COURSE NAME WITH CODE: MICROCONTROLLERS(EC45) **FACULTY NAME: RAMGOPAL SEGU G** **DATE:**	**UNIT-III**

OBJECTIVES OF THE LECTURE:

To Study on

- Assembly language programs,
- Software delay calculations,
- Software delay programming,
- Subroutine call, Stack operations with programming,
- Applications

REFERENCES:

1. " The 8051 Microcontroller Architecture Programming & Applications", 2e Kenneth J. Ayala
2. " The 8051 Microcontroller and Embedded Systems-using Assembly and C", Muhammad Ali Mazidi, Janice GillespieMazidi and Rllin D. McKinalay; PHI, 2006/Pearson,2006

Assembly language programs

Structure of Assembly language:

An Assembly language program consists of, among other things, a series of lines of Assembly language instructions. An Assembly language instruction consists of a mnemonic, optionally followed by one or two operands. The operands are the data items being manipulated, and the mnemonics are the commands to the CPU, telling it what to do with those items.

```
      ORG  0H          ;start (origin) at location 0
      MOV  R5,#25H     ;load 25H into R5
      MOV  R7,#34H     ;load 34H into R7
      MOV  A,#0        ;load 0 into A
      ADD  A,R5        ;add contents of R5 to A
                       ;now A = A + R5
      ADD  A,R7        ;add contents of R7 to A
                       ;now A = A + R7
      ADD  A,#12H      ;add to A value 12H
                       ;now A = A + 12H
HERE: SJMP HERE        ;stay in this loop
      END              ;end of asm source file
```

A given Assembly language program is a series of statements, or lines, which are either Assembly language instructions such as ADD and MOV, or statements called directives. While instructions tell the CPU what to do, directives (also called pseudo-instructions) give directions to the assembler. For example, in the above program while the MOV and ADD instructions are commands to the CPU, ORG and END are directives to the assembler. ORG tells the assembler to place the opcode at memory location 0 while END indicates to the assembler the end of the source code. In other words, one is for the start of the program and the other one for the end of the program.

Instructions tell the CPU what to do, directives (also called pseudo-instructions) give directions to the assembler. For example, in the above program while the MOV and ADD instructions are commands to the CPU, ORG and END are directives to the assembler. ORG tells the assembler to place the opcode at memory location 0 while END indicates to the assembler the end of the source code. In other words, one is for the start of the program and the other one for the end of the program.

An Assembly language instruction consists of four fields:

```
[label:]    mnemonic   [operands]   [;comment]
```

Brackets indicate that a field is optional, and not all lines have them. Brackets should not be typed in. Regarding the above format, the following points should be noted.

1. The label field allows the program to refer to a line of code by name. The label field cannot exceed a certain number of characters. Check your assembler for the rule.

2. The Assembly language mnemonic (instruction) and operand(s) fields together perform the real work of the program and accomplish the tasks for which the program was written. In Assembly language statements such as

ADD A, B MOV A, #67

ADD and MOV are the mnemonics, which produce opcodes; and "A, B" and "A, #67″ are the operands. Instead of a mnemonic and an operand, these two fields could contain assembler pseudo-instructions, or directives. Remember that directives do not generate any machine code (opcode) and are used only by the assembler, as opposed to instructions that are translated into machine code (opcode) for the CPU to execute. In Program 2-1 the commands ORG (origin) and END are examples of directives (some 8051 assemblers use .ORG and .END). Check your assembler for the rules. More of these pseudo-instructions are discussed later.

3. The comment field begins with a semicolon comment indicator ";". Comments may be at the end of a line or on a line by themselves. The assem bler ignores comments, but they are indispensable to programmers. Although comments are optional, it is recommended that they be used to describe the program and make it easier for someone else to read and understand, or for the programmer to remember what they wrote.

4. Notice the label "HERE" in the label field in Program 2-1. Any label referring to an instruction must be followed by a colon symbol, ":". In the SJMP (short jump instruction), the 8051 is told to stay in this loop indefinitely. If your system has a monitor program you do not need this line and it should be deleted from your program. In the next section we will see how to create a ready-to-run program.

