



# Individual Learning Program

## MICROPROCESSORS

*Unit 6*

**THE 6800 MICROPROCESSOR — PART 2**

EE-3401

HEATH COMPANY  
BENTON HARBOR, MICHIGAN 49022

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

## CONTENTS

Introduction .....	6-3
Unit Objectives .....	6-4
Unit Activity Guide .....	6-5
Stack Operations .....	6-6
Subroutines .....	6-17
Input-Output (I/O) Operations .....	6-27
Interrupts .....	6-37
Experiments .....	6-50
Unit Examination .....	6-51
Examination Answers .....	6-53

## *Unit 6*

# **THE 6800 MICROPROCESSOR — PART 2**

## **INTRODUCTION**

In the previous unit, you were introduced to the architecture and instruction set of the 6800 microprocessor. Much of the MPU's capabilities were discussed; however, three important areas were omitted. These include the microprocessor's stack operation, the use of subroutines, and the interrupt capability. These capabilities are discussed in detail in this unit. You are also introduced to input-output operations.

## UNIT OBJECTIVES

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

1. Explain the difference between a cascade stack and a memory stack.
2. Write simple programs that can store data in — and retrieve data from — the stack.
3. Write programs that use the stack and indexing to move a list from one place in memory to another.
4. Explain the operations performed by each of the following instructions: PULA, PULB, PSHA, PSHB, DES, INS, LDS, STS, TXS, and TSX.
5. Define stack, subroutine, nested subroutine, interrupt, interrupt vector, and interrupt masking.
6. Write programs that use subroutines and nested subroutines.
7. Explain the operations performed by each of the following instructions: JMP, JSR, BSR, and RTS.
8. Describe how the 6800 MPU performs input and output operations.
9. Draw flowcharts depicting the sequence of events that occur during reset, non-maskable interrupt, interrupt request, software interrupt, return from interrupt, and wait for interrupt.
10. Explain the operation performed by each of the following instructions: WAI, SWI, RTI, SEI, and CLI.

**UNIT ACTIVITY GUIDE****Completion  
Time**

- |   |       |
|---|-------|
| <input type="checkbox"/> Read Section on Stack Operations.          | _____ |
| <input type="checkbox"/> Complete Self-Test Review Questions 1-10.  | _____ |
| <input type="checkbox"/> Read Section on Subroutines.               | _____ |
| <input type="checkbox"/> Complete Self-Test Review Questions 11-20. | _____ |
| <input type="checkbox"/> Read Section on Input-Output Operations.   | _____ |
| <input type="checkbox"/> Complete Self-Test Review Questions 21-27. | _____ |
| <input type="checkbox"/> Read Section on Interrupts.                | _____ |
| <input type="checkbox"/> Complete Self-Test Review Questions 28-40. | _____ |
| <input type="checkbox"/> Perform Programming Experiments 9 and 10.  | _____ |
| <input type="checkbox"/> Complete Unit Examination.                 | _____ |
| <input type="checkbox"/> Check Examination Answers.                 | _____ |

## STACK OPERATIONS

In computer jargon, a *stack* is a group of temporary storage locations in which data can be stored and later retrieved. In this regard, a *stack* is somewhat like memory. In fact, many microprocessors use a section of memory as a stack. The difference between a stack and other forms of memory is the method by which the data is accessed or addressed. The discussion will begin by considering a simple stack arrangement used in some microprocessors. Then the more sophisticated stack arrangement used by the 6800 MPU will be discussed.

### Cascade Stack

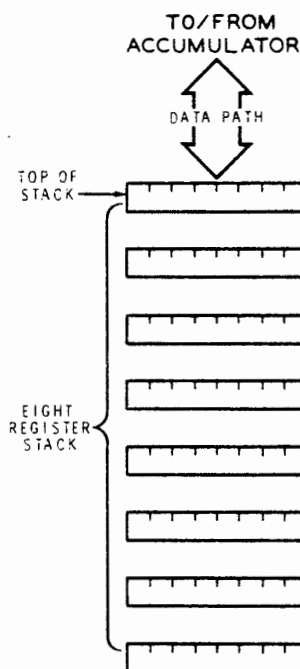


Figure 6-1.  
A cascade stack.

Some microprocessors have a special group of registers (usually 8 or 16) called a *cascade stack*. Each register can hold one 8-bit byte of data. Because these registers are right on the MPU chip, they make excellent temporary storage locations. If we need to free the accumulator for some reason, we can store its contents in the stack. Later, if that piece of data is needed again, we can retrieve the data from the stack. Of course, we could also have freed the accumulator by storing the data in memory. What then is the advantage of the stack?

One advantage of the stack is the method by which it is accessed or addressed. Recall that when a byte is stored in memory, an address is required. That is to store the contents of the accumulator in memory a 2-byte or 3-byte instruction is required. Depending on the addressing mode, the last one or two bytes is the address. Later, if the byte is retrieved, another instruction is required that also has an address.

An advantage of the stack is that data can be stored into it or read from it with single-byte instructions. That is, the instructions used with the stack do not require an address. Therefore, they are single-byte instructions.

Figure 6-1 shows an 8-register stack similar to that found in some microprocessors. This is called a cascade stack because of the method by which data is loaded and retrieved. All data transfers are between the top of the stack and the accumulator. That is, the accumulator communicates only with the top location on the stack. Data is transferred to the stack by a special instruction called PUSH.

**The PUSH Instruction.** Figure 6-2 illustrates how the PUSH instruction places data in the stack. The number  $01_{16}$  is in the accumulator and we wish to temporarily store it. While we could store the number in memory, this would require a 2-byte or a 3-byte instruction. So instead, we use the PUSH instruction to place this number in the stack. Notice that the number is placed in the top location of the stack as shown in Figure 6-2A. The number remains there until we retrieve it or until we push another byte into the stack.

Figure 6-2B shows what happens if, at some time later, we push another byte into the stack. Notice that the accumulator now contains  $03_{16}$ . If the PUSH instruction is executed, the contents of the accumulator are pushed into the top of the stack. To make room for this new number, the original number  $01_{16}$  is pushed deeper into the stack.

Figures 6-2C and 6-2D show two more numbers being pushed into the stack at later points in the program. Notice that new data is always pushed into the top of the stack. To make room for the new data, the old data is pushed deeper into the stack. For this reason, this arrangement is often called a *push-down* or *cascade stack*. The name *cascade stack* comes from the characteristic cascading of data down through the stack as each new byte is pushed in at the top.

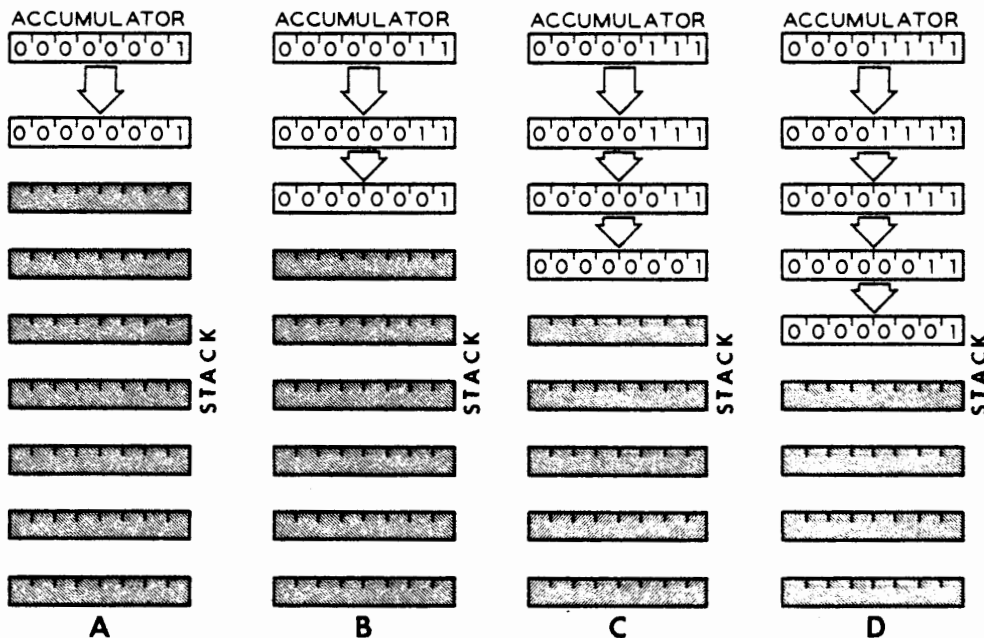


Figure 6-2.  
Pushing data into the stack.

**The PULL Instruction.** The MPU retrieves data from the stack by using the PULL instruction. In some microprocessors, this is referred to as a POP instruction.

Figure 6-3 illustrates how data can be pulled (or popped) from the stack. Figure 6-3A shows the stack as it appeared after the last push operation. Notice that it contains four bytes of data. The last byte of data that was entered is at the top of the stack.

The PULL instruction retrieves the byte that is at the top of the stack. As this byte is removed from the stack, all other bytes move up, filling in the space left by that byte. Figure 6-3B illustrates how  $0F_{16}$  is pulled from the stack. Notice that  $07_{16}$  is now at the top of the stack.

Figures 6-3C and 6-3D show how the next two bytes can be pulled from the stack. In each case, the remaining bytes move up in the stack, filling in the register vacated by the removed byte.

If you compare Figures 6-2 and 6-3, you will notice that the data must be pulled from the stack in the reverse order. That is, the last byte pushed into the stack is the first byte that is pulled from the stack. Another name for this arrangement is a last-in/first-out (LIFO) stack.

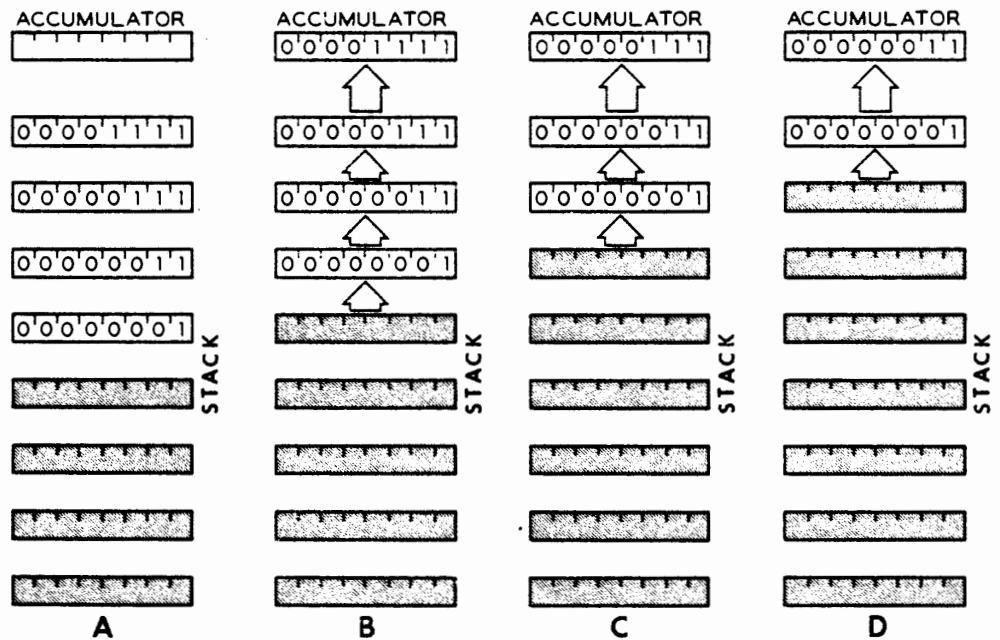


Figure 6-3.  
Pulling data from the stack.



## Memory Stack

While a cascade stack is valuable, it does have some limitations. For one thing, the number of registers is generally quite limited, with eight being typical. If more than eight pieces of data are pushed into the stack, the "older" bytes are pushed out the bottom and are lost. Also, the readout of the stack is destructive. When a byte is pulled from the stack, it no longer exists in the stack. This is fundamentally different from reading a byte from memory.

