary numbers. With four bits of binary information, you can represent a total of $2^4 = 16$ different states or the decimal number equivalents 0 through 15. In the BCD system, six of these states (10-15), are wasted. When the BCD number system is used, some efficiency is traded for the improved communications between the digital equipment and the human operator.

Decimal-to-BCD conversion is simple and straightforward. However, binary-to-BCD conversion is not direct. An intermediate conversion to decimal must be performed first. For example, the binary number 1011.01 is converted into its BCD equivalent.

First the binary number is converted to decimal.

$$\begin{aligned} 1011.01_2 &= (1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0) + (0 \times 2^{-1}) + (1 \times 2^{-2}) \\ &= 8 + 0 + 2 + 1 + 0 + 0.25 \\ &= 11.25_{10} \end{aligned}$$

Then the decimal result is converted to BCD.

$11.25_{10} = 0001\ 0001.0010\ 0101$

To convert from BCD to binary, the previous operation is reversed. For example, the BCD number 1001 0110.0110 0010 0101 is converted into its binary equivalent.

First, the BCD number is converted to decimal.
1001 0110.0110 0010 0101 = 96.625_{10}
Then the decimal result is converted to binary.
$96.625_{10} = 96_{10} + 0.625_{10}$

```
96 ÷ 2 = 48  with remainder 0 ←— LSB
48 ÷ 2 = 24                 0
24 ÷ 2 = 12                 0
12 ÷ 2 = 6                  0
 6 ÷ 2 =3                   0
 3 ÷ 2 = 1                  1
 1 ÷ 2 = 0                  1 ←— MSB
```

$\boxed{96_{10} = 1100000_2}$

```
0.625 × 2 = 1.25 = 0.25    with overflow    1 ←— MSB
0.250 × 2 = 0.50 = 0.50                     0
0.500 × 2 = 1.00 = 0                        1 ←— LSB
```

$\boxed{0.625_{10} = 0.101_2}$

$96.625_{10} = 96_{10} + 0.625_{10} = 1100000_2 + 0.101_2 = 1100000.101_2$

Therefore:

1001 0110.0110 0010 0101 = $96.625_{10} = 1100000.101_2$

Because the intermediate decimal number contains both an integer and fraction, each number portion is converted as described under "Binary Number System." The binary sum (integer plus fraction) 1100000.101 is equivalent to the BCD number 1001 0110.0110 0010 0101.

DECIMAL	'8421 BCD	GRAY	BINARY
0	0000	0000	0000
1	0001	0001	0001
2	0010	0011	0010
3	0011	0010	0011
4	0100	0110	0100
5	0101	0111	0101
6	0110	0101	0110
7	0111	0100	0111
8	1000	1100	1000
9	1001	1101	1001
10	0001 0000	1111	1010
11	0001 0001	1110	1011
12	0001 0010	1010	1100
13	0001 0011	1011	1101
14	0001 0100	1001	1110
15	0001 0101	1000	1111

Figure 1-11
Codes.

Special Binary Codes

Besides the standard pure binary coded form, the BCD numbering system is by far the most widely-used digital code. You will find one or the other in most of the applications that you encounter. However, there are several other codes that are used for special applications, such as the "Gray Code."

The Gray Code is a widely-used, non-weighted code system. Also known as the cyclic, unit distance or reflective code, the Gray code can exist in either the pure binary or BCD formats. The Gray code is shown in Figure 1-11. As with the pure binary code, the first ten codes are used in BCD operations. Notice that there is a change in only one bit from one code number to the next in sequence. You can get a better idea about the Gray code sequence by comparing it to the standard 4-bit 8421 BCD code and the pure binary code also shown in Figure 1-11. For example, consider the change from 7 (0111) to 8 (1000) in the pure binary code. When this change takes place, all bits change. Bits that were 1's are changed to 0's and 0's are changed to 1's. Now notice the code change from 7 to 8 in the Gray code. Here 7 (0100) changes to 8 (1100). Only the first bit changes.

The Gray code is generally known as an error minimizing code because it greatly reduces confusion in the electronic circuitry when changing from one state to the next. When binary codes are implemented with electronic circuitry, it takes a finite period of time for bits to change from 0 to 1 or 1 to 0. These state changes can create timing and speed problems. This is particularly true in the standard 8421 codes where many bits change from one combination to the next. When the Gray code is used, however, the timing and speed errors are greatly minimized because only one bit changes at a time. This permits code circuitry to operate at higher speeds with fewer errors.

The biggest disadvantage of the Gray code is that it is difficult to use in arithmetic computations. Where numbers must be added, subtracted or used in other computations, the Gray code is not applicable. In order to perform arithmetic operations, the Gray code number must generally be converted into pure binary form.

Alphanumeric Codes

Several binary codes are called alphanumeric codes because they are used to represent characters as well as numbers. The two most common codes that will be discussed are ASCII and BAUDOT.

ASCII Code The American Standard Code for Information Interchange commonly referred to as ASCII, is a special form of binary code that is widely used in microprocessors and data communications equipment. A new name for this code that is becoming more popular is the American National Standard Code for Information Interchange (ANSCII). However, this course will use the most recognized term, ASCII. ASCII is a 6-bit binary code that is used in transferring data between microprocessors and their peripheral devices, and in communicating data by radio and telephone. With six bits, a total of $2^6 = 64$ different characters can be represented. These characters comprise decimal numbers 0 through 9, upper-case letters of the alphabet, plus other special characters used for punctuation and data control. A 7-bit code called full ASCII, extended ASCII, or USASCII can be represented by $2^7 = 128$ different characters. In addition to the characters and numbers generated by 6-bit ASCII, 7-bit ASCII contains lower-case letters of the alphabet, and additional characters for punctuation and control. The 7-bit ASCII code is shown in Figure 1-12.

	COLUMN	0 (3)	1 (3)	2 (3)	3	4	5	6	7 (3)
ROW	BITS 4321 ↓ 765 ⇨	000	001	010	011	100	101	110	111
0	0000	NUL	DLE	SP	0	@	P	`	p
1	0001	SOH	DC1	!	1	A	Q	a	q
2	0010	STX	DC2	"	2	B	R	b	r
3	0011	ETX	DC3	#	3	C	S	c	s
4	0100	EOT	DC4	$	4	D	T	d	t
5	0101	ENQ	NAK	%	5	E	U	e	u
6	0110	ACK	SYN	&	6	F	V	f	v
7	0111	BEL	ETB	'	7	G	W	g	w
8	1000	BS	CAN	(	8	H	X	h	x
9	1001	HT	EM	)	9	I	Y	i	y
10	1010	LF	SUB	*	:	J	Z	j	z
11	1011	VT	ESC	+	;	K	[	k	{
12	1100	FF	FS	,	<	L	\	l	¦
13	1101	CR	GS	–	=	M	]	m	}
14	1110	SO	RS	.	>	N	^ (1)	n	~
15	1111	SI	US	/	?	O	— (2)	o	DEL

Figure 1-12
Table of 7-bit American Standard Code for Information Interchange.

NOTES:

(1) Depending on the machine using this code, the symbol may be a circumflex, an up-arrow, or a horizontal parenthetical mark.

(2) Depending on the machine using this code, the symbol may be an underline, a back-arrow, or a heart.

(3) Explanation of special control functions in columns 0, 1, 2, and 7.

NUL	Null	DLE	Data Link Escape
SOH	Start of Heading	DC1	Device Control 1
STX	Start of Text	DC2	Device Control 2
ETX	End of Text	DC3	Device Control 3
EOT	End of Transmission	DC4	Device Control 4
ENQ	Enquiry	NAK	Negative Acknowledge
ACK	Acknowledge	SYN	Synchronous Idle
BEL	Bell (audible signal)	ETB	End of Transmission Block
BS	Backspace	CAN	Cancel
HT	Horizontal Tabulation (punched card skip)	EM	End of Medium
		SUB	Substitute
LF	Line Feed	ESC	Escape
VT	Vertical Tabulation	FS	File Separator
FF	Form Feed	GS	Group Separator
CR	Carriage Return	RS	Record Separator
SO	Shift Out	US	Unit Separator
SI	Shift In	DEL	Delete
SP	Space (blank)		

Figure 1-12
(Continued.)

The 7-bit ASCII code for each number, letter or control function is made up of a 4-bit group and a 3-bit group. Figure 1-13 shows the arrangement of these two groups and the numbering sequence. The 4-bit group is on the right and bit 1 is the LSB. Note how these groups are arranged in rows and columns in Figure 1-12.

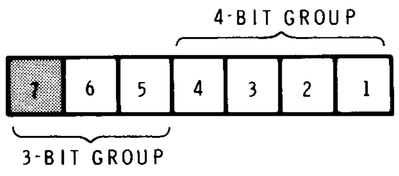


Figure 1-13
ASCII code word format.

To determine the ASCII code for a given number letter or control operation, locate that item in the table. Then use the 3- and 4-bit codes associated with the row and column in which the item is located. For example, the ASCII code for the letter L is 1001100. It is located in column 4, row 12. The most significant 3-bit group is 100, while the least significant 4-bit group is 1100. When 6-bit ASCII is used, the 3-bit group is reduced to a 2-bit group as shown in Figure 1-14.

In 7-bit ASCII code, an eighth bit is often used as a **parity** or check bit to determine if the data (character) has been transmitted correctly. The value of this bit is determined by the type of parity desired. **Even parity** means the sum of all the 1 bits, including the parity bit, is an even number. For example, if G is the character transmitted, the ASCII code is 1000111. Since four 1's are in the code, the parity bit is 0. The 8-bit code would be written 01000111.

Odd Parity means the sum of all the 1 bits, including the parity bit, is an odd number. If the ASCII code for G was transmitted with odd parity, the binary representation would be 11000111.

ROW	COLUMN	0	1	2	3
	BITS 4321 ⇩ 65 ⇨	10	11	00	01
0	0000	SP(3)	0	@	P
1	0001	!	1	A	Q
2	0010	"	2	B	R
3	0011	#	3	C	S
4	0100	$	4	D	T
5	0101	%	5	E	U
6	0110	&	6	F	V
7	0111	'	7	G	W
8	1000	(	8	H	X
9	1001	)	9	I	Y
10	1010	*	:	J	Z
11	1011	+	;	K	[
12	1100	,	<	L	\
13	1101	–	=	M	]
14	1110	.	>	N	^ (1)
15	1111	/	?	O	— (2)

Figure 1-14
Table of 6-bit American Standard Code for Information Interchange.

NOTES:

(1) Depending on the machine using this code, the symbol may be a circumflex, an up-arrow, or a horizontal parenthetical mark.

(2) Depending on the machine using this code, the symbol may be an underline, a back-arrow, or a heart.

(3) SP — Space (blank) for machine control.

BAUDOT Code While the ASCII code is used almost exclusively with microprocessor peripheral devices (CRT display, keyboard terminal, paper punch/reader, etc.), there are many older printer peripherals that use the 5-bit BAUDOT code. With five data bits, this code can represent only $2^5 = 32$ different characters. To obtain a greater character capability, 26 of the 5-bit codes are used to represent two separate characters. As shown in Figure 1-15, one set of 5-bit codes represents the 26 upper-case alphabet letters. The same 5-bit codes also represent various figures and the decimal number series 0 through 9.

The remaining six 5-bit codes are used for machine control and do not have a secondary function. Two of these 5-bit codes determine which of the 26 double (letter/figure) characters can be transmitted/received. Bit number 11111 forces the printer to recognize all following 5-bit codes as **letters**. Bit number 11011 forces **figure** recognition of all the following 5-bit codes. For example, to type 56 NORTH 10 STREET, the following method is used.

Type — Figures 5 6 Space

Then — Letters N O R T H Space

Then — Figures 1 0 Space

Finally — Letters S T R E E T

Bit Numbers 5 4 3 2 1	Letters Case	Figures Case
0 0 0 0 0	Blank	Blank
0 0 0 0 1	E	3
0 0 0 1 0	Line Feed	Line Feed
0 0 0 1 1	A	—
0 0 1 0 0	Space	Space
0 0 1 0 1	S	Bell
0 0 1 1 0	I	8
0 0 1 1 1	U	7
0 1 0 0 0	Car. Ret.	Car. Ret.
0 1 0 0 1	D	$
0 1 0 1 0	R	4
0 1 0 1 1	J	(Apos)'
0 1 1 0 0	N	(Comma),
0 1 1 0 1	F	!
0 1 1 1 0	C	:
0 1 1 1 1	K	(
1 0 0 0 0	T	5
1 0 0 0 1	Z	"
1 0 0 1 0	L	)
1 0 0 1 1	W	2
1 0 1 0 0	H	Stop
1 0 1 0 1	Y	6
1 0 1 1 0	P	0
1 0 1 1 1	Q	1
1 1 0 0 0	O	9
1 1 0 0 1	B	?
1 1 0 1 0	G	&
1 1 0 1 1	Figures	Figures
1 1 1 0 0	M	.
1 1 1 0 1	X	/
1 1 1 1 0	V	;
1 1 1 1 1	Letters	Letters

Figure 1-15
5-bit BAUDOT code table.

Self-Test Review

26. The BCD code is more convenient to use than the binary code because:

    A. it uses less bits.
    B. it is more compatible with the decimal number system.
    C. it is more adaptable to arithmetic computations.
    D. there are more different coding schemes available.

27. Convert the following decimal numbers to 8421 BCD code.

    A. 1049
    B. 267
    C. 835

28. Convert the following 8421 BCD code numbers to decimal.

    A. 1001 0110 0010
    B. 0111 0001 0100 0011
    C. 1010 1001 1000
    D. 1000 0000 0101

29. Convert the following binary numbers to 8421 BCD code.

    A. 101110.01
    B. 1001.0101
    C. 11011011.0001

30. Convert the following 8421 BCD codes to binary.

    A. 0001 1000 0010.0101
    B. 0010 1001 0000.0010 0101
    C. 1101 0110 0011.0101
    D. 0110 1000.0001 0010 0101

31. Which code is best for error minimizing?

    A. 8421 BCD
    B. pure binary
    C. Gray

32. The ASCII and BAUDOT codes are a form of ______________ ______________ codes.

33. To determine if the correct ASCII character has been transmitted, a ______________ bit is often added to the code.

34. Which type of parity is used when the 8-bit ASCII character 01000111 is transmitted?

    A. odd
    B. even

35. Refer to Figure 1-12 and convert the following characters into their ASCII 7-bit binary code.

    A. B
    B. X
    C. 3
    D. S

36. Refer to Figure 1-12 and convert the following ASCII 7-bit binary codes to their character equivalents.

    A. 0110010
    B. 1010110
    C. 1011010
    D. 1001110

Answers

26. B. More compatible with the decimal system.

27. A. 0001 0000 0100 1001
    B. 0010 0110 0111
    C. 1000 0011 0101

28. A. 962
    B. 7143
    C. Invalid (1010 represents a decimal digit greater than 9)
    D. 805

29. A.

$$\begin{aligned}101110.01_2 &= (1 \times 2^5) + (0 \times 2^4) + (1 \times 2^3) + (1 \times 2^2) + (1 \times 2^1)\\ &\quad + (0 \times 2^0) + (0 \times 2^{-1}) + (1 \times 2^{-2})\\ &= 32 + 0 + 8 + 4 + 2 + 0 + 0 + 0.25\\ &= 46.25_{10}\\ 46.25_{10} &= 0100\ 0110.0010\ 0101\\ 101110.01_2 &= 0100\ 0110.0010\ 0101\end{aligned}$$

B. $1001.0101_2 = 1001.0011\ 0001\ 0010\ 0101$

C. $11011011.0001_2 = 0010\ 0001\ 1001.0000\ 0110\ 0010\ 0101$

30. A. 0001 1000 0010.0101 = 182.5_{10}

$182.5_{10} = 182_{10} + 0.5_{10}$

```
182 ÷ 2 = 91     with remainder   0  ← LSB
 91 ÷ 2 = 45                      1
 45 ÷ 2 = 22                      1
 22 ÷ 2 = 11                      0
 11 ÷ 2 = 5                       1
  5 ÷ 2 = 2                       1
  2 ÷ 2 = 1                       0
  1 ÷ 2 = 0                       1  ← MSB
```

$\boxed{182_{10} = 10110110_2}$

$0.5 \times 2 = 1.00 = 0$ overflow 1 ← MSB ← LSB

$\boxed{0.5_{10} = 0.1_2}$

$182.5_{10} = 182_{10} + 0.5_{10} = 10110110_2 + 0.1_2 = 10110110.1_2$

0001 1000 0010.0101 = 10110110.1_2

B. 0010 1001 0000.0010 0101 = 100100010.01_2
C. invalid (1101 represents a decimal digit greater than 9)
D. 0110 1000.0001 0010 0101 = 1000100.001_2

31. Gray

32. Alpha numeric

33. Parity

34. Even

35. A. 1000010
B. 1011000
C. 0110011
D. 1010011

36. A. 2
B. V
C. Z
D. N

UNIT 1 SUMMARY

1. A basic distinguishing feature of a number system is its base or radix. The binary number system has a radix of 2, octal 8, decimal 10, and hexadecimal 16.

2. A number system is positional or weighted. That is, each digit position carries a particular weight which determines the magnitude of that number. Each position has a weight determined by the power of the number system. In the decimal number system the positional weights are 10^0 (units), 10^1 (tens), 10^2 (hundreds), 10^{-1} (tenths), 10^{-2} (hundredths), etc.

3. In a number, such as decimal 247429, the left-most digit (2) is called the most significant digit or MSD, while the right-most digit (9) is called the least significant digit or LSD. These terms apply to all number systems.

4. The simplest number system is the binary number system. It is called binary because it has a base or radix of 2. The numbers used in the binary number system are 0 and 1.

5. Binary data is represented by binary digits called bits. The left-most bit is generally called the most significant bit (MSB) rather than MSD. By the same token, the right-most bit is called the least significant bit (LSB) rather than LSD.

6. Microprocessors operate on groups of bits which are called words.

7. Positional weight in the binary number system is a positive or negative power of two.

8. To convert a binary number into its decimal equivalent, add together the weights of the positions in the number where the binary ones occur.

9. To convert a decimal whole number into its binary equivalent, successively divide the number by two and note the remainders. When you divide by two, the remainder will always be one or zero. The first remainder is always the LSB, while the last remainder is always the MSB.

10. To convert a decimal fraction into its binary equivalent, successively multiply the fraction by the number two and record the integers produced by the multiplication. The integer produced by the multiplication will always be one or zero. The first integer is the MSB. The MSB is located at the radix (binary) point of the fraction.

11. The octal number system uses the digits 0 through 7.

12. The positional weight in the octal number system is a positive or negative power of eight.

13. To convert an octal number into its decimal equivalent, add together the weights of the number positions.

14. To convert a decimal whole number into its octal equivalent, successively divide the number by eight and note the remainders. These will be some value between zero and seven. The first remainder will always be the MSD.

15. To convert a decimal fraction into its octal equivalent, successively multiply the fraction by the number eight and record the integers produced by the multiplication. The integer produced by the multiplication will always be some number between zero and seven. The first integer is the MSD. The MSD is located at the radix (octal) point of the fraction.

16. Three bits of a binary number exactly equal eight value combinations. Therefore, you can represent a 3-bit binary number with a 1-digit octal number.

17. To convert a binary number into octal, first separate the number into groups containing three bits, beginning with the LSB. Then convert each 3-bit group into its octal equivalent. This applies to both integers and fractions.

18. To convert an octal number into binary, simply convert each octal digit into its 3-bit binary equivalent.

19. The hexadecimal number system uses the digits 0 through 9 and the letters A through F.

20. The positional weight in the hexadecimal number system is a positive or negative power of sixteen.

21. To convert a hexadecimal number into its decimal equivalent, add together the weights of the number positions.

22. To convert a decimal whole number into its hexadecimal equivalent, successively divide the number by sixteen and note the remainders. These will be some value between zero and fifteen. The first remainder will always be the MSD. Any remainder greater than nine must be converted into its hexadecimal letter equivalent.

23. To convert a decimal fraction into its hexadecimal equivalent, successively multiply the fraction by the number sixteen and record the integers produced by the multiplication. The integer produced by the multiplication will always be some number between zero and fifteen. The first integer is the MSD. The MSD is located at the radix (octal) point of the fraction. Any remainder greater than nine must be converted into its hexadecimal letter equivalent.

24. Four bits of a binary number exactly equal sixteen value combinations. Therefore, you can represent a 4-bit binary number with a 1-digit hexadecimal number.

25. To convert a binary number into hexadecimal, first separate the number into groups containing four bits, beginning with the LSB. Then convert each 4-bit group into its hexadecimal equivalent. This applies to both integers and fractions.

26. To convert a hexadecimal number into binary, simply convert each hex digit into its 4-bit binary equivalent.

27. Converting a decimal number into its binary equivalent is called coding.

28. Binary coded decimal (BCD) code combines some of the characteristics of both the binary and decimal number systems. The BCD code is a system of representing the decimal digits zero through nine with a 4-bit binary code. The code uses the 8421 position weighting system of pure binary code. However, there are only ten valid 4-bit code arrangements.

29. The BCD code simplifies the man-machine interface but it is less efficient than the pure binary code. It takes more bits to represent a given decimal number in BCD than it does to represent a given decimal number in pure binary.

30. You cannot make direct conversion between BCD and binary. You must first convert the BCD code to decimal digits, then convert the decimal digits to binary.

31. Gray code is a widely-used, non-weighted code system. It is also known as the cyclic, unit distance, error minimizing, or reflective code.

32. When Gray code is incremented, only one bit changes state. While this permits code circuitry to operate at higher speeds with fewer errors, it is difficult to use in arithmetic computations.

33. American Standard Code for Information Interchange (ASCII) code is a special form of binary code that is widely used in microprocessors and data communications equipment.

34. While ASCII can be used as a 6-bit code, it is generally used in its 7- bit form. The seven code bits are used through a translation table to identify 128 different printable and non-printable characters. The printable characters are the upper-and lower-case alphabet, the numbers zero through nine, and punctuation symbols. The non-printable characters are used to control various "machine" operations such as carriage return, line feed, etc.

35. In 7-bit ASCII code, an eighth bit is often used as a parity or check bit. You use the value of the parity bit to control the parity of the 8-bit word.

36. Older printer peripherals use BAUDOT code for data transmission. It is a 5-bit code that can represent 32 different characters. These characters include the upper-case alphabet, the numbers zero through nine, punctuation symbols, and a number of machine control characters.

Perform Experiments 1 and 2 at the end of Book 2.

POSITIVE POWERS OF 2

n	2^n
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256
9	512
10	1024
11	2048
12	4096
13	8192
14	16384
15	32768
16	65536
17	13107 2
18	26214 4
19	52428 8
20	10485 76
21	20971 52
22	41943 04
23	83886 08
24	16777 216
25	33554 432
26	67108 864
27	13421 7728
28	26843 5456
29	53687 0912
30	10737 41824
31	21474 83648
32	42949 67296

NEGATIVE POWERS OF 2

n	2^{-n}
0	1.0
1	0.5
2	0.25
3	0.125
4	0.0625
5	0.03125
6	0.01562 5
7	0.00781 25
8	0.00390 625
9	0.00195 3125
10	0.00097 65625
11	0.00048 82812 5
12	0.00024 41406 25
13	0.00012 20703 125
14	0.00006 10351 5625
15	0.00003 05175 78125
16	0.00001 52587 89062 5
17	0.00000 76293 94531 25
18	0.00000 38146 97265 625
19	0.00000 19073 48632 8125
20	0.00000 09536 74316 40625
21	0.00000 04768 37158 20312 5
22	0.00000 02384 18579 10156 25
23	0.00000 01192 09289 55078 125
24	0.00000 00596 04644 77539 0625
25	0.00000 00298 02322 38769 53125
26	0.00000 00149 01161 19384 76562 5
27	0.00000 00074 50580 59692 38281 25
28	0.00000 00037 25290 29846 19140 625
29	0.00000 00018 62645 14923 09570 3125
30	0.00000 00009 31322 57461 54785 15625
31	0.00000 00004 65661 28730 77392 57812 5
32	0.00000 00002 32830 64365 38696 28906 25

POSITIVE POWERS OF 8

n	8^n
0	1
1	8
2	64
3	512
4	409 6
5	327 68
6	262 144
7	209 715 2
8	167 772 16

POSITIVE POWERS OF 16

n	16^n
0	1
1	16
2	256
3	409 6
4	655 36
5	104 857 6
6	167 772 16
7	268 435 456
8	429 496 729 6

NEGATIVE POWERS OF 16

n	16^{-n}
0	1.0
1	0.062 5
2	0.003 906 25
3	0.000 244 140 625
4	0.000 015 258 789 062 5

Unit 2

MICROCOMPUTER BASICS

INTRODUCTION

A microprocessor is a very complex electronic circuit. It consists of thousands of microscopic transistors squeezed onto a tiny chip of silicon that is often no more than one-eighth inch square. The chip is placed in a package containing anywhere from 40 to 64 leads.

The thousands of transistors that make up the microprocessor are arranged to form many different circuits within the chip. From the standpoint of learning how the microprocessor operates, the most important circuits on the chip are registers, counters, and decoders. In this unit, you will learn how these circuits can work together to perform simple but useful tasks.

UNIT OBJECTIVES

When you have completed this unit you will be able to:

1. Define the terms: microprocessor, microcomputer, input, output, I/O, I/O device, I/O port, instruction, program, stored program concept, word, byte, MPU, ALU, operand, memory, address, read, write, RAM, fetch, execute, MPU cycle, mnemonic, opcode, and bus.

2. Explain the purpose of the following circuits in a typical microprocessor: accumulator, program counter, instruction decoder, controller sequencer, data register, and address register.

3. Using a simplified block diagram of a hypothetical microprocessor, trace the data flow that takes place between the various circuits during the execution of a simple program.

4. Describe the difference between inherent, immediate, and direct addressing.

5. Write simple, straight-line programs that can be executed by the ET-3400 Microprocessor Trainer.

TERMS AND CONVENTIONS

A **microprocessor** is a logic device that is used in digital electronic systems. It is also being used by hobbyists, experimenters and low-budget research groups as a low-cost, general-purpose computer. But a distinction should be made between the microprocessor and the microcomputer.

The microprocessor unit, or MPU, is a complex logic element that performs arithmetic, logic, and control operations. The trend is to package it as a single integrated circuit.

A **microcomputer** contains a microprocessor, but it also contains other circuits such as memory devices to store information, interface adapters to connect it with the outside world, and a clock to act as a master timer for the system. Figure 2-1 shows a typical microcomputer in which these additional circuits are added. The arrows represent conductors over which binary information flows. The wide arrows represent several conductors connected in parallel. A group of parallel conductors which carry information is called a **bus**.

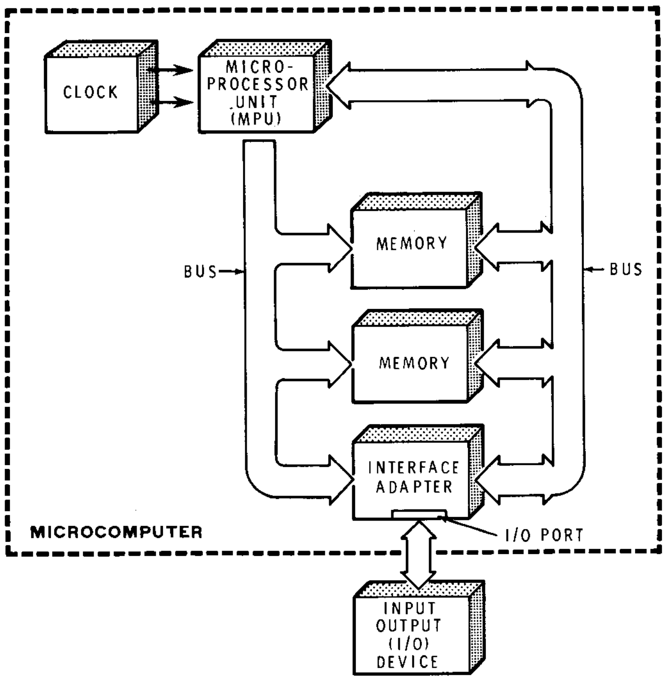


Figure 2-1
A Basic Microcomputer.

The microcomputer is composed of everything inside the dotted line. Everything outside the dotted line is referred to as the **outside world**, and all microcomputers must have some means of communicating with it. Information received by the microcomputer from the outside world is referred to as **input** data. Information transmitted to the outside world from the microcomputer is referred to as **output** data.

Input information may come from devices like paper tape readers, typewriters, mechanical switches, keyboards, or even other computers. Output information may be sent to video displays, output typewriters, paper tape punches, or line printers. Some devices such as the teletypewriter can serve as both an input and an output device. These devices are referred to as **input/output** or **I/O** devices. The point at which the I/O device connects to the microcomputer is called an **I/O port**.

Stored Program Concept

A microcomputer is capable of performing many different operations. It can add and subtract numbers and it can perform logical operations. It can read information from an input device and transmit information to an output device. In fact, depending on the microprocessor used, there may be 100 or more different operations that the microcomputer can perform. Moreover, two or more individual operations can be combined to perform much more complex operations.

In spite of all its capabilities, the computer will do nothing on its own accord. It will do only what it is told to do, nothing more and nothing less. You must tell the computer exactly what operations to perform and the order in which it should perform them. The operations that the computer can be told to perform are called **instructions**. A few typical instructions are ADD, SUBTRACT, LOAD INDEX REGISTER, STORE ACCUMULATOR, and HALT.

A group of instructions that allows the computer to perform a specific job is called a **program**. One who writes these instructions is called a **programmer**. To design with microprocessors, the engineer must become a programmer. To repair microprocessor-based equipment, the technician must understand programming. Generally, the length of the program is proportional to the complexity of the task that the computer is to perform. A program for adding a list of numbers may require only a dozen instructions. On the other hand, a program for controlling all the traffic lights in a small city may require thousands of instructions.

A computer is often compared to a calculator, which is told what to do by the operator via the keyboard. Even inexpensive calculators can perform several operations that can be compared to instructions in the computer. By depressing the right keys, you can instruct the calculator to add, subtract, multiply, divide, and clear the display. Of course, you must also enter the numbers that are to be added, subtracted, etc. With the calculator, you can add a list of numbers as quickly as you can enter the numbers and the instructions. That is, the operation is limited by the speed and accuracy of the operator.

From the start, computer designers recognized that it was the human operator that slowed the computation process. To overcome this, the **stored program concept** was developed. Using this approach, the program is stored in the computer's memory. Suppose, for example, that you have 20 numbers that are to be manipulated by a program that is composed of 100 instructions. Let's further suppose that 10 answers will be produced in the process.

Before any computation begins, the 100-instruction program plus the 20 numbers are loaded into the computer's memory. Furthermore, 10 memory locations are reserved for the 10 answers. Only then is the computer allowed to execute the program. The actual computation time might be less than one millisecond. Compare this to the time that it would take to manually enter the instructions and numbers, one at a time, while the computer is running. This automatic operation is one of the features that distinguishes the computer from the calculator.

Computer Words

In computer terminology, a **word** is a group of binary digits that can occupy a storage location. Although the word may be made up of several binary digits, the computer handles each word as if it were a single unit. Thus, the **word** is the fundamental unit of information used in the computer.

A word may be a binary number that is to be handled as data. Or, the word may be an instruction that tells the computer which operation it is to perform. It may be an ASCII character representing a letter of the alphabet. Finally, a word can be an "address" that tells the computer where a piece of data is located.

Word Length

In the past few years, a wide variety of microprocessors have been developed. Their cost and capabilities vary widely. One of the most important characteristics of any microprocessor is the word length it can handle. This refers to the length in bits of the most fundamental unit of information.

The most common word length for microprocessors is 8 bits. In these units; numbers, addresses, instructions, and data are represented by 8-bit binary numbers. The lowest 8-bit binary number is $0000\ 0000_2$ or 00_{16}. The highest is $1111\ 1111_2$ or FF_{16}. In decimal, this range is from 0 to 255_{10}. Thus, an 8-bit binary number can have any one of 256_{10} unique values.

An 8-bit word can specify positive numbers between 0 and 255_{10}. Or, if the 8-bit word is an instruction, it can specify any of 256_{10} possible operations. It is also entirely possible that the 8-bit word is an ASCII character. In this case, it can represent letters of the alphabet, punctuation marks, or numerals. As you can see, the 8-bit word can represent many different things, depending on how it is interpreted. The programmer must insure that an ASCII character or binary number is not interpreted as an instruction. Later, you will see the consequences of making this mistake.

While the 8-bit word length is the most popular, other word lengths are sometimes used. The earliest microprocessor used a 4-bit word length, and four-bit microprocessors are still used in many cases because of their low cost. A few 12-bit and 16-bit microprocessors have also been developed.

Longer word lengths allow us to work with larger numbers. For example, a 16-bit word can represent numbers up to $65{,}535_{10}$. However, this capability adds to the complexity and cost of the microprocessor. Because most microprocessors use 8-bit word lengths, we will restrict our discussion to units of this type.

It should be pointed out that just because the word length is 8-bits, it does not mean that we are restricted to numbers below 256_{10}. It simply means that you must use two or more words to represent larger numbers.

The 8-bit word length defines the size of many different components in the microprocessor system. For example, many of the important registers will have 8-bit capacity. Memory will be capable of holding a large number of 8-bit words. And, the bus which is used to transfer data words will consist of eight parallel conductors.

Even 16-bit microprocessors use 8-bit segments of data in many applications. For example, inputs from teletypewriters often consist of 8-bit ASCII characters. To distinguish these 8-bit segments of information from the 16-bit (or longer) word lengths, another term has come into general use: the term **byte**. A byte is a group of bits that are handled as a single unit. Generally, a byte is understood to consist of 8-bits. In the 8-bit microprocessor, each word consists of one byte. But in the 16-bit machines, each word contains two bytes. Figure 2-2 illustrates these points.

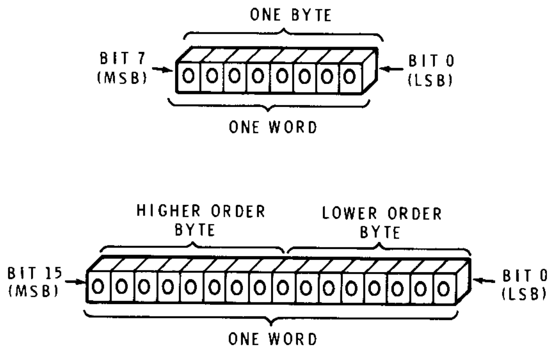


Figure 2-2
Words and Bytes.

Figure 2-2 also shows how the bits that make up the computer word are numbered. The least significant bit (LSB) is on the right while the most significant bit (MSB) is on the left. In the 8-bit word, the bits are numbered 0 through 7 from right to left. In the 16-bit word, the bits are numbered 0 through 15 as shown. The lower 8-hits are called the lower order byte while the upper 8-bits are called the higher order byte.

Self-Test Review

1. Explain the difference between a microprocessor and a microcomputer.

2. What is a bus?

3. Explain the difference between input and output data.

4. Define I/O.

5. The point at which data enters or leaves the computer is called an I/O____________.

6. The operations that the computer can be told to perform are called ____________.

7. What is a program?

8. Explain what is meant by "stored program concept."

9. A byte generally consists of ____________ bits.

10. A computer word may consist of one or more ____________.

11. What is the largest number that can be represented by an 8-bit computer word? By a 16-bit word?

Answers

1. A microprocessor is a logic element that can perform a variety of arithmetic, control, and logic operations. A microcomputer is a system that consists of a microprocessor, memory, interface adapters, clock, etc.

2. A bus is a group of conductors over which information can be transmitted. The conductors may be wires in a cable, foil patterns on a printed circuit board, or microscopic metal deposits in a silicon chip.

3. Input data is information that is entered into the computer from the outside world. Output data is information that is transmitted from the computer to the outside world. The outside world is defined as anything outside the computer.

4. I/O is the abbreviation for input-output. Thus, a device that can send data to and accept data from the computer is called an I/O device.

5. Port.

6. Instructions.

7. A program is a group of instructions that tell the computer the operations to be performed and the sequence in which they are to be performed.

8. The stored program concept refers to the technique of storing the instruction to be performed in the memory section along with the data that is to be operated upon.

9. 8.

10. Bytes.

11. $1111\ 1111_2$ or 255_{10}. $1111\ 1111\ 1111\ 1111_2$ or $65{,}535_{10}$.

AN ELEMENTARY MICROCOMPUTER

One of the difficulties you may encounter in learning about a microcomputer for the first time is the complexity of its main component — the microprocessor. The microprocessor may have a dozen or more registers varying in size from 1 bit to 16 bits. It will have scores of instructions, most of which can be implemented several different ways. It will have data, address, and control buses. In short, it can be very intimidating to start out by considering a full-blown microprocessor.

A better approach is to start with a "stripped down" version. By initially omitting some of the processor's advanced features, we arrive at a device that can be readily understood and yet maintains the characteristics of an actual microprocessor. Strictly speaking, the microprocessor developed in this unit is hypothetical in nature. However, it is so close to the real thing that the programs we develop for it will actually run on the ET-3400 Microprocessor Trainer. Also, as you will see later, one of the most popular microprocessors in use today is a vastly advanced version of our elementary model.

A block diagram of a basic microcomputer is shown in Figure 2-3, which shows the microprocessor, the memory, and the I/O circuitry. For simplicity, we will ignore the I/O circuitry in this unit. We can do this by assuming that the program and data are already in memory and that the results of any computations will be held in a register or stored in memory. Ultimately of course, the program and data must come from the outside world and the results must be sent to the outside world. But we will save these procedures until a later unit. This will allow us to concentrate on the microprocessor unit and the memory.

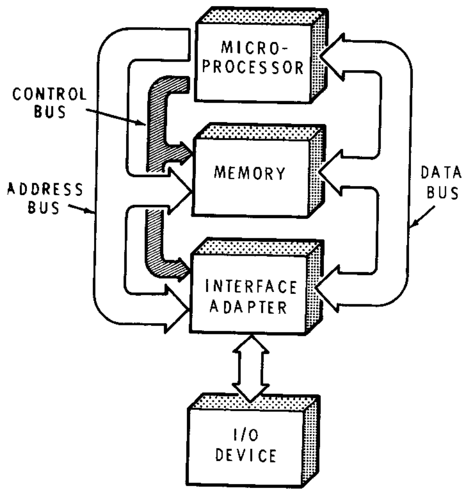


Figure 2-3
The Basic Microcomputer.

The Microprocessor Unit (MPU)

The microprocessor unit is shown in greater detail in Figure 2-4. For simplicity, only the major registers and circuits are shown. In our elementary unit most of the counters, registers, and buses are 8-bits wide. That is, they can accomodate 8-bit words.

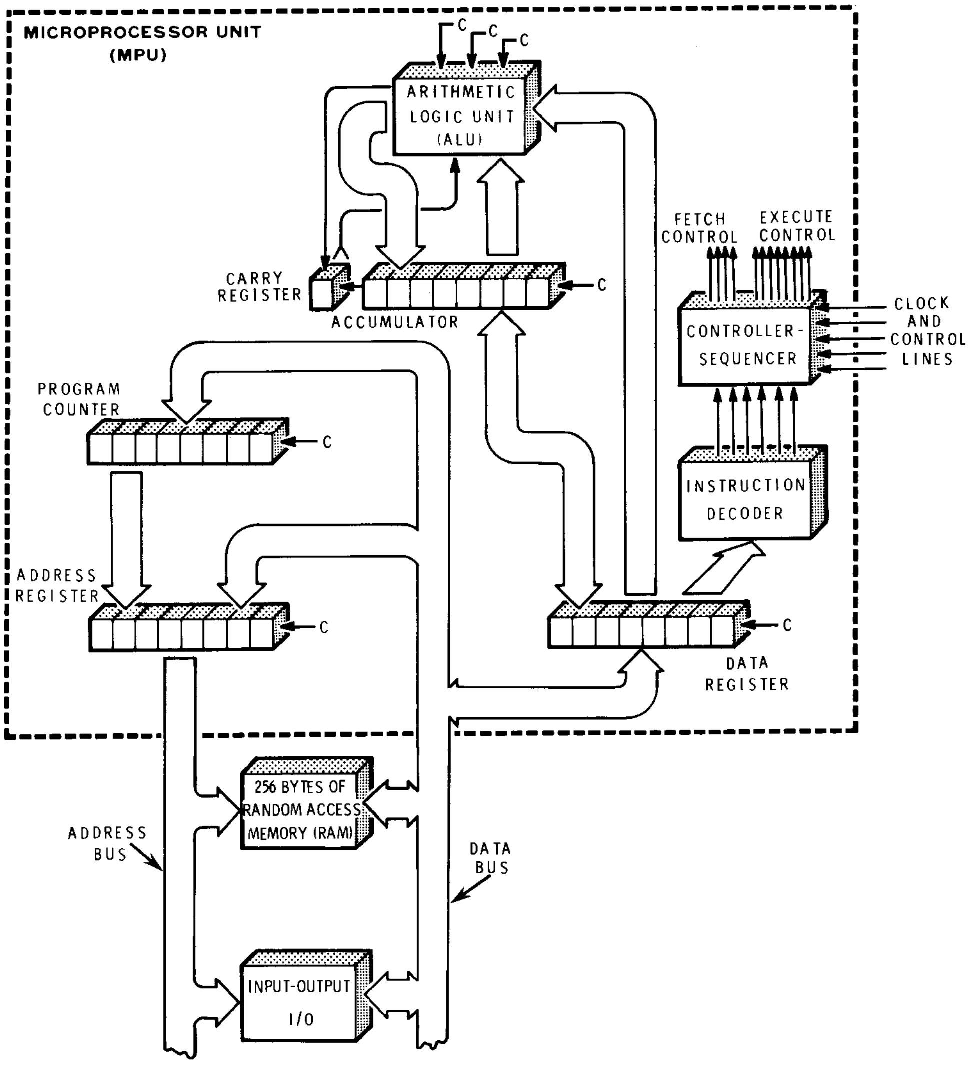
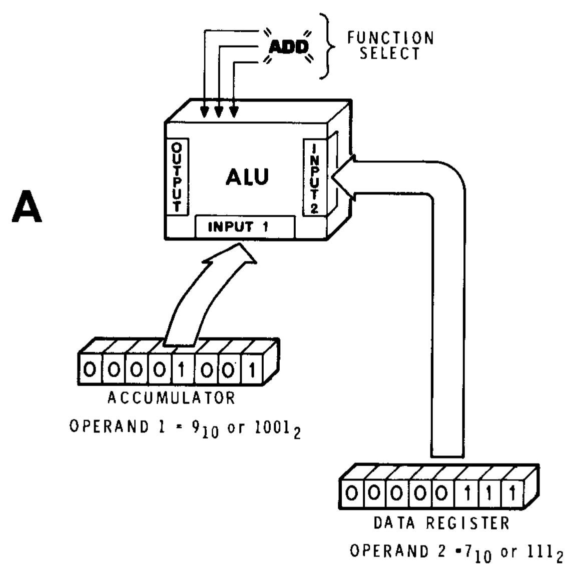


Figure 2-4
An Elementary Microprocessor.

One of the most important circuits in the microprocessor is the **arithmetic logic unit** (ALU). Its purpose is to perform arithmetic or logic operations on the data words that are delivered to it. The ALU has two main inputs. One comes from a register called the accumulator, and the other comes from the data register. The ALU can add the two input data words together, or it can subtract one from the other. It can also perform some logic operations which will be discussed in later units. The operation that the ALU performs is determined by signals on the various control lines (marked C on the block diagram).

Generally, the ALU receives two 8-bit binary numbers from the accumulator and the data register as shown in Figure 2-5A. Because some operation is performed on these data words, the two inputs are called **operands**.



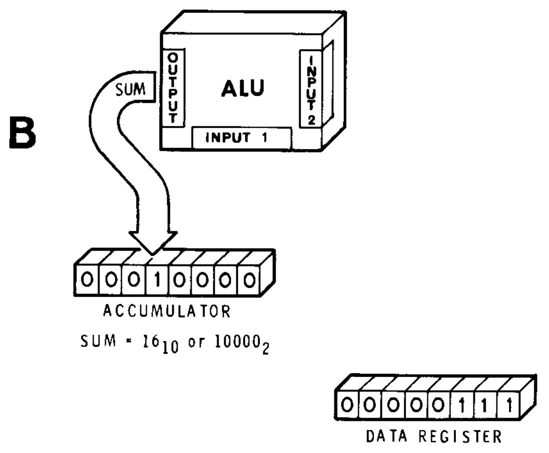


Figure 2-5
The Arithmetic Logic Unit.

The two operands may be added, subtracted, or compared in some way, and the result of the operation is stored back in the accumulator. For example, assume that two numbers (7 and 9) are to be added. Before the numbers can be added, one operand (9) is placed in the accumulator; the other (7) is placed in the data register. The proper control line is then activated to implement the add operation. The ALU adds the two numbers together, producing their sum (16_{10}) at the output. As shown in Figure 2-5B, the sum is stored in the accumulator, replacing the operand that was originally stored there. Notice that all the numbers involved are in binary form.

The **accumulator** is the most useful register in the microprocessor. During arithmetic and logic operations it performs a dual function. Before the operation, it holds one of the operands. After the operation, it holds the resulting sum, difference, or logical answer. The accumulator receives several instructions in every microprocessor. For example, the "load accumulator" instruction causes the contents of some specified memory location to be transferred to the accumulator. The "store accumulator" instruction causes the contents of the accumulator to be stored at some specified location in memory.

The **data register** is a temporary storage location for data going to or coming from the data bus. For example, it holds an instruction while the instruction is being decoded. Also, it holds a data byte while the word is being stored in memory.

The MPU also contains several other important registers and circuits: the address register, the program counter, the instruction decoder, and the controller-sequencer. These are shown in Figure 2-4.

The **address register** is another temporary storage location. It holds the address of the memory location or I/O device that is used in the operation presently being performed.

The **program counter** controls the sequence in which the instructions in a program are performed. Normally, it does this by counting in the sequence, 0, 1, 2, 3, 4, etc. At any given instant, the count indicates the location in memory from which the next byte of information is to be taken.

The **instruction decoder** does just what its name implies. After an instruction is pulled from memory and placed in the data register, the instruction is decoded by this circuit. The decoder examines the 8-bit code and decides which operation is to be performed.

The **controller-sequencer** produces a variety of control signals to carry out the instruction. Since each instruction is different, a different combination of control signals is produced for each instruction. This circuit determines the sequence of events necessary to complete the operation described by the instruction.

Later you will see how these various circuits work together to execute simple programs. But first, take a closer look at the memory for our microcomputer.

Memory

A simplified diagram of the 256- word, 8-bit read/write memory that is used in our hypothetical microcomputer is shown in Figure 2-6. The memory consists of 256_{10} locations, each of which can store an 8-bit word. This size memory is often referred to as 256 × 8. A read/write memory is one in which data can be written in and read out with equal ease.

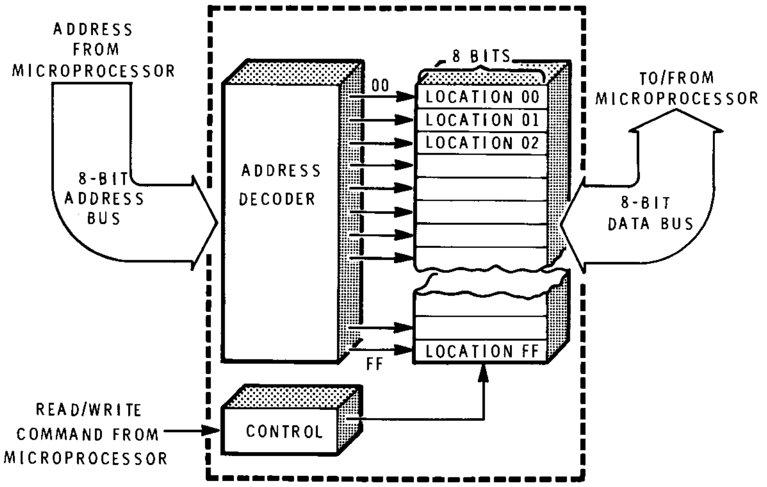


Figure 2-6
The Random Access Memory.

Two buses and a number of control lines connect the memory with the microprocessing unit. The address bus will carry an 8-bit binary number, which can specify 256_{10} locations, from the MPU to the memory address decoder. Each location is assigned a unique number called its address. The first location is given the address 0. The last location is given the address 255_{10}, which is 1111 1111 in binary and FF in hexadecimal. A specific location is selected by placing its 8-bit address on the address bus. The address decoder decodes the 8-bit number and selects the proper memory location.

The memory also receives a control signal from the MPU. This signal tells the memory the operation that is to be performed. A READ signal indicates that the selected location is to be read out. This means that the 8-bit number contained in the selected location is to be placed on the data bus where it can be transferred to the MPU.

The procedure is illustrated in Figure 2-7. Assume that the MPU is to read out the contents of memory location 04_{16}. Let's further assume that the number stored there is 97_{16}. First, the MPU places the address 04_{16} on the address bus. The decoder recognizes the address and selects the proper memory location. Second, the MPU sends a READ signal to the memory, indicating that the contents of the selected location is to be placed on the data bus. Third, the memory responds by placing the number 97_{16} on the data bus. The MPU can then pick up the number and use it as needed.

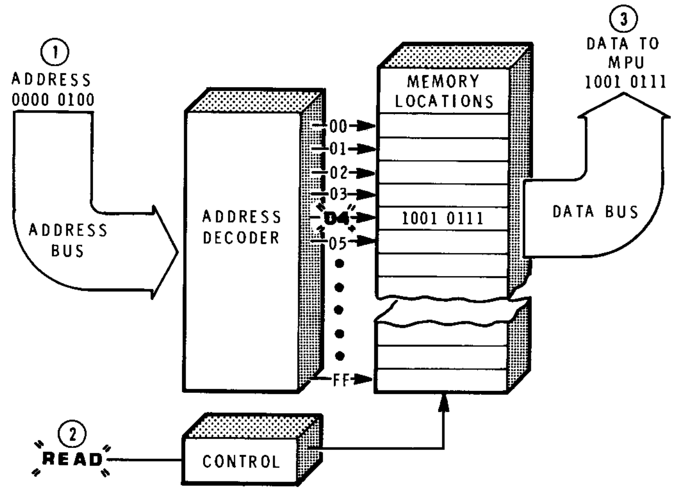


Figure 2-7
Reading from Memory.

It should be pointed out that the process of reading out a memory location does not disturb the contents of that location. That is, the number 97_{16} will still be present at memory location 04 after the read operation is finished. This characteristic is referred to as nondestructive readout (NDRO). It is an important feature because it allows us to read out the same data as many times as needed.

The MPU can also initiate a WRITE operation. This procedure is illustrated in Figure 2-8. During a WRITE operation, a data word is taken from the data bus and placed in the selected memory location. For example, let's see how the MPU can store the number 52_{16} at memory location 03. First, the MPU places the address 03 on the address bus. The decoder responds by selecting memory location 03. Second, the MPU places the number 52_{16} on the data bus. Third, the MPU sends the WRITE signal. The memory responds by storing the number contained on the data bus in the selected location. That is, 52_{16} is stored in location 03. The previous contents of the selected location are lost as the new number is written in that location.

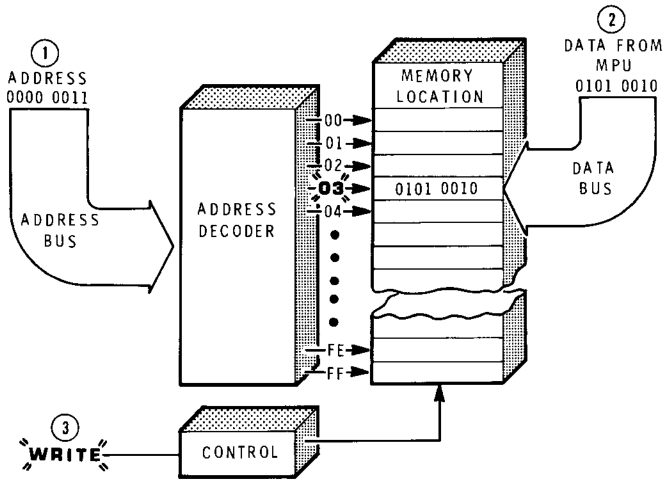


Figure 2-8
Writing into Memory.

The accepted name for a memory of this type is **Random Access Memory** (RAM). "Random access' means that all memory locations are equally accessible. However, in recent years RAM has come to mean a random access **read/write** memory. As you will see later, there is another type of memory called a read only memory (ROM). It is also randomly accessible, but it does not have a write capability. Today, the accepted definition of RAM is a random access **read/write** memory. A read only memory, although it is randomly accessible, is called a ROM and never a RAM.

Fetch-Execute Sequence

When the microcomputer is executing a program, it goes through a fundamental sequence that is repeated over and over again. Recall that a program consists of instructions that tell the microcomputer exactly what operations to perform. These instructions must be stored in an orderly manner in memory. Instructions must be fetched, one at a time, from memory by the MPU. The instruction is then executed by the MPU.

The operation of the microcomputer can be broken down into two phases, as shown in Figure 2-9. When the microprocessor is initially started, it enters the **fetch phase**. During the fetch phase, an instruction is taken from memory and decoded by the MPU. Once the instruction is decoded, the MPU switches to the **execute phase**. During this phase, the MPU carries out the operation dictated by the instruction.

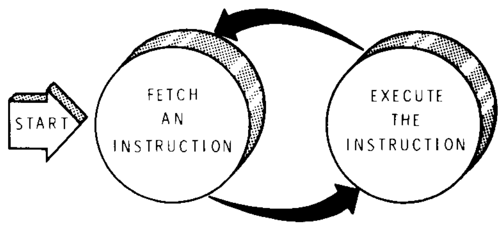


Figure 2-9
The Fetch-Execute Sequence.

The fetch phase always consists of the same series of operations. Thus, it always takes the same amount of time. However, the execute phase will consist of different sequences of events, depending on what type of instruction is being executed. Thus, the time of the execute phase may vary considerably from one instruction to the next.

A Sample Program

Now that you have a general idea of the registers and circuits found in a microcomputer, we will examine how all these circuits work together to execute a simple program. At this point, we are primarily interested in showing you how the microcomputer operates. Therefore, the program will be a very trivial one.

Let's see how the computer goes about solving a problem like 7 + 10 = ? While this seems like an incredibly easy problem, the computer, if left to its own resources, does not have the foggiest notion of how to solve it. You must tell the computer how to solve the problem right down to the smallest detail. You do this by writing a program.

Before you can write the program you must know what instructions are available to you and the computer. Every microprocessor comes with a listing of its instruction set. Assume that, after looking over the list, you decide that three instructions are necessary to solve the problem. These three instructions and a description of what they do are shown in Figure 2-10.

NAME	MNEMONIC	OPCODE	DESCRIPTION
Load Accumulator	LDA	$1000\ 0110_2$ or 86_{16}	Load the contents of the next memory location into the accumulator.
Add	ADD	$1000\ 1011_2$ or $8B_{16}$	Add the contents of the next memory location to the present contents of the accumulator. Place the sum in the accumulator.
Halt	HLT	$0011\ 1110_2$ or $3E_{16}$	Stop all operations.

Figure 2-10
Instructions Used in the Sample Program.

The first column in the table gives the name of the instruction. When writing programs, it is often inconvenient to write out the entire name. For this reason, each instruction is given an abbreviation or a memory aid called a **mnemonic**. The mnemonics are given in the second column. The third column is called the operation code or **opcode**. This is the binary number that the computer and the programmer use to represent the instruction. The opcode is given in both binary and hexadecimal form. The final column describes exactly what operation is performed when the instruction is executed. Study this table carefully; you will be using these instructions over and over again.

Assume that you wish to add 7 to 10_{10} and place the sum in the accumulator. The program is an elementary one. First you will load 7 into the accumulator with the LDA instruction. Next, you will add 10_{10} to the accumulator using the ADD instruction. Finally, you will stop the computer with the HLT instruction.

Using the mnemonics and the decimal representation of the numbers to be added, the program looks like this:

```
LDA 7
ADD 10
HLT
```

Unfortunately, the basic microcomputer cannot understand mnemonics or decimal numbers. It can interpret binary numbers and nothing else. Thus, you must write the program as a sequence of binary numbers. You can do this by replacing each mnemonic with its corresponding opcode and each decimal number with its binary counterpart.

That is:

LDA 7	becomes	1000 0110 opcode from Figure 2-10	0000 0111 binary representation for 7

And:

ADD 10	becomes	1000 1011 opcode from Figure 2-10	0000 1010 binary representation for 10_{10}

Finally,

HLT	becomes	0011 1110 opcode from Figure 2-10

Notice that the program consists of three instructions. The first two instructions have two parts: an 8-bit opcode followed by an 8-bit operand. The operands are the two numbers that are to be added (7 and 10_{10}).

Recall that the microprocessor and memory work with 8-bit words or bytes. Because the first two instructions consist of 16-bits of information, they must be broken into two 8-bit bytes before they can be stored in memory. Thus, when the program is stored in memory, it will look like this:

1st Instruction	1000	0110	Opcode for LDA
	0000	0111	Operand (7)
2nd Instruction	1000	1011	Opcode for ADD
	0000	1010	Operand (10_{10})
3rd Instruction	0011	1110	Opcode for HLT

Five bytes of memory are required. You can store this 5-byte program any place in memory you like. Assuming you store it at the first five memory locations, the memory can be diagrammed as shown in Figure 2-11.

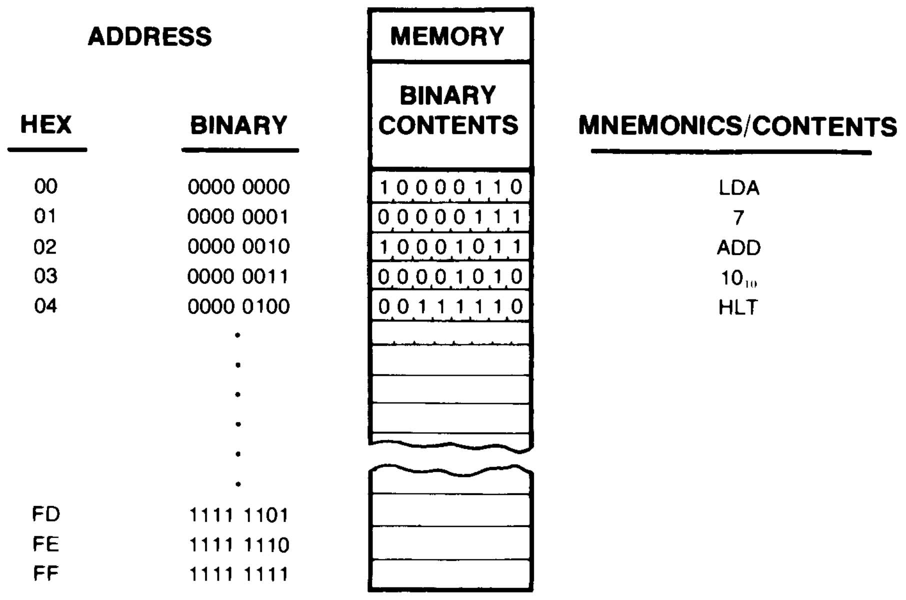


Figure 2-11
The Program in Memory.

Notice that each memory location has two 8-bit binary numbers associated with it. One is its address, the other is its contents. Be careful not to confuse these two numbers. The address is fixed. It is established when the microcomputer is built. However, the contents may be changed at any time by storing new data.

Before you see how this program is executed, let's review the material covered in this section.

Self-Test Review

12. The circuit in the microprocessor that performs arithmetic and logic operations is called the ________.

13. The numbers that are operated upon by the computer are called ________.

14. Where are the two operands held as they are transferred to the ALU?

15. Where is the result held after an arithmetic operation?

16. Which register in the MPU holds the instruction while the opcode is being decoded?

17. Which circuit in the MPU determines the memory location from which the next byte of information will be taken?

18. Which register holds the address while it is being decoded?

19. How many memory locations can be specified by an 8-bit address?

20. Explain the 3-step procedure for writing the number 34_{16} into memory location 9.

21. Define mnemonic.

22. Define opcode.

23. Define RAM.

24. An instruction is retrieved from memory and decoded during the ________ phase.

25. The operation indicated by the instruction is carried out during the ________ phase.

26. In our hypothetical microcomputer, the mnemonic for the load accumulator instruction is ________.

27. The opcode for the halt instruction is ________.

28. When the ADD instruction is executed, where is the SUM stored?

29. How many memory locations are required to store the following program?

```
LDA  13₁₀
ADD  17₁₀
ADD  10₁₀
HLT
```

30. What will be the contents of the accumulator after this program is executed?

Answers

12. Arithmetic logic unit (ALU).

13. Operands.

14. One is held in the accumulator; the other in the data register.

15. In the accumulator.

16. Data register.

17. The program counter.

18. The address register.

19. $2^8 = 256_{10}$.

20. Step 1. The address 9 is placed on the address bus.

    Step 2. The data 34_{16} is placed on the data bus.

    Step 3. The read/write line is switched to the write state.

21. A mnemonic is an abbreviation or memory aid.

22. An opcode is a binary number that tells the microprocessor which instruction to execute.

23. RAM has come to mean a random access read/write memory.

24. Fetch.

25. Execute.

26. LDA.

27. $0011\ 1110_2$.

28. In the accumulator.

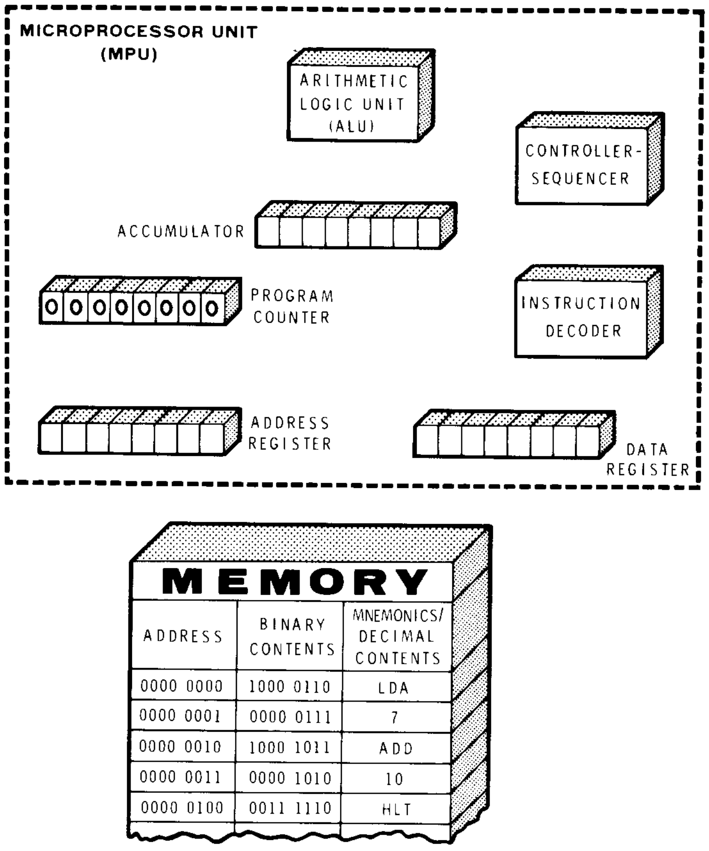
29. Seven.

30. 40_{10} or 101000_2.

EXECUTING A PROGRAM

Before a program can be run, it must be placed in memory. Later, you will see how this is done. For now, assume that we have already loaded the program developed in the previous section.

The important registers of the microcomputer are shown in Figure 2-12. Notice that our 5-byte program for adding 7 and 10_{10} is shown in memory. The following paragraphs will take you through the step-by-step procedure by which the computer executes this program.



ADDRESS	BINARY CONTENTS	MNEMONICS/ DECIMAL CONTENTS
0000 0000	1000 0110	LDA
0000 0001	0000 0111	7
0000 0010	1000 1011	ADD
0000 0011	0000 1010	10
0000 0100	0011 1110	HLT

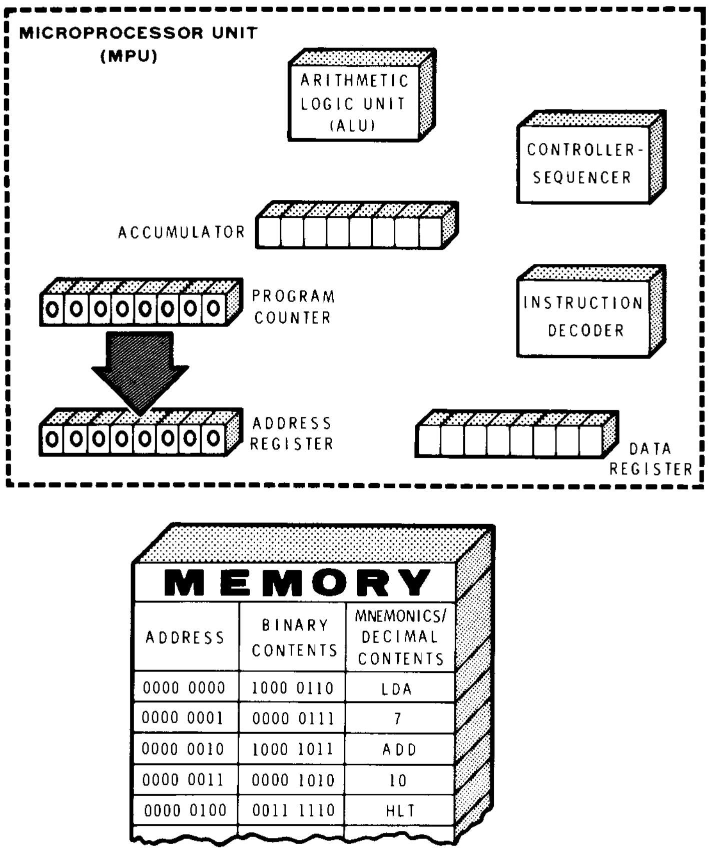
Figure 2-12
The Program Counter is Set to the Address of the First Instruction.

To begin executing the program, the program counter must be set to the address of the first instruction. In this case, the first instruction is in memory location 0000 0000, so the program counter is set accordingly. The procedure for setting the program counter to the proper address will be discussed later.

The Fetch Phase

The first step is to fetch the first instruction from memory. The sequence of events that happen during the fetch phase is controlled by the controller-sequencer. It produces a number of control signals which will cause the events illustrated in Figure 2-13 through 2-17 to occur.

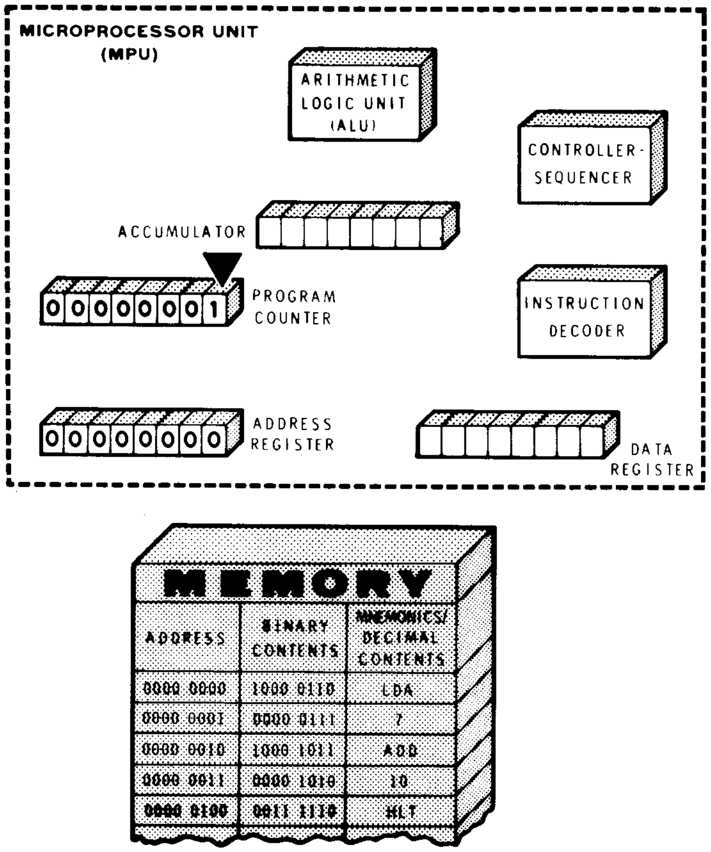
First the contents of the program counter are transferred to the address register as shown in Figure 2-13. Recall that this is the address of the first instruction.



ADDRESS	BINARY CONTENTS	MNEMONICS/ DECIMAL CONTENTS
0000 0000	1000 0110	LDA
0000 0001	0000 0111	7
0000 0010	1000 1011	ADD
0000 0011	0000 1010	10
0000 0100	0011 1110	HLT

Figure 2-13
The Contents of the Program Counter are Transferred to the Address Register.

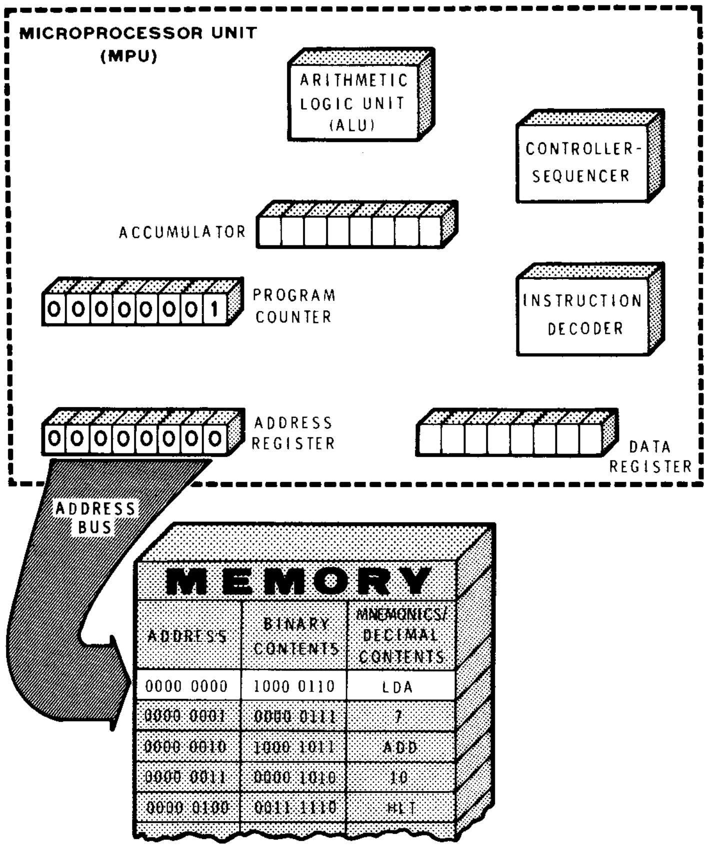
Once the address is safely in the address register, the program counter is incremented by one. That is, its contents change from 0000 0000 to 0000 0001 (Figure 2-14). Notice that this does not change the contents of the address register in any way.



ADDRESS	BINARY CONTENTS	MNEMONICS/ DECIMAL CONTENTS
0000 0000	1000 0110	LDA
0000 0001	0000 0111	7
0000 0010	1000 1011	ADD
0000 0011	0000 1010	10
0000 0100	0011 1110	HLT

Figure 2-14
The Program Counter is Incremented.

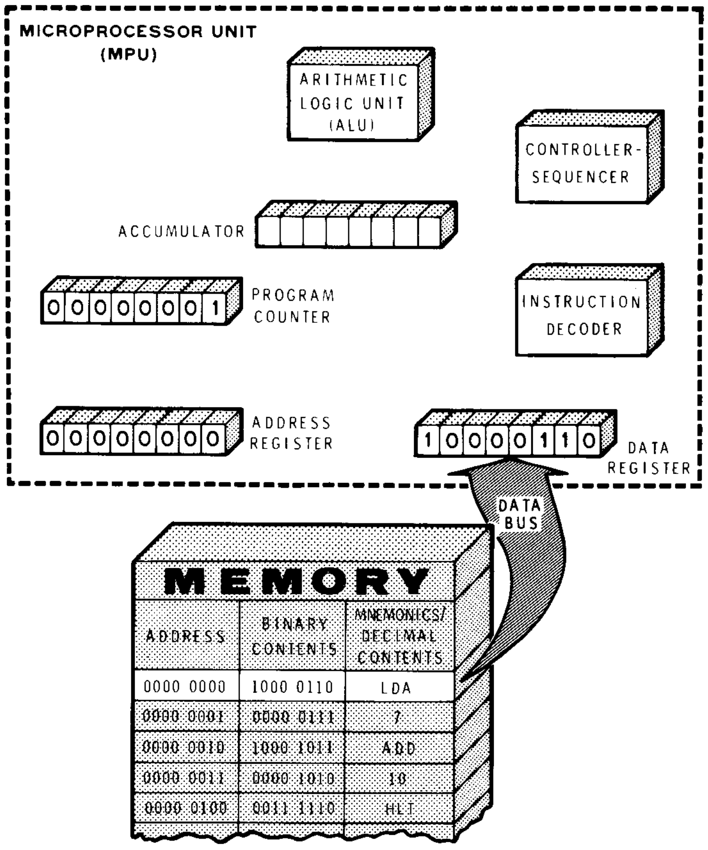
Next, the contents of the address register (0000 0000) are placed on the address bus (Figure 2-15). The memory circuits decode the address and select memory location 0000 0000.



ADDRESS	BINARY CONTENTS	MNEMONICS/ DECIMAL CONTENTS
0000 0000	1000 0110	LDA
0000 0001	0000 0111	7
0000 0010	1000 1011	ADD
0000 0011	0000 1010	10
0000 0100	0011 1110	HLT

Figure 2-15
The Address of the First Instruction is Placed on the Address Bus.

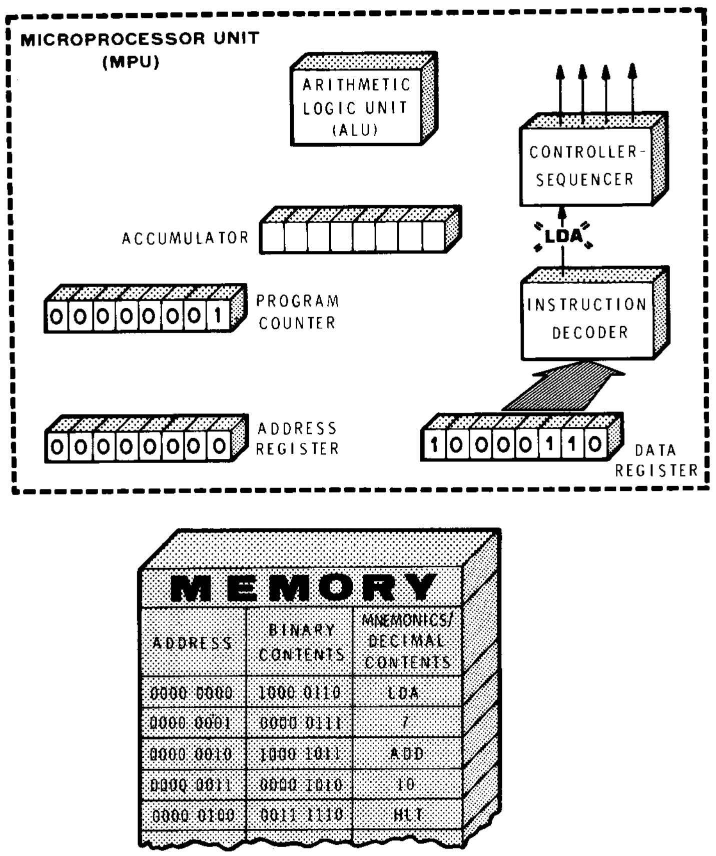
The contents of the selected memory location are placed on the data bus and transferred to the data register in the MPU. After this operation, the opcode for the LDA instruction will be in the data register as shown in Figure 2-16.



ADDRESS	BINARY CONTENTS	MNEMONICS/ DECIMAL CONTENTS
0000 0000	1000 0110	LDA
0000 0001	0000 0111	7
0000 0010	1000 1011	ADD
0000 0011	0000 1010	10
0000 0100	0011 1110	HLT

Figure 2-16
The Opcode of the First Instruction is Placed on the Data Bus.

The next step is to decode the instruction (Figure 2-17). The opcode is transferred to the instruction decoder. This circuit recognizes that the opcode is that of an LDA instruction. It informs the controller-sequencer of this fact and the sequencer produces the necessary control pulses to carry out the instruction. This completes the fetch phase of the first instruction.



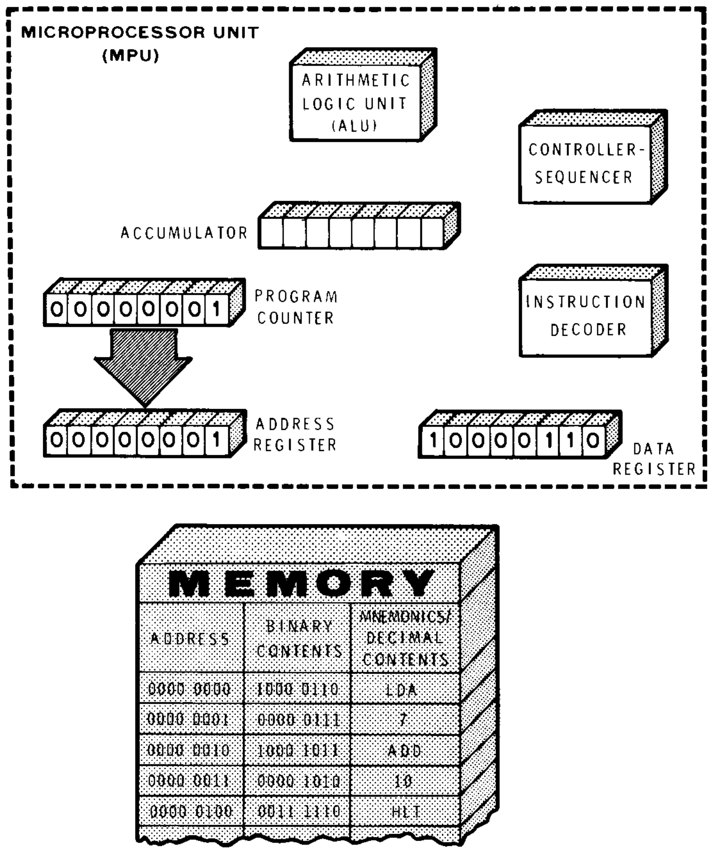
ADDRESS	BINARY CONTENTS	MNEMONICS/ DECIMAL CONTENTS
0000 0000	1000 0110	LDA
0000 0001	0000 0111	7
0000 0010	1000 1011	ADD
0000 0011	0000 1010	10
0000 0100	0011 1110	HLT

Figure 2-17
The Opcode is Decoded.

The Execute Phase

The first instruction was fetched from memory and decoded during the fetch phase. The MPU now knows that this is an LDA instruction. During the execute phase, it must carry out this instruction by reading out the next byte of memory and placing it in the accumulator.

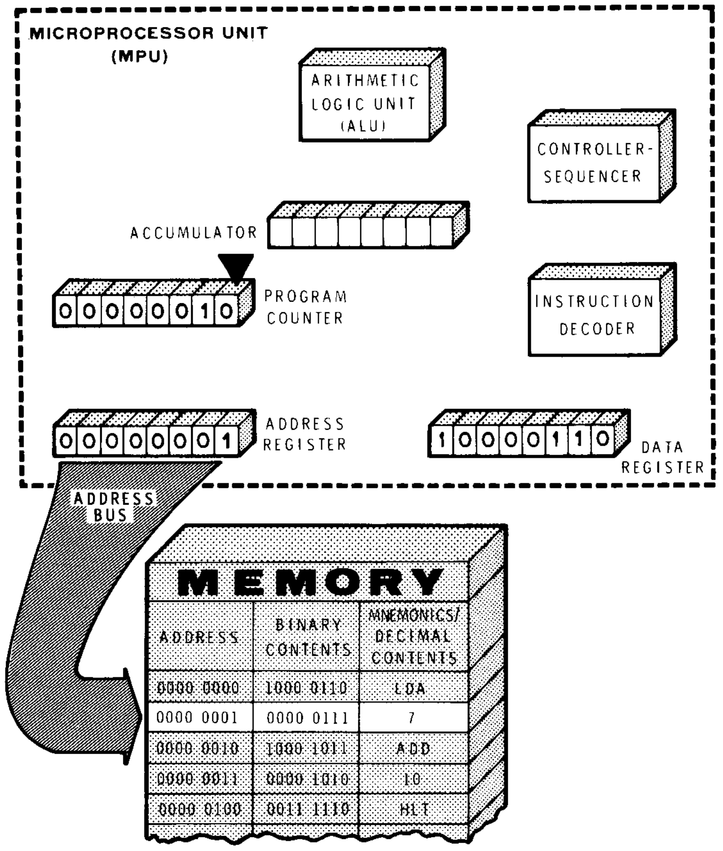
The first step is to transfer the address of the next byte from the program counter to the address register (Figure 2-18). You will recall that the program counter was incremented to the proper address (0000 0001) during the previous fetch phase.



ADDRESS	BINARY CONTENTS	MNEMONICS/ DECIMAL CONTENTS
0000 0000	1000 0110	LDA
0000 0001	0000 0111	7
0000 0010	1000 1011	ADD
0000 0011	0000 1010	10
0000 0100	0011 1110	HLT

Figure 2-18
The Contents of the Program Counter are Transferred to the Address Register.

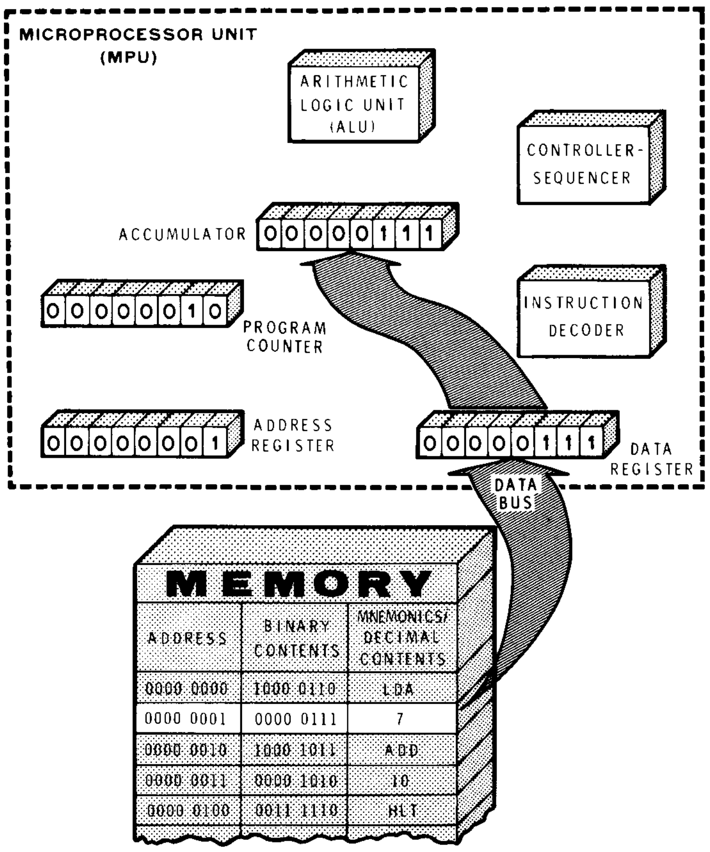
The next two operations are shown in Figure 2-19. First, the program counter is incremented to 0000 0010 in anticipation of the next fetch phase. Second, the contents of the address register (000 0001) are placed on the address bus.



ADDRESS	BINARY CONTENTS	MNEMONICS/ DECIMAL CONTENTS
0000 0000	1000 0110	LDA
0000 0001	0000 0111	7
0000 0010	1000 1011	ADD
0000 0011	0000 1010	10
0000 0100	0011 1110	HLT

Figure 2-19
The Program Counter is Incremented; the Contents of the Address Register are Placed on the Address Bus.

The address is decoded and the contents of memory location 0000 0001 are loaded into the data register as shown in Figure 2-20. Recall that this is the number 7. An instant later, the number is transferred to the accumulator. Thus, the first execute phase ends with the number 7 in the accumulator.


MICROPROCESSOR UNIT (MPU)
ARITHMETIC LOGIC UNIT (ALU)
CONTROLLER-SEQUENCER
ACCUMULATOR 0 0 0 0 0 1 1 1
0 0 0 0 0 0 1 0 PROGRAM COUNTER
INSTRUCTION DECODER
0 0 0 0 0 0 0 1 ADDRESS REGISTER
0 0 0 0 0 1 1 1 DATA REGISTER
DATA BUS
MEMORY

ADDRESS	BINARY CONTENTS	MNEMONICS/ DECIMAL CONTENTS
0000 0000	1000 0110	LDA
0000 0001	0000 0111	7
0000 0010	1000 1011	ADD
0000 0011	0000 1010	10
0000 0100	0011 1110	HLT

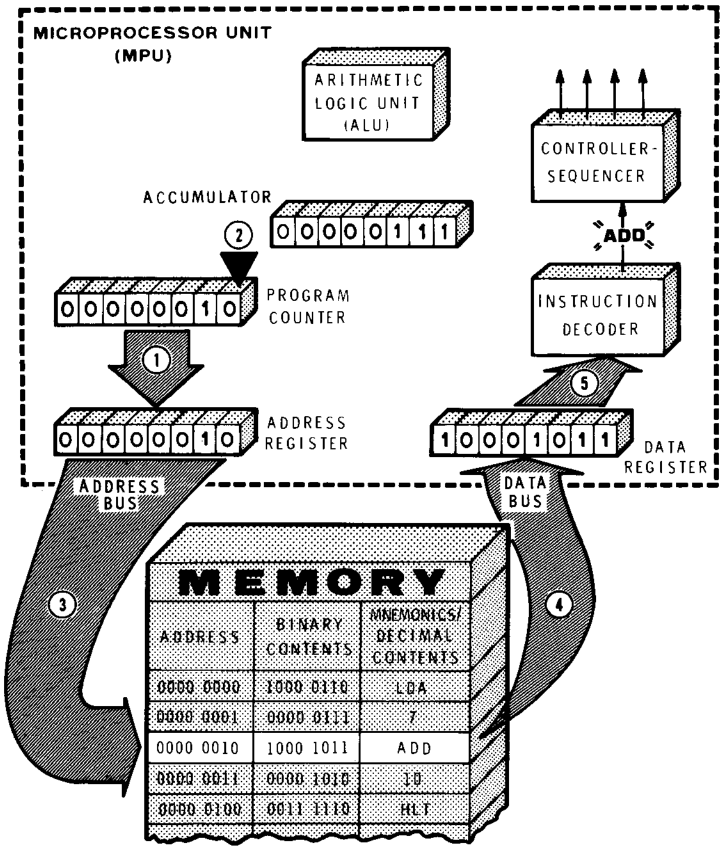
Figure 2-20
The First Operand is Transferred to the Accumulator Via the Data Register.

Fetching the Add Instruction

The next instruction in our program is the ADD instruction. It is fetched from memory using the same procedure outlined above. Figure 2-21 illustrates this five-step procedure:

1. The contents of the program counter (0000 0010) are transferred to the address register.

2. The program counter is incremented to 0000 0011.
3. The address is placed on the address bus.
4. The contents of the selected memory location are transferred to the data register.
5. The contents of the data register are decoded by the instruction decoder.



ADDRESS	BINARY CONTENTS	MNEMONICS/ DECIMAL CONTENTS
0000 0000	1000 0110	LDA
0000 0001	0000 0111	7
0000 0010	1000 1011	ADD
0000 0011	0000 1010	10
0000 0100	0011 1110	HLT

Figure 2-21
Fetching the ADD Instruction.

The data word fetched from memory is the opcode for the ADD instruction. Thus, the controller-sequencer produces the necessary control pulses to execute this instruction.

Executing the Add Instruction

The execution of the ADD instruction is a six-step procedure. This procedure is illustrated in Figure 2-22.

1. The contents of the program counter (0000 0011) are transferred to the address register.

2. The program counter is incremented to 0000 0100 in anticipation of the next fetch phase.

3. The address of the operand is placed on the address bus.

4. The operand (10_{10}) is transferred to the data register.

5A. The operand (10_{10}) is transferred into one input of the ALU.

5B. Simultaneously, the other operand (7) is transferred from the accumulator to the other input of the ALU.

6. The ALU adds the two operands. Their sum ($0001\ 0001_2$ or 17_{10}) is loaded into the accumulator, destroying the number (7) that was previously stored there.

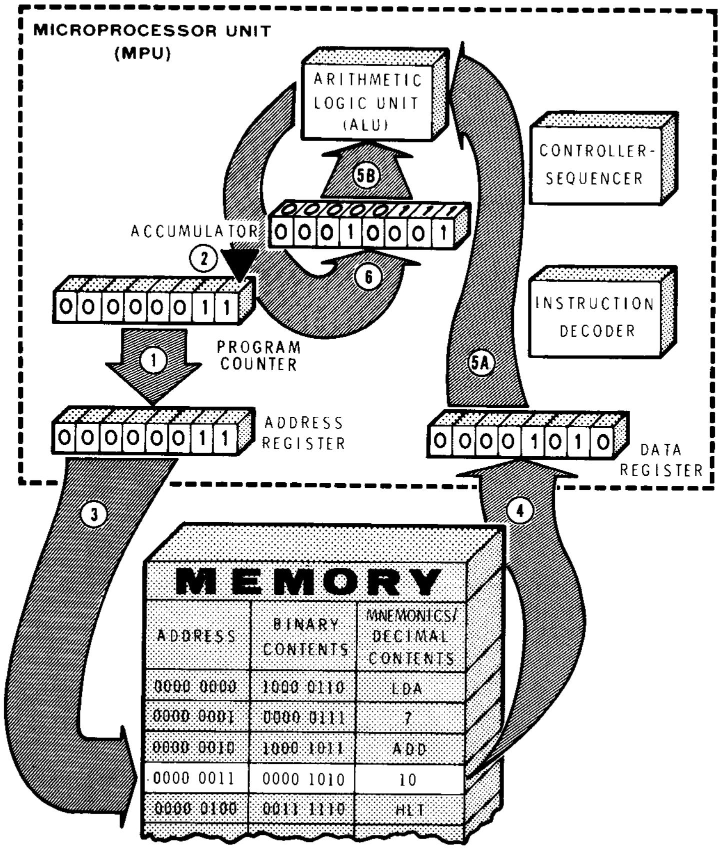


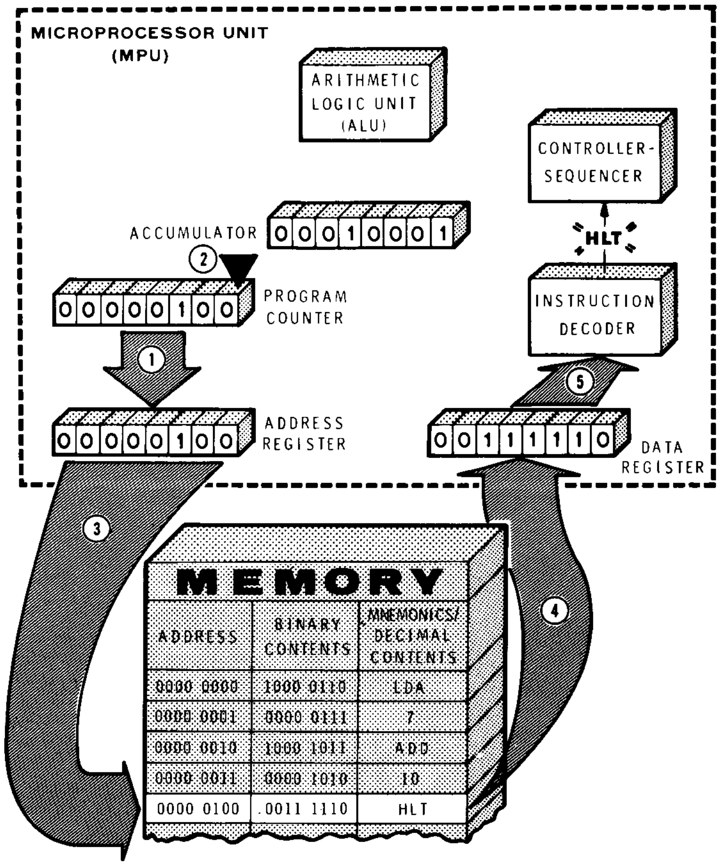
Figure 2-22
Executing the ADD Instruction.

The computation portion of our program ends with the sum of the two operands in the accumulator. However, the program is not finished until it tells the computer to stop executing instructions.

Fetching and Executing the HLT Instruction

The final instruction in the program is a HLT instruction. It is fetched using the same fetch procedure as before. The five steps are illustrated in Figure 2-23. The address comes from the program counter via the address register. When this address is placed on the address bus, memory location 0000 0100 is read out and the opcode for HLT is loaded into the data register. The opcode is decoded and the instruction is executed.

The execution of the HLT instruction is very simple. The controller-sequencer simply stops producing control signals. Consequently, all computer operations stop. Notice that the program has accomplished our objective of adding 7_{10} to 10_{10}. The resulting sum, 17_{10}, is in the accumulator.



ADDRESS	BINARY CONTENTS	MNEMONICS/ DECIMAL CONTENTS
0000 0000	1000 0110	LDA
0000 0001	0000 0111	7
0000 0010	1000 1011	ADD
0000 0011	0000 1010	10
0000 0100	0011 1110	HLT

Figure 2-23
Fetching and Executing the HLT Instruction.

Self-Test Review

Examine this sample program carefully and answer the questions below:

```
LDA  8
ADD  7
ADD  4
HLT
```

31. During the first fetch phase, what binary number is loaded into the data register?

32. During the first execute phase the number 0000 1000_2 is loaded into the ______________.

33. During the second fetch phase, what binary number is loaded into the data register?

34. If the first byte of the program is placed in address 0000 0000, what is the address of the first ADD instruction?

35. How many bytes of memory are taken up by the program?

36. What number is in the accumulator during the third fetch phase?

37. When the program is finished running, what number is in the accumulator?

38. What is the final number in the program counter?

39. What are the final contents of the address register?

40. What are the final contents of the data register?

Answers

31. The opcode for the LDA instruction, $1000\ 0110_2$.

32. Accumulator.

33. The opcode for ADD, 1000 1011.

34. $0000\ 0010_2$.

35. Seven.

36. 15_{10} or $0000\ 1111_2$.

37. 19_{10} or $0001\ 0011_2$.

38. 7_{10} or $0000\ 0111_2$.

39. 6_{10} or $0000\ 0110_2$.

40. The opcode for the HLT instruction, $0011\ 1110_2$.

ADDRESSING MODES

If you examine the program discussed in the previous section, you will find that it uses two distinctly different types of instructions. One type of instruction requires an operand. LDA and ADD are examples of this type. These are two-byte instructions. The first byte is the opcode; the second is the operand.

Microprocessors also have single-byte instructions. HLT is a good example. This instruction requires no operand; thus, it can be implemented with a single byte.

Instructions can be classified in several different ways. One of the most basic distinctions is their addressing mode. The addressing mode refers to the method by which an instruction addresses its operand.

Inherent or Implied Addressing

Single-byte instructions have no operand or the operand is implied by the opcode itself. For example, the HLT instruction has no operand at all. However, some single-byte instructions may have an implied operand. An example is the "increment accumulator" instruction. The number that is operated upon (incremented) is the number in the accumulator. This instruction simply adds one to the contents of the accumulator. This type of addressing will be referred to in this course as **implied addressing** or **inherent addressing**. The operations performed during an instruction of this type are illustrated in Figure 2-24.

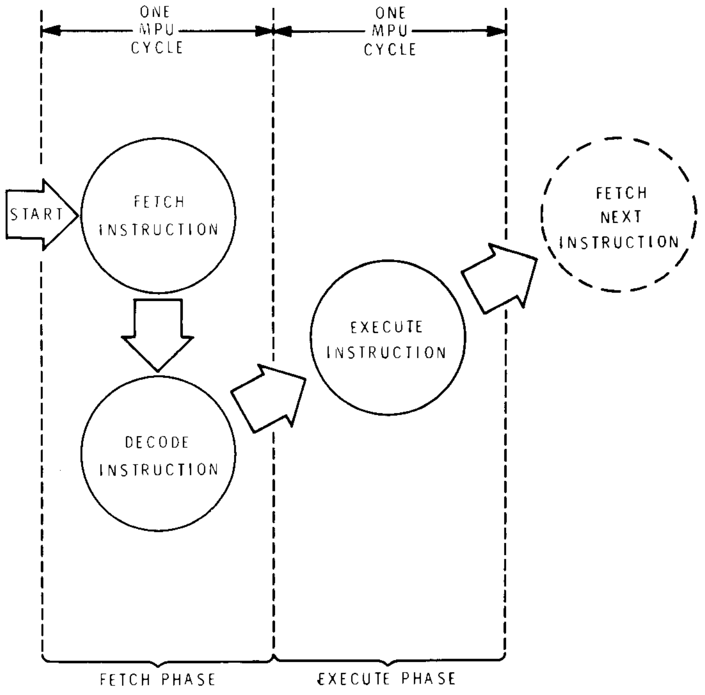


Figure 2-24
Operations Performed in the Inherent or Implied Addressing Mode.

Immediate Addressing

In our previous program, the two-byte instructions use the **immediate addressing** mode. In this mode, the operand is the byte immediately following the opcode. That is, the byte of data that is to be operated upon is the second byte of the instruction. When these two-byte instructions are stored in memory, the address of the operand is the memory location following the opcode. The operations performed during this type of instruction are illustrated in Figure 2-25.

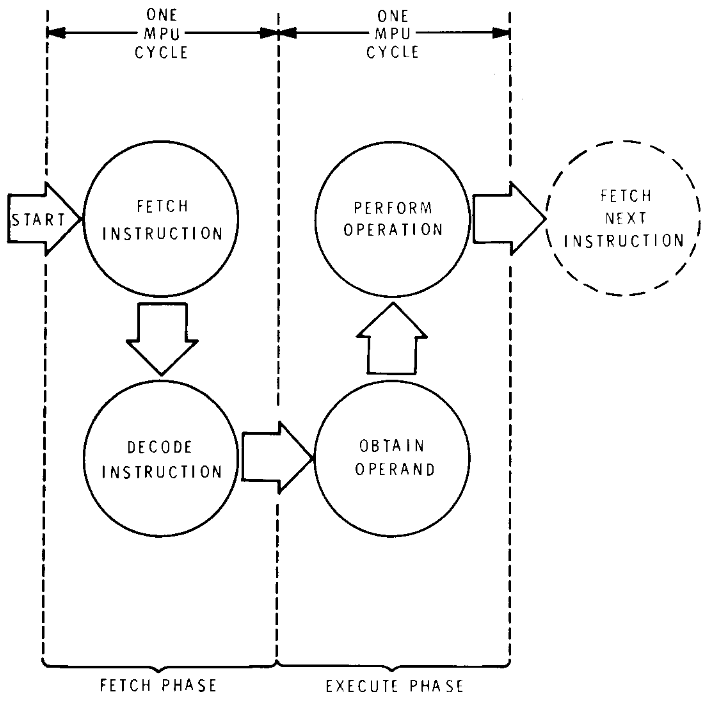


Figure 2-25
Operations Performed in the Immediate Addressing Mode.

