

LAB DIGITAL ASSIGNMENT 1

NAME: Ritik Gupta

REG. No: 18BCE0154

QUES:1 Write a Summary on Flag register, addressing mode, Assembler directive and a neat sketch of 8086 Architecture.

1.FLAG REGISTER

The Flag register is a Special Purpose Register. Depending upon the value of result after any arithmetic and logical operation the flag bits become set (1) or reset (0).

D ₁₅	D ₁₄	D ₁₃	D ₁₂	D ₁₁	D ₁₀	D ₉	D ₈	D ₇	D ₆	D ₅	D ₄	D ₃	D ₂	D ₁	D ₀
				O	D	I	T	S	Z		AC		P		CY

(a) **Status Flags** – There are 6 flag registers in 8086 microprocessor which become set(1) or reset(0) depending upon condition after either 8-bit or 16-bit operation. These flags are conditional/status flags. 5 of these flags are same as in case of 8085 microprocessor and their working is also same as in 8085 microprocessor. The sixth one is the overflow flag.

The 6 status flags are:

1. **Sign Flag (S)** -It is set if the MSB of the result is 1. For signed operations such a number is treated as negative.

2. **Zero Flag (Z)** -It is set if the result is zero.
3. **Auxiliary Carry Flag (AC)** -It is set if a carry is generated out of the lower nibble. It is used only in 8 bit operations like DAA and DAS.
4. **Parity Flag (P)** -It is set if the result has even parity. If parity is odd, PF is reset. This flag is normally used for data transmission errors.
5. **Carry Flag (CY)**-It is set whenever there is a carry or borrow out of the MSB (most significant bit) of a result. D7 bit for an 8 bit operation and D15 bit for a 16 bit operation.
6. **Overflow Flag (O)** – This flag will be set (1) if the result of a signed operation is too large to fit in the number of bits available to represent it, otherwise reset (0). After any operation, if D[6] generates any carry and passes to D[7] OR if D[6] does not generate carry but D[7] generates, overflow flag becomes set, i.e., 1. If D[6] and D[7] both generate carry or both do not generate any carry, then overflow flag becomes reset, i.e., 0.

(b) Control Flags – The control flags enable or disable certain operations of the microprocessor. There are 3 control flags in 8086 microprocessor and these are:

1. **Directional Flag (D)** – This flag is specifically used in string instructions.
If directional flag is set (1), then access the string data from higher memory location towards lower memory location.
If directional flag is reset (0), then access the string data from lower memory location towards higher memory location.
2. **Interrupt Flag (I)** – This flag is for interrupts.
If interrupt flag is set (1), the microprocessor will recognize interrupt requests from the peripherals.
If interrupt flag is reset (0), the microprocessor will not recognize any interrupt requests and will ignore them.
3. **Trap Flag (T)** – This flag is used for on-chip debugging. Setting trap flag puts the microprocessor into single step mode for debugging. In single stepping, the microprocessor executes a instruction and enters into single step ISR.
If trap flag is set (1), the CPU automatically generates an internal interrupt after each instruction, allowing a program to be inspected as it executes instruction

by instruction.

If trap flag is reset (0), no function is performed.

2.Addressing mode

The way of specifying data to be operated by an instruction is known as **addressing modes**. This specifies that the given data is an immediate data or an address. It also specifies whether the given operand is register or register pair.

Types of addressing modes:

1. **Register mode** – In this type of addressing mode both the operands are registers.

Example:

```
MOV AX, BX
XOR AX, DX
ADD AL, BL
```

2. **Immediate mode** – In this type of addressing mode the source operand is a 8 bit or 16 bit data. Destination operand can never be immediate data.

Example:

```
MOV AX, 2000
MOV CL, 0A
ADD AL, 45
AND AX, 0000
```

Note that to initialize the value of segment register an register is required.

```
MOV AX, 2000
MOV CS, AX
```

3. **Displacement or direct mode** – In this type of addressing mode the effective address is directly given in the instruction as displacement.

Example:

```
MOV AX, [DISP]
MOV AX, [0500]
```

4. **Register indirect mode** – In this addressing mode the effective address is in SI, DI or BX.

Example:

```
MOV AX, [DI]
ADD AL, [BX]
MOV AX, [SI]
```

5. **Based indexed mode** – In this the effective address is sum of base register and index register.

Base register: BX, BP
Index register: SI, DI

The physical memory address is calculated according to the base register.

Example:

```
MOV AL, [BP+SI]
MOV AX, [BX+DI]
```

6. **Indexed mode** – In this type of addressing mode the effective address is sum of index register and displacement.

Example:

```
MOV AX, [SI+2000]
MOV AL, [DI+3000]
```

7. **Based mode** – In this the effective address is the sum of base register and displacement.

Example:

```
MOV AL, [BP+ 0100]
```

8. **Based indexed displacement mode** – In this type of addressing mode the effective address is the sum of index register, base register and displacement.

Example:

```
MOV AL, [SI+BP+2000]
```

9. **String mode** – This addressing mode is related to string instructions. In this the value of SI and DI are auto incremented and decremented depending upon the value of directional flag.

Example:

```
MOVS B  
MOVS W
```

10. **Input/Output mode** – This addressing mode is related with input output operations.

Example:

```
IN A, 45  
OUT A, 50
```

11. **Relative mode** – In this the effective address is calculated with reference to instruction pointer.

Example:

```
JNZ 8 bit address  
IP=IP+8 bit address
```

12. **Implied mode**:: In implied addressing the operand is specified in the instruction itself. In this mode the data is 8 bits or 16 bits long and data is the part of instruction. Zero address instruction are designed with implied addressing mode.

Example: CLC (used to reset Carry flag to 0)

13. **Auto Indexed (increment mode)**: Effective address of the operand is the contents of a register specified in the instruction. After accessing the operand, the contents of this register are automatically incremented to point to the next consecutive memory location. **(R1)+**.

Here one register reference, one memory reference and one ALU operation is required to access the data.