Because of these limitations the 6800 MPU does not use a cascade stack. Instead, a section of RAM can be set aside by the programmer to act as a stack. This has several advantages. First, the stack can be any length that the programmer requires. Second, the programmer can set up more than one stack if he likes. Third he can address the data in the stack using any of the instructions that address memory.

**Stack Pointer.** Recall that the 6800 MPU has a 16-bit register called the stack pointer. In a memory-type stack, the stack pointer defines the memory location that acts as the top of the stack.

The cascade stack considered earlier generally does not require a stack pointer. The top of the stack is determined by hardware. During push and pull operations, the data bytes actually move from one register to another. That is, the top of the stack remains stationary and the data moves up or down in relation to the stack.

In the memory stack, data cannot be easily transferred from one location to the next. Therefore, instead of moving data up and down in relation to the stack, it is much easier to move the top of the stack in relation to the data.

Generally, when the microprocessor-based system is being planned, a section of RAM is reserved for the stack. This should be a section of RAM that is not being used for any other purpose.

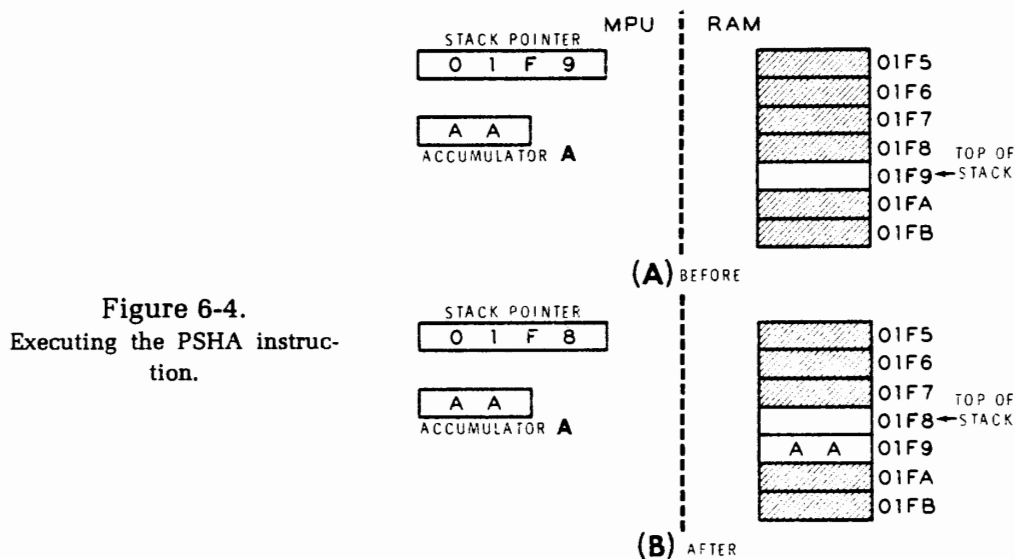
Once this is done, the stack can be set up by a program. The top of the stack is established by loading an address into the stack pointer. For example, suppose we wish to establish address  $01F9_{16}$  as the top of the stack. The following instruction could be used:

```
LDS#  
01  
F9
```

This loads the address  $01F9_{16}$  into the stack pointer and establishes that address as the top of the stack. However, as you will see, the top of the stack moves each time data is pushed into — or pulled from — the stack.

**The PUSH Instructions.** The 6800 MPU has two push instructions, PSHA and PSHB. These single-byte instructions push the contents of their respective accumulator onto the stack.

Figure 6-4 shows the effects of the PSHA instruction. Before the instruction is executed, the stack pointer contains the address  $01F9_{16}$  as a result of a previous LDS instruction. Accumulator A contains a data byte ( $AA_{16}$ ). If the PSHA instruction is now executed, the contents of accumulator A are pushed into memory location  $01F9_{16}$ . Then, the stack pointer is automatically decremented to  $01F8_{16}$ . This automatically moves the top of the stack as shown.



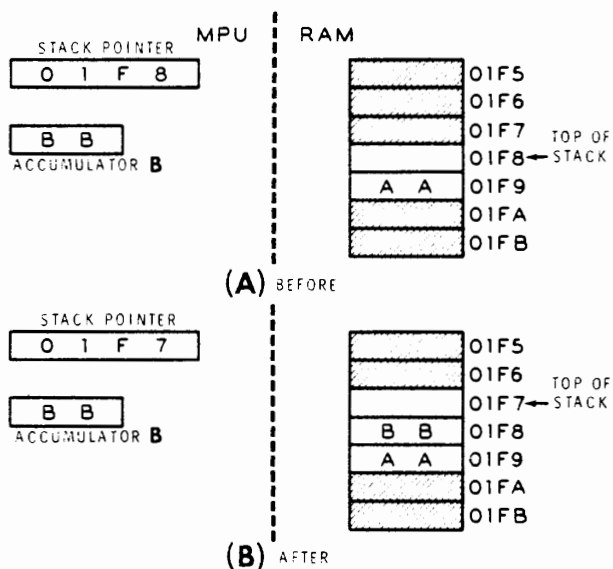
If you look at your Instruction Set Summary card, you will see that the operation is described as follows:

$$A \rightarrow M_{SP}, SP - 1 \rightarrow SP$$

This means that the contents of the A accumulator are transferred to the memory location specified by the stack pointer. Also, the contents of the stack pointer are replaced by the previous contents of the stack pointer minus one. In other words, after the accumulator-to-stack transfer takes place, the stack pointer is decremented by one.

To reinforce the idea, assume that at some later point in the program, the MPU executes a PSHB instruction. This is illustrated in Figure 6-5. Before PSHB is executed, the B accumulator contains  $BB_{16}$  and the stack pointer is still pointing to  $01F8_{16}$ . When PSHB is executed, the contents of accumulator B are pushed onto the stack and the stack pointer is decremented to  $01F7_{16}$ .

**The PULL Instructions.** Data bytes are removed from the stack with the pull instruction. The 6800 MPU has two pull instructions. PULA allows the MPU to pull data from the stack into the A accumulator. PULB performs a similar operation except the data byte goes into accumulator B. In each case, data is pulled from the top of the stack. Thus, the data byte available to the MPU is the last byte that was placed in the stack.

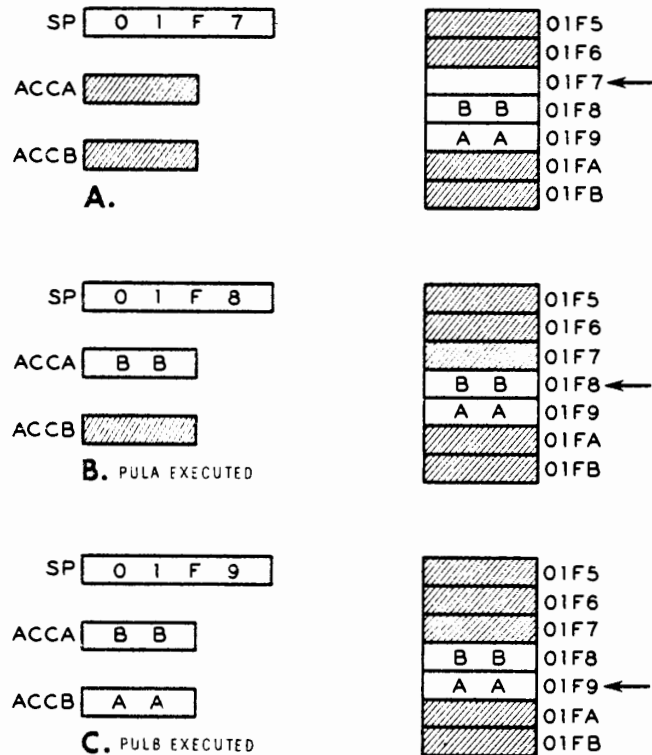


**Figure 6-5.**  
Executing the PSHB instruction.

For example, Figure 6-6A shows the stack as we left it after the last push instruction. Figure 6-6B shows what happens if the PULA instruction is executed. First, the stack pointer is automatically incremented by one to 01F8<sub>16</sub>. Then the contents of the memory location designated by the stack pointer are transferred to accumulator A. Thus, BB<sub>16</sub> goes into accumulator A. Notice that the stack pointer is incremented *before* the byte is pulled from the stack.

To be certain you have the idea, consider what happens if the PULB instruction is now executed. Figure 6-6C shows that the stack pointer is automatically incremented to 01F9<sub>16</sub>. The contents of that location are then pulled into accumulator B. This operation is described on your Instruction Set Summary card as:

$$SP + 1 \rightarrow SP, M_{SP} \rightarrow B.$$



**Figure 6-6.**  
Executing PULL instructions.

**Using the Stack.** Figure 6-7 summarizes all of the instructions that directly affect stack operations. The push and pull instructions were introduced in this unit while the other instructions were discussed briefly in the previous unit. Find these instructions on your Instruction Set Summary card. The push and pull instructions are listed with the Accumulator and Memory Operations. Those instructions that affect the stack pointer are listed under Index Register and Stack Pointer Operations.

**ADDRESSING MODES**

STACK AND STACK POINTER OPERATIONS		ADDRESSING MODES															BOOLEAN/ARITHMETIC OPERATION (All register labels refer to contents)
		IMMED			DIRECT			INDEX			EXTND			INHER			
OPERATIONS	MNEMONIC	OP	~	#	OP	~	#	OP	~	=	OP	~	#	OP	~	#	
Push Data	PSHA													36	4	1	A → M <sub>SP</sub> , SP - 1 → SP B → M <sub>SP</sub> , SP - 1 → SP SP + 1 → SP, M <sub>SP</sub> → A SP + 1 → SP, M <sub>SP</sub> → B  SP - 1 → SP SP + 1 → SP M → SP <sub>H</sub> , (M + 1) → SP <sub>L</sub> SP <sub>H</sub> → M, SP <sub>L</sub> → (M + 1) X - 1 → SP SP + 1 → X
	PSHB													37	4	1	
Pull Data	PULA													32	4	1	
	PULB													33	4	1	
Decrement Stack Pntr	DES													34	4	1	
Increment Stack Pntr	INS													31	4	1	
Load stack Pntr	LDS	8E	3	3	9E	4	2	AE	6	2	BE	5	3				
Store Stack Pntr	STS				9F	5	2	AF	7	2	BF	6	3				
Indx Reg → Stack Pntr	TXS													35	4	1	
Stack Pntr → Indx Reg	TSX													30	4	1	

Following are some examples of how the stack can be used. First consider a trivial example. Using only stack operations, swap the contents of accumulators A and B. Assuming the stack pointer has already been set up, the program segment might look like this:

```

PSHA
PSHB
PULA
PULB
    
```

Assume that accumulator A initially contains AA<sub>16</sub> and that accumulator B contains BB<sub>16</sub>. The first instruction pushes AA<sub>16</sub> onto the stack. Next BB<sub>16</sub> is pushed onto the stack. The third instruction pulls BB<sub>16</sub> from the top of the stack and places it in accumulator A. Finally, the last instruction pulls AA<sub>16</sub> from the stack and places it in accumulator B. As you can see, the contents of the two accumulators are reversed. The following routine accomplishes the same thing with one less instruction:

```

PSHA
TBA
PULB
    
```

Figure 6-7. Stack and stack pointer instructions.

Now look at a more complex example. Assume that you wish to transfer  $16_{10}$  bytes of data from one place in memory to another. As you saw in the previous unit, this type of problem is a good candidate for indexing. However, indexing alone becomes cumbersome if the two lists are over  $FF_{16}$  memory locations apart. The reason for this is that the offset address can only extend  $FF_{16}$  locations above the address in the index register.

