THE 6800 MICROPROCESSOR

A Self-Study Course with Applications

REVISED EDITION

LANCE A. LEVENTHAL
Engineering Technology Department
Grossmont College

HAYDEN BOOK COMPANY, INC.
Rochelle Park, New Jersey

ISBN 0-8104-5120-4
Library of Congress Catalog Card Number 78-56262

2	3	4	5	6	7	8	9	PRINTING
78	79	80	81	82	83	84	85	86 YEAR

PREFACE

I have planned this laboratory manual as a simple introduction to the use of microcomputers. It assumes almost no hardware beyond that supplied with a basic Motorola 6800 Computer. The last two exercises require an edge connector and a few simple switches and lights – all of which can be purchased from electronic or radio stores at very low cost. Thus the manual should be particularly suitable for use in colleges, trade schools, adult education courses, secondary schools, small companies, homestudy courses, and other situations where extensive laboratory facilities and materials are not available.

The exercises are aimed at electrical engineering and technology students. However, I have assumed no particular knowledge of either programming or digital logic so the manual should be useful for students of data processing, mathematics, computer science, other engineering or technical disciplines, health sciences, and other fields. It should also be suitable for those who are just curious about how computers work. Even if the student plans to use a high-level language in actual applications, a little knowledge about how the computer operates at the machine language level may be useful and interesting; it may also provide insight into more complex problems.

The emphasis here is on the use of the microcomputer as a controller which responds to inputs and prepares suitable outputs. Simple control applications are common in industry, easy to understand, and require little extra hardware. However, I should mention that the microcomputer is equally useful in industrial control, communications, data acquisition, business data processing, simulation, data logging, interfacing, human interaction, signal processing, and other applications. Perhaps subsequent manuals will explore some of these areas. The microcomputer can do anything that a large computer can do. Furthermore, the low cost of the microcomputer makes it an ideal teaching tool since students can have ready access to it and complete control over its facilities.

The laboratory exercises contain many short programs. I have used the notation from the standard Motorola 6800 assembler (# means immediate, $ means hexadecimal, % means binary) so that the programs have the same format as those available in manuals from Motorola and other sources and in textbooks and articles. I have provided hexadecimal listings of many programs to provide students with reasonable starting points for relatively brief laboratory sessions. I have tried to make the programs clear, simple, well-structured, and well-documented rather than efficient. All of the programs have, of course, been fully tested on the 6800 and I take responsibility for any errors. This manual does not describe the Motorola 6800 in great detail; more extensive discussions are available in the standard manuals and in my textbooks.*

I would like to thank Mr. Tim Mathis, Mr. Bill Sheets, and Mr. Joe Stapczynski of Southwestern College, Mr. Colin Walsh of Grossmont College, and Mr. Victor Wintriss and Ms. Patti Neumann of Electronic Product Associates, Inc., for their help in producing this manual. Mr. Karl Amatneek of KVA Associates has provided many useful suggestions and has kept me from straying too far from the main subject of interest.

Lance A. Leventhal

Solana Beach, CA

**Microprocessors*, published by Prentice-Hall, and *6800 Assembly Language Programming*, published by Adam Osborne and Associates.

INTRODUCTION

The 6800 Laboratory Manual is designed as the basis of a self-study course in microprocessor applications for use with a minimal Motorola 6800 computer.

We recommend that the Motorola M6800 Programming Reference Manual and the Motorola M6800 Microcomputer System Design Data Manual be used as reference sources with this Manual.

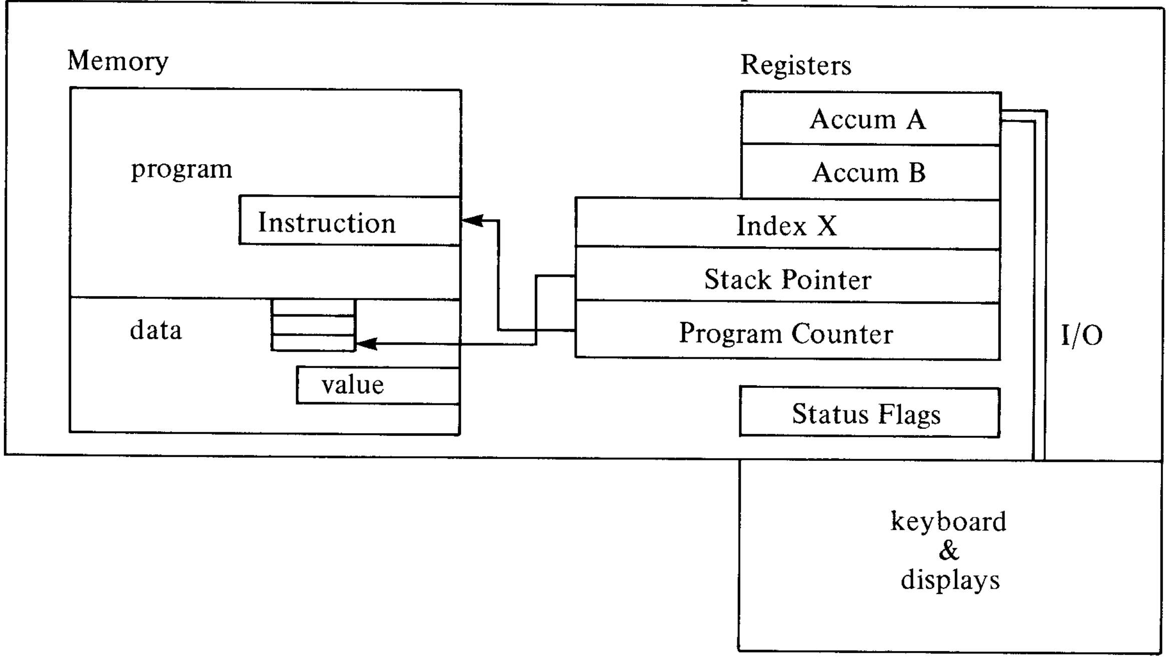
BRIEF INDEX

LABORATORY 0

Introduction to the 6800

The computer used throughout this book is called Micro-68. The Micro-68 is an inexpensive microcomputer based on the Motorola 6800 microprocessor. Besides the microprocessor, the Micro-68 has program and data memory, input/output circuitry, a 16 key keyboard, 6 seven-segment displays, power supply, and case. The instructions in this manual will be applicable to any 6800 processor.

Figure 0-1
Logical View of the 6800 Computer



The microcomputer is under control of a monitor program stored in read-only memory. This program allows you to place programs and data in read-write memory, execute programs, and examine the contents of memory. Each memory location has a 16 bit address (four hexadecimal digits) and contains 8 bits of data (two hexadecimal digits). Table 0-1 is a list of the hexadecimal digits and their binary equivalents.

The Micro-68 has two working registers, A and B. Each register is 8 bits wide and is able to hold two hex digits.

The index register X is a third internal register used to offset an address by an amount stored in X. The contents of the X register are added to the instruction address to determine the absolute address of an operand.

The stack pointer register contains the address of the top-of-stack element in memory.

The program counter register contains the address of the next instruction to be executed.

The status flags are single bits used to indicate conditions and processor status.

These registers and their values will be fully discussed in the following laboratory exercises.

Table 0-1
Hexadecimal Code Table

Hexadecimal Digit	Binary Equivalent	Decimal Equivalent
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
A	1010	10
B	1011	11
C	1100	12
D	1101	13
E	1110	14
F	1111	15

Note that the Micro-68 uses the lower case letters b and d for those hexadecimal digits.

LABORATORY 1

Simple Programming

ONES COMPLEMENT

The first program we will write is an inverter or complement program. It will simply take the contents of memory location 40, complement each bit, and place the result in memory location 41. The program is:

```
LDAA  $40        GET DATA
COMA             COMPLEMENT
STAA  $41        STORE RESULT
SWI
```

Let's now look at each instruction:

1) LDAA $40 loads accumulator A with the contents of memory location 0040 (hexadecimal). The "$" means hexadecimal rather than decimal. Remember that the address is 4 digits long. However, when the first two digits are zero we can often drop them and use the DIRECT mode (second addressing mode on the 6800 instruction card).

2) COMA complements accumulator A, i.e., replaces each "0" bit with a "1" and each "1" with a "0" just like a set of inverter gates.

3) STAA $41 stores the contents of accumulator A in memory location 41 (hex). Again, you can use the direct mode since the first two digits of the address are zero.

4) SWI (software interrupt) returns control to the monitor. You should put this instruction or some exit instruction at the end of all programs so that the computer goes back to the monitor rather than wandering off aimlessly.

To enter the program, you must look up the hexadecimal codes on the 6800 programmer's instruction card. Remember to use the direct mode for LDAA and STAA. The column marked # on the programmer's card tells how many words of memory the instruction requires, the column marked ~ tells how many clock cycles it takes to execute.

The hexadecimal program (starting in memory location 0) is:

Memory Address (Hex)	Instruction (Mnemonic)	Memory Contents (Hex)
00	LDAA $40	96
01		40
02	COMA	43
03	STAA $41	97
04		41
05	SWI	3F

Note that both the LDAA and STAA require a second byte for an address.

Enter the program and place the data (36 hex) in memory location 40. Then run the program. The answer should be C9 (hex). Why? (Look at the bit patterns.) Examine memory location 41 and see if it contains C9.

Run the program again with the data 00. The answer should be FF.

TWOS COMPLEMENT

The instruction NEGA (40 hex) will produce the twos complement (the ones complement plus 1). Modify the program so that it produces the twos complement and run the following examples. (The location enclosed in parentheses means "the contents of location.")

1) Data = (40) = 36
   Result = (41) = CA

2) Data = (40) = 00
   Result = (41) = 00

USING ACCUMULATOR B

The Motorola 6800 has two accumulators (A and B) which are generally equivalent. Rewrite the two earlier programs so that they use accumulator B and run them. Remember to use the codes:

LDAB	DIRECT	D6
STAB	DIRECT	D7
COMB		53
NEGB		50

LABORATORY 2

Introduction to Motorola 6800 Input/Output

The Motorola 6800 treats input/output devices just like memory locations. For example, the keyboard of the Micro-68 used in this book uses memory addresses 8004 and 8006. Memory location 8004 has 1 bit for each of keys 0-7 organized as follows:

Bit	7	6	5	4	3	2	1	0
8004								
Key	7	6	5	4	3	2	1	0

Memory location 8006 similarly has 1 bit for each of keys 8-F:

Bit	7	6	5	4	3	2	1	0
8006								
Key	F	E	D	C	B	A	9	8

The bit is "0" if the key is being pressed, "1" otherwise.

You may use memory locations 8004 and 8006 just like any other locations except that it doesn't make much sense to store data there. Note that the first two digits of these addresses are not both zero so you'll have to use the EXTENDED mode (the fourth addressing mode on the instruction card).

The programs for keyboard I/O are written in symbolic form, where STATUS has value $8004 in the following examples.

WAITING FOR A KEY CLOSURE

The following program will wait until you press key 5 and will then return control to the monitor:

```
WAITK     LDAA  STATUS        GET KEYS 0-7
          ANDA  #%00100000    IS KEY 5 BEING PRESSED?
          BNE   WAITK         NO, WAIT UNTIL IT IS
          SWI
```

Let us now look at each instruction:

1) LDAA STATUS loads accumulator A with the contents of memory location STATUS, i.e., with the state of keys 0 through 7. Remember that you must use the code for LDAA extended (B6).

2) ANDA #%00100000 logically ANDs the contents of accumulator A with the binary number 00100000. The # means "immediate" (i.e., the data is right there in the next word rather than somewhere else in memory), and the % means "binary". The result of the logical AND is 0 if key 5 is being pressed (bits = 0), and 00100000 if switch 5 is not being pressed. This process is called masking.

3) BNE WAITK causes the computer to execute the instruction at memory location WAITK next if the ZERO bit is 0. Otherwise, the computer proceeds to the next instruction in sequence (i.e., SWI). Remember that the ZERO bit is "1" if the result of the last operation was zero.

The program in hex is:

Memory Address (Hex)	Instruction (Mnemonic)	Memory Contents (Hex)
00	WAITK LDAA STATUS	B6
01		80
02		04
03	ANDA #%00100000	84
04		20
05	BNE WAITK	26
06		F9
07	SWI	3F

Note the following:

1) LDAA STATUS uses the extended mode and has the complete address in the next two words (most significant digits first).

2) ANDA #%00100000 uses the immediate mode and has the data in the next word. Note that the function of the mask is clearest in binary.

3) BNE WAITK uses a relative address which tells the computer how far to branch from the end of the instruction. The distance is a twos complement number so it can be either positive or negative. You can calculate the distance by subtracting the address of the next instruction from the address of the destination. In this case, the distance is:

```
 0000      0000
-0007  =  +FFF9
          -----
            F9
```

Hexadecimal subtraction is tricky and you should always check the result. You can drop the 2 most significant digits. Remember that subtraction is the same as adding the twos complement.

Enter and run the program. What happens when you press key 0? What happens when you press key 5?

Change the program so that it responds to key 6 and run it. Next make the program respond to key 3.

Key 7 is particularly simple to use since you can test bit 7 as the negative (N) bit. The following program will do the job:

```
WAITK     LDAA   STATUS     GET KEYS 0-7
          BMI    WAITK      WAIT IF KEY 7 IS NOT BEING PRESSED
          SWI
```

The hexadecimal program is:

Memory Address (Hex)	Instruction (Mnemonic)	Memory Contents (Hex)
00	WAITK LDAA STATUS	B6
01		80
02		04
03	BMI WAITK	2B
04		FB
05	SWI	3F

Note here that:

1) BMI causes a branch if the NEGATIVE bit is "1". The NEGATIVE bit is the most significant bit (bit 7) of the previous result.

2) The offset for BMI is:

$$\begin{array}{r} 00 \\ \underline{-05} \end{array} = \begin{array}{r} 00 \\ \underline{+FB} \\ FB \end{array}$$

(Check it!)

Enter and run the program for key 7. Key 0 is also simple to use with the following program:

```
WAITK     LSR    STATUS       IS KEY 0 BEING PRESSED?
          BCS    WAITK        NO, WAIT
          SWI
```

Enter and run this program. LSR extended is 74 and BCS is 25.

WAITING FOR 2 CLOSURES

You can combine programs to wait for more than one closure. The following program will wait for keys 2 and 5 in that order:

```
WAIT1     LDAA   STATUS
          ANDA   #%00000100   IS KEY 2 BEING PRESSED?
          BNE    WAIT1        NO, WAIT
WAIT2     LDAA   STATUS
          ANDA   #%00100000   IS KEY 5 BEING PRESSED?
          BNE    WAIT2
          SWI
```

The hexadecimal program is:

Memory Address (Hex)	Instruction (Mnemonic)			Memory Contents (Hex)
00	WAIT1	LDAA	STATUS	B6
01				80
02				04
03		ANDA	#%00000100	84
04				04
05		BNE	WAIT1	26
06				F9
07	WAIT2	LDAA	STATUS	B6
08				80
09				04
0A		ANDA	#%00100000	84
0B				20
0C		BNE	WAIT2	26
0D				F9
0E		SWI		3F

Enter this program and run it. Note that the two relative addresses in the program are:

1) BNE WAIT1

$$\begin{array}{r} 00 \\ \underline{-07} \end{array} = \begin{array}{r} 00 \\ \underline{+F9} \\ F9 \end{array}$$

2) BNE WAIT2

$$\begin{array}{r} 07 \\ \underline{-0E} \end{array} = \begin{array}{r} 07 \\ \underline{+F2} \\ F9 \end{array}$$

Modify the last program to perform the following tasks:

1) Wait for key 0 followed by key 7.

2) Wait for keys 2 and 5 followed by keys 0 and 7.

3) Wait for either key 0 followed by key 7 or key 7 followed by key 0.

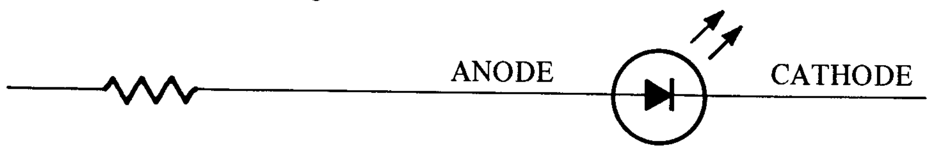
What is the difference if you replace ANDA with SUBA?

LABORATORY 3

Using the Micro-68 Displays

The Micro-68's LED (light-emitting diode) displays use memory locations 8008 and 800A. Memory location 8008 is the data, and memory location 800A determines which displays are on.

The basic LED configuration is simple:



Current flows (and the LED lights) when the anode is positive with respect to the cathode. The resistor limits the current through the LED. The computer can control the LED either by placing a logic "1" at the anode (if the cathode is grounded) or by placing a logic "0" at the cathode (if the anode is tied to +5 volts).

Each 7-segment display consists of 7 segment LEDs and a decimal point organized as shown in Figure 3-1. The Micro-68 uses the following arrangement to control the LEDs:

1) Memory location 8008 has one bit for each segment and the decimal point as follows:

Bit	7	6	5	4	3	2	1	0
8008	a	b	c	d	e	f	g	dp

A zero in the bit position turns the LED on, a one turns it off.

2) Memory location 800A has six bits which control whether entire seven-segment displays are on or off; it is organized as follows:

Bit	7	6	5	4	3	2	1	
800A	Not Used	LED 6	LED 5	LED 4	LED 3	LED 2	LED 1	Not Used

A one in the bit position turns the display on. The LED numbering is:

LED 1	LED 2	LED 3	LED 4	LED 5	LED 6

where LEDs 1 through 4 are the address displays (on the left) and 5 and 6 are the data displays (on the right). We will use LIGHTS in place of $800A, and GLOW in place of $8008 in the following programs, but remember their values.

TURNING ON AN LED

The following program will light all the decimal point LEDs.

```
LDAA   #$FF
STAA   LIGHTS        TURN ALL DISPLAYS ON
LDAA   #%11111110
STAA   GLOW          DECIMAL POINTS ON
SWI
```

The hexadecimal version of the program is:

Memory Address (Hex)	Instruction (Mnemonic)	Memory Contents (Hex)
00	LDAA #$FF	86
01		FF
02	STAA LIGHTS	B7
03		80
04		0A
05	LDAA #%11111110	86
06		FE
07	STAA GLOW	B7
08		80
09		08
0A	SWI	3F

Note that LDAA immediate (86) loads accumulator A with the data in the address following the instruction.

Enter and run the program. What happens?