Example:

Add R1, (R2)+ // OR
 $R1 = R1 + M[R2]$
 $R2 = R2 + d$

Useful for stepping through arrays in a loop. R2 – start of array d – size of an element

14. **Auto indexed (decrement mode):** Effective address of the operand is the contents of a register specified in the instruction. Before accessing the operand, the contents of this register are automatically decremented to point to the previous consecutive memory location. **–(R1)**
Here one register reference, one memory reference and one ALU operation is required to access the data.

Example:

Add R1, -(R2) //OR
 $R2 = R2 - d$
 $R1 = R1 + M[R2]$

Auto decrement mode is same as auto increment mode. Both can also be used to implement a stack as push and pop. Auto increment and Auto decrement modes are useful for implementing “Last-In-First-Out” data structures.

3.Assembler directive

Assembly languages are low-level languages for programming computers, microprocessors, microcontrollers, and other IC. They implement a symbolic representation of the numeric machine Codes and other constants needed to program a particular CPU architecture. This representation is usually defined by the hardware manufacturer, and is based on abbreviations that help the programmer to remember individual instructions, registers. An assembler directive is a statement to give direction to the assembler to perform task of the assembly process.

It control the organization if the program and provide necessary information to the assembler to understand the assembly language programs to generate necessary machine codes. They indicate how an operand or a section of the program is to be processed by the assembler.

An assembler supports directives to define data, to organise segments to control procedure, to define macros. It consists of two types of statements: instructions and directives. The instructions are translated to the machine code by the assembler whereas directives are not translated to the machine codes.

Assembler Directives of the 8086 Microprocessor

- (a) The DB directive
- (b) The DW directive
- (c) The DD directive
- (d) The STRUCT (or STRUC) and ENDS directives (counted as one)
- (e) The EQU Directive
- (f) The COMMENT directive
- (g) ASSUME
- (h) EXTERN
- (i) GLOBAL
- (j) SEGMENT
- (k) OFFSET
- (l) PROC
- (m) GROUP
- (n) INCLUDE

Data declaration directives:

1. DB – The DB directive is used to declare a BYTE -2-BYTE variable – A BYTE is made up of 8 bits.

Declaration examples:

Byte1 DB 10h

Byte2 DB 255 ; 0FFh, the max. possible for a BYTE

CRLF DB 0Dh, 0Ah, 24h ; Carriage Return, terminator BYTE

2. DW – The DW directive is used to declare a WORD type variable – A WORD occupies 16 bits or (2 BYTE).

Declaration examples:

Word DW 1234h

Word2 DW 65535; 0FFFFh, (the max. possible for a WORD)

3. DD – The DD directive is used to declare a DWORD – A DWORD double word is made up of 32 bits =2 Word's or 4 BYTE.

Declaration examples:

Dword1 DW 12345678h

Dword2 DW 4294967295 ;0FFFFFFFFh.

4. STRUCT and ENDS directives to define a structure template for grouping data items.

(1) The STRUCT directive tells the assembler that a user defined uninitialized data structure follows. The uninitialized data structure consists of a combination of the three supported data types. DB, DW, and DD. The labels serve as zero-based offsets into the structure. The first element's offset for any structure is 0. A structure element is referenced with the base "+" operator before the element's name.

A Structure ends by using the ENDS directive meaning END of Structure.

Syntax:

STRUCT

Structure_element_name element_data_type?

...

...

...

ENDS

(OR)

STRUC

Structure_element_name element_data_type?

...

...

...

ENDS

DECLARATION:

STRUCT

Byte1 DB?

Byte2 DB?

Word1 DW?

Word2 DW?

Dword1DW?

Dword2 DW?

ENDS

Use OF STRUCT:

The STRUCT directive enables us to change the order of items in the structure when, we reform a file header and shuffle the data. Shuffle the data items in the file header and reformat the sequence of data declaration in the STRUCT and off you go. No change in the code we write that processes the file header is necessary unless you inserted an extra data element.

(5) The EQU Directive

The EQU directive is used to give name to some value or symbol. Each time the assembler finds the given names in the program, it will replace the name with the value or a symbol. The value can be in the range 0 through 65535 and it can be another Equate declared anywhere above or below.

The following operators can also be used to declare an Equate:

THIS BYTE

THIS WORD

THIS DWORD

A variable – declared with a DB, DW, or DD directive – has an address and has space reserved at that address for it in the .COM file. But an Equate does not have an address or space reserved for it in the .COM file.

Example:

A – Byte EQU THIS BYTE

DB 10

A_ word EQU THIS WORD

DW 1000

A_ dword EQU THIS DWORD

DD 4294967295

Buffer Size EQU 1024

Buffer DB 1024 DUP (0)

Bufed_ptr EQU \$; actually points to the next byte after the; 1024th byte in buffer.

(6) Extern:

It is used to tell the assembler that the name or label following the directive are in some other assembly module. For example: if you call a procedure which is in program module assembled at a different time from that which contains the CALL instructions, you must tell the assembler that the procedure is external the assembler will put information in the object code file so that the linker can connect the two module together.

Example:

```
PROCEDURE -HERE SEGMENT
```

```
EXTERN SMART-DIVIDE: FAR ; found in the segment; PROCEDURES-HERE
```

```
PROCEDURES-HERE ENDS
```

(7) GLOBAL:

The GLOBAL directive can be used in place of PUBLIC directive. For a name defined in the current assembly module; the GLOBAL directive is used to make the symbol available to the other modules. Example:

GLOBAL DIVISOR:

WORD tells the assembler that DIVISOR is a variable of type of word which is in another assembly module or EXTERN.

(8) SEGMENT:

It is used to indicate the start of a logical segment. It is the name given to the segment. Example: the code segment is used to indicate to the assembler the start of logical segment.

(9) PROC: (PROCEDURE)

It is used to identify the start of a procedure. It follows a name we give the procedure.

After the procedure the term NEAR and FAR is used to specify the procedure

Example: SMART-DIVIDE PROC FAR identifies the start of procedure named SMART-DIVIDE and tells the assembler that the procedure is far.