In this example, assume you wish to move the data in memory locations  $0010_{16}$  through  $001F_{16}$  to locations  $01F0_{16}$  to  $01FF_{16}$ . While this could be accomplished using indexing alone, the program becomes unnecessarily complicated. Two separate indexes must be maintained; one for loading data from  $0010_{16}$  through  $001F_{16}$ , the other for storing data in  $01F0_{16}$  through  $01FF_{16}$ . A simpler approach is to use indexing for one operation and the stack capability for the other operation. That is, we could load data from the lower list using indexing and store it in the upper list using the stack capability.

A program that does this is shown in Figure 6-8. The first instruction loads the stack pointer with address  $01FF_{16}$ . This is the address of the last entry in the new list that will be formed. Recall that the new list is to be written in locations  $01F0_{16}$  through  $01FF_{16}$ . Once location  $01FF_{16}$  is established as the top of the stack, we can enter data into the new list simply by pushing data onto the stack. Because the stack pointer is decremented with each push operation, we must push the last entry in the list onto the stack first.

Figure 6-8.  
Moving a list of data using both  
indexing and stack operations.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0020	8E	LDS#	Load the stack pointer immediately with the address of the last entry in the new list.
0021	01	01	
0022	FF	FF	Load the index register immediately with the address of the last entry in the original list.
0023	CE	LDX#	
0024	00	00	Load accumulator A indexed from the original list.
0025	1F	1F	
0026	A6	LDAA, X	Push the contents of accumulator A into the new list.
0027	00	00	
0028	36	PSHA	Decrement the index register.
0029	09	DEX	
002A	8C	CPX#	Compare the contents of the index register with one less than the address of the first entry in the original list.
002B	00	00	
002C	0F	0F	If no match occurs, branch back this far.
002D	26	BNE	
002E	F7	F7	Otherwise, wait.
002F	3E	WAI	

The second instruction loads the index register with the address of the last entry in the original list. This is necessary for the reason pointed out above.

Next, the A accumulator is loaded using indexed addressing. Since the offset address is  $00_{16}$ , the accumulator is loaded with the contents of  $001F_{16}$ . That is, the last entry in the original list is loaded into accumulator A.

The PSHA instruction then pushes the contents of accumulator A onto the stack. Thus, the last entry in the original list is transferred to location  $01FF_{16}$ . In the process, the stack pointer is automatically decremented to  $01FE_{16}$ .

The index register is decremented to  $001E_{16}$  by the next instruction. Then, the CPX instruction compares the index register with  $000F_{16}$  to see if all entries in the list have been moved. If no match occurs, the MPU branches back and picks up the next entry in the list. The loop is repeated over and over again until the entire list has been moved to its new location.

Other uses of the stack will be revealed later. However, even if the stack did nothing more than has already been explained, it would be a very useful capability to have. But as you will see, the MPU uses the stack in several other ways that makes this capability even more important.

## Self-Test Review

1. What is a stack?
2. What is a cascade stack?
3. What is a memory stack?
4. Which type of stack does the 6800 MPU use?
5. What is the name of the instruction that stores data in the stack?
6. What type of instruction is used to retrieve data from the stack?
7. What is the purpose of the stack pointer?
8. The PUSH instruction transfers data from one of the accumulators to \_\_\_\_\_.
9. The PULB instruction transfers data from the top of the stack to \_\_\_\_\_.
10. Refer to Figure 6-8. How can we change this program so that the new list is placed in addresses  $0220_{16}$  through  $022F$ ?

## Answers

1. A stack is a group of registers or a section of memory that is used as a last-in, first-out memory.
2. A cascade stack is a group of hardware registers (usually 16 or less) that is used as a last-in, first-out memory.
3. A memory stack uses a section of RAM as a last-in, first-out memory.
4. A memory stack
5. PUSH
6. PULL
7. The stack pointer indicates the address of the top of the stack.
8. The top of the stack.
9. Accumulator B.
10. By changing the first instruction to: `LDS# 022F16`.



## SUBROUTINES

A subroutine is a group of instructions that performs some limited but frequently required task. A given subroutine may be used many times during the execution of the main program. In many cases, the easiest way to write a program is to break the overall job down into many simple operations, each of which can be performed by a subroutine.

Because subroutines are used so frequently, most microprocessors have special capabilities that allow them to handle subroutines efficiently. In this section, these capabilities will be examined. The discussion will start with the instructions associated with subroutines.

The 6800 MPU has three instructions that are used to handle subroutines. They are:

Jump to Subroutine (JSR)  
Branch to Subroutine (BSR)  
Return from Subroutine (RTS)

Each of these will be discussed in this section. One other instruction that has not yet been mentioned will also be discussed. It is the Jump (JMP) instruction. While not used exclusively with subroutines, the JMP instruction makes an excellent introduction to the Jump to Subroutine (JSR) instruction. Therefore, the Jump (JMP) instruction will be discussed first.

### Jump (JMP) Instruction

This instruction allows the MPU to jump from one point in a program to another. In this respect, it is somewhat like the Branch Always (BRA) instruction that was discussed earlier. The difference is the method of addressing used. Recall that the BRA instruction used relative addressing. This has the advantage that only a 2-byte instruction is required. Its disadvantage is that the branch must be within the range of -128 bytes to +127 bytes of the program count.

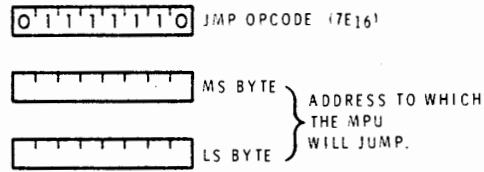


Figure 6-9.  
Format of the JMP instruction  
using extended addressing.

The JMP instruction can use either the indexed or the extended addressing mode. It does not use relative addressing. When using extended addressing, the format of the JMP instruction is as shown in Figure 6-9. Three bytes are required; the opcode followed by the 2-byte address to which the MPU is to jump. Since a 16-bit address is given, the jump may be to any point in the  $65,536_{10}$  byte memory range. This address is loaded into the program counter so that the next opcode is fetched from that address. The previous contents of the program counter are lost. Thus, the MPU starts executing instructions from a new point in memory.

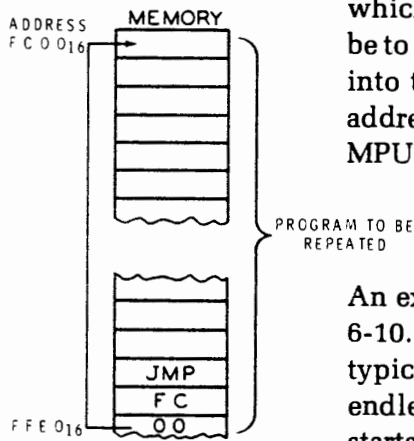


Figure 6-10.  
Using the JMP instruction to  
repeat a program.

An example of how the JMP instruction can be used is shown in Figure 6-10. Here, a long program is to be repeated over and over again. This is typical of applications such as controllers that repeat the same operations endlessly. The program is contained in the upper 1k bytes of memory. It starts at location  $FC00_{16}$  and ends at  $FFE0_{16}$ . Notice that the last instruction is  $JMP FC00_{16}$ . This sends the program back to its beginning so that the loop is repeated endlessly.

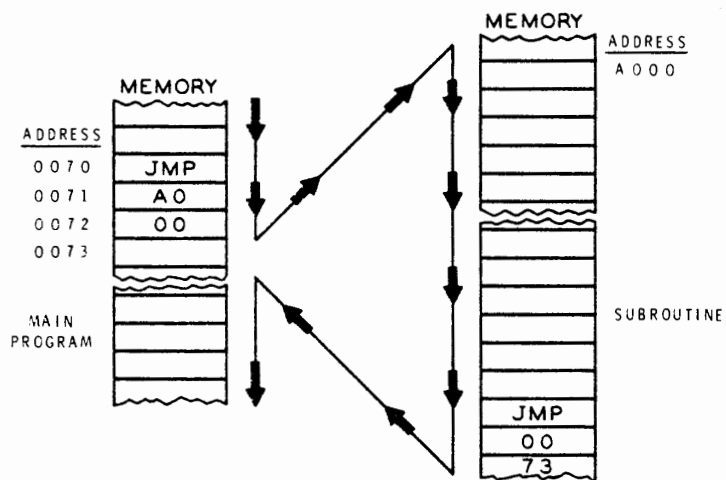


Figure 6-11.  
Using the JMP instruction to  
call a subroutine.

Another possible use of the JMP instruction is shown in Figure 6-11. Here, the main program is in the lower memory locations shown on the left. The main program requires a subroutine that is up at address A000 (shown on the right). The JMP instruction at address 0070 sends the MPU off to the subroutine as shown. The last instruction in the subroutine is another JMP instruction that sends the MPU back to the main program.

Jumping to a subroutine is often referred to as *calling* a subroutine. While we can call a subroutine using the JMP instruction, this approach has a distinct problem. What happens if the main program wants to call the same subroutine more than once? That is, suppose a situation like that shown in Figure 6-12 is required. Here, the main program (on the left) wishes to call the subroutine (on the right) at two separate points. Jumping to the subroutine is no problem. We can do that as many times as we like, using the instruction JMP A000. The problem is: how do we get back from the subroutine to the main program? The first time through the subroutine, the MPU should return to address 0073. The second time through, the MPU should return to address 0093.

A programmer could get around this problem by changing the last instruction in the subroutine before each call or by constructing a table of return addresses, etc. However, most microprocessors have some instructions that solve this problem for us. The following section will discuss the 6800 MPU's solution to this problem.

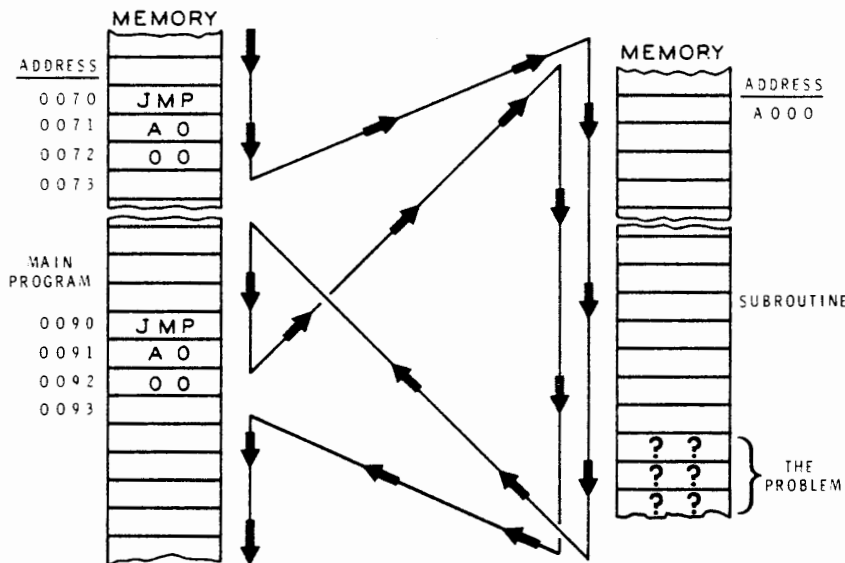


Figure 6-12.

The JMP instruction cannot handle situations like this one.

## JSR and RTS Instructions

If you refer to Figure 6-12 again, you will see that this problem arises because the old program count is not saved when the MPU jumps from one location to the next. However, the 6800 MPU has an instruction that will not only jump to a subroutine, it will also cause the old program count to be stored away. This instruction is called the Jump to Subroutine (JSR) instruction. Its format is exactly the same as the JMP instruction but its execution is different.

Figure 6-13 shows how the earlier problem can be solved using the JSR instruction. Notice that the two JMP instructions in the main program have been replaced by JSR instructions. Notice also that the last instruction in the subroutine is a Return from Subroutine (RTS) instruction. These new instructions ease the problem of calling the subroutine.

