HEATHKIT
CONTINUING
EDUCATION

Individual Learning Program

MICROPROCESSORS

Unit 4

INTRODUCTION TO PROGRAMMING

EE-3401

HEATH COMPANY
BENTON HARBOR, MICHIGAN 49022

Copyright © 1977
Heath Company
All Rights Reserved
Printed in the United States of America

CONTENTS

Introduction 4-3

Unit Objectives 4-4

Unit Activity Guide 4-5

Branching 4-6

Conditional Branching 4-20

Algorithms 4-29

Additional Instructions 4-46

Experiments 4-58

Unit Examination 4-59

Examination Answers 4-61

Unit 4
INTRODUCTION TO PROGRAMMING

INTRODUCTION

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

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

UNIT OBJECTIVES

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

1. Explain the difference between machine language, assembly language, interpretive language, and compiler language.

2. Define assembler, compiler, interpreter, object program, source program, BASIC, FORTRAN, and COBOL.

3. Draw the symbols used in flow charting and explain the purpose of each.

4. Develop flow charts that illustrate step-by-step procedures for solving simple problems.

5. Explain the purpose of conditional and unconditional branching.

6. Using the block diagram of the hypothetical microprocessor, trace the data flow during the execution of a branch instruction.

7. Compute the proper relative address for branching forward or backward from one point to another in a program.

8. Explain the purpose of the carry, negative, zero, and overflow flags. Give an example of a situation that can cause each to be set and another example that will cause each to clear. List eight instructions that test one of these flags.

9. Write programs that can: multiply by repeated addition; divide by repeated subtraction; convert binary to BCD; convert BCD to binary; add multiple-precision numbers; subtract multiple-precision numbers; add BCD numbers.

UNIT ACTIVITY GUIDE

	Completion Time
☐ Play Cassette Tape Section "Introduction to Programming."	________
☐ Read Section on Branching.	________
☐ Complete Self-Test Review Questions 1-9.	________
☐ Read Section on Conditional Branching.	________
☐ Complete Self-Test Review Questions 10-19.	________
☐ Read Section on Algorithms.	________
☐ Complete Self-Test Review Questions 20-29.	________
☐ Read Section on Additional Instructions.	________
☐ Complete Self-Test Review Questions 30-37.	________
☐ Perform Programming Experiments 5 and 6.	________
☐ Complete Unit Examination.	________
☐ Check Examination Answers.	________

BRANCHING

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

The real power of the microprocessor comes from its ability to execute a section of a program over and over again. In an earlier program we saw that two numbers could be multiplied by repeated addition. As long as the numbers are very small and we know the value of the two numbers, we can write a "straight line" program to multiply the numbers. For example, 9 could be multiplied by 4 with the following program:

Address	**Instruction/Data**	**Comments**
00	LDA 05	Load direct
01	ADD 05	Add direct
02	ADD 05	Add direct
03	ADD 05	Add direct
04	HLT	
05	09	

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

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

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

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

Relative Addressing

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

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

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

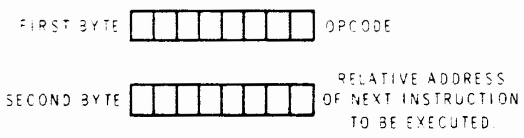


Figure 4-1
Format of the branch instruction.

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

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

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

BRANCH 30_{16}

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

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

Executing a Branch Instruction

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

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

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

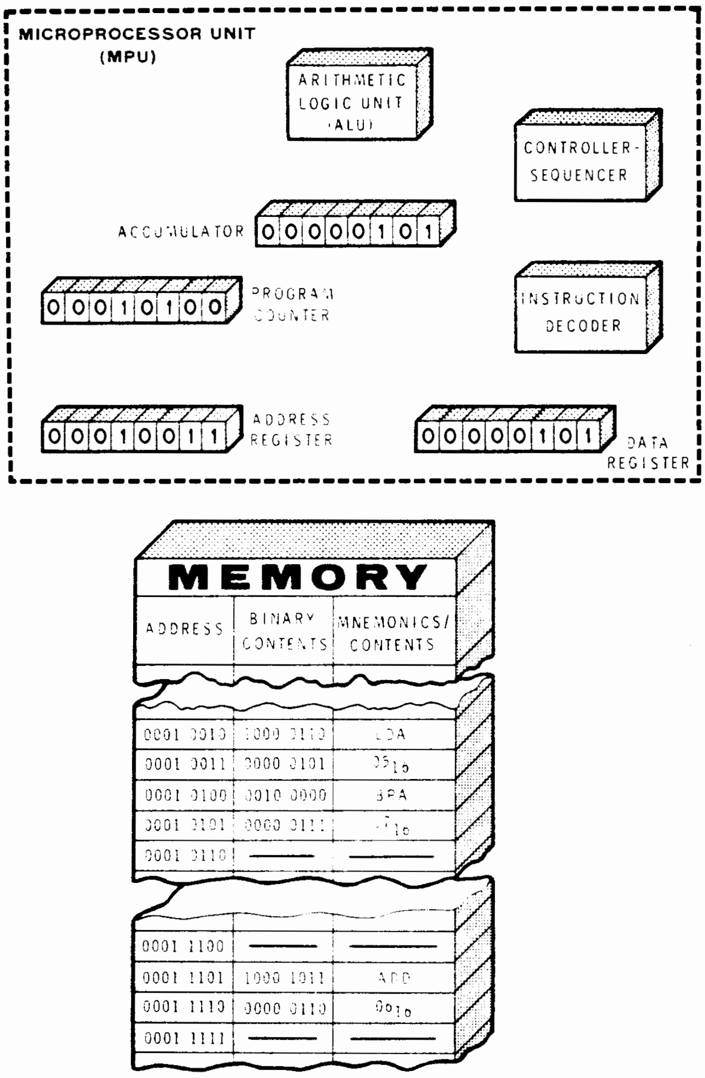


Figure 4-2
Status of the MPU registers after executing the LDA 05 instruction.

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

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

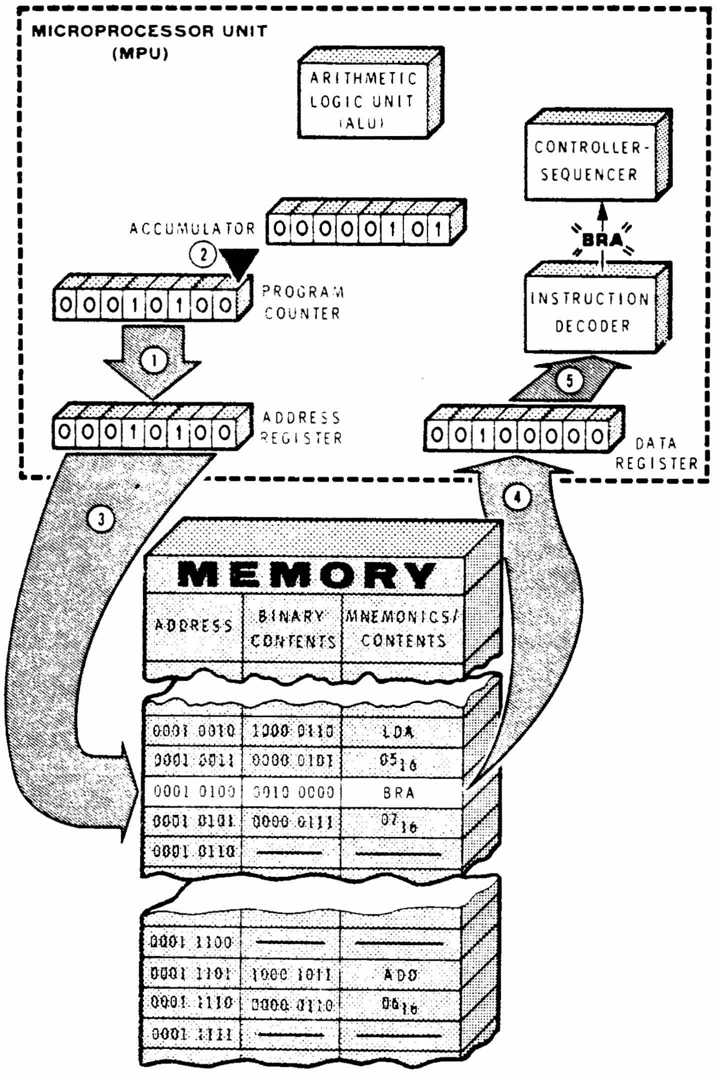

MICROPROCESSOR UNIT (MPU)
ARITHMETIC LOGIC UNIT (ALU)
CONTROLLER-SEQUENCER
ACCUMULATOR 0 0 0 0 0 1 0 1
BRA
PROGRAM COUNTER 0 0 0 1 0 1 0 0
INSTRUCTION DECODER
ADDRESS REGISTER 0 0 0 1 0 1 0 0
DATA REGISTER 0 0 1 0 0 0 0 0
1
2
3
4
5
MEMORY
ADDRESS
BINARY CONTENTS
MNEMONICS/CONTENTS
0001 0010 | 1000 0110 | LDA
0001 0011 | 0000 0101 | 05₁₆
0001 0100 | 0010 0000 | BRA
0001 0101 | 0000 0111 | 07₁₆
0001 0110
0001 1100
0001 1101 | 1000 1011 | ADD
0001 1110 | 0000 0110 | 06₁₆
0001 1111

Figure 4-3
Fetching the BRA instruction.

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

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

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

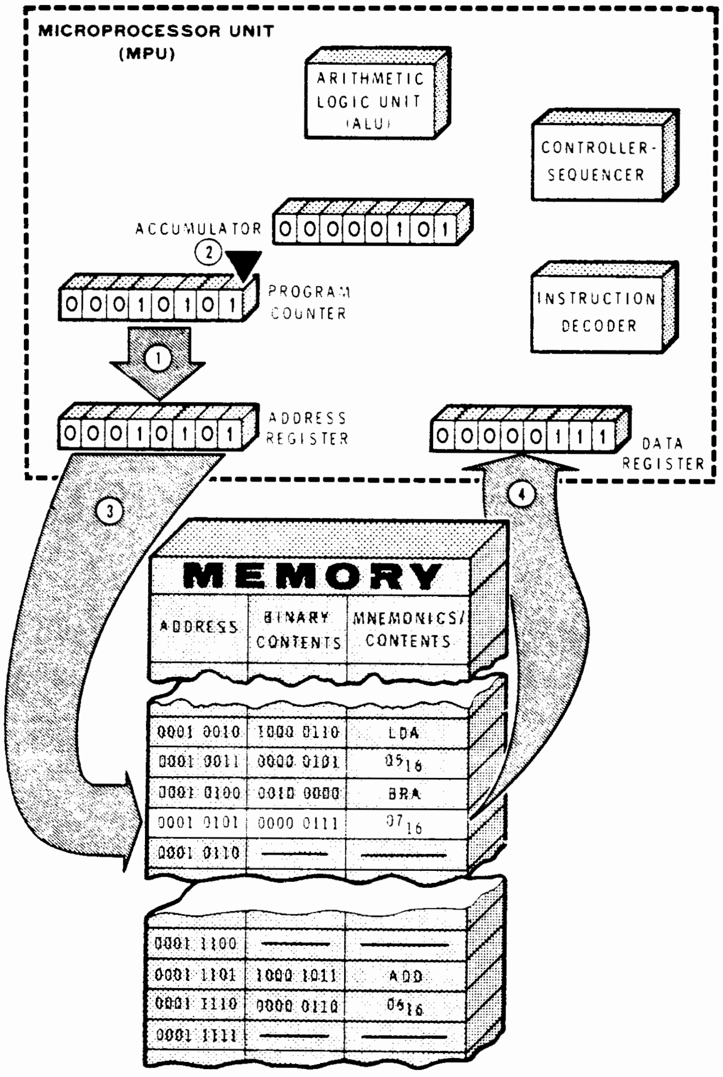


Figure 4-4
Fetching the relative address.