(10) NAME:

It is used to give a specific name to each assembly module when program consists of several modules.

Example: PC-BOARD used to name an assembly module which contains the instructions for controlling a printed circuit board.

(11) INCLUDE:

It is used to tell the assembler to insert a block of source code from the named file into the current source module. This shortens the source module. An alternative is use of editor block command to copy the file into the current source module.

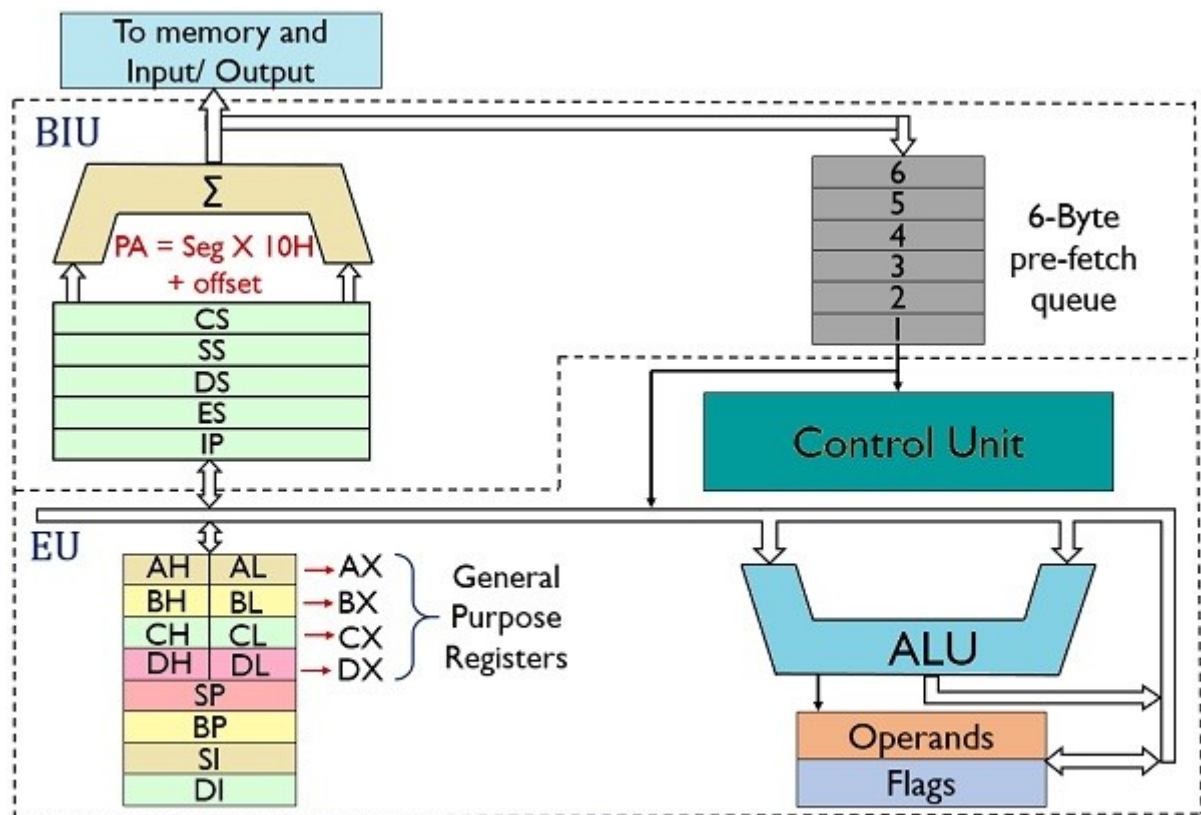
(12) OFFSET:

It is an operator which tells the assembler to determine the offset or displacement of a named data item from the start of the segment which contains it. It is used to load the offset of a variable into a register so that variable can be accessed with one of the addressed modes. Example: when the assembler read `MOV BX,OFFSET PRICES`, it will determine the offset of the prices.

(13) GROUP:

It can be used to tell the assembler to group the logical segments named after the directive into one logical group. This allows the contents of all the segments to be accessed from the same group. Example: `SMALL-SYSTEM GROUP CODE, DATA, STACK-SEG`.

4.Neat sketch of 8086 Architecture.



Block Diagram of 8086 Microprocessor

Ques 2. Write an ALP program to perform the 8 bit add, sub and multiplication[Direct and Immediate]

8 Bit Add Direct

.model small

.stack 64h

.data

.code

mov ax,@data

mov ds,ax

start:

mov AX,ds:[0500h]

mov BX,ds:[0600h]

ADD AX,BX

mov ah,4ch

int 21h

end

.end

;18BCE0154

Screenshot Of Code

```

.model small
.stack 64h
.data
.code
mov ax,@data
mov ds,ax
start:
mov AX,ds:[0500h]
mov BX,ds:[0600h]
ADD AX,BX
mov ah,4ch
int 21h
end
.end
;18BCE0154

```

OUTPUT:

```

List File [NUL.MAP]:
Libraries [.LIB]:

C:\MASM>debug add172.exe
-t
AX=0745 BX=0000 CX=0012 DX=0000 SP=0064 BP=0000 SI=0000 DI=0000
DS=0734 ES=0734 SS=0746 CS=0744 IP=0003 NU UP EI PL ZR NA PE NC
0744:0003 8ED8          MOV     DS,AX
-t
AX=0745 BX=0000 CX=0012 DX=0000 SP=0064 BP=0000 SI=0000 DI=0000
DS=0745 ES=0734 SS=0746 CS=0744 IP=0005 NU UP EI PL ZR NA PE NC
0744:0005 A10005      MOV     AX,[0500]          DS:0500=DF8B
-e DS:0500
0745:0500 8B.04   DF.00   D1.
-t
AX=0004 BX=0000 CX=0012 DX=0000 SP=0064 BP=0000 SI=0000 DI=0000
DS=0745 ES=0734 SS=0746 CS=0744 IP=0008 NU UP EI PL ZR NA PE NC
0744:0008 8B1E0006     MOV     BX,[0600]          DS:0600=5048
-e DS:0600
0745:0600 48.06   50.00   8D.
-t
AX=0004 BX=0006 CX=0012 DX=0000 SP=0064 BP=0000 SI=0000 DI=0000
DS=0745 ES=0734 SS=0746 CS=0744 IP=000C NU UP EI PL ZR NA PE NC
0744:000C 03C3          ADD     AX,BX
-

