APPENDIX B

x86 INSTRUCTIONS DESCRIPTION

OVERVIEW

In this appendix, we list the instructions of the 8086, give their format and expected operands, and describe the function of each instruction. Where pertinent, programming examples have been given. These instructions will operate on any x86 processor. There are additional instructions for the 80186 processor and above (80286, 80386, 80486, and Pentium); however, these instructions are not given in this list and can be found in their datasheets.

SECTION B.1: THE 8086 INSTRUCTION SET

AAA ASCII Adjust after Addition

```
Flags:      Affected: AF and CF. Unpredictable: OF, SF, ZF, PF.
Format:     AAA
```

Function: This instruction is used after an ADD instruction has added two digits in ASCII code. This makes it possible to add ASCII numbers without masking off the upper nibble "3". The result will be unpacked BCD in AL with the carry flag set if needed. This instruction adjusts only on the AL register. AH is incremented if the carry flag is set.

```
Example 1:
      MOV    AL,31H     ;AL=31 THE ASCII CODE FOR 1
      ADD    AL,37H     ;ADD 37 (ASCII FOR 7) TO AL;  AL=68H
      AAA               ;AL=08 AND CF=0
```

In the example above, ASCII 1 (31H) is added to ASCII 7 (37H). After the AAA instruction, AL will contain 8 in BCD and CF = 0. The following example shows another ASCII addition and then the adjustment:

```
Example 2:
      MOV   AL,'9'      ;AL=39 ASCII FOR 9
      ADD   AL,'5'      ;ADD 35 (ASCII FOR 5) TO AL THEN AL=6EH
      AAA               ;NOW AL=04 CF=1
      OR    AL,30H      ;converts result to ASCII
```

AAD ASCII Adjust before Division

```
Flags:      Affected: SF, ZF, PF.  Unpredictable: OF, AF, CF.
Format:     AAD
```

Function: Used before the DIV instruction to convert two unpacked BCD digits in AL and AH to binary. A better name for this would be BCD to binary conversion before division. This allows division of ASCII numbers. Before the AAD instruction is executed, the ASCII tag of 3 must be masked from the upper nibble of AH and AL.

```
Example:
      MOV   AX,3435H    ;AX=3435 THE ASCII FOR 45
      AND   AX,0F0FH    ;AX=0405H UNPACKED BCD FOR 45
      AAD               ;AX=002DH HEX FOR 45
      MOV   DL,07       ;DL=07
      DIV   DL          ;2DH DIV BY 07 GIVES AL=06,AH=03
      OR    AX,3030H    ;AL=36=QUOTIENT AND AH=33=REMAINDER
```

AAM ASCII Adjust after Multiplication

```
Flags:      Affected: AF, CF.  Unpredictable: OF, SF, ZF, PF.
Format:     AAM
```

Function: Again, a better name would have been BCD adjust after multiplication. It is used after the MUL instruction has multiplied two unpacked BCD numbers. It converts AX from binary to unpacked BCD. AAM adjusts only AL, and any digits greater than 9 are stored in AH.

```
Example:
      MOV   AL,'5'      ;AL=35
      AND   AL,0FH      ;AL=05 UNPACKED BCD FOR 5
      MOV   BL,'4'      ;BL=34
      AND   BL,0FH      ;BL=04 UNPACKED BCD FOR 4
      MUL   BL          ;AX=0014H=20 DECIMAL
      AAM               ;AX=0200
      OR    AX,3030H    ;AX=3230 ASCII FOR 20
```

AAS ASCII Adjust after Subtraction

```
Flags:      Affected: AF, CF.  Unpredictable: OF, SF, ZF, PF.
Format:     AAS
```

Function: After the subtraction of two ASCII digits, this instruction is used to convert the result in AL to packed BCD. Only AL is adjusted; the value in AH will be decremented if the carry flag is set.

```
Example:
      MOV   AL,32H      ;AL=32 ASCII FOR 2
      MOV   DH,37H      ;DH=37 ASCII FOR 7
      SUB   AL,DH       ;AL-DH=32-37=FBH WHICH IS -5 IN 2'S COMP
                        ;CF=1 INDICATING A BORROW
      AAS               ;NOW AL=05 AND CF=1
```

ADC Add with Carry

```
Flags:      Affected: OF, SF, ZF, AF, PF, CF.
Format:     ADC  dest,source        ;dest = dest + source + CF
```

Function: If CF = 1 prior to this instruction, then after execution of this instruction, source is added to destination plus 1. If CF = 0, source is added to destination plus 0. Used widely in multibyte and multiword additions.

