HEATHKIT
CONTINUING
EDUCATION

Individual Learning Program

MICROPROCESSORS

Unit 9

PROGRAMMING EXPERIMENTS

EE-3401

HEATH COMPANY
BENTON HARBOR, MICHIGAN 49022

CONTENTS

Introduction .... Page 9-3

Experiment 1. Binary/Decimal Training Program .... Page 9-4

Experiment 2. Hexadecimal/Decimal Training Program .... Page 9-11

Experiment 3. Straight Line Programs .... Page 9-19

Experiment 4. Arithmetic and Logic Instructions .... Page 9-35

Experiment 5. Program Branches .... Page 9-45

Experiment 6. Additional Instructions .... Page 9-80

Experiment 7. New Addressing Modes .... Page 9-100

Experiment 8. Arithmetic Operations .... Page 9-109

Experiment 9. Stack Operations .... Page 9-119

Experiment 10. Subroutines .... Page 9-127

UNIT 9

PROGRAMMING EXPERIMENTS

INTRODUCTION

This Unit contains ten programming experiments that are to be run on the Microprocessor Trainer. At the end of Units 1 through 6, you will be instructed to perform one or more of these experiments. Do not confuse these with the Interfacing Experiments which are in Unit 10.

The early programs given in this Manual are extremely simple. The later programs are more complex, but you will be able to accomplish them as you become familiar with the instruction set and programming techniques. Before you finish this course, you will be writing programs that will turn the trainer into a clock, a musical instrument, a digital voltmeter, etc.

When you complete an experiment, return to the activity guide of the unit that directed you to the experiment. This is important because you will be jumping from one point to another quite frequently.

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 ET-3400 Microprocessor Trainer.

To demonstrate the versatility of the ET-3400 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 IC14 through IC17 before starting the experiments in this unit.

If your Trainer is model number ET-3400A, reinstall the 2114 RAM IC's at IC14 and IC15 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.

5. Using the Trainer keys, enter the Binary-to-Decimal training program shown in Figure 9-1. 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.

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	C0	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	F0	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	86
0018	F2	0041	CB	006A	02
0019	BD	0042	0A	006B	CE
001A	FD	0043	4D	006C	00
001B	93	0044	26	006D	00
001C	B6	0045	F9	006E	09
001D	C0	0046	96	006F	26
001E	06	0047	00	0070	FD
001F	01	0048	84	0071	4A
0020	46	0049	0F	0072	26
0021	25	004A	15	0073	F7
0022	F9	004B	90	0074	96
0023	BD	004C	01	0075	01
0024	FC	004D	26	0076	84
0025	BC	004E	0D	0077	3F
0026	BD	004F	BD	0078	97
0027	FE	0050	FE	0079	01
0028	52	0051	52	007A	96
				007B	00
				007C	39
					RESET

*This data may change randomly.

Figure 9-1
Binary-to-decimal training program.

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 007C 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 siginficant digit first.) For values less than 10, enter a 0 before you enter the value. After a short period of time, the Trainer will indicate whether or not your answer is correct.

12. If your answer was correct, the Trainer will display YES. A moment later, the word GO will replace the decimal number.

    If your answer was incorrect, the Trainer will display NO. 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 9-2.

17. Now that you have entered the Decimal-to-Binary program, press the EXAM key and enter address 0000.

HEATH
Schlumberger
Continuing
Education

HEATH COMPANY
BENTON HARBOR, MICHIGAN 49022

IMPORTANT NOTICE

Dear Customer:

Please replace Page 9-9 of your Heath Continuing Education program — EE-3401 Microprocessors — with the enclosed page.

We are sorry for any inconvience this may have caused you.

Thank you,

HEATH COMPANY

ADDRESS	INST/DATA	ADDRESS	INST/DATA	ADDRESS	INST/DATA
0000	00*	0033	4F	0066	00
0001	CE	0034	E6	0067	00
0002	C1	0035	00	0068	80
0003	6F	0036	C5	0069	7E
0004	BD	0037	10	006A	00
0005	FE	0038	27	006B	01
0006	50	0039	03	006C	BD
0007	5E	003A	AB	006D	FE
0008	FE	003B	03	006E	52
0009	96	003C	19	006F	15
000A	00	003D	56	0070	1D
000B	8B	003E	24	0071	00
000C	01	003F	03	0072	00
000D	19	0040	AB	0073	00
000E	81	0041	06	0074	80
000F	63	0042	19	0075	BD
0010	23	0043	08	0076	00
0011	01	0044	8C	0077	7E
0012	4F	0045	00	0078	BD
0013	97	0046	88	0079	FC
0014	00	0047	26	007A	BC
0015	B6	0048	EB	007B	7E
0016	CO	0049	BD	007C	00
0017	03	004A	00	007D	1C
0018	01	004B	7E	007E	36
0019	46	004C	BD	007F	BD
001A	25	004D	FC	0080	00
001B	ED	004E	BC	0081	8E
001C	96	004F	D6	0082	32
001D	00	0050	00	0083	01
001E	BD	0051	11	0084	39
001F	FE	0052	26	0085	00
0020	20	0053	18	0086	00
0021	B6	0054	BD	0087	00
0022	CO	0055	FE	0088	32
0023	06	0056	52	0089	08
0024	01	0057	00	008A	02
0025	46	0058	00	008B	16
0026	25	0059	00	008C	04
0027	F9	005A	3B	008D	01
0028	BD	005B	4F	008E	86
0029	FC	005C	DB	008F	02
002A	BC	005D	BD	0090	CE
002B	C6	005E	00	0091	00
002C	03	005F	7E	0092	00
002D	CE	0060	CE	0093	09
002E	00	0061	C1	0094	26
002F	85	0062	3F	0095	FD
0030	BD	0063	BD	0096	49
0031	FD	0064	FE	0097	26
0032	25	0065	50	0098	F7
				0099	39
					RESET

*This data may change randomly

Figure 9-2
Decimal-to-binary training program.