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

```
0001  0110    program count
0000  0111    relative address
----------
0001  1101    new program count
```

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

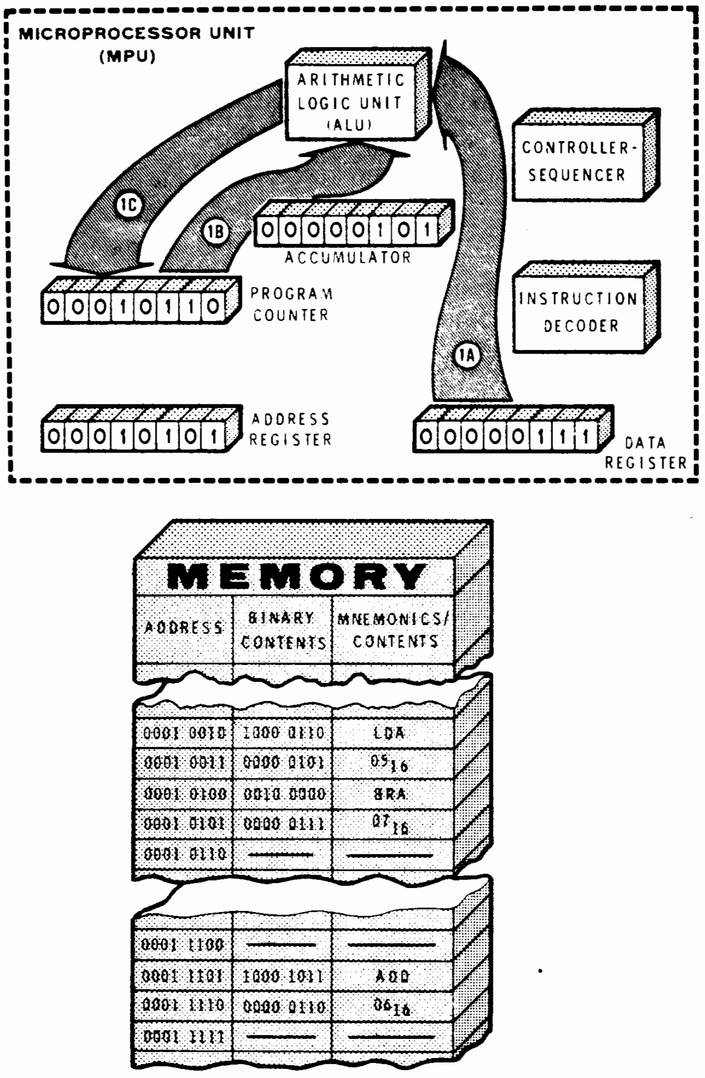


Figure 4-5
Computing the address of the
next instruction.

Branching Forward

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

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

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

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

A

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

Program will branch to here

B

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

Originating Address

Destination Address

We wish to Branch to here

Figure 4-6
Branching forward.

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

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

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

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

$0010\ 0100_2$	24_{16}	Destination address
$-0001\ 1010_2$	$1A_{16}$	Originating address
$0000\ 1010_2$	$0A_{16}$	Relative address

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

Branching Backward

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

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

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

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

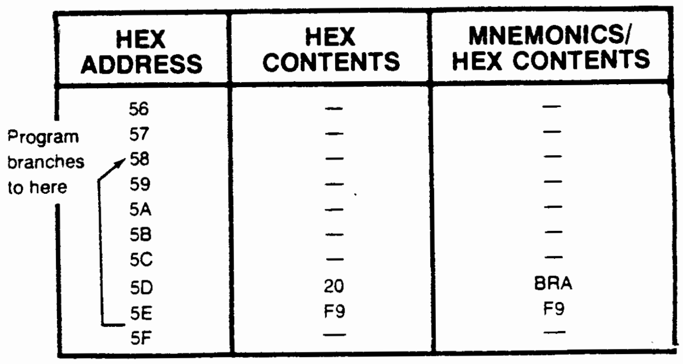
```
                     0101 1111   ←   Old program count
                   + 1111 1001   ←   Relative Address
                   ───────────
                   1 0101 1000   ←   New program count
                   ↑
Carry is ignored───┘
```

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

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

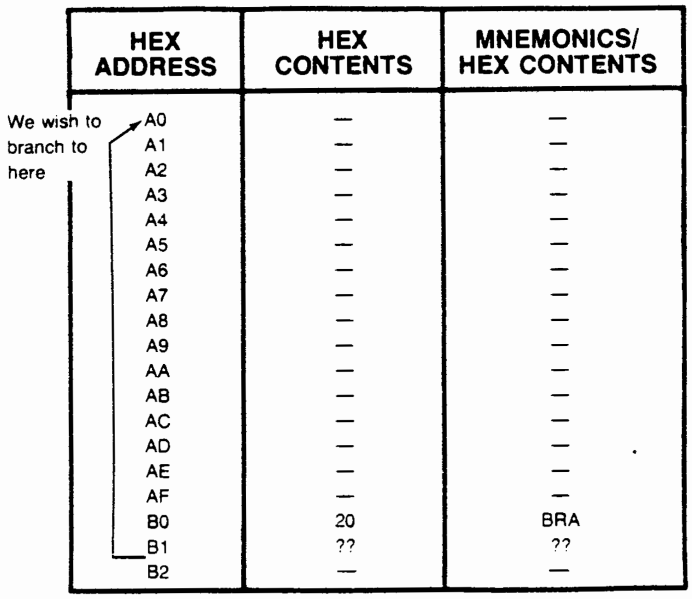
1. Subtract the destination address from the originating address.

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



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

A



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

B

Figure 4-7
Branching backwards.

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

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

Next you compute the relative address by taking the two's complement of the difference. The two's complement of $0001\ 0010_2$ is $1110\ 1110_2$. In hexadecimal this is EE_{16}. Thus, the required relative address is EE_{16}.

Self-Test Review

1. What addressing mode is used by the branch instruction?

2. What does the second byte of a branch instruction indicate?

3. What happens in the MPU during the execution of the branch instruction?

4. What type of relative address causes a branch forward?

5. What type of relative address causes a branch backwards?

6. What is the maximum number of memory locations that can be branched over during a forward branch?

7. What is the maximum number of memory locations that can be branched over during a backward branch?

8. The opcode for the branch instruction is at address 20_{16}. The relative address is 06_{16}. After the branch instruction is executed, from what address will the next opcode be fetched?

9. The opcode for the branch instruction is at address 20_{16}. The relative address is $F1_{16}$. After the branch instruction is executed, from what address will the next opcode be fetched?

Answers

1. Relative addressing.

2. The second byte of the branch instruction is the relative address. This number is added to the contents of the program counter to form the absolute address.

3. The relative address is retrieved from memory and is added to the program count. The new program count goes into the program counter.

4. A positive two's complement number.

5. A negative two's complement number.

6. $0111\ 1111_2$ or $+127_{10}$.

7. $1000\ 0000_2$ or -128_{10}.

8. 28_{16}. Recall that during the execution of the branch instruction, the program counter will be incremented twice to 22_{16}. Thus, when the relative address (06_{16}) is added, the new address becomes 28_{16}.

9. As in answer 8, the program counter is automatically advanced to 22_{16} ($0010\ \ 0010_2$) before the relative address is added. $F1_{16}$ is equal to $1111\ \ 0001_2$. When this is added to the program count, the new address becomes

$$\begin{array}{rrrl} & 0010 & 0010_2 & \text{Old program count} \\ & 1111 & 0001_2 & \text{Relative address} \\ \hline 1 & 0001 & 0011_2 & \text{New program count} \\ \uparrow & & & \\ \text{Ignore carry} & & & \end{array}$$

Thus, the next opcode will be fetched from address 13_{16}.

CONDITIONAL BRANCHING

The branch instruction allows the MPU to jump forward over a block of data or over a portion of a program. It also allows the MPU to jump backwards so a group of instructions can be repeated.

Until now we have been discussing the **unconditional** branch instruction. This type of instruction always results in a program branch. For this reason, it is called the **BR**anch **A**lways instruction. Its mnemonic is BRA.

There are other types of branch instructions that greatly expand the versatility of the MPU. These are called **conditional** branch instructions. Unlike BRA, these instructions cause a branch only if some specified condition is met.

A good example of a conditional branch instruction is the Branch If Minus (BMI). This instruction may or may not initiate a branch operation, depending on the result of some previous arithmetic or logic operation. This instruction might be placed after a subtract instruction. If the result of the subtraction is a negative number, the branch would be implemented. Otherwise, the MPU would continue to fetch and execute instructions in numerical order. An example may help to illustrate this.

Figure 4-8 shows part of a program that uses the branch if minus (BMI) instruction. Let's start with the instruction at address 95_{16}. This instruction causes the contents of location $B0_{16}$ to be loaded into the accumulator. Next, the SUB instruction subtracts the contents of location $B1_{16}$ from the number in the accumulator. The next instruction (BMI) examines the result of the subtraction. If the result was a minus number, the program will branch over the next three bytes. That is, the next instruction to be executed is the STA instruction at address $9E_{16}$. Thus, the resulting number in the accumulator is stored in location $B3_{16}$ and the MPU halts.

If the result of the subtraction is not minus, the BMI instruction has no effect. That is, the BMI instruction is fetched and executed but no branch occurs because the specified condition is not met. In this case, the next instruction to be executed is the STA instruction at address $9B_{16}$. Thus, the result of the subtraction will be stored in location $B2_{16}$.

HEX ADDRESS	HEX CONTENTS	MNEMONIC/HEX CONTENTS	COMMENTS
95	96	LDA	Load accumulator direct
96	B0	B0	with contents of this address.
97	90	SUB	Subtract
98	B1	B1	the contents of this address.
99	2B	BMI	If result is minus
9A	03	03	branch this far.
9B	97	STA	If result is not minus, store
9C	B2	B2	at this address;
9D	3E	HLT	then halt.
9E	97	STA	If result is minus, store
9F	B3	B3	it at this address;
A0	3E	HLT	then halt.

Figure 4-8
This program uses the BMI instruction
to make a simple decision.

Notice that the program flow can take one of two paths, depending on the result of the subtraction. The BMI instruction gives the MPU this capability. The conditional branch instructions are sometimes called "decision making instructions." The reason for this becomes obvious if you consider the implications of our sample program. Here the MPU decides if the number at address $B1_{16}$ is larger than that at $B0_{16}$. The program path is determined by the outcome of this decision. If the number in $B1_{16}$ is larger, the result of the subtraction is a negative number. In this case, the result is stored in location $B3_{16}$. Otherwise, the resulting difference is stored in location $B2_{16}$.

Virtually all programs must make some type of decision. Some frequently encountered decisions are:

"Which of two numbers is larger?"

"Does this byte represent a letter of the alphabet or a numeral?"

"Are these two numbers equal?"

"Is this an even number?"

"Has the program loop been repeated the proper number of times?"