The problem is that the program only turns the displays on for a few microseconds before the monitor takes control. You can solve the problem by not returning to the monitor. End the program with:

```
HERE            BRA HERE

          or

0A            HERE BRA HERE       20
0B                                FE
```

This instruction just jumps to itself until you reset the computer. The relative offset is:

```
 0A              0A
--0C     =      +F4
                ---
                 FE
```

Now run the program and see what happens. Table 3-1 contains the patterns required in LIGHTS to turn on the various displays; Table 3-2 contains the patterns required in GLOW to turn on various segments. Try some combinations. Remember that ZERO is on, ONE is off. To turn on more than one segment, clear each bit. For example:

Segments	Pattern (Hex)
a and b	3F
a and e	77
b and c	9F
b and g	8D
e and f	F3

Figure 3-1
Seven-Segment
Display

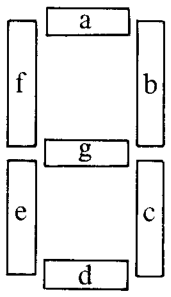


Table 3-1
Patterns for Turning on Displays
(Memory Location 800a)

Display Number	Pattern (Hex)
1	C1
2	A1
3	91
4	89
5	85
6	83

Remember that bits 0 and 7 really don't matter since they're not used. To turn on more than one display, just set each bit. For example:

Display Number	Pattern (Hex)
1 and 2	E1
1 and 5	C5
2 and 3	B1
3 and 5	95
4 and 6	8B

Table 3-2
Patterns for Turning on Segments
(Memory Location 8008)

Segments	Pattern (Hex)
a	7F
b	BF
c	DF
d	EF
e	F7
f	FB
g	FD
dp	FE

PROVIDING A DELAY

Of course, in most real applications, we'll want to turn things on for a specified amount of time. The following program provides a delay by counting with the index or X register:

```
                  LDX #0                    DELAY LOOP
         COUNT    INX
                  BNE COUNT
```

Remember that the X register is 16 bits or 4 hexadecimal digits long so the program counts from 0000 to FFFF before it reaches zero again. When the computer adds 1 to FFFF, the result is zero.

Enter the delay loop at the end of the display program, i.e.:

0A	LDX #0	CE
0B		00
0C		00
0D	COUNT INX	08
0E	BNE COUNT	26
0F		FD
10	SWI	3F

Run the program. What happens? You can change the length of the delay by changing memory locations 0B and 0C (the starting count). Try the values in the following table until you can no longer see the display.

Delay (seconds)	Contents of 000B (Hex)
1	00
1/2	80
1/4	C0
1/8	E0
1/16	F0
1/32	F8
1/64	FC
1/128	FE
1/256	FF

You can turn on first one segment (decimal point) and then another (g) with the following program:

```
         START    LDAA    #$FF          ALL DISPLAYS ON
                  STAA    LIGHTS
                  LDAA    #%11111110
                  STAA    GLOW          DECIMAL POINTS ON
                  LDX     #0
         DLY1     INX                   DELAY 1
                  BNE     DLY1
                  LDAA    #%11111101
                  STAA    GLOW          SEGMENT G ON
                  LDX     #0
```

```
DLY2        INX                     DELAY 2
            BNE     DLY2
            JMP     START           REPEAT FOREVER
```

The hexadecimal version is:

Memory Address (Hex)	Instruction (Mnemonic)	Memory Contents (Hex)
00	LDAA #$FF	86
01		FF
02	STAA LIGHTS	B7
03		80
04		0A
05	START LDAA #%11111110	86
06		FE
07	STAA GLOW	B7
08		80
09		08
0A	LDX #0	CE
0B		00
0C		00
0D	DLY1 INX	08
0E	BNE DLY1	26
0F		FD
10	LDAA #%11111101	86
11		FD
12	STAA GLOW	B7
13		80
14		08
15	LDX #0	CE
16		00
17		00
18	DLY2 INX	08
19	BNE DLY2	26
1A		FD
1B	JMP START	7E
1C		00
1D		05

Try running this program. What happens when you reduce the delays (location 000B and 0016) according to the previous table? If you want to form some digits or letters, use the codes in Tables 8-1 and 8-2.

LABORATORY 4

Using the Micro-68 Keyboard

Actually identifying which key has been pressed requires some manipulation. Note that the Micro-68 uses its keyboard both for data and for commands. Let's first investigate using the keyboard for data.

Table 4-1 contains the binary patterns which are produced in STATUS by pressing the various keys 0 to 7. If no keys are pressed, the pattern is all "1's" (i.e., FF hexadecimal). So the following program will wait until you press a key in the 0 to 7 group.

```
WAITK     LDAA   STATUS     GET KEYS 0-7
          CMPA   #$FF       ARE ANY BEING PRESSED?
          BEQ    WAITK      NO, WAIT
          SWI
```

The hexadecimal version of the program is:

Memory Address (Hex)	Instruction (Mnemonic)	Memory Contents (Hex)
00	WAITK LDAA STATUS	B6
01		80
02		04
03	CMPA #$FF	81
04		FF
05	BEQ WAITK	27
06		F9
07	SWI	3F

Note that the instruction CMPA #$FF subtracts FF from the contents of accumulator A and changes the flags appropriately but doesn't store the result anywhere.

Enter and run the program. Now change the program so that it waits until you stop pressing a key.

Table 4-1
Patterns Produced by Various Key Closures

Key Pressed	Pattern	
	Binary	Hex
0	11111110	FE
1	11111101	FD
2	11111011	FB
3	11110111	F7
4	11101111	EF
5	11011111	DF
6	10111111	BF
7	01111111	7F
None	11111111	FF

Now combine the two programs with the original version first. The combined program should wait for you to press a key and then release it.

The combined program is:

```
WAITC      LDAA   STATUS    GET KEYS 0-7
           CMPA   #$FF      ARE ANY BEING PRESSED?
           BEQ    WAITC     NO, WAIT UNTIL ONE IS
WAITO      LDAA   STATUS
           CMPA   #$FF      YES, WAIT FOR IT TO BE
                            RELEASED
           BNE    WAITO
           SWI
```

The hexadecimal version is:

Memory Address (Hex)	Instruction (Mnemonic)	Memory Contents (Hex)
00	WAITC LDAA STATUS	B6
01		80
02		04
03	CMPA #$FF	81
04		FF
05	BEQ WAITC	27
06		F9
07	WAITO LDAA STATUS	B6
08		80
09		04
0A	CMPA #$FF	81
0B		FF
0C	BNE WAITO	26
0D		F9
0E	SWI	3F

DEBOUNCING A KEY

If you try the combined program several times, you'll probably find that it often doesn't wait for you to release the key. This is because a mechanical key (or any other switch) doesn't provide a clean closure. Instead, the switch bounces back and forth for a while until it settles down. The computer, however, cannot tell the bounce from an actual release of the key since the effects of the bounce and the release are the same (a logic '1' in the corresponding bit).

The solution to this problem is not to examine the key again until it has stopped bouncing. Since the settling (or debouncing time) is usually less than 1 millisecond, a delay of one millisecond will do the job. A simple delay program is:

```
        LDX    #N
DLY     DEX
        BNE    DLY
```

If you look up the instructions on your card, you'll find that they require the following number of clock cycles:

LDX	# (immediate)	–3
DEX		–4
BNE		–4

So the delay program takes:

$$t_c \times (N\times(4+4) + 3) \text{ microseconds}$$

where t_c is the number of microseconds in a clock cycle.

To get a delay of 1 millisecond (1000 microseconds) at the standard 500 KHz rate of the Micro-68 requires an N given by:

$$\begin{aligned} 2 \times (8N + 3) &= 1000 \\ 16N &= 994 \\ N &= 62 \text{ (decimal)} \\ &= 3E \text{ (hex)} \end{aligned}$$

(See, high school algebra really is good for something besides counting apples and oranges).

The one millisecond delay program is:

```
        LDX    #$3E
DLY     DEX
        BNE    DLY
```

or in hexadecimal:

Instruction (Mnemonic)	Memory Contents (Hex)
LDX #$3E	CE
	00
	3E
DLY DEX	09
BNE DLY	26
	FD

Add the delay routine between the two sections of the previous program and try it. The Micro-68 should now wait for you to press and release a key. What happens when you press several keys at once? How would you extend the program so that it responds to any of the sixteen keys? Does the monitor program respond to the operator pressing or releasing a key? How can you tell?

IDENTIFYING THE KEY

The next question is how to transform a key closure into the corresponding digit. Look at Table 4-1 again.

The bit which is "0" is the one which identifies the key, i.e., bit 0 is "0" for key 0, bit 1 for key 1, etc. So all you have to do is figure out which bit is "0." You can do that by counting the number of shifts required to get a "0" bit into the CARRY, i.e., if accumulator A contains the key closures from STATUS:

```
            CLRB            KEY NUMBER = 0
SRKEY       LSRA            IS NEXT BIT "0"?
            BCC DONE        YES, DONE
            INCB            NO, KEY NUMBER = KEY NUMBER + 1
            BRA SRKEY
DONE        SWI
```

Accumulator B contains the key number at the end of the program (see the flowchart in Figure 4-1).

An alternative identification program uses somewhat different initial conditions in order to eliminate one of the jump instructions, i.e.,

```
            LDAB    #$FF    KEY NUMBER = -1
SRKEY       INCB            KEY NUMBER = KEY NUMBER + 1
            LSRA            IS NEXT BIT "0"?
            BCS     SRKEY   NO, KEEP LOOKING FOR "0" BIT
            SWI
```

The entire program is:

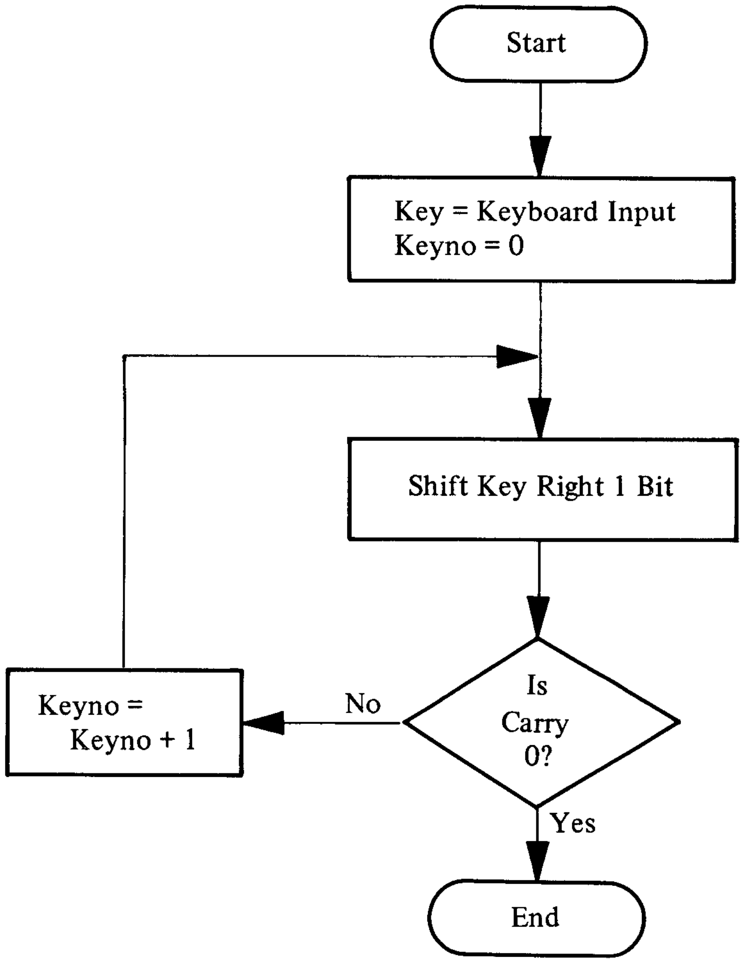
1) Wait for a key closure
2) Wait 1 ms. to debounce the key
3) Identify key

The assembly language version is:

```
WAITK       LDAA    STATUS  GET KEYS 0-7
            CMPA    #$FF    ARE ANY BEING PRESSED?
            BEQ     WAITK   NO, WAIT
            LDX     #$3E
DLY         DEX
            BNE     DLY
            CLRB            KEY NUMBER = 0
SRKEY       LSRA            IS NEXT BIT "0"?
            BCC     DONE    YES, DONE
            INCB            NO, KEY NUMBER = KEY NUMBER + 1
            BRA     SRKEY
DONE        STAB    $40
            SWI
```

This program saves the key code in memory location 40.

Figure 4-1
Flowchart for Key Identification



The machine language version is:

Memory Address (Hex)	Instruction (Mnemonic)			Memory Contents (Hex)
00	WAITK	LDAA	STATUS	B6
01				80
02				04
03		CMPA	#$FF	81
04				FF
05		BEQ	WAITK	27
06				F9
07		LDX	#$3E	CE
08				00
09				3E
0A	DLY	DEX		09
0B		BNE	DLY	26
0C				FD
0D		CLRB		5F
0E	SRKEY	LSRA		44
0F		BCC	DONE	24
10				03
11		INCB		5C
12		BRA	SRKEY	20
13				FA
14	DONE	STAB	$40	D7
15				40
16		SWI		3F

Enter this program and try it for keys 0 through 7. What happens if you press several keys at once? How could you change the program so that it always takes the highest numbered key? Now rewrite the program so that it handles all 16 keys (hint: set B to zero before examining keys 0-7 and to eight before examining keys 8-F).

LABORATORY 5

Handling Data Arrays on the Micro-68

Most computer tasks involve applying the same instructions to an entire set of data (or array). Typical examples of such tasks are calculating averages, finding the largest element for scaling purposes, editing a line of text, collecting data for storage on magnetic tape, and arranging a sequence of operations for a process or industrial control system.

INDEXING

The Motorola 6800 uses indexing to apply the same instructions to each element in an array of data. Indexing means that the processor adds the address in the index register to the offset (in the word following the instruction) to get the actual address of the data. The actual address used by the instruction is called the effective address. For example, the instruction LDAA $20, X works as follows (see Figure 5-1):

1) The processor adds 20 (hex) to the contents of the index register (1000 hex. The result is 1020.
2) The processor places the contents of memory location 1020 in accumulator A.

Note that the index register is 16 bits (4 digits) long while the offset is 8 bits (2 digits) long. The important thing, though, is that you can change the address from which the CPU gets the data in step 2 by changing the contents of the index register. For example, INX (08) adds 1 to the index register and DEX (09) substracts 1 from it.

SUM OF DATA

The following program will add the contents of memory locations 41, 42, 43, and 44 and place the sum (ignoring carries) in memory location 40.

```
        LDAB   #4       COUNT = 4
        CLRA            SUM = 0
        LDX    #$41     INDEX REGISTER = START OF ARRAY
ADDW    ADDA   X        SUM = SUM + DATA
        INX
        DECB
        BNE    ADDW
        STAA   $40      STORE SUM
        SWI
```

Note that ADDA X is shorthand for ADDA 0, X.

The hexadecimal version of the program is:

Memory Address (Hex)	Instruction (Mnemonic)			Memory Contents (Hex)
00		LDAB	#4	C6
01				04
02		CLRA		4F
03		LDX	#$41	CE
04				00
05				41
06	ADDW	ADDA	X	AB
07				00
08		INX		08
09		DECB		5A
0A		BNE	ADDW	26
0B				FA
0C		STAA	$40	97
0D				40
0E		SWI		3F

Note that we have used the indexed form of ADDA (AB). Indexing is the third addressing mode on the instruction card. The relative offset here is:

$$\begin{array}{rcl} 0006 & & 06 \\ -000C & = & \underline{+F4} \\ & & FA \end{array}$$

Try running this program with the following array of data:

```
(41) = 07
(42) = 23
(43) = 31
(44) = 20
```

The result should be:

```
(40) = 7B
```

Remember that all the numbers are hexadecimal.

Now replace (42) with F1. What is the result? Why?

Change the program so that it will handle the following array of numbers:

```
(41) = 07
(42) = 23
(43) = 31
(44) = 20
(45) = 16
(46) = 38
```

The result should be (40) = C9.

USING AN ENDING MARKER

If you're not sure how long your array of data is (or don't want to bother counting it), you can always end the array with a special marker. In this case, 0 makes a good marker since it doesn't add anything to the sum. The program is:

```
        CLRA                SUM = 0
        LDX     #$41        INDEX REGISTER = START OF ARRAY
ADDW    LDAB    X           IS NEXT ELEMENT 0?
        BEQ     DONE        YES, DONE
        ABA                 NO, SUM = SUM + DATA
        INX
        BRA     ADDW
DONE    STAA    $40
        SWI
```

The hexadecimal program is:

Memory Address (Hex)	Instruction (Mnemonic)			Memory Contents (Hex)
00		CLRA		4F
01		LDX	#$41	CE
02				00
03				41
04	ADDW	LDAB	X	E6
05				00
06		BEQ	DONE	27
07				04
08		ABA		1B
09		INX		08
0A		BRA	ADDW	20
0B				F8
0C	DONE	STAA	$40	97
0D				40
0E		SWI		3F

Note that:

1) ABA adds the A and B accumulators and places the result in A.

2) The relative offsets are:

BEQ DONE
$$\begin{array}{r} 0C \\ \underline{-08} \\ 04 \end{array}$$

BRA ADDW
$$\begin{array}{r} 04 \\ \underline{-0C} \end{array} = \begin{array}{r} 04 \\ \underline{+F4} \\ F8 \end{array}$$

Try this program on the data given before except put zero in the location after the last item. What happens if you forget the final zero? What happens if you place zero in memory location 42?

FORMING A CHECKSUM

Change the program so that it EXCLUSIVE ORs the numbers together instead of adding them. This result is called a logical sum or checksum and is often used to check for errors in tape records. Note that EXCLUSIVE OR is the same as addition except that there are no carries, i.e.,

A	B	SUM	CARRY	A⊕B
0	0	0	0	0
0	1	1	0	1
1	0	1	0	1
1	1	0	1	0

DISPLAYING AN ARRAY

We can use indexing to place different data on each of the six LED displays. The following program will do the job:

```
START     LDAB    #%00000010    TURN ON DISPLAY 1
          STAB    LIGHTS
          LDX     #$40          INDEX REGISTER =
                                  START OF ARRAY
DSPLY     LDAA    X             GET DATA FROM ARRAY
          STAA    GLOW          SEND DATA TO DISPLAY
          STX     $50
          LDX     #0            DELAY A WHILE (BUT
                                  SAVE X REGISTER)
DELAY     INX
          BNE     DELAY
          LDX     $50
          INX                   NEXT DATA IN ARRAY
          ASL     LIGHTS        TURN ON NEXT DISPLAY
          BPL     DSPLY         START OVER IF ALL
                                  DISPLAYS HANDLED
          JMP     START
```

The program start with a "1" in bit 1 of LIGHTS and continues until that "1" moves into the NEGATIVE (or sign bit). ASL (arithmetic shift left) shifts the contents of LIGHTS left one bit and clears the empty bit. BPL branches back if bit 7 is still "0." Remember that the Micro-68 does not use bits 0 and 7 of LIGHTS.

