HEATHKIT
CONTINUING
EDUCATION

Individual Learning Program

MICROPROCESSORS

Unit 5

THE 6800 MICROPROCESSOR — PART 1

EE-3401

HEATH COMPANY
BENTON HARBOR, MICHIGAN 49022

CONTENTS

INTRODUCTION

Until now, we have confined our study to a simple hypothetical microprocessor. Obviously, though, this hypothetical model must be very close to the real thing, since we have been running its programs on the ET-3400 Microprocessor Trainer. In this unit, you will begin your study of the actual microprocessors upon which our hypothetical model is based. The microprocessor in the ET-3400 Trainer is called the 6800. It was first released by Motorola in the mid 1970's. Today, it is also supplied by several other companies.

The microprocessor in the ET-3400A Trainer is called the 6808. The primary difference between the 6808 and 6800 microprocessors is the means by which clock signals are generated. The 6808 has an on-chip clock circuit, the 6800 does not. Because of this one difference, there will also be some chip pin assignments that are not identical. These differences will be discussed in more detail in Unit Seven of this course. However, in all other characteristics, such as the instruction set and internal registers, the 6800 and 6808 are identical. It is these identical characteristics which are the subject of this and the following unit. Therefore, this unit refers to the 6800 microprocessor only. But, the data presented also applies to the 6808 in the ET-3400A Trainer.

There are other microprocessors that have similar instruction sets, architecture, and addressing modes. Thus, by becoming familiar with the 6800 (6808), you should be able to understand and use a wide range of microprocessors.

You already know a great deal about the 6800 and/or 6808 microprocessor. You have been programming this device for the past several units. The main difference between the 6800 (6808) microprocessors and our hypothetical model is complexity. As you will see, the 6800 (6808) is a vastly expanded version of our hypothetical model.

UNIT OBJECTIVES

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

1. Draw a programming model of the 6800 MPU.

2. Explain the purpose of each block in a simplified block diagram of the 6800 MPU.

3. Using Appendix A and Figure 5-24 as references, explain the operation of all the instructions discussed in this unit.

4. Write simple programs that use indexed and extended addressing.

5. Using Figure 5-24 as a guide, find the opcode, number of MPU cycles, number of bytes, and effects on the condition code flags of every instruction discussed in this unit.

UNIT ACTIVITY GUIDE

	Completion Time
☐ Read Section on Architecture of the 6800 MPU.	________
☐ Complete Self-Test Review Questions 1 — 8.	________
☐ Read Section on Instruction Set of the 6800 MPU.	________
☐ Complete Self-Test Review Questions 9 — 26.	________
☐ Review Appendix A.	________
☐ Read Section on New Addressing Modes.	________
☐ Complete Self-Test Review Questions 27 — 43.	________
☐ Perform Programming Experiments 7 and 8.	________
☐ Complete Unit Examination.	________
☐ Check Examination Answers.	________

ARCHITECTURE OF THE 6800 MPU

In computer jargon, the word architecture is used to describe the computer's style of construction, its register size and arrangement, its bus configuration, etc. The architecture of our hypothetical microprocessor is shown for one last time in Figure 5-1. By now you should be quite familiar and comfortable with this architecture.

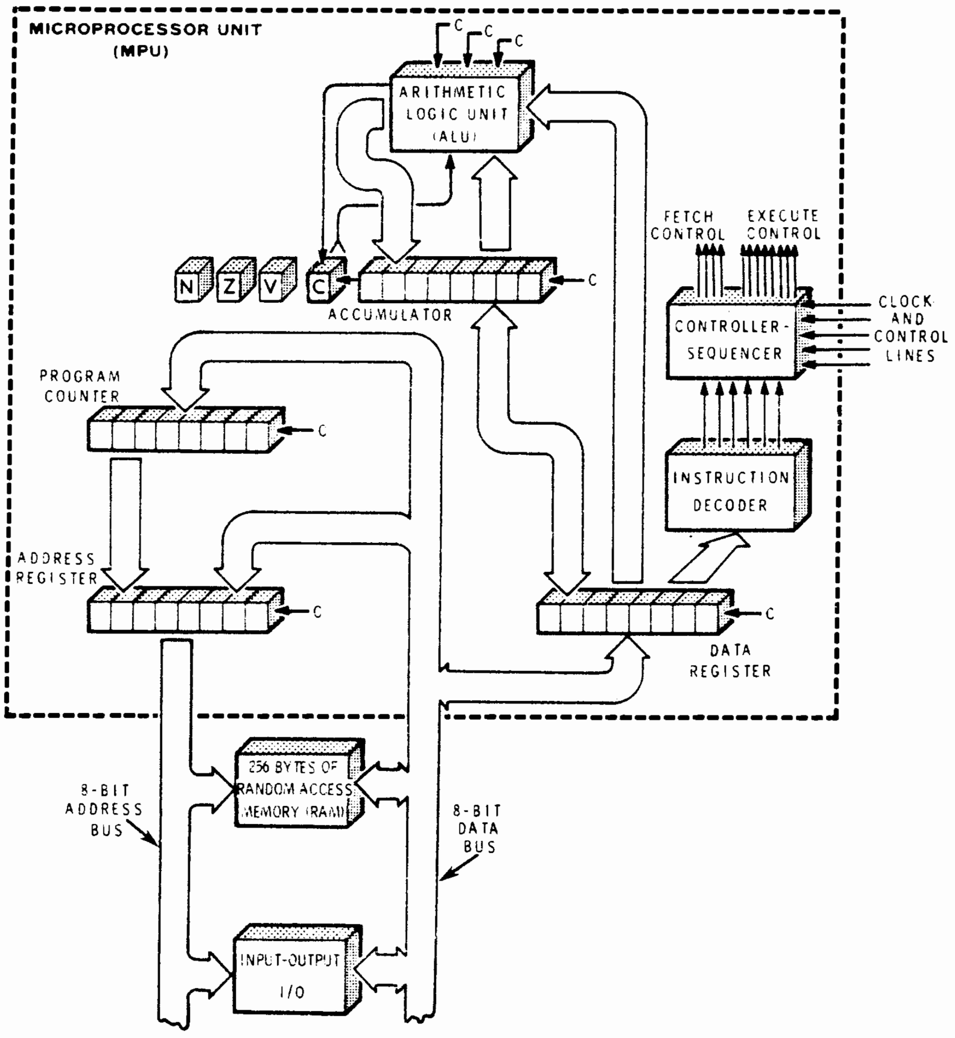


Figure 5-1
Architecture of the hypothetical microcomputer.

The only reason for showing the details of the model is to give you an idea of what goes on inside the integrated circuit. In an actual microprocessor the internal structure is often so complex that we become bogged down in details if we attempt to analyze it too closely. For this reason, a programming model is generally used when a microprocessor is being introduced for the first time. In the programming model, the emphasis is shifted upward by an order of magnitude. Any register or circuit that cannot be directly controlled by the programmer is simply ignored. Consider the data register for example. There are no instructions that give the programmer direct control over this register. That is, there are no instructions such as Load Data Register, Store Data Register, etc. All data register activity is controlled strictly by the MPU. Thus, the programmer can simply ignore the existence of this register. The same is true of the address register, the instruction decoder, the controller-sequencer, etc. Therefore, the programming model of our hypothetical MPU can be represented as shown in Figure 5-2. This simple diagram is sufficient for most programming applications since it shows all the registers that can be directly controlled by the program.

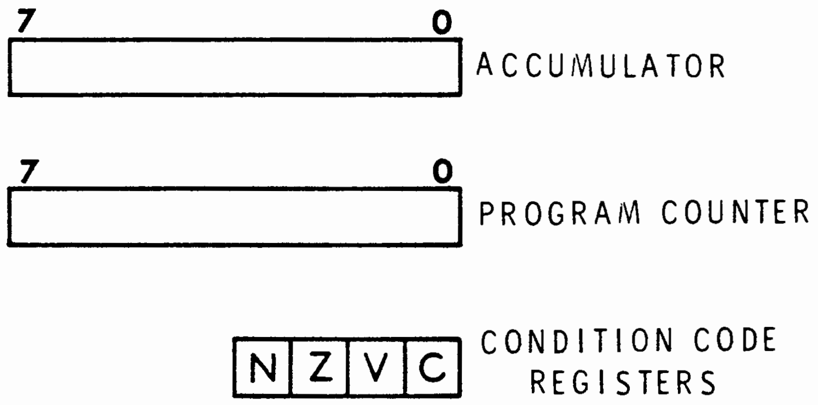


Figure 5-2
Programming model of the hypothetical MPU.

Programming Model of the 6800 MPU

The 6800 MPU is much more complex than our hypothetical MPU. Consequently, a programming model of the 6800 makes a good starting point. The programming model is shown in Figure 5-3.

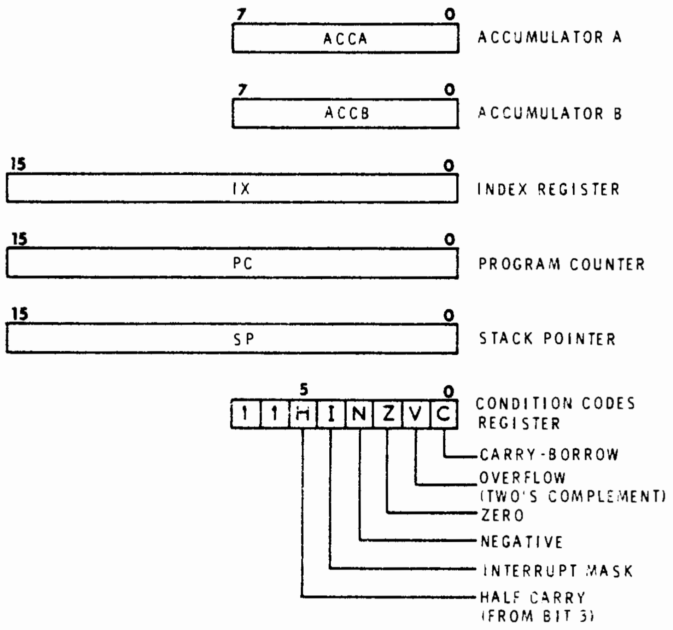


Figure 5-3
Programming model of the 6800 MPU.

You will notice immediately that the 6800 MPU has several additional registers. However, only two of these, the index register and the stack pointer, are actually new to you. Let's look at the major differences between this MPU and our hypothetical model.

Two Accumulators The 6800 MPU has two accumulators instead of one. They are called accumulator A (ACCA) and accumulator B (ACCB). Each has its own group of instructions associated with it. The names and mnemonics of the instructions specify which accumulator is to be used. Thus, there are instructions such as:

Load Accumulator A (LDAA)
Load Accumulator B (LDAB)
Store Accumulator A (STAA)
Store Accumulator B (STAB)

Notice that a letter is added to both the name and the mnemonic to indicate which accumulator is being used.

From your previous programming experience, you can visualize the value of a second accumulator. For example consider a program in which the MPU counts the number of times that some operation occurs. In the past, we stored the number that the accumulator was presently working on, loaded the count into the accumulator; incremented the count; stored the count; and reloaded the original number. With a second accumulator, none of this is necessary. We can simply maintain the count in accumulator B while working with the number in accumulator A. In fact, we can perform any arithmetic or logic operation on two different numbers without having to shift the numbers back and forth between memory.

16-Bit Program Counter The program counter in the 6800 has 16_{10} bits rather than 8. Thus, it can specify $65{,}536_{10}$ different addresses. This means that a 6800 based microcomputer can have up to $65{,}536_{10}$ bytes of memory. Most applications require substantially less memory than this maximum number. Fortunately, we can use as little or as much memory as we need up to the 2^{16} byte limit.

Since the program counter has 16_{10} bits, the address bus must also be 16-bits wide. Contrast this with the 8-bit address bus of our hypothetical machine.