Conditional branch instructions are used in making all of these decisions.

Condition Codes

As the name implies, a conditional branch instruction causes a program branch only if some specified condition is met. Some commonly monitored conditions are:

1. Did a previous operation result in a negative number in the accumulator?
2. Did a previous operation result in zero in the accumulator?
3. Did a previous operation result in a carry from bit 7 of the accumulator?

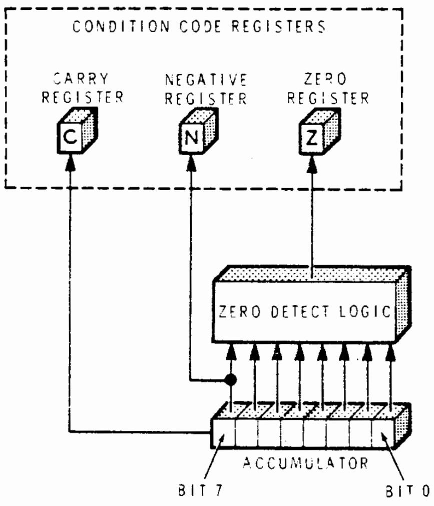


Figure 4-9
Condition code registers monitor the operations in the accumulator.

To keep track of these conditions, most microprocessors have a group of single bit registers called condition code registers. Three of these registers are shown in Figure 4-9. They are the Negative (N) Register, the Zero (Z) Register, and the Carry (C) Register.

Negative (N) Register Recall that negative numbers are expressed in two's complement form. Using this system, the most significant bit determines whether or not the number is negative. In an 8-bit byte, bit 7 is a 1 if the two's complement number is negative. Thus, the N register monitors bit 7 of the accumulator. Immediately after an operation that involves the accumulator, the N register looks at bit 7 to see if the number is negative. If so, the N register is set to 1. If the number in the accumulator is not negative, the N register is reset to 0.

Most operations that involve the accumulator affect the N register in this way — **but not all**. In a later unit we will point out how this register is affected by each instruction. In this unit, we will assume that the N register is affected as outlined above any time a number is added to, subtracted from, loaded into, or stored from the accumulator.

Another name for a condition code is a flag. Thus, the N register is sometimes called the N flag or the negative flag.

Zero (Z) Register This register monitors the accumulator looking for all zeros. Immediately after an operation that involves the accumulator, the zero-detect circuit looks at the resulting number. If all 8 bits are 0, the Z register is set to 1. Otherwise, the Z register is reset to 0. Most operations that involve the accumulator affect the Z register in this way.

Carry (C) Register The C register acts somewhat like an extension of the accumulator. You have seen that when two unsigned 8-bit numbers are added, the sum is frequently a 9-bit number. For example:

```
           1001  0010     8-bit addend
       +   1100  0110     8-bit augend
       ------------------ ------------
         1 0101  1000     9-bit sum
         ↑
  carry ─┘
```

Since the accumulator is an 8-bit register, the sum will not fit. The most significant bit (the carry) would be lost if you did not have another 1-bit register to hold it. This is the purpose of the C register. Any operation that causes a carry out of bit 7 will set the carry register to 1. Arithmetic operations that do not result in a carry will reset this register to 0.

The carry register is also used to keep track of "borrows" during subtract operations. If a subtraction requires a borrow for bit 7, the carry flag will also be set. For example, suppose you subtract an unsigned, binary number from a smaller unsigned binary number. The result will, of course, be a negative number. Moreover, bit 7 will have to "borrow" a bit to complete the subtraction. As a simple example, let's subtract 2 from 1. The subtraction looks like this

```
Borrow →  1
             0000  0001     Minuend
          -  0000  0010     Subtrahend
          -------------     ----------
             1111  1111     Difference
```

The carry bit is set to 1 to indicate that a borrow operation occurred. Many subtraction operations do not require borrows. In these cases, the carry bit is reset to 0 to indicate that no borrow occurred.

Notice that the carry code can have different meanings, depending on the operation involved. That is, a 1 can mean either that a carry occurred or that a borrow occurred. The precise meaning of the 1 depends on whether the operation was an addition or a subtraction. We will discuss some additional aspects of the carry register in a later unit.

Overflow (V) Register The final condition code that is to be considered in this unit keeps track of two's complement overflow. Figure 4-10 shows how this register is connected in the MPU. A special circuit detects an overflow condition by monitoring bit 7 of the ALU's input and output lines. This circuit sets the V flag when an overflow occurs but clears it if no overflow occurs.

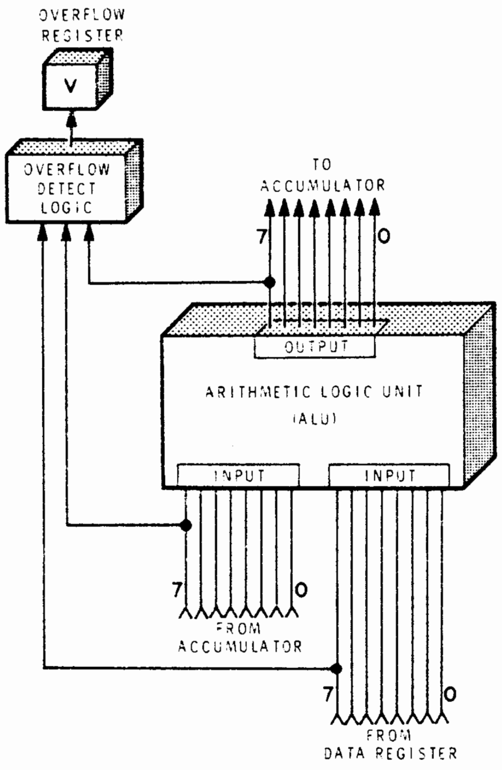


Figure 4-10
The overflow register monitors bit 7 of the ALU's input and output lines.

Let's see what is meant by two's complement overflow. Recall that the ALU adds numbers as if they were unsigned binary numbers. Even so, it can handle signed binary numbers if the proper bit patterns represent the negative numbers. This is the reason that the two's complement method of representing signed numbers has become so popular. A disadvantage of this system is that the magnitude of the number must be represented by 7 bits, since the eighth bit is used as the sign. Remember that a 1 in the MSB defines the number as negative.

Unfortunately, if two signed numbers are added and their sum exceeds 7-bits, the sign bit will be changed. For example, assume that a program adds $+73_{10}$ and $+96_{10}$. The addition looks like this:

$$\begin{array}{rcr}
\underline{0}100\ 1001_2 & & +73_{10} \\
\underline{0}110\ 0000_2 & & +96_{10} \\
\hline
\underline{1}010\ 1001_2 & & 169_{10}
\end{array}$$

The answer is correct if all the binary numbers represent unsigned quantities. However, using two's complement, the underlined bits represent sign bits. Therefore, the answer does **not** represent 169_{10}. Instead, it represents -87_{10}. The reason for this error is that there was an overflow from bit 6 into the sign bit (bit 7). This is one of the situations that the V flag indicates.

When two's complement numbers having the same sign are added, the sum should have the same sign. That is, when two positive numbers are added, the sum should be positive. By the same token, when two negative numbers are added, the sum should be negative. However, an overflow can cause the sign to be reversed. The overflow logic detects this situation and sets the V flag whenever an overflow occurs.

The sign bit can also be upset during subtract operations. For example, when a negative number is subtracted from a positive number, the results should be positive. Remember that subtracting a negative number is tantamount to adding a positive number. However, in certain cases, an overflow can reverse the sign bit. This type of overflow occurs when the signs of the minuend and subtrahend are opposite and the difference has the sign of the subtrahend. This condition also sets the V flag.

Conditional Branch Instructions

The conditional branch instructions available in our hypothetical microprocessor are shown in Figure 4-11. While these are largely self-explanatory, a couple of points should be mentioned.

INSTRUCTION	MNEMONIC	OPCODE	BRANCH IF
Branch If Carry Clear	BCC	24	C=0
Branch If Carry Set	BCS	25	C=1
Branch If Not Equal Zero	BNE	26	Z=0
Branch If Equal Zero	BEQ	27	Z=1
Branch If Plus	BPL	2A	N=0
Branch If Minus	BMI	2B	N=1
Branch If Overflow Clear	BVC	28	V=0
Branch If Overflow Set	BVS	29	V=1

Figure 4-11
Conditional Branch Instructions.

The first instruction, Branch If Carry Clear (BCC), monitors the C register. If the carry register is reset to 0, the branch is implemented. Notice that the words "clear" and "reset" are used interchangeably in this regard. They both mean the register contains a 0.

The branch instructions that monitor the Z register can also be confusing. The Branch If Equal Zero (BEQ) instruction implements a branch when the Z register is set to 1. Recall that the Z register is set to 1 when the number in the accumulator is zero. Thus, you must remember that a 0 in the Z register means that the number in the accumulator is **not** zero.

These conditional branch instructions can be used with other instructions to make a wide range of decisions. They greatly increase the power of the microprocessor. More than any other type of instruction, the conditional branches are responsible for the MPU's "intelligence." In the next section, you will see how these instructions are used.

Self-Test Review

10. What is the difference between an unconditional branch instruction and a conditional branch instruction?

11. What condition is tested by the branch if minus (BMI) instruction?

12. When is the N flag set?

13. When is the Z flag set?

14. During an add operation, the C flag is set. What does this represent?

15. During a subtract operation, the C flag is set. What does this indicate?

16. Often, when two positive 2's complement numbers are added, the sign bit of the answer will indicate a negative sum. This "error" can be spotted by checking which flag?

17. Under what condition will the BEQ instruction cause a branch to occur?

18. Under what condition will the BPL instruction cause a branch to occur?

19. When subtracting unsigned binary numbers, which flag indicates that the difference is a negative number?

Answers

10. An unconditional branch instruction always causes a branch operation to occur. On the other hand, the conditional branch instruction implements a branch operation only if some specified condition is met.

11. The BMI instruction tests the Negative (N) register to see if it is set.

12. Generally speaking, the N flag is set if the previous instruction left a 1 in the MSB of the accumulator.

13. Generally, the Z flag is set if the previous instruction left all zeros in the accumulator.

14. During an add operation, the carry bit is set if there is a carry from bit 7 of the accumulator.

15. During a subtract operation, the carry bit is set if bit 7 had to "borrow" a bit to complete the subtraction.

16. This condition results from a two's complement overflow. Thus, the V flag will be set if this condition occurs.

17. The BEQ instruction causes a branch to occur only if the Z register is set.

18. The BPL instruction causes a branch to occur only if the N register is clear.

19. The carry flag.

ALGORITHMS

An algorithm is a step-by-step procedure for doing a particular job. It generally involves doing a complex task by stringing together a series of simple steps. To illustrate the use of an algorithm, consider the following very simple example.

Multiplying by Repeated Addition

Most microprocessors do not have hardware multiply capabilities. That is, they do not have a multiplication circuit nor a multiply instruction. Nevertheless, the microprocessor can be made to multiply by use of an algorithm. One procedure for doing this was discussed earlier. It involved adding the multiplicand to itself the number of times indicated by the multiplier. In the previous example, this was done by using a separate ADD instruction for each addition. This procedure is unsatisfactory for two reasons. First, it results in excessively long programs. Second, you must know the value of the multiplier so that you know how many ADD instructions to include.

A better approach, although still far from ideal, is to use a program loop that will multiply two numbers by repeated addition. For the time being, assume that the two numbers are both positive and that the product does not exceed 255_{10}. Let's further assume that we use only the instructions which have been discussed up to this point. In fact, we will restrict ourselves to the instructions shown in Figure 4-12.