The hexadecimal version of the program is:

Memory Address (Hex)	Instruction (Mnemonic)			Memory Contents (Hex)
00	START	LDAB	#$00000010	C6
01				02
02		STAB	LIGHTS	F7
03				80
04				0A
05		LDX	#$40	CE
06				00
07				40
08	DSPLY	LDAA	X	A6
09				00
0A		STAA	GLOW	B7
0B				80
0C				08
0D		STX	$50	DF
0E				50
0F		LDX	#0	CE
10				00
11				00
12	DELAY	INX		08
13		BNE	DELAY	26
14				FD
15		LDX	$50	DE
16				50
17		INX		08
18		ASL	LIGHTS	78
19				80
1A				0A
1B		BPL	DSPLY	2A
1C				EB
1D		JMP	START	7E
1E				00
1F				00

The offsets are:

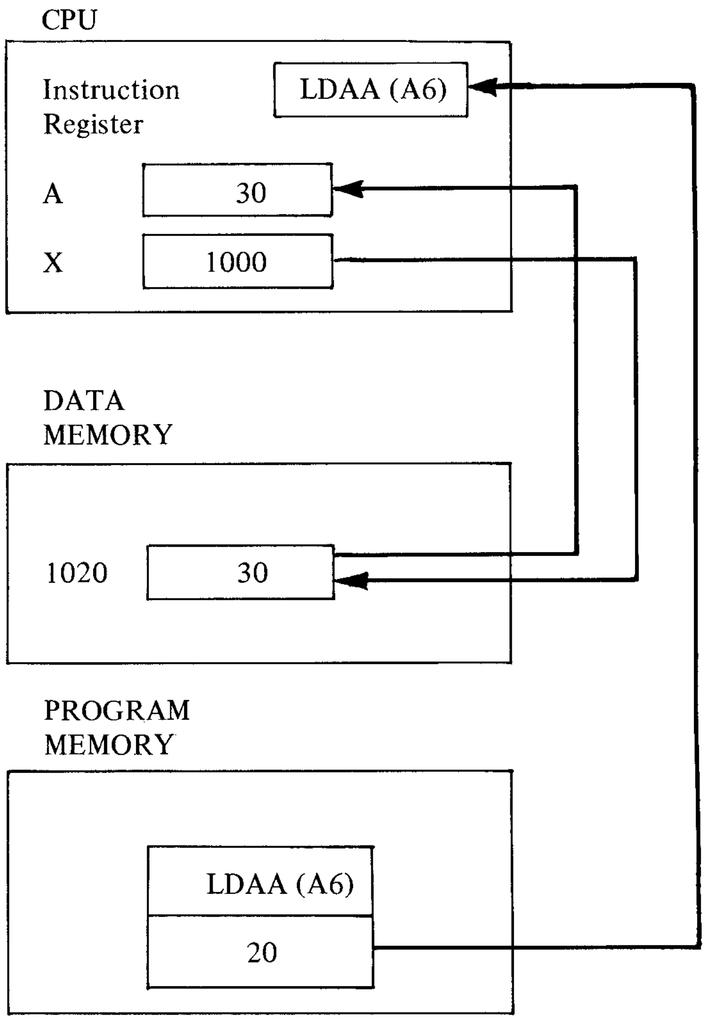
```
BNE DELAY      12       12
              -15   =  +EB
                        FD

BPL DSPLY      08       08
              -1D   =  +E3
                        EB
```

Run the program with the following data:

Display 1	=	(40)	=	91
Display 2	=	(41)	=	61
Display 3	=	(42)	=	E3
Display 4	=	(43)	=	E3
Display 5	=	(44)	=	03
Display 6	=	(45)	=	FF

What happens when you reduce the delay by changing memory location 0010 to 80, C0, E0, F0, F8, FC, FE, FF? Try devising and displaying some other data. You can use the patterns in Tables 8-1 and 8-2.

Figure 5-1
Execution of the LDAA $20, X Instructions



We can change the location in data memory by changing the value in the index register. If, for example, the CPU executes an INX instruction, the next LDAA $20, X fill fetch the data from address 1021.

LABORATORY 6

Forming Arrays on the Micro-68

We have already seen how you can handle an array. The question now is how to form one. The procedure requires a counter and a pointer. The pointer contains the address of the next empty location in the array; the counter contains the length of the array.

The basic procedure is:

<u>Step 1</u> Initialization

Pointer	=	Start of Array
Counter	=	0

<u>Step 2</u> Place Data in Array

(Pointer) = Data. Remember that the parentheses mean "contents of".

<u>Step 3</u> Update Counter and Pointer

Pointer	=	Pointer + 1
Counter	=	Counter +1

Of course, this simple procedure assumes that you have all the data available. It also provides no way to end the array formation. One method is:

<u>Step 4</u> See if Enough Data has been Collected

If Counter = length, then done; Otherwise, Return to Step 2

CLEARING AN ARRAY

A simple example just clears an area of memory, say memory locations 40 through 47:

```
         CLRA                DATA = 0
         NOP
         LDX     #$40        POINTER = START OF ARRAY
         LDAB    #8          LENGTH OF ARRAY = 8
CLEAR1   STAA    X           CLEAR AN ELEMENT IN ARRAY
         INX
         DECB
         BNE     CLEAR1
         SWI
```

(The NOP – no operation – doesn't do anything except make the program easier to change).

The hexadecimal version of this program is:

Memory Address (Hex)	Instruction (Mnemonic)	Memory Contents (Hex)
00	CLRA	4F
01	NOP	01
02	LDX #$40	CE
03		00
04		40
05	LDAB #8	C6
06		08
07	CLEAR1 STAA X	A7
08		00
09	INX	08
0A	DECB	5A
0B	BNE CLEAR1	26
0C		FA
0D	SWI	3F

Enter the program and run it. Make the necessary changes to have the program do the following tasks and test each revision.

1) Clear memory locations 40 through 49

2) Clear memory locations 50 through 59

3) Place 80 hex in memory locations 50 through 59

PLACING VALUES IN AN ARRAY

The next step, of course, is to put different numbers in each entry. What happens if, in the original program, you change STAA X to STAB X (i.e., E7 instead of A7)? How would you change the program to reverse the order of the numbers? Hint: use INCA but remember to adjust the offset in BNE CLEAR1.

The following program will set each element to twice the preceding element, starting with 1:

```
         LDAA    #1      FIRST ELEMENT IS 1
         LDX     #$40    POINTER = START OF ARRAY
         LDAB    #8      LENGTH OF ARRAY = 8
SETUP    STAA    X       PLACE ELEMENT IN ARRAY
         ASLA            NEXT ELEMENT = ELEMENT X2
         INX
         DECB
         BNE     SETUP
         SWI
```

The hexadecimal version of this program is:

Memory Address (Hex)	Instruction (Mnemonic)			Memory Contents (Hex)
00		LDAA	#1	86
01				01
02		LDX	#$40	CE
03				00
04				40
05		LDAB	#8	C6
06				08
07	SETUP	STAA	X	A7
08				00
09		ASLA		48
0A		INX		08
0B		DECB		5A
0C		BNE	SETUP	26
0D				F9
0E		SWI		3F

This program really doesn't require a counter since you can just let the "1" bit in position 0 move across the word. Change the program to eliminate the counter. Rewrite the program so that the sequence is:

```
(40) = 80 (10000000)
(41) = C0 (11000000)
(42) = E0 (11100000)
(43) = F0 (11110000)
(44) = F8 (11111000)
(45) = FC (11111100)
(46) = FE (11111110)
(47) = FF (11111111)
```

Hint: Use ARITHMETIC SHIFT RIGHT. Why do you think this instruction is part of the set? What are the values of the last set of numbers if you consider them as signed twos complement numbers?

ENTERING INPUT DATA INTO AN ARRAY

The next step is to form an array from the keyboard using input data. For the time being, let's just use keys 0 – 7.

The procedure for forming the array will be:

<u>Step 1</u> Initialization

Pointer = Start of Array (40)
Counter = Length of Array (4)

<u>Step 2</u> Wait for Key to be Pressed

Step 3 Debounce Key with 1 ms. Delay

Step 4 Identify Key

Step 5 Place Key in Array

(Pointer) = Key

Step 6 Update Counter and Pointer

Pointer = Pointer + 1
Counter = Counter + 1

Step 7 Wait for Key to be Released

Step 8 If Counter ≠ 0, Return to Step 2

The assembly language program is (see flowchart in Figure 6-1):

```
        LDX     #$40     POINTER = START OF ARRAY
        STX     $50
        LDAA    #4       COUNTER = LENGTH OF ARRAY
        STAA    $52
WAITC   LDAA    STATUS   GET KEYS 0 TO 7
        CMPA    #$FF     ARE ANY BEING PRESSED?
        BEQ     WAITC    NO, WAIT
        LDX     #$3E
DLY     DEX              1 MS. TO DEBOUNCE
        BNE     DLY
        CLRB             KEY NUMBER = 0
SRKEY   LSRA             IS NEXT BIT "0"?
        BCC     STKEY    YES, DONE
        INCB             NO, KEY NUMBER = KEY NUMBER +1
STKEY   BRA     SRKEY
        LDX     $50      GET POINTER
        STAB    X        SAVE KEY NUMBER IN ARRAY
        INX              UPDATE AND SAVE POINTER
        STX     $50
WAITO   LDAA    STATUS   GET KEYS 0 TO 7
        CMPA    #$FF     ARE ANY BEING PRESSED?
        BNE     WAITO    YES, WAIT UNTIL ALL KEYS RELEASED
        DEC     $52
        BNE     WAITC
        SWI
```

The hexadecimal program is:

Memory Address (Hex)	Label	Instruction (Mnemonic)	Operand	Memory Contents (Hex)
00		LDX	#$40	CE
01				00
02				40
03		STX	$50	DF
04				50
05		LDAA	#4	86
06				04
07		STAA	$52	97
08				52
09	WAITC	LDAA	STATUS	B6
0A				80
0B				04
0C		CMPA	#$FF	81
0D				FF
0E		BEQ	WAITC	27
0F				F9
10		LDX	#$3E	CE
11				00
12				3E
13	DLY	DEX		09
14		BNE	DLY	26
15				FD
16		CLRB		5F
17	SRKEY	LSRA		44
18		BCC	STKEY	24
19				03
1A		INCB		5C
1B		BRA	SRKEY	20
1C				FA
1D	STKEY	LDX	$50	DE
1E				50
1F		STAB	X	E7
20				00
21		INX		08
22		STX	$50	DF
23				50
24	WAIT0	LDAA	STATUS	B6
25				80
26				04
27		CMPA	#$FF	81
28				FF
29		BNE	WAIT0	26
2A				F9

Memory Address (Hex)	Instruction (Mnemonic)	Memory Contents (Hex)
2B	DEC $52	7A
2C		00
2D		52
2E	BNE WAITC	26
2F		D9
30	SWI	3F

Enter this program and run it. Try several different key sequences. Which sequences do you think would make an adequate test?

Change the program to perform the following tasks:

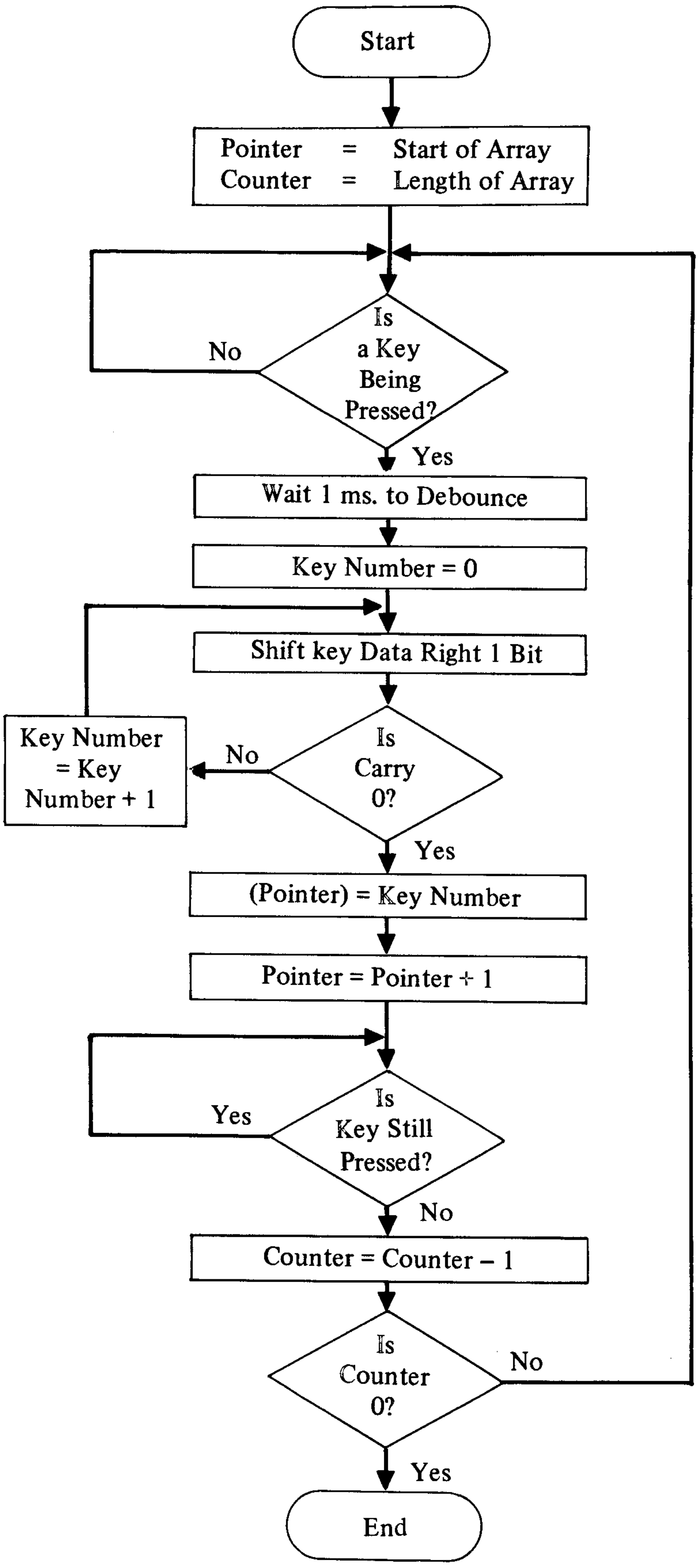
1) Save six key numbers in memory locations 40 through 45

2) Save four key numbers in memory locations 48 through 4B

What happens if you try to use memory locations 50 through 53 for the closures? How would you solve this problem?

What happens if you don't wait for the key to be released? You can try this by branching around the section that waits for the end, i.e., place 20 (BRA) in memory location 24 and 05 in memory location 25. See also what happens if you press several keys at once.

Figure 6-1
Flowchart for Forming Array from Keyboard



LABORATORY 7

Designing and Debugging Programs on the Micro-68

Now that you've seen some fairly complex programs, you're probably wondering how such programs get written and tested. In practice, each program goes through many stages, i.e.,

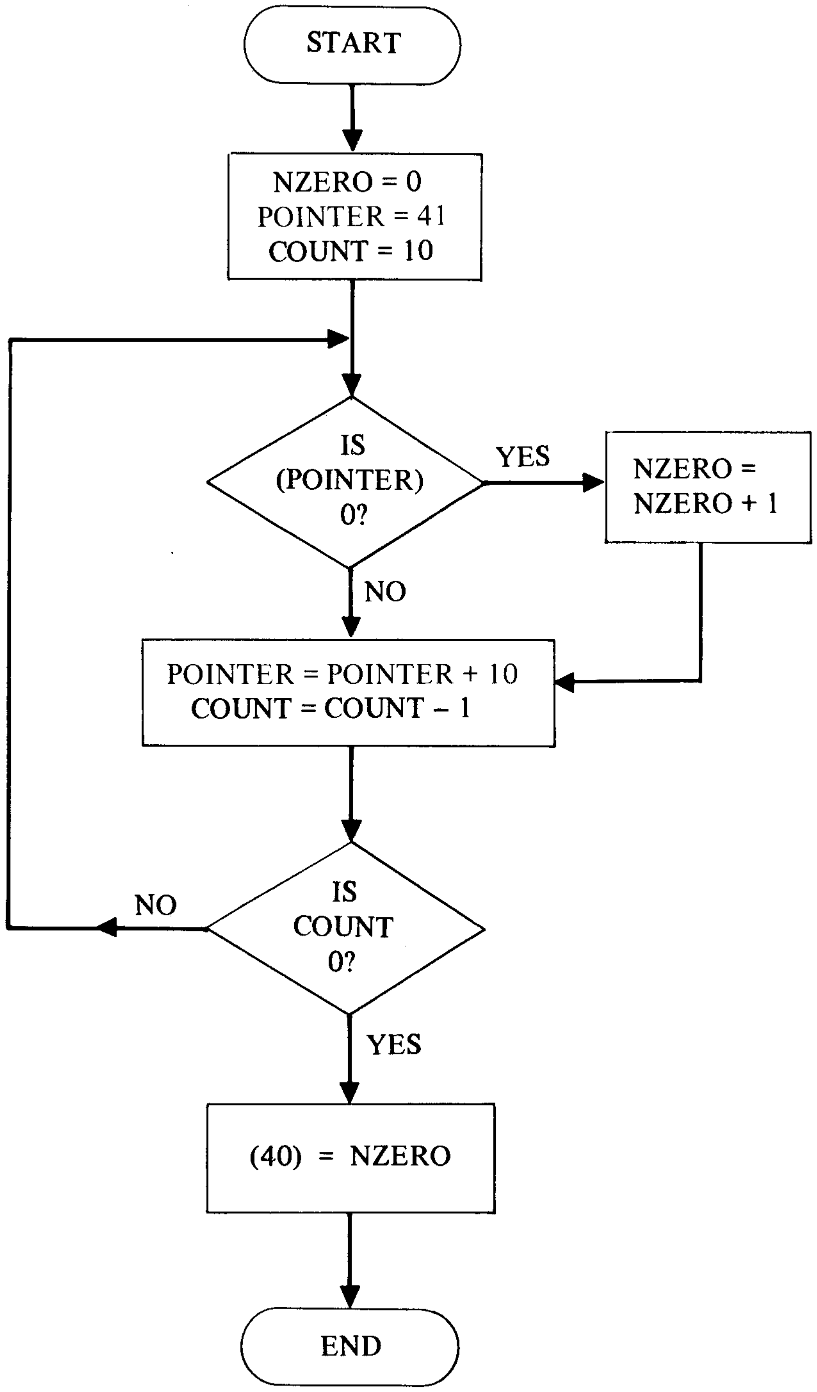
1) problem definition
2) program design
3) coding (the actual writing of instructions)
4) debugging
5) testing
6) documentation
7) extension and redesign

For the present we will concentrate on simple, well-defined problems which can be designed with flowcharts and can be debugged and tested at the same time. Flowcharting is the traditional program design method and is useful for small problems. It has a standard set of symbols and is well-understood even by those with no expertise in computer programming.