You may wonder how we specify a 16-bit address with an 8-bit byte. The obvious answer is that two 8-bit bytes are required. Recall that in the direct addressing mode, the address was specified by a single 8-bit byte. The 6800 microprocessor retains this addressing mode. However, since an 8-bit address can specify only 256_{10} addresses, the 6800 MPU can use this mode only if the operand is in the first 256_{10} bytes of memory. To reach higher addresses, a new addressing mode called **extended addressing** must be used. In the extended addressing mode, two bytes are used to represent each address. This addressing mode will be discussed in more detail later. For now, keep in mind that there are $65{,}536_{10}$ possible addresses. The lowest address is 0000_{16} and the highest is $FFFF_{16}$. Using extended addressing, we have access to any location in memory, but a 2-byte address is required.

Condition Code Registers The 6800 MPU has six condition codes. Four of these are almost identical to those discussed in an earlier unit. These include the negative (N), zero (Z), overflow (V) and carry (C) condition codes. The difference arises because there are two accumulators in the 6800 MPU. Thus, the carry flag is set whenever there is a carry from either accumulator. By the same token, an overflow in either accumulator will set the V flag. Later in this unit, you will see how the condition codes are affected by each instruction.

Two new condition codes are shown in Figure 5-3. The I flag is called an interrupt mask. We will discuss this flag later when you study interrupts. The other is called the half carry flag (H). The H flag is set when there is a carry from bit 3 of the accumulator. The MPU uses this flag to determine how to implement the decimal adjust instruction.

These six flags make up bits 0 through 5 of an 8-bit register. Bits 6 and 7 of the condition code register are not used and are always set to 1. Additional details of the condition codes will be brought out as the need arises.

Index Register The index register is a special-purpose, 16-bit register that greatly increases the power of the microprocessor. It allows a powerful address mode called indexed addressing. We will examine this addressing mode later in this unit. For now, consider the index register to be just another working register. The fact that it holds two bytes instead of one can be put to good use. The MPU has instructions that allow the index register to be loaded from two adjacent memory bytes. Another instruction allows us to store the contents of the index register in two adjacent memory locations. This allows us to move data in 2-byte groups. Also, the index register can be incremented and decremented. This lets us maintain 16-bit tallies.

Stack Pointer The stack pointer is another special-purpose 16-bit register. It allows the MPU and the programmer to use a section of RAM as a last in, first out (LIFO) memory. This capability is extremely valuable when using subroutines or when processing interrupts. These aspects of the stack pointer will be discussed in the next unit. For the time being let's consider the stack pointer to be another 16-bit working register. It too can be loaded from memory, stored in memory, incremented, and decremented.

Block Diagram of the 6800 MPU

Now that you have seen the programming model of the 6800 MPU, take a look at the block diagram. A simplified block diagram is shown in Figure 5-4. Several data paths, most control lines, and a temporary storage register have been omitted in favor of the major data paths and registers.

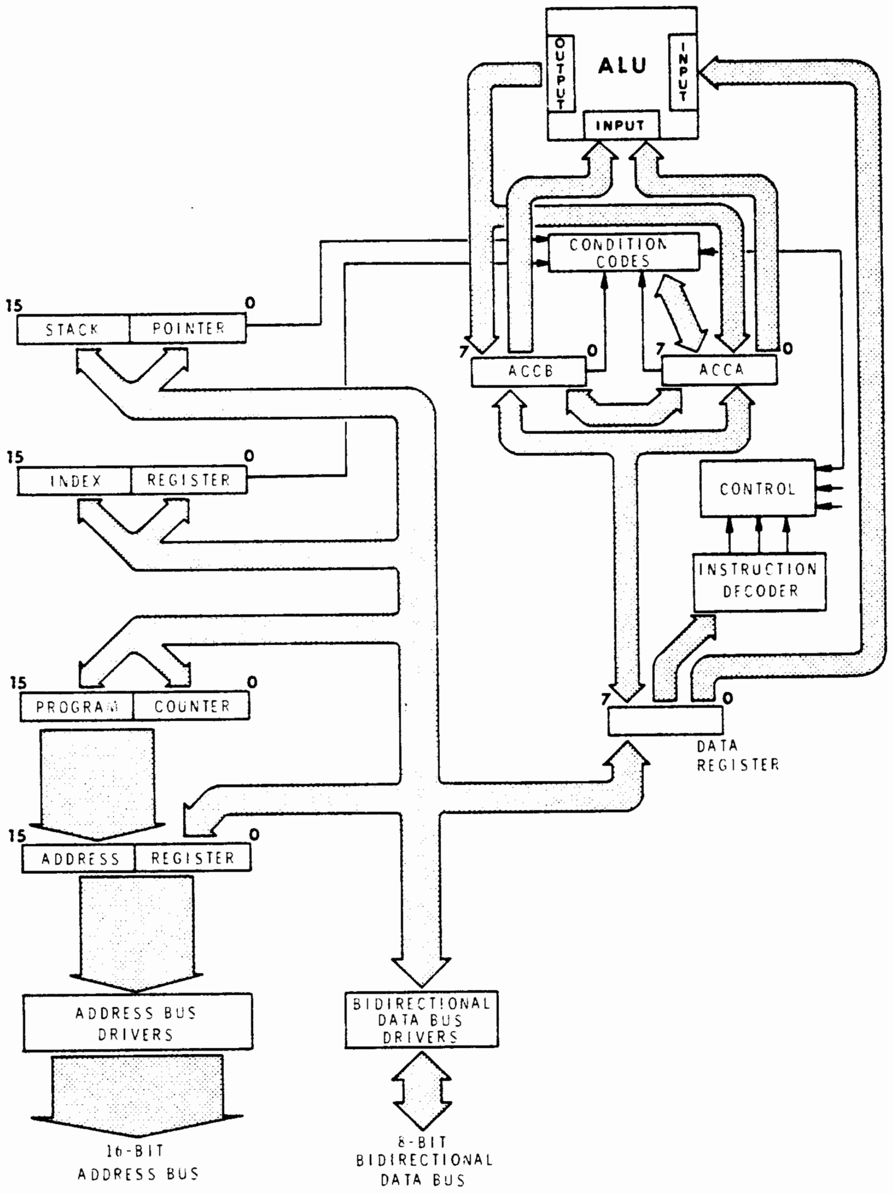


Figure 5-4
Simplified block diagram of the 6800 MPU.

The 16-bit registers are shown on the left. These registers are primarily concerned with addressing memory. Since the address bus has 16-bits, all registers associated with addressing must also have 16-bits. Any of the 16-bit registers can be loaded from the data bus. However, because the data bus has only 8-bits, two operations are required to load the 16-bit registers. The upper half of the affected register is always loaded first. Then, a second operation loads the lower half. Although this requires separate MPU cycles, the microprocessor takes care of these operations automatically. For example, a single instruction can load the 16-bit index register with two memory bytes.

The program counter and address register perform exactly the same functions in the 6800 MPU as they did in our hypothetical model. The fetch and execute phases for the immediate and direct addressing modes are virtually identical. The same is true of the relative addressing mode except that the 8-bit relative address is added to the 16-bit program count.

The 8-bit registers are shown on the right. Notice that these circuits are identical to those in our hypothetical model except that there are two accumulators. The condition code registers monitor both accumulators. Also, the two accumulators share the ALU. This allows you to keep track of two separate mathematical operations at more or less the same time. This arrangement is particularly flexible since the contents of one accumulator can be transferred to the other or the contents of the two accumulators can be added together.

Self-Test Review

1. The microprocessor on which our hypothetical model and the ET-3400 are based is the ______________ MPU.

2. A major difference between our hypothetical model and the 6800 MPU is that the latter has two ______________.

3. The program counter in the 6800 MPU has ______________ bits.

4. How wide is the address bus in a 6800-based microcomputer?

5. What is the range of addresses in the 6800 MPU?

6. List the six condition code flags.

7. Besides the program counter, what other 16-bit registers are used in the 6800 MPU?

8. In the 6800 MPU, does each accumulator have its own carry flag?

Answers

1. 6800.

2. Accumulators.

3. 16_{10}.

4. 16_{10} bits.

5. From 0000 to $65{,}535_{10}$ or 0000 to $FFFF_{16}$.

6. Carry — borrow (C)
   Overflow (V)
   Zero (Z)
   Negative (N)
   Interrupt Mask (I)
   Half Carry (H)

7. Index register and stack pointer.

8. No, the two accumulators share a common carry flag.

INSTRUCTION SET OF THE 6800 MPU

The 6800 MPU has about 100_{10} basic instructions. Moreover, when all the different addressing modes are considered, there are 197_{10} different op-codes to which the MPU will respond.

These instructions can be broken down into seven general categories. While some of the categories overlap, the general classifications of instructions are: arithmetic, data handling, logic, data test, index register and stack pointer, jump and branch, and condition code. In this unit we will discuss most of these instructions in detail. The handful of instructions that are not discussed in this unit will be described in the following unit.

In this section we will not be concerned with addressing modes. Therefore, no opcodes are given. Later, we will look at the various addressing modes and opcodes. For now, though, let's identify the instructions by their names, mnemonics, and operations. You will see what each instruction does and how it affects the various condition code registers.

Because of the large number of instructions covered in this section, the explanations will be general and brief. You are **not** expected to remember all the details of every instruction. Appendix A of this course contains a detailed listing of each instruction. It explains every detail of the various instructions. After reading this section, turn to Appendix A and look over the explanations given there. In the future, when you are in doubt as to exactly what a particular instruction does, look it up in Appendix A.

Arithmetic Instructions

Figure 5-5 shows the arithmetic instructions of the 6800 MPU. The name of each instruction is given on the left. The next column contains the mnemonics. The center column gives a shorthand description of what the instruction does. The right-hand columns show how the various condition code registers are affected.

ACCUMULATOR AND MEMORY OPERATIONS	MNEMONIC	BOOLEAN/ARITHMETIC OPERATION (All register labels refer to contents)	COND. CODE REG.					
			5	4	3	2	1	0
			H	I	N	Z	V	C
Add	ADDA	A + M → A	↕	●	↕	↕	↕	↕
	ADDB	B + M → B	↕	●	↕	↕	↕	↕
Add Acmltrs	ABA	A + B → A	↕	●	↕	↕	↕	↕
Add with Carry	ADCA	A + M + C → A	↕	●	↕	↕	↕	↕
	ADCB	B + M + C → B	↕	●	↕	↕	↕	↕
Complement, 2's (Negate)	NEG	00 − M → M	●	●	↕	↕	①	②
	NEGA	00 − A → A	●	●	↕	↕	①	②
	NEGB	00 − B → B	●	●	↕	↕	①	②
Decimal Adjust, A	DAA	Converts Binary Add. of BCD Characters into BCD Format*	●	●	↕	↕	↕	③
Subtract	SUBA	A − M → A	●	●	↕	↕	↕	↕
	SUBB	B − M → B	●	●	↕	↕	↕	↕
Subract Acmltrs.	SBA	A − B → A	●	●	↕	↕	↕	↕
Subtr. with Carry	SBCA	A − M − C → A	●	●	↕	↕	↕	↕
	SBCB	B − M − C → B	●	●	↕	↕	↕	↕

*Used after ABA, ADC, and ADD in BCD arithmetic operation; each 8-bit byte regarded as containing two 4-bit BCD numbers. DAA adds 0110 to lower half-byte if least significant number >1001 or if preceding instruction caused a Half-carry. Adds 0110 to upper half-byte if most significant number >1001 or if preceding instruction caused a Carry. Also adds 0110 to upper half-byte if least significant number >1001 and most significant number = 9.

(Bit set if test is true and cleared otherwise)

① (Bit V) Test: Result = 10000000?

② (Bit C) Test: Result = 00000000?

③ (Bit C) Test: Decimal value of most significant BCD Character greater than nine? (Not cleared if previously set.)

Figure 5-5
Arithmetic instructions.