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.

24. If your answer was correct, the Trainer will display YES a short time after you enter the last binary bit. A moment later, the Trainer will display GO.

    If your answer was incorrect, the Trainer will display NO a short time after you enter the last binary bit. A moment 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.

ADDRESS	INST/DATA	ADDRESS	INST/DATA	ADDRESS	INST/DATA
0000	00*	0033	4F	0062	3F
0001	CE	0034	E6	0063	BD
0002	C1	0035	00	0064	FE
0003	6F	0036	C5	0065	50
0004	BD	0037	10	006A	00
0005	FE	0038	27	006B	01
0006	50	0039	03	006C	BD
0007	5E	003A	AB	006D	FE
0008	FE	003B	03	006E	52
0009	96	003C	19	006F	15
000A	00	003D	56	0070	1D
000B	8B	003E	24	0071	00
000C	01	003F	03	0072	00
000D	19	0040	AB	0073	00
000E	81	0041	06	0074	80
000F	63	0042	19	0075	BD
0010	23	0043	08	0076	00
0011	01	0044	8C	0077	7E
0012	4F	0045	00	0078	BD
0013	97	0046	88	0079	FC
0014	00	0047	26	007A	BC
0015	B6	0048	EB	007B	7E
0016	CO	0049	BD	007C	00
0017	03	004A	00	007D	1C
0018	01	004B	7E	007E	36
0019	46	004C	BD	007F	BD
001A	25	004D	FC	0080	00
001B	ED	004E	BC	0081	8E
001C	96	004F	D6	0082	32
001D	00	0050	00	0083	01
001E	BD	0051	11	0084	39
001F	FE	0052	26	0085	00
0020	20	0053	18	0086	00
0021	B6	0054	BD	0087	00
0022	CO	0055	FE	0088	32
0023	06	0056	52	0089	08
0024	01	0057	00	008A	02
0025	46	0058	00	008B	16
0026	25	0059	00	008C	04
0027	F9	005A	3B	008D	01
0028	BD	005B	4F	008E	86
0029	FC	005C	DB	008F	02
002A	BC	005D	BD	0090	CE
002B	C6	0066	00	0091	00
002C	03	0067	00	0092	00
002D	CE	0068	80	0093	09
002E	00	0069	7E	0094	26
002F	85	005E	00	0095	FD
0030	BD	005F	7E	0096	49
0031	FD	0060	CE	0097	26
0032	25	0061	C1	0098	F7
				0099	39
					RESET

*This data may change randomly

Figure 9-2
Decimal-to-binary training program.

18. Using the FWD key, compare the Trainer memory contents with 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.

24. If your answer was correct, the Trainer will display YES a short time after you enter the last binary bit. A moment later, the Trainer will display GO.

    If your answer was incorrect, the Trainer will display NO a short time after you enter the last binary bit. A moment 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.

$$\begin{array}{lll} & & 5 \\ & 16 & \overline{)\,92} \\ & & -\,80 \\ \hline \text{Remainder} & & 12_{10} = C_{16} \leftarrow \text{LSD} \end{array}$$

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.

$$\begin{array}{lll} & & 0 \\ & 16 & \overline{)\,5} \\ & & -\,0 \\ \hline \text{Remainder} \rightarrow & & 5_{10} = 5_{16} \leftarrow \text{MSD} \end{array}$$

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 9-3, 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	0073	86
0008	FE	002C	80	0050	7E	0074	02
0009	96	002D	BD	0051	00	0075	CE
000A	00	002E	FC	0052	01	0076	00
000B	8B	002F	BC	0053	BD	0077	00
000C	01	0030	BD	0054	FE	0078	09
000D	19	0031	FE	0055	52	0079	26
000E	97	0032	09	0056	00	007A	FD
000F	00	0033	36	0057	00	007B	4A
0010	B6	0034	4F	0058	15	007C	26
0011	CO	0035	D6	0059	9D	007D	F7
0012	03	0036	00	005A	BD	007E	39
0013	46	0037	CO	005B	00		RESET
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	36		
001C	CO	0040	10	0064	BD		
001D	06	0041	1B	0065	00		
001E	46	0042	33	0066	73		
001F	25	0043	11	0067	32		
0020	FA	0044	26	0068	01		
0021	BD	0045	OD	0069	BD		
0022	FC	0046	BD	006A	FD		
0023	BC	0047	FE	006B	8D		

* This data may change randomly

Figure 9-3
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 9-3. 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: $1 \times 16^1 = 16$ ←		↳ $1 \times 16^0 = 1$
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:	$6_{16} = 6_{10}$ ←		→ $B_{16} = 11_{10}$
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 9-4 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 9-4. Use the CHAN key to correct any errors that you find.

    When you are satisfied that the program is correct, press the RESET key.

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	0074	86
0009	96	002D	00	0051	7E	0075	02
000A	00	002E	00	0052	00	0076	CE
000B	4C	002F	80	0053	01	0077	00
000C	81	0030	BD	0054	BD	0078	00
000D	63	0031	FC	0055	FE	0079	09
000E	23	0032	BC	0056	52	007A	26
000F	01	0033	BD	0057	00	007B	FD
0010	4F	0034	FE	0058	00	007C	4A
0011	97	0035	09	0059	15	007D	26
0012	00	0036	5F	005A	9D	007E	F7
0013	B6	0037	80	005B	BD	007F	39
0014	CO	0038	10	005C	00		RESET
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	36		
001D	20	0041	1B	0065	BD		
001E	B6	0042	D6	0066	00		
001F	CO	0043	00	0067	74		
0020	06	0044	11	0068	32		
0021	46	0045	26	0069	01		
0022	25	0046	OD	006A	BD		
0023	FA	0047	BD	006B	FD		

Figure 9-4
Hexadecimal-to-decimal training program

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.