EXAMPLE 1 – COUNTING ZEROES

Purpose: Count the number of zeroes in memory locations 41 through 4A and place the result in memory location 40.

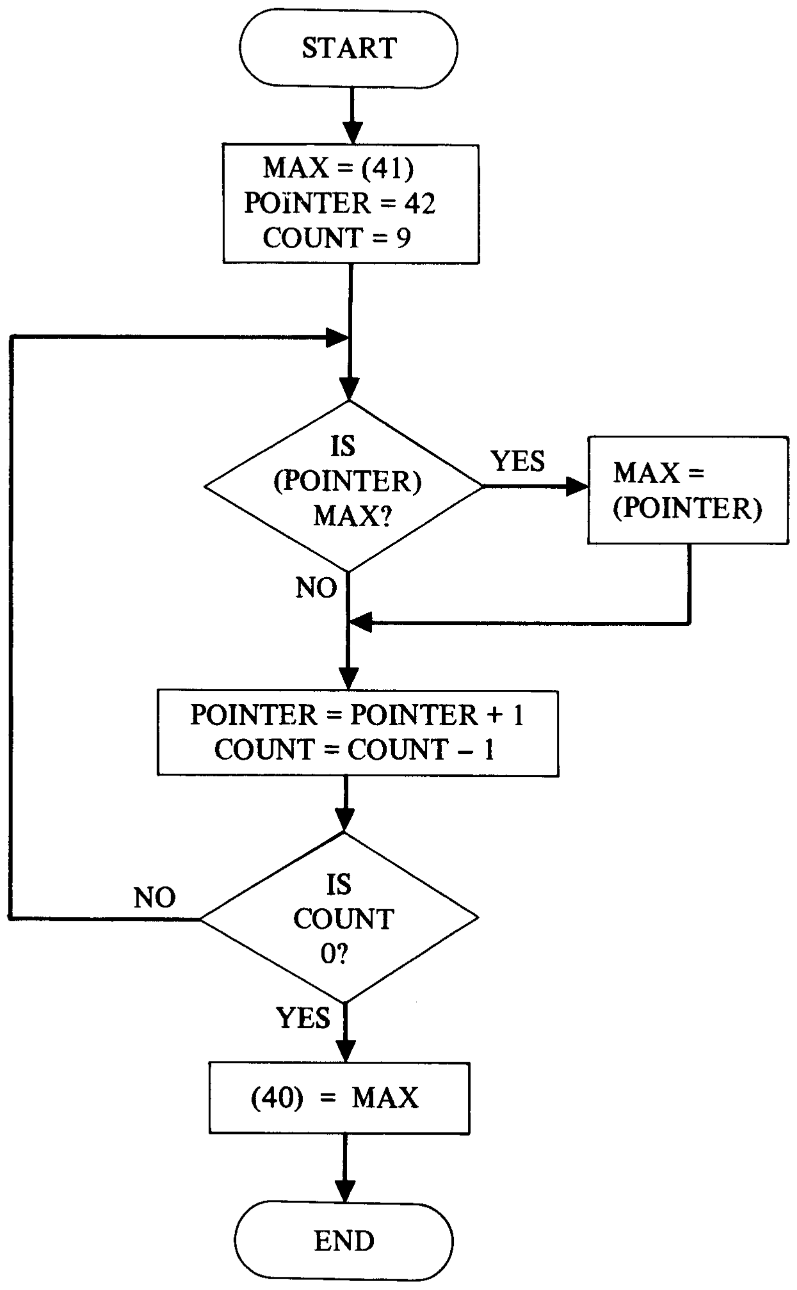
Flowchart:


START
NZERO = 0
POINTER = 41
COUNT = 10
IS (POINTER) 0?
YES
NZERO = NZERO + 1
NO
POINTER = POINTER + 10
COUNT = COUNT – 1
IS COUNT 0?
NO
YES
(40) = NZERO
END

EXAMPLE 2 – FIND MAXIMUM

Purpose: Find the largest unsigned number in memory locations 41 through 4A and place it in memory location 40.

Flowchart:



Note that you don't have to make the flowchart very elaborate or very detailed. In fact, the flowchart is most useful when it is a simple, straightforward guide to the logic of the program. A flowchart with too much detail is difficult to understand.

CODING AND DEBUGGING

The next step is to translate the flowchart into a trial program. Make sure that the program includes everything in the flowchart, but don't try to go through it in detail by hand. Use the debugging facilities in the computer instead.

You can have the program return to the monitor at any point by replacing an operation code with the instruction SOFTWARE INTERRUPT (SWI or 3F hex). You can resume the program after the SWI by pressing the "8" or RTI key. This feature is called a breakpoint.

Not only does SWI return control to the monitor, but it also saves the contents of all the registers so that you can easily examine them. To find out where these contents are, look in memory locations 0068 and 0069; these locations will contain the address just below (i.e., one less than) where the registers are saved. If that address is SAVE, the order is:

Table 7-1
Map of the Micro-68 Stack
(SAVE is address in memory locations 68 and 69)
(MSB=Most significant bit, LSB=least significant bit)

SAVE+1	Condition codes register
SAVE+2	Accumulator B
SAVE+3	Accumulator A
SAVE+4	Index Register, 8 MSB's
SAVE+5	Index Register, 8 LSB's
SAVE+6	Program Counter, 8 MSB's
SAVE+7	Program Counter, 8 LSB's

Remember that the index register, program counter, and stack pointer are 16 bits long while the accumulators are only 8 bits long. The condition codes register consists of the various flags organized as follows:

Table 7-2
Organization of Condition Code Registers

Bit 7	always 1
Bit 6	always 1
Bit 5	half-carry (from bit 3) H
Bit 4	interrupt (disable) I
Bit 3	negative (sign) N
Bit 2	zero Z
Bit 1	overflow O
Bit 0	carry C

DEBUGGING WITH THE MICRO-68: AN EXAMPLE

COUNTING ZEROES (as in earlier flowchart)

Our initial program is:

```
         CLRA             NUMBER OF ZEROES = 0
         LDAB   $10       COUNT = 10
SRZRO    LDX    #$41      POINTER = START OF ARRAY
         TST    X         CHECK AN ELEMENT
         BEQ    COUNT     IS ELEMENT ZERO?
         INCB             YES, ADD 1 TO NUMBER OF ZEROES
COUNT    INX
         DECB
         BNE    SRZRO
         STAB   $40       SAVE NUMBER OF ZEROES
         SWI
```

Note that TST just sets flags as if zero had been subtracted from the contents of the addressed register or memory location.

The hexadecimal version is:

Memory Address (Hex)	Instruction (Mnemonic)	Memory Contents (Hex)
00	CLRA	4F
01	LDAB $10	D6
02		10
03	SRZRO LDX #$41	CE
04		00
05		41
06	TST X	6D
07		00
08	BEQ COUNT	27
09		03
0A	INCB	5C
0B	COUNT INX	80
0C	DECB	5A
0D	BNE SRZRO	26
0E		F4
0F	STAB $40	D7
10		40
11	SWI	3F

Enter this program. The first debugging step will be to place a breakpoint after the initialization, i.e., put 3F in memory location 0006. The register should contain:

```
NUMBER OF ZEROES = (A) = 00
           COUNT = (B) = 0A (10 decimal)
   ARRAY POINTER = (X) = 0041
```

To check the program, enter the SWI and run the program. Now examine memory locations 0068 and 0069; they should be:

```
(0068) = 00
(0069) = 60
```

So the register storage starts in memory location 0061, i.e.,

```
(B) = (0062)
(A) = (0063)
(X) = (0064)(0065)
```

Examine those locations. Are they correct? My results were:

```
(B) = 40
(A) = 00
(X) = 0041
```

Clearly B is incorrect, so the instruction LDAB $10 is wrong. It should be LDAB #10 since we want to put the number 10 in B, not the contents of memory location 10. And, furthermore, we want the decimal number 10, not the hexadecimal number 10. The instruction should be:

```
01    LDAB    #10    C6
02                   0A
```

Make this correction and run the program again with the SWI in location 6. Are the answers right now?

To go further, we must add some data. Let's try:

```
(41) through (4A) = 00
```

Now put the breakpoint at the end of the loop, i.e.,

```
(0006) = 6D (as in original)
(000D) = 3F (SWI)
```

After you run this program, since (41) = 00, the register contents should be:

```
(B) = 09
(A) = 01
(X) = 0042
```

Make sure you understand why these are the proper results.

Run the program. What are the register contents? Mine were:

```
(B) = 0A
(A) = 0D
(X) = 0041
```

At least these last results are uniform. Everything is wrong. Clearly A and X aren't being incremented and B isn't being decremented. INCB should be INCA and INX should be 08 instead of 80. The corrections are:

```
0A          INCA        4C
0B          INX         08
```

Running this program gives the results:

```
(B) = 0A
(A) = 00
(X) = 0041
```

Everything is still wrong. Obviously, the jump instruction is wrong. We can try it by hand. What we want is:

If (A) $\neq$ 0, skip to memory location 000B instead of proceeding normally to memory location 000A.

The proper instruction is

```
08      BNE   COUNT   26
09                    01
```

since the branch should send the computer 1 location ahead if A is not equal to zero.

The results after this change are:

```
(B) = 09
(A) = 01
(X) = 0042
```

Now try another iteration. Resume the program at memory location 000D with the following changes:

```
(000D) = 26 (the original value)
(0008) = 3F (SWI
```

The results should be:

```
(B) = 09
(A) = 01
(X) = 0042
```

Instead, (X) = 0041 because the jump is to the wrong place. The program should jump back to LDAA X, i.e.,

(000E) = F7

Try this correction by inserting the breakpoint first in memory location 000D, then in 0008. Remember to put the proper code into the location where you don't have a breakpoint (i.e., 26 in 0008 or 000D).

Now try running the program without breakpoints. Are the results correct? Test the program with some other data that isn't all zeroes.

The final program is:

```
          CLRA                  NUMBER OF ZEROES = 0
          LDAB    #10           COUNT = 10
          LDX     #$41          POINTER = START OF ARRAY
SRZRO     TST     X             EXAMINE AN ELEMENT
          BNE     COUNT         IS ELEMENT ZERO?
          INCA                  YES, ADD 1 TO NUMBER OF ZEROES
COUNT     INX
          DECB
          BNE     SRZRO
          STAA    $40           SAVE NUMBER OF ZEROES
          SWI
```

In hexadecimal, the program is:

Memory Address (Hex)	Instruction (Mnemonic)			Memory Contents (Hex)
00		CLRA		4F
01		LDAB	#10	C6
02				0A
03		LDX	#$41	CE
04				00
05				41
06	SRZRO	TST	X	6D
07				00
08		BNE	COUNT	26
09				01
0A		INCA		4C
0B	COUNT	INX		08
0C		DECB		5A
0D		BNE	SRZRO	26
0E				F7
0F		STAA	$40	97
10				40
11		SWI		3F

Note that common errors to look for are:

1) Forgetting to initialize variables. Don't assume anything is zero when you start.

2) Confusing data and addresses. The contents of memory location 10 could be anything, including (but not necessarily) the number 10.

3) Branching on the wrong condition, i.e., branching on not equal instead of an equal.

4) Calculating relative offsets incorrectly. Remember that you have to start at the instruction following the branch.

5) Confusing decimal and hexadecimal numbers. Decimal 10 is 0A hex; hex 10 is 16 decimal.

6) Accidentally re-initializing a register or memory location by branching to the wrong place.

7) Incorrect keyboard entries. You should examine the entire program before you run it.

8) Forgetting to update a counter or pointer. Watch for the loop controls which must be updated regardless of which path the program follows.

9) Confusing the index register with the address in the index register. LDAA X loads accumulator A from the address in the index register. Be especially careful with instructions like CLR X TST X, etc.

Try writing and debugging a program for Example 2 which finds an unsigned maximum. Then try flowcharting, coding, and debugging the following program:

EXAMPLE 3 – KEY IN A DELAY

Purpose: The program should wait for the D key to be pressed. It should then delay for N seconds (where you can enter N from keys 0 to 7).

I.e., if you press D and then 5, it will delay for 5 seconds.

Hint: The following routine is a 1 second delay (check it):

```
     LDX    #$F423
DLY  DEX
     BNE    DLY
```

What happens if the delay is '0'?

Use the breakpoints to debug the program. Remember the common errors which we noted.

Micro-68 Displays II

We saw earlier how to turn the displays on and off and how to pulse and scan them. Now we'll discuss how to send data to the displays and how to combine display control with other tasks.

The following program places the contents of memory location 40 on all the displays for one second:

```
        LDAA   #$FF
        STAA   LIGHTS     ENABLE ALL LEDS
        LDAB   $40        GET DATA
        STAB   GLOW       SEND TO LEDS
        LDX    #$F423     WAIT 1 SECOND
DLY     DEX
        BNE    DLY
        SWI
```

The hexadecimal version of the program is:

Memory Address (Hex)	Instruction (Mnemonic)	Memory Contents (Hex)
00	LDAA #$FF	86
01		FF
02	STAA LIGHTS	B7
03		80
04		0A
05	LDAB $40	D6
06		40
07	STAB GLOW	F7
08		80
09		08
0A	LDX #$F423	CE
0B		F4
0C		23
0D	DLY DEX	09
0E	BNE DLY	26
0F		FD
10	SWI	3F

Enter and run this program. Try the following data in memory location 40: 03, 9F, 25, 0D, 99, 49, 41, 1F, 01, 09. Try some other combinations of segments.

Table 8-1 contains the seven-segment codes for the hexadecimal digits. Table 8-2 contains the codes for some letters and symbols. Clearly there is no very obvious relationship between input and output when we could use to perform the conversion.

Table 8-1
Seven-Segment Code Table
Hexadecimal Digits

Digit	Hex code
0	03
1	9F
2	25
3	0D
4	99
5	49
6	41
7	1F
8	01
9	09
A	11
b	C1
C	63
d	85
E	61
F	71

Table 8-2
Seven-Segment Code Table
Other Symbols

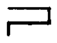
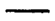
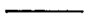
Capital letters

Letter	Hex code
A	11
C	63
E	61
F	71
H	91
J	87
L	E3
O	03
P	31
U	83
Y	89

Lower case letters

Letter	Hex code
b	C1
c	E5
d	85
h	D1
n	D5
o	C5
r	F5
u	C7

Symbols

Symbol	Hex code
	35
	FD
	EF

You can solve the conversion problem by just placing Table 8-1 in memory. Put it in memory locations 0044 through 0053. The following program will convert a hexadecimal digit in memory location 40 to a seven-segment code in memory location 41.

```
LDX     #$44
STX     $42     SAVE BASE ADDRESS OF TABLE
LDAA    $40     GET HEXADECIMAL DIGIT
ADDA    $43     USE DIGIT TO INDEX TABLE
STAA    $43
LDX     $42     GET INDEXED ADDRESS
LDAA    X       AND USE IT TO FETCH 7-SEGMENT CODE
STAA    $41
SWI
```

Note that STX $42 places the 8 most significant bits of register X in memory location 0042 and the 8 least significant bits in memory location 0043.

If, for example, (40) =0003, then

1) After STX $42

(0042) = 00
(0043) = 44

2) After STAA $43

(0043) = (0043) + (0040)
= 47

3) After LDX $42

(X) = 0047

4) After LDAA X

(A) = ((X)) = 0D

Note that the program takes advantage of the fact that the 8 most significant bits of all the table addresses are the same so 8 bit operations can be used.

The hexadecimal version of the program is:

Memory Address (Hex)	Instruction (Mnemonic)	Memory Contents (Hex)
00	LDX #$44	CE
01		00
02		44
03	STX $42	DF
04		42
05	LDAA $40	96
06		40
07	ADDA $43	9B
08		43
09	STAA $43	97
0A		43
0B	LDX $42	DE
0C		42
0D	LDAA X	A6
0E		00
0F	STAA $41	97
10		41
11	SWI	3F

Enter the program and run it for all the hexadecimal digits. Normally the table of codes will be in ROM or PROM since it never changes. Revise the program so as to eliminate the addition instruction (hint: use the instruction LDAA $E8,X).

COUNTING ON THE DISPLAYS

Now combine the display program and the table so that the program counts on the displays, i.e.,

```
                LDAA    #$FF
                STAA    LIGHTS      ENABLE ALL LEDS
        STCNT   LDX     #TABLE      GET BASE ADDRESS OF TABLES
        NXDIG   LDAA    X           GET DATA
                STAA    GLOW        SEND TO DISPLAYS
                STX     $50         WAIT 1 SECOND
                LDX     #$F423
        DLY     DEX
                BNE     DLY
                LDX     $50         UPDATE COUNT
                INX
                CMPA    #$71        WAS COUNT F?
                BNE     NXDIG       NO, KEEP COUNTING
                JMP     STCNT       YES, START OVER AT ZERO.
```

Memory Address (Hex)	Instruction (Mnemonic)			Memory Contents (Hex)
10		LDAA	#$FF	86
11				FF
12		STAA	LIGHTS	B7
13				80
14				0A
15	STCNT	LDX	#TABLE	CE
16				FF
17				E8
18	NXDIG	LDAA	X	A6
19				00
1A		STAA	GLOW	B7
1B				80
1C				08
1D		STX	$50	DF
1E				50
1F		LDX	#$F423	CE
20				F4
21				23
22	DLY	DEX		09
23		BNE	DLY	26
24				FD
25		LDX	$50	DE
26				50
27		INX		08
28		CMPA	#$71	81
29				71
2A		BNE	NXDIG	26
2B				EC
2C		BRA	STCNT	20
2D				E7

Enter and run this program. Change the counting delay and see what happens. Note that we started the program at 0010 to allow for some additions.

Change the program so that it counts down instead of up, i.e., it starts at F and counts down to zero. Remember to change the starting address (to the end of the table), the direction of the updating (DEX instead of INX), and the final comparison.

Now change the program so that it waits for the key F to be pressed then counts up continuously. Looking for key F is simple, i.e.,

```
WAITF   LDAA    $8006     IS KEY F BEING PRESSED?
        BMI     WAITF     NO, WAIT
```

or in hexadecimal

```
0B    LDAA  $8006    B6
0C                   80
0D                   06
0E    BMI   WAITF    2B
0F                   FB
```

Change the program so that it waits for key B to be pressed and then counts down continuously.

Checking for key B requires a logical AND, i.e.,

```
WAITB   LDAA   $8006        GET KEYS 8-F
        ANDA   #%00001000   IS KEY B PRESSED?
        BNE    WAITB        NO, WAIT
```

Change the program so that it counts up but checks key B after each one second delay and waits as long as key B is pressed. Remember to use accumulator B since A holds the seven-segment code. You can improve the responsiveness of the system by dividing the delay into tenths of a second. The following program is a tenth of a second delay:

```
        LDX    #$1869
DLYT    DEX
        BNE    DLYT
```

Introduce the tenth of a second response. (Use A as a counter but save its old contents!) Can you notice the difference? Clearly a processor does not have to check keys very often to provide almost instantaneous response as far as an operator is concerned.