ADD Signed or Unsigned ADD

```
Flags:      Affected: OF, SF, ZF, AF, PF, CF.
Format:     ADD  dest,source        ;dest = dest + source
```

Function: Adds source operand to destination operand and places the result in destination. Both source and destination operands must match (e.g., both byte size or word size) and only one of them can be in memory.

Unsigned addition:

In addition of unsigned numbers, the status of CF, ZF, SF, AF, and PF may change, but only CF, ZF, and AF are of any use to programmers. The most important of these flags is CF. It becomes 1 when there is carry from D7 out in 8-bit (D0–D7) operations, or a carry from D15 out in 16-bit (D0–D15) operations.

```
Example 1:
      MOV   BH,45H      ;BH=45H
      ADD   BH,4FH      ;BH=94H (45H+4FH=94H)
                        ;CF=0,ZF=0,SF=1,AF=1,and PF=0
Example 2:
      MOV   AL,FEH      ;AL=FEH
      MOV   DL,75H      ;DL=75H
      ADD   AL,DL       ;AL=FE+75=73H
                        ;CF=1,ZF=0,AF=0,SF=0,PF=0
```

```
Example 3:
      MOV   DX,126FH     ;DX=126FH
      ADD   DX,3465H     ;DX=46D4H (126F=3465=46D4H)
                         ;CF=0,ZF=0,AF=1,SF=0,PF=1
      MOV   BX,0FFFFH
      ADD   BX,1               ;BX=0000 (FFFFH+1=0000)
                         ;AND CF=1,ZF=1,AF=1,SF=0,PF=1
```

Signed addition:

In addition of signed numbers, the status of OF, ZF, and SF must be noted. Special attention should be given to the overflow flag (OF) since this indicates if there is an error in the result of the addition. There are two rules for setting OF in signed number operation. The overflow flag is set to 1:

1. If there is a carry from D6 to D7 and no carry from D7 out in an 8-bit operation or a carry from D14 to D15 and no carry from D15 out in a 16-bit operation
2. If there is a carry from D7 out and no carry from D6 to D7 in an 8-bit operation or a carry from D15 out but no carry from D14 to D15 in a 16-bit operation

Notice that if there is a carry both from D7 out and from D6 to D7, then OF = 0 in 8-bit operations. In 16-bit operations, OF = 0 if there is both a carry out from D15 and a carry from D14 to D15.

```
Example 4:
      MOV   BL,+8        ;BL=0000 1000
      MOV   DH,+4        ;DH=0000 0100
      ADD   BL,DH        ;BL=0000 1100 SF=0,ZF=0,OF=0,CF=0
```

Notice SF = D7 = 0 since the result is positive and OF = 0 since there is neither a carry from D6 to D7 nor any carry beyond D7. Since OF = 0, the result is correct [(+8) + (+4) = (+12)].

```
Example 5:
      MOV   AL,+66       ;AL=0100 0010
      MOV   CL,+69       ;CL=0100 0101
      ADD   CL,AL        ;CL=1000 0111 = -121 (INCORRECT)
                         ;CF=0,SF=1,ZF=0, AND OF=1
```

In Example 5, the correct result is +135 [(+66) + (+69) = (+135)], but the result was –121. The OF = 1 is an indication of this error. Notice that SF = D7 = 1 since the result is negative; OF = 1 since there is a carry from D6 to D7 and CF = 0.

```
Example 6:
      MOV   AL,-12       ;AL=1111 0100
      MOV   BL,+18       ;BL=0001 0010
      ADD   BL,AL        ;BL=0000 0110 (WHICH IS +6 )
                         ;SF=0,ZF=0,OF=0, AND CF=1
```

Notice above that OF = 0 since there is a carry from D6 to D7 and a carry from D7 out.

```
Example 7:
      MOV   AH,-30       ;AH=1110 0010
      MOV   DL,+14       ;DL=0000 1110
      ADD   DL,AH        ;DL=1111 0000 (WHICH IS -16 AND CORRECT)
                         ;AND SF=1,ZF=0,OF=0, AND CF=0
```

OF = 0 since there is no carry from D7 out nor any carry from D6 to D7.

```
Example 8:
      MOV   AL,-126     ;AL=1000 0010
      MOV   BH,-127     ;BH=1000 0001
      ADD   AL,BH       ;AL=0000 0011 (WHICH IS +3AND WRONG)
                        ;AND SF=0,ZF=0 AND OF=1
```

OF = 1 since there is carry from D7 out but no carry from D6 to D7.