```

```

-t
AX=0004 BX=0006 CX=0012 DX=0000 SP=0064 BP=0000 SI=0000 DI=0000
DS=0745 ES=0734 SS=0746 CS=0744 IP=000C NV UP EI PL ZR NA PE NC
0744:000C 03C3          ADD     AX,BX
-t
AX=000A BX=0006 CX=0012 DX=0000 SP=0064 BP=0000 SI=0000 DI=0000
DS=0745 ES=0734 SS=0746 CS=0744 IP=000E NV UP EI PL NZ NA PE NC
0744:000E B44C          MOV     AH,4C
-s_

```

8 Bit Add Immediate

CODE:

```

%18BCE0154
.model small
.stack 64
.data
.code

mov ax,@data
mov ds,ax
start:
MOV AX,04H
MOV BX,05H
ADD AX,BX
mov ah,4ch
int 21h
end
.end

```

OUTPUT:

```
C:\MASM>debug add1.exe
-t 5
AX=0745 BX=0000 CX=0011 DX=0000 SP=0040 BP=0000 SI=0000 DI=0000
DS=0734 ES=0734 SS=0746 CS=0744 IP=0003 NU UP EI PL ZR NA PE NC
0744:0003 8ED8          MOV     DS,AX
AX=0745 BX=0000 CX=0011 DX=0000 SP=0040 BP=0000 SI=0000 DI=0000
DS=0745 ES=0734 SS=0746 CS=0744 IP=0005 NU UP EI PL ZR NA PE NC
0744:0005 B80400      MOV     AX,0004
AX=0004 BX=0000 CX=0011 DX=0000 SP=0040 BP=0000 SI=0000 DI=0000
DS=0745 ES=0734 SS=0746 CS=0744 IP=0008 NU UP EI PL ZR NA PE NC
0744:0008 BB0500      MOV     BX,0005
AX=0004 BX=0005 CX=0011 DX=0000 SP=0040 BP=0000 SI=0000 DI=0000
DS=0745 ES=0734 SS=0746 CS=0744 IP=000B NU UP EI PL ZR NA PE NC
0744:000B 03C3        ADD     AX,BX
AX=0009 BX=0005 CX=0011 DX=0000 SP=0040 BP=0000 SI=0000 DI=0000
DS=0745 ES=0734 SS=0746 CS=0744 IP=000D NU UP EI PL NZ NA PE NC
0744:000D B44C        MOV     AH,4C
-r AX
AX 0009 :           //18BCE0154
```

8 Bit Sub Direct

.model small

.stack 64h

.data

.code

mov ax,@data

mov ds,ax

start:


```
mov AX,ds:[0502h]
```

```
mov BX,ds:[0504h]
```

```
SUB AX,BX
```

```
mov ah,4ch
```

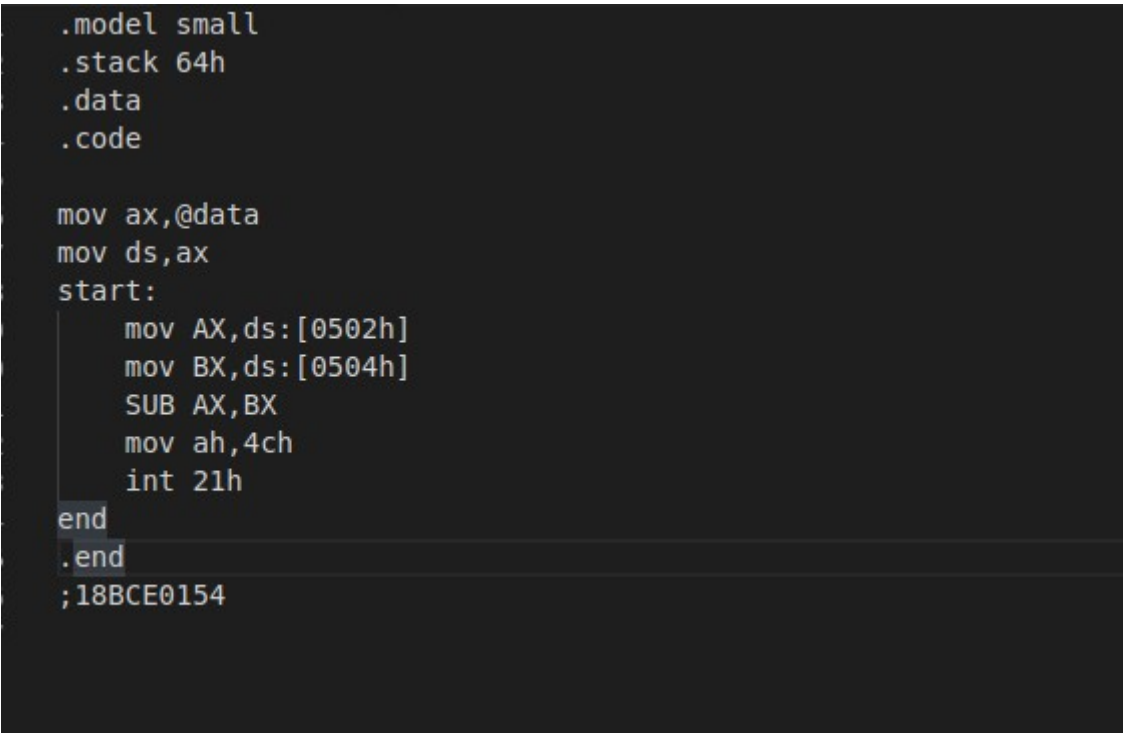
```
int 21h
```

```
end
```

```
.end
```

```
;18BCE0154
```