Finally, change the program so that it checks keys B and F every second. Key B causes the program to wait, while key F causes it to resume counting. How would you modify the program so that key F acts as a stop/start button, i.e., the first closure stops the program, the second closure restarts it, etc.?

How would you design a program which started counting forward, changed to counting backward when key B was pressed, and resumed counting forward when key F was pressed?

Hints:

1) Use memory location 52 as an UP/DOWN flag, i.e., (52) = 0 if the count is forward, FF if the count is backward.

2) Use memory locations 53 and 54 as the starting address, i.e.,

(53)(54) = FFE8 if the count is forward

(53)(54) = FFF7 if the count is backward.

3) Use memory location 55 for the last code, i.e.,

(55) = 71 (code for F) if the count is forward

(55) = 03 (code for 0) if the count is backward.

The procedures are:

1) If key F is pressed and to start

```
CLR     $52       UP/DOWN FLAG = UP
LDX     #$FFE8    START AT BEGINNING OF TABLE
STX     $53
LDX     #$71      LAST CODE = F
STX     $55
```

2) If key B is pressed

```
LDAA    #$FF
STAA    $52       UP/DOWN FLAG = DOWN
LDX     #$FFF7    START AT END OF TABLE
STX     $53
LDX     #$03      LAST CODE = 0
STX     $55
```

LABORATORY 9

Arithmetic on the Micro-68

One common task for small computers is simple arithmetic. Typical applications involving arithmetic include: averaging sets of readings, forming checksums, comparing levels to thresholds, scaling inputs and outputs, linearizing non-linear inputs (such as thermocouples), and calculating frequency responses. Decimal arithmetic is necessary in calculators, business terminals, instruments, appliances, and games.

An earlier exercise showed how to form an 8 bit sum. We will now extend that exercise to saving carries, forming a 16 bit sum, and performing some simple multiplication, division, and rounding, and handling multi-word binary and decimal arithmetic.

AN 8 BIT SUM

Purpose: Add together the array of elements starting in memory location 43 and place the sum in memory location 42 (ignoring carries). The length of the array is memory location 40.

Assembly language program:

```
        CLRA             SUM = 0
        LDX     #$43     POINT TO START OF ARRAY
ADDW    ADDA    X        SUM = SUM + DATA
        NOP
        INX
        DEC     $40
        BNE     ADDW
DONE    STAA    $42      STORE SUM
        SWI
```

Hexadecimal program:

Memory Address (Hex)	Instruction (Mnemonic)	Memory Contents (Hex)
00	CLRA	4F
01	LDX #$43	CE
02		00
03		43
04	ADDW ADDA X	AB
05		00
06	NOP	01
07	INX	08
08	DEC $40	7A
09		00
0A		40
0B	BNE ADDW	26
0C		F7
0D	DONE STAA $42	97
0E		42
0F	SWI	3F

Enter this program and run it. What happens if (40) = 0? How would you correct the program to solve this problem:

```
SAMPLE DATA:  (40) = 02
              (43) = 6E
              (44) = 39

     RESULT:  (42) = A7
```

DECIMAL ARITHMETIC

The instruction DAA (decimal adjust accumulator A) corrects a binary sum in accumulator A to a decimal sum, i.e., the two instructions

```
ADDA
DAA
```

perform a decimal addition. Note that you cannot decimal adjust accumulator B.

Try the earlier 8 bit addition program with and without the DAA instruction, i.e.,

```
        CLRA             SUM = 0
        LDX     #$43     POINT TO START OF ARRAY
ADDW    ADDA    X        SUM = SUM + DATA
        (NOP)   (DAA)    DECIMAL OR BINARY SUM
        INX
        DEC     $40
        BNE     ADDW
        STAA    $42
        SWI
```

Replace the 01 in memory location 0006 with DAA (19).

Use the following set of data:

```
(40) = 03
(43) = 16
(44) = 26
(45) = 35

Result:  (42) = 77 (with DAA)
              = 71 (without DAA)
```

What is the reason for this difference?

SAVING THE CARRIES

You can easily save the carries by placing them in accumulator B. A useful instruction is:

```
ADCB   #0
```

The result is (B) = (B) + CARRY + 0
= (B) + CARRY

Use this instruction to save the carries from the 8-bit addition. Store the carries in memory location 41.

Run the revised program with the following data:

```
(40) = 04
(43) = BF
(44) = 78
(45) = E1
(46) = F1
```

Result:

```
(41) = 03
(42) = 09
```

Try a decimal version with the following data:

```
(40) = 0C
(43) = 93
(44) = 88
(45) = 98
(46) = 97
(47) = 94
(48) = 92
(49) = 90
(4A) = 97
(4B) = 93
(4C) = 96
(4D) = 95
(4E) = 97
```

Result:

```
(41) = 11
(42) = 30
```

Remember to decimal adjust both the sum and the carries. But remember that DAA only works on accumulator A. You can use the following instruction sequence to produce a 16-bit decimal sum:

```
ADDW    ADDA    X       SUM = SUM + DATA
        DAA             DECIMAL SUM
        STAA    $52
        TBA
        ADCA    #0      ADD IN CARRIES
        DAA             AND MAKE THAT DECIMAL
        TAB
        LDAA    $52
```

The various codes are:

TBA		17
ADCA	#	89
TAB		16

PERFORMING 16 BIT ARITHMETIC

You can use the two accumulators together to perform 16-bit arithmetic. The following program will add a set of 16-bit numbers stored starting in memory location 43 (least significant bits first). The length of the set is in memory location 40.

```
        CLRA            SUM = 0
        CLRB
        LDX     #$43    POINT TO START OF ARRAY
ADD16   ADDA    X       ADD IN 8 LSB's OF ENTRY
        INX
        ADCB    X       ADD IN 8 MSB's OF ENTRY
        INX
        DEC     $40
        BNE     ADD16
        STAB    $41     STORE MSB's OF SUM
        STAA    $42     STORE LSB's OF SUM
        SWI
```

Hexadecimal version:

Memory Address (Hex)	Instruction (Mnemonic)	Memory Contents (Hex)
00	CLRA	4F
01	CLRB	5F
02	LDX #$43	CE
03		00
04		43
05	ADD16 ADDA X	AB
06		00
07	INX	08
08	ADCB X	E9
09		00
0A	INX	08
0B	DEC $40	7A
0C		00
0D		40
0E	BNE ADD16	26
0F		F5
10	STAA $41	97
11		41
12	STAB $42	D7
13		42
14	SWI	3F

Enter this program and try it on the following data:

```
         (40) = 03

         (43) = F8
         (44) = 37

         (45) = 19
         (46) = 26

         (47) = EC
         (48) = 0B

       Result =  37F8
                +2619
                +0BEC
                 ----
                 69FD

         (41) = FD
         (42) = 69
```

Change the program so that it performs a 16 bit logical sum (EXCLUSIVE OR). How about a 16 bit logical AND? Revise the program so that it uses a final "zero" element rather than a specific count. Try the last program on the following sets of data:

```
a)  (43) = 00
    (44) = 00

    Result:  (41) = 00
             (42) = 00

b)  (43) = 67
    (44) = 02
    (45) = 80
    (46) = 80
    (47) = 00
    (48) = 00

    Result:  (41) = E7
             (42) = 82
```

Remember that X + Y = 0 does not mean either X or Y is zero (why?). How can you determine if a 16-bit number is zero?

ROUNDING BINARY NUMBERS

Rounding binary numbers to a specified bit length is simple since the only bit values are '0' and '1'. All you have to do is look at the most significant bit that you plan to drop:

(1) If MSB = 1, round up by adding 1.

(2) If MSB = 0, round down by truncating.

The following program will round a 16 bit number in memory locations 41 and 42 (MSB's in 41) to an 8 bit number in memory location 40.

```
        LDAA   $41    GET MSB's
        LDAB   $42    GET LSB's
        BPL    STRR   DO LSB'S REQUIRE ROUNDING?
        INCA          YES, ADD 1 TO MSB'S
STRR    STAA   $40
        SWI
```

Memory Address (Hex)	Instruction (Mnemonic)	Memory Contents (Hex)
00	LDAA $41	96
01		41
02	LDAB $42	D6
03		42
04	BPL STRR	2A
05		01
06	INCA	4C
07	STRR STAA $40	97
08		40
09	SWI	3F

Try this program on these sets of data:

(a) (41) = 26
(42) = 88

Result = (40) = 27

(b) (41) = 43
(42) = 7F

Result = (40) = 43

You can scale a number down by dividing it by 2; i.e., you can truncate an 8 bit number to seven bits with the single instruction LSR. You can shift both accumulators right by using the logical shift on the MSB's and the rotate instruction on the LSB's. A left shift uses the opposite sequence.

16 – BIT RIGHT SHIFT A AND B
(MSB's IN A)

```
LSRA
RORB
```

16 – BIT LEFT SHIFT A AND B
(MSB's IN A)

```
ASLB
ROLA
```

Similar sequences will work on two consecutive memory locations, i.e.;

```
LSR    $43
ROL    $42
```

So the following program will scale the contents of memory locations 42 and 43 by a factor of 2, round the result, and save it in memory locations 40 and 41 (MSB's in 40 and 42).

```
LDAA    $42         GET MSB's
LDAB    $43         AND LSB's
LSRA                SCALE DOWN BY 2
RORB
ADCB    #0          ROUND LSB's
ADCA    #0          ROUND MSB's
STAA    $40         STORE MSB's
STAB    $41         STORE LSB's
SWI
```

Memory Address (Hex)	Instruction (Mnemonic)	Memory Contents (Hex)
00	LDAA $42	96
01		42
02	LDAB $43	D6
03		43
04	LSRA	44
05	RORB	56
06	ADCB #0	C9
07		00
08	ADCA #0	89
09		00
0A	STAA $40	97
0B		40
0C	STAB $41	D7
0D		41
0E	SWI	3F

Try this program on the following data:

a) (42) = 57
   (43) = 83

   Result = (40) = 2B
            (41) = C2

b) (42) = 16
   (43) = 80

   Result = (40) = 0B
            (41) = 40

Change the program so that it scales the original number by a factor of 4 and rounds. The results should be:

```
a)  (42) = 57
    (43) = 83

    Result = (40) = 15
             (41) = E1

b)  (42) = 16
    (43) = 80

    Result = (40) = 05
             (41) = A0
```

Now change the program so that the number of bits to be dropped (>0) is in memory location 44; i.e.,

```
a)  (42) = 57
    (43) = 83
    (44) = 01

    Result = (40) = 2B
             (41) = C2

b)  (42) = 57
    (43) = 83
    (44) = 03

    Result = (40) = 0A
             (41) = F0
```

MULTI-WORD ARITHMETIC

Operations involving more than 16 bits require a slightly different approach than 8 or 16-bit arithmetic because there are only two accumulators. The procedure is simply to perform the correct number of 8-bit operations; e.g.,

40 BIT ADDITION (LSB's FIRST)

```
        LDAB   #5     40 BITS = 5 WORDS
        LDX    #$40   POINT TO START OF NUMBER
        CLC           CLEAR CARRY TO START
ADD8    LDAA   X      GET 8 BITS OF 1ST NUMBER
        ADCA   5, X   ADD 8 BITS OF 2ND NUMBER
        STAA   10, X  STORE 8 BITS OF RESULT
        INX
        DECB
        BNE    ADD8
        SWI
```

Memory Address (Hex)	Instruction (Mnemonic)			Memory Contents (Hex)
00		LDAB	#5	C6
01				05
02		LDX	#$40	CE
03				00
04				40
05		CLC		0C
06	ADD8	LDAA	X	A6
07				00
08		ADCA	5, X	A9
09				05
0A		STAA	10, X	A7
0B				0A
0C		INX		08
0D		DECB		5A
0E		BNE	ADD8	26
0F				F6
10		SWI		3F

Note that you must clear the CARRY initially since there is never a carry into the least significant bits.

Modify this program to perform the following tasks:

a) 48 bit addition

b) 48 bit addition with the numbers stored with most significant bits first.

c) 12 digit decimal addition (use DAA)

d) 12 digit decimal subtraction

Hint: X - Y = X + (99 - Y) + 1 - 100

so the procedure is to set the CARRY originally and let it be 1 whenever no borrow is needed; i.e.,

```
        SEC
DECSB   LDAA    #$99    GET 99
        ADCA    #0      100 – BORROW
        SUBA    X       100 – BORROW – Y
        ADDA    6, X    100 – BORROW – Y + X
        DAA             DECIMAL SUBTRACTION
        STAA    12, X   STORE RESULT
```

Subroutines and the Micro-68 Monitor

Clearly some instruction sequences will be used over and over again, e.g., the delay routine, keyboard input, display output, etc. You will probably find it convenient to write these routines once, place them in memory, and simply refer to them as needed. Such a standard routine is called a subroutine.

You may call the subroutine with the instructions BSR or JSR; these instructions perform an unconditional jump to the start of the subroutine and save the old value of the program counter in memory. An RTS instruction at the end of the subroutine restores the old program counter and returns control to the main program.

A DELAY SUBROUTINE

For example, the one second delay routine could serve as a subroutine. The following program waits for key C to be pressed and then delays for one second. KEY is location $8006.

```
WAITC   LDAA   KEY          GET KEYS 8-F
        ANDA   #%00010000   IS KEY C BEING PRESSED?
        BNE    WAITC        NO, WAIT UNTIL IT IS
        JSR    DELAY        WAIT 1 SECOND
        SWI

DELAY   LDX    #F423        WAIT 1 SECOND
DLY     DEX
        BNE    DLY
        RTS
```

The hexadecimal versions of the program and the subroutine are:

Memory Address (Hex)	Instruction (Mnemonic)			Memory Contents (Hex)
00	WAITC	LDAA	KEY	B6
01				80
02				06
03		ANDA	#%00010000	84
04				10
05		BNE	WAITC	26
06				F9
07		JSR	DELAY	BD
08				00
.09				30
0A		SWI		3F

Memory Address (Hex)	Instruction (Mnemonic)			Memory Contents (Hex)
30	DELAY	LDX	#$F423	CE
31				F4
32				23
33	DLY	DEX		09
34		BNE	DLY	26
35				FD
36		RTS		39

Enter this program and run it.

What are the final contents of memory locations 68 and 69? Remember that these locations contain the address one below where the SWI instruction has stored the contents of all the registers. What are the contents of the registers?

Now replace DEX in memory location 33 with SWI and run the program. What are the contents of memory locations 68 and 69? How about 66 and 67 and the various registers?

The reason for the change is that JSR (like SWI) uses the stack pointer to store data in memory. The procedure for each byte is:

((Stack Pointer)) = Data
(Stack Pointer) = (Stack Pointer) - 1

Each byte goes into the address contained in the stack pointer and that address is decremented so the next byte will go into the next lower address. The stack grows downward (instead of upward as you might expect) so that you can start it at the end of a block of memory and it won't interfere with other data.

Since the computer decrements the stack pointer with each use, subroutines can call other subroutines and so on as long as there is room in the stack. For example, change the program counter in memory locations 0064 and 65 to 0007 and execute RTI. What are the contents of memory locations 68 and 69 at the end of the program? Explain what happened.

The instructions RTS and RTI use the stack pointer to retrieve data and addresses from memory. The procedure for each byte is:

(Stack Pointer) = (Stack Pointer) + 1
Data = ((Stack Pointer))

The address in the stack pointer is incremented before a byte of data is retrieved. The next byte will be obtained from the next higher address. So RTS and RTI reverse the actions of JSR and SWI. Note that the program counter is 2 bytes long and the least significant byte is stored first and retrieved last.

A KEYBOARD SUBROUTINE

The following subroutine waits for one of keys 0 to 7 to be pressed and returns with the key identification in accumulator B.

```
WAITC   LDAA   STATUS   GET KEYS 0-7
        CMPA   #$FF     ARE ANY PRESSED?
        BEQ    WAITC    NO, WAIT
        CLRB            YES, KEY NUMBER = 0
SRKEY   LSRA            IS NEXT BIT 0?
        BCC    DONE     YES, DONE
        INCB            NO, ADD 1 TO KEY NUMBER
        BRA    SRKEY
DONE    RTS
```

Memory Address (Hex)	Instruction (Mnemonic)			Memory Contents (Hex)
37	WAITC	LDAA	STATUS	B6
38				80
39				04
3A		CMPA	#$FF	81
3B				FF
3C		BEQ	WAITC	27
3D				F9
3E		CLRB		5F
3F	SRKEY	LSRA		44
40		BCC	DONE	24
41				03
42		INCB		5C
43		BRA	SRKEY	20
44				FA
45	DONE	RTS		39

Revise the earlier program so that it waits for key C, then waits until one of keys 0 to 7 is pressed, and then delays for the correct number of seconds. Place an SWI in memory location 3A and see what happens to the stack. Be sure that the program works properly for key 0.

A DISPLAY SUBROUTINE

The following program converts the contents of accumulator B to a seven-segment code and sends the result to the LEDs. The original contents of accumulator B are unchanged.

```
DSPB    LDAA   #$FF     GET BASE PAGE OF 7 SEGMENT
                          CODE TABLE
        STAA   $2E
        STAB   $2F      INDEX TABLE WITH DATA
        LDX    $2E      GET CODE ADDRESS
        LDAA   $E8, X   GET CODE
        STAA   STATUS
        RTS
```

Remember that the table of seven-segment codes starts in memory location FFE8. Note that the subroutine uses registers A and X and memory locations 2E and 2F.

Memory Address (Hex)	Instruction (Mnemonic)			Memory Contents (Hex)
46	DSRB	LDAA	#$FF	86
47				FF
48		STAA	$2E	97
49				2E
4A		STAB	$2F	D7
4B				2F
4C		LDX	$2E	DE
4D				2E
4E		LDAA	$E8, X	A6
4F				E8
50		STAA	GLOW	B7
51				80
52				08
53		RTS		39

Revise the earlier program so that it displays the number of seconds remaining. How would you revise the program so that key F turns the displays on and off; i.e., pressing key F once turns the displays off, pressing it again turns the displays on, etc.

This feature is often convenient when the displays may be distracting to an operator as in an airplane cockpit.

USING THE MONITOR SUBROUTINES

You can also use JSR to reach the subroutines in the Micro-68 monitor. The subroutines are:

DISPLAY (Address FFCB) – displays six characters from memory locations 0078 through 007D in a single left-to-right scan of the LEDs. Uses A, B, and X registers. Each display is pulsed for about 3 ms.

WRDATA (Address FFB6) – converts a 4 bit binary number in A into a seven-segment display code stored at (0078 + (0077)), increments 0077, and saves the original binary number in 007F. Uses A and X. Note that 0077 tells the computer which address to use between 0078 and 007D.