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 9-5 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 9-5
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 9-6.

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 9-6
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 9-7. 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 9-7
Addition 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 9-7. 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 9-8. 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 9-8

Multiplication of a number by another through multiple addition in the direct addressing mode.

21. Load the program shown in Figure 9-8 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 9-9.

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.

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 9-9. 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 9-10.

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.

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 9-10.

30. Now you will examine some common programming pitfalls. Without modifying the previous program, except as directed in Figure 9-11, enter the program listed in Figure 9-11.

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 9-11

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 9-11. 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 9-10 and 9-11, 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 9-12, enter the program listed in Figure 9-12.

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 9-12

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 9-12. Note that the Trainer executed the instructions beyond address 0007. This occured 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 9-13.

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 9-13

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 9-13. 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 9-13 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 9-14.

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 9-14

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 9-15 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 9-16 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 9-15

This program decrements the contents of the accumulator from +5 to −8.

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 9-16
Record results here.

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 9-16 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.

4. Enter the program shown in Figure 9-17. 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? ________.

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.

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 9-17
Adding signed 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 9-18. 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 9-18
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 9-16, 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 9-19 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 9-19
Adding unsigned numbers.

14. Figure 9-20 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 9-19 with that of Figure 9-20.

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 9-14. Either immediate or direct addressing can be used.

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 9-20
Adding signed numbers.

Procedure (Continued)

16. Load the program shown in Figure 9-21. 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? ________.

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 9-14. 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 most significant bits to 0011.

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 9-21
Using the subtract instruction.

Procedure (Continued)

18. Load the program shown in Figure 9-22. 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 9-22
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.

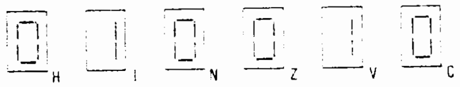


Figure 9-23
Displaying the conditions of the flags.

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 "CC" 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 9-23.

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 9-24 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 9-24. 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 9-24
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.

$$\begin{array}{llcl} 1001 & 0010 & = & 92_{16} \\ 1100 & 0110 & = & C6_{16} \\ \hline 0101 & 1000 & = & 158_{16} \end{array}$$

1
CARRY

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.

$$\begin{array}{lllcl} \overset{1}{\uparrow} & & & & \\ \text{Borrow} & 0000 & 0001 & = & 01_{16} \\ & \underline{0000} & \underline{0010} & = & 02_{16} \\ & 1111 & 1111 & = & FF_{16} \end{array}$$

Notice that the difference is FF_{16}. This will $\underset{\text{set/clear}}{\underline{\qquad}}$ 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 $\underset{\text{set/cleared}}{\underline{\qquad}}$.

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.

$$\begin{array}{llcl} 0100 & 1001 & = & 49_{16}\\ 0110 & 0000 & = & 60_{16}\\ \hline \underline{1}010 & 1001 & = & A9_{16}\end{array}$$

Overflow changes sign bit.

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 ______ the V flag.
set/clear

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 9-25 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 it's name that a conditional branch will occur when the ________ flag is set.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000		CLRA	Clear the accumulator.
0001		STA	Store the product
0002		13	in location 13_{16}.
0003		LDA	Load the accumulator with the
0004		12	multiplier from location 12_{16}.
0005		BEQ	If the multiplier is equal to zero,
0006		09	branch down to the Halt instruction.
0007		DECA	Otherwise, decrement the multiplier.
0008		STA	Store the new value of the
0009		12	multiplier back in location 12_{16}.
000A		LDA	Load the accumulator with the
000B		13	product from location 13_{16}.
000C		ADD	Add
000D		11	the multiplicand to the product.
000E		BRA	Branch back to instruction
000F		F1	in location 01.
0010		HLT	Halt.
0011		05	Multiplicand.
0012		02	Multiplier.
0013		—	Product.

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 9-26. 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 9-26
Single-stepping through the Multiply by repeated addition program.

Discussion

The chart that you completed should be similar to the one shown in Figure 9-27. 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}.

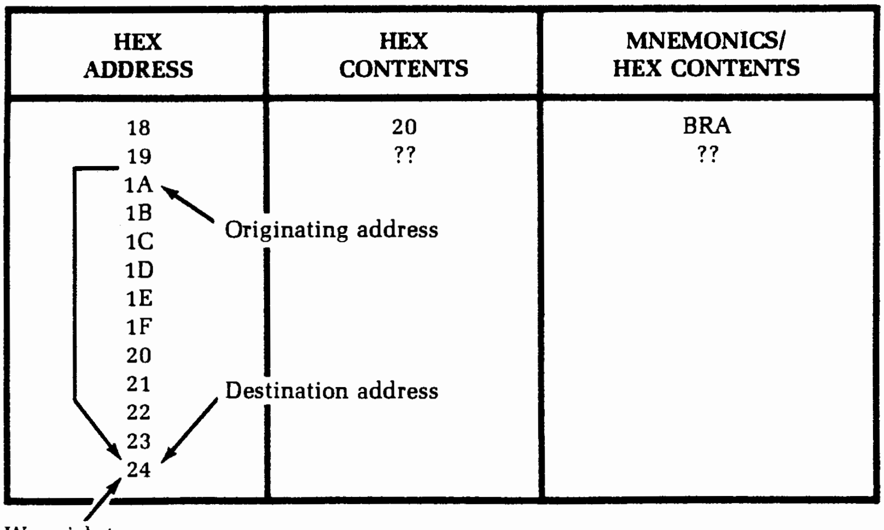
The multiplier is now loaded and, since it's been decremented to 00_{16}, it sets the Z flag. The BEQ instruction checks the Z flag, finds that it's set and branches to the halt instruction at address 0010_{16}. Therefore, the program makes two complete passes, before the multiplier becomes 00_{16}. 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.

