BIRSA INSTITUTE OF TECHNOLOGY (TRUST)

NH-33, GETLATU, RANCHI

Department: - Electronics and Communication Engineering
Lecture notes
Semester: - 4th
Subject: - Digital Technologies and Microprocessor
Lecturer: - Alok Kumar Singh

ASSEMBLER DIRECTIVES

There are some instructions in the assembly language program which are not a part of processor instruction set. These instructions are instructions to the assembler, linker and loader. These are referred to as pseudo-operations or as assembler directives. The assembler directives enable us to control the way in which a program assembles and lists. They act during the assembly of a program and do not generate any executable machine code.

There are many specialized assembler directives. Let us see the commonly used assembler directive in 8086 assembly language programming.

1. ASSUME:

It is used to tell the name of the logical segment the assembler to use for a specified segment.

E.g.: ASSUME CS: CODE tells that the instructions for a program are in a logical segment named CODE.

2. DB -Define Byte:

The DB directive is used to reserve byte or bytes of memory locations in the available memory. While preparing the EXE file, this directive directs the assembler to allocate the specified number of memory bytes to the said data type that may be a constant, variable, string, etc. Another option of this directive also initializes the reserved memory bytes with the ASCII codes of the characters specified as a string. The following examples show how the DB directive is used for different purposes.

1) RANKS DB 01H,02H,03H,04H

This statement directs the assembler to reserve four memory locations for a list named RANKS and initialize them with the above specified four values.

2) MESSAGE DB „GOOD MORNING"

This makes the assembler reserve the number of bytes of memory equal to the number of characters in the string named MESSAGE and initializes those locations by the ASCII equivalent of these characters.

3) VALUE DB 50H

This statement directs the assembler to reserve 50H memory bytes and leave them uninitialized for the variable named VALUE.

3. DD -Define Double word - used to declare a double word type variable or to reserve memory locations that can be accessed as double word.

E.g.: ARRAY _POINTER DD 25629261H declares a double word named ARRAY_POINTER.

4. DQ -Define Quad word

This directive is used to direct the assembler to reserve 4 words (8 bytes) of memory for the specified variable and may initialize it with the specified values.

E.g.: BIG_NUMBER DQ 2432987456292612H

declares a quad word named BIG_NUMBER.

5. DT -Define Ten Bytes:

The DT directive directs the assembler to define the specified variable requiring 10-bytes for its storage and initialize the 10-bytes with the specified values. The directive may be used in case of variables facing heavy numerical calculations, generally processed by numerical processors.

E.g.: PACKED_BCD 11223344556677889900 declares an array that is 10 bytes in length.

6. DW -Define Word:

The DW directives serves the same purposes as the DB directive, but it now makes the assembler reserve the number of memory words (16-bit) instead of bytes. Some examples are given to explain this directive.

1) WORDS DW 1234H, 4567H, 78ABH, 045CH

This makes the assembler reserve four words in memory (8 bytes), and initialize the words with the specified values in the statements. During initialization, the lower bytes are stored at the lower memory addresses, while the upper bytes are stored at the higher addresses.

2) NUMBER1 DW 1245H

This makes the assembler reserve one word in memory.

7. END-End of Program:

The END directive marks the end of an assembly language program. When the assembler comes across this END directive, it ignores the source lines available later on. Hence, it should be ensured that the END statement should be the last statement in the file and should not appear in between. Also, no useful program statement should lie in the file, after the END statement.

8. ENDP-End Procedure - Used along with the name of the procedure to indicate the end of a procedure.

E.g.: SQUARE_ROOT PROC: start of procedure
SQUARE_ROOT ENDP: End of procedure

9. ENDS-End of Segment:

This directive marks the end of a logical segment. The logical segments are assigned with the names using the ASSUME directive. The names appear with the ENDS directive as prefixes to mark the end of those particular segments. Whatever are the contents of the segments, they should appear in the program before ENDS. Any statement appearing after ENDS will be neglected from the segment. The structure shown below explains the fact more clearly.