INPUT (Address FF93) – scans the keyboard and refreshes the LED displays. Displays key input unless (74) ≠ 0. Uses A, B, X. If (74) ≠ 0, the LEDs remain as they were. The routine clears memory location 74.

BYTEOUT (Address FF47) – converts an 8 bit number in A into two hexadecimal characters. Uses WRDATA twice to convert the characters into seven-segment codes. Returns with original 8 bit number in B. Uses A, B, X.

ADROUT (Address FED4) – converts a 16 bit number in memory locations 0072 (MSB's) and 0073 (LSB's) to four seven-segment codes in memory locations 0078-007B. Uses A, B, X.

BYTEIN (Address FF54) – builds an 8 bit binary number in A from two keyboard entries (first one is 4 MSB's). Uses A, B, X.

Remember that DISPLAY only scans the LEDs once. The following program will hold the contents of memory locations 0040 through 0045 on the LEDs.

Note that we have to move our data to the display locations since the monitor uses the display locations also.

```
DSP       LDX    $40       POINT TO START OF DATA ARRAY
          LDAB   #6        COUNT = 6 WORDS
MVDSP     LDAA   X         GET A WORD OF DATA
          STAA   $38, X    MOVE IT TO DISPLAY LOCATION
          INX
          DECB
          BNE    MVDSP
SCAND     JSR    DISPLAY   SCAN DISPLAYS
          BRA    SCAND
```

Memory Address (Hex)	Instruction (Mnemonic)			Memory Contents (Hex)
00	DSP	LDX	#$40	CE
01				00
02				40
03		LDAB	#6	C6
04				06
05	MVDSP	LDAA	X	A6
06				00
07		STAA	$38, X	A7
08				38
09		INX		08
0A		DECB		5A
0B		BNE	MVDSP	26
0C				F8
0D	SCAND	JSR	DISPLAY	BD
0E				FF
0F				CB
10		BRA	SCAND	20
11				FB

Try the following patterns:

1) (40) = 91
   (41) = 61
   (42) = E3
   (43) = E3
   (44) = 03
   (45) = FF

2) (40) = FF
   (41) = 71
   (42) = E3
   (43) = 03
   (44) = 31
   (45) = FF

Make up some patterns of your own and display them.

Change the program so that it only scans the display a certain number of times. How many times do you have to scan the display in order to make it visible? Remember that DISPLAY uses the registers so you'll have to keep your counter in memory.

You can build the data from the keys using INPUT; i.e.,

```
        CLR     $77        DISPLAY OFFSET = 0
        LDAA    #6         NUMBER OF DISPLAYS = 6
        STAA    $50
BUILDE  JSR     INPUT      GET A KEYBOARD INPUT
        DEC     $50
        BNE     BUILDE
DSP     JSR     DISPLAY
        BRA     DSP
```

INPUT uses WRDATA to convert the keyboard entry into a seven-segment code. Note that we use a counter in memory since INPUT uses the registers. You must be very careful about exactly what changes a subroutine produces; a single subroutine call can have many different effects on registers and memory locations.

The hexadecimal version of the program is:

Memory Address (Hex)	Instruction (Mnemonic)	Memory Contents (Hex)
00	CLR $77	7F
01		00
02		77
03	LDAA #6	86
04		06
05	STAA $50	97
06		50
07	BUILDE JSR INPUT	BD
08		FF
09		93
0A	DEC $50	7A
0B		00
0C		50
0D	BNE BUILDE	26
0E		F8
0F	DSP JSR DISPLAY	BD
10		FF
11		CB
12	BRA DSP	20
13		FB

Enter and run this program. What are the final contents of memory location 0077? Why? How could you change the program so that it only accepts four keys? Try the program. Which displays does it use? How could you change the program so that it uses the rightmost four displays? Hint: blank the first two displays by placing FF in locations 78 and 79 and 2 in location 77.

BUILDING MONITOR COMMANDS

The basic monitor routines are, of course, used to build an address from 4 key entries and display the contents of that address:

```
            CLR     $77
            JSR     BYTEIN      GET 1ST TWO DIGITS
            STAA    $72
            JSR     BYTEIN      GET 2ND TWO DIGITS
            STAA    $73
            LDX     $72         MAKE DIGITS INTO ADDRESS
            LDAA    X           GET DATA FROM ADDRESS
            JSR     BYTEOUT     DATA TO DISPLAY BUFFER
DSP         JSR     DISPLAY
            BRA     DSP
```

Memory Address (Hex)		Instruction (Mnemonic)		Memory Contents (Hex)
00		CLR	$77	7F
01				00
02				77
03		JSR	BYTEIN	BD
04				FF
05				54
06		STAA	$72	97
07				72
08		JSR	BYTEIN	BD
09				FF
0A				54
0B		STAA	$73	97
0C				73
0D		LDX	$72	DE
0E				72
0F		LDAA	X	A6
10				00
11		JSR	BYTEOUT	BD
12				FF
13				47
14	DSP	JSR	DISPLAY	BD
15				FF
16				CB
17		BRA	DSP	20
18				FB

Enter this program and run it. You can also end it by simply returning to the BYTEIN routine. In fact, this routine automatically refreshes the displays while waiting for the next key to be pressed.

Using the Peripheral Interface Adapter

The most complex and expensive part of modern computers is the input/output section. Each peripheral represents a unique interfacing problem which requires a special circuit board. However, LSI techniques can simplify interfacing by providing devices which can be programmed to handle many different situations. The Motorola 6820 Peripheral Interface Adapter (PIA) is such a programmable device.

The PIA has two I/O ports (A and B) each of which contains the following:

an 8-bit data register which is latched when used for output but unlatched when used for input

an 8-bit data direction register which determines whether individual data pins are inputs ('0') or outputs ('1')

an 8-bit control register which configures the port

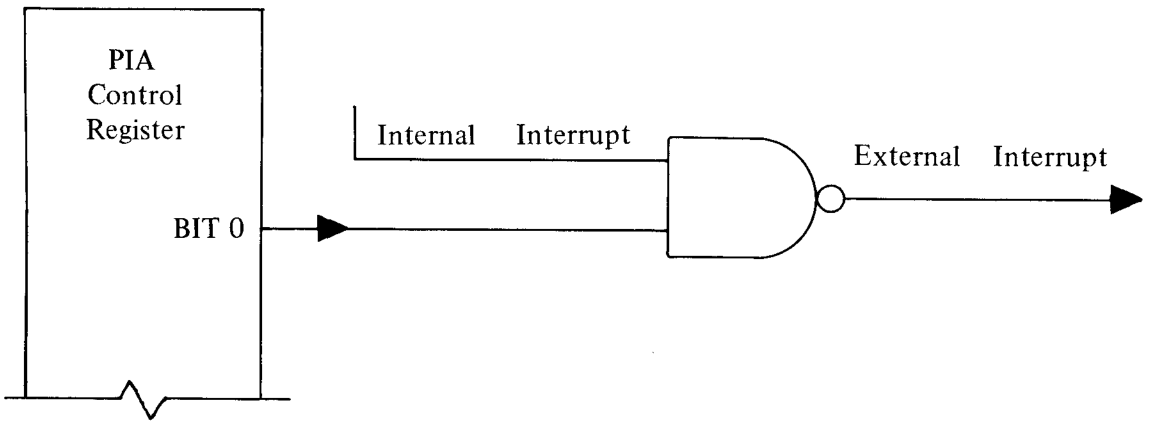
a serial input line CA(B) 1

a serial input or output line CA(B)2

an active-low interrupt output line IRQA(B)

The control register is the key to choosing the configuration for the PIA. The bits which the programmer places in that register determine how the PIA operates. For example, bit 0 of the control register determines whether the internal interrupt appears externally as shown in Figure B-1. If bit 0 is '0' the output from the NAND gate is always '1' regardless of the other input. If bit 0 is '1', the output from the NAND gate is the inverse of the other input. The programmer can thus decide whether the PIA will operate in an interrupt or non-interrupt mode.

Figure B-1
Controlling the Interrupt Output with Control Register Bit 0

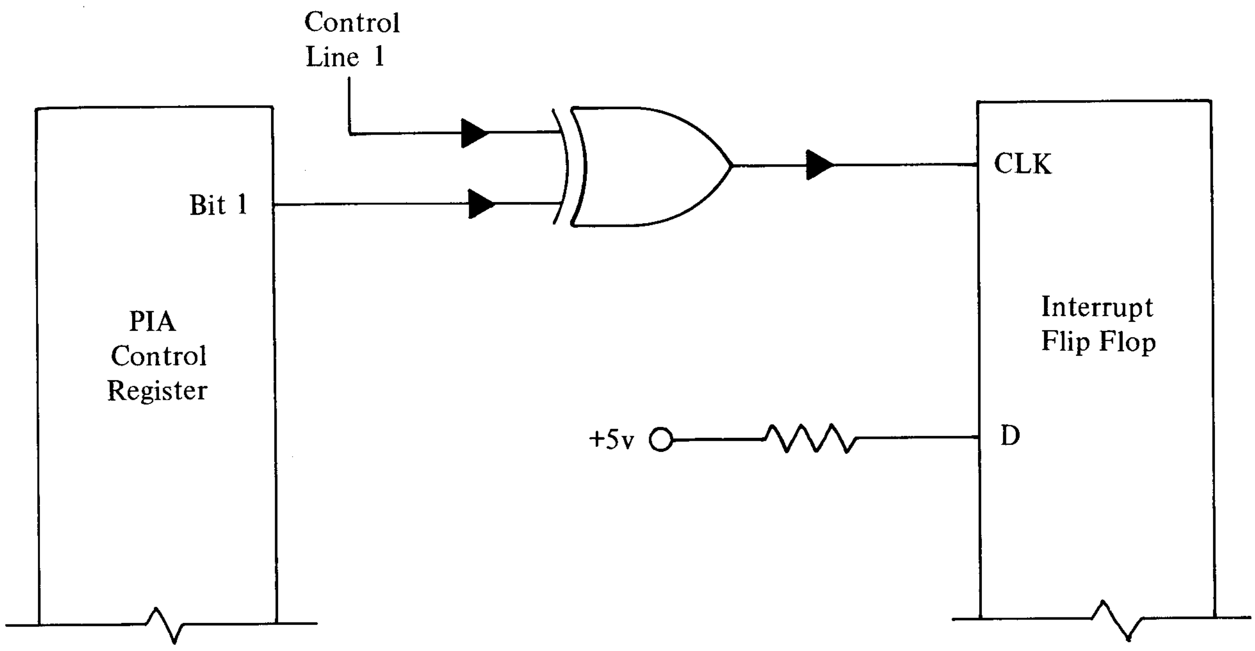


The programmer can either permit or inhibit the external interrupt by placing a '0' or '1' in bit 0 of the control register. The external interrupt is active-low.

Bit 1 of the control register determines whether the internal interrupt flip-flop will be set on a high-to-low ('0') or low-to-high ('1') transition on control line 1. Figure B-2 shows how this circuitry could be designed. The D flip-flop changes state (to '1' – the D input) on a high-to-low clock transition. If control register bit 1 is '1', control line 1 is inverted so the actual transition is low-to-high. The PIA can therefore recognize either transition on the control line without the need for an external inverter. Sometimes a high-to-low transition is called a 'trailing edge' and a low-to-high transition is called a 'leading edge'.

Bit 2 of the control register determines whether the CPU is accessing the data register (1) or the data direction register (0). The two registers share a system address. Figure B-3 shows how this circuitry could be designed. If bit 2 = '0', the data direction register is clocked; if bit 2 = '1', the data register is clocked.

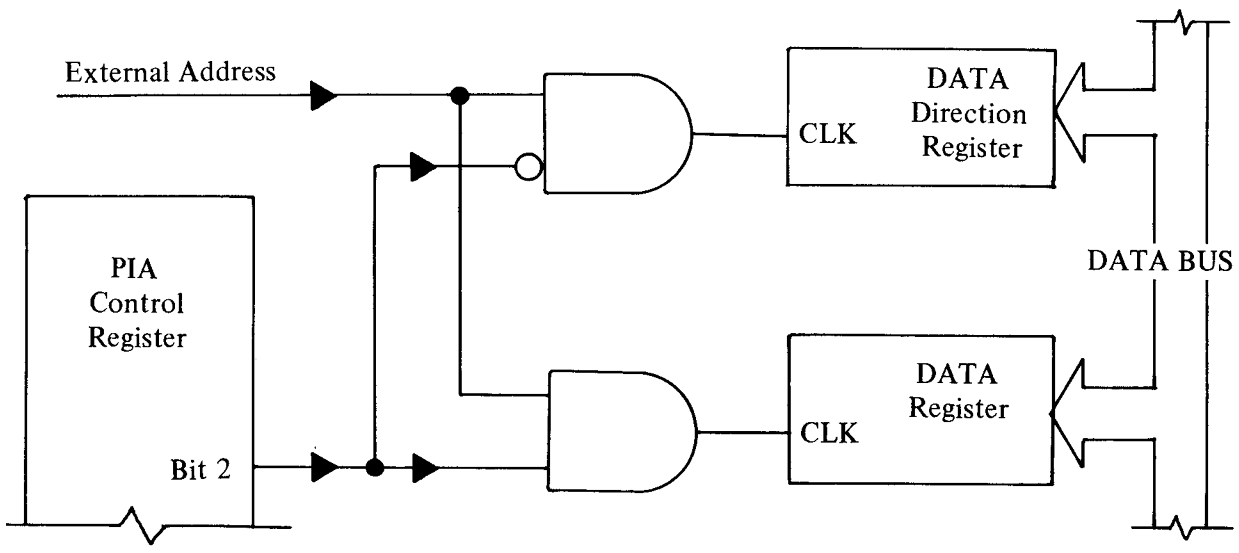
Figure B-2
Controlling the Interrupt Flip-Flop with
Control Register Bit 1



If control register bit 1 is '0', the output from the EXCLUSIVE OR gate is the same as control line 1. If control register bit 1 is '1', the output is the inverse of control line 1. Remember the truth table for EXCLUSIVE OR, i.e.,

A	B	A + B
0	0	0
0	1	1
1	0	1
1	1	0

Figure B-3
Controlling the Internal Addresses with
Control Register Bit 2



If bit 2 = 1, the data register is clocked while if bit 2 = 0, the data direction register is clocked.

These examples are typical of how the programmer can determine the logic structure of the PIA by placing data in the control registers. The bit assignments (these are arbitrary) are as follows:

Bit 7 – internal interrupt latch for control line 1.

Bit 6 – internal interrupt latch for control line 2.

Bit 5 – determines whether control line 2 is an input ('0') or output ('1').

Bits 3 and 4 – determine the use of control line 2.

Bit 2 – determines whether the CPU will access data direction register ('0') or data register ('1').

Bit 1 – determines whether bit 7 is set by high-to-low ('0') or low-to-high ('1') transitions on control line 1.

Bit 0 – determines whether the external interrupt from bit 7 is disabled ('0') or enabled ('1').

Note the following features of the PIA:

1. Reset clears all the registers, making the data and control lines inputs, selecting the data direction registers, and disabling the interrupts.
2. The CPU can't write into the interrupt latches, i.e., control register bits 6 and 7.
3. Reading the data register clears bits 6 and 7 so that they can act as DATA READY signals. Note that writing into the data register does not clear those bits.

AN INPUT PIA

The EPA keyboard PIA is available for external use at the 40 pin connector. Its addresses are:

DATA OR DATA DIRECTION REGISTER A	8004	DATAA
CONTROL REGISTER A	8005	STATA
DATA OR DATA DIRECTION REGISTER B	8006	DATAB
CONTROL REGISTER B	8007	STATB

The pin connections are:

Table B-1
Connections for 40 Pin Connector

PA0 – PA7	2 – 9
PB0 – PB7	10 – 17
CA1	A
CA2	1
CB1	18
CB2	V

The following program will make the "A" side into an input port:

```
CLR     STATA
CLR     DATAA          MAKE ALL LINES INPUTS
LDAA    #%00000100     ACCESS DATA REGISTER
STAA    STATA
```

So that you can see the contents of the A side data and data direction registers at the same time, the following program places them in registers A and B respectively,

```
CLR     STATA
CLR     DATAA          MAKE ALL LINES INPUTS
LDAB    DATAA          SAVE DATA DIRECTION REGISTER IN B
LDAA    #%00000100     ACCESS DATA REGISTER
STAA    STATA
LDAA    DATAA          SAVE DATA REGISTER IN A
SWI
```

The hexadecimal version of the program is:

Memory Address (Hex)	Instruction (Mnemonic)	Memory Contents (Hex)
00	CLR STATA	7F
01		80
02		05
03	CLR DATAA	7F
04		80
05		04
06	LDAB DATAA	F6
07		80
08		04
09	LDAA #%00000100	86
0A		04
0B	STAA STATA	B7
0C		80
0D		05
0E	LDAA DATAA	B6
0F		80
10		04
11	SWI	3F

Enter and run the program. What are the values of the data and data direction registers at the end? Remember that we obtained both values from the same address. Try the same program but use the PIA "B" side (addresses 8006 and 8007).

Attach one end of an SPST (single-pole, single-throw) switch to data bit PA7 and the other end to ground. The following program will make the PIA into an input port and wait for you to close the switch:

```
        CLR   STATA
        CLR   DATAA        ALL LINES INPUTS
        LDAA  #%00000100   ACCESS DATA REGISTER
        STAA  STATA
WAITS   LDAA  DATAA        GET SWITCH STATUS
        BMI   WAITS        WAIT IF OPEN
        SWI
```

Memory Address (Hex)	Instruction (Mnemonic)	Memory Contents (Hex)
00	CLR STATA	7F
01		80
02		05
03	CLR DATAA	7F
04		80
05		04

Memory Address (Hex)	Instruction (Mnemonic)			Memory Contents (Hex)
06		LDAA	#%00000100	86
07				04
08		STAA	STATA	B7
09				80
0A				05
0B	WAITS	LDAA	DATAA	B6
0C				80
0D				04
0E		BMI	WAITS	2B
0F				FB
10		SWI		3F

Note that clearing the control register to start allows us to place values in the data direction register. Enter this program and run it. Try attaching the switch to some of the other data bits on side A.

