M6800

Assembly Language Programming

3. MC6802 MICROPROCESSOR

MC6802 microprocessor runs in 1MHz clock cycle. It has 64 Kbyte memory address capacity using 16-bit addressing path (A0-A15). The 8-bit data path (D0-D7) is bidirectional and has three states. It has 72 instructions which are 1, 2 or 3 byte instructions.

MC6802 microprocessor has 3 interrupt inputs. One of them is maskable (IRQ), the other one is unmaskable (NMI) and the last one is the reset (RESET). It also has 2 special instructions: SWI (software interrupt) and WAI (wait for interrupt). MC6802's pin numbers and their connections are shown in Figure 3.1.

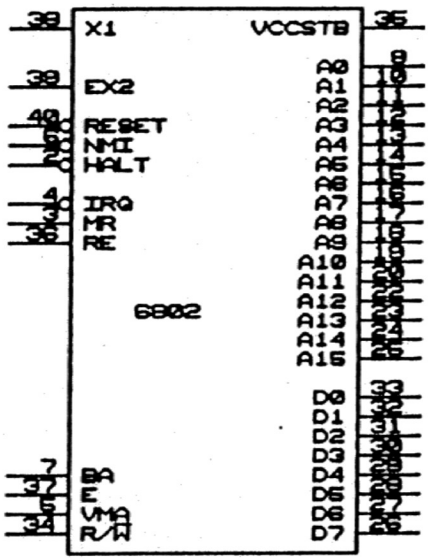

Figure 3.1 - MC6082 Microprocessor

3.1 REGISTERS

MC6802 Microprocessor has three 16-bit registers and three 8-bit registers available for use by the programmer (Figure 3.2).

- 2 Accumulators (Accumulator A, Accumulator B)
- Program Counter (PC)
- Stack Pointer (SP)
- Index Register (X)
- Condition Code Register (CCR).

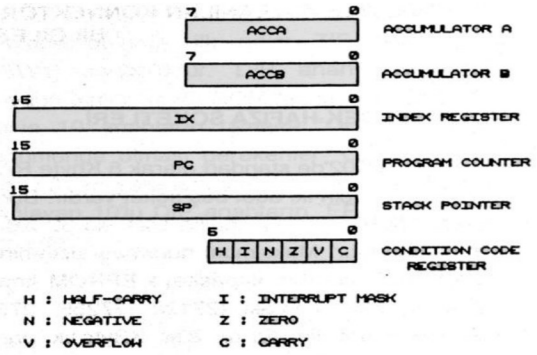

Figure 3.2 - Registers of the MC6802 Microrprocessor

Accumulators: The Microprocessor unit (MPU) contains two 8-bit accumulators (Accumulator A and Accumulator B) that are used to hold operands and/or result produced by the arithmetic logic unit (ALU).

Program Counter: It is a 2-byte (16-bit) register that points to the current program address.

Stack Pointer: It is a 2-byte register that contains the address of the next available location in an external push-down/pop–up stack. The contents of the stack pointer defines the top of the stack in RAM.

Index Register: It is a 2-byte register that is used to store data or a 2-byte memory address for indexed memory addressing.

Condition Code Register: It shows the conditions occurs as a result of an Arithmetic Logic Unit operation (Figure 3.2):

Bit 0: carry from bit 7 of an arithmetic operation (C)
Bit 1: Overflow flag (V)
Bit 2: Zero flag (Z)
Bit 3: Negative flag (N)
Bit 4: Interrupt Mask (I)
Bit 5: Half carry from bit 3 of an arithmetic operation (H)
Bit 6: Unused
Bit 7: Unused

These bits of the Condition Code Register are used as testable conditions for the conditional branch instructions. Bit 4 of the CCR is the interrupt mask bit (I). The unused bits of the Condition Code Register (bit 6 and bit 7) are 1.

Figure 3.3 shows the internal connections of the registers and the other units of the MC6802.

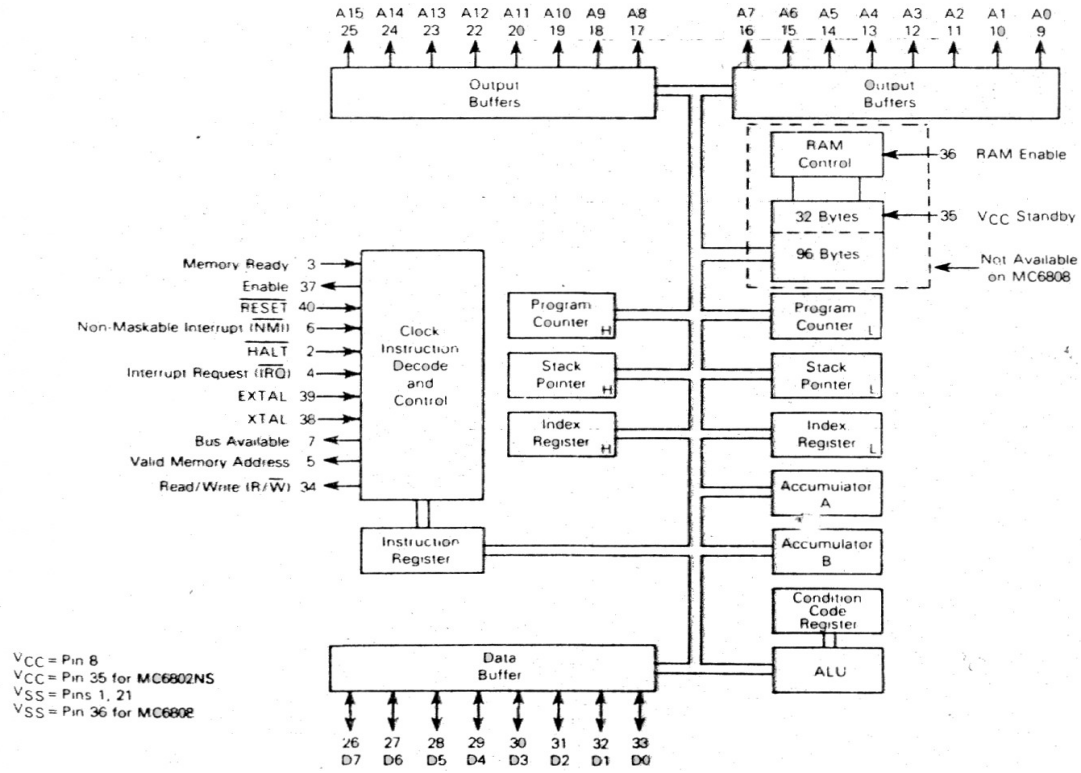

Figure 3.3 - Functional block diagram of 6802 MPU

3.2 ADDRESSING MODES

MC6802 Microprocessor has 7 addressing modes that can be used by the programmer:

1. Accumulator
2. Immediate
3. Direct
4. Extended
5. Indexed
6. Implied (Inherent)
7. Relative

MC6802 instructions may be used with one or more of these addressing modes. The instruction set and their addressing modes are given in Appendix A.

Accumulator Addressing

In accumulator addressing, either accumulator A or accumulator B is specified. These are 1-byte instructions.

Ex: ABA adds the contetns of accumulators and stores the result in accumulator A

Immediate Addressing

In immediate addressing, operand is located immediately after the opcode in the second byte of the instruction in program memory (except LDS and LDX where the operand is in the second and third bytes of the instruction). These are 2-byte or 3-byte instructions.

Ex: LDAA #25H loads the number $(25)_H$ into accumulator A

Direct Addressing

In direct addressing, the address of the operand is contained in the second byte of the instruction. Direct addressing allows the user to directly address the lowest 256 bytes of the memory, i.e, locations 0 through 255. Enhanced execution times are achieved by storing data in these locations. These are 2-byte instructions.

Ex: LDAA 25H loads the contents of the memory address $(25)_H$ into accumulator A

Extended Addressing

In extended addressing, the address contained in the second byte of the instruction is used as the higher eight bits of the address of the operand. The third byte of the instruction is used as the lower eight bits of the address for the operand. This is an absolute address in the memory. These are 3-byte instructions.

Ex: LDAA 1000H loads the contents of the memory address $(1000)_H$ into accumulator A

Indexed Addressing

In indexed addressing, the address contained in the second byte of the instruction is added to the index register's lowest eight bits. The carry is then added to the higher order eight bits of the index register. This result is then used to address memory. The modified address is held in a temporary address register so there is no change to the index register. These are 2-byte instructions.

Ex: **LDX #1000H**
LDAA 10H,X

Initially, LDX #1000H instruction loads 1000H to the index register (X) using immediate addressing. Then LDAA 10H,X instruction, using indexed addressing, loads the contents of memory address $(10)_H + X = 1010_H$ into accumulator A.

Implied (Inherent) Addressing

In the implied addressing mode, the instruction gives the address inherently (i.e, stack pointer, index register, etc.). Inherent instructions are used when no operands need to be fetched. These are 1-byte instructions.

Ex: **INX** increases the contents of the Index register by one. The address information is "inherent" in the instruction itself.
INCA increases the contents of the accumulator A by one.
DECB decreases the contents of the accumulator B by one.

Relative Addressing

The relative addressing mode is used with most of the branching instructions on the 6802 microprocessor. The first byte of the instruction is the opcode. The second byte of the instruction is called the ***offset***. The offset is interpreted as a ***signed 7-bit number***. If the MSB (most significant bit) of the offset is 0, the number is positive, which indicates a forward branch. If the MSB of the offset is 1, the number is negative, which indicates a backward branch. This allows the user to address data in a range of -126 to +129 bytes of the present instruction. These are 2-byte instructions.

Ex:	**PC**	**Hex Label**	**Instruction**
	0009	2004	BRA 0FH

Figure 3.4 shows the address calculation in the execution of the unconditional branch instruction (BRA). Program counter (PC) before the operation is 0009_H. The opcode of the "branch always" instruction (20_H) is fetched from location 0009_H in program memory with the offset 04_H (00000100_2). Then the program counter is incremented to the address of the next instruction ($000B_H$) just before the actual operand fetch. The 6802 processor internally adds the offset (04_H) to the current contents of program counter ($000B_H$). The new address in the program counter after the "branch always" operation is $000B+04=000F_H$ ($0000\ 0000\ 0000\ 1111_2$). The processor then jumps to this new address and fetches an instruction from location $000F_H$. Note that the offset's most significant bit (MSB) is 0. This indicates a positive offset, which causes a forward branch.

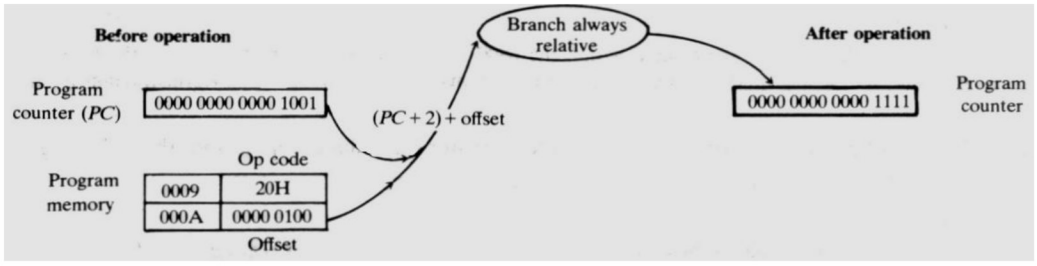

Figure 3.4 - Relative Addressing (branching forward)

All branch operations use relative addressing mode. Branches can be forward or backward. The program in Figure 3.5 is an example for the use of branch instructions. In the first branch instruction (BRA NEXT), the address to be branched is 109_H. As relative addressing is used, the offset is calculated as

$$109_H - 105_H = 04_H$$

where 105_H is the contents of PC which points to the next instruction. The offset is written in the machine code program as the operand of the branch instruction (20 04_H). The second branch instruction (BRA LAST) is a backward branch. The displacement (offset) is calculated as

$$105_H - 10E_H = - 09_H$$

where $10E_H$ is the contents of PC. As the offset is a negative number, its 2's complement ($F7_H$) is used as the offset (20 $F7_H$).

Memory Address	Machine Code Program	**Assembly Language Program**			PC after instruction execution
			ORG	100H	
0100	B6 0110	BEGIN:	LDAA	110H	**0103**
0103	20 04		BRA	NEXT	**0109**
0105	B7 0130	LAST:	STAA	130H	**0108**
0108	3F		SWI		**-**
0109	BB 0120	NEXT:	ADDA	120H	**010C**
010C	20 F7		BRA	LAST	**0105**

Figure 3.5 - A program using branch instruction

4. 6802 ASSEMBLY LANGUAGE PROGRAMMING I

4.1 Flags

The 6802 MPU uses six condition code bits or flags (Figure 4.1). These flags are grouped into an 8-bit register called the Condition Code Register (CCR). The branch instructions test these flags to determine whether a branch will be taken or not.

As on the generic, the carry flag (C) is set to 1 whenever a carry (or 'borrow') is generated out by the most significant bit (MSB) of the accumulator. A sum larger than the capacity of the 8-bit accumulator sets the C flag to 1.

The overflow flag (V) in the condition code register of the MPU indicates a 2's complement overflow. When dealing with signed numbers, the MSB (B_7) of accumulator(s) is the sign bit. The remaining 7 bits are written in 2's complement form. These 7 bits can hold numbers between decimal +127 to –128. This is the range of signed numbers. If the result of an arithmetic operation exceeds this range, an overflow occurs and the overflow flag (V) is set to 1.

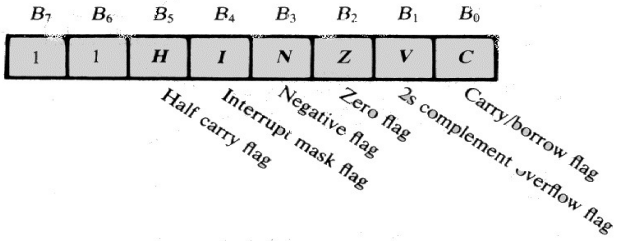

Figure 4.1 - Condition Code Register

Consider adding the positive numbers 79_{10} and 64_{10}. Decimal +79 is 01001111 in 2's complement and decimal +64 is 01000000 in 2's complement. These 2's complement numbers are added in Figure 4.2(a). Due to the carry from B_6 to B_7, the sign bit of the result changes to 1, (which indicates a negative number). This is an error. Figure 4.2(b) shows how the overflow flag is set in the microprocessor if two such numbers (in accumulator A and B) are added. The sum (10001111 in this example) is deposited in accumulator A after add operation. The overflow flag (V) is set to 1, indicating that the sum is larger than $+127_{10}$ (sum=79_{10}+64_{10}=143_{10}).