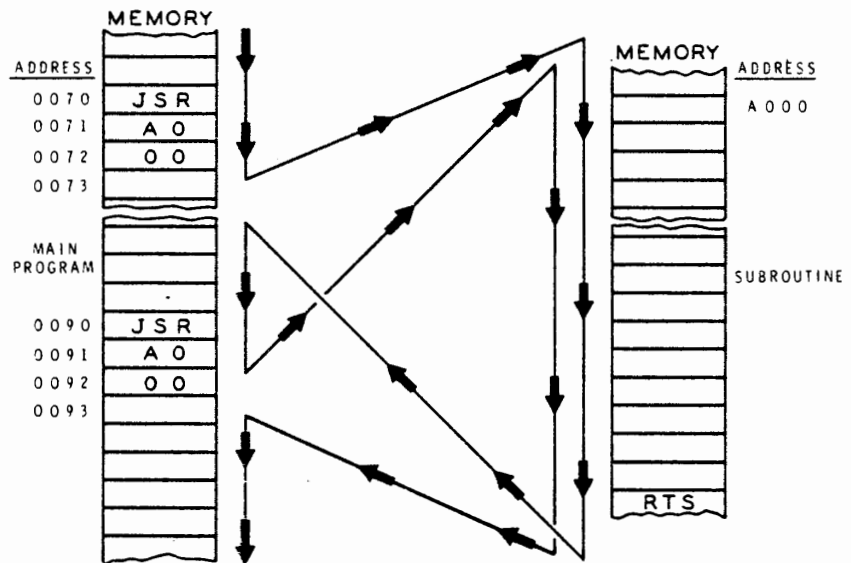


Figure 6-13.  
The JSR and RTS instructions  
can be used to handle this situ-  
ation.

When the first JSR instruction is executed, the subroutine address  $A000_{16}$  is placed in the program counter. However, just prior to this, the program counter was incremented to the address of the next instruction in sequence. That is, the program counter was advanced to  $0073_{16}$  while the contents of address  $0072_{16}$  were being retrieved. This count ( $0073_{16}$ ) is automatically pushed onto the stack. By saving the old program count, the MPU can tell where to return after the subroutine is finished. As soon as the old program count is tucked away safely in the stack, the subroutine address  $A000_{16}$  is placed in the program counter. Thus, the MPU fetches the next instruction from address  $A000_{16}$ .

Notice that the last instruction in the subroutine is an RTS instruction. When the MPU encounters this single-byte instruction, it will jump back to the point where it left off in the main program. It does this by pulling the old program count ( $0073_{16}$ ) from the stack and placing it in the program counter. Consequently, the next instruction will be fetched from address  $0073_{16}$ . As you can see, this returns the MPU to the correct point in the main program.

Notice that the programmer does not specify a return address at the end of the subroutine. The return address is automatically pulled from the stack. This allows us to call the subroutine repeatedly from several different points in the main program.

Figure 6-13 shows that the subroutine is called again by the JSR A000 instruction in location  $0090_{16}$ . As this instruction and address are decoded, the program count is incremented to  $0093_{16}$ . This program count is pushed onto the stack. Then  $A000_{16}$  is placed in the program counter. Thus, the MPU jumps off to the subroutine. When the subroutine is finished, the RTS instruction causes the old program count to be pulled from the stack into the program counter. This causes the MPU to jump back to address  $0093_{16}$  which contains the next instruction in the main program.

## Nested Subroutines

Figure 6-14 shows a situation in which the main program calls subroutine A. In turn, subroutine A calls subroutine B. In this situation, subroutine B is called a *nested* subroutine. That is, a nested subroutine is a program segment that is called by another subroutine. If control is to be eventually returned to the main program, two program counts must be saved. Figure 6-14 shows how the two program counts are saved in the stack.

At the start of the main program, the stack pointer is loaded with the address of the area in memory that has been set aside to act as the stack. If no stack instructions have been executed when the main program arrives at the first JSR instruction, the stack pointer will still be pointing to where it was originally set. The contents of the stack are of no interest until this point.

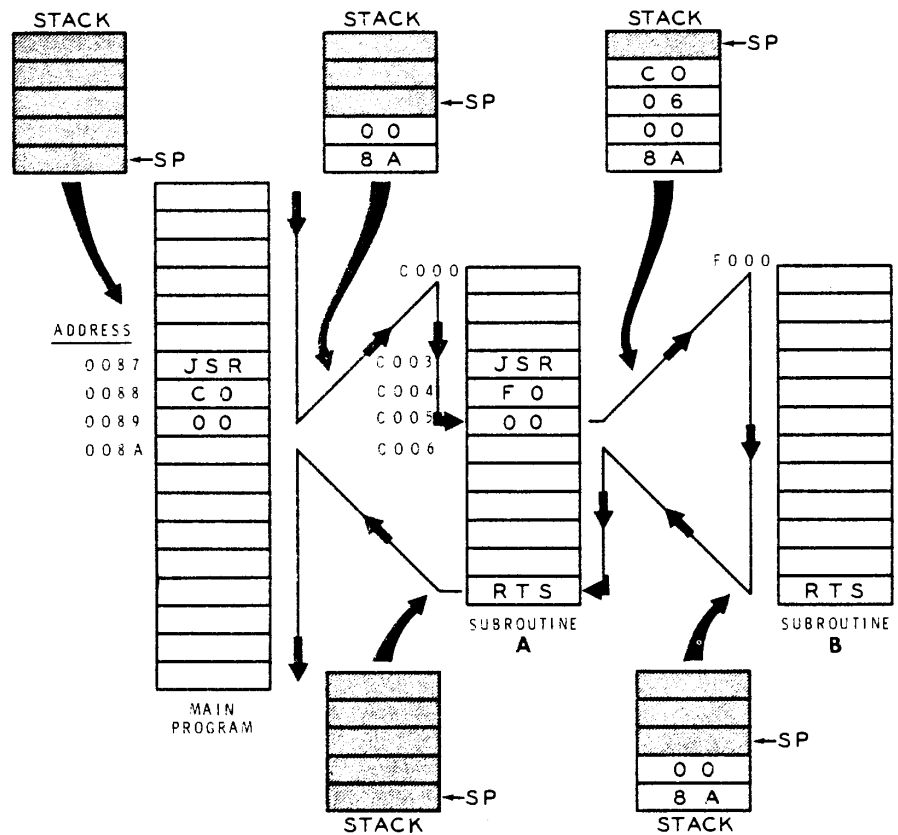


Figure 6-14.  
Handling nested subroutines.

When the main program reaches the JSR instruction, the program count is advanced to the address of the next instruction in sequence ( $008A_{16}$ ). When the JSR instruction is executed, this address ( $008A_{16}$ ) is pushed onto the stack as shown. The low order byte goes in first, followed by the high order byte. In the process, the stack pointer is decremented twice. Finally, the new address ( $C000_{16}$ ) is placed in the program counter. This causes the MPU to jump off to subroutine A which starts at  $C000_{16}$ .

Notice that halfway through subroutine A, subroutine B is called. Consequently, the return address in subroutine A ( $C006_{16}$ ) must be saved. That is, when the program reaches the JSR instruction in subroutine A, the return address ( $C006_{16}$ ) is pushed onto the stack as shown. Notice that there are now two return addresses in the stack. The starting address of subroutine B ( $F000_{16}$ ) is then placed in the program counter and the MPU jumps off to this subroutine.

Subroutine B has no nested subroutines of its own, so the program flow is through the subroutine as shown. The last instruction in subroutine B is the RTS instruction. At this point, the MPU pulls the return address ( $C006_{16}$ ) from the top of the stack and places it in the program counter. This causes the MPU to jump back to the instruction at address  $C006_{16}$  in subroutine A.

The remainder of subroutine A is then executed down to the RTS instruction. This instruction causes the MPU to pull the next address ( $008A_{16}$ ) from the stack and place it in the program counter. Notice that this sends the MPU back to the main program.

For simplicity, a single level of subroutine nesting is shown in this example. However, in practice, many levels of nesting may be used. For example, subroutine B could call subroutine C; etc. Any level of nesting can be used as long as enough memory is set aside for the stack. Remember, each return address requires two bytes in the stack.

## Branch to Subroutine (BSR) Instruction

Quite often, the subroutine we wish to call is within the  $-128_{10}$  to  $+127_{10}$  byte range of the relative address. When it is, we can save one byte by using the Branch to Subroutine (BSR) instruction. The execution of BSR is identical to that of JSR except that relative addressing is used. The old program count is saved in the stack before the branch occurs. Thus, the RTS instruction at the end of the subroutine will cause the old program count to be restored.

## Summary of Subroutine Instructions

Figure 6-15 shows the four instructions discussed in this section. Notice that the BSR instruction uses relative addressing. The JMP and JSR instructions can use either indexed or extended addressing. The RTS instruction uses inherent addressing since its address is pulled from the top of the stack.

Find these instructions on your Instruction Set Summary card. The operations performed by these instructions are illustrated under "Special Operations" on the back of the card. Also, Appendix A of this course gives a concise description of the operations performed by each of these instructions.

JUMP AND BRANCH OPERATIONS		RELATIVE			INDEX			EXTND			INHER		
		OP	~	#	OP	--	#	OP	~	#	OP	~	#
Branch To Subroutine	BSR	8D	8	2									
Jump	JMP				6E	4	2	7E	3	3			
Jump To Subroutine	JSR				AD	8	2	8D	9	3			
Return From Subroutine	RTS										39	5	1

Figure 6-15.  
Subroutine and jump instructions.



## Self-Test Review

11. What is a subroutine?
12. What addressing modes can the JMP instruction use?
13. How does the JMP instruction differ from the BRA instruction?
14. How does the execution of the JSR instruction differ from that of the JMP instruction?
15. Why is the program count saved when the JSR or BSR instructions are executed?
16. Where is the program count saved?
17. How is the stack pointer affected by the JSR instruction?
18. Generally, the last instruction in the subroutine will be a \_\_\_\_\_ instruction.
19. What is a nested subroutine?
20. How is the stack pointer affected by the RTS instruction?

## Answers

11. A subroutine is a group of instructions that performs some specific, limited task that is used more than once by the main program.
12. Indexed and extended.
13. Since the BRA instruction uses relative addressing, it can branch only in a  $-128_{10}$  to  $+127_{10}$  byte range. The JMP instruction uses indexed or extended addressing. Therefore, it can jump to any point in memory.
14. When the JSR instruction is executed, the program count is saved in the stack.
15. The program count is saved so that when the subroutine is finished, the MPU can return to the point it left off.
16. The program count is pushed into the top two locations of the stack.
17. The stack pointer is automatically decremented twice as the program count is pushed onto the stack.
18. Return from Subroutine (RTS).
19. When subroutine A calls subroutine B, subroutine B is said to be nested.
20. The stack pointer is automatically incremented twice as the old program count is pulled from the stack.

## INPUT — OUTPUT (I/O) OPERATIONS

A full explanation of input-output (I/O) operations will be given in the next units, but a brief introduction to I/O is necessary at this point. In this section, you will learn what is involved in sending data to — or taking data from — the MPU.

To be useful, a microprocessor system must accept data from the outside world, process it in some way, and present results to the outside world. The input device may be nothing more than a group of switches while the output device can be as simple as a bank of indicator lamps. On the other hand, a single microprocessor might handle several teletypewriters, printers, papertape machines, etc. The point is that the I/O requirements can vary greatly from one application to the next. This section will be concerned with the simplest form of I/O operations.

In the short history of microprocessors, two distinctly different methods have been developed for handling I/O operations. In some microprocessors, I/O operations are handled by I/O instructions. These microprocessors generally have one *input* instruction and one *output* instruction. When the *input* instruction is executed, a byte is transferred from the selected I/O device to a register (usually one of the accumulators) in the MPU. The I/O device is selected by sending out a device selection byte on the address bus. By using an 8-bit byte for device selection, the MPU can specify up to  $256_{10}$  different I/O devices. Of course, no microprocessor system uses that many devices, but the capability is there. The *output* instruction causes a data transfer from the accumulator to the selected I/O device. While this method of handling I/O operations is used in many microprocessors, the 6800 MPU uses a different technique.