The inherent and immediate addressing modes have two advantages. First, they require little memory space; one and two bytes respectively. This is important because memory locations cost money. Generally, the less memory space taken by a program, the better off we are. Second, these addressing modes require a minimum of execution time. The execution time can become important in long programs.

The time required for an instruction to be fetched and executed is often given in **MPU cycles**. We will define an MPU cycle as the minimum time required to fetch a data byte from memory. Thus, the fetch phase of an instruction requires one MPU cycle. In inherent and immediate addressing modes, the execute phase also requires one MPU cycle. Therefore, the **minimum** time required to fetch and execute any instruction is two MPU cycles. As you will see later, other addressing modes will require more MPU cycles.

Because of their fast execution time and their efficient use of memory, the inherent and immediate modes of addressing should be used wherever possible. However, there are many situations in which these addressing modes are simply not suitable. For this reason, every microprocessor will have instructions which use other addressing modes.

Direct Addressing

Most computer operations involve an operand. To realize its full potential, the computer must be able to manipulate the operand in many different ways. Immediate addressing of an operand is most useful when the operand is a constant that is used by only one instruction in the program. In this case, the operand can be placed immediately after that instruction's opcode.

However, there are many cases in which the operand is a variable which may be operated upon by many different instructions. In these cases, the immediate addressing mode is simply not practical and a more sophisticated form of addressing is necessary.

One way of solving this problem is to use the **direct addressing mode**. In this mode, an instruction requires two bytes of memory. The first byte is the opcode of the instruction just as before. However, the second byte is **not** the operand. Instead, the second byte is the **address** of the operand. The format of the instruction as it appears in memory is shown in Figure 2-26.

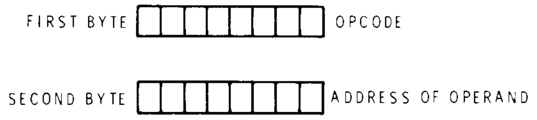


Figure 2-26
Format of an Instruction which uses the Direct Addressing Mode.

Three typical direct-addressing-mode instructions are listed in Figure 2-27. The first is the load accumulator instruction (LDA). Read the description carefully and note the difference between this instruction and the LDA immediate instruction discussed earlier. An example of each may help. In the **immediate addressing** mode, the instruction

LDA 50_{10}

means "load 50_{10} into the accumulator." But in the **direct addressing** mode, the same instruction means "load the number at memory location 50_{10} into the accumulator."

NAME	MNEMONIC	OPCODE	DESCRIPTION
Load Accumulator	LDA	$1001\ 0110_2$ or 96_{16}	Load the contents of the memory location whose address is given by the next byte into the accumulator.
Add	ADD	$1001\ 1011_2$ or $9B_{16}$	Add the contents of the memory location whose address is given by the next byte to the present contents of the accumulator. Place the sum in the accumulator.
Store Accumulator	STA	$1001\ 0111_2$ or 97_{16}	Store the contents of the accumulator in the memory location whose address is given by the next byte.

Figure 2-27
Direct Addressing Mode Instructions.

You may have noticed that this instruction has a different opcode from the LDA immediate instruction. This is necessary to tell the MPU the exact nature of the instruction. That is, the opcode tells the MPU the addressing mode as well as the operation that is to be performed.

The ADD instruction also has a slightly different meaning in the direct addressing mode. Recall that, in the **immediate** addressing mode,

ADD 10_{10}

means "add 10_{10} to the contents of the accumulator." However, in the **direct** addressing mode.

ADD 10_{10}

means "add the contents of memory location 10_{10} to the contents of the accumulator." Once again, the opcode tells the MPU the addressing mode. An opcode of $8B_{16}$ means ADD immediate whereas an opcode of $9B_{16}$ means ADD direct.

The last instruction shown is a store accumulator (STA) instruction. It tells the MPU to store the contents of the accumulator in the address indicated by the second byte of the instruction.

For example:

STA 20_{10}

means "store the contents of the accumulator in memory location 20_{10}." Because the second byte of the instruction must be an address, there is no STA immediate instruction.

The direct addressing mode instructions require one or more additional MPU cycles to execute. A typical fetch-execute sequence is illustrated in Figure 2-28. The instruction fetch is the same regardless of the addressing mode. However, the execution phase is extended since the MPU must first obtain the address of the operand from memory and then retrieve the operand itself from memory.

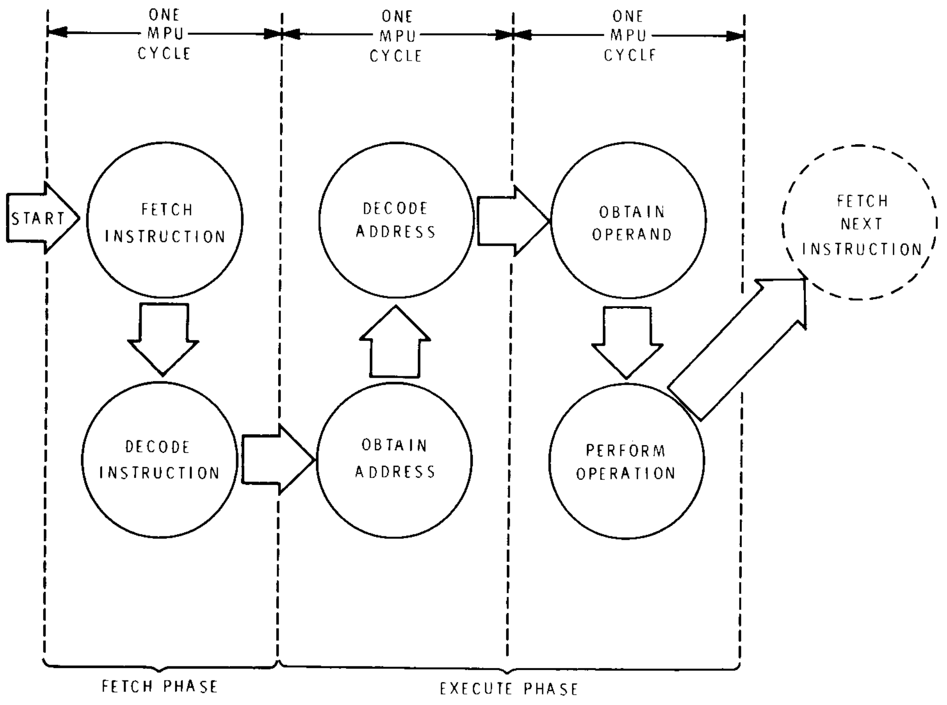


Figure 2-28
Most Direct Addressing Mode Instructions Require Three MPU Cycles.

Direct addressing generally requires more memory and longer execution times. However, there are many cases in which the added flexibility makes direct addressing worthwhile in spite of these disadvantages.

Sample Program Using Direct Addressing

The differences between direct and immediate addressing can be illustrated by a sample program. Earlier we examined a program which added two numbers (7 and 10). The sum was placed in the accumulator. Now let's look at the same program using direct addressing, only this time the sum will be stored in memory.

Figure 2-29 shows the program as it would look when stored in memory. Assume that we have arbitrarily stored the program starting at memory location or address 0001 0000_2 (16_{10}). Addresses 16_{10} and 17_{10} contain the first instruction:

LDA 23_{10}

BINARY ADDRESS	BINARY CONTENTS	MNEMONICS/ CONTENTS	
0001 0000	1001 0110	LDA	1st Instruction
0001 0001	0001 0111	23_{10}	
0001 0010	1001 1011	ADD	2nd Instruction
0001 0011	0001 1000	24_{10}	
0001 0100	1001 0111	STA	3rd Instruction
0001 0101	0001 1001	25_{10}	
0001 0110	0011 1110	HLT	4th Instruction
0001 0111	0000 0111	7_{10}	Data
0001 1000	0000 1010	10_{10}	
0001 1001	0000 0000	Reserved for sum	

Figure 2-29
Sample Program Using Direct Addressing.

This instruction tells the MPU to load the contents of memory location 23_{10} into the accumulator. Looking down to address 23_{10} ($0001\ 0111_2$), you see that it contains the operand 7. Thus, the first instruction causes 7 to be loaded into the accumulator.

The second instruction is in memory locations 18_{10} and 19_{10}. It is:

ADD 24_{10}

This tells the MPU to add the number at address 24_{10} to the number in the accumulator. Address 24_{10} ($0001\ 1000_2$) contains the number 10_{10}. Therefore, the second instruction causes 10_{10} to be added to the contents of the accumulator. The sum (17_{10}) is placed back in the accumulator.

The third instruction, in locations 20_{10} and 21_{10}, is:

STA 25_{10}

This tells the MPU to store the contents of the accumulator in memory location 25_{10}. After this instruction is executed, the number 17_{10} will appear in location 25_{10}.

The final instruction tells the MPU to halt. The program illustrates the value of the HLT instruction. Let's assume that the HLT instruction is inadvertently omitted. In this case, the MPU would fetch the next byte in sequence and attempt to execute it as if it were an instruction. The next byte is the number 7_{10}. This is a data word and was never intended to be an instruction. Nevertheless, the MPU probably has an instruction with an opcode of 7_{10}. After executing this instruction, the MPU will continue to fetch and execute whatever it finds in the remaining memory locations. Without a HLT instruction, it has no way of knowing where the instructions end and data begins.

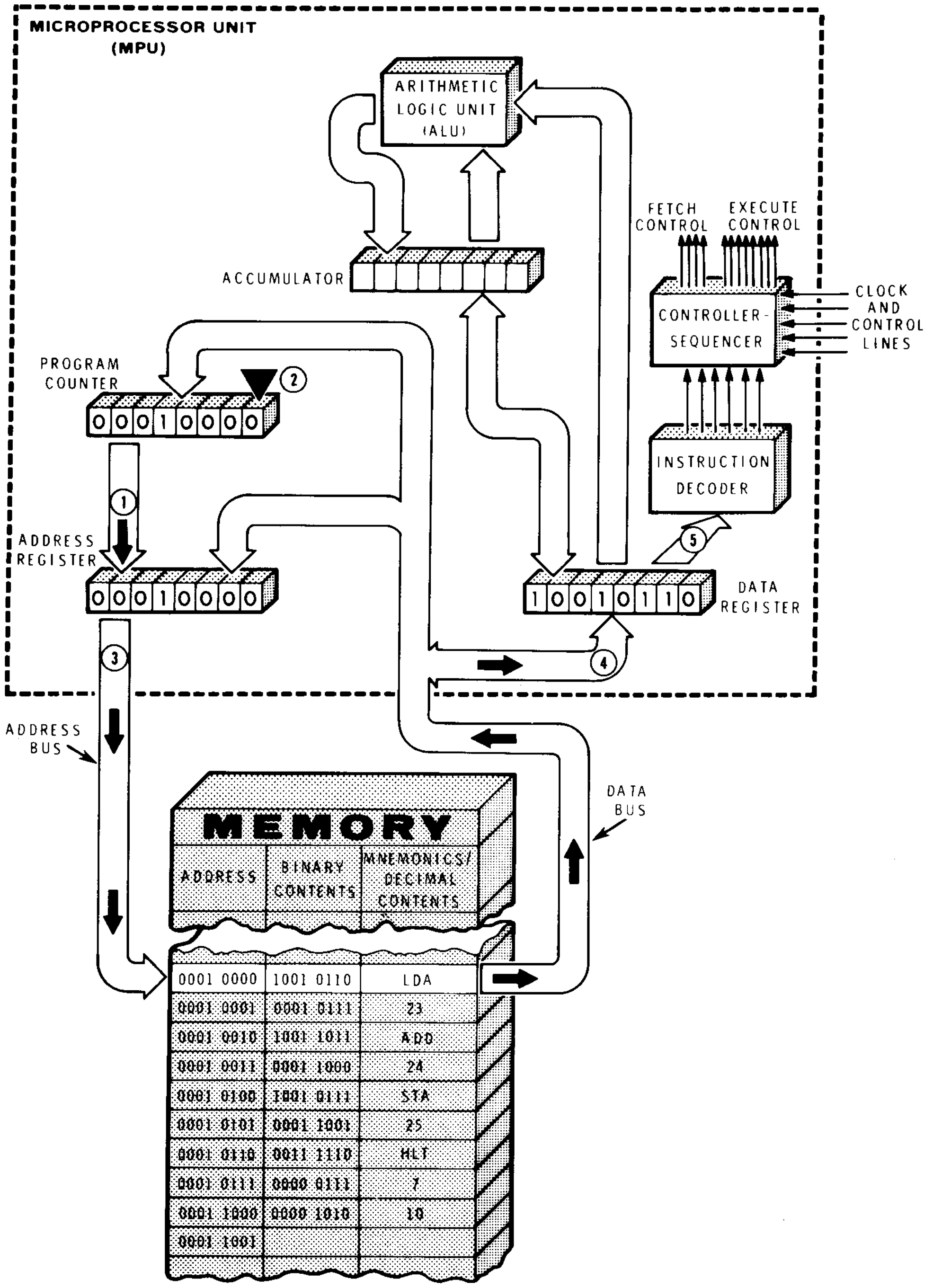
Executing the Sample Program

The data flow within the microcomputer is slightly different for the direct addressing mode. Figure 2-30 shows several of the data paths within the microcomputer. Using this type of diagram, let's examine the data manipulations that occur during the execution of our sample program.

Notice that our program is loaded in memory starting at address 16_{10}. The program counter is set to 16_{10}, so the MPU is ready to begin executing the program.

The first fetch phase is illustrated in Figure 2-30. During this phase:

1. The contents of the program counter are loaded into the address register.
2. The program counter is incremented to 17_{10}.
3. The contents of the address register are placed on the address bus.
4. The contents of the selected memory location are transferred via the data bus to the data register.
5. The contents of the data register are decoded.
6. The MPU recognizes that an LDA direct operation is indicated. This concludes the fetch phase.



ADDRESS	BINARY CONTENTS	MNEMONICS/ DECIMAL CONTENTS
0001 0000	1001 0110	LDA
0001 0001	0001 0111	23
0001 0010	1001 1011	ADD
0001 0011	0001 1000	24
0001 0100	1001 0111	STA
0001 0101	0001 1001	25
0001 0110	0011 1110	HLT
0001 0111	0000 0111	7
0001 1000	0000 1010	10
0001 1001		

Figure 2-30
Fetching the Opcode of the First Instruction.

The execute phase has two parts when direct addressing is indicated. Figure 2-31 illustrates the first half of the execute phase. During this half:

1. The contents of the program counter are transferred to the address register. This number is the memory location that holds the address of the operand.

2. The program counter is incremented to 18_{10}.

3. The contents of the address register are placed on the address bus.

4. The contents of the selected memory location are placed on the data bus. However, in the direct addressing mode, this data is transferred to the address register. Thus, 23_{10} goes into the address register, replacing the previous contents. After this cycle, the address register will appear as shown in Figure 2-32.

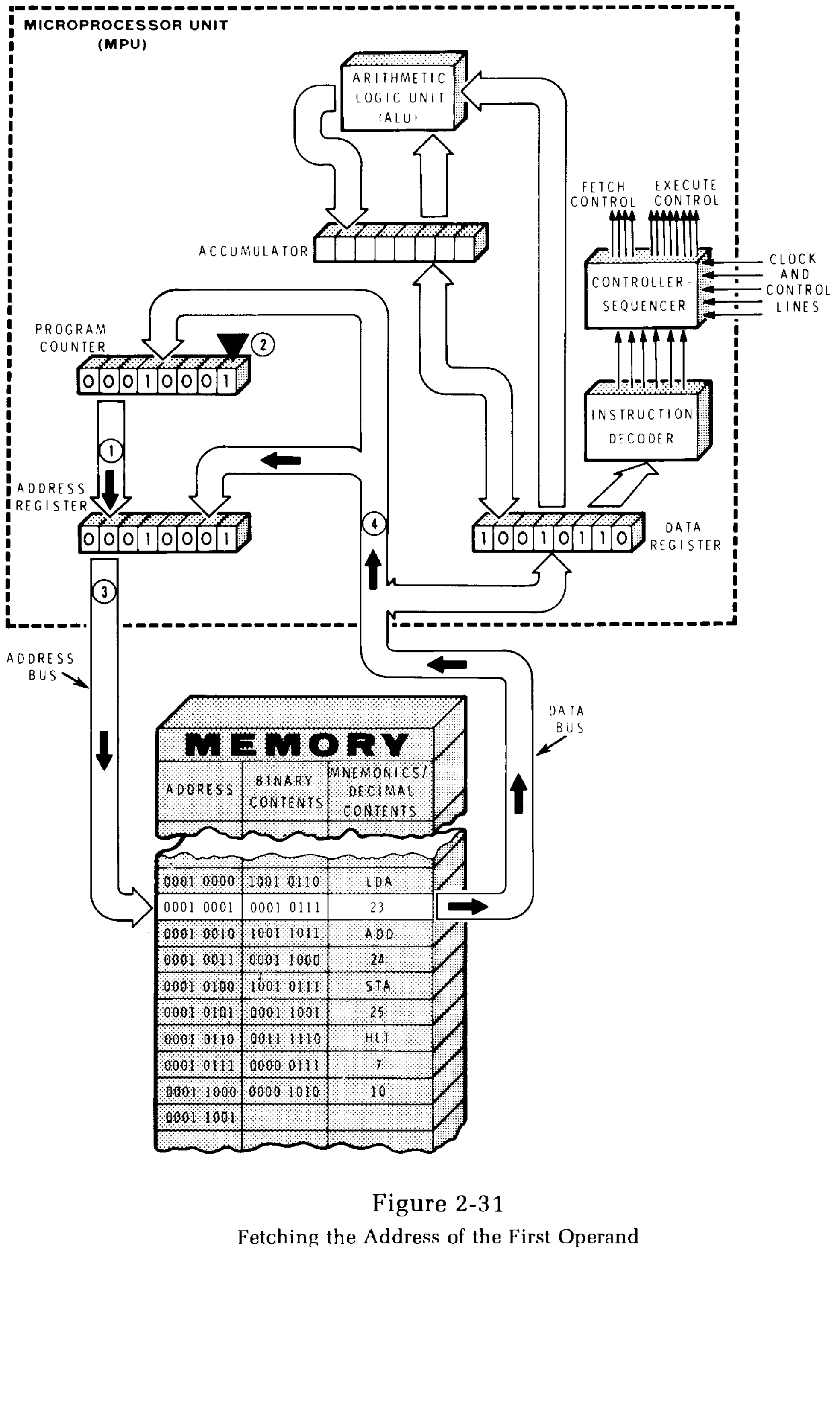


Figure 2-31
Fetching the Address of the First Operand

During the second half of the execute phase, the operand is loaded into the accumulator as shown in Figure 2-32. The procedure is:

1. The address of the operand which is in the address register is placed on the address bus.
2. The operand is read out of memory location 23_{10} and is transferred via the data bus to the data register.
3. The operand is transferred from the data register to the accumulator.

This completes the execution of the first instruction. Notice that the first operand (7_{10}) is now in the accumulator.

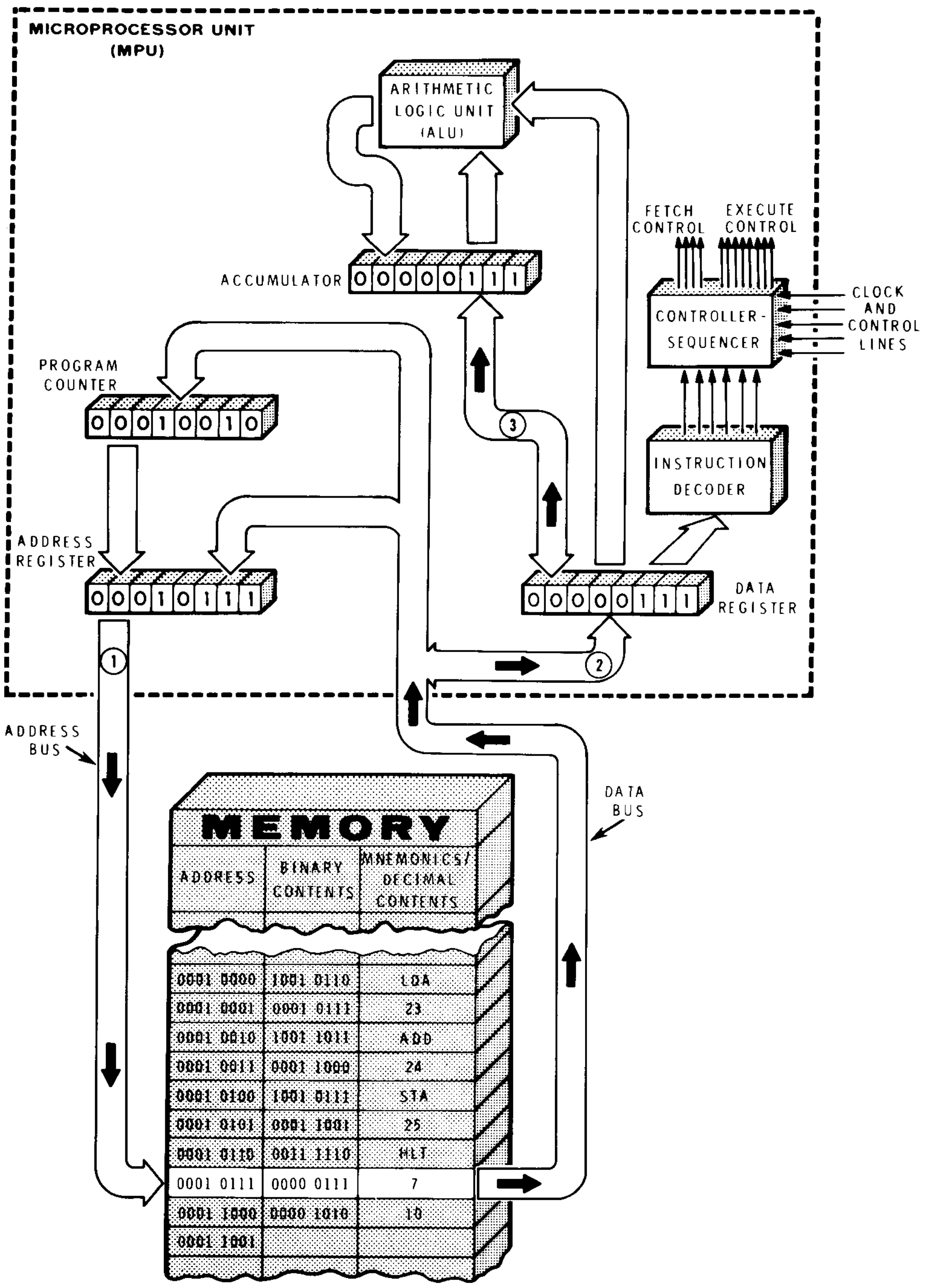
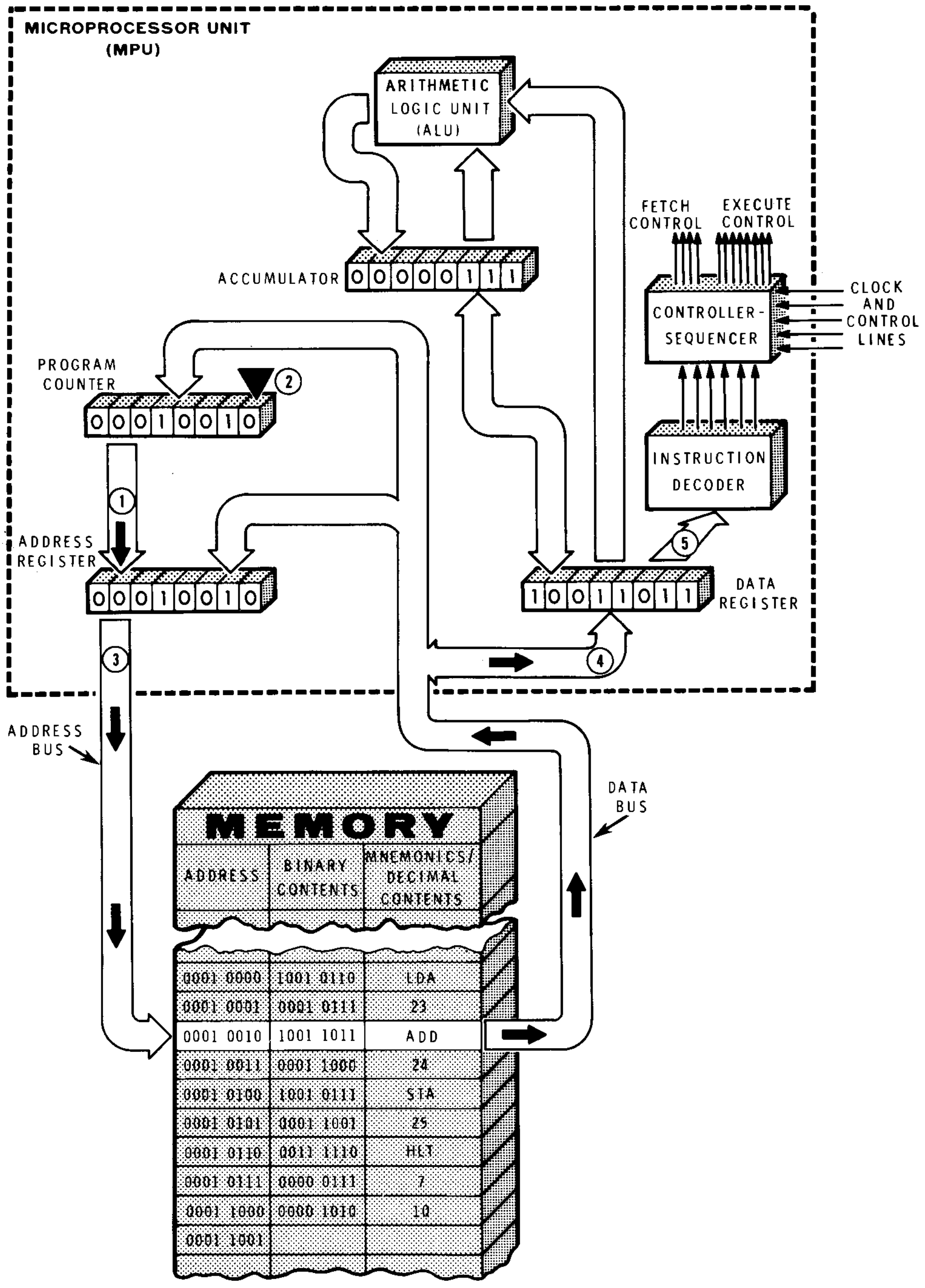


Figure 2-32
Fetching the First Operand.

The fetch phase for the second instruction is similar to that of the first. As shown in Figure 2-33, it causes the opcode of the ADD instruction to be read out of address 18_{10}. The opcode is transferred to the instruction decoder via the data bus and data register. In the process, the program counter is incremented to 19_{10}.



ADDRESS	BINARY CONTENTS	MNEMONICS/ DECIMAL CONTENTS
0001 0000	1001 0110	LDA
0001 0001	0001 0111	23
0001 0010	1001 1011	ADD
0001 0011	0001 1000	24
0001 0100	1001 0111	STA
0001 0101	0001 1001	25
0001 0110	0011 1110	HLT
0001 0111	0000 0111	7
0001 1000	0000 1010	10
0001 1001		

Figure 2-33
Fetching the Opcode of the Second Instruction.

The first half of the execute phase is illustrated in Figure 2-34. Here, the address of the second operand is read out of memory location 19_{10} and is placed in the address register.

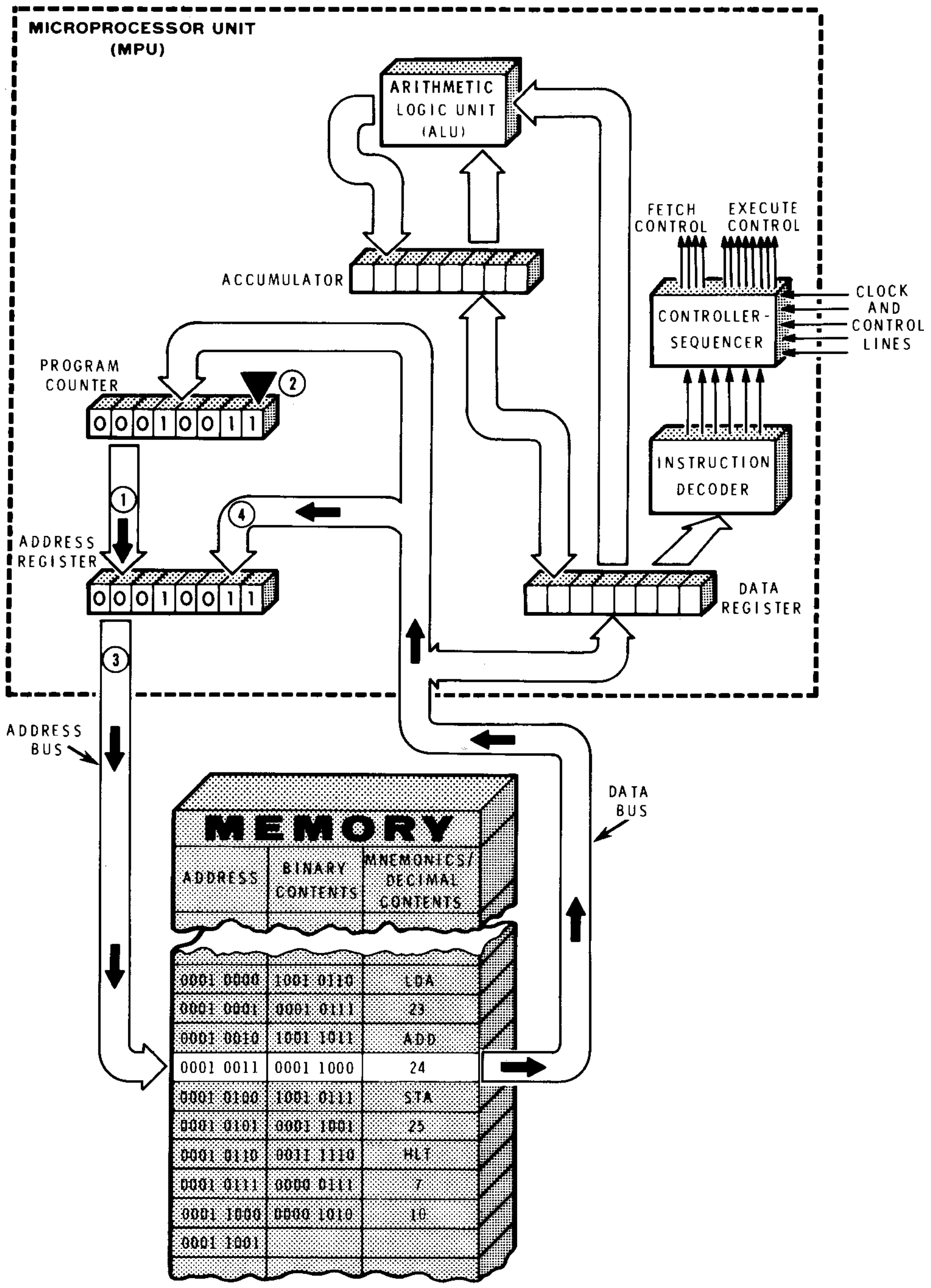
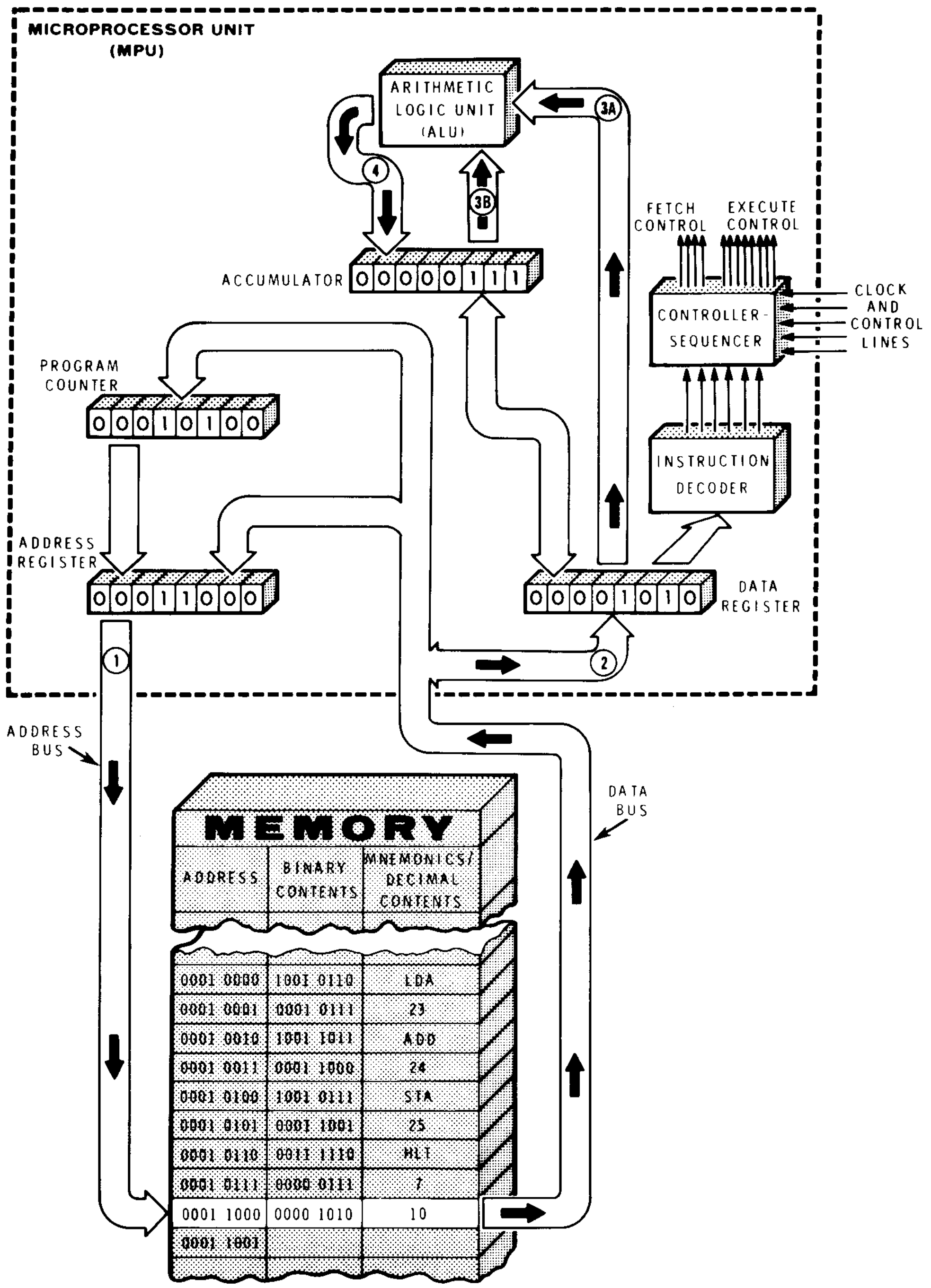


Figure 2-34
Fetching the Address of the Second Operand.

Figure 2-35 illustrates the second cycle of the execute phase. Here the address of the second operand is transferred from the address register to the address bus. The address is 24_{10}. Therefore, the contents of location 24_{10} are placed on the data bus and transferred to the data register. That is, the second operand 10_{10}, is loaded into the data register. Then, the operand from the data register is made available at one input to the ALU. Simultaneously, the first operand which has been waiting in the accumulator is made available at the other input to the ALU. The ALU adds the two operands together, producing a result of 17_{10}. This sum is put back in the accumulator, replacing the previous number.



ADDRESS	BINARY CONTENTS	MNEMONICS/ DECIMAL CONTENTS
0001 0000	1001 0110	LDA
0001 0001	0001 0111	23
0001 0010	1001 1011	ADD
0001 0011	0001 1000	24
0001 0100	1001 0111	STA
0001 0101	0001 1001	25
0001 0110	0011 1110	HLT
0001 0111	0000 0111	7
0001 1000	0000 1010	10
0001 1001		

Figure 2-35
Adding the Two Operands.

Now all that remains is to place the sum in memory. This is done by the STA 25_{10} instruction. Since this is the next instruction in sequence, it will be fetched and executed next. The fetch phase is illustrated in Figure 2-36. It ends with the STA opcode being decoded.

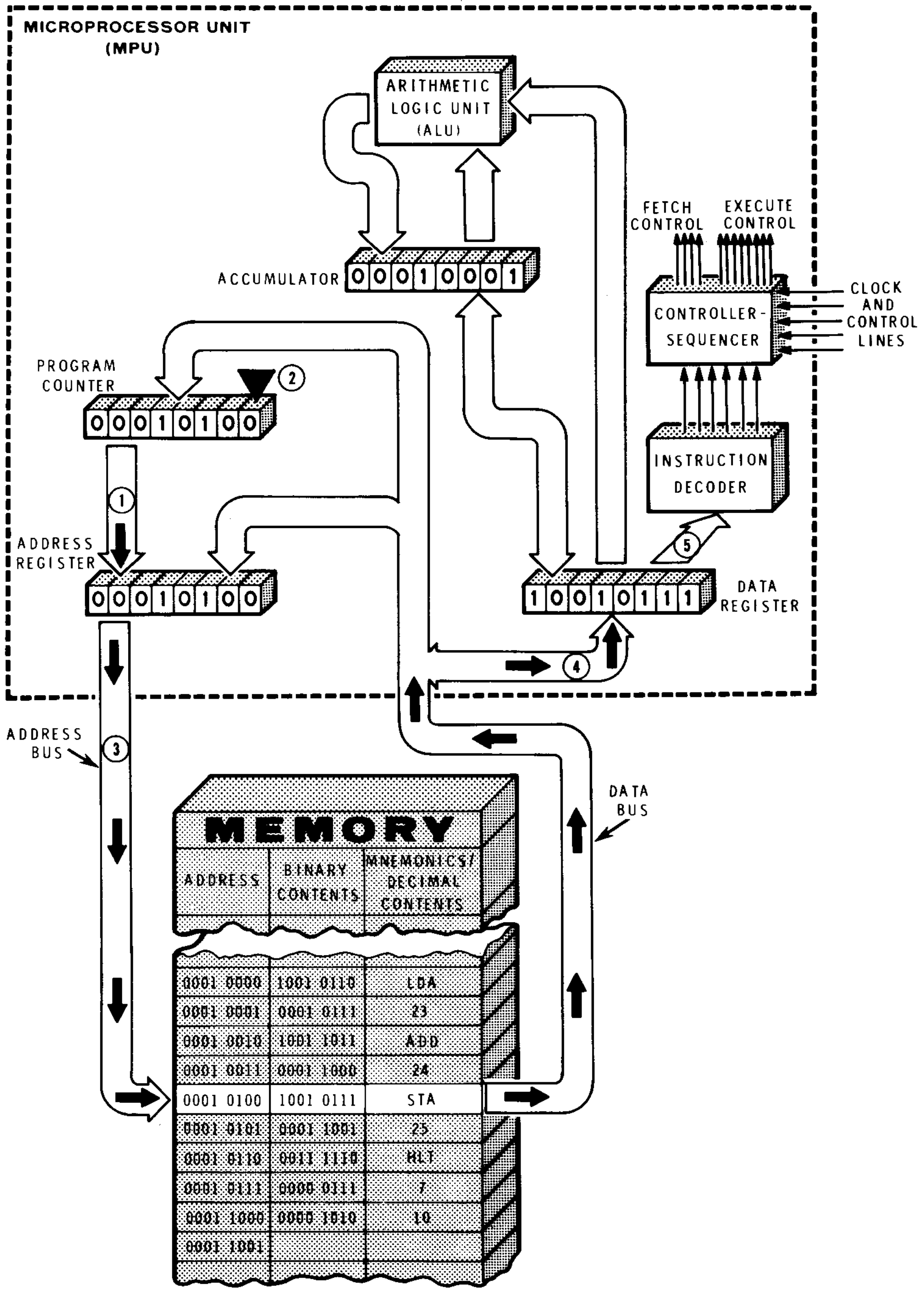


Figure 2-36
Fetching the Third Opcode.

The first half of the execution phase of the STA instruction involves loading the address of the storage location into the address register. Figure 2-37 illustrates that this four-step procedure is identical to that performed for the previous two instructions. It ends with the address 25_{10} in the address register.

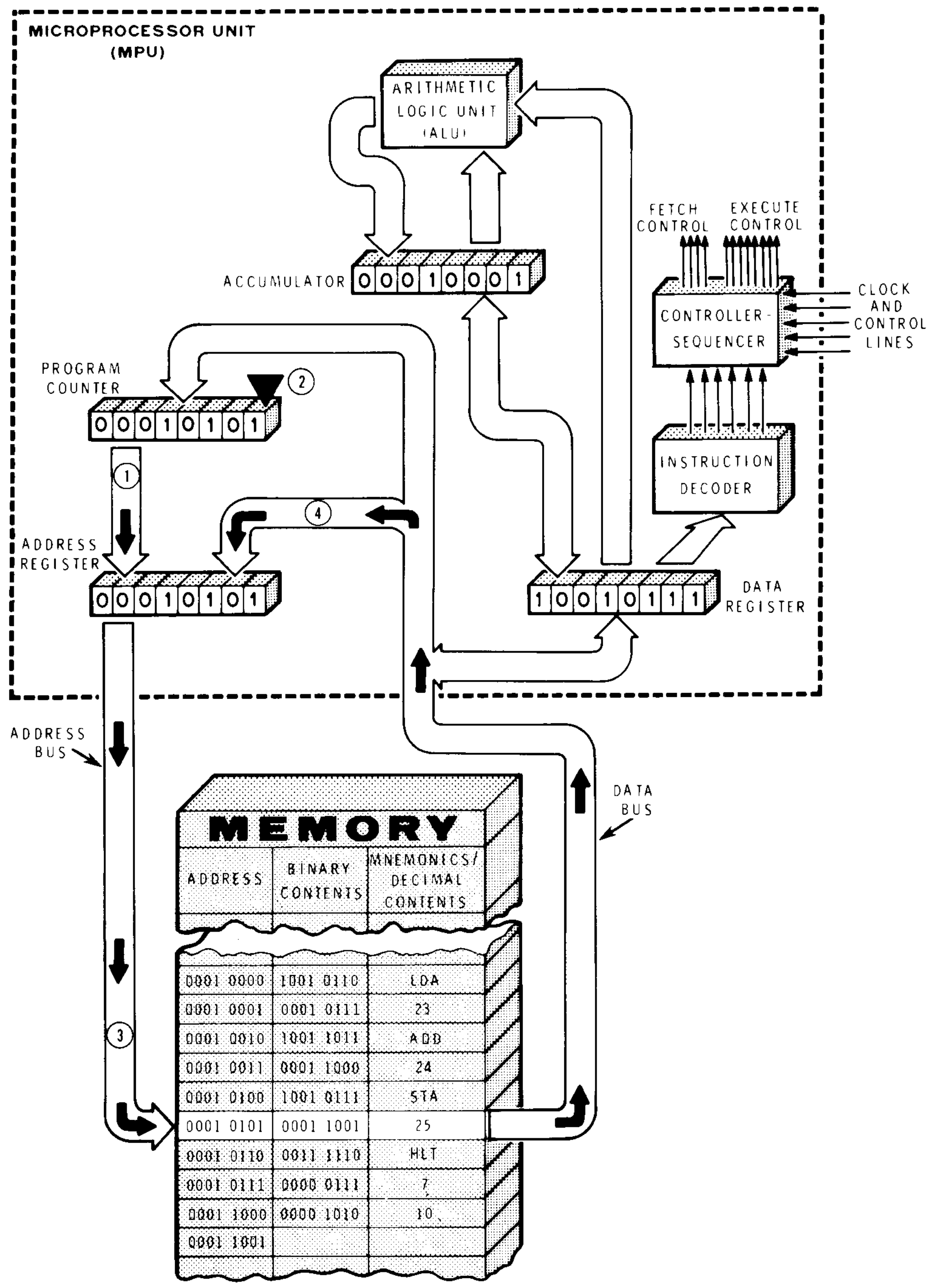


Figure 2-37
Fetching the Third Address.

During the final half of the execute phase, the contents of the accumulator are transferred to the data register and are then stored in the selected memory location. We have not yet discussed this operation in detail. Therefore, the step-by-step procedure is presented below. Refer to Figure 2-38 for the following stèps:

1. The contents of the accumulator (17_{10}) are transferred to the data register. At this point, the number 17_{10} exists in both the accumulator and the data register.
2. The address at which this data is to be stored is placed on the address bus.
3. The contents of the data register are placed on the data bus.
4. The number on the data bus is written into the selected memory location. That is, 17_{10} is written into memory location 25_{10}.

Notice that, after this operation, the number 17_{10} appears at memory location 25_{10}, but it also appears in the accumulator. Thus, the number is merely "copied" into memory. It is also important to note that the previous contents of memory location 25_{10} are lost whenever you write new data into this location. For this reason, you must be certain that you do not write into a location that contains an instruction or some byte of data that you will need later.

The program has now accomplished its goal. It has added 10 to 7 and has stored the sum back in memory. The last step in the program is the HLT instruction. The MPU fetches and executes this instruction next. The fetch and execute sequence for this instruction were discussed earlier and need not be repeated here.

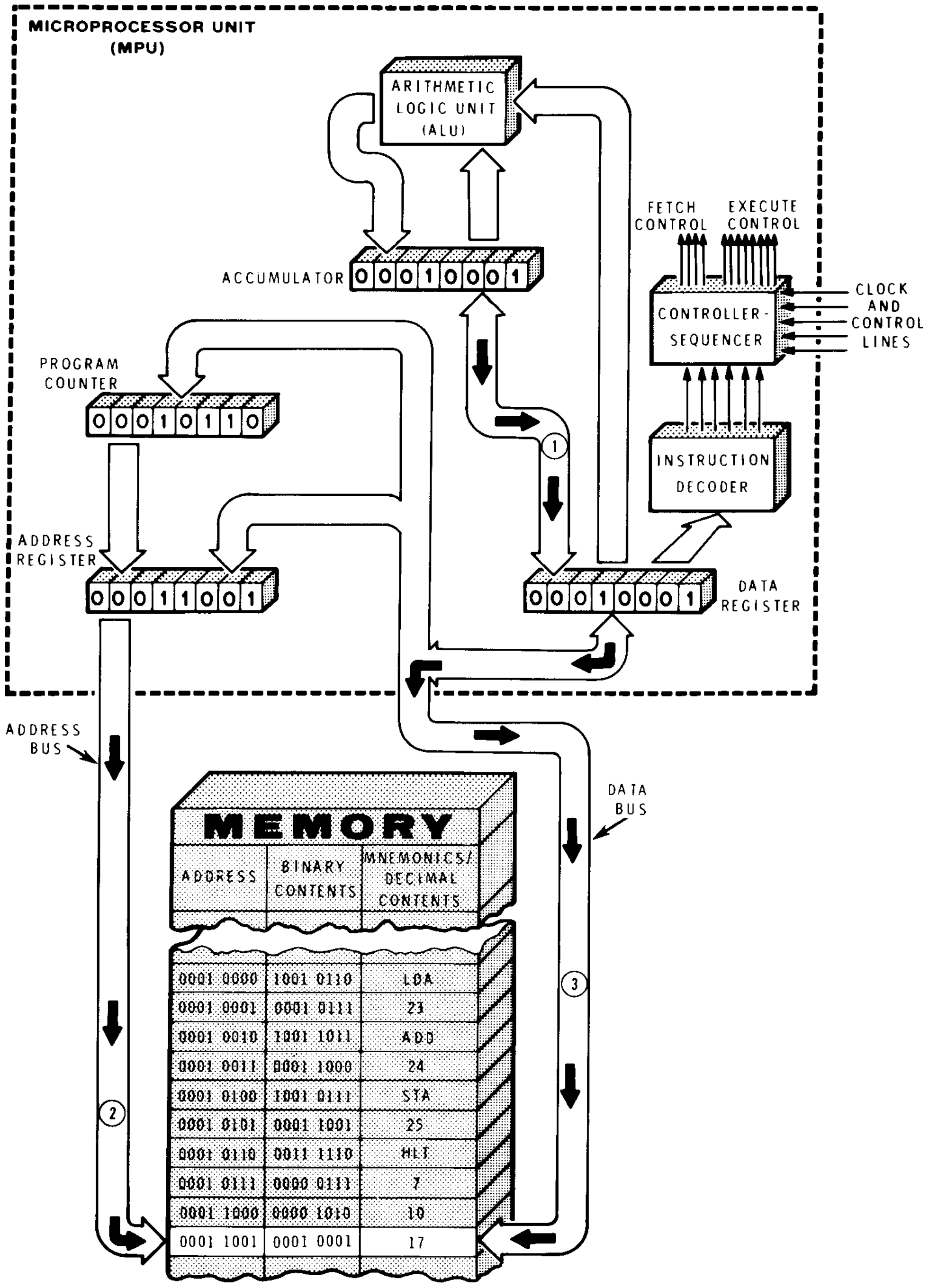

MICROPROCESSOR UNIT (MPU)
ARITHMETIC LOGIC UNIT (ALU)
FETCH CONTROL
EXECUTE CONTROL
ACCUMULATOR 0 0 0 1 0 0 0 1
CONTROLLER-SEQUENCER
CLOCK AND CONTROL LINES
PROGRAM COUNTER 0 0 0 1 0 1 1 0
INSTRUCTION DECODER
1
ADDRESS REGISTER 0 0 0 1 1 0 0 1
DATA REGISTER 0 0 0 1 0 0 0 1
ADDRESS BUS
DATA BUS
MEMORY
ADDRESS | BINARY CONTENTS | MNEMONICS/DECIMAL CONTENTS
0001 0000 | 1001 0110 | LDA
0001 0001 | 0001 0111 | 23
0001 0010 | 1001 1011 | ADD
0001 0011 | 0001 1000 | 24
0001 0100 | 1001 0111 | STA
0001 0101 | 0001 1001 | 25
0001 0110 | 0011 1110 | HLT
0001 0111 | 0000 0111 | 7
0001 1000 | 0000 1010 | 10
0001 1001 | 0001 0001 | 17
2
3

Figure 2-38
Storing the Sum.

Combining Addressing Modes

When writing programs, you can use the addressing mode that best suits your application. For example, the program that was just discussed can be shortened by using the immediate addressing mode with the first two instructions. Figure 2-39 compares two programs that do the same job.

Using direct addressing only, the program required ten bytes of memory. Its execution requires eleven MPU cycles. If you use immediate addressing for the first two instructions, the program requires eight bytes of memory. Furthermore, it can be executed in nine MPU cycles. Everything else being equal, the second approach would probably be preferred.

A. USING DIRECT ADDRESSING

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
00	96	LDA	Load accumulator direct with operand 1
01	07	07_{10}	which is stored at this address.
02	9B	ADD	Add to accumulator direct with operand 2
03	08	08_{10}	which is stored at this address.
04	97	STA	Store the sum
05	09	09_{10}	at this address.
06	3E	HLT	Stop
07	21	33_{10}	Operand 1
08	17	23_{10}	Operand 2
09	—	—	Reserved for sum.

B. COMBINING ADDRESSING MODES

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
00	86	LDA	Load accumulator immediately with
01	21	33_{10}	Operand 1.
02	8B	ADD	Add to accumulator immediately with
03	17	23_{10}	Operand 2.
04	97	STA	Store the sum
05	07	07_{10}	at this address.
06	3E	HLT	Stop
07	—	—	Reserved for sum.

Figure 2-39
By Combining the Addressing Modes,
We Can Save Memory Space And
Computer Time.

Self-Test Review

41. What addressing mode is used by single byte instructions?

42. In the immediate addressing mode, what is the second byte of the instruction?

43. In the direct addressing mode, what is the second byte of the instruction?

44. In all addressing modes, what is the first byte of the instruction?

45. Define MPU cycle.

46. Which of the three addressing modes discussed so far requires the longest execution time?

47. Refer to Figure 2-39A. What number is loaded into the accumulator by the first instruction?

48. What number is added to the accumulator by the second instruction?

49. When the computer halts, what number will be in memory location 09?

50. Refer to Figure 2-39B. When the computer halts, what number will be in memory location 07?

Answers

41. Inherent or implied.

42. The operand.

43. The address of the operand.

44. The opcode.

45. An MPU cycle is the time required to fetch a byte from memory.

46. The direct addressing mode.

47. 21_{16} or 33_{10}.

48. 17_{16} or 23_{10}.

49. 38_{16} or 56_{10}.

50. 38_{16} or 56_{10}.

UNIT 2 SUMMARY

1. A microprocessor is a logic device that is used in digital electronic systems.
2. The microprocessor is the "brains" of the microcomputer. It performs arithmetic, logic, and control operations.
3. The microcomputer is composed of RAM, ROM, a clock, and an I/O interface.
4. Parallel conductors within the microcomputer that carry address and data information are called a bus.
5. The microcomputer communicates with the "outside world" through one or more I/O ports.
6. The operation of the microprocessor is controlled by a list of instructions called a program.
7. One who writes these instructions is called a programmer.
8. The computer program is stored outside of the computer and later loaded into memory when needed to perform an operation. This is known as the stored program concept.
9. A computer word is a group of binary bits. It is the fundamental unit of information used in the computer.
10. The most common word length is eight bits. Thus, an 8-bit binary word can have any one of 256_{10} unique values. It can also represent any of the 128 ASCII code characters.
11. Some microprocessors use a 4-bit word length, while others may use a 12- or 16-bit word length.
12. A 16-bit word can represent $65{,}536_{10}$ unique values.
13. The 8-bit word length defines the size of many different components in the microprocessor/microcomputer system. Many important microprocessor registers have an 8-bit capacity, and the bus which is used to transfer data words consists of eight parallel conductors.

14. Microprocessors that use 16-bit words also use 8-bit segments of data. These 8-bit segments are called bytes.

15. A 16-bit word is composed of two 8-bit bytes. The first eight bits in a 16-bit word is called the low byte, while the last eight bits is called the high byte.

16. The bits in an 8-bit byte or word are numbered from 0 through 7, with 0 being the least significant bit. In a 16-bit word, the bits are numbered from 0 through 15.

17. The microprocessor (MPU) contains many specialized circuits. They include the data register, the instruction decoder, the controller-sequencer, arithmetic logic unit (ALU), program counter, address register, and accumulator.

18. The ALU performs arithmetic or logic operations on the data supplied to the MPU. The data supplied to the ALU are called operands.

19. The accumulator is a specialized register that performs two functions. First, it is used to hold an operand before it is processed by the ALU. Second, it is used to store the result of the ALU process.

20. Specific MPU instructions load data into the accumulator from memory and store data from the accumulator into memory.

21. The data register is a temporary storage location for data going to or coming from the data bus.

22. The address register is a temporary storage location that holds the address of the memory location or I/O port that is used in the operation that is presently being performed.

23. The program counter controls the sequence the instructions in a program are executed.

24. The instruction decoder decodes the instruction pulled from memory to determine the operation the MPU will perform.

25. The controller-sequencer determines the sequence of events necessary to complete the operation described by the program instruction.

26. Memory is a series of locations outside the MPU where program data and instructions are stored. If the MPU operates on 8-bit words, each memory location contains storage for eight bits of data.

27. Memory size is identified by the number of storage locations and the number of data bits that can be stored in one location. A 256 $\times$ 8 memory contains 256_{10} storage locations, each eight bits wide.

28. Two buses connect memory to the MPU. One bus is used to transfer data to and from the MPU, the other is used by the MPU to address a specific memory location.

29. Reading the contents of a memory will not destroy the contents of that memory location. This is known and nondestructive read-out (NDRO).

30. You can read from or write to random access memory (RAM).

31. You can only read from read only memory (ROM).

32. When executing a program, the MPU goes through a fundamental sequence of steps or phases called the fetch-execute sequence.

33. In the fetch phase, the MPU reads and decodes an instruction from memory.

34. In the execute phase, the MPU performs the operation described by the instruction. The time to perform the operation will vary with the operation.

35. A mnemonic is an abbreviation or memory aid used by a programmer for identifying an instruction.

36. An opcode is the number or value assigned to mnemonic. Each mnemonic has its own opcode. This is the information that is decoded by the MPU.

37. An instruction operand is the data that the MPU will process during the execution of an instruction.

38. The time required for an instruction to be fetched and executed is often given in MPU cycles.

39. An instruction addressing mode refers to the method the MPU uses to address an instruction's operand.

40. In the inherent or implied addressing mode, the instruction has no operand or the operand is implied by the instruction opcode.

41. In the immediate addressing mode, the operand is stored in the next sequential memory location following the instruction opcode.

42. In the direct addressing mode, the next sequential memory location does not contain the operand. Rather, it contains the address of the operand, or the address of the memory location where the operand will be stored.

43. Many instructions can be written using different addressing modes. That is, you can use immediate addressing to add a value to the accumulator or you can use direct addressing to add a value to the accumulator.

Perform Experiment 3 at the end of Book 2.

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.

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
Augend:	31863
Addend:	+ 56719
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
Augend:     1101
Addend:   + 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
Augend:	10110101
Addend:	+ 10001111
Sum:	101000100

Carry:	11111000
Augend:	11001100
Addend:	+ 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 addend 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.

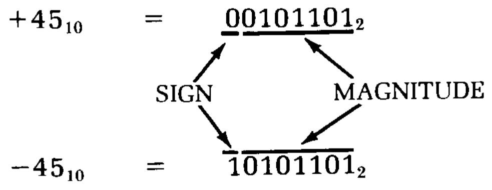
```
                 000101     Quotient
Divisor    1000 )101000     Dividend
                 1000↓↓
                 ------
                   1000     Remainder
                   1000
                   ----
                      0     Remainder

                 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:



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.

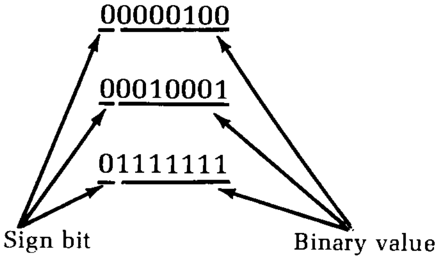
While this method of representing negative numbers may seem logical, its popularity was short-lived. Because it required complex and slow arithmetic circuitry, it was abandoned long before microprocessors were invented.

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:

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

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:

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

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:

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

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
Augend:                       10011011
Addend:                     + 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
Divisor:   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:

$$\begin{array}{r} 00011001 \\ \underline{1} \\ 00011010 \end{array}$$

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:

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

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

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

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:

Minuend:	973
Subtrahend:	−654
Difference:	319

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

$$\underset{\uparrow}{1}319 \quad \text{Discard final carry}$$

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:

$$\begin{array}{llr} & \text{Minuend:} & 01000101 \\ \text{Two's complement of} & \text{Subtrahend:} & \underline{+11100110} \\ & \text{Difference:} & 100101011 \end{array}$$

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:

$$\begin{array}{rr} \underline{0}0000111 & +7 \\ \underline{+\underline{0}0000011} & \underline{+\ 3} \\ \underline{0}0001010 & +10 \end{array}$$

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:

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

Discard final carry

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:

Augend:	00001000	(+8)
Addend:	+ 11110111	+ (−9)
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:

```
Augend:      11111100       (−4)
Addend:    + 11111101     + (−3)
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}.

```
               11101110      (− 18)
             + 10001000    + (−120)
              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 • 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.

$1 \cdot 1 = 1$ MSB
$0 \cdot 1 = 0$
$0 \cdot 0 = 0$
$1 \cdot 0 = 0$
$1 \cdot 0 = 0$
$1 \cdot 1 = 1$
$0 \cdot 1 = 0$
$1 \cdot 0 = 0$ 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 $\odot$) 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. •

   B. +

   C. ⊕

   D. ×

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. $\overline{\mathbf{A}}$ 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.

UNIT 3 SUMMARY

1. A number system can be used to perform two basic operations: addition and subtraction. From these, you can then perform multiplication, division, and any other numerical operation.

2. The addend is the number that is added to another number.

3. The augend is the number to which the addend is added.

4. The sum the result of adding the addend to the augend.

5. When performing decimal addition, you add the addend to the augend in each column of numbers beginning with the column that contains the LSD. If the sum of any column produces a value greater than nine, you add the carry to the next column on the left.

6. Binary addition is similar to decimal addition. There are four rules you should follow:

   A. Adding zero to zero produces the sum of zero.
   B. Adding zero to one produces the sum of one.
   C. Adding one to one produces the sum of zero with a carry of one.
   D. Adding one to one to one produces the sum of one with a carry of one.

7. The subtrahend is the number that is subtracted from another number.

8. The minuend is the number from which the subtrahend is subtracted.

9. The difference is the result of subtracting the subtrahend from the minuend.

10. When performing decimal subtraction, you subtract the subtrahend from the minuend in each column of numbers beginning with the column that contains the LSD. If the minuend of any column is smaller than the subtrahend, one is borrowed from the minuend in the column on the left, reducing that minuend by one, and ten is added to the minuend that required the borrow.

11. Binary subtraction is similar to decimal subtraction. There are four rules you should follow:

    A. Subtracting zero from zero produces the difference of zero.
    B. Subtracting one from one produces the difference of zero.
    C. Subtracting zero from one produces the difference of one.
    D. Subtracting one from zero produces the difference of one with a borrow of one.

12. The multiplicand is the number that is added to itself.

13. The multiplier is the number that specifies how many times the multiplicand is added to itself.

14. The product is the result of the multiplication.

15. Multiplication is a short method for adding a multiplicand to itself as many times as it is specified by the multiplier.

16. Binary multiplication is a process of shift left and add.

17. There are four rules you must follow when multiplying binary numbers:

    A. Multiplying zero times zero gives the product zero.
    B. Multiplying zero times one gives the product zero.
    C. Multiplying one times zero gives the product zero.
    D. Multiplying one times one gives the product one.

18. The dividend is the number that is divided in a division operation.

19. The divisor is the number that is used to divide the dividend.

20. The quotient is the number of times the divisor will divide into the dividend.

21. The remainder is what is left of the dividend after the last division.

22. Division is a short method for subtracting a divisor from a dividend until what is left of the dividend is less than the divisor or zero. The number of subtractions is recorded as the quotient. The value remaining after the subtraction is the remainder.

23. Binary division is a process of shift right and add.

24. The rules of binary division are the same as the rules for binary subtraction.

25. The sign and magnitude of a number can be represented in binary.

26. A signed binary number is normally eight bits in length. The sign of the number is represented by the MSB. This is called the sign bit.

27. In the sign and magnitude method of representing a value, the sign bit is zero for positive and one for negative. The remaining seven bits represent the magnitude of the value.

28. In the one's complement method of representing a signed value, the positive values are the same as in the sign and magnitude method. Negative values are represented by first presenting the value as a positive value, and then inverting all of the bits, including the sign bit.

29. In the two's complement method of representing a signed value, the positive values are the same as in the one's complement method. Negative values are represented by first obtaining the one's complement of the value, and then adding one to the value. This is the method used by the microprocessor to represent a signed value.

30. The maximum value that can be represented by a two's complement binary number is $+127_{10}$ or -128_{10}.

31. The microprocessor can only add binary numbers. To subtract one number from another, the subtrahend must be converted to its negative two's complement form. Then the minuend is added to the subtrahend. Any carries into the ninth bit are discarded.

32. Boolean operations include AND, OR, Exclusive OR, and Invert.

33. The AND operation produces the logical product of two or more binary variables. The possible results are:

    A. Zero AND zero equals zero.
    B. Zero AND one equals zero.
    C. One AND zero equals zero.
    D. One AND one equals one.

34. The OR operation produces the logical sum of two or more binary variables. The possible results are:

    A. Zero OR zero equals zero.
    B. Zero OR one equals one.
    C. One OR zero equals one.
    D. One OR one equals one.

35. The Exclusive OR (EOR or XOR) operation produces the logical test for "equalness" between two binary variables. The possible results are:

    A. Zero XOR zero equals zero.
    B. Zero XOR one equals one.
    C. One XOR zero equals one.
    D. One XOR one equals zero.

36. The Invert operation performs a direct complement of a single binary variable.

Perform Experiment 4 at the end of Book 2.

Unit 4

INTRODUCTION TO PROGRAMMING

INTRODUCTION

In the final analysis there are only two things you can do with a microprocessor. You can program it and you can interface it with the outside world. In this course, you learn to program the microprocessor first. This unit, along with the associated experiments, will serve as an introduction to programming.

The programs you encounter in this unit are simple enough that anyone can understand them, and yet they illustrate many important concepts. By studying these programs, you will develop an understanding of how the microprocessor handles complex tasks. At the same time, you will gain practice using the instruction set.

UNIT OBJECTIVES

When you have completed this unit, you will be able to:

1. Draw the symbols used in flow charting and explain the purpose of each.
2. Develop flow charts that illustrate step-by-step procedures for solving simple problems.
3. Explain the purpose of conditional and unconditional branching.
4. Using the block diagram of the hypothetical microprocessor, trace the data flow during the execution of a branch instruction.
5. Compute the proper relative address for branching forward or backward from one point to another in a program.
6. Explain the purpose of the carry, negative, zero, and overflow flags. Give an example of a situation that can cause each to be set and another example that will cause each to clear. List eight instructions that test one of these flags.
7. Write programs that can: multiply by repeated addition; divide by repeated subtraction; convert binary to BCD; convert BCD to binary; add multiple-precision numbers; subtract multiple-precision numbers; add BCD numbers.

BRANCHING

The programs discussed earlier were all "straight line" programs; the instructions were executed one after another in the order in which they were written. Programs of this type are extremely limited because they use only a fraction of the microprocessor's power.

The real power of the microprocessor comes from its ability to execute a section of a program over and over again. In an earlier program we saw that two numbers could be multiplied by repeated addition. As long as the numbers are very small and we know the value of the two numbers, we can write a "straight line" program to multiply the numbers. For example, 5 could be multiplied by 4 with the following program:

Address	Instruction/Data	Comments
00	LDA 09	Load direct
02	ADD 09	Add direct
04	ADD 09	Add direct
06	ADD 09	Add direct
08	HLT	
09	05	

This technique is very crude for a number of reasons. If the two numbers are large, such as 98 and 112, the number of ADD instructions becomes excessive. Moreover, the values of the two numbers to be multiplied are generally not known. Therefore, even if we were willing to write enough ADD instructions, we simply would not know how many to write. Obviously, some better technique must be available.

A technique that is used in virtually every program is called **looping.** This allows a section of the program to be run as often as needed. Every microprocessor has a group of instructions called JUMP or BRANCH instructions that allow it to execute these program loops. These allow the microprocessor to escape the normal instruction sequence.

The microprocessor discussed in this course has both jump and branch instructions. In this unit, we will confine our discussion to the branch instructions. In a later unit we will discuss the jump instructions.

Before discussing the types of branch instructions, we must first discuss a new addressing mode called relative addressing.

Relative Addressing

In previous units, we discussed immediate addressing and direct addressing. Recall that in the immediate addressing mode no address is specified. The data is assumed to be the byte following the opcode. In direct addressing, an address is given. The data is assumed to be at that address.

Branch instructions are somewhat different from the instructions discussed earlier. While the branch instruction has an address associated with it, the address does not indicate the location of data. Instead, the address indicates the location of the next instruction that is to be executed.

The format of the branch instruction is shown in Figure 4-1. All branch instructions are 2-byte instructions. The first byte is the 8-bit opcode. This code identifies the particular type of branch instruction. As you will see later, a microprocessor may have a dozen or more different branch instructions. Each has its own opcode that uniquely identifies it.

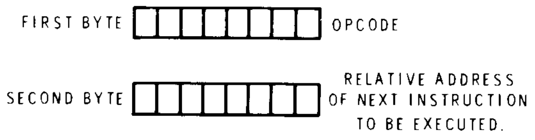


Figure 4-1
Format of the branch instruction.

The second byte of the branch instruction indicates the point to which the program is to branch. That is, it specifies the address of the next instruction that is to be executed.

In some microprocessors, the address is absolute. That is, the address is the memory location that holds the next instruction. In this case, the instruction BRANCH 30_{16} would mean that the instruction to be executed next is at address 30_{16}. In other words, some microprocessors use direct addressing when branching.

Our hypothetical microprocessor uses a different technique called relative addressing. In this addressing mode, the byte following the opcode does not represent an absolute address. Instead, it is a number that must be added to the program counter to form the new address. Consider the instruction:

BRANCH 30_{16}

Using relative addressing, this does not mean that the next instruction is to be taken from memory location 30_{16}. Rather, it means that 30_{16} must be added to the present contents of the program counter. Thus, if the program counter is at 08_{16} when the BRANCH 30_{16} instruction is executed, the next instruction will be fetched from location $08_{16} + 30_{16} = 38_{16}$.

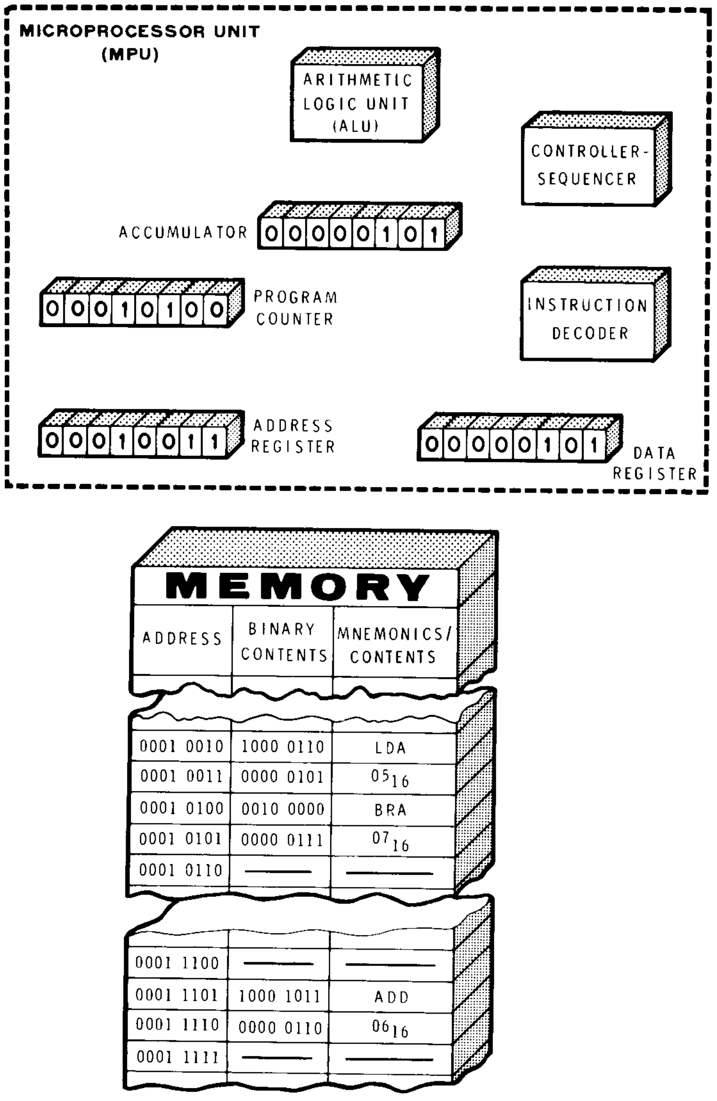
By the same token, if the contents of the program counter is FA_{16} when a BRANCH 03 is encountered, the next instruction will be fetched from location $FA_{16} + 03 = FD_{16}$. Notice that this allows the MPU to jump over the instructions at addresses FB_{16} and FC_{16}.

Executing a Branch Instruction

Determining the relative address to use as the second byte of the branch instruction can be confusing unless you keep in mind the method by which the MPU executes a program. Therefore, let's go through the manipulations that take place within the MPU during the execution of the branch instruction.

Figure 4-2 shows sections of a program stored in memory. Let's assume that the MPU has been executing this program. Let's further assume that the MPU just completed the execution of the LDA 05 instruction at addresses 12_{16} and 13_{16}. The address register still holds the address of the last byte that was read from memory. The accumulator and data register hold the contents (05) of the last location that was read out.

Notice that the program counter contains the address of the next instruction to be executed. This address points to the branch instruction in memory location 14_{16}. Let's pick up the action at this point.


MICROPROCESSOR UNIT (MPU)
ARITHMETIC LOGIC UNIT (ALU)
CONTROLLER-SEQUENCER
ACCUMULATOR 0 0 0 0 0 1 0 1
0 0 0 1 0 1 0 0 PROGRAM COUNTER
INSTRUCTION DECODER
0 0 0 1 0 0 1 1 ADDRESS REGISTER
0 0 0 0 0 1 0 1 DATA REGISTER
MEMORY

ADDRESS	BINARY CONTENTS	MNEMONICS/ CONTENTS
0001 0010	1000 0110	LDA
0001 0011	0000 0101	05_{16}
0001 0100	0010 0000	BRA
0001 0101	0000 0111	07_{16}
0001 0110	———	———
0001 1100	———	———
0001 1101	1000 1011	ADD
0001 1110	0000 0110	06_{16}
0001 1111	———	———

Figure 4-2
Status of the MPU registers after executing the LDA 05 instruction.

Figure 4-3 shows how the first byte of the branch instruction is fetched. This is the standard fetch operation that was discussed earlier:

1. The address (14_{16}) is transferred from the program counter to the address register.
2. The program counter is incremented to 15_{16}.
3. The address is strobed onto the address bus.
4. The contents of the selected memory location are transferred via the data bus to the data register.
5. The instruction decoder examines this opcode and finds it to be a branch instruction.

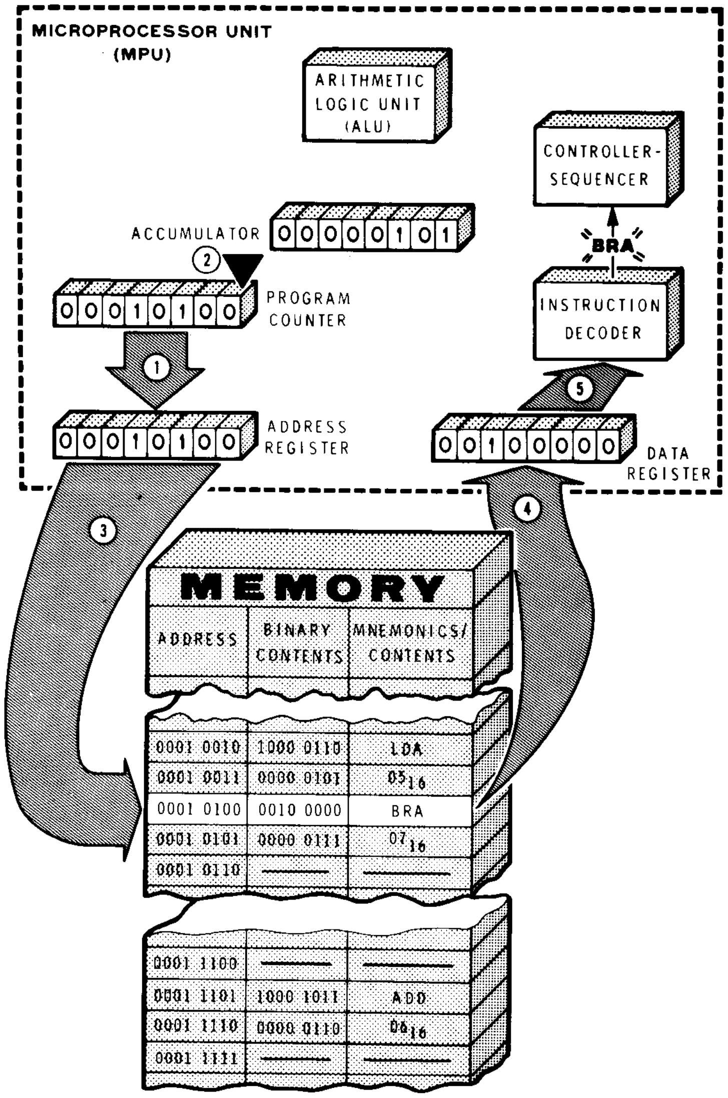


Figure 4-3
Fetching the BRA instruction.

Therefore, the controller-sequencer starts the procedure for executing a branch instruction.

During the next machine cycle, the relative address is fetched. This procedure is shown in Figure 4-4. The major events are:

1. The address (15_{16}) is transferred from the program counter to the address register.
2. The program counter is incremented to 16_{16}.
3. The address (15_{16}) is strobed onto the address bus.
4. The contents of location 15_{16} are transferred to the data register via the data bus.

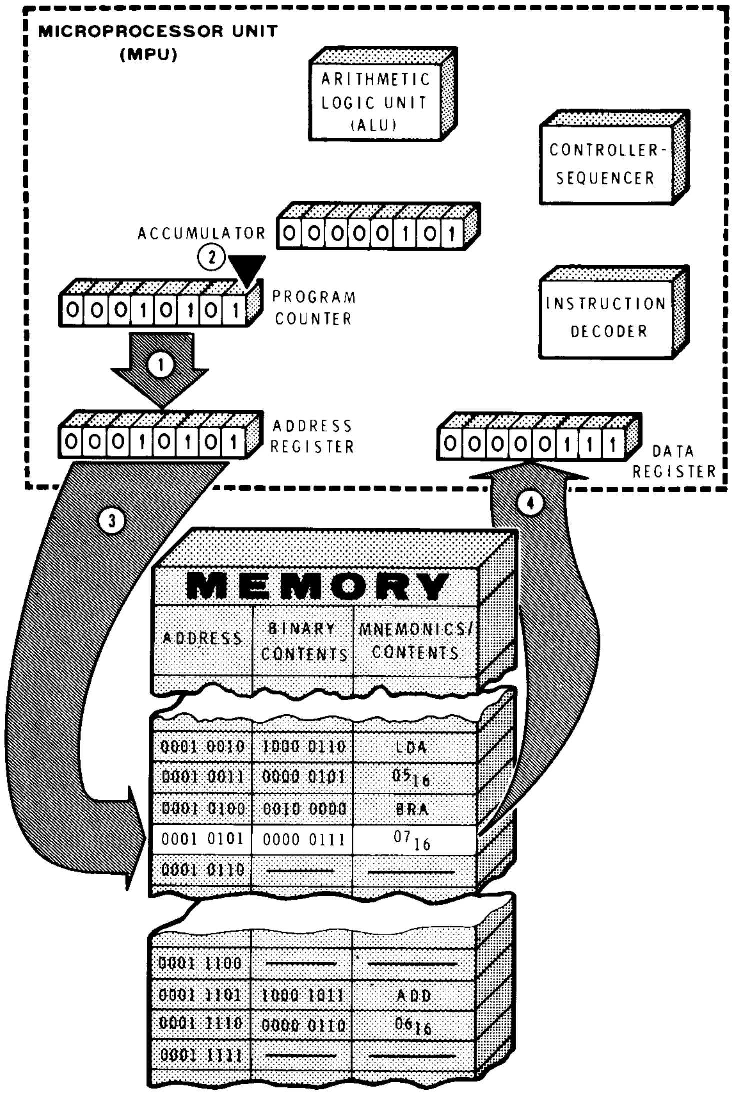

MICROPROCESSOR UNIT (MPU)
ARITHMETIC LOGIC UNIT (ALU)
CONTROLLER-SEQUENCER
ACCUMULATOR 0 0 0 0 0 1 0 1
2
0 0 0 1 0 1 0 1 PROGRAM COUNTER
INSTRUCTION DECODER
1
0 0 0 1 0 1 0 1 ADDRESS REGISTER
0 0 0 0 0 1 1 1 DATA REGISTER
3
4
MEMORY
ADDRESS
BINARY CONTENTS
MNEMONICS/CONTENTS
0001 0010 | 1000 0110 | LDA
0001 0011 | 0000 0101 | 05 16
0001 0100 | 0010 0000 | BRA
0001 0101 | 0000 0111 | 07 16
0001 0110
0001 1100
0001 1101 | 1000 1011 | ADD
0001 1110 | 0000 0110 | 06 16
0001 1111

Figure 4-4
Fetching the relative address.

Figure 4-5 shows the state of the various registers after the relative address is fetched. The relative address (07_{16}) is in the data register. Now look at the program counter. Notice that it points to address 16_{16}. However, the MPU has not yet finished executing the branch instruction. It must now compute the new address by adding the relative address to the program count. It uses the addition capabilities of the ALU to perform this function. That is, the program count and relative address are strobed into the ALU. The ALU adds the two together and produces a sum of

$$\begin{array}{ll} 0001\ \ 0110 & \text{program count} \\ \underline{0000\ \ 0111} & \text{relative address} \\ 0001\ \ 1101 & \text{new program count} \end{array}$$

This sum is loaded into the program counter. Thus, the next instruction is fetched from memory location $1D_{16}$. That is, the next instruction to be executed is the ADD 06_{16} instruction.

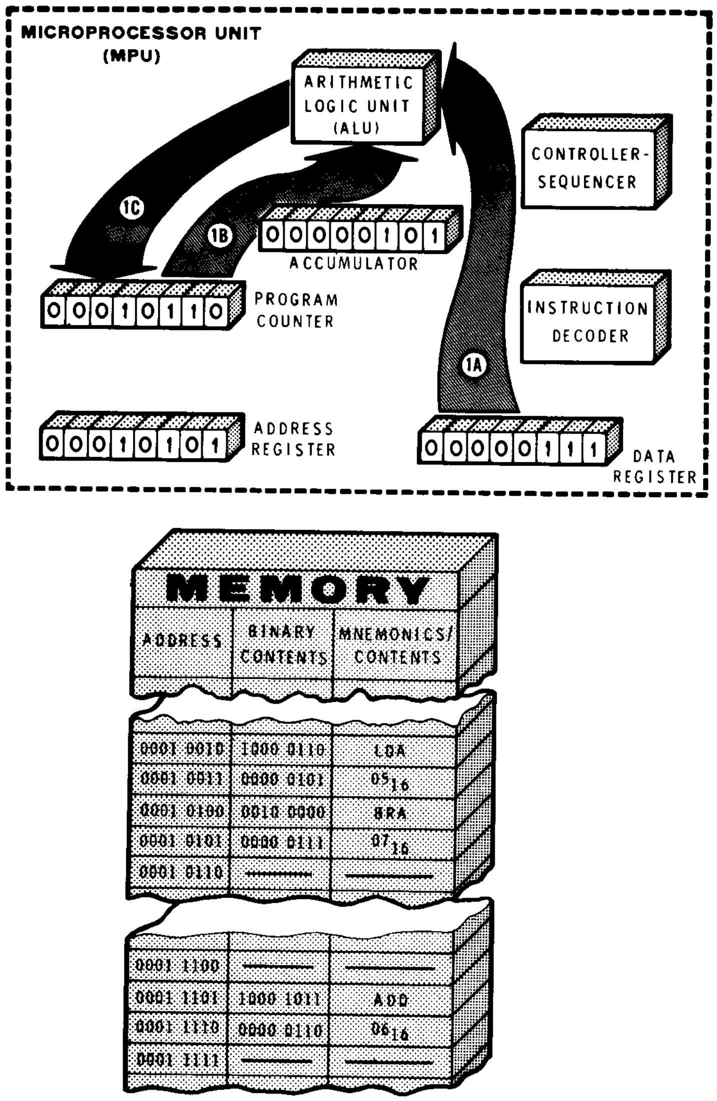


Figure 4-5
Computing the address of the next instruction.

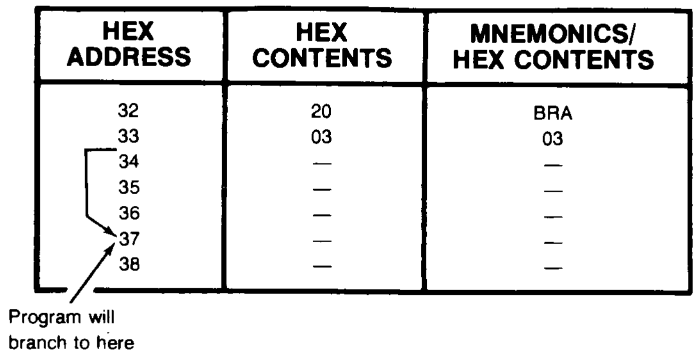
Branching Forward

Branching in the forward direction is a simple task if you know the value of the program count when the relative address is added. A couple of examples will illustrate the procedure.

In Figure 4-6A, the BRANCH 03 instruction is placed in locations 32_{16} and 33_{16}. Assuming this instruction is executed, from which location will the next instruction be fetched? Remember that the program counter will always point to the next byte in sequence. Since the last byte fetched was the relative address from location 33_{16}, the program counter must be at 34_{16} when the relative address is added. Adding the relative address produces a new program count of

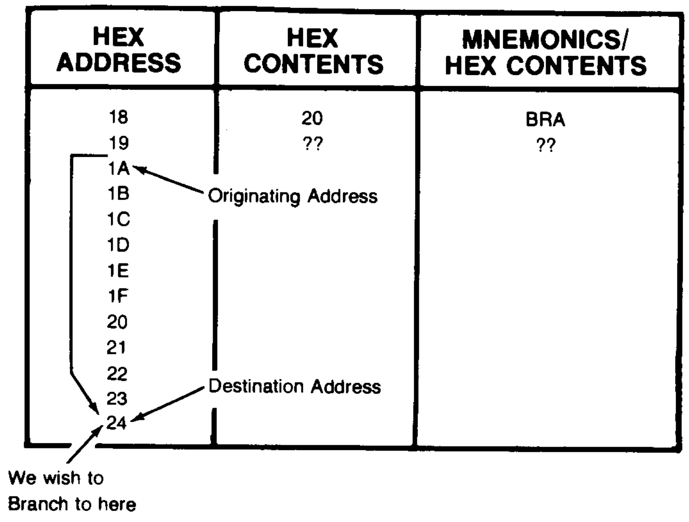
$$\begin{array}{r} 34_{16} \\ +\ 3_{16} \\ \hline 37_{16} \end{array}$$

Thus, the next instruction will be fetched from location 37_{16}.



HEX ADDRESS	HEX CONTENTS	MNEMONICS/ HEX CONTENTS
32	20	BRA
33	03	03
34	—	—
35	—	—
36	—	—
37	—	—
38	—	—

A



HEX ADDRESS	HEX CONTENTS	MNEMONICS/ HEX CONTENTS
18	20	BRA
19	??	??
1A		
1B		
1C		
1D		
1E		
1F		
20		
21		
22		
23		
24		

B

Figure 4-6
Branching forward.

Figure 4-6B shows a slightly different situation. Here we wish to branch to the instruction at address 24_{16}. The opcode for the branch instruction is at address 18_{16}. What relative address is required at location 19_{16} in order to implement this branch?

Keep in mind that the program count will automatically advance to $1A_{16}$ after the relative address is fetched from address 19_{16}. Also, remember that the relative address is added to the program count. Thus, a relative address of 00 would result in a "branch" to location $1A_{16}$. A relative address of 01 would result in a branch to location $1B_{16}$. Continuing this procedure until location 24_{16} is reached, you find that a relative address of 10_{10} is required. That is, the relative address must be $0A_{16}$ or 10_{10}.

There is a simple procedure for determining the relative address when branching forward. Subtract the originating address from the destination address. The difference is the relative address.

In our example, the originating address is $1A_{16}$. Remember this is the program count at the time the relative address is added. The destination address or the address to which you wish to branch is 24_{16}. Subtracting the originating address from the destination address, you find that the required relative address is

$$\begin{array}{rrl} 0010\ 0100_2 & 24_{16} & \text{Destination address} \\ \underline{-0001\ 1010_2} & \underline{1A_{16}} & \text{Originating address} \\ 0000\ 1010_2 & 0A_{16} & \text{Relative address} \end{array}$$

As you can see, a relative address of $0A_{16}$ is called for.

Branching Backward

A backward branch is used when a part of the program is to be repeated. The technique used for branching backward is similar to that used in branching forward. The difference is that a negative number is used as the relative address. As you learned earlier, two's complement representation is used to signify negative and positive numbers. Therefore, the relative address portion of any branch instruction is interpreted as a two's complement number.

This means that bit 7 of the relative address byte is a sign bit. A 0 at bit 7 tells the MPU to branch forward; a 1 tells it to branch backward. Thus, the positive values for the relative address extend from $0000\ 0000_2$ to $0111\ 1111_2$. This is 00_{16} to $7F_{16}$ and 00_{10} to $+127_{10}$.

The negative values extend from $1111\ 1111_2$ to $1000\ 0000_2$. This is FF_{16} to 80_{16} and -1 to -128_{10}. But remember, the relative address is with respect to the present program count. At the time the relative address is added, the program count points to the next byte after the relative address. Let's look at two examples of branching backward.

The first example is shown in Figure 4-7A. To what point does the MPU branch when the branch instruction at address $5D_{16}$ is executed? Notice that the relative address is $F9_{16}$. In binary this is $1111\ 1001_2$. Recall that this is the two's complement representation of -7. Thus, the program count should jump backwards 7 bytes — but from what point? Recall that after the byte at address 5E is fetched, the program count will automatically advance to $5F_{16}$ or $0101\ 1111_2$. When the relative address ($F9_{16}$ or $1111\ 1001_2$) is added, the result is

```
                   0101 1111    ←  Old program count
                +  1111 1001    ←  Relative Address
                1  0101 1000    ←  New program count
                ↑
Carry is ignored┘
```

The carry bit is ignored, leaving a new program count of 58_{16}. Thus, the next instruction will be fetched from address 58_{16}.

Figure 4-7B shows a different problem. Here we want the branch instruction at addresses B0 and B1 to direct the MPU back to address A0. What relative address is required? A simple procedure is:

1. Subtract the destination address from the originating address.

2. Take the two's complement of the difference.

Program branches to here

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ HEX CONTENTS
56	—	—
57	—	—
58	—	—
59	—	—
5A	—	—
5B	—	—
5C	—	—
5D	20	BRA
5E	F9	F9
5F	—	—

A

We wish to branch to here

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ HEX CONTENTS
A0	—	—
A1	—	—
A2	—	—
A3	—	—
A4	—	—
A5	—	—
A6	—	—
A7	—	—
A8	—	—
A9	—	—
AA	—	—
AB	—	—
AC	—	—
AD	—	—
AE	—	—
AF	—	—
B0	20	BRA
B1	??	??
B2	—	—

B

Figure 4-7
Branching backwards.

In our example, the program count will be advanced to $B2_{16}$ after the relative address is fetched. This is our originating address. The point to which we wish to branch is $A0_{16}$. This is our destination address. Subtracting yields a difference of

$$\begin{array}{rrl} 1011\ 0010_2 & B2_{16} & \text{Originating address} \\ -\ \underline{1010\ 0000_2} & \underline{A0_{16}} & \text{Destination address} \\ 0001\ 0010_2 & 12_{16} & \text{Difference} \end{array}$$

Next you compute the relative address by taking the two's complement of the difference. The two's complement of $0001\ 0010_2$ is $1110\ 1110_2$. In hexadecimal this is EE_{16}. Thus, the required relative address is EE_{16}.

Self-Test Review

1. What addressing mode is used by the branch instruction?

2. What does the second byte of a branch instruction indicate?

3. What happens in the MPU during the execution of the branch instruction?

4. What type of relative address causes a branch forward?

5. What type of relative address causes a branch backwards?

6. What is the maximum number of memory locations that can be branched over during a forward branch?

7. What is the maximum number of memory locations that can be branched over during a backward branch?

8. The opcode for the branch instruction is at address 20_{16}. The relative address is 06_{16}. After the branch instruction is executed, from what address will the next opcode be fetched?

9. The opcode for the branch instruction is at address 20_{16}. The relative address is $F1_{16}$. After the branch instruction is executed, from what address will the next opcode be fetched?

Answers

1. Relative addressing.

2. The second byte of the branch instruction is the relative address. This number is added to the contents of the program counter to form the absolute address.

3. The relative address is retrieved from memory and is added to the program count. The new program count goes into the program counter.

4. A positive two's complement number.

5. A negative two's complement number.

6. $0111\ 1111_2$ or $+127_{10}$.

7. $1000\ 0000_2$ or -128_{10}.

8. 28_{16}. Recall that during the execution of the branch instruction, the program counter will be incremented twice to 22_{16}. Thus, when the relative address (06_{16}) is added, the new address becomes 28_{16}.

9. As in answer 8, the program counter is automatically advanced to 22_{16} ($0010\ \ 0010_2$) before the relative address is added. $F1_{16}$ is equal to $1111\ \ 0001_2$. When this is added to the program count, the new address becomes

$$\begin{array}{rrl}
 & 0010\ \ 0010_2 & \text{Old program count} \\
 & \underline{1111\ \ 0001_2} & \text{Relative address} \\
 1 & 0001\ \ 0011_2 & \text{New program count} \\
 \text{Ignore carry} \rightarrow \uparrow & &
\end{array}$$

Thus, the next opcode will be fetched from address 13_{16}.

CONDITIONAL BRANCHING

The branch instruction allows the MPU to jump forward over a block of data or over a portion of a program. It also allows the MPU to jump backwards so a group of instructions can be repeated.

Until now we have been discussing the **unconditional** branch instruction. This type of instruction always results in a program branch. For this reason, it is called the **BR**anch **A**lways instruction. Its mnemonic is BRA.

There are other types of branch instructions that greatly expand the versatility of the MPU. These are called **conditional** branch instructions. Unlike BRA, these instructions cause a branch only if some specified condition is met.

A good example of a conditional branch instruction is the Branch If Minus (BMI). This instruction may or may not initiate a branch operation, depending on the result of some previous arithmetic or logic operation. This instruction might be placed after a subtract instruction. If the result of the subtraction is a negative number, the branch would be implemented. Otherwise, the MPU would continue to fetch and execute instructions in numerical order. An example may help to illustrate this.

Figure 4-8 shows part of a program that uses the branch if minus (BMI) instruction. Let's start with the instruction at address 95_{16}. This instruction causes the contents of location $B0_{16}$ to be loaded into the accumulator. Next, the SUB instruction subtracts the contents of location $B1_{16}$ from the number in the accumulator. The next instruction (BMI) examines the result of the subtraction. If the result was a minus number, the program will branch over the next three bytes. That is, the next instruction to be executed is the STA instruction at address $9E_{16}$. Thus, the resulting number in the accumulator is stored in location $B3_{16}$ and the MPU halts.

If the result of the subtraction is not minus, the BMI instruction has no effect. That is, the BMI instruction is fetched and executed but no branch occurs because the specified condition is not met. In this case, the next instruction to be executed is the STA instruction at address $9B_{16}$. Thus, the result of the subtraction will be stored in location $B2_{16}$.

HEX ADDRESS	HEX CONTENTS	MNEMONIC/HEX CONTENTS	COMMENTS
95	96	LDA	Load accumulator direct
96	B0	B0	with contents of this address.
97	90	SUB	Subtract
98	B1	B1	the contents of this address.
99	2B	BMI	If result is minus
9A	03	03	branch this far.
9B	97	STA	If result is not minus, store
9C	B2	B2	at this address;
9D	3E	HLT	then halt.
9E	97	STA	If result is minus, store
9F	B3	B3	it at this address;
A0	3E	HLT	then halt.

Figure 4-8
This program uses the BMI instruction
to make a simple decision.

Notice that the program flow can take one of two paths, depending on the result of the subtraction. The BMI instruction gives the MPU this capability. The conditional branch instructions are sometimes called "decision making instructions." The reason for this becomes obvious if you consider the implications of our sample program. Here the MPU decides if the number at address $B1_{16}$ is larger than that at $B0_{16}$. The program path is determined by the outcome of this decision. If the number in $B1_{16}$ is larger, the result of the subtraction is a negative number. In this case, the result is stored in location $B3_{16}$. Otherwise, the resulting difference is stored in location $B2_{16}$.

Virtually all programs must make some type of decision. Some frequently encountered decisions are:

"Which of two numbers is larger?"

"Does this byte represent a letter of the alphabet or a numeral?"

"Are these two numbers equal?"

"Is this an even number?"

"Has the program loop been repeated the proper number of times?"

Conditional branch instructions are used in making all of these decisions.

Condition Codes

As the name implies, a conditional branch instruction causes a program branch only if some specified condition is met. Some commonly monitored conditions are:

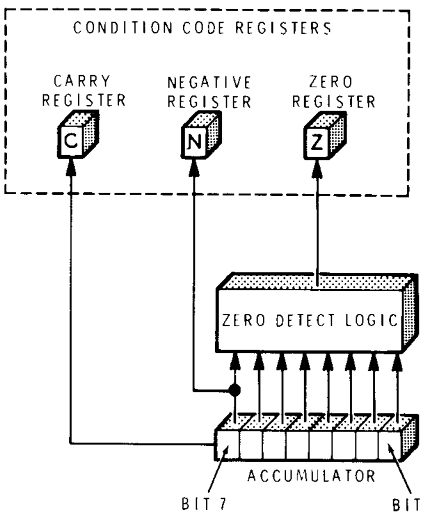


Figure 4-9
Condition code registers monitor the operations in the accumulator.

1. Did a previous operation result in a negative number in the accumulator?
2. Did a previous operation result in zero in the accumulator?
3. Did a previous operation result in a carry from bit 7 of the accumulator?

To keep track of these conditions, most microprocessors have a group of single bit registers called condition code registers. Three of these registers are shown in Figure 4-9. They are the Negative (N) Register, the Zero (Z) Register, and the Carry (C) Register.

Negative (N) Register Recall that negative numbers are expressed in two's complement form. Using this system, the most significant bit determines whether or not the number is negative. In an 8-bit byte, bit 7 is a 1 if the two's complement number is negative. Thus, the N register monitors bit 7 of the accumulator. Immediately after an operation that involves the accumulator, the N register looks at bit 7 to see if the number is negative. If so, the N register is set to 1. If the number in the accumulator is not negative, the N register is reset to 0.

Most operations that involve the accumulator affect the N register in this way — **but not all.** In a later unit we will point out how this register is affected by each instruction. In this unit, we will assume that the N register is affected as outlined above any time a number is added to, subtracted from, loaded into, or stored from the accumulator.

Another name for a condition code is a flag. Thus, the N register is sometimes called the N flag or the negative flag.

Zero (Z) Register This register monitors the accumulator looking for all zeros. Immediately after an operation that involves the accumulator, the zero-detect circuit looks at the resulting number. If all 8 bits are 0, the Z register is set to 1. Otherwise, the Z register is reset to 0. Most operations that involve the accumulator affect the Z register in this way.

Carry (C) Register The C register acts somewhat like an extension of the accumulator. You have seen that when two unsigned 8-bit numbers are added, the sum is frequently a 9-bit number. For example:

```
             1001  0010     8-bit addend
          +  1100  0110     8-bit augend
           1 0101  1000     9-bit sum
           ↑
   carry ──┘
```

Since the accumulator is an 8-bit register, the sum will not fit. The most significant bit (the carry) would be lost if you did not have another 1-bit register to hold it. This is the purpose of the C register. Any operation that causes a carry out of bit 7 will set the carry register to 1. Arithmetic operations that do not result in a carry will reset this register to 0.

The carry register is also used to keep track of "borrows" during subtract operations. If a subtraction requires a borrow for bit 7, the carry flag will also be set. For example, suppose you subtract an unsigned, binary number from a smaller unsigned binary number. The result will, of course, be a negative number. Moreover, bit 7 will have to "borrow" a bit to complete the subtraction. As a simple example, let's subtract 2 from 1. The subtraction looks like this

```
   Borrow →  1
               0000  0001     Minuend
             − 0000  0010     Subtrahend
               1111  1111     Difference
```

The carry bit is set to 1 to indicate that a borrow operation occurred. Many subtraction operations do not require borrows. In these cases, the carry bit is reset to 0 to indicate that no borrow occurred.

Notice that the carry code can have different meanings, depending on the operation involved. That is, a 1 can mean either that a carry occurred or that a borrow occurred. The precise meaning of the 1 depends on whether the operation was an addition or a subtraction. We will discuss some additional aspects of the carry register in a later unit.

Overflow (V) Register The final condition code that is to be considered in this unit keeps track of two's complement overflow. Figure 4-10 shows how this register is connected in the MPU. A special circuit detects an overflow condition by monitoring bit 7 of the ALU's input and output lines. This circuit sets the V flag when an overflow occurs but clears it if no overflow occurs.

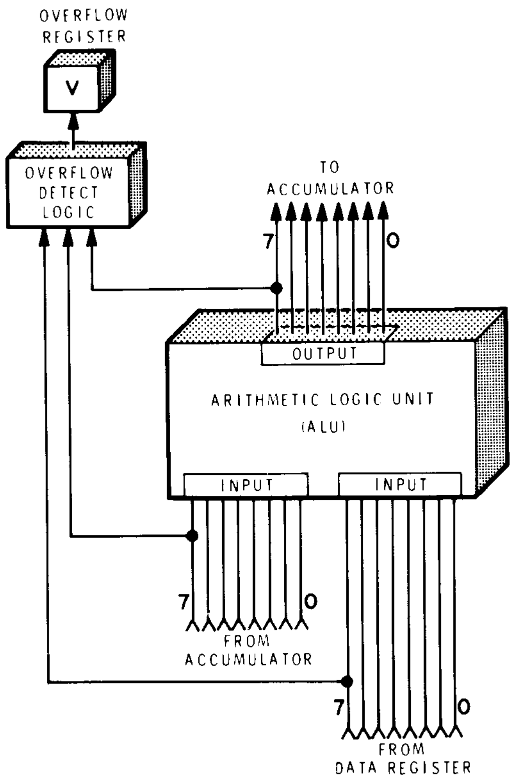


Figure 4-10
The overflow register monitors bit 7 of the ALU's input and output lines.

Let's see what is meant by two's complement overflow. Recall that the ALU adds numbers as if they were unsigned binary numbers. Even so, it can handle signed binary numbers if the proper bit patterns represent the negative numbers. This is the reason that the two's complement method of representing signed numbers has become so popular. A disadvantage of this system is that the magnitude of the number must be represented by 7 bits, since the eighth bit is used as the sign. Remember that a 1 in the MSB defines the number as negative.

Unfortunately, if two signed numbers are added and their sum exceeds 7-bits, the sign bit will be changed. For example, assume that a program adds $+73_{10}$ and $+96_{10}$. The addition looks like this:

$$\begin{array}{rr} \underline{0}100\ \ 1001_2 & +73_{10} \\ \underline{0}110\ \ 0000_2 & +96_{10} \\ \hline \underline{1}010\ \ 1001_2 & 169_{10} \end{array}$$

The answer is correct if all the binary numbers represent unsigned quantities. However, using two's complement, the underlined bits represent sign bits. Therefore, the answer does **not** represent 169_{10}. Instead, it represents -87_{10}. The reason for this error is that there was an overflow from bit 6 into the sign bit (bit 7). This is one of the situations that the V flag indicates.

When two's complement numbers having the same sign are added, the sum should have the same sign. That is, when two positive numbers are added, the sum should be positive. By the same token, when two negative numbers are added, the sum should be negative. However, an overflow can cause the sign to be reversed. The overflow logic detects this situation and sets the V flag whenever an overflow occurs.

The sign bit can also be upset during subtract operations. For example, when a negative number is subtracted from a positive number, the results should be positive. Remember that subtracting a negative number is tantamount to adding a positive number. However, in certain cases, an overflow can reverse the sign bit. This type of overflow occurs when the signs of the minuend and subtrahend are opposite and the difference has the sign of the subtrahend. This condition also sets the V flag.

Conditional Branch Instructions

The conditional branch instructions available in our hypothetical microprocessor are shown in Figure 4-11. While these are largely self-explanatory, a couple of points should be mentioned.

INSTRUCTION	MNEMONIC	OPCODE	BRANCH IF
Branch If Carry Clear	BCC	24	C=0
Branch If Carry Set	BCS	25	C=1
Branch If Not Equal Zero	BNE	26	Z=0
Branch If Equal Zero	BEQ	27	Z=1
Branch If Plus	BPL	2A	N=0
Branch If Minus	BMI	2B	N=1
Branch If Overflow Clear	BVC	28	V=0
Branch If Overflow Set	BVS	29	V=1

Figure 4-11
Conditional Branch Instructions.

The first instruction, Branch If Carry Clear (BCC), monitors the C register. If the carry register is reset to 0, the branch is implemented. Notice that the words "clear" and "reset" are used interchangeably in this regard. They both mean the register contains a 0.

The branch instructions that monitor the Z register can also be confusing. The Branch If Equal Zero (BEQ) instruction implements a branch when the Z register is set to 1. Recall that the Z register is set to 1 when the number in the accumulator is zero. Thus, you must remember that a 0 in the Z register means that the number in the accumulator is **not** zero.

These conditional branch instructions can be used with other instructions to make a wide range of decisions. They greatly increase the power of the microprocessor. More than any other type of instruction, the conditional branches are responsible for the MPU's "intelligence." In the next section, you will see how these instructions are used.

Self-Test Review

10. What is the difference between an unconditional branch instruction and a conditional branch instruction?

11. What condition is tested by the branch if minus (BMI) instruction?

12. When is the N flag set?

13. When is the Z flag set?

14. During an add operation, the C flag is set. What does this represent?

15. During a subtract operation, the C flag is set. What does this indicate?

16. Often, when two positive 2's complement numbers are added, the sign bit of the answer will indicate a negative sum. This "error" can be spotted by checking which flag?

17. Under what condition will the BEQ instruction cause a branch to occur?

18. Under what condition will the BPL instruction cause a branch to occur?

19. When subtracting unsigned binary numbers, which flag indicates that the difference is a negative number?

Answers

10. An unconditional branch instruction always causes a branch operation to occur. On the other hand, the conditional branch instruction implements a branch operation only if some specified condition is met.

11. The BMI instruction tests the Negative (N) register to see if it is set.

12. Generally speaking, the N flag is set if the previous instruction left a 1 in the MSB of the accumulator.

13. Generally, the Z flag is set if the previous instruction left all zeros in the accumulator.

14. During an add operation, the carry bit is set if there is a carry from bit 7 of the accumulator.

15. During a subtract operation, the carry bit is set if bit 7 had to "borrow" a bit to complete the subtraction.

16. This condition results from a two's complement overflow. Thus, the V flag will be set if this condition occurs.

17. The BEQ instruction causes a branch to occur only if the Z register is set.

18. The BPL instruction causes a branch to occur only if the N register is clear.

19. The carry flag.

MICROCOMPUTER PROGRAMMING

As the programs you write increase in complexity and length, the need for some type of organization for problem solving becomes quite evident.

The exact method of problem solving will vary from person to person. Some may have an intuitive "feel" for program development and may write programs "off the top of their head". If you can do this, you are fortunate indeed. Most people, however, need a formal "plan of attack" when solving problems and writing programs.

Essentially, there are three steps to writing a program. The first is to **define the problem.** This may seem like a relatively easy task but it sometimes proves to be the most difficult. Unfortunately, there is no set procedure for defining problems. Some problems will lend themselves to mathematical definitions while others may require a logical or graphical approach. We can only advise you to analyze each situation carefully before you attempt to define the problem.

The last step in creating a program is the **writing of that program** using the instruction set for your microprocessor. You already know a little bit about writing programs and you will be learning much more as you progress through the course. What we are primarily concerned with here is the second step in creating a program: mapping or **flowcharting** the solution.

The Flowchart

Given a well structured blueprint, a carpenter can construct even the most complex project with little or no difficulty. The same is true of a programmer with a good flowchart. As a matter of fact, some flowcharts are so detailed that little remains for the programmer to do other than fill in the appropriate instructions. For our purposes, we'll use a simple flowcharting technique using the five most common symbols. These are shown in Figure 4-12.

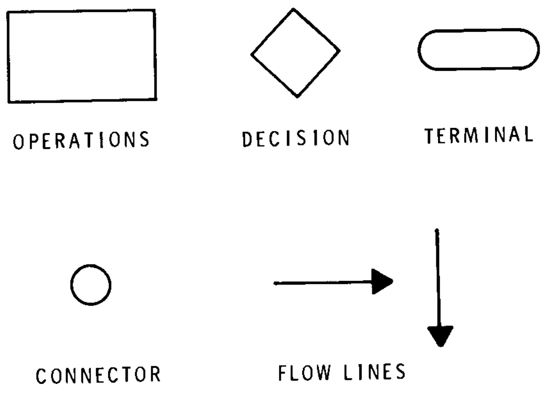


Figure 4-12
Flowchart Symbols.

THE SYMBOLS

All flowcharts have a beginning, and most flowcharts will have an end. These are referred to as the "terminal" points of the flowchart and they are represented by the **terminal** symbol. The terminal symbol seen in Figure 4-13 is simply labeled "start" or "stop", to denote its meaning.

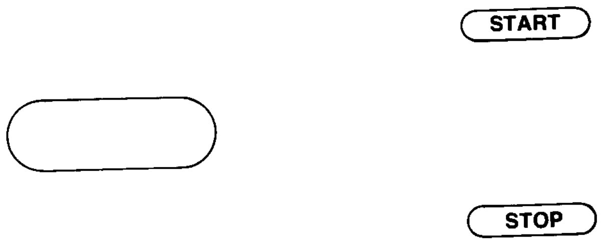


Figure 4-13
Terminal Symbol.

The **operations box** is probably the most frequently used flowchart symbol. It may represent a transfer process such as moving or loading data or an algebraic process like addition or multiplication. It can also be used for such statements as "print" or "set"; which we will get into later. If you have a process that is not specified by one of the other symbols, use the operations box. Later we will introduce some additional flowchart symbols. Some examples of the operations box use are shown in Figure 4-14.

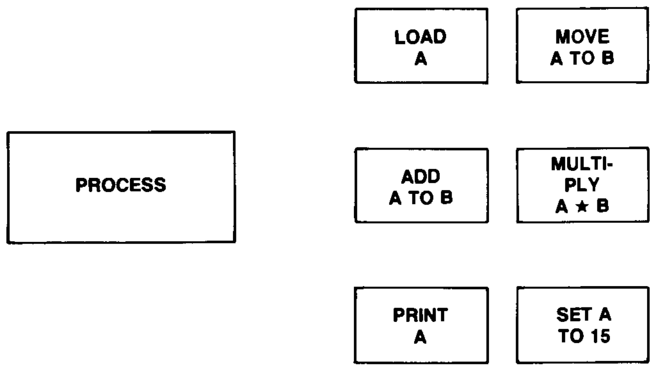


Figure 4-14
Operation Box.

The diamond-shaped **decision box** is really the "heart" of the flowchart. It indicates a logical choice between two conditions and, therefore, it controls the direction of program flow. If a condition is satisfied, the "yes" route is taken. If the condition is not satisfied, then the "no" route is selected. Typical examples for using the decision box are shown in Figure 4-15. The yes and no routes can originate in any corner of the diamond.

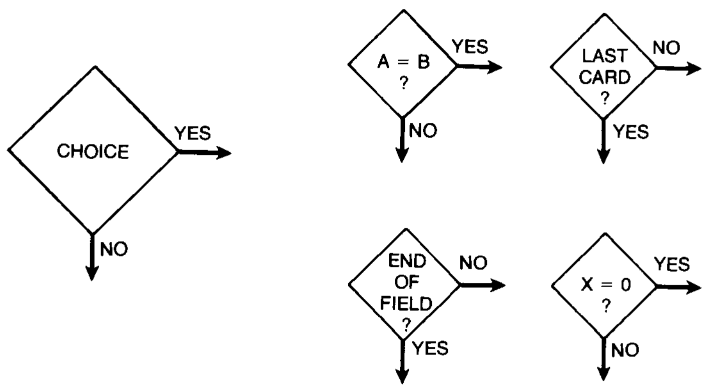


Figure 4-15
Decision Box.

Flow lines and **connectors** are used to tie the symbols and sections of the flowchart together. Normally, the chart will be arranged to flow from top to bottom and from left to right. However, there is no set rule in regard to this and the flowchart could just as well flow in the opposite direction. The use of flow lines as well as connectors is demonstrated in Figure 4-16. The connector symbols direct you from one section of the program to another. The connectors are labeled with values to indicate where the connection takes place; 1A connects with 1A, 2A connects with 2A, etc.

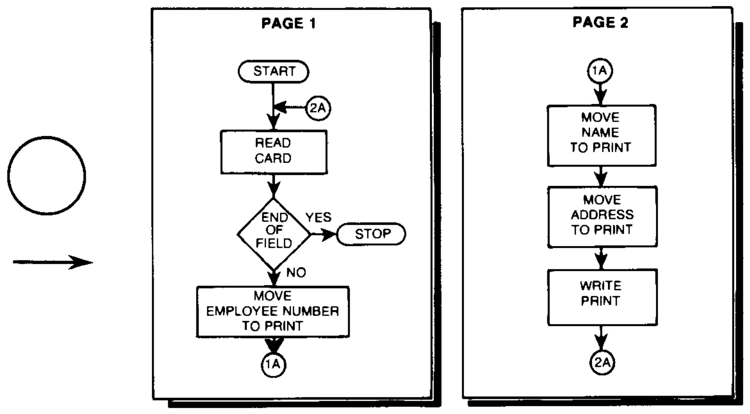


Figure 4-16
Connectors and Flowlines.

MATHEMATICAL SYMBOLS

Usually, sophisticated flowcharts use mathematical symbols to represent the decision making process. These symbols allow us to illustrate our decision quickly and concisely. In Figure 4-17A you see the five most frequently used symbols and their meanings. Figure 4-17B shows some examples of their use.

A

	SYMBOL	EXAMPLE	DESCRIPTION
1.	$=$	$A = B$	A is equal to B.
2.	$>$	$A > B$	A is greater than B.
3.	$<$	$A < B$	A is less than B.
4.	$\geqq$	$A \geqq B$	A is greater than or equal to B.
5.	$\leqq$	$A \leqq B$	A is less than or equal to B.

B

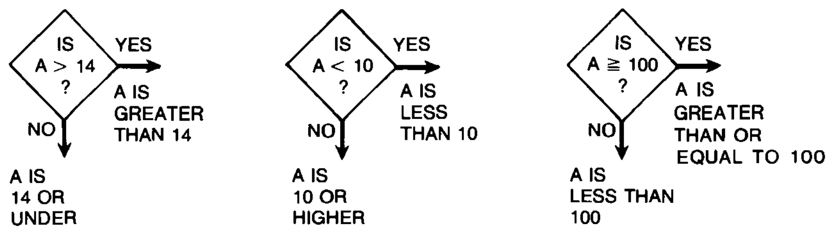


Figure 4-17
Mathematical Decision Making Symbols and Their Use.

CONSTRUCTING THE FLOWCHART

Now that you can identify flowchart symbols, it's time to solve a few problems using flowcharts. The first, to add three numbers and produce a sum, is shown in Figure 4-18. The problem and the mathematical expression of the problem are shown on the left. The mathematical expression serves as a definition of the problem. Our flowchart is on the right.

PROBLEM:

1. Add 3 numbers:
   A, B, and C producing sum, S.
2. Print sum.

$$S=A+B+C$$

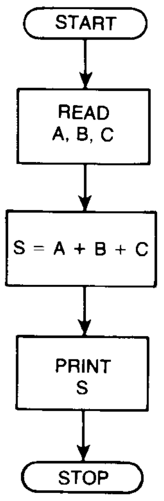


Figure 4-18
Flowchart Construction.

It begins with the start terminal symbol. In a flowchart of this size you might think that this symbol is unnecessary. But, no matter how simple you think a flowchart is, always use the start terminal symbol. You will see later on in this course that flowcharts can get quite lengthy, and it is often necessary to return to the beginning of your flowchart. The start terminal symbol allows you to do this with ease. If you use it all the time, even when it seems unnecessary, then you'll never get into difficulty locating the beginning of the program.

Next we find an operation box that tells the computer to read, or identify, the variables A, B, and C. This process would actually move the numbers from memory into the microcomputer. Again, you might think that this box is unnecessary or that it could be combined with the next operations box. But, remember, the more detailed the flowchart, the easier it is to write your program from it.

In the second operations box, the computer is instructed to add A, B, and C, giving you the total S. The final operations box tells the computer to print the solution, S. Finally, the stop terminal symbol is used to end the process.

Now that you have seen how it is done, it's time to look at a more complex problem. In Figure 4-19 you see a drawing of a traffic intersection. Main Street is a busy thoroughfare while Side Street handles only a moderate amount of traffic. A traffic study shows that if traffic on Main Street is allowed to move for two minutes, while traffic on Side Street is allowed to move for one minute, the intersection controls traffic very efficiently. Light A has been assigned for control of traffic on Main Street and light B for control of traffic on Side Street.

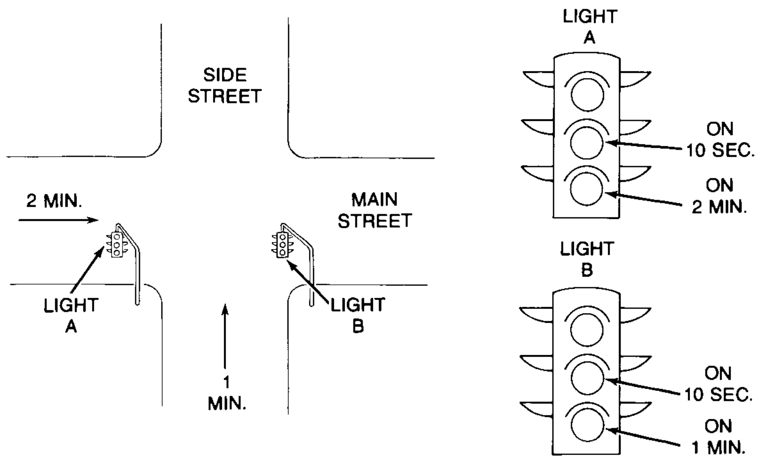


Figure 4-19
The Traffic Problem.

It is not necessary at this point, but you might like to try to draw this flowchart yourself. If not, just read along while we work our way through this problem. Before you can attempt to draw a flowchart, you must carefully examine the problem. That is, you must carefully determine exactly what each light must do.

Light A must remain green for two minutes. Naturally, you want a caution light between red and green. We suggest a yellow light duration of 10 seconds. The duration of the red light will be controlled by light B. Light B, on the other hand, will be green for one minute and yellow for ten seconds. When light B turns red, light A turns green and the cycle repeats.

With this information, you should encounter little difficulty drawing the flowchart for this problem. Defining the problem in detail will always simplify problem solving. If you care to try your hand at this flowchart, grab a pencil and some paper and give it a try. If not, just continue reading. You'll find our solution in Figure 4-20.

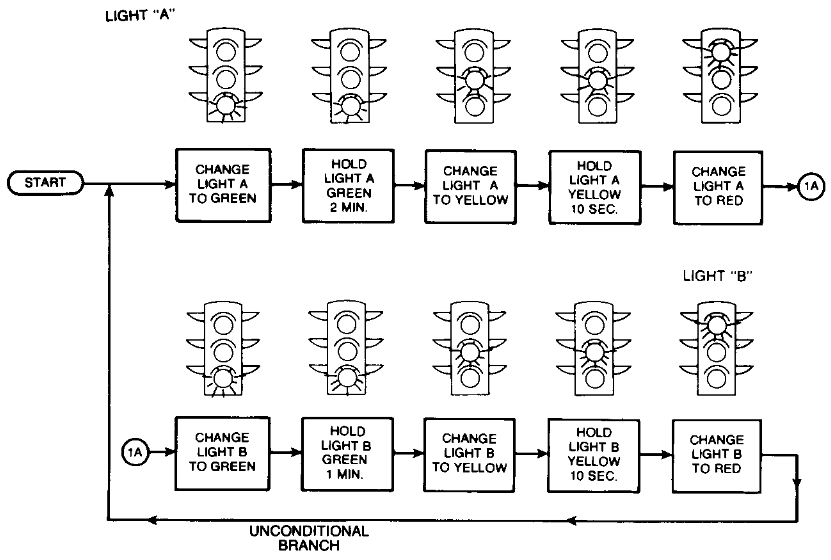


Figure 4-20
The Traffic Solution.

The flowchart begins in the top left-hand corner of Figure 4-20. The first five operation blocks control the operation of light A. You begin the program at the box that changes light A to green. The second process holds the light green for two minutes. After this, light A must change to yellow and hold there for ten seconds as the next two blocks indicate. Finally, light A is changed red by the fifth operation.

Connector 1A tells you that the flow of the program picks up at the corresponding 1A connector in the bottom left-hand corner of the illustration. Here, you enter the light B routine that controls the traffic on Side Street. Now that the traffic on Main Street has a red light, it is safe to change light B to green. Light B is held green for one minute, then it is changed to yellow and held for ten seconds. Finally, light B changes to red. Of course, you want this process to repeat so you draw a flow line from the end of the light B cycle back to the beginning of the light A cycle.

This type of a flow line is called an unconditional branch because it must always point to a particular point in the program. The unconditional branch in a flowchart would correspond to an unconditional branch in the actual program.

There are other flow lines that originate at a decision box rather than at an operation box. These are called conditional branches because the flow of the program along these lines is based on the conditions stated in the box. Conditional branches in a flowchart are related to conditional branches in an actual program.

Figure 4-21 demonstrates the use of the decision box and the conditional branch. The problem is stated on the left and the flowchart solution is shown on the right.

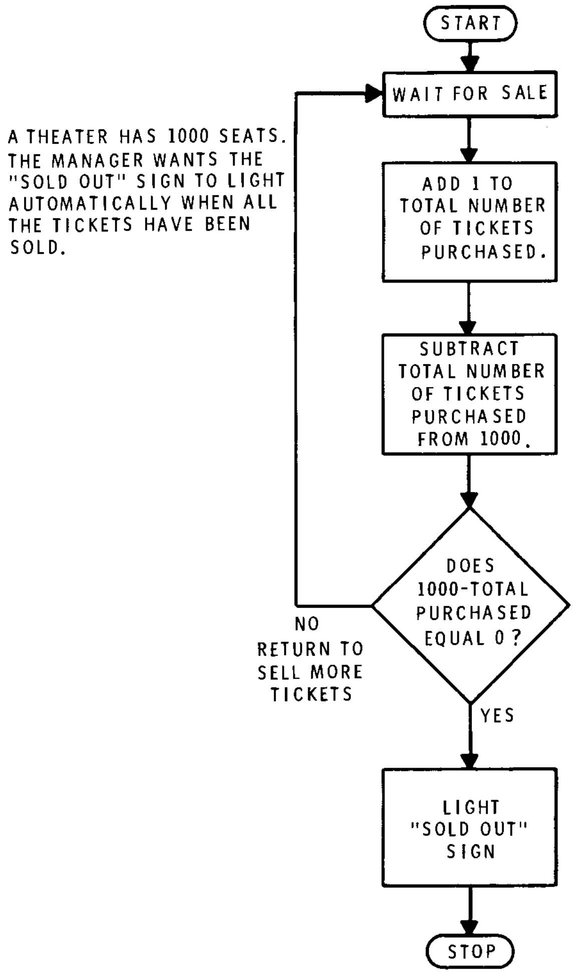

START
WAIT FOR SALE
A THEATER HAS 1000 SEATS. THE MANAGER WANTS THE "SOLD OUT" SIGN TO LIGHT AUTOMATICALLY WHEN ALL THE TICKETS HAVE BEEN SOLD.
ADD 1 TO TOTAL NUMBER OF TICKETS PURCHASED.
SUBTRACT TOTAL NUMBER OF TICKETS PURCHASED FROM 1000.
DOES 1000-TOTAL PURCHASED EQUAL 0 ?
NO
RETURN TO SELL MORE TICKETS
YES
LIGHT "SOLD OUT" SIGN
STOP

Figure 4-21
The Use of the Decision Box.

Again, the chart begins with the terminal start symbol. As each ticket is purchased, it is added to the total number of tickets purchased. Then the total number of tickets sold is subtracted from the number of seats available.

Now the decision must be made. If all tickets have not been sold, you must branch to the beginning of the flowchart and resume the count. If all tickets are sold, that is if the result of the subtraction is zero, then branch forward and light the "SOLD OUT" sign. Once the sign is lit, you complete the flowchart with the stop terminal symbol. Although the problem is very simple, there is no way to solve it without using a decision box and conditional branches in the flowchart.

Self Review Questions

20. The five most common flowcharting symbols are the:

    A. ______________________
    B. ______________________
    C. ______________________
    D. ______________________
    E. ______________________

21. You are permitted to use mathematical symbols in a flowchart.

    ____
    (True/False)

22. The flowchart should always begin with a start terminal symbol.

    ____
    (True/False)

23. The ______________________ ______________ in a flowchart corresponds to the unconditional branch in the actual program.

24. The ______________________ ______________ in a flowchart are related to the conditional branch in an actual program.

Self Review Answers

20. The five most common flowcharting symbols are the:

    A. **Terminal Symbol**
    B. **Operations Box**
    C. **Decision Box**
    D. **Flow Lines**
    E. **Connectors**

21. **True.** Flowcharts make use of mathematical symbols.

22. **True.** The flowchart should always begin with a start terminal symbol.

23. The **unconditional branch** in a flowchart corresponds to the unconditional branch in the actual program.

24. The **conditional branch** in a flowchart is related to the conditional branch in an actual program.

ALGORITHMS

An algorithm is a step-by-step procedure for doing a particular job. It generally involves doing a complex task by stringing together a series of simple steps. To illustrate the use of an algorithm, consider the following very simple example.

Multiplying by Repeated Addition

Most microprocessors do not have hardware multiply capabilities. That is, they do not have a multiplication circuit nor a multiply instruction. Nevertheless, the microprocessor can be made to multiply by use of an algorithm. One procedure for doing this was discussed earlier. It involved adding the multiplicand to itself the number of times indicated by the multiplier. In the previous example, this was done by using a separate ADD instruction for each addition. This procedure is unsatisfactory for two reasons. First, it results in excessively long programs. Second, you must know the value of the multiplier so that you know how many ADD instructions to include.

A better approach, although still far from ideal, is to use a program loop that will multiply two numbers by repeated addition. For the time being, assume that the two numbers are both positive and that the product does not exceed 255_{10}. Let's further assume that we use only the instructions which have been discussed up to this point. In fact, we will restrict ourselves to the instructions shown in Figure 4-22.

		ADDRESSING MODE			
INSTRUCTION	MNEMONIC	IMMEDIATE	DIRECT	RELATIVE	INHERENT
Load Accumulator	LDA	86	96		
Clear Accumulator	CLRA				4F
Decrement Accumulator	DECA				4A
Increment Accumulator	INCA				4C
Store Accumulator	STA		97		
Add	ADD	8B	9B		
Subtract	SUB	80	90		
Branch Always	BRA			20	
Branch If Carry Set	BCS			25	
Branch if Equal Zero	BEQ			27	
Branch if Minus	BMI			2B	
Halt	HLT				3E

Figure 4-22
Instructions to be used.

The algorithm for multiplying by repeated addition is quite simple. To multiply A times B, you merely add A to a specific location B times. For example, to multiply 5 times 3, you clear a location and then add 5. You continue the addition until 5 has been added 3 times. The number in the affected location will then be 15_{10} which is the product of 5 times 3.

The success of this operation depends on the microprocessor knowing when to stop. It must add 5 three times, but only three times. One way to keep track of the number of additions is to decrement the multiplier (3) each time an addition is made. When the multiplier reaches 0, the proper number of multiplications has been carried out.

Figure 4-23 is a flow chart that illustrates the algorithm. In the first two steps, the MPU clears the accumulator and stores the resulting number (0) in the product. This ensures that the product is zero before the first number is added. Next, it loads the multiplier and checks to see if the multiplier is 0. If so, the process is stopped since a multiplier of 0 dictates a product of 0.

In our example, the multiplier is 3; therefore, we exit the decision block via the "no" line. The next step tells us to decrement the multiplier. The new value of the multiplier (2) is stored for future use. Next, the product whose present value is 0 is loaded. Then, the multiplicand (5) is added so that the new value of the product becomes 5. This completes our first pass through the program. Remember that the multiplicand has been added once and that the multiplier has been reduced by one.

Notice that the program loops back to the input of the second block. The product which now has a value of 5 is stored back in memory. The multiplier (which is now 2) is loaded and tested. Because its value is not yet 0, the multiplier is decremented to 1 and stored again. The product (whose value is now 5) is then loaded and the multiplicand is added so that a new value of 10_{10} is obtained.

The program loops again and the new product (10_{10}) is stored. The multiplier (whose value is now 1) is loaded and tested. Because its value is still not 0, it is decremented again. Notice that the value of the multiplier is now 0. This value is stored away, the product (10_{10}) is fetched, and the multiplicand is added once more. The new value of the product becomes 15_{10}.

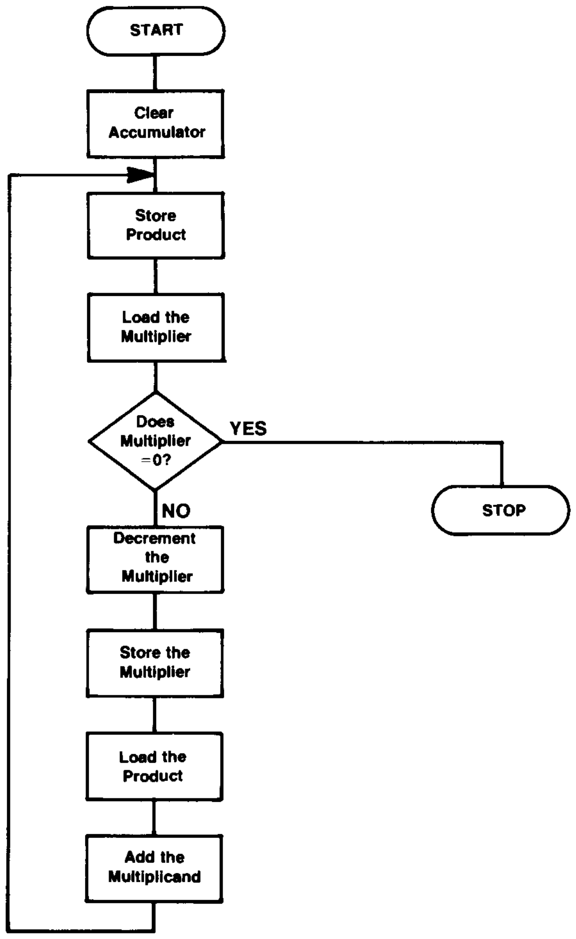


Figure 4-23
Flow chart for multiplying by repeated addition.

The program loops again and the product is stored. The multiplier is loaded and tested. Recall that the value of the multiplier is now 0. Consequently, we exit the decision block via the "yes" line. The program has accomplished its task and it now stops. Notice that the value of the product is 15_{10} which is the proper answer for 5×3.

The next task is to convert the flow chart to a program that the computer can execute. Figure 4-24 shows such a program. Carefully compare this program to the flow chart paying particular attention to the comments column. Work through the program on paper and verify that it will multiply the numbers at addresses 11_{16} and 12_{16}. Although 3 and 5 are used in this example, the program will work for any values of multiplier and multiplicand as long as the product does not exceed 255_{10}.

HEX ADDRESS	HEX CONTENTS	MNEMONIC/HEX CONTENTS	COMMENTS
00	4F	CLRA	Clear the accumulator.
01	97	STA	Store the product
02	13	13	in location 13_{16}.
03	96	LDA	Load the accumulator with the
04	12	12	multiplier from location 12_{16}
05	27	BEQ	If the multiplier is equal to zero,
06	09	09	branch down to the Halt instruction.
07	4A	DECA	Otherwise, decrement the multiplier.
08	97	STA	Store the new value of the
09	12	12	multiplier back in location 12_{16}.
0A	96	LDA	Load the accumulator with the
0B	13	13	product from location 13_{16}.
0C	9B	ADD	Add
0D	11	11	the multiplicand to the product.
0E	20	BRA	Branch back to instruction
0F	F1	F1	in location 01.
10	3E	HLT	Halt.
11	05	05	Multiplicand.
12	03	03	Multiplier.
13	–	–	Product.

Figure 4-24
This program multiplies the numbers at addresses 11_{16} and 12_{16}, and places their product at address 13_{16}.

Dividing by Repeated Subtraction

Another interesting algorithm is one that allows the microprocessor to divide by repeated subtraction. The technique is to keep track of the number of times that the divisor can be subtracted from the dividend. For example, suppose you wish to divide 47_{10} by 15_{10}. The divisor can be subtracted 3 times:

First subtraction Second Subtraction Third Subtraction

$$\begin{tikzcd} \begin{array}{r}47_{10}\\ -15_{10}\\ \hline 32_{10}\end{array} \arrow[r, bend left] & \begin{array}{r}32_{10}\\ -15_{10}\\ \hline 17_{10}\end{array} \arrow[r, bend left] & \begin{array}{r}17_{10}\\ -15_{10}\\ \hline 2_{10}\end{array}\end{tikzcd}$$

Because three subtractions occurred, the quotient is 3. Also, because 2 was left after the last subtraction, the remainder is 2. We can verify this by long division:

$$\begin{array}{lcrll} & & 3_{10} & \leftarrow & \text{Quotient} \\ \text{divisor} & \rightarrow & 15_{10}\,\overline{)\,47_{10}} & \leftarrow & \text{Dividend} \\ & & \underline{45_{10}} & & \\ & & 2_{10} & \leftarrow & \text{Remainder} \end{array}$$

The microprocessor keeps track of the number of subtractions by incrementing the quotient by one each time a subtraction occurs. Of course, the quotient must be initially set to zero.

The divisor is subtracted from the dividend until any further subtraction would result in a negative number. The MPU can use the BMI instruction to check for a negative result on each loop. The negative result is the indication that the process is finished.

A flow chart for this algorithm is shown in Figure 4-25. The actual program is shown in Figure 4-26. the program is arbitrarily placed in locations 00 through 10_{16}. The dividend (47_{10}) is at address 11_{16} while the divisor (15_{10}) is at address 12_{16}. When executed, the program will produce the quotient at location 13_{16} and the remainder at location 11_{16}.

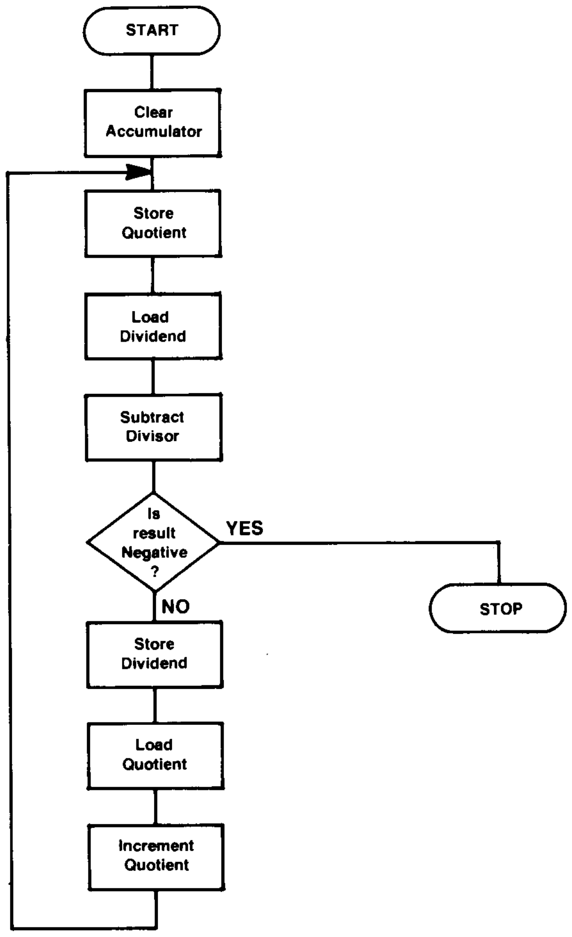


Figure 4-25
Flow chart for dividing by repeated subtraction.

HEX ADDRESS	HEX CONTENTS	MNEMONIC/HEX CONTENTS	COMMENTS
00	4F	CLRA	Clear the accumulator.
01	97	STA	Store in the quotient which
02	13	13	is at address location 13_{16}.
03	96	LDA	Load the accumulator with the
04	11	11	dividend from location 11_{16}.
05	90	SUB	Subtract the
06	12	12	divisor from the dividend.
07	2B	BMI	If the difference is negative, brach
08	07	07	down to the Halt instruction.
09	97	STA	Otherwise, store the difference
0A	11	11	back in location 11_{16}.
0B	96	LDA	Load the accumulator with the
0C	13	13	quotient.
0D	4C	INCA	Increment the quotient by one.
0E	20	BRA	Brach back to instruction
0F	F1	F1	in location 01.
10	3E	HLT	Halt.
11	2F	2F	Dividend (47_{10}).
12	0F	0F	Divisor (15_{10}).
13	—	—	Quotient.

Figure 4-26
This program divides by repeatedly subtracting the divisor from the dividend.

Refer to the flow chart and the comments column of the program. Before reading further, try running through the program on paper. This will give you a feel for how the computer solves the problem.

Now let's go through the program to see what it does. The first two instructions clear the quotient. Next, the dividend (47_{10}) is loaded into the accumulator and the divisor (15_{10}) is subtracted. The BMI instruction is used to examine the difference (32_{10}). Since the difference is not minus, the branch does not occur. Consequently, the next instruction stores the difference (32_{10}) back in the location from which the dividend came. In effect, the difference becomes the new dividend. Next the quotient (0) is loaded and is incremented to 1. The program then branches back to the instruction in location 01. This instruction stores the quotient (1) back in location 13_{16}.

On the next pass through the program, the new dividend (32_{10}) is loaded and the divisor (15_{10}) is subtracted again. This produces a difference of (17_{10}). Since the difference is not negative, the BMI instruction does not cause a branch. Thus, the difference is stored back in location 11_{16}. The quotient is loaded into the accumulator and is incremented to 2. The BRA instruction causes the program to loop once again. The STA instruction in location 01 stores the quotient (2) back in location 13_{16}.

On the third pass the dividend (17_{10}) is loaded and the divisor (15_{10}) is subtracted a third time. The difference (02) is still not negative so no branch occurs. The difference is stored away; the quotient is loaded and is incremented to 03. Notice that this is the proper final value for the quotient. Therefore, on the next pass, the MPU should be able to break out of the loop.

The quotient is stored back in location 13_{16}. The dividend, which now has a value of 2, is loaded. The divisor (15_{10}) is subtracted, leaving a negative number (-13_{10}) in the accumulator. The BMI instruction recognizes that this is a negative number and implements a branch operation. Notice that the MPU branches forward to the HLT instruction. Thus, the program ends with the quotient set to 3. The remainder is at address 11_{16}. That is, the remainder is what remains of the dividend after the third subtraction.

It may bother you that there were four subtractions and that a negative difference resulted from the last subtraction. However, you will recall that the quotient was incremented only on the first three of these subtractions. Thus, the final quotient is 3. Moreover, the negative difference that resulted during the last subtraction was never stored. Consequently, the remainder was 2 when the program ended.

This program does have some drawbacks. For one thing, neither the dividend nor the divisor can exceed 127_{10}. Also, only positive numbers can be used. Finally, the program gets hung up in an endless loop if the initial value of the divisor is zero. While division by zero is not allowed in mathematics, some provision would be made in a practical program to recognize this eventuality. Since the program is for demonstration purposes, we will live with these shortcomings for the time being.

Converting BCD to Binary

When a microprocessor is used with a terminal such as a teletypewriter, numerals are entered as ASCII characters. For example, the number 237_{10} is entered into memory as three ASCII characters:

Numeral	ASCII Character
2	0011 0010
3	0011 0011
7	0011 0111

Notice that the four least significant bits of the ASCII character represent the BCD value of the corresponding numeral. Thus, we can convert these ASCII characters to BCD numbers simply by eliminating the four most significant bits. This technique was demonstrated in an earlier experiment.

While the microprocessor does have some BCD capability, it is often desirable to convert BCD numbers to binary. The technique for doing this illustrates another useful algorithm.

The BCD representation for 237_{10} is:

0010	←	hundreds BCD digit
0011	←	tens BCD digit
0111	←	units BCD digit

Notice that in this example 0010 represents two hundred, 0011 represents thirty, and 0111 represents seven. Because of this, there is a simple procedure for converting BCD to binary. Starting with an initial value of zero, the MPU adds 100_{10} as many times as indicated by the hundreds digit. It then adds 10_{10} as indicated by the tens digit. Finally, the value of the units digit is added on to the result. The steps involved look like this:

$$\begin{array}{rrl}
1100100_2 & 100_{10} & \left.\begin{array}{l}\\ \\ \end{array}\right\} \text{One hundred added 2 times} \\
1100100_2 & 100_{10} & \\
1010_2 & 10_{10} & \left.\begin{array}{l}\\ \\ \\ \end{array}\right\} \text{Ten added three times} \\
1010_2 & 10_{10} & \\
1010_2 & 10_{10} & \\
\underline{0111_2} & \underline{7_{10}} & \text{7 units added} \\
11101101_2 & = 237_{10} &
\end{array}$$

As you can see, this procedure ends with a binary result of 1110 1101, which is the binary representation for 237_{10}.

A flow chart for this procedure is shown in Figure 4-27. Here, the first step is to clear the binary result. We will be adding to this result, so it must start out at zero.

Next the program enters a loop in which it adds 100_{10} to the binary result the number of times indicated by the hundreds digit of the BCD number. The hundreds digit is loaded and tested for zero. If it is not zero, the hundreds digit is decremented and stored back in memory. Then the binary result is loaded and 100_{10} is added. The result is stored away and the loop is repeated. In our example, the hundreds digit was initially 2. Thus, this loop is repeated twice. The binary result will have the value 1100 1000_2 (200_{10}) when the hundreds digit is reduced to zero. At that time, the program exits the decision block via the "yes" line and immediately encounters a second loop.

The second loop is exactly like the first except that 10_{10} is added to the binary result each time the tens digit of the BCD number is decremented. Because the tens digit was initially 3, this loop is repeated three times. Ten is added to the binary result three times, bringing the result to 1110 0110_2 (230_{10}). The program exits this loop via the "yes" line on the pass after the tens digit is reduced to zero.

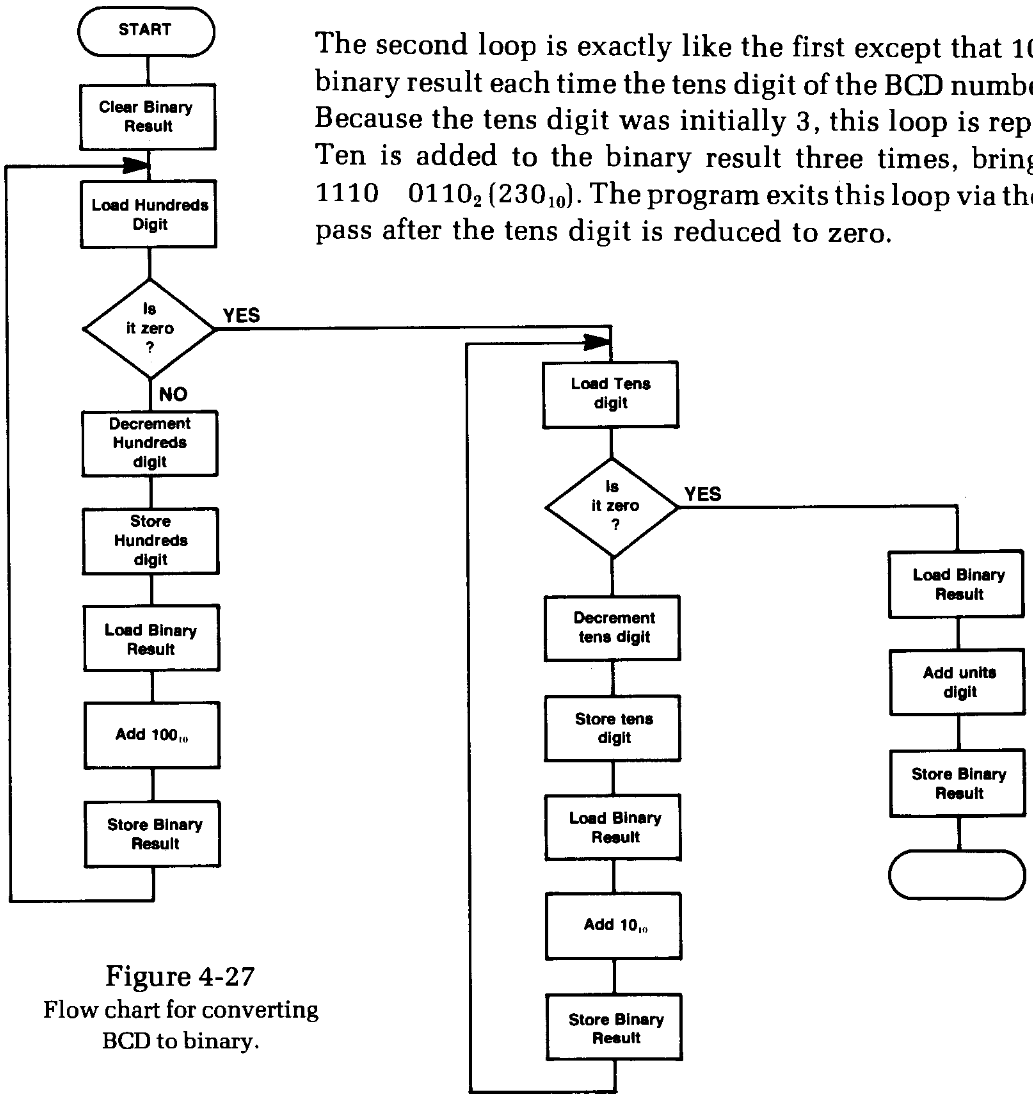


Figure 4-27
Flow chart for converting BCD to binary.

The final three blocks add the units digit to the binary result. In our example, the units digit was 7_{10}. This brings the final binary result to $1110\ \ 1101_2$. Notice that this is the proper binary representation for the unsigned number 237_{10}.

A program that carries out this operation is shown in Figure 4-28. The three digit BCD number is stored in locations 28_{16}, 29_{16}, and $2A_{16}$. the binary equivalent will be computed and placed in location $2B_{16}$. Before reading further, try to work through the program. Refer to the flow chart and the comments column as you trace out the sequence that the MPU will follow.

HEX ADDRESS	HEX CONTENTS	MNEMONIC/HEX CONTENTS	COMMENTS
00	4F	CLRA	Clear the accumulator.
01	97	STA	Store 00
02	2B	2B	in location 2B. This clears the binary result.
03	96	LDA	Load direct
04	28	28	the hundreds BCD digit.
05	27	BEQ	If the hundreds digit is zero, branch
06	0B	0B	forward to the instruction in location 12_{16}.
07	4A	DECA	Otherwise, decrement the accumulator.
08	97	STA	Store the result as the new
09	28	28	hundreds BCD digit.
0A	96	LDA	Load direct
0B	2B	2B	the binary result.
0C	8B	ADD	Add immediate
0D	64	64	100_{10} to the binary result.
0E	97	STA	Store away the new
0F	2B	2B	binary result.
10	20	BRA	Branch
11	F1	F1	back to the instruction in location 03_{16}.
12	96	LDA	Load direct
13	29	29	the tens BCD digit.
14	27	BEQ	If the tens BCD digit is zero, branch
15	0B	0B	forward to the instruction in location 21_{16}.
16	4A	DECA	Otherwise, decrement the accumulator.
17	97	STA	Store the result as the new
18	29	29	tens BCD digit.
19	96	LDA	Load direct
1A	2B	2B	the binary result.
1B	8B	ADD	Add immediate
1C	0A	0A	10_{10} to the binary result.
1D	97	STA	Store away the new
1E	2B	2B	binary result.
1F	20	BRA	Branch
20	F1	F1	back to the instruction in location 12_{16}.
21	96	LDA	Load direct
22	2B	2B	the binary result.
23	9B	ADD	Add direct
24	2A	2A	the units BCD digit.
25	97	STA	Store away the new
26	2B	2B	binary result.
27	3E	HLT	Halt.
28	02	02	Hundreds BCD digit.
29	03	03	Tens BCD digit.
2A	07	07	Unit BCD digit.
2B	—	—	Reserved for the binary result.

Figure 4-28
Program for converting BCD to binary.

Now let's briefly go through the program. The first two instructions clear the location at which the binary number will be formed.

Next, the program enters the first loop, which is shown as the first shaded area. In this loop, the hundreds digit is loaded and tested for zero. If not zero, it is decremented and stored away. Then the binary result is loaded and 100_{10} is added. The result is stored away and the loop is repeated. Because the hundreds digit was 02 initially, 100_{10} will be added to the binary result twice. Thus, upon leaving this loop, the binary result will have the value 200_{10}. The MPU escapes this loop when the BEQ instruction at address 05 detects that the hundreds digit has been reduced to zero. The branch is to the second loop which is shown as an unshaded area.

In the second loop, the tens digit is loaded and tested for zero. If not zero, it is decremented and stored away. Then the binary number is loaded, 10_{10} is added, and the result is stored away. This loop is repeated until the tens digit is reduced to zero. Because the tens digit was initially three, the loop is repeated three times so that thirty is added to the binary number. The BEQ instruction at address 14_{16} allows the MPU to escape the loop and branch to the final program segment.

This final segment is the last shaded area. Here, the binary result is loaded and the units digit is added. This brings the binary result to 237_{10}. Then the result is stored and the program halts. While the number 237_{10} was used in this example, the program will convert any BCD number between 000 and 255_{10} to its binary equivalent.

Converting Binary to BCD

A microprocessor generally manipulates data in the form of straight binary numbers. However, before the results can be transmitted to the outside world, the data is often converted back to BCD. Frequently, this is an intermediate step in converting back to ASCII.

The binary-to-BCD conversion is the reverse of the process that occurred in the previous program. The MPU must determine how many times 100_{10} can be subtracted from the binary number. The answer becomes the hundreds BCD digit. After the 100_{10} has been subtracted as many times as possible, 10_{10} is subtracted repeatedly from the remaining number. The number of subtractions becomes the tens BCD digit. Finally, after 10_{10} has been subtracted as many times as possible, the remaining number becomes the units BCD digit.

For the number 1110 1101_2 (237_{10}), the process looks likes this:

```
 1110  1101          237
-0110  0100         -100
 1000  1001          137    hundreds digit  =  2
-0110  0100         -100
 0010  0101           37
-0000  1010         - 10
 0001  1011           27    tens digit  =  3
-0000  1010         - 10
 0001  0001           17
-0000  1010         - 10
 0000  0111            7 <---units digit  =  7
```

One hundred can be subtracted twice. Thus, the hundreds digit is 2_{10} or 0010_2. From the remainder, ten can be subtracted three times. Thus, the tens digit is 3_{10} or 0011_2. Finally, the remainder of 7_{10} or 0111_2 becomes the units digit. The BCD representation is 0010 0011 0111.

Figure 4-29 shows the flow chart for this procedure. The first three blocks clear the hundreds, tens, and units digits of the BCD result. Then the binary number that is to be converted to BCD is loaded and 100_{10} is subtracted. The outcome is tested to see if a negative number resulted. If not, the result is stored away. The hundreds digit is loaded, incremented, and stored away. The loop is repeated until 100_{10} can no longer be subtracted. In our example, 100_{10} can be subtracted twice. Therefore, the hundreds digit is incremented to 2. The third subtraction of 100_{10} gives a negative result. This allows the MPU to escape the first loop.

The second loop increments the tens digit to the proper value by subtracting 10_{10} repeatedly and keeping track of the number of subtractions. In our example, this loop is repeated three times. Consequently, the tens digit is incremented to 3. The binary number that is left over after 10_{10} is subtracted the proper number of times becomes the units digit. That is, upon escaping the second loop, the remaining binary number is stored in the units digit. In our example, the remaining number, and therefore the units digit, is 7.

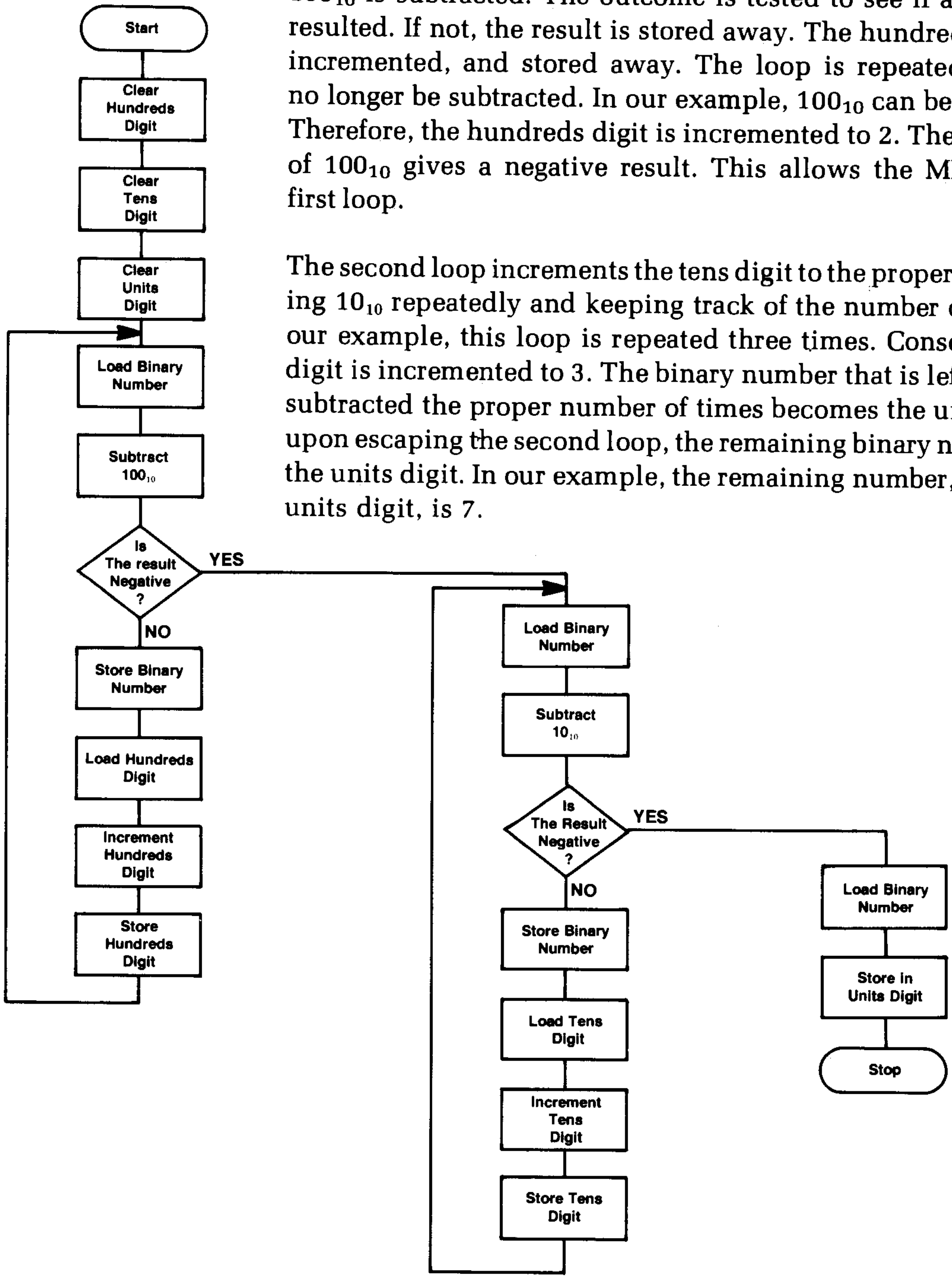


Figure 4-29
Flow chart for converting a binary number to a BCD number.

The program that carries out this procedure is shown in Figure 4.30. At this point, you should be able to interpret the program from the comments given. However, a couple of points should be explained briefly. Any unsigned binary number from 0000 0000 to 1111 1111 can be placed at address $2A_{16}$. The computer will convert this number into its BCD equivalent. The hundreds digit will appear at address $2B_{16}$, the tens digit at $2C_{16}$, and the units digit at $2D_{16}$. The decision making instructions at addresses $0B_{16}$ and $1A_{16}$ are Branch if Carry Set (BCS) instructions. Because these instructions follow immediately after SUB instructions, the carry flag will indicate whether or not a borrow occurred. In effect, the BCS instructions decide: "Was the result of the subtraction a negative number?"

HEX ADDRESS	HEX CONTENTS	MNEMONIC/HEX CONTENTS	COMMENTS
00	4F	CLRA	Clear the accumulator.
01	97	STA	Store 00
02	2B	2B	in location $2B_{16}$. This clears the hundreds digit.
03	97	STA	Store 00
04	2C	2C	in location $2C_{16}$. This clears the tens digit.
05	97	STA	Store 00
06	2D	2D	in location $2D_{16}$. This clears the units digit.
07	96	LDA	Load direct
08	2A	2A	the binary number to be converted.
09	80	SUB	Subtract immediate
0A	64	64	100_{10}.
0B	25	BCS	If a borrow occurred, branch
0C	09	09	forward to the instruction in location 16_{16}.
0D	97	STA	Otherwise, store the result of the subtraction
0E	2A	2A	as the new binary number.
0F	96	LDA	Load direct
10	2B	2B	the hundreds digit of the BCD result.
11	4C	INCA	Increment the hundreds digit.
12	97	STA	Store the hundreds digit
13	2B	2B	back where it came from.
14	20	BRA	Branch
15	F1	F1	back to the instruction at address 07_{16}
16	96	LDA	Load direct
17	2A	2A	the binary number.
18	80	SUB	Subtract immediate
19	0A	0A	10_{10}.
1A	25	BCS	If a borrow occurred, branch
1B	09	09	forward to the instruction in location 25_{16}.
1C	97	STA	Otherwise, store the result of the subtraction
1D	2A	2A	as the new binary number.
1E	96	LDA	Load direct
1F	2C	2C	the tens digit.
20	4C	INCA	Increment the tens digit.
21	97	STA	Store the tens digit
22	2C	2C	back where it came from.
23	20	BRA	Branch
24	F1	F1	back to the instruction at address 16_{16}.
25	96	LDA	Load direct
26	2A	2A	the binary number.
27	97	STA	Store it in
28	2D	2D	the units digit.
29	3E	HLT	Halt.
2A	—	—	Place binary number to be converted at this address.
2B	—	—	Hundreds digit } Reserved for BCD result.
2C	—	—	Tens digit } Reserved for BCD result.
2D	—	—	Units digit } Reserved for BCD result.

Figure 4-30
Program for converting a binary number to a BCD number.

Self-Test Review

25. What is an algorithm?

26. What type of instruction is used to make a decision?

27. Refer to the program in Figure 4-24. If the multiplier is 8_{16} and the multiplicand is 15_{16}, how many times will the BEQ instruction be executed?

28. Refer to the program in Figure 4-26. What is the largest number that can be used as a dividend?

29. How could this program be modified so it could handle unsigned dividends up to 255_{10}?

30. When this program halts, where will the remainder be located?

31. Refer to the program in Figure 4-28. Assume that addresses 28_{16}, 29_{16}, and $2A_{16}$ contain 01, 09, and 08 respectively. How many times will 100_{10} be added to address $2B_{16}$?

32. How many times will 10_{10} be added?

33. Refer to the program in Figure 4-30. What is the purpose of the first four instructions?

34. What is the largest binary number that this program can convert to BCD?

Answers

25. An algorithm is a step-by-step procedure for doing a particular job.

26. Conditional branch instruction.

27. Nine times.

28. $+127_{10}$ or $0111\ 1111_2$.

29. Change the BMI instruction to BCS.

30. At address 11_{16}.

31. Once.

32. Nine times.

33. The first four instructions clear the locations where the BCD equivalent will be stored.

34. $1111\quad 1111_2$ or 255_{10}.

ADDITIONAL INSTRUCTIONS

Before leaving this unit, you should also look at four additional instructions. The names, opcodes and mnemonics of these instructions are shown in Figure 4-31.

NAME	MNEMONIC	HEX OPCODE		
		IMMEDIATE	DIRECT	INHERENT
ADD WITH CARRY	ADC	89	99	
SUBTRACT WITH CARRY	SBC	82	92	
ARITHMETIC SHIFT ACCUMULATOR LEFT	ASLA			48
DECIMAL ADJUST ACCUMULATOR	DAA			19

Figure 4-31
Four new instructions.

Recall that the ALU always adds numbers as if they were unsigned binary numbers. When it adds 8-bit numbers, a carry often occurs from the MSB, setting the C flag. Thus, you can think of the carry flag as an extension of the accumulator. Let's look at some instructions that use the carry flag.

Add With Carry (ADC) Instruction

This instruction is similar to the ADD instruction discussed earlier with one important difference. If the carry bit is set to 1 before this instruction is executed, 1 is added to the LSB of the sum. However, if the carry bit is 0 prior to execution, then no carry is added. The effect is the same as having the carry bit from the previous operation added to the result of the present operation.

Like the ADD instruction, the ADC instruction has two addressing modes: immediate and direct. As shown in Figure 4-31, the opcode for "ADD With Carry Immediate" is 89_{16}, while the opcode for "Add With Carry Direct" is 99_{16}.

A primary use of the ADC instruction is to simplify **multiple-precision arithmetic**. Multiple-precision means that two or more bytes are used to represent a number. Recall that a single byte can represent unsigned binary numbers with values up to 255_{10}. However, much larger numbers can be represented by using two or more bytes. Two bytes (16 bits) can represent unsigned binary values up to $2^{16}-1$ or $65{,}535_{10}$. Three bytes can represent values to $16{,}777{,}215_{10}$; etc. Thus, the MPU can handle numbers of virtually any size simply by stringing the proper number of bytes together.

Suppose, for example, that two very large numbers are to be added. Figure 4-32 shows how the addition might look on paper. Notice that two 24-bit numbers are being added to form a 24-bit sum. The MPU is restricted to operating on data in 8-bit bytes. Thus, each quantity involved must be represented by three bytes.

1 1 11	111	1	← Carries
0100 1010	1100 0000	1110 1010	← Addend
+ 0110 0110	0001 1011	1001 0011	← Augend
1011 0000	1101 1100	0111 1101	← Sum
Byte 3	Byte 2	Byte 1	

Figure 4-32
Multiple-precision addition.

The MPU must be instructed to add the first byte of the addend to the first byte of the augend. This forms the first byte of the sum. Next the MPU must add the second bytes of the addend and augend. However, you will notice that there was a carry from the first byte to the second byte. If this carry is not added with the second bytes, the sum will be in error. The ADC instruction performs this operation automatically.

The program for adding the multiple-byte numbers could be written as shown in Figure 4-33. The three byte addend is stored in locations 13_{16} through 15_{16} while the augend is stored in locations 16_{16} through 18_{16}. Verify that the hexadecimal contents shown are the same as the binary values given in Figure 4-32.

HEX ADDRESS	HEX CONTENTS	MNEMONIC/HEX CONTENTS	COMMENTS
00	96	LDA	Load accumulator direct with
01	13	13	least significant byte of addend.
02	9B	ADD	Add direct
03	16	16	least significant byte of augend.
04	97	STA	Store result in
05	19	19	least significant byte of sum.
06	96	LDA	Load accumulator direct with
07	14	14	next byte of addend.
08	99	ADC	Add with carry direct
09	17	17	next byte of augend.
0A	97	STA	Store result in
0B	1A	1A	next byte of sum.
0C	96	LDA	Load accumulator direct with
0D	15	15	most significant byte of addend.
0E	99	ADC	Add with carry direct
0F	18	18	most significant byte of augend.
10	97	STA	Store result in
11	1B	1B	most significant byte of sum.
12	3E	HLT	Halt.
13	EA	EA	Least significant byte } Addend.
14	C0	C0	
15	4A	4A	Most significant byte
16	93	93	Least significant byte } Augent
17	1B	1B	
18	66	66	Most significant byte
19	—	—	Least significant byte } Reserved for for sum.
1A	—	—	
1B	—	—	Most significant byte.

Figure 4-33
Program for multiple-precision addition.

The first two instructions add the least significant bytes of the addend and augend. The ADD instruction is used because the MPU need not consider earlier carries. The first byte of the resulting sum is stored in location 19_{16}.

The next two instructions add the next two bytes. This time the ADC instruction is used because the MPU must consider the carry from the previous addition. The second byte of the sum is placed in location $1A_{16}$.

Finally, the last two bytes are added using the ADC instruction. The final byte of the sum is stored in location $1B_{16}$. The program halts when the addition is completed.

Subtract With Carry (SBC) Instruction

This instruction simplifies multiple-precision subtraction. You will recall that during subtract operations the carry flag indicates whether or not a borrow operation occurred. For this reason, this instruction can be thought of as a subtract with borrow operation.

The SBC instruction subtracts the subtrahend from the minuend just as the SUB instruction did. However, the SBC instruction has an additional step in that the carry bit is also subtracted. As with the other add and subtract instructions, both immediate and direct addressing modes are possible. The opcodes for both modes are shown in Figure 4-31.

Figure 4-34 illustrates how multiple-precision numbers can be subtracted. Notice that, during the course of this subtraction, byte 1 must "borrow" a 1 from byte 2. The SBC instruction allows the MPU to do this.

```
   11               |   1              Borrows
   1001  0111       |   0010  0001  ← Minuend
 - 0111  0101       |   1000  0001  ← Subtrahend
 ------------------------------------
   0010  0001       |   1010  0000  ← Difference
                    |
   Byte 2           |   Byte 1
```

Figure 4-34
Multiple-precision subtraction.

HEX ADDRESS	HEX CONTENTS	MNEMONIC/HEX CONTENTS	COMMENTS
00	96	LDA	Load accumulator direct with
01	0D	0D	least significant byte of minuend.
02	90	SUB	Subtract direct
03	0F	0F	least significant byte of subtrahend.
04	97	STA	Store result in
05	11	11	least significant byte of difference.
06	96	LDA	Load accumulator direct with
07	0E	0E	most significant byte of minuend.
08	92	SBC	Subtract with carry
09	10	10	most significant byte of the subtrahend.
0A	97	STA	Store result in
0B	12	12	most significant byte of the difference.
0C	3E	HLT	Halt
0D	21	21	Least significant byte } Minuend.
0E	97	97	Most significant byte } Minuend.
0F	81	81	Least significant byte } Subtrahend.
10	75	75	Most significant byte } Subtrahend.
11	—	—	Least significant byte } Difference.
12	—	—	Most significant byte } Difference.

Figure 4-35
Program for multiple-precision subtraction.

Figure 4-35 shows a simple program for performing the subtraction. The double-precision minuend is at addresses $0D_{16}$ and OE_{16}, while the subtrahend is at addresses $0F_{16}$ and 10_{16}. The program computes the difference and stores it in location 11_{16} and 12_{16}.

The first instruction loads the least significant byte of the minuend. Next, the corresponding byte of the subtrahend is subtracted. Since the subtrahend byte is larger, a borrow is indicated. Consequently, the carry flag is set to 1. Notice that the SUB rather than the SBC instruction is used. This is done because the first byte should not be affected by any previous borrow or carry. The result of the subtraction is stored away to become the least significant byte of the difference.

The most significant byte of the minuend is loaded next and the corresponding byte of the subtrahend is subtracted. However, this time the SBC instruction is used. And since the carry flag is set, an additional 1 is subtracted from the minuend to complete the borrow operation. The result of the subtraction becomes the most significant byte of the difference.

Arithmetic Shift Accumulator Left (ASLA) Instruction

The ASLA instruction shifts the contents of the accumulator to the left by one space. Figure 4-36 illustrates the repeated execution of this instruction. Figure 4-36A shows the condition of the accumulator and carry bit. In this example, the number in the accumulator is arbitrarily assumed to be 10_{10}. Also, the carry bit is arbitrarily assumed to be cleared.

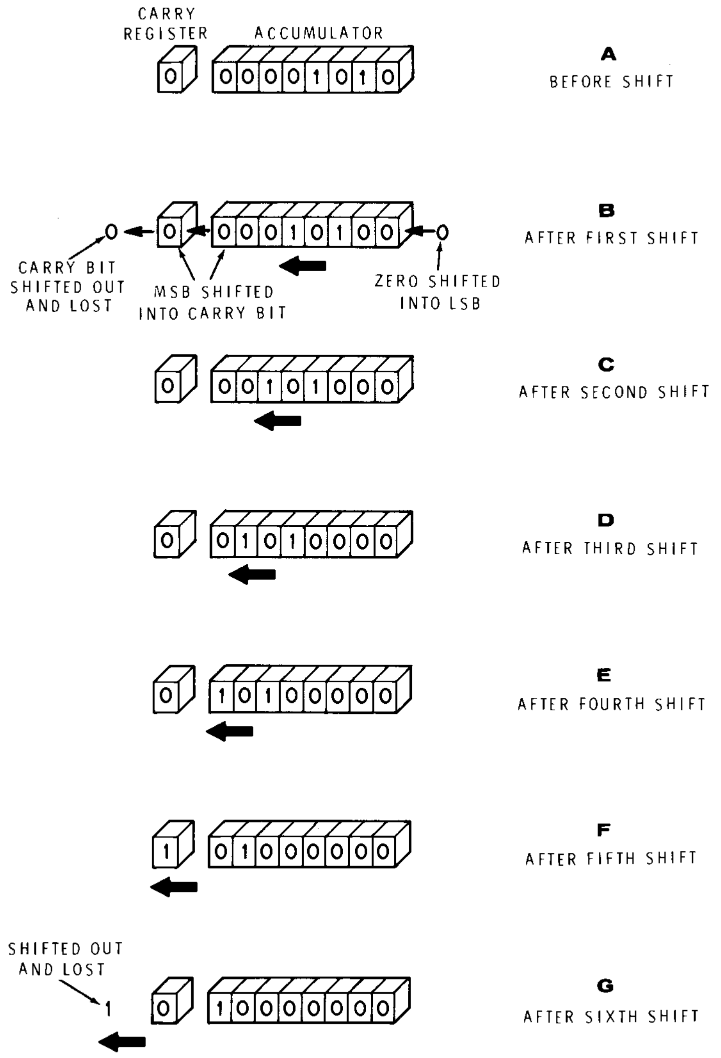


Figure 4-36
Repeatedly implementing the ASLA instruction.

Figure 4-36B shows the contents of the accumulator and carry bit after the ASLA instruction is executed. Notice that the number is shifted one bit to the left. Also, a 0 is shifted into the LSB. At the same time, the MSB is shifted into the carry bit. The old carry bit is shifted out and is lost.

You can understand one purpose of this instruction by examining the numbers in the accumulator before and after the instruction is executed. Before the shift, the number is 10_{10}; afterwards the number is 20_{10}. The number has been doubled. If you will try several different examples, you will see that any binary number can be multiplied by two simply by shifting the number one bit to the left. This holds true as long as the capacity of the accumulator is not exceeded.

Figures 4-36C through G show what happens if the MPU continues to execute ASLA instructions. The number continues to double. The number in the accumulator becomes 40_{10}, then 80_{10}, then 160_{10}. Each shift multiplies the number by two. On the fifth shift, the capacity of the accumulator is exceeded as the most significant 1 bit shifts into the carry bit. After the sixth shift, the leading 1 is lost altogether. When you use this technique to multiply by two or by a power of two, you must not exceed the capacity of the accumulator.

Another use of the ASLA instruction is to pack two BCD digits in a single byte. Earlier when we worked with BCD numbers, we assumed that each BCD digit resided in a separate memory byte. However, because a BCD digit has only 4 bits, memory space is wasted by assigning each digit a separate byte. Frequently, it is more desirable to "pack" two BCD digits into a single byte. A simple routine for doing this is shown in Figure 4-37. If dozens of BCD numbers are to be manipulated, a routine that uses a procedure similar to this can save substantial memory space. At the same time, it puts the BCD numbers into a more convenient and usable form.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/HEX CONTENTS	COMMENTS
00	96	LDA	Load into the accumulator direct
01	0C	0C	the unpacked most significant BCD digit.
02	48	ASLA	} Shift it four places to the left.
03	48	ASLA	
04	48	ASLA	
05	48	ASLA	
06	9B	ADD	Add
07	0D	0D	the unpacked least significant BCD digit.
08	97	STA	Store the result as
09	0B	0B	two packed BCD digits.
0A	3E	HLT	Halt
0B	—	—	Packed BCD digits,
0C	—	—	Most significant BCD digit (unpacked).
0D	—	—	Least significant BCD digit (unpacked).

Figure 4-37
Program for packing two BCD digits into a single byte.

The procedure carried out by the program is quite simple. The most significant BCD digit is loaded into the accumulator. It is then shifted four places to the left to make room for the least significant BCD digit. The least significant digit is then added to form a packed BCD number. The resulting single byte number is stored back in memory.

Decimal Adjust Accumulator (DAA) Instruction

Earlier in this unit, the problems of converting from BCD to binary and back again were considered. While this conversion is frequently necessary, many microprocessors have some limited BCD arithmetic capabilities. Our hypothetical MPU has an instruction that greatly simplifies BCD arithmetic. It is called the Decimal Adjust Accumulator (DAA) Instruction. When used in conjunction with the ADD or ADC instruction, it allows the MPU to add BCD numbers directly without an intermediate binary conversion.

Recall that the ALU adds input data bytes as if they were unsigned binary numbers. Therefore, if two BCD digits are added, the sum may be incorrect. For example, assume that the MPU adds the BCD digits 0111 and 0101. The ALU produces the result

$$\begin{array}{r} 1 \\ 0111 \\ +\ 0101 \\ \hline 1100 \end{array}$$

This answer is the correct **binary** result, 12_{10}; but it is not the proper BCD result. Recall that in BCD, 12_{10} is represented as 0001 0010. Notice that you can obtain the proper BCD result by adding 0110_2 to the binary result. The addition of 0110_2 is necessary anytime that the binary result exceeds 1001_2.

To produce the proper BCD result when adding two BCD digits, the MPU must follow this procedure:

1. If the sum is 1001_2 or less, use the sum as the single digit BCD result.

2. If the sum is greater than 1001_2, add 0110_2 and use the result as a 2-digit BCD number.

The situation becomes more complex when packed BCD numbers are added. Consider adding 0111 1001_{BCD} (79_{10}) to 0111 0011_{BCD} (73_{10}). The ALU adds these packed BCD numbers as if they were unsigned binary numbers. The result is

```
  11    1
0111 1001
0111 0011
---------
1110 1100
```

Notice that the result is not a BCD number, since both 4-bit groups exceed 1001_2. Even so, the sum can be converted to BCD by adding 0110_2 to each 4-bit group. The result is