Screenshot of code:

A screenshot of assembly code in a dark-themed editor. The code is as follows:

```
.model small
.stack 64h
.data
.code

mov ax,@data
mov ds,ax
start:
    mov AX,ds:[0502h]
    mov BX,ds:[0504h]
    SUB AX,BX
    mov ah,4ch
    int 21h
end
.end
;18BCE0154
```

OUTPUT:

List File [NUL.MAP]:

Libraries [.LIB]:

C:\MASM>debug subdir.exe

-t

AX=0745 BX=0000 CX=0012 DX=0000 SP=0064 BP=0000 SI=0000 DI=0000
DS=0734 ES=0734 SS=0746 CS=0744 IP=0003 NV UP EI PL ZR NA PE NC
0744:0003 8ED8 MOV DS,AX

-t

AX=0745 BX=0000 CX=0012 DX=0000 SP=0064 BP=0000 SI=0000 DI=0000
DS=0745 ES=0734 SS=0746 CS=0744 IP=0005 NV UP EI PL ZR NA PE NC
0744:0005 A10205 MOV AX,[0502]

DS:0502=E3D1

-e DS:0502

0745:0502 D1.06 E3.00 03.

-t

AX=0006 BX=0000 CX=0012 DX=0000 SP=0064 BP=0000 SI=0000 DI=0000
DS=0745 ES=0734 SS=0746 CS=0744 IP=0008 NV UP EI PL ZR NA PE NC
0744:0008 8B1E0405 MOV BX,[0504]

DS:0504=1E03

-e DS:0504

0745:0504 03.04 1E.00 F0.

-t

AX=0006 BX=0004 CX=0012 DX=0000 SP=0064 BP=0000 SI=0000 DI=0000
DS=0745 ES=0734 SS=0746 CS=0744 IP=000C NV UP EI PL ZR NA PE NC
0744:000C 2BC3 SUB AX,BX

-

-t

AX=0006 BX=0004 CX=0012 DX=0000 SP=0064 BP=0000 SI=0000 DI=0000
DS=0745 ES=0734 SS=0746 CS=0744 IP=000C NV UP EI PL ZR NA PE NC
0744:000C 2BC3 SUB AX,BX

-t

AX=0002 BX=0004 CX=0012 DX=0000 SP=0064 BP=0000 SI=0000 DI=0000
DS=0745 ES=0734 SS=0746 CS=0744 IP=000E NV UP EI PL NZ NA PO NC
0744:000E B44C MOV AH,4C

-s_

8 Bit Sub Immediate

Code:

```
%18BCE0154
.model small
.stack 64
.data
.code

mov ax,@data
mov ds,ax
start:
MOV AX,09H
MOV BX,02H
SUB AX,BX
mov ah,4ch
int 21h
end
.end
```

OUTPUT:

```
C:\MASM>debug sub1.exe
-t 5
AX=0745 BX=0000 CX=0011 DX=0000 SP=0040 BP=0000 SI=0000 DI=0000
DS=0734 ES=0734 SS=0746 CS=0744 IP=0003 NU UP EI PL ZR NA PE NC
0744:0003 8ED8          MOV     DS,AX
AX=0745 BX=0000 CX=0011 DX=0000 SP=0040 BP=0000 SI=0000 DI=0000
DS=0745 ES=0734 SS=0746 CS=0744 IP=0005 NU UP EI PL ZR NA PE NC
0744:0005 B80900     MOV     AX,0009
AX=0009 BX=0000 CX=0011 DX=0000 SP=0040 BP=0000 SI=0000 DI=0000
DS=0745 ES=0734 SS=0746 CS=0744 IP=0008 NU UP EI PL ZR NA PE NC
0744:0008 BB0200     MOV     BX,0002
AX=0009 BX=0002 CX=0011 DX=0000 SP=0040 BP=0000 SI=0000 DI=0000
DS=0745 ES=0734 SS=0746 CS=0744 IP=000B NU UP EI PL ZR NA PE NC
0744:000B 2BC3          SUB     AX,BX
AX=0007 BX=0002 CX=0011 DX=0000 SP=0040 BP=0000 SI=0000 DI=0000
DS=0745 ES=0734 SS=0746 CS=0744 IP=000D NU UP EI PL NZ NA PO NC
0744:000D B44C          MOV     AH,4C
-r AX
AX 0007  :
-r BX
BX 0002  :
-%18BCE0154_
```

8 Bit Multiplication Direct

.model small

.stack 64h

.data

.code

mov ax,@data

mov ds,ax

start:

 mov AX,ds:[0700h]

 mov BX,ds:[0702h]

 MUL BX

 mov ah,4ch

 int 21h

end

.end

;18BCE0154

Screenshot Of Code:

```
.model small
.stack 64h
.data
.code

mov ax,@data
mov ds,ax
start:
    mov AX,ds:[0700h]
    mov BX,ds:[0702h]
    MUL BX
    mov ah,4ch
    int 21h
end
.end
;18BCE0154
```