The other method for handling I/O operations is to treat all I/O transfers as memory transfers. This is the method used by the 6800 MPU and many other microprocessors. In fact, even those microprocessors that have I/O instructions can ignore those instructions and handle I/O operations as memory transfers.

The 6800 MPU has no I/O instructions. An I/O device is assigned an address and is treated as a memory location. For example, assume that an input keyboard has been assigned an address of  $8000_{16}$ . We can input data into accumulator A by using the instruction:

LDAA  $8000_{16}$

By the same token, an output display may have been assigned the address  $9000_{16}$ . In this case, we can output from accumulator B by using the instruction:

STAB  $9000_{16}$ .

As you can see, the I/O device is treated as a memory location. The system block diagram shown in Figure 6-16 shows how an I/O device is connected to the microcomputer. Notice that both the data bus and the address bus connect to the I/O interface. As you will see in the next unit, the interface can consist of an address decoder, an output or input latch, and buffers or drivers.

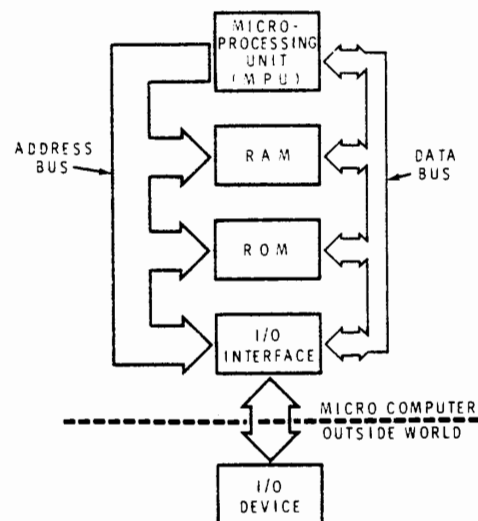


Figure 6-16.  
Adding I/O to the mi-  
crocomputer.

The address decoder monitors the address bus and enables the interface circuitry whenever the proper address is detected. This prevents the I/O interface from interfering when data is being transferred between memory and the MPU.

The I/O interface will generally have an output latch if it is to be used for an output operation. The reason for this is that the data from the MPU will appear on the data lines for only an instant (usually less than one microsecond). By storing the output data in a latch, the I/O device is given a much longer period of time to examine and respond to the data.

Buffers or drivers are also included in the I/O interface. As you will see later, these are frequently necessary when several different circuits are sharing the same bus.

## Output Operations

Figure 6-17 shows a simplified output circuit. Here, the output device is a bank of eight light emitting diodes (LEDs). Enough detail is shown to illustrate how an output operation can be performed. The address decoder monitors the address bus, looking for the address  $9000_{16}$ . It also monitors some of the control lines that connect to the MPU. One of those lines is called a read-write line. It goes to its low state when a write (output) operation is initiated by the MPU. The other control lines will be discussed in the next unit.

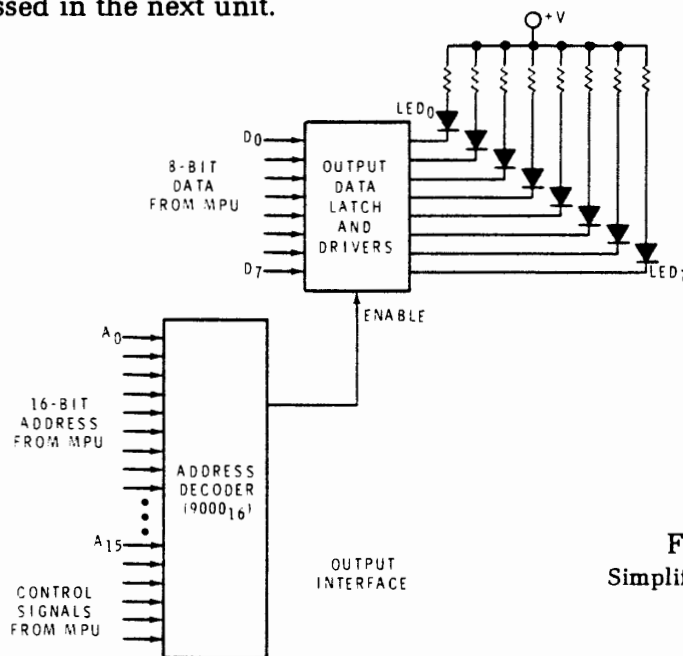


Figure 6-17.  
Simplified output circuit.

Notice that the output of the address decoder is used to enable the output data latch and drivers. When these are enabled, the byte on the data lines is stored in the latch. The data bits stored in the latch cause the appropriate LEDs to light up. By outputting appropriate bit patterns, the MPU can cause different binary numbers to be displayed.

Notice that the address decoder (and therefore the display) is given the address  $9000_{16}$ . We can output data to the display in several different ways. For example, we can load the appropriate pattern to be displayed into accumulator A. Then by executing a "store accumulator A" extended instruction, we can transfer the contents of the accumulator to the display. The instruction would be: STAA  $9000_{16}$ . Or, we could output data from accumulator B by using the instruction: STAB  $9000_{16}$ .

In either case, the address  $9000_{16}$  goes out on the address bus for a brief interval of time. The address decoder recognizes this address. At the same time, the control lines indicate that an output operation is called for. In particular, the read-write line goes low. This causes the address decoder to enable the output data latch for an instant. Simultaneously, the 8-bit data byte appears on the data bus. The output latch stores the data byte. The data appears at the input of the latch for less than a microsecond (typically). However, once the data is stored, it appears at the output of the latch until new data is written in. Thus, the output data will be displayed until the next byte of data is outputted by the MPU.

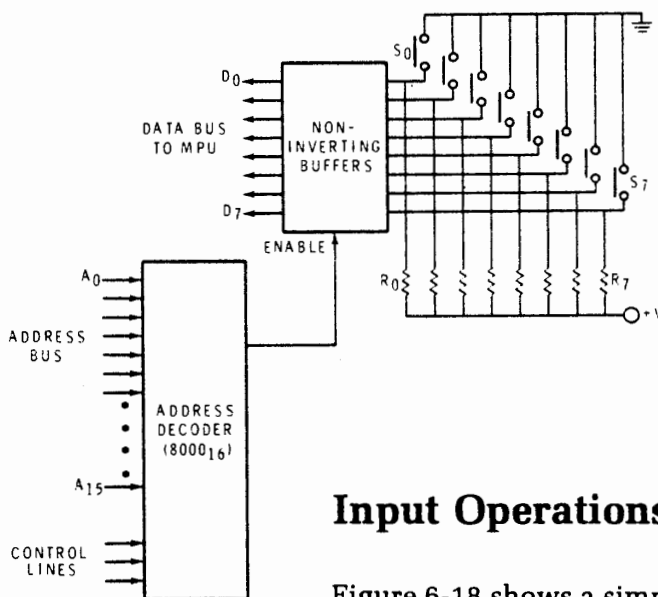


Figure 6-18.  
Simplified input circuit.

## Input Operations

Figure 6-18 shows a simplified input circuit. Here, the input device is a bank of eight switches. When a switch is open, its respective input line to the buffer is held high by the pull-up resistors. However, when a switch is closed, its respective input line is pulled low because the switch connects it to ground.

In this simple circuit, no latch is required between the switches and the data bus. However, a buffer is used so that the switch bank can be effectively disconnected from the data bus when the switches are not being addressed.

As with the output circuit, an address decoder monitors the address and control lines. Notice that the assigned address is  $8000_{16}$ . To input data from the switch bank to accumulator A, we use the instruction: LDAA  $8000_{16}$ . Or, we could input the data to accumulator B by using the instruction: LDAB  $8000_{16}$ .

In either case, the address  $8000_{16}$  is placed on the address line. The address decoder recognizes this address and enables the buffer. For a brief interval (typically less than one microsecond), the lines of the data bus assume the same state as the lines on the right side of the buffer. If no switch is depressed, all data lines will be high and all 1's ( $FF_{16}$ ) will be loaded into the accumulator. However, if one of the switches ( $S_0$ , for example) is depressed, its respective data line ( $D_0$ ) will be low. In this case, the number read into the accumulator will be  $FE_{16}$ . By examining the byte that is read in, the MPU can determine which switch is depressed.

## Input — Output Programming

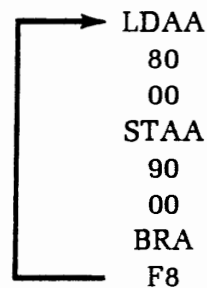
You now know enough about simple input/output circuits to perform some I/O operations. Refer to Figures 6-17 and 6-18. For the first example, assume that you would like one of the LEDs to light when the corresponding switch is pushed. That is,  $LED_0$  should light when  $S_0$  is pushed;  $LED_1$  should light when  $S_1$  is pushed, etc.

If you refer to Figure 6-17, you will see that an LED is caused to light by placing a 0 in the proper bit in the latch. For example, a 0 in bit 0 will cause  $LED_0$  to be forward biased. Thus, the diode will conduct and emit light. Notice that a 1 at bit 0 will not allow the diode to conduct and emit light. Consequently, a 0 turns the LED on and a 1 turns it off.

Refer to Figure 6-18, and you will find that, when one of the switches is closed, its corresponding line goes to 0. If the switch is not closed, its corresponding line is at 1.

If we load data into one of the accumulators from address  $8000_{16}$  and then store the data at address  $9000_{16}$ , the switches will appear to control the LED's. The program could look like this:

```
LDAA  
80  
00  
STAA  
90  
00  
BRA  
F8
```

A diagram showing a vertical line on the left side of the assembly code, with a horizontal arrow pointing to the right at the top, indicating that the code block is enclosed within a specific memory range or context.

If  $S_0$ , and only  $S_0$ , is closed when the LDAA 8000 instruction is executed,  $11111110_2$  will be loaded into accumulator A. The next instruction stores this data byte in the output latch. This causes LED<sub>0</sub>, and only LED<sub>0</sub>, to light. The BRA instruction holds the MPU in a tight loop. Try a few examples and verify that each time a switch is closed, the corresponding LED will light. If the switches are set to some 8-bit binary number, the LED's will display that 8-bit number.

Now, suppose we change our mind and decide that the LEDs should display the one's complement of the binary number set on the switches. We do not have to touch the hardware. Instead, we just change the program. The new program might look like this:

```

      → LDAA
        80
        00
        COMA
        STAA
        90
        00
        BRA
        F7
  
```

Notice that we have simply inserted the one's complement instruction between the input and output operations.

As another example, suppose we wish to display a number that is four times greater than the number set on the switches. Our program could be changed to this:

```

      → LDAA
        80
        00
        ASLA
        ASLA
        STAA
        90
        00
        BRA
        F6
  
```

Once again, no hardware change is needed. We simply insert two ASLA instructions between the input and output operations.



Although these examples are very simple, they illustrate the flexibility of this I/O arrangement. Data is pulled from the input device as if it were being pulled from memory. Once in the MPU, the data byte can be modified in any way we like. The data can then be transferred to the output device as if it were being stored in memory. While the data is in the MPU, it can be modified in any number of ways. The input byte can be shifted left or right. It can be added to — or subtracted from — another number. It can be ANDed or ORed with another byte. The possibilities are endless and yet none of these involve a hardware change. All data manipulations can be accomplished by the program.

## Program Control of I/O Operations

In the preceding examples, all I/O transfers are controlled by the program and the program alone. The program is in a tight loop that inputs data from the switches, modifies the data (if required), and outputs the data to the displays.

When this arrangement is used, the MPU never knows if the data at the input has changed. It simply reads in the data a number of times each second. By the same token, the MPU outputs the data over and over again. This system works well for simple I/O operations. However, as the I/O requirements become more sophisticated, this technique becomes cumbersome.

The program must be in a loop if it is to repeatedly check for inputs and refresh the output. As the number of data manipulations increase, the loop becomes longer and the MPU must check the inputs less frequently. When several I/O devices are used, it must check each input and refresh each output repeatedly. If the loop becomes too long, the MPU may miss a momentary switch closure. This may be acceptable in some applications but in many others it may be intolerable. Obviously then, a more sophisticated method of handling I/O operations must be available to the microcomputer.