DATA SEGMENT

--------------------- DATA
ENDS
ASSUME CS: CODE, DS: DATA CODE
SEGMENT

--------------------- CODE
ENDS ENDS

10. EQU-Equate - Used to give a name to some value or symbol. Each time the assembler finds the given name in the program, it will replace the name with the vale.

E.g.: CORRECTION_FACTOR EQU 03H
MOV AL, CORRECTION_FACTOR

11. EVEN - Tells the assembler to increment the location counter to the next even address if it is not already at an even address.

Used because the processor can read even addressed data in one clock cycle

12. EXTRN - Tells the assembler that the names or labels following the directive are in some other assembly module.

For example if a procedure in a program module assembled at a different time from that which contains the CALL instruction ,this directive is used to tell the assembler that the procedure is external

13. GLOBAL - Can be used in place of a PUBLIC directive or in place of an EXTRN directive.

It is used to make a symbol defined in one module available to other modules.

E.g.: GLOBAL DIVISOR makes the variable DIVISOR public so that it can be accessed from other modules.

14. GROUP-Used to tell the assembler to group the logical statements named after the directive into one logical group segment, allowing the contents of all the segments to be accessed from the same group segment base.

E.g.: SMALL_SYSTEM GROUP CODE, DATA, STACK_SEG

15. INCLUDE - Used to tell the assembler to insert a block of source code from the named file into the current source module.

This will shorten the source code.

16. LABEL- Used to give a name to the current value in the location counter.

This directive is followed by a term that specifies the type you want associated with that name.

E.g: ENTRY_POINT LABEL FAR

NEXT: MOV AL, BL

17. NAME- Used to give a specific name to each assembly module when programs consisting of several modules are written.

E.g.: NAME PC_BOARD

18. OFFSET- Used to determine the offset or displacement of a named data item or procedure from the start of the segment which contains it.

E.g.: MOV BX, OFFSET PRICES

19. ORG- The location counter is set to 0000 when the assembler starts reading a segment. The ORG directive allows setting a desired value at any point in the program.

E.g.: ORG 2000H

20. PROC- Used to identify the start of a procedure.

E.g.: SMART_DIVIDE PROC FAR identifies the start of a procedure named SMART_DIVIDE and tells the assembler that the procedure is far

21. PTR- Used to assign a specific type to a variable or to a label.

E.g.: INC BYTE PTR[BX] tells the assembler that we want to increment the byte pointed to by BX

22. PUBLIC- Used to tell the assembler that a specified name or label will be accessed from other modules.

E.g.: PUBLIC DIVISOR, DIVIDEND makes the two variables DIVISOR and DIVIDEND available to other assembly modules.

23. SEGMENT- Used to indicate the start of a logical segment.

E.g.: CODE SEGMENT indicates to the assembler the start of a logical segment called CODE

24. SHORT- Used to tell the assembler that only a 1 byte displacement is needed to code a jump instruction.

E.g.: JMP SHORT NEARBY_LABEL

25. TYPE - Used to tell the assembler to determine the type of a specified variable.

E.g.: ADD BX, TYPE WORD_ARRAY is used where we want to increment BX to point to the next word in an array of words.

Macros:

Macro is a group of instruction. The macro assembler generates the code in the program each time where the macro is “called”. Macros can be defined by MACROP and ENDM assembler directives. Creating macro is very similar to creating a new opcode that can used in the program, as shown below.

Example:

```
INIT MACRO MOV
    AX,@DATA MOV DS
MOV ES, AX ENDM
```

It is important to note that macro sequences execute faster than procedures because there is no CALL and RET instructions to execute. The assembler places the macro instructions in the program each time when it is invoked. This procedure is known as Macro expansion.

WHILE:

In Macro, the WHILE statement is used to repeat macro sequence until the expression specified with it is true. Like REPEAT, end of loop is specified by ENDM statement. The WHILE statement allows to use relational operators in its expressions.

The table-1 shows the relational operators used with WHILE statements.