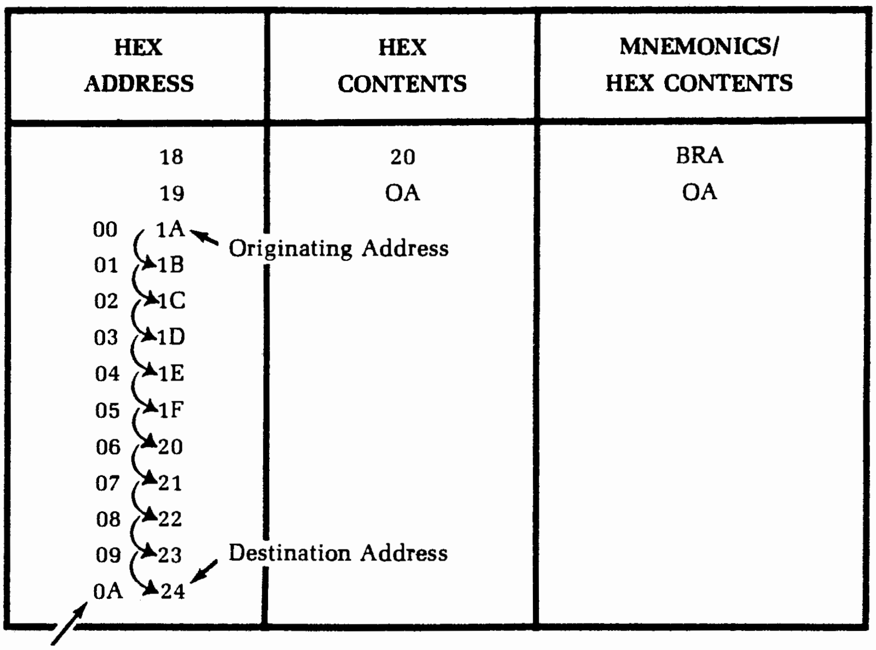
Figure 9-27



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

Figure 9-28

For example, in the program of Figure 9-28, 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 9-29. 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}.



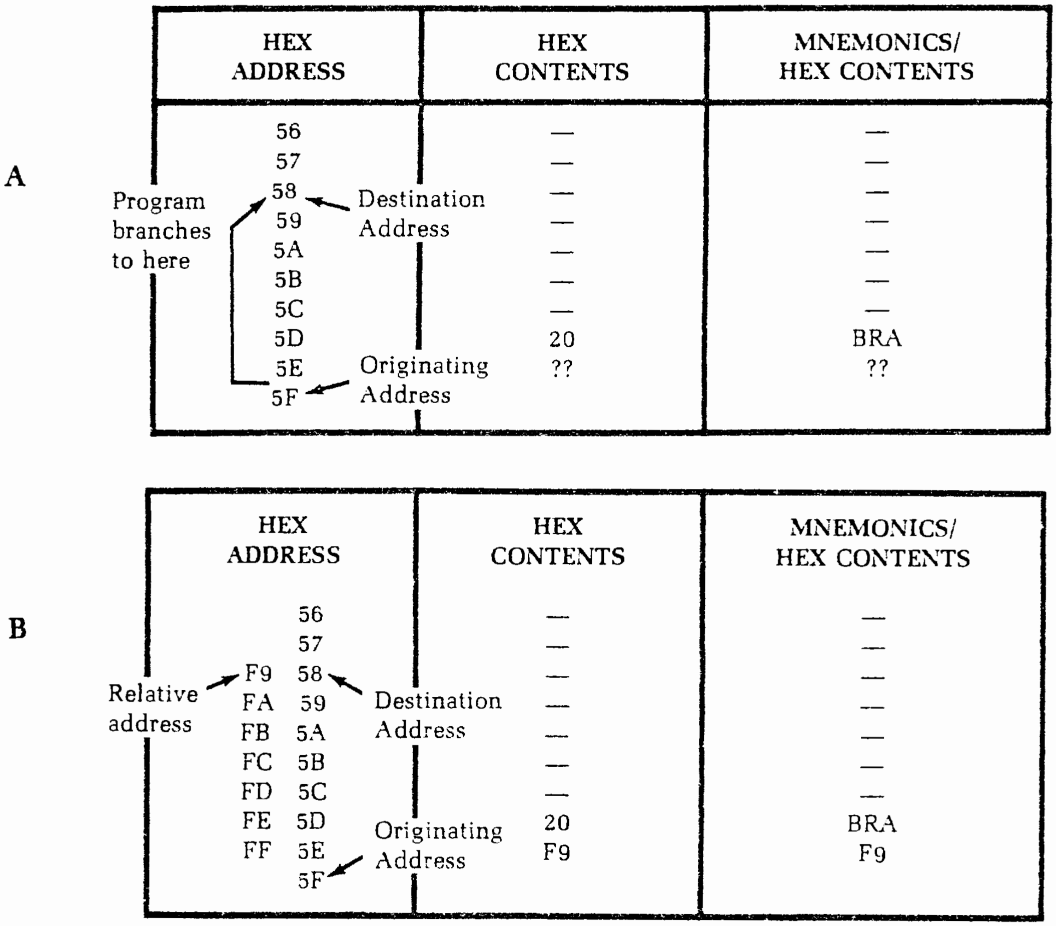
HEX ADDRESS	HEX CONTENTS	MNEMONICS/ HEX CONTENTS
18	20	BRA
19	OA	OA
00 1A		
01 1B		
02 1C		
03 1D		
04 1E		
05 1F		
06 20		
07 21		
08 22		
09 23		
0A 24		

Figure 9-29
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 9-30A, 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 9-30B 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 $F9_{16}$.

Figure 9-31 shows another example of computing the relative address for a larger branch. The branch instruction is at address $B0_{16}$ and therefore, the origination address is $B2_{16}$. We calculate the relative address as shown in Figure 9-31B. Starting with FF_{16} at address $B1_{16}$ we count down to the destination address $A0_{16}$. As the count indicates, the relative address to get to $A0_{16}$ is EE_{16}.