To be certain you have the idea, let's go through the first instruction in detail. The first instruction is the add instruction. Actually, since the 6800 has two accumulators, there are two add instructions. Their mnemonics are ADDA and ADDB. Notice that the final letter of the mnemonic indicates which accumulator (A or B) is involved. The shorthand representation of the operation is: $A+M \rightarrow A$. The note at the top of this column tells you that the register labels refer to the contents of the register. Thus, A means the contents of accumulator A and M means the contents of the affected memory location. The symbol ($\rightarrow$) means "Transfer into." Therefore, $A+M \rightarrow A$ means "Add the contents of accumulator A to the contents of the affected memory location and transfer the sum into accumulator A."

To see how the condition code flags are affected, you simply look over to the right under whatever condition code you are interested in. Generally, the condition code is either unaffected or is tested and set accordingly. When the condition code is unaffected, this is represented by the symbol (•). For example, none of the arithmetic instructions affect the I flag. Most of the arithmetic instructions test the condition codes and set them if the condition exists. For example, if the result of an arithmetic operation is zero, the Z flag is set to 1. If this condition does not exist, the Z flag is reset or cleared to 0. The symbol ($\updownarrow$) means "test and set if true; clear otherwise." Occasionally, a note is necessary to describe some unusual situation regarding the condition code. This is represented by a number within a circle. The notes are given at the bottom of the drawing.

The ADDA and ADDB instructions are self-explanatory. The ABA instruction adds the contents of accumulator A to the contents of accumulator B. The result is stored in accumulator A.

The add with carry instructions are identical to those discussed earlier for our hypothetical machine. Notice that the carry bit is added in with the sum.

Because two's complement arithmetic is used in the 6800 MPU, instructions are provided that allow us to take the two's complement of a number. The negate instruction subtracts the contents of the affected register from 00_{16}. This is the same as taking the two's complement of the number. The affected register can be any memory location (M) or either accumulator (A or B). Thus, there are three different negate instructions. Keep in mind that NEG means "take the two's complement of the affected memory location;" NEGA means "take the two's complement of accumulator A;" etc.

Notice that the NEG instruction allows us to operate on a byte in memory without first fetching the operand from memory. In the past, we have loaded the operand, performed the operation, and then stored the new operand. However, the 6800 allows us to perform certain operations on the operand without first fetching it from memory. Several examples of this will be pointed out as we progress through the instruction set.

The decimal adjust instruction performs exactly as it did in our hypothetical machine. The note immediately under the table summarizes its operation. It must also be pointed out that this instruction works only with accumulator A.

The subtract and the subtract with carry instructions are self-explanatory. They perform as described earlier for our hypothetical MPU. The 6800 MPU has an additional subtract instruction. The SBA instruction subtracts the contents of accumulator B from the contents of accumulator A. The resulting difference is placed in accumulator A.

Data Handling Instructions

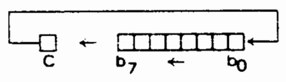
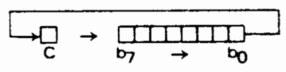
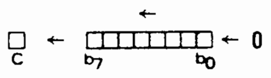
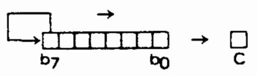
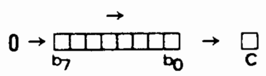
Figure 5-6 shows the largest group of instructions used by the 6800 MPU. These can be loosely categorized as data handling instructions.

The clear instructions allow us to clear a memory location or either accumulator. In the past, we have cleared bytes of memory by first clearing the accumulator and then storing the resulting 00_{16} in the proper memory location. However, the CLR instruction allows us to clear a memory location with a single instruction. Notice that some new entries appear in the condition code registers column. R means that the condition code is always reset or cleared to 0. S means that the code is always set to 1.

The decrement instruction allows us to subtract 1 from a memory location or from either accumulator. The DEC instruction is especially valuable since it allows us to decrement a byte in memory with a single instruction. Previously we have loaded the byte, decremented it, and then stored it back in memory.

The increment instructions are similar except they allow us to add 1 to a memory location or one of the accumulators. Notice that the INC instruction allows us to maintain a tally in memory without having to load it, increment it, and then store it away.

The load accumulator instructions are self-explanatory. Notice that either accumulator can be loaded from memory.

ACCUMULATOR AND MEMORY		BOOLEAN/ARITHMETIC OPERATION	COND. CODE REG.					
		(All register labels	5	4	3	2	1	0
OPERATIONS	MNEMONIC	refer to contents)	H	I	N	Z	V	C
Clear	CLR	00 → M	●	●	R	S	R	R
	CLRA	00 → A	●	●	R	S	R	R
	CLRB	00 → B	●	●	R	S	R	R
Decrement	DEC	M – 1 → M	●	●	↕	↕	④	●
	DECA	A – 1 → A	●	●	↕	↕	④	●
	DECB	B – 1 → B	●	●	↕	↕	④	●
Increment	INC	M + 1 → M	●	●	↕	↕	⑤	●
	INCA	A + 1 → A	●	●	↕	↕	⑤	●
	INCB	B + 1 → B	●	●	↕	↕	⑤	●
Load Acmltr	LDAA	M → A	●	●	↕	↕	R	●
	LDAB	M → B	●	●	↕	↕	R	●
Rotate Left	ROL	M 	●	●	↕	↕	⑥	↕
	ROLA	A	●	●	↕	↕	⑥	↕
	ROLB	B	●	●	↕	↕	⑥	↕
Rotate Right	ROR	M 	●	●	↕	↕	⑥	↕
	RORA	A	●	●	↕	↕	⑥	↕
	RORB	B	●	●	↕	↕	⑥	↕
Shift Left, Arithmetic	ASL	M 	●	●	↕	↕	⑥	↕
	ASLA	A	●	●	↕	↕	⑥	↕
	ASLB	B	●	●	↕	↕	⑥	↕
Shift Right, Arithmetic	ASR	M 	●	●	↕	↕	⑥	↕
	ASRA	A	●	●	↕	↕	⑥	↕
	ASRB	B	●	●	↕	↕	⑥	↕
Shift Right, Logic	LSR	M 	●	●	R	↕	⑥	↕
	LSRA	A	●	●	R	↕	⑥	↕
	LSRB	B	●	●	R	↕	⑥	↕
Store Acmltr	STAA	A → M	●	●	↕	↕	R	●
	STAB	B → M	●	●	↕	↕	R	●
Transfer Acmltrs	TAB	A → B	●	●	↕	↕	R	●
	TBA	B → A	●	●	↕	↕	R	●

④ (Bit V) Test: Operand = 10000000 prior to execution?
⑤ (Bit V) Test: Operand = 01111111 prior to execution?
⑥ (Bit V) Test: Set equal to result of $N \oplus C$ after shift has occurred.

Figure 5-6
Data handling instructions.

The rotate left instructions allow us to shift the contents of the accumulator or a memory location without losing bits of data. Consider the ROLA instruction as an example. When this instruction is executed, the A accumulator and the carry bit form a 9-bit circulating register. That is, they form a closed loop as shown in Figure 5-7A. When ROLA is executed, the data is rotated clockwise. The MSB of A shifts into the carry register. Simultaneously, the contents of A are shifted left. Notice that the carry bit is not lost. Instead it is shifted into the LSB of the accumulator.

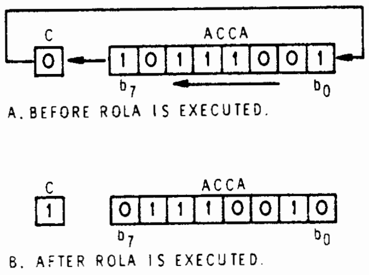


Figure 5-7
Executing the ROLA instruction.

While the usefulness of this instruction may not be obvious, it is a valuable tool. For example, it could be used to determine parity. By repeatedly rotating left and testing the C flag, you could determine the number of 1's in the byte. Once you know this, you could easily generate the proper parity bit.

The ROL instruction allows you to rotate a memory byte to the left while it is still in memory. ROLB allows you to rotate the B accumulator to the left. In each case, the C register is used as a ninth bit.

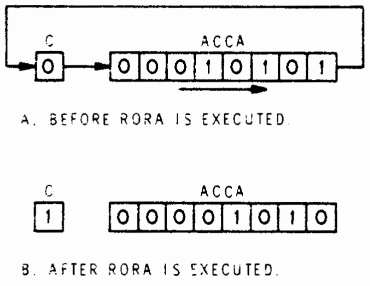


Figure 5-8
Executing the RORA instruction.

The rotate right instructions are identical except that the direction of rotation is reversed. Figure 5-8 illustrates the execution of the RORA instruction. This instruction is also valuable. Suppose for example that we wish to know if the number in the accumulator is even or odd. This is determined by the LSB of the number. If LSB = 1, the number is odd; if LSB = 0, the number is even. One way to determine this is to rotate the number to the right so that the LSB is in the C register. We could then test the C register to see if it is set or cleared. Notice that the number could then be restored to its original value by the ROLA instruction.

The arithmetic shift left instruction was discussed earlier in our hypothetical MPU. The ASLA instruction performs exactly as described in the previous unit. However, notice that the 6800 MPU also has an ASLB instruction that performs the same operation with accumulator B. Also, it has an ASL instruction that allows us to perform this operation on a byte that is in memory. Figure 5-9 illustrates the execution of this instruction.

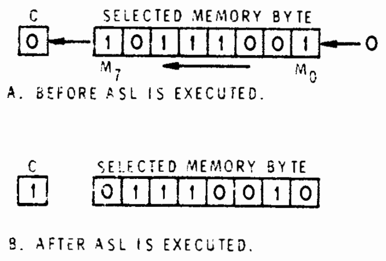


Figure 5-9
Executing the ASL instruction.

While there is only one type of shift left instruction, there are two types of shift right instructions. Let's discuss the arithmetic shift right instructions first.

When an **arithmetic** shift right instruction is executed, the number in the affected register is shifted right one position. The LSB goes into the C register. B_1 shifts to B_0, etc. B_7 shifts into B_6. However, B_7 itself remains unchanged. Figure 5-10 illustrates the execution of the ASRB instruction. Notice that there are also ASRA and ASR instructions listed in Figure 5-6. These perform the same type of shift operation but on accumulator A and the selected memory byte respectively.

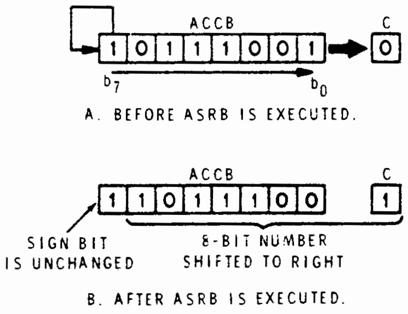

ACCB
C
1 0 1 1 1 0 0 1
0
b7
b0
A. BEFORE ASRB IS EXECUTED.
ACCB
C
1 1 0 1 1 1 0 0
1
SIGN BIT IS UNCHANGED
8-BIT NUMBER SHIFTED TO RIGHT
B. AFTER ASRB IS EXECUTED.

Figure 5-10
Executing the ASRB instruction.

The logic shift right instructions are different in that they do not retain the sign bit. When a logic shift right is executed, the contents of the affected register are shifted to the right. The LSB goes into the carry register. The MSB is filled with a 0. For example, the LSR instruction is illustrated in Figure 5-11. While this instruction shifts the selected memory locations, LSRA and LSRB can be used to perform similar operations on accumulators A and B respectively.

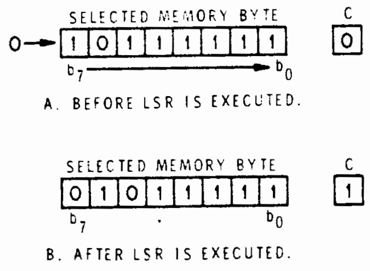


Figure 5-11
Executing the LSR instruction.

Referring back to Figure 5-6, the store accumulator instructions are self-explanatory.

The final data handling instructions are the transfer accumulator instructions. TAB copies the contents of accumulator A into accumulator B. After this instruction is executed, the number originally in accumulator A will be in both accumulators. TBA does just the opposite. It copies the contents of accumulator B into accumulator A. After TBA is executed, the number originally in accumulator B will be in both accumulators.