INSTRUCTION	MNEMONIC	ADDRESSING MODE			
		IMMEDIATE	DIRECT	RELATIVE	INHERENT
Load Accumulator	LDA	86	96		
Clear Accumulator	CLRA				4F
Decrement Accumulator	DECA				4A
Increment Accumulator	INCA				4C
Store Accumulator	STA		97		
Add	ADD	8B	9B		
Subtract	SUB	80	90		
Branch Always	BRA			20	
Branch If Carry Set	BCS			25	
Branch if Equal Zero	BEQ			27	
Branch if Minus	BMI			2B	
Halt	HLT				3E

Figure 4-12
Instructions to be used.

The algorithm for multiplying by repeated addition is quite simple. To multiply A times B, you merely add A to a specific location B times. For example, to multiply 5 times 3, you clear a location and then add 5. You continue the addition until 5 has been added 3 times. The number in the affected location will then be 15_{10} which is the product of 5 times 3.

The success of this operation depends on the microprocessor knowing when to stop. It must add 5 three times, but only three times. One way to keep track of the number of additions is to decrement the multiplier (3) each time an addition is made. When the multiplier reaches 0, the proper number of multiplications has been carried out.

Figure 4-13 is a flow chart that illustrates the algorithm. In the first two steps, the MPU clears the accumulator and stores the resulting number (0) in the product. This ensures that the product is zero before the first number is added. Next, it loads the multiplier and checks to see if the multiplier is 0. If so, the process is stopped since a multiplier of 0 dictates a product of 0.

In our example, the multiplier is 3; therefore, we exit the decision block via the "no" line. The next step tells us to decrement the multiplier. The new value of the multiplier (2) is stored for future use. Next, the product whose present value is 0 is loaded. Then, the multiplicand (5) is added so that the new value of the product becomes 5. This completes our first pass through the program. Remember that the multiplicand has been added once and that the multiplier has been reduced by one.

Notice that the program loops back to the input of the second block. The product which now has a value of 5 is stored back in memory. The multiplier (which is now 2) is loaded and tested. Because its value is not yet 0, the multiplier is decremented to 1 and stored again. The product (whose value is now 5) is then loaded and the multiplicand is added so that a new value of 10_{10} is obtained.

The program loops again and the new product (10_{10}) is stored. The multiplier (whose value is now 1) is loaded and tested. Because its value is still not 0, it is decremented again. Notice that the value of the multiplier is now 0. This value is stored away, the product (10_{10}) is fetched, and the multiplicand is added once more. The new value of the product becomes 15_{10}.

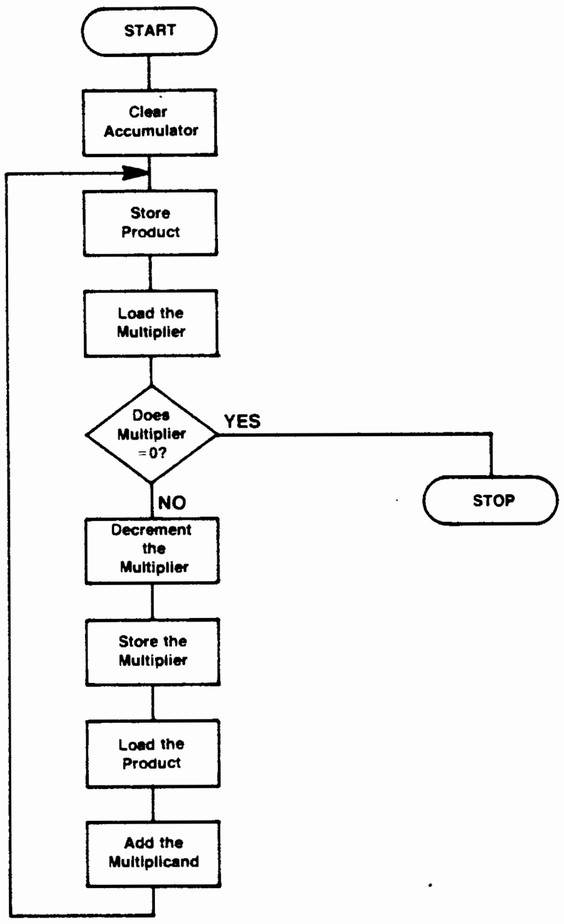


Figure 4-13
Flow chart for multiplying by repeated addition.

The program loops again and the product is stored. The multiplier is loaded and tested. Recall that the value of the multiplier is now 0. Consequently, we exit the decision block via the "yes" line. The program has accomplished its task and it now stops. Notice that the value of the product is 15_{10} which is the proper answer for 5×3.

The next task is to convert the flow chart to a program that the computer can execute. Figure 4-14 shows such a program. Carefully compare this program to the flow chart paying particular attention to the comments column. Work through the program on paper and verify that it will multiply the numbers at addresses 11_{16} and 12_{16}. Although 3 and 5 are used in this example, the program will work for any values of multiplier and multiplicand as long as the product does not exceed 255_{10}.

HEX ADDRESS	HEX CONTENTS	MNEMONIC/HEX CONTENTS	COMMENTS
00	4F	CLRA	Clear the accumulator.
01	97	STA	Store the product
02	13	13	in location 13_{16}.
03	96	LDA	Load the accumulator with the
04	12	12	multiplier from location 12_{16}
05	27	BEQ	If the multiplier is equal to zero,
06	09	09	branch down to the Halt instruction.
07	4A	DECA	Otherwise, decrement the multiplier.
08	97	STA	Store the new value of the
09	12	12	multiplier back in location 12_{16}.
0A	96	LDA	Load the accumulator with the
0B	13	13	product from location 13_{16}.
0C	9B	ADD	Add
0D	11	11	the multiplicand to the product.
0E	20	BRA	Branch back to instruction
0F	F1	F1	in location 01.
10	3E	HLT	Halt.
11	05	05	Multiplicand.
12	03	03	Multiplier.
13	—	—	Product.

Figure 4-14
This program multiplies the numbers at addresses 11_{16} and 12_{16}, and places their product at address 13_{16}.

Dividing by Repeated Subtraction

Another interesting algorithm is one that allows the microprocessor to divide by repeated subtraction. The technique is to keep track of the number of times that the divisor can be subtracted from the dividend. For example, suppose you wish to divide 47_{10} by 15_{10}. The divisor can be subtracted 3 times:

$$\begin{tikzcd}[column sep=large]
\text{First subtraction} & \text{Second Subtraction} & \text{Third Subtraction} \\
\begin{array}{r} 47_{10} \\ -15_{10} \\ \hline 32_{10} \end{array} \arrow[r, bend left] & \begin{array}{r} 32_{10} \\ -15_{10} \\ \hline 17_{10} \end{array} \arrow[r, bend left] & \begin{array}{r} 17_{10} \\ -15_{10} \\ \hline 2_{10} \end{array}
\end{tikzcd}$$

Because three subtractions occurred, the quotient is 3. Also, because 2 was left after the last subtraction, the remainder is 2. We can verify this by long division:

$$\begin{array}{lcrl}
 & & 3_{10} & \leftarrow \text{Quotient} \\
\text{divisor} & \rightarrow & 15_{10} \,\big)\overline{\,47_{10}\,} & \leftarrow \text{Dividend} \\
 & & \cdot 45_{10} & \\
\hline
 & & 2_{10} & \leftarrow \text{Remainder}
\end{array}$$

The microprocessor keeps track of the number of subtractions by incrementing the quotient by one each time a subtraction occurs. Of course, the quotient must be initially set to zero.

The divisor is subtracted from the dividend until any further subtraction would result in a negative number. The MPU can use the BMI instruction to check for a negative result on each loop. The negative result is the indication that the process is finished.

A flow chart for this algorithm is shown in Figure 4-15. The actual program is shown in Figure 4-16. The program is arbitrarily placed in locations 00 through 10_{16}. The dividend (47_{10}) is at address 11_{16} while the divisor (15_{10}) is at address 12_{16}. When executed, the program will produce the quotient at location 13_{16} and the remainder at location 11_{16}.

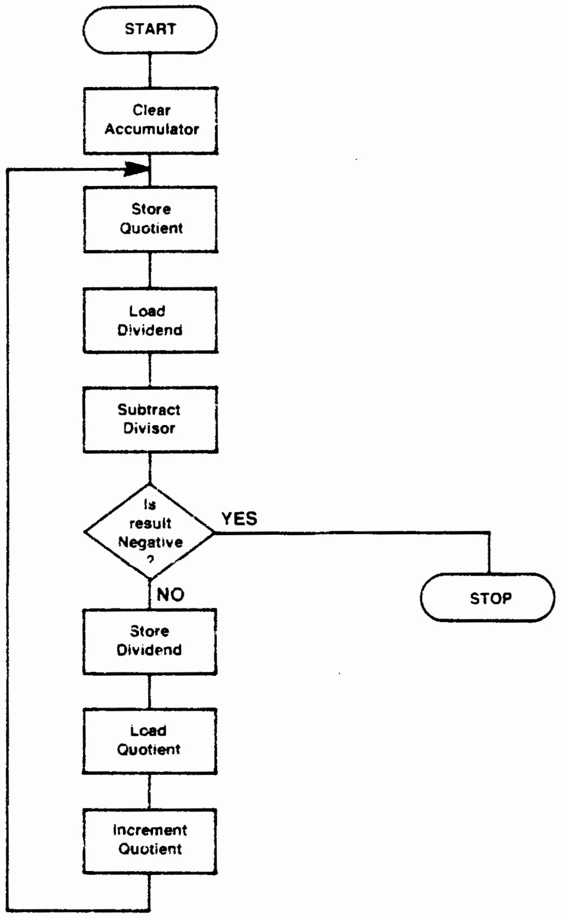


Figure 4-15
Flow chart for dividing by repeated subtraction.

HEX ADDRESS	HEX CONTENTS	MNEMONIC/HEX CONTENTS	COMMENTS
00	4F	CLRA	Clear the accumulator.
01	97	STA	Store in the quotient which
02	13	13	is at address location 13_{16}.
03	96	LDA	Load the accumulator with the
04	11	11	dividend from location 11_{16}.
05	9C	SUB	Subtract the
06	12	12	divisor from the dividend.
07	2B	BMI	If the difference is negative, branch
08	07	07	down to the Halt instruction.
09	97	STA	Otherwise, store the difference
0A	11	11	back in location 11_{16}.
0B	96	LDA	Load the accumulator with the
0C	13	13	quotient.
0D	4C	INCA	Increment the quotient by one.
0E	20	BRA	Branch back to instruction
0F	F1	F1	in location 01.
10	3E	HLT	Halt.
11	2F	2F	Dividend (47_{10}).
12	0F	0F	Divisor (15_{10}).
13	—	—	Quotient.

Figure 4-16
This program divides by repeatedly subtracting the divisor from the dividend.

Refer to the flow chart and the comments column of the program. Before reading further, try running through the program on paper. This will give you a feel for how the computer solves the problem.

Now let's go through the program to see what it does. The first two instructions clear the quotient. Next, the dividend (47_{10}) is loaded into the accumulator and the divisor (15_{10}) is subtracted. The BMI instruction is used to examine the difference (32_{10}). Since the difference is not minus, the branch does not occur. Consequently, the next instruction stores the difference (32_{10}) back in the location from which the dividend came. In effect, the difference becomes the new dividend. Next the quotient (0) is loaded and is incremented to 1. The program then branches back to the instruction in location 01. This instruction stores the quotient (1) back in location 13_{16}.