OPERATOR	FUNCTION
EQ	Equal
NE	Not equal
LE	Less than or equal
LT	Less than
GE	Greater than or equal
GT	Greater than
NOT	Logical inversion
AND	Logical AND
OR	Logical OR

Table-1: Relational operators used in WHILE statement.

FOR statement:

A FOR statement in the macro repeats the macro sequence for a list of data. For example, if we pass two arguments to the macro then in the first iteration the FOR statement gives the macro sequence using first argument and in the second iteration it gives the macro sequence using second argument. Like WHILE statement, end of FOR is indicated by ENDM statement. The program shows the use of FOR statement in the macro.

Example1:

```
DISP MACRO CHR MOV AH,
    02H FOR ARG, <CHR>
    MOV DL, ARG INT 21H
ENDM ENDM
. MODEL SMALL

. CODE

START: DISP „M‟, „A‟, „C‟, „R‟, „O‟ END START
```

CODE FOR 8 BIT ADDER

```
DATA SEGMENT
    A1 DB 50H
    A2 DB 51H
    RES DB ?
DATA ENDS
CODE SEGMENT
ASSUME CS: CODE, DS:DATA
START:   MOV AX,DATA
         MOV DS,AX
         MOV AL,A1
         MOV BL,A2
         ADD AL,BL
         MOV RES,AL
         MOV AX,4C00H
         INT 21H
         CODE ENDS
         END START
```

CODE FOR 16 BIT ADDER

```
DATA SEGMENT
     A1 DW 0036H
     A2 DW 0004H
     SUM DW ?
DATA ENDS
CODE SEGMENT
ASSUME   CS:CODE,DS:DATA
  START: MOV AX,DATA
         MOV DS,AX
         MOV AX,A1
         MOV BX,A2
         DIV BX
         MOV SUM,AX
         MOV AX,0008H
         INT 21H
         CODE ENDS
         END START
```

ADD33 MATRIX

```
        .MODEL SMALL
        .DATA
        M1 DB 10H,20H,30H,40H,50H,60H,70H,80H,90H
        M2 DB 10H,20H,30H,40H,50H,60H,70H,80H,90H
        RESULT DW 9 DUP (0)
        .CODE
START: MOV AX,@DATA
        MOV DS,AX
        MOV CX,9
        MOV DI,OFFSET M1
        MOV BX,OFFSET M2
        MOV SI,OFFSET
        RESULT
BACK: MOV AH,00
        MOV AL,[DI]
        ADD AL,[BX]
        ADC AH,00
        MOV [SI],AX
        INC DI
        INC BX
        INC SI
        INC SI
        LOOP BACK
        MOV AH,4CH
        INT 21H
        END START
        END
```

ARRAY SUM

```
        .MODEL SMALL
        .DATA
        ARRAY DB 12H, 24H, 26H, 63H, 25H, 86H, 2FH, 33H, 10H, 35H
        SUM DW 0
        .CODE
        START:MOV AX, @DATA
              MOV DS, AX
              MOV CL, 10
              XOR DI, DI
              LEA BX, ARRAY
      BACK:  MOV AL, [BX+DI]
              MOV AH, 00H
              ADD SUM, AX
              INC DI
              DEC CL
```

```
JNZ BACK
 END START
```

ASCIITOHEX

```
DATA SEGMENT
    A DB 41H
    R DB ?
DATA ENDS
CODE SEGMENT
ASSUME CS: CODE, DS:DATA
    START:     MOV AX,DATA
               MOV DS,AX
               MOV AL,A
               SUB AL,30H
               CMP AL,39H
               JBE L1
               SUB AL,7H
           L1: MOV R,AL
               INT 3H
CODE ENDS
END START
```

AVERAGE

```
.MODEL SMALL
.STACK 100
.DATA
    NO1  DB  63H
    NO2  DB  2EH
    AVG DB ?
.CODE
START: MOV AX,@DATA
        MOV DS,AX
       MOV AL,NO1
       ADD AL,NO2
       ADC AH,00H
       SAR AX,1
       MOV AVG,AL
END START
```