AND Logical AND

```
Flags:      Affected: CF = 0, OF = 0, SF, ZF, PF.
            Unpredictable: AF.
Format:     AND  dest,source
```

Function: Performs logical AND on the operands, bit by bit, storing the result in the destination.

```
Example:
      MOV   BL,39H      ;BL=39
      AND   BL,09H      ;BL=09
      ;39   0011 1001
      ;09   0000 1001
      ;--   ---------
      ;09   0000 1001
```

A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1

CALL Call a Procedure

```
Flags:      Unchanged.
Format:     CALL proc   ;transfer control to procedure
```

Function: Transfers control to a procedure. RET is used to return control to the instruction after the call. There are two types of CALLs: NEAR and FAR. If the target address is within the same code segment, it is a NEAR call. If the target address is outside the current code segment, it is a FAR CALL. Each is described below.

NEAR CALL: If calling a near procedure (the procedure is in the same code segment as the CALL instruction) then the content of the IP register (which is the address of the instruction after the CALL) is pushed onto the stack and SP is decremented by 2. Then IP is loaded with the new value, which is the offset of the procedure. At the end of the procedure when the RET is executed, IP is popped off the stack, which returns control to the instruction after the CALL. There are three ways to code the address of the called NEAR procedure:

1. Direct:

```
      CALL  proc1
      ...
proc1 PROC NEAR
      ...
      RET
proc1 ENDP
```

2. Register indirect:

```
      CALL  [ SI]  ;transfer control to address in SI
```

3. Memory indirect:

```
      CALL  WORD  PTR [ DI]    ;DI points to the address that

                               ;contains IP address of proc
```

FAR CALL: When calling a far procedure (the procedure is in a different segment from the CALL instruction), the SP is decremented by 4 after CS:IP of the instruction following the CALL is pushed onto the stack. CS:IP is then loaded with the segment and offset address of the called procedure. In pushing CS:IP onto the stack, CS is pushed first and then IP. When the RETF is executed, CS and IP are restored from the stack and execution continues with the instruction following the CALL. The following addressing modes are supported:

1. Direct (but outside the present segment):

```
      CALL  proc1
      ...
proc1 PROC FAR
      ...
      RETF
proc1 ENDP
```

2. Memory indirect:

```
      CALL  DWORD PTR [ DI]     ;transfer control to CS:IP where
                                ;DI and DI+1 point to location of
                                ;CS and DI+2 and DI+4 point to
                                ;location of IP
```

CBW Convert Byte to Word

```
Flags:      Unchanged.
Format:     CBW
```

Function: Copies D7 (the sign flag) to all bits of AH. Used widely to convert a signed byte in AL to a signed word to avoid the overflow problem in signed number arithmetic.

```
Example:
      MOV   AX,0
      MOV   AL,-5       ;AL=(-5)=FB in 2's complement
                        ;AX = 0000 0000 1111 1011
      CBW               ;now AX=FFFB
                        ;AX = 1111 1111 1111 1011
```

CLC Clear Carry Flag

```
Flags:      Affected: CF.
Format:     CLC
```

Function: Resets CF to zero (CF = 0).

CLD Clear Direction Flag

```
Flags:      Affected: DF.
Format:     CLD
```

Function: Resets DF to zero (DF = 0). In string instructions if DF = 0, the pointers are incremented with each execution of the instruction. If DF = 1, the pointers are decremented. Therefore, CLD is used before string instructions to make the pointers increment.

CLI Clear Interrupt Flag

```
Flags:      Affected: IF.
Format:     CLI
```

Function: Resets IF to zero, thereby masking external interrupts received on INTR input. Interrupts received on NMI input are not blocked by this instruction.

CMC Complement Carry Flag

```
Flags:      Affected: CF.
Format:     CMC
```

Function: Changes CF from 0 to 1 or from 1 to 0.

CMP Compare Operands

```
Flags:      Affected: OF, SF, ZF, AF, PF, CF.
Format:     CMP dest,source   ;sets flags as if "SUB dest,source"
```

Function: Compares two operands of the same size. The source and destination operands are not altered. Performs comparison by subtracting the source operand from the destination and sets flags as if SUB were performed. The relevant flags are as follows:

	CF	ZF	SF	OF
dest > source	0	0	0	SF
dest = source	0	1	0	SF
dest < source	1	0	1	inverse of SF

CMPS/CMPSB/CMPSW Compare Byte or Word String

```
Flags:      Affected: OF, SF, ZF, AF, PF, CF.
Format:     CMPSx
```

Function: Compares strings a byte or word at a time. DS:SI is used to address the first operand; ES:DI is used to address the second. If DF = 0, it increments the pointers SI and DI. If DF = 1, it decrements the pointers. It can be used with prefix REPE or REPNE to compare strings of any length. The comparison is done by subtracting the source operand from the destination and sets flags as if SUB were performed.