## Interrupt Control of I/O Operations

A more effective way of handling I/O operations involves a concept called *interrupts*. Interrupts are a means by which an I/O device can notify the MPU that it is ready to send input data or to accept output data. Generally, when an interrupt occurs, the MPU suspends its current operation and takes care of the interrupt. That is, it might read in or write out a byte of data. After it has taken care of the interrupt, the MPU returns to its original task and takes up where it left off.

An analogy may help you to visualize an interrupt operation. Compare the MPU to the president of a corporation who is writing a report. The interrupt can be compared to a telephone call. The president's main task is the report. However, if the telephone rings (an interrupt), she finishes writing the present word or sentence then answers the phone call. After she has attended to the phone call, she returns to the report and takes up where she left off. In this analogy, the ringing of the telephone notifies the president of the interrupt request.

This analogy shows the difficulty of the program controlled I/O technique discussed earlier. If we remove the interrupt request (the ringing of the phone), we are left with an almost comical situation. The president writes a few words of the report. She then picks up the phone to see if anyone is on the other end. If not, she hangs up the phone, writes a few more words, and checks the phone again. Clearly, this technique wastes an important resource — the president's time.

This simple analogy shows the importance of an interrupt capability. Without it, a great deal of the MPU's time can be wasted doing routine operations. The next section will examine the interrupt capabilities of the 6800 MPU.

## Self-Test Review

21. What are the two methods by which microprocessors handle I/O operations?
22. Which method does the 6800 MPU use?
23. Which instruction can be used for transferring data from an I/O device to accumulator A?
24. Which instruction can be used for transferring data from accumulator B to an I/O device.
25. Write a program segment that will: read in data from the switch bank shown in Figure 6-18; double the number; and display the result on the LED bank shown in Figure 6-17.
26. What is meant by program control of an I/O operation?
27. What is meant by interrupt control of an I/O operation?

## Answers

21. Some microprocessors have input-output instructions; others treat I/O as memory.
22. The 6800 MPU treats I/O as memory.
23. LDAA
24. STAB
25. One solution is:  
  

LDAA  
80  
00  
ASLA  
STAA  
90  
00
26. Using this method, the program regularly reads in or writes out data. All I/O operations are controlled by the program.
27. Using this method, the I/O device itself signals the MPU that it is ready to transmit or receive data. The I/O operations are controlled largely by the I/O device itself.

## INTERRUPTS

Interrupts were introduced in the previous section in connection with I/O operations. While I/O operations use part of the interrupt capability of the MPU, interrupts are also used in other ways. The 6800 MPU has four different types of interrupts:

Reset  
Non-Maskable Interrupt (NMI)  
Interrupt Request (IRQ)  
Software Interrupt (SWI)

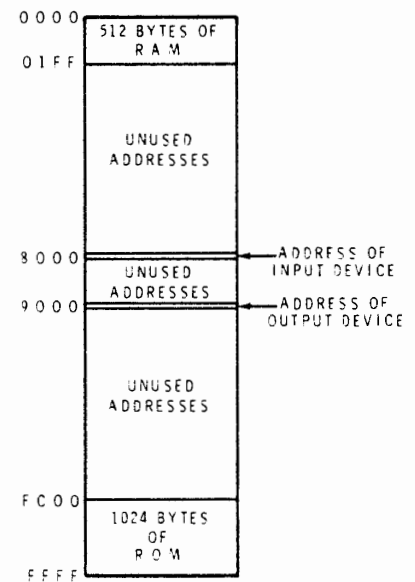
This section will examine each of these interrupts in detail.

### Reset

In a typical application, the microcomputer has a control or monitor program in a read-only-memory (ROM). Also, a random access read-write memory (RAM) is used for holding input data, intermediate answers, output data, etc. As we have seen, the 6800 MPU has the capability of addressing up to  $65,536_{10}$  memory locations. Most microprocessor applications do not require this much memory. In many applications, the control program requires less than ten percent of the possible locations. The RAM probably uses less than two percent. Generally, the monitor program is placed at the high memory addresses. The RAM is usually given the low memory addresses so that the direct addressing mode can be used. The I/O devices are given intermediate addresses. Thus, the memory addresses may be allocated as shown in Figure 6-19.

Notice that the control or monitor program is placed in a ROM at the very top of memory. In this example, a  $1024_{10}$  byte ROM is used. The addresses of the ROM are  $FC00_{16}$  through  $FFFF_{16}$ . A small RAM is placed at the low end of memory. Addresses  $0000_{16}$  through  $01FF_{16}$  are used. Notice that all other addresses are unused except for two. The input device is assigned address  $8000_{16}$ , while the output device is assigned address  $9000_{16}$ .

The monitor program stored in the ROM, controls all the activities of the MPU. At all times, the entire system is being run by this program. In this example, when the microprocessor is initially turned on, it should start executing instructions at address  $FC00_{16}$ . Also, we should be able to restart the program at this address at any time. In order to accomplish this, the 6800 MPU has a built-in reset capability.



**Figure 6-19.**  
Memory allocations in a typical microcomputer system.

The 6800 MPU has a signal line or control pin that is called  $\overline{\text{Reset}}$ . This pin or line is connected to a reset switch of some kind. If this line goes low for a prescribed period of time (to be explained later) and then swings high, the MPU will initiate a reset interrupt sequence. The main purpose of the reset interrupt sequence is to load the address of the first instruction to be executed into the program counter. This would be easy to accomplish if, in every application, the starting address were the same. However, the starting address differs from one application to the next. Therefore, a convenient means is provided to allow the designer to specify any starting address that he likes.

In any 6800 based microprocessor system, the upper eight bytes of ROM are reserved for *interrupt vectors*. An interrupt vector is simply an address that is loaded into the program counter when an interrupt occurs. Figure 6-20 shows how these eight reserved memory bytes are allocated. Notice that addresses  $\text{FFFE}_{16}$  and  $\text{FFFF}_{16}$  contain the reset vector. That is, these two memory locations contain the address of the first instruction that is to be executed when the microcomputer is initially started. In our example, the first instruction in the monitor program is at address  $\text{FC00}_{16}$ . Consequently, this is our reset vector. Location  $\text{FFFE}_{16}$  must contain the high byte of the address ( $\text{FC}_{16}$ ) and  $\text{FFFF}_{16}$  must contain the low byte of the address ( $\text{00}_{16}$ ).

Remember locations  $\text{FFFE}_{16}$  and  $\text{FFFF}_{16}$  are in the read-only-memory. Therefore, the designer must provide the proper reset vector at the time he is writing the monitor program.

Figure 6-20.  
Interrupt vector assignments.

Address	
F F F 8	Interrupt Request Vector (high order address)
F F F 9	Interrupt Request Vector (low order address)
F F F A	Software Interrupt Vector (high order address)
F F F B	Software Interrupt Vector (low order address)
F F F C	Non-Maskable-Interrupt Vector (high order address)
F F F D	Non-Maskable-Interrupt Vector (low order address)
F F F E	Reset Vector (high order address)
F F F F	Reset Vector (low order address)

Figure 6-21 shows the sequence of events that occurs when the MPU is reset. First, the interrupt (I) mask bit is set. You will recall that the I flag is one of the condition code registers. As you will see later, if this flag is set, it prevents one of the other interrupts from occurring. Thus, the MPU sets the interrupt mask bit so that the reset sequence will not be interrupted by a request for interrupt by one of the I/O devices.

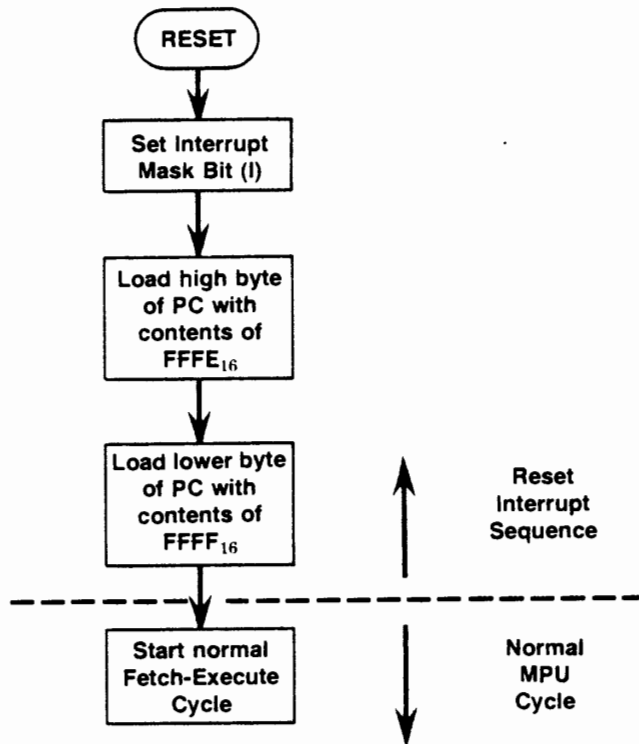


Figure 6-21.  
Reset interrupt sequence.

Second, the contents of location FFFE<sub>16</sub> are loaded into the high byte of the program counter. This is done by sending the address FFFE<sub>16</sub> out on the address bus. The memory location is read out and its contents are placed on the data bus. The MPU picks up this byte and places it in the upper eight bits of the program counter. In our example, the byte in location FFFE<sub>16</sub> is FC<sub>16</sub>.

Next, the contents of location FFFF<sub>16</sub> are loaded into the lower eight bits of the program counter. This is done by setting the address bus to FFFF<sub>16</sub>. Thus, the contents of the highest memory location are placed on the data bus. In our example, this byte is 00<sub>16</sub>. At this point, the program counter contains the address of the first instruction which is FC00<sub>16</sub>.

The reset sequence is then terminated by switching the MPU to its normal fetch-execute machine cycle. Thus, the instruction at address FC00<sub>16</sub> is fetched and executed. From this point on, all MPU activities are controlled by the program.

The microprocessor system will have a reset switch somewhere in the system. This will allow the operator to restart the system if the system locks up or runs away for some reason. In addition, some systems will have an automatic reset feature that will allow the system to reset itself after a power failure. In both cases, the reset capability of the MPU is used.

This reset capability can be considered an interrupt, since the MPU leaves whatever it is doing and jumps off to the start of the monitor program. In most cases, the monitor program would start with a short subroutine that initializes the system. It would do things like set up the stack pointer, initialize displays, etc.

## Non-Maskable Interrupts

The 6800 has two other types of hardware interrupts. One of these interrupts is maskable; the other is not. A maskable interrupt is one that the MPU can ignore under certain conditions. Whereas, a non-maskable interrupt cannot be ignored. To illustrate the difference, recall the corporation president analogy.

The president's report writing can be interrupted by the telephone. However, by telling her secretary to hold all calls, she has effectively masked one source of interruptions. In this analogy it is impractical to mask all interrupts. For example, it could be counterproductive to mask the fire alarm.

Somewhat the same situation can exist in a microprocessor controlled system. Some interrupts can be ignored for a few seconds while the MPU is performing a more important task. This type of interrupt can be masked. Others must not be ignored at all. These cannot be masked. Of course, it is up to the designer to decide which interrupts can be masked and which cannot. The 6800 MPU has provisions for handling both types. How the MPU handles the non-maskable type will be discussed first.

The 6800 MPU has a control line called the non-maskable interrupt ( $\overline{\text{NMI}}$ ) line. A high-to-low transition on this line forces the MPU to initiate a *non-maskable interrupt sequence*. The purpose of this sequence is to provide an orderly means by which the MPU can jump off to a service routine that will take care of the interrupt.

This becomes somewhat involved because the MPU must be able to go back to its main program after the interrupt service routine is finished. It must be able to pick up exactly where it left off. Furthermore, all registers must hold exactly the same data and addresses that they held when the



interrupt occurred. In other words, when an interrupt occurs, the program count must be saved so that the MPU can later return to this point in the program. Also, the contents of the accumulators, index register, and even the condition code registers must be saved so that the MPU can be restored to the exact condition that existed at the instant the interrupt occurred.