Logic Instructions

The logic instructions in the 6800 MPU are similar to those in our hypothetical MPU. Figure 5-12 shows the 6800's logic instructions.

ACCUMULATOR AND MEMORY		BOOLEAN/ARITHMETIC OPERATION	COND. CODE REG.					
		(All register labels	5	4	3	2	1	0
OPERATIONS	MNEMONIC	refer to contents)	H	I	N	Z	V	C
And	ANDA	$A \bullet M \to A$	●	●	↕	↕	R	●
	ANDB	$B \bullet M \to B$	●	●	↕	↕	R	●
Complement, 1's	COM	$\overline{M} \to M$	●	●	↕	↕	R	S
	COMA	$\overline{A} \to A$	●	●	↕	↕	R	S
	COMB	$\overline{B} \to B$	●	●	↕	↕	R	S
Exclusive OR	EORA	$A \oplus M \to A$	●	●	↕	↕	R	●
	EORB	$B \oplus M \to B$	●	●	↕	↕	R	●
Or, Inclusive	ORA	$A + M \to A$	●	●	↕	↕	R	●
	ORB	$B + M \to B$	●	●	↕	↕	R	●

Figure 5-12
Logic instructions.

There is one AND instruction for each accumulator. The contents of the specified accumulator are ANDed bit-for-bit with the contents of the selected memory location. The result is placed back in the accumulator. This is identical to the AND instruction in our hypothetical machine.

The complement instructions allow you to take the 1's complement of the number in the affected register. COM allows you to complement a byte in memory.

COMA and COMB allow you to complement the contents of accumulators A and B respectively. In each case, all 1's are changed to 0's and all 0's are changed to 1's.

The exclusive OR instructions work like the one in our hypothetical MPU. The contents of the specified accumulator are exclusively ORed bit-for-bit with the contents of the selected memory location. The result is stored back in the specified accumulator.

The inclusive OR is similar except that the contents of the specified accumulator are inclusively ORed with the contents of the selected memory location.

Data Test Instructions

These are a powerful group of instructions that allow us to compare operands in several different ways. In previous units, you had experience comparing operands. The most frequently used method was to subtract one operand from another and test the result for zero or negative. In many cases, the numeric result of the subtraction was unimportant. We needed to know only if the result was zero or minus. The data test instructions allow us to make several different tests without actually producing an unwanted numeric result. These instructions are shown in Figure 5-13.

ACCUMULATOR AND MEMORY		BOOLEAN/ARITHMETIC OPERATION (All register labels	COND. CODE REG.					
			5	4	3	2	1	0
OPERATIONS	MNEMONIC	refer to contents)	H	I	N	Z	V	C
Bit Test	BITA	A • M	●	●	↕	↕	R	●
	BITB	B • M	●	●	↕	↕	R	●
Compare	CMPA	A – M	●	●	↕	↕	↕	↕
	CMPB	B – M	●	●	↕	↕	↕	↕
Compare Acmltrs	CBA	A – B	●	●	↕	↕	↕	↕
Test, Zero or Minus	TST	M – 00	●	●	↕	↕	R	R
	TSTA	A – 00	●	●	↕	↕	R	R
	TSTB	B – 00	●	●	↕	↕	R	R

Figure 5-13
Data test instructions.

The bit test instructions are very similar to the AND instructions. In both cases, the contents of the specified accumulator are ANDed with the contents of the selected memory location. The difference is that with the bit test instruction no logical product is produced. Neither the contents of the accumulator nor memory are altered in any way. However, the condition code registers are affected just as if the AND operation had taken place. Consider the BITA instruction. When executed, A is ANDed with M. If the result is 00_{16}, the Z register is set. Otherwise, the Z register is cleared. If the MSB of the result is 1, the N flag is set. However, the contents of the accumulator and memory are unaffected.

In the same way, the compare instructions are similar to subtract instructions except that the resulting numeric difference is ignored. For example, when the CMPA instruction is executed, the contents of the selected memory location are subtracted from the contents of accumulator A. The condition codes are affected just as if a difference had been produced. However, the original contents of accumulator A and memory are unaffected.

The compare accumulators instruction (CBA) works the same way. The condition codes are set as if the contents of B were subtracted from the contents A. However, the contents of the accumulators are unaffected.

Finally, the test for zero or minus instruction allows you to test the number in one of the accumulators or the memory to see if it is negative or zero. When this instruction is executed, the MPU looks at the number in question and sets the N and Z flags accordingly. The number itself is not changed.

Index Register and Stack Pointer Instructions

The index register and stack pointer are 16-bit registers. Figure 5-14 shows eleven instructions that allow us to control the operation of these registers. Because of the 16-bit format, the load, store, and compare instructions are slightly different from those discussed earlier.

INDEX REGISTER AND STACK			5	4	3	2	1	0
POINTER OPERATIONS	**MNEMONIC**	**BOOLEAN/ARITHMETIC OPERATION**	**H**	**I**	**N**	**Z**	**V**	**C**
Compare Index Reg	CPX	$(X_H/X_L) - (M/M+1)$	●	●	①	↕	②	●
Decrement Index Reg	DEX	$X - 1 \rightarrow X$	●	●	●	↕	●	●
Decrement Stack Pntr	DES	$SP - 1 \rightarrow SP$	●	●	●	●	●	●
Increment Index Reg	INX	$X + 1 \rightarrow X$	●	●	●	↕	●	●
Increment Stack Pntr	INS	$SP + 1 \rightarrow SP$	●	●	●	●	●	●
Load Index Reg	LDX	$M \rightarrow X_H, (M+1) \rightarrow X_L$	●	●	③	↕	R	●
Load Stack Pntr	LDS	$M \rightarrow SP_H, (M+1) \rightarrow SP_L$	●	●	③	↕	R	●
Store Index Reg	STX	$X_H \rightarrow M, X_L \rightarrow (M+1)$	●	●	③	↕	R	●
Store Stack Pntr	STS	$SP_H \rightarrow M, SP_L \rightarrow (M+1)$	●	●	③	↕	R	●
Indx Reg → Stack Pntr	TXS	$X - 1 \rightarrow SP$	●	●	●	●	●	●
Stack Pntr → Indx Reg	TSX	$SP + 1 \rightarrow X$	●	●	●	●	●	●

① (Bit N) Test: Sign bit of most significant (MS) byte of result = 1?

② (Bit V) Test: 2's complement overflow from subtraction of LS bytes?

③ (Bit N) Test: Result less than zero? (Bit 15 = 1)

Figure 5-14
Index register and stack pointer instructions.

The compare index register (CPX) instruction allows us to compare the 16-bit number in the index register with any two consecutive bytes in memory. Recall that the index register (X) will hold two bytes. The higher byte is identified as X_H while the lower byte is called X_L. When the CPX instruction is executed, X_H is compared with the 8-bit byte in the specified memory location (M). Also, X_L is compared with the byte immediately following the specified memory location (M+1). The comparison is the same as if M and M+1 were subtracted from X_H and X_L except that no numeric difference is produced. Neither X nor M is changed in any way. However, the N, Z, and V condition codes are affected as shown in Figure 5-14. Generally, the Z code is the one we are interested in since it tells us whether or not an exact match exists between the index register and the two bytes in memory.

The next four instructions are self-explanatory. They allow us to increment and decrement either the index register or the stack pointer. For one thing, these instructions allow us to maintain two separate 16-bit tallies simultaneously. However, the real value of these instructions and their associated registers will be discussed later.

The load and store instructions for the 16-bit registers are shown next in Figure 5-14. Since these are two byte registers, the LDX and LDS instructions must load two bytes from memory. In the case of the index register, the specified memory byte (M) is loaded into the upper half of the index register (X_H). An instant later, the next byte in memory (M+1) is automatically loaded into the lower half of the index register (X_L). Thus, the operation can be described as: $M \rightarrow X_H$, $(M+1) \rightarrow X_L$.

Because the stack pointer is also a 16-bit register, the load stack pointer instruction (LDS) works the same way. Its operation can be described as: $M \rightarrow SP_H$, $(M+1) \rightarrow SP_L$. Here, SP_H refers to the upper half of the stack pointer while SP_L refers to the lower half.

When the contents of the 16-bit registers are being stored, the operation is reversed. For example, the STX instruction stores X_H in M and X_L in M+1. A similar instruction, STS, allows us to store the contents of the stack pointer in the same way.

The final two instructions in this group allow us to transfer numbers between these two 16-bit registers. The TXS instruction loads the stack pointer with the contents of the index register **minus one**. The TSX instruction loads the index register with the contents of the stack pointer **plus one.** A more detailed discussion of these two important registers and their associated instructions will be given in the next unit.

Branch Instructions

The branch instructions are shown in Figure 5-15. Two additional instructions are also thrown in since they affect the program counter.

BRANCH OPERATIONS	MNEMONIC	BRANCH TEST	5 H	4 I	3 N	2 Z	1 V	0 C
Branch Always	BRA	None	●	●	●	●	●	●
Branch If Carry Clear	BCC	C = 0	●	●	●	●	●	●
Branch If Carry Set	BCS	C = 1	●	●	●	●	●	●
Branch If = Zero	BEQ	Z = 1	●	●	●	●	●	●
Branch If ⩾ Zero	BGE	$N \oplus V = 0$	●	●	●	●	●	●
Branch If > Zero	BGT	$Z + (N \oplus V) = 0$	●	●	●	●	●	●
Branch If Higher	BHI	$C + Z = 0$	●	●	●	●	●	●
Branch If ⩽ Zero	BLE	$Z + (N \oplus V) = 1$	●	●	●	●	●	●
Branch If Lower Or Same	BLS	$C + Z = 1$	●	●	●	●	●	●
Branch If < Zero	BLT	$N \oplus V = 1$	●	●	●	●	●	●
Branch If Minus	BMI	N = 1	●	●	●	●	●	●
Branch If Not Equal Zero	BNE	Z = 0	●	●	●	●	●	●
Branch If Overflow Clear	BVC	V = 0	●	●	●	●	●	●
Branch If Overflow Set	BVS	V = 1	●	●	●	●	●	●
Branch If Plus	BPL	N = 0	●	●	●	●	●	●
No Operation	NOP	Advances Prog. Cntr. Only	●	●	●	●	●	●
Wait for Interrupt	WAI		●	①	●	●	●	●

① (Bit I) Set when interrupt occurs. If previously set, a Non-Maskable Interrupt is required to exit the wait state.

Figure 5-15
Jump and branch instructions.

Nine of these instructions were discussed in the previous unit. These are: Branch Always (BRA); Branch If Carry Clear (BCC); Branch If Carry Set (BCS); Branch If Equal Zero (BEQ); Branch If Not Equal Zero (BNE); Branch If Minus (BMI); Branch If Plus (BPL); Branch If Overflow Clear (BVC); and Branch If Overflow Set (BVS).

Before we discuss the new branch instructions, here are some of the symbols we will be using. The symbol ($\geqslant$) means "is greater than or is equal to"; ($>$) means "is greater than"; ($\leqslant$) means "is less than or is equal to"; ($<$) means "is less than"; and ($\neq$) means "is not equal to."

Now consider the Branch If Greater Than or Equal instruction (BGE). This instruction is normally used after a subtract or compare instruction. It will cause a branch operation if the two's complement value in the accumulator is greater than or equal to the two's complement operand in memory. This condition is indicated by the N and V flags having the same value. The MPU determines if this condition is met by exclusively ORing N and V and examining the result.

Three simple examples may help illustrate the operation of this instruction. Let's start with a number in the accumlator that is greater than the operand in memory:

Number in Accumulator	=	00000010_2
Operand in Memory	=	00000001_2

When the operand is subtracted, the result is 00000001_2. With this result, both N and V are cleared to 0. Notice that N and V are equal and $N \oplus V = 0$. If the BGE instruction followed the subtract operation, the branch would be implemented.