```
1  1 1  1
  1110 1100
  0110 0110
-----------
1 0101 0010
```

There is a carry from bit 7 that sets the carry bit. This carry bit becomes the most significant BCD digit. Thus, the final BCD result is 0001 0101 0010_{BCD} or 152_{10}.

If you consider all possible combinations of BCD numbers, you will find that four different situations exist:

1. When some BCD numbers are added, the binary result produced by the ALU is equal to the proper BCD representation. This occurs when both BCD digits of the result are 1001_2 or less.

2. The binary sum is adjusted by adding 06_{16} if the least significant BCD digit exceeds 1001_2 but the most significant BCD digit does not.

3. The binary sum is adjusted by adding 60_{16} if the most significant BCD digit exceeds 1001_2 but the least significant BCD digit does not.

4. The binary sum is adjusted by adding 66_{16} if both BCD digits exceed 1001_2.

While this procedure could be programmed, it would be much better if the MPU performed these operations automatically. Fortunately, our hypothetical microprocessor does this. The programmer simply informs the MPU that the numbers being added are BCD numbers. The MPU automatically computes the proper BCD result. The way the programmer informs the MPU is via the DAA instruction. When the DAA instruction is placed immediately after an ADD or ADC instruction, the MPU automatically converts the sum to the proper BCD number.

Suppose, for example, that you wish to add two BCD numbers. Assume the numbers are 3792_{10} and 5482_{10}. Naturally, the sum should be 9274_{10}. A program for solving this problem is shown in Figure 4-38. The BCD addend (3792_{10}) is in addresses OF_{16} and 10_{16}. The augend (5482_{10}) is in locations 11_{16} and 12_{16}. The BCD sum will be placed in locations 13_{16} and 14_{16}.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/HEX CONTENTS	COMMENTS
00	96	LDA	Load into the accumulator direct
01	10	10	the least significant half of the addend.
02	9B	ADD	Add
03	12	12	the least significant half of the augend.
04	19	DAA	Decimal adjust the sum to BCD.
05	97	STA	Store the result as the
06	14	14	least significant half of the sum.
07	96	LDA	Load
08	0F	0F	the most significant half of the addend.
09	99	ADC	Add
0A	11	11	the most significant half of the augend.
0B	19	DAA	Decimal adjust the sum to BCD.
0C	97	STA	Store the result as the
0D	13	13	most significant half of the sum.
0E	3E	HLT	Halt
0F	37	37	} BCD Addend
10	92	92	
11	54	54	} BCD Augend.
12	82	82	
13	—	—	} Reserved for BCD sum.
14	—	—	

Figure 4-38
Program for adding multiple-precision BCD numbers.

The first two instructions add the least significant halves of the addend and augend. The ADD instruction is followed immediately by the DAA instruction. Therefore, the sum is adjusted to a packed BCD number. The result is stored in location 14_{16} as the lower half of the BCD sum.

Next, the upper halves of the addend and augend are added. This time, the ADC instruction is used because the carry from the previous addition must be added in. Again, the DAA instruction adjusts the sum to BCD. The result is stored as the upper half of the BCD sum.

The DAA instruction must be used properly. It can be used only with addition. Also, it must be used immediately after the addition instruction. It can not be used to convert just any binary number to BCD. It only converts the sum of BCD numbers to the BCD format.

Self-Test Review

35. How is the ADC instruction different from the ADD instruction?

36. How is the SBC instruction different from the SUB instruction?

37. A primary use of the ADC and SBC instructions is in ____________-____________ arithmetic.

38. The accumulator contains the number 7_{10}. If two ASLA instructions are executed, what number will be in the accumulator?

39. What is the difference between packed and unpacked BCD numbers?

40. When adding unpacked BCD numbers, under what condition must 0110_2 be added to the sum in order to form a BCD sum?

41. What instruction is used to automatically adjust the sum to the proper BCD format when two BCD numbers are added?

42. Can the DAA instruction be used after a SUB instruction to produce the proper BCD difference when two BCD numbers are subtracted?

Answers

35. When the ADC instruction is executed, an additional 1 is added to the sum if the carry flag is set.

36. When the SBC instruction is executed, an additional 1 is subtracted from the difference if the carry flag is set.

37. Multiple-precision.

38. The first ASLA instruction multiplies the number by two, giving 14_{10}. The second ASLA doubles this number, giving 28_{10}.

39. With packed BCD numbers, each byte contains two BCD digits. With unpacked BCD numbers, each byte contains one BCD digit.

40. When two BCD digits are added, 0110_2 must be added to the sum if the sum exceeds 1001_2.

41. The decimal adjust accumulator (DAA) instruction.

42. No. The DAA instruction is used in conjunction with add instructions only.

UNIT 4 SUMMARY

1. The instructions in "straight line" programs are executed one after another in the order in which they are written.
2. The process called looping allows a section of a program to be repeated as often as it is needed.
3. Jump and branch instructions allow a programmer to create program loops.
4. Branch instructions use relative addressing to control the operation of the program loop.
5. The relative address is a value that is added to the MPU program counter to determine the address of the next instruction to be executed in the program.
6. The relative address is found in the second byte of a branch instruction.
7. Branching forward, the relative address is a positive value. Branching backward, the relative address is a negative value written in two's complement form.
8. The unconditional branch instruction forces the MPU to always branch when the instruction is executed.
9. The conditional branch instruction will only force the MPU to branch when a specified condition is met. If the condition is not met, the branch instruction will be ignored.
10. Conditional execution of a branch instruction is determined by the status of four condition code registers. These single bit registers are called carry, negative, zero, and overflow.
11. The carry condition code register acts as an extension of the accumulator. If a carry or borrow is generated during an arithmetic operation, the carry register bit is set. If no carry or borrow occurred, the bit is cleared.

12. The negative condition code register monitors the MSB of the accumulator. If that bit is one, the negative register is set. If the MSB of the accumulator is zero, the negative register bit is cleared. Not all accumulator operations affect the negative condition code register.

13. The zero condition code register monitors all bits of the accumulator. If they are all zero after an operation, the zero register bit is set. If any of the accumulator bits are one, the zero register bit is cleared. Most accumulator operations affect the status of the zero register.

14. The overflow condition code register monitors the status of the sign bits of the two numbers being added, subtracted, or compared, and the sign bit of the result. During addition, if the sign bits of the two numbers are different, there is no overflow, the register is cleared. If the sign bits of the two numbers, and the result are the same, there is, again, no overflow. If the sign bits of the two numbers are the same, but the sign bit of the result is different, there is an overflow, and the overflow register is set.

15. Writing complex and lengthy programs requires some form of organization for problem solving. Generally there are three steps to writing a program: define the problem, map or flowchart the solution, write the program.

16. A flowchart is a form of project blueprint.

17. Standard flowchart symbols should be used to allow any programmer to interpret the program solution.

18. The flowchart symbol terminal indicates the beginning and end of a flowchart.

19. The flowchart symbol operations box is used to represent a transfer process, an algebraic process, or any process that doesn't have a specific symbol.

20. The flowchart symbol decision box is used to indicate a logical choice between two conditions.

21. Flowchart flow lines and connectors are used to tie the symbols and sections in a flowchart together.

22. The five most common mathematical symbols used in a flowchart are: equal to, greater than, less than, greater than or equal to, and less than or equal to.

23. The unconditional branch in a flowchart corresponds to the unconditional branch in a program.

24. The conditional branch in a flowchart corresponds to the conditional branch in a program.

25. An algorithm is a step-by-step procedure for doing a particular job that cannot be handled directly by the microcomputer, such as multiplication.

26. Multiplication in a 6800-based microcomputer is best handled by a algorithm that uses a program loop to repeatedly add a number to itself the number of times specified by the multiplier.

27. Division in a 6800-based microcomputer is best handled by an algorithm that uses a program loop to repeatedly subtract the divisor from the dividend, and keep track of the count and any remainder.

28. Most data received by or transmitted from a microcomputer is in ASCII coded form.

29. The least significant four bits of an ASCII coded character represent the BCD value of the corresponding numeral.

30. To convert ASCII coded numerals to BCD code, subtract the binary value 00110000 from the ASCII code.

31. Multiple-digit BCD code is converted to binary by adding the positional weight of each code digit to itself, in binary, the number of times specified by the digit. Then all of the binary sums are added together.

32. Binary data is converted to BCD code by repeatedly subtracting the binary equivalent of the hundreds digit, the tens digit, and the ones digit from the binary value, beginning with the hundreds digit. The number of times each digit can be subtracted represents the BCD value for that decimal positional weight.

33. To convert BCD code to ASCII code, add the binary value 00110000 to the BCD code.

34. Multiple-precision numbers use two or more bytes to represent the number.

35. The add with carry and subtract with carry (borrow) instructions are used to simplify multiple-precision arithmetic.

36. The ASLA instruction shifts the MSB of the accumulator into the carry condition code register and a zero into the LSB.

37. The DAA instruction is used to adjust the result after a BCD arithmetic operation. A carry into the next digit position is recorded in the carry condition code register.

Perform Experiments 5 and 6 at the end of Book 2.

Unit 5

THE 6800 MICROPROCESSOR — PART 1

INTRODUCTION

Until now, we have confined our study to a simple hypothetical microprocessor. Obviously, though, this hypothetical model must be very close to the real thing, since we have been running its programs on the ET-3400 or ET-3400A Microprocessor Trainer. In this unit you will begin your study of the actual microprocessor upon which our hypothetical model is based. The microprocessor in the ET-3400 Trainer is called the 6800. It was first released by Motorola in the mid 1970's. Today, it is being second sourced by several other companies.

The microprocessor in the ET-3400A Trainer is called the 6808. The primary difference between the 6808 and 6800 microprocessors is the means by which clock signals are generated. The 6808 has an on-chip clock circuit, the 6800 does not. Because of this one difference, there will also be some chip pin assignments that are not identical. These differences will be discussed in more detail in Unit Seven of this course. However, in all other characteristics, such as the instruction set and internal registers, the 6800 and 6808 are identical. It is these identical characteristics which are the subject of this and the following unit. Therefore, throughout this unit reference is made to the 6800 microprocessor only. But, the data presented also applies to the 6808 in the ET-3400A Trainer.

There are other microprocessors that have similar instruction sets, architecture, and addressing modes. Thus, by becoming familiar with the 6800 (6808) you should be able to understand and use a wide range of microprocessors.

You already know a great deal about the 6800 and/or 6808 microprocessor. You have been programming this device for the past several units. The main difference between the 6800 (6808) microprocessor and our hypothetical model is complexity. As you will see, the 6800 (6808) is a vastly expanded version of our hypothetical model.

UNIT OBJECTIVES

When you have completed this unit you will be able to:

1. Draw a programming model of the 6800 MPU.
2. Explain the purpose of each block in a simplified block diagram of the 6800 MPU.
3. Using Appendix A and Figure 5-24 as references, explain the operation of all the instructions discussed in this unit.
4. Write simple programs that use indexed and extended addressing.
5. Using Figure 5-24 as a guide, find the opcode, number of MPU cycles, number of bytes, and effects on the condition code flags of every instruction discussed in this unit.

ARCHITECTURE OF THE 6800 MPU

In computer jargon, the word architecture is used to describe the computer's style of construction, its register size and arrangement, its bus configuration, etc. The architecture of our hypothetical microprocessor is shown for one last time in Figure 5-1. By now you should be quite familiar and comfortable with this architecture.

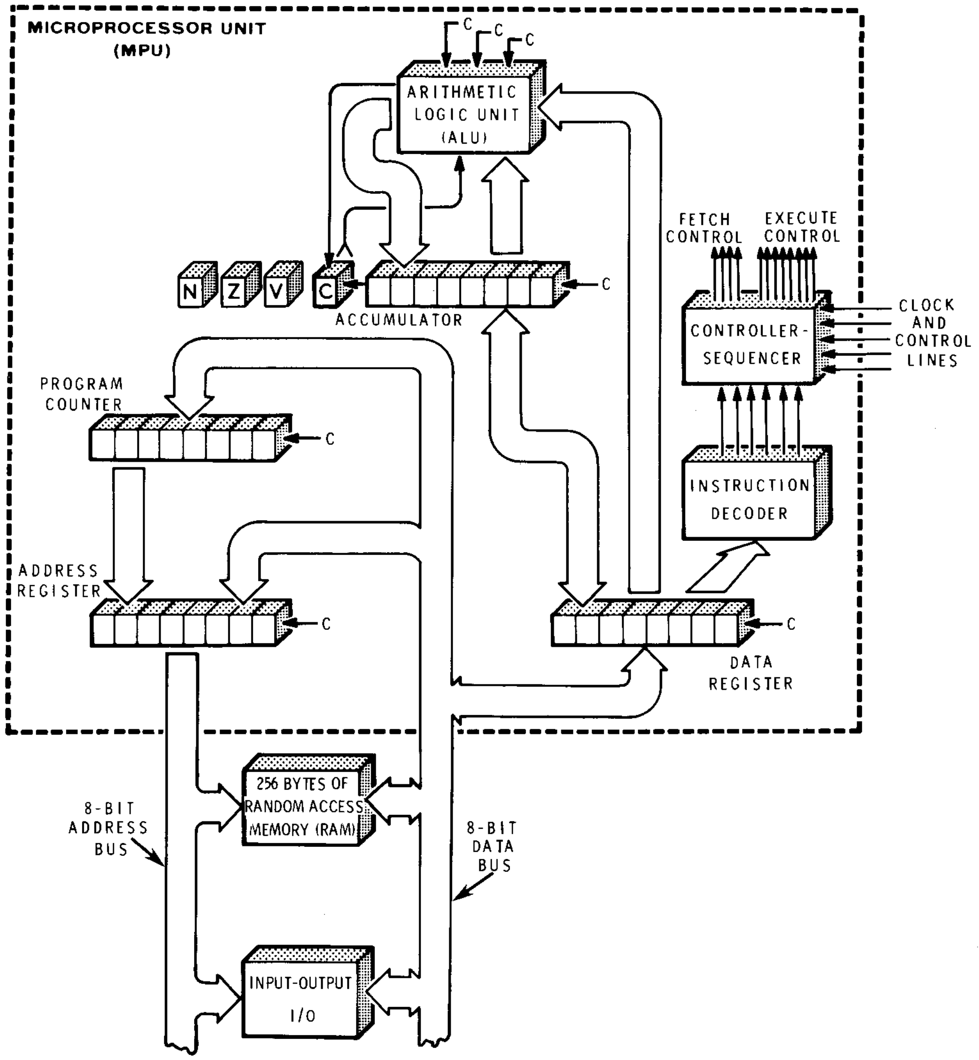


Figure 5-1
Architecture of the hypothetical microcomputer.

The only reason for showing the details of the model is to give you an idea of what goes on inside the integrated circuit. In an actual microprocessor the internal structure is often so complex that we become bogged down in details if we attempt to analyze it too closely. For this reason, a programming model is generally used when a microprocessor is being introduced for the first time. In the programming model, the emphasis is shifted upward by an order of magnitude. Any register or circuit that cannot be directly controlled by the programmer is simply ignored. Consider the data register for example. There are no instructions that give the programmer direct control over this register. That is, there are no instructions such as Load Data Register, Store Data Register, etc. All data register activity is controlled strictly by the MPU. Thus, the programmer can simply ignore the existence of this register. The same is true of the address register, the instruction decoder, the controller-sequencer, etc. Therefore, the programming model of our hypothetical MPU can be represented as shown in Figure 5-2. This simple diagram is sufficient for most programming applications since it shows all the registers that can be directly controlled by the program.

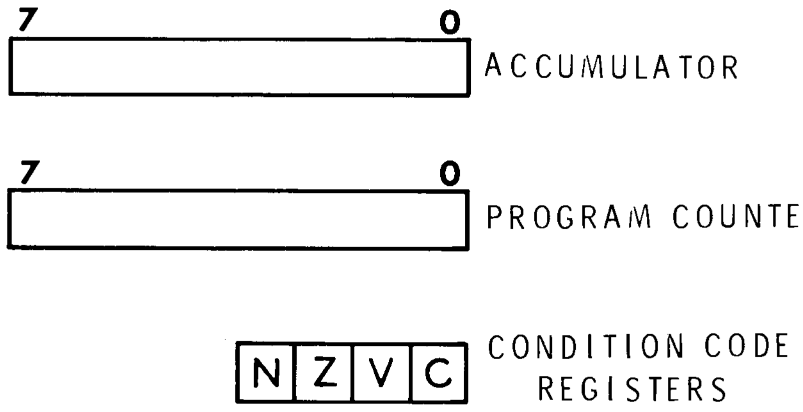


Figure 5-2
Programming model of the hypothetical MPU.

Programming Model of the 6800 MPU

The 6800 MPU is much more complex than our hypothetical MPU. Consequently, a programming model of the 6800 makes a good starting point. The programming model is shown in Figure 5-3.

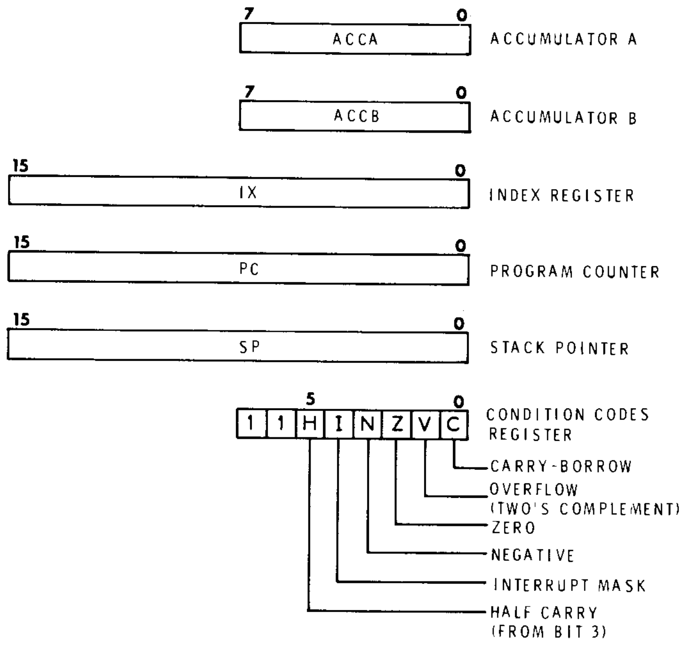


Figure 5-3
Programming model of the 6800 MPU.

You will notice immediately that the 6800 MPU has several additional registers. However, only two of these, the index register and the stack pointer, are actually new to you. Let's look at the major differences between this MPU and our hypothetical model.

Two Accumulators The 6800 MPU has two accumulators instead of one. They are called accumulator A (ACCA) and accumulator B (ACCB). Each has its own group of instructions associated with it. The names and mnemonics of the instructions specify which accumulator is to be used. Thus, there are instructions such as:

Load Accumulator A (LDAA)
Load Accumulator B (LDAB)
Store Accumulator A (STAA)
Store Accumulator B (STAB)

Notice that a letter is added to both the name and the mnemonic to indicate which accumulator is being used.

From your previous programming experience, you can visualize the value of a second accumulator. For example consider a program in which the MPU counts the number of times that some operation occurs. In the past, we stored the number that the accumulator was presently working on, loaded the count into the accumulator; incremented the count; stored the count; and reloaded the original number. With a second accumuator, none of this is necessary. We can simply maintain the count in accumulator B while working with the number in accumulator A. In fact, we can perform any arithmetic or logic operation on two different numbers without having to shift the numbers back and forth between memory.

16-Bit Program Counter The program counter in the 6800 has 16_{10} bits rather than 8. Thus, it can specify $65{,}536_{10}$ different addresses. This means that a 6800 based microcomputer can have up to $65{,}536_{10}$ bytes of memory. Most applications require substantially less memory than this maximum number. Fortunately, we can use as little or as much memory as we need up to the 2^{16} byte limit.

Since the program counter has 16_{10} bits, the address bus must also be 16-bits wide. Contrast this with the 8-bit address bus of our hypothetical machine.

You may wonder how we specify a 16-bit address with an 8-bit byte. The obvious answer is that two 8-bit bytes are required. Recall that in the direct addressing mode, the address was specified by a single 8-bit byte. The 6800 microprocessor retains this addressing mode. However, since an 8-bit address can specify only 256_{10} addresses, the 6800 MPU can use this mode only if the operand is in the first 256_{10} bytes of memory. To reach higher addresses, a new addressing mode called **extended addressing** must be used. In the extended addressing mode, two bytes are used to represent each address. This addressing mode will be discussed in more detail later. For now, keep in mind that there are $65{,}536_{10}$ possible addresses. The lowest address is 0000_{16} and the highest is $FFFF_{16}$. Using extended addressing, we have access to any location in memory, but a 2-byte address is required.

Condition Code Registers The 6800 MPU has six condition codes. Four of these are almost identical to those discussed in an earlier unit. These include the negative (N), zero (Z), overflow (V) and carry (C) condition codes. The difference arises because there are two accumulators in the 6800 MPU. Thus, the carry flag is set whenever there is a carry from either accumulator. By the same token, an overflow in either accumulator will set the V flag. Later in this unit, you will see how the condition codes are affected by each instruction.

Two new condition codes are shown in Figure 5-3. The I flag is called an interrupt mask. We will discuss this flag later when you study interrupts. The other is called the half carry flag (H). The H flag is set when there is a carry from bit 3 of the accumulator. The MPU uses this flag to determine how to implement the decimal adjust instruction.

These six flags make up bits 0 through 5 of an 8-bit register. Bits 6 and 7 of the condition code register are not used and are always set to 1. Additional details of the condition codes will be brought out as the need arises.

Index Register The index register is a special-purpose, 16-bit register that greatly increases the power of the microprocessor. It allows a powerful address mode called indexed addressing. We will examine this addressing mode later in this unit. For now, consider the index register to be just another working register. The fact that it holds two bytes instead of one can be put to good use. The MPU has instructions that allow the index register to be loaded from two adjacent memory bytes. Another instruction allows us to store the contents of the index register in two adjacent memory locations. This allows us to move data in 2-byte groups. Also, the index register can be incremented and decremented. This lets us maintain 16-bit tallies.

Stack Pointer The stack pointer is another special-purpose 16-bit register. It allows the MPU and the programmer to use a section of RAM as a last in, first out (LIFO) memory. This capability is extremely valuable when using subroutines or when processing interrupts. These aspects of the stack pointer will be discussed in the next unit. For the time being let's consider the stack pointer to be another 16-bit working register. It too can be loaded from memory, stored in memory, incremented, and decremented.

Block Diagram of the 6800 MPU

Now that you have seen the programming model of the 6800 MPU, take a look at the block diagram. A simplified block diagram is shown in Figure 5-4. Several data paths, most control lines, and a temporary storage register have been omitted in favor of the major data paths and registers.

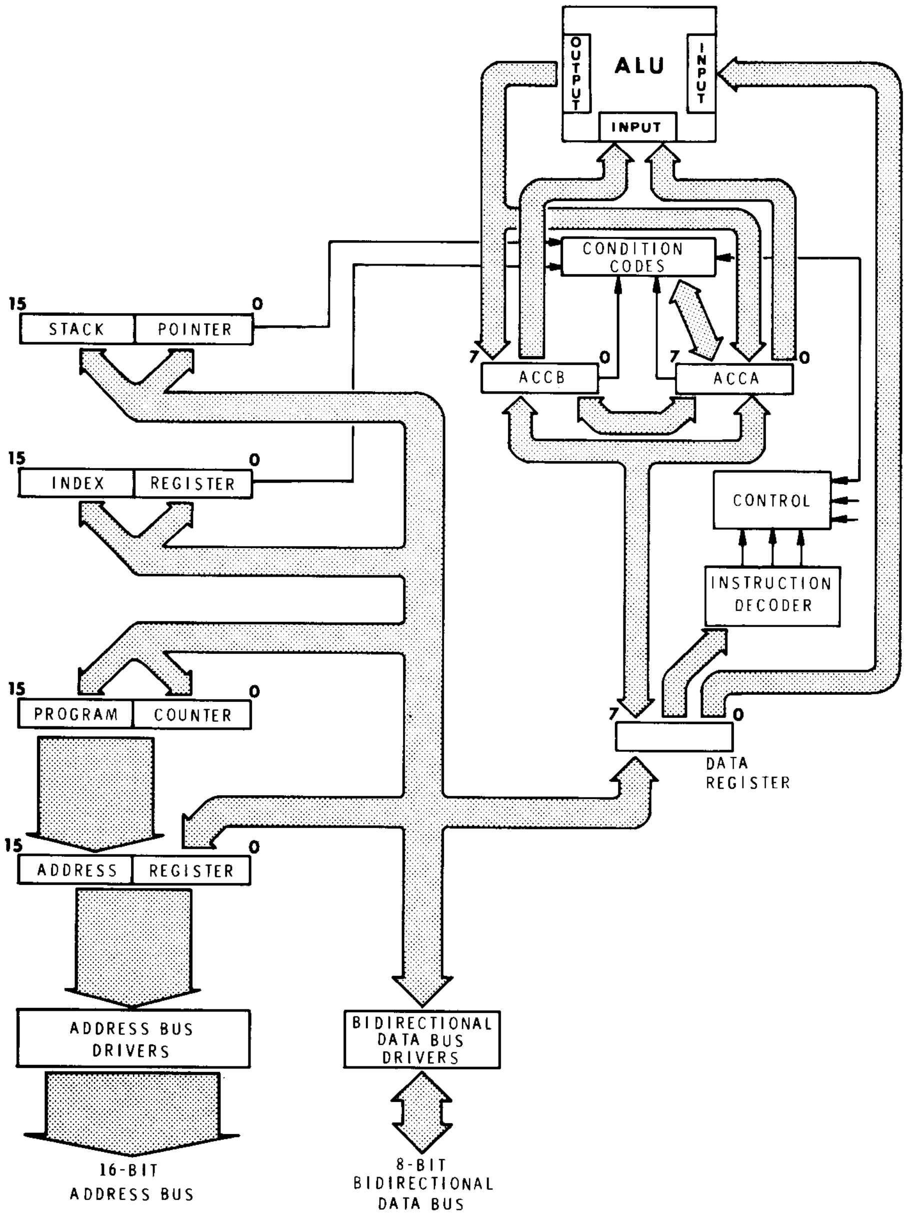


Figure 5-4
Simplified block diagram of the 6800 MPU.

The 16-bit registers are shown on the left. These registers are primarily concerned with addressing memory. Since the address bus has 16-bits, all registers associated with addressing must also have 16-bits. Any of the 16-bit registers can be loaded from the data bus. However, because the data bus has only 8-bits, two operations are required to load the 16-bit registers. The upper half of the affected register is always loaded first. Then, a second operation loads the lower half. Although this requires separate MPU cycles, the microprocessor takes care of these operations automatically. For example, a single instruction can load the 16-bit index register with two memory bytes.

The program counter and address register perform exactly the same functions in the 6800 MPU as they did in our hypothetical model. The fetch and execute phases for the immediate and direct addressing modes are virtually identical. The same is true of the relative addressing mode except that the 8-bit relative address is added to the 16-bit program count.

The 8-bit registers are shown on the right. Notice that these circuits are identical to those in our hypothetical model except that there are two accumulators. The condition code registers monitor both accumulators. Also, the two accumulators share the ALU. This allows you to keep track of two separate mathematical operations at more or less the same time. This arrangement is particularly flexible since the contents of one accumulator can be transferred to the other or the contents of the two accumulators can be added together.

Self-Test Review

1. The microprocessor on which our hypothetical model and the ET-3400 are based is the ______________ MPU.

2. A major difference between our hypothetical model and the 6800 MPU is that the latter has two ______________.

3. The program counter in the 6800 MPU has ______________ bits.

4. How wide is the address bus in a 6800-based microcomputer?

5. What is the range of addresses in the 6800 MPU?

6. List the six condition code flags.

7. Besides the program counter, what other 16-bit registers are used in the 6800 MPU?

8. In the 6800 MPU, does each accumulator have its own carry flag?

Answers

1. 6800.

2. Accumulators.

3. 16_{10}.

4. 16_{10} bits.

5. From 0000 to $65{,}535_{10}$ or 0000 to $FFFF_{16}$.

6. Carry — borrow (C)
   Overflow (V)
   Zero (Z)
   Negative (N)
   Interrupt Mask (I)
   Half Carry (H)

7. Index register and stack pointer.

8. No, the two accumulators share a common carry flag.

INSTRUCTION SET OF THE 6800 MPU

The 6800 MPU has about 100_{10} basic instructions. Moreover, when all the different addressing modes are considered, there are 197_{10} different opcodes to which the MPU will respond.

These instructions can be broken down into seven general categories. While some of the categories overlap, the general classifications of instructions are: arithmetic, data handling, logic, data test, index register and stack pointer, jump and branch, and condition code. In this unit we will discuss most of these instructions in detail. The handful of instructions that are not discussed in this unit will be described in the following unit.

In this section we will not be concerned with addressing modes. Therefore, no opcodes are given. Later, we will look at the various addressing modes and opcodes. For now, though, let's identify the instructions by their names, mnemonics, and operations. You will see what each instruction does and how it affects the various condition code registers.

Because of the large number of instructions covered in this section, the explanations will be general and brief. You are **not** expected to remember all the details of every instruction. Appendix A of this course contains a detailed listing of each instruction. It explains every detail of the various instructions. After reading this section, turn to Appendix A and look over the explanations given there. In the future, when you are in doubt as to exactly what a particular instruction does, look it up in Appendix A.

Arithmetic Instructions

Figure 5-5 shows the arithmetic instructions of the 6800 MPU. The name of each instruction is given on the left. The next column contains the mnemonics. The center column gives a shorthand description of what the instruction does. The right-hand columns show how the various condition code registers are affected.

ACCUMULATOR AND MEMORY		BOOLEAN/ARITHMETIC OPERATION	CONO. CODE REG.					
		(All register labels	5	4	3	2	1	0
OPERATIONS	MNEMONIC	refer to contents)	H	I	N	Z	V	C
Add	ADDA	$A + M \rightarrow A$	↕	●	↕	↕	↕	↕
	ADDB	$B + M \rightarrow B$	↕	●	↕	↕	↕	↕
Add Acmltrs	ABA	$A + B \rightarrow A$	↕	●	↕	↕	↕	↕
Add with Carry	ADCA	$A + M + C \rightarrow A$	↕	●	↕	↕	↕	↕
	ADCB	$B + M + C \rightarrow B$	↕	●	↕	↕	↕	↕
Complement, 2's (Negate)	NEG	$00 - M \rightarrow M$	●	●	↕	↕	①	②
	NEGA	$00 - A \rightarrow A$	●	●	↕	↕	①	②
	NEGB	$00 - B \rightarrow B$	●	●	↕	↕	①	②
Decimal Adjust, A	DAA	Converts Binary Add. of BCD Characters into BCD Format*	●	●	↕	↕	↕	③
Subtract	SUBA	$A - M \rightarrow A$	●	●	↕	↕	↕	↕
	SUBB	$B - M \rightarrow B$	●	●	↕	↕	↕	↕
Subract Acmltrs.	SBA	$A - B \rightarrow A$	●	●	↕	↕	↕	↕
Subtr. with Carry	SBCA	$A - M - C \rightarrow A$	●	●	↕	↕	↕	↕
	SBCB	$B - M - C \rightarrow B$	●	●	↕	↕	↕	↕

*Used after ABA, ADC, and ADD in BCD arithmetic operation; each 8-bit byte regarded as containing two 4-bit BCD numbers. DAA adds 0110 to lower half-byte if least significant number >1001 or if preceding instruction caused a Half-carry. Adds 0110 to upper half-byte if most significant number >1001 or if preceding instruction caused a Carry. Also adds 0110 to upper half-byte if least significant number >1001 and most significant number = 9.

(Bit set if test is true and cleared otherwise)

①	(Bit V)	Test: Result = 10000000?
②	(Bit C)	Test: Result $\neq$ 00000000?
③	(Bit C)	Test: Decimal value of most significant BCD Character greater than nine? (Not cleared if previously set.)

Figure 5-5
Arithmetic instructions.

To be certain you have the idea, let's go through the first instruction in detail. The first instruction is the add instruction. Actually, since the 6800 has two accumulators, there are two add instructions. Their mnemonics are ADDA and ADDB. Notice that the final letter of the mnemonic indicates which accumulator (A or B) is involved. The shorthand representation of the operation is: $A+M \rightarrow A$. The note at the top of this column tells you that the register labels refer to the contents of the register. Thus, A means the contents of accumulator A and M means the contents of the affected memory location. The symbol ($\rightarrow$) means "Transfer into." Therefore, $A+M \rightarrow A$ means "Add the contents of accumulator A to the contents of the affected memory location and transfer the sum into accumulator A."

To see how the condition code flags are affected, you simply look over to the right under whatever condition code you are interested in. Generally, the condition code is either unaffected or is tested and set accordingly. When the condition code is unaffected, this is represented by the symbol (•). For example, none of the arithmetic instructions affect the I flag. Most of the arithmetic instructions test the condition codes and set them if the condition exists. For example, if the result of an arithmetic operation is zero, the Z flag is set to 1. If this condition does not exist, the Z flag is reset or cleared to 0. The symbol ($\updownarrow$) means "test and set if true; clear otherwise." Occasionally, a note is necessary to describe some unusual situation regarding the condition code. This is represented by a number within a circle. The notes are given at the bottom of the drawing.

The ADDA and ADDB instructions are self-explanatory. The ABA instruction adds the contents of accumulator A to the contents of accumulator B. The result is stored in accumulator A.

The add with carry instructions are identical to those discussed earlier for our hypothetical machine. Notice that the carry bit is added in with the sum.

Because two's complement arithmetic is used in the 6800 MPU, instructions are provided that allow us to take the two's complement of a number. The negate instruction subtracts the contents of the affected register from 00_{16}. This is the same as taking the two's complement of the number. The affected register can be any memory location (M) or either accumulator (A or B). Thus, there are three different negate instructions. Keep in mind that NEG means "take the two's complement of the affected memory location;" NEGA means "take the two's complement of accumulator A;" etc.

Notice that the NEG instruction allows us to operate on a byte in memory without first fetching the operand from memory. In the past, we have loaded the operand, performed the operation, and then stored the new operand. However, the 6800 allows us to perform certain operations on the operand without first fetching it from memory. Several examples of this will be pointed out as we progress through the instruction set.

The decimal adjust instruction performs exactly as it did in our hypothetical machine. The note immediately under the table summarizes its operation. It must also be pointed out that this instruction works only with accumulator A.

The subtract and the subtract with carry instructions are self-explanatory. They perform as described earlier for our hypothetical MPU. The 6800 MPU has an additional subtract instruction. The SBA instruction subtracts the contents of accumulator B from the contents of accumulator A. The resulting difference is placed in accumulator A.

Data Handling Instructions

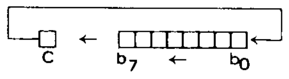
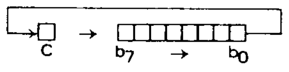
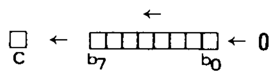
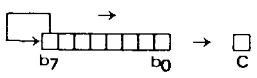
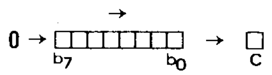
Figure 5-6 shows the largest group of instructions used by the 6800 MPU. These can be loosely categorized as data handling instructions.

The clear instructions allow us to clear a memory location or either accumulator. In the past, we have cleared bytes of memory by first clearing the accumulator and then storing the resulting 00_{16} in the proper memory location. However, the CLR instruction allows us to clear a memory location with a single instruction. Notice that some new entries appear in the condition code registers column. R means that the condition code is always reset or cleared to 0. S means that the code is always set to 1.

The decrement instruction allows us to subtract 1 from a memory location or from either accumulator. The DEC instruction is especially valuable since it allows us to decrement a byte in memory with a single instruction. Previously we have loaded the byte, decremented it, and then stored it back in memory.

The increment instructions are similar except they allow us to add 1 to a memory location or one of the accumulators. Notice that the INC instruction allows us to maintain a tally in memory without having to load it, increment it, and then store it away.

The load accumulator instructions are self-explanatory. Notice that either accumulator can be loaded from memory.

ACCUMULATOR AND MEMORY OPERATIONS	MNEMONIC	BOOLEAN/ARITHMETIC OPERATION (All register labels refer to contents)	COND. CODE REG. 5 H	4 I	3 N	2 Z	1 V	0 C
Clear	CLR	00 → M	●	●	R	S	R	R
	CLRA	00 → A	●	●	R	S	R	R
	CLRB	00 → B	●	●	R	S	R	R
Decrement	DEC	M – 1 → M	●	●	↕	↕	④	●
	DECA	A – 1 → A	●	●	↕	↕	④	●
	DECB	B – 1 → B	●	●	↕	↕	④	●
Increment	INC	M + 1 → M	●	●	↕	↕	⑤	●
	INCA	A + 1 → A	●	●	↕	↕	⑤	●
	INCB	B + 1 → B	●	●	↕	↕	⑤	●
Load Acmltr	LDAA	M → A	●	●	↕	↕	R	●
	LDAB	M → B	●	●	↕	↕	R	●
Rotate Left	ROL	M 	●	●	↕	↕	⑥	↕
	ROLA	A	●	●	↕	↕	⑥	↕
	ROLB	B	●	●	↕	↕	⑥	↕
Rotate Right	ROR	M 	●	●	↕	↕	⑥	↕
	RORA	A	●	●	↕	↕	⑥	↕
	RORB	B	●	●	↕	↕	⑥	↕
Shift Left, Arithmetic	ASL	M 	●	●	↕	↕	⑥	↕
	ASLA	A	●	●	↕	↕	⑥	↕
	ASLB	B	●	●	↕	↕	⑥	↕
Shift Right, Arithmetic	ASR	M 	●	●	↕	↕	⑥	↕
	ASRA	A	●	●	↕	↕	⑥	↕
	ASRB	B	●	●	↕	↕	⑥	↕
Shift Right, Logic	LSR	M 	●	●	R	↕	⑥	↕
	LSRA	A	●	●	R	↕	⑥	↕
	LSRB	B	●	●	R	↕	⑥	↕
Store Acmltr	STAA	A → M	●	●	↕	↕	R	●
	STAB	B → M	●	●	↕	↕	R	●
Transfer Acmltrs	TAB	A → B	●	●	↕	↕	R	●
	TBA	B → A	●	●	↕	↕	R	●

④ (Bit V) Test: Operand = 10000000 prior to execution?

⑤ (Bit V) Test: Operand = 01111111 prior to execution?

⑥ (Bit V) Test: Set equal to result of N ⊕ C after shift has occurred.

Figure 5-6
Data handling instructions.

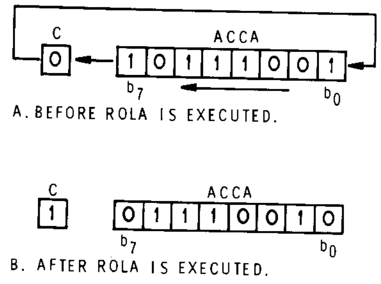


Figure 5-7
Executing the ROLA instruction.

The rotate left instructions allow us to shift the contents of the accumulator or a memory location without losing bits of data. Consider the ROLA instruction as an example. When this instruction is executed, the A accumulator and the carry bit form a 9-bit circulating register. That is, they form a closed loop as shown in Figure 5-7A. When ROLA is executed, the data is rotated clockwise. The MSB of A shifts into the carry register. Simultaneously, the contents of A are shifted left. Notice that the carry bit is not lost. Instead it is shifted into the LSB of the accumulator.

While the usefulness of this instruction may not be obvious, it is a valuable tool. For example, it could be used to determine parity. By repeatedly rotating left and testing the C flag, you could determine the number of 1's in the byte. Once you know this, you could easily generate the proper parity bit.

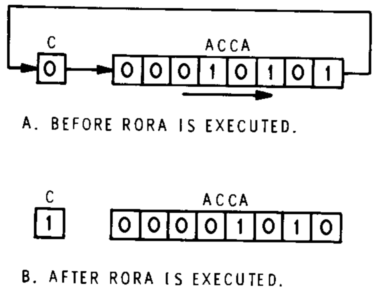


Figure 5-8
Executing the RORA instruction.

The ROL instruction allows you to rotate a memory byte to the left while it is still in memory. ROLB allows you to rotate the B accumulator to the left. In each case, the C register is used as a ninth bit.

The rotate right instructions are identical except that the direction of rotation is reversed. Figure 5-8 illustrates the execution of the RORA instruction. This instruction is also valuable. Suppose for example that we wish to know if the number in the accumulator is even or odd. This is determined by the LSB of the number. If LSB = 1, the number is odd; if LSB = 0, the number is even. One way to determine this is to rotate the number to the right so that the LSB is in the C register. We could then test the C register to see if it is set or cleared. Notice that the number could then be restored to its original value by the ROLA instruction.

The arithmetic shift left instruction was discussed earlier in our hypothetical MPU. The ASLA instruction performs exactly as described in the previous unit. However, notice that the 6800 MPU also has an ASLB instruction that performs the same operation with accumulator B. Also, it has an ASL instruction that allows us to perform this operation on a byte that is in memory. Figure 5-9 illustrates the execution of this instruction.

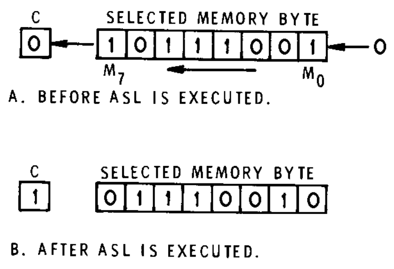


Figure 5-9
Executing the ASL instruction.

While there is only one type of shift left instruction, there are two types of shift right instructions. Let's discuss the arithmetic shift right instructions first.

When an **arithmetic** shift right instruction is executed, the number in the affected register is shifted right one position. The LSB goes into the C register. B_1 shifts to B_0, etc. B_7 shifts into B_6. However, B_7 itself remains unchanged. Figure 5-10 illustrates the execution of the ASRB instruction. Notice that there are also ASRA and ASR instructions listed in Figure 5-6. These perform the same type of shift operation but on accumulator A and the selected memory byte respectively.

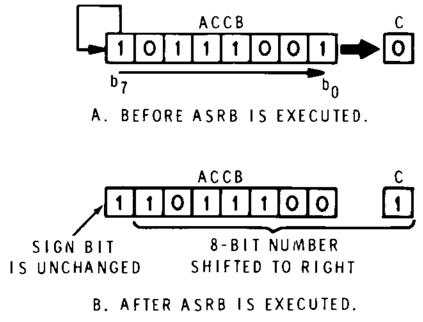

ACCB
C
1 0 1 1 1 0 0 1
0
b7
b0
A. BEFORE ASRB IS EXECUTED.
ACCB
C
1 1 0 1 1 1 0 0
1
SIGN BIT IS UNCHANGED
8-BIT NUMBER SHIFTED TO RIGHT
B. AFTER ASRB IS EXECUTED.

Figure 5-10
Executing the ASRB instruction.

The logic shift right instructions are different in that they do not retain the sign bit. When a logic shift right is executed, the contents of the affected register are shifted to the right. The LSB goes into the carry register. The MSB is filled with a 0. For example, the LSR instruction is illustrated in Figure 5-11. While this instruction shifts the selected memory locations, LSRA and LSRB can be used to perform similar operations on accumulators A and B respectively.

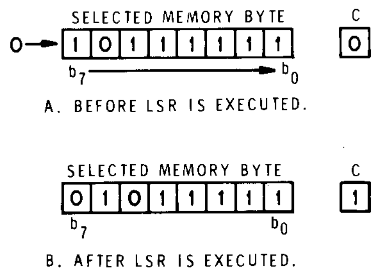


Figure 5-11
Executing the LSR instruction.

Referring back to Figure 5-6, the store accumulator instructions are self-explanatory.

The final data handling instructions are the transfer accumulator instructions. TAB copies the contents of accumulator A into accumulator B. After this instruction is executed, the number originally in accumulator A will be in both accumulators. TBA does just the opposite. It copies the contents of accumulator B into accumulator A. After TBA is executed, the number originally in accumulator B will be in both accumulators.

Logic Instructions

The logic instructions in the 6800 MPU are similar to those in our hypothetical MPU. Figure 5-12 shows the 6800's logic instructions.

ACCUMULATOR AND MEMORY		BOOLEAN/ARITHMETIC OPERATION (All register labels refer to contents)	COND. CODE REG.					
OPERATIONS	MNEMONIC		5	4	3	2	1	0
			H	I	N	Z	V	C
And	ANDA	$A \bullet M \rightarrow A$	●	●	↕	↕	R	●
	ANDB	$B \bullet M \rightarrow B$	●	●	↕	↕	R	●
Complement, 1's	COM	$\bar{M} \rightarrow M$	●	●	↕	↕	R	S
	COMA	$\bar{A} \rightarrow A$	●	●	↕	↕	R	S
	COMB	$\bar{B} \rightarrow B$	●	●	↕	↕	R	S
Exclusive OR	EORA	$A \oplus M \rightarrow A$	●	●	↕	↕	R	●
	EORB	$B \oplus M \rightarrow B$	●	●	↕	↕	R	●
Or, Inclusive	ORA	$A + M \rightarrow A$	●	●	↕	↕	R	●
	ORB	$B + M \rightarrow B$	●	●	↕	↕	R	●

Figure 5-12
Logic instructions.

There is one AND instruction for each accumulator. The contents of the specified accumulator are ANDed bit-for-bit with the contents of the selected memory location. The result is placed back in the accumulator. This is identical to the AND instruction in our hypothetical machine.

The complement instructions allow you to take the 1's complement of the number in the affected register. COM allows you to complement a byte in memory.

COMA and COMB allow you to complement the contents of accumulators A and B respectively. In each case, all 1's are changed to 0's and all 0's are changed to 1's.

The exclusive OR instructions work like the one in our hypothetical MPU. The contents of the specified accumulator are exclusively ORed bit-for-bit with the contents of the selected memory location. The result is stored back in the specified accumulator.

The inclusive OR is similar except that the contents of the specified accumulator are inclusively ORed with the contents of the selected memory location.

Data Test Instructions

These are a powerful group of instructions that allow us to compare operands in several different ways. In previous units, you had experience comparing operands. The most frequently used method was to subtract one operand from another and test the result for zero or negative. In many cases, the numeric result of the subtraction was unimportant. We needed to know only if the result was zero or minus. The data test instructions allow us to make several different tests without actually producing an unwanted numeric result. These instructions are shown in Figure 5-13.

ACCUMULATOR AND MEMORY		BOOLEAN/ARITHMETIC OPERATION (All register labels refer to contents)	COND. CODE REG.					
			5	4	3	2	1	0
OPERATIONS	MNEMONIC		H	I	N	Z	V	C
Bit Test	BITA	A • M	●	●	↕	↕	R	●
	BITB	B • M	●	●	↕	↕	R	●
Compare	CMPA	A − M	●	●	↕	↕	↕	↕
	CMPB	B − M	●	●	↕	↕	↕	↕
Compare Acmltrs	CBA	A − B	●	●	↕	↕	↕	↕
Test, Zero or Minus	TST	M − 00	●	●	↕	↕	R	R
	TSTA	A − 00	●	●	↕	↕	R	R
	TSTB	B − 00	●	●	↕	↕	R	R

Figure 5-13
Data test instructions.

The bit test instructions are very similar to the AND instructions. In both cases, the contents of the specified accumulator are ANDed with the contents of the selected memory location. The difference is that with the bit test instruction no logical product is produced. Neither the contents of the accumulator nor memory are altered in any way. However, the condition code registers are affected just as if the AND operation had taken place. Consider the BITA instruction. When executed, A is ANDed with M. If the result is 00_{16}, the Z register is set. Otherwise, the Z register is cleared. If the MSB of the result is 1, the N flag is set. However, the contents of the accumulator and memory are unaffected.

In the same way, the compare instructions are similar to subtract instructions except that the resulting numeric difference is ignored. For example, when the CMPA instruction is executed, the contents of the selected memory location are subtracted from the contents of accumulator A. The condition codes are affected just as if a difference had been produced. However, the original contents of accumulator A and memory are unaffected.

The compare accumulators instruction (CBA) works the same way. The condition codes are set as if the contents of B were subtracted from the contents A. However, the contents of the accumulators are unaffected.

Finally, the test for zero or minus instruction allows you to test the number in one of the accumulators or the memory to see if it is negative or zero. When this instruction is executed, the MPU looks at the number in question and sets the N and Z flags accordingly. The number itself is not changed.

Index Register and Stack Pointer Instructions

The index register and stack pointer are 16-bit registers. Figure 5-14 shows eleven instructions that allow us to control the operation of these registers. Because of the 16-bit format, the load, store, and compare instructions are slightly different from those discussed earlier.

INDEX REGISTER AND STACK POINTER OPERATIONS	MNEMONIC	BOOLEAN/ARITHMETIC OPERATION	5	4	3	2	1	0
			H	I	N	Z	V	C
Compare Index Reg	CPX	$(X_H/X_L) - (M/M + 1)$	●	●	①	↕	②	●
Decrement Index Reg	DEX	$X - 1 \rightarrow X$	●	●	●	↕	●	●
Decrement Stack Pntr	DES	$SP - 1 \rightarrow SP$	●	●	●	●	●	●
Increment Index Reg	INX	$X + 1 \rightarrow X$	●	●	●	↕	●	●
Increment Stack Pntr	INS	$SP + 1 \rightarrow SP$	●	●	●	●	●	●
Load Index Reg	LDX	$M \rightarrow X_H, (M + 1) \rightarrow X_L$	●	●	③	↕	R	●
Load Stack Pntr	LDS	$M \rightarrow SP_H, (M + 1) \rightarrow SP_L$	●	●	③	↕	R	●
Store Index Reg	STX	$X_H \rightarrow M, X_L \rightarrow (M + 1)$	●	●	③	↕	R	●
Store Stack Pntr	STS	$SP_H \rightarrow M, SP_L \rightarrow (M + 1)$	●	●	③	↕	R	●
Indx Reg → Stack Pntr	TXS	$X - 1 \rightarrow SP$	●	●	●	●	●	●
Stack Pntr → Indx Reg	TSX	$SP + 1 \rightarrow X$	●	●	●	●	●	●

① (Bit N) Test: Sign bit of most significant (MS) byte of result = 1?
② (Bit V) Test: 2's complement overflow from subtraction of LS bytes?
③ (Bit N) Test: Result less than zero? (Bit 15 = 1)

Figure 5-14
Index register and stack pointer instructions.

The compare index register (CPX) instruction allows us to compare the 16-bit number in the index register with any two consecutive bytes in memory. Recall that the index register (X) will hold two bytes. The higher byte is identified as X_H while the lower byte is called X_L. When the CPX instruction is executed, X_H is compared with the 8-bit byte in the specified memory location (M). Also, X_L is compared with the byte immediately following the specified memory location (M+1). The comparison is the same as if M and M+1 were subtracted from X_H and X_L except that no numeric difference is produced. Neither X nor M is changed in any way. However, the N, Z, and V condition codes are affected as shown in Figure 5-14. Generally, the Z code is the one we are interested in since it tells us whether or not an exact match exists between the index register and the two bytes in memory.

The next four instructions are self-explanatory. They allow us to increment and decrement either the index register or the stack pointer. For one thing, these instructions allow us to maintain two separate 16-bit tallies simultaneously. However, the real value of these instructions and their associated registers will be discussed later.

The load and store instructions for the 16-bit registers are shown next in Figure 5-14. Since these are two byte registers, the LDX and LDS instructions must load two bytes from memory. In the case of the index register, the specified memory byte (M) is loaded into the upper half of the index register (X_H). An instant later, the next byte in memory (M+1) is automatically loaded into the lower half of the index register (X_L). Thus, the operation can be described as: $M \rightarrow X_H$, $(M+1) \rightarrow X_L$.

Because the stack pointer is also a 16-bit register, the load stack pointer instruction (LDS) works the same way. Its operation can be described as: $M \rightarrow SP_H$, $(M+1) \rightarrow SP_L$. Here, SP_H refers to the upper half of the stack pointer while SP_L refers to the lower half.

When the contents of the 16-bit registers are being stored, the operation is reversed. For example, the STX instruction stores X_H in M and X_L in M+1. A similar instruction, STS, allows us to store the contents of the stack pointer in the same way.

The final two instructions in this group allow us to transfer numbers between these two 16-bit registers. The TXS instruction loads the stack pointer with the contents of the index register **minus one**. The TSX instruction loads the index register with the contents of the stack pointer **plus one**. A more detailed discussion of these two important registers and their associated instructions will be given in the next unit.

Branch Instructions

The branch instructions are shown in Figure 5-15. Two additional instructions are also thrown in since they affect the program counter.

BRANCH OPERATIONS	MNEMONIC	BRANCH TEST	5	4	3	2	1	0
			H	I	N	Z	V	C
Branch Always	BRA	None	●	●	●	●	●	●
Branch If Carry Clear	BCC	C = 0	●	●	●	●	●	●
Branch If Carry Set	BCS	C = 1	●	●	●	●	●	●
Branch If = Zero	BEQ	Z = 1	●	●	●	●	●	●
Branch If ⩾ Zero	BGE	N ⊕ V = 0	●	●	●	●	●	●
Branch If > Zero	BGT	Z + (N ⊕ V) = 0	●	●	●	●	●	●
Branch If Higher	BHI	C + Z = 0	●	●	●	●	●	●
Branch If ⩽ Zero	BLE	Z + (N ⊕ V) = 1	●	●	●	●	●	●
Branch If Lower Or Same	BLS	C + Z = 1	●	●	●	●	●	●
Branch If < Zero	BLT	N ⊕ V = 1	●	●	●	●	●	●
Branch If Minus	BMI	N = 1	●	●	●	●	●	●
Branch If Not Equal Zero	BNE	Z = 0	●	●	●	●	●	●
Branch If Overflow Clear	BVC	V = 0	●	●	●	●	●	●
Branch If Overflow Set	BVS	V = 1	●	●	●	●	●	●
Branch If Plus	BPL	N = 0	●	●	●	●	●	●
No Operation	NOP	Advances Prog. Cntr. Only	●	●	●	●	●	●
Wait for Interrupt	WAI		●	①	●	●	●	●

① (Bit I) Set when interrupt occurs. If previously set, a Non-Maskable Interrupt is required to exit the wait state.

Figure 5-15
Jump and branch instructions.

Nine of these instructions were discussed in the previous unit. These are: Branch Always (BRA); Branch If Carry Clear (BCC); Branch If Carry Set (BCS); Branch If Equal Zero (BEQ); Branch If Not Equal Zero (BNE); Branch If Minus (BMI); Branch If Plus (BPL); Branch If Overflow Clear (BVC); and Branch If Overflow Set (BVS).

Before we discuss the new branch instructions, here are some of the symbols we will be using. The symbol ($\geq$) means "is greater than or is equal to"; ($>$) means "is greater than"; ($\leq$) means "is less than or is equal to"; ($<$) means "is less than"; and ($\neq$) means "is not equal to."

Now consider the Branch If Greater Than or Equal instruction (BGE). This instruction is normally used after a subtract or compare instruction. It will cause a branch operation if the two's complement value in the accumulator is greater than or equal to the two's complement operand in memory. This condition is indicated by the N and V flags having the same value. The MPU determines if this condition is met by exclusively ORing N and V and examining the result.

Three simple examples may help illustrate the operation of this instruction. Let's start with a number in the accumlator that is greater than the operand in memory:

Number in Accumulator	=	00000010_2
Operand in Memory	=	00000001_2

When the operand is subtracted, the result is 00000001_2. With this result, both N and V are cleared to 0. Notice that N and V are equal and $N \oplus V = 0$. If the BGE instruction followed the subtract operation, the branch would be implemented.

Now see what happens when the number in the accumulator is equal to the operand:

Number in Accumulator	=	00000010_2
Operand in Memory	=	00000010_2

When the operand is subtracted, the result is 00000000_2. Again N and V are cleared to 0. Thus, N and V are still equal and $N \oplus V = 0$. Again, the BGE instruction would cause a branch to occur.

Finally, note what happens when the number in the accumulator is smaller:

Number in Accumulator	=	00000001_2
Operand in Memory	=	00000010_2

When the operand is subtracted, the result is 11111111_2. This time N is set but V is cleared. Thus, N and V are not equal. Therefore, $N \oplus V = 1$. In this case, the BGE conditions are not met and no branch will occur. The branch occurs if the two's complement value in the accumulator is greater than or equal to the two's complement operand in memory.

Next, consider the Branch If Greater Than (BGT) instruction. This instruction is normally used immediately after a subtract or compare operation. The branch will occur only if the two's complement minuend was greater than the two's complement subtrahend. By trying several examples as we did above, you will find that the branch conditions are met when $Z = 0$ and $N = V$.

The Branch If Higher (BHI) instruction is similar to the BGT instruction except that it is concerned with **unsigned** numbers. BHI is normally used after a subtract or compare operation. The branch will occur only if the unsigned minuend was greater than the unsigned subtrahend. By trying several different examples, you can prove that this occurs only when the C and Z flags are both 0.

The Branch If Less Than or Equal (BLE) instruction allows you to compare two's complement numbers in another way. If it is executed immediately after a subtract or compare operation, the branch will occur only if the two's complement minuend was less than or equal to the two's complement subtrahend.

The Branch If Lower Or Same (BLS) instruction is similar to the BLE instruction except that **unsigned** numbers are compared. When it is executed immediately after a subtract or compare operation, the branch will occur only if the unsigned minuend was lower than or equal to the unsigned subtrahend.

The Branch If Less Than Zero (BLT) instruction is also similar to the BLE instruction except that the equal qualification is removed. If BLT is executed immediately after a subtract or compare operation, the branch occurs only if the two's complement minuend was less than the two's complement subtrahend.

Two additional instructions are included in Figure 5-15. Although they are not branch instructions, they are included here since they do not seem to fit any of the other categories.

The No Operation (NOP) instruction is a "do-nothing" instruction that simply consumes a small increment of time. It does not change the contents of any register except the program counter. It does increment the program counter by one and consumes two MPU cycles. In spite of this, the NOP is a very useful instruction. When writing a program, we frequently use too many instructions. Once the program is loaded in memory, it is often inconvenient to simply remove an instruction. The hole left in memory can be filled by moving back all instructions that follow. However, a faster way is to simply fill the hole with one or more NOP instructions.

The Wait For Interrupt (WAI) instruction is the 6800's version of a HLT instruction. In earlier units we used this instruction at the end of all our programs. We will continue to use it in the same manner in the future. However, as you will see in the next unit, there is more involved in executing the WAI instruction than simply stopping the MPU. For now, though, continue to think of the WAI as a simple halt instruction.

Condition Code Register Instructions

The 6800 MPU has eight instructions that allow us direct access to the condition codes. These are listed in Figure 5-16.

CONDITION CODE REGISTER OPERATIONS	MNEMONIC	BOOLEAN OPERATION	5 H	4 I	3 N	2 Z	1 V	0 C
Clear Carry	CLC	0 → C	●	●	●	●	●	R
Clear Interrupt Mask	CLI	0 → I	●	R	●	●	●	●
Clear Overflow	CLV	0 → V	●	●	●	●	R	●
Set Carry	SEC	1 → C	●	●	●	●	●	S
Set Interrupt Mask	SEI	1 → I	●	S	●	●	●	●
Set Overflow	SEV	1 → V	●	●	●	●	S	●
Acmltr A → CCR	TAP	A → CCR	——— ① ———					
CCR → Acmltr A	TPA	CCR → A	●	●	●	●	●	●

R = Reset
S = Set
● = Not affected
① (ALL) Set according to the contents of Accumulator A.

Figure 5-16
Condition code register instructions.

Most of these instructions are self-explanatory. The Clear Carry (CLC) instruction resets the C flag to 0 while the Set Carry (SEC) sets it to 1. In the same way, the CLV and SEV instructions allow us to clear and set the overflow flag. Also, the CLI and SEI instructions can be used to clear or set the interrupt flag.

You will notice that there are no instructions for individually clearing the N, Z, or H flags. However, we can still set or clear these flags with the Transfer Accumulator A to the Processor Status Register (TAP) instruction. Figure 5-17 illustrates the execution of this instruction. The contents of bits 0 through 5 of accumulator A are transferred to the condition code registers. Thus, this instruction allows us to set or clear all the condition codes at once.

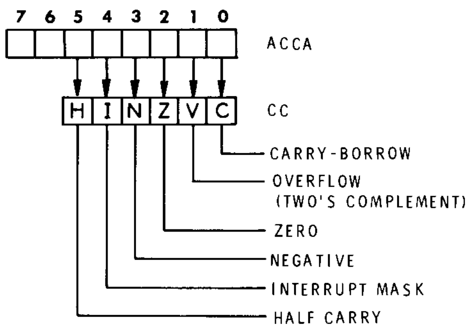


Figure 5-17
Executing the TAP instruction.

The final instruction is the Transfer Processor Status to Accumulator A (TPA) instruction. When this instruction is executed, the contents of the condition code registers are transferred to bits 0 through 5 of accumulator A. This operation is illustrated in Figure 5-18. Notice that bits 6 and 7 of the accumulator are set to 1.

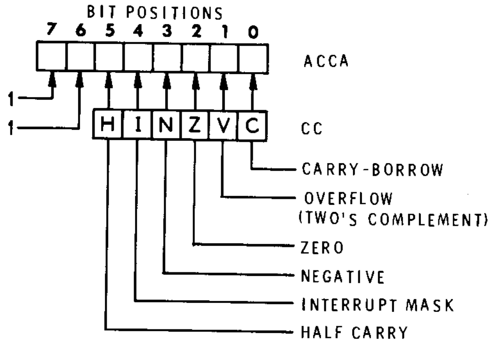


Figure 5-18
Executing the TPA instruction.

Summary of Instruction Set

As you can see, the 6800 MPU has a wide variety of instructions. In this section, most of the instructions have been mentioned briefly. However, a full explanation of some instructions must wait until additional new concepts have been covered.

In one short section, it is very difficult to cover every instruction in detail. And, it is virtually impossible for the reader to remember all the details of each instruction. Remember, all of the instructions available to the 6800 MPU are explained in detail in Appendix A of this program. Also, they are arranged alphabetically by their mnemonics for easy reference. Refer to Appendix A any time you are in doubt about what an instruction does. Be sure to look over the introductory material in the Appendix so that you understand all the conventions and symbols.

Self-Test Review

9. List the seven general categories of instructions.

10. What is meant by the shorthand notation: $A+B \rightarrow A$?

11. How is the C flag affected by the "add" and "add with carry" instructions?

12. Is the C flag changed when the AND instruction is executed?

13. Explain the difference between the NEG instruction and the COM instruction.

14. Explain the difference between the ANDA instruction and the BITA instruction.

15. The decimal adjust instruction is associated with which accumulator?

16. When the RORA instruction is executed the LSB of accumulator A is shifted into the ________________ register.

17. List eleven operations that can be performed directly to an operand in memory without first loading it into one of the MPU registers.

18. Explain the difference between the SUBB instruction and the CMPB instruction.

19. List the four types of logic operations that the 6800 MPU can perform.

20. When the LDX instruction is executed, from where is the index register loaded?

21. List four conditional branch instructions that are commonly used after a compare or subtract instruction to compare two's complement numbers.

22. Explain the difference between the BGT and BHI instructions.

23. Which instruction is often used to fill in a hole left in a program after an unwanted byte is removed?

24. Which instruction in the 6800 roughly corresponds to the halt instruction in our hypothetical machine?

25. Which of the condition codes can be individually set or cleared?

26. When you have some doubt as to exactly what operation is performed by a given instruction, where can you look to find the answer?

Answers

9. Arithmetic, data handling, logic, data test, index register and stack pointer, jump and branch, condition code.

10. Add the contents of accumulator A to the contents of accumulator B; transfer the result to accumulator A.

11. The C flag is set if a carry occurs; it is cleared otherwise.

12. No, the C flag is unaffected by the AND instruction.

13. The COM instruction replaces the operand with its 1's complement. The NEG instruction replaces the operand with its 2's complement.

14. With the ANDA instruction, the result of the AND operation is placed in accumulator A. With the BITA instruction, the condition code registers are set according to the result but the result is not retained.

15. The decimal adjust instruction works only with the A accumulator.

16. Carry (C).

17. A byte in memory can be: cleared, incremented, decremented, complemented, negated, rotated left, rotated right, shifted left arithmetically, shifted right arithmetically, shifted right logically, and tested.

18. With the SUBB instruction, a difference is produced and placed in accumulator B. With CMPB, the flags are set as if a difference were produced, but the difference is not retained.

19. Complement, AND, inclusive OR, and exclusive OR.

20. The upper half of the index register is loaded from the specified memory location; the lower half from the byte following the specified memory location.

21. BGE, BGT, BLE, BLT.

22. BGT is used to test the result of subtracting two's complement numbers. BHI is used to test the result of subtracting unsigned numbers.

23. NOP.

24. WAI.

25. C, I, and V.

26. Appendix A of this course.

NEW ADDRESSING MODES

In previous units, we have discussed four addressing modes. Let's briefly review these.

In the immediate addressing mode, the operand is the memory byte immediately following the opcode. These are generally two byte instructions. The first byte is the opcode, the second is the operand. However, there are exceptions to the two-byte rule. Some operations involve the 16-bit index register and stack pointer. In these cases, the operand is the **two** bytes immediately following the opcode. These are three byte instructions. The first byte is the opcode, the second and third are the operand.

In the direct addressing mode, the byte following the opcode is the address of the operand. These are always two byte instructions. The first byte is the opcode; the second is the address of the operand. An eight-bit byte can specify addresses from 00 to FF_{16}. Thus, when the direct addressing mode is being used, the operand must be in the first 256_{10} bytes of memory. Since the 6800 MPU can have up to $65{,}536_{10}$ bytes of memory, another means must be used to address the upper portion of memory.

The relative addressing mode is used for branching. These are two byte instructions. The first byte is the opcode, the second is the relative address. Recall that the relative address is added to the program count to form the absolute address. Since the 8-bit relative address is a two's complement number, the branch limits are $+127_{10}$ and -128_{10}.

In the inherent addressing mode either there is no operand or the operand is implied by the instruction. These are one byte instructions.

In this section, we will discuss two new addressing modes. These are called **extended addressing** and **indexed addressing**. We will discuss extended addressing first.

Extended Addressing

Extended addressing is similar to direct addressing but with one significant difference. Recall that with direct addressing the operand must be in the first 256_{10} bytes of memory. Since this represents less than one percent of the addresses available to the 6800 MPU, a more powerful addressing mode is needed. The extended addressing mode fills this need.

The format of an instruction that uses extended addressing is shown in Figure 5-19. The instruction will always have three bytes. The first byte is the opcode. The second and third bytes form a 16-bit address. Notice that the most significant part of the address is the byte immediately following the opcode. Since this instruction has a 16-bit address, the operand can be at any one of the $65,536_{10}$ possible addresses.

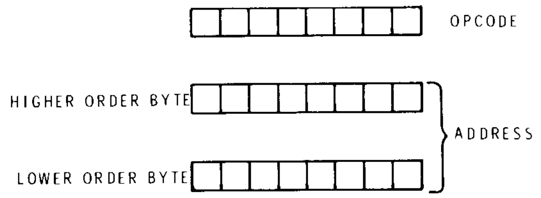


Figure 5-19
Format of an instruction that uses the extended addressing mode.

Suppose, for example, that you wish to load the operand at memory location 2134_{16} into accumulator B. The instruction would look like this:

F6	Opcode for LDAB extended
21	Higher order address
34	Lower order address

By the same token, if you wish to increment the number in memory location $AA00_{16}$, the instruction would be:

7C	Opcode for INC extended
AA	Higher order address
00	Lower order address

The extended addressing mode allows us to address an operand at any address including the first 256_{10} bytes of memory. Thus, if you wish to load the operand at address 0013_{16} into accumulator A, you can use extended addressing:

B6	Opcode for LDAA extended
00	Higher order address
13	Lower order address

Or, you can use direct addressing:

96	Opcode for LDAA direct
13	Address

Notice that, with direct addressing, the higher order address can be ignored since it is always 00. Because it saves one memory byte and one MPU cycle, direct addressing is normally used when the operand is in the first 256_{10} bytes of memory. Extended addressing is used when the operand is above address $00FF_{16}$. However, as you will see later, some instructions do not have a direct addressing mode. In these cases, extended addressing must be used even if the operand is in the first 256_{10} memory locations.

Indexed Addressing

The most powerful mode available to the 6800 MPU is indexed addressing. Recall that the 6800 MPU has a 16-bit index register. There are several instructions associated with this register. They allow us to load the register from memory and to store its contents in memory. Also, we can increment and decrement the index register. We can even compare its contents with two consecutive bytes in memory. These capabilities alone make the index register a very handy 16-bit counter. However, the real power of the index register comes from the fact that we can use this counter as an address pointer. Since this is a 16-bit register, it can point to any address in memory.

Purpose Before going into the details of how indexed addressing works, let's see why it is needed. Let's assume that we wish to add a list of 20_{16} numbers, and that the numbers are in 20_{16} consecutive memory locations starting at address 0050. Using the addressing modes discussed earlier, our program might look like this:

```
 CLRA    Clear Accumulator A.
 ADDA    Add the first number
  50     To accumulator A.
 ADDA    Add the second
  51     number to accumulator A.
 ADDA    Add the third number
  52     to accumulator A.
   •           •
   •           •
   •           •
 ADDA    Add the last number
  6F     to accumulator A.
 WAI     Wait.
```

While this accomplishes the desired result, it requires a long repetitive program. The above program would require 66_{10} bytes of memory. Notice that all the ADDA instructions are identical except that each successive address is one larger than the previous address. Indexed addressing can greatly simplify programs of this type.

Instruction Format The format of an instruction that uses indexed addressing is shown in Figure 5-20. Notice that this is a two-byte instruction. The first byte is the opcode, and the second is called an offset address. The offset address is an **unsigned** 8-bit binary number. It is added to the contents of the index register to determine the address at which the operand is located.

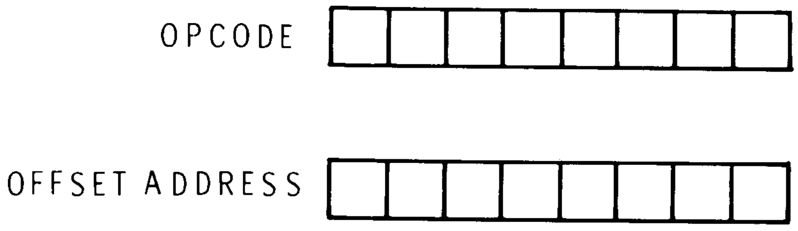


Figure 5-20
Format of an instruction that uses the indexed addressing mode.

Every instruction that involves an operand in memory can use the indexed addressing mode. In this unit, we will use the following convention to indicate indexed addressing:

```
LDAA, X
STAA, X
ADDB, X
   etc.
```

In each case, the X tells us that indexed addressing is used. For example, the first instruction means: "using indexed addressing, load the contents of the specified memory location into accumulator A." Now let's see how the address of the operand is determined.

Determining the Operand Address When indexed addressing is being used, the address of the operand is determined by the offset address and the number in the index register. Specifically, the 8-bit offset address is added to the 16-bit address in the index register. The 16-bit sum becomes the address of the operand. Figure 5-21 illustrates this.

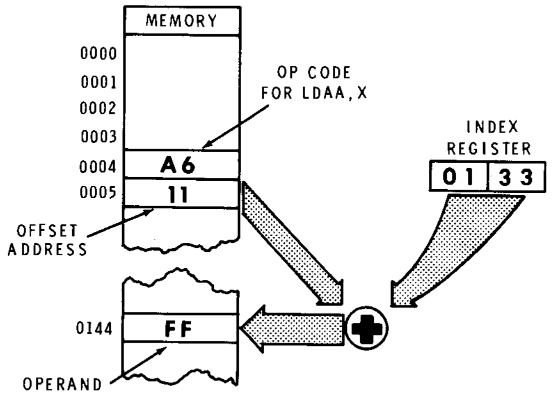


Figure 5-21
The operand address is formed by adding the offset address to the contents of the index register.

Here, the instruction in memory location 0004_{16} is LDAA, X. The offset address is 11_{16}. The contents of the index register are 0133_{16}. When the LDAA, X instruction is executed, the address of the operand is formed by adding the offset address to the number in the index register. In this case, the operand address will be:

$$\begin{array}{r} 0133_{16} \\ +\quad 11_{16} \\ \hline 0144_{16} \end{array}$$

The operand at this address is loaded into accumulator A. In this example, the operand FF is loaded into accumulator A when the instruction at location 0004 is executed. It is important to remember that this does not change the contents of the index register in any way. That is, the index register will still contain 0133_{16} after the instruction is executed.

Adding a List of Numbers To see how this addressing mode saves instructions, consider the problem given earlier. Recall that we were to add 20_{16} numbers stored in consecutive memory locations starting at address 0050. Using indexed addressing for the add instruction, our program looks like the one shown in Figure 5-22.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ HEX CONTENTS	COMMENTS
0010	CE	LDX #	Load index register immediate with the
0011	00	00	address of the
0012	50	50	first number in list.
0013	4F	CLRA	Clear accumulator A
0014	AB	ADDA, X	Add to accumulator A using indexed addressing
0015	00	00	with an offset address of 00.
0016	08	INX	Increment index register.
0017	8C	CPX #	Compare the contents of the index register
0018	00	00	with an address that is one greater than the
0019	70	70	address of the last number in the list.
001A	26	BNE	If not equal, branch back
001B	F8	F8	to the ADDA, X instruction.
001C	3E	WAI	Otherwise, halt.

Figure 5-22
Program for adding a list of 20_{16} numbers.

The first instruction is load index register immediate. Notice that a new symbol is used in this program. The symbol # is used to indicate the immediate addressing mode. Thus, the LDX# instruction causes the operand immediately following the opcode to be loaded into the index register. Recall that the index register can hold two 8-bit bytes. The operand is the two-byte number 0050_{16}. You may recognize that this is the address of the first number in the list of numbers that is to be added.

The next instruction clears accumulator A. The sum will be accumulated in this register, so it is important that it be cleared initially.

The third instruction (ADDA, X) is the only instruction in the program that uses indexed addressing. Notice that the symbol X indicates the indexed addressing mode. The offset address is 00. Recall that the operand address is determined by adding the offset to the contents of the index register. The index register contains 0050_{16} from a previous instruction. Since the offset is 00, the operand address is 0050_{16}. That is, the contents of memory location 0050 are added to the contents of accumulator A. Recall that 0050_{16} is the address of the first number in the list.

The fourth instruction increments the index register to 0051_{16}. Notice that the index register now points to the address of the second number in the list.

The fifth instruction compares the number in the index register with a number that is one greater than the address of the last number in the list.

If a match occurs, the Z flag will be set. Of course in this case, no match occurs yet. Notice once again that the symbol # indicates the immediate addressing mode. Thus, the contents of the index register are compared with the next two bytes in the program or 0070.

The BNE instruction tests the Z flag to see if the two numbers matched. If no match is indicated, the relative address (F8) directs the program back to the ADDA, X instruction. The first pass through the loop ends with the first number in accumulator A.

The second pass through the loop begins with the ADDA, X instruction being executed again. This time the index register points to address 0051. Therefore, the second number in the list is added to accumulator A. Accumulator A now contains the sum of the first two numbers. The index register is then incremented to 0052. Its contents are again compared with 0070. No match exists so the BNE instruction causes the loop to be repeated again.

The loop is repeated over and over again. Each time, the next number in the list is added to accumulator A. This process continues until the last number in the list is added. At that time, the index register will be incremented to 0070. Thus, when the CPX# instruction is executed, the Z flag will be set because the two numbers match. The BNE instruction recognizes that a match has occurred. Consequently, it does not allow the branch to occur and the next instruction in sequence is executed. Because this is the WAI instruction, the program halts. At this time, the sum of the 20_{16} numbers in the list will be in accumulator A.

Adding a list of numbers is a classic example of how indexing can be used to shorten a program. However, this example does not illustrate the full power of indexed addressing. For example, it does not illustrate the advantage of the offset address. Because indexed addressing is so important, let's look at another example.

Copying a List Let's assume we have a list of 10_{16} numbers that we wish to copy from one location to another. For simplicity, assume that the list is presently in addresses 0030 through 003F and that we wish to copy the list in location 0040 through 004F. Without using indexed addressing, our program might look like this:

```
LDAA
 30
STAA
 40
LDAA
 31
STAA
 41
  •
  •
  •
LDAA
 3F
STAA
 4F
 WAI
```

As you have seen; long, repetitive programs such as this are excellent candidates for indexed addressing.

Using indexed addressing, our program might look like that shown in Figure 5-23. The first step is to load the index register with the first address in the original list. The LDAA, X instruction has an offset address of 00. Therefore, accumulator A is loaded from the address specified by the index register (0030). That is, the first number in the original list is loaded into accumulator A when the LDAA, X instruction is executed.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ HEX CONTENTS	COMMENTS
0010	CE	LDX #	Load index register immediate with
0011	00	00	the first address of the original
0012	30	30	list.
0013	A6	LDAA, X	Load accumulator A indexed with
0014	00	00	an offset of 00.
0015	A7	STAA, X	Store accumulator A indexed with
0016	10	10	an offset of 10_{16}.
0017	08	INX	Increment index register.
0018	8C	CPX #	Compare index with one greater
0019	00	00	than last
001A	40	40	address in original list.
001B	26	BNE	If not equal, branch back to the
001C	F6	F6	LDAA, X instruction.
001D	3E	WAI	Otherwise, halt.

Figure 5-23
Program for copying a list from addresses 0030 — 003F into addresses 0040 — 004F.

The STAA, X instruction illustrates the use of the offset address. Notice that the offset is 10. This number is added to the address in the index register to form the effective address at which the contents of accumulator A are stored. Thus, the contents of accumulator A are stored at address 0040. Remember, this does not change the number in the index register in any way. By using the offset, we can load the accumulator indexed from one address and store the accumulator indexed at another.

Next, the index register is incremented to 0031. It is then compared with 0040. Since no match exists, the BNE instruction directs the program back to the LDAA, X instruction. The loop is repeated until the entire list is rewritten in locations 0040 through 004F. After the last entry in the list is copied, the index register is incremented to 0040. Thus, the CPX# instruction sets the Z flag allowing the BNE instruction to divert the program from the loop. The program halts after the last entry in the list is written in its new position in memory.

Instruction Set Summary

You have now been introduced to most of the instructions available to the 6800 MPU. You have also been introduced to all of the addressing modes. Now let's look at the complete instruction set.

Figure 5-24 summarizes the 6800's instructions and addressing modes. This 2-page Figure contains a wealth of information. For your convenience, this information is repeated on the Instruction Set Summary card provided with the course. You should keep this card handy. After a while, you will be able to write long, complex programs using only the card for reference.

The left-hand column of Figure 5-24 lists the names and mnemonics for each of the instructions. In many cases, a single name such as "add" is associated with more than one mnemonic. For example, ADDA is an add operation that involves accumulator A while ADDB is an add operation that involves accumulator B.

The center column gives important information about the addressing modes. Notice that the ADDA instruction can have any one of four addressing modes: immediate, direct, indexed, or extended. Three facts are given for each addressing mode. The hexadecimal opcode is given in the OP column. For example, the opcode for ADDA immediate is 8B while the opcode for ADDA direct is 9B.

The column labeled (~) tells the number of MPU cycles required to execute the instruction. This information is important because it allows us to determine exactly how long it will take to run a given program. As you will see later, an MPU cycle is equal to one cycle of the MPU clock. For example, if the clock frequency is 1 MHz, one MPU cycle will be one microsecond. With this clock rate, 2 microseconds are required to execute the ADDA immediate instruction while 5 microseconds are required for the ADDA indexed instruction.

The column labeled (#) indicates the number of bytes required by the instruction. ADDA immediate, ADDA direct, and ADDA indexed are two-byte instructions while ADDA extended is a three-byte instruction.

The next column to the right gives the shorthand notation for the Boolean or arithmetic operations performed. Finally, the right-hand column indicates how the condition code registers are affected by each instruction.

If you study the instruction set carefully, you will find that there are a few instructions that we have not yet discussed. These will be described in the next unit.

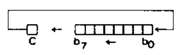
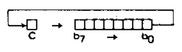
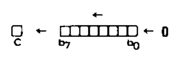
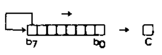
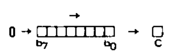
ACCUMULATOR AND MEMORY		ADDRESSING MODES															BOOLEAN/ARITHMETIC OPERATION	COND. CODE REG.					
		IMMED			DIRECT			INDEX			EXTND			INHER			(All register labels	5	4	3	2	1	0
OPERATIONS	MNEMONIC	OP	~	#	OP	~	#	OP	~	#	OP	~	#	OP	~	#	refer to contents)	H	I	N	Z	V	C
Add	ADDA	8B	2	2	9B	3	2	AB	5	2	BB	4	3				A + M → A	↕	●	↕	↕	↕	↕
	ADDB	CB	2	2	DB	3	2	EB	5	2	FB	4	3				B + M → B	↕	●	↕	↕	↕	↕
Add Acmltrs	ABA													1B	2	1	A + B → A	↕	●	↕	↕	↕	↕
Add with Carry	ADCA	89	2	2	99	3	2	A9	5	2	B9	4	3				A + M + C → A	↕	●	↕	↕	↕	↕
	ADCB	C9	2	2	D9	3	2	E9	5	2	F9	4	3				B + M + C → B	↕	●	↕	↕	↕	↕
And	ANDA	84	2	2	94	3	2	A4	5	2	B4	4	3				A • M → A	●	●	↕	↕	R	●
	ANDB	C4	2	2	D4	3	2	E4	5	2	F4	4	3				B • M → B	●	●	↕	↕	R	●
Bit Test	BITA	85	2	2	95	3	2	A5	5	2	B5	4	3				A • M	●	●	↕	↕	R	●
	BITB	C5	2	2	D5	3	2	E5	5	2	F5	4	3				B • M	●	●	↕	↕	R	●
Clear	CLR							6F	7	2	7F	6	3				00 → M	●	●	R	S	R	R
	CLRA													4F	2	1	00 → A	●	●	R	S	R	R
	CLRB													5F	2	1	00 → B	●	●	R	S	R	R
Compare	CMPA	81	2	2	91	3	2	A1	5	2	B1	4	3				A − M	●	●	↕	↕	↕	↕
	CMPB	C1	2	2	D1	3	2	E1	5	2	F1	4	3				B − M	●	●	↕	↕	↕	↕
Compare Acmltrs	CBA													11	2	1	A − B	●	●	↕	↕	↕	↕
Complement, 1's	COM							63	7	2	73	6	3				$\overline{M}$ → M	●	●	↕	↕	R	S
	COMA													43	2	1	$\overline{A}$ → A	●	●	↕	↕	R	S
	COMB													53	2	1	$\overline{B}$ → B	●	●	↕	↕	R	S
Complement, 2's	NEG							60	7	2	70	6	3				00 − M → M	●	●	↕	↕	①	②
(Negate)	NEGA													40	2	1	00 − A → A	●	●	↕	↕	①	②
	NEGB													50	2	1	00 − B → B	●	●	↕	↕	①	②
Decimal Adjust, A	DAA													19	2	1	Converts Binary Add. of BCD Characters into BCD Format	●	●	↕	↕	↕	③
Decrement	DEC							6A	7	2	7A	6	3				M − 1 → M	●	●	↕	↕	④	●
	DECA													4A	2	1	A − 1 → A	●	●	↕	↕	④	●
	DECB													5A	2	1	B − 1 → B	●	●	↕	↕	④	●
Exclusive OR	EORA	88	2	2	98	3	2	A8	5	2	B8	4	3				A ⊕ M → A	●	●	↕	↕	R	●
	EORB	C8	2	2	D8	3	2	E8	5	2	F8	4	3				B ⊕ M → B	●	●	↕	↕	R	●
Increment	INC							6C	7	2	7C	6	3				M + 1 → M	●	●	↕	↕	⑤	●
	INCA													4C	2	1	A + 1 → A	●	●	↕	↕	⑤	●
	INCB													5C	2	1	B + 1 → B	●	●	↕	↕	⑤	●
Load Acmltr	LDAA	86	2	2	96	3	2	A6	5	2	B6	4	3				M → A	●	●	↕	↕	R	●
	LDAB	C6	2	2	D6	3	2	E6	5	2	F6	4	3				M → B	●	●	↕	↕	R	●
Or, Inclusive	ORAA	8A	2	2	9A	3	2	AA	5	2	BA	4	3				A + M → A	●	●	↕	↕	R	●
	ORAB	CA	2	2	DA	3	2	EA	5	2	FA	4	3				B + M → B	●	●	↕	↕	R	●
Push Data	PSHA													36	4	1	A → M_{SP}, SP − 1 → SP	●	●	●	●	●	●
	PSHB													37	4	1	B → M_{SP}, SP − 1 → SP	●	●	●	●	●	●
Pull Data	PULA													32	4	1	SP + 1 → SP, M_{SP} → A	●	●	●	●	●	●
	PULB													33	4	1	SP + 1 → SP, M_{SP} → B	●	●	●	●	●	●
Rotate Left	ROL							69	7	2	79	6	3				M 	●	●	↕	↕	⑥	↕
	ROLA													49	2	1	A	●	●	↕	↕	⑥	↕
	ROLB													59	2	1	B	●	●	↕	↕	⑥	↕
Rotate Right	ROR							66	7	2	76	6	3				M 	●	●	↕	↕	⑥	↕
	RORA													46	2	1	A	●	●	↕	↕	⑥	↕
	RORB													56	2	1	B	●	●	↕	↕	⑥	↕
Shift Left, Arithmetic	ASL							68	7	2	78	6	3				M 	●	●	↕	↕	⑥	↕
	ASLA													48	2	1	A	●	●	↕	↕	⑥	↕
	ASLB													58	2	1	B	●	●	↕	↕	⑥	↕
Shift Right, Arithmetic	ASR							67	7	2	77	6	3				M 	●	●	↕	↕	⑥	↕
	ASRA													47	2	1	A	●	●	↕	↕	⑥	↕
	ASRB													57	2	1	B	●	●	↕	↕	⑥	↕
Shift Right, Logic.	LSR							64	7	2	74	6	3				M 	●	●	R	↕	⑥	↕
	LSRA													44	2	1	A	●	●	R	↕	⑥	↕
	LSRB													54	2	1	B	●	●	R	↕	⑥	↕
Store Acmltr.	STAA				97	4	2	A7	6	2	B7	5	3				A → M	●	●	↕	↕	R	●
	STAB				D7	4	2	E7	6	2	F7	5	3				B → M	●	●	↕	↕	R	●
Subtract	SUBA	80	2	2	90	3	2	A0	5	2	B0	4	3				A − M → A	●	●	↕	↕	↕	↕
	SUBB	C0	2	2	D0	3	2	E0	5	2	F0	4	3				B − M → B	●	●	↕	↕	↕	↕
Subract Acmltrs.	SBA													10	2	1	A − B → A	●	●	↕	↕	↕	↕
Subtr. with Carry	SBCA	82	2	2	92	3	2	A2	5	2	B2	4	3				A − M − C → A	●	●	↕	↕	↕	↕
	SBCB	C2	2	2	D2	3	2	E2	5	2	F2	4	3				B − M − C → B	●	●	↕	↕	↕	↕
Transfer Acmltrs	TAB													16	2	1	A → B	●	●	↕	↕	R	●
	TBA													17	2	1	B → A	●	●	↕	↕	R	●
Test, Zero or Minus	TST							6D	7	2	7D	6	3				M − 00	●	●	↕	↕	R	R
	TSTA													4D	2	1	A − 00	●	●	↕	↕	R	R
	TSTB													5D	2	1	B − 00	●	●	↕	↕	R	R

Figure 5-24
The 6800 instruction set.

INDEX REGISTER AND STACK POINTER OPERATIONS	MNEMONIC	IMMED			DIRECT			INDEX			EXTND			INHER			BOOLEAN/ARITHMETIC OPERATION	5	4	3	2	1	0
		OP	~	#	OP	~	#	OP	~	#	OP	~	#	OP	~	#		H	I	N	Z	V	C
Compare Index Reg	CPX	8C	3	3	9C	4	2	AC	6	2	BC	5	3				$(X_H/X_L) - (M/M+1)$	●	●	⑦	↕	⑧	●
Decrement Index Reg	DEX													09	4	1	$X - 1 \to X$	●	●	●	↕	●	●
Decrement Stack Pntr	DES													34	4	1	$SP - 1 \to SP$	●	●	●	●	●	●
Increment Index Reg	INX													08	4	1	$X + 1 \to X$	●	●	●	↕	●	●
Increment Stack Pntr	INS													31	4	1	$SP + 1 \to SP$	●	●	●	●	●	●
Load Index Reg	LDX	CE	3	3	DE	4	2	EE	6	2	FE	5	3				$M \to X_H, (M+1) \to X_L$	●	●	⑨	↕	R	●
Load Stack Pntr	LDS	8E	3	3	9E	4	2	AE	6	2	BE	5	3				$M \to SP_H, (M+1) \to SP_L$	●	●	⑨	↕	R	●
Store Index Reg	STX				DF	5	2	EF	7	2	FF	6	3				$X_H \to M, X_L \to (M+1)$	●	●	⑨	↕	R	●
Store Stack Pntr	STS				9F	5	2	AF	7	2	BF	6	3				$SP_H \to M, SP_L \to (M+1)$	●	●	⑨	↕	R	●
Indx Reg → Stack Pntr	TXS													35	4	1	$X - 1 \to SP$	●	●	●	●	●	●
Stack Pntr → Indx Reg	TSX													30	4	1	$SP + 1 \to X$	●	●	●	●	●	●

JUMP AND BRANCH OPERATIONS	MNEMONIC	RELATIVE			INDEX			EXTND			INHER			BRANCH TEST	5	4	3	2	1	0
		OP	~	#	OP	~	#	OP	~	#	OP	~	#		H	I	N	Z	V	C
Branch Always	BRA	20	4	2										None	●	●	●	●	●	●
Branch If Carry Clear	BCC	24	4	2										C = 0	●	●	●	●	●	●
Branch If Carry Set	BCS	25	4	2										C = 1	●	●	●	●	●	●
Branch If = Zero	BEQ	27	4	2										Z = 1	●	●	●	●	●	●
Branch If ≥ Zero	BGE	2C	4	2										$N \oplus V = 0$	●	●	●	●	●	●
Branch If > Zero	BGT	2E	4	2										$Z + (N \oplus V) = 0$	●	●	●	●	●	●
Branch If Higher	BHI	22	4	2										$C + Z = 0$	●	●	●	●	●	●
Branch If ≤ Zero	BLE	2F	4	2										$Z + (N \oplus V) = 1$	●	●	●	●	●	●
Branch If Lower Or Same	BLS	23	4	2										$C + Z = 1$	●	●	●	●	●	●
Branch If < Zero	BLT	2D	4	2										$N \oplus V = 1$	●	●	●	●	●	●
Branch If Minus	BMI	2B	4	2										N = 1	●	●	●	●	●	●
Branch If Not Equal Zero	BNE	26	4	2										Z = 0	●	●	●	●	●	●
Branch If Overflow Clear	BVC	28	4	2										V = 0	●	●	●	●	●	●
Branch If Overflow Set	BVS	29	4	2										V = 1	●	●	●	●	●	●
Branch If Plus	BPL	2A	4	2										N = 0	●	●	●	●	●	●
Branch To Subroutine	BSR	8D	8	2											●	●	●	●	●	●
Jump	JMP				6E	4	2	7E	3	3				See Special Operations	●	●	●	●	●	●
Jump To Subroutine	JSR				AD	8	2	BD	9	3					●	●	●	●	●	●
No Operation	NOP										01	2	1	Advances Prog. Cntr. Only	●	●	●	●	●	●
Return From Interrupt	RTI										3B	10	1		——	——	⑩	——	——	——
Return From Subroutine	RTS										39	5	1		●	●	●	●	●	●
Software Interrupt	SWI										3F	12	1	See special Operations	●	S	●	●	●	●
Wait for Interrupt	WAI										3E	9	1		●	⑪	●	●	●	●

CONDITIONS CODE REGISTER OPERATIONS	MNEMONIC	INHER			BOOLEAN OPERATION	5	4	3	2	1	0
		OP	~	#		H	I	N	Z	V	C
Clear Carry	CLC	0C	2	1	$0 \to C$	●	●	●	●	●	R
Clear Interrupt Mask	CLI	0E	2	1	$0 \to I$	●	R	●	●	●	●
Clear Overflow	CLV	0A	2	1	$0 \to V$	●	●	●	●	R	●
Set Carry	SEC	0D	2	1	$1 \to C$	●	●	●	●	●	S
Set Interrupt Mask	SEI	0F	2	1	$1 \to I$	●	S	●	●	●	●
Set Overflow	SEV	0B	2	1	$1 \to V$	●	●	●	●	S	●
Acmltr A → CCR	TAP	06	2	1	$A \to CCR$	——	——	——	⑫	——	——
CCR → Acmltr A	TPA	07	2	1	$CCR \to A$	●	●	●	●	●	●

CONDITION CODE REGISTER NOTES:

(Bit set if test is true and cleared otherwise)

①	(Bit V)	Test: Result = 10000000?
②	(Bit C)	Test: Result = 00000000?
③	(Bit C)	Test: Decimal value of most significant BCD Character greater than nine? (Not cleared if previously set.)
④	(Bit V)	Test: Operand = 10000000 prior to execution?
⑤	(Bit V)	Test: Operand = 01111111 prior to execution?
⑥	(Bit V)	Test: Set equal to result of $N \oplus C$ after shift has occurred.
⑦	(Bit N)	Test: Sign bit of most significant (MS) byte of result = 1?
⑧	(Bit V)	Test: 2's complement overflow from subtraction of LS bytes?
⑨	(Bit N)	Test: Result less than zero? (Bit 15 = 1)
⑩	(All)	Load Condition Code Register from Stack. (See Special Operations)
⑪	(Bit I)	Set when interrupt occurs. If previously set, a Non-Maskable Interrupt is required to exit the wait state.
⑫	(ALL)	Set according to the contents of Accumulator A.

LEGEND:

OP	Operation Code (Hexadecimal);	00	Byte = Zero;
~	Number of MPU Cycles;	H	Half-carry from bit 3;
#	Number of Program Bytes;	I	Interrupt mask
+	Arithmetic Plus;	N	Negative (sign bit)
–	Arithmetic Minus;	Z	Zero (byte)
•	Boolean AND;	V	Overflow, 2's complement
M_{SP}	Contents of memory location pointed to be Stack Pointer;	C	Carry from bit 7
		R	Reset Always
		S	Set Always
+	Boolean Inclusive OR;	↕	Test and set if true, cleared otherwise
⊕	Boolean Exclusive OR;	●	Not Affected
$\overline{M}$	Complement of M;	CCR	Condition Code Register
→	Transfer Into;	LS	Least Significant
0	Bit = Zero;	MS	Most Significant

Figure 5-24
(continued)

Self-Test Review

27. A disadvantage of direct addressing is that the operand must be in the first ________ bytes of memory.

28. The advantage of direct addressing is that only ________ bytes are required for each instruction.

29. Extended addressing can address ________ bytes of memory.

30. A disadvantage of extended addressing is that each instruction requires ________ bytes.

31. Can extended addressing be used to address an operand in the first 256_{10} bytes of memory?

32. The most powerful addressing mode available to the 6800 is called ________ addressing.

33. Indexed addressing requires ________ bytes for each instruction.

34. The second byte of an indexed addressing instruction is called the ________ address.

35. How is the address of the operand determined when indexed addressing is used?

36. Carefully examine the program shown in Figure 5-25. Determine what the program does and fill in the comments column. What number is loaded into the index register by the first instruction?

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ HEX CONTENTS	COMMENTS
0010	CE	LDX #	
0011	00	00	
0012	50	50	
0013	6F	CLR, X	
0014	00	00	
0015	08	INX	
0016	8C	CPX #	
0017	00	00	
0018	60	60	
0019	26	BNE	
001A	F8	F8	
001B	3E	WAI	

Figure 5-25
Program for Self-Test Review

37. What location is cleared by the CLR, X instruction?

38. What is the number in the index register after the INX instruction is executed for the first time?

39. The loop will be repeated until the number in the index register is ______________.

40. What does this program do?

41. Refer to Figure 5-24. What is the hexadecimal opcode for the LDAB extended instruction?

42. How many MPU cycles are required by the INC, X instruction?

43. How many bytes in the LDS # instruction?

Answers

27. 256_{10}.

28. Two.

29. $65{,}536_{10}$.

30. Three.

31. Yes. Although direct addressing is normally used when the operand is in the first 256_{10} bytes of memory, extended addressing can be used also.

32. Indexed.

33. Two.

34. Offset.

35. The offset address is added to the contents of the index register.

36. 0050_{16}.

37. 0050_{16}.

38. 0051_{16}.

39. 0060_{16}.

40. The program clears memory locations 0050_{16} through $005F_{16}$.

41. $F6_{16}$.

42. Seven.

43. Three.

UNIT 5 SUMMARY

1. In computer jargon, the word architecture is used to describe a microprocessor's style of construction.

2. From a programmer's point of view, the only parts of a microprocessor that are important are the registers that are accessible — the registers that can be directly modified by program code.

3. The programming model of the 6800 MPU contains the A and B Accumulators, the Index register, the Program Counter, the Stack Pointer, and Condition Codes register.

4. The A and B Accumulators in the 6800 MPU are 8-bit registers that operate as independent accumulators.

5. The 16-bit Program Counter can address up to 65,536 different addresses in memory.

6. The 6800 MPU has a 16-bit wide address bus.

7. Direct addressing is limited to the first 256 address locations in memory, because it uses an 8-bit address value.

8. Extended addressing uses a 16-bit (2-byte) address value, therefore it can access any address location in memory.

9. The Condition Code register in the 6800 MPU contains six 1-bit code registers. They are the:

   A. Carry code register that indicates if there was a carry from or borrow into the MSB of either accumulator.
   B. Overflow code register that indicates if there was an overflow from either accumulator.
   C. Zero code register that indicates if either accumulator is clear of 1-bits.
   D. Negative code register that indicates the state of the sign bit of either accumulator.
   E. Interrupt Mask code register that is used to determine how the 6800 MPU is effected by external interrupt requests.
   F. Half Carry code register that indicates if there was a carry from or borrow into the fourth bit of either 8-bit accumulator.

10. The Index register is a 16-bit register that can be used to hold 16-bit values and increment or decrement those values. It is also used to hold the 16-bit address values used in indexed addressing.

11. The Stack Pointer is another 16-bit working register that can be used to hold 16-bit values and increment or decrement those values. It is also used as an address pointer for "stack" operations.

12. When a 16-bit register is loaded with data, two 8-bit memory operations are used to supply that data. The first operation loads the upper half, or high-byte, of the register. The second memory operation loads the lower half, or low-byte, of the register.

13. The 6800 MPU has about 100_{10} basic instructions. Adding the different addressing modes brings the count to 197_{10} different opcodes.

14. There are seven general classifications of instructions: arithmetic, data handling, logic, data test, index register and stack pointer, jump and branch, and condition code.

15. Two's complement arithmetic is used within the 6800 MPU.

16. The add instructions add data in memory to an accumulator, or add the two accumulators together and store the sum in an accumulator.

17. The negate instructions perform a two's complement operation on data in memory or in either of the accumulators.

18. The decimal adjust instruction is used to convert the contents of the A accumulator into a valid BCD number after a BCD add operation.

19. The subtract instructions subtract data in memory or an accumulator from another accumulator and store the difference in an accumulator.

20. The clear instructions are used to clear (zero) a memory location or an accumulator.

21. The decrement instructions are used to subtract one from a value in a memory location, an accumulator, the Index register, or the Stack Pointer, and store the new value in the original location.

22. The increment instructions are used to add one to a value in a memory location, an accumulator, the Index register, or the Stack Pointer, and store the new value in the original location.

23. The rotate instructions are used to shift the eight bits in a memory location or accumulator and the carry condition code register bit one bit position left or right. On a rotate left, the MSB is shifted into carry, and the carry bit is shifted into the LSB. On a rotate right, the carry bit is shifted into the MSB, and the LSB is shifted into carry.

24. The shift instructions are used to shift the eight bits in a memory location or accumulator one bit position left or right. In each instance the carry condition code register bit is replaced by the bit shifted out of the accumulator or memory location. On a shift arithmetic left, the MSB is shifted into carry, and a zero is shifted into the LSB. On a shift arithmetic right, the LSB is shifted into carry, and the MSB is shifted back into the MSB as well as into the next bit position right. On a shift logical right, the LSB is shifted into carry, and a zero is shifted into the MSB.

25. The load instructions are used to move the contents of memory into either accumulator, the Index register, or the Stack Pointer.

26. The store instructions are used to save the contents of either accumulator, the Index register, or the Stack Pointer in memory.

27. The transfer instructions are used to load the contents of one accumulator into the other, load the contents of the Index register into the Stack Pointer, or load the contents of the Stack Pointer into the Index register.

28. The AND instructions are used to logically AND, bit-for-bit, the contents of an accumulator with a memory location, and store the result in the accumulator.

29. The OR instructions are used to logically OR, bit-for-bit, the contents of an accumulator with a memory location, and store the result in the accumulator.

30. The Exclusive OR instructions are used to logically XOR (EOR), bit-for-bit, the contents of an accumulator with a memory location, and store the result in the accumulator.

31. The complement instructions are used to perform a one's complement operation on a memory location or an accumulator.

32. The bit test instructions are used to logically AND, bit-for-bit, the contents of an accumulator with a memory location, and set the appropriate condition code register bits. The contents of the accumulator and memory location are not affected.

33. The compare instructions are used to subtract the contents of a memory location from an accumulator, an accumulator from an accumulator, or two memory locations from the Index register, and set the appropriate condition code register bits. The contents of the accumulators, Index register, and memory are not affected.

34. The test instructions are used to subtract zero from a memory location or an accumulator and set the appropriate condition code register bits. The contents of the accumulators or memory are not affected.

35. The unconditional branch instruction forces the MPU to execute the instruction pointed to by the relative address immediately following the branch instruction opcode. The relative address value is limited to $+127_{10}$ and -128_{10} bytes.

36. The conditional branch instruction causes the MPU to test the contents of the condition code register. If the proper conditions are met, the MPU is forced to execute the instruction pointed to by the relative address immediately following the branch instruction opcode. If the conditions are not met, the branch is ignored and the next sequential instruction is executed. The relative address is limited to $+127_{10}$ and -128_{10} bytes.

37. The no operation instruction is used to advance the Program Counter by one. You would use it to waste time in a program or reserve space in a program for future instructions.

38. The wait for interrupt instruction causes the MPU to go into a halt condition where it will only respond to an external interrupt.

39. The clear condition code instructions are used to clear to a zero state the carry, interrupt mask, or overflow register bits.

40. The set condition code instructions are used to set to a one state the carry, interrupt mask, or overflow register bits.

41. The transfer condition code register instructions are used to move the contents of the condition code register into the A accumulator, or the contents of the A accumulator into the condition code register.

42. In extended addressing, the second and third bytes of the instruction contain the high-order and low-order bytes, respectively, of the memory location being addressed.

43. In indexed addressing, the contents of the Index register are used as a pointer to the memory location being addressed. The second byte of the instruction contains an 8-bit, unsigned, offset value that is added to the Index register value to indicate the actual memory location being addressed.

Microprocessors

Book 2

Including Programming Experiments

THE 6800 MICROPROCESSOR — PART 2

Unit 6

INTRODUCTION

In the previous unit, you were introduced to the architecture and instruction set of the 6800 microprocessor. Much of the MPU's capabilities were discussed; however, three important areas were omitted. These include the microprocessor's stack operation, the use of subroutines, and the interrupt capability. These capabilities are discussed in detail in this unit. You are also introduced to input-output operations.

UNIT OBJECTIVES

When you have completed this unit, you will be able to:

1. Explain the difference between a cascade stack and a memory stack.

2. Write simple programs that can store data in — and retrieve data from — the stack.

3. Write programs that use the stack and indexing to move a list from one place in memory to another.

4. Explain the operations performed by each of the following instructions: PULA, PULB, PSHA, PSHB, DES, INS, LDS, STS, TXS, and TSX.

5. Define stack, subroutine, nested subroutine, interrupt, interrupt vector, and interrupt masking.

6. Write programs that use subroutines and nested subroutines.

7. Explain the operations performed by each of the following instructions: JMP, JSR, BSR, and RTS.

8. Describe how the 6800 MPU performs input and output operations.

9. Draw flowcharts depicting the sequence of events that occur during reset, non-maskable interrupt, interrupt request, software interrupt, return from interrupt, and wait for interrupt.

10. Explain the operation performed by each of the following instructions: WAI, SWI, RTI, SEI, and CLI.

STACK OPERATIONS

In computer jargon, a *stack* is a group of temporary storage locations in which data can be stored and later retrieved. In this regard, a *stack* is somewhat like memory. In fact, many microprocessors use a section of memory as a stack. The difference between a stack and other forms of memory is the method by which the data is accessed or addressed. The discussion will begin by considering a simple stack arrangement used in some microprocessors. Then the more sophisticated stack arrangement used by the 6800 MPU will be discussed.

Cascade Stack

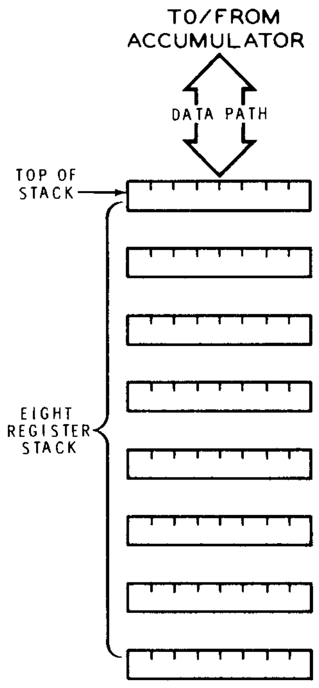


Figure 6-1
A cascade stack.

Some microprocessors have a special group of registers (usually 8 or 16) called a *cascade stack*. Each register can hold one 8-bit byte of data. Because these registers are right on the MPU chip, they make excellent temporary storage locations. If we need to free the accumulator for some reason, we can store its contents in the stack. Later, if that piece of data is needed again, we can retrieve the data from the stack. Of course, we could also have freed the accumulator by storing the data in memory. What then is the advantage of the stack?

One advantage of the stack is the method by which it is accessed or addressed. Recall that when a byte is stored in memory, an address is required. That is to store the contents of the accumulator in memory a 2-byte or 3-byte instruction is required. Depending on the addressing mode, the last one or two bytes is the address. Later, if the byte is retrieved, another instruction is required that also has an address.

An advantage of the stack is that data can be stored into it or read from it with single-byte instructions. That is, the instructions used with the stack do not require an address. Therefore, they are single-byte instructions.

Figure 6-1 shows an 8-register stack similar to that found in some microprocessors. This is called a cascade stack because of the method by which data is loaded and retrieved. All data transfers are between the top of the stack and the accumulator. That is, the accumulator communicates only with the top location on the stack. Data is transferred to the stack by a special instruction called PUSH.

The PUSH Instruction. Figure 6-2 illustrates how the PUSH instruction places data in the stack. The number 01_{16} is in the accumulator and we wish to temporarily store it. While we could store the number in memory, this would require a 2-byte or a 3-byte instruction. So instead, we use the PUSH instruction to place this number in the stack. Notice that the number is placed in the top location of the stack as shown in Figure 6-2A. The number remains there until we retrieve it or until we push another byte into the stack.

Figure 6-2B shows what happens if, at some time later, we push another byte into the stack. Notice that the accumulator now contains 03_{16}. If the PUSH instruction is executed, the contents of the accumulator are pushed into the top of the stack. To make room for this new number, the original number 01_{16} is pushed deeper into the stack.

Figures 6-2C and 6-2D show two more numbers being pushed into the stack at later points in the program. Notice that new data is always pushed into the top of the stack. To make room for the new data, the old data is pushed deeper into the stack. For this reason, this arrangement is often called a *push-down* or *cascade* stack. The name *cascade stack* comes from the characteristic cascading of data down through the stack as each new byte is pushed in at the top.

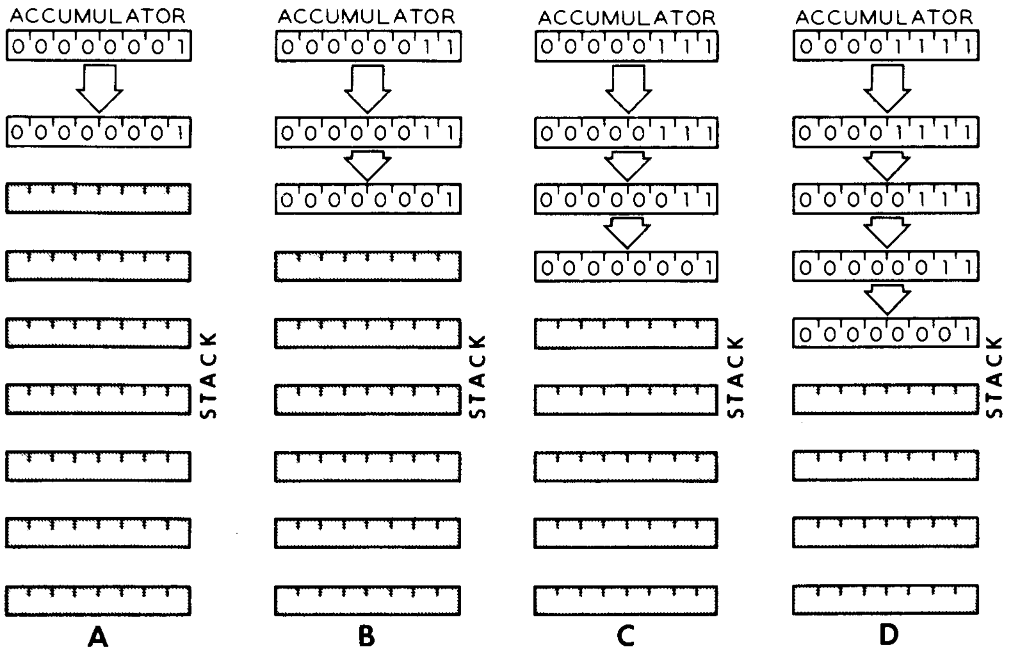


Figure 6-2.
Pushing data into the stack.

The PULL Instruction. The MPU retrieves data from the stack by using the PULL instruction. In some microprocessors, this is referred to as a POP instruction.

Figure 6-3 illustrates how data can be pulled (or popped) from the stack. Figure 6-3A shows the stack as it appeared after the last push operation. Notice that it contains four bytes of data. The last byte of data that was entered is at the top of the stack.

The PULL instruction retrieves the byte that is at the top of the stack. As this byte is removed from the stack, all other bytes move up, filling in the space left by that byte. Figure 6-3B illustrates how OF_{16} is pulled from the stack. Notice that 07_{16} is now at the top of the stack.

Figures 6-3C and 6-3D show how the next two bytes can be pulled from the stack. In each case, the remaining bytes move up in the stack, filling in the register vacated by the removed byte.

If you compare Figures 6-2 and 6-3, you will notice that the data must be pulled from the stack in the reverse order. That is, the last byte pushed into the stack is the first byte that is pulled from the stack. Another name for this arrangement is a last-in/first-out (LIFO) stack.

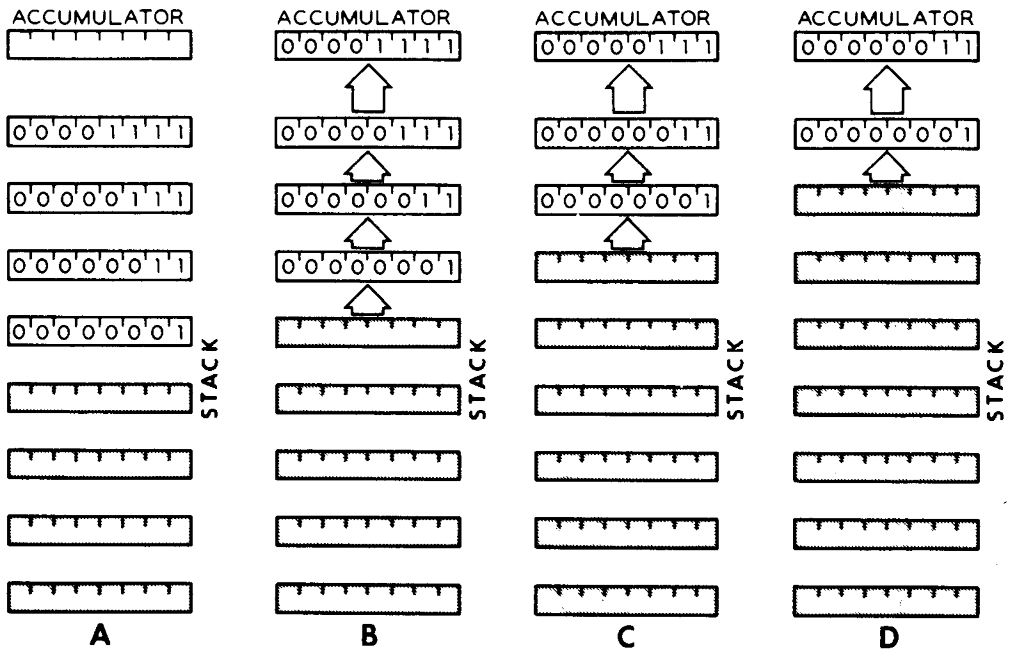


Figure 6-3.
Pulling data from the stack.

Memory Stack

While a cascade stack is valuable, it does have some limitations. For one thing, the number of registers is generally quite limited, with eight being typical. If more than eight pieces of data are pushed into the stack, the "older" bytes are pushed out the bottom and are lost. Also, the readout of the stack is destructive. When a byte is pulled from the stack, it no longer exists in the stack. This is fundamentally different from reading a byte from memory.

Because of these limitations the 6800 MPU does not use a cascade stack. Instead, a section of RAM can be set aside by the programmer to act as a stack. This has several advantages. First, the stack can be any length that the programmer requires. Second, the programmer can set up more than one stack if he likes. Third he can address the data in the stack using any of the instructions that address memory.

Stack Pointer. Recall that the 6800 MPU has a 16-bit register called the stack pointer. In a memory-type stack, the stack pointer defines the memory location that acts as the top of the stack.

The cascade stack considered earlier generally does not require a stack pointer. The top of the stack is determined by hardware. During push and pull operations, the data bytes actually move from one register to another. That is, the top of the stack remains stationary and the data moves up or down in relation to the stack.

In the memory stack, data cannot be easily transferred from one location to the next. Therefore, instead of moving data up and down in relation to the stack, it is much easier to move the top of the stack in relation to the data.

Generally, when the microprocessor-based system is being planned, a section of RAM is reserved for the stack. This should be a section of RAM that is not being used for any other purpose.

Once this is done, the stack can be set up by a program. The top of the stack is established by loading an address into the stack pointer. For example, suppose we wish to establish address $01F9_{16}$ as the top of the stack. The following instruction could be used:

```
LDS#
 01
 F9
```

This loads the address $01F9_{16}$ into the stack pointer and establishes that address as the top of the stack. However, as you will see, the top of the stack moves each time data is pushed into — or pulled from — the stack.

The PUSH Instructions. The 6800 MPU has two push instructions, PSHA and PSHB. These single-byte instructions push the contents of their respective accumulator onto the stack.

Figure 6-4 shows the effects of the PSHA instruction. Before the instruction is executed, the stack pointer contains the address $01F9_{16}$ as a result of a previous LDS instruction. Accumulator A contains a data byte (AA_{16}). If the PSHA instruction is now executed, the contents of accumulator A are pushed into memory location $01F9_{16}$. Then, the stack pointer is automatically decremented to $01F8_{16}$. This automatically moves the top of the stack as shown.

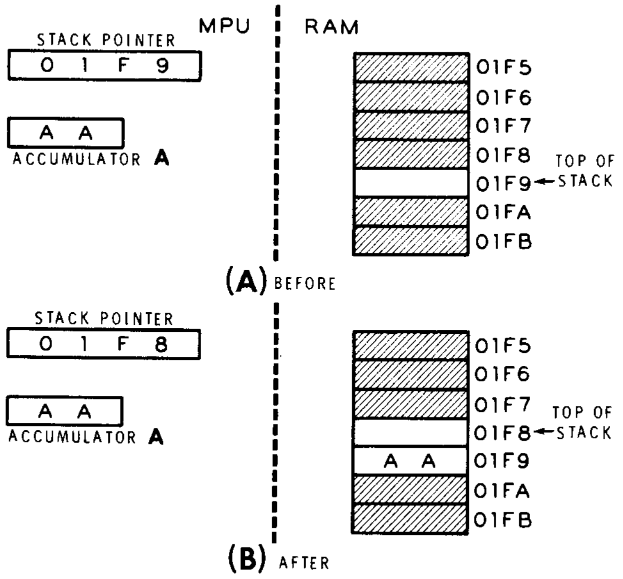


Figure 6-4
Executing the PSHA instruction.

If you look at your Instruction Set Summary card, you will see that the operation is described as follows:

$$A \rightarrow M_{SP},\ SP - 1 \rightarrow SP$$

This means that the contents of the A accumulator are transferred to the memory location specified by the stack pointer. Also, the contents of the stack pointer are replaced by the previous contents of the stack pointer minus one. In other words, after the accumulator-to-stack transfer takes place, the stack pointer is decremented by one.

To reinforce the idea, assume that at some later point in the program, the MPU executes a PSHB instruction. This is illustrated in Figure 6-5. Before PSHB is executed, the B accumulator contains BB_{16} and the stack pointer is still pointing to $01F8_{16}$. When PSHB is executed, the contents of accumulator B are pushed onto the stack and the stack pointer is decremented to $01F7_{16}$.

The PULL Instructions. Data bytes are removed from the stack with the pull instruction. The 6800 MPU has two pull instructions. PULA allows the MPU to pull data from the stack into the A accumulator. PULB performs a similar operation except the data byte goes into accumulator B. In each case, data is pulled from the top of the stack. Thus, the data byte available to the MPU is the last byte that was placed in the stack.

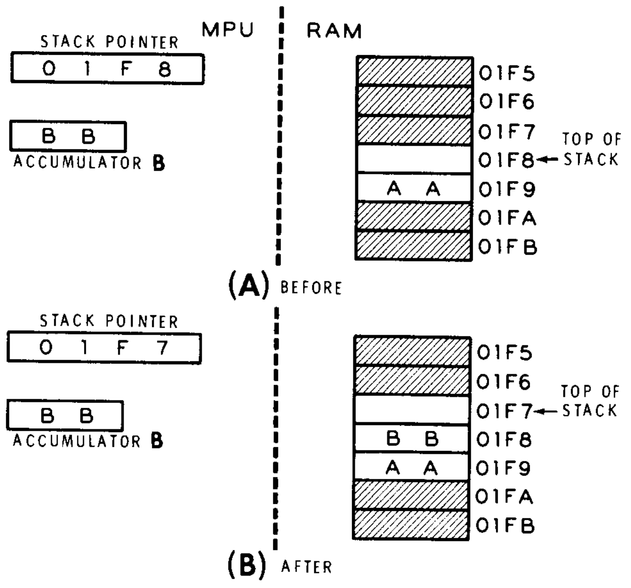


Figure 6-5
Executing the PSHB instruction.

For example, Figure 6-6A shows the stack as we left it after the last push instruction. Figure 6-6B shows what happens if the PULA instruction is executed. First, the stack pointer is automatically incremented by one to $01F8_{16}$. Then the contents of the memory location designated by the stack pointer are transferred to accumulator A. Thus, BB_{16} goes into accumulator A. Notice that the stack pointer is incremented *before* the byte is pulled from the stack.

To be certain you have the idea, consider what happens if the PULB instruction is now executed. Figure 6-6C shows that the stack pointer is automatically incremented to $01F9_{16}$. The contents of that location are then pulled into accumulator B. This operation is described on your Instruction Set Summary card as:

$$SP + 1 \rightarrow SP, M_{SP} \rightarrow B .$$

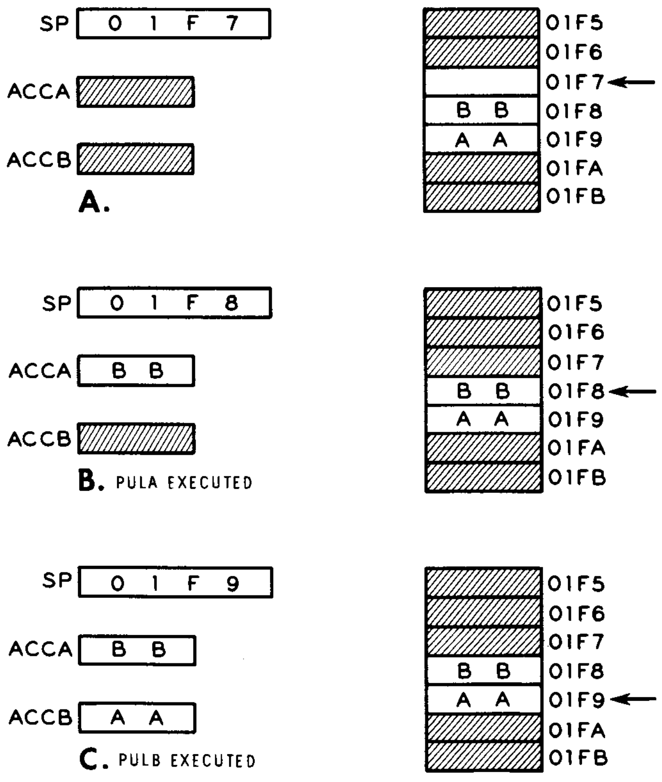


Figure 6-6
Executing PULL instructions.

Using the Stack. Figure 6-7 summarizes all of the instructions that directly affect stack operations. The push and pull instructions were introduced in this unit while the other instructions were discussed briefly in the previous unit. Find these instructions on your Instruction Set Summary card. The push and pull instructions are listed with the Accumulator and Memory Operations. Those instructions that affect the stack pointer are listed under Index Register and Stack Pointer Operations.

STACK AND STACK POINTER		ADDRESSING MODES															
		IMMED			DIRECT			INDEX			EXTND			INHER			BOOLEAN/ARITHMETIC OPERATION (All register labels refer to contents)
OPERATIONS	MNEMONIC	OP	~	#	OP	~	#	OP	~	=	OP	~	#	OP	~	#	
Push Data	PSHA													36	4	1	$A \rightarrow M_{SP}$, $SP - 1 \rightarrow SP$
	PSHB													37	4	1	$B \rightarrow M_{SP}$, $SP - 1 \rightarrow SP$
Pull Data	PULA													32	4	1	$SP + 1 \rightarrow SP$, $M_{SP} \rightarrow A$
	PULB													33	4	1	$SP + 1 \rightarrow SP$, $M_{SP} \rightarrow B$
Decrement Stack Pntr	DES													34	4	1	$SP - 1 \rightarrow SP$
Increment Stack Pntr	INS													31	4	1	$SP + 1 \rightarrow SP$
Load stack Pntr	LDS	8E	3	3	9E	4	2	AE	6	2	BE	5	3				$M \rightarrow SP_H$, $(M + 1) \rightarrow SP_L$
Store Stack Pntr	STS				9F	5	2	AF	7	2	BF	6	3				$SP_H \rightarrow M$, $SP_L \rightarrow (M + 1)$
Indx Reg → Stack Pntr	TXS													35	4	1	$X - 1 \rightarrow SP$
Stack Pntr → Indx Reg	TSX													30	4	1	$SP + 1 \rightarrow X$

Figure 6-7
Stack and stack pointer instructions.

Following are some examples of how the stack can be used. First consider a trivial example. Using only stack operations, swap the contents of accumulators A and B. Assuming the stack pointer has already been set up, the program segment might look like this:

```
PSHA
PSHB
PULA
PULB
```

Assume that accumulator A initially contains AA_{16} and that accumulator B contains BB_{16}. The first instruction pushes AA_{16} onto the stack. Next BB_{16} is pushed onto the stack. The third instruction pulls BB_{16} from the top of the stack and places it in accumulator A. Finally, the last instruction pulls AA_{16} from the stack and places it in accumulator B. As you can see, the contents of the two accumulators are reversed. The following routine accomplishes the same thing with one less instruction:

```
PSHA
TBA
PULB
```

Now look at a more complex example. Assume that you wish to transfer 16_{10} bytes of data from one place in memory to another. As you saw in the previous unit, this type of problem is a good candidate for indexing. However, indexing alone becomes cumbersome if the two lists are over FF_{16} memory locations apart. The reason for this is that the offset address can only extend FF_{16} locations above the address in the index register.

In this example, assume you wish to move the data in memory locations 0010_{16} through $001F_{16}$ to locations $01F0_{16}$ to $01FF_{16}$. While this could be accomplished using indexing alone, the program becomes unnecessarily complicated. Two separate indexes must be maintained; one for loading data from 0010_{16} through $001F_{16}$, the other for storing data in $01F0_{16}$ through $01FF_{16}$. A simpler approach is to use indexing for one operation and the stack capability for the other operation. That is, we could load data from the lower list using indexing and store it in the upper list using the stack capability.

A program that does this is shown in Figure 6-8. The first instruction loads the stack pointer with address $01FF_{16}$. This is the address of the last entry in the new list that will be formed. Recall that the new list is to be written in locations $01F0_{16}$ through $01FF_{16}$. Once location $01FF_{16}$ is established as the top of the stack, we can enter data into the new list simply by pushing data onto the stack. Because the stack pointer is decremented with each push operation, we must push the last entry in the list onto the stack first.

Figure 6-8.
Moving a list of data using both indexing and stack operations.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0020	8E	LDS#	Load the stack pointer immediately with the
0021	01	01	address of the last entry in the
0022	FF	FF	new list.
0023	CE	LDX#	Load the index register immediately with the
0024	00	00	address of the last entry in the
0025	1F	1F	original list.
0026	A6	LDAA, X	Load accumulator A indexed from
0027	00	00	the original list.
0028	36	PSHA	Push the contents of accumulator A into the new list.
0029	09	DEX	Decrement the index register.
002A	8C	CPX#	Compare the contents of the index register
002B	00	00	with one less than the address of the
002C	0F	0F	first entry in the original list.
002D	26	BNE	If no match occurs, branch back
002E	F7	F7	this far.
002F	3E	WAI	Otherwise, wait.

The second instruction loads the index register with the address of the *last* entry in the original list. This is necessary for the reason pointed out above.

Next, the A accumulator is loaded using indexed addressing. Since the offset address is 00_{16}, the accumulator is loaded with the contents of $001F_{16}$. That is, the last entry in the original list is loaded into accumulator A.

The PSHA instruction then pushes the contents of accumulator A onto the stack. Thus, the last entry in the original list is transferred to location $01FF_{16}$. In the process, the stack pointer is automatically decremented to $01FE_{16}$.

The index register is decremented to $001E_{16}$ by the next instruction. Then, the CPX instruction compares the index register with $000F_{16}$ to see if all entries in the list have been moved. If no match occurs, the MPU branches back and picks up the next entry in the list. The loop is repeated over and over again until the entire list has been moved to its new location.

Other uses of the stack will be revealed later. However, even if the stack did nothing more than has already been explained, it would be a very useful capability to have. But as you will see, the MPU uses the stack in several other ways that makes this capability even more important.

Self-Test Review

1. What is a stack?
2. What is a cascade stack?
3. What is a memory stack?
4. Which type of stack does the 6800 MPU use?
5. What is the name of the instruction that stores data in the stack?
6. What type of instruction is used to retrieve data from the stack?
7. What is the purpose of the stack pointer?
8. The PUSH instruction transfers data from one of the accumulators to ____________________.
9. The PULB instruction transfers data from the top of the stack to ____________________.
10. Refer to Figure 6-8. How can we change this program so that the new list is placed in addresses 0220_{16} through 022F?

Answers

1. A stack is a group of registers or a section of memory that is used as a last-in, first-out memory.

2. A cascade stack is a group of hardware registers (usually 16 or less) that is used as a last-in, first-out memory.

3. A memory stack uses a section of RAM as a last-in, first-out memory.

4. A memory stack

5. PUSH

6. PULL

7. The stack pointer indicates the address of the top of the stack.

8. The top of the stack.

9. Accumulator B.

10. By changing the first instruction to: LDS# $022F_{16}$.

SUBROUTINES

A subroutine is a group of instructions that performs some limited but frequently required task. A given subroutine may be used many times during the execution of the main program. In many cases, the easiest way to write a program is to break the overall job down into many simple operations, each of which can be performed by a subroutine.

Because subroutines are used so frequently, most microprocessors have special capabilities that allow them to handle subroutines efficiently. In this section, these capabilities will be examined. The discussion will start with the instructions associated with subroutines.

The 6800 MPU has three instructions that are used to handle subroutines. They are:

Jump to Subroutine (JSR)
Branch to Subroutine (BSR)
Return from Subroutine (RTS)

Each of these will be discussed in this section. One other instruction that has not yet been mentioned will also be discussed. It is the Jump (JMP) instruction. While not used exclusively with subroutines, the JMP instruction makes an excellent introduction to the Jump to Subroutine (JSR) instruction. Therefore, the Jump (JMP) instruction will be discussed first.

Jump (JMP) Instruction

This instruction allows the MPU to jump from one point in a program to another. In this respect, it is somewhat like the Branch Always (BRA) instruction that was discussed earlier. The difference is the method of addressing used. Recall that the BRA instruction used relative addressing. This has the advantage that only a 2-byte instruction is required. Its disadvantage is that the branch must be within the range of −128 bytes to +127 bytes of the program count.

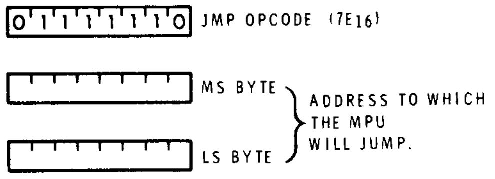


Figure 6-9
Format of the JMP instruction using extended addressing.

The JMP instruction can use either the indexed or the extended addressing mode. It does not use relative addressing. When using extended addressing, the format of the JMP instruction is as shown in Figure 6-9. Three bytes are required; the opcode followed by the 2-byte address to which the MPU is to jump. Since a 16-bit address is given, the jump may be to any point in the $65{,}536_{10}$ byte memory range. This address is loaded into the program counter so that the next opcode is fetched from that address. The previous contents of the program counter are lost. Thus, the MPU starts executing instructions from a new point in memory.

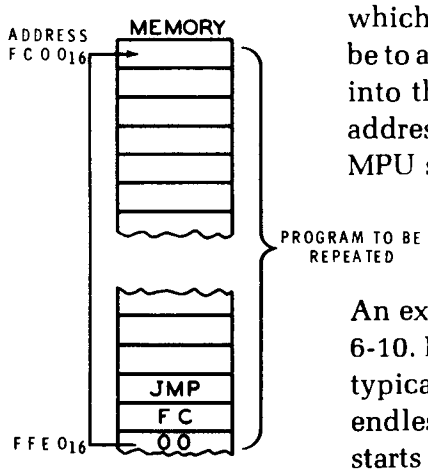


Figure 6-10
Using the JMP instruction to repeat a program.

An example of how the JMP instruction can be used is shown in Figure 6-10. Here, a long program is to be repeated over and over again. This is typical of applications such as controllers that repeat the same operations endlessly. The program is contained in the upper 1k bytes of memory. It starts at location $FC00_{16}$ and ends at $FFE0_{16}$. Notice that the last instruction is JMP $FC00_{16}$. This sends the program back to its beginning so that the loop is repeated endlessly.

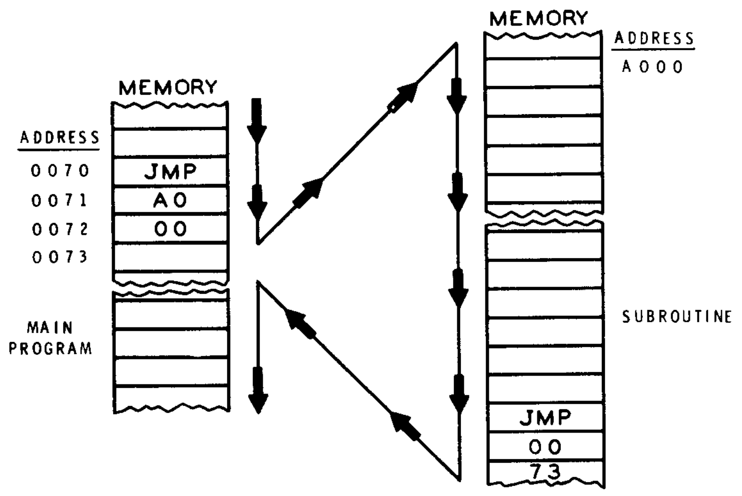


Figure 6-11
Using the JMP instruction to call a subroutine.

Another possible use of the JMP instruction is shown in Figure 6-11. Here, the main program is in the lower memory locations shown on the left. The main program requires a subroutine that is up at address A000 (shown on the right). The JMP instruction at address 0070 sends the MPU off to the subroutine as shown. The last instruction in the subroutine is another JMP instruction that sends the MPU back to the main program.

Jumping to a subroutine is often referred to as *calling* a subroutine. While we can call a subroutine using the JMP instruction, this approach has a distinct problem. What happens if the main program wants to call the same subroutine more than once? That is, suppose a situation like that shown in Figure 6-12 is required. Here, the main program (on the left) wishes to call the subroutine (on the right) at two separate points. Jumping to the subroutine is no problem. We can do that as many times as we like, using the instruction JMP A000. The problem is: how do we get back from the subroutine to the main program? The first time through the subroutine, the MPU should return to address 0073. The second time through, the MPU should return to address 0093.

A programmer could get around this problem by changing the last instruction in the subroutine before each call or by constructing a table of return addresses, etc. However, most microprocessors have some instructions that solve this problem for us. The following section will discuss the 6800 MPU's solution to this problem.

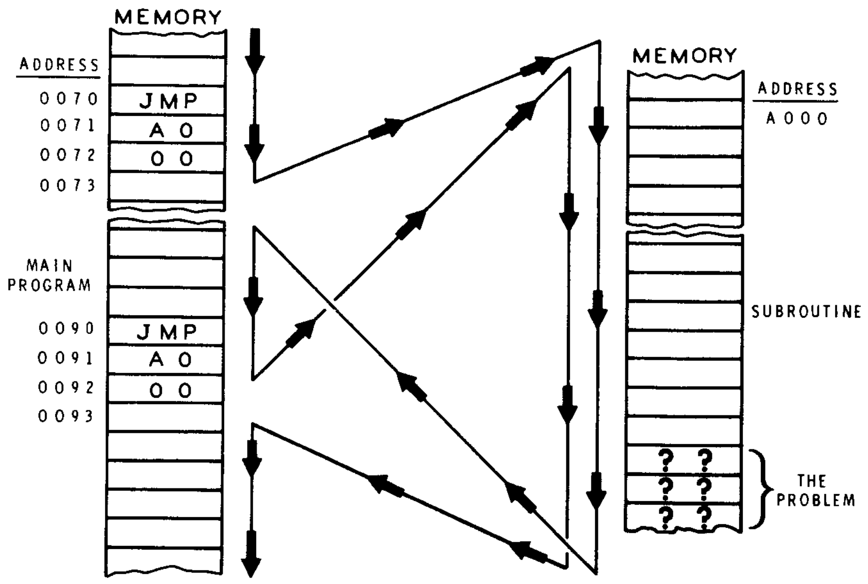


Figure 6-12
The JMP instruction cannot handle situations like this one.

JSR and RTS Instructions

If you refer to Figure 6-12 again, you will see that this problem arises because the old program count is not saved when the MPU jumps from one location to the next. However, the 6800 MPU has an instruction that will not only jump to a subroutine, it will also cause the old program count to be stored away. This instruction is called the Jump to Subroutine (JSR) instruction. Its format is exactly the same as the JMP instruction but its execution is different.

Figure 6-13 shows how the earlier problem can be solved using the JSR instruction. Notice that the two JMP instructions in the main program have been replaced by JSR instructions. Notice also that the last instruction in the subroutine is a Return from Subroutine (RTS) instruction. These new instructions ease the problem of calling the subroutine.

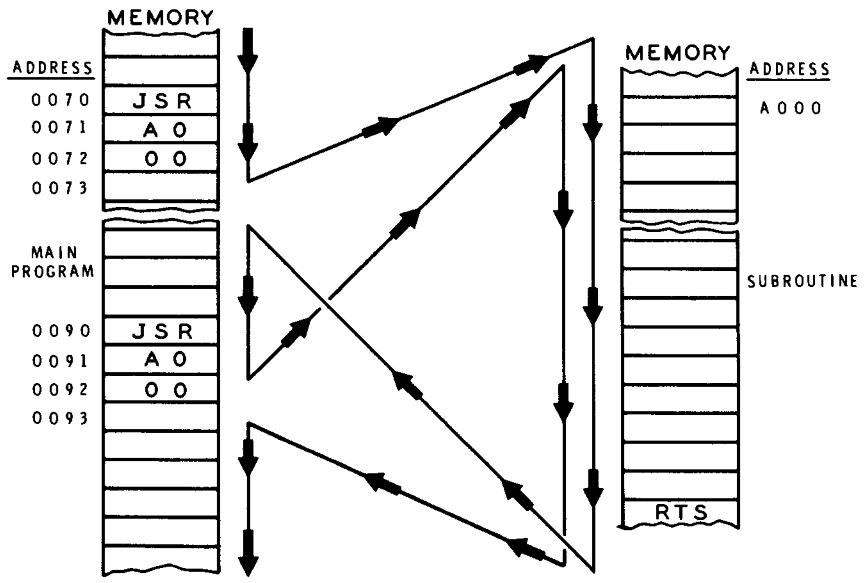


Figure 6-13
The JSR and RTS instructions can be used to handle this situation.

When the first JSR instruction is executed, the subroutine address $A000_{16}$ is placed in the program counter. However, just prior to this, the program counter was incremented to the address of the next instruction in sequence. That is, the program counter was advanced to 0073_{16} while the contents of address 0072_{16} were being retrieved. This count (0073_{16}) is automatically pushed onto the stack. By saving the old program count, the MPU can tell where to return after the subroutine is finished. As soon as the old program count is tucked away safely in the stack, the subroutine address $A000_{16}$ is placed in the program counter. Thus, the MPU fetches the next instruction from address $A000_{16}$.

Notice that the last instruction in the subroutine is an RTS instruction. When the MPU encounters this single-byte instruction, it will jump back to the point where it left off in the main program. It does this by pulling the old program count (0073_{16}) from the stack and placing it in the program counter. Consequently, the next instruction will be fetched from address 0073_{16}. As you can see, this returns the MPU to the correct point in the main program.

Notice that the programmer does not specify a return address at the end of the subroutine. The return address is automatically pulled from the stack. This allows us to call the subroutine repeatedly from several different points in the main program.

Figure 6-13 shows that the subroutine is called again by the JSR A000 instruction in location 0090_{16}. As this instruction and address are decoded, the program count is incremented to 0093_{16}. This program count is pushed onto the stack. Then $A000_{16}$ is placed in the program counter. Thus, the MPU jumps off to the subroutine. When the subroutine is finished, the RTS instruction causes the old program count to be pulled from the stack into the program counter. This causes the MPU to jump back to address 0093_{16} which contains the next instruction in the main program.

Nested Subroutines

Figure 6-14 shows a situation in which the main program calls subroutine A. In turn, subroutine A calls subroutine B. In this situation, subroutine B is called a *nested* subroutine. That is, a nested subroutine is a program segment that is called by another subroutine. If control is to be eventually returned to the main program, two program counts must be saved. Figure 6-14 shows how the two program counts are saved in the stack.

At the start of the main program, the stack pointer is loaded with the address of the area in memory that has been set aside to act as the stack. If no stack instructions have been executed when the main program arrives at the first JSR instruction, the stack pointer will still be pointing to where it was originally set. The contents of the stack are of no interest until this point.

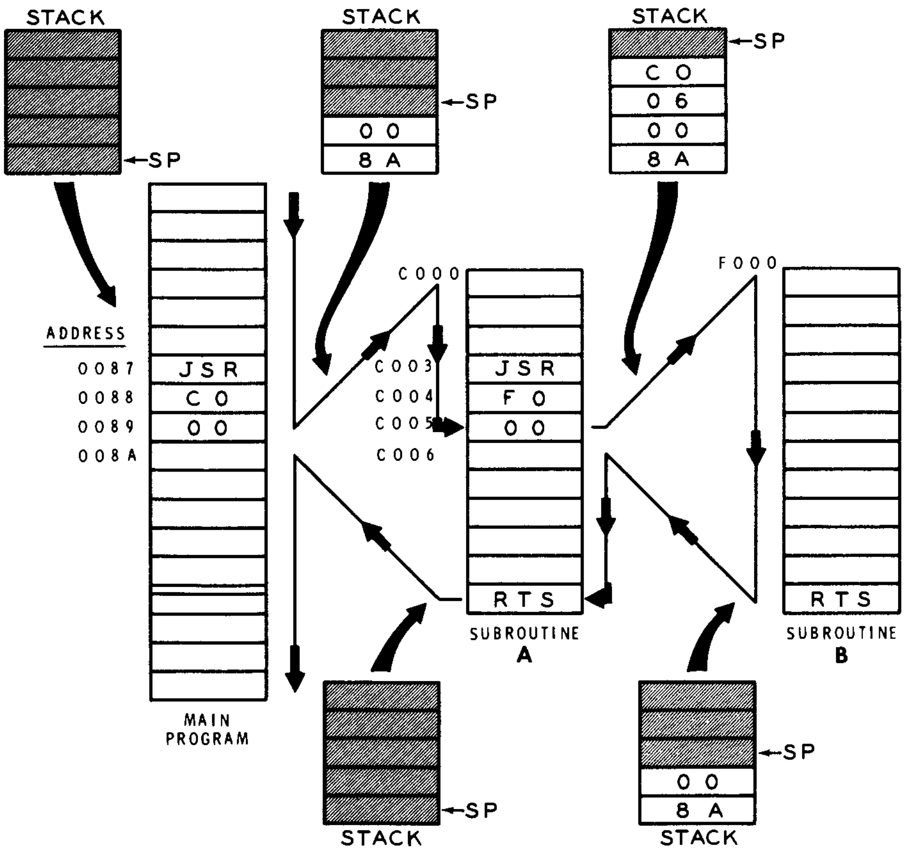


Figure 6-14
Handling nested subroutines.

When the main program reaches the JSR instruction, the program count is advanced to the address of the next instruction in sequence ($008A_{16}$). When the JSR instruction is executed, this address ($008A_{16}$) is pushed onto the stack as shown. The low order byte goes in first, followed by the high order byte. In the process, the stack pointer is decremented twice. Finally, the new address ($C000_{16}$) is placed in the program counter. This causes the MPU to jump off to subroutine A which starts at $C000_{16}$.

Notice that halfway through subroutine A, subroutine B is called. Consequently, the return address in subroutine A ($C006_{16}$) must be saved. That is, when the program reaches the JSR instruction in subroutine A, the return address ($C006_{16}$) is pushed onto the stack as shown. Notice that there are now two return addresses in the stack. The starting address of subroutine B ($F000_{16}$) is then placed in the program counter and the MPU jumps off to this subroutine.

Subroutine B has no nested subroutines of its own, so the program flow is through the subroutine as shown. The last instruction in subroutine B is the RTS instruction. At this point, the MPU pulls the return address ($C006_{16}$) from the top of the stack and places it in the program counter. This causes the MPU to jump back to the instruction at address $C006_{16}$ in subroutine A.

The remainder of subroutine A is then executed down to the RTS instruction. This instruction causes the MPU to pull the next address ($008A_{16}$) from the stack and place it in the program counter. Notice that this sends the MPU back to the main program.

For simplicity, a single level of subroutine nesting is shown in this example. However, in practice, many levels of nesting may be used. For example, subroutine B could call subroutine C; etc. Any level of nesting can be used as long as enough memory is set aside for the stack. Remember, each return address requires two bytes in the stack.

Branch to Subroutine (BSR) Instruction

Quite often, the subroutine we wish to call is within the -128_{10} to $+127_{10}$ byte range of the relative address. When it is, we can save one byte by using the Branch to Subroutine (BSR) instruction. The execution of BSR is identical to that of JSR except that relative addressing is used. The old program count is saved in the stack before the branch occurs. Thus, the RTS instruction at the end of the subroutine will cause the old program count to be restored.

Summary of Subroutine Instructions

Figure 6-15 shows the four instructions discussed in this section. Notice that the BSR instruction uses relative addressing. The JMP and JSR instructions can use either indexed or extended addressing. The RTS instruction uses inherent addressing since its address is pulled from the top of the stack.

Find these instructions on your Instruction Set Summary card. The operations performed by these instructions are illustrated under "Special Operations" on the back of the card. Also, Appendix A of this course gives a concise description of the operations performed by each of these instructions.

JUMP AND BRANCH		RELATIVE			INDEX			EXTND			INHER		
OPERATIONS	MNEMONIC	OP		#	OP		#	OP		#	OP		#
Branch To Subroutine	BSR	8D	8	2									
Jump	JMP				6E	4	2	7E	3	3			
Jump To Subroutine	JSR				AD	8	2	BD	9	3			
Return From Subroutine	RTS										39	5	1

Figure 6-15.
Subroutine and jump instructions.

Self-Test Review

11. What is a subroutine?

12. What addressing modes can the JMP instruction use?

13. How does the JMP instruction differ from the BRA instruction?

14. How does the execution of the JSR instruction differ from that of the JMP instruction?

15. Why is the program count saved when the JSR or BSR instructions are executed?

16. Where is the program count saved?

17. How is the stack pointer affected by the JSR instruction?

18. Generally, the last instruction in the subroutine will be a ____________________ instruction.

19. What is a nested subroutine?

20. How is the stack pointer affected by the RTS instruction?

Answers

11. A subroutine is a group of instructions that performs some specific, limited task that is used more than once by the main program.

12. Indexed and extended.

13. Since the BRA instruction uses relative addressing, it can branch only in a -128_{10} to $+127_{10}$ byte range. The JMP instruction uses indexed or extended addressing. Therefore, it can jump to any point in memory.

14. When the JSR instruction is executed, the program count is saved in the stack.

15. The program count is saved so that when the subroutine is finished, the MPU can return to the point it left off.

16. The program count is pushed into the top two locations of the stack.

17. The stack pointer is automatically decremented twice as the program count is pushed onto the stack.

18. Return from Subroutine (RTS).

19. When subroutine A calls subroutine B, subroutine B is said to be nested.

20. The stack pointer is automatically incremented twice as the old program count is pulled from the stack.

INPUT — OUTPUT (I/O) OPERATIONS

A full explanation of input-output (I/O) operations will be given in the next units, but a brief introduction to I/O is necessary at this point. In this section, you will learn what is involved in sending data to — or taking data from — the MPU.

To be useful, a microprocessor system must accept data from the outside world, process it in some way, and present results to the outside world. The input device may be nothing more than a group of switches while the output device can be as simple as a bank of indicator lamps. On the other hand, a single microprocessor might handle several teletypewriters, printers, papertape machines, etc. The point is that the I/O requirements can vary greatly from one application to the next. This section will be concerned with the simplest form of I/O operations.

In the short history of microprocessors, two distinctly different methods have been developed for handling I/O operations. In some microprocessors, I/O operations are handled by I/O instructions. These microprocessors generally have one *input* instruction and one *output* instruction. When the *input* instruction is executed, a byte is transferred from the selected I/O device to a register (usually one of the accumulators) in the MPU. The I/O device is selected by sending out a device selection byte on the address bus. By using an 8-bit byte for device selection, the MPU can specify up to 256_{10} different I/O devices. Of course, no microprocessor system uses that many devices, but the capability is there. The *output* instruction causes a data transfer from the accumulator to the selected I/O device. While this method of handling I/O operations is used in many microprocessors, the 6800 MPU uses a different technique.

The other method for handling I/O operations is to treat all I/O transfers as memory transfers. This is the method used by the 6800 MPU and many other microprocessors. In fact, even those microprocessors that have I/O instructions can ignore those instructions and handle I/O operations as memory transfers.

The 6800 MPU has no I/O instructions. An I/O device is assigned an address and is treated as a memory location. For example, assume that an input keyboard has been assigned an address of 8000_{16}. We can input data into accumulator A by using the instruction:

LDAA 8000_{16}

By the same token, an output display may have been assigned the address 9000_{16}. In this case, we can output from accumulator B by using the instruction:

STAB 9000_{16}.

As you can see, the I/O device is treated as a memory location. The system block diagram shown in Figure 6-16 shows how an I/O device is connected to the microcomputer. Notice that both the data bus and the address bus connect to the I/O interface. As you will see in the next unit, the interface can consist of an address decoder, an output or input latch, and buffers or drivers.

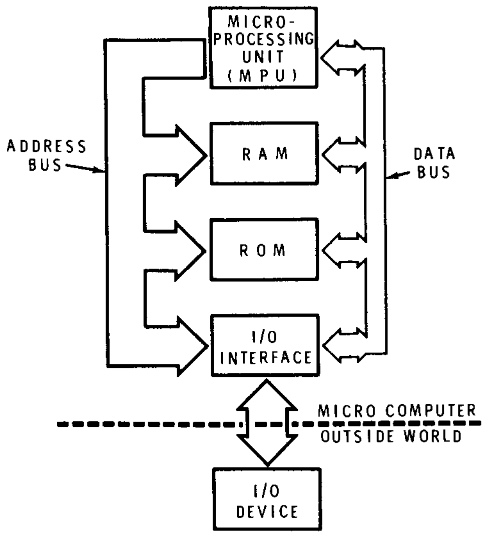


Figure 6-16
Adding I/O to the microcomputer.

The address decoder monitors the address bus and enables the interface circuitry whenever the proper address is detected. This prevents the I/O interface from interfering when data is being transferred between memory and the MPU.

The I/O interface will generally have an output latch if it is to be used for an output operation. The reason for this is that the data from the MPU will appear on the data lines for only an instant (usually less than one microsecond). By storing the output data in a latch, the I/O device is given a much longer period of time to examine and respond to the data.

Buffers or drivers are also included in the I/O interface. As you will see later, these are frequently necessary when several different circuits are sharing the same bus.

Output Operations

Figure 6-17 shows a simplified output circuit. Here, the output device is a bank of eight light emitting diodes (LEDs). Enough detail is shown to illustrate how an output operation can be performed. The address decoder monitors the address bus, looking for the address 9000_{16}. It also monitors some of the control lines that connect to the MPU. One of those lines is called a read-write line. It goes to its low state when a write (output) operation is initiated by the MPU. The other control lines will be discussed in the next unit.

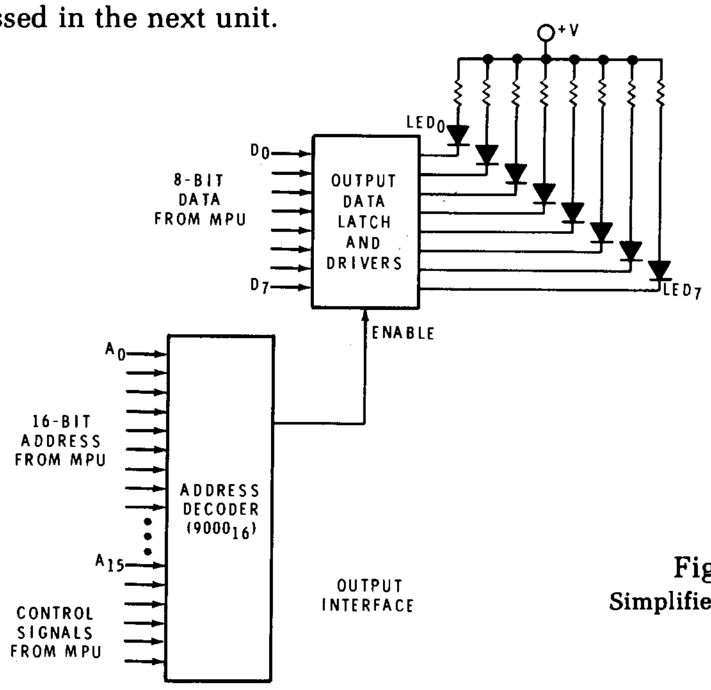


Figure 6-17
Simplified output circuit.

Notice that the output of the address decoder is used to enable the output data latch and drivers. When these are enabled, the byte on the data lines is stored in the latch. The data bits stored in the latch cause the appropriate LEDs to light up. By outputting appropriate bit patterns, the MPU can cause different binary numbers to be displayed.

Notice that the address decoder (and therefore the display) is given the address 9000_{16}. We can output data to the display in several different ways. For example, we can load the appropriate pattern to be displayed into accumulator A. Then by executing a "store accumulator A" extended instruction, we can transfer the contents of the accumulator to the display. The instruction would be: STAA 9000_{16}. Or, we could output data from accumulator B by using the instruction: STAB 9000_{16}.

In either case, the address 9000_{16} goes out on the address bus for a brief interval of time. The address decoder recognizes this address. At the same time, the control lines indicate that an output operation is called for. In particular, the read-write line goes low. This causes the address decoder to enable the output.data latch for an instant. Simultaneously, the 8-bit data byte appears on the data bus. The output latch stores the data byte. The data appears at the input of the latch for less than a microsecond (typically). However, once the data is stored, it appears at the output of the latch until new data is written in. Thus, the output data will be displayed until the next byte of data is outputted by the MPU.

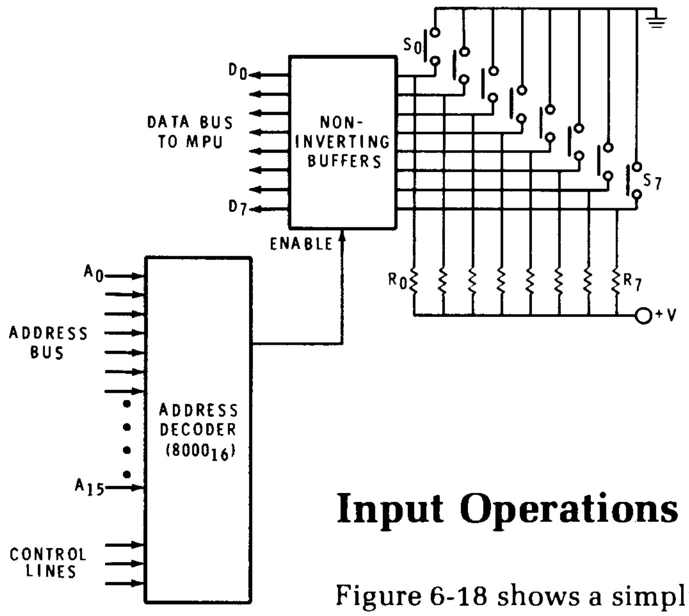


Figure 6-18
Simplified input circuit.

Input Operations

Figure 6-18 shows a simplified input circuit. Here, the input device is a bank of eight switches. When a switch is open, its respective input line to the buffer is held high by the pull-up resistors. However, when a switch is closed, its respective input line is pulled low because the switch connects it to ground.

In this simple circuit, no latch is required between the switches and the data bus. However, a buffer is used so that the switch bank can be effectively disconnected from the data bus when the switches are not being addressed.

As with the output circuit, an address decoder monitors the address and control lines. Notice that the assigned address is 8000_{16}. To input data from the switch bank to accumulator A, we use the instruction: LDAA 8000_{16}.Or, we could input the data to accumulator B by using the instruction: LDAB 8000_{16}.

In either case, the address 8000_{16} is placed on the address line. The address decoder recognizes this address and enables the buffer. For a brief interval (typically less than one microsecond), the lines of the data bus assume the same state as the lines on the right side of the buffer. If no switch is depressed, all data lines will be high and all 1's (FF_{16}) will be loaded into the accumulator. However, if one of the switches (S_0, for example) is depressed, its respective data line (D_0) will be low. In this case, the number read into the accumulator will be FE_{16}. By examining the byte that is read in, the MPU can determine which switch is depressed.

Input — Output Programming

You now know enough about simple input/output circuits to perform some I/O operations. Refer to Figures 6-17 and 6-18. For the first example, assume that you would like one of the LEDs to light when the corresponding switch is pushed. That is, LED_0 should light when S_0 is pushed; LED_1 should light when S_1 is pushed, etc.

If you refer to Figure 6-17, you will see that an LED is caused to light by placing a 0 in the proper bit in the latch. For example, a 0 in bit 0 will cause LED_0 to be forward biased. Thus, the diode will conduct and emit light. Notice that a 1 at bit 0 will not allow the diode to conduct and emit light. Consequently, a 0 turns the LED on and a 1 turns it off.

Refer to Figure 6-18, and you will find that, when one of the switches is closed, its corresponding line goes to 0. If the switch is not closed, its corresponding line is at 1.

If we load data into one of the accumulators from address 8000_{16} and then store the data at address 9000_{16}, the switches will appear to control the LED's. The program could look like this:

```
┌──► LDAA
│     80
│     00
│    STAA
│     90
│     00
│     BRA
└──── F8
```

If S_0, and only S_0, is closed when the LDAA 8000 instruction is executed, 11111110_2 will be loaded into accumulator A. The next instruction stores this data byte in the output latch. This causes LED_0, and only LED_0, to light. The BRA instruction holds the MPU in a tight loop. Try a few examples and verify that each time a switch is closed, the corresponding LED will light. If the switches are set to some 8-bit binary number, the LED's will display that 8-bit number.

Now, suppose we change our mind and decide that the LEDs should display the one's complement of the binary number set on the switches. We do not have to touch the hardware. Instead, we just change the program.The new program might look like this:

```
LDAA
 80
 00
COMA
STAA
 90
 00
 BRA
 F7
```

Notice that we have simply inserted the one's complement instruction between the input and output operations.

As another example, suppose we wish to display a number that is four times greater than the number set on the switches. Our program could be changed to this:

```
LDAA
 80
 00
ASLA
ASLA
STAA
 90
 00
 BRA
 F6
```

Once again, no hardware change is needed. We simply insert two ASLA instructions between the input and output operations.

Although these examples are very simple, they illustrate the flexibility of this I/O arrangement. Data is pulled from the input device as if it were being pulled from memory. Once in the MPU, the data byte can be modified in any way we like. The data can then be transferred to the output device as if it were being stored in memory. While the data is in the MPU, it can be modified in any number of ways. The input byte can be shifted left or right. It can be added to — or subtracted from — another number. It can be ANDed or ORed with another byte. The possibilities are endless and yet none of these involve a hardware change. All data manipulations can be accomplished by the program.

Program Control of I/O Operations

In the preceding examples, all I/O transfers are controlled by the program and the program alone. The program is in a tight loop that inputs data from the switches, modifies the data (if required), and outputs the data to the displays.

When this arrangement is used, the MPU never knows if the data at the input has changed. It simply reads in the data a number of times each second. By the same token, the MPU outputs the data over and over again. This system works well for simple I/O operations. However, as the I/O requirements become more sophisticated, this technique becomes cumbersome.

The program must be in a loop if it is to repeatedly check for inputs and refresh the output. As the number of data manipulations increase, the loop becomes longer and the MPU must check the inputs less frequently. When several I/O devices are used, it must check each input and refresh each output repeatedly. If the loop becomes too long, the MPU may miss a momentary switch closure. This may be acceptable in some applications but in many others it may be intolerable. Obviously then, a more sophisticated method of handling I/O operations must be available to the microcomputer.

Interrupt Control of I/O Operations

A more effective way of handling I/O operations involves a concept called *interrupts*. Interrupts are a means by which an I/O device can notify the MPU that it is ready to send input data or to accept output data. Generally, when an interrupt occurs, the MPU suspends its current operation and takes care of the interrupt. That is, it might read in or write out a byte of data. After it has taken care of the interrupt, the MPU returns to its original task and takes up where it left off.

An analogy may help you to visualize an interrupt operation. Compare the MPU to the president of a corporation who is writing a report. The interrupt can be compared to a telephone call. The president's main task is the report. However, if the telephone rings (an interrupt), she finishes writing the present word or sentence then answers the phone call. After she has attended to the phone call, she returns to the report and takes up where she left off. In this analogy, the ringing of the telephone notifies the president of the interrupt request.

This analogy shows the difficulty of the program controlled I/O technique discussed earlier. If we remove the interrupt request (the ringing of the phone), we are left with an almost comical situation. The president writes a few words of the report. She then picks up the phone to see if anyone is on the other end. If not, she hangs up the phone, writes a few more words, and checks the phone again. Clearly, this technique wastes an important resource — the president's time.

This simple analogy shows the importance of an interrupt capability. Without it, a great deal of the MPU's time can be wasted doing routine operations. The next section will examine the interrupt capabilities of the 6800 MPU.

Self-Test Review

21. What are the two methods by which microprocessors handle I/O operations?

22. Which method does the 6800 MPU use?

23. Which instruction can be used for transferring data from an I/O device to accumulator A?

24. Which instruction can be used for transferring data from accumulator B to an I/O device.

25. Write a program segment that will: read in data from the switch bank shown in Figure 6-18; double the number; and display the result on the LED bank shown in Figure 6-17.

26. What is meant by program control of an I/O operation?

27. What is meant by interrupt control of an I/O operation?

Answers

21. Some microprocessors have input-output instructions; others treat I/O as memory.

22. The 6800 MPU treats I/O as memory.

23. LDAA

24. STAB

25. One solution is:

```
LDAA
 80
 00
ASLA
STAA
 90
 00
```

26. Using this method, the program regularly reads in or writes out data. All I/O operations are controlled by the program.

27. Using this method, the I/O device itself signals the MPU that it is ready to transmit or receive data. The I/O operations are controlled largely by the I/O device itself.

INTERRUPTS

Interrupts were introduced in the previous section in connection with I/O operations. While I/O operations use part of the interrupt capability of the MPU, interrupts are also used in other ways. The 6800 MPU has four different types of interrupts:

Reset
Non-Maskable Interrupt (NMI)
Interrupt Request (IRQ)
Software Interrupt (SWI)

This section will examine each of these interrupts in detail.

Reset

In a typical application, the microcomputer has a control or monitor program in a read-only-memory (ROM). Also, a random access read-write memory (RAM) is used for holding input data, intermediate answers, output data, etc. As we have seen, the 6800 MPU has the capability of addressing up to $65{,}536_{10}$ memory locations. Most microprocessor applications do not require this much memory. In many applications, the control program requires less than ten percent of the possible locations. The RAM probably uses less than two percent. Generally, the monitor program is placed at the high memory addresses. The RAM is usually given the low memory addresses so that the direct addressing mode can be used. The I/O devices are given intermediate addresses. Thus, the memory addresses may be allocated as shown in Figure 6-19.

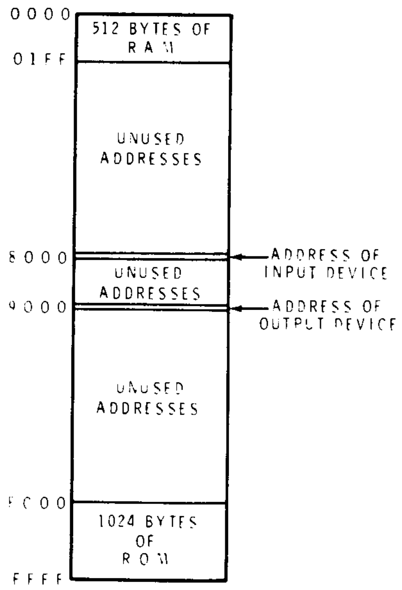


Figure 6-19.
Memory allocations in a typical microcomputer system.

Notice that the control or monitor program is placed in a ROM at the very top of memory. In this example, a 1024_{10} byte ROM is used. The addresses of the ROM are $FC00_{16}$ through $FFFF_{16}$. A small RAM is placed at the low end of memory. Addresses 0000_{16} through $01FF_{16}$ are used. Notice that all other addresses are unused except for two. The input device is assigned address 8000_{16}, while the output device is assigned address 9000_{16}.

The monitor program stored in the ROM, controls all the activities of the MPU. At all times, the entire system is being run by this program. In this example, when the microprocessor is initially turned on, it should start executing instructions at address $FC00_{16}$. Also, we should be able to restart the program at this address at any time. In order to accomplish this, the 6800 MPU has a built-in reset capability.

The 6800 MPU has a signal line or control pin that is called $\overline{\text{Reset}}$. This pin or line is connected to a reset switch of some kind. If this line goes low for a prescribed period of time (to be explained later) and then swings high, the MPU will initiate a *reset interrupt sequence*. The main purpose of the reset interrupt sequence is to load the address of the first instruction to be executed into the program counter. This would be easy to accomplish if, in every application, the starting address were the same. However, the starting address differs from one application to the next. Therefore, a convenient means is provided to allow the designer to specify any starting address that he likes.

In any 6800 based microprocessor system, the upper eight bytes of ROM are reserved for *interrupt vectors*. An interrupt vector is simply an address that is loaded into the program counter when an interrupt occurs. Figure 6-20 shows how these eight reserved memory bytes are allocated. Notice that addresses $FFFE_{16}$ and $FFFF_{16}$ contain the reset vector. That is, these two memory locations contain the address of the first instruction that is to be executed when the microcomputer is initially started. In our example, the first instruction in the monitor program is at address $FC00_{16}$. Consequently, this is our reset vector. Location $FFFE_{16}$ must contain the high byte of the address (FC_{16}) and $FFFF_{16}$ must contain the low byte of the address (00_{16}).

Remember locations $FFFE_{16}$ and $FFFF_{16}$ are in the read-only-memory. Therefore, the designer must provide the proper reset vector at the time he is writing the monitor program.

Address	
F F F 8	Interrupt Request Vector (high order address)
F F F 9	Interrupt Request Vector (low order address)
F F F A	Software Interrupt Vector (high order address)
F F F B	Software Interrupt Vector (low order address)
F F F C	Non-Maskable-Interrupt Vector (high order address)
F F F D	Non-Maskable-Interrupt Vector (low order address)
F F F E	Reset Vector (high order address)
F F F F	Reset Vector (low order address)

Figure 6-20.
Interrupt vector assignments.

Figure 6-21 shows the sequence of events that occurs when the MPU is reset. First, the interrupt (I) mask bit is set. You will recall that the I flag is one of the condition code registers. As you will see later, if this flag is set, it prevents one of the other interrupts from occurring. Thus, the MPU sets the interrupt mask bit so that the reset sequence will not be interrupted by a request for interrupt by one of the I/O devices.

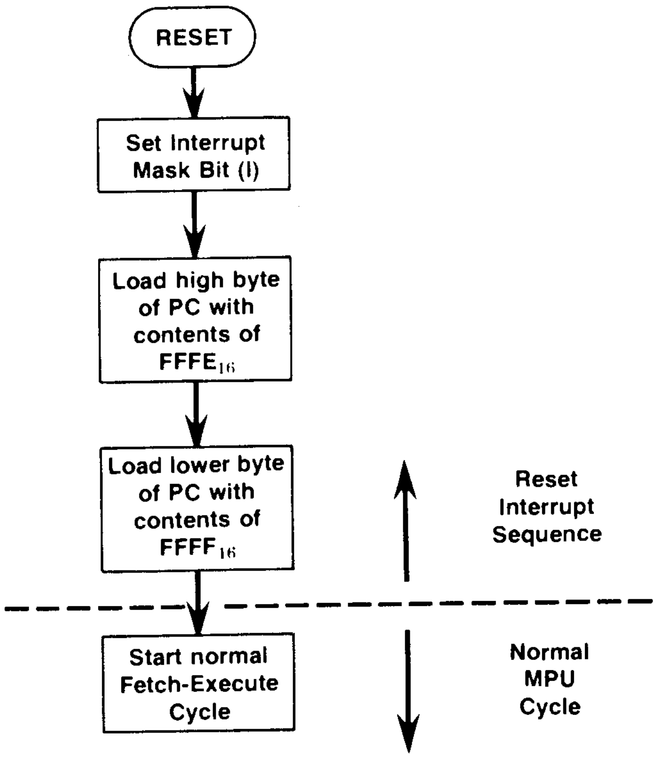

RESET
Set Interrupt Mask Bit (I)
Load high byte of PC with contents of FFFE16
Load lower byte of PC with contents of FFFF16
Reset Interrupt Sequence
Start normal Fetch-Execute Cycle
Normal MPU Cycle

Figure 6-21.
Reset interrupt sequence.

Second, the contents of location $FFFE_{16}$ are loaded into the high byte of the program counter. This is done by sending the address $FFFE_{16}$ out on the address bus. The memory location is read out and its contents are placed on the data bus. The MPU picks up this byte and places it in the upper eight bits of the program counter. In our example, the byte in location $FFFE_{16}$ is FC_{16}.

Next, the contents of location $FFFF_{16}$ are loaded into the lower eight bits of the program counter. This is done by setting the address bus to $FFFF_{16}$. Thus, the contents of the highest memory location are placed on the data bus. In our example, this byte is 00_{16}. At this point, the program counter contains the address of the first instruction which is $FC00_{16}$.

The reset sequence is then terminated by switching the MPU to its normal fetch-execute machine cycle. Thus, the instruction at address $FC00_{16}$ is fetched and executed. From this point on, all MPU activities are controlled by the program.

The microprocessor system will have a reset switch somewhere in the system. This will allow the operator to restart the system if the system locks up or runs away for some reason. In addition, some systems will have an automatic reset feature that will allow the system to reset itself after a power failure. In both cases, the reset capability of the MPU is used.

This reset capability can be considered an interrupt, since the MPU leaves whatever it is doing and jumps off to the start of the monitor program. In most cases, the monitor program would start with a short subroutine that initializes the system. It would do things like set up the stack pointer, initialize displays, etc.

Non-Maskable Interrupts

The 6800 has two other types of hardware interrupts. One of these interrupts is maskable; the other is not. A maskable interrupt is one that the MPU can ignore under certain conditions. Whereas, a non-maskable interrupt cannot be ignored. To illustrate the difference, recall the corporation president analogy.

The president's report writing can be interrupted by the telephone. However, by telling her secretary to hold all calls, she has effectively masked one source of interruptions. In this analogy it is impractical to mask all interrupts. For example, it could be counterproductive to mask the fire alarm.

Somewhat the same situation can exist in a microprocessor controlled system. Some interrupts can be ignored for a few seconds while the MPU is performing a more important task. This type of interrupt can be masked. Others must not be ignored at all. These cannot be masked. Of course, it is up to the designer to decide which interrupts can be masked and which cannot. The 6800 MPU has provisions for handling both types. How the MPU handles the non-maskable type will be discussed first.

The 6800 MPU has a control line called the non-maskable interrupt ($\overline{\text{NMI}}$) line. A high-to-low transition on this line forces the MPU to initiate a *non-maskable interrupt sequence.* The purpose of this sequence is to provide an orderly means by which the MPU can jump off to a service routine that will take care of the interrupt.

This becomes somewhat involved because the MPU must be able to go back to its main program after the interrupt service routine is finished. It must be able to pick up exactly where it left off. Furthermore, all registers must hold exactly the same data and addresses that they held when the interrupt occurred. In other words, when an interrupt occurs, the program count must be saved so that the MPU can later return to this point in the program. Also, the contents of the accumulators, index register, and even the condition code registers must be saved so that the MPU can be restored to the exact condition that existed at the instant the interrupt occurred.

The 6800 MPU accomplishes this by pushing all the pertinent data onto the stack. Then, after the interrupt has been serviced, the MPU returns to its previous status by pulling the data from the stack.

The non-maskable interrupt sequence is shown in Figure 6-22. A non-maskable interrupt is initiated when the $\overline{\text{NMI}}$ line goes from its high state to its low state. The MPU finishes the execution of the current instruction. However, before another instruction is fetched, the MPU pushes the contents of its registers onto the stack. Recall that the stack pointer always points to the top of the stack. For this example, assume that the stack pointer was set by an earlier instruction to address 0068_{16}.

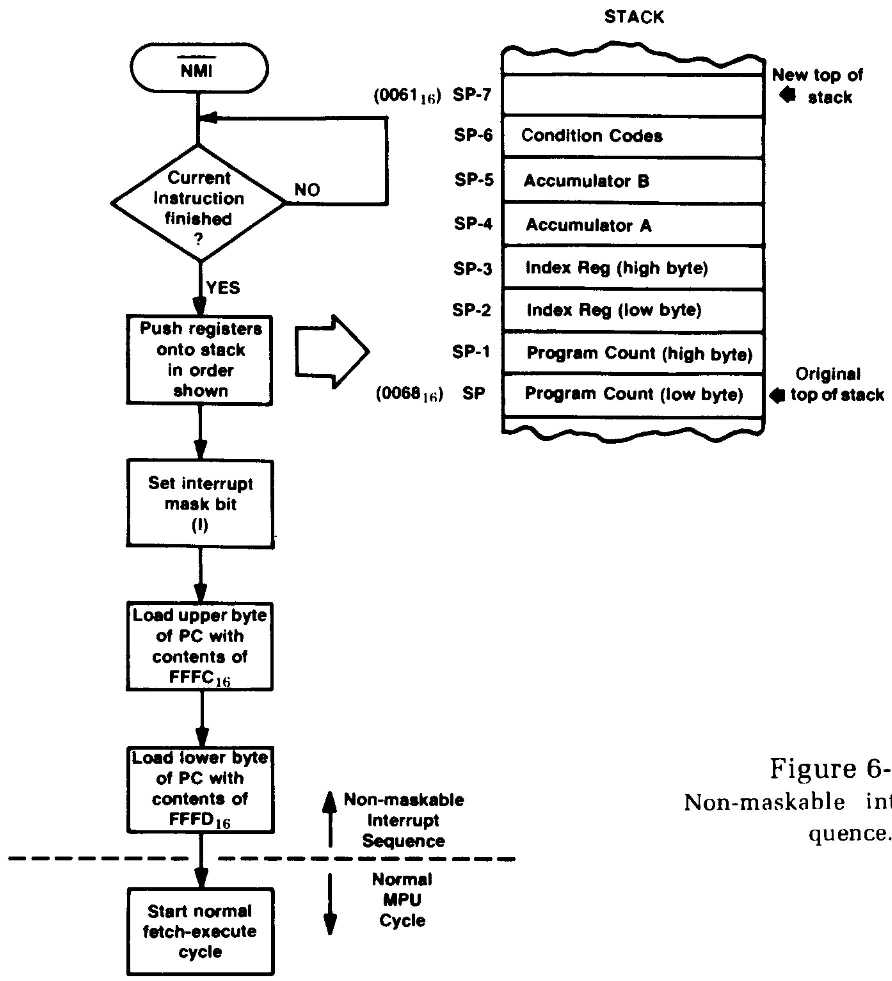

STACK
NMI
Current Instruction finished ?
NO
YES
Push registers onto stack in order shown
Set interrupt mask bit (I)
Load upper byte of PC with contents of FFFC16
Load lower byte of PC with contents of FFFD16
Start normal fetch-execute cycle
Non-maskable Interrupt Sequence
Normal MPU Cycle
(006116) SP-7
SP-6 Condition Codes
SP-5 Accumulator B
SP-4 Accumulator A
SP-3 Index Reg (high byte)
SP-2 Index Reg (low byte)
SP-1 Program Count (high byte)
(006816) SP Program Count (low byte)
New top of stack
Original top of stack

Figure 6-22.
Non-maskable interrupt sequence.

The MPU pushes the lower eight bits of the program counter into memory location 0068_{16}. Then it decrements the stack pointer so that the upper eight bits of the program counter are pushed into address 0067_{16}. Next, the contents of the index register are pushed into addresses 0066_{16} and 0065_{16}. The contents of accumulators A and B and the condition codes are also pushed in as shown. When all this has been done, the stack pointer will have been decremented seven times to 0061_{16}.

Return to the flow chart and notice that the next step is to set the interrupt mask bit. This allows the MPU to ignore any interrupt requests that occur while the non-maskable interrupt is being serviced.

At this point, the MPU is ready to jump to the interrupt service routine. But, what is the address of this routine? Recall the interrupt vector chart that was shown earlier in Figure 6-20. The non-maskable interrupt vector is at addresses FFFC_{16} and FFFD_{16}. Thus, the upper byte of the program counter is loaded from FFFC_{16} while the lower byte is loaded from FFFD_{16}. This directs the MPU to the first instruction in the non-maskable interrupt service routine. From this point on, the MPU returns to its normal fetch-execute cycle until the service routine is finished.

The sequence of events shown in Figure 6-22 happen automatically when a non-maskable interrupt sequence is initiated. The $\overline{\text{NMI}}$ line gives external hardware a method of forcing a jump-to-subroutine to occur. In this case, the subroutine is a short program that performs some action to take care of the interrupt.

Return From Interrupt (RTI) Instruction

The non-maskable interrupt is used when some situation exists that cannot be ignored. You can probably visualize applications that would require such a capability. For example, assume that a microprocessor is being used in a numerically controlled drill press. The non-maskable interrupt could be used in conjunction with limit switches to prevent drilling holes in the work surface. Or, it could be used to shut down the machine if someone's hand got too close.

The purpose of the service routine is to direct the operation of the computer to take care of the interrupt. Typically, it would first determine which external device initiated the interrupt. Then it would determine the nature of the interrupt. Finally, it would take whatever action was necessary to take care of the interrupt. In many cases, the interrupt is of a routine nature and can be easily serviced. In these situations, the MPU should return to the main program and take up where it left off. There is an instruction that allows the MPU to do this. It is called the "Return-From-Interrupt" (RTI) instruction. Look on your Instruction Set Summary card, and you will see that this is a one-byte instruction whose opcode is $3B_{16}$.

Figure 6-23 shows how the RTI instruction is used. The main program is shown on the left, while the interrupt service routine is shown on the right. Assume that the interrupt signal occurs while the LDAB# instruction is being executed. The MPU finishes that one instruction and pushes all pertinent data onto the stack. It then jumps to an address determined by the $\overline{\text{NMI}}$ vector in address FFFC and FFFD. The contents of these two locations determine the starting address of the $\overline{\text{NMI}}$ service routine. Notice that the last instruction in the service routine is the return-from-interrupt instruction. This instruction returns program control to the point in the main program that the MPU left when the interrupt occurred.

This can be done because the previous status of the MPU was preserved in the stack. The RTI instruction causes the accumulators, the index registers, the condition code register, and the program counter to be loaded from the stack. Thus, the same information that went into the stack when the interrupt occurred comes out of the stack when the RTI instruction is executed. This allows the MPU to return to the main program and take up where it left off.

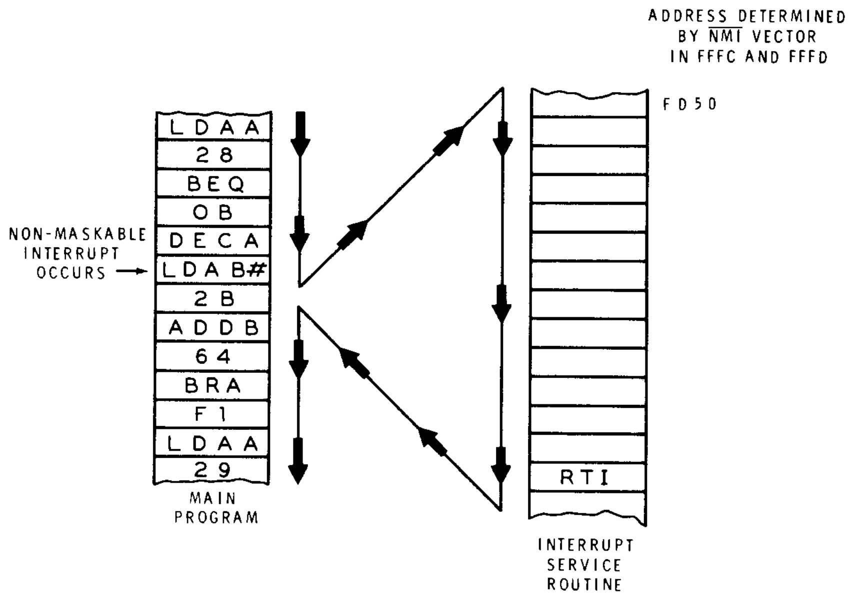


Figure 6-23
The RTI instruction returns control to the main program after the interrupt has been serviced.

Interrupt Request (IRQ)

The interrupt request is very similar to the non-maskable interrupt. The main difference between the two is that the interrupt request is maskable.

The 6800 MPU has a control line called the interrupt request ($\overline{\text{IRQ}}$) line. When this line is low, an interrupt sequence is requested. However, the MPU may or may not initiate the interrupt sequence depending on the state of the interrupt mask (I) bit in the condition code register. If the I bit is set, the MPU ignores the interrupt request. If the I bit is not set, the MPU initiates the interrupt sequence. This procedure is very similar to the $\overline{\text{NMI}}$ procedure discussed earlier. Figure 6-24 shows the interrupt procedure.

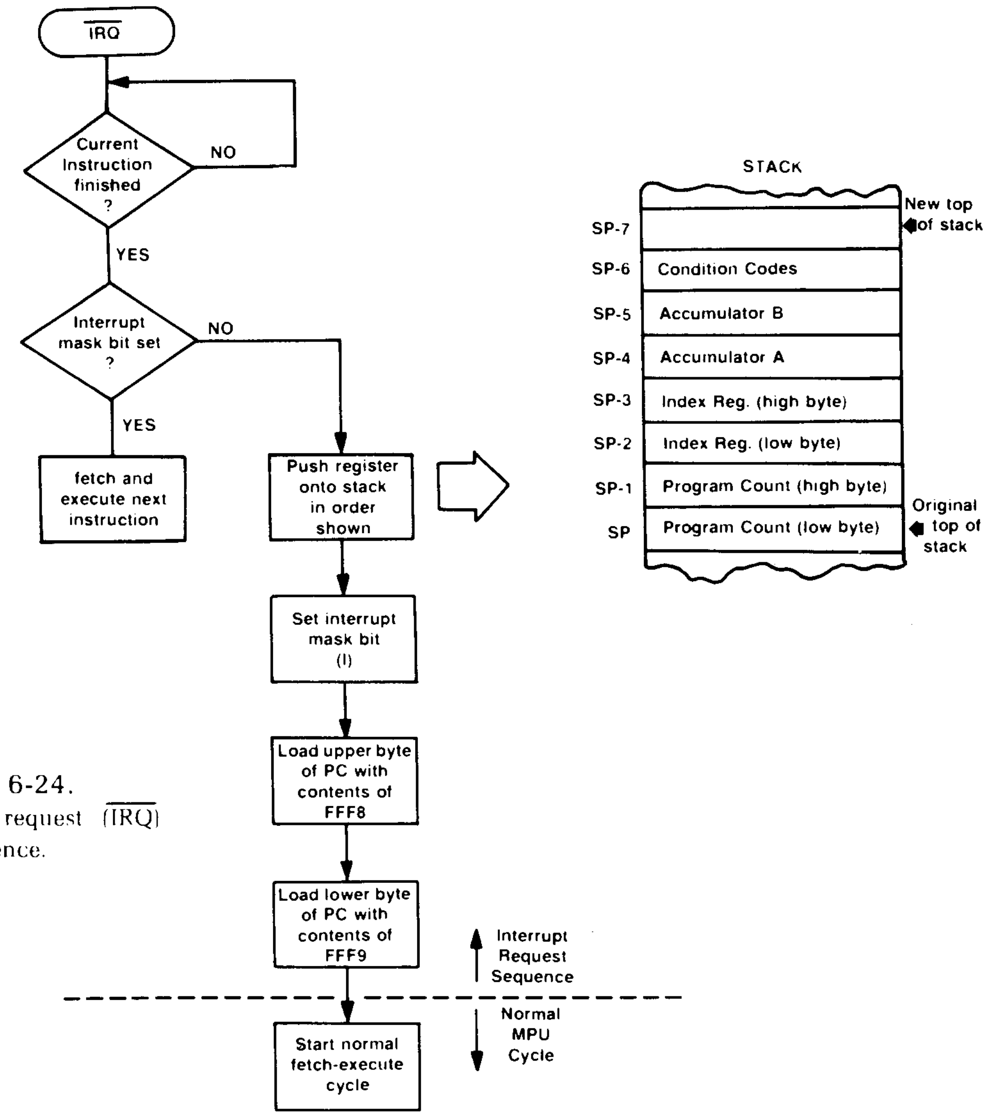

IRQ
Current Instruction finished ?
NO
YES
Interrupt mask bit set ?
NO
YES
fetch and execute next instruction
Push register onto stack in order shown
Set interrupt mask bit (I)
Load upper byte of PC with contents of FFF8
Load lower byte of PC with contents of FFF9
Start normal fetch-execute cycle
Interrupt Request Sequence
Normal MPU Cycle
STACK
New top of stack
SP-7
SP-6 Condition Codes
SP-5 Accumulator B
SP-4 Accumulator A
SP-3 Index Reg. (high byte)
SP-2 Index Reg. (low byte)
SP-1 Program Count (high byte)
SP Program Count (low byte)
Original top of stack

Figure 6-24.
The interrupt request ($\overline{\text{IRQ}}$) sequence.

When the $\overline{\text{IRQ}}$ line is low, the MPU finishes the current instruction. It then checks the interrupt mask bit. If I is set to 1, the MPU ignores the interrupt request and executes the next instruction in sequence. However, if I=0, the MPU pushes the contents of the various registers onto the stack in the order shown.

Next, the interrupt mask bit is set to 1. This prevents the MPU from honoring other interrupt requests until the present interrupt has been serviced.

The address of the $\overline{\text{IRQ}}$ service routine is at addresses $FFF8_{16}$ and $FFF9_{16}$. The program counter is loaded from these addresses. Thus, the next instruction to be executed will be the first instruction in the interrupt request service routine.

Once in the service routine, the MPU goes into its normal fetch-execute cycle. When the interrupt has been serviced, control can be returned to the main program by an RTI instruction.

Interrupt Mask Instructions

The 6800 MPU has two instructions that allow software control of the interrupt mask bit. You have seen that the I bit in the condition code register is set any time an interrupt sequence is initiated. This prevents an $\overline{\text{IRQ}}$ from being honored while a previous $\overline{\text{IRQ}}$ or $\overline{\text{NMI}}$ is being serviced. This is an example of setting the interrupt flag with *hardware*.

In many cases, it is necessary to set the interrupt flag with software. Therefore, the 6800 MPU has an instruction that can do this. It is called the "Set-Interrupt-Mask" (SEI) instruction. If you refer to your Instruction Set Summary card, you will see that this is a one-byte instruction whose opcode is $0F_{16}$. The flag may be set to prevent an interruption on a part of the program that we do not wish to be interrupted. It has the effect of disabling interrupt requests.

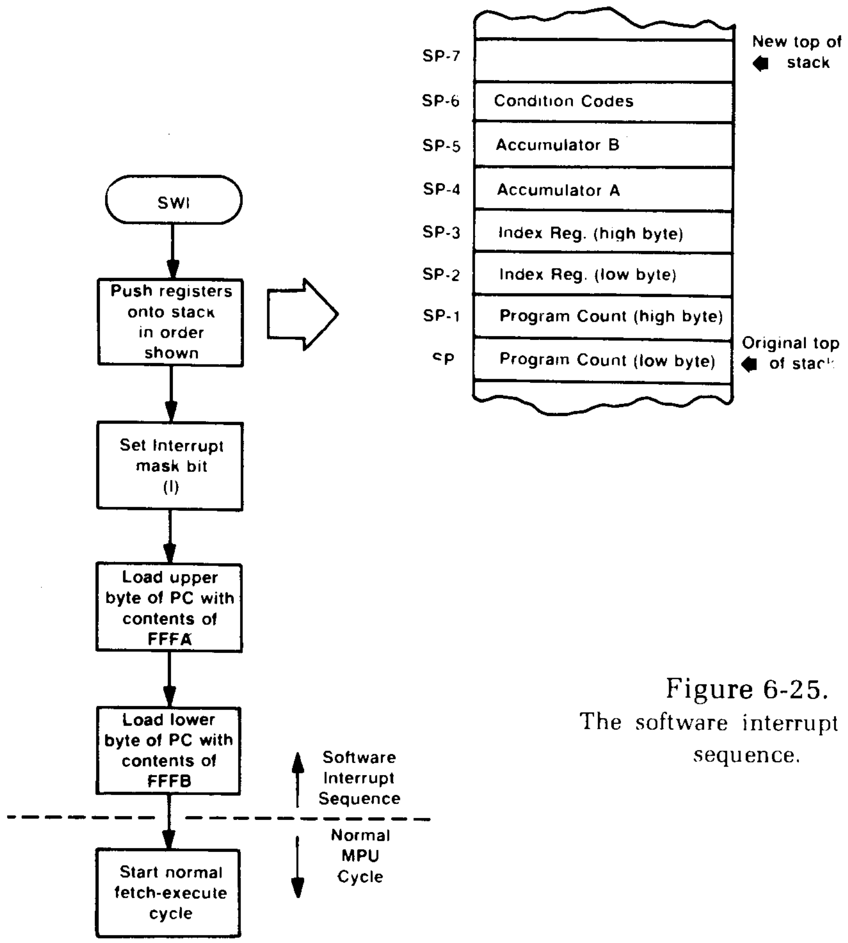
Of course, we do not wish to permanently disable the interrupt capability. Therefore, some means must be provided for enabling the interrupt request capability. An instruction called "Clear-Interrupt-Mask" (CLI) is available for this purpose. This is a one-byte instruction whose opcode is $0E_{16}$.

While we can disable or enable the interrupt request line with these instructions, they do not affect the non-maskable interrupt. As the name implies, the $\overline{\text{NMI}}$ line cannot be disabled by the I flag.

Software Interrupt (SWI) Instruction

The 6800 MPU has a software equivalent of an interrupt. It is an instruction called the "Software Interrupt" (SWI). When executed, the instruction causes the MPU to perform an interrupt sequence that is very similar to the hardware interrupt sequences already discussed. As shown on your Instruction Set Summary card, this is a one-byte instruction whose opcode is $3F_{16}$.

Figure 6-25 shows the sequence of events that occurs when this instruction is executed. First the contents of all the pertinent registers are pushed onto the stack in the order shown. Next, the interrupt mask is set so that interrupt requests cannot interfere. Finally, the software interrupt vector is obtained from addresses $FFFA_{16}$ and $FFFB_{16}$. This vector is loaded into the program counter so that the next instruction will be fetched from this address. As with the other interrupts, the MPU will return to the original program when a return-from-interrupt instruction is encountered.



SWI
Push registers onto stack in order shown
Set Interrupt mask bit (I)
Load upper byte of PC with contents of FFFA
Load lower byte of PC with contents of FFFB
Start normal fetch-execute cycle
Software Interrupt Sequence
Normal MPU Cycle
SP-7
SP-6 Condition Codes
SP-5 Accumulator B
SP-4 Accumulator A
SP-3 Index Reg. (high byte)
SP-2 Index Reg. (low byte)
SP-1 Program Count (high byte)
SP Program Count (low byte)
New top of stack
Original top of stack

Figure 6-25.
The software interrupt (SWI) sequence.

The software interrupt instruction can be used to simulate hardware interrupts. It is also helpful for inserting pauses in a program. For example, the ET-3400 Microprocessor Trainer uses the software interrupt to perform the single-step function and to implement the breakpoint capability.

Wait for Interrupt (WAI) Instruction

One of the first instructions introduced in this course was the halt instruction (opcode $3E_{16}$). In the previous unit, you learned that this instruction is actually called a Wait-for-Interrupt (WAI). What exactly does this instruction do? It does cause the MPU to halt, but there is more to it than that.

When the WAI instruction is executed, the program counter is incremented by one. Then the contents of the program counter, index register, accumulators, and condition code register are pushed onto the stack. The order is exactly the same as if an interrupt occurs. The MPU then enters a wait state, doing nothing further until, and unless, an interrupt occurs.

The MPU can be forced back into action either by an interrupt request or by a non-maskable interrupt. The $\overline{\text{NMI}}$ sequence is the same as that described earlier except for one important difference. Remember that the contents of the registers have already been pushed onto the stack. Thus, this part of the $\overline{\text{NMI}}$ sequence is omitted. This allows the MPU to respond faster to the interrupt.

The $\overline{\text{IRQ}}$ sequence is also the same as that described earlier except that the registers are not pushed onto the stack again. As always, the $\overline{\text{IRQ}}$ signal is ignored if the interrupt mask bit is set.

Of course, the reset signal can override the wait state. Thus, there are three ways of escaping the wait state.

Self-Test Review

28. List the four types of interrupts available to the MPU.

29. Which interrupt is ignored by the MPU if the interrupt mask bit is set?

30. What is the purpose of the reset interrupt sequence?

31. From what addresses is the reset interrupt vector taken?

32. What is an interrupt vector?

33. List the sequence of events that takes place when a non-maskable interrupt occurs.

34. What is an interrupt service routine?

35. What is usually the last instruction in the interrupt service routine?

36. How does the RTI instruction affect the stack pointer?

37. Which of the interrupts does not cause data to be pushed into the stack?

38. Which instruction can be used to disable the interrupt request capability?

39. List three methods by which the MPU can be released from the wait state following the execution of a WAI instruction.

40. Under what condition will the $\overline{\text{IRQ}}$ and $\overline{\text{NMI}}$ interrupts not cause data to be pushed into the stack?

Answers

28. Reset, non-maskable interrupt, interrupt request, and software interrupt.

29. Interrupt request ($\overline{IRQ}$).

30. To direct the MPU to the first instruction in the monitor or control program.

31. $FFFE_{16}$ and $FFFF_{16}$.

32. The address of the interrupt service routine.

33. A. The current instruction is executed.
    B. The contents of the pertinent registers are pushed onto the stack.
    C. The interrupt mask bit is set.
    D. The $\overline{NMI}$ vector from addresses $FFFC_{16}$ and $FFFD_{16}$ is loaded into the program counter.
    E. The instruction at the address specified by the $\overline{NMI}$ vector is fetched and executed.

34. A routine that takes care of the interrupt and then returns control to the main program.

35. The Return-From-Interrupt (RTI) instruction.

36. The stack pointer is incremented seven times as the previous MPU status is pulled from the stack.

37. Reset.

38. Set Interrupt Mask (SEI).

39. A. By a reset signal.
    B. By a non-maskable interrupt.
    C. By an interrupt request (if I=0).

40. If the MPU is waiting for an interrupt.

UNIT 6 SUMMARY

1. In computer jargon, a stack is a group of temporary storage locations in which data can be stored and later retrieved.
2. A cascade stack is a group of registers devoted to the storage of data.
3. A memory stack is an area of memory reserved for stack operations.
4. Data is pushed into and pulled from a stack in a last-in/first-out basis.
5. When the amount of data pushed into a cascade stack exceeds the capacity of the stack, the first data pushed is lost.
6. A memory stack has no fixed length. However, if too much data is pushed into the stack, it can overwrite program code or data.
7. An advantage of a stack transfer operation over a memory transfer operation is that the instruction to push or pull data in a stack does not require an address value.
8. More than one stack can be setup in memory.
9. The Stack Pointer is a 16-bit register in the 6800 MPU that is used to identify the location of the top of the stack in memory.
10. The push instructions are used to move the contents of either accumulator into the stack.
11. The pull instructions are used to move the data at the top of the stack into either accumulator.
12. A subroutine is a group of instructions that perform some limited but frequently required task.
13. The jump instruction is similar to the branch always instruction only it uses the indexed or the extended addressing mode of operation.

14. Jumping to a subroutine is often referred to as calling a subroutine.

15. The jump instruction does not tell the MPU to save a return address.

16. The jump to subroutine instruction tells the MPU to save the address of the next sequential instruction in the stack before it loads the Program Counter with the address of the subroutine.

17. The return from subroutine instruction tells the MPU to pull the address located at the top of the stack, place it in the Program Counter, and execute the instruction found at that address.

18. Subroutine nesting occurs when a called subroutine calls another subroutine. Many levels of nesting can exist in a program.

19. A branch to subroutine instruction is the same as a jump to subroutine instruction except that a single-byte relative address is used instead of an indexed or extended address.

20. To be useful, a microprocessor system must be able to interface with the "outside world."

21. Some microprocessors use specific I/O instructions to handle I/O operations. The 6800 has no I/O instructions. Rather, each I/O interface is assigned a memory location, and all I/O operations are treated as memory operations.

22. The I/O interface generally has a latch on its output port to allow the slower peripheral devices sufficient time to respond to the data. The input port generally uses a simple buffer to isolate the peripheral from the MPU data bus.

23. To input data from an I/O port, use the indexed or extended forms of addressing with either load accumulator instruction. To store data, use the indexed or extended forms of addressing with either store accumulator instruction.

24. A peripheral can notify the MPU that it needs servicing with an interrupt signal.

25. There are four types of interrupts: reset, non-maskable interrupt, interrupt request, and software interrupt.

26. Every type of interrupt has a related interrupt vector address. Each vector address identifies two sequential memory locations that contain the address of the routine, or program, that processes the interrupt. The interrupt vectors occupy the last eight memory locations.

27. The monitor program in ROM controls all MPU/system related activities. These control programs are addressed by the MPU immediately after a reset interrupt.

28. When power is applied to the 6800 MPU and its support circuitry, the MPU $\overline{\text{RESET}}$ line is momentarily pulled low. This operation triggers a sequence of events within the MPU that end with the interrupt vector address for the reset interrupt being loaded into the Program Counter. The interrupt vector points to the last two bytes in the ROM that contain the address of the system start-up routine. This address is loaded into the Program Counter and the start-up routine is executed.

29. A system reset operation can be initiated at any time by pulling the MPU $\overline{\text{RESET}}$ line low.

30. The first operation the MPU performs when it processes a reset interrupt is set the interrupt mask bit in the condition code register.

31. A non-maskable interrupt is a hardware interrupt that the MPU cannot ignore. A high-to-low transition on the $\overline{\text{NMI}}$ control line forces the MPU to initiate a non-maskable interrupt sequence. Before the sequence begins, the MPU is allowed to finish the current instruction. Then, the MPU saves all of its registers (except the Stack Pointer) in the stack. Next, it sets the interrupt mask bit in the condition code register. Finally, it gets the address of the service routine.

32. A maskable interrupt is a hardware interrupt that the MPU can ignore. A low on the $\overline{\text{IRQ}}$ line indicates that an interrupt is being requested. It is serviced in the same manner as a non-maskable interrupt; except that before the interrupt is serviced, the condition code register is checked to make sure the interrupt mask bit is clear.

33. The end of an interrupt service routine is indicated by the return from interrupt instruction. When the instruction is executed, all of the register values that were saved in the stack are returned to the MPU, and program execution continues from were it was originally halted.

34. Two interrupt mask instructions are used to set and clear the interrupt mask bit in the condition code register. When the mask bit is set, all external maskable interrupt requests are ignored.

35. A software interrupt is a program initiated interrupt request. As soon as the instruction is executed the interrupt is serviced in the same manner as a non-maskable interrupt.

36. When the wait for interrupt instruction is executed, the Program Counter is incremented by one. Then, the contents of the Program Counter, Index register, condition code register, and accumulators are pushed into the stack. Finally, the MPU goes into a wait state. In this state, the MPU will only respond to a hardware interrupt of some form.

Perform Experiments 9 and 10 at the end of Book 2.

PROGRAMMING EXPERIMENTS

Experiment 1

Binary/Decimal Training Program

OBJECTIVES:

To improve your ability to convert binary numbers to their decimal equivalent.

To improve your ability to convert decimal numbers to their binary equivalent.

To present the proper procedure for entering a program into the Microprocessor Trainer.

To demonstrate the versatility of the Microprocessor Trainer and microprocessors in general.

INTRODUCTION

In Unit 1, you were introduced to the binary number system. As you proceed through this course, you will find the need to convert binary numbers to decimal, and decimal numbers to binary. To improve your ability to make these conversions, you will enter a program into the Microprocessor Trainer to allow it to act as your instructor. In the first half of this experiment, you will use the Trainer to practice binary-to-decimal conversion.

When you use the Trainer, carefully follow all of the operating instructions. A microprocessor can only perform properly if it is programmed properly. However, you do not need programming experience at this time; just follow the instructions provided in this experiment. Do not worry about what you are entering.

The Trainer Manual contains a great amount of useful information in the Operation Section. You should review that section before you proceed with this experiment.

If your Trainer has been modified for use with the Heathkit Memory I/O Accessory, unplug the Trainer from the AC wall receptacle. Disconnect the 40-pin plug that connects the Trainer to the Memory I/O Accessory.

If your Trainer is Model number ET-3400, reinstall the 2112 RAM IC's at IC-14 through IC-17 before starting the experiments in this unit.

If your Trainer is Model number ET-3400A, reinstall the 2114 RAM IC's at IC-14 and IC-15 before starting the experiments in this unit.

PROCEDURE

1. Plug in the Trainer and push the POWER switch on. Then momentarily press the RESET key. The display should show CPU UP.

2. Push the AUTO (automatic) key. Displays H, I, N, and Z will show "prompt" characters (bottom segment of each digit illuminated), and displays V and C will show Ad. NOTE: The letters identifying each display are located near their bottom right corners.

3. Push the 0 key three times. 0's will appear in displays H, I; and N.

4. Push, but do not release the 0 key. A 0 will appear in display Z. Now release the key. The 0 will not change, but displays V and C will now show prompt characters.

   NOTE: The Trainer is now ready to receive program data. If you make a data error while entering the program, do not attempt to correct the error; continue programming. Any errors will be located and corrected when you examine your program.

ADDRESS	INST/DATA	ADDRESS	INST/DATA	ADDRESS	INST/DATA
0000	00*	0029	08	0052	00
0001	00*	002A	08	0053	3B
0002	BD	002B	00	0054	4F
0003	FC	002C	00	0055	DB
0004	BC	002D	00	0056	BD
0005	BD	002E	80	0057	00
0006	FE	002F	BD	0058	69
0007	52	0030	FC	0059	7E
0008	5E	0031	BC	005A	00
0009	FE	0032	BD	005B	02
000A	7C	0033	FE	005C	BD
000B	00	0034	09	005D	FE
000C	01	0035	97	005E	52
000D	B6	0036	00	005F	00
000E	CO	0037	BD	0060	00
000F	03	0038	00	0061	15
0010	01	0039	69	0062	9D
0011	46	003A	5F	0063	BD
0012	25	003B	84	0064	00
0013	F6	003C	FO	0065	69
0014	CE	003D	27	0066	7E
0015	00	003E	07	0067	00
0016	00	003F	80	0068	14
0017	DF	0040	10	0069	CE
0018	F2	0041	CB	006A	00
0019	BD	0042	OA	006B	00
001A	FD	0043	4D	006C	09
001B	93	0044	26	006D	26
001C	B6	0045	F9	006E	FD
001D	CO	0046	96	006F	96
001E	06	0047	00	0070	01
001F	01	0048	84	0071	84
0020	46	0049	0F	0072	3F
0021	25	004A	1B	0073	97
0022	F9	004B	90	0074	01
0023	BD	004C	01	0075	96
0024	FC	004D	26	0076	00
0025	BC	004E	OD	0077	39
0026	BD	004F	BD		RESET
0027	FE	0050	FE		
0028	52	0051	52		

*This data may change randomly.

Figure 1
Binary-to-decimal training program.

5. Using the Trainer keys, enter the Binary-to-Decimal training program shown in Figure 1. At each address specified, press the appropriate inst/data (program instruction or data) number keys (most significant number first). Displays V and C will show the inst/data word you have entered. Note that as you release the second data key, address displays H, I, N, and Z will increment (count up one), and displays V and C will again show prompt characters. When you get to the end of the program, press the RESET key as indicated.

6. Now that you have entered the Binary-to-Decimal training program, you must examine the data for errors. Use the following sequence to examine the data and correct any errors.

   A. Press the EXAM (examine) key. Note that the display is now asking for a 4-digit address (_ _ _ _ Ad.)

   B. Enter the beginning address of the program (0000). As soon as the last address digit is entered, displays V and C show the contents of that memory location. NOTE: The address is a memory location in the Trainer.

   C. Now compare the displayed address and data with the address and inst/data columns in the program.

   D. If the displayed data is incorrect, press the CHAN (change) key. The data displays will now show prompt characters. Enter the correct data.

   E. Press the FWD (forward) key. The address will increment and the data for that memory location will be shown. Correct the data if necessary.

   F. Continue to step through the program with the FWD key, and correct data as necessary, until you reach the end of the program. It is not necessary to examine or modify the memory beyond address 0077 since it will have no effect on the program.

7. Press the RESET key.

8. Press the DO key, then enter address 0002. The display should show GO. If the display shows a different number or word, or goes blank, your program contains an error. Repeat steps 6 through 8.

9. Press the F key. A 6-bit binary number should appear in the display. This is a random number and should change in value when you are told to "GO" next time.

10. Examine the binary number and determine its decimal value. Then press the D key. Two prompt characters should appear in the display.

11. Enter the decimal value of the binary number previously displayed (most significant digit first.) For values less than 10, enter a 0 before you enter the value. After approximately one second, the Trainer will indicate whether or not your answer is correct.

12. If your answer was correct, the Trainer will display YES. After approximately one second, the word GO will replace the decimal number.

    If your answer was incorrect, the Trainer will display NO. After approximately one second, the same binary number will again be displayed. Determine and enter the decimal value as described in steps 10 and 11.

13. Refer again to steps 9 through 12 and practice converting binary numbers to their decimal equivalent. You should obtain 10 correct answers in succession before you continue with this experiment.

DISCUSSION

Now that you have used the Trainer and its microprocessor, you have accomplished three objectives. First, you are becoming proficient in binary-to-decimal conversion. Second, you have been introduced to the correct method for entering, examining, and modifying a program. Third, you have been shown how a simple set of instructions can produce a powerful training aid. However, you should remember, a microprocessor can only perform what you tell it. One incorrect instruction can produce totally unexpected results.

Now, reprogram the Trainer for decimal-to-binary instruction. Since you will be using the same memory locations used in the first half of this experiment, the Binary-To-Decimal program will disappear.

PROCEDURE (Continued)

14. Press the RESET key.

15. Press the AUTO key, and enter address 0000.

16. Using the Trainer keys, enter the Decimal-to-Binary training program shown in Figure 2.

17. Now that you have entered the Decimal-to-Binary program, press the EXAM key and enter address 0000.

18. Using the FWD key, compare the Trainer memory contents with the program address and inst/data listing. If you must correct any data, press the CHAN key and enter the proper data.

19. After you have checked the program, press the RESET key.

20. Press the DO key, then enter address 0001. The display should show GO. If the display shows a different number or word, or goes blank, your program contains an error. Repeat steps 17 through 20.

21. Press the F key. A 2-digit decimal number should appear in the display, next to the word GO. This is a random number and should change in value when you are told to "GO" next time.

22. Examine the decimal number and determine its binary value. Then press the D key. Six prompt characters should appear in the display.

23. Enter the binary value of the decimal number previously displayed, beginning with the most significant bit (MSB). If the decimal value is less than 32, be sure to enter any leading zeros. NOTE: Although the program will accept any number combination, you should use only 1's and 0's.

ADDRESS	INST/DATA	ADDRESS	INST/DATA	ADDRESS	INST/DATA
0000	00*	002F	85	005E	00
0001	CE	0030	BD	005F	7E
0002	C1	0031	FD	0060	CE
0003	6F	0032	25	0061	C1
0004	BD	0033	4F	0062	3F
0005	FE	0034	E6	0063	BD
0006	50	0035	00	0064	FE
0007	5E	0036	C5	0065	50
0008	FE	0037	10	0066	00
0009	96	0038	27	0067	00
000A	00	0039	03	0068	80
000B	8B	003A	AB	0069	7E
000C	01	003B	03	006A	00
000D	19	003C	19	006B	01
000E	81	003D	56	006C	BD
000F	63	003E	24	006D	FE
0010	23	003F	03	006E	52
0011	01	0040	AB	006F	15
0012	4F	0041	06	0070	1D
0013	97	0042	19	0071	00
0014	00	0043	08	0072	00
0015	B6	0044	8C	0073	00
0016	CO	0045	00	0074	80
0017	03	0046	88	0075	BD
0018	01	0047	26	0076	00
0019	46	0048	EB	0077	7E
001A	25	0049	BD	0078	BD
001B	ED	004A	00	0079	FC
001C	96	004B	7E	007A	BC
001D	00	004C	BD	007B	7E
001E	BD	004D	FC	007C	00
001F	FE	004E	BC	007D	1C
0020	20	004F	D6	007E	CE
0021	B6	0050	00	007F	00
0022	CO	0051	11	0080	00
0023	06	0052	26	0081	09
0024	01	0053	18	0082	26
0025	46	0054	BD	0083	FD
0026	25	0055	FE	0084	39
0027	F9	0056	52	0085	00
0028	BD	0057	00	0086	00
0029	FC	0058	00	0087	00
002A	BC	0059	00	0088	32
002B	C6	005A	3B	0089	08
002C	03	005B	4F	008A	02
002D	CE	005C	DB	008B	16
002E	00	005D	BD	008C	04
				008D	01
					RESET

*This data may change randomly

Figure 2
Decimal-to-binary training program.

24. If your answer was correct, the Trainer will display YES approximately one second after you enter the last binary bit. Approximately one second later, the Trainer will display GO.

    If your answer was incorrect, the Trainer will display NO approximately one second after you enter the last binary bit. Approximately one second later, the same decimal number will be displayed again. Determine and enter the binary value as described in steps 22 and 23.

25. Refer again to steps 21 through 24 and practice converting decimal numbers to their binary equivalent. You should obtain 10 correct answers in succession before you continue with this experiment.

DISCUSSION

In this half of the experiment, you were given further experience in programming with the ET-3400 Microprocessor Trainer. You also improved your ability to readily translate decimal numbers into binary. This ability will become very useful as you progress through the Microprocessor Course.

Experiment 2

Hexadecimal/Decimal Training Program

OBJECTIVES:

To practice the conversion of decimal numbers to their hexadecimal equivalent.

To practice the conversion of hexadecimal numbers to their decimal equivalent.

INTRODUCTION

Binary numbers are used in all microprocessors to represent data and instructions. But binary numbers are difficult to work with.... especially when the number contains 8_{10}bits or more. To simplify programming, microprocessor designers usually use other number systems, like octal or hexadecimal, to represent binary data. Both octal and hexadecimal are just shorthand notations of binary numbers. Although the numbers are entered in hexadecimal or octal, the microprocessor "sees" them as binary. This simplifies programming.

For example, the binary number 10011111_2 requires eight key closures for entry. Fortunately, this same number can be represented in hexadecimal as $9F_{16}$ and requires only two key closures for entry. Fewer key closures means less programming errors and more efficient programming.

Your Microprocessor Trainer is based on the hexadecimal number system. You probably noticed this when you loaded the programs in the previous experiment; all instructions were coded in hexadecimal. The Microprocessor Trainer normally displays data in hexadecimal form. Of course, special programs allow the Trainer to accept binary or decimal numbers, as you saw in the first experiment. However, these special programs waste a portion of the microprocessors potential power and aren't necessary because you can make the conversion from decimal to hexadecimal with a little practice. That's the purpose of this experiment. . . to sharpen your conversion skills.

Again, you will use the Microprocessor Trainer for this purpose. First, you'll enter a program that allows you to practice conversion from decimal to hexadecimal. Then you'll load the second program that reverses the process. You'll find that it's not as difficult as it might appear.

Now briefly review decimal-to-hexadecimal conversion. Initially, it's helpful to make up a chart of decimal numbers and their hexadecimal equivalents, as shown here.

DECIMAL	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
HEXADECIMAL	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

Recall that hexadecimal is a base 16_{10} number system. Both systems use identical numbers from 0 through 9. However, at decimal number 10, the hexadecimal system shifts to characters of the alphabet, as shown by the letters A through F. Conversion of a decimal number to it's hexadecimal equivalent is a simple process where the decimal number is repeatedly divided by 16_{10}, with the remainder producing the equivalent hexadecimal number. This example will use only 2-digit numbers, since that's what you'll be converting in this experiment.

Suppose you want to convert 92_{10} to hexadecimal. The first step is to divide 92_{10} by 16_{10} as shown below.

$$\begin{array}{rl} & 5 \\ 16 & \overline{)92} \\ & \underline{-80} \\ \text{Remainder} & 12_{10} = C_{16} \leftarrow \text{LSD} \end{array}$$

The quotient is 5, but remember, we aren't concerned with this at the moment. We're interested in the remainder, in this case 12_{10}, because it forms the LSD of the equivalent hexadecimal number. Now, refer to the chart and find that $12_{10} = C_{16}$ and write this down as the LSD of the hex equivalent. The next step is to take the quotient of the previous division, in this case 5_{10}, and divide it by 16_{10}, as shown below.

$$\begin{array}{rl} & 0 \\ 16 & \overline{)5} \\ & \underline{-0} \\ \text{Remainder} \rightarrow & 5_{10} = 5_{16} \leftarrow \text{MSD} \end{array}$$

Of course, the quotient of this division is 0, signifying that the remainder, 5_{10}, is the MSD of the hexadecimal number. Checking the chart, you find that $5_{10} = 5_{16}$. Combining the MSD (5_{16}) and LSD (C_{16}), you find that the hex equivalent of 92_{10} is $5C_{16}$. You'll find that, after you've made the conversion a few times, you'll be able to do them in your head. You'll get that practice in this experiment.

PROCEDURE

1. Turn on the Trainer and press the RESET key.

2. Press AUTO and then enter address 0000.

3. Now enter the Decimal-to-Hexadecimal training program, shown in Figure 3, into the Trainer. When you've entered the last program instruction press the RESET key as shown at the end of the program.

ADDRESS	INST/DATA	ADDRESS	INST/DATA	ADDRESS	INST/DATA	ADDRESS	INST/DATA
0000	00*	0024	BD	0048	52	006C	00
0001	CE	0025	FE	0049	00	006D	00
0002	C1	0026	52	004A	3B	006E	00
0003	6F	0027	08	004B	4F	006F	00
0004	BD	0028	08	004C	DB	0070	00
0005	FE	0029	00	004D	BD	0071	80
0006	50	002A	00	004E	00	0072	39
0007	5E	002B	00	004F	63		RESET
0008	FE	002C	80	0050	7E		
0009	96	002D	BD	0051	00		
000A	00	002E	FC	0052	01		
000B	8B	002F	BC	0053	BD		
000C	01	0030	BD	0054	FE		
000D	19	0031	FE	0055	52		
000E	97	0032	09	0056	00		
000F	00	0033	36	0057	00		
0010	B6	0034	4F	0058	15		
0011	CO	0035	D6	0059	9D		
0012	03	0036	00	005A	BD		
0013	46	0037	CO	005B	00		
0014	25	0038	10	005C	63		
0015	F3	0039	25	005D	BD		
0016	96	003A	04	005E	FC		
0017	00	003B	8B	005F	BC		
0018	BD	003C	OA	0060	7E		
0019	FE	003D	20	0061	00		
001A	20	003E	F8	0062	16		
001B	B6	003F	CB	0063	CE		
001C	CO	0040	10	0064	00		
001D	06	0041	1B	0065	00		
001E	46	0042	33	0066	09		
001F	25	0043	11	0067	26		
0020	FA	0044	26	0068	FD		
0021	BD	0045	OD	0069	BD		
0022	FC	0046	BD	006A	FD		
0023	BC	0047	FE	006B	8D		

* This data may change randomly

Figure 3
Decimal-to-hexadecimal training program.

4. Check the stored program by first pressing the EXAM key and then entering address 0000. Now use the FWD key to step through the program, comparing the contents of memory with the program in Figure 3. Remember, the four left-most digits of the display represent the memory address and the two digits at the right are the contents of memory that should correspond with the INST/DATA listing of the program. If you find a mistake, correct it by first pressing the CHAN key and then entering the proper data.

5. When you're satisfied that the program is correct, press the RESET key.

6. Now it's time to execute the program. Do this by pressing the DO key and then entering address 0001. The word "GO" should now appear in the two left-most digits. If the display is blank, or if other numbers or letters appear, there is an error in the program and steps 4 and 5 should be repeated.

7. Now press the F key. A 2-digit "decimal" number will appear on the display. The Trainer is asking you to convert this decimal number to its hexadecimal equivalent. Therefore, examine the decimal number and then convert it to hexadecimal.

8. Enter your answer by first pressing key D. Two prompt characters will appear in the left-most digits. Now enter your hexadecimal number.

   If you respond correctly, the Trainer will display "YES" for a short period and then give you another "GO." Pressing the F key will cause another random number to be displayed.

   An incorrect response will result in the word "NO" on the display. After a short delay, the original decimal number will reappear and you should try the conversion process again. This cycle continues until you arrive at the correct answer.

9. Repeat steps 7 and 8, practicing conversion until you're confident of your ability. A good guideline to follow is . . . . when you answer 10 consecutive queries correctly, you're probably proficient.

DISCUSSION

As you worked through the exercises in this experiment, you probably developed your own shorthand method of conversion. After a few queries, you probably found that you didn't need the decimal-to-hexadecimal conversion chart any longer . . . you had the chart committed to memory. Perhaps you noticed that when 16_{10} is divided into the 2-digit decimal numbers used in this experiment, the resulting quotient always equals the MSD of the hexadecimal equivalent. Naturally, the remainder is the LSD. However, this only works for decimal numbers less than 159_{10}. For larger numbers, the procedure studied earlier must be used.

Since the Microprocessor Trainer displays data in hexadecimal and we naturally think in decimal, the conversion process must be reversed to interpret output data from the Trainer. For example, if the Trainer is programmed to add the numbers $1A_{16}$ and $9B_{16}$, the result $B5_{16}$ will be displayed. This hexadecimal number means very little. To understand the result, you must convert the sum ($B5_{16}$) to its decimal equivalent (181_{10}). Now the answer is clear.

Several methods can be used to change hexadecimal numbers to decimal. One process uses double conversion; first, the hexadecimal number is reduced to its binary equivalent; next, the resulting binary number is transformed into the resulting decimal equivalent.

Another, more commonly used method, is to use positional notation, inherent in any number system, and multiply each digit by its weighted value and then add the products. For example, the decimal equivalent of the hex number 11_{16} is derived as shown below:

Assign Weights: 16^1 16^0 Positional Weights

1 1

Weight × Digit: $1 \times 16^1 = 16 \leftarrow$ $\hookrightarrow 1 \times 16^0 = 1$

Add Products: $16 + 1 = 17$

Final Result: $11_{16} = 17_{10}$

The first step is to assign positioned weights to each digit. Since the number is hexadecimal, each position represents a power of 16_{10}. Next, multiply each digit by its positional weight. Finally, add the products. The resulting sum is the decimal equivalent. Therefore, as shown in the example, 11_{16} is equal to 17_{10}.

Now try a problem that's a bit more difficult . . . converting $6B_{16}$ to decimal. To begin with, this expression hardly looks like a number. Instead, it's a combination of a number and a letter. However, the notation at the bottom of the expression denotes a base 16 number so we know it's hexadecimal. The translation process is almost identical to the previous example. The only difference being that the hexadecimal "letter" must be changed to decimal before it can be multiplied by the positional weight. The conversion process is shown below.

Assign Weights:	16^1	16^0
	6	B
Convert to Decimal:	$6_{16} = 6_{10} \leftarrow$	$\rightarrow B_{16} = 11_{10}$
Weight × Digit:	$6 \times 16^1 = 96$	$11 \times 16^0 = 11$
Add Products:	$96 + 11 = 107$	
Final Result:	$6B_{16} = 107_{10}$	

Again, we begin by assigning positional weights to each digit. However, now the second step is to convert the hexadecimal characters to decimal numbers. Recall that 6_{16} is equal to 6_{10} and that B_{16} equals 11_{10}. Now multiply the weight by the decimal numbers, add the products and obtain the final result. As shown, the decimal equivalent of $6B_{16}$ is 107_{10}.

In the next section of this experiment, you will load a hexadecimal-to-decimal training program in the Trainer and then practice hexadecimal-to-decimal conversion.

PROCEDURE (Continued)

10. Prepare to enter the new program by pressing the RESET key. Next press the AUTO key and then enter address 0000.

11. Refer to Figure 4 and enter the Hexadecimal-to-Decimal training program listed there. When you've entered all of the instructions, press the RESET key as indicated at the end of the program.

12. Check the program that you've loaded by pressing the EXAM key and then entering address 0000. Use the FWD key to step through the program, comparing the stored program with the program listing in Figure 4. Use the CHAN key to correct any errors that you find.

    When you are satisfied that the program is correct, press the RESET key.

13. Now execute the program by first pressing the DO key and then entering address 0001. The word "GO" should appear on the display. The absence of this word indicates a programming error and you should go back and recheck the program as outlined in step 12.

14. Now press the F key. A 2-digit "hexadecimal" number will appear. The Trainer is asking for the decimal equivalent of this number. Convert the hexadecimal number into its decimal equivalent. Then enter your answer by pressing the D key. Two prompt characters will appear. Now enter your answer.

    If your response is correct, the Trainer will display "YES." You can then continue these conversion exercises by again pressing the F key.

    However, if your answer is incorrect, the Trainer will display "NO." After a short delay, the original hexadecimal number will reappear, and you can try again.

15. Continue the conversion training program until you are confident of your ability to change hexadecimal numbers to decimal numbers. The standard of ten correct conversions in a row is a good guideline.

ADDRESS	INST/DATA	ADDRESS	INST/DATA	ADDRESS	INST/DATA	ADDRESS	INST/DATA
0000	00	0024	BD	0048	FE	006C	8D
0001	CE	0025	FC	0049	52	006D	00
0002	C1	0026	BC	004A	00	006E	00
0003	6F	0027	BD	004B	3B	006F	00
0004	BD	0028	FE	004C	4F	0070	00
0005	FE	0029	52	004D	DB	0071	00
0006	50	002A	08	004E	BD	0072	80
0007	5E	002B	08	004F	00	0073	39
0008	FE	002C	00	0050	64		RESET
0009	96	002D	00	0051	7E		
000A	00	002E	00	0052	00		
000B	4C	002F	80	0053	01		
000C	81	0030	BD	0054	BD		
000D	63	0031	FC	0055	FE		
000E	23	0032	BC	0056	52		
000F	01	0033	BD	0057	00		
0010	4F	0034	FE	0058	00		
0011	97	0035	09	0059	15		
0012	00	0036	5F	005A	9D		
0013	B6	0037	80	005B	BD		
0014	CO	0038	10	005C	00		
0015	03	0039	25	005D	64		
0016	46	003A	04	005E	BD		
0017	25	003B	CB	005F	FC		
0018	FO	003C	OA	0060	BC		
0019	96	003D	20	0061	7E		
001A	00	003E	F8	0062	00		
001B	BD	003F	8B	0063	19		
001C	FE	0040	10	0064	CE		
001D	20	0041	1B	0065	00		
001E	B6	0042	D6	0066	00		
001F	CO	0043	00	0067	09		
0020	06	0044	11	0068	26		
0021	46	0045	26	0069	FD		
0022	25	0046	OD	006A	BD		
0023	FA	0047	BD	006B	FD		

Figure 4
Hexadecimal-to-decimal training program.

DISCUSSION

The translation of hexadecimal numbers into decimal equivalent numbers is an important part of your training.

You will find this skill is extremely handy when you begin to write programs later in this course. Now you should be able to convert between hexadecimal and decimal numbers with ease. Perhaps you even developed your own shorthand methods for these translations. If so, use them. However, a word of caution . . . be sure they work for all numbers. As mentioned previously, some techniques work with small numbers, but not with large numbers.

Experiment 3

Straight Line Programs

OBJECTIVES:

To demonstrate the instructions presented in Unit 2 with simple programs.

To present three new instructions and use them in simple programs.

To demonstrate some programming pitfalls.

To demonstrate the difference between RAM and ROM.

INTRODUCTION

Unit 2 introduced you to the basic microprocessor and its internal structure. You also learned six basic microprocessor instructions that are represented by 8-bit binary numbers called "op codes." Op codes allow you to use the microprocessor for data manipulation. Figure 5 lists the six instructions and their op codes. It also lists three new instructions that you will use in this experiment. These new instructions use the inherent addressing mode described in Unit 2.

This is the first experiment to introduce microprocessor instructions that you can identify. There are a number of Trainer keyboard commands that you must learn in order to examine and use the microprocessor instructions. The Trainer commands that you should know for this experiment are:

DO — Execute the program, beginning at the address specified after this key is pressed.

EXAM (examine) — Display the address and memory contents at the address specified after this key is pressed. Memory contents can be changed by pressing the CHAN key and entering new data.

FWD (forward) — Advance to the next memory location and display the contents.

CHAN (change) — Open the memory location being examined so that new data can be entered.

NAME	MNEMONIC	OPCODE	DESCRIPTION
Load Accumulator (Immediate)	LDA	$1000\ 0110_2$ or 86_{16}	Load the contents of the next memory location into the accumulator.
Add (Immediate)	ADD	$1000\ 1011_2$ or $8B_{16}$	Add the contents of the next memory location to the present contents of the accumulator. Place the sum in the accumulator.
Load Accumulator (Direct)	LDA	$1001\ 0110_2$ or 96_{16}	Load the contents of the memory location whose address is given by the next byte into the accumulator.
Add (Direct)	ADD	$1001\ 1011_2$ or $9B_{16}$	Add the contents of the memory location whose address is given by the next byte to the present contents of the accumulator. Place the sum in the accumulator.
Store Accumulator (Direct)	STA	$1001\ 0111_2$ or 97_{16}	Store the contents of the accumulator in the memory location whose address is given by the next byte.
Halt (Inherent)	HLT	$0011\ 1110_2$ or $3E_{16}$	Stop all operations.
Clear Accumulator (Inherent)	CLRA	$0100\ 1111_2$ or $4F_{16}$	Reset all bits in the accumulator to 0.
Increment Accumulator (Inherent)	INCA	$0100\ 1100_2$ or $4C_{16}$	Add 1 to the contents of the accumulator.
Decrement Accumulator (Inherent)	DECA	$0100\ 1010_2$ or $4A_{16}$	Subtract 1 from the contents of the accumulator.

Figure 5

Instructions used in Experiment 3.

BACK — Go back to the previous memory location and display the contents.

AUTO (automatic) — Open the memory location specified, after this key is pressed, so that data can be entered. After data has been entered, automatically advance to the next memory location and wait for data.

SS (single step) — Go to the address specified by the program counter and execute the instruction at that address. Wait at the next instruction.

ACCA (accumulator) — Display the contents of the accumulator when this key is pressed. Accumulator contents can be changed by pressing the CHAN key and entering new data.

PC (program counter) — Display the contents of the program counter. This points to the next location in memory that the microprocessor will "fetch" from. Program counter contents can be changed by pressing the CHAN key and entering the new address.

RESET — Clear any Trainer keyboard commands and display "CPU UP." Memory contents and microprocessor contents are not disturbed.

You have access to all of these keyboard commands after the RESET key is pressed.

In this experiment, you will load some simple straight-line programs into the Trainer and examine how the microprocessor executes them. In its normal mode of operation, the microprocessor executes programs much too fast for a person to follow. It can execute hundreds of thousands of instructions each second. To allow us to witness the operation of the MPU, this high speed operation must be slowed down. The Microprocessor Trainer has a mode of operation that allows us to control the execution of single instructions. In this single-step mode, we can look at the contents of the accumulator, the program counter, and various memory locations, after each instruction is executed. In this way, we can follow exactly how the computer performs each step of the program. For this reason, you will use the single-step mode for most of the programs in this experiment.

PROCEDURE

1. Switch your Trainer on, and press the RESET key.

2. Your first program will use the immediate addressing mode to add two numbers. Press AUTO and enter starting address 0000. Then load the hex contents of the program listed in Figure 6.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	86	LDA	Load accumulator immediately with
0001	21	33_{10}	Operand 1.
0002	8B	ADD	Add to accumulator immediately with
0003	17	23_{10}	Operand 2.
0004	3E	HLT	Stop.

Figure 6
Addition of two numbers through the immediate addressing mode.

3. Press the RESET key, then examine your program to make sure it was properly entered. **Always** examine your program after it is entered.

4. Press the ACCA key and record the value _ _. This is a random number since no data has been loaded.

5. Press the PC key, then change the contents of the program counter to 0000 (the starting address of your program).

6. Press the SS key. This lets the Trainer execute the first instruction. The display should show 00028b. 0002 represents the address of the next instruction; 8b is the next instruction.

7. Press the ACCA key and record the value _ _. The first program instruction was LDA, and the next byte contained the data (operand) to be loaded, which is 21_{16}. This should be the value you recorded in this step.

8. Press the PC key and record the value ____. This value points to the next memory location, which should be 0002.

   You may have noted that the address 0002 and instruction 8b were displayed when you first pressed the SS key. This would seem to indicate that 8b was already fetched and the program counter should point to address 0003. However, the control program allows the Trainer to "look" at the next instruction.

9. Press the SS key and record the value _ _ _ _ _ _. The second instruction has been executed and the display should show the next instruction and its address.

10. Press the ACCA key and record the value _ _. The second operand has been added to the first operand and the sum is stored in the accumulator.

11. Press the SS key. Note that the display does not change. This is because the next instruction was a halt instruction ($3E_{16}$). The Trainer is preprogrammed to stop at a halt instruction. It also loses control of the single-step function when the halt instruction is implemented.

12. Enter the program (HEX contents) listed in Figure 7. Then examine the program to make sure it is properly entered.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	96	LDA	Load accumulator direct with
0001	07	07_{16}	operand 1 which is stored at this address.
0002	9B	ADD	Add to accumulator direct with operand 2
0003	08	08_{16}	which is stored at this address.
0004	97	STA	Store the sum
0005	09	09_{16}	at this address.
0006	3E	HLT	Stop.
0007	20	32_{10}	Operand 1.
0008	17	23_{10}	Operand 2.
0009	00	00	Reserved for sum.

Figure 7
Additional of two numbers through the direct addressing mode.

13. Press the ACCA key and record the value _ _. This is the value obtained in the previous program, a value you entered prior to this program, or a random value produced when you plugged in the Trainer.

14. Enter the program starting address into the program counter and single-step through the program. Record the specified information after each step.

    Step 1 display _ _ _ _ _ _.

    ACCA _ _.

    Step 2 display _ _ _ _ _ _.

    ACCA _ _.

    Step 3 display _ _ _ _ _ _.

    ACCA_ _.

15. Examine address 0009. Its value is _ _. This value should be identical to the value now stored in the ACCA.

16. Now compare your recorded data with the program in Figure 7. This will give you a general picture of how the microprocessor uses various instructions and data to perform a desired function.

17. Change the data in the ACCA and at address 0009 to FF, then execute the program with the DO key. This is done by depressing the DO key and then entering the address of the first instruction (0000). This allows the MPU to execute the program at its normal speed. After the program runs, you must press RESET to return control to the keyboard.

18. The data in the ACCA is _ _ and the data in address 0009 is _ _. These should be the same and equal to the sum of the two operands.

19. The program counter contains the address _ _ _ _. This should be the address of the next memory location after the HLT instruction.

20. Now write a program of your own. Using the **direct** addressing mode, write a program that will multiply 4 times 4, by adding 4 to itself in three consecutive steps. The final answer should be held in the accumulator. After you write your program, enter it into the Trainer and execute it. Keep trying until it produces a final result of 10_{16} (which is 16_{10}) in the accumulator.

One solution to the problem is shown in Figure 8. Yours should be similar, although not necessarily identical.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/DECIMAL CONTENTS	COMMENTS
0000	96	LDA	Load accumulator direct with
0001	09	09_{16}	operand 1 which is stored at this address.
0002	9B	ADD	Add to accumulator direct with
0003	09	09_{16}	operand 1 which is stored at this address.
0004	9B	ADD	Add to accumulator direct with
0005	09	09_{16}	operand 1 which is stored at this address.
0006	9B	ADD	Add to accumulator direct with
0007	09	09_{16}	operand 1 which is stored at this address.
0008	3E	HLT	Stop.
0009	04	04_{10}	Operand 1.

Figure 8
Multiplication of a number by another through multiple addition in the direct addressing mode.

21. Load the program shown in Figure 8 into the Trainer. Enter the program starting address into the program counter and single-step through the program. Record the specified information after each step.

Step 1 display _ _ _ _ _ _. ACCA _ _.

Step 2 display _ _ _ _ _ _. ACCA _ _.

Step 3 display _ _ _ _ _ _. ACCA _ _.

Step 4 display _ _ _ _ _ _. ACCA _ _.

22. According to the microprocessor, the product of 4_{16} times 4_{16} is _ _$_{16}$.

23. Now that you are becoming acquainted with the instructions described in Unit 2, examine the three instructions introduced in this Experiment. Enter the program listed in Figure 9.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	4F	CLRA	Clear accumulator.
0001	97	STA	Store the contents
0002	OA	OA_{16}	at this address.
0003	4C	INCA	Increment accumulator.
0004	97	STA	Store the contents
0005	OB	OB_{16}	at this address.
0006	4A	DECA	Decrement accumulator.
0007	97	STA	Store the contents
0008	OC	OC_{16}	at this address.
0009	3E	HLT	Stop.
000A	FF	FF_{16}	Reserved for data.
000B	FF	FF_{16}	Reserved for data.
000C	FF	FF_{16}	Reserved for data.

Figure 9
Implementation of the Clear, Increment, and Decrement instructions.

24. Set the program counter to 0000 and single-step through the program. Record the specified information after each step.

Step 1 display _ _ _ _ _ _. ACCA _ _.

Step 2 display _ _ _ _ _ _. ACCA _ _.

Step 3 display _ _ _ _ _ _. ACCA _ _.

Step 4 display _ _ _ _ _ _. ACCA _ _.

Step 5 display _ _ _ _ _ _. ACCA _ _.

Step 6 display _ _ _ _ _ _.

25. Compare your accumulated data with the program in Figure 9. Note that when op codes $4F_{16}$, $4C_{16}$, and $4A_{16}$ are executed, the single-step display advances only one address location. This is because of their inherent addressing mode; immediate and direct addressing modes require two locations in memory.

26. Shown below is a program to swap the contents of two memory locations. Now examine the process using the Trainer. Enter the program listed in Figure 10.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	96	LDA	Load accumulator direct with operand 1
0001	10	10_{16}	stored at this address.
0002	97	STA	Store operand 1
0003	12	12_{16}	at this address.
0004	96	LDA	Load accumulator direct with operand 2
0005	11	11_{16}	stored at this address.
0006	97	STA	Store operand 2
0007	10	10_{16}	at this address.
0008	96	LDA	Load accumulator direct with operand 1
0009	12	12_{16}	stored at this address.
000A	97	STA	Store operand 1
000B	11	11_{16}	at this address.
000C	4F	CLRA	Clear the accumulator.
000D	97	STA	Store the contents
000E	12	12_{16}	at this address.
000F	3E	HLT	Stop.
0010	AA	170_{10}	Operand 1.
0011	BB	187_{10}	Operand 2.
0012	00	00	Temporary storage.

Figure 10
Data transfer between two addresses.

27. Set the program counter to starting address 0000 and single-step through the program. Record the specified information after each step.

Step 1 display _ _ _ _ _ _.	ACCA _ _.
Step 2 display _ _ _ _ _ _.	ACCA _ _.
Step 3 display _ _ _ _ _ _.	ACCA _ _.
Step 4 display _ _ _ _ _ _.	ACCA _ _.
Step 5 display _ _ _ _ _ _.	ACCA _ _.
Step 6 display _ _ _ _ _ _.	ACCA _ _.
Step 7 display _ _ _ _ _ _.	ACCA _ _.
Step 8 display _ _ _ _ _ _.	ACCA _ _.

28. Examine address:

0010 _ _.

0011 _ _.

0012 _ _.

29. Compare your accumulated data with the program in Figure 10.

30. Now you will examine some common programming pitfalls. Without modifying the previous program, except as directed in Figure 11, enter the program listed in Figure 11.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	86	LDA	Load accumulator immediately with
0001	4F	79_{10}	operand 1.
0002	97	STA	Store operand 1
0003	05	05_{16}	at this address.
0004	4A	DECA	Decrement accumulator
0005	3E	HLT	Stop.

Figure 11
Storing data at an address in the program.

31. Set the program counter to 0000 and single-step through the program. Record the specified information after each step.

Step 1 display ______. ACCA __.

Step 2 display ______. ACCA __.

Step 3 display ______. ACCA __.

Step 4 display ______. ACCA __.

Step 5 display ______. ACCA __.

Step 6 display ______. ACCA __.

Step 7 display ______. ACCA __.

Step 8 display ______. ACCA __.

Step 9 display ______. ACCA __.

32. Compare your accumulated data with the program in Figure 11. Note that the data in the accumulator (operand 1) has been stored at address 0005. This removed the HLT instruction and allowed the microprocessor to continue executing any valid instructions in memory. In this case, the remaining unaltered instructions from the previous program are used. When you write a program, **make sure** you do not store data at an address that contains a needed instruction or data.

33. Using the data you accumulated in step 31 of this experiment, plus the programs listed in Figures 10 and 11, determine the contents of address:

    0010 _ _.

    0011 _ _.

    0012 _ _.

34. Now examine the Trainer contents at address:

    0010 _ _.

    0011 _ _.

    0012 _ _.

    Your estimated data from step 33, and the actual contents should be identical. If they are not, re-examine your calculations and the contents of each memory location from 0000 to 0012. You might have inadvertently modified the contents of an address in the previous steps.

35. Without modifying the previous program, except as directed in Figure 12, enter the program listed in Figure 12.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	86	LDA	Load accumulator immediately with
0001	40	64_{10}	operand 1.
0002	8B	ADD	Add to accumulator immediately with
0003	0A	10_{10}	operand 2.
0004	97	STA	Store the sum
0005	07	07_{16}	at this address.
0006	4F	CLRA	Clear accumulator.
0007	00	00	Reserved for data.

Figure 12
Addition of two numbers with immediate addressing.

36. Set the program counter to 0000 and single-step through the program. Record the specified information after each step.

Step 1 display ______. ACCA __.

Step 2 display ______. ACCA __.

Step 3 display ______. ACCA __.

Step 4 display ______. ACCA __.

Step 5 display ______. ACCA __.

Step 6 display ______. ACCA __.

Step 7 display ______. ACCA __.

Step 8 display ______. ACCA __.

Step 9 display ______. ACCA __.

37. Compare your accumulated data with the program in Figure 12. Note that the Trainer executed the instructions beyond address 0007. This occurred because there was no halt instruction in the program. Always end your program with a halt instruction. If you don't, the microprocessor will try to execute all of the information contained in memory, thinking it is part of the program. In the process, the program you entered may get modified.

38. This final programming pitfall illustrates a problem almost everybody experiences. Enter the program listed in Figure 13.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	96	LDA	Load accumulator direct with
0001	07	07_{16}	operand 1 stored at this address.
0002	8B	ADD	Add to accumulator direct with
0003	07	07_{16}	operand 1 stored at this address.
0004	8B	ADD	Add to accumulator direct with
0005	07	07_{16}	operand 1 stored at this address.
0006	3E	HLT	Stop.
0007	05	05_{10}	Operand 1.

Figure 13
Multiplication of two numbers using successive addition in the direct addressing mode.

39. Set the program counter to 0000 and single-step through the program. Record the specified information after each step.

Step 1 display _ _ _ _ _ _. ACCA _ _.

Step 2 display _ _ _ _ _ _. ACCA _ _.

Step 3 display _ _ _ _ _ _. ACCA _ _.

40. Compare your accumulated data with the program in Figure 13. The program should have added 05 three times (5 × 3) for the answer OF. The Trainer indicates the answer is 13. This discrepancy occurred because the program contains the wrong **addressing mode op code** for the ADD function. It should be 9B rather than 8B. Return to Figure 13 and change the two ADD op codes to 9B so the program will be correct.

41. In Unit 2, you were shown that RAM (random access memory) was a read/write type memory, while ROM (read only memory) is a preprogrammed memory that can only be read and not written into. To examine these memory types, enter FF at address 0000 through 000F.

42. Examine the following memory locations and write down the contents next to each address. Use the first data column for each address. You will use the second column later.

ADDRESS	DATA	DATA	ADDRESS	DATA	DATA
0000	--	--	FD00	--	--
0001	--	--	FD01	--	--
0002	--	--	FD02	--	--
0003	--	--	FD03	--	--
0004	--	--	FD04	--	--
0005	--	--	FD05	--	--
0006	--	--	FD06	--	--
0007	--	--	FD07	--	--
0008	--	--	FD08	--	--
0009	--	--	FD09	--	--
000A	--	--	FD0A	--	--
000B	--	--	FD0B	--	--
000C	--	--	FD0C	--	--
000D	--	--	FD0D	--	--
000E	--	--	FD0E	--	--
000F	--	--	FD0F	--	--

43. Turn the Trainer power off, then unplug the line cord. Wait twenty seconds, then plug in the line cord and turn on the Trainer.

44. Examine the memory locations listed in step 42, and write down the contents next to each address, in the second data column. Compare the two sets of data. Notice the data obtained at address 0000 through 000F changed when all Trainer power was removed. However, the data at address FD00 through FD0F is unchanged. Address 0000 is RAM, while address FD00 is ROM. Memory is lost from RAM when power is removed. When power is reapplied, random data will appear in the memory.

    Enter FF at address FD00 through FD0F. Now examine address FD00 through FD0F. Notice the data is identical to that obtained in step 42. This shows that ROM can not be written into. You can send data down the data bus, but the memory will not accept it.

SUGGESTION: Use the nine instructions presented and write a few sample programs of your own. It's quite simple and can be great fun.

Experiment 4

Arithmetic and Logic Instructions

OBJECTIVES:

To present seven new instructions and use them in simple programs.

To demonstrate 2's complement conversion.

To demonstrate binary subtraction.

To demonstrate binary addition of signed numbers.

To demonstrate logical manipulation of data using the AND and OR instructions.

INTRODUCTION

In Experiment 3, you used nine instructions to write various programs. These instructions were:

MNEMONIC	OP CODE	ADDRESSING MODE
LDA	86_{16}	Immediate
LDA	96_{16}	Direct
ADD	$8B_{16}$	Immediate
ADD	$9B_{16}$	Direct
STA	97_{16}	Direct
CLRA	$4F_{16}$	Inherent
INCA	$4C_{16}$	Inherent
DECA	$4A_{16}$	Inherent
HLT	$3E_{16}$	Inherent

Seven new instructions are presented in this experiment. Each is listed in Figure 14.

Unit 3 examined the process of binary arithmetic, 2's complement arithmetic, signed number addition, and Boolean logic. Through sample programs, this experiment will illustrate some of the operations presented in Unit 3.

NAME	MNEMONIC	OPCODE	DESCRIPTION
Complement 2's or Negate (Inherent)	NEGA	$0100\ 0000_2$ or 40_{16}	Replace the contents of the accumulator with its complement plus 1.
Subtract (Immediate)	SUB	$1000\ 0000_2$ or 80_{16}	Subtract the contents of the next memory location from the contents of the accumulator. Place the difference in the accumulator.
Subtract (Direct)	SUB	$1001\ 0000_2$ or 90_{16}	Subtract the contents of the memory location whose address is given by the next byte from the present contents of the accumulator. Place the difference in the accumulator.
AND (Immediate)	ANDA	$1000\ 0100_2$ or 84_{16}	Perform the logical AND between the contents of the accumulator and the contents of the next memory location. Place the result in the accumulator.
AND (Direct)	ANDA	$1001\ 0100_2$ or 94_{16}	Perform the logical AND between the contents of the accumulator and the contents of the memory location whose address is given by the next byte. Place the result in the accumulator.
OR, Inclusive (Immediate)	ORA	$1000\ 1010_2$ or $8A_{16}$	Perform the logical OR between the contents of the accumulator and the contents of the next memory location. Place the result in the accumulator.
OR, Inclusive (Direct)	ORA	$1001\ 1010_2$ or $9A_{16}$	Perform the logical OR between the contents of the accumulator and the contents of the memory location whose address is given by the next byte. Place the result in the accumulator.

Figure 14
Instructions introduced in this experiment.

PROCEDURE

1. In the first part of the experiment, you will determine how the microprocessor represents negative and positive numbers. The program shown in Figure 15 loads a positive number into the accumulator and then repeatedly decrements the number until it is negative. Enter this program into the Trainer. Verify that you entered it properly by examining each address.

2. Go to the single-step mode by: pressing the PC key; pressing the CHAN key; and entering the starting address (0000). Single-step through the program by repeatedly pressing the SS key. Notice that the first instruction places $+5_{10}$ in the accumulator. Refer to Figure 16 and record the contents of the accumulator (in both hexadecimal and binary) after each DECA instruction is executed.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	86	LDA	Load accumulator immediate
0001	05	05	with 05.
0002	4A	DECA	
0003	4A	DECA	Repeatedly decrement
0004	4A	DECA	the accumulator.
0005	4A	DECA	
0006	4A	DECA	
0007	4A	DECA	
0008	4A	DECA	
0009	4A	DECA	
000A	4A	DECA	
000B	4A	DECA	
000C	4A	DECA	
000D	4A	DECA	
000E	4A	DECA	
000F	3E	HLT	Halt

Figure 15
This program decrements the contents of the accumulator from +5 to −8.

3. In step 7, the number in the accumulator changed from 0 to -1. The microprocessor expresses −1 as _ _$_{16}$ or _ _ _ _ _ _ _ _$_2$. The table you have developed in Figure 16 shows how the microprocessor expresses the signed number from + 5 to −5 in both hexadecimal and binary. The next program will add signed numbers like these.

AFTER STEP	CONTENTS OF ACCUMULATOR		
	DECIMAL	HEXADECIMAL	BINARY
1	+5	05	0000 0101
2	+4		
3	+3		
4	+2		
5	+1		
6	0		
7	−1		
8	−2		
9	−3		
10	−4		
11	−5	FB	1111 1011

Figure 16
Record results here.

4. Enter the program shown in Figure 17. Use the single step mode to execute the program. What number is in the accumulator after the first instruction is executed? _ _$_{16}$ or _ _ _ _ _ _ _ _$_2$. What signed decimal number does this represent? ________.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	86	LDA	Load accumulator immediate
0001	05	+5	with +5.
0002	8B	ADD	Add immediate
0003	FB	−5	−5.
0004	8B	ADD	Add immediate
0005	FC	−4	−4
0006	3E	HLT	

Figure 17
Adding signed numbers.

5. What number is in the accumulator after the second instruction is executed? _ $\_{}_{16}$ or _ _ _ _ _ _ _ $\_{}_{2}$. What decimal number does this represent? ________.

6. What number is in the accumulator after the third instruction is executed? _ $\_{}_{16}$ or _ _ _ _ _ _ _ $\_{}_{2}$. What signed decimal number does this represent? ________.

DISCUSSION

These very simple examples illustrate how the microprocessor represents signed numbers. Further experiments will show that the microprocessor can represent signed numbers between $+127_{10}$ and -128_{10}. You could determine the bit pattern for each negative number by clearing the accumulator and decrementing the required number of times. However, there are much simpler ways of determining the proper bit pattern for negative numbers.

The simplest way is to start with the positive binary equivalent and take the two's complement by changing all 0's to 1's and 1's to 0's and adding 1. The microprocessor has an instruction that will do this for us. It is called the two's complement or Negate instruction. Its mnemonic is NEGA. This instruction changes the number in the accumulator to its two's complement. It is used to change the sign of a number.

PROCEDURE (Continued)

7. Load the program shown in Figure 18. Use the single-step mode to execute the program. Execute the first instruction by depressing the SS key. What number is in the accumulator? $\_\_{}_{16}$ or $\_\_\_\_\_\_{}_{2}$. What signed decimal number does this represent? ________.

8. Execute the second instruction. What number is in the accumulator? $\_\_{}_{16}$ or $\_\_\_\_\_\_\_\_{}_{2}$. What signed decimal number does this represent? ________. Compare this with the number in step 7. What affect did the NEGA instruction have? ________________

________________________________.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	86	LDA	Load accumulator immediate
0001	05	+5	with +5.
0002	40	NEGA	Change the number to −5.
0003	40	NEGA	Change it back to +5.
0004	4A	DECA	Decrement the number to +4.
0005	40	NEGA	Change the number to −4.
0006	40	NEGA	Change it back to +4.
0007	3E	HLT	Halt

Figure 18
Using the NEGA instruction.

9. Execute the third instruction. What number is in the accumulator? $\_\_{}_{16}$ or $\_\_\_\_\_\_\_\_{}_{2}$. What signed decimal number does this represent? ________. Is your answer the same as that found in step 7? ________.

10. Execute the fourth instruction. This decrements the accumulator so that it now contains the signed decimal number ______.

11. Execute the fifth instruction. What number is in the accumulator? $\_\_{}_{16}$ or $\_\_\_\_\_\_\_\_{}_{2}$. What signed decimal number does this represent? ________.

12. Execute the sixth instruction. The number in the accumulator is $\_\_{}_{16}$ once more.

DISCUSSION

The program used the NEGA instruction four times. The first time, the NEGA instruction changed 05_{16} to its two's complement FB_{16}. Referring back to the table you developed in Figure 16, this is the representation for -5_{10}. Thus, the NEGA instruction effectively changes the sign of the number in the accumulator. The next step proved this again by converting -5_{10} back to $+5_{10}$. To further emphasize the point, the number was decremented to $+4_{10}$. The next NEGA instruction changed this to FC_{16} which is the representation for -4_{10}. The final NEGA instruction converts this back to $+4_{10}$. This instruction allows us to convert a positive number to its negative equivalent and vice versa.

In Unit 3, you learned that the MPU can work with signed numbers in the range of $+127_{10}$ to -128_{10} or unsigned numbers in the range of 0 to 255_{10}. This capability results from the way we interpret bit patterns. The following steps will demonstrate this.

PROCEDURE (Continued)

13. Figure 19 shows a program for adding the unsigned numbers 220_{10} and 27_{10}. Load this program into the Trainer and execute it. The final result in the accumulator is $\_\,\__{16}$ or $\_\,\_\,\_\,\_\,\_\,\_\,\_\,\__{2}$. What unsigned decimal number does this represent? ________.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	86	LDA	Load accumulator immediate
0001	DC	220_{10}	with 220_{10}.
0002	8B	ADD	Add immediate
0003	1B	27_{10}	27_{10}.
0004	3E	HLT	Halt.

Figure 19
Adding unsigned numbers.

14. Figure 20 shows a program for adding the signed numbers -36_{10} and 27_{10}. Load and execute this program. The final result in the accumulator is $\_\_{}_{16}$ or $\_\_\_\_\_\_\_\_{}_{2}$. What signed decimal number does this represent? ________.

15. Compare the results obtained in steps 13 and 14. Compare the HEX Contents columns of Figure 19 with that of Figure 20.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	86	LDA	Load accumulator immediate
0001	DC	-36_{10}	with -36_{10}
0002	8B	ADD	Add immediate
0003	1B	$+27_{10}$	$+27_{10}$
0004	3E	HLT	Halt.

Figure 20
Adding signed numbers.

DISCUSSION

This demonstrates that the MPU simply adds bit patterns. It is our interpretation of these patterns that decide whether we are using signed or unsigned numbers. After all, the two programs are identical except for our interpretation of the input and output data.

Negative numbers are often encountered when performing subtract operations. The subtract instruction was shown earlier in Figure 14. Either immediate or direct addressing can be used.

PROCEDURE (Continued)

16. Load the program shown in Figure 21. Execute the program using the single-step mode. What is the number in the accumulator after the first subtract instruction is executed? $\_\_{}_{16}$ or $\_\_\_\_\_\_\_\_{}_{2}$ or $\_\_{}_{10}$.

17. What is the number in the accumulator after the second subtract instruction is executed? $\_\_{}_{16}$ or $\_\_\_\_\_\_\_\_{}_{2}$. What signed decimal number does this represent? ________.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	86	LDA	Load accumulator immediate
0001	2F	47_{10}	with 47_{10}.
0002	80	SUB	Subtract immediate
0003	10	16_{10}	16_{10}
0004	80	SUB	Subtract immediate
0005	23	35_{10}	35_{10}
0006	3E	HLT	Halt

Figure 21
Using the subtract instruction.

DISCUSSION

The first subtract instruction subtracted 16_{10} from 47_{10}, leaving 31_{10}. The second one subtracted 35_{10} from 31_{10}. This produced a result of -4_{10}. However, the MPU expressed −4 in two's complement form (FC_{16} or 1111 1100_2). You will find this to be the case anytime the MPU produces a negative result.

Now let's look at some of the logical instructions available to the microprocessor. The AND and OR instructions are described in Figure 14. Carefully read the description of these instructions given there. While these instructions have many uses, we will demonstrate only one here. Earlier you learned that certain peripheral devices communicate with computers using the ASCII code. Thus, when the "2" key on a teletypewriter is pushed, the computer receives the ASCII code for 2, which is 0011 0010. The ASCII code for 6 is 0011 0110. Notice that the four least significant bits of the ASCII character are the binary value of the corresponding numeral. Thus, we can convert the ASCII characters for the numerals 0 through 9 to binary simply by setting the four most significant bits to 0's. Likewise, we can convert the binary numbers 0000 0000 through 0000 1001 to ASCII by changing the four mos significant bits to 0011.

PROCEDURE (Continued)

18. Load the program shown in Figure 22. Single-step through the first instruction. The number in the accumulator is $\_\_\_\_\_\_\_\_{}_2$.

19. Execute the second instruction. This AND's the contents of the accumulator with the "mask" ________. The number in the accumulator after this AND operation is $\_\_\_\_\_\_\_\_{}_2$. Compare this with the number that was in the accumulator in step 18. Compare both numbers with the mask. A 1 in the original number is retained only if there is a ______ in the corresponding bit position of the mask.

20. Execute the third instruction. In what memory location is the number in the accumulator stored? $\_\_\_\_\_\_{}_{16}$. What number is now in the accumulator? $\_\_\_\_\_\_\_\_{}_2$. Does the number still appear in the accumulator after being stored in memory? ______.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	96	LDA	Load the accumulator with
0001	OB	OB	the ASCII character at this address.
0002	84	AND	AND it with
0003	OF	OF	this "mask".
0004	97	STA	Store the binary equivalent
0005	OC	OC	at this address.
0006	8A	ORA	OR the number with
0007	30	30	this "mask".
0008	97	STA	Store the result
0009	OD	OD	here.
000A	3E	HLT	Stop
000B	37	0011 0111	ASCII character for numeral 7.
000C	—	—	Reserved
000D	—	—	Reserved

Figure 22
Using the AND and OR instruction.

21. Execute the fourth instruction. This OR's the contents of the accumulator with the "mask" _ _ _ _ _ _ _ $_2$. The number in the accumulator is _ _ _ _ _ _ _ $_2$. Compare this with the mask and the number that was in the accumulator in step 20. A 1 is produced in the result whenever there is a ______ in the corresponding bit position of either the original number, the mask, or both.

22. Execute the fifth instruction. This stores the number in memory location ________$_{16}$.

23. Examine memory locations $000B_{16}$, $000C_{16}$, and $000D_{16}$ and compare their contents.

DISCUSSION

The program first converts the ASCII code for the number "7" to the binary number 0000 0111. It does this by ANDing the ASCII code with the "mask" 0000 1111_2. Notice that a 1 bit in the mask allows the corresponding bit in the original number to be retained. The four most significant bits of the original number are "masked off" because they are ANDed with 0's.

The OR operation restores the ASCII character by attaching 0011 as the four most significant bits.

Experiment 5

Program Branches

OBJECTIVES:

To manipulate the N, Z, V, and C condition code registers and determine the conditions that set and reset these flags.

To verify the operation of a simple multiply by repeated addition program that uses the BEQ conditional branch instruction and the BRA instruction.

To demonstrate the ability to write a program that divides by repeated subtraction and uses a conditional branch and BRA instruction.

To introduce a shorthand method of calculating relative addresses.

To verify the operation of a program that converts BCD numbers to their binary equivalent.

To demonstrate the effect an incorrect relative address can have on a program operation and how the microprocessor trainer can be used to debug programs.

INTRODUCTION

As mentioned previously, conditional branch instructions give the computer the power to make decisions. As the name implies, a certain condition must be met before a branch takes place. The condition code registers monitor the accumulator and signal the presence of a specific condition. If the MPU encounters a conditional branch instruction, it merely checks the condition code registers, or flags, to see if the condition is satisfied. If the specific flag is set, the program branches off to another section. If not, the normal program continues.

Therefore, the conditional branch instructions inherit their power from these simple condition code registers. A sound knowledge of how these flags are set and cleared will enhance your ability as a programmer.

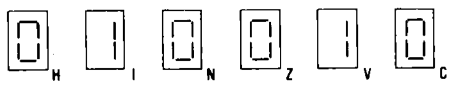


Figure 23
Displaying the conditions of the flags.

Since condition code registers are very important, your Trainer was designed with a special key to allow you to examine these flags. The key is labelled "C" for "Condition Code." When this key is pressed, the state of the condition code registers will be displayed. Each LED displays the contents of one register. The letter just to the right of each LED denotes the corresponding register as shown in Figure 23.

Notice that there are six flag registers. For the moment we aren't concerned with the two left-most flags. They will be covered in a later unit. However, we are interested in the N, Z, V, and C flags, because they indicate conditions that can lead to conditional branches. Notice that the flags can either be set as indicated by a 1 or they can be cleared as indicated by a 0.

In this first portion of the experiment, you will implement a "do-nothing" program that manipulates the condition code registers. Then single-stepping through the program, you will examine how the accumulator changes these flags.

PROCEDURE

1. Turn on the Trainer and then press the RESET key.

2. Now, load the program listed in Figure 24 into the Trainer. Once the program is loaded, go back and examine it to insure that it's entered correctly.

   Now look at the first instruction of the program in Figure 24. It has the op code 01 and the mnemonic is "NOP". As the comments column points out, this is a "do-nothing" type of instruction called a "No-Op." In other words, it performs no operation. In this program, the NOP's primary function is to allow you to see the first instruction before it's executed.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	01	NOP	"DO Nothing" Instruction
0001	86	LDA	Load the accumulator immediate
0002	FF	FF_{16}	with FF_{16}.
0003	86	LDA	Load the accumulator immediate
0004	77	77_{16}	with 77_{16}.
0005	86	LDA	Load the accumulator immediate
0006	00	00_{16}	with 00_{16}.
0007	86	LDA	Load the accumulator immediate
0008	01	01_{16}	with 01_{16}
0009	86	LDA	Load the accumulator immediate
000A	92	92_{16}	with 92_{16}
000B	8B	ADD	Add Immediate
000C	C6	$C6_{16}$	$C6_{16}$
000D	86	LDA	Load the accumulator immediate
000E	08	08_{16}	with 08_{16}.
000F	8B	ADD	Add Immediate
0010	08	08_{16}	08_{16}.
0011	86	LDA	Load the accumulator immediate
0012	01	01_{16}	with 01_{16}.
0013	80	SUB	Subtract immediate
0014	02	02_{16}	02_{16}.
0015	86	LDA	Load the accumulator immediate
0016	77	77_{16}	with 77_{16}.
0017	80	SUB	Subtract immediate
0018	66	66_{16}	66_{16}.
0019	86	LDA	Load the accumulator immediate
001A	49	49_{16}	with 49_{16}.
001B	8B	ADD	Add immediate
001C	60	60_{16}	60_{16}.
001D	86	LDA	Load the accumulator immediate
001E	10	10_{16}	with 10_{16}.
001F	3E	HLT	Halt.

Figure 24
Program to illustrate the condition code registers.

In previous experiments, you probably noticed that when you single-stepped through programs, you never saw the first instruction. This is because in the "SS" mode, the Trainer executes the first instruction automatically and then stops on the second instruction. This can be somewhat confusing.

To offset this problem, we merely insert the NOP. The Trainer "sees" this as the first instruction, although nothing is accomplished by the NOP. Therefore, the Trainer displays the next instruction, which is the first "real" instruction of the program, permitting you to view it before it's executed.

3. Load the program counter with address 0000 and then press the SS key. Recall that the first four displays represent the address that's currently in the program counter. The two right-most displays show the op code stored at this address. Record the information below.

   PC _ _ _ _ OP CODE _ _

   Now, press the ACCA key and record the contents of the accumulator.

   ACCA _ _

   The contents of the accumulator will be a random number, since we haven't yet executed a program instruction.

   Now, press the CC key and record the contents of the N, Z, V, and C condition code registers below.

   _ _ _ _

   N Z V C

   Again, the states of the flags are random at this time.

4. Now, press the SS key and then the ACCA key. Record the contents of the accumulator below.

   ACCA _ _

   Press key CC and record the state of the N flag below.

   _ _ _ _
   N Z V C

   With the negative number FF_{16} in the accumulator, the negative (N) flag is set.

5. Press the SS key again. The program count should now be 0005_{16} and the op code at this address is 86. Now check and record the contents of the accumulator and the N flag.

ACCA _ _   _ _ _ _
N Z V C

With the positive number 77_{16} in the accumulator, the N flag is cleared, or reset, to 0.

From the information gathered in steps 4 and 5, what conclusions do you reach with respect to the N flag and the contents of the accumulator?

____

6. Single-step the program again. The program count is now 0007_{16}. Record the contents of the accumulator and the condition of the Z flag below.

ACCA _ _   _ _ _ _
N Z V C

With 00_{16} in the accumulator, the Z flag is set.

Press SS and again record the contents of the accumulator and the Z flag below.

ACCA _ _   _ _ _ _
N Z V C

The accumulator now contains 01_{16} and the Z flag is cleared. What is the relation between the contents of the accumulator and the Z, or zero flag?

____

7. Single-step again and record the information below.

ACCA _ _   _ _ _ _
N Z V C

This step loads the number 92_{16} into the accumulator. Bit 7 of the accumulator contains a 1_2 so the N flag is set. Naturally, the Z flag is cleared. The next instruction will add $C6_{16}$ to the contents of the accumulator. As shown below, this operation should generate a carry.

$$\begin{array}{llcl}
1001 & 0010 & = & 92_{16} \\
1100 & 0110 & = & C6_{16} \\
\hline
0101 & 1000 & = & 158_{16}
\end{array}$$

$$\text{CARRY} \longrightarrow 1$$

Press the SS key and record the information below.

ACCA _ _   _ _ _ _
N Z V C

The 8-bit accumulator cannot hold the 9-bit sum. However, the carry generated by the addition sets the C flag.

8. This step loads the number 08_{16} into the accumulator. Press the SS key and record the information below.

ACCA _ _   _ _ _ _
N Z V C

Notice that loading this new number into the accumulator didn't affect the carry (C) flag. The next step will add 08_{16} to the contents of the accumulator (08_{16}).

9. Press the SS key and record the information below.

ACCA _ _   _ _ _ _
N Z V C

The accumulator now contains the sum of the addition (10_{16}) and the carry flag is cleared.

From the results of steps 8 and 9, you might conclude that the carry flag can be cleared by another ______ ______ that does not result in a carry.

10. Press the SS key. The program count should now be 0013. Record the information below.

ACCA _ _   _ _ _ _
N Z V C

This shows that the accumulator contains 01_{16} and that the N, Z, and C flags are all cleared. When the next instruction is executed, the number 02_{16} will be subtracted from 01_{16} (the contents of the accumulator). As shown below, the subtraction should result in a borrow, setting the C flag.

$$\begin{array}{lrrcl} & \overset{1}{\uparrow} & & & \\ \text{Borrow} & 0000 & 0001 & = & 01_{16} \\ & 0000 & 0010 & = & 02_{16} \\ \hline & 1111 & 1111 & = & FF_{16} \end{array}$$

Notice that the difference is FF_{16}. This will $\underset{\text{set/clear}}{\underline{\hspace{4em}}}$ the N flag.

11. Press the SS key and record the information below.

ACCA _ _   _ _ _ _
N Z V C

As expected, the difference produced is FF_{16}. Also, the N flag is set, indicating a negative number is in the accumulator and the C flag indicates a borrow occurred.

The next step will execute the instruction that loads 77_{16} into the accumulator. After this LDA operation, the C flag will be $\underset{\text{set/cleared}}{\underline{\hspace{4em}}}$.

12. Press the SS key and record the information below.

    ACCA _ _   _ _ _ _

                       N Z V C

    Notice that the C flag is still set and that 77_{16} is in the accumulator. Now we will subtract 66_{16} from the accumulator contents (77_{16}).

    Press the SS key and record the information below.

    ACCA _ _   _ _ _ _

                       N Z V C

    The difference (11_{16}) is now stored in the accumulator and, since no borrow is generated, the C flag is cleared.

13. In this step, the first instruction loads the accumulator with the number 49_{16}. The next instruction adds the number 60_{16} to 49_{16}. As shown below, the addition of these numbers causes an overflow into the sign bit (bit 7) and the sum, $A9_{16}$, appears to be a negative number.

$$\begin{array}{ll}
 & \begin{array}{llcl} 0100 & 1001 & = & 49_{16} \\ 0110 & 0000 & = & 60_{16} \\ \hline \underline{1}010 & 1001 & = & A9_{16} \end{array} \\
\text{Overflow changes sign bit.} \longrightarrow \uparrow &
\end{array}$$

Of course, this is incorrect and the MPU must be notified of this overflow. This is the purpose of the V flag.

Press the SS key and record the information below.