A
Program branches to here
Destination Address
Originating Address
B
Relative address
Destination Address
Originating Address

A

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

B

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 9-30
Branching back

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 9-31

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 9-32. Using this flow chart as a guide and the instructions presented in Figure 9-33, write a program that divides by repeated subtraction.

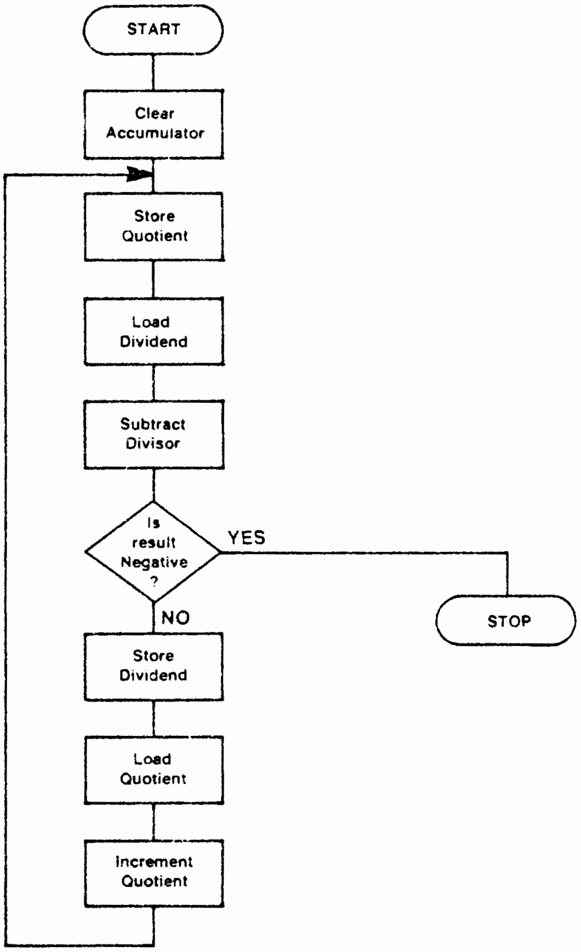


Figure 9-32
Flow chart for dividing by repeated subtraction.

		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 9-33
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 9-34.

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 9-34

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 9-35. 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 9-35
Dividing by repeated subtraction.

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 9-36

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 9-36 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}.

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 9-37 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.

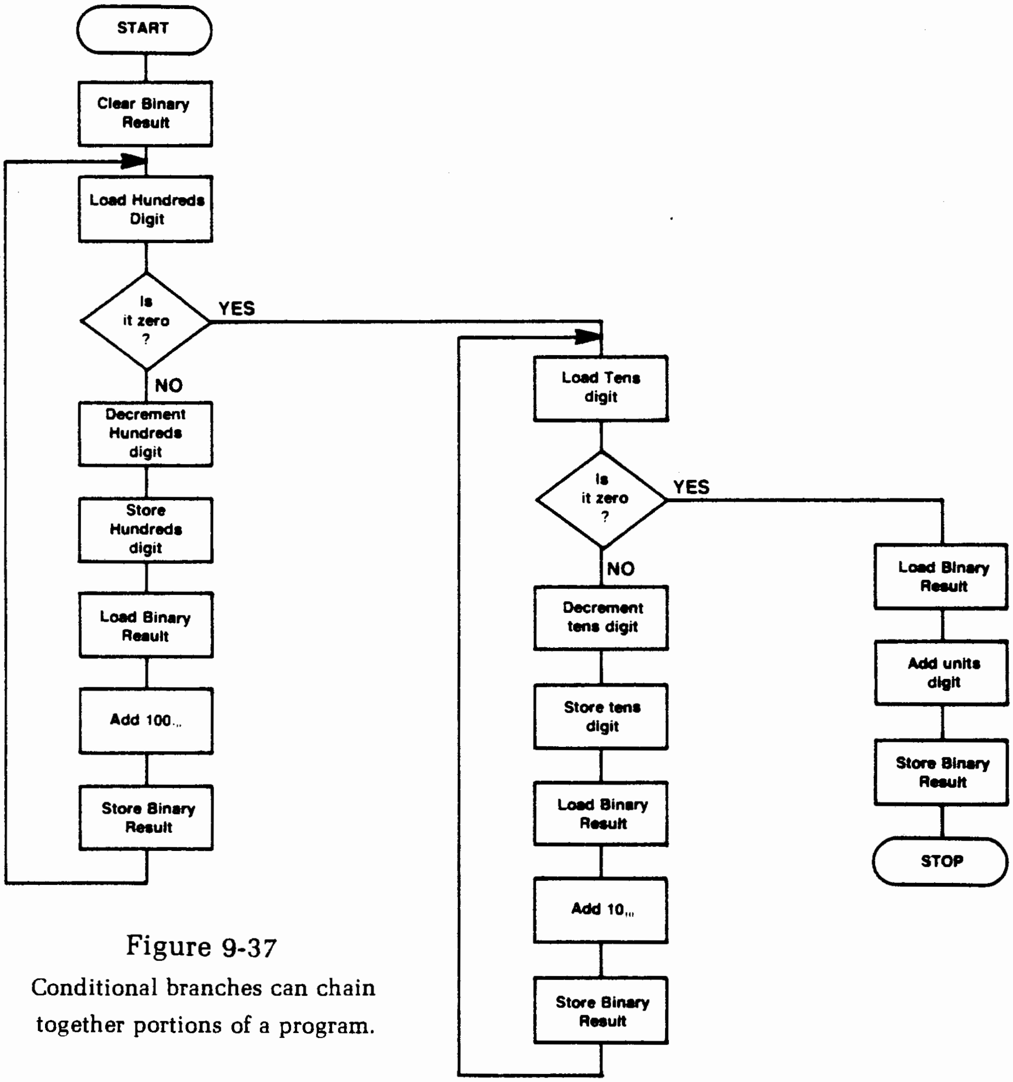


Figure 9-37
Conditional branches can chain together portions of a program.