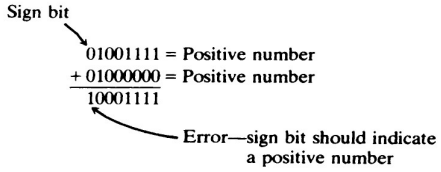

Figure 4.2 (a)

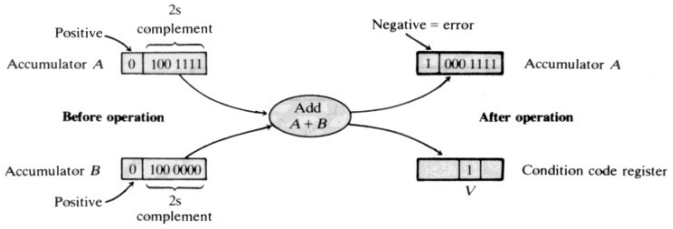

Figure 4.2 (b)

Figure 4.2 - Addition of positive numbers using 2's complement and CCR
(a) 2's complement addition showing effect on sign bit
(b) Effect on overflow flag

Consider adding two negative numbers -79_{10} and -64_{10}. Decimal –79 is 10110001 in 2's complement and decimal –64 is 11000000 in 2's complement. Since the most significant bits of both 2's complement numbers are 1 they represent negative numbers between –1 and –128. These 2's complement numbers are added in Figure 4.3(a). The result is 1 01110001. Although the sign bit must be 1 (negative), the addition results with a 0. This is an error because the sum exceeds the limit -128_{10}.

Addition of the negative numbers -79_{10} (10110001 in 2's complement) and -64_{10} (11000000 in 2's complement) using the 6802 MPU is shown in Figure 4.3(b). The 2's complement numbers are held in the accumulators A and B, and the sum is stored in accumulator A after the add operation. As the addition causes an overflow, the overflow flag (V) is set to 1, warning the user that the range of the 6802 microprocessor register is exceeded. The carry flag (C) is also set to 1, indicating the carry out from the B_7 position.

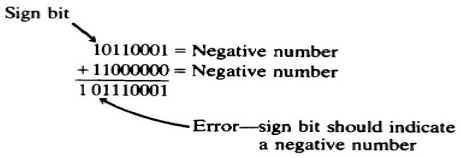

Figure 4.3 (a)

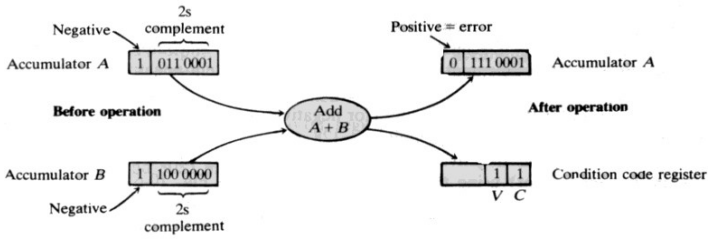

Figure 4.3(b)

Figure 4.3 - Addition of negative numbers using 2's complement and CCR
(a) 2's complement addition showing the effect on sign bit
(b) Effect on overflow flag

The zero flag (Z) in the condition code register of the 6802 MPU is set to 1 whenever the accumulator becomes zero as a result of an operation or data transfer. The zero flag resets to 0, indicating the accumulator does not contain a zero.

The negative flag (N) in the condition code register of the 6802 MPU indicates a negative result. Assume B_7 is the sign bit of the accumulator. If the result of the last arithmetic, logical or data transfer operation is negative, the N flag is set to 1. If the result is positive, the N flag is resets to 0. The N flag reflects the MSB of the accumulator.

Ex: The following program adds two 1-byte signed numbers in memory locations $(0120)_H$ and $(0121)_H$. After the addition, if the overflow flag is set, then 10_{10} is stored into location $(0040)_H$. Otherwise 20_{10} is stored into the same location.

```
        ORG     0H
        LDAA    120H        ; load the first number
        ADDA    121H        ; add them
        BVS     OVOCC       ; branch if overflow is set
        LDAA    #20         ; load (10)10 to accumulator A
        STAA    40H         ; store it in memory location (0040)H
        BRA     STOP        ; jump to the end of the program
OVOCC:  LDAA    #10
        STAA    40H
STOP:   SWI                 ; end program
```

Ex: The following program adds two 1-byte unsigned numbers in memory locations $(0120)_H$ and $(0121)_H$ and stores the result, represented as a 2-byte number, into two consecutive memory locations $(0122)_H$ and $(0123)_H$. After the addition operation accumulator A is stored ~~at~~ into address $(0123)_H$. If there is no carry, 0 is stored into the location $(0122)_H$. Otherwise carry flag is stored into $(0122)_H$ (using ADC 122H and STAA 122H instructions).

```
            ORG     100H

            LDAA    #0H     ; Clear the most significant bit
            STAA    122H
            LDAA    120H    ; load the first number
            ADDA    121H    ; add them
            STAA    123H    ; store result
            BCS     CROCC   ; branch if carry occurs
            LDAA    #0H     ; clear the most significant bit
            STAA    122H
            BRA     STOP    ; jump to the end of the program

CROCC:      LDAA    #0H     ; set accumulator to 0
            ADCA    122H    ; save carry bit in accumulator A
            STAA    122H
STOP:       SWI             ; end program
```

4.2 Looping

Loops help to repeat a section of a program for a number of times. There are three main types of loops :

1. Repeating a program section indefinitely

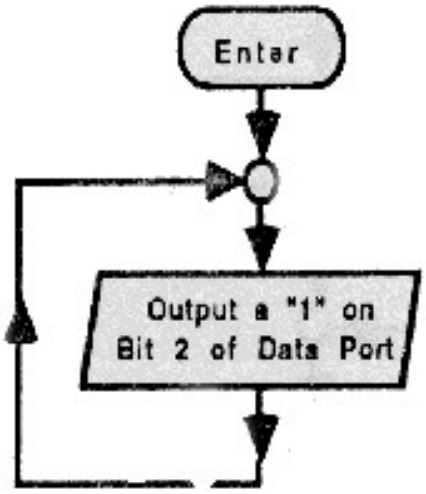

Figure 4.1 - Infinite Loop. Above code outputs a “1” on bit 2 of a data port indefinitely.

2. Repeating a program section until some predetermined condition becomes true (Figure 4.2)

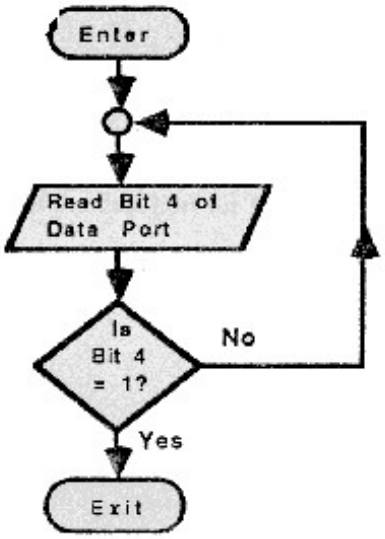

Figure 4.2 - Conditional loop. The loop is repeated until a "1" appears at input bit 4 of the data port

3. Repeating a program section for a predetermined number of passes.

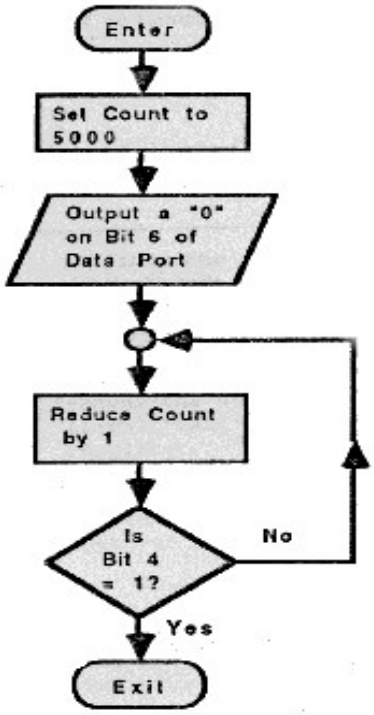

Figure 4.3 - Loop with a loop count. Above code outputs a "0" on bit 6 of a data port 5000 times

For looping in assembly language programs, branch instructions are needed. Jump and branch instructions of the 6802 microprocessor are shown in Table A.2. These instructions transfer the control from one point to another in the program.

Ex: In the following program, accumulator A is incremented by 2 during each iteration of the loop. Accumulator B is used as a counter and decremented by 1 at each iteration, until it reduces to 0.

```
                ORG             100H

                LDAA            #00H            ; load (00)H to accumulator A
                LDAB            #10H            ; load (10)H to accumulator B
COMPARISON:     BEQ             STOP            ; exit from the loop if accumulator B is 0
                ADDA            #2H             ; increment accumulator A by 2
                DECB                            ; decrement counter
                BRA             COMPARISON      ; branch to the beginning of the loop
STOP:           STAA            150H            ; store the number in accumulator A
                SWI                             ; end program
```

Ex: The index register is often used when the program must deal with data in the form of a table. The assembly language program listed in Figure 4.8(a) adds numbers from tables of the augends and addends in Figure 4.8(b) and places the sum in the table of sums to the bottom of this memory map. For instance, the program first adds $01_H + 02_H$, placing the sum 03_H in the "table of sums" to the memory location 0040H. Then it repeats this process by adding $03_H + 04_H$, placing the sum of 07_H in the "table of sums" to the memory location 0041_H, etc. The program in Figure 4.6(a) also has a feature that supports the termination of the program if the sum of the numbers exceeds FF_H (using BCS instruction).

Label	Mnemonic	Operand	Comments
	LDX	#0020H	; Initialize index register at 0020H
LOOP	LDAA	00H,X	; Load augend from first table in memory (X + offset of 00H) into accumulator A
	ADDA	10H,X	; Add addend from second table in memory (X + offset of 10H) into accumulator A
	BCS	STOP	If C flag = 1, then branch forward to STOP (end program if any sum is greater than FFH)
	STAA	20H,X	; Store accumulator A (sum) in third table in memory (X + offset 20H)
	INX		; Increment contents of index reg.
	CPX	#0025	;Compare index register with 0025H (subtract 0025H from contents of index register)
	BNE	LOOP	; If Z flag = 0, then branch back to symbolic address called LOOP
STOP	SWI		; End program

Figure 4.8(a) - Assembly language program

Program memory

Address (hex)	Contents (hex)	
0000		Program
. .	. .	
0011		

Data memory

0020	01	Table of augends (data)
0021	03	
0022	05	
0023	FF	
0024	7F	
0030	02	Table of addends (data)
0031	04	
0032	06	
0033	B	
0034	80	
0040		Table of sums (data)
0041		
0042		
0043		
0044		

Figure 4.8(b) - Memory map

The first instruction in the program listed in Figure 4.8(a) initializes the index register to 0020_H. LDAA 00H, X instruction loads a number from the table of augends in data memory into accumulator A. The first number to be loaded is 01_H from the memory location 0020_H ($0020_H + 00_H = 0020_H$). Note that the instruction in line 2 has a label LOOP and is the target of a backward branch from the BNE LOOP operation towards the bottom of the program.

ADDA 10H,X instruction in line 3 adds the addend in data memory to the augend which is in accumulator A. The addend's memory location is 0030_H ($0020_H + 10_H = 0030_H$).

The fourth instruction (BCS STOP) checks whether the carry flag is set to 1. If C = 1, this indicates that the sum exceeded FF_H and the control is transferred to the end of the program. While C=0, execution continues from line 5. The STAA 20H,X instruction causes the sum in accumulator A to be stored in the "table of sums". In the first pass of the loop, the sum is stored into the memory location 0040_H ($0020_H + 20_H = 0040_H$).

The INX instruction in line 6 increments the contents of the index register. The CPX #0025H instruction in line 7 compares the current contents of the index register with 0025_H to see whether the end of the table of augends is reached or not. The compare instruction is a subtract operation that is used to set or reset the Z flag. The BNE LOOP instruction in line 8 checks Z flag. If Z flag = 0, the branch test is true for the BNE instruction and the program branches back to the symbolic address LOOP in line 2. When the index register reaches 0025_H, the compare operation sets the Z flag to 1, branch test of the BNE instruction becomes false, and the program continues with the next instruction in sequence. This is SWI instruction, which terminates the run.

5. 6802 ASSEMBLY LANGUAGE PROGRAMMING II

5.1 Increment and Decrement Instructions

Increment (INC) and decrement (DEC) instructions allow the contents of a register or memory location to be increased or decreased by 1 respectively.

5.2 Compare Instruction

Consider the problem of testing the accumulator contents, e.g., whether it contains 37_H or not. This can be achieved using substraction and BEQ instructions as shown in the following program.

```
        SUBA    #37H    ; Subtracts the value 37H from the AccumulatorA
        BEQ     PASS    ; If the Zero Flag is set, branch to the label "PASS"
FAIL:   LDAB    #01H    ; Zero flag is not set so place 001H in AccumulatorB
        BRA     STOP    ; Returns to start
PASS:   LDAB    #FFH    ; Zero flag is set so place FFH in Acc B
STOP:   SWI
```

Subtracting 37_H from the accumulator causes the zero flag to be set if the accumulator containes 37_H. Accumulator B is loaded with either FF_H or 01_H, to indicate an accumulator value of 37_H or non-37_H respectively. The difficulty with this technique is that it destroys the contents of the accumulator. Since this is a very common problem in assembly language programming, 6802 provides compare instructions (CMPA, CMPB, CBA, and CPX) which operate like subtraction but do not destroy the register contents.

Compare instructions subtract the contents of the accumulator or index register from the destination and change the condition of flags in CCR according to the result. Contents of the accumulator (or index register) and destination are unaffected by the execution of this instruction.

The above example program can be rewritten using CMPA instruction as follows:

```
        CMPA    #37H
        BEQ     PASS
FAIL:   LDAB    #01H
        BRA     STOP
PASS:   LDAB    #FFH
STOP:   SWI
```

The CMPA instruction subtracts the value 37_H from the Accumulator but does not place the result in the accumulator. It only changes the flags of the CCR.

If the value 37_H is equal to the contents of the Accumulator:	Zero Flag = 1 Carry Flag = 0
If the value 37_H is greater than the contents of the Accumulator:	Zero Flag = 0 Carry Flag = 1
If the value 37_H is less than the contents of the Accumulator:	Zero Flag = 0 Carry Flag = 0

5.3 Logic and bit manipulation Instructions

Logical Operators

Logical instructions (AND, OR, Exclusive OR) can be used to test or change group of bits. AND, OR and Exclusive OR are logical operators:

AND	OR	Exclusive OR
0 AND 0 = 0	0 OR 0 = 0	0 XOR 0 = 0
0 AND 1 = 0	0 OR 1 = 1	0 XOR 1 = 1
1 AND 0 = 0	1 OR 0 = 1	1 XOR 0 = 1
1 AND 1 = 1	1 OR 1 = 1	1 XOR 1 = 0

Example:

```
        0110
        0101
AND
-------------
        0100
```

Notice that any given bit in the result can only be 1 if both of the numbers have a 1 in that position. This property can be used to change specific bits in a register or memory location. In the following example the contents of a register is 99_H. To change the rightmost 4 bits to 0 and keep the other bits unchanged, the AND operation with $F0_H$ can be used.

```
99H =       10011001
F0H =       11110000
        AND
-------------------------
            10010000 = 90H
```

The 6802 AND instructions (ANDA, ANDB) can operate upon memory, register or immediate data:

```
ANDA    0FFH      ; ANDs accumulator A with the contents of
                    address location 00FFH
ANDA    10H,X     ; ANDs accumulator A with the contents of
                    address location (10H + offset of index register)
ANDA    #20H      ; ANDs accumulator A with the value 20H
```

Other logical operations also use the same addressing modes:

```
ANDA   AND with accumulator A
ANDB   AND with accumulator B
ORAA   OR with accumulator B
ORAB   OR with accumulator B
EORA   XOR with accumulator A
EORB   XOR with accumulator B
```

Complement Instruction

COMA, COMB and COM instructions complement the contents of the specified accumulator or a memory location. Complement instructions offer indexed and extended addressing modes.