```
ACCA __   _ _ _ _
          N Z V C
```

The number 49_{16} is in the accumulator and the N, Z, V, and C flags are cleared.

Single-step once more and then record the information below.

```
ACCA __   _ _ _ _
          N Z V C
```

The sum $A9_{16}$ is now in the accumulator. Notice that the N **and** V flags are set, indicating that the number in the accumulator is negative and that an overflow occurred.

14. When the next instruction is executed, the number 10_{16} will be loaded into the accumulator.

Single-step the program and record the information below. Notice that the op code 3E (a halt) is the next instruction, so the program is finished.

```
ACCA __   _ _ _ _
          N Z V C
```

The accumulator contains the number 10_{16}, and all flags cleared. From this, you might conclude that any instruction that doesn't produce an overflow in the accumulator will $\underset{\text{set/clear}}{\underline{\hspace{3em}}}$ the V flag.

DISCUSSION

In this portion of the experiment, you stepped through a simple program that manipulated the condition code registers. In step 4, the negative number FF_{16} was loaded into the accumulator. This set the N flag to 1_2. In step 5, the positive number 77_{16} was loaded into the accumulator. And, as you noted, the N flag was cleared or reset to 0_2. From these two steps you should have concluded that when the number in the accumulator is negative, the N flag is set. And when the accumulator contains a positive number, the N flag is cleared.

In step 6, the accumulator was loaded with 00_{16}. This set the Z flag to 1_2. Next, when 01_{16} was loaded, the Z flag was reset or cleared to 0_2. Your conclusion should have been that when the accumulator contains 00_{16}, the Z flag is set. If it contains any number other than 00_{16}, the Z flag is cleared.

Next, you examined the C flag. When a carry was generated by the addition of the two numbers, 92_{16} and $C6_{16}$, the C flag was set. In step 8, you noted that merely loading a new number into the accumulator did not clear the C flag. The carry flag was cleared by another addition that did not result in a carry. Your conclusion should have been that the C flag can only be cleared by an arithmetic operation that does not result in a carry.

As you proved in steps 10 and 11, a subtraction that results in a borrow also sets the C flag. Again, the C flag was cleared by an arithmetic operation, in this case a subtraction, that did not generate a borrow. Therefore, the C flag can only be cleared or reset to 0_2 by an arithmetic operation that does not result in a borrow or carry.

You concluded this phase of the experiment by adding two positive numbers, the sum of which overflowed into the sign bit of the accumulator. This set the V or overflow flag, showing that the sum should not be a negative number as the N flag indicated. The next LDA instruction cleared the V flag. From this, you should conclude that the V flag is cleared by any instruction that doesn't produce an overflow.

In the next sections of this experiment, you will step through a few branching programs that illustrate the use of the branch always (BRA) instruction and certain conditional branch instructions. These branch instructions were discussed in Unit 4, and you will verify their operation. We'll begin with the multiply by repeated addition program.

PROCEDURE (Continued)

15. Enter the program listed in Figure 25 into the Trainer. This program multiplies 05_{16} and 02_{16} and stores the product in address 0013_{16}. Recheck the program to insure that it's entered correctly.

16. This is the same program that you stepped through in Unit 4. Notice that the program contains two branch instructions; the BEQ (Branch if Equal Zero) at address $0005_{16}$and the BRA (Branch Always) at address $000E_{16}$.

    The branch if equal zero (BEQ) instruction implies by its name that a conditional branch will occur when the ________ flag is set.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	4F	CLRA	Clear the accumulator.
0001	97	STA	Store the product
0002	13	13	in location 13_{16}.
0003	96	LDA	Load the accumulator with the
0004	12	12	multiplier from location 12_{16}.
0005	27	BEQ	If the multiplier is equal to zero,
0006	09	09	branch down to the Halt instruction.
0007	4A	DECA	Otherwise, decrement the multiplier.
0008	97	STA	Store the new value of the
0009	12	12	multiplier back in location 12_{16}.
000A	96	LDA	Load the accumulator with the
000B	13	13	product from location 13_{16}.
000C	9B	ADD	Add
000D	11	11	the multiplicand to the product.
000E	20	BRA	Branch back to instruction
000F	F1	F1	in location 01.
0010	3E	HLT	Halt.
0011	05	05	Multiplicand.
0012	02	02	Multiplier.
0013	—	—	Product.

Figure 25
Program to multiply by repeated addition.

17. Now, set the program counter to 0000 and single-step through the program, recording the information in the chart of Figure 26. Notice that you will be monitoring the Z flag. A comments column is provided so you can make notes about each step. Use the program listing as a reference for each op code and the corresponding operand.

18. When the BEQ instruction is executed and the Z flag is set, the program branches to the ______ instruction.

When the multiplier was 02_{16}, the program halted on the ______ pass through the program.

If the multiplier is changed to 06_{16}, how many passes would the program make before it halts? ______.

19. Examine the contents of address 0013_{16} and record below.

0013 _ _.

STEP	PROGRAM COUNTER	OPCODE	ACCA	Z FLAG	COMMENTS
1					
2					
3					
4					
5					
6					
7					
8					
9					
10					
11					
12					
13					
14					
15					
16					
17					
18					
19					
20					

Figure 26
Single-stepping through the Multiply by repeated addition program.

DISCUSSION

The chart that you completed should be similar to the one shown in Figure 27. Compare the charts.

The first step we don't see, since it's executed before the Trainer stops at address 0001. Nevertheless, we do see the result of this clear accumulator instruction because the accumulator contains 00. When step 1 is executed, 00_{16} is stored in location 0013_{16}. Step 2 brings us to address 0003_{16} which loads the accumulator with the multiplier, in this example, 02_{16}. The BEQ instruction is next, but the Z flag is cleared so the program continues on the normal route. Next the multiplier is decremented to 01_{16} and then stored in location 0012_{16}. Now the product (00_{16}) is loaded and the multiplicand (05_{16}) is added directly. This produces the new product, 05_{16}. Now the program encounters the BRA, or branch always instruction and it branches back to address 0001_{16}.

Here the new product is stored away and the multiplier is loaded again. It's 01_{16} this time, so the program continues on through the BEQ instruction, the multiplier is decremented to 00_{16}, and the multiplicand 05_{16} is added to the product. The new product ($0A_{16}$) is still in the accumulator. Once again, the BRA instruction loops flow back to address 0001_{16} and the product is stored in address 0013_{16}.

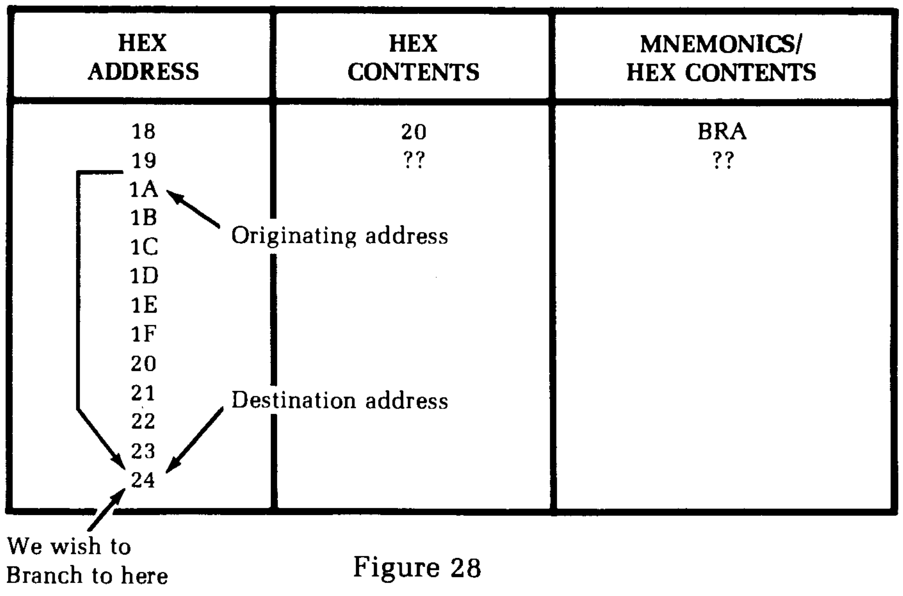
The multiplier is now loaded and, since it's been decremented to 00_{16}, it sets the Z flag. The BEQ instruction checks the Z flag, finds that it's set and branches to the halt instruction at address 0010_{16}. Therefore, the program makes two complete passes, before the multiplier becomes 00_{16}. On the third pass through, BEQ terminates the program because the Z flag is set.

The multiplier sets the count and determines how many additions will be performed. If the multiplier is changed to 06_{16}, the program will make six complete loops, halting on the seventh loop. The BEQ will only be satisfied when the multiplier has been reduced to 00.

All branch instructions use relative addressing. In Unit 4, we discussed the method used to calculate the destination address for a branch instruction. However, another shorthand type procedure that's quite popular with programmers can be used. With this technique, you simply count in hexadecimal. For a forward branch, you begin at 00_{16} and count up to the destination address.

STEP	PROGRAM COUNTER	OPCODE	ACCA	Z FLAG	COMMENTS
1	0001	97	00	1	Store the product (oo$_{16}$) in address 0013$_{16}$.
2	0003	96	00	1	Load the accumulator with the multiplier (02$_{16}$) from address 0012$_{16}$.
3	0005	27	02 ↑ Multiplier	0	BEQ. Check the Z flag. It's not set so continue.
4	0007	4A	02	0	Decrement the multiplier (02$_{16}$).
5	0008	97	01 ↑ New Multiplier	0	Store the new multiplier (01$_{16}$) at address 0012$_{16}$.
6	000A	96	01	0	Load the accumulator with the product (00) at address 0013$_{16}$.
7	000C	9B	00	1	Add the multiplicand (05) giving new product.
8	000E	20	05 ↑ New Product	0	Branch back to address 0001$_{16}$.
9	0001	97	05	0	Store the product (05$_{16}$) in address 0013$_{16}$.
10	0003	96	05	0	Load the accumulator with the multiplier (01$_{16}$) located at address 0012$_{16}$.
11	0005	27	01	0	BEQ. Check Z flag. It's not set so continue.
12	0007	4A	01	0	Decrement the multiplier (01$_{16}$).
13	0008	97	00 ↑ New Multiplier	1	Store the new Multiplier (00$_{16}$) at address 0012$_{16}$.
14	000A	96	00	1	Load the accumulator with the product (05$_{16}$) at address 0013$_{16}$.
15	000C	9B	05	0	Add the multiplicand (05$_{16}$) giving new product.
16	000E	20	OA ↑ New Product	0	Branch back to address 0001$_{16}$.
17	0001	97	OA	0	Store the product (OA$_{16}$) in address 0013$_{16}$.
18	0003	96	OA	0	Load the accumulator with the multiplier (00$_{16}$) from address 0012$_{16}$.
19	0005	27	00	1	BEQ. Check the Z flag. Now it's set. Branch to address 0010$_{16}$.
20	0010	3E	00	1	Halt.

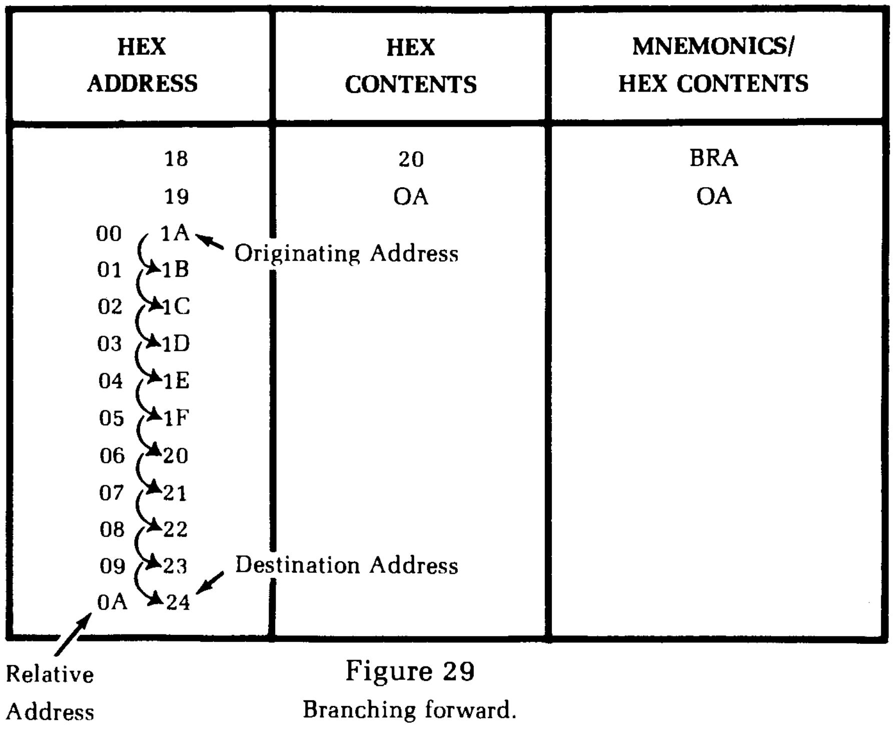
Figure 27



HEX ADDRESS	HEX CONTENTS	MNEMONICS/ HEX CONTENTS
18	20	BRA
19	??	??
1A		
1B		
1C		
1D		
1E		
1F		
20		
21		
22		
23		
24		

Figure 28

For example, in the program of Figure 28, we want to branch from address 18_{16} to address 24_{16}. Recall that the relative address is added to the contents of the program counter. After the BRA instruction and its operand (the relative address) have been fetched, the program counter is pointing to address $1A_{16}$. Therefore, we begin our count at address $1A_{16}$. Then we count forward in hex as shown in Figure 29. When we reach the destination address, the hexadecimal count is the relative address. In this case, it's $0A_{16}$, and we insert this operand at address 19_{16}.



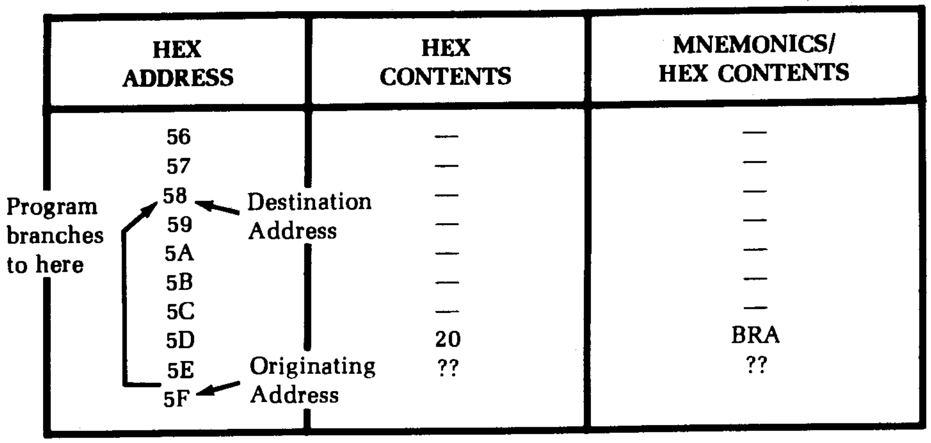
	HEX ADDRESS	HEX CONTENTS	MNEMONICS/ HEX CONTENTS
	18	20	BRA
	19	OA	OA
00	1A		
01	1B		
02	1C		
03	1D		
04	1E		
05	1F		
06	20		
07	21		
08	22		
09	23		
0A	24		

Figure 29
Branching forward.

To branch backward in the program, we simply count down using negative hex numbers. It may sound more difficult, but once you are accustomed to it, you will find it easier to use than the previous method you learned.

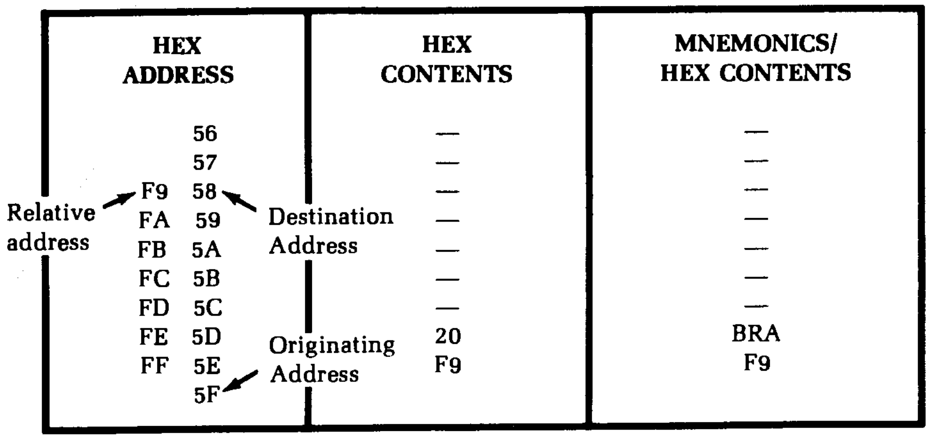
For example, in the program shown in Figure 30A, we wish to branch back to address 58_{16}. The BRA instruction, at address $5D_{16}$ is fetched and the program count points to address $5F_{16}$. Figure 30B shows how we calculate the address for this backward branch. We begin with FF_{16}, and count down. When we reach the destination address (58_{16}), the count at that point is the relative address, in this case $F9_{16}$.

A


Program branches to here
Destination Address
Originating Address

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ HEX CONTENTS
56	—	—
57	—	—
58	—	—
59	—	—
5A	—	—
5B	—	—
5C	—	—
5D	20	BRA
5E	??	??
5F		

B


Relative address
Destination Address
Originating Address

	HEX ADDRESS	HEX CONTENTS	MNEMONICS/ HEX CONTENTS
	56	—	—
	57	—	—
F9	58	—	—
FA	59	—	—
FB	5A	—	—
FC	5B	—	—
FD	5C	—	—
FE	5D	20	BRA
FF	5E	F9	F9
	5F		

Figure 30
Branching back.

Figure 31 shows another example of computing the relative address for a larger branch. The branch instruction is at address BO_{16} and therefore, the origination address is $B2_{16}$. We calculate the relative address as shown in Figure 31B. Starting with FF_{16} at address $B1_{16}$, we count down to the destination address AO_{16}. As the count indicates, the relative address to get to AO_{16} is EE_{16}.

We wish to branch to here

Destination Address

Originating Address

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ HEX CONTENTS
AO	—	—
A1	—	—
A2	—	—
A3	—	—
A4	—	—
A5	—	—
A6	—	—
A7	—	—
A8	—	—
A9	—	—
AA	—	—
AB	—	—
AC	—	—
AD	—	—
AE	—	—
AF	—	—
BO	26	BNE
B1	??	??
B2		

A

Relative Address

HEX ADDRESS		HEX CONTENTS	MNEMONICS/ HEX CONTENTS
EE	AO	—	—
EF	A1	—	—
FO	A2	—	—
F1	A3	—	—
F2	A4	—	—
F3	A5	—	—
F4	A6	—	—
F5	A7	—	—
F6	A8	—	—
F7	A9	—	—
F8	AA	—	—
F9	AB	—	—
FA	AC	—	—
FB	AD	—	—
FC	AE	—	—
FD	AF	—	—
FE	BO	26	BNE
FF	B1	EE	EE
	B2		

B

Figure 31

In the next section of this experiment, you will write a program that will divide by repeated subtraction. You will probably have two branches in this program; a forward branch and a branch back. Use this new technique to calculate the relative addresses for both branches.

PROCEDURE (Continued)

20. In Unit 4, we discussed a program that divides by repeated subtraction. The flow chart for this program is shown in Figure 32. Using this flow chart as a guide and the instructions presented in Figure 33, write a program that divides by repeated subtraction.

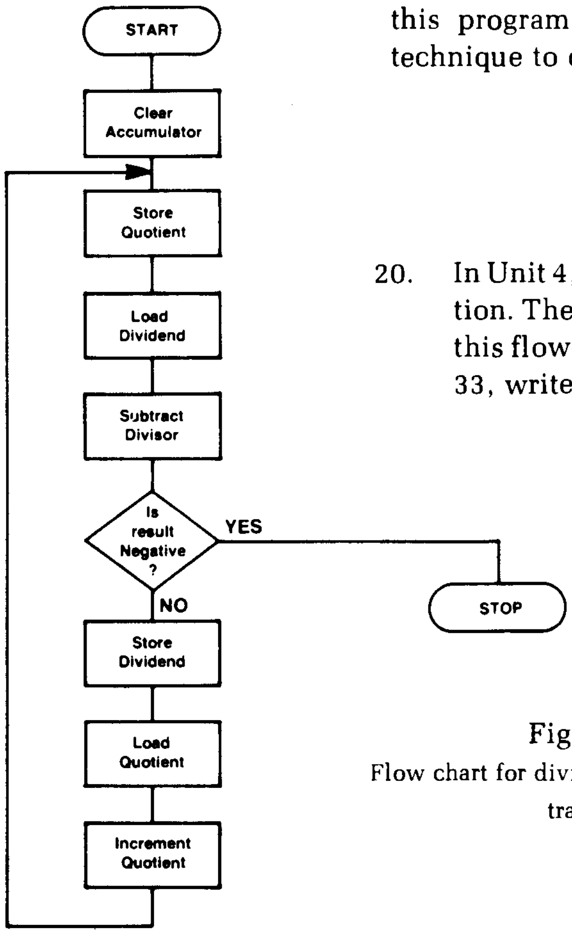


Figure 32
Flow chart for dividing by repeated subtraction.

INSTRUCTION	MNEMONIC	ADDRESSING MODE			
		IMMEDIATE	DIRECT	RELATIVE	INHERENT
Load Accumulator	LDA	86	96		
Clear Accumulator	CLRA				4F
Decrement Accumulator	DECA				4A
Increment Accumulator	INCA				4C
Store Accumulator	STA		97		
Add	ADD	8B	9B		
Subtract	SUB	80	90		
Branch Always	BRA			20	
Branch if Carry Set	BCS			25	
Branch if Equal Zero	BEQ			27	
Branch if Minus	BMI			2B	
Halt	HLT				3E

Figure 33
Instructions to be used.

21. Now load the program into the Trainer. Let the dividend be $0B_{16}$ and the divisor be 05_{16}. Change the program counter to the starting address of your program and single-step through the program, recording the information in the chart of Figure 34.

22. Examine the contents of the address that stores the dividend and the quotient. If you followed the flow chart, the address where the dividend is stored should now contain the remainder from the division. Record the contents below.

Quotient ______ ______ Remainder ______ ______

STEP	PROGRAM COUNTER	OPCODE	ACCA	N FLAG.	COMMENTS

Figure 34

DISCUSSION (Continued)

Now you've written a program that incorporates an unconditional branch and a conditional branch. Hopefully, you calculated the relative addresses using the shorthand technique just discussed. Our program for the divide by repeated subtraction is listed in Figure 35. If you followed the flow chart, your program should be similar to this.

HEX ADDRESS	HEX CONTENTS	MNEMONIC/HEX CONTENTS	COMMENTS
0000	4F	CLRA	Clear the accumulator.
0001	97	STA	Store in the quotient which
0002	13	13	is at address location 13_{16}.
0003	96	LDA	Load the accumulator with the
0004	11	11	dividend from location 11_{16}.
0005	90	SUB	Subtract the
0006	12	12	divisor from the dividend.
0007	2B	BMI	If the difference is negative,
0008	07	07	branch down to the Halt instruction.
0009	97	STA	Otherwise, store the difference
000A	11	11	back in location 11_{16}.
000B	96	LDA	Load the accumulator with the
000C	13	13	quotient.
000D	4C	INCA	Increment the quotient by one.
000E	20	BRA	Branch back to instruction
000F	F1	F1	in location 01.
0010	3E	HLT	Halt.
0011	OB	OB	Dividend (11_{16}).
0012	05	05	Divisor (5_{16}).
0013	—	—	Quotient.

Figure 35
Dividing by repeated subtraction.

Notice that we used the BMI (Branch if Minus) conditional branch instruction. Therefore, the N or negative flag will satisfy the branch when it's set. Figure 36 charts our program as we single-stepped through it. Since the program subtracts the divisor from the dividend and stores the difference as the new dividend, at the conclusion of the program the dividend is actually the remainder of the division. When $0B_{16}$ is divided by 05_{16}, the quotient should be 02_{16} and the remainder 01_{16}.

STEP	PROGRAM COUNTER	OPCODE	ACCA	N FLAG	COMMENTS
1	0001	97	00	0	Store the quotient (00_{16}) at address 0013_{16}.
2	0003	96	00	0	Load the accumulator with the dividend from address 0011_{16}.
3	0005	90	OB ↑ Dividend	0	Subtract the divisor (05_{16}) at address 0012_{16} from the accumulator.
4	0007	2B	06 ↑ After subtraction	0	BMI. Check the N flag. It's not set so continue.
5	0009	97	06	0	Store the difference (06_{16}) back in address 0011_{16}.
6	000B	96	06	0	Load the accumulator with the quotient (00_{16}) at address 0013_{16}.
7	000D	4C	00	0	Increment the quotient.
8	000E	20	01 ↑ Quotient after INC	0	Branch back to the instruction at address 0001_{16}.
9	0001	97	01	0	Store the quotient (01_{16}) at address 0013_{16}.
10	0003	96	01	0	Load the accumulator with the dividend (06_{16}) at address 0011_{16}.
11	0005	90	06 ↑ Dividend Now	0	Subtract the divisor (05_{16}) at address 0012_{16} from the accumulator.
12	0007	2B	01 ↑ After Subtraction	0	BMI. Check the N flag. It's not set so continue.
13	0009	97	01	0	Store the difference (01_{16}) back in address 0011_{16}.
14	000B	96	01	0	Load the accumulator with the quotient (01_{16}) at address 0013_{16}.
15	000D	4C	01	0	Increment the quotient.
16	000E	20	02 ↑ Quotient after INC.	0	Branch back to the instruction at address 0001_{16}.
17	0001	97	02	0	Store the quotient (02_{16}) at address 0013_{16}.
18	0002	96	02	0	Load the accumulator with the dividend (01_{16}) at address 0011_{16}.
19	0005	90	01	0	Subtract the divisor (05_{16}) at address 0012_{16} from the accumulator.
20	0007	2B	FC ↑ Negative Number	1	BMI. Check the N flag. Now it's set so branch to the instruction at address 0010_{16}.
21	0010	3E	FC	1	Halt.

Figure 36

So far, we've used the conditional branch instructions only to exit a loop and then halt program execution. However, these branch instructions become even more powerful when they are used to "chain" together different portions of a program. Figure 37 shows an example of this chaining effect. The program starts and runs through the first loop until the conditional branch BEQ is satisfied. Then it exits this loop and starts another. When the BEQ condition is satisfied in the second loop, another exit is performed, and another portion of the program is executed.

A strategically placed conditional branch at the end of the program can cause a branch back to the beginning that will repeat the program again and again. In the next portion of this experiment, you will load the BCD-to-binary conversion program that you studied earlier. Then you will step through the program and watch as the Trainer executes each instruction.

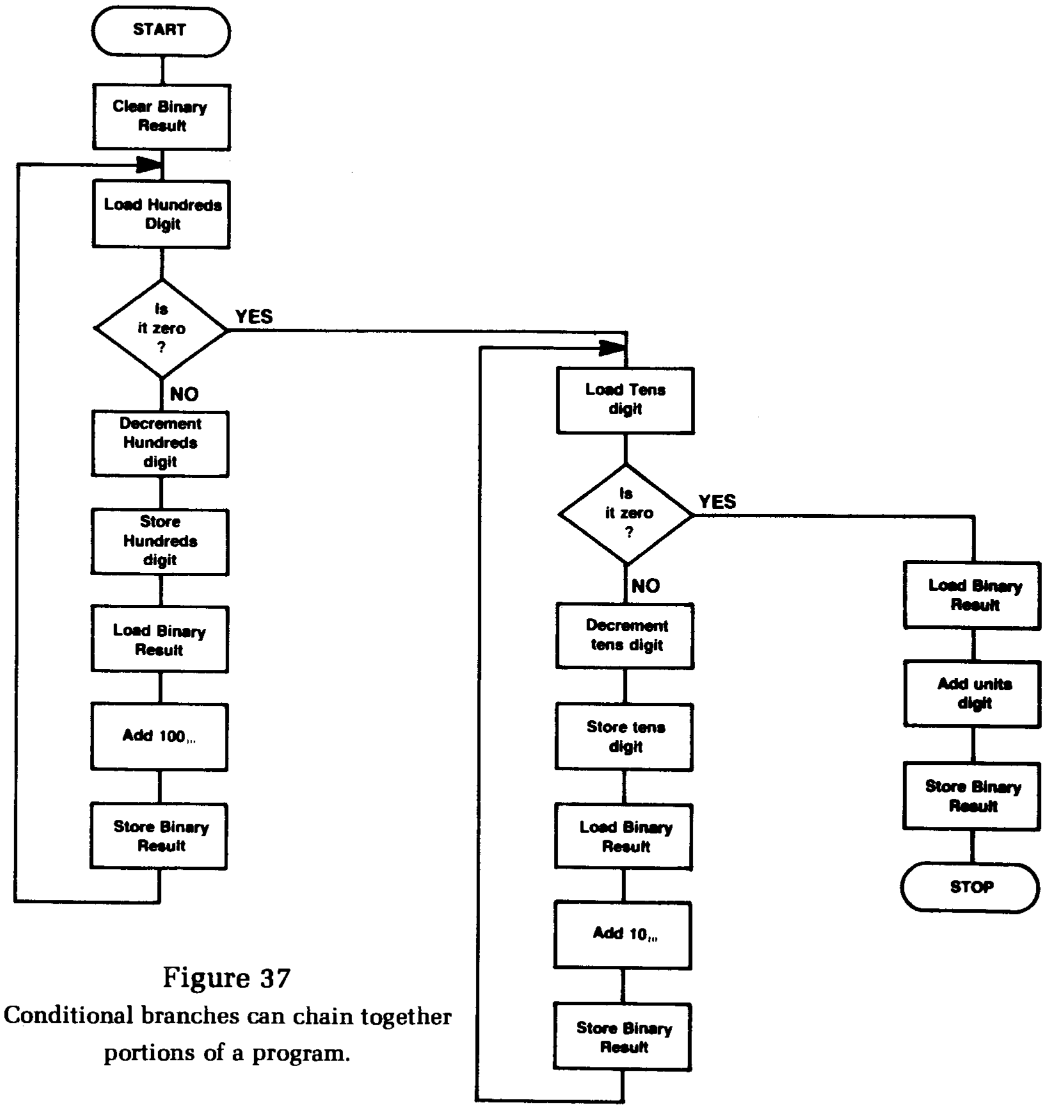


Figure 37
Conditional branches can chain together portions of a program.

PROCEDURE (Continued)

23. Load the program listed in Figure 38 into the Trainer. The BCD number 117_{10} will be converted to binary by this program.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	4F	CLRA	Clear the Accumulator.
0001	97	STA	Store 00
0002	2B	2B	in location 2B. This clears the binary result.
0003	96	LDA	Load direct into the accumulator
0004	28	28	the hundreds BCD digit.
0005	27	BEQ	If the hundreds digit is zero, branch
0006	OB	OB	forward to the instruction in location 12_{16}.
0007	4A	DECA	Otherwise, decrement the accumulator.
0008	97	STA	Store the result as the new
0009	28	28	hundreds BCD digit.
000A	96	LDA	Load direct into the accumulator
000B	2B	2B	the binary result.
000C	8B	ADD	Add immediate
000D	64	64	100_{10} to the binary result.
000E	97	STA	Store away the new
000F	2B	2B	binary result.
0010	20	BRA	Branch
0011	F1	F1	back to the instruction in location 03_{16}.
0012	96	LDA	Load direct into the accumulator
0013	29	29	the tens BCD digit.
0014	27	BEQ	If the tens BCD digit is zero, branch
0015	OB	OB	forward to the instruction in location 21_{16}.
0016	4A	DECA	Otherwise, decrement the accumulator.
0017	97	STA	Store the result as the new
0018	29	29	tens BCD digit.
0019	96	LDA	Load direct into the accumulator
001A	2B	2B	the binary result.
001B	8B	ADD	Add immediate
001C	OA	OA	10_{10} to the binary result.
001D	97	STA	Store away the new
001E	2B	2B	binary result.
001F	20	BRA	Branch
0020	F1	F1	back to the instruction in location 12_{16}.
0021	96	LDA	Load direct into the accumulator
0022	2B	2B	the binary result.
0023	9B	ADD	Add direct
0024	2A	2A	the units BCD digit.
0025	97	STA	Store away the new
0026	2B	2B	binary result.
0027	3E	HLT	Halt.
0028	01	01	Hundreds BCD digit.
0029	01	01	Tens BCD digit.
002A	07	07	Units BCD digit.
002B	—	—	Reserved for the binary result.

Figure 38
Program for converting BCD to binary.

The BEQ instruction is used for the conditional branches in this program. This means that MPU will monitor the ____ flag to determine if the condition is set.

24. Now set the program counter to 0000 and single-step through the program recording the information in the chart of Figure 39. Notice that, at strategic steps, you should stop and answer questions before you continue.

25. What is the hundreds BCD digit at this time? ______ The result is now 64_{16}, which is ______ in the decimal number system.

Now return to the Trainer and continue stepping through the program.

26. What is the tens BCD digit at this time? ______.

The result is now $6E_{16}$. This is the equivalent of ____ in the decimal number system.

Now return to the Trainer and step through the remainder of the program.

27. Examine address $002B_{16}$ and record the result below.

$\_\_\_\_\_\__{16}$

Convert this number to its decimal equivalent.

$75_{16} = \_\_\_\_\_\__{10}$

STEP	PROGRAM COUNTER	OPCODE	ACCA	Z FLAG	COMMENTS
1					
2					
3					
4					
5					
6					
7					
8					
Stop! Return to Step 25.					
9					
10					
11					
12					
13					
14					
15					
16					
17					
18					
Stop! Return to step 26.					
19					
20					
21					
22					
23					
24					
25					

Figure 39

DISCUSSION

Now you've verified the operation of the BCD-to-binary conversion program. The chart that you completed should match the one shown in Figure 40.

Since the BEQ instruction is used for the conditional branches in the program, we monitored the Z flag. In this example, the BCD number 117_{10} was converted to its binary equivalent 75_{16}. This program will convert BCD numbers as high as 255_{10}, to their binary equivalent.

The program isn't as complicated as it might appear. The hundreds and tens BCD digits are used to set a count. Each pass through a loop decrements the BCD digit, or count, and then adds the equivalent hexadecimal positional value for that BCD digit. For example, in the hundreds conversion loop, 64_{16} is added to the binary result for each hundreds BCD digit. Hence, the BCD digit sets the count. Then the count is decremented by one and the program loops back and runs through again. When the count is zero, that BCD digit has been added the correct number of times and the program branches off to another loop. This continues until the program halts.

Stepping through the program, you found that after Step 8, the Trainer had completed one loop through the hundreds BCD portion of the program. The count was 00_{16} and the binary result was 64_{16}, or the binary equivalent of 100_{10}. On the next pass through, the program branches to the tens BCD loop.

The first loop through, the tens BCD portion of the program was completed at step 18. The binary result was $6E_{16}$, which is the equivalent of 110_{10}. The tens BCD digit had been decremented to 00_{16}. Then all that remained was to add the units BCD digit (07_{10}) and the conversion process was complete.

You verified the final result by checking the binary result at location $002B_{16}$. Here you found the hex number 75_{16}. When you converted this number to its decimal equivalent, you found that 75_{16} equals 117_{10}. Also, if you converted 75_{16} to binary, you would find the number $0111\ 0101_2$; which is the (binary) equivalent of 117_{10}, so the program works.

The most frequent mistake made by programmers when using the branch instructions is the improper computation of the relative address. An improperly coded relative address not only prevents the program from executing properly, but can even wipe out portions of the program. In the next section of this experiment, you will witness the result of an incorrect relative address and the effect it has on the program. In this example, we will use the binary-to-BCD conversion program you studied earlier.

STEP	PROGRAM COUNTER	OPCODE	ACCA	Z FLAG	COMMENTS
1	0001	97	00	1	Store 00 in address 002B$_{16}$. This clears the binary result.
2	0003	96	00	1	Load the accumulator with the Hundreds BCD digit (01$_{16}$).
3	0005	27	Hundreds BCD→ Digit 01	0	BEQ. Check the Z flag. It's clear so continue.
4	0007	4A	01	0	Decrement the BCD Hundreds Digit.
5	0008	97	New→ Hundreds Digit 00	1	Store the new Hundreds Digit (00).
6	000A	96	00	1	Load the accumulator with the Binary Result (00$_{16}$).
7	000C	8B	00	1	Add to the binary result 64$_{16}$.
8	000E	97	Binary→ Result Now 64	0	Store away the new binary result.
9	0010	20	64	0	Branch back to address 0003$_{16}$.
10	0003	96	64	0	Load the accumulator with the Hundreds BCD digit (00).
11	0005	27	00	1	BEQ. Check the Z flag. It's set so branch to address 0012$_{16}$.
12	0012	96	00	1	Load the accumulator with the tens BCD digit (01$_{16}$).
13	0014	27	Tens BCD→ Digit 01	0	BEQ. Check the Z flag. It's clear so continue.
14	0016	4A	01	0	Decrement the tens BCD digit (01$_{16}$).
15	0017	97	New Tens→ Digit 00	1	Store the new tens BCD digit.
16	0019	96	00	1	Load the accumulator with the binary result (64$_{16}$).
17	001B	8B	64	0	Add OA$_{16}$ to the binary result.
18	001D	97	New Binary→ Result 6E	0	Store away the new binary result.
19	001F	20	6E	0	Branch back to address 0012$_{16}$.
20	0012	96	6E	0	Load the accumulator with the tens BCD digit (00).
21	0014	27	00	1	BEQ. Check the Z flag. It's set so branch to address 0021$_{16}$.
22	0021	96	00	1	Load the accumulator with the binary result (6E$_{16}$).
23	0023	9B	6E	0	Add the units BCD digit (07$_{16}$).
24	0025	97	New Binary→ Result 75	0	Store the new binary result (75$_{16}$).
25	0027	3E	75	0	Halt.

Figure 40
Single-stepping through the BCD-to-binary conversion program.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	4F	CLRA	Clear the accumulator.
0001	97	STA	Store 00
0002	2B	2B	in location $002B_{16}$. This clears the hundreds digit.
0003	97	STA	Store 00.
0004	2C	2C	in location $002C_{16}$. This clears the tens digit.
0005	97	STA	Store 00.
0006	2D	2D	in location $002D_{16}$. This clears the units digit.
0007	96	LDA	Load direct into the accumulator
0008	2A	2A	the binary number to be converted.
0009	80	SUB	Subtract immediate
000A	64	64	100_{16}.
000B	25	BCS	If a borrow occurred, branch
000C	0A	0A	forward to the instruction in location 0016_{16}.
000D	97	STA	Otherwise, store the result of the subtraction
000E	2A	2A	as the new binary number.
000F	96	LDA	Load direct into the accumulator
0010	2B	2B	the hundreds digit of the BCD result.
0011	4C	INCA	Increment the hundreds digit.
0012	97	STA	Store the hundreds digit
0013	2B	2B	back where it came from.
0014	20	BRA	Branch
0015	F1	F1	back to the instruction at address 0007_{16}.
0016	96	LDA	Load direct into the accumulator
0017	2A	2A	the binary number.
0018	80	SUB	Subtract immediate
0019	0A	0A	10_{16}.
001A	25	BCS	If a borrow occurred, branch
001B	09	09	forward to the instruction in location 0025_{16}.
001C	97	STA	Otherwise, store the result of the subtraction
001D	2A	2A	as the new binary number.
001E	96	LDA	Load direct into the accumulator
001F	2C	2C	the tens digit.
0020	4C	INCA	Increment the tens digit.
0021	97	STA	Store the tens digit.
0022	2C	2C	back where it came from.
0023	20	BRA	Branch
0024	F1	F1	back to the instruction at address 0016_{16}.
0025	96	LDA	Load direct into the accumulator
0026	2A	2A	the binary number.
0027	97	STA	Store it in
0028	2D	2D	the units digit.
0029	3E	HLT	Halt.
002A	75	75	Place binary number to be converted at this address.
002B	—	—	Hundreds digit } Reserved for BCD result.
002C	—	—	Tens digit } Reserved for BCD result.
002D	—	—	Units digit } Reserved for BCD result.

Figure 41
A program with an incorrect relative address.

PROCEDURE (Continued)

28. Load the program listed in Figure 41 into the Trainer. This program should convert the binary number 0111 0101_2 (75_{16}) into its BCD equivalent. However, one of the relative addresses is **incorrect**. Part of this exercise is to locate the incorrect relative address and correct it.

29. Now set the program counter to 0000 and single-step through the program. Record the results in the chart of Figure 42. Notice that we're monitoring the carry (C) flag because the program uses the BCS (Branch if Carry Set) instruction.

STEP	PROGRAM COUNTER	OPCODE	ACCA	C FLAG	COMMENTS
1					
2					
3					
4					
5					
6					
7					
8					
9					
10					
11					
12					
13					
14					
15					
16					
17					
18					
19					
20					

Figure 42
Single-Stepping through the binary-to-BCD conversion program.

30. Examine addresses $002B_{16}$, $002C_{16}$, and $002D_{16}$; record the results below.

    002B __ __ Hundreds BCD Digit

    002C __ __ Tens BCD Digit

    002D __ __ Units BCD Digit

    Obviously, there is something wrong with the program. Although the hundreds and tens digits are believable, the units digit of 11 is impossible. Remember, a decimal number can only have a units digit of from 0 to 9_{10}.

31. Use the program listing and the chart that you've compiled and locate the error in the program. Then record the address of the instruction below.

    HINT: The problem is with the relative address for one of the branch instructions. When one of these addresses is incorrect, the program branches to the wrong address, possibly skipping portions of the program. Therefore, first determine the portions of the program that produced the wrong result and work back until you find the problem.

    Address __ __ __ __ Incorrect Relative Address __ __

32. Now calculate the correct relative address (operand) and record it below.

    Correct Relative Address __ __.

DISCUSSION

This exercise should have demonstrated the versatility of your Trainer to assist you in "debugging" programs. When you examined addresses $002B_{16}$, $002C_{16}$, and $002D_{16}$, you found these results.

002B	0 1	Hundreds BCD Digit
002C	0 0	Tens BCD Digit
002D	1 1	Units BCD Digit

Obviously, the units BCD digit is incorrect. Since the units digit is wrong, we begin to debug at this portion of the program. This happens to be the least complex section of the program because the binary number is simply loaded into the accumulator and stored in address $002D_{16}$. Comparing the chart that you compiled against the program listing, we find that this portion of the program seems to be executing correctly.

Therefore, we move back to the tens BCD digit portion of the program. Checking the program listing, we find that the tens BCD portion of the program begins at address 0016_{16}. But as the chart in Figure 43 shows, when the program is single-stepped the tens BCD digit loop actually starts at address 0017_{16}. This is the wrong address. We find the problem when we move back to step 14 of the chart. This is the BCS (Branch if Carry Set) instruction at address $000B_{16}$. However, instead of branching to address 0016_{16} as the comments column suggests, the program goes to address 0017_{16}. Therefore, the relative address at address $000C_{16}$ must be incorrect. When we check this relative address, we find that it should be 09_{16}, instead of $0A_{16}$.

But, how did this incorrect operand affect the program? Following the chart in Figure 43, we find that the hundreds BCD portion of the program worked correctly. On the second loop through this portion of the program, the subtraction resulted in a borrow and the C flag was set. Hence, the BCD instruction produced the desired branch.

But, instead of branching to address 0016_{16}, where we would have found a load accumulator instruction (96_{16}) with an operand of $2A_{16}$, the program branches to address 0017_{16}. The Trainer now interprets the operand ($2A_{16}$) as an instruction or op code. The op code 2A, as you may recall, represents a valid instruction which is "Branch if Plus." The MPU checks the N flag and finds it set, because at this time, the negative number AD_{16} is in the accumulator. Therefore, the condition is not satisfied, and the Trainer continues on to the next instruction.

Single-stepping again (now we are at step 16) the next op code is 0A. Actually, this should be the operand for the subtract instruction at address 0018_{16}. But since we are off by one, it appears to be the op code. The Trainer checks the op code 0A and finds that it's an inherent instruction to "clear the overflow flag." It executes this instruction.

Step 17 finds the program at address $001A_{16}$. Here, we encounter another BCS conditional branch instruction. The C flag is still set so we branch to address 0025_{16}. The program works properly from this point on.

Therefore, this one incorrect relative address caused the program to skip the tens BCD portion of the program. The tens unit was never subtracted, so it carried over into the units BCD digit. This produced the wrong units digit of 11_{10}.

STEP	PROGRAM COUNTER	OPCODE	ACCA	C FLAG	COMMENTS
1	0001	97	00	0	Store 00 in Hundreds Digit.
2	0003	97	00	0	Store 00 in tens Digit.
3	0005	97	00	0	Store 00 in units Digit.
4	0007	96	00	0	Load the accumulator with the Binary number (75_{16}).
5	0009	80	75	0	Subtract 64_{16} from accumulator
6	000B	25	11	0	BCS. Check C flag for borrow. It's clear so continue.
7	000D	97	11	0	Store away the new binary number.
8	000F	96	11	0	Load the accumulator with the Hundreds Digit (00).
9	0011	4C	00	0	Increment the Hundreds Digit.
10	0012	97	01	0	Store the Hundreds Digit.
11	0014	20	01	0	Branch back to address 0007_{16}.
12	0007	96	01	0	Load the accumulator with the Binary Number (11_{16}).
13	0009	80	11	0	Subtract 64_{16} from accumulator. BCS. Check C Flag for borrow.
14	000B	25	AD	1	It's set so branch to address 0016_{16}.
15	$\boxed{\text{Tens BCD}}$ 0017	← Wrong Address 2A	AD	1	What's this?
16	0019	OA	AD	1	
17	001A	25	AD	1	BCS. Check C Flag. It's still set so branch to address 0025_{16}.
18	$\boxed{\text{Units BCD}}$ 0025	96	AD	1	Load the accumulator with the Binary number.
19	0027	97	11	1	Store it in the units Digit.
20	0029	3E	11	1	Halt.

Figure 43
Locating the incorrect relative address.

PROCEDURE (Continued)

33. Now change the operand at address $000C_{16}$ from $0A_{16}$ to 09_{16}.

34. Also change the number at address $002A_{16}$ to 75_{16}. This is the number that the program will convert to its BCD equivalent.

35. Reset the program counter to 0000 and single-step through the program comparing the program listing with the results that you obtain.

36. Examine the addresses listed below and record the information stored there.

002B	__ __	Hundreds BCD Digit
002C	__ __	Tens BCD Digit
002D	__ __	Units BCD Digit

Is this the correct BCD representation for the number 75_{16}? ______.

DISCUSSION

When the program is corrected by inserting the relative address (09_{16}) at address $000C_{16}$, we find that it works perfectly. After single-stepping through the program, we examine the BCD digits stored at addresses $002B_{16}$, $002C_{16}$, and $002D_{16}$. The hundreds digit is 01_{10}, the tens digit is 01_{10}, and the units digit is 07_{10}. Therefore, the BCD equivalent of the binary number $0111\ 0101_2$ (75_{16}) is 117_{10}.

Experiment 6

Additional Instructions

OBJECTIVES:

To verify the operation of the ADC instruction when used in a multiple-precision addition program.

To investigate the hazard of using the ADC instruction when a carry is not desired.

To demonstrate your ability to write a multiple-precision subtraction program using the SBC instruction.

To demonstrate your ability to write a routine that will multiply any 4-bit binary number times 16_{10} using the ASLA instruction.

To verify the operation of a BCD packing program that uses the ASLA instruction.

To verify the operation of the DAA instruction when used in a BCD multiple-precision addition program.

INTRODUCTION

One of the measures of a microprocessor's power is the size of the instruction set. In other words, more instructions generally mean more potential power. You saw the economy that resulted with the addition of branch instructions in the previous experiment. In this experiment, we will examine four additional instructions; the ADC or add with carry, the SBC or subtract with carry, the ASLA or arithmetic shift accumulator left, and the DAA or decimal adjust accumulator.

The discussion in Unit 4 explained the purpose of each instruction. In this experiment, we will restrict our activity to verifying that each instruction works as explained.

In the previous experiment, you examined the condition code registers and how the MPU monitors these flag registers to initiate conditional branches. Yet, these condition code registers are also monitored for other instructions. For example, the ADC (add with carry) and SBC (subtract with carry) instructions key on the C or carry flag. If an ADC instruction is executed and the carry flag is set, one is added to the least significant bit in the accumulator. Likewise, if the C flag is set when an SBC instruction is executed, one is subtracted from the least-significant bit of the accumulator. Remember, the C flag represents a "borrow" to the subtract instruction.

In the first portion of this experiment, we will verify the operation of the ADC instruction with a program for multiple precision arithmetic. Then we will examine one of the hazards of using this instruction.

PROCEDURE

1. Turn on the Trainer and press the RESET key.

2. Load the program listed in Figure 44 into the Trainer. This program performs multiple-precision addition of two 16_{10} bit numbers. The augend $1B93_{16}$ will be added to the addend $C0EA_{16}$ by this program. Of course, the program can add any numbers that are 16_{10} bits or less.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	01	NOP	No operation
0001	96	LDA	Load the accumulator direct with the
0002	OE	OE	least significant byte of the addend.
0003	9B	ADD	Add direct the
0004	10	10	least significant byte of the augend.
0005	97	STA	Store the result in the
0006	12	12	least significant byte of the sum.
0007	96	LDA	Load the accumulator direct with the
0008	OF	OF	most significant byte of the addend.
0009	99	ADC	Add with carry direct the
000A	11	11	most significant byte of the augend.
000B	97	STA	Store the result in the
000C	13	13	most significant byte of the sum.
000D	3E	HLT	Halt
000E	EA	EA	Least significant byte } addend
000F	CO	CO	Most significant byte } addend
0010	93	93	Least significant byte } augend
0011	1B	1B	Most significant byte } augend
0012	—	—	Least significant byte } sum
0013	—	—	Most significant byte } sum

Figure 44
Program for multiple-precision addition.

3. Change the program counter to 0000 and single-step through the program, recording the information in the chart of Figure 45. Notice that we are monitoring the carry (C) flag.

STEP	PROGRAM COUNTER	OPCODE	ACCA	C FLAG	COMMENTS
1					
2					
3					
4					
5					
6					
7					

Figure 45

4. Examine memory location 0012_{16} and 0013_{16} and record the sum below.

SUM _ _ _ _

5. Add the binary numbers below. These numbers are the binary equivalent of the two hex numbers added by the program just executed.

		MSB		LSB	
$COEA_{16}$	=	1100	0000	1110	1010
$1B93_{16}$	=	0001	1011	1001	0011
SUM	=				

Now, convert the binary sum to its hexadecimal equivalent and record below.

SUM _ _ _ _

Does this match the sum obtained in step 4? ________

6. Now load the program of Figure 46 into the Trainer. This program simply adds two binary numbers and produces a carry. Hence, it will set the C flag. You will see its purpose in a moment.

   Execute the program by pressing the DO key and then entering address 0000.

7. Examine the carry (C) condition code register. The C flag is __________.
   set/reset

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	86	LDA	Load the accumulator immediate
0001	EA	EA	with EA_{16}.
0002	8B	ADD	Add immediate
0003	93	93	93
0004	3E	HLT	Halt

Figure 46
Program adds two numbers and produces carry.

8. Enter the program listed in Figure 47 into the Trainer. Notice that this is the same multiple-precision addition program previously executed, with the exception that the ADD Instruction has been replaced by the ADC instruction, as shown by the shaded section.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	01	NOP	No operation
0001	96	LDA	Load the accumulator direct with the
0002	OE	OE	least significant byte of the addend
0003	99	ADC	Add with carry direct the
0004	10	10	least significant byte of the augend.
0005	97	STA	Store the result in the
0006	12	12	least significant byte of the sum.
0007	96	LDA	Load the accumulator direct with the
0008	OF	OF	most significant byte of the addend.
0009	99	ADC	Add with carry direct the
000A	11	11	most significant byte of the augend.
000B	97	STA	Store the result in the
000C	13	13	most significant byte of the sum.
000D	3E	HLT	Halt
000E	EA	EA	Least significant byte } addend
000F	CO	CO	Most significant byte } addend
0010	93	93	Least significant byte } augend
0011	1B	1B	Most significant byte } augend
0012	—	—	Least significant byte } sum
0013	—	—	Most significant byte } sum

Figure 47
Multiple-precision addition program with instruction at address 003_{16} changed.

9. Set the program counter to 0000 and single-step through the program, recording the information in the chart of Figure 48.

10. Examine memory locations 0012_{16} and 0013_{16}. Record the sum below.

    SUM _ _ _ _

    Compare this sum to the previous sum recorded in step 4. Are they the same? ______.
    yes/no

    Why are the sums different? ____________________

    ____

    ____

    From this demonstration, what conclusion can you draw concerning the use of the ADC instruction? ____________________

    ____

    ____

STEP	PROGRAM COUNTER	OPCODE	ACCA	C FLAG	COMMENTS
1					
2					
3					
4					
5					
6					
7					

Figure 48

DISCUSSION

In steps 1 through 3 of this experiment, you loaded a multiple-precision addition program similar to the one you studied in Unit 4. Single-stepping through the program, you witnessed the operation of the ADC instruction. The chart you complied should be similar to the chart in Figure 49. When you checked memory locations 0012_{16} and 0013_{16}, you found the LSB and MSB respectively of the 16_{10}-bit sum. The sum should have been $DC7D_{16}$.

In step 5 you added the binary equivalents of the hex numbers, $COEA_{16}$ and $1B93_{16}$. The sum was the binary equivalent of the sum produced by the program, as shown below.