OUTPUT:

```
List File [NUL.MAP]:
Libraries [.LIB]:

C:\MASM>debug muldir.exe
-t
AX=0745 BX=0000 CX=0012 DX=0000 SP=0064 BP=0000 SI=0000 DI=0000
DS=0734 ES=0734 SS=0746 CS=0744 IP=0003 NU UP EI PL ZR NA PE NC
0744:0003 8ED8          MOV     DS,AX
-t
AX=0745 BX=0000 CX=0012 DX=0000 SP=0064 BP=0000 SI=0000 DI=0000
DS=0745 ES=0734 SS=0746 CS=0744 IP=0005 NU UP EI PL ZR NA PE NC
0744:0005 A10007       MOV     AX,[0700]          DS:0700=C700
-e DS:0700
0745:0700 00.05    C7.00    06.
-t
AX=0005 BX=0000 CX=0012 DX=0000 SP=0064 BP=0000 SI=0000 DI=0000
DS=0745 ES=0734 SS=0746 CS=0744 IP=0008 NU UP EI PL ZR NA PE NC
0744:0008 8B1E0207     MOV     BX,[0702]          DS:0702=7C06
-e DS:0702
0745:0702 06.04    7C.00    5D.
-t
AX=0005 BX=0004 CX=0012 DX=0000 SP=0064 BP=0000 SI=0000 DI=0000
DS=0745 ES=0734 SS=0746 CS=0744 IP=000C NU UP EI PL ZR NA PE NC
0744:000C F7E3          MUL     BX
```

```

-t
AX=0005 BX=0004 CX=0012 DX=0000 SP=0064 BP=0000 SI=0000 DI=0000
DS=0745 ES=0734 SS=0746 CS=0744 IP=000C NV UP EI PL ZR NA PE NC
0744:000C F7E3          MUL     BX
-t
AX=0014 BX=0004 CX=0012 DX=0000 SP=0064 BP=0000 SI=0000 DI=0000
DS=0745 ES=0734 SS=0746 CS=0744 IP=000E NV UP EI PL NZ NA PE NC
0744:000E B44C          MOV     AH,4C

```

8 Bit Multiplication Immediate

Code:

```

%18BCE0154
.model small
.stack 64
.data
.code

mov ax,0data
mov ds,ax
start:
MOV AX,09H
MOV BX,02H
MUL BX
int 3
end
.end

```

OUTPUT:

```

C:\MASM>debug mul1.exe
-t 5
AX=0744 BX=0000 CX=000E DX=0000 SP=0040 BP=0000 SI=0000 DI=0000
DS=0734 ES=0734 SS=0745 CS=0744 IP=0003 NV UP EI PL ZR NA PE NC
0744:0003 8ED8          MOV     DS,AX
AX=0744 BX=0000 CX=000E DX=0000 SP=0040 BP=0000 SI=0000 DI=0000
DS=0744 ES=0734 SS=0745 CS=0744 IP=0005 NV UP EI PL ZR NA PE NC
0744:0005 B80900      MOV     AX,0009
AX=0009 BX=0000 CX=000E DX=0000 SP=0040 BP=0000 SI=0000 DI=0000
DS=0744 ES=0734 SS=0745 CS=0744 IP=0008 NV UP EI PL ZR NA PE NC
0744:0008 BB0200      MOV     BX,0002
AX=0009 BX=0002 CX=000E DX=0000 SP=0040 BP=0000 SI=0000 DI=0000
DS=0744 ES=0734 SS=0745 CS=0744 IP=000B NV UP EI PL ZR NA PE NC
0744:000B F7E3      MJL     BX
AX=0012 BX=0002 CX=000E DX=0000 SP=0040 BP=0000 SI=0000 DI=0000
DS=0744 ES=0734 SS=0745 CS=0744 IP=000D NV UP EI PL NZ NA PE NC
0744:000D CC          INT     3
-r AX
AX 0012 :
-r BX
BX 0002 :
-~18BCE0154_

```

Ques 3. Write a program in ALP to add the data byte located at offset 0500h in 3000h segment to another data byte available at 0502h in the same segment and store the result at 0504h in the same segment.

```
.model small
```

```
.stack 64h
```

```
.data
```

```
.code
```

```
mov ax,@data
```

```
mov ds,ax
```

```
start:
```

```
    mov CX,3000h
```

```
    mov DS,CX
```

```
    mov AX,ds:[0500h]
```

```
mov BX,ds:[0502h]
```

```
ADD AX,BX
```

```
mov ds:[0504h],AX
```

```
HLT
```

```
end start
```

```
.end
```

```
;18BCE0154
```


Screenshot Of Code:

```
.model small
.stack 64h
.data
.code

mov ax,@data
mov ds,ax
start:
    mov CX,3000h
    mov DS,CX
    mov AX,ds:[0500h]
    mov BX,ds:[0502h]
    ADD AX,BX
    mov ds:[0504h],AX
    HLT
end start
.end
;18BCE0154
```

OUTPUT:

```
C:\MASM>debug q3.exe
-t
AX=FFFF BX=0000 CX=3000 DX=0000 SP=0064 BP=0000 SI=0000 DI=0000
DS=0734 ES=0734 SS=0746 CS=0744 IP=0008 NU UP EI PL ZR NA PE NC
0744:0008 8ED9          MOV     DS,CX
-t
AX=FFFF BX=0000 CX=3000 DX=0000 SP=0064 BP=0000 SI=0000 DI=0000
DS=3000 ES=0734 SS=0746 CS=0744 IP=000A NU UP EI PL ZR NA PE NC
0744:000A A10005       MOV     AX,[0500]          DS:0500=0000
-e DS:0500
3000:0500  00.05  00.00  00.
-t
AX=0005 BX=0000 CX=3000 DX=0000 SP=0064 BP=0000 SI=0000 DI=0000
DS=3000 ES=0734 SS=0746 CS=0744 IP=000D NU UP EI PL ZR NA PE NC
0744:000D 8B1E0205     MOV     BX,[0502]          DS:0502=0000
-e DS:0502
3000:0502  00.03  00.00  00.
-t
AX=0005 BX=0003 CX=3000 DX=0000 SP=0064 BP=0000 SI=0000 DI=0000
DS=3000 ES=0734 SS=0746 CS=0744 IP=0011 NU UP EI PL ZR NA PE NC
0744:0011 03C3          ADD     AX,BX
```

```

-t
AX=0005 BX=0000 CX=3000 DX=0000 SP=0064 BP=0000 SI=0000 DI=0000
DS=3000 ES=0734 SS=0746 CS=0744 IP=000D NV UP EI PL ZR NA PE NC
0744:000D 8B1E0205      MOV     BX,[0502]      DS:0502=0000
-e DS:0502
3000:0502 00.03 00.00 00.
-t
AX=0005 BX=0003 CX=3000 DX=0000 SP=0064 BP=0000 SI=0000 DI=0000
DS=3000 ES=0734 SS=0746 CS=0744 IP=0011 NV UP EI PL ZR NA PE NC
0744:0011 03C3      ADD     AX,BX
-t
AX=0008 BX=0003 CX=3000 DX=0000 SP=0064 BP=0000 SI=0000 DI=0000
DS=3000 ES=0734 SS=0746 CS=0744 IP=0013 NV UP EI PL NZ NA PO NC
0744:0013 A30405      MOV     [0504],AX      DS:0504=0000
-t
AX=0008 BX=0003 CX=3000 DX=0000 SP=0064 BP=0000 SI=0000 DI=0000
DS=3000 ES=0734 SS=0746 CS=0744 IP=0016 NV UP EI PL NZ NA PO NC
0744:0016 F4      HLT
-t
AX=0008 BX=0003 CX=3000 DX=0000 SP=0064 BP=0000 SI=0000 DI=0000
DS=3000 ES=0734 SS=0746 CS=0744 IP=0017 NV UP EI PL NZ NA PO NC
0744:0017 008B46FC      ADD     [BX+SI+FC46],CL      DS:FC49=00
-