The 6800 MPU accomplishes this by pushing all the pertinent data onto the stack. Then, after the interrupt has been serviced, the MPU returns to its previous status by pulling the data from the stack.

The non-maskable interrupt sequence is shown in Figure 6-22. A non-maskable interrupt is initiated when the  $\overline{\text{NMI}}$  line goes from its high state to its low state. The MPU finishes the execution of the current instruction. However, before another instruction is fetched, the MPU pushes the contents of its registers onto the stack. Recall that the stack pointer always points to the top of the stack. For this example, assume that the stack pointer was set by an earlier instruction to address  $0068_{16}$ .

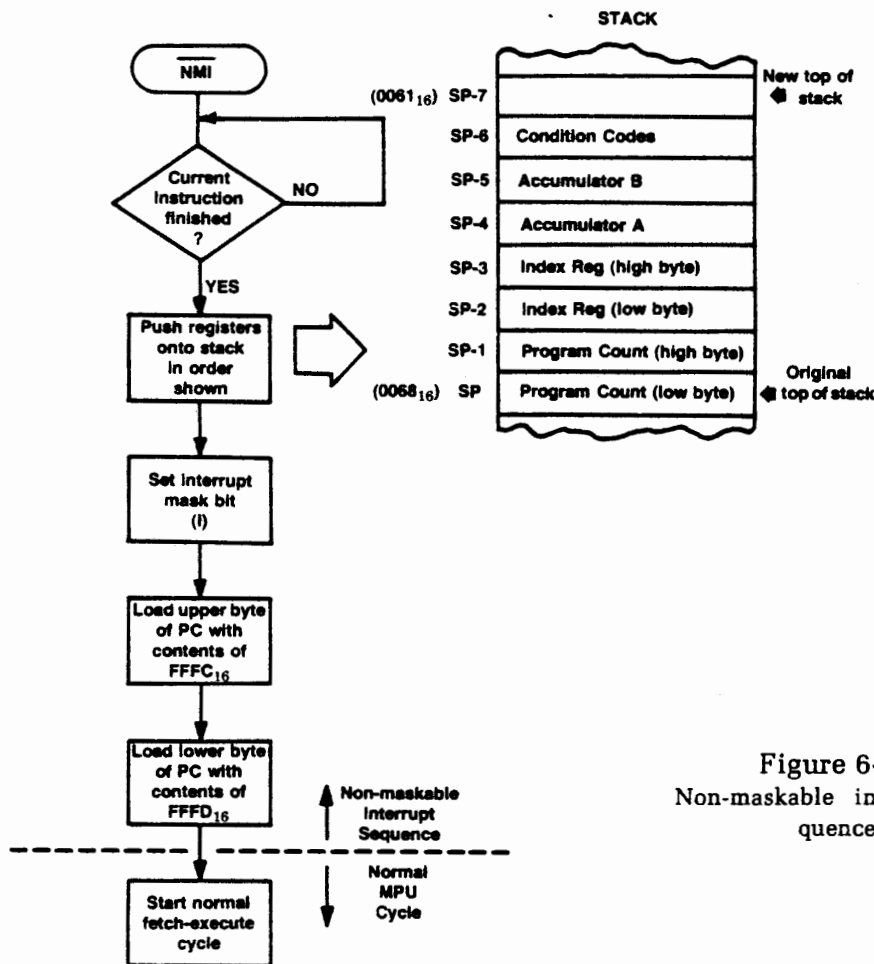


Figure 6-22.  
Non-maskable interrupt sequence.

The MPU pushes the lower eight bits of the program counter into memory location  $0068_{16}$ . Then it decrements the stack pointer so that the upper eight bits of the program counter are pushed into address  $0067_{16}$ . Next, the contents of the index register are pushed into addresses  $0066_{16}$  and  $0065_{16}$ . The contents of accumulators A and B and the condition codes are also pushed in as shown. When all this has been done, the stack pointer will have been decremented seven times to  $0061_{16}$ .

Return to the flow chart and notice that the next step is to set the interrupt mask bit. This allows the MPU to ignore any interrupt requests that occur while the non-maskable interrupt is being serviced.

At this point, the MPU is ready to jump to the interrupt service routine. But, what is the address of this routine? Recall the interrupt vector chart that was shown earlier in Figure 6-20. The non-maskable interrupt vector is at addresses  $FFFC_{16}$  and  $FFFD_{16}$ . Thus, the upper byte of the program counter is loaded from  $FFFC_{16}$  while the lower byte is loaded from  $FFFD_{16}$ . This directs the MPU to the first instruction in the non-maskable interrupt service routine. From this point on, the MPU returns to its normal fetch-execute cycle until the service routine is finished.

The sequence of events shown in Figure 6-22 happen automatically when a non-maskable interrupt sequence is initiated. The  $\overline{NMI}$  line gives external hardware a method of forcing a jump-to-subroutine to occur. In this case, the subroutine is a short program that performs some action to take care of the interrupt.

## Return From Interrupt (RTI) Instruction

The non-maskable interrupt is used when some situation exists that cannot be ignored. You can probably visualize applications that would require such a capability. For example, assume that a microprocessor is being used in a numerically controlled drill press. The non-maskable interrupt could be used in conjunction with limit switches to prevent drilling holes in the work surface. Or, it could be used to shut down the machine if someone's hand got too close.

The purpose of the service routine is to direct the operation of the computer to take care of the interrupt. Typically, it would first determine which external device initiated the interrupt. Then it would determine the nature of the interrupt. Finally, it would take whatever action was necessary to take care of the interrupt. In many cases, the interrupt is of a routine nature and can be easily serviced. In these situations, the MPU

should return to the main program and take up where it left off. There is an instruction that allows the MPU to do this. It is called the “Return-From-Interrupt” (RTI) instruction. Look on your Instruction Set Summary card, and you will see that this is a one-byte instruction whose opcode is  $3B_{16}$ .

Figure 6-23 shows how the RTI instruction is used. The main program is shown on the left, while the interrupt service routine is shown on the right. Assume that the interrupt signal occurs while the LDAB# instruction is being executed. The MPU finishes that one instruction and pushes all pertinent data onto the stack. It then jumps to an address determined by the  $\overline{NMI}$  vector in address FFFC and FFFD. The contents of these two locations determine the starting address of the  $\overline{NMI}$  service routine. Notice that the last instruction in the service routine is the return-from-interrupt instruction. This instruction returns program control to the point in the main program that the MPU left when the interrupt occurred.

This can be done because the previous status of the MPU was preserved in the stack. The RTI instruction causes the accumulators, the index registers, the condition code register, and the program counter to be loaded from the stack. Thus, the same information that went into the stack when the interrupt occurred comes out of the stack when the RTI instruction is executed. This allows the MPU to return to the main program and take up where it left off.

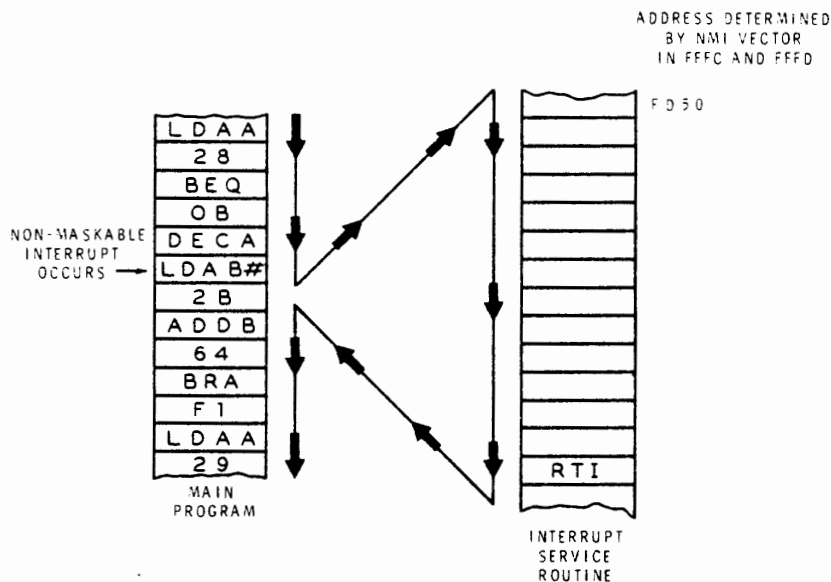


Figure 6-23.

The RTI instruction returns control to the main program after the interrupt has been serviced.

## Interrupt Request (IRQ)

The interrupt request is very similar to the non-maskable interrupt. The main difference between the two is that the interrupt request is maskable.

The 6800 MPU has a control line called the interrupt request ( $\overline{\text{IRQ}}$ ) line. When this line is low, an interrupt sequence is requested. However, the MPU may or may not initiate the interrupt sequence depending on the state of the interrupt mask (I) bit in the condition code register. If the I bit is set, the MPU ignores the interrupt request. If the I bit is not set, the MPU initiates the interrupt sequence. This procedure is very similar to the  $\overline{\text{NMI}}$  procedure discussed earlier. Figure 6-24 shows the interrupt procedure.

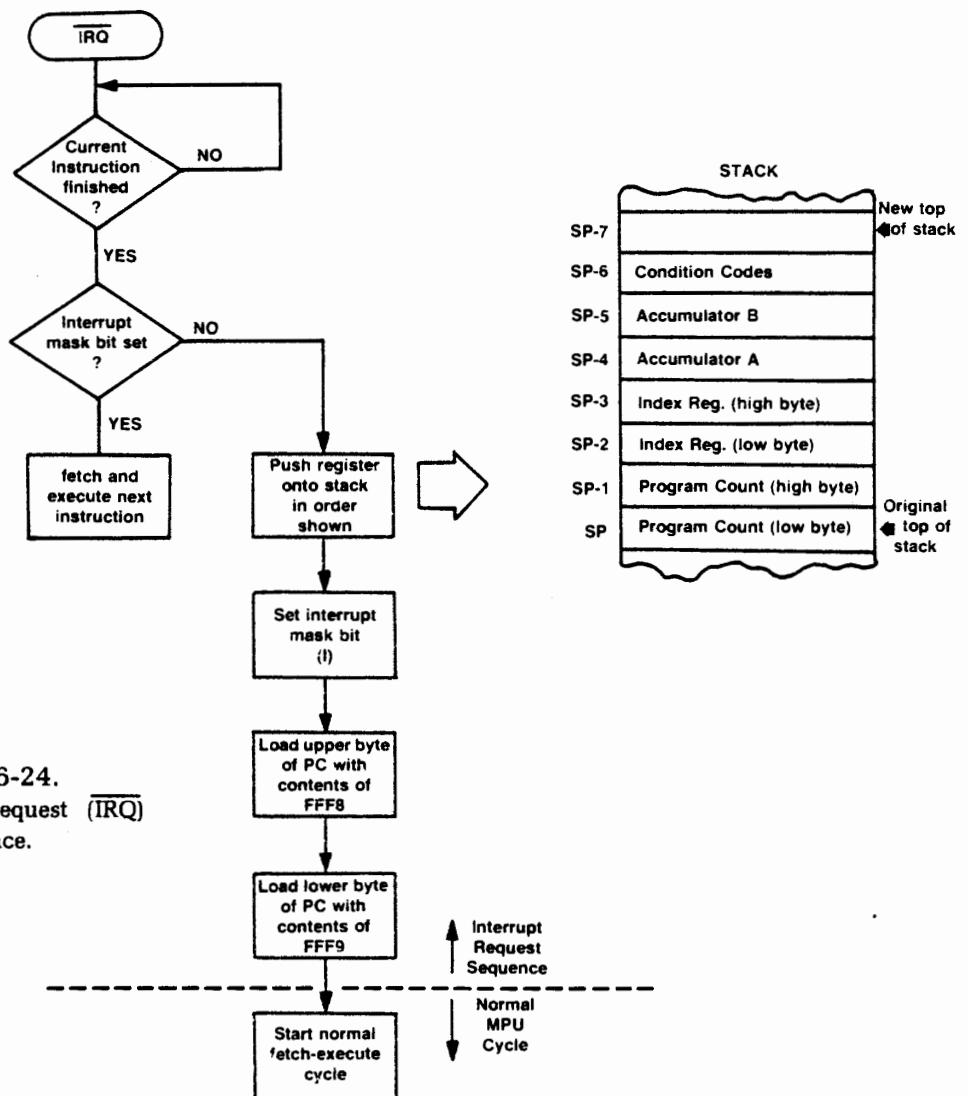


Figure 6-24.  
The interrupt request ( $\overline{\text{IRQ}}$ ) sequence.