```
                    MSB           LSB

                       1
COEA16  =    1100  0000     1110  1010

1B9316  =    0001  1011     1001  0011

SUM     =    1101  1100     0111  1101
```

As you noticed, a carry is generated by the addition of the least significant bytes of the two numbers. When you were single-stepping through the program, you observed this carry because the C flag was set. The addition of the most significant bytes did not produce a carry. Therefore, the carry flag was cleared.

STEP	PROGRAM COUNTER	OPCODE	ACCA	C FLAG	COMMENTS
1	0001	96	Random	Random	Load the accumulator with the LSB of Addend (EA_{16}).
2	0003	9B	.EA	Random	Add the LSB of the Augend (93_{16}).
3	0005	97	7D	1	Store result in LSB of sum.
4	0007	96	7D	1	Load the accumulator with the MSB of the Addend (CO_{16}).
5	0009	99	CO	1	Add with carry the MSB of the Augend ($1B_{16}$).
6	000B	97	DC	0	Store result in MSB of Sum.
7	000D	3E	DC	0	Halt.

Figure 49

When you converted the binary number to hexadecimal, you found that the sum was the same as that produced by the program.

1101	1100	0111	1101
D	C	7	D

In step 6, you loaded a simple program that added the numbers EA_{16} and 93_{16}. Of course, the addition generated a carry, as you witnessed when you checked the C flag and found it set.

In step 8, you loaded another multiple-precision addition program into the Trainer. The only difference between this program and the previous multiple-precision addition program was that the first add instruction was the ADC (add with carry), rather than the ADD. Then you single-stepped through the program and completed the chart of Figure 48. Your chart should be similar to the one shown in Figure 50.

STEP	PROGRAM COUNTER	OPCODE	ACCA	C FLAG	COMMENTS
1	0001	96	Random	1 or 0	Load the accumulator with the LSB of Addend (EA_{16}).
2	0003	99	EA	1 or 0	Add with carry the LSB of the Augend (93_{16}).
3	0005	97	7E	1	Store result in LSB of sum. Load the accumulator with the MSB
4	0007	96	7E	1	of Addend (CO_{16}).
5	0009	99	CO	1	Add with carry the MSB of the Augend ($1B_{16}$).
6	000B	97	DC	0	Store result in MSB of sum.
7	000D	3E	DC	0	Halt.

Figure 50
Single-stepping through the multiple-precision addition program where both add instructions are ADC.

When you examined the sum at addresses 0012_{16} and 0013_{16}, you found $DC7E_{16}$. The correct sum, as you verified earlier, should have been $DC7D_{16}$. If you checked the chart compiled while single-stepping through the program, the reason for this incorrect answer should have been evident. The carry flag was set even before the program was executed. Therefore, when the Trainer executed the first ADC instruction, it automatically added the carry (1_2) to the sum of the least significant bytes. Hence, the result 7E was one greater than the correct sum of 7D.

From this demonstration you should have reached the conclusion that the ADC instruction should not be used unless you are positive of the condition of the C flag. You must remember that the C flag is only reset by an arithmetic operation that doesn't produce a **carry** or a **borrow**. For example, in the program that worked properly, we used the simple ADD instruction for the first addition. Naturally, this instruction ignores the condition of the C flag, so it doesn't matter if it's set or reset. This is a simple way of playing it safe. The second addition used the ADC instruction because we wanted any carry from the least significant byte to be reflected in the most significant byte.

The SBC (subtract with carry) instruction is similar to the ADC instruction because it also monitors the C flag to indicate a borrow. In the next section of this experiment, you will write a program that uses the SBC instruction for multiple-precision subtraction of 16_{10}-bit numbers.

PROCEDURE (Continued)

11. Write a program that will perfrom multiple-precision subtraction of two 16_{10}-bit (2-byte) numbers. The following guidelines define the problem.

    a. The program must subtract a 16_{10}-bit subtrahend from a 16_{10}-bit minuend and store the difference in memory.

    b. Use the direct addressing mode.

    c. Select the op codes from the instruction listing in Figure 51.

12. Now load the program. Enter 9721_{16} in the locations reserved for the minuend and 7581_{16} in the locations reserved for the subtrahend.

13. Single-step through the program and observe its operation. Examine the locations where the difference is stored and record the 2-byte difference below.

    DIFFERENCE ____ ____ ____ ____

INSTRUCTION	MNEMONIC	ADDRESSING MODE			
		IMMEDIATE	DIRECT	RELATIVE	INHERENT
Load Accumulator	LDA	86	96		
Clear Accumulator	CLRA				4F
Decrement Accumulator	DECA				4A
Increment Accumulator	INCA				4C
Store Accumulator	STA		97		
Add	ADD	8B	9B		
Subtract	SUB	80	90		
Add with Carry	ADC	89	99		
Subtract with Carry	SBC	82	92		
Arithmetic Shift Accumulator Left	ASLA				48
Decimal Adjust Accumulator	DAA				19
Halt	HLT				3E

Figure 51
Instructions.

DISCUSSION

If you made a flow chart of the problem, your flow chart probably looks like the one shown in Figure 52. Your program should be similar to the solution shown in Figure 53. After stepping through the program on the Trainer, the difference of the subtraction should have been $21A0_{16}$. If you didn't obtain this answer, go back and recheck your program.

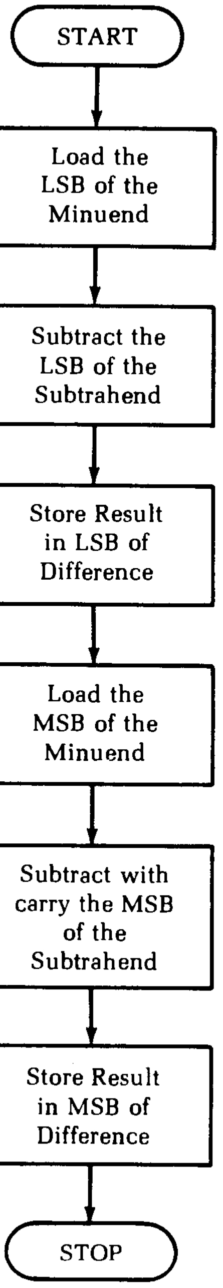

START
Load the LSB of the Minuend
Subtract the LSB of the Subtrahend
Store Result in LSB of Difference
Load the MSB of the Minuend
Subtract with carry the MSB of the Subtrahend
Store Result in MSB of Difference
STOP

Figure 52
Flow chart for multiple-precision subtraction.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	96	LDA	Load accumulator direct with
0001	0D	0D	least significant byte of minuend
0002	90	SUB	Subtract direct
0003	0F	0F	least significant byte of subtrahend
0004	97	STA	Store result in
0005	11	11	least significant byte of difference
0006	96	LDA	Load accumulator direct with
0007	0E	0E	most significant byte of minuend
0008	92	SBC	Subtract with carry
0009	10	10	most significant byte of the subtrahend
000A	97	STA	Store result in
000B	12	12	most significant byte of difference
000C	3E	HLT	Halt
000D	21	21	Least significant byte } Minuend
000E	97	97	Most significant byte } Minuend
000F	81	81	Least significant byte } Subtrahend
0010	75	75	Most significant byte } Subtrahend
0011	—	—	Least significant byte } Difference
0012	—	—	Most significant byte } Difference

Figure 53
Program for multiple-precision subtraction.

You may have used the SBC instruction for the first subtraction. If you did, this might explain the problem, because if the C flag is set when this instruction is executed a 1 will be borrowed from the difference. Therefore, your answer would have been 1 less than the correct answer, or $219F_{16}$. If the carry flag was cleared before you executed the program, the result would still be correct.

In the next section of this experiment, we will examine the ASLA (arithmetic shift accumulator left) instruction. You will also write a simple program that uses this instruction to multiply any 4_{10}-bit number by 16_{10}. This simple routine will prove it's usefulness later.

Recall from the discussion in Unit 4 that each ASLA operation multiplies the contents of the accumulator by two.

PROCEDURE (Continued)

14. Use the instructions listed in Figure 51 and write a program that uses the ASLA instruction to multiply any 4_{10}-bit number by 16_{10}.

15. Enter your program into the Trainer and then have your program multiply OF_{16} (15_{10}) by 16_{10}. Record the product below.

    $OF_{16} \times 16_{10} =$ ______$_{16}$.

16. Convert the product obtained to its decimal equivalent.

    Decimal equivalent ______$_{10}$.

    Now check your result by multiplying 15_{10} times 16_{10}.

    $15_{10} \times 16_{10} =$ ______$_{10}$.

17. In this program, the multiplier is determined by the number of ASLA instructions. How many ASLA instructions are required to produce a multiplier of 4_{10}? ________.

DISCUSSION

The program for this simple routine is shown in Figure 54. Notice that is uses 4_{10} ASLA instructions to produce the required multiplier of 16_{10}. If your program worked properly, the final product should have been $F0_{16}$. Converting this number to its decimal equivalent, we find that $F0_{16}$ equals 240_{10}. When we multiplied 15_{10} times 16_{10}, we also found the product was 240_{10}. Therefore, the program works.

Only two ASLA instructions are necessary to produce a multiplier of 4_{10}; three ASLA instructions will result in a multiplier of 8_{10}.

Another use for the ASLA instruction is to pack two BCD digits into a single byte. This "packing" can result in a significant savings of memory if many BCD numbers are used. Let's verify the operation of the BCD packing program that was presented in Unit 4.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	96	LDA	Load the accumulator with the
0001	09	09	4-bit multiplicand
0002	48	ASLA	Shift the accumulator
0003	48	ASLA	four places to the left
0004	48	ASLA	multiplying the multiplicand by
0005	48	ASLA	16_{10}.
0006	97	STA	Store the product
0007	0A	0A	at this location
0008	3E	HLT	Halt
0009	0F	0F	4-bit multiplicand
000A	—	—	Product

Figure 54
Program that uses the ASLA instruction to multiply a 4-bit number times 16_{10}.

PROCEDURE (Continued)

18. Enter the BCD packing program listed in Figure 55 into the Trainer. The unpacked BCD numbers are 09_{10} and 03_{10}.

HEX ADDRESS	OPCODES/ CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	01	NOP	Do nothing
0001	96	LDA	Load into the accumulator direct
0002	0D	0D	the unpacked most significant BCD digit.
0003	48	ASLA	Shift it four places to the left.
0004	48	ASLA	
0005	48	ASLA	
0006	48	ASLA	
0007	9B	ADD	Add the
0008	0E	0E	unpacked least significant BCD digit.
0009	97	STA	Store the result
000A	0C	0C	in the packed BCD number
000B	3E	HLT	Halt
000C	00	00	Packed BCD number
000D	09	09	Unpacked most significant BCD digit.
000E	03	03	Unpacked least significant BCD digit.

Figure 55
Program to pack two BCD digits into a single byte.

19. Set the program counter to 0000 and single-step through the program, recording the information below. Where it is indicated, convert the hexadecimal contents of the accumulator to the binary equivalent.

Program Count	Op code	ACCA	Binary Equivalent
0001	96	Random	Random
0003	48	____	________
0004	48	____	________
0005	48	____	________
0006	48	____	________
0007	9B	____	________
0009	97	____	________
000B	3E	HALT	

20. Examine the packed BCD number at address $000C_{16}$ and record it below.

Packed BCD Number ______

DISCUSSION

As you can see, the BCD packing program is very simple. Nevertheless, simple routines such as this can be combined in many programs, easing the task of programming. Most programmers either commit these general purpose routines to memory or file them away for future reference.

The results you obtained by stepping through the program should be similar to those shown below.

PROGRAM COUNT	OP CODE	ACCA	BINARY EQUIVALENT
0001	96	Random	Random
0003	48	09	0000 1001
0004	48	12	0001 0010 After 1st shift
0005	48	24	0010 0100 After 2nd shift
0006	48	48	0100 1000 After 3rd shift
0007	9B	90	1001 0000 After 4th shift
0009	97	93	1001 0011
000B	3E		

As the listing shows, the most significant BCD digit (09_{10}) is loaded into the accumulator. Four ASLA shifts take place, moving this digit progressively to the left. Following these four shifts, the most significant BCD digit is properly positioned. Now the program simply adds the least significant BCD (03_{10}) to the contents of the accumulator and then stores the sum. Checking the address of the packed BCD number, we find 93_{10}.

When BCD numbers are added, we encounter yet another problem. Often, the sum is the correct BCD number. But, just as frequently, it isn't. In Unit 4, the reason for this inconsistency was discussed. However, your Trainer has an instruction, called the "Decimal Adjust Accumulator" (DAA), that can correct the sum of BCD numbers, producing the desired result.

In the next portion of this experiment, we will demonstrate the need for the DAA instruction by first adding two BCD numbers without using the DAA instruction. Then we will check the sum. Next, we will correct the program by inserting DAA instructions and again examine the BCD sum.

PROCEDURE (Continued)

21. Load the program listed in Figure 56 into your Trainer. This program adds the BCD numbers 3792_{10} and 5482_{10}, storing the sum in address 0011_{16} and 0012_{16}.

22. RESET the Trainer and execute the program by first pressing the DO key and entering address 0000.

23. Again, press the RESET key and then examine the sum stored at address 0011_{16} and 0012_{16}. The most significant byte of the sum is at address 0011_{16} and the least significant byte is at address 0012_{16}. Record the sum below.

SUM ______

Is this the correct BCD sum for the addition of the numbers 3792_{10} and 5482_{10}? ________.
yes/no

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	96	LDA	Load the accumulator direct with
0001	0E	0E	the least significant byte of addend.
0002	9B	ADD	Add direct
0003	10	10	the least significant byte of augend
0004	97	STA	Store the result in
0005	12	12	the least significant byte of BCD sum.
0006	96	LDA	Load the accumulator direct with
0007	0D	0D	the most significant byte of addend
0008	99	ADC	Add with carry
0009	0F	0F	the most significant byte of augend
000A	97	STA	Store the result in
000B	11	11	the most significant byte of BCD sum.
000C	3E	HLT	Halt
000D	37	37	Most significant byte } BCD Addend
000E	92	92	Least significant byte } BCD Addend
000F	54	54	Most significant byte } BCD Augend
0010	82	82	Least significant byte } BCD Augend
0011	—		Most significant byte } BCD Sum
0012	—		Least significant byte } BCD Sum

Figure 56
Incorrect program for multiple-precision addition of BCD numbers.

24. Now load the corrected multiple-precision BCD addition program listed in Figure 57 into your Trainer. Notice that the only changes between this program and the previous program are the additions of the NOP instructions and the two DAA instructions following the addition operations.

25. Change the program counter to 0000 and single-step through the program, recording the information below.

STEP 1	______________ PROGRAM COUNT	______ OP CODE		
STEP 2	______________ PROGRAM COUNT	______ OP CODE	____ ACCA	
STEP 3	______________ PROGRAM COUNT	______ OP CODE	____ ACCA	____ C FLAG

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	01	NOP	Do nothing
0001	96	LDA	Load the accumulator direct with the
0002	11	11	least significant byte of addend.
0003	9B	ADD	Add direct
0004	13	13	the least significant byte of augend.
0005	19	DAA	Decimal adjust the sum to BCD.
0006	97	STA	Store the result in the
0007	15	15	least significant byte of BCD sum
0008	96	LDA	Load the accumulator direct with the
0009	10	10	most significant byte of addend.
000A	99	ADC	Add with carry the
000B	12	12	most significant byte of augend.
000C	19	DAA	Decimal adjust the sum to BCD.
000D	97	STA	Store the result in the
000E	14	14	most significant byte of BCD sum.
000F	3E	HLT	Halt.
0010	37	37	Most significant byte } BCD Addend
0011	92	92	Least significant byte } BCD Addend
0012	54	54	Most significant byte } BCD Augend
0013	82	82	Least significant byte } BCD Augend
0014	—	—	Most significant byte } BCD Sum
0015	—	—	Least significant byte } BCD Sum

Figure 57
Program for adding multiple-precision BCD numbers.

The sum of the addition of the least significant bytes is now in the accumulator. Is this the correct BCD sum for the numbers 92_{10} and 82_{10}? ________
yes/no

When the DAA instruction (op code 19) is executed, will this number be corrected? ________.
yes/no

STEP 4	________	________	________	________
	PROGRAM COUNT	OP CODE	ACCA	C FLAG

As you can see, the DAA instruction did correct the left-most digit by adding 60_{16} to the sum. Since the result 14_{10} appears to be a legitimate BCD number, how did the MPU know it was not the valid BCD sum? ____

____

STEP 5	________	________	________	________
	PROGRAM COUNT	OP CODE	ACCA	C FLAG
STEP 6	________	________	________	________
	PROGRAM COUNT	OP CODE	ACCA	C FLAG
STEP 7	________	________	________	________
	PROGRAM COUNT	OP CODE	ACCA	C FLAG

It's obvious that this number ($8C_{16}$) is not the BCD sum of 37_{10} and 54_{10}. What number will the MPU add to $8C_{16}$ to produce the desired BCD sum? ________.

STEP 8	________	________	________	________
	PROGRAM COUNT	OP CODE	ACCA	C FLAG
STEP 9	________	________	________	
	PROGRAM COUNT	OP CODE	ACCA	

26. Now examine the BCD sum at addresses 0014_{16} and 0015_{16} and record below.

SUM $\_\_\_\_\_\__{10}$.

DISCUSSION

When you executed the first program to add BCD numbers, it was obvious that the sum 8C14 was not the correct BCD number. The answer should have been 9274_{10}. Naturally, the MPU considered these BCD numbers as hexadecimal numbers, hence, the hexadecimal sum.

However, when the program was modified by the addition of DAA (decimal adjust accumulator) instructions after each addition operation, the result was the correct BCD number. As you stepped through the program you saw the DAA instruction in operation.

At step 3, the BCD numbers 92_{10} and 82_{10} had been added and the accumulator was supposedly storing the sum 14_{10}. A carry was generated by the setting of the C flag. However, the sum was not correct. Instead of 14_{10}, the sum should have been 174_{10}. To the MPU, the addition looked something like this.

	1001	0010_2	=	92_{16}
C FLAG	1000	0010_2	=	82_{16}
1 Carry	0001	0100_2		114_{16}

If we ignore the carry, the sum 14_{16} appears to be a legitimate BCD number. Nevertheless, the sum would be incorrect. Taking the carry flag into consideration, remember it's just an extension of the accumulator, we find the sum is 114_{16}. In hex, this is the correct sum of the two numbers.

In step 4, the DAA instruction had been executed and, as you witnessed, the number 14_{16} had been adjusted to the correct BCD sum of 74_{10}. The carry flag was set, indicating that the sum of the two left-most 4-bit binary numbers was larger than 1001_2 (9_{16}). Actually, it was $1\ 0001_2$. When the DAA instruction was executed, the MPU followed the conversion rules and adjusted the sum by adding 60_{16} as shown below.

Carry				Carry	
1	0001	0100_2	=	1	14_{16}
	0110	0000_2	=		60_{16}
1	0111	0100_2	=	1	74_{16}

The result is 74_{16} with a carry of 1_{16}. This is the correct BCD sum for the two BCD numbers. If we include the carry, the result is 174_{10} which is indeed the decimal sum of 92_{10} and 82_{10}. However, this exceeds the capacity of our storage locations, since they're only 8-bits long, so the carry is carried forward to the addition of the most significant bytes of the numbers in the next step.

As you continued single-stepping through the program, the most significant bytes were loaded and added with the ADC instruction. At step 7, the sum of this addition was in the accumulator. It was obvious that the sum $8C_{16}$ wasn't a BCD number. To adjust this number to the correct BCD sum, 06_{16} was added by the DAA instruction. The BCD adjusted sum 92_{10} was the result.

In the final step of the experiment, you verified program operation by examining the BCD sum at locations 0014_{16} and 0015_{16}. Here you should have found the sum 9274_{10}.

Experiment 7

New Addressing Modes

OBJECTIVES:

To demonstrate the extended addressing mode.

To demonstrate the indexed addressing mode.

To gain experience using the instruction set and registers of the MPU.

NOTE 1: If the Trainer you are using has a Model number ET-3400A, it will not be necessary for you to add the two RAM IC's (listed under "Material Required") to your Trainer. After reading the Introduction, begin this experiment at Procedure step 6.

NOTE 2: If the Trainer you are using has a Model number ET-3400, check IC locations IC16 and IC17. If these two locations do not contain IC's (2112 Heath number 443-721), begin this experiment at Procedure step 1. If these two locations are equipped with the 2112 IC's, begin this experiment at Procedure step 6.

MATERIAL REQUIRED

1 — Microprocessor Trainer

2 — 2112-2 IC's (Heath Number 443-721)

INTRODUCTION

In Unit 5, you learned that the MPU has two new addressing modes called extended and indexed addressing. Either of these addressing modes can be used to reach operands anywhere in memory. By contrast, the direct addressing mode can be used only when the operand is in the first 256_{10} bytes of memory.

PROCEDURE

1. Turn your ET-3400 Microprocessor Trainer off and unplug it.

2. Locate the two 2112-2 IC's (Heath number 443-721) that were supplied with the EB-6401-30 Parts Pack. Notice that these IC's are packed in conductive foam.

NOTE: These IC's are rugged, reliable components. However, normal static electricity discharged from your body through an IC pin to an object can damage the IC. Install these IC's without interruption as follows:

A. Remove the IC from its package with both hands.

B. Hold the IC with one hand and straighten any bent pins with the other hand.

C. Refer to Figure 58. Position the pin 1 end of the IC over the index mark on the circuit board.

D. Be sure each IC pin is properly started into the socket. Then push the IC down.

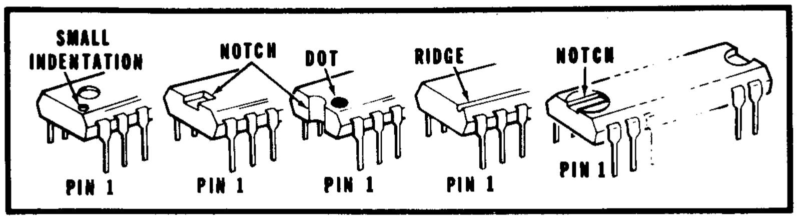


Figure 58

3. Install one of the IC's in the empty socket labelled IC16 on the ET-3400 Trainer.

4. Install the other IC in the socket labelled IC17.

NOTE: Until now, you could not use the extended addressing mode because the ET-3400 Trainer had only 256_{10} bytes of RAM memory. The installation of the two RAM IC's in the above steps has added an additional 256_{10} bytes of RAM memory necessary for the extended addressing mode.

5. Plug in your Trainer and turn it on.

6. Using the AUTO mode, load the numbers 00 through 0F into memory locations 0100 through 010F, respectively.

7. Using the EXAM and FWD keys, verify that the above numbers were stored in those addresses.

DISCUSSION

The ET-3400A Trainer required no hardware modification to acquire 512_{10} bytes of RAM in addresses 0000_{16} through $01FF_{16}$. The two 2114 RAM IC's at IC14 and IC15 already have this capacity. However, the ET-3400 Trainer uses 2112 RAM IC's. The two IC's at IC14 and IC15 contain only the first 256_{10} bytes of memory from 0000_{16} to $00FF_{16}$. Therefore, to extend the RAM capacity of the ET-3400 trainer an additional 256_{10} bytes, it may have been necessary to install two additional 2112 IC's at locations IC16 and IC17. The starting address of this new RAM is 0100_{16} and extends through $01FF_{16}$ for a total of 512_{10} bytes. When operands are placed at addresses above $00FF_{16}$, the extended addressing mode is generally used.

PROCEDURE (Continued)

8. Figure 59 shows a program for adding a list of numbers. Because the numbers are in addresses higher than $00FF_{16}$, the extended addressing mode is used. Load this program into the Trainer and verify that you have loaded it properly.

9. Execute the program using the single-step mode. The first instruction sets the contents of accumulator A to ________.

10. Examine the program counter and accumulator A after each instruction is executed. Each time an ADDA extended instruction is executed, the program counter is advanced ________bytes.

11. Examine the contents of accumulator A after the final instruction is executed. The number in accumulator A is ________.

12. Refer to your instruction set summary card. How many MPU cycles are required to execute this program? ________.

DISCUSSION

The program adds the ten numbers giving the sum 55_{10} or 37_{16}. It requires 51 MPU cycles. Notice that the program itself takes up 32_{10} bytes of memory. The data (the ten numbers) use another 10_{10} bytes.

A repetitive program like this one is an excellent candidate for indexed addressing. Let's see how the same job can be done using indexed addressing.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0100	4F	CLRA	Clear accumulator A
0101	BB	ADDA	Add the first number
0102	01	01	which is at this
0103	20	20	address.
0104	BB	ADDA	Add the second number.
0105	01	01	
0106	21	21	
0107	BB	ADDA	Add the third number.
0108	01	01	
0109	22	22	
010A	BB	ADDA	Continue until all numbers are added.
010B	01	01	
010C	23	23	
010D	BB	ADDA	
010E	01	01	
010F	24	24	
0110	BB	ADDA	
0111	01	01	
0112	25	25	
0113	BB	ADDA	
0114	01	01	
0115	26	26	
0116	BB	ADDA	
0117	01	01	
0118	27	27	
0119	BB	ADDA	
011A	01	01	
011B	28	28	
011C	BB	ADDA	
011D	01	01	
011E	29	29	
011F	3E	WAI	Stop.
0120	01	01	First number.
0121	02	02	Second number.
0122	03	03	Third number.
0123	04	04	
0124	05	05	•
0125	06	06	•
0126	07	07	•
0127	08	08	
0128	09	09	
0129	0A	0A	Tenth number.

Figure 59
Adding a list of numbers using extended addressing.

PROCEDURE (Continued)

13. Figure 60 shows a program for adding the same list of numbers. However it uses indexed addressing. Load this program into the Trainer and verify that you have loaded it correctly.

HEX ADDRESSES	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0130	4F	CLRA	Clear accumulator A
0131	CE	LDX#	Load the index register immediately
0132	01	01	with the address of
0133	20	20	the first number in the list.
0134	AB	ADDA, X	Add to accumulator A indexed
0135	00	00	with 00 offset.
0136	08	INX	Increment index register.
0137	8C	CPX#	Compare the index register immediately
0138	01	01	with one greater than the address
0139	2A	2A	of the last number in the list.
013A	26	BNE	If there is no match
013B	F8	F8	branch back to here.
013C	3E	WAI	Otherwise, halt.

Figure 60
Adding the list of numbers using indexed addressing.

14. Execute the program using the single-step mode. After each step, record the contents of the program counter, accumulator A, and the index register in Figure 61.

15. Compare the programs of Figures 59 and 60. Which requires fewer instructions?

16. Refer to the instruction set summary card. How many machine cycles are required to execute the program shown in Figure 59 ____________? Compare this with the number of machine cycles required for the program in Figure 60.

STEP NUMBER	CONTENTS AFTER EACH STEP		
	PC	ACCA	INDEX
1			
2			
3			
4			
5			
6			
7			
8			
9			
10			
11			
12			
13			
14			
15			
16			
17			
18			
19			
20			
21			
22			
23			
24			
25			
26			
27			
28			
29			
30			
31			
32			
33			
34			
35			
36			
37			
38			
39			
40			
41			
42			
43			

Figure 61
Record values here.

DISCUSSION

This example illustrates that when a repetitive task is to be done, indexed addressing can save many bytes of memory. In many cases, indexed addressing requires more MPU cycles and therefore, a longer time to execute. Generally, time is of little importance compared to saving a substantial number of memory bytes.

Let's look at some other ways that indexed addressing is used.

PROCEDURE (Continued)

17. Write a program that will clear memory locations 0120_{16} through $01A0_{16}$. It should use indexed addressing. The program should reside in the lower RAM addresses.

18. When you are sure your program is correct, load it into the ET-3400 Trainer. Verify that you loaded it correctly; then execute it using the DO command.

19. Examine memory locations 0120_{16} through $01A0_{16}$. Each should be cleared. Examine locations below 0120_{16} and above $01A0_{16}$. These locations should not be cleared.

20. Debug your program if necessary and repeat steps 18 and 19 until the desired results are obtained.

DISCUSSION

Our solution to the problem is shown in Figure 62. Your solution may be similar or quite different. If it achieves the proper result and requires about the same number of bytes, then it is perfectly acceptable.

We still have not demonstrated the full power of indexed addressing because we have not yet used the offset capability. Let's look at how the offset capability can be used. Figure 63 shows three tables. The first two tables contain signed numbers, the third is initially cleared. The entries in the first two tables are to be added and the resulting sums are to be placed in the third table. That is, the first entry in table 1 is to be added to the first entry in table 2. The resulting sum is to be stored as the first entry of table 3. The second entry in table 1 is to be added to the second entry in table 2, forming the second entry in table 3; etc.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	CE	LDX#	Load index register immediately with
0001	01	01	the address of the
0002	20	20	first location to be cleared.
0003	6F	→CLR, X	Clear the location whose
0004	00	00	address is indicated by the index register.
0005	08	INX	Increment the index register.
0006	8C	CPX#	Compare the number in the index
0007	01	01	register with one greater than
0008	A1	A1	the address of the last location to be cleared.
0009	26	BNE	If there is no match
000A	F8	F8	branch back to here.
000B	3E	WAI	Otherwise, stop.

Figure 62
Program for clearing addresses 0120_{16} through $01A0_{16}$.

TABLE 1		TABLE 2		TABLE 3	
ADDRESS	CONTENTS	ADDRESS	CONTENTS	ADDRESS	CONTENTS
0100	06	0110	FA	0150	00
0101	0F	0111	01	0151	00
0102	06	0112	1A	0152	00
0103	20	0113	10	0153	00
0104	2F	0114	11	0154	00
0105	00	0115	50	0155	00
0106	2F	0116	31	0156	00
0107	61	0117	0F	0157	00
0108	3E	0118	42	0158	00
0109	4F	0119	41	0159	00
010A	91	011A	0F	015A	00
010B	9F	011B	11	015B	00
010C	C0	011C	00	015C	00
010D	84	011D	4C	015D	00
010E	70	011E	70	015E	00
010F	E1	011F	0F	015F	00

Figure 63
Three tables.

PROCEDURE (Continued)

21. Enter the data shown in Figure 63 into the indicated addresses.

22. Write a program that will solve the problem described above.

23. Enter the program into the Trainer and execute it.

24. Examine addresses 0150_{16} through $015F_{16}$ to verify that the program performed properly.

25. If necessary, debug your program and try again.

DISCUSSION

The solution to the problem is shown in Figure 64.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	CE	LDX#	Load index register with address
0001	01	01	of first entry
0002	00	00	in Table 1.
0003	A6	→ LDAA, X	Load entry from Table 1 into
0004	00	00	accumulator A.
0005	AB	ADDA, X	Add the corresponding entry from
0006	10	10	Table 2.
0007	A7	STAA, X	Store the result in the
0008	50	50	corresponding location in Table 3
0009	08	INX	Increment the index register.
000A	8C	CPX#	Compare the number in the index
000B	01	01	register with one greater
000C	10	10	than the address of the last entry in Table 1.
000D	26	BNE	If there is no match,
000E	F4	F4	branch to here.
000F	3E	WAI	Otherwise, stop.

Figure 64
Program for adding two tables.

Experiment 8

Arithmetic Operations

OBJECTIVES:

To gain practice using the instruction set and registers of the 6800 MPU.

To demonstrate a fast method of performing multiplication.

To demonstrate multiple-precision arithmetic.

To demonstrate an algorithm for finding the square root of a number.

To gain experience writing programs.

INTRODUCTION

In Unit 5, you were exposed to the full architecture and instruction set of the 6800 microprocessor. In this experiment, you will use some of the new-found capabilities of the microprocessor to solve some simple problems.

Mathematical operations make excellent programming examples and at the same time illustrate useful procedures. For these reasons, the programs developed in this experiment are concerned with arithmetic operations.

In an earlier unit, you learned that a computer can multiply by repeated addition. However, this is a very slow method of multiplication when large numbers are used.

A much faster method of multiplying involves a shifting-and-adding process. To illustrate the procedure, consider the long hand method of multiplying two 4-bit binary numbers. The procedure looks like this.

$$\begin{array}{rcccr} 1101_2 & \leftarrow & \text{Multiplicand} & \rightarrow & 13_{10} \\ \underline{1011_2} & \leftarrow & \text{Multiplier} & \rightarrow & \underline{11_{10}} \\ 1101 & & & & 13 \\ 1101 & & & & \underline{13} \\ 0000 & & & & 143_{10} \\ \underline{1101} & & & & \\ 10001111_2 & \leftarrow & \text{Product} & \nearrow & \end{array}$$

The decimal equivalents are shown for comparison purposes. The product is formed by shifting and adding the multiplicand. Put in computer terms, the procedure goes like this:

1. Clear the product.

2. Examine the multiplier. If it is 0, stop. Otherwise, go to 3.

3. Examine the LSB of the multiplier. If it is 1, add the multiplicand to the product then go to 4. If it is a 0, go to 4 without adding.

4. Shift the multiplicand to the left.

5. Shift the multiplier to the right so that the next bit becomes the LSB.

6. Go to 2.

PROCEDURE

1. Write a program of any length that will perform multiplication in the manner indicated. Here are some guidelines:

   A. You may use any of the instructions discussed up to this point.

   B. To keep the program simple, only unsigned 4-bit binary numbers are to be used for the multiplier and the multiplicand.

   C. The final product should be in Accumulator A when the multiplication is finished.

   D. The multiplier may be destroyed during the multiplication process.

   E. Assume that the multiplier and multiplicand are initially in memory. That is, you should load them into memory along with the program.

2. Try to write the program before you read further. If after 30 minutes, you feel you are not making progress, go on to step 3.

3. If you feel you need help, read over the following hints and then write the program.

   A. The product should be formed in accumulator A.

   B. The first step is to clear the product.

   C. The multiplicand is shifted and added to Accumulator A. Accumulator B is a good place to hold the multiplicand during this process.

   D. The multiplier can be tested for zero while still in memory by using the TST instruction followed by the BEQ instruction.

   E. A good way to test the LSB of the multiplier is to shift the multiplier one bit to the right into the carry flag and then test the carry flag with a BCC instruction.

4. Once your program is written, load it into the Trainer and run it. Verify that it works for several different values of multipliers and multiplicands. Debug your program as necessary.

DISCUSSION

The real test of your program is "Does it work?" If it works, then you have successfully completed this part of the experiment. One solution to the problem is shown in Figure 65. Compare your program with this one. If you could not write a successful program, study this program carefully to see how it handles each phase of the operation.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0010	4F	CLRA	Set the product to 0.
0011	D6	LDAB	Load accumulator B with the
0012	22	22	multiplicand.
0013	7D	→ TST	Test
0014	00	00	the
0015	23	23	multiplier.
0016	27	BEQ	If it is 0, branch to the
0017	09	09	wait instruction.
0018	74	LSR	Shift the LSB of the
0019	00	00	multiplier to the
001A	23	23	right into the carry flag.
001B	24	BCC	If the carry flag is cleared
001C	01	01	skip the next instruction.
001D	1B	ABA	Add the multiplicand to the product.
001E	58	ASLB	Shift the multiplicand to the left.
001F	20	BRA	Branch back and go through again.
0020	F2	F2	
0021	3E	WAI	Wait.
0022	05	Multiplicand	
0023	03	Multiplier	

Figure 65
Multiplying by shifting and adding.

Obviously, this simple program has some serious drawbacks. The chief one is that the product cannot exceed eight bits. Fortunately, the basic procedure can be expanded so that much larger numbers can be handled. The solution is to use two bytes for the product. This will allow products up to $65,535_{10}$. In this example, the multiplier will be restricted to eight bits. However, the multiplicand can have up to 16 bits (two bytes) as long as the product does not exceed $65,535_{10}$. In an earlier unit, you learned that multiple-precision numbers can be added by a 2-step operation. The least significant (LS) byte of one number is added to the LS byte of the other. Then, the MS byte is added **with carry** to the MS byte of the other. Keep this in mind as you write your program.

The procedure for shifting a multiple-precision value will also come in handy. To shift a 2-byte number to the left, a 2-step procedure like that shown in Figure 66 can be used. First, the LS byte is shifted one place to the left into the carry bit by using the ASL instruction. Next the MS byte is rotated to the left. The result is that the 16-bit number has been shifted one bit to the left.

PROCEDURE (Continued)

5. Write a program that will multiply a double-precision multiplicand times an 8-bit multiplier. Assume that the double-precision product is to be stored in memory locations 0000_{16} and 0001_{16}. The double-precision multiplicand is initially in addresses 0002_{16} and 0003_{16}. The 8-bit multiplier is in address 0004_{16}.

6. Once again, you should try to write this program. If after 30 minutes or so you are not making progress, read the hints given in step 7.

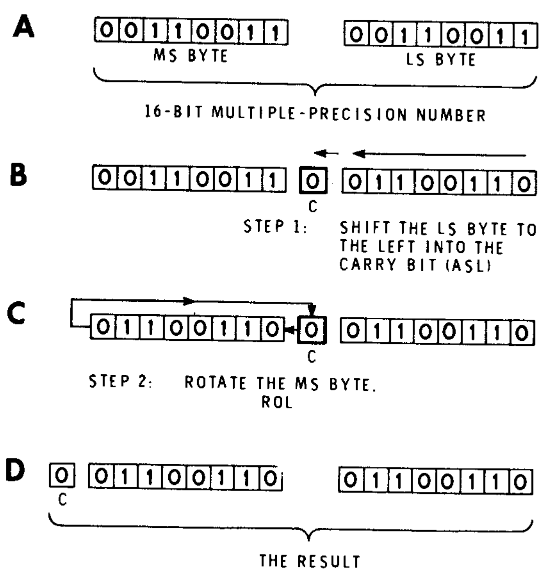


Figure 66
Shifting a multiple-precision number.

7. Read over the following hints (if necessary) and try again.

   A. Initially clear both bytes of the product.

   B. Test the multiplier for zero exactly as you did in the previous program.

   C. Test the LSB of the multiplier as you did in the previous program.

   D. When adding the multiplicand to the product, use the multiple-precision add technique.

   E. When shifting the multiplicand to the left, use the technique shown in Figure 66.

8. Once your program is written, load it into the Trainer and verify that it works properly. Debug the program as necessary.

DISCUSSION

There are dozens of ways in which this program could be written. If your program produces proper results, then you have been successful. One solution to the problem is shown in Figure 67. Compare your program with this one. If you were unsuccessful in writing a program, study Figure 67 very carefully until you understand the procedures involved.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	—	—	Product (LS byte)
0001	—	—	Product (MS byte)
0002	—	—	Multiplicand (LS byte)
0003	—	—	Multiplicand (MS byte)
0004	—	—	Multiplier
*	*	*	Instructions start at address 0010
0010	7F	CLR	Clear the product.
0011	00	00	
0012	00	00	
0013	7F	CLR	
0014	00	00	
0015	01	01	
0016	7D	TST	Test the multiplier.
0017	00	00	
0018	04	04	
0019	27	BEQ	If the multiplier is 0, branch to
001A	19	19	the WAI instruction.
001B	74	LSR	Otherwise, shift the right most
001C	00	00	bit of the multiplier into
001D	04	04	the C flag.
001E	24	BCC	If the C flag is 0 branch to
001F	0C	0C	here.
0020	96	LDAA	Otherwise, load the LS byte of
0021	00	00	the product into accumulator A.
0022	9B	ADDA	Then add the LS byte of the
0023	02	02	multiplicand.
0024	D6	LDAB	Load the MS byte of the product
0025	01	01	into accumulator B.
0026	D9	ADCB	Add (with carry) the MS byte of the
0027	03	03	multiplicand.
0028	97	STAA	Store the contents of accumulator A
0029	00	00	as the LS byte of the product.
002A	D7	STAB	Store the contents of accumulator B
002B	01	01	as the MS byte of the product.
002C	78	ASL	Shift the LS byte of the
002D	00	00	multiplicand to the left.
002E	02	02	
002F	79	ROL	Rotate the MS byte of the
0030	00	00	multiplicand to the left.
0031	03	03	
0032	20	BRA	Repeat the process.
0033	E2	E2	
0034	3E	WAI	Stop.

Figure 67
Program for multiplying a double-precision multiplicand by an 8-bit multiplier.

Another problem that makes a good programming exercise is finding the square root of a number. Writing the program is not too difficult once you develop the proper algorithm. While there are many different ways to find the square root of a number, the easiest method from the programmer's point of view involves the subtraction of successive odd integers.

This method works because of the relationship between perfect squares. The first several perfect squares are $0^2 = 0, 1^2 = 1, 2^2 = 4, 3^2 = 9, 4^2 = 16, 5^2 = 25$, etc. Notice:

The relationship between the numbers 0, 1, 4, 9, 16, 25, etc.

The difference between 0 and 1 is 1, the first odd integer.

The difference between 1 and 4 is 3, the second odd integer.

The difference between 4 and 9 is 5, the third odd integer; etc.

This relationship gives us a simple method of finding the exact square root of perfect squares and of approximating the square root of non-perfect squares.

The procedure for finding the square root of a number looks like this:

1. Subtract successive odd integers (1, 3, 5, 7, 9, etc.) from the number until the number is reduced to 0 or a negative value.
2. Count the number of subtractions required. The count is the exact square root of the number if the number was a perfect square. The count is the approximate square root if the number was not a perfect square.

For example, let's find the square root of 49_{10}.

49	Original Number.
–1	Subtract the first odd integer.
48	
–3	Subtract the second odd integer.
45	
–5	Subtract the third odd integer.
40	
–7	Subtract the fourth odd integer.
33	
–9	Subtract the fifth odd integer.
24	
–11	Subtract the sixth odd integer.
13	
–13	Subtract the seventh odd integer.
0	Stop subtracting because the original number has been reduced to 0.

We simply count the number of subtractions required.

Since 7 subtractions were required, the square root of 49 is 7.

PROCEDURE (Continued)

9. With pencil and paper, use the above algorithm to find the square root of 81_{10}. Does the answer give the exact square? ________. Was the result of the final subtraction 0? ________.

10. With pencil and paper, use the above algorithm to find the square root of 119_{10}. How many subtractions are required to reduce the number to a negative value. Does this count approximate the square root of 119_{10}? __________.

11. Write a program that uses the above algorithm to find or approximate the square root of any unsigned 8-bit number.

12. Load your program into the Trainer and run it. Verify that it works for several different values.

DISCUSSION

Our solution to the problem is shown in Figure 68. The number is loaded into accumulator A, where it will be gradually reduced to a negative value. The odd integer is maintained in accumulator B. Each new odd integer is formed by incrementing twice. The SBA instruction is used to subtract the odd integer from the number. The BCS instruction is used to determine when the number goes negative (a borrow occurs at that point). You could have used the BMI instruction but this would limit the original number to a value below $+128_{10}$. A few bytes are saved by not maintaining a separate count of the number of subtractions. Instead, the final odd integer value is converted to the count. This is possible because of the relationship between the odd integer value and the number of subtractions. As the program is written, the final odd integer is always one more than twice the number of subtractions. By shifting the final odd integer to the right, the correct count is created.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	96	LDAA	Load the number that is at
0001	0F	0F	this address into accumulator A.
0002	C6	LDAB#	Load accumulator B with the
0003	01	01	first odd integer.
0004	10	SBA	Subtract the odd integer from the number.
0005	25	BCS	If the carry is set, branch
0006	04	04	to here.
0007	5C	INCB	Otherwise, form the next higher odd
0008	5C	INCB	integer by incrementing B twice.
0009	20	BRA	Branch back
000A	F9	F9	to here.
000B	54	LSRB	Shift the odd integer to the right.
000C	D7	STAB	Store the answer at
000D	10	10	this address.
000E	3E	WAI	Wait.
000F	—	Number	Number to be operated upon.
0010	—	Answer	Final answer appears here.

Figure 68
Square root subroutine.

Of course, any square root program that is limited to numbers below 256_{10} is of limited use. However, this same technique can be applied to multiple-precision numbers. Figure 69 shows a program that can find or approximate the square root of numbers up to $16{,}385_{10}$. Before you study this program, try to write your own program to do this.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	96	LDAA	Load accumulator A with the
0001	1A	1A	LS byte of the number.
0002	D6	LDAB	Load accumulator B with the
0003	19	19	MS byte of the number.
0004	7F	CLR	Clear
0005	00	00	the odd
0006	1B	1B	integer.
0007	7C	INC	Increment.
0008	00	00	the odd
0009	1B	1B	integer.
000A	90	SUBA	Subtract the odd
000B	1B	1B	integer from the LS byte of the number.
000C	C2	SBCB#	Take care of any borrow
000D	00	00	from the MS byte of the number.
000E	25	BCS	If the carry is set, branch
000F	05	05	to here.
0010	7C	INC	Otherwise, form the next
0011	00	00	higher odd integer by
0012	1B	1B	incrementing
0013	20	BRA	and branching
0014	F2	F2	to here.
0015	74	LSR	Convert the odd integer to
0016	00	00	the answer by shifting
0017	1B	1B	right.
0018	3E	WAI	Stop.
0019	—	Number (MS)	Number to be
001A	—	Number (LS)	operated upon.
001B	—	Odd integer	Form the odd integer and the answer here.

Figure 69
Routine for finding the square root of a double precision number.

Experiment 9

Stack Operations

OBJECTIVES:

To demonstrate the stack operations that occur automatically.

To demonstrate ways that the programmer can use the stack.

To demonstrate the break-point capability of the Trainer.

INTRODUCTION

As you learned in Unit 6, the stack is used by the MPU to perform some automatic functions. When an interrupt occurs or a WAI is encountered, the MPU pushes the contents of the program counter, index register, accumulators, and condition codes on to the stack. We can easily verify this.

PROCEDURE

1. Figure 70 shows a program for setting the MPU registers to a known state. Examine the program and determine the hex contents of the following registers immediately after the WAI is executed.

Condition Code Register	________
Accumulator B	________
Accumulator A	________
Index Register	________
Program Counter	________

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	8E	LDS#	Load 0020 into
0001	00	00	the stack pointer
0002	20	20	
0003	CE	LDX#	Load EEDD into the index register.
0004	EE	EE	
0005	DD	DD	
0006	C6	LDAB#	Load BB into ACCB.
0007	BB	BB	
0008	86	LDAA#	Load AA into ACCA.
0009	AA	AA	
000A	36	PSHA	Push AA onto the stack.
000B	86	LDAA#	Load CC into ACCA.
000C	CC	CC	
000D	06	TAP	Transfer CC into the condition codes.
000E	32	PULA	Pull AA from the stack.
000F	3E	WAI	Wait.
0010			

Figure 70
This routine sets the contents of all MPU registers to known values.

2. Load the program into the Trainer and verify that you loaded it properly.

3. Execute the program using the DO command.

4. Examine the following memory locations and record their hex contents.

Address	Contents	Register
001A	____	____
001B	____	____
001C	____	____
001D	____	____
001E	____	____
001F	____	____
0020	____	____

5. Identify the register from which these numbers came.

6. Try to examine the contents of ACCA, ACCB, PC, SP, and INDEX register. Do their contents agree with the number loaded there?

DISCUSSION

When the WAI instruction is executed, the contents of the MPU registers are pushed onto the stack. Since the stack pointer is initially at 0020, the contents of the registers are stored as follows.

Address	Contents	Where it came from
001A	CC	Condition Codes
001B	BB	Accumulator B
001C	AA	Accumulator A
001D	EE	Index Register (high byte)
001E	DD	Index Register (low byte)
001F	00	Program Counter (high byte)
0020	10	Program Counter (low byte)

When you tried to examine the contents of ACCA, ACCB, SP, etc., you found that their contents did not agree with what was loaded. The reason for this **apparent** error is that the Trainer does not actually examine the contents of these registers. Instead, it examines what is placed in the stack by the WAI instruction. However, when the Trainer is reset, the monitor program assumes that the stack starts at address 00D1. Since our program moved the location of the stack, we can not use the ACCA, ACCB, PC, SP, CC, or INDEX commands after changing the stack pointer and then resetting the Trainer.

This demonstrates how the MPU uses the stack. A similar operation occurs for the SWI instruction or when a hardware interrupt occurs. Of course, the programmer can also use the stack.

PROCEDURE (Continued)

7. Figure 71 shows a program that will clear memory locations 0001 through 001F. It then transfers a list of numbers to these addresses. The numbers come from addresses 0151 through 016F.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ ADDRESS	COMMENTS
0020	CE	LDX#	Load the index register with highest address to be cleared.
0021	00	00	
0022	1F	1F	
0023	6F	→ CLR, X	Clear it.
0024	00	00	
0025	09	DEX	Decrement index register to next lower address.
0026	26	BNE	Finished? If not, go back and clear the indicated address.
0027	FB	FB	
0028	08	INX	Set index register to first entry in new list.
0029	8E	LDS#	Set the stack pointer to one less than the first entry in the old list.
002A	01	01	
002B	50	50	
002C	32	→ PULA	Pull the entry from the old list.
002D	A7	STAA, X	Store it in the new list.
002E	00	00	
002F	08	INX	Increment index register to next entry in list.
0030	8C	CPX#	Finished?
0031	00	00	
0032	20	20	
0033	26	BNE	If not, go back and pull next entry.
0034	F7	F7	
0035	3E	WAI	Otherwise, wait.

Figure 71
Program for demonstrating stack operations and breakpoints.

8. Load this program into the Trainer and verify that you loaded it properly.

9. At address 0151 through 016F, load the numbers 01 through $1F_{16}$, respectively.

10. Execute the program using the DO command.

11. Examine addresses 0001 through 001F. They should contain the numbers 01 through 1F, respectively.

DISCUSSION

This illustrates how the stack can be used in conjunction with indexing to move a list of numbers.

When this program is executed using the DO command, everything happens so fast that it is impossible to see intermediate results. Of course, you could use the single-step mode and examine the result produced by every single instruction. But in many programs, this is a long, tedious process. Therefore, the Trainer provides another way to examine programs. It allows us to set four different breakpoints in our program. The Trainer will execute instructions at its normal speed until it reaches one of these breakpoints. At that point, the Trainer will stop with the address and op code of the next instruction displayed. While the Trainer is stopped, you can examine and change the contents of any register or memory location. When you are ready to resume, you depress the return (RTI) key and the Trainer executes instructions at its normal speed until the next breakpoint or a WAI instruction is encountered.

PROCEDURE (Continued)

12. Verify that the program is still in memory.

13. Depress the RESET key. Do not depress RESET again as you perform the following steps. To do so will erase any breakpoints that you set.

14. Refer to the program listing in Figure 71. Let's assume we wish to stop and examine memory and the MPU registers just before the BNE instruction at address 0026 is executed.

15. Depress the BR key. The display should be _ _ _ _ Br. The Trainer is now ready to accept the first breakpoint address. Enter the address at which the Trainer is to stop: 0026. The breakpoint is now entered.

16. Without hitting RESET, depress the DO key. Enter the address of the first instruction in the program: 0020.

17. Immediately, the display will show the address 0026 and op code 26 at which the breakpoint occurred.

18. Without hitting RESET, examine the contents of the index register. It should now read 001E.

19. Depress the EXAM key and examine address 001F. It should now be cleared.

20. Notice that you can examine the contents of any MPU register or memory location from this breakpoint mode.

21. When you are ready for the program to resume, depress the RTI key once. Again, the display will read 002626 because the MPU is back at the same breakpoint on the second pass through the first loop.

22. Examine the index register again. It should now read 001D. Examine location 001E and verify that it has been cleared.

23. The loop will be repeated 31_{10} times. On the 32nd pass, the program will escape the loop.

24. Before you go further, set a second breakpoint at the INX instruction. Do this by depressing the BR key and entering the address of the instruction (0028).

25. Depress the RTI key again. Notice that the program is still stopping at the first breakpoint. It will continue to do so until it escapes the first loop.

26. You have now pushed the RTI key three times. Repeatedly push the RTI key until the display changes to 0028 08. The RTI key should have been depressed a total of 32_{10} times, counting the first three times.

27. The program is now waiting at the second break point.

28. To demonstrate a point, let's set two additional break points.

29. Depress the BR key and enter address 0029. This sets the third break point at the LDS# instruction.

30. Depress the BR key again and enter address 0033. This sets the fourth break point at the last BNE instruction.

31. The Trainer will accept only four breakpoints. We have now reached this limit. Depress the BR key again in an attempt to enter a fifth breakpoint. Notice that the word "FULL!" appears on the display.

32. Depress the RTI key so that the Trainer resumes program execution. It should stop at the third breakpoint.

33. Depress the RTI key again. The program should stop at the fourth breakpoint. Notice that the program is again in a loop. On each pass through the loop, the program will stop at this fourth breakpoint.

34. Analyze the operation of the program by examining the pertinent registers and memory locations on each pass through the loop.

DISCUSSION

The breakpoint capability of the Trainer can be a powerful aid in writing, analyzing and debugging a program. It allows us to stop at four distinct points in the program. Here are some tips to remember when using this capability:

1. A maximum of four breakpoints can be used.
2. These may be entered all at once or during a previous breakpoint pause.
3. The RESET key erases all breakpoints.
4. The contents of the address at which the breakpoint is set must be an op code.

Experiment 10

Subroutines

OBJECTIVES:

To demonstrate the use of subroutines.

To demonstrate that the monitor program of the ET-3400 Trainer contains some useful subroutines that can be called when needed.

To gain experience writing programs.

INTRODUCTION

Most of the subroutines that you will develop and use in this experiment deal with lighting the displays on the Trainer. For this reason, we will begin by discussing how the displays are accessed.

The ET-3400 Microprocessor Trainer has six hexadecimal displays. Each display contains eight light-emitting diodes (LED's) arranged as shown in Figure 72. Each LED is given two addresses. The addresses for the left-most display are shown. To light a particular LED, we simply store an odd number at the proper address. An odd number is used because the LED responds to a 1 in bit 0 of the byte that is stored. To turn an LED off, we store an even number at the proper address. The following procedure will demonstrate this.

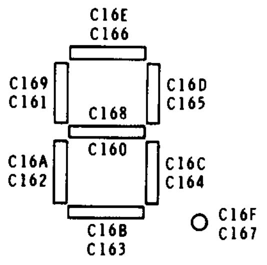


Figure 72
Addresses of the various segments in the left LED display.

PROCEDURE

1. Write a program that will halt after storing an odd number (such as 01) at address $C167_{16}$.

2. Load the program into the Trainer and execute it using the DO command. The microprocessor should halt with the decimal point of the left-most display lit.

3. Notice that the LED remains lit until it is deliberately turned off.

DISCUSSION

To form characters, the LED's in the display must be turned on in combination. For example, to form the letter "A", the segments at addresses C162, C161, C166, C165, C164, and C160 must be turned on.

PROCEDURE (Continued)

4. Write a program that will halt after storing an odd number (such as 01) at the six addresses listed above.

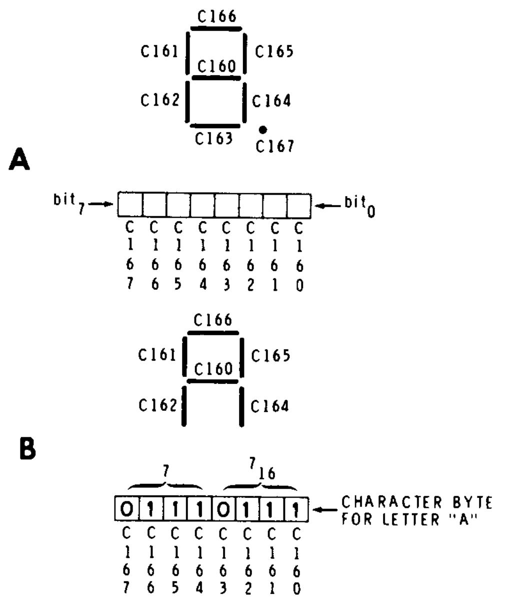
5. Load the program into the Trainer and execute it using the DO command. The microprocessor should halt with the letter A in the left-most display.

DISCUSSION

Your program probably took this form:

```
LDAA   #     01
STAA         C162
STAA         C161
STAA         C166
       •
       •
       •
WAI
```

While this approach works, the program would have to be rewritten for each new character. What is needed is a program that will form many characters. One approach is to store characters as 8-bit character bytes. Since there are eight LED's in each display, each bit of the character byte can be assigned to a different LED segment. Figure 73A shows how each bit in a character byte is assigned to each segment of the display. To light a corresponding LED, the proper bit in the character byte must be 1. For example, Figure 73B shows the character byte for the letter A. To form this letter, all display segments except C163 and C167 must be lit. Therefore, a 1 is placed in the character byte at all bits except the two that correspond to these addresses.



A
C166
C161
C165
C160
C162
C164
C163
C167
bit7
bit0
C167 C166 C165 C164 C163 C162 C161 C160
B
C166
C161
C165
C160
C162
C164
7
7 16
0 1 1 1 0 1 1 1
CHARACTER BYTE FOR LETTER "A"
C167 C166 C165 C164 C163 C162 C161 C160

Figure 73
Assigning the bits of the character byte.

The display responds only to bit 0 of the character byte. To make each segment bit appear in turn at bit 0, the character byte must be shifted to the right. After each shift, the contents of the character byte must be stored at the address whose corresponding bit is now at bit 0. The procedure is:

1. Store the contents of the character byte at $C160_{16}$.
2. Shift the character byte to the right.
3. Store it at $C161_{16}$.
4. Shift it to the right again.
5. Store it at $C162_{16}$.

Etc.

A program that will do this is shown in Figure 74.

HEX ADDRESS	HEX CONTENTS	MNEMONIC/ CONTENTS	COMMENTS
0000	86	LDAA#	Load accumulator A immediate with the
0001	77	77	character byte.
0002	CE	LDX#	Load the index register immediate with
0003	C1	C1	the address.
0004	60	60	of the left display.
0005	A7	→ STAA, X	Store the character byte at the address indicated by the index register.
0006	00	00	
0007	44	LSRA	Shift the character bit to the right.
0008	08	INX	Advance index register to the address of the next segment.
0009	8C	CPX	Compare index register with one greater
000A	C1	C1	than the address of the
000B	68	68	last segment.
000C	26	BNE	If no match occurs branch
000D	F7	F7	back to here.
000E	3E	WAI	Otherwise, stop.

Figure 74
Program for lighting a display.

PROCEDURE (Continued)

6. Load the program into the Trainer and verify that you loaded it correctly.

7. Execute the program using the DO command. The left-most digit should display the letter A.

8. The character byte is at address 0001. Change this byte to 47_{16}.

9. Execute the program again using the DO command. What letter appears in the display? ________.

10. Change the character byte so that the letter H is displayed. What character byte is required? ________.

11. Change the character byte to 79_{16}. Execute the program. What character is displayed? ______.

12. Refer to Figure 75. This figure shows the addresses of the LED's in each of the six displays. You have seen that the left display has an address of $C16X_{16}$. The X stands for some number between 0 and F, depending on which segment of that display we wish to use. The next display to the right has an address of $C15X_{16}$; etc.

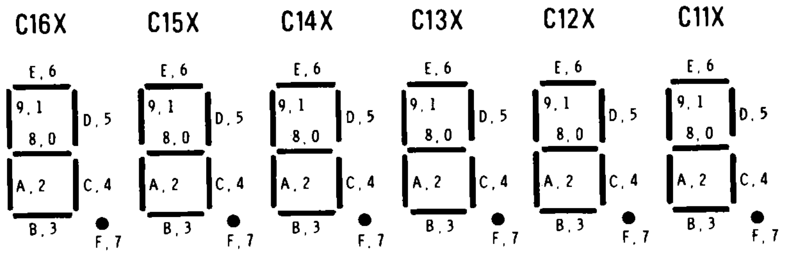


Figure 75
Addresses of the various display segments.

13. Now return to the program shown in Figure 74. Addresses 0003 and 0004 contain the address of the affected display. By changing this address, we can move the character to a different display. Actually since all display addresses start with C1, we need only change the number at address 0004.

14. Change the byte at 0004 to 50_{16}. Change the byte at $000B_{16}$ to 58. Execute the program using the DO command. The character should appear in the second display from the left.

15. Change the byte at 0004 to 10_{16} and the byte at 000B to 18_{16}. Execute the program using the DO command. The character should appear in the right-most display.

DISCUSSION

It has probably occurred to you that the monitor program must have a subroutine that performs this same function. Fortunately, this subroutine is written in such a way that we can use it. It is called OUTCH for OUTput CHaracter. It starts at address $FE3A_{16}$. We can call this subroutine anytime we like by using the JSR instruction. This subroutine assumes that the character byte is in accumulator A.

PROCEDURE

16. Load the program shown in Figure 76. Verify that you loaded it properly.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	86	LDAA#	Load accumulator A immediate with the
0001	37	37	character byte for the letter H.
0002	BD	JSR	Jump to subroutine
0003	FE	FE	OUTCH
0004	3A	3A	
0005	86	LDAA#	Load ACCA with
0006	4F	4F	next character byte.
0007	BD	JSR	
0008	FE	FE	Display it.
0009	3A	3A	
000A	86	LDAA#	Load next character.
000B	0E	0E	
000C	BD	JSR	
000D	FE	FE	Display it.
000E	3A	3A	
000F	86	LDAA#	Load next character.
0010	67	67	
0011	BD	JSR	
012	FE	FE	Display it.
0013	3A	3A	
0014	3E	WAI	Stop.

Figure 76
This program uses the OUTCH subroutine in the monitor program to display a message.

17. Execute the program using the DO command. What message does the program write? ________.

18. Notice that each character is written in a different display. Thus, the subroutine OUTCH automatically changes the address to that of the next display after each character is written.

DISCUSSION

The monitor program writes several messages of its own. Examples are: ACCA, ACCB, CPU UP, and FULL! Thus, the monitor has a subroutine that can be used to write messages. It is called OUTSTR for OUTput a STRing of characters. Its starting address is at $FE52_{16}$. There is a special convention for calling this subroutine. The JSR $FE52_{16}$ instruction must be followed immediately by the character bytes that make up the message. Up to six characters can be displayed. The last character must have the decimal point lit. After the message is displayed, control is returned to the instruction immediately following the last character.

PROCEDURE (Continued)

19. Load the program shown in Figure 77 into the Trainer and verify that you loaded it properly.

HEX ADDRESS	HEX CONTENTS	MNEMONIC/ CONTENTS	COMMENTS
0000	BD	JSR	Jump to the subroutine that
0001	FE	FE	will display the following message.
0002	52	52	
0003	37	37	H
0004	4F	4F	E
0005	0E	0E	L
0006	E7	E7	P. ← Decimal point must be lit in last character.
0007	3E	WAI	Then stop.

Figure 77
The OUTSTR subroutine in the monitor is used to display a message.

20. Execute the program using the DO command. What message does it display? ________.

21. Modify the program so that it displays HELLO.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	BD	JSR	
0001	FE	FE	Cal OUTSTR.
0002	52	52	
0003	76	76	N
0004	FE	FE	O. ◄— Decimal point lit (last character).
0005	BD	JSR	
0006	FE	FE	Call OUTSTR again.
0007	52	52	
0008	5E	5E	G
0009	FE	FE	O. ◄— Decimal point lit (last character).
000A	3E	WAI	Then stop.

Figure 78
OUTSTR is called twice.

22. The program shown in Figure 78 calls the OUTSTR subroutine twice. Load this program into the Trainer.

23. Execute it using the DO command. What message is displayed? ___________.

24. Notice that the second message (GO.) is written to the right of the first. Thus, subroutine OUTSTR does not reset the display to the left for the second message.

25. Rewrite the program so that two blank displays appear between NO. and GO.

DISCUSSION

When displaying long messages such as: "HELLO CAN I HELP YOU?", the display must be given no more than six characters at a time. Also, a short delay must be placed between the various parts of the message. A delay of over one second can be achieved by loading the index register with FFFF and decrementing it to 0000. We can write a display subroutine and call it between each part of the message.

Also, because we are using the same displays over again for each part of the message, each new word should start on the left. The subroutine called OUTSTR has an alternate entry point at address $FD8C_{16}$ called OUTSTJ. The calling convention for this subroutine is the same as that for OUTSTR. However, each new message starts in the left-most display.

PROCEDURE (Continued)

26. Load the program shown in Figure 79. Verify that you loaded it properly.

27. Execute the program using the DO command. What message is displayed? ______________________.

28. Change the number in address 0030_{16} to 40_{16}.

29. Execute the program using the DO command. What affect does this have?

30. Write a program of your own that will display "LOAD 2 IS BAD."

DISCUSSION

The monitor program in the Trainer contains some other useful subroutines. These are outlined in the manual for the ET-3400 Microprocessor Trainer. Two of the most useful are REDIS and OUTBYT.

OUTBYT is a subroutine that displays the contents of accumulator A as two hex digits. Its address is $FE20_{16}$. When this subroutine is called for the first time, the two left displays are used. If it is called again without being reset, the two center displays are used. The third time, the two right displays are used.

The display can be reset to the left by calling the REDIS subroutine. This subroutine is located in address $FCBC_{16}$. If OUTBYT is called after REDIS is called, the two left displays will be used.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	BD	JSR	
0001	FD	FD	Call OUTSTJ.
0002	8C	8C	
0003	37	37	H
0004	4F	4F	E
0005	0E	0E	L
0006	0E	0E	L
0007	FE	FE	O.
0008	BD	JSR	
0009	00	00	Call Delay Subroutine
000A	2F	2F	
000B	BD	JSR	
000C	FD	FD	Call OUTSTJ again.
000D	8C	8C	
000E	4E	4E	C
000F	77	77	A
0010	76	76	N
0011	00	00	blank
0012	B0	B0	I.
0013	BD	JSR.	
0014	00	00	Call Delay Subroutine
0015	2F	2F	
0016	BD	JSR	
0017	FD	FD	Call OUTSTJ again.
0018	8C	8C	
0019	37	37	H
001A	4F	4F	E
001B	0E	0E	L
001C	67	67	P
001D	80	80	•
001E	BD	JSR	Call Delay Subroutine
001F	00	00	
0020	2F	2F	
0021	BD	JSR	Call OUTSTJ again.
0022	FD	FD	
0023	8C	8C	
0024	3B	3B	Y
0025	7E	7E	O
0026	3E	3E	U
0027	00	00	blank
0028	80	80	•
0029	BD	JSR	
002A	00	00	
002B	2F	2F	
002C	7E	JMP	Do it all again.
002D	00	00	
002E	00	00	
002F	CE	LDX#	Delay subroutine.
0030	FF	FF	
0031	FF	FF	
0032	09	DEX	
0033	26	BNE	
0034	FD	FD	
0035	39	RTS	

Figure 79
This program makes extensive use of the subroutine call.

PROCEDURE (Continued)

31. Load the program shown in Figure 80. Verify that you loaded it properly.

Figure 80
Using the OUTBYT and REDIS subroutines.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	4F	CLRA	Clear accumulator A
0001	BD	JSR	
0002	FE	FE	Call OUTBYT
0003	20	20	
0004	BD	JSR	
0005	00	00	Call Delay Subroutine
0006	0E	0E	
0007	4C	INCA	Increment accumulator A
0008	BD	JSR	
0009	FC	FC	Call REDIS
000A	BC	BC	
000B	7E	JMP	
000C	00	00	Do it again.
000D	01	01	
000E	CE	LDX#	
000F	FF	FF	
0010	FF	FF	
0011	09	DEX	Delay Subroutine.
0012	26	BNE	
0013	FD	FD	
0014	39	RTS	

32. Execute the program using the DO command.

33. Which digits are used by the display? ________.

34. Notice that the JSR instruction at address 0008 calls the subroutine that resets the display to the left.

35. To illustrate why this is necessary, let's see what happens when this important step is omitted. Change the contents of locations 0008, 0009, and 000A to 01. This replaces the JSR instruction with three NOPs.

36. Execute the program using the DO command. Notice that, without calling the REDIS subroutine, the display advances to the right and is lost after the third time through the loop.

37. Restore the program to its original state. How can the count be speeded up?

DISCUSSION

The speed of the count can be varied by changing the contents of addresses 000F and 0010. It probably has occurred to you that the trainer could be turned into a digital clock. In the following procedure, you will develop a program that will do this.

PROCEDURE

38. Write a program that will count seconds from 00 to 99_{10}. The seconds count should be maintained in the two left-most displays. It should count as the above program did, but in decimal instead of hexadecimal.

39. If you have problems, remember that the DAA instruction can be used to convert the addition of BCD numbers to a BCD sum. However, the DAA instruction works only if preceeded immediately by an ADDA or ADCA instruction.

40. Load your program into the Trainer and execute it using the DO command.

DISCUSSION

One solution is shown in Figure 81. Carefully study this program. This routine counts the seconds in decimal. However in a real digital clock, the seconds reset to 00 after 59_{10} rather than after 99_{10}.

There are two one-second delay sub-routines listed in the following experiments. You must use the one that matches the clock frequency of your ET-3400 Trainer.

The original trainer has a clock frequency of approximately 500 kHz. If your Trainer has not been modified, you must use the "**Slow Clock** One-Second Delay Subroutine."

If your Trainer has been modified for use with the Heathkit Memory I/O Accessory ETA-3400, it has a clock frequency of 1 MHz. In this case, you must use the "**Fast Clock** One-Second Delay Subroutine."

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	4F	CLRA	Clear seconds.
0001	BD	JSR	
0002	FE	FE	Call OUTBYT
0003	20	20	
0004	BD	JSR	
0005	00	00	Call Delay subroutine
0006	10	10	
0007	8B	ADDA#	Increment seconds
0008	01	01	
0009	19	DAA	Make it decimal
000A	BD	JSR	
000B	FC	FC	Call REDIS
000C	BC	BC	
000D	7E	JMP	
000E	00	00	Do it all again.
000F	01	01	
* 0010	CE	LDX#	Slow Clock One-Second Delay Subroutine
0011	C5	C5	
0012	00	00	
0013	09	DEX	
0014	26	BNE	
0015	FD	FD	
0016	39	RTS	

cont'd.

cont'd.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0010	36	PSHA	
0011	86	LDAA#	
0012	02	02	
0013	CE	LDX#	
0014	F3	F3	
0015	80	80	Fast Clock
0016	09	DEX	One-Second
0017	26	BNE	
0018	FD	FD	Delay Subroutine
0019	4A	DECA	
001A	26	BNE	
001B	F7	F7	
001C	32	PULA	
001D	39	RTS	

*Use either the Fast Clock or the Slow Clock One-Second Delay Subroutine.

Figure 81
This routine counts seconds from 00 to 99.

PROCEDURE (Continued)

41. Modify your program (or the one in this Experiment) so that it displays seconds from 00 to 59 and then returns to 00 and starts over again.

42. Load your program into the Trainer and execute it using the DO command.

43. Debug your program if necessary until it performs properly.

DISCUSSION

One solution is shown in Figure 82. The seconds count is compared to 60 each time it is incremented. When it reaches 60, it is reset to 00.

The next step is to add a minutes count. This can be done by incrementing a decimal number each time the seconds count "rolls over" from 59 to 00. The decimal number is then displayed as minutes.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	C6	LDAB#	Load number for comparison
0001	60	60	
0002	4F	CLRA	Clear seconds.
0003	BD	JSR	
0004	FE	FE	Call OUTBYT
0005	20	20	
0006	BD	JSR	
0007	00	00	Call Delay Subroutine
0008	14	14	
0009	BD	JSR	
000A	FC	FC	Call REDIS
000B	BC	BC	
000C	8B	ADDA#	Increment seconds.
000D	01	01	
000E	19	DAA	Make it decimal
000F	11	CBA	Time to clear seconds
0010	27	BEQ	Yes.
0011	F0	F0	
0012	20	BRA	No.
0013	EF	EF	
* 0014	CE	LDX#	Slow Clock One-Second Delay Subroutine
0015	C5	C5	
0016	00	00	
0017	09	DEX	
0018	26	BNE	
0019	FD	FD	
001A	39	RTS	
0014	36	PSHA	Fast Clock One-Second Delay Subroutine
0015	86	LDAA#	
0016	02	02	
0017	CE	LDX#	
0018	F3	F3	
0019	80	80	
001A	09	DEX	
001B	26	BNE	
001C	FD	FD	
001D	4A	DECA	
001E	26	BNE	
001F	F7	F7	
0020	32	PULA	
0021	39	RTS	

*Use either the Fast Clock or the Slow Clock One-Second Delay Subroutine.

Figure 82
This routine counts seconds from 00 to 59.

PROCEDURE (Continued)

44. Write a program that will display minutes and seconds properly. The minutes should be displayed in the two left displays; the seconds in the two center displays. Like the seconds, the minutes should return to 00 after 59.

45. Load your program and execute it.

46. Debug your program as necessary.

DISCUSSION

A solution is shown in Figure 83. Your approach may be more straightforward, but may require more memory.

The final step is to include the hours display.

PROCEDURE (Continued)

47. Modify your program so that it displays hours, minutes and seconds.

48. Load your program and execute it.

49. Debug your program as necessary.

A solution is shown in Figure 84. This program evolved over a period of time and is extremely compact. It is virtually impossible for a beginning programmer to write a program this compact on the first try. Your program may require substantially more memory, but the important thing is: does it work?

While you can "fine tune" the slow-clock period by changing the numbers in addresses 0004 and 0005, the clock will never be very accurate because it is temperature sensitive. The fast clock period is much more accurate because the oscillator is crystal controlled. You can fine tune it by changing the numbers in addresses 003A and 003B. In a later experiment, you will rectify this problem and produce an extremely accurate clock.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	00	00	Reserved for seconds
0001	00	00	Reserved for minutes
* 0002	CE (36)	LDX# (PSHA)	Slow Clock One-Second delay. (call Fast Clock One-Second Delay)
0003	C5 (BD)	C5 (JSR)	
0004	00 (00)	00 (00)	
0005	09 (2F)	DEX (2F)	
0006	26 (32)	BNE (PULA)	
0007	FD (01)	FD (NOP)	
0008	C6	LDAB#	Load number for comparison.
0009	60	60	
000A	0D	SEC	Set carry bit.
000B	8D	BSR	Branch to subroutine to
000C	11	11	increment seconds.
000D	8D	BSR	Branch to the same subroutine
000E	0F	0F	to increment minutes.
000F	BD	JSR	
0010	FC	FC	Call REDIS
0011	BC	BC	
0012	96	LDAA	Load minutes
0013	01	01	
0014	BD	JSR	
0015	FE	FE	Call OUTBYT to
0016	20	20	display minutes.
0017	96	LDAA	Load seconds
0018	00	00	
0019	BD	JSR	Call OUTBYT to
001A	FE	FE	display seconds
001B	20	20	
001C	20	BRA	Do it all again.
001D	E4	E4	
001E	A6	LDAA, X	Load seconds (or minutes) into A. (Increment subroutine)
001F	00	00	
0020	89	ADCA#	Increment if necessary
0021	00	00	
0022	19	DAA	Adjust to decimal
0023	11	CBA	Time to clear?
0024	26	BNE	No.
0025	01	01	
0026	4F	CLRA	Yes.
0027	A7	STAA, X	Store seconds (or minutes)
0028	00	00	
0029	08	INX	
002A	07	TPA	
002B	88	EORA#	Complement carry bit
002C	01	01	
002D	06	TAP	
002E	39	RTS	

cont'd.

cont'd.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
002F	(86)	(LDAA#)	Fast Clock One-Second Delay Subroutine
0030	(02)	(02)	
0031	(CE)	(LDX#)	
0032	(F3)	(F3)	
0033	(80)	(80)	
0034	(09)	(DEX)	
0035	(26)	(BNE)	
0036	(FD)	(FD)	
0037	(4A)	(DECA)	
0038	(26)	(BNE)	
0039	(F7)	(F7)	
003A	(39)	(RTS)	

*Numbers in parenthesis are for Fast Clock One-Second Delay only.

Figure 83
Routine for displaying minutes and seconds.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS	
0000	00	00	Reserved for seconds	
0001	00	00	Reserved for minutes	
0002	00	00	Reserved for hours	
* 0003	CE (36)	LDX# (PSHA)		
0004	C5 (BD)	C5 (JSR)	Slow Clock (Call Fast Clock	
0005	00 (00)	00 (00)	One-Second One-Second	
0006	09 (37)	DEX (37)	Delay Delay)	
0007	26 (32)	BNE (PULA)		
0008	FD (01)	FD (NOP)		
0009	C6	LDAB#	Minutes and seconds will	
000A	60	60	be compared with sixty.	
000B	0D	SEC	Prepare to increment seconds	
000C	8D	BSR	Go to subroutine that will	
000D	11	11	increment seconds.	
000E	8D	BSR	Go to same subroutine. It will increment	
000F	0F	0F	minutes if necessary.	
0010	C6	LDAB#	Hours will be compared	
0011	12	12	with twelve.	
0012	8D	BSR	Go to same subroutine. It will increment	
0013	0B	0B	hours if necessary.	
0014	BD	JSR		
0015	FC	FC	Call REDIS	
0016	BC	BC		
0017	8D	BSR	Call display subroutine to display	
0018	17	17	hours.	
0019	8D	BSR	Call display subroutine to display	
001A	15	15	minutes.	
001B	8D	BSR	Call display subroutine to display	
001C	13	13	seconds.	
001D	20	BRA	Do it all again.	
001E	E4	E4		
001F	A6	LDAA, X	Load seconds (or minutes or hours).	
0020	00	00		
0021	89	ADCA#	Increment if necessary.	
0022	00	00		
0023	19	DAA	Adjust to decimal.	
0024	11	CBA	Time to clear?	
0025	25	BCS	No.	
0026	01	01		
0027	4F	CLRA	Yes.	Increment Subroutine
0028	A7	STAA, X	Store seconds (or minutes or hours).	
0029	00	00		
002A	08	INX	Point index register at minutes (or hours).	
002B	07	TPA		
002C	88	EORA#	Complement carry bit	
002D	01	01		
002E	06	TAP		
002F	39	RTS		
0030	09	DEX	Point index register at hours (or minutes or seconds)	
0031	A6	LDAA, X	Load hours (or minutes or seconds)	
0032	00	00		
0033	7E	JSR	Display hours (or minutes or seconds)	Display Subroutine
0034	FE	FE		
0035	20	20		
0036	39	RTS		

cont'd.

cont'd.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0037	(86)	(LDAA#)	Fast Clock One-Second Delay Subroutine
0038	(02)	(02)	
0039	(CE)	(LDX#)	
003A	(F3)	(F3)	
003B	(80)	(80)	
003C	(09)	(DEX)	
003D	(26)	(BNE)	
003E	(FD)	(FD)	
003F	(4A)	(DECA)	
0040	(26)	(BNE)	
0041	(F7)	(F7)	
0042	(39)	(RTS)	

*Numbers in parentheses are for Fast Clock One-Second Delay only.

Figure 84
Twelve-hour clock program.

EXAMINATIONS

Unit 1 Examination

Number Systems and Codes

This and the later unit examinations will test your knowledge of the important facts in each unit. These exams will show you what you have learned and what you need to review. Your instructor has the answers.

To take the exam, read each question completely and select the correct answer. Circle the letter (A, B, C, or D) beside the answer that you feel is most correct. It is usually a good idea to read the question and answer choices twice to be sure you fully understand what is being asked.

1. What is the decimal equivalent for the binary number 11010101?

   A. 106
   B. 185
   C. 213
   D. 426

2. What is the decimal equivalent for the binary number 0.110110?

   A. 0.0185185
   B. 0.140625
   C. 0.54
   D. 0.84375

3. What is the binary equivalent for the decimal number 105.0625?

   A. 1101001.0001
   B. 1010110.0001
   C. 1101001.1001
   D. 1010110.1001

4. What is the binary equivalent for the octal number 752.31?

A. 100011001.010001
B. 101100111.001100
C. 111101010.011001
D. 111110100.011001

5. What is the octal equivalent for the binary number 1110101011.1111?

A. 1653.74
B. 3523.17
C. 4653.71
D. 7251.33

6. What is the binary equivalent for the hexadecimal number A8C.5F?

A. 100110001010.01011111
B. 101010001100.01011111
C. 101010001100.01101111
D. 101110001101.10101111

7. What is the hexadecimal equivalent for the binary number 1101011010.00101?

A. 35A.05
B. 35A.28
C. D62.05
D. D62.28

8. What is the hexadecimal equivalent for the decimal number 536.0625

A. 134.01
B. 178.1
C. 218.1
D. 268.01

9. Which of the following is considered an alphanumeric code?

A. 8421 BCD
B. Gray
C. Binary
D. BAUDOT

10. Which of the following columns contain Gray code?

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

Unit 2 Examination

Microcomputer Basics

1. In microprocessor terminology, the number or piece of data that is operated upon is called the:

   A. Operand.
   B. Opcode.
   C. Address.
   D. Instruction.

2. The part of the instruction that tells the microprocessor what operation to perform is called the:

   A. Operand.
   B. Opcode.
   C. Address.
   D. Mnemonic.

3. The portion of the microcomputer in which instructions and data are stored is called the:

   A. ALU.
   B. MPU.
   C. RAM.
   D. Data bus.

4. An 8-bit byte in memory can represent an:

   A. Opcode.
   B. Operand.
   C. Address.
   D. All of the above.

5. During the fetch phase:

   A. The opcode is fetched from memory and is decoded.
   B. The address of the operand is fetched from memory and is decoded.
   C. The operand is fetched from memory and is operated upon.
   D. The program count is fetched from memory.

6. In what register is the result of an arithmetic operation normally placed?

   A. The data register.
   B. The address register.
   C. The arithmetic logic unit (ALU).
   D. The accumulator.

7. During the fetch and execute phases of the "load accumulator direct" instruction, the information on the data bus will be:

   A. The operand address followed by the operand.
   B. The program count, followed by the opcode, followed by the operand address, followed by the operand.
   C. The opcode, followed by the operand address, followed by the operand.
   D. The opcode, followed by the operand.

8. In the immediate addressing mode, the second byte of the instruction is the:

   A. Opcode of the instruction.
   B. Number that is to be operated upon.
   C. Address of the operand.
   D. Address of the opcode.

9. The program shown in Figure 85 below:

   A. Adds the contents of memory location 09, 0A, and 0B.
   B. Stores 00 in locations 09, 0A, 0B.
   C. Stores 09 in location 03, 0A in location 05, and 0B in location 07.
   D. Stores 0B in the accumulator.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS
00	86	LDA
01	00	00_{16}
02	97	STA
03	09	09_{16}
04	97	STA
05	0A	$0A_{16}$
06	97	STA
07	0B	$0B_{16}$
08	3E	HLT
09	—	—
0A	—	—
0B	—	—

Figure 85
Program for Question 13.

10. The program shown in Figure 86:

A. Swaps the contents of memory location 0D and 0E.
B. Stores AA_{16} in locations 0D, 0E, and 0F.
C. Stores BB_{16} in locations 0D, 0E, and 0F.
D. Adds AA and BB, storing the sum at location 0F.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS
00	96	LDA
01	0D	$0D_{16}$
02	97	STA
03	0F	$0F_{16}$
04	96	LDA
05	0E	$0E_{16}$
06	97	STA
07	0D	$0D_{16}$
08	96	LDA
09	0F	$0F_{16}$
0A	97	STA
0B	0E	$0E_{16}$
0C	3E	HLT
0D	AA	AA_{16}
0E	BB	BB_{16}
0F	—	—

Figure 86
Program for Question 15.

Unit 3 Examination

Computer Arithmetic

1. What is the sum of binary 10010110 and binary 1101?

   A. 10010110
   B. 10011011
   C. 10100011
   D. 10111111

2. If you subtract binary 1011 from binary 10110110, what is the difference?

   A. 10110100
   B. 10101011
   C. 10110001
   D. 10111010

3. The product of binary 1001 and binary 1100 is:

   A. 1101100
   B. 1101101
   C. 1110011
   D. 1110100

4. Divide binary 100111 with binary 110, and the quotient is:

   A. 110
   B. 110.1
   C. 110.11
   D. 111

5. What is the 2's complement of binary 00110110?

   A. 00110110
   B. 11001000
   C. 11001001
   D. 11001010

6. Using 2's complement arithmetic, what is the binary and decimal sum of decimal +75 and −6?

   A. 11000101 = −69
   B. 11010001 = −81
   C. 01010001 = 81
   D. 01000101 = 69

7. Using 2's complement arithmetic, what is the binary and decimal difference of decimal −15 subtracted from −85.

   A. 11100100 = −100
   B. 01100100 = 100
   C. 10111010 = −70
   D. 01000110 = 70

8. If you logically AND binary 11011010 with binary 10010110, the output is:

   A. 00100101
   B. 01001100
   C. 11011110
   D. 10010010

9. If you logically OR binary 11011010 with binary 10010110, the output is:

   A. 00100101
   B. 01001100
   C. 11011110
   D. 10010010

10. If you logically EOR binary 11011010 with binary 10010110, the output is:

   A. 00100101
   B. 01001100
   C. 11011110
   D. 10010010

Unit 4 Examination

Introduction to Programming

1. The BRA instruction will cause a branch to occur:

   A. Anytime that it is executed.
   B. Only if the Z flag is set.
   C. Only if the N flag is set.
   D. Only if the C flag is set.

2. The address that follows the opcode of an unconditional branch instruction is:

   A. The address of the operand.
   B. The address of the next opcode to be executed.
   C. Added to the program count to form the address of the next opcode to be executed.
   D. Added to the program count to form the address of the operand that is to be tested to see if a branch operation is required.

3. The opcode for an unconditional branch instruction is at address AF_{16}. The relative address is $0F_{16}$. From what address will the next opcode be fetched?

   A. $A0_{16}$.
   B. $C0_{16}$.
   C. BE_{16}.
   D. $B1_{16}$.

4. The opcode for an unconditional branch instruction is at address 30_{16}. The relative address is EF_{16}. From what address will the next opcode be fetched?

   A. 21_{16}.
   B. EF_{16}.
   C. 32_{16}.
   D. 19_{16}.

5. The carry register:

   A. Acts like the ninth bit of the accumulator.
   B. Is set when a "borrow" for bit 7 of the accumulator occurs.
   C. Is set when a carry from bit 7 occurs.
   D. All the above.

6. The numbers $0101\ 1000_2$ and $0110\ 0011_2$ are added using the ADD instruction. Immediately after the ADD instruction is executed, the condition code registers will indicate the following:

   A. C=1, N=1, V=1, Z=0.
   B. C=0, N=1, V=1, Z=0.
   C. C=0, N=0, V=1, Z=1.
   D. C=0, N=0, V=1, Z=1.

7. The divide program shown in Figure 4-16 of the text works only if the dividend is initially less than $+\ 128_{10}$. The program can be modified to work for dividends up to 255_{10} by replacing the BMI instruction with the:

   A. BEQ instruction.
   B. BNE instruction.
   C. BCC instruction.
   D. BCS instruction.

8. A binary number can be converted to BCD by repeatedly:

   A. Dividing by powers of two.
   B. Subtracting powers of ten.
   C. Multiplying by powers of two.
   D. Adding powers of ten.

9. The DAA instruction is used:

   A. To convert a binary number to BCD.
   B. To convert a BCD number to binary.
   C. After an add instruction to adjust the sum to a BCD number.
   D. After a subtract instruction to adjust the difference to a BCD number.

10. When you are adding multiple-precision binary numbers, all bytes except the least significant ones must be:

    A. Added using the ADD instruction.
    B. Added using the DAA instruction.
    C. Added using the ADC instruction.
    D. Decimal adjusted before addition takes place.

Unit 5 Examination

The 6800 Microprocessor — Part 1

1. Which of the following program segments will **not** clear both accumulators?

   A. CLRA
      CLRB

   B. CLRA
      TAB

   C. CLRB
      TBA

   D. CLRA
      ABA

2. Which of the following contains an operation that can **not** be performed directly on a byte in memory using a single instruction?

   A. Increment, decrement, shift left arithmetically.
   B. Clear, complement, compare.
   C. Rotate left, negate, test for zero.
   D. Shift right logically, rotate right, test for minus.

3. Which addressing mode is best suited for adding a list of numbers?

   A. Direct.
   B. Extended.
   C. Indexed.
   D. Relative.

4. Which of the following program segments will sucessfully swap the contents of accumulators A and B?

```
A.  TAB            C.  TAB
    TBA                ABA

B.  STAA           D.  STAA
    10                 10
    TBA                LDAB
    LDAB               10
    10                 TBA
```

5. Which of the following program segments will cause a branch if the number in memory location 8310 is odd?

```
A.  ROR            C.  RORA
    83                 BCS
    10                 07
    BCS
    07             D.  LDAA
                       83
B.  ASL                10
    83                 ROLA
    10                 BCS
    BCS                07
    07
```

6. Examine the following program segment:

```
      CLRA
  ┌──►INCA
  │    BNE
  └──── FD
      WAI
```

If an MPU cycle is 1 microsecond, how much time elapses from the time this segment starts running until the WAI instruction is fetched?

A. Approximately 8 microseconds.
B. Approximately 2050 microseconds.
C. Approximately 1538 microseconds.
D. Approximately 3 microseconds.

7. Which of the following instructions can be used to clear the Z flag?

   A. BEQ
   B. BNE
   C. NOP
   D. TAP

8. Which of the following instructions can be used to test the result of the subtraction of unsigned binary numbers?

   A. BGE.
   B. BGT.
   C. BCS.
   D. BLT

9. Examine the following program segment:

```
   LDX #
      00
      50
┌──→ DEX
│    BNE
└─── FD
    WAI
```

How many times will the DEX instruction be executed?

   A. Once
   B. 50_{16} times.
   C. $65,536_{10}$.
   D. The number of times will depend on the contents of memory location 0050.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ HEX CONTENTS	COMMENTS
0010	4F	CLRA	Clear Accumulator A.
0011	7D	TST	Test
0012	00	00	the
0013	1E	1E	multiplier.
0014	27	BEQ	If it is zero branch to wait.
0015	07	07	
0016	7A	DEC	Otherwise decrement
0017	00	00	the
0018	1E	1E	multiplier.
0019	9B	ADDA	Add the
001A	1F	1F	multiplicand to the product.
001B	20	BRA	Repeat the loop.
001C	F4	F4	
001D	3E	WAI	Wait.
001E	05	Multiplier	
001F	04	Multiplicand	

Figure 87
This program multiplies by repeated addition.

NOTE: Refer to the program shown in Figure 87 for questions 10 through 16

10. What addressing mode does the TST instruction use?

A. Immediate
B. Direct.
C. Extended.
D. Indexed.

11. The BEQ instruction checks to see if the TST instruction set the:

A. Z flag
B. C flag.
C. H flag.
D. V flag.

12. The DEC instruction decrements the number in:

A. Accumulator A.
B. Memory location 001E.
C. Accumulator B.
D. The index register.

13. Which instruction is executed immediately after the BRA instruction?

    A. WAI.
    B. BEQ.
    C. CLRA.
    D. TST.

14. With the values given for the multiplier and multiplicand, how many times will the main program loop be repeated?

    A. Four times.
    B. Five times.
    C. Twenty times.
    D. Twice.

15. After the program has been executed, memory location 001E will contain:

    A. 05_{16}.
    B. 04_{16}.
    C. 20_{16}.
    D. 00_{16}.

16. After the program has been executed, the product will appear in:

    A. Memory location 001E.
    B. Memory location 001F.
    C. Accumulator A.
    D. Accumulator B.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ HEX CONTENTS	COMMENTS
0010	CE	LDX #	
0011	00	00	
0012	00	05	
0013	A6	LDAA. X	
0014	20	20	
0015	AB	ADDA. X	
0016	30	30	
0017	A7	STAA. X	
0018	40	40	
0019	08	INX	
001A	8C	CPX #	
001B	00	00	
001C	15	15	
001D	26	BNE	
001E	F4	F4	
001F	3E	WAI	

Figure 88
Program for Questions 17 through 20.

NOTE: Refer to Figure 88 for Questions 17 through 20. Analyze the program, determine what it does, and fill in appropriate comments.

17. On the first pass through the main program loop, the LDAA, X instruction takes its operand from memory location:

A. 0005.
B. 0020.
C. 0025.
D. 0014.

18. On the first pass, the ADDA, X adds the contents of what memory loction to accumulator A?

    A. 0005.
    B. 0030.
    C. 0035.
    D. 0016.

19. On the second pass through the program loop, the contents of memory location:

    A. 0021 are added to the contents of 0031 and the result is stored in 0041.
    B. 0026 are added to the contents of 0036 and the result is stored in 0046.
    C. 0025 are added to the contents of 0035 and the result is stored in 0045.
    D. 0020 are added to the contents of 0030 and the result is stored in 0040.

20. How many times is the main program loop repeated?

    A. 10_{16} times.
    B. 05_{16} times.
    C. 30_{16} times.
    D. 15_{16} times.

Unit 6 Examination

The 6800 Microprocessor — Part 2

1. If the I bit in the condition code register is set, the MPU will ignore:
   A. The reset signal.
   B. The non-maskable interrupt signal.
   C. The interrupt request signal.
   D. The software interrupt instruction.

2. Which of the following lists contains instructions that do **not** change the contents of the stack pointer?
   A. PULA, DES, RTI.
   B. PSHB, INS, SWI.
   C. TXS, BSR, LDS.
   D. JMP, TSX, STS.

3. Which of the following program segments will successfully swap the contents of the two accumulators?

```
A.  PSHA     B.  PSHB     C.  PSHA     D.  PSHB
    TAB          TAB          TBA          TBA
    PULA         PULA         PULA         PULB
```

4. The stack pointer is automatically:
   A. Decremented before data is pushed onto the stack.
   B. Incremented before data is pushed onto the stack.
   C. Decremented after data is pushed onto the stack.
   D. Incremented after data is pushed onto the stack.

5. One difference between the JMP and JSR instruction is:
   A. JMP can use either extended or indexed addressing.
   B. The program count is saved when JSR is executed.
   C. The JSR will be executed even if the interrupt mask is set.
   D. JMP is an unconditional jump.

6. The last instruction in a subroutine is generally:
   A. A JMP instruction.
   B. An RTS instruction.
   C. An RTI instruction.
   D. A JSR instruction.

7. In the 6800 MPU, which of the following instructions could be used to transfer data from an I/O device to accumulator A?
   A. INPA.
   B. LDAA.
   C. STAA.
   D. OUTA.

8. Refer to Figures 6-17 and 6-18. Which of the following program segments will read in data from the switch bank and, if the number is larger than $2A_{16}$, display it on the LED's?