```

QUES:4 Identify the instruction AAA,AAM,AAS,AAD,NEG.

1.AAA

Description

AAA converts the result of the addition of two valid unpacked BCD digits to a valid 2-digit BCD number and takes the AL register as its implicit operand.

For the previous addition to have had any meaning, each of the two operands of the addition must have had its lower 4 bits contain a number in the range from 0 to 9. The AAA instruction then adjusts AL so that it contains a correct BCD digit. If the addition produced a decimal carry (AF=1), the AH register is incremented and the carry (CF) and auxiliary carry (AF) flags are set to 1. If the addition did not produce a decimal carry, CF and AF are cleared to 0 and AH is not altered. In both cases, the high-order 4 bits of AL are cleared to 0.

Traditionally, this instruction is labeled as ASCII Adjust After Addition. And AAA will adjust the result of the addition of two ASCII characters that were in the range from 30h ("0") to 39h ("9"). This is because the lower 4 bits of those characters fall

in the range from 0 to 9. The result of the addition, however, is not an ASCII character; it is a BCD digit.

The following example shows how to add BCD numbers then adjust the result:

```
MOV     AH,0           ; Clear AH for most significant digit
MOV     AL,6           ; BCD 6 in AL
ADD     AL,5           ; Add BCD 5 to digit in AL
AAA                     ; AH=1, AL=1 representing BCD 11.
```

Algorithm

```
IF ((AL AND 0Fh)>9 OR (AF=1)) THEN
    IF (8086 OR 8088) THEN ;See note 1
        AL=AL+6
    ELSE                     ;80286 or later
        AX=AX+6
    ENDIF
    AH=AH+1
    AF=1
    CF=1
ELSE
    AF=0
    CF=0
ENDIF
AL=AL AND 0FH
```

Notes

1. The 8086 and 8088 implement AAA differently than later processors. On the 80286 and later processors, the first addition is performed on AX instead of AL, incrementing the AH register if a carry is generated out of AL. If AX contains 00FFh, executing AAA on an 8088 will leave AX=0105h. On an 80386, the same operation will leave AX=0205h. Despite the different implementation, this instruction does operate as intended for all valid operands.
2. The upper 4 bits of the AL register are always cleared to 0. This is not noted correctly in Intel's documentation for the 80386 and 80486.

2.AAM

AAM--ASCII Adjust AX After Multiply

Opcode	Instruction	Description
--------	-------------	-------------

D4 0A	AAM	ASCII adjust AX after multiply
-------	-----	--------------------------------

D4 <i>ib</i>	(No mnemonic)	Adjust AX after multiply to number base <i>imm8</i>
--------------	------------------	--

Description

Adjusts the result of the multiplication of two unpacked BCD values to create a pair of unpacked (base 10) BCD values. The AX register is the implied source and destination operand for this instruction. The AAM instruction is only useful when it follows an MUL instruction that multiplies (binary multiplication) two unpacked BCD values and stores a word result in the AX register. The AAM instruction then adjusts the contents of the AX register to contain the correct 2-digit unpacked (base 10) BCD result.

The generalized version of this instruction allows adjustment of the contents of the AX to create two unpacked digits of any number base (see the "Operation" section below). Here, the *imm8* byte is set to the selected number base (for example, 08H for octal, 0AH for decimal, or 0CH for base 12 numbers). The AAM mnemonic is interpreted by all assemblers to mean adjust to ASCII (base 10) values. To adjust to values in another number base, the instruction must be hand coded in machine code (D4 *imm8*).

Operation

tempAL \leftarrow AL;

AH \leftarrow tempAL / *imm8*; (* *imm8* is set to 0AH for the AAD mnemonic *)

AL \leftarrow tempAL MOD *imm8*;

The immediate value (*imm8*) is taken from the second byte of the instruction.

Flags Affected

The SF, ZF, and PF flags are set according to the result. The OF, AF, and CF flags are undefined.

Exceptions (All Operating Modes)

None with the default immediate value of 0AH. If, however, an immediate value of 0 is used, it will cause a #DE (divide error) exception.

3.AAS

AAS — ASCII Adjust AL After Subtraction

Opcode	Instruction	Op/En	64-bit Mode	Compat/Leg Mode	Description
3F	AAS	ZO	Invalid	Valid	ASCII adjust AL after subtraction.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	NA	NA	NA	NA

Description

Adjusts the result of the subtraction of two unpacked BCD values to create a unpacked BCD result. The AL register is the implied source and destination operand for this instruction. The AAS instruction is only useful when it follows a SUB instruction that subtracts (binary subtraction) one unpacked BCD value from another and stores a byte result in the AL register. The AAA instruction then adjusts the contents of the AL register to contain the correct 1-digit unpacked BCD result.

If the subtraction produced a decimal carry, the AH register decrements by 1, and the CF and AF flags are set. If no decimal carry occurred, the CF and AF flags are cleared, and the AH register is unchanged. In either case, the AL register is left with its top four bits set to 0.