CWD Convert Word to Doubleword

```
Flags:      Unchanged.
Format:     CWD
```

Function: Converts a signed word in AX into a signed doubleword by copying the sign bit of AX into all the bits of DX. Often used to avoid the overflow problem in signed number arithmetic.

```
Example:
      MOV   DX,0
      MOV   AX,-5       ;AX=(-5)=FFFB in 2's complement
      ;DX = 0000H
      CWD
      ;DX = FFFFH
```

DAA Decimal Adjust after Addition

```
Flags:      Affected: SF, ZF, AF, PF, CF, OF.
Format:     DAA
```

Function: This instruction is used after addition of BCD numbers to convert the result back to BCD. It adds 6 to the lower 4 bits of AL if it is greater than 9 or if AF = 1. Then it adds 6 to the upper 4 bits of AL if it is greater than 9 or if CF = 1.

```
Example 1:
      MOV   AL,47H      ;AL=0100 0111
      ADD   AL,38H      ;AL=47H+38H=7FH.    invalid BCD
      DAA               ;NOW AL=1000 0101 (85H IS VALID BCD)
```

In this example, since the lower nibble was larger than 9, DAA added 6 to AL. If the lower nibble is smaller than 9 but AF = 1, it also adds 6 to the lower nibble.

```
Example 2:
      MOV   AL,29H      ;AL=0010 1001
      ADD   AL,18H      ;AL=0100 0001 INCORRECT RESULT
      DAA               ;AL=0100 0111 A VALID BCD FOR 47H.
```

The same thing can happen for the upper nibble.

```
Example 3:
      MOV   AL,52H      ;AL=0101 0010
      ADD   AL,91H      ;AL=1110 0011 AN INVALID BCD
      DAA               ;AL=0100 0011 AND CF=1
```

Again the upper nibble can be smaller than 9 but because CF = 1, it must be corrected.

```
Example 4:
      MOV   AL,94H      ;AL=1001 0100
      ADD   AL,91H      ;AL=0010 0101 INCORRECT RESULT
      DAA               ;AL=1000 0101 A VALID BCD FOR 85 AND CF=1
```

It is entirely possible that 6 is added to both the high and low nibbles.

```
Example 5:
      MOV   AL,54H      ;AL=0101 0100
      ADD   AL,87H      ;AL=1101 1011 INVALID BCD
      DAA               ;AL=0100 0001 AND CF=1 (141 IN BCD)
```

DAS Decimal Adjust after Subtraction

```
Flags:      Affected: SF, ZF, AF, PF, CF.  Unpredictable: OF.
Format:     DAS
```

Function: This instruction is used after subtraction of BCD numbers to convert the result to BCD. If the lower 4 bits of AL represent a number greater than 9 or if AF = 1, then 6 is subtracted from the lower nibble. If the upper 4 bits of AL are now greater than 9 or if CF = 1, 6 is subtracted from the upper nibble.

```
Example:
      MOV   AL,45H      ;AL=0100 0101 BCD for 45
      SUB   AL,17H      ;AL=0010 1110 AN INVALID BCD
      DAS               ;AL=0010 1000 BCD FOR 28(45-17=28)
```

For more examples of problems associated with BCD arithmetic, see DAA.

DEC Decrement

```
Flags:      Affected: OF, SF, ZF, AF, PF.  Unchanged: CF.
Format:     DEC dest    ;dest = dest-1
```

Function: Subtracts 1 from the destination operand. Note that CF (carry/ borrow) is unchanged even if a value 0000 is decremented and becomes FFFF.

DIV Unsigned Division

```
Flags:      Unpredictable: OF, SF, ZF, AF, PF, CF.
Format:     DIV source        ;divide AX or DX:AX by source
```

Function: Divides either an unsigned word (AX) by a byte or an unsigned doubleword (DX:AX) by a word. If dividing a word by a byte, the quotient will be in AL and the remainder in AH. If dividing a doubleword by a word, the quotient will be in AX and the remainder in DX. Divide by zero causes interrupt type 0.

ESC Escape

```
Flags:      Unchanged.
Format:     ESC
```

Function: This instruction facilitates the use of math coprocessors (such as the 8087), which share data and address buses with the microprocessor. ESC is used to pass an instruction to a coprocessor and is usually treated as NOP (no operation) by the main processor.