Now see what happens when the number in the accumulator is equal to the operand:

Number in Accumulator	=	00000010_2
Operand in Memory	=	00000010_2

When the operand is subtracted, the result is 00000000_2. Again N and V are cleared to 0. Thus, N and V are still equal and $N \oplus V = 0$. Again, the BGE instruction would cause a branch to occur.

Finally, note what happens when the number in the accumulator is smaller:

Number in Accumulator	=	00000001_2
Operand in Memory	=	00000010_2

When the operand is subtracted, the result is 11111111_2. This time N is set but V is cleared. Thus, N and V are not equal. Therefore, $N \oplus V = 1$. In this case, the BGE conditions are not met and no branch will occur. The branch occurs if the two's complement value in the accumulator is greater than or equal to the two's complement operand in memory.

Next, consider the Branch If Greater Than (BGT) instruction. This instruction is normally used immediately after a subtract or compare operation. The branch will occur only if the two's complement minuend was greater than the two's complement subtrahend. By trying several examples as we did above, you will find that the branch conditions are met when $Z = 0$ and $N = V$.

The Branch If Higher (BHI) instruction is similar to the BGT instruction except that it is concerned with **unsigned** numbers. BHI is normally used after a subtract or compare operation. The branch will occur only if the unsigned minuend was greater than the unsigned subtrahend. By trying several different examples, you can prove that this occurs only when the C and Z flags are both 0.

The Branch If Less Than or Equal (BLE) instruction allows you to compare two's complement numbers in another way. If it is executed immediately after a subtract or compare operation, the branch will occur only if the two's complement minuend was less than or equal to the two's complement subtrahend.

The Branch If Lower Or Same (BLS) instruction is similar to the BLE instruction except that **unsigned** numbers are compared. When it is executed immediately after a subtract or compare operation, the branch will occur only if the unsigned minuend was lower than or equal to the unsigned subtrahend.

The Branch If Less Than Zero (BLT) instruction is also similar to the BLE instruction except that the equal qualification is removed. If BLT is executed immediately after a subtract or compare operation, the branch occurs only if the two's complement minuend was less than the two's complement subtrahend.

Two additional instructions are included in Figure 5-15. Although they are not branch instructions, they are included here since they do not seem to fit any of the other categories.

The No Operation (NOP) instruction is a "do-nothing" instruction that simply consumes a small increment of time. It does not change the contents of any register except the program counter. It does increment the program counter by one and consumes two MPU cycles. In spite of this, the NOP is a very useful instruction. When writing a program, we frequently use too many instructions. Once the program is loaded in memory, it is often inconvenient to simply remove an instruction. The hole left in memory can be filled by moving back all instructions that follow. However, a faster way is to simply fill the hole with one or more NOP instructions.

The Wait For Interrupt (WAI) instruction is the 6800's version of a HLT instruction. In earlier units we used this instruction at the end of all our programs. We will continue to use it in the same manner in the future. However, as you will see in the next unit, there is more involved in executing the WAI instruction than simply stopping the MPU. For now, though, continue to think of the WAI as a simple halt instruction.

Condition Code Register Instructions

The 6800 MPU has eight instructions that allow us direct access to the condition codes. These are listed in Figure 5-16.

CONDITION CODE REGISTER OPERATIONS	MNEMONIC	BOOLEAN OPERATION	5 H	4 I	3 N	2 Z	1 V	0 C
Clear Carry	CLC	0→C	●	●	●	●	●	R
Clear Interrupt Mask	CLI	0→I	●	R	●	●	●	●
Clear Overflow	CLV	0→V	●	●	●	●	R	●
Set Carry	SEC	1→C	●	●	●	●	●	S
Set Interrupt Mask	SEI	1→I	●	S	●	●	●	●
Set Overflow	SEV	1→V	●	●	●	●	S	●
Acmltr A→CCR	TAP	A→CCR	——— ① ———					
CCR→Acmltr A	TPA	CCR→A	●	●	●	●	●	●

R = Reset
S = Set
● = Not affected
① (ALL) Set according to the contents of Accumulator A.

Figure 5-16
Condition code register instructions.

Most of these instructions are self-explanatory. The Clear Carry (CLC) instruction resets the C flag to 0 while the Set Carry (SEC) sets it to 1. In the same way, the CLV and SEV instructions allow us to clear and set the overflow flag. Also, the CLI and SEI instructions can be used to clear or set the interrupt flag.

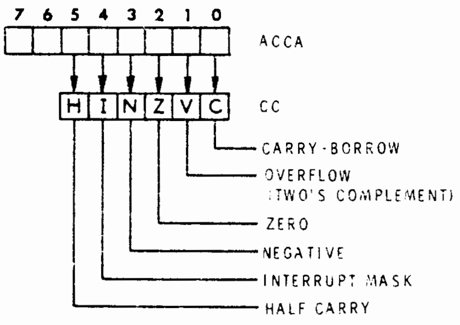


Figure 5-17
Executing the TAP instruction.

You will notice that there are no instructions for individually clearing the N, Z, or H flags. However, we can still set or clear these flags with the Transfer Accumulator A to the Processor Status Register (TAP) instruction. Figure 5-17 illustrates the execution of this instruction. The contents of bits 0 through 5 of accumulator A are transferred to the condition code registers. Thus, this instruction allows us to set or clear all the condition codes at once.

The final instruction is the Transfer Processor Status to Accumulator A (TPA) instruction. When this instruction is executed, the contents of the condition code registers are transferred to bits 0 through 5 of accumulator A. This operation is illustrated in Figure 5-18. Notice that bits 6 and 7 of the accumulator are set to 1.

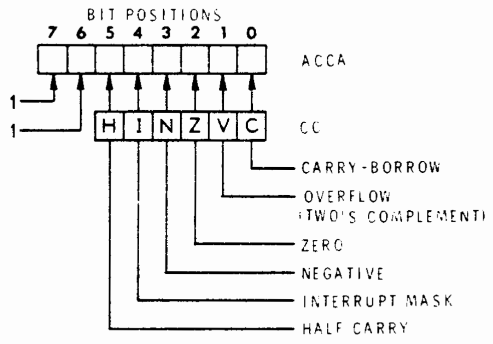


Figure 5-18
Executing the TPA instruction.

Summary of Instruction Set

As you can see, the 6800 MPU has a wide variety of instructions. In this section, most of the instructions have been mentioned briefly. However, a full explanation of some instructions must wait until additional new concepts have been covered.

In one short section, it is very difficult to cover every instruction in detail. And, it is virtually impossible for the reader to remember all the details of each instruction. Remember, all of the instructions available to the 6800 MPU are explained in detail in Appendix A of this program. Also, they are arranged alphabetically by their mnemonics for easy reference. Refer to Appendix A any time you are in doubt about what an instruction does. Be sure to look over the introductory material in the Appendix so that you understand all the conventions and symbols.

Self-Test Review

9. List the seven general categories of instructions.

10. What is meant by the shorthand notation: $A+B \rightarrow A$?

11. How is the C flag affected by the "add" and "add with carry" instructions?

12. Is the C flag changed when the AND instruction is executed?

13. Explain the difference between the NEG instruction and the COM instruction.

14. Explain the difference between the ANDA instruction and the BITA instruction.

15. The decimal adjust instruction is associated with which accumulator?

16. When the RORA instruction is executed the LSB of accumulator A is shifted into the ________________ register.

17. List eleven operations that can be performed directly to an operand in memory without first loading it into one of the MPU registers.

18. Explain the difference between the SUBB instruction and the CMPB instruction.

19. List the four types of logic operations that the 6800 MPU can perform.

20. When the LDX instruction is executed, from where is the index register loaded?

21. List four conditional branch instructions that are commonly used after a compare or subtract instruction to compare two's complement numbers.

22. Explain the difference between the BGT and BHI instructions.

23. Which instruction is often used to fill in a hole left in a program after an unwanted byte is removed?

24. Which instruction in the 6800 roughly corresponds to the halt instruction in our hypothetical machine?

25. Which of the condition codes can be individually set or cleared?

26. When you have some doubt as to exactly what operation is performed by a given instruction, where can you look to find the answer?

Answers

9. Arithmetic, data handling, logic, data test, index register and stack pointer, jump and branch, condition code.

10. Add the contents of accumulator A to the contents of accumulator B; transfer the result to accumulator A.

11. The C flag is set if a carry occurs; it is cleared otherwise.

12. No, the C flag is unaffected by the AND instruction.

13. The COM instruction replaces the operand with its 1's complement. The NEG instruction replaces the operand with its 2's complement.

14. With the ANDA instruction, the result of the AND operation is placed in accumulator A. With the BITA instruction, the condition code registers are set according to the result but the result is not retained.

15. The decimal adjust instruction works only with the A accumulator.

16. Carry (C).

17. A byte in memory can be: cleared, incremented, decremented, complemented, negated, rotated left, rotated right, shifted left arithmetically, shifted right arithmetically, shifted right logically, and tested.

18. With the SUBB instruction, a difference is produced and placed in accumulator B. With CMPB, the flags are set as if a difference were produced, but the difference is not retained.

19. Complement, AND, inclusive OR, and exclusive OR.

20. The upper half of the index register is loaded from the specified memory location; the lower half from the byte following the specified memory location.

21. BGE, BGT, BLE, BLT.

22. BGT is used to test the result of subtracting two's complement numbers. BHI is used to test the result of subtracting unsigned numbers.

23. NOP.

24. WAI.

25. C, I, and V.

26. Appendix A of this course.

NEW ADDRESSING MODES

In previous units, we have discussed four addressing modes. Let's briefly review these.

In the immediate addressing mode, the operand is the memory byte immediately following the opcode. These are generally two byte instructions. The first byte is the opcode, the second is the operand. However, there are exceptions to the two-byte rule. Some operations involve the 16-bit index register and stack pointer. In these cases, the operand is the **two** bytes immediately following the opcode. These are three byte instructions. The first byte is the opcode, the second and third are the operand.

In the direct addressing mode, the byte following the opcode is the address of the operand. These are always two byte instructions. The first byte is the opcode; the second is the address of the operand. An eight-bit byte can specify addresses from 00 to FF_{16}. Thus, when the direct addressing mode is being used, the operand must be in the first 256_{10} bytes of memory. Since the 6800 MPU can have up to $65{,}536_{10}$ bytes of memory, another means must be used to address the upper portion of memory.

The relative addressing mode is used for branching. These are two byte instructions. The first byte is the opcode, the second is the relative address. Recall that the relative address is added to the program count to form the absolute address. Since the 8-bit relative address is a two's complement number, the branch limits are $+127_{10}$ and -128_{10}.

In the inherent addressing mode either there is no operand or the operand is implied by the instruction. These are one byte instructions.

In this section, we will discuss two new addressing modes. These are called **extended addressing** and **indexed addressing**. We will discuss extended addressing first.

Extended Addressing

Extended addressing is similar to direct addressing but with one significant difference. Recall that with direct addressing the operand must be in the first 256_{10} bytes of memory. Since this represents less than one percent of the addresses available to the 6800 MPU, a more powerful addressing mode is needed. The extended addressing mode fills this need.

The format of an instruction that uses extended addressing is shown in Figure 5-19. The instruction will always have three bytes. The first byte is the opcode. The second and third bytes form a 16-bit address. Notice that the most significant part of the address is the byte immediately following the opcode. Since this instruction has a 16-bit address, the operand can be at any one of the $65,536_{10}$ possible addresses.

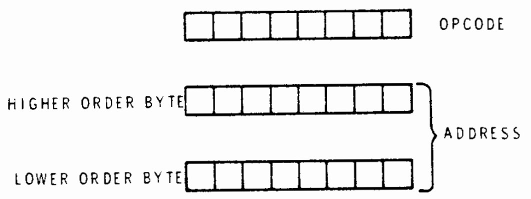


Figure 5-19
Format of an instruction that uses the extended addressing mode.