On the next pass through the program, the new dividend (32_{10}) is loaded and the divisor (15_{10}) is subtracted again. This produces a difference of (17_{10}). Since the difference is not negative, the BMI instruction does not cause a branch. Thus, the difference is stored back in location 11_{16}. The quotient is loaded into the accumulator and is incremented to 2. The BRA instruction causes the program to loop once again. The STA instruction in location 01 stores the quotient (2) back in location 13_{16}.

On the third pass the dividend (17_{10}) is loaded and the divisor (15_{10}) is subtracted a third time. The difference (02) is still not negative so no branch occurs. The difference is stored away; the quotient is loaded and is incremented to 03. Notice that this is the proper final value for the quotient. Therefore, on the next pass, the MPU should be able to break out of the loop.

The quotient is stored back in location 13_{16}. The dividend, which now has a value of 2, is loaded. The divisor (15_{10}) is subtracted, leaving a negative number (-13_{10}) in the accumulator. The BMI instruction recognizes that this is a negative number and implements a branch operation. Notice that the MPU branches forward to the HLT instruction. Thus, the program ends with the quotient set to 3. The remainder is at address 11_{16}. That is, the remainder is what remains of the dividend after the third subtraction.

It may bother you that there were four subtractions and that a negative difference resulted from the last subtraction. However, you will recall that the quotient was incremented only on the first three of these subtractions. Thus, the final quotient is 3. Moreover, the negative difference that resulted during the last subtraction was never stored. Consequently, the remainder was 2 when the program ended.

This program does have some drawbacks. For one thing, neither the dividend nor the divisor can exceed 127_{10}. Also, only positive numbers can be used. Finally, the program gets hung up in an endless loop if the initial value of the divisor is zero. While division by zero is not allowed in mathematics, some provision would be made in a practical program to recognize this eventuality. Since the program is for demonstration purposes, we will live with these shortcomings for the time being.

Converting BCD to Binary

When a microprocessor is used with a terminal such as a teletypewriter, numerals are entered as ASCII characters. For example, the number 237_{10} is entered into memory as three ASCII characters:

Numeral	ASCII Character
2	0011 0010
3	0011 0011
7	0011 0111

Notice that the four least significant bits of the ASCII character represent the BCD value of the corresponding numeral. Thus, we can convert these ASCII characters to BCD numbers simply by eliminating the four most significant bits. This technique was demonstrated in an earlier experiment.

While the microprocessor does have some BCD capability, it is often desirable to convert BCD numbers to binary. The technique for doing this illustrates another useful algorithm.

The BCD representation for 237_{10} is:

0010	←	hundreds BCD digit
0011	←	tens BCD digit
0111	←	units BCD digit

Notice that in this example 0010 represents two hundred, 0011 represents thirty, and 0111 represents seven. Because of this, there is a simple procedure for converting BCD to binary. Starting with an initial value of zero, the MPU adds 100_{10} as many times as indicated by the hundreds digit. It then adds 10_{10} as indicated by the tens digit. Finally, the value of the units digit is added on to the result. The steps involved look like this:

$$\begin{array}{rcrl}
1100100_2 & & 100_{10} & \left.\begin{array}{l} \\ \end{array}\right\} \text{One hundred added 2 times} \\
1100100_2 & & 100_{10} & \\
1010_2 & & 10_{10} & \left.\begin{array}{l} \\ \\ \end{array}\right\} \text{Ten added three times} \\
1010_2 & & 10_{10} & \\
1010_2 & & 10_{10} & \\
\underline{0111_2} & & \underline{7_{10}} & \quad \text{7 units added} \\
11101101_2 & = & 237_{10} &
\end{array}$$

As you can see, this procedure ends with a binary result of 1110 1101, which is the binary representation for 237_{10}.

A flow chart for this procedure is shown in Figure 4-17. Here, the first step is to clear the binary result. We will be adding to this result, so it must start out at zero.

Next the program enters a loop in which it adds 100_{10} to the binary result the number of times indicated by the hundreds digit of the BCD number. The hundreds digit is loaded and tested for zero. If it is not zero, the hundreds digit is decremented and stored back in memory. Then the binary result is loaded and 100_{10} is added. The result is stored away and the loop is repeated. In our example, the hundreds digit was initially 2. Thus, this loop is repeated twice. The binary result will have the value $1100\ \ 1000_2$ (200_{10}) when the hundreds digit is reduced to zero. At that time, the program exits the decision block via the "yes" line and immediately encounters a second loop.

The second loop is exactly like the first except that 10_{10} is added to the binary result each time the tens digit of the BCD number is decremented. Because the tens digit was initially 3, this loop is repeated three times. Ten is added to the binary result three times, bringing the result to $1110\ \ 0110_2$ (230_{10}). The program exits this loop via the "yes" line on the pass after the tens digit is reduced to zero.

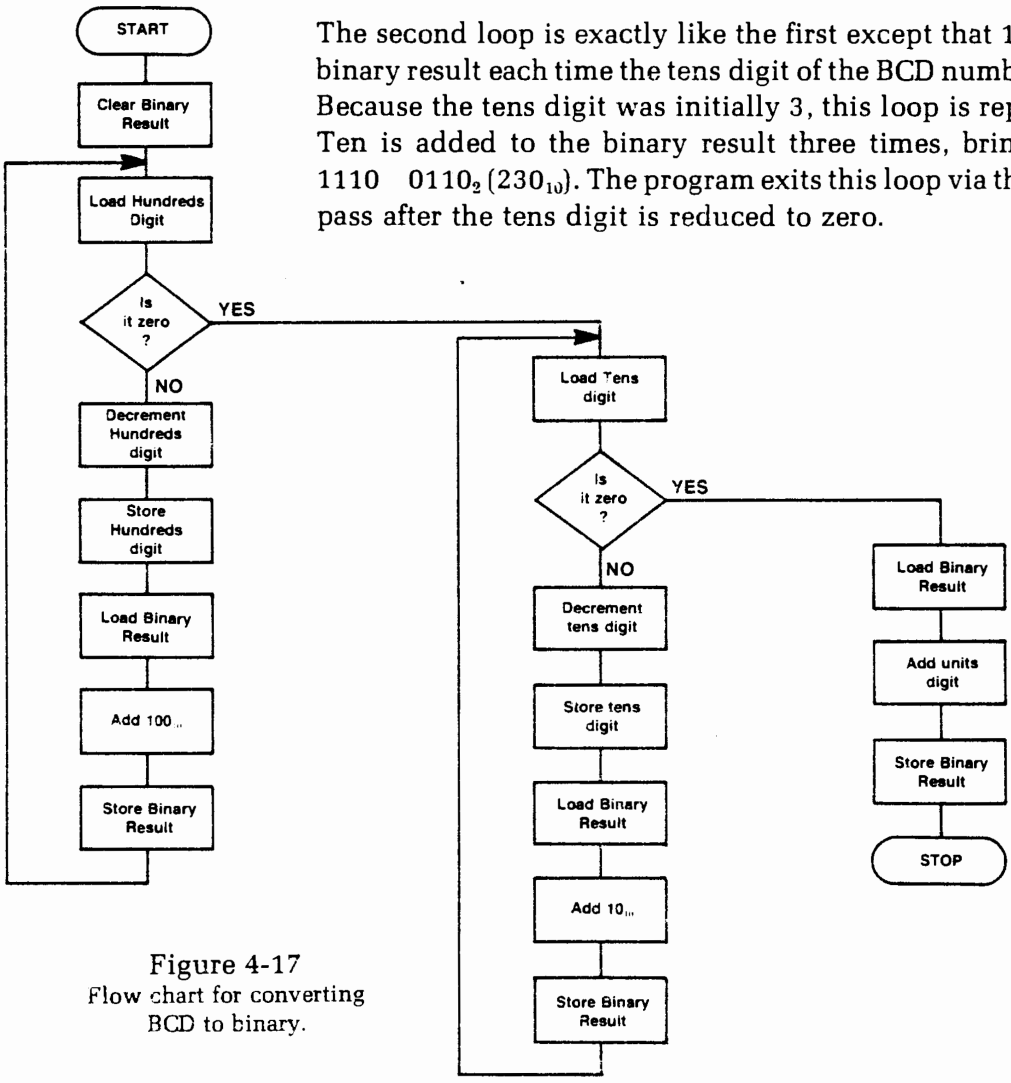


Figure 4-17
Flow chart for converting BCD to binary.

The final three blocks add the units digit to the binary result. In our example, the units digit was 7_{10}. This brings the final binary result to $1110\ \ 1101_2$. Notice that this is the proper binary representation for the unsigned number 237_{10}.

A program that carries out this operation is shown in Figure 4-18. The three digit BCD number is stored in locations 28_{16}, 29_{16}, and $2A_{16}$. The binary equivalent will be computed and placed in location $2B_{16}$. Before reading further, try to work through the program. Refer to the flow chart and the comments column as you trace out the sequence that the MPU will follow.

HEX ADDRESS	HEX CONTENTS	MNEMONIC/HEX CONTENTS	COMMENTS
00	4F	CLRA	Clear the accumulator.
01	97	STA	Store 00
02	2B	2B	in location 2B. This clears the binary result.
03	96	LDA	Load direct
04	28	28	the hundreds BCD digit.
05	27	BEQ	If the hundreds digit is zero, branch
06	0B	0B	forward to the instruction in location 12_{16}.
07	4A	DECA	Otherwise, decrement the accumulator.
08	97	STA	Store the result as the new
09	28	28	hundreds BCD digit.
0A	96	LDA	Load direct
0B	2B	2B	the binary result.
0C	8B	ADD	Add immediate
0D	64	64	100_{10} to the binary result.
0E	97	STA	Store away the new
0F	2B	2B	binary result.
10	20	BRA	Branch
11	F1	F1	back to the instruction in location 03_{16}.
12	96	LDA	Load direct
13	29	29	the tens BCD digit.
14	27	BEQ	If the tens BCD digit is zero, branch
15	0B	0B	forward to the instruction in location 21_{16}.
16	4A	DECA	Otherwise, decrement the accumulator.
17	97	STA	Store the result as the new
18	29	29	tens BCD digit.
19	96	LDA	Load direct
1A	2B	2B	the binary result.
1B	8B	ADD	Add immediate
1C	0A	0A	10_{10} to the binary result.
1D	97	STA	Store away the new
1E	2B	2B	binary result.
1F	20	BRA	Branch
20	F1	F1	back to the instruction in location 12_{16}.
21	96	LDA	Load direct
22	2B	2B	the binary result.
23	9B	ADD	Add direct
24	2A	2A	the units BCD digit.
25	97	STA	Store away the new
26	2B	2B	binary result.
27	3E	HLT	Halt.
28	02	02	Hundreds BCD digit.
29	03	03	Tens BCD digit.
2A	07	07	Unit BCD digit.
2B	—	—	Reserved for the binary result.

Figure 4-18
Program for converting BCD to binary.

Now let's briefly go through the program. The first two instructions clear the location at which the binary number will be formed.