COMA or COMB instruction complements the contents of the specified accumulator. No other status bit or register contents are affected. If accumulator B contains $3A_H$ (00111010_2), after the **COMB** instruction is executed, accumulator B contains $C5_H$ (11000101_2).

Complement instruction can also be used to complement the contents of the specified memory location. If the contents of the index register are 0100_H and contents of the memory location 0113_H is 23_H (00100011_2), after **COM 13H,X** instruction is executed, the memory location 0113_H contains DC_H (11011100_2).

The Bit Test Instruction

The Bit Test instruction (BIT) is similar to the logical Compare. The contents of the accumulator or memory location are ANDed with a mask. However, neither the accumulator nor the destination is modified by this instruction; only the Flags are affected.

BITA #07H tests the bits 1,2 and 3 of accumulator A and sets the zero flag if the condition is true

Example: Following program examines the byte at location 120_H. If bit 1 of location 120_H is set, 55_H is stored in location 130_H, otherwise program terminates.

```
        ORG     100H
        LDAA    120H    ; load byte
        BITA    #01H    ; is bit 1 set?
        BEQ     STOP    ; if not set, end program
        LDAB    #55H    ; store 55H in memory location 130H
        STAB    130H
STOP:   SWI             ; end program
```

5.4 Arithmetic Operations

5.4.1 Addition Instructions

Add Accumulators

ABA instruction adds the contents of accumulator B to the contents of accumulator A and stores the result in accumulator A. If accumulator A contains $B4_H$ and accumulator B contains $2D_H$, after the ABA instruction is executed accumulator A contains $E1_H$.

Add Memory to Accumulator

ADDA, ADDB instructions add the contents of a memory location to accumulator A or B respectively without considering the carry status. The same memory addressing options as ADC instruction are supported.

Ex (8-bit addition): The following program adds the contents of memory locations 0040_H and 0041_H, and place the result in the memory location 0042_H.

```
        ORG     0H

        LDAA    40H
        ADDA    41H
        STAA    42H
        SWI             ; end program
```

Add Memory, with carry, to Accumulator

ADCA or ADCB instructions add the contents of a memory location to accumulator A or B respectively. 4 addressing modes are supported:

1. Immediate
2. Direct
3. Extended
4. Indexed

Addition with carry using Immediate Data

This type of instruction adds the immediate data with the carry bit to accumulator A. If accumulator A contains $3A_H$, the carry bit is 1, after the instruction **ADCA #7CH** is executed, the accumulator A contains $B7_H$.

Addition with carry using Direct Memory Addressing.

This type of instruction adds the contents of a specified direct memory address and the carry bit to accumulator B. If accumulator B contains $3A_H$ and memory address $1F_H$ contains $7C_H$ and carry bit contains 1. After the instruction **ADCB 1FH** is executed, accumulator B contains $B7_H$.

Addition with carry using Extended Addressing

This type of instruction is similar to the addition with carry using direct addressing. Only difference is that **ADCA $3FF2_H$** instruction allows extended addressing.

Addition with carry using Indexed Addressing

This type of instruction adds the carry bit and the contents of a memory location addressed by the sum of index register and the first operand of ADCA instruction to accumulator A. If accumulator A contains $3A_H$, Index register contains $50D_H$, memory address 523_H contains 76_H, and the carry bit is 1, After the instruction **ADCA 16H,X** is executed, accumulator A contains $B1_H$.

Ex (16-bit addition): Following program adds 16-bit number in memory locations 0040_H and 0041_H to the 16-bit number in memory locations 0042_H and 0043_H. The most significant eight bits are in memory locations 0040_H and 0042_H. Then the result is stored into memory locations 0044_H and 0045_H, where the most significant bits are in 0044_H.

```
ORG     0H

LDAA    41H
ADDA    43H     , add least significant bits
STAA    45H
LDAA    40H
ADCA    42H     ; add most significant bits with carry
STAA    44H
SWI             ; end program
```

ADCA 42H adds the contents of accumulator A and the memory location 0042, plus the contents of Carry (C) bit. The carry from the addition of the least significant eight bits is thus included in the addition of the most significant eight bits.

5.4.2 Subtraction

Subtract Memory from Accumulator

SUBA, SUBB instructions subtract the contents of the selected memory byte from the contents of accumulator A ~~or~~ and B respectively. The same addressing modes of ADC instruction are supported. If the memory address 0031_H contains $A0_H$ and accumulator B contains $E3_H$, after the **SUBB 31H** instruction is executed accumulator B contains 43_H.

5.4.3 Shift Operations

Logical Shift Operations

LSRA, LSRB, LSR instructions perform a one-bit logical right shift on accumulator A, B ~~or~~ and a specified memory location respectively. The least significant bit is shifted into the carry bit in CCR and 0 is inserted as a most significant bit. If accumulator B contains $7A_H$ (01111010_2), after LSRB instruction is executed, accumulator B contains $3D_H$ (00111101_2) and the carry status bit is set to 0.

LSR instruction shifts the contents of the specified memory location towards right 1-bit. Indexed and extended addressing modes are available for LSR instruction. If the contents of the memory location $04FA_H$ is $0D_H$ (00001101_2), after **LSR 04FAH** is executed, the carry bit is 1 and the contents of location $04FA_H$ is 06_H (00000110_2).

Ex: Following program separates the contents of memory location 0040_H into two 4-bit numbers and stores them in memory locations 0041_H and 0042_H. It places the four consequitive most significant bits of memory location 0040_H into the four least significant bit positions of memory location 0041_H; and the four least significant bit positions of memory location 0040_H into the four least significant bit positions of memory location 0042_H

```
LDAA    40H     ; load data
ANDA    #0FH    ; mask off four MSBs
STAA    42H     ; store at address 0042H
LDAA    40H     ; reload data
LSRA            ; shift accumulator to right 4 bits, clearing the most significant bits.
LSRA            ;
LSRA            ;
LSRA            ;
STAA    41H     ; store at address 0041H
SWI             ; end program
```

Arithmetic Shift Operations

ASLA or ASLB instructions perform a one-bit arithmetic left shift on the contents of accumulator A or B. If accumulator A contains $7A_H$ (01111010_2), after ASLA is executed $F4_H$ (11110100_2) is stored in accumulator A, carry bit is set to 0, sign bit is set 1 (as the leftmost bit is 1) and Zero bit is set to 0.

ASL instruction performs a 1-bit arithmetic left shift on the contents of a memory location. The extended and indexed addressing modes are supported. If the Index register contains $3F3C_H$, and the memory address $3F86_H$ contains CB_H, after the **ASL 4AH,X** instruction is executed, the memory address $3F86_H$ contains 96_H and Carry flag is set to 1. The ASL instruction is often used in multiplication routines. Note that execution of a single ASL instruction results with its operand multiplied by a factor of 2.

ASR instruction performs a one-bit arithmetic right shift on the contents of accumulator A or B or the contents of a selected memory byte. ASR is frequently used in division routines.

Rotate Operations

ROLA or ROLB instructions rotate the specified accumulator or a selected memory byte one bit towards left through the carry bit. ROLA or ROLB instructions rotate contents of the specified accumulator and the carry bit as a block towards left one bit. If accumulator A contains $7A_H$ (01111010_2) and the carry bit is 1, after **ROLA** instruction is executed, accumulator A contains $F5_H$ (11110101_H) and the carry bit is reset to 0.

ROL instruction rotates the contents of the specified memory location one bit to the left through the carry. Indexed and Extended addressing modes can be used with ROL instruction.

Example: If the contents of memory location 1403_H is $2E_H$ (00101110_2) and the Carry bit is 0, after **ROL 1403H** is executed, memory location 1403_H contains $5C_H$ (01011100_2).

RORA, RORB or ROR instructions rotate the specified accumulator or contents of a selected memory location and the carry bit as a block one bit towards right. These instructions operate similarly with the ROL instruction.

Ex (8-bit binary multiplication): Following program multiplies an 8-bit unsigned number in memory location 0041_H by another 8-bit unsigned number in the memory location 0040_H and places the most significant bits of the result in memory location 0042_H and eight least significant bits in memory location 0043_H.

Multiplying a number by zero results with zero, multiplying by one results with the number itself. Therefore the multiplication can be reduced to the following operation: If the current bit is 1, add the multiplicand to the partial product.

```
        CLRA        ; product MSB = Zero
        CLRB        ; product LSB = Zero
        LDX   #8    ; load number of bits of the multiplier to index register
SHIFT:  ASLB        ; shift product left 1 bit
        ROLA
        ASL   40H   ; shift multiplier left to examine next bit
        BCC   DECR  ;
        ADDB  41H   ; add multiplicand to the product if carry is 1
        ADCA  #0    ;
DECR:   DEX
        BNE   SHIFT ; repeat until index register is 0
        STAA  42H   ; store result
        STAB  43H
        SWI         ; end program
```

The following operations are performed to ensure that everything is lined up correctly every time:

1) Shift multiplier left one bit so that the bit to be examined is placed in the Carry.
2) Shift product left one bit so that the next addition is lined up correctly.

The complete process for binary multiplication is as follows:

Step 1 - Initialization

The Index register is used as a counter. CLRA and CLRB set the product and carry bit to 0.

Product=0
Counter=8

Step 2 - Shift Product to left so as to line up properly.

The instructions ASLB and ROLA together act as a 16-bit arithmetic left shift of the product in accumulators A and B (MSBs in A).

Step 3 - Shift Multiplier to left so the next bit goes to Carry to multiply.

The instruction ASL 40H shifts the contents of memory location 0040H left one bit, placing the most significant bit in the Carry and clearing the least significant bit.

Step 4- Add Multiplicand to Product if carry is 1

The instruction ADDB 41H adds the multiplicand to the product. The instruction ADCA #0 adds the carry flag from that 8-bit addition to the most significant eight bits of the product (in accumulator A).

If Carry=1, Product = Product + Multiplicand

Step 5 - Decrement counter and check for zero

Counter = Counter - 1
If Counter $\neq$ 0, go to Step 2

If multiplier is 61_H and the multiplicand is $6F_H$, the process works as follows:

Initialisation:

Product	0000
Multiplier	61
Multiplicand	6F
Counter	08

After first iteration of steps:

Product	0000
Multiplier	C2
Multiplicand	6F
Counter	07
Carry from Multiplier	0

After second iteration:

Product	006F
Multiplier	84
Multiplicand	6F
Counter	06
Carry from Multiplier	1

After third iteration:

Product	014D
Multiplier	08
Multiplicand	6F
Counter	05
Carry from Multiplier	1

Goes like this until counter is equal to 0.

Ex (8-bit binary division): Following program divides an 8-bit unsigned number in memory location 0040_H by another 8-bit unsigned number in the memory location 0041_H and stores the quotient in memory location 0042_H and the remainder in memory location 0043_H. Initially accumulator A is cleared and dividend is loaded into accumulator B. During the division, the quotient replaces the dividend in accumulator B as the dividend is shifted left to accumulator A. At the end, remainder is found in accumulator A and quotient in accumulator B.

```
            LDX   #8          ; load number of bits of the divisor into index register
            CLRA
            LDAB  40H         ; load dividend to accumulator B
DIVIDE:     ASLB              ; shift left dividend-quotient.
            ROLA
            CMPA  41H         ; compare accumulator A and divisor
            BCS   CHKCNT      ; branch if accumulator A is smaller than divisor
            SUBA  41H         ; subtract divisor from accumulator A
            INCB              ; increment quotient
CHKCNT:     DEX               ; decrement counter
            BNE   DIVIDE
            STAB  42H         ; store quotient
            STAA  43H         ; store remainder
            SWI               ; end program
```

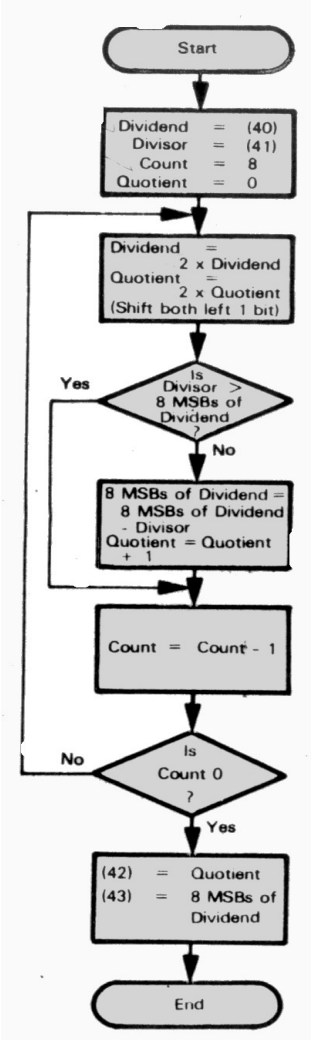

Figure 5.1 - Flowchart of division program

6. 6802 ASSEMBLY LANGUAGE PROGRAMMING III

6.1 STACK OPERATIONS

A “stack” is simply an area of memory where a list of data items is stored consecutively. It consists of any number of locations in RAM memory. The restriction in the list of the elements is that, the elements can be added or removed at one end of the list only. This end is usually called “top of stack” and the structure is sometimes referred to as a “push-down” stack. This type of storage mechanism is Last-In-First-Out “LIFO”; the last data item placed on the stack, is the first one removed when retrival begins.

PUSH operation places a new item on the stack.
PULL operation removes the top item from the stack.

MC6802 microprocessor allows a porion of memory to be used as a stack. The microprocessing unit has a 16-bit stack pointer (Figure 6.1). When a byte of information is stored in the stack, it is stored at the address which is contained in the stack pointer. The stack pointer is decremented (by one) immediately following the storage of each byte of information in the stack. Conversely, the stack pointer is incremented (by one) immediately before retrieving each byte of information from the stack, and the byte is then obtained from the address contained in the stack pointer. **The programmer must ensure that the stack pointer is initialized to the required address before the first execution of an instruction which manipulates the stack. Stack pointer can be initialized to use any portion of read-write memory, usually to the highest address of RAM (Figure 6.2).**

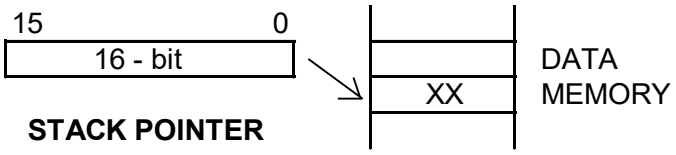

Figure 6.1 - Stack Pointer

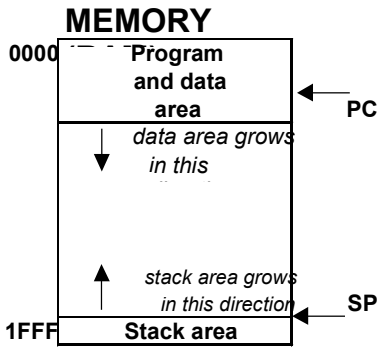

Figure 6.2 - Memory usage

Stack operations are used:

- while executing subroutines
- while handling interrupts
- while doing arithmetic operations

6.1.1 PUSH Operation

PSH instruction is used for storing a single byte of data in the stack. This instruction addresses either accumulator A or accumulator B (PSHA and PSHB respectively). The contents of the specified accumulator are stored in the stack. The address contained in the stack pointer is decremented. If accumulator A contains $3A_H$ and the stack pointer contains $1FFF_H$. After the instruction **PSHA** is executed $3A_H$ is stored into the location $1FFF_H$ and the stack pointer is altered to $1FFE_H$, the PC is 0011_H (Figure 6.3).