Suppose, for example, that you wish to load the operand at memory location 2134_{16} into accumulator B. The instruction would look like this:

F6	Opcode for LDAB extended
21	Higher order address
34	Lower order address

By the same token, if you wish to increment the number in memory location $AA00_{16}$, the instruction would be:

7C	Opcode for INC extended
AA	Higher order address
00	Lower order address

The extended addressing mode allows us to address an operand at any address including the first 256_{10} bytes of memory. Thus, if you wish to load the operand at address 0013_{16} into accumulator A, you can use extended addressing:

B6	Opcode for LDAA extended
00	Higher order address
13	Lower order address

Or, you can use direct addressing:

96	Opcode for LDAA direct
13	Address

Notice that, with direct addressing, the higher order address can be ignored since it is always 00. Because it saves one memory byte and one MPU cycle, direct addressing is normally used when the operand is in the first 256_{10} bytes of memory. Extended addressing is used when the operand is above address $00FF_{16}$. However, as you will see later, some instructions do not have a direct addressing mode. In these cases, extended addressing must be used even if the operand is in the first 256_{10} memory locations.

Indexed Addressing

The most powerful mode available to the 6800 MPU is indexed addressing. Recall that the 6800 MPU has a 16-bit index register. There are several instructions associated with this register. They allow us to load the register from memory and to store its contents in memory. Also, we can increment and decrement the index register. We can even compare its contents with two consecutive bytes in memory. These capabilities alone make the index register a very handy 16-bit counter. However, the real power of the index register comes from the fact that we can use this counter as an address pointer. Since this is a 16-bit register, it can point to any address in memory.

Purpose Before going into the details of how indexed addressing works, let's see why it is needed. Let's assume that we wish to add a list of 20_{16} numbers, and that the numbers are in 20_{16} consecutive memory locations starting at address 0050. Using the addressing modes discussed earlier, our program might look like this:

```
  CLRA     Clear Accumulator A.
  ADDA     Add the first number
   50      To accumulator A.
  ADDA     Add the second
   51      number to accumulator A.
  ADDA     Add the third number
   52      to accumulator A.
   •              •
   •              •
   •              •
  ADDA     Add the last number
   6F      to accumulator A.
  WAI      Wait.
```

While this accomplishes the desired result, it requires a long repetitive program. The above program would require 66_{10} bytes of memory. Notice that all the ADDA instructions are identical except that each successive address is one larger than the previous address. Indexed addressing can greatly simplify programs of this type.

Instruction Format The format of an instruction that uses indexed addressing is shown in Figure 5-20. Notice that this is a two-byte instruction. The first byte is the opcode, and the second is called an offset address. The offset address is an **unsigned** 8-bit binary number. It is added to the contents of the index register to determine the address at which the operand is located.

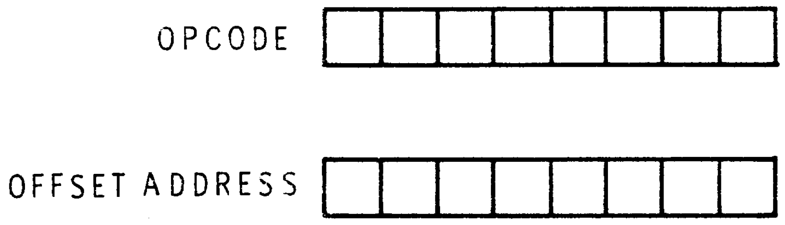


Figure 5-20
Format of an instruction that uses the indexed addressing mode.

Every instruction that involves an operand in memory can use the indexed addressing mode. In this unit, we will use the following convention to indicate indexed addressing:

LDAA, X
STAA, X
ADDB, X
etc.

In each case, the X tells us that indexed addressing is used. For example, the first instruction means: "using indexed addressing, load the contents of the specified memory location into accumulator A." Now let's see how the address of the operand is determined.

Determining the Operand Address When indexed addressing is being used, the address of the operand is determined by the offset address and the number in the index register. Specifically, the 8-bit offset address is added to the 16-bit address in the index register. The 16-bit sum becomes the address of the operand. Figure 5-21 illustrates this.

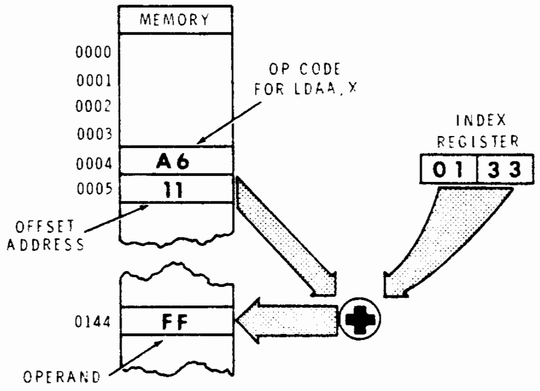


Figure 5-21
The operand address is formed by adding the offset address to the contents of the index register.

Here, the instruction in memory location 0004_{16} is LDAA, X. The offset address is 11_{16}. The contents of the index register are 0133_{16}. When the LDAA, X instruction is executed, the address of the operand is formed by adding the offset address to the number in the index register. In this case, the operand address will be:

$$\begin{array}{r} 0133_{16} \\ +\quad 11_{16} \\ \hline 0144_{16} \end{array}$$

The operand at this address is loaded into accumulator A. In this example, the operand FF is loaded into accumulator A when the instruction at location 0004 is executed. It is important to remember that this does not change the contents of the index register in any way. That is, the index register will still contain 0133_{16} after the instruction is executed.

Adding a List of Numbers To see how this addressing mode saves instructions, consider the problem given earlier. Recall that we were to add 20_{16} numbers stored in consecutive memory locations starting at address 0050. Using indexed addressing for the add instruction, our program looks like the one shown in Figure 5-22.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ HEX CONTENTS	COMMENTS
0010	CE	LDX #	Load index register immediate with the
0011	00	00	address of the
0012	50	50	first number in list.
0013	4F	CLRA	Clear accumulator A
0014	AB	ADDA, X	Add to accumulator A using indexed addressing
0015	00	00	with an offset address of 00.
0016	08	INX	Increment index register.
0017	8C	CPX #	Compare the contents of the index register
0018	00	00	with an address that is one greater than the
0019	70	70	address of the last number in the list.
001A	26	BNE	If not equal, branch back
001B	F8	F8	to the ADDA, X instruction.
001C	3E	WAI	Otherwise, halt.

Figure 5-22
Program for adding a list of 20_{16} numbers.

The first instruction is load index register immediate. Notice that a new symbol is used in this program. The symbol # is used to indicate the immediate addressing mode. Thus, the LDX# instruction causes the operand immediately following the opcode to be loaded into the index register. Recall that the index register can hold two 8-bit bytes. The operand is the two-byte number 0050_{16}. You may recognize that this is the address of the first number in the list of numbers that is to be added.

The next instruction clears accumulator A. The sum will be accumulated in this register, so it is important that it be cleared initially.

The third instruction (ADDA, X) is the only instruction in the program that uses indexed addressing. Notice that the symbol X indicates the indexed addressing mode. The offset address is 00. Recall that the operand address is determined by adding the offset to the contents of the index register. The index register contains 0050_{16} from a previous instruction. Since the offset is 00, the operand address is 0050_{16}. That is, the contents of memory location 0050 are added to the contents of accumulator A. Recall that 0050_{16} is the address of the first number in the list.

The fourth instruction increments the index register to 0051_{16}. Notice that the index register now points to the address of the second number in the list.

The fifth instruction compares the number in the index register with a number that is one greater than the address of the last number in the list.

If a match occurs, the Z flag will be set. Of course in this case, no match occurs yet. Notice once again that the symbol # indicates the immediate addressing mode. Thus, the contents of the index register are compared with the next two bytes in the program or 0070.

The BNE instruction tests the Z flag to see if the two numbers matched. If no match is indicated, the relative address (F8) directs the program back to the ADDA, X instruction. The first pass through the loop ends with the first number in accumulator A.

The second pass through the loop begins with the ADDA, X instruction being executed again. This time the index register points to address 0051. Therefore, the second number in the list is added to accumulator A. Accumulator A now contains the sum of the first two numbers. The index register is then incremented to 0052. Its contents are again compared with 0070. No match exists so the BNE instruction causes the loop to be repeated again.

The loop is repeated over and over again. Each time, the next number in the list is added to accumulator A. This process continues until the last number in the list is added. At that time, the index register will be incremented to 0070. Thus, when the CPX# instruction is executed, the Z flag will be set because the two numbers match. The BNE instruction recognizes that a match has occurred. Consequently, it does not allow the branch to occur and the next instruction in sequence is executed. Because this is the WAI instruction, the program halts. At this time, the sum of the 20_{16} numbers in the list will be in accumulator A.

Adding a list of numbers is a classic example of how indexing can be used to shorten a program. However, this example does not illustrate the full power of indexed addressing. For example, it does not illustrate the advantage of the offset address. Because indexed addressing is so important, let's look at another example.

Copying a List Let's assume we have a list of 10_{16} numbers that we wish to copy from one location to another. For simplicity, assume that the list is presently in addresses 0030 through 003F and that we wish to copy the list in location 0040 through 004F. Without using indexed addressing, our program might look like this:

```
LDAA
 30
STAA
 40
LDAA
 31
STAA
 41
  •
  •
  •
LDAA
 3F
STAA
 4F
WAI
```

As you have seen; long, repetitive programs such as this are excellent candidates for indexed addressing.

Using indexed addressing, our program might look like that shown in Figure 5-23. The first step is to load the index register with the first address in the original list. The LDAA, X instruction has an offset address of 00. Therefore, accumulator A is loaded from the address specified by the index register (0030). That is, the first number in the original list is loaded into accumulator A when the LDAA, X instruction is executed.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ HEX CONTENTS	COMMENTS
0010	CE	LDX #	Load index register immediate with
0011	00	00	the first address of the original
0012	30	30	list.
0013	A6	LDAA, X	Load accumulator A indexed with
0014	00	00	an offset of 00.
0015	A7	STAA, X	Store accumulator A indexed with
0016	10	10	an offset of 10_{16}.
0017	08	INX	Increment index register.
0018	8C	CPX #	Compare index with one greater
0019	00	00	than last
001A	40	40	address in original list.
001B	26	BNE	If not equal, branch back to the
001C	F6	F6	LDAA, X instruction.
001D	3E	WAI	Otherwise, halt.

Figure 5-23
Program for copying a list from addresses 0030 — 003F into addresses 0040 — 004F.

The STAA, X instruction illustrates the use of the offset address. Notice that the offset is 10. This number is added to the address in the index register to form the effective address at which the contents of accumulator A are stored. Thus, the contents of accumulator A are stored at address 0040. Remember, this does not change the number in the index register in any way. By using the offset, we can load the accumulator indexed from one address and store the accumulator indexed at another.

Next, the index register is incremented to 0031. It is then compared with 0040. Since no match exists, the BNE instruction directs the program back to the LDAA, X instruction. The loop is repeated until the entire list is rewritten in locations 0040 through 004F. After the last entry in the list is copied, the index register is incremented to 0040. Thus, the CPX# instruction sets the Z flag allowing the BNE instruction to divert the program from the loop. The program halts after the last entry in the list is written in its new position in memory.

Instruction Set Summary

You have now been introduced to most of the instructions available to the 6800 MPU. You have also been introduced to all of the addressing modes. Now let's look at the complete instruction set.

Figure 5-24 summarizes the 6800's instructions and addressing modes. This 2-page Figure contains a wealth of information. For your convenience, this information is repeated on the Instruction Set Summary card provided with the course. You should keep this card handy. After a while, you will be able to write long, complex programs using only the card for reference.

The left-hand column of Figure 5-24 lists the names and mnemonics for each of the instructions. In many cases, a single name such as "add" is associated with more than one mnemonic. For example, ADDA is an add operation that involves accumulator A while ADDB is an add operation that involves accumulator B.