When the  $\overline{\text{IRQ}}$  line is low, the MPU finishes the current instruction. It then checks the interrupt mask bit. If I is set to 1, the MPU ignores the interrupt request and executes the next instruction in sequence. However, if I=0, the MPU pushes the contents of the various registers onto the stack in the order shown.

Next, the interrupt mask bit is set to 1. This prevents the MPU from honoring other interrupt requests until the present interrupt has been serviced.

The address of the  $\overline{\text{IRQ}}$  service routine is at addresses  $\text{FFF8}_{16}$  and  $\text{FFF9}_{16}$ . The program counter is loaded from these addresses. Thus, the next instruction to be executed will be the first instruction in the interrupt request service routine.

Once in the service routine, the MPU goes into its normal fetch-execute cycle. When the interrupt has been serviced, control can be returned to the main program by an RTI instruction.

## Interrupt Mask Instructions

The 6800 MPU has two instructions that allow software control of the interrupt mask bit. You have seen that the I bit in the condition code register is set any time an interrupt sequence is initiated. This prevents an  $\overline{\text{IRQ}}$  from being honored while a previous  $\overline{\text{IRQ}}$  or  $\overline{\text{NMI}}$  is being serviced. This is an example of setting the interrupt flag with hardware.

In many cases, it is necessary to set the interrupt flag with software. Therefore, the 6800 MPU has an instruction that can do this. It is called the "Set-Interrupt-Mask" (SEI) instruction. If you refer to your Instruction Set Summary card, you will see that this is a one-byte instruction whose opcode is  $0\text{F}_{16}$ . The flag may be set to prevent an interruption on a part of the program that we do not wish to be interrupted. It has the effect of disabling interrupt requests.

Of course, we do not wish to permanently disable the interrupt capability. Therefore, some means must be provided for enabling the interrupt request capability. An instruction called "Clear-Interrupt-Mask" (CLI) is available for this purpose. This is a one-byte instruction whose opcode is  $0\text{E}_{16}$ .

While we can disable or enable the interrupt request line with these instructions, they do not affect the non-maskable interrupt. As the name implies, the  $\overline{\text{NMI}}$  line cannot be disabled by the I flag.

## Software Interrupt (SWI) Instruction

The 6800 MPU has a software equivalent of an interrupt. It is an instruction called the "Software Interrupt" (SWI). When executed, the instruction causes the MPU to perform an interrupt sequence that is very similar to the hardware interrupt sequences already discussed. As shown on your Instruction Set Summary card, this is a one-byte instruction whose opcode is  $3F_{16}$ .

Figure 6-25 shows the sequence of events that occurs when this instruction is executed. First the contents of all the pertinent registers are pushed onto the stack in the order shown. Next, the interrupt mask is set so that interrupt requests cannot interfere. Finally, the software interrupt vector is obtained from addresses  $FFFA_{16}$  and  $FFFB_{16}$ . This vector is loaded into the program counter so that the next instruction will be fetched from this address. As with the other interrupts, the MPU will return to the original program when a return-from-interrupt instruction is encountered.

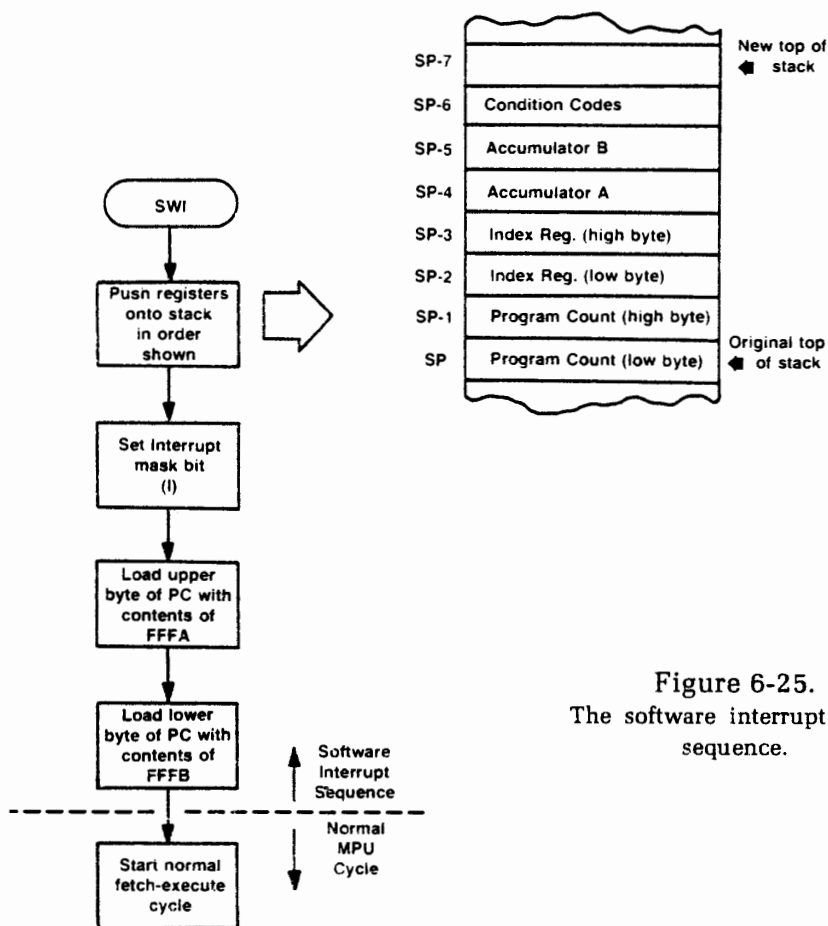


Figure 6-25.  
The software interrupt (SWI)  
sequence.

The software interrupt instruction can be used to simulate hardware interrupts. It is also helpful for inserting pauses in a program. For example, the ET-3400 Microprocessor Trainer uses the software interrupt to perform the single-step function and to implement the breakpoint capability.

## Wait for Interrupt (WAI) Instruction

One of the first instructions introduced in this course was the halt instruction (opcode  $3E_{16}$ ). In the previous unit, you learned that this instruction is actually called a Wait-for-Interrupt (WAI). What exactly does this instruction do? It does cause the MPU to halt, but there is more to it than that.

When the WAI instruction is executed, the program counter is incremented by one. Then the contents of the program counter, index register, accumulators, and condition code register are pushed onto the stack. The order is exactly the same as if an interrupt occurs. The MPU then enters a wait state, doing nothing further until, and unless, an interrupt occurs.

The MPU can be forced back into action either by an interrupt request or by a non-maskable interrupt. The  $\overline{NMI}$  sequence is the same as that described earlier except for one important difference. Remember that the contents of the registers have already been pushed onto the stack. Thus, this part of the  $\overline{NMI}$  sequence is omitted. This allows the MPU to respond faster to the interrupt.

The  $\overline{IRQ}$  sequence is also the same as that described earlier except that the registers are not pushed onto the stack again. As always, the  $\overline{IRQ}$  signal is ignored if the interrupt mask bit is set.

Of course, the reset signal can override the wait state. Thus, there are three ways of escaping the wait state.





## Answers

28. Reset, non-maskable interrupt, interrupt request, and software interrupt.
29. Interrupt request ( $\overline{\text{IRQ}}$ ).
30. To direct the MPU to the first instruction in the monitor or control program.
31.  $\text{FFFE}_{16}$  and  $\text{FFFF}_{16}$ .
32. The address of the interrupt service routine.
33.
  - A. The current instruction is executed.
  - B. The contents of the pertinent registers are pushed onto the stack.
  - C. The interrupt mask bit is set.
  - D. The  $\overline{\text{NMI}}$  vector from addresses  $\text{FFFC}_{16}$  and  $\text{FFFD}_{16}$  is loaded into the program counter.
  - E. The instruction at the address specified by the  $\overline{\text{NMI}}$  vector is fetched and executed.
34. A routine that takes care of the interrupt and then returns control to the main program.
35. The Return-From-Interrupt (RTI) instruction.
36. The stack pointer is incremented seven times as the previous MPU status is pulled from the stack.
37. Reset.
38. Set Interrupt Mask (SEI).
39.
  - A. By a reset signal.
  - B. By a non-maskable interrupt.
  - C. By an interrupt request (if  $I=0$ ).
40. If the MPU is waiting for an interrupt.

## EXPERIMENTS

Perform Experiments 9 and 10 in the Programming Experiment Section (Unit 9) of this course. After you finish these experiments, return to this unit and complete the unit examination.

## UNIT EXAMINATION

1. If the I bit in the condition code register is set, the MPU will ignore:
  - A. The reset signal.
  - B. The non-maskable interrupt signal.
  - C. The interrupt request signal.
  - D. The software interrupt instruction.
  
2. Which of the following lists contains instructions that do **not** change the contents of the stack pointer?
  - A. PULA, DES, RTI, WAI.
  - B. PSHB, INS, RTS, SWI.
  - C. TXS, BSR, PULB, LDS.
  - D. PSHA, JMP, TSX, STS.
  
3. Which of the following program segments will successfully swap the contents of the two accumulators?

A. PSHA	B. PSHB	C. PSHA	D. PSHB
TAB	TAB	TBA	TBA
PULA	PULA	PULA	PULB
  
4. The stack pointer is automatically:
  - A. Decrementd before data is pushed onto the stack.
  - B. Incremented before data is pushed onto the stack.
  - C. Decrementd after data is pushed onto the stack.
  - D. Incremented after data is pushed onto the stack.
  
5. One difference between the JMP and JSR instruction is:
  - A. JMP can use either extended or indexed addressing.
  - B. The program count is saved when JSR is executed.
  - C. The JSR will be executed even if the interrupt mask is set.
  - D. JMP is an unconditional jump.
  
6. The last instruction in a subroutine is generally:
  - A. A JMP instruction.
  - B. An RTS instruction.
  - C. An RTI instruction.
  - D. A JSR instruction.

7. In the 6800 MPU, which of the following instructions could be used to transfer data from an I/O device to accumulator A?
- INPA.
  - LDAA.
  - STAA.
  - OUTA.
8. Refer to Figures 6-17 and 6-18. Which of the following program segments will read in data from the switch bank and, if the number is larger than  $2A_{16}$ , display it on the LED's?
- |         |         |         |         |
|---------|---------|---------|---------|
| A. LDAA | B. LDAA | C. LDAA | D. LDAA |
| 80      | 80      | 80      | 90      |
| 00      | 00      | 00      | 00      |
| CMPA#   | SUBA#   | STAA    | SUBA#   |
| 2A      | 2A      | 90      | 2A      |
| BHI     | BHI     | 00      | BHI     |
| 01      | 01      |         | 01      |
| WAI     | WAI     |         | WAI     |
| STAA    | STAA    |         | STA     |
| 90      | 90      |         | 80      |
| 00      | 00      |         | 00      |
9. Which of the following types of interrupts does **not** cause data to be pushed into the stack?
- Software interrupt.
  - Non-maskable interrupt.
  - Reset interrupt.
  - Interrupt request.
10. Generally, the last instruction in an interrupt service routine will be:
- An RTI instruction.
  - An SWI instruction.
  - An RTS instruction.
  - An NMI instruction.