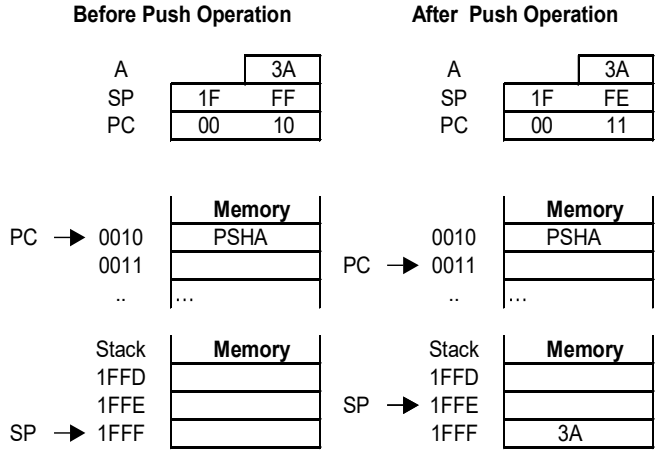

Figure 6.3 - Push (PSHA) instruction

6.1.2 PULL Operation

PUL instruction retrives data from the stack. This instruction addresses either accumulator A or accumulator B (PULA and PULB respectively). The address contained in the stack pointer is incremented. A single byte of data is then obtained from the stack and is loaded into the specified accumulator. If the stack pointer contains $1FFE_H$ and location $1FFF_H$ contains CE_H. After the instruction **PULB** is executed, accumulator B contains CE_H and the stack pointer contains $1FFF_H$ (Figure 6.4).

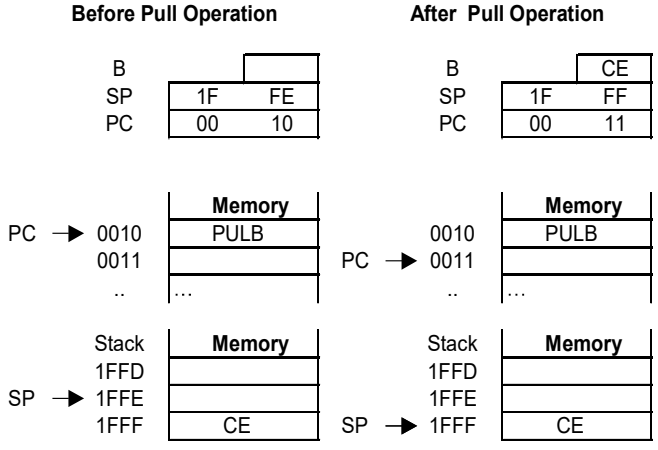

Figure 6.4 – Pull (PULB) instruction

6.1.3 Other Stack Operations

The address stored in the stack pointer is affected by the execution of the instructions PSH, PUL, SWI, WAI, RTI, BSR, JSR, and RTS, and also by the servicing of a non-maskable interrupt or an interrupt request from a peripheral device.

The address in the stack pointer may also be changed without storing or retrieving information in the stack. This is carried out by the following instructions:

DES	decrement stack pointer
INS	increment stack pointer
LDS	load the stack pointer
TXS	transfer index register to stack pointer

The contents of the stack pointer is also involved in the execution of the following instructions:

STS	store the stack pointer
TSX	transfer stack pointer to index register

STS instruction stores the contents of the stack pointer into two contiguous memory locations. This instruction offers direct, indexed and extended addressing modes. The **STS** instruction stores the high byte of the stack pointer into the specified memory address and the low byte into the memory address immediately following it. In figure 6.5, the contents of the stack pointer is $1FFF_H$ and the memory address to store the stack pointer is given as 0080_H in the **STS #80H** instruction. After **STS** instruction is executed, memory location 0080_H contains the high byte of stack pointer ($1F_H$) and memory location 0081_H contains the low byte of stack pointer (FF_H).

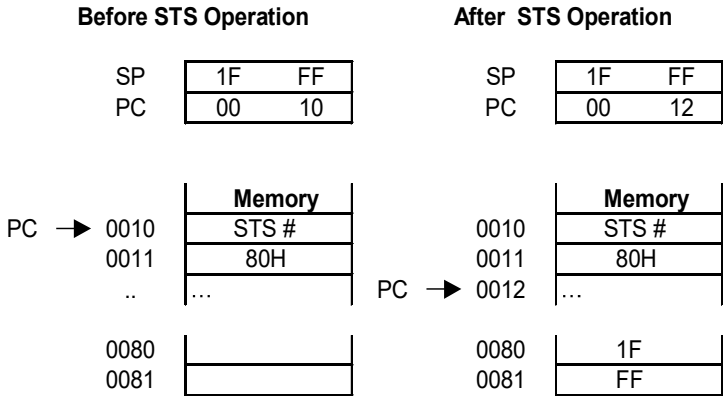

Figure 6.5 - Store Stack Pointer (STS) instruction

TSX instruction moves the contents of the stack pointer plus one to the index register so that the index register points directly to the bottom of the stack. In Figure 6.6, SP is $1FFE_H$. After the execution of **TSX** instruction, index register contains $1FFF_H$. The MC6802 employs a decrement after write, increment before read stack implementation scheme.

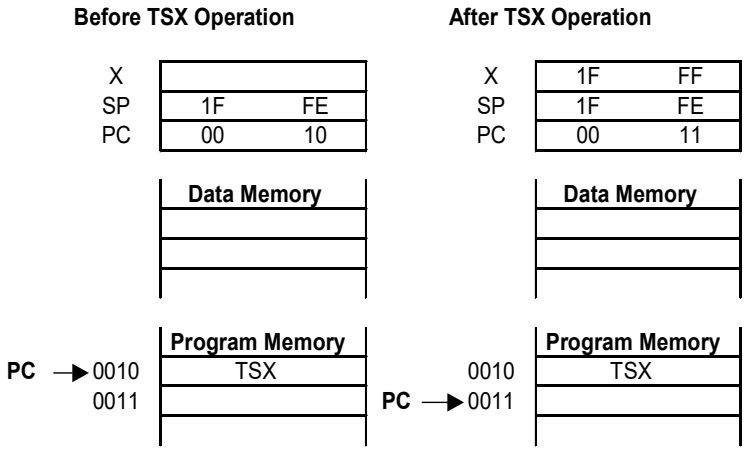

Figure 6.6 - TSX Instruction

TXS instruction moves the contents of the index register minus 1 to the stack pointer. In Figure 6.7, contents of the index register is $1FFF_H$. After **TXS** is executed, stack pointer contains $1FFE_H$.

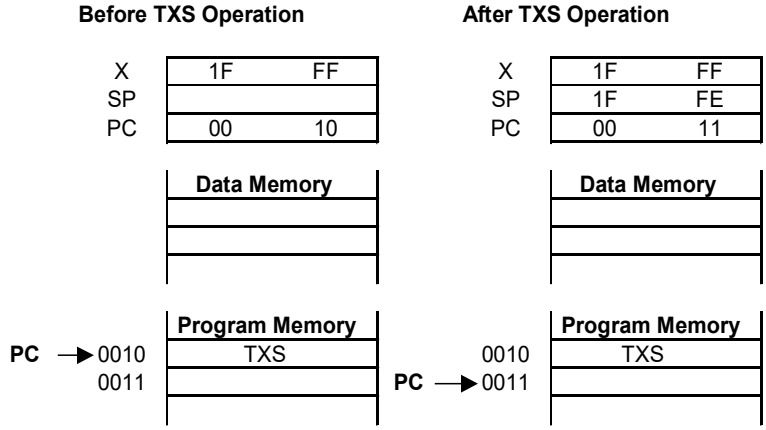

Figure 6.7 - Move From Index Register to Stack Pointer (TXS) instruction

Example: In the following program, the contents of the accumulators are stored in the stack before branching to a routine and at the end of routine, the original values of the accumulator are restored from the stack.

```
ORG     100H

LDAA    #03H
LDAB    #04H
LDS     #1FFFH

PSHA
PSHB
BRA     LDRT
```

```
CONT:   PULA
        PULB
        ABA
        STAA   61H
        SWI

LDRT:   LDAA   #01H
        LDAB   #02H
        ABA
        STAA   60H
        BRA    CONT
```

At the beginning of the program accumulator A and B is loaded with 03_H and 04_H and stack pointer is initialized to point the end of memory ($1FFF_H$). Then, before branching to the routine LDRT these numbers are stored in the stack using PSH instructions. In LDRT routine, 01_H and 02_H are added using ABA instruction and (03_H) is stored in memory location 0060_H and CONT is branched. Then the original values of the accumulators are loaded from the stack using PUL instructions, and the result (07_H) is stored in 0061_H and the program ends with the execution of SWI instruction.

6.2 SUBROUTINES

In a given program it is often necessary to perform a particular task a number of times on the same or on different data values (such as subroutines to obtain time delay, subroutine to sort a list of values, etc.) including the block of instructions to perform the task at every place where it is needed in the program causes a waste of memory. Instead, it is better to place only one copy of this block of machine instructions in the main memory as a subroutine, and in the program branch to the beginning of the subroutine whenever required.

In figure 6.8, the execution of a jump to subroutine instruction **(JSR)** and return from subroutine instruction **(RTS)** are given. After **JSR 0100H** instruction is executed, PC is set to the defined address and execution continues with the **ABA** instruction. The program returns from this subroutine after the **RTS** instruction is executed. Program counter (PC) is set to 0023_H and the program execution continues with **ANDA #7FH** instruction.

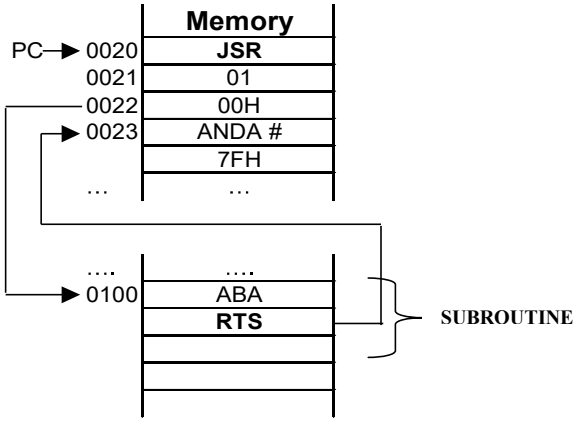

Figure 6.8 - A subroutine execution process

6.2.1 Jump to a subroutine

JSR instruction uses extended and indexed addressing modes. After **JSR** instruction is executed, program counter is decremented by 3 (if extended addressing is used) or 2 (if indexed addressing is used), and then is pushed onto the stack. The stack pointer is adjusted to point to the next empty location in the stack. The specified memory address is then loaded into the program counter and execution continues with the first instruction in the subroutine.

Using Extended Addressing

Execution of JSR instruction with extended addressing mode is shown in Figure 6.9. After **JSR 0100H** instruction is executed, address of the next instruction (**AND #7FH** instruction) is stored on top of the stack, and SP is decremented by 2. Address of the first instruction in the subroutine (0100_H) is stored into the program counter, and the program continues from this point. ABA instruction is the next instruction to be executed.

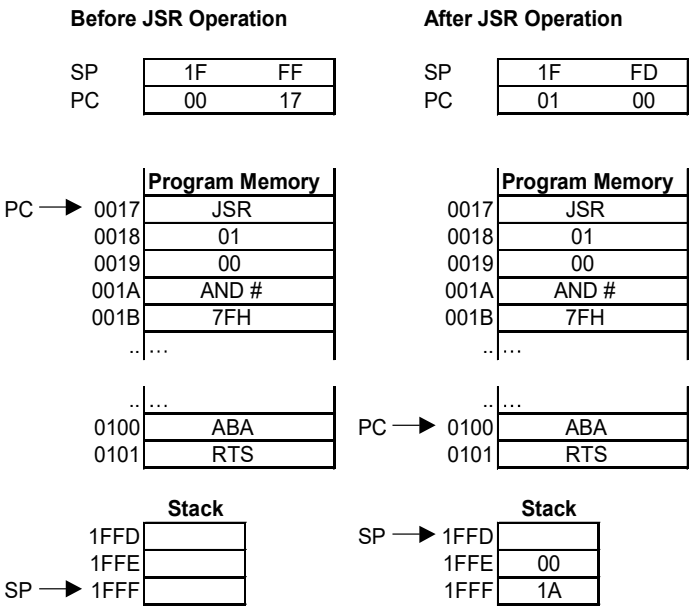

Figure 6.9 - Jump to subroutine (JSR) instruction, using extended addressing

Using Indexed Addressing

Execution of JSR instruction with indexed addressing mode is shown in Figure 6.10. After **JSR 30H,X** instruction is executed, address of the next instruction (0019_H) is stored on top of the stack and SP is decremented by 2. Jump address is calculated as the value of the index register plus the address part of the instruction ($1100_H + 30_H = 1130_H$) and it is stored in PC. The program continues with the ABA instruction in the subroutine.

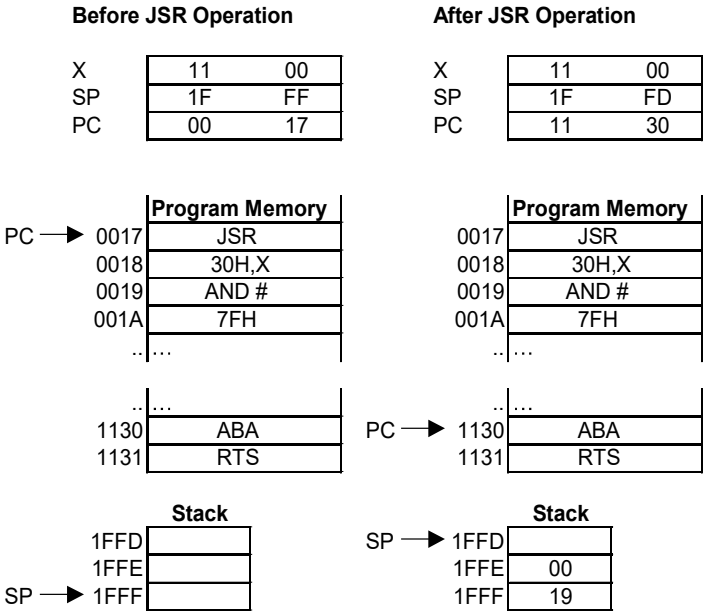

Figure 6.10 - Jump to subroutine (JSR) instruction, using indexed addressing

6.2.2 Return from a subroutine

RTS instruction moves the contents of the top two stack bytes (which is the address of the next instruction after JSR) to program counter. These two bytes provide the address of the next instruction to be executed. Previous program counter contents are lost. RTS instruction also increments the stack pointer by 2 (Fig 6.11). Every subroutine must contain at least one Return instruction; which is the last instruction to be executed in the subroutine.