The center column gives important information about the addressing modes. Notice that the ADDA instruction can have any one of four addressing modes: immediate, direct, indexed, or extended. Three facts are given for each addressing mode. The hexadecimal opcode is given in the OP column. For example, the opcode for ADDA immediate is 8B while the opcode for ADDA direct is 9B.

The column labeled (~) tells the number of MPU cycles required to execute the instruction. This information is important because it allows us to determine exactly how long it will take to run a given program. As you will see later, an MPU cycle is equal to one cycle of the MPU clock. For example, if the clock frequency is 1 MHz, one MPU cycle will be one microsecond. With this clock rate, 2 microseconds are required to execute the ADDA immediate instruction while 5 microseconds are required for the ADDA indexed instruction.

The column labeled (#) indicates the number of bytes required by the instruction. ADDA immediate, ADDA direct, and ADDA indexed are two-byte instructions while ADDA extended is a three-byte instruction.

The next column to the right gives the shorthand notation for the Boolean or arithmetic operations performed. Finally, the right-hand column indicates how the condition code registers are affected by each instruction.

If you study the instruction set carefully, you will find that there are a few instructions that we have not yet discussed. These will be described in the next unit.

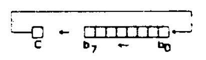
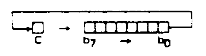
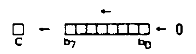
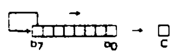
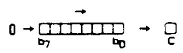
ACCUMULATOR AND MEMORY		ADDRESSING MODES: IMMED			DIRECT			INDEX			EXTND			INHER			BOOLEAN/ARITHMETIC OPERATION (All register labels refer to contents)	COND. CODE REG. 5	4	3	2	1	0
OPERATIONS	MNEMONIC	OP	~	#	OP	~	#	OP	~	#	OP	~	#	OP	~	#		H	I	N	Z	V	C
Add	ADDA	8B	2	2	9B	3	2	AB	5	2	BB	4	3				A + M → A	↕	●	↕	↕	↕	↕
	ADDB	CB	2	2	DB	3	2	EB	5	2	FB	4	3				B + M → B	↕	●	↕	↕	↕	↕
Add Acmltrs	ABA													1B	2	1	A + B → A	↕	●	↕	↕	↕	↕
Add with Carry	ADCA	89	2	2	99	3	2	A9	5	2	B9	4	3				A + M + C → A	↕	●	↕	↕	↕	↕
	ADCB	C9	2	2	D9	3	2	E9	5	2	F9	4	3				B + M + C → B	↕	●	↕	↕	↕	↕
And	ANDA	84	2	2	94	3	2	A4	5	2	B4	4	3				A • M → A	●	●	↕	↕	R	●
	ANDB	C4	2	2	D4	3	2	E4	5	2	F4	4	3				B • M → B	●	●	↕	↕	R	●
Bit Test	BITA	85	2	2	95	3	2	A5	5	2	B5	4	3				A • M	●	●	↕	↕	R	●
	BITB	C5	2	2	D5	3	2	E5	5	2	F5	4	3				B • M	●	●	↕	↕	R	●
Clear	CLR							6F	7	2	7F	6	3				00 → M	●	●	R	S	R	R
	CLRA													4F	2	1	00 → A	●	●	R	S	R	R
	CLRB													5F	2	1	00 → B	●	●	R	S	R	R
Compare	CMPA	81	2	2	91	3	2	A1	5	2	B1	4	3				A − M	●	●	↕	↕	↕	↕
	CMPB	C1	2	2	D1	3	2	E1	5	2	F1	4	3				B − M	●	●	↕	↕	↕	↕
Compare Acmltrs	CBA													11	2	1	A − B	●	●	↕	↕	↕	↕
Complement, 1's	COM							63	7	2	73	6	3				$\bar{M}$ → M	●	●	↕	↕	R	S
	COMA													43	2	1	$\bar{A}$ → A	●	●	↕	↕	R	S
	COMB													53	2	1	$\bar{B}$ → B	●	●	↕	↕	R	S
Complement, 2's (Negate)	NEG							60	7	2	70	6	3				00 − M → M	●	●	↕	↕	①	②
	NEGA													40	2	1	00 − A → A	●	●	↕	↕	①	②
	NEGB													50	2	1	00 − B → B	●	●	↕	↕	①	②
Decimal Adjust, A	DAA													19	2	1	Converts Binary Add. of BCD Characters into BCD Format	●	●	↕	↕	↕	③
Decrement	DEC							6A	7	2	7A	6	3				M − 1 → M	●	●	↕	↕	④	●
	DECA													4A	2	1	A − 1 → A	●	●	↕	↕	④	●
	DECB													5A	2	1	B − 1 → B	●	●	↕	↕	④	●
Exclusive OR	EORA	88	2	2	98	3	2	A8	5	2	B8	4	3				A ⊕ M → A	●	●	↕	↕	R	●
	EORB	C8	2	2	D8	3	2	E8	5	2	F8	4	3				B ⊕ M → B	●	●	↕	↕	R	●
Increment	INC							6C	7	2	7C	6	3				M + 1 → M	●	●	↕	↕	⑤	●
	INCA													4C	2	1	A + 1 → A	●	●	↕	↕	⑤	●
	INCB													5C	2	1	B + 1 → B	●	●	↕	↕	⑤	●
Load Acmltr	LDAA	86	2	2	96	3	2	A6	5	2	B6	4	3				M → A	●	●	↕	↕	R	●
	LDAB	C6	2	2	D6	3	2	E6	5	2	F6	4	3				M → B	●	●	↕	↕	R	●
Or, Inclusive	ORAA	8A	2	2	9A	3	2	AA	5	2	BA	4	3				A + M → A	●	●	↕	↕	R	●
	ORAB	CA	2	2	DA	3	2	EA	5	2	FA	4	3				B + M → B	●	●	↕	↕	R	●
Push Data	PSHA													36	4	1	A → M_{SP}, SP − 1 → SP	●	●	●	●	●	●
	PSHB													37	4	1	B → M_{SP}, SP − 1 → SP	●	●	●	●	●	●
Pull Data	PULA													32	4	1	SP + 1 → SP, M_{SP} → A	●	●	●	●	●	●
	PULB													33	4	1	SP + 1 → SP, M_{SP} → B	●	●	●	●	●	●
Rotate Left	ROL							69	7	2	79	6	3				M 	●	●	↕	↕	⑥	↕
	ROLA													49	2	1	A	●	●	↕	↕	⑥	↕
	ROLB													59	2	1	B	●	●	↕	↕	⑥	↕
Rotate Right	ROR							66	7	2	76	6	3				M 	●	●	↕	↕	⑥	↕
	RORA													46	2	1	A	●	●	↕	↕	⑥	↕
	RORB													56	2	1	B	●	●	↕	↕	⑥	↕
Shift Left, Arithmetic	ASL							68	7	2	78	6	3				M 	●	●	↕	↕	⑥	↕
	ASLA													48	2	1	A	●	●	↕	↕	⑥	↕
	ASLB													58	2	1	B	●	●	↕	↕	⑥	↕
Shift Right, Arithmetic	ASR							67	7	2	77	6	3				M 	●	●	↕	↕	⑥	↕
	ASRA													47	2	1	A	●	●	↕	↕	⑥	↕
	ASRB													57	2	1	B	●	●	↕	↕	⑥	↕
Shift Right, Logic	LSR							64	7	2	74	6	3				M 	●	●	R	↕	⑥	↕
	LSRA													44	2	1	A	●	●	R	↕	⑥	↕
	LSRB													54	2	1	B	●	●	R	↕	⑥	↕
Store Acmltr	STAA				97	4	2	A7	6	2	B7	5	3				A → M	●	●	↕	↕	R	●
	STAB				D7	4	2	E7	6	2	F7	5	3				B → M	●	●	↕	↕	R	●
Subtract	SUBA	80	2	2	90	3	2	A0	5	2	B0	4	3				A − M → A	●	●	↕	↕	↕	↕
	SUBB	C0	2	2	D0	3	2	E0	5	2	F0	4	3				B − M → B	●	●	↕	↕	↕	↕
Subract Acmltrs.	SBA													10	2	1	A − B → A	●	●	↕	↕	↕	↕
Subtr. with Carry	SBCA	82	2	2	92	3	2	A2	5	2	B2	4	3				A − M − C → A	●	●	↕	↕	↕	↕
	SBCB	C2	2	2	D2	3	2	E2	5	2	F2	4	3				B − M − C → B	●	●	↕	↕	↕	↕
Transfer Acmltrs	TAB													16	2	1	A → B	●	●	↕	↕	R	●
	TBA													17	2	1	B → A	●	●	↕	↕	R	●
Test, Zero or Minus	TST							6D	7	2	7D	6	3				M − 00	●	●	↕	↕	R	R
	TSTA													4D	2	1	A − 00	●	●	↕	↕	R	R
	TSTB													5D	2	1	B − 00	●	●	↕	↕	R	R

Figure 5-24

The 6800 instruction set.

INDEX REGISTER AND STACK		IMMED			DIRECT			INDEX			EXTND			INHER				5	4	3	2	1	0
POINTER OPERATIONS	MNEMONIC	OP	~	#	OP	~	=	OP	~	=	OP	~	#	OP	~	#	BOOLEAN/ARITHMETIC OPERATION	H	I	N	Z	V	C
Compare Index Reg	CPX	8C	3	3	9C	4	2	AC	6	2	BC	5	3				$(X_H/X_L) - (M/M+1)$	●	●	⑦	↕	⑧	●
Decrement Index Reg	DEX													09	4	1	$X - 1 \rightarrow X$	●	●	●	↕	●	●
Decrement Stack Pntr	DES													34	4	1	$SP - 1 \rightarrow SP$	●	●	●	●	●	●
Increment Index Reg	INX													08	4	1	$X + 1 \rightarrow X$	●	●	●	↕	●	●
Increment Stack Pntr	INS													31	4	1	$SP + 1 \rightarrow SP$	●	●	●	●	●	●
Load Index Reg	LDX	CE	3	3	DE	4	2	EE	6	2	FE	5	3				$M \rightarrow X_H, (M+1) \rightarrow X_L$	●	●	⑨	↕	R	●
Load Stack Pntr	LDS	8E	3	3	9E	4	2	AE	6	2	BE	5	3				$M \rightarrow SP_H, (M+1) \rightarrow SP_L$	●	●	⑨	↕	R	●
Store Index Reg	STX				DF	5	2	EF	7	2	FF	6	3				$X_H \rightarrow M, X_L \rightarrow (M+1)$	●	●	⑨	↕	R	●
Store Stack Pntr	STS				9F	5	2	AF	7	2	BF	6	3				$SP_H \rightarrow M, SP_L \rightarrow (M+1)$	●	●	⑨	↕	R	●
Indx Reg → Stack Pntr	TXS													35	4	1	$X - 1 \rightarrow SP$	●	●	●	●	●	●
Stack Pntr → Indx Reg	TSX													30	4	1	$SP + 1 \rightarrow X$	●	●	●	●	●	●