Next, the program enters the first loop, which is shown as the first shaded area. In this loop, the hundreds digit is loaded and tested for zero. If not zero, it is decremented and stored away. Then the binary result is loaded and 100_{10} is added. The result is stored away and the loop is repeated. Because the hundreds digit was 02 initially, 100_{10} will be added to the binary result twice. Thus, upon leaving this loop, the binary result will have the value 200_{10}. The MPU escapes this loop when the BEQ instruction at address 05 detects that the hundreds digit has been reduced to zero. The branch is to the second loop which is shown as an unshaded area.

In the second loop, the tens digit is loaded and tested for zero. If not zero, it is decremented and stored away. Then the binary number is loaded, 10_{10} is added, and the result is stored away. This loop is repeated until the tens digit is reduced to zero. Because the tens digit was initially three, the loop is repeated three times so that thirty is added to the binary number. The BEQ instruction at address 14_{16} allows the MPU to escape the loop and branch to the final program segment.

This final segment is the last shaded area. Here, the binary result is loaded and the units digit is added. This brings the binary result to 237_{10}. Then the result is stored and the program halts. While the number 237_{10} was used in this example, the program will convert any BCD number between 000 and 255_{10} to its binary equivalent.

Converting Binary to BCD

A microprocessor generally manipulates data in the form of straight binary numbers. However, before the results can be transmitted to the outside world, the data is often converted back to BCD. Frequently, this is an intermediate step in converting back to ASCII.

The binary-to-BCD conversion is the reverse of the process that occurred in the previous program. The MPU must determine how many times 100_{10} can be subtracted from the binary number. The answer becomes the hundreds BCD digit. After the 100_{10} has been subtracted as many times as possible, 10_{10} is subtracted repeatedly from the remaining number. The number of subtractions becomes the tens BCD digit. Finally, after 10_{10} has been subtracted as many times as possible, the remaining number becomes the units BCD digit.

For the number 1110 1101_2 (237_{10}), the process looks likes this:

```
  1110  1101            237
 -0110  0100           -100
 -----------           ----
  1000  1001            137     hundreds digit  =  2
 -0110  0100           -100
 -----------           ----
  0010  0101             37
 -0000  1010           - 10
 -----------           ----
  0001  1011             27     tens digit  =  3
 -0000  1010           - 10
 -----------           ----
  0001  0001             17
 -0000  1010           - 10
 -----------           ----
  0000  0111              7  <-- units digit  =  7
```

One hundred can be subtracted twice. Thus, the hundreds digit is 2_{10} or 0010_2. From the remainder, ten can be subtracted three times. Thus, the tens digit is 3_{10} or 0011_2. Finally, the remainder of 7_{10} or 0111_2 becomes the units digit. The BCD representation is 0010 0011 0111.

Figure 4-19 shows the flow chart for this procedure. The first three blocks clear the hundreds, tens, and units digits of the BCD result. Then the binary number that is to be converted to BCD is loaded and 100_{10} is subtracted. The outcome is tested to see if a negative number resulted. If not, the result is stored away. The hundreds digit is loaded, incremented, and stored away. The loop is repeated until 100_{10} can no longer be subtracted. In our example, 100_{10} can be subtracted twice. Therefore, the hundreds digit is incremented to 2. The third subtraction of 100_{10} gives a negative result. This allows the MPU to escape the first loop.

The second loop increments the tens digit to the proper value by subtracting 10_{10} repeatedly and keeping track of the number of subtractions. In our example, this loop is repeated three times. Consequently, the tens digit is incremented to 3. The binary number that is left over after 10_{10} is subtracted the proper number of times becomes the units digit. That is, upon escaping the second loop, the remaining binary number is stored in the units digit. In our example, the remaining number, and therefore the units digit, is 7.

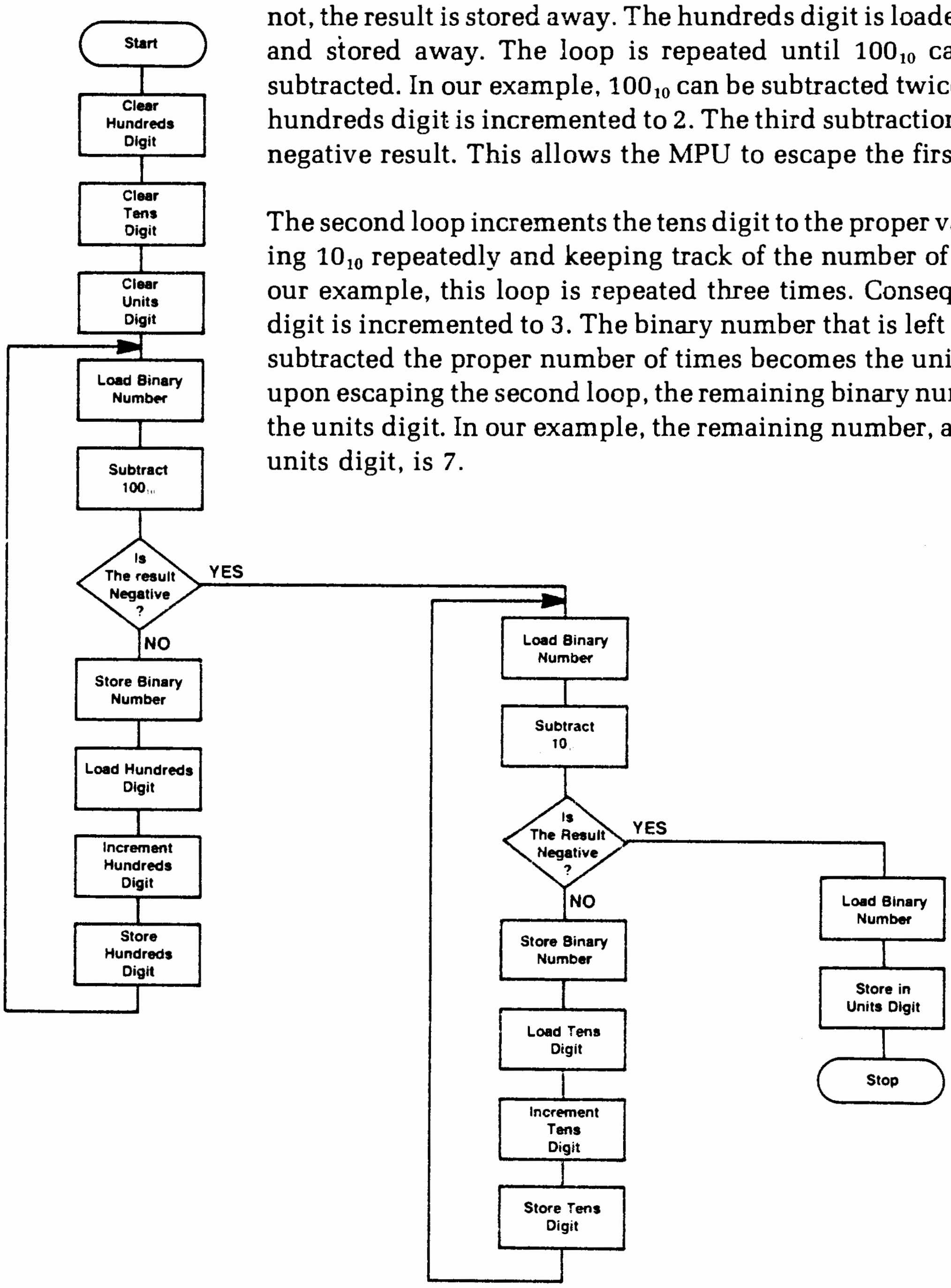


Figure 4-19
Flow chart for converting a binary number to a BCD number.

The program that carries out this procedure is shown in Figure 4-20. At this point, you should be able to interpret the program from the comments given. However, a couple of points should be explained briefly. Any unsigned binary number from 0000 0000 to 1111 1111 can be placed at address $2A_{16}$. The computer will convert this number into its BCD equivalent. The hundreds digit will appear at address $2B_{16}$, the tens digit at $2C_{16}$, and the units digit at $2D_{16}$. The decision making instructions at addresses $0B_{16}$ and $1A_{16}$ are Branch if Carry Set (BCS) instructions. Because these instructions follow immediately after SUB instructions, the carry flag will indicate whether or not a borrow occurred. In effect, the BCS instructions decide: "Was the result of the subtraction a negative number?"

HEX ADDRESS	HEX CONTENTS	MNEMONIC/HEX CONTENTS	COMMENTS
00	4F	CLRA	Clear the accumulator.
01	97	STA	Store 00
02	2B	2B	in location $2B_{16}$. This clears the hundreds digit.
03	97	STA	Store 00
04	2C	2C	in location $2C_{16}$. This clears the tens digit.
05	97	STA	Store 00
06	2D	2D	in location $2D_{16}$. This clears the units digit.
07	96	LDA	Load direct
08	2A	2A	the binary number to be converted.
09	80	SUB	Subtract immediate
0A	64	64	100_{10}.
0B	25	BCS	If a borrow occurred, branch
0C	09	09	forward to the instruction in location 16_{16}.
0D	97	STA	Otherwise, store the result of the subtraction
0E	2A	2A	as the new binary number.
0F	96	LDA	Load direct
10	2B	2B	the hundreds digit of the BCD result.
11	4C	INCA	Increment the hundreds digit.
12	97	STA	Store the hundreds digit
13	2B	2B	back where it came from.
14	20	BRA	Branch
15	F1	F1	back to the instruction at address 07_{16}
16	96	LDA	Load direct
17	2A	2A	the binary number.
18	80	SUB	Subtract immediate
19	0A	0A	10_{10}.
1A	25	BCS	If a borrow occurred, branch
1B	09	09	forward to the instruction in location 25_{16}.
1C	97	STA	Otherwise, store the result of the subtraction
1D	2A	2A	as the new binary number.
1E	96	LDA	Load direct
1F	2C	2C	the tens digit.
20	4C	INCA	Increment the tens digit.
21	97	STA	Store the tens digit
22	2C	2C	back where it came from.
23	20	BRA	Branch
24	F1	F1	back to the instruction at address 16_{16}.
25	96	LDA	Load direct
26	2A	2A	the binary number.
27	97	STA	Store it in
28	2D	2D	the units digit.
29	3E	HLT	Halt.
2A	—	—	Place binary number to be converted at this address.
2B	—	—	Hundreds digit } Reserved for
2C	—	—	Tens digit } BCD result.
2D	—	—	Units digit }

Figure 4-20
Program for converting a binary number to a BCD number.

Self-Test Review

20. What is an algorithm?

21. What type of instruction is used to make a decision?

22. Refer to the program in Figure 4-14. If the multiplier is 8_{16} and the multiplicand is 15_{16}, how many times will the BEQ instruction be executed?

23. Refer to the program in Figure 4-16. What is the largest number that can be used as a dividend?

24. How could this program be modified so it could handle unsigned dividends up to 255_{10}?

25. When this program halts, where will the remainder be located?

26. Refer to the program in Figure 4-18. Assume that addresses 28_{16}, 29_{16}, and $2A_{16}$ contain 01, 09, and 08 respectively. How many times will 100_{10} be added to address $2B_{16}$?

27. How many times will 10_{10} be added?

28. Refer to the program in Figure 4-20. What is the purpose of the first four instructions?

29. What is the largest binary number that this program can convert to BCD?

Answers

20. An algorithm is a step-by-step procedure for doing a particular job.

21. Conditional branch instruction.

22. Nine times.

23. $+127_{10}$ or $0111\ 1111_2$.

24. Change the BMI instruction to BCS.

25. At address 11_{16}.

26. Once.

27. Nine times.

28. The first four instructions clear the locations where the BCD equivalent will be stored.

29. $1111\quad 1111_2$ or 255_{10}.