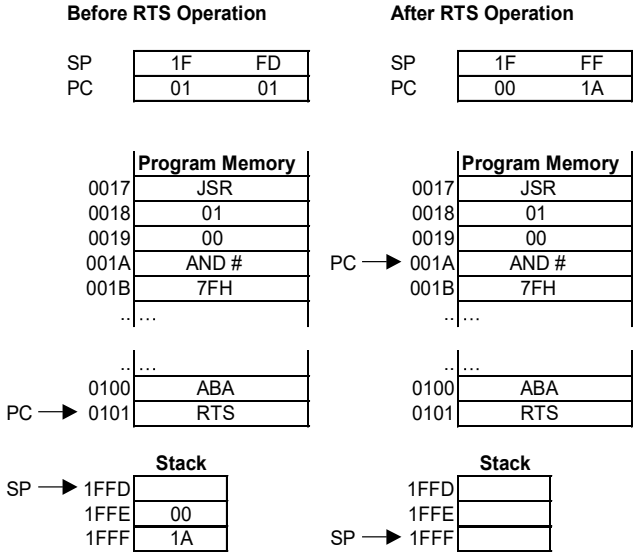

Figure 6.11 - Return from subroutine (RTS) instruction

Example:

Determine the lengths of two strings of ASCII characters. Starting addresses of the strings are 43_H and 63_H. End of a string is marked by a carriage return character ($0D_H$). Write a subroutine which calculates the length of a string (excluding carriage return character) and places it in accumulator B. Also write a calling program which calls this subroutine to calculate the lenghts of two strings and stores the results (in accumulator B) in memory addresses 42H and 62H.

Calling Program:

```
        ORG     0H
        LDS     #500H       ; Start stack at location 500H
        LDX     #43H        ; Get starting address of the first string
        JSR     STLEN       ; Determine string length
        STAB    42H         ; Store string length
        LDX     #63H        ; Get starting address of second string
        JSR     STLEN       ; Determine string length
        STAB    62H         ; Store string length
        SWI
```

Subroutine:

```
        ORG     100H
STLEN:  CLRB                ; String length = 0
        LDAA    #0DH        ; Get 'CR' for comparison
CHKCR:  CMPA    X           ; Is character 'CR'?
        BEQ     DONE        ; Yes, end of string
        INCB                ; No, add 1 to string length
        INCX
        BRA     CHKCR
DONE:   RTS
```

The calling program initializes the stack pointer to 500_H and then performs the following steps for each string:

1. places the starting address of the string in the Index register
2. calls the subroutine
3. stores the result in accumulator B to memory.

The stack pointer must be initialized to an appropriate area in memory (in this example 500_H) so that the stack does not use the addresses in the program area.

The subroutine determines the length of a string of ASCII characters and places the length in accumulator B. Starting address of the string is a parameter to the subroutine. It is placed in the index register before the subroutine is jumped. The result is returned in accumulator B.

If the first string is only a carriage return character and index register (X) contains 0043_H, after the first call of the subroutine, contents of the registers and memory are as follows:

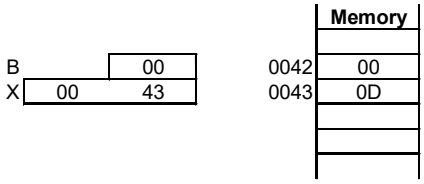

Figure 6.12 - Memory allocated for the first string

If the second string is 'RATHER' and index register contains 0063_H, after the second call of the subroutine, contents of the registers and memory are as follows:

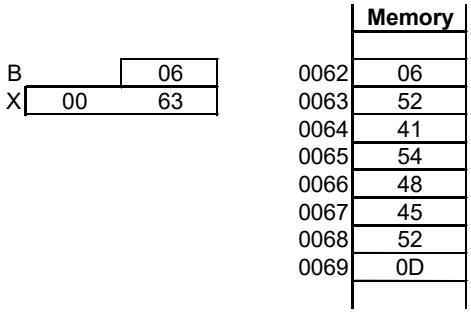

Figure 6.13 - Memory allocated for the second string

5.2.3 Subroutine Nesting

A subroutine may jump to another subroutine as shown in Figure 6.14. This situation can be extended to the case where the second subroutine jumps to a third subroutine and so on. This is called *subroutine nesting,* and can be carried out to any depth.

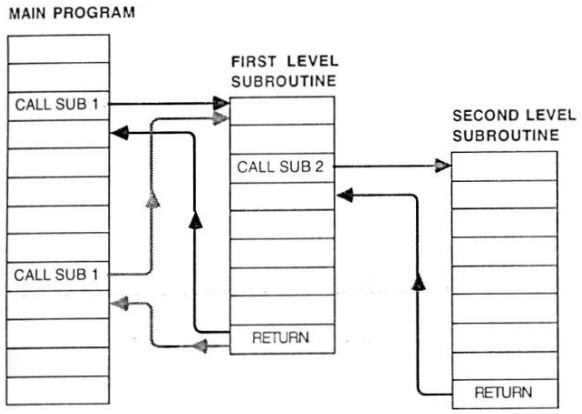

Figure 6.14 - Subroutine nesting

Example : The following program uses subroutine nesting to write the number FF_H into a memory block. Starting address of the memory block is 00_H and the end of the block is determined by the number in accumulator B. Subroutine FILL, in a loop, stores 00_H to the memory location pointed by the index register (X) and calls subroutine DECR to decrement accumulator B.

```
        ORG     100H
        LDS     #1FFFH      ; initialize stating address of stack area
        LDX     #00H        ; initialize index register
        LDAA    #0FFH       ; keep FFH in accumulator A
        LDAB    #25H        ; length of the memory block is 25H
        JSR     FILL        ; start filling
        SWI

        ORG     500H        ;
FILL:   STAA    0H,X        ; store FFH in the memory address pointed by X
        INX                 ; increment X to point to the next location
        JSR     DECR        ; jump to DECR subroutine
        CMPB    #0H         ; Is the end of  memory block reached?
        BNE     FILL        ; No, continue with the filling procedure
        RTS                 ; Yes, end of subroutine

        ORG     1000FH      ;
DECR:   DECB                ; decrement accumulator B
        RTS                 ; return to FILL subroutine
```

5.3 MACROS

In source programs, particular sequences of instructions may occur many times. Programmer can avoid repeatedly writing out the same instruction sequence by using a macro.

Macros allow the programmer to assign a name to an instruction sequence. This macro name is then used in the source program in the place of the related instruction sequence. Before assembling macroprocessor replaces the macro name with the appropriate sequence of instructions.

Macros are not the same as subroutines. A subroutine occurs in a program, and program execution branches to the subroutine. A macro is expanded to an actual instruction sequence each time the macro occurs; thus a macro does not cause any branching. Figure 6.15 shows the source program with macro calls and the object program after the macro expansion.

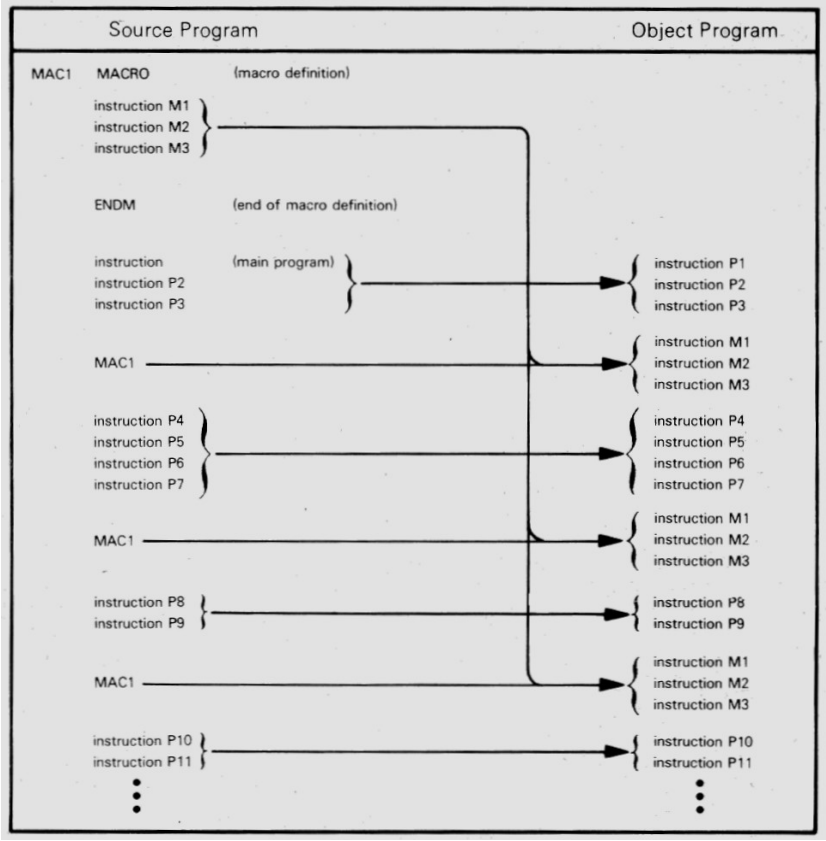

Figure 6.15 - A macro call example

5.3.1 Macros in Cross-32 Cross Assembler

MACRO and ENDM directives define the sequence of instructions in the source file as macro. Format of a macro definition is as follows:

```
Label:  MACRO       exp(1), exp(2), … exp (n)       ; comment

            line 1
            line 2
        .
        .
        ENDM
```

Upon encountering a MACRO directive, Cross-32 stores the source code between the MACRO directive and the next ENDM directive, assigning the label on the MACRO line to it. Although the code within the macro definition is checked for syntax errors, the resulting machine code is not written to either the list or hexadecimal files. When the macro's label is found as a macro call later in the assembly source code, the entire MACRO is expanded at this location. Any expressions appearing after the macro definition are replaced by those appearing after the macro call in the expanded code. These are character by character replacements, so ensure that the expressions in the macro definition are truly unique. **The number of expressions in the macro definition must be equal to the number of expressions in the macro call**. Nested macros are not permitted.

Ex: Following program is an example for macro definitions and macro calls.

```
            CPU         "6801.TBL"
            HOF         "MOT8"

ADDITION:   MACRO       Z,Y     ; macro definition steps
            LDAA        #Z      ;
            ADDA        #Y      ;
            ENDM                ; macro ends

            ORG         0H      ; begin program

            ADDITION    10H,20  ; macro Call
            STAA        1000H   ; store result in memory location (1000)H
            ADDITION    12H,15  ; macro Call
            STAA        1001H   ; store result in memory location (1001)H
            SWI                 ; End Program
            END                 ;
```

Examine the list file of the program to see that the Macro calls are replaced by the Macro definition.

IF, ELSE, and ENDI - Conditional Assembly

Cross-32 supports conditionaly assembly using the IF, ELSE and ENDI directives. This feature is usually used to re-configure a single assembly language program for different hardware environments.

Conditional assembly has the following syntax:

```
IF      expression      ; comment
        line 1
        line 2
        .
        .
        line n
ELSE                    ; comment
        line 1
        line 2
        .
        .
        line n
ENDI                    ; comment
```

Upon encountering an IF statement Cross-32 evaluates the single expression following it. All labels used in this expression must be defined previous to the IF. If the expression evaluates to zero, the statements between the IF and either an ELSE or an ENDI are not assembled. If the expression results in a non-zero value, the statements between the IF and either an ELSE or an ENDI are assembled. ELSE is an optional directive, allowing only one of the two sections of the source file within the IF block to be assembled. All conditional blocks must have an IF directive and an ENDI directive, the ELSE directive being optional. If blocks may be nested 16 deep before a fatal error occurs.

An example of conditionally assembly follows, where a microprocessor type is selected.

```
                CPU     "6801.TBL"
                HOF     "MOT8"
OPTION:         EQU     1

                IF OPTION=1                     ; If 6800 selected
BRAVAR:         EQU     100H                    ;
                ENDI
                IF OPTION=2                     ; If 6802 selected
BRAVAR:         EQU     110H                    ;
                ENDI

                ORG     70H
                BRA     BRAVAR
CONT:           STAA    85H
                SWI

                ORG     100H
                LDAA    #02H
                BRA     CONT
                ORG     110H
                LDAA    #03H
                BRA     CONT
                END
```

In this example Option 1 is selected so BRAVAR variable is set to 100H. Accumulator A is loaded with 02_H and then this value is stored in memory location 85_H. If Option 2 is selected then BRAVAR variable is set to 110_H which means that accumulator A is to be loaded with 03_H and this value is going to be stored in memory location 85_H.

7. MICROPROCESSOR INPUT/OUPUT TECHNIQUES

7.1 MONITOR SUBROUTINES

The DT6802 Microprocessor Training Kit Monitor Program has subroutines dealing with the keypad and the seven segment display. These subroutines accept an input from the keypad, and convert the output data into a suitable format that can be displayed on the seven segment display. DT6802 Monitor subroutines and their starting addresses are shown in Table 7.1.

Memory Address	*Monitor Subroutine*
DFA0H	CLEARD
DFA3H	PATCON
DFA6H	DISPAT
DFA9H	RKEYC

Table 7.1 - Monitor Subroutines for I/O

Label : **CLEARD**
Memory Address : $DFA0_H$
Function : Clears the contents of the seven segment displays.
Input Parameter : None
Output Parameter : None
Example : JSR 0DFA0H ; clears 7-segment display

Label : **PATCON**
Memory Address : 0DFA3H
Function : Converts the output into a binary number which can be displayed on the seven segment display. This subroutine sets the bits, which light the segments of the seven-segment display to 1.
Input Parameter : The data to be displayed must be loaded into accumulator A
Output Parameter : The converted binary number is in accumulator A
Example :
```
LDAA    #0H
JSR     0DFA3H
```

In this example, accumulator A is loaded with “0” and then the program jumps to the subroutine at addressed 0DFA3H. After the subroutine is executed, accumulator A contains 03FH which is the code to light the segments of the display for displaying character “0”.

Label : **DISPAT**
Memory Address : 0DFA6H
Function : Selects a seven segment display for displaying a character
Input Parameter : The converted data must be in accumulator A, and code of the selected 7-segment display must be in accumulator B.
Output Parameter : None
Example :
```
LDAA    #0H
JSR     0DFA0H
LDAB    #01H
JSR     0DFA6H
```

In this example, character 0 is displayed on the first seven segment display. Note that before jumping to DISPAT subroutine, the data must have been converted using PATCON subroutine and the code of the selected seven segment display must exist in accumulator B. Codes for selecting the seven segment displays are given in Table 7.2 .

Seven segment display	Code
1	01H
2	02H
3	04H
4	08H
5	10H
6	20H
7	40H
8	80H

Table 7.2 - Codes for the seven segment displays

Label : **RKEYC**
Memory Address : 0DFA9H
Function : Waits for an input from the keypad. When a key is pressed, the predetermined value of this key is loaded to accumulator A.
Input Parameter : None
Output Parameter : Value of the key pressed is in accumulator A
Example : JSR 0DFA9H
ANDA #3FH

When a key is pressed, its predetermined value is loaded to accumulator A (Table 7.3). Then accumulator A is ANDed with 3FH. The keys and their predetermined values are given in Table 7.3.