1) Write an assembly language program to transfer n = 10 bytes of data from location 8035h to location 8050h.

Algorithm:

1. Initialize origin of program at 0000H.

2. Jump to 30H and initialize origin at 30H.

3. Initialize registers to hold count & also the source & destination addresses of code memory.

4. Load lower byte of address into DPL register.

5. Get data from source location into accumulator

6. Move destination address into DPL register.
7. Transfer data to the destination location.
8. Increment source and destination addresses.
9. Decrement the counting register and check if it has reached Zero.
10. Repeat step 5 to 9 till count is zero.

Program :

Label	Mnemonic/ Operands	Comments
	ORG 0000H	//Origins program from 0000H location
	SJMP 30H	//Unconditional jump to 30H
	ORG 30H	//Program starts from 30H
	MOV DPH,#80H	//Higher byte of address is stored in DPH
	MOV R0,#35H	//Lower byte of source address
	MOV R1,#50H	//Lower byte of destination address
	MOV R3,#0AH	//count- Number of bytes to be transferred
BACK:	MOV DPL, R0	//DPTR stores complete source address
	MOVX A,@DPTR	//Read content at source address
	MOV DPL, R1	//Update DPTR with destination address
	MOVX @DPTR,A	//Write data at destination
	INC R0	//Increment source address lower byte
	INC R1	//Increment destination address lower byte
	DJNZ R3, BACK	//Decrement R3 & jump to BACK if its not Zero
HERE:	SJMP HERE	//Infinite looping
	END	//End directive

2). Write a program to find the average of five 8 bit numbers. Store the result in H. (Assume that after adding five 8 bit numbers, the result is 8 bit only).

```
ORG 0000H
MOV 40H, #05H
MOV 41H, #55H
MOV 42H, #06H
MOV 43H, #1AH
MOV 44H, #09H
MOV R0, #40H
MOV R5, #05H
MOV B, R5
CLR A
Loop: ADD A,@RO
        INC RO
DJNZ R5, Loop
DIV AB
MOV 55H, A
END
```

3) Write a program to find the cube of an 8 bit number program is as follows

```
ORG 0000H
MOV R1, #N
MOV A, R1
MOV B, R1
MUL AB   //SQUARE IS COMPUTED
MOV R2, B
MOV B, R1
MUL AB
MOV 50, A
MOV 51, B
MOV A, R2
MOV B, R1
MUL AB
ADD A, 51H
MOV 51H, A
MOV 52H, B
MOV A, # 00H
ADDC A, 52H
MOV 52H, A   //CUBE IS STORED IN 52H, 51H, 50H
END
```

4) write a program to store data FFH into RAM memory locations 50H to 58H using direct Addressing mode

```
ORG 0000H   ; Set program counter 0000H
MOV A, #0FFH; Load FFH into A
MOV 50H, A   ; Store contents of A in location 50H
MOV 51H, A   ; Store contents of A in location 5IH
MOV 52H, A   ; Store contents of A in location 52H
MOV 53H, A   ; Store contents of A in location 53H
MOV 54H, A   ; Store contents of A in location 54H
MOV 55H, A   ; Store contents of A in location 55H
MOV 56H, A   ; Store contents of A in location 56H
MOV 57H, A   ; Store contents of A in location 57H
MOV 58H, A   ; Store contents of A in location 58H
END
```

5)Write a program to add two 16 bit numbers stored at locations 51H-52H and 55H-56H and Store the result in locations 40H, 41H and 42H. Assume that the least significant byte of Data and the result is stored in low address and the most significant byte of data or the result is stored in high address.

```
ORG 0000H; Set program counter 0000H
MOV A, 51H   ; Load the contents of memory location 51H into A
ADD A, 55H   ; Add the contents of 55H with contents of A
MOV 40H, A; Save the LS byte of the result in location 40H
MOV A, 52H   ; Load the contents of 52H into A
ADDC A, 56H   ; Add the contents of 56H and CY flag with A
MOV 41H, A; Save the second byte of the result in 41H
MOV A, #00; Load 00H into A
ADDC A, #00; Add the  immediate data 00H and CY to A
MOV 42H, A; Save the MS byte of the result in location 42H
END
```