Attach the switch to control bit CA1. The following program will configure the PIA, wait for you to close the switch, and place the contents of the control register before and after the data register has been read in accumulators A and B respectively.

```
        CLR    STATA
        CLR    STATA        ALL LINES INPUTS
        LDAA   #%00000100   ACCESS DATA REGISTER
        STAA   STATA
WAITS   LDAA   STATA        HAS SWITCH BEEN CLOSED?
        BPL    WAITS        NO, WAIT
        LDAB   DATAA        CLEAR STATUS BIT
        LDAB   STATA        GET NEW CONTROL REGISTER
                              CONTENTS
        SWI
```

Memory Address (Hex)	Instruction (Mnemonic)		Memory Contents (Hex)
00	CLR	STATA	7F
01			80
02			05
03	CLR	DATAA	7F
04			80
05			04
06	LDAA	#%00000100	86
07			04
08	STAA	STATA	B7
09			80
0A			05

Memory Address (Hex)	Instruction (Mnemonic)			Memory Contents (Hex)
0B	WAITS	LDAA	STATA	B6
0C				80
0D				05
0E		BPL	WAITS	2A
0F				FB
10		LDAB	DATAA	F6
11				80
12				04
13		LDAB	STATA	F6
14				80
15				05
16		SWI		3F

Run the program. What are the final contents of accumulators A and B? Change the program so that it debounces the switch by waiting for 1 ms., i.e.,

Memory Address (Hex)	Instruction (Mnemonic)			Memory Contents (Hex)
10		LDX	#$3E	CE
11				00
12				3E
13	DLY	DEX		09
14		BNE	DLY	26
15				FD
16		LDAB	DATAA	F6
17				80
18				04
19		LDAB	STATA	F6
1A				80
1B				05
1C		SWI		3F

Now what are the final contents of accumulators A and B? Explain the difference. What happens if you replace LDAB DATAA with STAB DATAA? How about ADDB DATAA, TST DATAA, CLR DATAA, or ROR DATAA? Why is this feature useful and what happens if the program doesn't read the data register? How would you change the program to use line CA2 rather than CA1?

AN OUTPUT PIA

The following program makes the PIA B side into an output port and stores the data and direction registers in registers A and B respectively,

```
CLR     STATB
LDAA    #$FF
STAA    DATAB         ALL LINES OUTPUTS
LDAB    DATAB         SAVE DATA DIRECTION REG. IN B
LDAA    #%00000100    ACCESS DATA REGISTER
STAA    STATB
LDAA    DATAB         SAVE DATA REGISTER IN A
SWI
```

Memory Address (Hex)	Instruction (Mnemonic)	Memory Contents (Hex)
00	CLR STATB	7F
01		80
02		07
03	LDAA #$FF	86
04		FF
05	STAA DATAB	B7
06		80
07		06
08	LDAB DATAB	F6
09		80
0A		06
0B	LDAA #%00000100	86
0C		04
0D	STAA STATB	B7
0E		80
0F		07
10	LDAA DATAB	B6
11		80
12		06
13	SWI	3F

Enter and run the program. What are the values of the data and direction registers at the end? Change LDAA #%00000100 to LDAA #%11000100 and change LDAA DATAB to LDAA STATB. What is the final value of accumulator A? Explain this.

Attach the cathode of an LED to data line PB7 and its anode to +5 volts. The following program will configure the PIA and light the LED.

```
        CLR     STATB
        LDAA    #$FF          ALL LINES OUTPUTS
        STAA    DATAB
        LDAA    #%00000100    ACCESS DATA REGISTER
        STAA    STATB
        CLR     DATAB         LIGHT THE LED
HERE    BRA     HERE          AND WAIT
```

Memory Address (Hex)	Instruction (Mnemonic)			Memory Contents (Hex)
00		CLR	STATB	7F
01				80
02				07
03		LDAA	#$FF	86
04				FF
05		STAA	DATAB	B7
06				80
07				06
08		LDAA	#%00000100	86
09				04
0A		STAA	STATB	B7
0B				80
0C				07
0D		CLR	DATAB	7F
0E				80
0F				06
10	HERE	BRA	HERE	20
11				FE

Run the program. Change it so that it turns the LED on for one second. How would you change the program so that it only affects one bit in the PIA data register?

THE BIDIRECTIONAL CONTROL LINE

Control line 1 is always an input line. Transitions on that line generally indicate the presence of new data from an input device or the readiness of an output device to accept data. Control line 2 may be either an input or an output line. As an output line, it may indicate the presence of new output data, mark the completion of a transfer (or the readiness of the CPU to accept data), or serve as a latched serial output.

Attach a switch to CB1 and an LED to CB2. The following program will use CB2 as a serial output to turn the LED on for one second.

```
        CLR     STATB
        CLR     DATAB         DATA LINES INPUTS
        LDAA    #%00110100    ACCESS DATA REGISTER AND
                                TURN LED ON
        STAA    STATB
        LDX     #$F423        DELAY 1 SECOND
DLY     DEX
        BNE     DLY
        LDAA    #%00111100    TURN LED OFF
        STAA    STATB
        SWI
```

Memory Address (Hex)	Instruction (Mnemonic)			Memory Contents (Hex)
00		CLR	STATB	7F
01				80
02				07
03		CLR	DATAB	7F
04				80
05				06
06		LDAA	#%00110100	86
07				34
08		STAA	STATB	B7
09				80
0A				07
0B		LDX	#$F423	CE
0C				F4
0D				23
0E	DLY	DEX		09
0F		BNE	DLY	26
10				FD
11		LDAA	#%00111100	86
12				3C
13		STAA	STATB	B7
14				80
15				07
16		SWI		3F

Does the switch have any effect on the program? In this configuration, the level and pulse length of the serial output are controlled by the program.

Change the program so that key 7 turns the LED on and key 0 turns it off. Remember that you can wait until key D is pressed with the instructions.

```
WAIT0      LSR       STATUS     IS KEY 0 PRESSED?
           BCS       WAIT0      NO, WAIT

WAIT0      LSR       STATUS     74
                                80
                                04
           BCS       WAIT0      25
                                FB
```

CB2 can also act as an acknowledge or "COMPUTER READY" signal. In this case, it goes low after the CPU writes into the PIA and remains low until there is a transition on CB1. The following program will produce this type of CB2 signal.

```
         CLR     STATB
         CLR     DATAB        DATA LINES INPUTS
         LDAA    #%00100100   TURN LED ON
         STAA    STATB
         CLR     DATAB
HERE     BRA     HERE
```

Memory Address (Hex)	Instruction (Mnemonic)			Memory Contents (Hex)
00		CLR	STATB	7F
01				80
02				07
03		CLR	DATAB	7F
04				80
05				06
06		LDAA	#%00100100	86
07				24
08		STAA	STATB	B7
09				80
0A				07
0B		CLR	DATAB	7F
0C				80
0D				06
0E	HERE	BRA	HERE	20
0F				FE

What happens when you open and close the switch on CB1? What happens if you replace CLR DATAB with LDAA DATAB? How about STAA DATAB, ADDA DATAB, LSR DATAB, TST DATAB? Why is this feature useful? Note that PIA side 'A' produces an equivalent signal in response to a read operation. Try side A and see which instructions turn the LED on.

Another control line option produces a brief pulse which can indicate the presence of new data to the peripheral. You can employ this option by replacing LDAA #%00100100 in the last program with LDAA #%00101100. Run the program with this change. What happens?

This pulse or strobe is too brief for you to see. You can check for its presence by attaching CB2 to CB1. The following program will end with the old contents of the control register (before the strobe) in accumulator A and the contents after the strobe in accumulator B.

```
LDAA    STATB        A = OLD CONTROL REGISTER
CLR     DATAB        PRODUCE STROBE
LDAB    STATB        B = NEW CONTROL REGISTER
SWI
```

Memory Address (Hex)	Instruction (Mnemonic)	Memory Contents (Hex)
0B	LDAA STATB	B6
0C		80
0D		07
0E	CLR DATAB	7F
0F		80
10		06
11	LDAB STATB	F6
12		80
13		07
14	SWI	3F

What are the final contents of accumulators A and B? Why?

So the output control line options are (control register bits):

Bit 5 = 1 makes CB2 an output

Bit 4 = 1 makes CB2 a latched serial output (level) with the value of bit 3

Bit 4 = 0 makes CB2 an active-low pulse which is either a long strobe deactivated by CB1 (bit 3 = 0) or a brief strobe (bit 3 = 1).

Note the variety of circuit configurations that you can obtain from a single PIA under program control.

If you need to clear the READY bit on an output port or produce a write strobe from side A of a PIA, you can always use the sequence

```
STAA    PIADR        OUTPUT DATA
LDAA    PIADR        DUMMY READ
```

The LDAA instruction does not change any registers (why?) or do anything else except waste time. Similarly, if you want a read strobe from side B of a PIA, the sequence

```
LDAA    PIADR        INPUT DATA
STAA    PIADR        DUMMY WRITE
```

will do the job with no side effects (why?). But be sure to document these seemingly useless instructions so that you don't accidentally eliminate them.

LABORATORY C

Interrupts

Interrupts provide direct serial inputs into the CPU. With an interrupt, an external device can directly inform the CPU that it has data, is ready to receive data, or has some other requirement. The program does not have to check the status bit (i.e., have the CPU poll it) or include precautions to avoid missing an event. The computer, instead, goes about its normal business or simply waits for the external event.

The Micro-68 computer responds as follows to an interrupt:

1. It saves the contents of all the registers in the stack.
2. It places the contents of memory locations 006A and 006B in the program counter. We will call this 16-bit address the NEWPC pointer.

So you must place the address of the interrupt service routine in NEWPC.

You can enable or disable the interrupt system as follows:

CLI (0E) enables interrupts (i.e., clears the interrupt disable bit).

SEI (0F) disables interrupts (i.e., sets the interrupt disable bit).

Remember that the processor automatically disables interrupts on reset or on accepting an interrupt (why do you think it does this?)

Each PIA also has its own interrupt enable bit, bit 0 of the control register. It must be 1 to allow an interrupt from the PIA. Reading the data from the PIA clears the interrupt bit (control register bit 7).

A SIMPLE INTERRUPT

You can generate a simple interrupt from a switch attached to line CA1 (connector pin A). The following program will wait for you to close the switch and then return to the monitor. Note that you must enable both the overall interrupt in the CPU with the CLI instruction and the PIA interrupt. Reading the PIA data register (after debouncing) clears the interrupt.

```
         SEI                      DISABLE INTERRUPT
         LDX    #$30              STORE INTERRUPT SERVICE ADDRESS
         STX    NEWPC
         LDAA   #%00000101        ENABLE PIA INTERRUPT
         STAA   STATA
         CLI                      ENABLE INTERRUPT
WAITI    BRA    WAITI             AND WAIT
         ORG    $30
         LDX    #$3E              DEBOUNCE SWITCH
DLY      DEX
         BNE    DLY
         LDAA   DATAA             CLEAR INTERRUPT
         SWI
```

Note that you should disable the interrupt initially since the monitor does not do it automatically.

Memory Address (Hex)	Instruction (Mnemonic)			Memory Contents (Hex)
00		SEI		0F
01		LDX	#$30	CE
02				00
03				30
04		STX	NEWPC	DF
05				6A
06		LDAA	#%00000101	86
07				05
08		STAA	STATA	B7
09				80
0A				05
0B		CLI		0E
0C	WAITI	BRA	WAITI	20
0D				FE
30		LDX	#$3E	CE
31				00
32				3E
33	DLY	DEX		09
34		BNE	DLY	26
35				FD
36		LDAA	DATAA	B6
37				80
38				04
39		SWI		3F

Enter and run the program.

What are the final contents of the stack? The external interrupt, just like an SWI instruction, causes the CPU to save all the registers in the stack. What is the value of register A at the end of the interrupt service routine? The original value of A in the main program is still in the stack (where?).

What is the value of the interrupt flag at the end of the interrupt service routine? The CPU automatically disables the interrupt when it accepts one. Again, note that the original value of the interrupt flag is still in the stack.

Change the program so that it uses control line CB1 (connector pin 18) instead of CA1. Now change it to use CB2 (connector pin V). Note that you must set control register bit 3 to 1 to enable the interrupt on control line 2.

COMMUNICATION BETWEEN MAIN PROGRAM AND INTERRUPT

Since the 6800 saves the contents of all its registers on accepting an interrupt and restores them all (via the RTI instruction) after servicing the interrupt, you must use the memory to pass information between the main program and the interrupt service routine. The following program uses a flag in memory location 40 to determine if the interrupt from CA1 has occurred.

(40) = 0 before interrupt
(40) = 1 after interrupt

Main Program:

```
        SEI                      DISABLE INTERRUPT
        LDX     #$30             STORE INTERRUPT SERVICE ADDRESS
        STX     NEWPC
        LDAA    #%00000101       ENABLE PIA INTERRUPT
        STAA    STATA
        CLR     $40              INTERRUPT FLAG=0
        CLI                      ENABLE INTERRUPT
WAITM   TST     $40              HAS INTERRUPT OCCURRED?
        BEQ     WAITM            NO, WAIT
        SWI
```

Interrupt service routine:

```
        ORG     $30
        LDX     #$3E             WAIT 1 MS. TO DEBOUNCE SWITCH
DLY     DEX
        BNE     DLY
        LDAA    DATAA            CLEAR INTERRUPT
        INC     $40              MARKER=1
        RTI
```

Memory Address (Hex)	Instruction (Mnemonic)		Memory Contents (Hex)
00	SEI		0F
01	LDX	#$30	CE
02			00
03			30
04	STX	NEWPC	DF
05			6A
06	LDAA	#%00000101	86
07			05
08	STAA	STATA	B7
09			80

Memory Address (Hex)	Instruction (Mnemonic)			Memory Contents (Hex)
0A				05
0B		CLR	$40	7F
0C				00
0D				40
0E		CLI		0E
0F	WAITM	TST	$40	7D
10				00
11				40
12		BEQ	WAITM	27
13				FB
14		SWI		3F
30		LDX	#$3E	CE
31				00
32				3E
33	DLY	DEX		09
34		BNE	DLY	26
35				FD
36		LDAA	DATAA	B6
37				80
38				04
39		INC	$40	7C
3A				00
3B				40
3C		RTI		3B

Enter and run the program. Try using accumulator B instead of memory location 40. What happens? Note that you do not have to reorganize the program – put NOP (01) in the unused spots, i.e., replace CLR $40 with CLRB, NOP, NOP.

The problem is that you have not changed the value of register B in the stack. Instead of INCB, try

```
TSX                    USE STACK POINTER AS DATA POINTER
INX                    AND INCREMENT B IN STACK
INC       X

39                     TSX                30
3A                     INX                08
3B                     INC   X            6C
3C                                        00
3D                     RTI                3B
```

Explain why this works. How would you set the interrupt flag in the stack so as to have the interrupts disabled on returning?

USING THE INTERRUPTS

Note that using the interrupt means that the program does not have to examine the status bit. Instead, the status bit informs the processor that it is active. The result is that the computer can be doing useful work until it is interrupted. The following program simply counts using three memory locations (40, 41, and 42). See how high it counts before you can interrupt it.

```
            SEI                       CLEAR INTERRUPT
            LDX      #$30             STORE INTERRUPT SERVICE ADDRESS
            STX      $6A
            LDAA     #%00000101       ENABLE PIA INTERRUPT
            STAA     STATA            ALL LOCATIONS = 0
            CLR      $40
            CLR      $41
            CLR      $42
            CLI                       ENABLE INTERRUPT
    CT1     INC      $42              AND COUNT
            BNE      CT1
            INC      $41
            BNE      CT1
            INC      $40
            BRA      CT1
```

The interrupt service routine is the same as in the first example.

Memory Address (Hex)	Instruction (Mnemonic)	Memory Contents (Hex)
00	SEI	0F
01	LDX #$30	CE
02		00
03		30
04	STX NEWPC	DF
05		6A
06	LDAA #%00000101	86
07		05
08	STAA STATA	B7
09		80
0A		05
0B	CLR $40	7F
0C		00
0D		40
0E	CLR $41	7F
0F		00
10		41
11	CLR $42	7F
12		00
13		42

Memory Address (Hex)	Instruction (Mnemonic)	Memory Contents (Hex)
14	CLI	0E
15 CT1	INC $42	7C
16		00
17		42
18	BNE CT1	26
19		FB
1A	INC $41	7C
1B		00
1C		41
1D	BNE CT1	26
1E		F6
1F	INC $40	7C
20		00
21		40
22	BRA CT1	20
23		F1

You can reset the computer to clear the stack. Try the program several times. With the interrupt, no time is wasted looking for the switch input yet the response is immediate.

Change the program so that the interrupt is initially disabled but is enabled by pressing key F. Check for key F during each cycle. Close the switch while the interrupt is disabled and then reopen it. What happens when you press key F? Does it matter if the initialization disables the PIA interrupt or the entire system? Change the program so that it includes an LDAA DATAA instruction in each cycle. Now does the PIA remember the interrupt? An unserviced interrupt remains active unless the program clears it.

POLLING INTERRUPTS

If there is more than one source for an interrupt, the CPU must check the status bits of the PIAs. The advantage of the interrupt here is that the CPU knows that one bit is active. Attach switches to both CA1 and CA2. The following program will wait for the interrupt and then display either 1 or 2 on the LEDs.

Main program:

```
        SEI                    DISABLE INTERRUPT
        LDX     #$30           STORE INTERRUPT SERVICE ADDRESS
        STX     $6A
        LDAA    #%00000101     ENABLE PIA INTERRUPTS
        STAA    STATA
        CLI                    ENABLE INTERRUPT
WAITI   BRA     WAITI          AND WAIT
```

Interrupt service routine:

```
              ORG     $30
              LDAB    #$9F           CODE TO DISPLAY 1
              LDAA    STATA          IS INTERRUPT FROM LINE 1?
              BMI     DSPLY          YES, DISPLAY 1
              ASLA                   IS INTERRUPT FROM LINE 2?
              BPL     LAST           NO, JUST WAIT
              LDAB    #$25           YES, GET CODE TO DISPLAY 2
DSPLY         LDAA    DATAA          CLEAR INTERRUPT
              LDAA    #$FF           TURN ON ALL DISPLAYS
              STAA    LIGHTS
              STAB    GLOW           AND DISPLAY 1 OR 2
LAST          BRA     LAST           WAIT
```

Memory Address (Hex)	Instruction (Mnemonic)			Memory Contents (Hex)
00		SEI		0F
01		LDX	#$30	CE
02				00
03				30
04		STX	$6A	DF
05				6A
06		LDAA	#%00000101	86
07				0D
08		STAA	STATA	B7
09				80
0A				05
0B		CLI		0E
0C	WAITI	BRA	WAITI	20
0D				FE
30		LDAB	#9F	C6
31				9F
32		LDAA	STATA	B6
33				80
34				05
35		BMI	DSPLY	2B
36				05
37		ASLA		48
38		BPL	LAST	2A
39				0E
3A		LDAB	#$25	C6
3B				25
3C	DSPLY	LDAA	DATAA	B6
3D				80
3E				04

Memory Address (Hex)	Instruction (Mnemonic)			Memory Contents (Hex)
3F		LDAA	#$FF	86
40				FF
41		STAA	LIGHTS	B7
42				80
43				0A
44		STAB	GLOW	F7
45				80
46				08
47	LAST	BRA	LAST	20
48				FE

Enter and run the program. Change the main program so that it waits for key F to be pressed before enabling the interrupts. What happens if you close one of the switches before pressing key F? What happens if you close both switches? The interrupt which takes precedence over the other one is said to have higher priority. Change the priority. Note that the order in which the interrupt flag bits are examined determines the priority of the interrupt. A system which uses flag bits to identify the source of an interrupt is called a polling interrupt system.