Key	Predetermined value	Key	Predetermined value
0	00H	A	0AH
1	01H	B	0BH
2	02H	C	0CH
3	03H	D	0DH
4	04H	E	0EH
5	05H	F	0FH
6	06H	←	10H
7	07H	→	11H
8	08H	↑	12H
9	09H	↓	13H

Table 7.3 - Predetermined values of keypad keys

7.2 Basic Input/Output

Figure 7.1 shows the MPU bus and control structure where processor's inputs and outputs in four functional categories; data, adress, control and supervisory. MC6802 Microporcessor has an 8-bit bidirectional bus to faciliate data flow throught the system(Data bus). Address Bus does not only specify memory addresses, but also it is a tool to specify I/O devices. By means of its connections to Data Bus, Control Bus, and selected address lines, the I/O interface is allocated as an area of memory. User may converse with I/O using any of the memory interface reference instructions, selecting the desired peripheral with memory address. Control Bus is provided for the memory and interface devices. It consists of a heterogenous mix of signals to regulate system operation. MPU supervisory, is used for timing and control of the MC6802 itself. Three signals are shared with the control bus and affect the memory and I/O device as well.

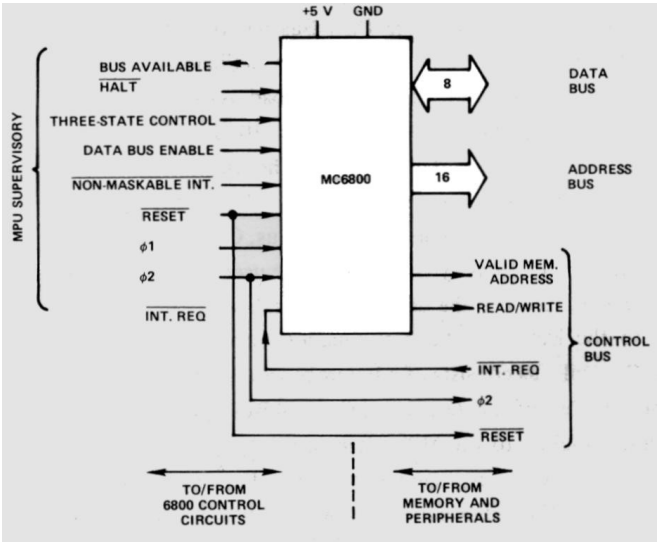

Figure 7.1 - MC6802 Bus and Control Signals

Every peripheral device is assigned a block of memory addresses (These blocks do not intersect). During instruction decoding, the address decoding circuit of the microprocessor enables only one peripheral device and other devices are set to high impedance mode.

Inputs of the address decoding circuit are connected to the address bus and necessary control outputs (such as R\W, E, …) and outputs of this circuit are connected to chip select lines (CS) of each peripheral device. 74ALS27 (NOR Gate) and 74LS30 (NAND Gate) in Figure 7.2 are used for address decoding circuit (chip select). 74LS125 (Tri-state buffer) and toggle switches are used for input.

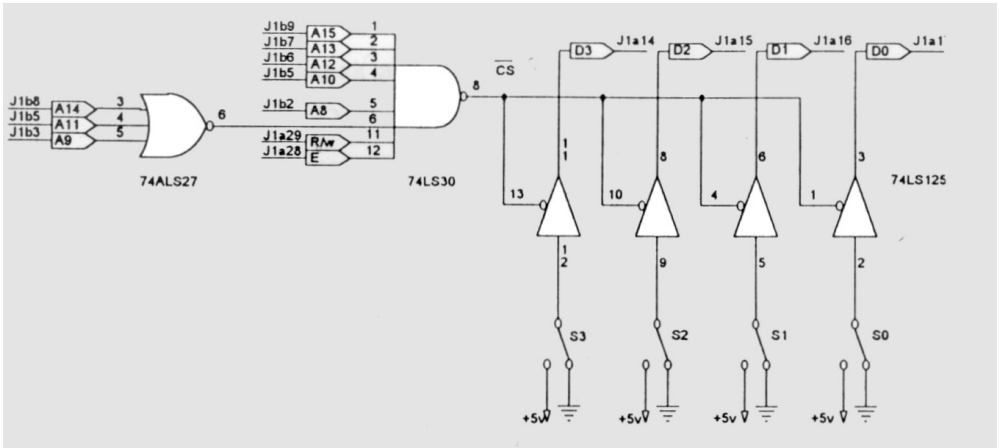

Figure 7.2 – Input to a microprocesor unit

In Figure 7.2 $\overline{CS}$ output is reset to Logic "0" only when:

A15=A13=A12=A10=A8=R\W=E= **Logic "1"** and A14=A11=A9= **Logic "0"**

where A is the address line, R\W is the Read\Write output , E is the Enable output.

That is:

A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0
1	0	1	1	0	1	0	1	X	X	X	X	X	X	X	X
	⇩				⇩				⇩				⇩		
	B				5				X				X		

When the address output of the microprocessor is between $B500_H$ and $B5FF_H$, and R/W is "Read", and E is 1, 74LS125 tri-state buffer (whose state diagram and truth table is shown in Figure 7.3) is enabled, which means that the inputs from the toggle switches are read and the data is sent to MPU using data lines. Otherwise the buffer is in high impedance mode.

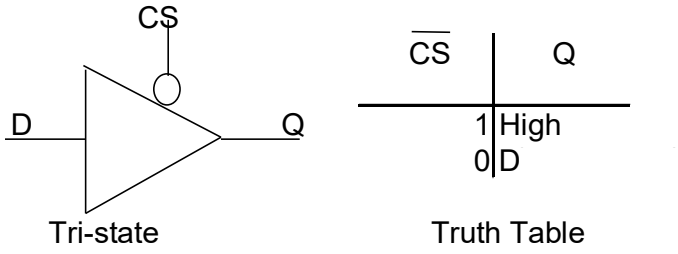

Figure 7.3 - Tri-state buffer and its state diagram

Example: The following program applies an input to the microprocessor unit:

```
        ORG     100H
        LDAB    B500    ; read toggle switches
        ANDB    #0FH    ; mask four MSBs
        STAB    0100H   ; write data read from toggle switches into address 0100H
        SWI
```

Example: The following program clears the display then starts an infinite loop which reads the status of the toggle switches and displays the result on the seven segment display.

```
        JSR     0DFA0H      ; CLEARD
RETOG:  LDAA    B500H       ; read toggle switches
        ANDA    #0FH        ; mask four MSBs
        JSR     0DFA3H      ; PATCON
        LDAB    #01H        ; select seven segment display
        JSR     0DFA6H      ; DISPAT
        BRA     RETOG       ; infinite loop
```

JSR 0DFA0H instruction clears the display. Then, program reads the input from toggle switches (when $B500_H$ is given as input to the address decoding circuit), and jumps to the subroutine at address $0DFA3_H$ (PATCON). This subroutine, converts a data into a code which can be displayed on the seven segment display. Then the program jumps to the subroutine at address $0DFA6_H$ (DISPAT), which displays the converted data on the seven segment display. This process is repeated in an infinite loop. In this loop when the condition of the toggle switches changes, the data displayed on the seven segment display also changes.

7.3 MC6820 PIA (Peripheral Interface Adapter)

The MC6820 Peripheral Interface Adapter (PIA), in Figure 7.4, provides a flexible method of connecting byte-oriented peripherals to the MPU. The PIA, while relatively complex itself, permits the MPU to handle a wide variety of equipment types with minimum additional logic and simple programming.

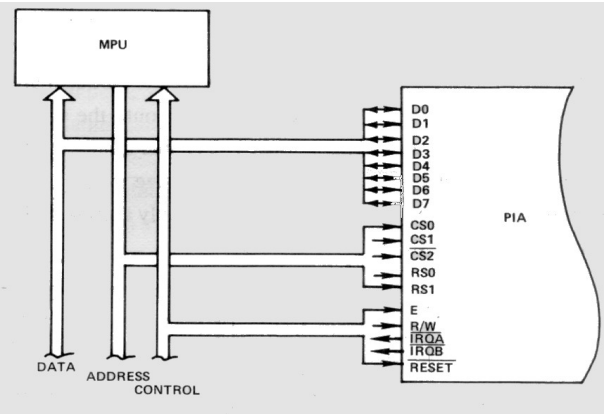

Figure 7.4 - MPU/PIA Interface

Data flows between the MPU and the PIA on the System Data Bus via eight bi-directional data lines, D0 through D7. The direction of data flow is controlled by the MPU via the Read/Write input to the PIA.

The "MPU side" of the PIA also includes three chip select lines, CS0, CS1, $\overline{CS2}$, for selecting a particular PIA. Two register select inputs, RS0 and RS1, are used in conjuction with a control bit (b_2) within the PIA for selecting specific registers in the PIA. Figure 7.6 shows the PIA control register format. The MPU can read or write into the PIA's internal registers by addressing PIA via the System Address Bus using these five input lines and the R/W signal. From the MPU's point of view, each PIA is simply four memory locations that are treated in the same manner as any other read/write memory. Table 7.5 shows the memory addresses that can be accessed by the MPU and the corresponding PIA registers.

Address	PIA Register
$E000_H$	Output and data direction register (Port A)
$E001_H$	Control register (Port A)
$E002_H$	Output and data direction register (Port B)
$E003_H$	Control register (Port B)

Table 7.5 - PIA Register Addresses

The MPU also provides a timing signal to the PIA via the Enable input. The Enable (E) pulse is used to condition the PIA's internal interrupt control circuitry and for the timing of peripheral control signals.

The "Peripheral side" of the PIA includes two 8-bit bi-directional data buses (PA0 - PA7 and PB0 - PB7) and four interrupt/control lines (CA1, CA2, CB1, CB2) (Figure 7.5). All of the lines on the "Peripheral Side" of the PIA are compatible with standard TTL Logic. In addition, all lines serving as outputs on the "B" side of each PIA (PB0-PB7, CB1, CB2) will supply up to one miliamp of drive current at 1.5 volts, therefore resulting in a more suitable part to use with current requesting peripherals (such as leds etc.)

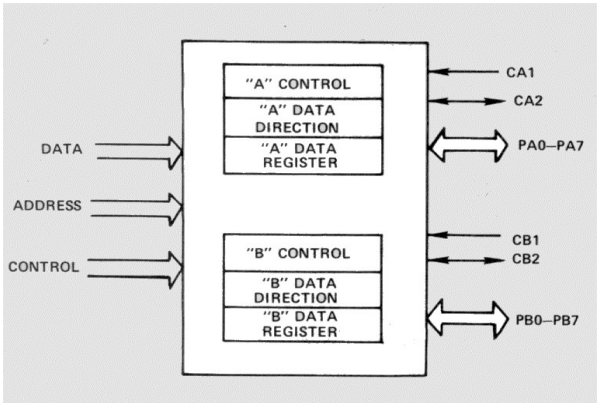

Figure 7.5 - PIA Registers

Internal Organization

Internally, the PIA is divided into symmetrical independent register configurations. Each half has three main features: an Output Register, a Control Register, and a Data Direction Register (Figure 7.5). These registers are addressed by MPU as memory locations from which data can be either read or written. The Output and Data Direction Registers on each side represents a single memory location to the MPU. The selection between them is internal to the PIA and determined by a bit in their Control Register (it is common convention that bit 2 of a data structure implies the third bit from the least significant).

Data Direction Registers (DDR) are used to establish each individual peripheral bus line as either an input or an output. This is accomplished by having the MPU write "ones" or "zeros" into the eight bit positions of the DDR. Zeros or ones cause the corresponding peripheral data lines to function as inputs or outputs, respectively.

Output Registers, when addressed, store the data present on the MPU Data Bus during MPU write operation. This data will immediately appear on those peripheral lines that have been programmed as outputs. During an MPU Read operation, the data present on peripheral lines, programmed as inputs, is transferred directly to the system Data Bus.

Two Control Registers, allow the MPU to establish and control the operating modes of the PIA. It is by means of these four lines that control information is passed back and forth between the MPU and peripheral devices.

Data Direction Register access is used in conjunction with the register select lines to select between internal registers. For a given register select combination, the status bit b_2 of the Data Direction Register determines whether the Data Direction Register (if b2=0) or the Output Register (if b2=1) is addressed by the MPU.

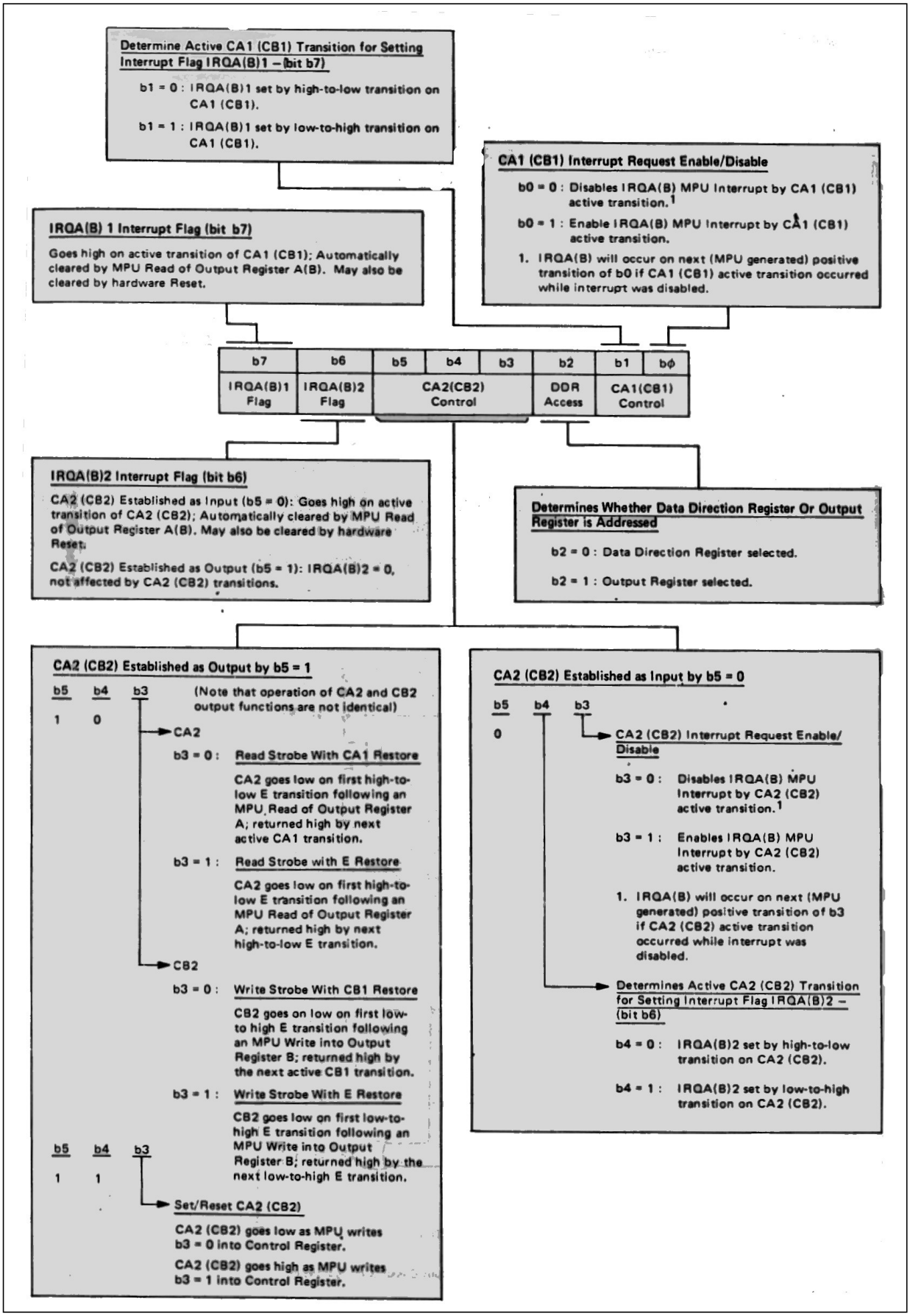

Figure 7.6 - PIA Control Register Format

The PIA (6821) has got three chip select inputs (CS0, CS1, $\overline{CS2}$), and two register select inputs (RS0 and RS1) (Figure 7.7). CS0 and CS1 are high active, $\overline{CS2}$ is low active. CS0 and CS1 is connected to the Vcc (+5V). So these two inputs are always in high active position. $\overline{CS2}$ input is connected to the Y0 ouput of 74LS138 3×8 decoder. In the address decoding circuit, when memory block between E000 and E3FF is decoded, the output Y0 is logic(0). When microprocessor addresses a memory location in this range, PIA is selected via $\overline{CS2}$. The microprocessor's A0 and A1 address lines are connected to PIA's RS0 and RS1 (register select) inputs and they select one of the PIA registers in Table 7.5 depending on the value of $\overline{CS2}$, RS0, RS1, then the data in D7-D0 (from the MPU) is transferred to PIA registers PB7-PB0 (or PIA7-PIA0), ot the data in PIA registers is transferred through D7-D0 to the MPU.