16 BIT SUB

```
DATA SEGMENT
    A1 DW 1001H
    A2 DW 1000H
    SUB DW ?
DATA ENDS
CODE SEGMENT
ASSUME CS:CODE,DS:DATA
START:  MOV AX,DATA
        MOV DS,AX
        MOV AX,A1
        MOV BX,A2
        SBB AX,BX
        MOV SUB,AX
        MOV AX,4C00H
        INT 21H
CODE ENDS
END START
```

16BIT SUM

```
DATA SEGMENT
    A1 DW 1000H
    A2 DW 1001H
    SUM DW ?
DATA ENDS
CODE SEGMENT
ASSUME CS:CODE,DS:DATA
START:  MOV AX,DATA
        MOV DS,AX
        MOV AX,A1
        MOV BX,A2
        ADC AX,BX
        MOV SUM,AX
        MOV AX,4C00H
        INT 21H
CODE ENDS
END START
```

8BMUL

```
DATA SEGMENT
    A1 DB 25H
    A2 DB 25H
    A3 DB ?
DATA ENDS
CODE SEGMENT
ASSUME CS: CODE, DS:DATA
    START:MOV AX,DATA
          MOV DS,AX
          MOV AL,A1
          MOV BL,A2
          MUL BL
          MOV A3,AL
          MOV AX,4C00H
          INT 21H
CODE ENDS
END START
```

16BIT MUL

```
DATA SEGMENT
    A1 DW 1000H
    A2 DW 1000H
    A3 DW ?
    A4 DW ?
DATA ENDS
CODE SEGMENT
ASSUME CS: CODE, DS:DATA
START:MOV AX,DATA
        MOV DS,AX
        MOV AX,A1
        MOV BX,A2
        MUL BX
        MOV A3,DX
        MOV A4,AX
        MOV AX,4C00H
        INT 21H
CODE ENDS
END START
```

EVENODD

```
DATA SEGMENT
        ORG 2000H
        FIRST DW 3H
DATA ENDS
CODE SEGMENT
ASSUME CS:CODE,DS:DATA
START: MOV AX,DATA
       MOV DS,AX
       MOV AX,FIRST
       SHR AX,1
       JC L1
       MOV BX,00
       INT 3H
    L1: MOV BX,01
       INT 3H
CODE ENDS
END START
```

FACTORIAL

```
DATA SEGMENT
     ORG  2000H
     FIRST DW 3H
     SEC DW 1H
DATA ENDS
CODE SEGMENT
ASSUME CS:CODE,DS:DATA
START: MOV AX,DATA
       MOV DS,AX
       MOV AX,SEC
       MOV CX,FIRST
   L1:   MUL CX
       DEC CX
       JCXZ L2
       JMP L1
   L2:   INT 3H
CODE ENDS
END START
```

FIBONOCCI

```
DATA SEGMENT
    ORG 2000H
    FIRST DW 0H
    SEC DW 01H
    THIRD DW 50H
    RESULT DW ?
DATA ENDS
CODE SEGMENT
ASSUME CS: CODE, DS:DATA
START:    MOV AX,DATA
          MOV DS,AX
          MOV SI,OFFSET RESULT
          MOV AX,FIRST
          MOV BX,SEC
          MOV CX,THIRD
          MOV [SI],AX
      L1: INC SI
          INC SI
          MOV [SI],BX
          ADD AX,BX
          XCHG AX,BX
          CMP BX,CX
          INT 3H
  CODE ENDS
  END START
```

FIND NUMBER

```
.MODEL SMALL
.STACK 100
.DATA
ARRAY DB 63H,32H,45H,75H,12H,42H,09H,14H,56H,38H
SER_NO DB 09H
SER_POS DB ?
.CODE
START:MOV AX,@DATA
      MOV DS,AX
      MOV ES,AX
      MOV CX,000AH
      LEA DI,ARRAY
      MOV AL,SER_NO
```

```
        CLD
        REPNE SCAS ARRAY
        MOV AL,10
        SUB AL CL
        MOV SER_POS,AL
    END START
```