6)Write a program to multiply two 8 bit numbers stored at locations 70H and 71H and store the Result at memory locations 52H and 53H. Assume that the least significant byte of the result is stored in low address.

```
ORG 0000H; Set program counter 00 OH
MOV A, 70H; Load the contents of memory location 70h into A
MOV B, 71H; Load the contents of memory location 71H into B
MUL AB; Perform multiplication
MOV 52H, A ; Save the least significant byte of the result in location 52H MOV 53H,B ;
                Save the most significant byte of the result in location 53
END
```

7)Write a program to subtract a 16 bit number stored at locations 51H-52H from 55H-56H and Store the result in locations 40H and 41H. Assume that the least significant byte of data or the Result is stored in low address. If the result is positive, then store 00H, else store 01H in 42H.

```
ORG 0000H; Set program counter 0000H
MOV A, 55H; Load the contents of memory location 55 into A
CLR C; Clear the borrow flag
SUBB A, 51H; Sub the contents of memory 51H from contents of A
MOV 40H, A; Save the LSByte of the result in location 40H
MOV A, 56H; Load the contents of memory location 56H into A
SUBB A, 52H; Subtract the content of memory 52H from the content A
MOV 41H, A; Save the MSbyte of the result in location 415.
MOV A, #00; Load 00 into A
ADDC A, #00; Add the immediate data and the carry flag to A
MOV 42H, A; If result is positive, store00H, else store 0lH in 42H
END
```

8) Two 8 bit numbers are stored in location 1000h and 1001h of external data memory. Write a program to find the GCD of the numbers and store the result in 2000h.

ALGORITHM

- Step 1 :Initialize external data memory with data and DPTR with address
- Step 2 :Load A and TEMP with the operands
- Step 3 : Are the two operands equal? If yes, go to step 9
- Step 4 :Is (A) greater than (TEMP) ? If yes, go to step 6
- Step 5 :Exchange (A) with (TEMP) such that A contains the bigger number
- Step 6 :Perform division operation (contents of A with contents of TEMP)
- Step 7 :If the remainder is zero, go to step 9
- Step 8 :Move the remainder into A and go to step 4
- Step 9 :Save the contents 'of TEMP in memory and terminate the program

```
ORG 0000H  ; Set program counter 0000H
TEMP EQU 70H
TEMPI EQU 71H
MOV DPTR, #1000H ; Copy address 100011 to DPTR
MOVX A, @DPTR ; Copy First number to A
MOV TEMP, A  ; Copy First number to temp INC DPTR
MOVX A, @DPTR ; Copy Second number to A
LOOPS: CJNE A, TEMP, LOOP1  ; (A) /= (TEMP) branch to LOOP1
AJMP LOOP2   ; (A) = (TEMP) branch to L00P2
LOOP1:  JNC LOOP3   ; (A) > (TEMP) branch to LOOP3
NOV TEMPI, A  ; (A) < (TEMP) exchange (A) with (TEMP)
MOV A, TEMP
MOV TEMP, TEMPI
LOOP3:  MOV B, TEMP
DIV AB   ; Divide (A) by (TEMP)
```

```
MOV A, B   ; Move remainder to A
CJNE A,#00, LOOPS  ; (A)/=00 branch to LOOPS
LOOP2:  MOV A, TEMP
MOV DPTR, #2000H
MOVX @DPTR, A  ; Store the result in 2000H
```

9) Write an ALP to implement (display) an eight bit up/down BCD counters on watch window.

Algorithm:

1. Move 00 to A register
2. Call the delay subroutine for 1 second (in delay program move FFH to registers R1, R2 and R3, loop and decrement until 0).
3. Increment A register(add 99h for down counter)
4. Decimal adjust accumulator for the BCD up/down counter.