ADDITIONAL INSTRUCTIONS

Before leaving this unit, you should also look at four additional instructions. The names, opcodes and mnemonics of these instructions are shown in Figure 4-21.

NAME	MNEMONIC	HEX OPCODE		
		IMMEDIATE	DIRECT	INHERENT
ADD WITH CARRY	ADC	89	99	
SUBTRACT WITH CARRY	SBC	82	92	
ARITHMETIC SHIFT ACCUMULATOR LEFT	ASLA			48
DECIMAL ADJUST ACCUMULATOR	DAA			19

Figure 4-21
Four new instructions.

Recall that the ALU always adds numbers as if they were unsigned binary numbers. When it adds 8-bit numbers, a carry often occurs from the MSB, setting the C flag. Thus, you can think of the carry flag as an extension of the accumulator. Let's look at some instructions that use the carry flag.

Add With Carry (ADC) Instruction

This instruction is similar to the ADD instruction discussed earlier with one important difference. If the carry bit is set to 1 before this instruction is executed, 1 is added to the LSB of the sum. However, if the carry bit is 0 prior to execution, then no carry is added. The effect is the same as having the carry bit from the previous operation added to the result of the present operation.

Like the ADD instruction, the ADC instruction has two addressing modes: immediate and direct. As shown in Figure 4-21, the opcode for "ADD With Carry Immediate" is 89_{16}, while the opcode for "Add With Carry Direct" is 99_{16}.

A primary use of the ADC instruction is to simplify **multiple-precision arithmetic**. Multiple-precision means that two or more bytes are used to represent a number. Recall that a single byte can represent unsigned binary numbers with values up to 255_{10}. However, much larger numbers can be represented by using two or more bytes. Two bytes (16 bits) can represent unsigned binary values up to $2^{16}-1$ or $65{,}535_{10}$. Three bytes can represent values to $16{,}777{,}215_{10}$; etc. Thus, the MPU can handle numbers of virtually any size simply by stringing the proper number of bytes together.

Suppose, for example, that two very large numbers are to be added. Figure 4-22 shows how the addition might look on paper. Notice that two 24-bit numbers are being added to form a 24-bit sum. The MPU is restricted to operating on data in 8-bit bytes. Thus, each quantity involved must be represented by three bytes.

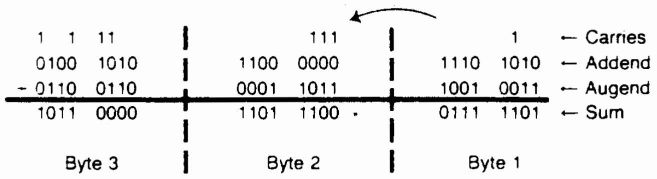

1 1 11 | 111 | 1 ← Carries
0100 1010 | 1100 0000 | 1110 1010 ← Addend
+ 0110 0110 | 0001 1011 | 1001 0011 ← Augend
1011 0000 | 1101 1100 | 0111 1101 ← Sum
Byte 3 | Byte 2 | Byte 1

Figure 4-22
Multiple-precision addition.

The MPU must be instructed to add the first byte of the addend to the first byte of the augend. This forms the first byte of the sum. Next the MPU must add the second bytes of the addend and augend. However, you will notice that there was a carry from the first byte to the second byte. If this carry is not added with the second bytes, the sum will be in error. The ADC instruction performs this operation automatically.

The program for adding the multiple-byte numbers could be written as shown in Figure 4-23. The three byte addend is stored in locations 13_{16} through 15_{16} while the augend is stored in locations 16_{16} through 18_{16}. Verify that the hexadecimal contents shown are the same as the binary values given in Figure 4-22.

Page 4-48

Reserved for sum.

HEX ADDRESS	HEX CONTENTS	MNEMONIC/HEX CONTENTS	COMMENTS	
00	96	LDA	Load accumulator direct with	
01	13	13	least significant byte of addend.	
02	9B	ADD	Add direct	
03	16	16	least significant byte of augend.	
04	97	STA	Store result in	
05	19	19	least significant byte of sum.	
06	96	LDA	Load accumulator direct with	
07	14	14	next byte of addend.	
08	99	ADC	Add with carry direct	
09	17	17	next byte of augend.	
0A	97	STA	Store result in	
0B	1A	1A	next byte of sum.	
0C	96	LDA	Load accumulator direct with	
0D	15	15	most significant byte of addend.	
0E	99	ADC	Add with carry direct	
0F	18	18	most significant byte of augend.	
10	97	STA	Store result in	
11	1B	1B	most significant byte of sum.	
12	3E	HLT	Halt.	
13	EA	EA	Least significant byte	Addend.
14	C0	C0		
15	4A	4A	Most significant byte	
16	93	93	Least significant byte	Augent
17	1B	1B		
18	66	66	Most significant byte	
19	—	—	Least significant byte	Reserved for sum.
1A	—	—		
1B	—	—	Most significant byte.	

Figure 4-23
Program for multiple-precision addition.

The first two instructions add the least significant bytes of the addend and augend. The ADD instruction is used because the MPU need not consider earlier carries. The first byte of the resulting sum is stored in location 19_{16}.

The next two instructions add the next two bytes. This time the ADC instruction is used because the MPU must consider the carry from the previous addition. The second byte of the sum is placed in location $1A_{16}$.

Finally, the last two bytes are added using the ADC instruction. The final byte of the sum is stored in location $1B_{16}$. The program halts when the addition is completed.

Subtract With Carry (SBC) Instruction

This instruction simplifies multiple-precision subtraction. You will recall that during subtract operations the carry flag indicates whether or not a borrow operation occurred. For this reason, this instruction can be thought of as a subtract with borrow operation.

The SBC instruction subtracts the subtrahend from the minuend just as the SUB instruction did. However, the SBC instruction has an additional step in that the carry bit is also subtracted. As with the other add and subtract instructions, both immediate and direct addressing modes are possible. The opcodes for both modes are shown in Figure 4-21.

Figure 4-24 illustrates how multiple-precision numbers can be subtracted. Notice that, during the course of this subtraction, byte 1 must "borrow" a 1 from byte 2. The SBC instruction allows the MPU to do this.

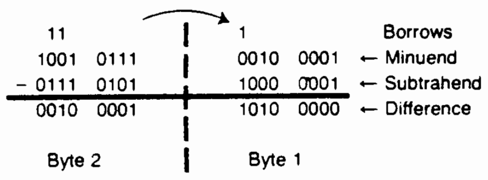


Figure 4-24
Multiple-precision subtraction.

HEX ADDRESS	HEX CONTENTS	MNEMONIC/HEX CONTENTS	COMMENTS
00	96	LDA	Load accumulator direct with
01	0D	0D	least significant byte of minuend.
02	90	SUB	Subtract direct
03	0F	0F	least significant byte of subtrahend.
04	97	STA	Store result in
05	11	11	least significant byte of difference.
06	96	LDA	Load accumulator direct with
07	0E	0E	most significant byte of minuend.
08	92	SBC	Subtract with carry
09	10	10	most significant byte of the subtrahend.
0A	97	STA	Store result in
0B	12	12	most significant byte of the difference.
0C	3E	HLT	Halt
0D	21	21	Least significant byte } Minuend.
0E	97	97	Most significant byte }
0F	81	81	Least significant byte } Subtrahend.
10	75	75	Most significant byte }
11	—	—	Least significant byte } Difference.
12	—	—	Most significant byte }

Figure 4-25
Program for multiple-precision subtraction.

Figure 4-25 shows a simple program for performing the subtraction. The double-precision minuend is at addresses $0D_{16}$ and OE_{16}, while the subtrahend is at addresses $0F_{16}$ and 10_{16}. The program computes the difference and stores it in locations 11_{16} and 12_{16}.

The first instruction loads the least significant byte of the minuend. Next, the corresponding byte of the subtrahend is subtracted. Since the subtrahend byte is larger, a borrow is indicated. Consequently, the carry flag is set to 1. Notice that the SUB rather than the SBC instruction is used. This is done because the first byte should not be affected by any previous borrow or carry. The result of the subtraction is stored away to become the least significant byte of the difference.

The most significant byte of the minuend is loaded next and the corresponding byte of the subtrahend is subtracted. However, this time the SBC instruction is used. And since the carry flag is set, an additional 1 is subtracted from the minuend to complete the borrow operation. The result of the subtraction becomes the most significant byte of the difference.

Arithmetic Shift Accumulator Left (ASLA) Instruction

The ASLA instruction shifts the contents of the accumulator to the left by one space. Figure 4-26 illustrates the repeated execution of this instruction. Figure 4-26A shows the condition of the accumulator and carry bit. In this example, the number in the accumulator is arbitrarily assumed to be 10_{10}. Also, the carry bit is arbitrarily assumed to be cleared.

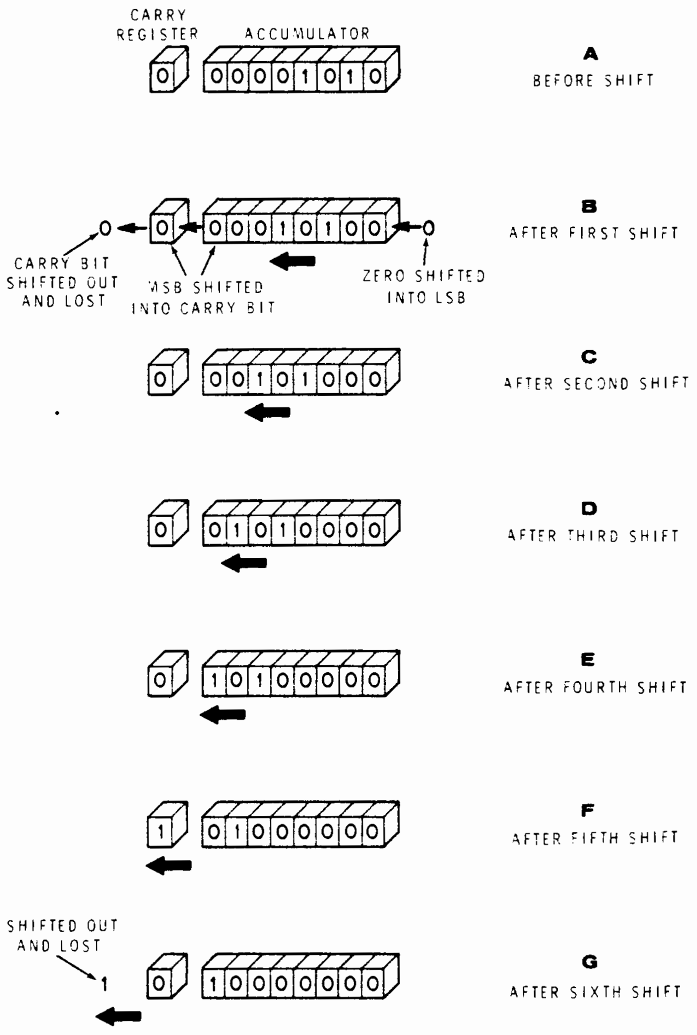


Figure 4-26
Repeatedly implementing the ASLA instruction.

Figure 4-26B shows the contents of the accumulator and carry bit after the ASLA instruction is executed. Notice that the number is shifted one bit to the left. Also, a 0 is shifted into the LSB. At the same time, the MSB is shifted into the carry bit. The old carry bit is shifted out and is lost.