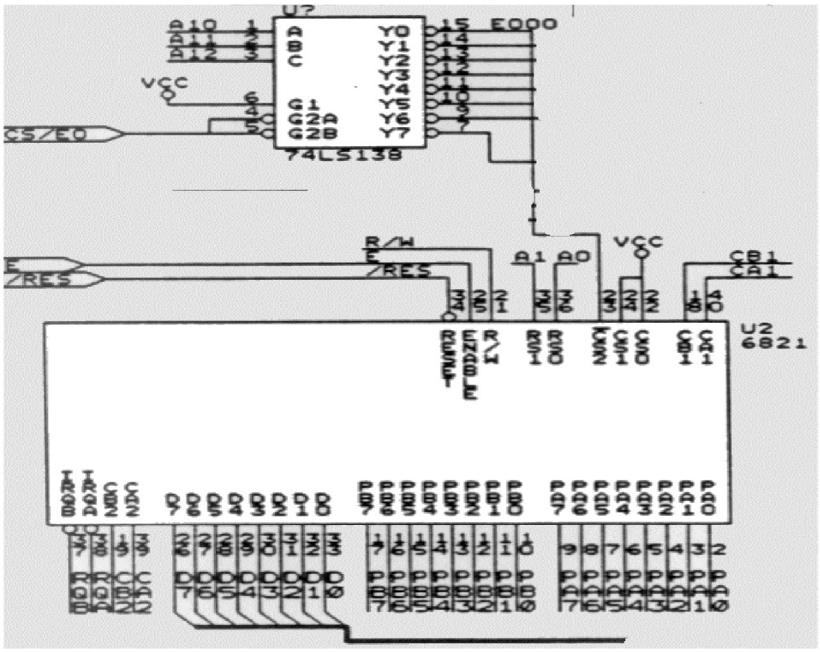

Figure 7.7 – 6821 PIA's hardware diagram

The PIA's registers are reset when the RESET input of PIA is activated (logic '0'). This is usually used when initializing the system. Therefore this input is connected to the microprocessor's RESET line.

PIA's $R/\overline{W}$ input determines the data transfer direction, so it is connected to the microprocessor $R/\overline{W}$ line. PIA's E signal synchronises the data transfer between PIA and microprocessor, so it is connected to microprocessor E clock line. D0-D7 data inputs are connected to the microprocessor data lines for data transfer. PIA's Port A (PA0-PA7) can be programmed as output or input. Similarly PIA's Port B (PB0-PB7) can be programmed as output or input.

Example: In PIA configuration, Control Register is addressed by a unique memory location E001 for port A and E003 for port B. But Data Direction Register (DDR) and Output Register are addressed by the same memory location that is E000 for port A and E002 for port B. Therefore it is necessary to choose whether DDR or Output Register is used by means of b2 of appropriate control register. DDR selection can be achieved by clearing b2 of E000 for port A and E002 for port B.

In the following program, PIA's Port A pins are programmed as input and PIA's Port B pins are programmed as output.

```
CLR     E001H   ; DDR is selected for port A
CLR     E003H   ; DDR is selected for port B
CLR     E000H   ; All of the port A pins are programmed as input
LDAA    #FFH    ;
STAA    E002H   ; All of the port B pins are programmed as output
LDAA    #04H    ;
STAA    E001H   ; Output register is selected for port A
STAA    E003H   ; Output register is selected for port B
LDAA    E000H   ; Data is read from Port A and written into accumulator A.
STAA    E002H   ; Data in accumulator A is stored in pins of port B.
```

8. TIME DELAYS

While handling input/output in a microprocessor system, one important problem is the generation of time intervals with specific lenghts. Such intervals are necessary to debounce mechanical switches, to refresh displays, and to provide timing for devices that transfer data regularly.

Timing intervals can be produced in several ways:

1) By hardware with one-shots or monostable multivibrations. These devices produce a single pulse of fixed duration in response to a pulse input.

2) By a combination of hardware and software with a flexible programmable timer. Motorola MC6840 can provide timing intervals of various lengths with a variety of starting and ending conditions.

3) By software with delay routines. These routines use the processor as a counter. This is possible since the processor has a stable clock reference.

The software method is inexpensive but may overburden the processor. The programmable timers are relatively expensive, but are easy to interface and may be able to handle many complex timing tasks.

8.1 Software generated time delays

A simple delay routine works as follows:

Step 1) Load a register with a specified value.
Step 2) Decrement the register.
Step 3) Repeat Step 2 until it is equal to zero,

This routine does nothing except consuming time. The amount of time consumed depends on the execution time of the instructions used. Maximum delay time is limited by the size of the register.

Ex 1: Following program code uses a single 8-bit register to produce delay.

```
TLOOP:   LDAA   #0AH   ------>  2 clock cycles
DELAY:   DECA          ------>  2 clock cycles
         BNE    DELAY  ------>  4 clock cycles
         RTS
```

Delay time is calculated using the time (in clock cycles) consumed by the execution of each instruction. In this example LDAA instruction is executed once, and DELAY loop is repeated 10 times (as the number in accumulator A is $0A_H$).

The delay (N_C) generated by the above code in number of cycles is:
$N_C = 2 + (2 + 4)*10 = 62$ clock cycles

If processor clock is 1MHz then the period $T_C = 1\ \mu s$
Therefore the delay time of the above program is:
Delay Time = $N_C * T_C = 62\ \mu s = 0{,}062$msec.

Accumulator A is an 8-bit register. Maximum number that can be stored in this register is FF_H. Therefore the maximum delay can be obtained if 00_H is stored in accumulator A initially. Note that DELAY loop begins with a DECA instruction and if accumulator A initially contains FF_H the loop is repeated FE_H times, until accumulator A becomes 00_H. However storing 00_H to accumulator A initially, causes the loop to be repeated FF_H times, and the maximum delay is calculated as:

$[2 + 6*2^8]\ \mu s = 1538\ \mu s = 1{,}538$ msec. $= 0{,}01538$ seconds.

Ex 2: To obtain longer delays with 8-bit registers, two 8-bit registers can be used.

```
TLOOP:    LDAB    #xH          ⟶  2 clock cycles
DELAY2:   LDAA    #yH          ⟶  2 clock cycles
DELAY1:   DECA                 ⟶  2 clock cycles
          BNE     DELAY1       ⟶  4 clock cycles
          DECB                 ⟶  2 clock cycles
          BNE     DELAY2       ⟶  4 clock cycles
          RTS
```

In this example two 8-bit registers (accumulators A and B) are used. LDAB instruction loads value x_H to accumulator B and is executed once. Therefore, DELAY2 loop is repeated x_H times. At each repetition of DELAY2 loop, value y_H is loaded to accumulator A, which causes DELAY1 loop to be repeated $x_H * y_H$. Total number of cycles is:

$N_C = 2 + (2 + (2 + 4)*\mathbf{y}_H + 2 + 4) * \mathbf{x}_H$ cycles (Note: while calculating N_C, convert the x_H and y_H into decimal numbers)

Accumulators A and B are 8-bit registers. Maximum number that can be stored in each register is FF_H. As the delay loops begin with a DEC instruction, maximum delay time is obtained if both accumulators contain 00_H initially.

$N_C = 2 + (2 + (2 + 4)*2^8 + 2 + 4) * 2^8$ cycles, and
maximum delay time = 395266 $\mu s \cong$ 0,4 second.

Ex 3: Following code uses Index Register (16-bit register) to produce delay.

```
TLOOP:    LDX     # xH          ⟶  3 clock cycles
DELAY:    DEX                   ⟶  4 clock cycles
          BNE     DELAY         ⟶  4 clock cycles
          RTS
```

LDX instruction is executed once. Then, DELAY loop is repeated x_H times (as the number loaded in index register is x_H).

$N_C = 3 + (4 + 4)*\mathbf{x}_H$ (Note: while calculating N_C, x_H must be converted into decimal)
Maximum delay time(when $X=0000_H$) = $3 + 8*2^{16} = 524291\ \mu s \cong 0{,}52$ seconds.

Ex 4: To obtain 1 ms delay using index register

$$N_C = \frac{DelayTime}{T_C} = \frac{1ms}{1\mu s} = 1000$$ clock cycles are required

For the loop in Ex. 3

$N_C = 3 + 8*x = 1000$

Then,

$$x = \frac{1000 - 3}{8} = 124_{10} = (007C)_H$$

Therefore initially $7C_H$ must be loaded to index register (LDX #007CH)

9. MC6802 Interrupts

In a typical application, peripheral devices may be continuously generating asynchronous signals (interrupts) that must be acted on by the MPU. The interrupts may be either requests for service or acknowledgements of services performed earlier by the MPU. The MC6802 MPU provides several methods for automatically responding to such interrupts in an orderly manner.

During the execution of the program when an interrupt occurs; the status of MPU is stored (i.e. accumulators, program counter, index register, and CCR) is stored in the memory addressed by the Stack Pointer (Figures 9.1, 9.2). Then program counter is loaded with the starting address of the related Interrupt Service Routine (ISR). After the execution of the ISR, the RTI instruction causes the contents of the Program Counter to be restored from the stack, and program execution continues with the next instruction.

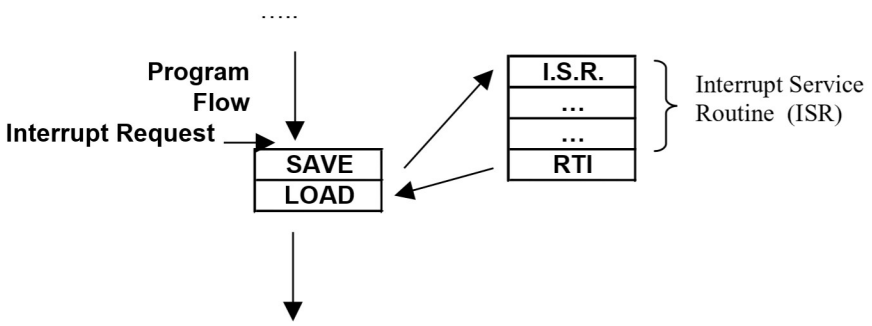

Figure 9.1 - Basic flow diagram of an interrupt service

In the control of interrupts, three general problems must be considered:

1. It is the characteristic of most applications that interrupts must be handled without permanently disrupting the task in process when the interrupt occurs. The MC6802 handles this by saving the results of its current activity so that processing can be resumed after the interrupt has been serviced.

2. There must be a method of handling multiple interrupts since several peripherals may be requesting service simultaneously.

3. If some signals are more important to system operation or if certain peripherals require faster servicing than others, there must be a method of prioritizing the interrupts.

The status of the microprocessing unit is stored in the stack during the following operations (Figure 9.2):

- in response to an external condition indicated by a negative edge on the “Non-maskable Interrupt” control input signal to the MPU.
- during the execution of a machine code corresponding to either of the source language instruction SWI or WAI.
- during servicing of an interrupt from a peripheral device, in response to a negative edge on the “Interrupt request” control input signal to the MPU.

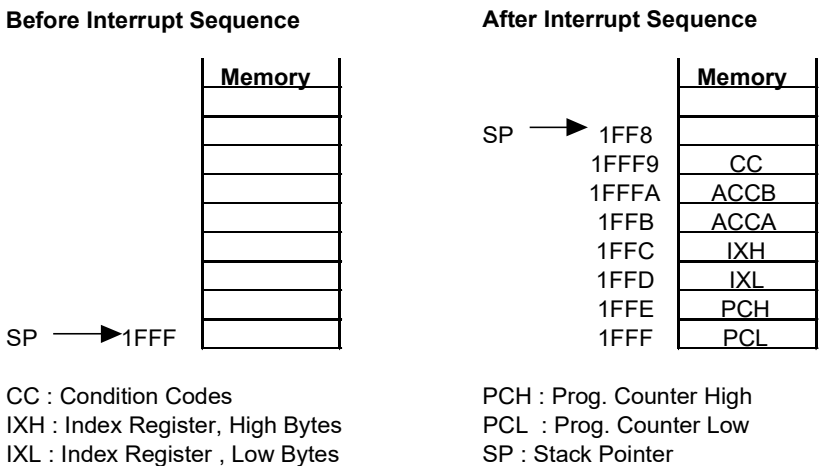

Fig 9.2 – Saving the Status of the Microprocessor in the Stack

When an interrupt occurs;

a) memory address of the next instruction to be executed is stored in the stack
b) contents of the other registers are stored in the stack

9.1 Interrupt Pointers

The MPU has three hardware interrupt inputs, Reset $\overline{(RES)}$[1], Non-Maskable Interrupt $\overline{(NMI)}$, and Interrupt Request $\overline{(IRQ)}$. An interrupt sequence can be initiated by applying a suitable control signal to any of these three inputs or by using the software SWI instruction. The resulting sequence is different for each case.

A block of memory, called interrupt vector is reserved for pointers to the interrupt service rotines which are to be executed in the event of a reset (or power down), a non-maskable interrupt signalled by a "low state" of the "Non-maskable Interrupt" control input, a software interrupt, or a response to an interrupt signal from a peripheral device. Figure 9.3 shows the memory addresses reserved as the interrupt vector and the associated interrupt types.

Interrupt Type	Addresses Used
Reset ($\overline{RESET}$)	FFFE,FFFF
Non-Maskable Interrupt ($\overline{NMI}$)	FFFC,FFFD
Software Interrupt Instruction (*SWI*)	FFFA,FFFB
Interrupt Request ($\overline{IRQ}$)	FFF8,FFF9

Figure 9.3 - Interrupt Vector, Permanent Memory Assignments

9.2 Interrupt Request ($\overline{IRQ}$)

Inputs to $\overline{IRQ}$ are normally generated in PIAs and ACIAs but may also come from other user-defined hardware. In either case, various interrupts may be wired-ORed and applied to the MPU's $\overline{IRQ}$ input. This input is level sensitive, a logic zero causes the MPU to initiate the interrupt sequence[2]. A flow chart of the $\overline{IRQ}$ sequence is given in Figure 9.4.