```
A.  LDAA        B.  LDAA        C.  LDAA        D.  LDAA
     80              80              80              90
     00              00              00              00
    CMPA#           SUBA#           STAA            SUBA#
     2A              2A              90              2A
     BHI             BHI             00              BHI
     01              01                              01
     WAI             WAI                             WAI
    STAA            STAA                             STA
     90              90                              80
     00              00                              00
```

9. Which of the following types of interrupts does **not** cause data to be pushed into the stack?
   A. Software interrupt.
   B. Non-maskable interrupt.
   C. Reset interrupt.
   D. Interrupt request.

10. Generally, the last instruction in an interrupt service routine will be:
   A. An RTI instruction.
   B. An SWI instruction.
   C. An RTS instruction.
   D. An NMI instruction.

Appendix A

DEFINITION OF THE EXECUTABLE INSTRUCTIONS

A.1 Nomenclature

The following nomenclature is used in the subsequent definitions.

(a) *Operators*

( )	=	contents of
$\leftarrow$	=	is transferred to
$\uparrow$	=	"is pulled from stack"
$\downarrow$	=	"is pushed into stack"
$\cdot$	=	Boolean AND
$\odot$	=	Boolean (Inclusive) OR
$\oplus$	=	Exclusive OR
$\approx$	=	Boolean NOT

(b) *Registers in the MPU*

ACCA	=	Accumulator A
ACCB	=	Accumulator B
ACCX	=	Accumulator ACCA or ACCB
CC	=	Condition codes register
IX	=	Index register, 16 bits
IXH	=	Index register, higher order 8 bits
IXL	=	Index register, lower order 8 bits
PC	=	Program counter, 16 bits
PCH	=	Program counter, higher order 8 bits
PCL	=	Program counter, lower order 8 bits
SP	=	Stack pointer
SPH	=	Stack pointer high
SPL	=	Stack pointer low

(c) *Memory and Addressing*

M	=	A memory location (one byte)
M +1	=	The byte of memory at 0001 plus the address of the memory location indicated by "M."
Rel	=	Relative address (i.e. the two's complement number stored in the second byte of machine code corresponding to a branch instruction).

(d) *Bits 0 thru 5 of the Condition Codes Register*

C	=	Carry — borrow	bit — 0
V	=	Two's complement overflow indicator	bit — 1
Z	=	Zero indicator	bit — 2
N	=	Negative indicator	bit — 3
I	=	Interrupt mask	bit — 4
H	=	Half carry	bit — 5

(e) *Status of Individual Bits BEFORE Execution of an Instruction*

An	=	Bit n of ACCA (n=7,6,5,...,0)
Bn	=	Bit n of ACCB (n=7,6,5,...,0)
IXHn	=	Bit n of IXH (n=7,6,5,...,0)

IXLn	=	Bit n of IXL (n=7,6,5,...,0)
Mn	=	Bit n of M (n=7,6,5,...,0)
SPHn	=	Bit n of SPH (n=7,6,5,...,0)
SPLn	=	Bit n of SPL (n=7,6,5,...,0)
Xn	=	Bit n of ACCX (n=7,6,5,...,0)

(f) *Status of Individual Bits of the RESULT of Execution of an Instruction*

(i) For 8-bit Results

Rn = Bit n of the result (n =7,6,5,...,0)

This applies to instructions which provide a result contained in a single byte of memory or in an 8-bit register.

(ii) For 16-bit Results

RHn = Bit n of the more significant byte of the result (n =7,6,5,...,0)

RLn = Bit n of the less significant byte of the result (n =7,6,5,...,0)

This applies to instructions which provide a result contained in two consecutive bytes of memory or in a 16-bit register.

A.2 Executable Instructions (definition of)

Detailed definitions of the 72 executable instructions of the source language are provided on the following pages.

Add Accumulator B to Accumulator A **ABA**

Operation: ACCA ← (ACCA) + (ACCB)

Description: Adds the contents of ACCB to the contents of ACCA and places the result in ACCA.

Condition Codes:
H: Set if there was a carry from bit 3; cleared otherwise.
I: Not affected.
N: Set if most significant bit of the result is set; cleared otherwise.
Z: Set if all bits of the result are cleared; cleared otherwise.
V: Set if there was two's complement overflow as a result of the operation; cleared otherwise.
C: Set if there was a carry from the most significant bit of the result; cleared otherwise.

Boolean Formulae for Condition Codes:

$$H = A_3 \cdot B_3 + B_3 \cdot \overline{R_3} + \overline{R_3} \cdot A_3$$
$$N = R_7$$
$$Z = \overline{R_7} \cdot \overline{R_6} \cdot \overline{R_5} \cdot \overline{R_4} \cdot \overline{R_3} \cdot \overline{R_2} \cdot \overline{R_1} \cdot \overline{R_0}$$
$$V = A_7 \cdot B_7 \cdot \overline{R_7} + \overline{A_7} \cdot \overline{B_7} \cdot R_7$$
$$C = A_7 \cdot B_7 + B_7 \cdot \overline{R_7} + \overline{R_7} \cdot A_7$$

Addressing Modes, Execution Time, and Machine Code (hexadecimal/octal/decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
Inherent	2	1	1B	033	027

ADC

Add with Carry

Operation: ACCX ← (ACCX) + (M) + (C)

Description: Adds the contents of the C bit to the sum of the contents of ACCX and M, and places the result in ACCX.

Condition Codes:
- H Set if there was a carry from bit 3; cleared otherwise.
- I: Not affected.
- N: Set if most significant bit of the result is set; cleared otherwise.
- Z: Set if all bits of the result are cleared; cleared otherwise.
- V: Set if there was two's complement overflow as a result of the operation; cleared otherwise.
- C: Set if there was a carry from the most significant bit of the result; cleared otherwise.

Boolean Formulae for Condition Codes:

$$H = X_3 \cdot M_3 + M_3 \cdot \overline{R}_3 + \overline{R}_3 \cdot X_3$$
$$N = R_7$$
$$Z = \overline{R}_7 \cdot \overline{R}_6 \cdot \overline{R}_5 \cdot \overline{R}_4 \cdot \overline{R}_3 \cdot \overline{R}_2 \cdot \overline{R}_1 \cdot \overline{R}_0$$
$$V = X_7 \cdot M_7 \cdot \overline{R}_7 + \overline{X}_7 \cdot \overline{M}_7 \cdot R_7$$
$$C = X_7 \cdot M_7 + M_7 \cdot \overline{R}_7 + \overline{R}_7 \cdot X_7$$

Addressing Formats:

See Table A-1

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

(DUAL OPERAND)

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A IMM	2	2	89	211	137
A DIR	3	2	99	231	153
A EXT	4	3	B9	271	185
A IND	5	2	A9	251	169
B IMM	2	2	C9	311	201
B DIR	3	2	D9	331	217
B EXT	4	3	F9	371	249
B IND	5	2	E9	351	233

Add Without Carry **ADD**

Operation: ACCX ← (ACCX) + (M)

Description: Adds the contents of ACCX and the contents of M and places the result in ACCX.

Condition Codes:
- H: Set if there was a carry from bit 3; cleared otherwise.
- I: Not affected.
- N: Set if most significant bit of the result is set; cleared otherwise.
- Z: Set if all bits of the result are cleared; cleared otherwise.
- V: Set if there was two's complement overflow as a result of the operation; cleared otherwise.
- C: Set if there was a carry from the most significant bit of the result; cleared otherwise.

Boolean Formulae for Condition Codes:

$$H = X_3 \cdot M_3 + M_3 \cdot \overline{R}_3 + \overline{R}_3 \cdot X_3$$
$$N = R_7$$
$$Z = \overline{R}_7 \cdot \overline{R}_6 \cdot \overline{R}_5 \cdot \overline{R}_4 \cdot \overline{R}_3 \cdot \overline{R}_2 \cdot \overline{R}_1 \cdot \overline{R}_0$$
$$V = X_7 \cdot M_7 \cdot \overline{R}_7 + \overline{X}_7 \cdot \overline{M}_7 \cdot R_7$$
$$C = X_7 \cdot M_7 + M_7 \cdot \overline{R}_7 + \overline{R}_7 \cdot X_7$$

Addressing Formats:

See Table A-1

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):
(DUAL OPERAND)

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A IMM	2	2	8B	213	139
A DIR	3	2	9B	233	155
A EXT	4	3	BB	273	187
A IND	5	2	AB	253	171
B IMM	2	2	CB	313	203
B DIR	3	2	DB	333	219
B EXT	4	3	FB	373	251
B IND	5	2	EB	353	235

AND

Logical AND

Operation: ACCX ← (ACCX) · (M)

Description: Performs logical "AND" between the contents of ACCX and the contents of M and places the result in ACCX. (Each bit of ACCX after the operation will be the logical "AND" of the corresponding bits of M and of ACCX before the operation.)

Condition Codes:
H: Not affected.
I: Not affected.
N: Set if most significant bit of the result is set; cleared otherwise.
Z: Set if all bits of the result are cleared; cleared otherwise.
V: Cleared.
C: Not affected.

Boolean Formulae for Condition Codes:

$N = R_7$

$Z = \overline{R}_7 \cdot \overline{R}_6 \cdot \overline{R}_5 \cdot \overline{R}_4 \cdot \overline{R}_3 \cdot \overline{R}_2 \cdot \overline{R}_1 \cdot \overline{R}_0$

$V = 0$

Addressing Formats:

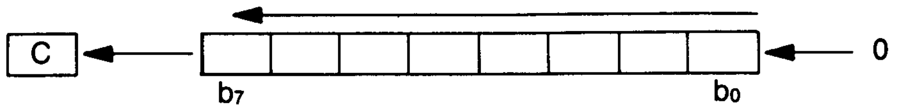
See Table A-1

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	**OCT.**	**DEC.**
A IMM	2	2	84	204	132
A DIR	3	2	94	224	148
A EXT	4	3	B4	264	180
A IND	5	2	A4	244	164
B IMM	2	2	C4	304	196
B DIR	3	2	D4	324	212
B EXT	4	3	F4	364	244
B IND	5	2	E4	344	228

Arithmetic Shift Left **ASL**

Operation:



Description: Shifts all bits of the ACCX or M one place to the left. Bit 0 is loaded with a zero. The C bit is loaded from the most significant bit of ACCX or M.

Condition Codes:
- H: Not affected.
- I: Not affected.
- N: Set if most significant bit of the result is set; cleared otherwise.
- Z: Set if all bits of the result are cleared; cleared otherwise.
- V: Set if, after the completion of the shift operation, EITHER (N is set and C is cleared) OR (N is cleared and C is set); cleared otherwise.
- C: Set if, before the operation, the most significant bit of the ACCX or M was set; cleared otherwise.

Boolean Formulae for Condition Codes:

$N = R_7$

$Z = \overline{R}_7 \cdot \overline{R}_6 \cdot \overline{R}_5 \cdot \overline{R}_4 \cdot \overline{R}_3 \cdot \overline{R}_2 \cdot \overline{R}_1 \cdot \overline{R}_0$

$V = N \oplus C = [N \cdot \overline{C}] \odot [\overline{N} \cdot C]$

(the foregoing formula assumes values of N and C after the shift operation)

$C = M_7$

Addressing Formats

See Table A-3

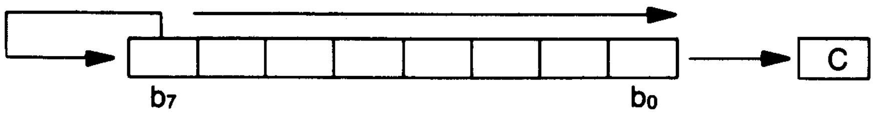
Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A	2	1	48	110	072
B	2	1	58	130	088
EXT	6	3	78	170	120
IND	7	2	68	150	104

ASR

Arithmetic Shift Right

Operation:



Description: Shifts all bits of ACCX or M one place to the right. Bit 7 is held constant. Bit 0 is loaded into the C bit.

Condition Codes:
H: Not affected.
I: Not affected.
N: Set if the most significant bit of the result is set; cleared otherwise.
Z: Set if all bits of the result are cleared; cleared otherwise.
V: Set if, after the completion of the shift operation, EITHER (N is set and C is cleared) OR (N is cleared and C is set); cleared otherwise.
C: Set if, before the operation, the least significant bit of the ACCX or M was set; cleared otherwise.

Boolean Formulae for Condition Codes:

$N = R_7$

$Z = \overline{R_7} \cdot \overline{R_6} \cdot \overline{R_5} \cdot \overline{R_4} \cdot \overline{R_3} \cdot \overline{R_2} \cdot \overline{R_1} \cdot \overline{R_0}$

$V = N \oplus C = [N \cdot \overline{C}] \odot [\overline{N} \cdot C]$

(the foregoing formula assumes values of N and C after the shift operation)

$C = M_0$

Addressing Formats:

See Table A-3

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A	2	1	47	107	071
B	2	1	57	127	087
EXT	6	3	77	167	119
IND	7	2	67	147	103

BCC

Branch if Carry Clear

Operation: PC ← (PC) + 0002 + Rel if (C)=0

Description: Tests the state of the C bit and causes a branch if C is clear.

See BRA instruction for further details of the execution of the branch.

Condition Codes: Not affected.

Addressing Formats:

See Table A-8.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
REL	4	2	24	044	036

BCS

Branch if Carry Set

Operation: PC ← (PC) + 0002 + Rel if (C)=1

Description: Tests the state of the C bit and causes a branch if C is set.
See BRA instruction for further details of the execution of the branch.

Condition Codes: Not affected.

Addressing Formats:
See Table A-8.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
REL	4	2	25	045	037

Branch if Equal **BEQ**

Operation: PC ← (PC) + 0002 + Rel if (Z)=1

Description: Tests the state of the Z bit and causes a branch if the Z bit is set.
See BRA instruction for further details of the execution of the branch.

Condition Codes: Not affected.

Addressing Formats:
See Table A-8.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
REL	4	2	27	047	039

BGE

Branch if Greater than or Equal to Zero

Operation: PC ← (PC) + 0002 + Rel if (N) $\oplus$ (V) = 0

i.e. if (ACCX) $\geqslant$ (M)

(Two's complement numbers)

Description: Causes a branch if (N is set and V is set) OR (N is clear and V is clear).

If the BGE instruction is executed immediately after execution of any of the instructions CBA, CMP, SBA, or SUB, the branch will occur if and only if the two's complement number represented by the minuend (i.e. ACCX) was greater than or equal to the two's complement number represented by the subtrahend (i.e. M).

See BRA instruction for details of the branch.

Condition Codes: Not affected.

Addressing Formats:

See Table A-8.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	**Execution Time (No. of cycles)**	**Number of bytes of machine code**	**Coding of First (or only) byte of machine code**		
			HEX.	**OCT.**	**DEC.**
REL	4	2	2C	054	044

Branch if Greater than Zero **BGT**

Operation: $PC \leftarrow (PC) + 0002 + Rel$ if $(Z) \odot [(N) \oplus (V)] = 0$
i.e. if (ACCX) > (M)
(two's complement numbers)

Description: Causes a branch if [Z is clear] AND [(N is set and V is set) OR (N is clear and V is clear)].

If the BGT instruction is executed immediately after execution of any of the instructions CBA, CMP, SBA, or SUB, the branch will occur if and only if the two's complement number represented by the minuend (i.e. ACCX) was greater than the two's complement number represented by the subtrahend (i.e. M).

See BRA instruction for details of the branch.

Condition Codes: Not affected.

Addressing Formats:

See Table A-8.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
REL	4	2	2E	056	046

BHI

Branch if Higher

Operation: PC ← (PC) + 0002 + Rel if (C) · (Z)=0

i.e. if (ACCX) > (M)
(unsigned binary numbers)

Description: Causes a branch if (C is clear) AND (Z is clear).

If the BHI instruction is executed immediately after execution of any of the instructions CBA, CMP, SBA, or SUB, the branch will occur if and only if the unsigned binary number represented by the minuend (i.e. ACCX) was greater than the unsigned binary number represented by the subtrahend (i.e. M).

See BRA instruction for details of the execution of the branch.

Condition Codes: Not affected.

Addressing Formats:

See Table A-8.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	**Execution Time (No. of cycles)**	**Number of bytes of machine code**	**Coding of First (or only) byte of machine code**		
			HEX.	**OCT.**	**DEC.**
REL	4	2	22	042	034

Bit Test

BIT

Operation: (ACCX) · (M)

Description: Performs the logical "AND" comparison of the contents of ACCX and the contents of M and modifies condition codes accordingly. Neither the contents of ACCX or M operands are affected. (Each bit of the result of the "AND" would be the logical "AND" of the corresponding bits of M and ACCX.)

Condition Codes:
- H: Not affected.
- I: Not affected.
- N: Set if the most significant bit of the result of the "AND" would be set; cleared otherwise.
- Z: Set if all bits of the result of the "AND" would be cleared; cleared otherwise.
- V: Cleared.
- C: Not affected.

Boolean Formulae for Condition Codes:

$N = R_7$

$Z = \overline{R_7} \cdot \overline{R_6} \cdot \overline{R_5} \cdot \overline{R_4} \cdot \overline{R_3} \cdot \overline{R_2} \cdot \overline{R_1} \cdot \overline{R_0}$

$V = 0$

Addressing Formats:

See Table A-1.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/octal/decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A IMM	2	2	85	205	133
A DIR	3	2	95	225	149
A EXT	4	3	B5	265	181
A IND	5	2	A5	245	165
B IMM	2	2	C5	305	197
B DIR	3	2	D5	325	213
B EXT	4	3	F5	365	245
B IND	5	2	E5	345	229

BLE

Branch if Less than or Equal to Zero

Operation: PC ← (PC) + 0002 + Rel if $(Z) \odot [(N) \oplus (V)] = 1$

i.e. if (ACCX) ≤ (M)

(two's complement numbers)

Description: Causes a branch if [Z is set] OR [(N is set and V is clear) OR (N is clear and V is set)].

If the BLE instruction is executed immediately after execution of any of the instructions CBA, CMP, SBA, or SUB, the branch will occur if and only if the two's complement number represented by the minuend (i.e. ACCX) was less then or equal to the two's complement number represented by the subtrahend (i.e. M).

See BRA instruction for details of the branch.

Condition Codes: Not affected.

Addressing Formats:

See Table A-8.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
REL	4	2	2F	057	047

Branch if Lower or Same **BLS**

Operation: PC ← (PC) + 0002 + Rel if $(C) \odot (Z) = 1$
i.e. if (ACCX) ≤ (M)
(unsigned binary numbers)

Description: Causes a branch if (C is set) OR (Z is set).

If the BLS instruction is executed immediately after execution of any of the instructions CBA, CMP, SBA, or SUB, the branch will occur if and only if the unsigned binary number represented by the minuend (i.e. ACCX) was less than or equal to the unsigned binary number represented by the subtrahend (i.e. M).

See BRA instruction for details of the execution of the branch.

Condition Codes: Not affected.

Addressing Formats:

See Table A-8.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	**Execution Time (No. of cycles)**	**Number of bytes of machine code**	**Coding of First (or only) byte of machine code**		
			HEX.	**OCT.**	**DEC.**
REL	4	2	23	043	035

BLT

Branch if Less than Zero

Operation: PC ← (PC) + 0002 + Rel if (N) ⊕ (V) = 1

i.e. if (ACCX) < (M)
(two's complement numbers)

Description: Causes a branch if (N is set and V is clear) OR (N is clear and V is set).

If the BLT instruction is executed immediately after execution of any of the instructions CBA, CMP, SBA, or SUB, the branch will occur if and only if the two's complement number represented by the minuend (i.e. ACCX) was less than the two's complement number represented by the subtrahend (i.e. M).

See BRA instruction for details of the branch.

Condition Codes: Not affected.

Addressing Formats:

See Table A-8.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	**Execution Time (No. of cycles)**	**Number of bytes of machine code**	**Coding of First (or only) byte of machine code**		
			HEX.	**OCT.**	**DEC.**
REL	4	2	2D	055	045

BMI

Branch if Minus

Operation: PC← (PC) + 0002 + Rel if (N) =1

Description: Tests the state of the N bit and causes a branch if N is set.

See BRA instruction for details of the execution of the branch.

Condition Codes: Not affected.

Addressing Formats:

See Table A-8.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
REL	4	2	2B	053	043

BNE

Branch if Not Equal

Operation: PC ← (PC) + 0002 + Rel if (Z) = 0

Description: Tests the state of the Z bit and causes a branch if the Z bit is clear.

See BRA instruction for details of the execution of the branch.

Condition Codes: Not affected.

Addressing Formats:

See Table A-8.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
REL	4	2	26	046	038

Branch if Plus **BPL**

Operation: PC ← (PC) + 0002 + Rel if (N) =0

Description: Tests the state of the N bit and causes a branch if N is clear.

See BRA instruction for details of the execution of the branch.

Condition Codes: Not affected.

Addressing Formats:

See Table A-8.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
REL	4	2	2A	052	042

BRA

Branch Always

Operation: PC ← (PC) + 0002 + Rel

Description: Unconditional branch to the address given by the foregoing formula, in which R is the relative address stored as a two's complement number in the second byte of machine code corresponding to the branch instruction.

Note: The source program specifies the destination of any branch instruction by its absolute address, either as a numerical value or as a symbol or expression which can be numerically evaluated by the assembler. The assembler obtains the relative address R from the absolute address and the current value of the program counter PC.

Condition Codes: Not affected.

Addressing Formats:

See Table A-8.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	**Execution Time (No. of cycles)**	**Number of bytes of machine code**	**Coding of First (or only) byte of machine code**		
			HEX.	**OCT.**	**DEC.**
REL	4	2	20	040	032

Branch to Subroutine **BSR**

Operation: PC ← (PC) + 0002

↓ (PCL)

SP ← (SP) − 0001

↓ (PCH)

SP ← (SP) − 0001

PC ← (PC) + Rel

Description: The program counter is incremented by 2. The less significant byte of the contents of the program counter is pushed into the stack. The stack pointer is then decremented (by 1). The more significant byte of the contents of the program counter is then pushed into the stack. The stack pointer is again decremented (by 1). A branch then occurs to the location specified by the program.

See BRA instruction for details of the execution of the branch.

Condition Codes: Not affected.

Addressing Formats:

See Table A-8.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
REL	8	2	8D	215	141

BRANCH TO SUBROUTINE EXAMPLE

			Memory Location	Machine Code (Hex)	Label	Assembler Language Operator	Operand
A.	*Before*						
	PC	←	$1000	8D		BSR	CHARLI
			$1001	50			
	SP	←	$EFFF				
B.	*After*						
	PC	←	$1052	**	CHARLI	***	*****
	SP	←	$EFFD				
			$EFFE	10			
			$EFFF	02			

BVC

Branch if Overflow Clear

Operation: PC ← (PC) + 0002 + Rel if (V) = 0

Description: Tests the state of the V bit and causes a branch if the V bit is clear.

See BRA instruction for details of the execution of the branch.

Condition Codes: Not affected.

Addressing Formats:

See Table A-8.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
REL	4	2	28	050	040

Branch if Overflow Set **BVS**

Operation: PC ← (PC) + 0002 + Rel if (V) = 1

Description: Tests the state of the V bit and causes a branch if the V bit is set.

See BRA instruction for details of the execution of the branch.

Condition Codes: Not affected.

Addressing Formats:

See Table A-8.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
REL	4	2	29	051	041

CBA

Compare Accumulators

Operation: (ACCA) − (ACCB)

Description: Compares the contents of ACCA and the contents of ACCB and sets the condition codes, which may be used for arithmetic and logical conditional branches. Both operands are unaffected.

Condition Codes:

H: Not affected.

I: Not affected.

N: Set if the most significant bit of the result of the subtraction would be set; cleared otherwise.

Z: Set if all bits of the result of the subtraction would be cleared; cleared otherwise.

V: Set if the subtraction would cause two's complement overflow; cleared otherwise.

C: Set if the subtraction would require a borrow into the most significant bit of the result; clear otherwise.

Boolean Formulae for Condition Codes:

$N = R_7$

$Z = \overline{R}_7 \cdot \overline{R}_6 \cdot \overline{R}_5 \cdot \overline{R}_4 \cdot \overline{R}_3 \cdot \overline{R}_2 \cdot \overline{R}_1 \cdot \overline{R}_0$

$V = A_7 \cdot \overline{B}_7 \cdot \overline{R}_7 + \overline{A}_7 \cdot B_7 \cdot R_7$

$C = \overline{A}_7 \cdot B_7 + B_7 \cdot R_7 + R_7 \cdot \overline{A}_7$

Addressing Modes, Execution Time, and Machine Code (hexadecimal/octal/decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	2	1	11	021	017

CLC

Clear Carry

Operation:	C bit ← 0
Description:	Clears the carry bit in the processor condition codes register.
Condition Codes:	H: Not affected.
	I: Not affected.
	N: Not affected.
	Z: Not affected.
	V: Not affected.
	C: Cleared

Boolean Formulae for Condition Codes:

$C = 0$

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	**Execution Time (No. of cycles)**	**Number of bytes of machine code**	**Coding of First (or only) byte of machine code**		
			HEX.	**OCT.**	**DEC.**
INHERENT	2	1	0C	014	012

CLI

Clear Interrupt Mask

Operation: I bit ← 0

Description: Clears the interrupt mask bit in the processor condition codes register. This enables the microprocessor to service an interrupt from a peripheral device if signalled by a high state of the "Interrupt Request" control input.

Condition Codes:

H:	Not affected.
I:	Cleared.
N:	Not affected.
Z:	Not affected.
V:	Not affected.
C:	Not affected.

Boolean Formulae for Condition Codes:

$I = 0$

Addressing Modes, Execution Time, and Machine Code (hexadecimal/octal/decimal):

Addressing Modes	**Execution Time (No. of cycles)**	**Number of bytes of machine code**	**Coding of First (or only) byte of machine code**		
			HEX.	**OCT.**	**DEC.**
INHERENT	2	1	0E	016	014

Clear

CLR

Operation: ACCX ← 00
or: M ← 00

Description: The contents of ACCX or M are replaced with zeros.

Condition Codes:

H:	Not affected.
I:	Not affected.
N:	Cleared
Z:	Set
V:	Cleared
C:	Cleared

Boolean Formulae for Condition Codes:

$N = 0$
$Z = 1$
$V = 0$
$C = 0$

Addressing Formats:
See Table A-3.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A	2	1	4F	117	079
B	2	1	5F	137	095
EXT	6	3	7F	177	127
IND	7	2	6F	157	111

CLV

Clear Two's Complement Overflow Bit

Operation: V bit ← 0

Description: Clears the two's complement overflow bit in the processor condition codes register.

Condition Codes:

H:	Not affected.
I:	Not affected.
N:	Not affected.
Z:	Not affected.
V:	Cleared.
C:	Not affected.

Boolean Formulae for Condition Codes:

$V = 0$

Addressing Modes, Execution Time, and Machine Code (hexadecimal/octal/decimal):

Addressing Modes	**Execution Time (No. of cycles)**	**Number of bytes of machine code**	**Coding of First (or only) byte of machine code**		
			HEX.	**OCT.**	**DEC.**
INHERENT	2	1	0A	012	010

CMP

Compare

Operation: (ACCX) − (M)

Description: Compares the contents of ACCX and the contents of M and determines the condition codes, which may be used subsequently for controlling conditional branching. Both operands are unaffected.

Condition Codes:

- H: Not affected.
- I: Not affected.
- N: Set if the most significant bit of the result of the subtraction would be set; cleared otherwise.
- Z: Set if all bits of the result of the subtraction would be cleared; cleared otherwise.
- V: Set if the subtraction would cause two's complement overflow; cleared otherwise.
- C: Carry is set if the absolute value of the contents of memory is larger than the absolute value of the accumulator; reset otherwise.

Boolean Formulae for Condition Codes:

$$N = R_7$$
$$Z = \overline{R}_7 \cdot \overline{R}_6 \cdot \overline{R}_5 \cdot \overline{R}_4 \cdot \overline{R}_3 \cdot \overline{R}_2 \cdot \overline{R}_1 \cdot \overline{R}_0$$
$$V = X_7 \cdot \overline{M}_7 \cdot \overline{R}_7 + \overline{X}_7 \cdot M_7 \cdot R_7$$
$$C = \overline{X}_7 \cdot M_7 + M_7 \cdot R_7 + R_7 \cdot \overline{X}_7$$

Addressing Formats:

See Table A-1.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

(DUAL OPERAND)

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	**OCT.**	**DEC.**
A IMM	2	2	81	201	129
A DIR	3	2	91	221	145
A EXT	4	3	B1	261	177
A IND	5	2	A1	241	161
B IMM	2	2	C1	301	193
B DIR	3	2	D1	321	209
B EXT	4	3	F1	361	241
B IND	5	2	E1	341	225

COM

Complement

Operation: ACCX $\leftarrow \approx$ (ACCX) = FF − (ACCX)
or: M $\leftarrow \approx$ (M) = FF − (M)

Description: Replaces the contents of ACCX or M with its one's complement. (Each bit of the contents of ACCX or M is replaced with the complement of that bit.)

Condition Codes:
H: Not affected.
I: Not affected.
N: Set if most significant bit of the result is set; cleared otherwise.
Z: Set if all bits of the result are cleared; cleared otherwise.
V: Cleared.
C: Set.

Boolean Formulae for Condition Codes:

$N = R_7$
$Z = \overline{R}_7 \cdot \overline{R}_6 \cdot \overline{R}_5 \cdot \overline{R}_4 \cdot \overline{R}_3 \cdot \overline{R}_2 \cdot \overline{R}_1 \cdot \overline{R}_0$
$V = 0$
$C = 1$

Addressing Formats:
See Table A-3.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A	2	1	43	103	067
B	2	1	53	123	083
EXT	6	3	73	163	115
IND	7	2	63	143	099

CPX

Compare Index Register

Operation: (IXL) − (M+1)
(IXH) − (M)

Description: The more significant byte of the contents of the index register is compared with the contents of the byte of memory at the address specified by the program. The less significant byte of the contents of the index register is compared with the contents of the next byte of memory, at one plus the address specified by the program. The Z bit is set or reset according to the results of these comparisons, and may be used subsequently for conditional branching.

The N and V bits, though determined by this operation, are not intended for conditional branching.

The C bit is not affected by this operation.

Condition Codes:

- H: Not affected.
- I: Not affected.
- N: Set if the most significant bit of the result of the subtraction from the more significant byte of the index register would be set; cleared otherwise.
- Z: Set if all bits of the results of both subtractions would be cleared; cleared otherwise.
- V: Set if the subtraction from the more significant byte of the index register would cause two's complement overflow; cleared otherwise.
- C: Not affected.

Boolean Formulae for Condition Codes:

$$N = RH_7$$
$$Z = (\overline{RH_7} \cdot \overline{RH_6} \cdot \overline{RH_5} \cdot \overline{RH_4} \cdot \overline{RH_3} \cdot \overline{RH_2} \cdot \overline{RH_1} \cdot \overline{RH_0}) \cdot (\overline{RL_7} \cdot \overline{RL_6} \cdot \overline{RL_5} \cdot \overline{RL_4} \cdot \overline{RL_3} \cdot \overline{RL_2} \cdot \overline{RL_1} \cdot \overline{RL_0})$$
$$V = IXH_7 \cdot \overline{M_7} \cdot \overline{RH_7} + \overline{IXH_7} \cdot M_7 \cdot RH_7$$

Addressing Formats:
See Table A-5.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/octal/decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
IMM	3	3	8C	214	140
DIR	4	2	9C	234	156
EXT	5	3	BC	274	188
IND	6	2	AC	254	172

DAA

Decimal Adjust ACCA

Operation: Adds hexadecimal numbers 00, 06, 60, or 66 to ACCA, and may also set the carry bit, as indicated in the following table:

State of C-bit before DAA (Col. 1)	Upper Half-byte (bits 4-7) (Col. 2)	Initial Half-carry H-bit (Col.3)	Lower to ACCA (bits 0-3) (Col. 4)	Number Added after by DAA (Col. 5)	State of C-bit DAA (Col. 6)
0	0-9	0	0-9	00	0
0	0-8	0	A-F	06	0
0	0-9	1	0-3	06	0
0	A-F	0	0-9	60	1
0	9-F	0	A-F	66	1
0	A-F	1	0-3	66	1
1	0-2	0	0-9	60	1
1	0-2	0	A-F	66	1
1	0-3	1	0-3	66	1

Note: Columns (1) through (4) of the above table represent all possible cases which can result from any of the operations ABA, ADD, or ADC, with initial carry either set or clear, applied to two binary-coded-decimal operands. The table shows hexadecimal values.

Description: If the contents of ACCA and the state of the carry-borrow bit C and the half-carry bit H are all the result of applying any of the operations ABA, ADD, or ADC to binary-coded-decimal operands, with or without an initial carry, the DAA operation will function as follows.

Subject to the above condition, the DAA operation will adjust the contents of ACCA and the C bit to represent the correct binary-coded-decimal sum and the correct state of the carry.

Condition Codes:

H:	Not affected.
I:	Not affected.
N:	Set if most significant bit of the result is set; cleared otherwise.
Z:	Set if all bits of the result are cleared; cleared otherwise.
V:	Not defined.
C:	Set or reset according to the same rule as if the DAA and an immediately preceding ABA, ADD, or ADC were replaced by a hypothetical binary-coded-decimal addition.

Boolean Formulae for Condition Codes:

$N = R_7$

$Z = \overline{R}_7 \cdot \overline{R}_6 \cdot \overline{R}_5 \cdot \overline{R}_4 \cdot \overline{R}_3 \cdot \overline{R}_2 \cdot \overline{R}_1 \cdot \overline{R}_0$

C = See table above.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	2	1	19	031	025

DEC

Decrement

Operation: ACCX ← (ACCX) − 01

or: M ← (M) − 01

Description: Subtract one from the contents of ACCX or M.

The N, Z, and V condition codes are set or reset according to the results of this operation.

The C bit is not affected by the operation.

Condition Codes:

H: Not affected.
I: Not affected.
N: Set if most significant bit of the result is set; cleared otherwise.
Z: Set if all bits of the result are cleared; cleared otherwise.
V: Set if there was two's complement overflow as a result of the operation; cleared otherwise. Two's complement overflow occurs if and only if (ACCX) or (M) was 80 before the operation.
C: Not affected.

Boolean Formulae for Condition Codes:

$N = R_7$

$Z = \overline{R}_7 \cdot \overline{R}_6 \cdot \overline{R}_5 \cdot \overline{R}_5 \cdot \overline{R}_4 \cdot \overline{R}_3 \cdot \overline{R}_2 \cdot \overline{R}_1 \cdot \overline{R}_0$

$V = X_7 \cdot \overline{X}_6 \cdot \overline{X}_5 \cdot \overline{X}_4 \cdot \overline{X}_3 \cdot \overline{X}_2 \cdot \overline{X}_0 = \overline{R}_7 \cdot R_6 \cdot R_5 \cdot R_4 \cdot R_3 \cdot R_2 \cdot R_1 \cdot R_0$

Addressing Formats:

See Table A-3.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A	2	1	4A	112	074
B	2	1	5A	132	090
EXT	6	3	7A	172	122
IND	7	2	6A	152	106

Decrement Stack Pointer

DES

Operation: SP ← (SP) − 0001

Description: Subtract one from the stack pointer.

Condition Codes: Not affected.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	4	1	34	064	052

DEX

Decrement Index Register

Operation: $IX \leftarrow (IX) - 0001$

Description: Subtract one from the index register.

Only the Z bit is set or reset according to the result of this operation.

Condition Codes:

H:	Not affected.
I:	Not affected.
N:	Not affected.
Z:	Set if all bits of the result are cleared; cleared otherwise.
V:	Not affected.
C:	Not affected.

Boolean Formulae for Condition Codes:

$$Z = (\overline{RH_7} \cdot \overline{RH_6} \cdot \overline{RH_5} \cdot \overline{RH_4} \cdot \overline{RH_3} \cdot \overline{RH_2} \cdot \overline{RH_1} \cdot \overline{RH_0})$$
$$(\overline{RL_7} \cdot \overline{RL_6} \cdot \overline{RL_5} \cdot \overline{RL_4} \cdot \overline{RL_3} \cdot \overline{RL_2} \cdot \overline{RL_1} \cdot \overline{RL_0})$$

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	4	1	09	011	009

Exclusive OR

EOR

Operation: ACCX ← (ACCX) ⊕ (M)

Description: Perform logical "EXCLUSIVE OR" between the contents of ACCX and the contents of M, and place the result in ACCX. (Each bit of ACCX after the operation will be the logical "EXCLUSIVE OR" of the corresponding bit of M and ACCX before the operation.)

Condition Codes:
H: Not affected.
I: Not affected.
N: Set if most significant bit of the result is set; cleared otherwise.
Z: Set if all bits of the result are cleared; cleared otherwise.
V: Cleared
C: Not affected.

Boolean Formulae for Condition Codes:

$N = R_7$

$Z = \overline{R_7}\cdot\overline{R_6}\cdot\overline{R_5}\cdot\overline{R_4}\cdot\overline{R_3}\cdot\overline{R_2}\cdot\overline{R_1}\cdot\overline{R_0}$

$V = 0$

Addressing Formats:
See Table A-1.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A IMM	2	2	88	210	136
A DIR	3	2	98	230	152
A EXT	4	3	B8	270	184
A IND	5	2	A8	250	168
B IMM	2	2	C8	310	200
B DIR	3	2	D8	330	216
B EXT	4	3	F8	370	248
B IND	5	2	E8	350	232

INC

Increment

Operation: ACCX ← (ACCX) + 01

or: M ← (M) + 01

Description: Add one to the contents of ACCX or M.

The N, Z, and V condition codes are set or reset according to the results of this operation.

The C bit is not affected by the operation.

Condition Codes:

H: Not affected.
I: Not affected.
N: Set if most significant bit of the result is set; cleared otherwise.
Z: Set if all bits of the result are cleared; cleared otherwise.
V: Set if there was two's complement overflow as a result of the operation; cleared otherwise. Two's complement overflow will occur if and only if (ACCX) or (M) was 7F before the operation.
C: Not affected.

Boolean Formulae for Condition Codes:

$N = R_7$

$Z = \overline{R}_7 \cdot \overline{R}_6 \cdot \overline{R}_5 \cdot \overline{R}_4 \cdot \overline{R}_3 \cdot \overline{R}_2 \cdot \overline{R}_1 \cdot \overline{R}_0$

$V = \overline{X}_7 \cdot X_6 \cdot X_5 \cdot X_4 \cdot X_3 \cdot X_2 \cdot X_1 \cdot X_0$

$C = \overline{R}_7 \cdot \overline{R}_6 \cdot \overline{R}_5 \cdot \overline{R}_4 \cdot \overline{R}_3 \cdot \overline{R}_2 \cdot \overline{R}_1 \cdot \overline{R}_0$

Addressing Formats:

See Table A-3.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A	2	1	4C	114	076
B	2	1	5C	134	092
EXT	6	3	7C	174	124
IND	7	2	6C	154	108

Increment Stack Pointer **INS**

Operation: SP ← (SP) + 0001

Description: Add one to the stack pointer.

Condition Codes: Not affected.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	**Execution Time (No. of cycles)**	**Number of bytes of machine code**	**Coding of First (or only) byte of machine code**		
			HEX.	**OCT.**	**DEC.**
INHERENT	4	1	31	061	049

INX

Increment Index Register

Operation: IX ← (IX) + 0001

Description: Add one to the index register.

Only the Z bit is set or reset according to the result of this operation.

Condition Codes:

H: Not affected.
I: Not affected.
N: Not affected.
Z: Set if all 16 bits of the result are cleared; cleared otherwise.
V: Not affected.
C: Not affected.

Boolean Formulae for Condition Codes:

$$Z = (\overline{RH_7}\cdot\overline{RH_6}\cdot\overline{RH_5}\cdot\overline{RH_4}\cdot\overline{RH_3}\cdot\overline{RH_2}\cdot\overline{RH_1}\cdot\overline{RH_0})\cdot(\overline{RL_7}\cdot\overline{RL_6}\cdot\overline{RL_5}\cdot\overline{RL_4}\cdot\overline{RL_3}\cdot\overline{RL_2}\cdot\overline{RL_1}\cdot\overline{RL_0})$$

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	**Execution Time (No. of cycles)**	**Number of bytes of machine code**	**Coding of First (or only) byte of machine code**		
			HEX.	**OCT.**	**DEC.**
INHERENT	4	1	08	010	008

Jump JMP

Operation: PC ← numerical address

Description: A jump occurs to the instruction stored at the numerical address. The numerical address is obtained according to the rules for EXTended or INDexed addressing.

Condition Codes: Not affected.

Addressing Formats:

See Table A-7.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	**Execution Time (No. of cycles)**	**Number of bytes of machine code**	**Coding of First (or only) byte of machine code**		
			HEX.	**OCT.**	**DEC.**
EXT	3	3	7E	176	126
IND	4	2	6E	156	110

JSR

Jump to Subroutine

Operation:

Either:	PC ← (PC) + 0003 (for EXTended addressing)
or:	PC ← (PC) + 0002 (for INDexed addressing)
Then:	↓ (PCL)
	SP ← (SP) − 0001
	↓ (PCH)
	SP ← (SP) − 0001
	PC ← numerical address

Description: The program counter is incremented by 3 or by 2, depending on the addressing mode, and is then pushed onto the stack, eight bits at a time. The stack pointer points to the next empty location in the stack. A jump occurs to the instruction stored at the numerical address. The numerical address is obtained according to the rules for EXTended or INDexed addressing.

Condition Codes: Not affected.

Addressing Formats:

See Table A-7.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
EXT	9	3	BD	275	189
IND	8	2	AD	255	173

JUMP TO SUBROUTINE EXAMPLE (extended mode)

			Memory Location	Machine Code (Hex)	Label	Operator	Operand
A.	*Before:*						
	PC	→	$0FFF	BD		JSR	CHARLI
			$1000	20			
			$1001	77			
	SP	←	$EFFF				
B.	*After:*						
	PC	→	$2077	**	CHARLI	***	*****
	SP	→	$EFFD				
			$EFFE	10			
			$EFFF	02			

Load Accumulator LDA

Operation: ACCX ← (M)

Description: Loads the contents of memory into the accumulator. The condition codes are set according to the data.

Condition Codes:
H: Not affected.
I: Not affected.
N: Set if most significant bit of the result is set; cleared otherwise.
Z: Set if all bits of the result are cleared; cleared otherwise.
V: Cleared.
C: Not affected.

Boolean Formulae for Condition Codes:

$N = R_7$

$Z = \overline{R_7} \cdot \overline{R_6} \cdot \overline{R_5} \cdot \overline{R_4} \cdot \overline{R_3} \cdot \overline{R_2} \cdot \overline{R_1} \cdot \overline{R_0}$

$V = 0$

Addressing Formats:

See Table A-1.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):
(DUAL OPERAND)

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A IMM	2	2	86	206	134
A DIR	3	2	96	226	150
A EXT	4	3	B6	266	182
A IND	5	2	A6	246	166
B IMM	2	2	C6	306	198
B DIR	3	2	D6	326	214
B EXT	4	3	F6	366	246
B IND	5	2	E6	346	230

LDS

Load Stack Pointer

Operation: SPH ← (M)
SPL ← (M+1)

Description: Loads the more significant byte of the stack pointer from the byte of memory at the address specified by the program, and loads the less significant byte of the stack pointer from the next byte of memory, at one plus the address specified by the program.

Condition Codes:
H: Not affected.
I: Not affected.
N: Set if the most significant bit of the stack pointer is set by the operation; cleared otherwise.
Z: Set if all bits of the stack pointer are cleared by the operation; cleared otherwise.
V: Cleared.
C: Not affected.

Boolean Formulae for Condition Codes:

$N = RH_7$

$Z = (\overline{RH_7} \cdot \overline{RH_6} \cdot \overline{RH_5} \cdot \overline{RH_4} \cdot \overline{RH_3} \cdot \overline{RH_2} \cdot \overline{RH_1} \cdot \overline{RH_0}) \cdot (\overline{RL_7} \cdot \overline{RL_6} \cdot \overline{RL_5} \cdot \overline{RL_4} \cdot \overline{RL_3} \cdot \overline{RL_2} \cdot \overline{RL_1} \cdot \overline{RL_0})$

$V = 0$

Addressing Formats:

See Table A-5.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/octal/decimal):

Addressing Modes	**Execution Time (No. of cycles)**	**Number of bytes of machine code**	**Coding of First (or only) byte of machine code**		
			HEX.	**OCT.**	**DEC.**
IMM	3	3	8E	216	142
DIR	4	2	9E	236	158
EXT	5	3	BE	276	190
IND	6	2	AE	256	174

Load Index Register **LDX**

Operation: $IXH \leftarrow (M)$
$IXL \leftarrow (M+1)$

Description: Loads the more significant byte of the index register from the byte of memory at the address specified by the program, and loads the less significant byte of the index register from the next byte of memory, at one plus the address specified by the program.

Condition Codes:
- H: Not affected.
- I: Not affected.
- N: Set if the most significant bit of the index register is set by the operation; cleared otherwise.
- Z: Set if all bits of the index register are cleared by the operation; cleared otherwise.
- V: Cleared.
- C: Not affected.

Boolean Formulae for Condition Codes:

$N = RH_7$

$Z = (\overline{RH_7} \cdot \overline{RH_6} \cdot \overline{RH_5} \cdot \overline{RH_4} \cdot \overline{RH_3} \cdot \overline{RH_2} \cdot \overline{RH_1} \cdot \overline{RH_0}) \cdot (\overline{RL_7} \cdot \overline{RL_6} \cdot \overline{RL_5} \cdot \overline{RL_4} \cdot \overline{RL_3} \cdot \overline{RL_2} \cdot \overline{RL_1} \cdot \overline{RL_0})$

$V = 0$

Addressing Formats:

See Table A-5.

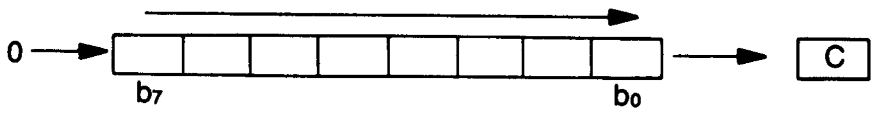
Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
IMM	3	3	CE	316	206
DIR	4	2	DE	336	222
EXT	5	3	FE	376	254
IND	6	2	EE	356	238

LSR

Logical Shift Right

Operation:



Description: Shifts all bits of ACCX or M one place to the right. Bit 7 is loaded with a zero. The C bit is loaded from the least significant bit of ACCX or M.

Condition Codes:
H: Not affected.
I: Not affected.
N: Cleared.
Z: Set if all bits of the result are cleared; cleared otherwise.
V: Set if, after the completion of the shift operation, EITHER (N is set and C is cleared) OR (N is cleared and C is set); cleared otherwise.
C: Set if, before the operation, the least significant bit of the ACCX or M was set; cleared otherwise.

Boolean Formulae for Condition Codes:

$N = 0$

$Z = \overline{R_7} \cdot \overline{R_6} \cdot \overline{R_5} \cdot \overline{R_4} \cdot \overline{R_3} \cdot \overline{R_2} \cdot \overline{R_1} \cdot \overline{R_0}$

$V = N \oplus C = [N \cdot \overline{C}] \odot [\overline{N} \cdot C]$

(the foregoing formula assumes values of N and C after the shift operation).

$C = M_0$

Addressing Formats:

See Table A-3.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A	2	1	44	104	068
B	2	1	54	124	084
EXT	6	3	74	164	116
IND	7	2	64	144	100

Negate NEG

Operation: ACCX ← − (ACCX) = 00 − (ACCX)
or: M ← − (M) = 00 − (M)

Description: Replaces the contents of ACCX or M with its two's complement. Note that 80 is left unchanged.

Condition Codes:
- H: Not affected.
- I: Not affected.
- N: Set if most significant bit of the result is set; cleared otherwise.
- Z: Set if all bits of the result are cleared; cleared otherwise.
- V: Set if there would be two's complement overflow as a result of the implied subtraction from zero; this will occur if and only if the contents of ACCX or M is 80.
- C: Set if there would be a borrow in the implied subtraction from zero; the C bit will be set in all cases except when the contents of ACCX or M is 00.

Boolean Formulae for Condition Codes:

$$N = R_7$$
$$Z = \overline{R_7}\cdot\overline{R_6}\cdot\overline{R_5}\cdot\overline{R_4}\cdot\overline{R_3}\cdot\overline{R_2}\cdot\overline{R_1}\cdot\overline{R_0}$$
$$V = R_7\cdot\overline{R_6}\cdot\overline{R_5}\cdot\overline{R_4}\cdot\overline{R_3}\cdot\overline{R_2}\cdot\overline{R_1}\cdot\overline{R_0}$$
$$C = R_7+R_6+R_5+R_4+R_3+R_2+R_1+R_0$$

Addressing Formats:

See Table A-3.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A	2	1	40	100	064
B	2	1	50	120	080
EXT	6	3	70	160	112
IND	7	2	60	140	096

NOP

No Operation

Description: This is a single-word instruction which causes only the program counter to be incremented. No other registers are affected.

Condition Codes: Not affected.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	2	1	01	001	001

Inclusive OR **ORA**

Operation: ACCX ← (ACCX)$\odot$(M)

Description: Perform logical "OR" between the contents of ACCX and the contents of M and places the result in ACCX. (Each bit of ACCX after the operation will be the logical "OR" of the corresponding bits of M and of ACCX before the operation).

Condition Codes:

H:	Not affected.
I:	Not affected.
N:	Set if most significant bit of the result is set; cleared otherwise.
Z:	Set if all bits of the result are cleared; cleared otherwise.
V:	Cleared.
C:	Not affected.

Boolean Formulae for Condition Codes:

$N = R_7$

$Z = \overline{R}_7 \cdot \overline{R}_6 \cdot \overline{R}_5 \cdot \overline{R}_4 \cdot \overline{R}_3 \cdot \overline{R}_2 \cdot \overline{R}_1 \cdot \overline{R}_0$

$V = 0$

Addressing Formats:

See Table A-1.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/octal/decimal):

(DUAL OPERAND)

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A IMM	2	2	8A	212	138
A DIR	3	2	9A	232	154
A EXT	4	3	BA	272	186
A IND	5	2	AA	252	170
B IMM	2	2	CA	312	202
B DIR	3	2	DA	332	218
B EXT	4	3	FA	372	250
B IND	5	2	EA	352	234

PSH

Push Data Onto Stack

Operation: ↓ (ACCX)
SP ← (SP) − 0001

Description: The contents of ACCX is stored in the stack at the address contained in the stack pointer. The stack pointer is then decremented.

Condition Codes: Not affected.

Addressing Formats:

See Table A-4.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A	4	1	36	066	054
B	4	1	37	067	055

Pull Data from Stack

PUL

Operation: SP ← (SP) + 0001
↑ ACCX

Description: The stack pointer is incremented. The ACCX is then loaded from the stack, from the address which is contained in the stack pointer.

Condition Codes: Not affected.

Addressing Formats:

See Table A-4.

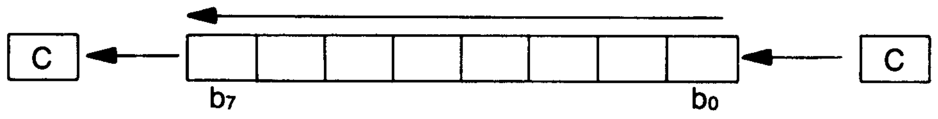
Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A	4	1	32	062	050
B	4	1	33	063	051

ROL

Rotate Left

Operation:



Description: Shifts all bits of ACCX or M one place to the left. Bit 0 is loaded from the C bit. The C bit is loaded from the most significant bit of ACCX or M.

Condition Codes:

H:	Not affected.
I:	Not affected.
N:	Set if most significant bit of the result is set; cleared otherwise.
Z:	Set if all bits of the result are cleared; cleared otherwise.
V:	Set if, after the completion of the operation, EITHER (N is set and C is cleared) OR (N is cleared and C is set); cleared otherwise.
C:	Set if, before the operation, the most significant bit of the ACCX or M was set; cleared otherwise.

Boolean Formulae for Condition Codes:

$N = R_7$

$Z = \overline{R_7} \cdot \overline{R_6} \cdot \overline{R_5} \cdot \overline{R_4} \cdot \overline{R_3} \cdot \overline{R_2} \cdot \overline{R_1} \cdot \overline{R_0}$

$V = N \oplus C = [N \cdot \overline{C}] \odot [\overline{N} \cdot C]$

(the foregoing formula assumes values of N and C after the rotation)

$C = M_7$

Addressing Formats:

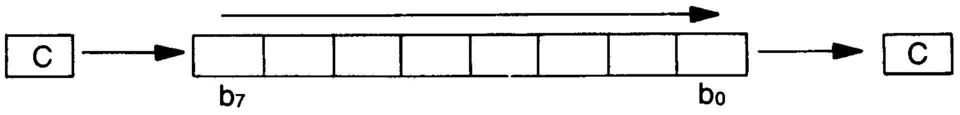
See Table A-3

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	**OCT.**	**DEC.**
A	2	1	49	111	073
B	2	1	59	131	089
EXT	6	3	79	171	121
IND	7	2	69	151	105

Rotate Right **ROR**

Operation:



Description: Shifts all bits of ACCX or M one place to the right. Bit 7 is loaded from the C bit. The C bit is loaded from the least significant bit of ACCX or M.

Condition Codes:
- H: Not affected.
- I: Not affected.
- N: Set if most significant bit of the result is set; cleared otherwise.
- Z: Set if all bits of the result are cleared; cleared otherwise.
- V: Set if, after the completion of the operation, EITHER (N is set and C is cleared) OR (N is cleared and C is set); cleared otherwise.
- C: Set if, before the operation, the least significant bit of the ACCX or M was set; cleared otherwise.

Boolean Formulae for Condition Codes:

$N = R_7$

$Z = \overline{R_7} \cdot \overline{R_6} \cdot \overline{R_5} \cdot \overline{R_4} \cdot \overline{R_3} \cdot \overline{R_2} \cdot \overline{R_1} \cdot \overline{R_0}$

$V = N \oplus C = [N \cdot \overline{C}] \odot [\overline{N} \cdot C]$

(the foregoing formula assumes values of N and C after the rotation)

$C = M_0$

Addressing Formats:

See Table A-3

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A	2	1	46	106	070
B	2	1	56	126	086
EXT	6	3	76	166	118
IND	7	2	66	146	102

RTI

Return from Interrupt

Operation:

$SP \leftarrow (SP) + 0001,\ \uparrow CC$
$SP \leftarrow (SP) + 0001,\ \uparrow ACCB$
$SP \leftarrow (SP) + 0001,\ \uparrow ACCA$
$SP \leftarrow (SP) + 0001,\ \uparrow IXH$
$SP \leftarrow (SP) + 0001,\ \uparrow IXL$
$SP \leftarrow (SP) + 0001,\ \uparrow PCH$
$SP \leftarrow (SP) + 0001,\ \uparrow PCL$

Description: The condition codes, accumulators B and A, the index register, and the program counter, will be restored to a state pulled from the stack. Note that the interrupt mask bit will be reset if and only if the corresponding bit stored in the stack is zero.

Condition Codes: Restored to the states pulled from the stack.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/octal/decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	10	1	3B	073	059

Return from Interrupt

Example

		Memory Location	Machine Code (Hex)	Label	Operator	Operand
A.	*Before*					
	PC →	$D066	3B		RTI	
	SP →	$EFF8				
		$EFF9	11HINZVC	(binary)		
		$EFFA	12			
		$EFFB	34			
		$EFFC	56			
		$EFFD	78			
		$EFFE	55			
		$EFFF	67			
B.	*After*					
	PC →	$5567	**		***	*****
		$EFF8				
		$EFF9	11HINZVC	(binary)		
		$EFFA	12			
		$EFFB	34			
		$EFFC	56			
		$EFFD	78			
		$EFFE	55			
	SP →	$EFFF	67			

CC = HINZVC (binary)

ACCB = 12 (Hex) IXH = 56 (Hex)

ACCA = 34 (Hex) IXL = 78 (Hex)

Return from Subroutine RTS

Operation: SP ← (SP) + 0001
↑ PCH
SP ← (SP) + 0001
↑ PCL

Description: The stack pointer is incremented (by 1). The contents of the byte of memory, at the address now contained in the stack pointer, are loaded into the 8 bits of highest significance in the program counter. The stack pointer is again incremented (by 1). The contents of the byte of memory, at the address now contained in the stack pointer, are loaded into the 8 bits of lowest significiance in the program counter.

Condition Codes: Not affected.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	5	1	39	071	057

Return from Subroutine

EXAMPLE

		Memory Location	Machine Code (Hex)	Label	Assembler Language Operator	Operand
A.	*Before*					
	PC	$30A2	39		RTS	
	SP	$EFFD				
		$EFFE	10			
		$EFFF	02			
B.	*After*					
	PC	$1002	**		***	*****
		$EFFD				
		$EFFE	10			
	SP	$EFFF	02			

SBA

Subtract Accumulators

Operation: ACCA ← (ACCA) − (ACCB)

Description: Subtracts the contents of ACCB from the contents of ACCA and places the result in ACCA. The contents of ACCB are not affected.

Condition Codes:
- H: Not affected.
- I: Not affected.
- N: Set if most significant bit of the result is set; cleared otherwise.
- Z: Set if all bits of the result are cleared; cleared otherwise.
- V: Set if there was two's complement overflow as a result of the operation.
- C: Carry is set if the absolute value of accumulator B plus previous carry is larger than the absolute value of accumulator A; reset otherwise.

Boolean Formulae for Condition Codes:

$N = R_7$

$Z = \overline{R_7}\cdot\overline{R_6}\cdot\overline{R_5}\cdot\overline{R_4}\cdot\overline{R_3}\cdot\overline{R_2}\cdot\overline{R_1}\cdot\overline{R_0}$

$V = A_7\cdot\overline{B_7}\cdot\overline{R_7}+\overline{A_7}\cdot B_7\cdot R_7$

$C = \overline{A_7}\cdot B_7+B_7\cdot R_7+R_7\cdot\overline{A_7}$

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	2	1	10	020	016

Subtract with Carry SBC

Operation: ACCX ← (ACCX) − (M) − (C)

Description: Subtracts the contents of M and C from the contents of ACCX and places the result in ACCX.

Condition Codes:
- H: Not affected.
- I: Not affected.
- N: Set if most significant bit of the result is set; cleared otherwise.
- Z: Set if all bits of the result are cleared; cleared otherwise.
- V: Set if there was two's complement overflow as a result of the operation; cleared otherwise.
- C: Carry is set if the absolute value of the contents of memory plus previous carry is larger than the absolute value of the accumulator; reset otherwise.

Boolean Formulae for Condition Codes:

$$N = R_7$$
$$Z = \overline{R_7} \cdot \overline{R_6} \cdot \overline{R_5} \cdot \overline{R_4} \cdot \overline{R_3} \cdot \overline{R_2} \cdot \overline{R_1} \cdot \overline{R_0}$$
$$V = X_7 \cdot \overline{M_7} \cdot \overline{R_7} + \overline{X_7} \cdot M_7 \cdot R_7$$
$$C = \overline{X_7} \cdot M_7 + M_7 \cdot R_7 + R_7 \cdot \overline{X_7}$$

Addressing Formats:

See Table A-1.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

(DUAL OPERAND)

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A IMM	2	2	82	202	130
A DIR	3	2	92	222	146
A EXT	4	3	B2	262	178
A IND	5	2	A2	242	162
B IMM	2	2	C2	302	194
B DIR	3	2	D2	322	210
B EXT	4	3	F2	362	242
B IND	5	2	E2	342	226

SEC

Set Carry

Operation: C bit ← 1

Description: Sets the carry bit in the processor condition codes register.

Condition Codes:

H:	Not affected.
I:	Not affected.
N:	Not affected.
Z:	Not affected.
V:	Not affected.
C:	Set.

Boolean Formulae for Condition Codes:

$C = 1$

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	2	1	0D	015	013

Set Interrupt Mask **SEI**

Operation:	I bit ← 1
Description:	Sets the interrupt mask bit in the processor condition codes register. The microprocessor is inhibited from servicing an interrupt from a peripheral device, and will continue with execution of the instructions of the program, until the interrupt mask bit has been cleared.

Condition Codes:

H:	Not affected.
I:	Set.
N:	Not affected.
Z:	Not affected.
V:	Not affected.
C:	Not affected.

Boolean Formulae for Condition Codes:

I = 1

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	2	1	0F	017	015

SEV

Set Two's Complement Overflow Bit

Operation: V bit ← 1

Description: Sets the two's complement overflow bit in the processor condition codes register.

Condition Codes:

H:	Not affected.
I:	Not affected.
N:	Not affected.
Z:	Not affected.
V:	Set.
C:	Not affected.

Boolean Formulae for Condition Codes:

V = 1

Addressing Modes, Execution Time, and Machine Code (hexadecimal/octal/decimal):

Addressing Modes	**Execution Time (No. of cycles)**	**Number of bytes of machine code**	**Coding of First (or only) byte of machine code**		
			HEX.	**OCT.**	**DEC.**
INHERENT	2	1	0B	013	011

Store Accumulator **STA**

Operation: M ← (ACCX)

Description: Stores the contents of ACCX in memory. The contents of ACCX remains unchanged.

Condition Codes:
- H: Not affected.
- I: Not affected.
- N: Set if the most significant bit of the contents of ACCX is set; cleared otherwise.
- Z: Set if all bits of the contents of ACCX are cleared; cleared otherwise.
- V: Cleared.
- C: Not affected.

Boolean Formulae for Condition Codes:

$N = X_7$

$Z = \overline{X_7} \cdot \overline{X_6} \cdot \overline{X_5} \cdot \overline{X_4} \cdot \overline{X_3} \cdot \overline{X_2} \cdot \overline{X_1} \cdot \overline{X_0}$

$V = 0$

Addressing Formats:

See Table A-2.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/octal/decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A DIR	4	2	97	227	151
A EXT	5	3	B7	267	183
A IND	6	2	A7	247	167
B DIR	4	2	D7	327	215
B EXT	5	3	F7	367	247
B IND	6	2	E7	347	231

STS

Store Stack Pointer

Operation: $M \leftarrow (SPH)$
$M + 1 \leftarrow (SPL)$

Description: Stores the more significant byte of the stack pointer in memory at the address specified by the program, and stores the less significant byte of the stack pointer at the next location in memory, at one plus the address specified by the program.

Condition Codes:
- H: Not affected.
- I: Not affected.
- N: Set if the most significant bit of the stack pointer is set; cleared otherwise.
- Z: Set if all bits of the stack pointer are cleared; cleared otherwise.
- V: Cleared.
- C: Not affected.

Boolean Formulae for Condition Codes:

$$N = SPH_7$$

$$Z = (\overline{SPH_7} \cdot \overline{SPH_6} \cdot \overline{SPH_5} \cdot \overline{SPH_4} \cdot \overline{SPH_3} \cdot \overline{SPH_2} \cdot \overline{SPH_1} \cdot \overline{SPH_0}) \cdot (\overline{SPL_7} \cdot \overline{SPL_6} \cdot \overline{SPL_5} \cdot \overline{SPL_4} \cdot \overline{SPL_3} \cdot \overline{SPL_2} \cdot \overline{SPL_1} \cdot \overline{SPL_0})$$

$$V = 0$$

Addressing Formats:
See Table A-6.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
DIR	5	2	9F	237	159
EXT	6	3	BF	277	191
IND	7	2	AF	257	175

STX

Store Index Register

Operation: M ← (IXH)
M + 1 ← (IXL)

Description: Stores the more significant byte of the index register in memory at the address specified by the program, and stores the less significant byte of the index register at the next location in memory, at one plus the address specified by the program.

Condition Codes:
H: Not affected.
I: Not affected.
N: Set if the most significant bité of the index register is set; cleared otherwise.
Z: Set if all bits of the index register are cleared; cleared otherwise.
V: Cleared.
C: Not affected.

Boolean Formulae for Condition Codes:

$N = IXH_7$

$Z = (\overline{IXH_7} \cdot \overline{IXH_6} \cdot \overline{IXH_5} \cdot \overline{IXH_4} \cdot \overline{IXH_3} \cdot \overline{IXH_2} \cdot \overline{IXH_1} \cdot \overline{IXH_0}) \cdot (\overline{IXL_7} \cdot \overline{IXL_6} \cdot \overline{IXL_5} \cdot \overline{IXL_4} \cdot \overline{IXL_3} \cdot \overline{IXL_2} \cdot \overline{IXL_1} \cdot \overline{IXL_0})$

$V = 0$

Addressing Formats:

See Table A-6.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
DIR	5	2	DF	337	223
EXT	6	3	FF	377	255
IND	7	2	EF	357	239

SUB

Subtract

Operation: ACCX ← (ACCX) − (M)

Description: Subtracts the contents of M from the contents of ACCX and places the result in ACCX.

Condition Codes:
- H: Not affected.
- I: Not affected.
- N: Set if most significant bit of the result is set; cleared otherwise.
- Z: Set if all bits of the result are cleared; cleared otherwise.
- V: Set if there was two's complement overflow as a result of the operation; cleared otherwise.
- C: Set if the absolute value of the contents of memory are larger than the absolute value of the accumulator; reset otherwise.

Boolean Formulae for Condition Codes:

$$N = R_7$$
$$Z = \overline{R}_7 \bullet \overline{R}_6 \bullet \overline{R}_5 \bullet \overline{R}_4 \bullet \overline{R}_3 \bullet \overline{R}_2 \bullet \overline{R}_1 \bullet \overline{R}_0$$
$$V = X_7 \bullet \overline{M}_7 \bullet \overline{R}_7 + \overline{X}_7 \bullet M_7 \bullet R_7$$
$$C = \overline{X}_7 \bullet M_7 + M_7 \bullet R_7 + R_7 \bullet \overline{X}_7$$

Addressing Formats:

See Table A-1.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/octal/decimal):
(DUAL OPERAND)

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A IMM	2	2	80	200	128
A DIR	3	2	90	220	144
A EXT	4	3	B0	260	176
A IND	5	2	A0	240	160
B IMM	2	2	C0	300	192
B DIR	3	2	D0	320	208
B EXT	4	3	F0	360	240
B IND	5	2	E0	340	224

Software Interrupt SWI

Operation: PC ← (PC) + 0001
↓ (PCL) , SP ← (SP)-0001
↓ (PCH) , SP ← (SP)-0001
↓ (IXL) , SP ← (SP)-0001
↓ (IXH) , SP ← (SP)-0001
↓ (ACCA) , SP ← (SP)-0001
↓ (ACCB) , SP ← (SP)-0001
↓ (CC) , SP ← (SP)-0001
I ← 1
PCH ← (n-0005)
PCL ← (n-0004)

Description: The program counter is incremented (by 1). The program counter, index register, and accumulator A and B, are pushed into the stack. The condition codes register is then pushed into the stack, with condition codes H, I, N, Z, V, C going respectively into bit positions 5 thru 0, and the top two bits (in bit positions 7 and 6) are set (to the 1 state). The stack pointer is decremented (by 1) after each byte of data is stored in the stack.

The interrupt mask bit is then set. The program counter is then loaded with the address stored in the software interrupt pointer at memory locations (n-5) and (n-4), where n is the address corresponding to a high state on all lines of the address bus.

Condition Codes:

H:	Not affected.
I:	Set.
N:	Not affected.
Z:	Not affected.
V:	Not affected.
C:	Not affected.

Boolean Formula for Condition Codes:
I = 1

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	12	1	3F	077	063

Software Interrupt

EXAMPLE

A. *Before:*

CC = HINZVC (binary)

ACCB = 12 (Hex) IXH = 56 (Hex)

ACCA = 34 (Hex) IXL = 78 (Hex)

		Memory Location	Machine Code (Hex)	Assembler Language Label	Operator	Operand
PC	→	$5566	3F		SWI	
SP	→	$EFFF				
		$FFFA	D0			
		$FFFB	55			

B. *After:*

PC	→	$D055		
SP	→	$EFF8		
		$EFF9	11HINZVC	(binary)
		$EFFA	12	
		$EFFB	34	
		$EFFC	56	
		$EFFD	78	
		$EFFE	55	
		$EFFF	67	

Note: This example assumes that FFFF is the memory location addressed when all lines of the address bus go to the high state.

Transfer from Accumulator A to Accumulator B **TAB**

Operation: ACCB ← (ACCA)

Description: Moves the contents of ACCA to ACCB. The former contents of ACCB are lost. The contents of ACCA are not affected.

Condition Codes:
- H: Not affected.
- I: Not affected.
- N: Set if the most significant bit of the contents of the accumulator is set; cleared otherwise.
- Z: Set if all bits of the contents of the accumulator are cleared; cleared otherwise.
- V: Cleared.
- C: Not affected.

Boolean Formulae for Condition Codes:

$N = R_7$

$Z = \overline{R_7} \cdot \overline{R_6} \cdot \overline{R_5} \cdot \overline{R_4} \cdot \overline{R_3} \cdot \overline{R_2} \cdot \overline{R_1} \cdot \overline{R_0}$

$V = 0$

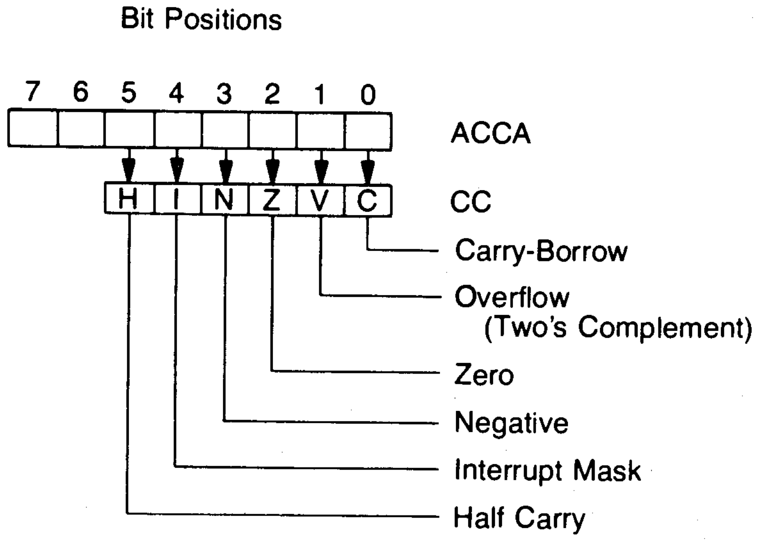
Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	2	1	16	026	022

TAP

Transfer from Accumulator A to Processor Condition Codes Register

Operation: CC ← (ACCA)



Description: Transfers the contents of bit positions 0 thru 5 of accumulator A to the corresponding bit positions of the processor condition codes register. The contents of accumulator A remain unchanged.

Condition Codes: Set or reset according to the contents of the respective bits 0 thru 5 of accumulator A.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	**Execution Time (No. of cycles)**	**Number of bytes of machine code**	**Coding of First (or only) byte of machine code**		
			HEX.	**OCT.**	**DEC.**
INHERENT	2	1	06	006	006

Transfer from Accumulator B to Accumulator A **TBA**

Operation: ACCA ← (ACCB)

Description: Moves the contents of ACCB to ACCA. The former contents of ACCA are lost. The contents of ACCB are not affected.

Condition Codes:

H:	Not affected.
I:	Not affected.
N:	Set if the most significant accumulator bit is set; cleared otherwise.
Z:	Set if all accumulator bits are cleared; cleared otherwise.
V:	Cleared.
C:	Not affected.

Boolean Formulae for Condition Codes:

$N = R_7$

$Z = \overline{R_7} \cdot \overline{R_6} \cdot \overline{R_5} \cdot \overline{R_4} \cdot \overline{R_3} \cdot \overline{R_2} \cdot \overline{R_1} \cdot \overline{R_0}$

$V = 0$

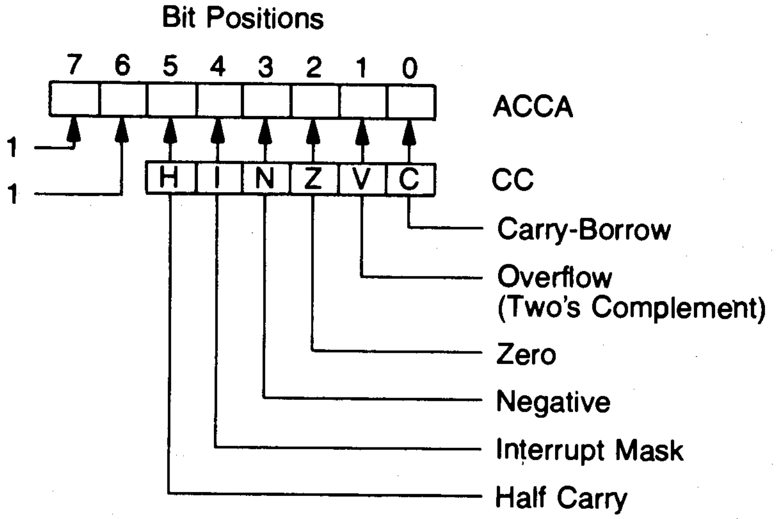
Addressing Modes, Execution Time, and Machine Code (hexadecimal/octal/decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	2	1	17	027	023

TPA

Transfer from Processor Condition Codes Register to Accumulator A

Operation: ACCA ← (CC)



Description: Transfers the contents of the processor condition codes register to corresponding bit positions 0 thru 5 of accumulator A. Bit positions 6 and 7 of accumulator A are set (i.e. go to the "1" state). The processor condition codes register remains unchanged.

Condition Codes: Not affected.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	2	1	07	007	007

Test TST

Operation: (ACCX) − 00
(M) − 00

Description: Set condition codes N and Z according to the contents of ACCX or M.

Condition Codes:
H: Not affected.
I: Not affected.
N: Set if most significant bit of the contents of ACCX or M is set; cleared otherwise.
Z: Set if all bits of the contents of ACCX or M are cleared; cleared otherwise.
V: Cleared.
C: Cleared.

Boolean Formulae for Condition Codes:

$N = M_7$

$Z = \overline{M_7} \cdot \overline{M_6} \cdot \overline{M_5} \cdot \overline{M_4} \cdot \overline{M_3} \cdot \overline{M_2} \cdot \overline{M_1} \cdot \overline{M_0}$

$V = 0$

$C = 0$

Addressing Formats:

See Table A-3.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/octal/decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A	2	1	4D	115	077
B	2	1	5D	135	093
EXT	6	3	7D	175	125
IND	7	2	6D	155	109

TSX

Transfer from Stack Pointer to Index Register

Operation: IX ← (SP) + 0001

Description: Loads the index register with one plus the contents of the stack pointer. The contents of the stack pointer remain unchanged.

Condition Codes: Not affected.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	**Execution Time (No. of cycles)**	**Number of bytes of machine code**	**Coding of First (or only) byte of machine code**		
			HEX.	**OCT.**	**DEC.**
INHERENT	4	1	30	060	048

Transfer From Index Register to Stack Pointer **TXS**

Operation: SP ← (IX) − 0001

Description: Loads the stack pointer with the contents of the index register, minus one. The contents of the index register remain unchanged.

Condition Codes: Not affected.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	**Execution Time (No. of cycles)**	**Number of bytes of machine code**	**Coding of First (or only) byte of machine code**		
			HEX.	**OCT.**	**DEC.**
INHERENT	4	1	35	.065	053

WAI

Wait for Interrupt

Operation: PC ← (PC) + 0001
↓ (PCL) , SP ← (SP)-0001
↓ (PCH) , SP ← (SP)-0001
↓ (IXL) , SP ← (SP)-0001
↓ (IXH) , SP ← (SP)-0001
↓ (ACCA) , SP ← (SP)-0001
↓ (ACCB) , SP ← (SP)-0001
↓ (CC) , SP ← (SP)-0001

Condition Codes: Not affected.

Description: The program counter is incremented (by 1). The program counter, index register, and accumulators A and B, are pushed into the stack. The condition codes register is then pushed into the stack, with condition codes H, I, N, Z, V, C going respectively into bit positions 5 thru 0, and the top two bits (in bit positions 7 and 6) are set (to the 1 state). The stack pointer is decremented (by 1) after each byte of data is stored in the stack.

Execution of the program is then suspended until an interrupt from a peripheral device is signalled, by the interrupt request control input going to a low state.

When an interrupt is signalled on the interrupt request line, and provided the I bit is clear, execution proceeds as follows. The interrupt mask bit is set. The program counter is then loaded with the address stored in the internal interrupt pointer at memory locations (n-7) and (n-6), where n is the address corresponding to a high state on all lines of the address bus.

Condition Codes:
H: Not affected.
I: Not affected until an interrupt request signal is detected on the interrupt request control line. When the interrupt request is received the I bit is set and further execution takes place, provided the I bit was initially clear.
N: Not affected.
Z: Not affected.
V: Not affected.
C: Not affected.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	9	1	3E	076	062

Addressing Mode of Second Operand	First Operand	
	Accumulator A	Accumulator B
IMMediate	CCC A #number CCC A #symbol CCC A #expression CCC A #'C	CCC B #number CCC B #symbol CCC B #expression CCC B #'C
DIRect or EXTended	CCC A number CCC A symbol CCC A expression	CCC B number CCC B symbol CCC B expression
INDexed	CCC A X CCC Z ,X CCC A number,X CCC A symbol,X CCC A expression,X	CCC B X CCC B ,X CCC B number,X CCC B symbol,X CCC B expression,X

Notes: 1. CCC = mnemonic operator of source instruction.
2. "symbol" may be the special symbol "*".
3. "expression" may contain the special symbol "*".
4. space may be omitted before A or B.

Applicable to the following source instructions:

ADC ADD AND BIT CMP
EOR LDA ORA SBC SUB

*Special symbol indicating program-counter.

TABLE A-1. Addressing Formats (1)

Addressing Mode of Second Operand	First Operand	
	Accumulator A	Accumulator B
DIRect or EXTended	STA A number STA A symbol STA A expression	STA B number STA B symbol STA B expression
INDexed	STA A X STA A ,X STA A number,X STA A symbol,X STA A expression,X	STA B X STA B ,X STA B number,X STA B symbol,X STA B expression,X

Notes: 1. "symbol" may be the special symbol "*".
2. "expression" may contain the special symbol "*".
3. Space may be omitted before A or B.

Applicable to the source instruction:

STA

*Special symbol indicating program-counter.

TABLE A-2. Addressing Formats (2)

Operand or Addressing Mode	Formats
Accumulator A	CCC A
Accumulator B	CCC B
EXTended	CCC number CCC symbol CCC expression
INDexed	CCC X CCC ,X CCC number,X CCC symbol,X CCC expression,X

Notes: 1. CCC = mnemonic operator of source instruction.
2. "symbol" may be the special symbol "*".
3. "expression" may contain the special symbol "*".
4. Space may be omitted before A or B.

Applicable to the following source instructions:

ASL ASR CLR COM DEC INC
LSR NEG ROL ROR TST

*Special symbol indicating program-counter.

TABLE A-3. Addressing Formats (3)

Operand	Formats
Accumulator A	CCC A
Accumulator B	CCC B

Notes: 1. CCC = mnemonic operator of source instruction.
2. Space may be omitted before A or B.

Applicable to the following source instructions:

PSH PUL

TABLE A-4. Addressing Formats (4)

Addressing Mode	Formats
IMMediate	CCC #number CCC #symbol CCC #expression CCC #'C
DIRect or EXTended	CCC number CCC symbol CCC expression
INDexed	CCC X CCC ,X CCC number,X CCC symbol,X CCC expression,X

Notes: 1. CCC = mnemonic operator of source instruction.
2. "symbol" may be the special symbol "*".
3. "expression" may contain the special symbol "*".

Applicable to the following source instructions:
CPX LDS LDX

*Special symbol indicating program-counter.

TABLE A-5. Addressing Formats (5)

Addressing Mode	Formats
DIRect or EXTended	CCC number CCC symbol CCC expression
INDexed	CCC X CCC ,X CCC number,X CCC symbol,X CCC expression,X

Notes: 1. CCC = mnemonic operator of source instruction.
2. "symbol" may be the special symbol "*".
3. "expression" may contain the special symbol "*".

Applicable to the following source instructions:
STS STX

*Special symbol indicating program-counter.

TABLE A-6. Addressing Formats (6)

Addressing Mode	Formats
EXTended	CCC number CCC symbol CCC expression
INDexed	CCC X CCC ,X CCC number,X CCC symbol,X CCC expression,X

Notes: 1. CCC = mnemonic operator of source instruction.
2. "symbol" may be the special symbol "*".
3. "expression" may contain the special symbol "*".

Applicable to the following source instructions:

JMP JSR

*Special symbol indicating program-counter.

TABLE A-7. Addressing Formats (7)

Addressing Mode	Formats
RELative	CCC number CCC symbol CCC expression

Notes: 1. CCC = mnemonic operator of source instruction.
2. "symbol" may be the special symbol "*".
3. "expression" may contain the special symbol "*".

Applicable to the following source instructions:

BCC BCS BEQ BGE BGT BHI BLE BLS
BLT BMI BNE BPL BRA BSR BVC BVS

*Special symbol indicating program-counter.

TABLE A-8. Addressing Formats (8)

INDEX

A

Accumulator, 2-15, 5-5
Addend, 3-4
Address Register, 2-15
Address, 2-45
Addressing Modes, 2-41, 64
Addressing, Direct, 2-44
Addressing, Extended, 5-35
Addressing, Immediate, 2-43
Addressing, Implied, 2-42
Addressing, Indexed, 5-37
Addressing, Inherent, 2-42
Addressing, Relative, 4-5
Algorithms, 4-27
Alphanumeric Codes, 1-46
AND, 3-33
Arithmetic Instructions, 5-14
Arithmetic Logic Unit, 2-14
ASCII Code, 1-46, 2-7
Augend, 3-4

B

Base, 1-4
BAUDOT Code, 1-50
BCD-to-Binary, 4-45
Binary, 1-9
Binary Addition, 3-4
Binary Coded Decimal, 1-40
Binary Division, 3-12
Binary Fraction, 1-23
Binary Integer, 1-23
Binary Multiplication, 3-9
Binary Number System, 1-40
Binary Point, 1-11
Binary Subtraction, 3-6
Binary-to-BCD, 4-49
Binary-to-Decimal, 1-11
Binary-to-Hexadecimal, 1-34
Binary-to-Octal, 1-22
Bit, 2-7
Boolean Operations, 3-33
Branch Instructions, 5-24
Branching, 6-22
Branching Backward, 4-14
Branching Forward, 4-12
Bus, 2-4
Byte, 2-8

C

Carry Register, 4-21
Cascade Stack, 6-4
Computer Word, 2-7
Condition Code Instructions, 5-28
Condition Code Register, 5-5
Condition Codes, 4-20
Conditional Branching, 4-18
Controller-Sequencer, 2-16

D

Data Handling Instructions, 5-16
Data Register, 2-15
Data Test Instructions, 5-21
Decimal Adjust Accumulator, 4-61
Decimal-to-BCD,1-42
Decimal-to-Binary, 1-12
Decimal-to-Hexadecimal, 1-31
Decimal-to-Octal, 1-19
Difference, 3-6
Direct Addressing, 2-44
Display, 6-27
Dividend, 3-12
Division by Subtraction, 4-41
Divisor, 3-12

E

Exclusive OR, 3-36
Execute Phase, 2-19, 32
Extended Addressing, 5-35

F

Fetch Phase, 2-19, 27
Fetch-Execute Sequence, 2-19
Flowchart, 4-28
Fractional Numbers, 1-5, 13, 18, 29

G

Gray Code, 1-41, 45

H

Hexadecimal, 1-27
Hexadecimal-to-Binary, 1-34
High Order Byte, 2-9

I

Immediate Addressing, 2-43
Implied Addressing, 2-42
Index Register Instructions, 5-22
Index Register, 5-8
Indexed Addressing, 5-37
Inherent Addressing, 2-42
Input, 2-5, 6-25
Instruction, 2-6
Instruction Decoder, 2-15
Instruction Sequence, 2-21
Instruction Set Summary, 5-45
Instructions, Arithmetic, 5-14
Instructions, Branch, 5-24, 6-21
Instructions, Condition Code, 5-28
Instructions, Data Handling, 5-16
Instructions, Data Test, 5-21
Instructions, Index Register, 5-22
Instructions, Interrupt Mask, 6-43
Instructions, Logic, 5-20
Instructions, Stack Pointer, 5-22
Instructions, Subroutine, 6-18
Integer, 1-5
Interrupt Mask Instructions, 6-43
Interrupt Request, 6-42
Interrupt Vectors, 6-36
Interrupts, 6-35
Invert, 3-38
I/O, 2-5, 6-25
I/O Interrupts, 6-32
I/O Programming, 6-29

J

Jumps, 6-16

L

Least Significant Bit, 1-11, 2-9
Least Significant Digit, 1-6
LED Display, 6-27
LIFO, 6-6
Logic Instructions, 5-20
Looping, 4-4
Low Order Byte, 2-9

M

Mathematical Symbols, Flowchart, 4-30
Memory, 2-16
Memory Map, 6-35
Memory Stack, 6-7
Microcomputer, 2-4, 12
Microprocessor, 2-4, 12
Minuend, 3-6
Mnemonic, 2-20
Most Significant Bit, 1-11, 2-9
Most Significant Digit, 1-6
MPU Architecture, 5-4
MPU Block Diagram, 5-9
MPU Cycle, 2-44
MPU Programming Model, 5-6
Multiple-Precision Arithmetic,4-55
Multiplicand, 3-9
Multiplication by Addition, 4-37
Multiplier, 3-9

N

Negative Numbers, 3-14
Negative Powers, 1-5
Negative Register, 4-20
Nesting, 6-20
Non-Maskable Interrupt, 6-38

O

Octal, 1-18
Octal-to-Binary, 1-22
One's Complement, 3-15
Opcode, 2-20
OR, 3-34
Output, 2-5, 6-25
Outside World, 2-5
Overflow Register, 4-22

P

Parity, 1-48
Port, 2-5
Positional Notation, 1-4, 9, 18, 27
Product, 3-9
Program, 2-6
Program Counter, 2-15, 5-5
Program Execution, 2-26
Programmer, 2-6
Programming, 4-27
Pull, 6-6, 9
Push, 6-5, 8

Q

Quotient, 3-12

R

Radix, 1-4
Random Access Memory, 2-18
Read Only Memory, 2-18
Read Operation, 2-17
Relative Addressing, 4-5
Remainder, 3-12
Reset, 6-35
Return From Interrupt, 6-40

S

Sign and Magnitude, 3-14
Signed Numbers, 3-28
Software Interrupt, 6-44
Special Binary Codes, 1-45
Stack Operations, 6-4
Stack Pointer, 5-8, 6-7
Stack Pointer Instructions, 5-22
Stored Program Concept, 2-6
Straight Line Programs, 4-4
Subroutine Instructions, 6-18
Subroutine Nesting, 6-20
Subroutines, 6-15
Subtrahend, 3-6
Sum, 3-5
Switch Interface, 6-28
Symbols, Flowchart, 4-28

T

Ten's Complement, 3-25
Two's Complement, 3-16, 24, 27, 4-14

U

Unconditional Branching, 4-4

V

Vector Assignments, 6-36

W

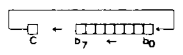
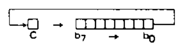
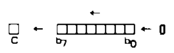
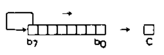
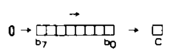
Wait For Interrupt, 6-45
Weighting System, 1-40
Word, 2-7
Write Operation, 2-18

Z

Zero Register, 4-20

ISBN 0-87119-105-9 TEXT BOOK — EB-6401-A 595-3429-07

PROGRAMMING CARD
6800/6808 INSTRUCTION SET

ACCUMULATOR AND MEMORY		ADDRESSING MODES															BOOLEAN/ARITHMETIC OPERATION	COND. CODE REG.					
		IMMED			DIRECT			INDEX			EXTND			INHER			(All register labels refer to contents)	5	4	3	2	1	0
OPERATIONS	MNEMONIC	OP	~	#	OP	~	#	OP	~	#	OP	~	#	OP	~	#		H	I	N	Z	V	C
Add	ADDA	8B	2	2	9B	3	2	AB	5	2	BB	4	3				A + M → A	↕	●	↕	↕	↕	↕
	ADDB	CB	2	2	DB	3	2	EB	5	2	FB	4	3				B + M → B	↕	●	↕	↕	↕	↕
Add Acmltrs	ABA													1B	2	1	A + B → A	↕	●	↕	↕	↕	↕
Add with Carry	ADCA	89	2	2	99	3	2	A9	5	2	B9	4	3				A + M + C → A	↕	●	↕	↕	↕	↕
	ADCB	C9	2	2	D9	3	2	E9	5	2	F9	4	3				B + M + C → B	↕	●	↕	↕	↕	↕
And	ANDA	84	2	2	94	3	2	A4	5	2	B4	4	3				A • M → A	●	●	↕	↕	R	●
	ANDB	C4	2	2	D4	3	2	E4	5	2	F4	4	3				B • M → B	●	●	↕	↕	R	●
Bit Test	BITA	85	2	2	95	3	2	A5	5	2	B5	4	3				A • M	●	●	↕	↕	R	●
	BITB	C5	2	2	D5	3	2	E5	5	2	F5	4	3				B • M	●	●	↕	↕	R	●
Clear	CLR							6F	7	2	7F	6	3				00 → M	●	●	R	S	R	R
	CLRA													4F	2	1	00 → A	●	●	R	S	R	R
	CLRB													5F	2	1	00 → B	●	●	R	S	R	R
Compare	CMPA	81	2	2	91	3	2	A1	5	2	B1	4	3				A − M	●	●	↕	↕	↕	↕
	CMPB	C1	2	2	D1	3	2	E1	5	2	F1	4	3				B − M	●	●	↕	↕	↕	↕
Compare Acmltrs	CBA													11	2	1	A − B	●	●	↕	↕	↕	↕
Complement, 1's	COM							63	7	2	73	6	3				$\bar{M}$ → M	●	●	↕	↕	R	S
	COMA													43	2	1	$\bar{A}$ → A	●	●	↕	↕	R	S
	COMB													53	2	1	$\bar{B}$ → B	●	●	↕	↕	R	S
Complement, 2's	NEG							60	7	2	70	6	3				00 − M → M	●	●	↕	↕	①	②
(Negate)	NEGA													40	2	1	00 − A → A	●	●	↕	↕	①	②
	NEGB													50	2	1	00 − B → B	●	●	↕	↕	①	②
Oecimal Adjust, A	OAA													19	2	1	Converts Binary Add. of BCO Characters into BCD Format	●	●	↕	↕	↕	③
Decrement	DEC							6A	7	2	7A	6	3				M − 1 → M	●	●	↕	↕	④	●
	DECA													4A	2	1	A − 1 → A	●	●	↕	↕	④	●
	DECB													5A	2	1	B − 1 → B	●	●	↕	↕	④	●
Exclusive DR	EORA	88	2	2	98	3	2	A8	5	2	B8	4	3				A ⊕ M → A	●	●	↕	↕	R	●
	EORB	C8	2	2	D8	3	2	E8	5	2	F8	4	3				B ⊕ M → B	●	●	↕	↕	R	●
Increment	INC							6C	7	2	7C	6	3				M + 1 → M	●	●	↕	↕	⑤	●
	INCA													4C	2	1	A + 1 → A	●	●	↕	↕	⑤	●
	INCB													5C	2	1	B + 1 → B	●	●	↕	↕	⑤	●
Load Acmltr	LDAA	86	2	2	96	3	2	A6	5	2	B6	4	3				M → A	●	●	↕	↕	R	●
	LOAB	C6	2	2	06	3	2	E6	5	2	F6	4	3				M → B	●	●	↕	↕	R	●
Or, Inclusive	DRAA	8A	2	2	9A	3	2	AA	5	2	BA	4	3				A + M → A	●	●	↕	↕	R	●
	ORAB	CA	2	2	DA	3	2	EA	5	2	FA	4	3				B + M → B	●	●	↕	↕	R	●
Push Data	PSHA													36	4	1	A → M_{SP}, SP − 1 → SP	●	●	●	●	●	●
	PSHB													37	4	1	B → M_{SP}, SP − 1 → SP	●	●	●	●	●	●
Pull Oata	PULA													32	4	1	SP + 1 → SP, M_{SP} → A	●	●	●	●	●	●
	PULB													33	4	1	SP + 1 → SP, M_{SP} → B	●	●	●	●	●	●
Rotate Left	ROL							69	7	2	79	6	3				M 	●	●	↕	↕	⑥	↕
	ROLA													49	2	1	A	●	●	↕	↕	⑥	↕
	ROLB													59	2	1	B	●	●	↕	↕	⑥	↕
Rotate Right	RDR							66	7	2	76	6	3				M 	●	●	↕	↕	⑥	↕
	RORA													46	2	1	A	●	●	↕	↕	⑥	↕
	RORB													56	2	1	B	●	●	↕	↕	⑥	↕
Shift Left, Arithmetic	ASL							68	7	2	78	6	3				M 	●	●	↕	↕	⑥	↕
	ASLA													48	2	1	A	●	●	↕	↕	⑥	↕
	ASLB													58	2	1	B	●	●	↕	↕	⑥	↕
Shift Right, Arithmetic	ASR							67	7	2	77	6	3				M 	●	●	↕	↕	⑥	↕
	ASRA													47	2	1	A	●	●	↕	↕	⑥	↕
	ASRB													57	2	1	B	●	●	↕	↕	⑥	↕
Shift Right, Logic.	LSR							64	7	2	74	6	3				M 	●	●	R	↕	⑥	↕
	LSRA													44	2	1	A	●	●	R	↕	⑥	↕
	LSRB													54	2	1	B	●	●	R	↕	⑥	↕
Store Acmltr.	STAA				97	4	2	A7	6	2	B7	5	3				A → M	●	●	↕	↕	R	●
	STAB				D7	4	2	E7	6	2	F7	5	3				B → M	●	●	↕	↕	R	●
Subtract	SUBA	80	2	2	90	3	2	A0	5	2	B0	4	3				A − M → A	●	●	↕	↕	↕	↕
	SUBB	C0	2	2	D0	3	2	E0	5	2	F0	4	3				B − M → B	●	●	↕	↕	↕	↕
Subract Acmltrs.	SBA													10	2	1	A − B → A	●	●	↕	↕	↕	↕
Subtr. with Carry	SBCA	82	2	2	92	3	2	A2	5	2	B2	4	3				A − M − C → A	●	●	↕	↕	↕	↕
	SBCB	C2	2	2	D2	3	2	E2	5	2	F2	4	3				B − M − C → B	●	●	↕	↕	↕	↕
Transfer Acmltrs	TAB													16	2	1	A → B	●	●	↕	↕	R	●
	TBA													17	2	1	B → A	●	●	↕	↕	R	●
Test, Zero or Minus	TST							6D	7	2	7D	6	3				M − 00	●	●	↕	↕	R	R
	TSTA													4D	2	1	A − 00	●	●	↕	↕	R	R
	TSTB													5D	2	1	B − 00	●	●	↕	↕	R	R

INDEX REGISTER AND STACK POINTER OPERATIONS	MNEMONIC	IMMED OP	IMMED ~	IMMED #	DIRECT OP	DIRECT ~	DIRECT #	INDEX OP	INDEX ~	INDEX #	EXTND OP	EXTND ~	EXTND #	INHER OP	INHER ~	INHER #	BOOLEAN/ARITHMETIC OPERATION	5 H	4 I	3 N	2 Z	1 V	0 C
Compare Index Reg	CPX	8C	3	3	9C	4	2	AC	6	2	BC	5	3				$(X_H/X_L) - (M/M + 1)$	●	●	⑦	↕	⑧	●
Decrement Index Reg	DEX													09	4	1	$X - 1 \rightarrow X$	●	●	●	↕	●	●
Decrement Stack Pntr	DES													34	4	1	$SP - 1 \rightarrow SP$	●	●	●	●	●	●
Increment Index Reg	INX													08	4	1	$X + 1 \rightarrow X$	●	●	●	↕	●	●
Increment Stack Pntr	INS													31	4	1	$SP + 1 \rightarrow SP$	●	●	●	●	●	●
Load Index Reg	LDX	CE	3	3	DE	4	2	EE	6	2	FE	5	3				$M \rightarrow X_H, (M + 1) \rightarrow X_L$	●	●	⑨	↕	R	●
Load Stack Pntr	LDS	8E	3	3	9E	4	2	AE	6	2	BE	5	3				$M \rightarrow SP_H, (M + 1) \rightarrow SP_L$	●	●	⑨	↕	R	●
Store Index Reg	STX				DF	5	2	EF	7	2	FF	6	3				$X_H \rightarrow M, X_L \rightarrow (M + 1)$	●	●	⑨	↕	R	●
Store Stack Pntr	STS				9F	5	2	AF	7	2	BF	6	3				$SP_H \rightarrow M, SP_L \rightarrow (M + 1)$	●	●	⑨	↕	R	●
Indx Reg → Stack Pntr	TXS													35	4	1	$X - 1 \rightarrow SP$	●	●	●	●	●	●
Stack Pntr → Indx Reg	TSX													30	4	1	$SP + 1 \rightarrow X$	●	●	●	●	●	●

JUMP AND BRANCH OPERATIONS	MNEMONIC	RELATIVE OP	RELATIVE ~	RELATIVE #	INDEX OP	INDEX ~	INDEX #	EXTND OP	EXTND ~	EXTND #	INHER OP	INHER ~	INHER #	BRANCH TEST	5 H	4 I	3 N	2 Z	1 V	0 C
Branch Always	BRA	20	4	2										None	●	●	●	●	●	●
Branch If Carry Clear	BCC	24	4	2										$C = 0$	●	●	●	●	●	●
Branch If Carry Set	BCS	25	4	2										$C = 1$	●	●	●	●	●	●
Branch If = Zero	BEQ	27	4	2										$Z = 1$	●	●	●	●	●	●
Branch If ≥ Zero	BGE	2C	4	2										$N \oplus V = 0$	●	●	●	●	●	●
Branch If > Zero	BGT	2E	4	2										$Z + (N \oplus V) = 0$	●	●	●	●	●	●
Branch If Higher	BHI	22	4	2										$C + Z = 0$	●	●	●	●	●	●
Branch If ≤ Zero	BLE	2F	4	2										$Z + (N \oplus V) = 1$	●	●	●	●	●	●
Branch If Lower Or Same	BLS	23	4	2										$C + Z = 1$	●	●	●	●	●	●
Branch If < Zero	BLT	2D	4	2										$N \oplus V = 1$	●	●	●	●	●	●
Branch If Minus	BMI	2B	4	2										$N = 1$	●	●	●	●	●	●
Branch If Not Equal Zero	BNE	26	4	2										$Z = 0$	●	●	●	●	●	●
Branch If Overflow Clear	BVC	28	4	2										$V = 0$	●	●	●	●	●	●
Branch If Overflow Set	BVS	29	4	2										$V = 1$	●	●	●	●	●	●
Branch If Plus	BPL	2A	4	2										$N = 0$	●	●	●	●	●	●
Branch To Subroutine	BSR	8D	8	2										See Special Operations	●	●	●	●	●	●
Jump	JMP				6E	4	2	7E	3	3				See Special Operations	●	●	●	●	●	●
Jump To Subroutine	JSR				AD	8	2	BD	9	3				See Special Operations	●	●	●	●	●	●
No Operation	NOP										01	2	1	Advances Prog. Cntr. Only	●	●	●	●	●	●
Return From Interrupt	RTI										3B	10	1		⑩	⑩	⑩	⑩	⑩	⑩
Return From Subroutine	RTS										39	5	1	See special Operations	●	●	●	●	●	●
Software Interrupt	SWI										3F	12	1	See special Operations	●	S	●	●	●	●
Wait for Interrupt	WAI										3E	9	1	See special Operations	●	⑪	●	●	●	●

CONDITIONS CODE REGISTER OPERATIONS	MNEMONIC	INHER OP	INHER ~	INHER =	BOOLEAN OPERATION	5 H	4 I	3 N	2 Z	1 V	0 C
Clear Carry	CLC	0C	2	1	$0 \rightarrow C$	●	●	●	●	●	R
Clear Interrupt Mask	CLI	0E	2	1	$0 \rightarrow I$	●	R	●	●	●	●
Clear Overflow	CLV	0A	2	1	$0 \rightarrow V$	●	●	●	●	R	●
Set Carry	SEC	0D	2	1	$1 \rightarrow C$	●	●	●	●	●	S
Set Interrupt Mask	SEI	0F	2	1	$1 \rightarrow I$	●	S	●	●	●	●
Set Overflow	SEV	0B	2	1	$1 \rightarrow V$	●	●	●	●	S	●
Acmltr A → CCR	TAP	06	2	1	$A \rightarrow CCR$	⑫	⑫	⑫	⑫	⑫	⑫
CCR → Acmltr A	TPA	07	2	1	$CCR \rightarrow A$	●	●	●	●	●	●

CONDITION CODE REGISTER NOTES:

(Bit set if test is true and cleared otherwise)

①	(Bit V)	Test: Result = 10000000?
②	(Bit C)	Test: Result = 00000000?
③	(Bit C)	Test: Decimal value of most significant BCD Character greater than nine? (Not cleared if previously set.)
④	(Bit V)	Test: Operand = 10000000 prior to execution?
⑤	(Bit V)	Test: Operand = 01111111 prior to execution?
⑥	(Bit V)	Test: Set equal to result of $N \oplus C$ after shift has occurred.
⑦	(Bit N)	Test: Sign bit of most significant (MS) byte of result = 1?
⑧	(Bit V)	Test: 2's complement overflow from subtraction of LS bytes?
⑨	(Bit N)	Test: Result less than zero? (Bit 15 = 1)
⑩	(All)	Load Condition Code Register from Stack. (See Special Operations)
⑪	(Bit I)	Set when interrupt occurs. If previously set, a Non-Maskable Interrupt is required to exit the wait state.
⑫	(ALL)	Set according to the contents of Accumulator A.

LEGEND:

OP	Operation Code (Hexadecimal);	00	Byte = Zero;
~	Number of MPU Cycles;	H	Half-carry from bit 3;
#	Number of Program Bytes;	I	Interrupt mask
+	Arithmetic Plus;	N	Negative (sign bit)
–	Arithmetic Minus;	Z	Zero (byte)
•	Boolean AND;	V	Overflow, 2's complement
M_{SP}	Contents of memory location pointed to be Stack Pointer;	C	Carry from bit 7
+	Boolean Inclusive OR;	R	Reset Always
⊕	Boolean Exclusive OR;	S	Set Always
$\overline{M}$	Complement of M;	↕	Test and set if true, cleared otherwise
→	Transfer Into;	●	Not Affected
0	Bit = Zero;	CCR	Condition Code Register
		LS	Least Significant
		MS	Most Significant