GREATER

```
DATA SEGMENT
ORG 2000H
FIRST     DW     5H,2H,3H,1H,4H
    COUNT  EQU  (($-FIRST)/2)-1
    DATA ENDS
CODE SEGMENT
ASSUME CS: CODE, DS:DATA
    START:  MOV AX,DATA
            MOV DS,AX
            MOV CX,COUNT
            MOV SI,OFFSET FIRST
            MOV AX,[SI]
        L2: INC SI
              INC SI
              MOV BX,[SI]
              CMP AX,BX
              JGE L1
              XCHG AX,BX
              JMP L1
        L1:   DEC CX
             JCXZ L4
             JMP L2
        L4:  INT 3H
    CODE ENDS
    END START
```

HEX TO ASCII

```
DATA SEGMENT
      A DB 08H
      C DB ?
DATA ENDS
CODE SEGMENT
ASSUME CS: CODE, DS: DATA
START: MOV AX,DATA
             MOV DS,AX
             MOV AL,A
             ADD AL,30H
             CMP AL,39H
             JBE L1
             ADD AL,7H
    L1:      MOV C,AL
             INT 3H
CODE ENDS
END START
```

MAX

```
.MODEL SMALL
.STACK 100
.DATA
ARRAY DB 63H,32H,45H,75,12H,42H,09H,14H,56H,38H
MAX DB 0
.CODE
START:MOV AX,@DATA
            MOV DS,AX
            XOR DI,DI
            MOV CL,10
            LEA BX,ARRAY
            MOV AL,MAX
BACK:       CMP AL,[BX+DI]
            JNC SKIP
            MOV DL,[BX+DI]
            MOV AL,DL
SKIP:       INC DI
            DEC CL
            JNZ BACK
            MOV MAX,AL
            MOV AX,4C00H
            INT 21H
    END START
```

NO OF 1S

```
DATA SEGMENT
      ORG 2000H
      FIRST DW 7H
      DATA ENDS
      CODE SEGMENT
ASSUME CS: CODE, DS: DATA
      START: MOV AX,DATA
                MOV DS,AX
                MOV AX,FIRST
                MOV BX,00
                MOV CX,16
            L2:   SHR AX,1
                JC L1
            L4:   DEC CX
                JCXZ L3
               JMP L2
            L1: INC BX
               JMP L4
            L3: INT 3H
 CODE ENDS
 END START
```

SMALLER

```
DATA SEGMENT
ORG 2000H
      FIRST DW 5H,2H,3H,1H,4H
      COUNT EQU (($-FIRST)/2)-1
      RESULT DW ?
DATA ENDS
CODE SEGMENT
ASSUME CS: CODE, DS:DATA
START: MOV AX,DATA
          MOV DS,AX
          MOV CX,COUNT
          MOV SI,OFFSET FIRST
          MOV AX,[SI]
      L2:  INC SI
          INC SI
          MOV BX,[SI]
          CMP AX,BX
          JB L1
          XCHG AX,BX
          JMP L1
       L1: DEC CX
```

```
JCXZ L4
JMP L2
L4:  MOV RESULT,AX
CODE ENDS
END START
```

SUM OF CUBES

```
DATA SEGMENT
        ORG 2000H
        NUM DB 1H
        RES DW ?
DATA ENDS
CODE SEGMENT
ASSUME CS: CODE, DS: DATA
START: MOV DX,DATA
        MOV DS,AX
        MOV CL,NUM
        MOV BX,00
    L1: MOV AL,CL
        MOV CH,CL MUL AL
        MUL CH
        ADD BX,AX
        DEC CL
        JNZ L1
        MOV RES,BX
        INT 3H
CODE ENDS
END START
```

SUM OF SQUARES

```
DATA SEGMENT
NUM DW 5H
RES DW ?
DATA ENDS
CODE SEGMENT
ASSUME CS: CODE, DS: DATA
START:    MOV AX,DATA
          MOV DS,AX MOV
          CX,NUM MOV BX,00
     L1:  MOV AX,CX
          MUL CX
          ADD BX,AX
          DEC CX
          JNZ L1
          MOV RES,BX
          INT 3H
 CODE ENDS
 END START
```