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 9-38

Program for converting BCD to binary.

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.

Procedure (Continued)

23. Load the program listed in Figure 9-38 into the Trainer. The BCD number 117_{10} will be converted to binary by this program.

    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 9-39. 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 9-39

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 9-40.

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.

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→ 01 Digit	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 00 Digit	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 64 Now	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→ 01 Digit	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→ 00 Digit	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→ 6E Result	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→ 75 Result	0	Store the new binary result (75_{16}).
25	0027	3E	75	0	Halt.

Figure 9-40

Single-stepping through the BCD-to-binary conversion program.

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.

Procedure (Continued)

28. Load the program listed in Figure 9-41 into the Trainer. This program should convert the binary number 0111 0101_2 (75_{16}) into it's 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 9-42. Notice that we're monitoring the carry (C) flag because the program uses the BCS (Branch if Carry Set) instruction.

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}.

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
002D	—	—	Units digit

Figure 9-41

A program with an incorrect relative address.

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 9-42
Single-Stepping through the binary-to-BCD conversion program.

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 9-43 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 9-43, 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 BCS instruction produced the desired branch.

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	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	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 9-43
Locating the incorrect relative address.

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}.

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.

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	
0010	93	93	Least significant byte	augend
0011	1B	1B	Most significant byte	
0012	—	—	Least significant byte	sum
0013	—	—	Most significant byte	

Figure 9-44
Program for multiple-precision addition.

Procedure

1. Turn on the Trainer and press the RESET key.

2. Load the program listed in Figure 9-44 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 $COEA_{16}$ by this program. Of course, the program can add any numbers that are 16_{10} bits or less.

3. Change the program counter to 0000 and single-step through the program, recording the information in the chart of Figure 9-45. Notice that we are monitoring the carry (C) flag.

4. Examine memory location 0012_{16} and 0013_{16} and record the sum below.

SUM _ _ _ _

STEP	PROGRAM COUNTER	OPCODE	ACCA	C FLAG	COMMENTS
1					
2					
3					
4					
5					
6					
7					

Figure 9-45

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 9-46 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 9-46
Program adds two numbers and produces carry.

8. Enter the program listed in Figure 9-47 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 }
0010	93	93	Least significant byte } augend
0011	1B	1B	Most significant byte }
0012	—	—	Least significant byte } sum
0013	—	—	Most significant byte }

Figure 9-47
Mutiple-precision addition program with instruction at address 0003_{16} changed.

9. Set the program counter to 0000 and single-step through the program, recording the information in the chart of Figure 9-48.

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 9-48

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 compiled should be similar to the chart in Figure 9-49. 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		
COEA$_{16}$	=	1100	0000	1110	1010
1B93$_{16}$	=	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 9-49

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 9-48. Your chart should be similar to the one shown in Figure 9-50.

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.

STEP	PROGRAM COUNTER	OPCODE	ACCA	C FLAG	COMMENTS
1	0001	96	Random	1	Load the accumulator with the LSB of Addend (EA$_{16}$).
2	0003	99	EA	1	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 9-50

Single-stepping through the multiple-precision addition program where both add instructions are ADC.

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 9-51.

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 ______ ______ ______ ______

		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		
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 9-51
Instructions.

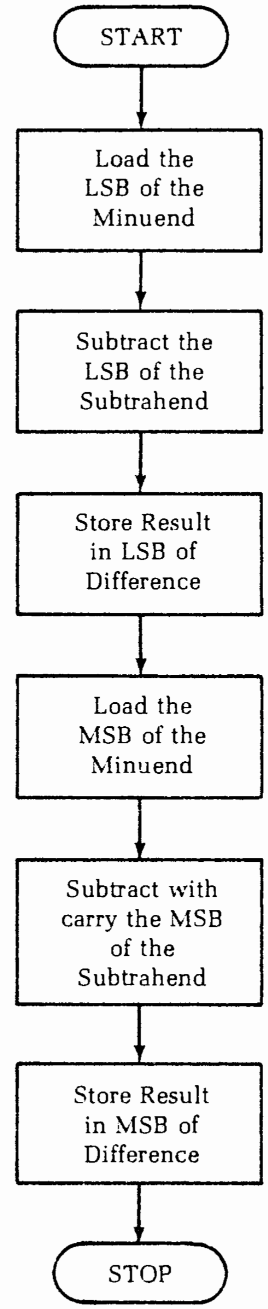

START
Load the LSB of the Minuend
Subtract the LSB of the Subtrahend
Store Result in LSB of Difference
Load the MSB of the Minuend
Subtract with carry the MSB of the Subtrahend
Store Result in MSB of Difference
STOP

Figure 9-52
Flow chart for multiple-precision subtraction.

Discussion

If you made a flow chart of the problem, your flow chart probably looks like the one shown in Figure 9-52. Your program should be similar to the solution shown in Figure 9-53. 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.

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 9-51 and write a program that uses the ASLA instruction to multiply any 4_{10}-bit number by 16_{10}.

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
000F	81	81	Least significant byte Subtrahend
0010	75	75	Most significant byte
0011	—	—	Least significant byte Difference
0012	—	—	Most significant byte

Figure 9-53

Program for multiple-precision subtraction.

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$_{16}$ × 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$_{10}$ × 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 9-54. Notice that it 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 4-bit multiplicand
0001	09	09	
0002	48	ASLA	Shift the accumulator four places to the left multiplying the multiplicand by 16_{10}.
0003	48	ASLA	
0004	48	ASLA	
0005	48	ASLA	
0006	97	STA	Store the product at this location
0007	0A	0A	
0008	3E	HLT	Halt
0009	0F	0F	4-bit multiplicand
000A	—	—	Product

Figure 9-54
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 9-55 into the Trainer. The unpacked BCD numbers are 09_{10} and 03_{10}.