Examining Computer Signals

This laboratory exercise assumes that you have a dual-trace oscilloscope and an AMP 86-pin connector which fits on the front of the Micro-68 computer.

Enter the following simple loop program into the computer. It branches back to itself endlessly.

```
LOOP    BRA    LOOP
```

Memory Address (Hex)	Instruction (Mnemonic)	Memory Contents (Hex)	(Binary)
00	LOOP BRA LOOP	20	00100000
01		FE	11111110

Attach the probe from one channel of the oscilloscope to clock phase φ_2. You can get φ_2 from the resistor just above the 6800 microprocessor and on the right-hand side (toward the keyboard). φ_2 should be a regular series of pulses with a frequency of 1 mHz. The pulse width is about 1/2 microsecond. Be sure that you have φ_2 on the oscilloscope before you proceed.

Since the only memory addresses used by the program are 0000 and 0001, only address line A0 will change. Attach the probe from the other channel of the oscilloscope to address line A0 (Prior V on the 86-pin connector). You should note that some 100-pin connectors are upside-down with respect to the AMP connector. We will therefore give both designations from here on – AMP and its opposite. The pins are in the same position but in opposite vertical planes. So A0 is pin V, 40 (V on AMP, 40 on the other connector).

Try to figure out how the instruction cycle works by lining up φ_2 and A0. The computer executes the BRA LOOP instruction as follows:

CYCLE	ADDRESS	DATA	PURPOSE
1	0000	20	FETCH
2	0001	FE	FETCH
3	?	?	RELATIVE ADDRESSING
4	?	?	RELATIVE ADDRESSING

The first two cycles are used to fetch the instruction and the relative offset. The second two cycles are used to add the offset and the 16-bit program counter. You can determine when cycle 2 starts by noting when A0 goes high.

Cycles 3 and 4 do not use the memory so VMA is low. Attach the second probe to VMA, connector pin L, 10. VMA should be '1' during cycles 1 and 2, '0' during cycles 3 and 4. Remember that φ_2 is low during the first half of each cycle and high during the second half.

Now look at the data lines. Note that 20 (BRA) consists of data bit 5 (0's) high and all others low; FE (the offset) has D0 low and all others high. So the data lines should be:

CYCLE	D7, D6, D4, D3, D2, D1	D5	D0
1	0	1	0
2	1	1	0

Attach the second probe to D5, connector pin $\overline{J}$,30. When does D5 become '1'? Try D0, connector pin $\overline{K}$, 30 and then D1, connector pin $\overline{H}$, 29. Note that the address changes occur during the first half of the cycle while the data changes occur during the second half of the cycle. The processor does not use the data bus float during the first half of the cycle when addresses may be changing. Why? Can you see a delay between the rising edge of φ_2 and a change on the data lines?

Now add another instruction to the loop. The new program is:

Memory Address (Hex)	Instruction (Mnemonic)			Memory Contents (Hex)	(Binary)
00	LOOP	NOP		01	00000001
01		BRA	LOOP	20	00100000
02				FD	11111101

The execution of NOP takes two cycles, one to fetch the instruction and one to execute it. Compare address lines A0 and A1 (pin 40, $\overline{V}$) to φ_2 and draw them on a piece of paper. Can you determine where the loop begins? What happens if you change memory location 0002 from FD to FE? Why?

Now examine the data lines. Note that they are all the same (0, 0, 1) except lines 0 (1, 0, 1), 1 (0, 0, 0), and 5 (0, 1, 1). Examine the READ/WRITE line. Does its value change?

Extend the loop by one more instruction as follows:

Memory Address (Hex)	Instruction (Mnemonic)			Memory Contents (Hex)	(Binary)
00	LOOP	SUBA	#0	80	10000000
01				00	00000000
02		BRA	LOOP	20	00100000
03				FC	11111100

Examine address lines A0 and A1, VMA, R/W, and the data lines. We chose the instruction SUBA #0 because of its simple bit pattern. It executes in two cycles (can you see it on the scope?). What happens during those two cycles? What is the value of VMA?

Now change memory location 0000 from 80 to 90. What happens to the data and address lines and VMA? Examine address line A7 (pin $\overline{S}$). The difference is that 80 is SUBA IMMEDIATE while 90 is SUBA DIRECT. Direct addressing means that the computer needs an extra cycle to fetch the data from the direct address (0000 in this case). So SUBA 0 requires three cycles instead of two.

Now try the following program:

Memory Address (Hex)	Instruction (Mnemonic)			Memory Contents (Hex)	(Binary)
00	LOOP	STX	$20	FF	11111111
01				00	00000000
02				20	00100000
03		BRA	LOOP	20	00100000
04				FB	11111011

Explain how this instruction is executed by examining:

A0	pin $\overline{V}$, 40
A1	pin 40, $\overline{V}$
A2	pin 39, $\overline{U}$
A5	pin 38, $\overline{T}$
D0	pin $\overline{K}$, 31
D2	pin 31, $\overline{K}$
D5	pin $\overline{J}$, 30
VMA	pin L, 10
R/W	pin #, 6

What happens if you change STX (FF) to LDX (FE)? Put 00 in memory location 0020 and FF in 0021.

The contents of the index register are underlined in the program with STX above. What happens if you enter the following program?

```
        LDX    #$00FF
LOOP    STX    $20
        BRA    LOOP
```

LABORATORY E

External Clocks

Computers often must determine the timing of a peripheral from an external clock. The clock is simply a regular series of pulses; once the computer determines when the pulses start and how long they are, the computer can then transfer data to or from a peripheral governed by the clock. The clock can also inform the computer of the passage of a certain amount of time. A real-time clock interrupts the computer at regular intervals. The computer can count interrupts and thus handle inputs and outputs at specified points in time.

This laboratory assumes that you have a variable external clock source. The TTL output from a function generator will do. So will any other source that can produce a clock in the 1 to 100 Hz range.

Attach the clock to control line CA1 (pin A of the 36-pin connector). The following program will wait for the first high-to-low transition (trailing edge) on the clock line:

```
WAITON    LDAA    STATA     CLOCK TRANSITION?
          BPL     WAITON    NO, WAIT
          LDAA    DATAA     CLEAR CLOCK STATUS FLAG
          SWI
```

The hexadecimal version of the program is:

Memory Address (Hex)	Instruction (Mnemonic)	Memory Contents (Hex)
00	WAITON LDAA STATA	B6
01		80
02		05
03	BPL WAITON	2A
04		FB
05	LDAA DATAA	B6
06		80
07		04
08	SWI	3F

Enter and run the program. Why is the LDAA DATAA instruction necessary? How would you change the program so that it waited for the first low-to-high transition (leading edge) on the clock line? How would you change it to use CA2 (pin 1) instead of CA1?

The following program will wait for ten high-to-low transitions:

```
          LDAB    #10       COUNT = 10
WAITON    LDAA    STATA     CLOCK TRANSITION?
          BPL     WAITON
          LDAA    DATAA     CLEAR CLOCK STATUS FLAG
          DECB              HAVE 10 TRANSITIONS OCCURRED?
          BNE     WAITON    NO, WAIT FOR NEXT TRANSITION
          SWI
```

The hexadecimal version is:

Memory Address (Hex)	Instruction (Mnemonic)	Memory Contents (Hex)
00	LDAB #10	C6
01		0A
02	WAITON LDAA STATA	B6
03		80
04		05
05	BPL WAITON	2A
06		FB
07	LDAA DATAA	B6
08		80
09		04
0A	DECB	5A
0B	BNE WAITON	26
0C		F5
0D	SWI	3F

Enter the program and run it with a 5 Hz clock rate. How long a delay do you get? How can you make the delay 10 seconds? How accurate will the delay be? How accurate could you make it with a 10 Hz clock?

Note the importance of the LDDA DATAA instruction. What happens if you replace this instruction with:

(a) 3 NOP's

(b) STAA DATAA

(c) LDAA STATA

(d) TST DATAA

(e) CLR DATAA

The computer can also determine the clock period. The following program will store the clock period (in milliseconds) in memory location (50).

```
           LDAA    DATAA     CLEAR STATUS
           CLRB              PERIOD = 0
WAITST     LDAA    STATA     HAS PULSE STARTED?
           BPL     WAITST    NO, WAIT FOR PULSE TO START
           LDAA    DATAA     CLEAR STATUS FLAG
COUNT      INCB              PERIOD = PERIOD + 1 MS.
           JSR     D1MS      WAIT 1 MS.
           LDAA    STATA     HAS NEXT PULSE STARTED?
           BPL     COUNT     NO, KEEP COUNTING
           STAB    $50       SAVE PERIOD
           LDAA    DATAA     CLEAR STATUS FLAG
           SWI
```

The delay program (from Laboratory 4) is:

```
DIMS     LDX     #$3E       WAIT 1 MS.
DLY      DEX
         BNE     DLY
         RTS
```

Try this program with clocks of 10, 20, and 50 Hz. How accurate is it? How could you make it more accurate? Note that no delay program was necessary in the earlier examples since the external clock provided the timing.

A real-time clock allows the computer to go about its normal business and still handle inputs and outputs at particular times. No delay routines are needed.

The following program uses a 1 Hz real-time clock to wait for one second before turning the displays on. We'll assume that the program and the clock start at the same time.

```
         SEI                    DISABLE CPU INTERRUPT
         LDX     #$30           STORE INTERRUPT SERVICE ADDRESS
         STX     NEWPC
         LDAA    #%00000101     ENABLE PIA INTERRUPT
         STAA    STATA
         CLR     $40            CLOCK COUNTER = 0
         LDAA    #1             DESIRED CLOCK COUNT = 1
         CLI                    ENABLE CPU INTERRUPT
WAITI    CMPA    $40            HAS DESIRED COUNT ELAPSED?
         BNE     WAITI          NO, WAIT
         LDAA    #$FF           YES, TURN DISPLAYS ON
         STAA    LIGHTS
         CLR     GLOW           TURN SEGMENTS ON
HERE     BRA     HERE           AND WAIT

         ORG     $30
         LDAA    DATAA          CLEAR CLOCK STATUS FLAG
         INC     $40            INCREMENT CLOCK COUNTER
         RTI
```

Note that we leave the interrupt on. Does it have any effect? Enter and run the program. Change it so that it delays for 10 seconds before turning the displays on. How would you get a 10 second delay with a 10 Hz clock.

The real-time clock is particularly useful when the computer must perform actions regularly. For example, to have the computer turn the displays on for 1 of every 2 seconds, simply change the end of the program to:

```
         LDAB    #2
WAITON   CMPB    $40        HAS ANOTHER SECOND ELAPSED?
         BNE     WAITON     NO, WAIT
         CLR     LIGHTS     YES, TURN DISPLAYS OFF
         BRA     STTIME     AND START AGAIN
```

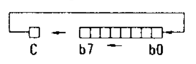
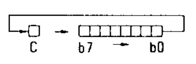
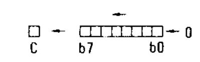
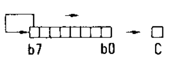
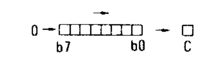
The program could also take a variable set of time intervals from a table. How would you write a program to handle the following table?

(41)	=	NO. OF SECONDS OFF	=	08
(42)	=	NO. OF SECONDS ON	=	06
(43)	=	NO. OF SECONDS OFF	=	03
(44)	=	NO. OF SECONDS ON	=	05
(45)	=	NO. OF SECONDS OFF	=	02
(46)	=	NO. OF SECONDS ON	=	10
(47)	=	NO. OF SECONDS OFF	=	04
(48)	=	NO. OF SECONDS ON	=	05

Let the program first run through the table once. Then revise the program so that it goes through the table indefinitely, i.e., it goes back to 0041 after finishing 0048.

STTIME is the instruction (CLR $40) which clears the clock counter. Make the program turn the displays on for 1 of every 10 seconds, 4 of every 20, 3 of every 8. Note that all you have to do to change the duty cycle is vary the constants in memory locations 000F and 001E.

ACCUMULATOR AND MEMORY INSTRUCTIONS

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

LEGEND:

OP	Operation Code (Hexadecimal);	+	Boolean Inclusive OR;
~	Number of MPU Cycles;	⊕	Boolean Exclusive OR;
#	Number of Program Bytes;	$\overline{M}$	Complement of M;
+	Arithmetic Plus;	→	Transfer Into;
−	Arithmetic Minus;	0	Bit = Zero;
·	Boolean AND;	00	Byte = Zero;
M_{SP}	Contents of memory location pointed to be Stack Pointer;		

Note – Accumulator addressing mode instructions are included in the column for IMPLIED addressing

CONDITION CODE SYMBOLS:

H	Half-carry from bit 3;
I	Interrupt mask
N	Negative (sign bit)
Z	Zero (byte)
V	Overflow, 2's complement
C	Carry from bit 7
R	Reset Always
S	Set Always
↕	Test and set if true, cleared otherwise
●	Not Affected

INDEX REGISTER AND STACK MANIPULATION INSTRUCTIONS

		IMMED			DIRECT			INDEX			EXTND			IMPLIED				COND. CODE REG. 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				$X_H - M, X_L - (M + 1)$	●	●	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	$SP + 1 \rightarrow X$	●	●	●	●	●	●

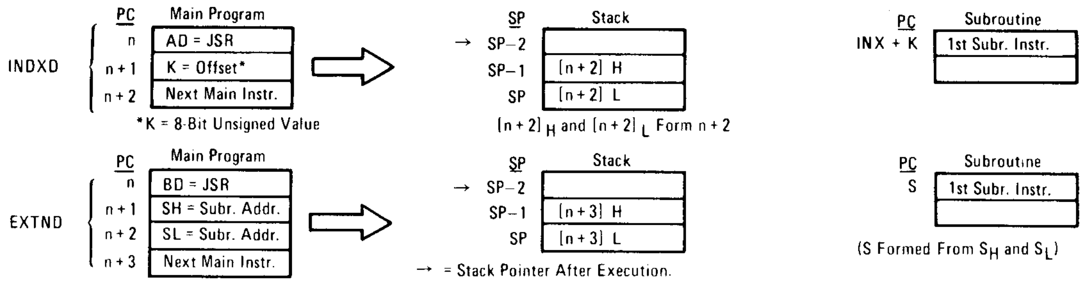
JUMP AND BRANCH INSTRUCTIONS

		RELATIVE			INDEX			EXTND			IMPLIED				COND. CODE REG. 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		⑩					
Return From Subroutine	RTS										39	5	1	See Special Operations	●	●	●	●	●	●
Software Interrupt	SWI										3F	12	1		●	●	●	●	●	●
Wait for Interrupt*	WAI										3E	9	1		●	⑪	●	●	●	●

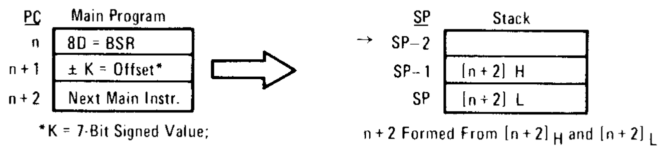
*WAI puts Address Bus, R/W, and Data Bus in the three-state mode while VMA is held low.

SPECIAL OPERATIONS

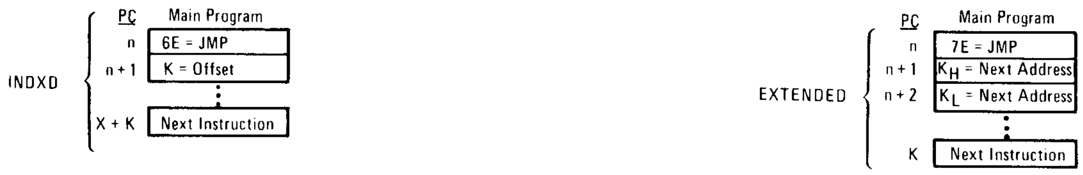
JSR, JUMP TO SUBROUTINE:



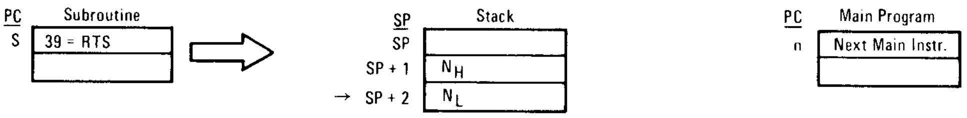
BSR, BRANCH TO SUBROUTINE:



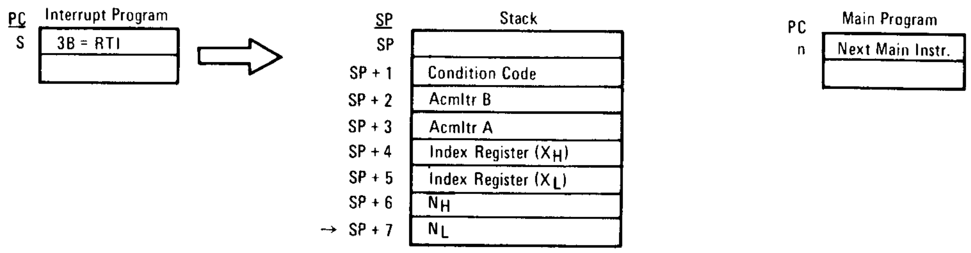
JMP, JUMP:



RTS, RETURN FROM SUBROUTINE:



RTI, RETURN FROM INTERRUPT:



CONDITION CODE REGISTER MANIPULATION INSTRUCTIONS

		IMPLIED				COND. CODE REG.					
						5	4	3	2	1	0
OPERATIONS	MNEMONIC	OP	~	#	BOOLEAN OPERATION	H	I	N	Z	V	C
Clear Carry	CLC	0C	2	1	0 → C	●	●	●	●	●	R
Clear Interrupt Mask	CLI	0E	2	1	0 → I	●	R	●	●	●	●
Clear Overflow	CLV	0A	2	1	0 → V	●	●	●	●	R	●
Set Carry	SEC	0D	2	1	1 → C	●	●	●	●	●	S
Set Interrupt Mask	SEI	0F	2	1	1 → I	●	S	●	●	●	●
Set Overflow	SEV	0B	2	1	1 → V	●	●	●	●	S	●
Acmltr A → CCR	TAP	06	2	1	A → CCR	(12)					
CCR → Acmltr A	TPA	07	2	1	CCR → 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⊕C after shift has occurred.
7	(Bit N)	Test: Sign bit of most significant (MS) byte = 1?
8	(Bit V)	Test: 2's complement overflow from subtraction of MS 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.