You can understand one purpose of this instruction by examining the numbers in the accumulator before and after the instruction is executed. Before the shift, the number is 10_{10}; afterwards the number is 20_{10}. The number has been doubled. If you will try several different examples, you will see that any binary number can be multiplied by two simply by shifting the number one bit to the left. This holds true as long as the capacity of the accumulator is not exceeded.

Figures 4-26C through G show what happens if the MPU continues to execute ASLA instructions. The number continues to double. The number in the accumulator becomes 40_{10}, then 80_{10}, then 160_{10}. Each shift multiplies the number by two. On the fifth shift, the capacity of the accumulator is exceeded as the most significant 1 bit shifts into the carry bit. After the sixth shift, the leading 1 is lost altogether. When you use this technique to multiply by two or by a power of two, you must not exceed the capacity of the accumulator.

Another use of the ASLA instruction is to pack two BCD digits in a single byte. Earlier when we worked with BCD numbers, we assumed that each BCD digit resided in a separate memory byte. However, because a BCD digit has only 4 bits, memory space is wasted by assigning each digit a separate byte. Frequently, it is more desirable to "pack" two BCD digits into a single byte. A simple routine for doing this is shown in Figure 4-27. If dozens of BCD numbers are to be manipulated, a routine that uses a procedure similar to this can save substantial memory space. At the same time, it puts the BCD numbers into a more convenient and usable form.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/HEX CONTENTS	COMMENTS
00	96	LDA	Load into the accumulator direct
01	0C	0C	the unpacked most significant BCD digit.
02	48	ASLA	
03	48	ASLA	Shift it four places to
04	48	ASLA	the left.
05	48	ASLA	
06	9B	ADD	Add
07	0D	0D	the unpacked least significant BCD digit.
08	97	STA	Store the result as
09	0B	0B	two packed BCD digits.
0A	3E	HLT	Halt
0B	—	—	Packed BCD digits.
0C	—	—	Most significant BCD digit (unpacked).
0D	—	—	Least significant BCD digit (unpacked).

Figure 4-27
Program for packing two
BCD digits into a single byte.

The procedure carried out by the program is quite simple. The most significant BCD digit is loaded into the accumulator. It is then shifted four places to the left to make room for the least significant BCD digit. The least significant digit is then added to form a packed BCD number. The resulting single byte number is stored back in memory.

Decimal Adjust Accumulator (DAA) Instruction

Earlier in this unit, the problems of converting from BCD to binary and back again were considered. While this conversion is frequently necessary, many microprocessors have some limited BCD arithmetic capabilities. Our hypothetical MPU has an instruction that greatly simplifies BCD arithmetic. It is called the Decimal Adjust Accumulator (DAA) Instruction. When used in conjunction with the ADD or ADC instruction, it allows the MPU to add BCD numbers directly without an intermediate binary conversion.

Recall that the ALU adds input data bytes as if they were unsigned binary numbers. Therefore, if two BCD digits are added, the sum may be incorrect. For example, assume that the MPU adds the BCD digits 0111 and 0101. The ALU produces the result

```
     1
   0111
 + 0101
 ------
   1100
```

This answer is the correct **binary** result, 12_{10}; but it is not the proper BCD result. Recall that in BCD, 12_{10} is represented as 0001 0010. Notice that you can obtain the proper BCD result by adding 0110_2 to the binary result. The addition of 0110_2 is necessary anytime that the binary result exceeds 1001_2.

To produce the proper BCD result when adding two BCD digits, the MPU must follow this procedure:

1. If the sum is 1001_2 or less, use the sum as the single digit BCD result.

2. If the sum is greater than 1001_2, add 0110_2 and use the result as a 2-digit BCD number.

The situation becomes more complex when packed BCD numbers are added. Consider adding 0111 1001_{BCD} (79_{10}) to 0111 0011_{BCD} (73_{10}). The ALU adds these packed BCD numbers as if they were unsigned binary numbers. The result is

```
 11     1
0111 1001
0111 0011
---------
1110 1100
```

Notice that the result is not a BCD number, since both 4-bit groups exceed 1001_2. Even so, the sum can be converted to BCD by adding 0110_2 to each 4-bit group. The result is

```
1   1 1  1
   1110 1100
   0110 0110
------------
1  0101 0010
```

There is a carry from bit 7 that sets the carry bit. This carry bit becomes the most significant BCD digit. Thus, the final BCD result is 0001 0101 0010_{BCD} or 152_{10}.

If you consider all possible combinations of BCD numbers, you will find that four different situations exist:

1. When some BCD numbers are added, the binary result produced by the ALU is equal to the proper BCD representation. This occurs when both BCD digits of the result are 1001_2 or less.

2. The binary sum is adjusted by adding 06_{16} if the least significant BCD digit exceeds 1001_2 but the most significant BCD digit does not.

3. The binary sum is adjusted by adding 60_{16} if the most significant BCD digit exceeds 1001_2 but the least significant BCD digit does not.

4. The binary sum is adjusted by adding 66_{16} if both BCD digits exceed 1001_2.

While this procedure could be programmed, it would be much better if the MPU performed these operations automatically. Fortunately, our hypothetical microprocessor does this. The programmer simply informs the MPU that the numbers being added are BCD numbers. The MPU automatically computes the proper BCD result. The way the programmer informs the MPU is via the DAA instruction. When the DAA instruction is placed immediately after an ADD or ADC instruction, the MPU automatically converts the sum to the proper BCD number.

Suppose, for example, that you wish to add two BCD numbers. Assume the numbers are 3792_{10} and 5482_{10}. Naturally, the sum should be 9274_{10}. A program for solving this problem is shown in Figure 4-28. The BCD addend (3792_{10}) is in addresses OF_{16} and 10_{16}. The augend (5482_{10}) is in locations 11_{16} and 12_{16}. The BCD sum will be placed in locations 13_{16} and 14_{16}.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/HEX CONTENTS	COMMENTS
00	96	LDA	Load into the accumulator direct
01	10	10	the least significant half of the addend.
02	9B	ADD	Add
03	12	12	the least significant half of the augend.
04	19	DAA	Decimal adjust the sum to BCD.
05	97	STA	Store the result as the
06	14	14	least significant half of the sum.
07	96	LDA	Load
08	0F	0F	the most significant half of the addend.
09	99	ADC	Add
0A	11	11	the most significant half of the augend.
0B	19	DAA	Decimal adjust the sum to BCD.
0C	97	STA	Store the result as the
0D	13	13	most significant half of the sum.
0E	3E	HLT	Halt
0F	37	37	} BCD Addend
10	92	92	
11	54	54	} BCD Augend.
12	82	82	
13	—	—	} Reserved for BCD sum.
14	—	—	

Figure 4-28
Program for adding multiple-precision BCD numbers.

The first two instructions add the least significant halves of the addend and augend. The ADD instruction is followed immediately by the DAA instruction. Therefore, the sum is adjusted to a packed BCD number. The result is stored in location 14_{16} as the lower half of the BCD sum.

Next, the upper halves of the addend and augend are added. This time, the ADC instruction is used because the carry from the previous addition must be added in. Again, the DAA instruction adjusts the sum to BCD. The result is stored as the upper half of the BCD sum.

The DAA instruction must be used properly. It can be used only with addition. Also, it must be used immediately after the addition instruction. It can not be used to convert just any binary number to BCD. It only converts the sum of BCD numbers to the BCD format.

Self-Test Review

30. How is the ADC instruction different from the ADD instruction?

31. How is the SBC instruction different from the SUB instruction?

32. A primary use of the ADC and SBC instructions is in ______________-______________ arithmetic.

33. The accumulator contains the number 7_{10}. If two ASLA instructions are executed, what number will be in the accumulator?

34. What is the difference between packed and unpacked BCD numbers?

35. When adding unpacked BCD numbers, under what condition must 0110_2 be added to the sum in order to form a BCD sum?

36. What instruction is used to automatically adjust the sum to the proper BCD format when two BCD numbers are added?

37. Can the DAA instruction be used after a SUB instruction to produce the proper BCD difference when two BCD numbers are subtracted?

Answers

30. When the ADC instruction is executed, an additional 1 is added to the sum if the carry flag is set.

31. When the SBC instruction is executed, an additional 1 is subtracted from the difference if the carry flag is set.

32. Multiple-precision.

33. The first ASLA instruction multiplies the number by two, giving 14_{10}. The second ASLA doubles this number, giving 28_{10}.

34. With packed BCD numbers, each byte contains two BCD digits. With unpacked BCD numbers, each byte contains one BCD digit.

35. When two BCD digits are added, 0110_2 must be added to the sum if the sum exceeds 1001_2.

36. The decimal adjust accumulator (DAA) instruction.

37. No. The DAA instruction is used in conjunction with add instructions only.

EXPERIMENTS

Perform Programming Experiments 5 and 6. You will find these experiments in Unit 9. After you finish these experiments, return to this unit and complete the Unit Examination.

UNIT EXAMINATION

1. The BRA instruction will cause a branch to occur:

   A. Anytime that it is executed.
   B. Only if the Z flag is set.
   C. Only if the N flag is set.
   D. Only if the C flag is set.

2. The address that follows the opcode of an unconditional branch instruction is:

   A. The address of the operand.
   B. The address of the next opcode to be executed.
   C. Added to the program count to form the address of the next opcode to be executed.
   D. Added to the program count to form the address of the operand that is to be tested to see if a branch operation is required.

3. The opcode for an unconditional branch instruction is at address AF_{16}. The relative address is $0F_{16}$. From what address will the next opcode be fetched?

   A. $A0_{16}$.
   B. $C0_{16}$.
   C. BE_{16}.
   D. $B1_{16}$.

4. The opcode for an unconditional branch instruction is at address 30_{16}. The relative address is EF_{16}. From what address will the next opcode be fetched?

   A. 21_{16}.
   B. EF_{16}.
   C. 32_{16}.
   D. 19_{16}.

5. The carry register:

   A. Acts like the ninth bit of the accumulator.
   B. Is set when a "borrow" for bit 7 of the accumulator occurs.
   C. Is set when a carry from bit 7 occurs.
   D. All the above.

6. The numbers 0101 1000_2 and 0110 0011_2 are added using the ADD instruction. Immediately after the ADD instruction is executed, the condition code registers will indicate the following:

   A. C=1, N=1, V=1, Z=0.
   B. C=0, N=1, V=1, Z=0.
   C. C=0, N=1, V=0, Z=0.
   D. C=0, N=0, V=1, Z=1.

7. The divide program shown in Figure 4-16 works only if the dividend is initially less than $+128_{10}$. The program can be modified to work for dividends up to 255_{10} by replacing the BMI instruction with the:

   A. BEQ instruction.
   B. BNE instruction.
   C. BCC instruction.
   D. BCS instruction.

8. A binary number can be converted to BCD by repeatedly:

   A. Dividing by powers of two.
   B. Subtracting powers of ten.
   C. Multiplying by powers of two.
   D. Adding powers of ten.

9. The DAA instruction is used:

   A. To convert a binary number to BCD.
   B. To convert a BCD number to binary.
   C. After an add instruction to adjust the sum to a BCD number.
   D. After a subtract instruction to adjust the difference to a BCD number.

10. When you are adding multiple-precision binary numbers, all bytes except the least significant ones must be:

   A. Added using the ADD instruction.
   B. Added using the DAA instruction.
   C. Added using the ADC instruction.
   D. Decimal adjusted before addition takes place.