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	

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 9-55
Program to pack two BCD digits into a single byte.

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 9-56 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	
000F	54	54	Most significant byte	BCD Augend
0010	82	82	Least significant byte	
0011	—		Most significant byte	BCD Sum
0012	—		Least significant byte	

Figure 9-56
Incorrect program for multiple-precision addition of BCD numbers.

24. Now load the corrected multiple-precision BCD addition program listed in Figure 9-57 into your Trainer. Notice that the only changes between this program and the previous program are the additions of the NOP instruction 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	
0012	54	54	Most significant byte	BCD Augend
0013	82	82	Least significant byte	
0014	—	—	Most significant byte	BCD Sum
0015	—	—	Least significant byte	

Figure 9-57
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.

NOTES:

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.

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

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 this course. 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 9-58. 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.

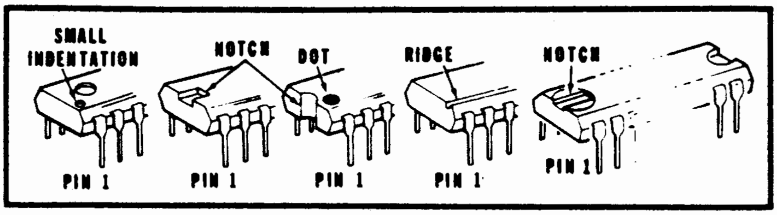


Figure 9-58

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 modifications 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 9-59 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 9-59

Adding a list of numbers using extended addressing.

Procedure (Continued)

13. Figure 9-60 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.

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 9-61.

15. Compare the programs of Figures 9-59 and 9-60. 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 9-59 __________. Compare this with the number of machine cycles required for the program in Figure 9-60.

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.

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 9-60
Adding the list of numbers using indexed addressing.

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 9-61
Record values here.

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 9-62. 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.

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 9-62

Program for clearing addresses 0120_{16} through $01A0_{16}$.

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 9-63 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.

Procedure (Continued)

21. Enter the data shown in Figure 9-63 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.

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 9-63
Three tables.

Discussion

The solution to the problem is shown in Figure 9-64.

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 9-64
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.

$$
\begin{array}{r c c c r}
1101_2 & \leftarrow & \text{Multiplicand} & \rightarrow & 13_{10} \\
\underline{1011_2} & \leftarrow & \text{Multiplier} & \rightarrow & \underline{11_{10}} \\
1101 & & & & 13 \\
1101 & & & & \underline{13} \\
0000 & & & & 143_{10} \\
\underline{1101} & & & & \\
10001111_2 & \leftarrow & \text{Product} & \nearrow &
\end{array}
$$

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 9-65. 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.

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.

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 9-65
Multiplying by shifting and adding.

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 9-66 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.

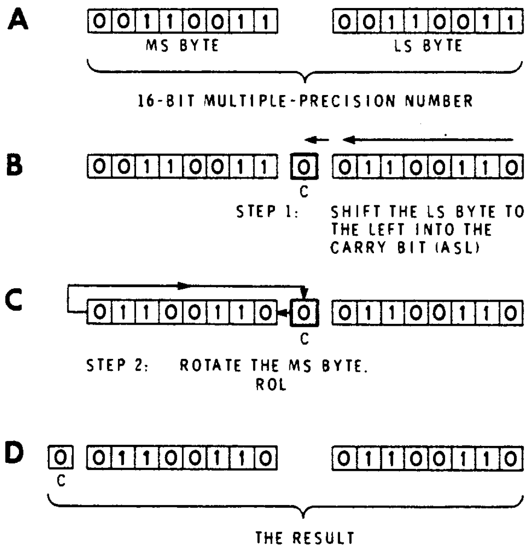


Figure 9-66
Shifting a multiple-precision number.

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 9-66.

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 9-67. Compare your program with this one. If you were unsuccessful in writing a program, study Figure 9-67 very carefully until you understand the procedures involved.

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.

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 9-67

Program for multiplying a double-precision multiplicand by an 8-bit multiplier.

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 9-68. 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

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 9-68
Square root subroutine

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.

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 9-69 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 9-69

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 9-70 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 __________

2. Load the program into the Trainer and verify that you loaded it properly.

3. Execute the program using the DO command.

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 9-70
This routine sets the contents of all MPU registers to known values.

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 9-71 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 9-71
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 9-71. 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 32^{nd} 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 (LEDs) arranged as shown in Figure 9-72. 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.

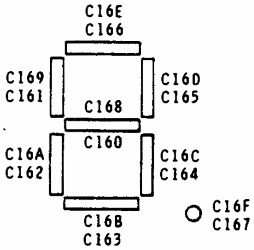


Figure 9-72
Addresses of the various segments in the left LED display.

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 9-73A shows how

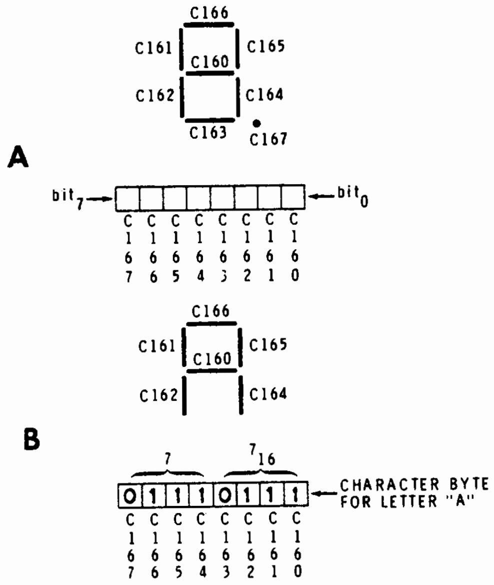

A
C166
C161
C160
C165
C162
C164
C163
C167
bit7
bit0
C167 C166 C165 C164 C163 C162 C161 C160
B
C166
C161
C160
C165
C162
C164
7
7 16
0 1 1 1 0 1 1 1
CHARACTER BYTE FOR LETTER "A"
C167 C166 C165 C164 C163 C162 C161 C160

Figure 9-73

Assigning the bits of the character byte.