JUMP AND BRANCH		RELATIVE			INDEX			EXTND			INHER				5	4	3	2	1	0
OPERATIONS	MNEMONIC	OP	~	=	OP	~	=	OP	~	=	OP	~	#	BRANCH TEST	H	I	N	Z	V	C
Branch Always	BRA	20	4	2										None	●	●	●	●	●	●
Branch If Carry Clear	BCC	24	4	2										C = 0	●	●	●	●	●	●
Branch If Carry Set	BCS	25	4	2										C = 1	●	●	●	●	●	●
Branch If = Zero	BEQ	27	4	2										Z = 1	●	●	●	●	●	●
Branch If ≥ Zero	BGE	2C	4	2										$N \oplus V = 0$	●	●	●	●	●	●
Branch If > Zero	BGT	2E	4	2										$Z + (N \oplus V) = 0$	●	●	●	●	●	●
Branch If Higher	BHI	22	4	2										$C + Z = 0$	●	●	●	●	●	●
Branch If ≤ Zero	BLE	2F	4	2										$Z + (N \oplus V) = 1$	●	●	●	●	●	●
Branch If Lower Or Same	BLS	23	4	2										$C + Z = 1$	●	●	●	●	●	●
Branch If < Zero	BLT	2D	4	2										$N \oplus V = 1$	●	●	●	●	●	●
Branch If Minus	BMI	2B	4	2										N = 1	●	●	●	●	●	●
Branch If Not Equal Zero	BNE	26	4	2										Z = 0	●	●	●	●	●	●
Branch If Overflow Clear	BVC	28	4	2										V = 0	●	●	●	●	●	●
Branch If Overflow Set	BVS	29	4	2										V = 1	●	●	●	●	●	●
Branch If Plus	BPL	2A	4	2										N = 0	●	●	●	●	●	●
Branch To Subroutine	BSR	8D	8	2										} See Special Operations	●	●	●	●	●	●
Jump	JMP				6E	4	2	7E	3	3					●	●	●	●	●	●
Jump To Subroutine	JSR				AD	8	2	BD	9	3					●	●	●	●	●	●
No Operation	NOP										01	2	1	Advances Prog. Cntr. Only	●	●	●	●	●	●
Return From Interrupt	RTI										3B	10	1	} See special Operations	—	—	⑩	—	—	—
Return From Subroutine	RTS										39	5	1		●	●	●	●	●	●
Software Interrupt	SWI										3F	12	1		●	S	●	●	●	●
Wait for Interrupt	WAI										3E	9	1		●	⑪	●	●	●	●

CONDITIONS CODE REGISTER		INHER				5	4	3	2	1	0
OPERATIONS	MNEMONIC	OP	~	=	BOOLEAN OPERATION	H	I	N	Z	V	C
Clear Carry	CLC	0C	2	1	$0 \rightarrow C$	●	●	●	●	●	R
Clear Interrupt Mask	CLI	0E	2	1	$0 \rightarrow I$	●	R	●	●	●	●
Clear Overflow	CLV	0A	2	1	$0 \rightarrow V$	●	●	●	●	R	●
Set Carry	SEC	0D	2	1	$1 \rightarrow C$	●	●	●	●	●	S
Set Interrupt Mask	SEI	0F	2	1	$1 \rightarrow I$	●	S	●	●	●	●
Set Overflow	SEV	0B	2	1	$1 \rightarrow V$	●	●	●	●	S	●
Acmltr A → CCR	TAP	06	2	1	$A \rightarrow CCR$	—	—	⑫	—	—	—
CCR → Acmltr A	TPA	07	2	1	$CCR \rightarrow A$	●	●	●	●	●	●

CONDITION CODE REGISTER NOTES:

(Bit set if test is true and cleared otherwise)

①	(Bit V)	Test: Result = 10000000?
②	(Bit C)	Test: Result = 00000000?
③	(Bit C)	Test: Decimal value of most significant BCD Character greater than nine? (Not cleared if previously set.)
④	(Bit V)	Test: Operand = 10000000 prior to execution?
⑤	(Bit V)	Test: Operand = 01111111 prior to execution?
⑥	(Bit V)	Test: Set equal to result of $N \oplus C$ after shift has occurred.
⑦	(Bit N)	Test: Sign bit of most significant (MS) byte of result = 1?
⑧	(Bit V)	Test: 2's complement overflow from subtraction of LS bytes?
⑨	(Bit N)	Test: Result less than zero? (Bit 15 = 1)
⑩	(All)	Load Condition Code Register from Stack. (See Special Operations)
⑪	(Bit I)	Set when interrupt occurs. If previously set, a Non-Maskable Interrupt is required to exit the wait state.
⑫	(ALL)	Set according to the contents of Accumulator A.

LEGEND:

OP	Operation Code (Hexadecimal);	00	Byte = Zero;
~	Number of MPU Cycles;	H	Half-carry from bit 3;
=	Number of Program Bytes;	I	Interrupt mask
+	Arithmetic Plus;	N	Negative (sign bit)
−	Arithmetic Minus;	Z	Zero (byte)
•	Boolean AND;	V	Overflow, 2's complement
M_{SP}	Contents of memory location pointed to be Stack Pointer,	C	Carry from bit 7
		R	Reset Always
		S	Set Always
+	Boolean Inclusive OR;	↕	Test and set if true, cleared otherwise
⊕	Boolean Exclusive OR;	●	Not Affected
$\overline{M}$	Complement of M;	CCR	Condition Code Register
→	Transfer Into;	LS	Least Significant
0	Bit = Zero;	MS	Most Significant

Figure 5-24
(continued)

Self-Test Review

27. A disadvantage of direct addressing is that the operand must be in the first ____________ bytes of memory.

28. The advantage of direct addressing is that only ____________ bytes are required for each instruction.

29. Extended addressing can address ____________ bytes of memory.

30. A disadvantage of extended addressing is that each instruction requires ____________ bytes.

31. Can extended addressing be used to address an operand in the first 256_{10} bytes of memory?

32. The most powerful addressing mode available to the 6800 is called ____________ addressing.

33. Indexed addressing requires ____________ bytes for each instruction.

34. The second byte of an indexed addressing instruction is called the ____________ address.

35. How is the address of the operand determined when indexed addressing is used?

36. Carefully examine the program shown in Figure 5-25. Determine what the program does and fill in the comments column. What number is loaded into the index register by the first instruction?

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ HEX CONTENTS	COMMENTS
0010	CE	LDX #	
0011	00	00	
0012	50	50	
0013	6F	→ CLR, X	
0014	00	00	
0015	08	INX	
0016	8C	CPX #	
0017	00	00	
0018	60	60	
0019	26	BNE	
001A	F8	F8	
001B	3E	WAI	

Figure 5-25
Program for Self-Test Review

37. What location is cleared by the CLR, X instruction?

38. What is the number in the index register after the INX instruction is executed for the first time?

39. The loop will be repeated until the number in the index register is ________________.

40. What does this program do?

41. Refer to Figure 5-24. What is the hexadecimal opcode for the LDAB extended instruction?

42. How many MPU cycles are required by the INC, X instruction?

43. How many bytes in the LDS # instruction?

Answers

27. 256_{10}.

28. Two.

29. $65{,}536_{10}$.

30. Three.

31. Yes. Although direct addressing is normally used when the operand is in the first 256_{10} bytes of memory, extended addressing can be used also.

32. Indexed.

33. Two.

34. Offset.

35. The offset address is added to the contents of the index register.

36. 0050_{16}.

37. 0050_{16}.

38. 0051_{16}.

39. 0060_{16}.

40. The program clears memory locations 0050_{16} through $005F_{16}$.

41. $F6_{16}$.

42. Seven.

43. Three.

EXPERIMENTS

Perform Programming Experiments 7 and 8. 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. Which of the following program segments will **not** clear both accumulators?

   A. CLRA
      CLRB

   B. CLRA
      TAB

   C. CLRB
      TBA

   D. CLRA
      ABA

2. Which of the following contains an operation that can **not** be performed directly on a byte in memory using a single instruction?

   A. Increment, decrement, shift left arithmetically.
   B. Clear, complement, compare.
   C. Rotate left, negate, test for zero.
   D. Shift right logically, rotate right, test for minus.

3. Which addressing mode is best suited for adding a list of numbers?

   A. Direct.
   B. Extended.
   C. Indexed.
   D. Relative.

4. Which of the following program segments will sucessfully swap the contents of accumulators A and B?

A.
```
TAB
TBA
```

B.
```
STAA
10
TBA
LDAB
10
```

C.
```
TAB
ABA
```

D.
```
STAA
10
LDAB
10
TBA
```

5. Which of the following program segments will cause a branch if the number in memory location 8310 is odd?

A.
```
ROR
83
10
BCS
07
```

B.
```
ASL
83
10
BCS
07
```

C.
```
RORA
BCS
07
```

D.
```
LDAA
83
10
ROLA
BCS
07
```

6. Examine the following program segment:

```
   CLRA
┌→ INCA
│  BNE
└─ FD
   WAI
```

If an MPU cycle is 1 microsecond, how much time elapses from the time this segment starts running until the WAI instruction is fetched?

A. Approximately 8 microseconds.
B. Approximately 2050 microseconds.
C. Approximately 1538 microseconds.
D. Approximately 3 microseconds.

7. Which of the following instructions can be used to clear the Z flag?

   A. BEQ
   B. BNE
   C. NOP
   D. TAP

8. Which of the following instructions can be used to test the result of the subtraction of unsigned binary numbers?

   A. BGE.
   B. BGT.
   C. BCS.
   D. BLT

9. Examine the following program segment:

```
   LDX #
      00
      50
┌─► DEX
│    BNE
└──── FD
     WAI
```

   How many times will the DEX instruction be executed?

   A. Once
   B. 50_{16} times.
   C. $65{,}536_{10}$.
   D. The number of times will depend on the contents of memory location 0050.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ HEX CONTENTS	COMMENTS
0010	4F	CLRA	Clear Accumulator A.
0011	7D	TST	Test
0012	00	00	the
0013	1E	1E	multiplier.
0014	27	BEQ	If it is zero branch to wait.
0015	07	07	
0016	7A	DEC	Otherwise decrement
0017	00	00	the
0018	1E	1E	multiplier.
0019	9B	ADDA	Add the
001A	1F	1F	multiplicand to the product.
001B	20	BRA	Repeat the loop.
001C	F4	F4	
001D	3E	WAI	Wait.
001E	05	Multiplier	
001F	04	Multiplicand	

Figure 5-26
This program multiplies by repeated addition.

NOTE: Refer to the program shown in Figure 5-26 for questions 10 through 16.

10. What addressing mode does the TST instruction use?

A. Immediate
B. Direct.
C. Extended.
D. Indexed.

11. The BEQ instruction checks to see if the TST instruction set the:

A. Z flag
B. C flag.
C. H flag.
D. V flag.

12. The DEC instruction decrements the number in:

A. Accumulator A.
B. Memory location 001E.
C. Accumulator B.
D. The index register.

13. Which instruction is executed immediately after the BRA instruction?

A. WAI.
B. BEQ.
C. CLRA.
D. TST.

14. With the values given for the multiplier and multiplicand, how many times will the main program loop be repeated?

A. Four times.
B. Five times.
C. Twenty times.
D. Twice.

15. After the program has been executed, memory location 001E will contain:

A. 05_{16}.
B. 04_{16}.
C. 20_{16}.
D. 00_{16}.

16. After the program has been executed, the product will appear in:

A. Memory location 001E.
B. Memory location 001F.
C. Accumulator A.
D. Accumulator B.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ HEX CONTENTS	COMMENTS
0010	CE	LDX #	
0011	00	00	
0012	00	05	
0013	A6	LDAA, X	
0014	20	20	
0015	AB	ADDA, X	
0016	30	30	
0017	A7	STAA, X	
0018	40	40	
0019	08	INX	
001A	8C	CPX #	
001B	00	00	
001C	15	15	
001D	26	BNE	
001E	F4	F4	
001F	3E	WAI	

Figure 5-27
Program for Questions 17 through 20.

NOTE: Refer to Figure 5-27 for Questions 17 through 20. Analyze the program, determine what it does, and fill in appropriate comments.

17. On the first pass through the main program loop, the LDAA, X instruction takes its operand from memory location:

A. 0005.
B. 0020.
C. 0025.
D. 0014.

18. On the first pass, the ADDA, X adds the contents of what memory location to accumulator A?

A. 0005.
B. 0030.
C. 0035.
D. 0016.

19. On the second pass through the program loop, the contents of memory location:

A. 0021 are added to the contents of 0031 and the result is stored in 0041.
B. 0026 are added to the contents of 0036 and the result is stored in 0046.
C. 0025 are added to the contents of 0035 and the result is stored in 0045.
D. 0020 are added to the contents of 0030 and the result is stored in 0040.

20. How many times is the main program loop repeated?

A. 10_{16} times.
B. 05_{16} times.
C. 30_{16} times.
D. 15_{16} times.