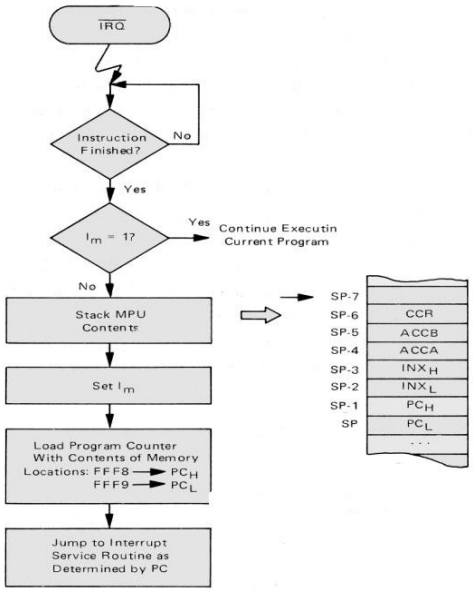

Figure 9.4 - Hardware Interrupt Request Sequence

After finishing its current instruction and testing the Interrupt Mask in the CCR, the MPU stores the contents of its programmable registers in the memory locations specified by the Stack Pointer. This stacking process takes seven memory cycles; two for each of the Index register and Program Counter, and one each for accumulator A, accumulator B, and the CCR. The Stack Pointer is decremented seven locations and is pointing to the next empty memory location.

The MPU's next step of setting Interrupt Mask to logic one allows the system interrupt control program to determine the order in which multiple interrupts will be handled. If it is desirable to recognize another interrupt (of higher priority, for example) before service of the first is complete, the Interrupt Mask can be cleared by a CLI instruction at the beginning of the current service routine. If each interrupt is to be completely serviced before another is recognized, the CLI instruction is omitted and a Return from Interrupt instruction, RTI, placed at the end of the service routine restores the Interrupt Mask status from the stack, thus enabling recognition of subsequent interrupts.

Note that if the former method is selected, the original interrupt service will still eventually be completed. This is due to the fact that the later interrupt also causes the current status to be put on the stack for later completion. This process is general and means that interrupts can be "nested" to any depth required by the system limited only by memory size. The status of the interrupted routines is returned in a LIFO basis.

After setting the Interrupt Mask, the MPU next obtains the address of the first interrupt service routine instruction from memory locations permanently assigned to the $\overline{IRQ}$ interrupt input. This is accomplished by loading the Program Counter's high and low bytes from memory locations responding to addresses, FFF8 and FFF9, respectively. The MPU then fetches the first instruction from the location now designated by the Program Counter.

[2] $\overline{IRQ}$ is a maskable input . If the Interrupt Mask Bit within the MPU is set, low levels on the $\overline{IRQ}$ line will not be recognized; the MPU will continue current program execution until the mask bit is cleared by encountering the Clear Interrupt (CLI) instruction in the control program, or an RTI is encountered.

This technique of indirect addressing (also called vectoring) is also used by other interrupt sequences. The "vectors" are placed in the memory locations corresponding to addresses FFF8 through FFFF during program development (Figure 9.3).

9.3 Non-Maskable Interrupt $(\overline{NMI})$

The Non-Maskable Interrupt $(\overline{NMI})$ must be recognized by the MPU as soon as the $\overline{NMI}$ line goes to logic zero. This interrupt is often used as a power-failure sensor or to provide interrupt service to a "hot" peripheral that must be allowed to interrupt.

Except for the fact that it cannot be masked, the $\overline{NMI}$ interrupt sequence is similar to $\overline{IRQ}$ (Figure 9.5). After completing its current instruction, the MPU stacks its registers, sets the Interrupt mask and fetches the starting address of the $\overline{NMI}$ interrupt service routine by vectoring to FFFC and FFFD. The MPU then starts execution of the Non-Maskable Interrupt Program, which begins with the instruction which is now addressed by the program counter.

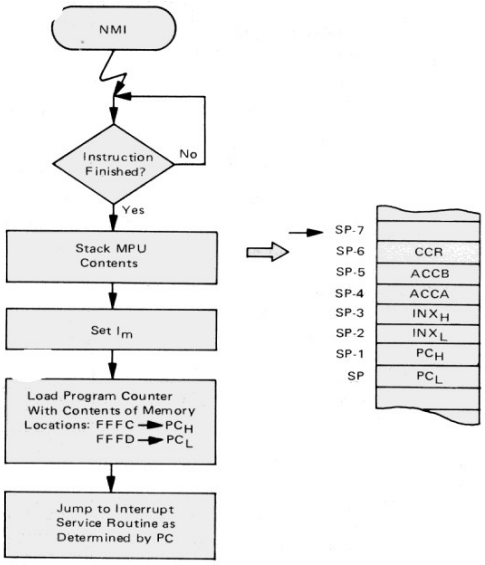

Figure 9.5 - Non-Maskable Interrupt Sequence

9.4 Reset $(\overline{RES})$

The Reset interrupt sequence differs from $\overline{NMI}$ and $\overline{IRQ}$ in two respect. When $\overline{RES}$ is low, the MPU places FFFE (the high order byte of the $\overline{RES}$ vector location) on the Address Bus in preparation for executing the $\overline{RES}$ interrupt sequence. It is normally used following power on to reach an initializing program that sets up system starting conditions such as initial value of the Program Counter, Stack Pointer, PIA Modes, etc. It is also available as a restart method in the event of system lockup or runaway. Because of its use for starting the MPU from a power down state, the $(\overline{RES})$ sequence is initiated by a positive going edge. Also, since it is normally used only in a start-up mode, there is no reason to store the MPU contents on the stack. After setting the Interrupt mask, the MPU loads the Program Counter from the memory locations responding to FFFE and FFFF and then proceeds with the initialization program (Figure 9.6).

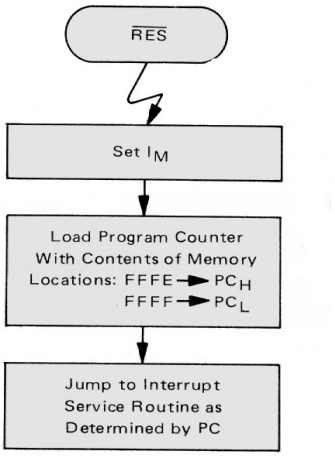

Figure 9.6 - Reset Interrupt Sequence

9.5 Software Interrupt (*SWI*)

The MPU also has a program initiated interrupt mode. Execution of the software interrupt (*SWI*) instruction by the MPU initiates the sequence (Figure 9.7). The sequence is similar to the hardware interrupts except that it is initiated by "software" and the vector is obtained from memory locations responding to FFFA and FFFB.

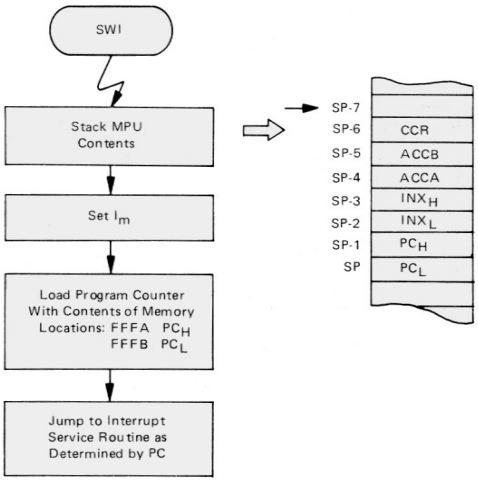

Figure 8.7 - Software Interrupt Sequence

The Software Interrupt is useful for inserting break-points in the program as an aid in debugging and troubleshooting. In effect, the SWI stops the process in place and puts the MPU register contents into memory where they can be examined or displayed.

During execution of the SWI instruction, the status of the MPU is stored in the stack, the value stored for the PC is the address of the SWI instruction plus one. After the status has been stored in, the interrupt mask bit "I" is set (I=1). The MPU will not respond to an interrupt request from a peripheral device while the interrupt mask is set. The program counter is then loaded with the address stored in software interrupt pointer, at location FFFA and FFFB. The MPU then proceeds with execution of a SWI program, which begins with the instruction pointed by the program counter. The MPU will remain insensitive to an interrupt request from any peripheral device until the interrupt mask bit has been reset by execution of the programmed instruction.

9.6 Wait Instruction *(WAI)*

During the execution of the WAI instruction, the status of the MPU is stored in the stack. The value stored for the PC is the address of the WAI instruction plus one. Execution of the WAI instruction does not change the interrupt mask bit.

If the interrupt mask bit is set (I=1), the MPU cannot respond to an interrupt request from any peripheral device. Execution will stop after saving the status of the MPU. In this case execution could be resumed only by a non-maskable interrupt or a reset interrupt. If the interrupt mask bit is in the reset state (I=0), the MPU will service any interrupt request which may be present.

9.7 Return from Interrupt *(RTI)*

This is a 1-byte machine instruction. Execution of this instruction consists of the restoration of the MPU to a state pulled from the stack. After the execution of the RTI instruction, seven bytes of information are pulled from the stack and stored in respective registers of the MPU. The address stored in the stack pointer is incremented before each byte of information is pulled from the stack.

9.8 Interrupt Prioritizing

If there is only one peripheral capable of requesting service, the source of the interrupt is known and the control program can immediately begin the service routine. More often, several devices are allowed to originate interrupt request and the first task of the interrupt routine is to identify the source of the interrupt.

There is also the possibility that several peripherals are simultaneously requesting service. In this case, the control program must also decide which interrupt to service first. The $\overline{(IRQ)}$ interrupt service routine in particular may be complex since most of the I/O interrupts are wire-ORed on this line.

The most common method of handling the multiple and/or simultaneous $\overline{(IRQ)}$ interrupts is to begin the service routine by "polling" the peripheral's signals coming in through a PIA or an ACIA. The polling procedure is very simple. In addition to causing $\overline{(IRQ)}$ to go low, the interrupting signal also sets a flag bit in the PIA's or ACIA's internal registers. Since these registers represent memory locations to the MPU, the polling consists of nothing more than stepping through the locations and testing the flag bits.

Establishing the priority of simultaneous interrupts can be handled in either of two ways. The simplest is to establish priority by the order in which the PIAs and ACIAs are polled. That is, the first I/O flag encountered gets the service, so higher priority devices are polled first. The second method first finds all the interrupt flags and then uses a special program to select the one of having the highest priority. This method permits a more sophisticated approach in that the priority can be modified by the control program.

Software techniques can, in theory, handle any number of devices to any sophistication level of prioritizing. In practice, if there are many sources of interrupt requests, the time required to find the appropriate interrupt can exceed the time available to do so. In this situation, external prioritizing hardware can be used to speed up the operation.

Example: Write a program which fills the seven segment displays with the number F_H. The program repeats itself when ↓ is entered. The program finishes when → is entered. When an interrupt request occurs, the program proceeds as follows:
If the interrupt request is **IRQ** interrupt, fill the displays with the number $\mathbf{A_H}$.
If the interrupt request is **SWI** interrupt, fill the displays with the number $\mathbf{B_H}$.
If the interrupt request is **NMI** interrupt, fill the displays with the number $\mathbf{C_H}$.
If the interrupt request is **RESET** interrupt, fill the displays with the number $\mathbf{E_H}$.

```
            ORG     100H

            LDS     #1FFFH      ; initialize the stack pointer
            CLC                 ; clear carry
            LDAA    #01H        ; load the first seven-segment display code
            STAA    600H        ; store the code in memory location 600H
            LDX     #200H       ; Initialise IRQ interrupt
            STX     FFF8H       ; service routine address
            LDX     #300H       ; Initialise SWI interrupt
            STX     FFFAH       ; service routine address
            LDX     #400H       ; Initialise NMI interrupt
            STX     FFFCH       ; service routine address
            LDX     #500H       ; Initialise RESET interrupt
            STX     FFFEH       ; service routine address
            CLI
DISP:       LDX     600H        ; load the seven-segment display code
            LDAA    #FH         ; load number to fill the display in accumulator A
            JSR     0DFA3H      ; PATCON
            LDAB    0,X         ; select seven-segment display
            JSR     0DFA6H      ; DISPAT
            CMPB    #80H        ; are all 7 segment displays FH?
            BEQ     REPLAY      ; YES, jump to decision
            ASL     600H        ; Arithmetic shift left contents of the 600H
            BRA     DISP        ; Loop instruction

STOP:       SWI

REPLAY:     LDAA    #01H        ; load the first seven-segment display code
            STAA    600H        ; store the code in memory location 600H
KLOOP:      JSR     0DFA9H      ; RKEYC
            ANDA    #3FH
            CMPA    #13H        ; is it ↓?
            JSR     0DFA0H      ; CLEARD
            BEQ     DISP        ; YES, replay routine
            CMPA    #11H        ; NO, is it → ?
            BEQ     STOP        ; YES, finish program
            BRA     KLOOP       ; None of them, replay KLOOP loop

            ORG     200H        ; Interrupt Service Routine for IRQ

            JSR     0DFA0H      ; CLEARD
            CLC                 ; Clear carry
            LDAA    #01H
            STAA    600H
DISP1:      LDX     #600H
            LDAA    #AH         ; displays AH
            JSR     0DFA3H
            LDAB    0,X
            JSR     0DFA6H
            JSR     DELAY
            CMPB    #80H
            BEQ     FINISH1
```

```
                ASL     600H
                BRA     DISP1

FINISH1:        RTI

                ORG     300H        ; Interrupt Service Routine for SWI

                JSR     0DFA0H      ; CLEARD
                CLC                 ; Clear carry
                LDAA    #01H
                STAA    600H
DISP2:          LDX     #600H
                LDAA    #BH
                JSR     0DFA3H
                LDAB    0,X
                JSR     0DFA6H
                JSR     DELAY
                CMPB    #80H
                BEQ     FINISH2
                ASL     600H
                BRA     DISP2
FINISH2:        RTI

                ORG     400H        ; Interrupt Service Routine for NMI

                JSR     0DFA0H      ; CLEARD
                CLC                 ; Clear carry
                LDAA    #01H
                STAA    600H
DISP3:          LDX     #600H
                LDAA    #CH
                JSR     0DFA3H
                LDAB    0,X
                JSR     0DFA6H
                JSR     DELAY
                CMPB    #80H
                BEQ     FINISH3
                ASL     600H
                BRA     DISP3
FINISH:         RTI

                ORG     500H        ; Interrupt Service Routine for RESET

                JSR     0DFA0H      ; CLEARD
                CLC                 ; Clear carry
                LDAA    #01H
                STAA    600H
DISP4:          LDX     #600H
                LDAA    #EH
                JSR     0DFA3H
                LDAB    0,X
                JSR     0DFA6H
                JSR     DELAY
                CMPB    #80H
                BEQ     FINISH4
                ASL     600H
                BRA     DISP4
FINISH4:        RTI

                ORG     700H        ; Delay Service Subroutine
```

```
DELAY:      STX     1000H
            PSHB
            LDX     #0FFFH
KDELAY:     LDAB    #02H
KDELAY1:    DECB
            BNE     KDELAY1
            DEX
            BNE     KDELAY
            PULB
            LDX     1000H
            RTS
```