Program:

Label	Mnemonic/Operands	Comments
	ORG 0000H	
	SJMP 30H	
	ORG 30H	
	MOV a,#00	
BACK:	ACALL DELAY	
	ADD A,#99H	
	DA A	
	JNZ BACK	
HERE:	SJMP HERE	
DELAY:	MOV R1,#35H	
DECR1:	MOV R2,#0FFH	
DECR:	MOV R3, #0FFH	
	DJNZ R3,$	
	DJNZ R2, DECR	
	DJNZ R1, DECR1	
	RET	
	END	

** Kindly go through all lab programs.

Software delay routine in 8051 microcontroller

In an 8051 microcontroller, it requires 12 cycles of the processor clock for executing a single instruction cycle. For an 8051 microcontroller clocked by a 12MHz crystal, the time taken for executing one instruction cycle is 1µS and it is according to the equation, *Time for 1 instruction cycle= 12 /12MHz = 1µS*. The shortest instructions will execute in 1µS and other instructions will take 2 or more micro seconds depending up on the size of the instruction. Thus a time delay of any magnitude can be generated by looping suitable instructions a required number of time keep one thing in mind that software delay is not very accurate because we cannot exactly predict how much time it takes for executing a single instruction. Generally an instruction will be executed in the theoretical amount of time but sometimes it may advance or retard due to other reasons. Therefore it is better to use 8051 Timer for generating delay in time critical applications. However software delay routines are very easy to develop and well enough for less critical and simple applications.

Program to generate delay of 1mS.

```
DELAY: MOV R6, #250
MOV R7, #259
LABEL1: DJNZ R6, LABEL1
LABEL2: DJNZ R7, LABEL2
RET
```

Program to delay 1 second.

The program shown below produces a delay of around 1 second. In this program subroutine for delaying 1mS (DELAY) is called 4 times back to back and the entire cycle is repeated 250 times. As result, a delay of 4 x 1mS x 250 = 1000mS = 1 second is produced.

```
DELAY1: MOV R5, 250
LABELI: ACALL DELAY
        ACALL DELAY
        ACALL DELAY
        ACALL DELAY
```

```
        DJNZ R5, LABELI
             RET
DELAY:  MOV R6,#250
             MOV R7,#250
LABEL1: DJNZ R6, LOOP1
LABEL2: DJNZ R7, LOOP2
             RET
```

****

Find the size of the delay in the following program, if the crystal frequency is 11.0592 MHz.

```
          MOV  A,#55H     ;load A with 55H
AGAIN:    MOV  P1,A       ;issue value in reg A to port 1
          ACALL DELAY     ;time delay
          CPL  A          ;complement reg A
          SJMP AGAIN      ;keep doing this indefinitely
;----Time delay
DELAY:    MOV  R3,#200    ;load R3 with 200
HERE:     DJNZ R3,HERE    ;stay here until R3 become 0
          RET             ;return to caller
```

Solution:

From Table A-1 in Appendix A, we have the following machine cycles for each instruction of the DELAY subroutine.

```
                          Machine Cycle
DELAY:    MOV  R3,#200        1
HERE:     DJNZ R3,HERE        2
          RET                 2
```

Therefore, we have a time delay of $[(200 \times 2) + 1 + 2] \times 1.085\ \mu s = 436.255\ \mu s$.

Loop inside loop delay

Another way to get a large delay is to use a loop inside a loop, which is also called a *nested loop.* .

For an 8051 system of 11.0592 MHz, find the time delay for the following subroutine:

```
                          Machine Cycle
DELAY:    MOV  R3,#250        1

HERE:     NOP                 1
          NOP                 1
          NOP                 1
          NOP                 1
          DJNZ R3,HERE        2

          RET                 2
```

The time delay inside the HERE loop is [250(1 + 1 + 1 + 1 + 2)] x 1.085 us =1500 x 1.085 us = 1627.5 us. Adding the two instructions outside the loop we have 1627.5 us + 3xl.085|is=1630.755 us. If machine cycle timing is critical to your system design, make sure that you check the manufacture's data sheets for the device specification.