This instruction executes as described in compatibility mode and legacy mode. It is not valid in 64-bit mode.

Operation

```
IF 64-bit mode
    THEN
        #UD;
    ELSE
        IF ((AL AND 0FH) > 9) or (AF = 1)
            THEN
                AX ← AX - 6;
                AH ← AH - 1;
                AF ← 1;
                CF ← 1;
                AL ← AL AND 0FH;
            ELSE
                CF ← 0;
```

AF ← 0;
AL ← AL AND 0FH;

FI;

FI;

Flags Affected

The AF and CF flags are set to 1 if there is a decimal borrow; otherwise, they are cleared to 0. The OF, SF, ZF, and PF flags are undefined.

Protected Mode Exceptions

#U If the LOCK prefix is
D used.

Real-Address Mode Exceptions

Same exceptions as protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as protected mode.

Compatibility Mode Exceptions

Same exceptions as protected mode.

4.AAD

AAD — ASCII Adjust AX Before Division

Opcode	Instruction	Op/En	64-bit Mode	Compat/Leg Mode	Description
D5 0A	AAD	ZO	Invalid	Valid	ASCII adjust AX before division.
D5 <i>ib</i>	AAD <i>imm8</i>	ZO	Invalid	Valid	Adjust AX before division to number base <i>imm8</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	NA	NA	NA	NA

Description

Adjusts two unpacked BCD digits (the least-significant digit in the AL register and the most-significant digit in the AH register) so that a division operation performed on the result will yield a correct unpacked BCD value. The AAD instruction is only

useful when it precedes a DIV instruction that divides (binary division) the adjusted value in the AX register by an unpacked BCD value.

The AAD instruction sets the value in the AL register to $(AL + (10 * AH))$, and then clears the AH register to 00H. The value in the AX register is then equal to the binary equivalent of the original unpacked two-digit (base 10) number in registers AH and AL.

The generalized version of this instruction allows adjustment of two unpacked digits of any number base (see the “Operation” section below), by setting the *imm8* byte to the selected number base (for example, 08H for octal, 0AH for decimal, or 0CH for base 12 numbers). The AAD mnemonic is interpreted by all assemblers to mean adjust ASCII (base 10) values. To adjust values in another number base, the instruction must be hand coded in machine code (D5 *imm8*).

This instruction executes as described in compatibility mode and legacy mode. It is not valid in 64-bit mode.

Operation

IF 64-Bit Mode

THEN

#UD;

ELSE

tempAL \leftarrow AL;

tempAH \leftarrow AH;

AL \leftarrow (tempAL + (tempAH * *imm8*)) AND FFH;

(* *imm8* is set to 0AH for the AAD mnemonic.*)

AH \leftarrow 0;

FI;

The immediate value (*imm8*) is taken from the second byte of the instruction.

Flags Affected

The SF, ZF, and PF flags are set according to the resulting binary value in the AL register; the OF, AF, and CF flags are undefined.

Protected Mode Exceptions

#U If the LOCK prefix is
D used.

Real-Address Mode Exceptions

Same exceptions as protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as protected mode.

Compatibility Mode Exceptions

Same exceptions as protected mode.

5.NEG

Purpose

Changes the arithmetic sign of the contents of a general-purpose register and places the result in another general-purpose register.

Syntax

Bits	<u>Value</u>
0 - 5	31
6 - 10	RT
11 - 15	RA
16 - 20	///
21	OE
22 - 30	104
31	Rc

Item	Description
neg	<i>RT, RA</i>
neg.	<i>RT, RA</i>
nego	<i>RT, RA</i>
nego	<i>RT, RA</i>
.	

Description

The **neg** instruction adds 1 to the one's complement of the contents of a general-purpose register (GPR) *RA* and stores the result in GPR *RT*.

If GPR *RA* contains the most negative number (that is, 0x8000 0000), the result of the instruction is the most negative number and signals the Overflow bit in the Fixed-Point Exception Register if OE is 1.

The **neg** instruction has four syntax forms. Each syntax form has a different effect on Condition Register Field 0 and the Fixed-Point Exception Register.

Item	Description			
Syntax Form	Overflow Exception (OE)	Fixed-Point Exception Register	Record Bit (Rc)	Condition Register Field 0
neg	0	None	0	None
neg.	0	None	1	LT,GT,EQ,SO
nego	1	SO,OV	0	None
nego.	1	SO,OV	1	LT,GT,EQ,SO

The four syntax forms of the **neg** instruction never affect the Carry bit (CA) in the Fixed-Point Exception Register. If the syntax form sets the Overflow Exception (OE) bit to 1, the instruction affects the Summary Overflow (SO) and Overflow (OV) bits in the Fixed-Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

Parameters

Item	Description
<i>RT</i>	Specifies target general-purpose register where result of operation is stored.
<i>RA</i>	Specifies source general-purpose register for operation.

Examples

1. The following code negates the contents of GPR 4 and stores the result in GPR 6:

```
# Assume GPR 4 contains 0x9000 3000.
neg 6,4
# GPR 6 now contains 0x6FFF D000.
```

2. The following code negates the contents of GPR 4, stores the result in GPR 6, and sets Condition Register Field 0 to reflect the result of the operation:

```
# Assume GPR 4 contains 0x789A 789B.
neg. 6,4
# GPR 6 now contains 0x8765 8765.
```

3. The following code negates the contents of GPR 4, stores the result in GPR 6, and sets the Fixed-Point Exception Register Summary Overflow and Overflow bits to reflect the result of the operation:

```
# Assume GPR 4 contains 0x9000 3000.  
nego 6,4  
# GPR 6 now contains 0x6FFF D000.
```

4. The following code negates the contents of GPR 4, stores the result in GPR 6, and sets Condition Register Field 0 and the Fixed-Point Exception Register Summary Overflow and Overflow bits to reflect the result of the operation:

```
# Assume GPR 4 contains 0x8000 0000.  
nego. 6,4  
# GPR 6 now contains 0x8000 0000.
```