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 9-73B 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.

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 9-74.

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? ________.

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
0006	00	00	address indicated by the index register.
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 9-74
Program for lighting a display.

11. Change the character byte to 79_{16}. Execute the program. What character is displayed? ______.

12. Refer to Figure 9-75. 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.

13. Now return to the program shown in Figure 9-74. 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.

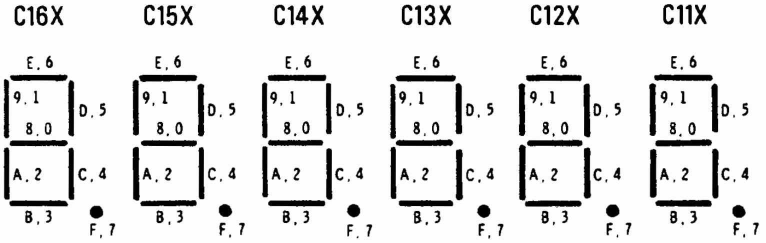


Figure 9-75
Addresses of the various display segments.

Procedure

16. Load the program shown in Figure 9-76. Verify that you loaded it properly.

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.

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 9-76
This program uses the OUTCH subroutine in the monitor program to display a message.

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 9-77 into the Trainer and verify that you loaded it properly.

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	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 9-77
The OUTSTR subroutine in the monitor is used to display a message.

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 9-78
OUTSTR is called twice.

22. The program shown in Figure 9-78 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. You can achieve a delay by loading the index register with FFFF and decrementing it to 0000. You can achieve an additional delay by using either accumulator in conjunction with the index register. 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 9-79. Verify that you loaded it properly.

27. Execute the program using the DO command. What message is displayed? ______________________.

28. Change the number in address $003C_{16}$, $003E_{16}$, and $003F_{16}$.

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."

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	BD	JSR	Call Delay Subroutine
0001	00	00	
0002	3B	3B	
0003	BD	JSR	Call OUTSTJ
0004	FD	FD	
0005	8C	8C	
0006	37	37	H
0007	F	4F	E
0008	0E	0E	L
0009	0E	0E	L
000A	FE	FE	O.
000B	BD	JSR	Call Delay Subroutine
000C	00	00	
000D	3B	3B	
000E	BD	JSR	Call OUTSTJ again
000F	FD	FD	
0010	8C	8C	
0011	4E	4E	C
0012	77	77	A
0013	76	76	N
0014	00	00	blank
0015	B0	B0	I.
0016	BD	JSR	Call Delay Subroutine
0017	00	00	
0018	3B	3B	
0019	BD	JSR	Call OUTSTJ again
001A	FD	FD	
001B	8C	8C	
001C	37	37	H
001D	4F	4F	E
001E	0E	0E	L
001F	67	67	P
0020	80	80	•
0021	BD	JSR	Call Delay Subroutine
0022	00	00	
0023	3B	3B	
0024	BD	JSR	Call SUTSTJ again
0025	FD	FD	
0026	8C	8C	
0027	3B	3B	Y
0028	7E	7E	O
0029	3E	3E	U
002A	00	00	blank
002B	80	80	•
002C	BD	JSR	Call Delay Subroutine
002D	00	00	
002E	3B	3B	
002F	BD	JSR	Call OUTSTJ again
0030	FD	FD	
0031	8C	8C	
0032	00	00	blank
0033	00	00	blank
0034	00	00	blank
0035	00	00	blank
0036	00	00	blank
0037	80	80	•
0038	7E	JMP	Do it all again

cont'd.

cont'd.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0039	00	00	
003A	00	00	
003B	86	LDA A#	Delay Subroutine
003C	02	02	
003D	CE	LDX#	
003E	00	00	
003F	00	00	
0040	09	DEX	
0041	26	BNE	
0042	FD	FD	
0043	4A	DECA	
0044	26	BNE	
0045	F7	F7	
0046	39	RTS	

Figure 9-79
This program makes extensive use of the subroutine call.

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.

Procedure (Continued)

31. Load the program shown in Figure 9-80. Verify that you loaded it properly.

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	

Figure 9-80
Using the OUTBYT and REDIS subroutines.

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 though 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 9-81. 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	Fast Clock One-Second Delay Subroutine
0011	86	LDAA#	
0012	02	02	
0013	CE	LDX#	
0014	F3	F3	
0015	80	80	
0016	09	DEX	
0017	26	BNE	
0018	FD	FD	
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 9-81
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 9-82. 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 9-82
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 9-83. 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 9-84. 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)	
0003	C5 (BD)	C5 (JSR)	Slow Clock (call Fast Clock
0004	00 (00)	00 (00)	One-Second One-Second
0005	09 (2F)	DEX (2F)	delay. Delay)
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.

002F	(86)	(I.DAA#)	Fast Clock One-Second Delay Subroutine
0030	(02)	(02)	
0031	(CE)	(I.DX#)	
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 9-83
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 hours.	
0018	17	17		
0019	8D	BSR	Call display subroutine to display minutes.	
001A	15	15		
001B	8D	BSR	Call display subroutine to display seconds.	
001C	13	13		
001D	20	BRA	Do it all again.	
001E	E4	E4		
001F	A6	LDAA, X	Load seconds (or minutes or hours).	Increment Subroutine
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.	
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)	Display Subroutine
0031	A6	LDAA, X	Load hours (or minutes or seconds)	
0032	00	00		
0033	7E	JSR	Display hours (or minutes or seconds)	
0034	FE	FE		
0035	20	20		
0036	39	RTS		

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 9-84
Twelve-hour clock program