HLT Halt

```
Flags:      Unchanged.
Format:     HLT
```

Function: Causes the microprocessor to halt execution of instructions. To get out of the halt state, activate an interrupt (NMI or INTR) or RESET.

IDIV Signed Number Division

```
Flags:      Unpredictable: OF, SF, ZF, AF, PF, CF.
Format:     IDIV  source      ;divide AX or DX:AX by source
```

Function: This division function divides either a signed word (AX) by a byte or a signed doubleword (DX:AX) by a word. If dividing a word by a byte, the signed quotient will be in AL and the signed remainder in AH. If dividing a doubleword by a word, the signed quotient will be in AX and the signed remainder in DX. Divide by zero causes interrupt type 0.

IMUL Signed Number Multiplication

```
Flags:      Affected: OF, CF.  Unpredictable: SF, ZF, AF, PF.
Format:     IMUL source   ;AX =source x AL or DX:AX =source x AX
```

Function: Multiplies a signed byte or word source operand by a signed byte or word in AL or AX with the result placed in AX or DX:AX.

IN Input Data from Port

```
Flags:      Unchanged.
Format:     IN  accumulator,port    ;input byte into AL or
                                    ;word into AX
```

Function: Transfers a byte or word to AL or AX from an input port specified by the second operand. The port address can be direct or register indirect:

1. Direct: the port address is specified directly and cannot be larger than FFH.

```
Example 1:
      IN    AL,99H      ;BRING A BYTE INTO AL FROM PORT 99H

Example 2:
      IN    AX,78H      ;BRING A WORD FROM PORT ADDRESSES 78H
                        ;AND 79H.  THE BYTE FROM PORT 78 GOES
                        ;TO AL AND BYTE FROM PORT 79H TO AH.
```

2. Register indirect: the port address is kept by the DX register. Therefore, it can be as high as FFFFH.

```
Example 3:
      MOV   DX,481H     ;DX=481H
      IN          AL,DX       ;BRING THE BYTE TO AL FROM THE
PORT
                              ;WHOSE ADDRESS IS POINTED BY DX

Example 4:
      IN    AX,DX       ;BRING A WORD FROM PORT ADDRESS POINTED
                        ;TO BY DX. THE BYTE FROM PORT AT
                        ;DX GOES TO AL AND BYTE FROM PORT AT
                        ;DX+1 TO AH.
```

INC Increment

```
Flags:      Affected: OF, SF, ZF, AF, PF.  Unchanged: CF.
Format:     INC destination   ;dest = dest + 1
```

Function: Adds 1 to the register or memory location specified by the operand. Note that CF is not affected even if a value FFFF is incremented to 0000.

INT Interrupt

```
Flags:      Affected:  IF, TF.
Format:     INT  type   ;transfer control to INT type
```

Function: Transfers execution to one of the 256 interrupts. The vector address is specified by the type number, which cannot be greater than FFH (0 to FF = 256 interrupts).

The following steps are performed for the interrupt:

1. SP is decremented by 2 and the flags are pushed onto the stack.
2. SP is decremented by 2 and CS is pushed onto the stack.
3. SP is decremented by 2 and the IP of the next instruction after the interrupt is pushed onto the stack.
4. Multiplies the type number by 4 to get the address of the vector table. Starting at this address, the first 2 bytes are the value of IP and the next 2 bytes are the value for CS of the interrupt handler (interrupt handler is also called interrupt service routine).
5. Resets IF and TF.

Interrupts are used to get the attention of the microprocessor. In the 8088/86 there are a total of 256 interrupts: INT 00, INT 01, INT 02, ... , INT FF. As mentioned above, the address that an interrupt jumps to is always four times the value of the interrupt number. For example, INT 03 will jump to memory address 0000CH (4 × 03 = 12 = 0CH). Table B-1 is a partial list of the interrupts, commonly referred to as the interrupt vector table.

Every interrupt has a program associated with it called the interrupt service routine (ISR). When an interrupt is invoked, the CS:IP address of its ISR is retrieved from the vector table (shown above). The lowest 1024 bytes (256 × 4 = 1024) of RAM are set aside for the interrupt vector table and must not be used for any other function.

Table B-1: Interrupt Vector Table

INT # (hex)	Physical Address	Logical Address
INT 00	00000	0000:0000
INT 01	00004	0000:0004
INT 02	00008	0000:0008
INT 03	0000C	0000:000C
INT 04	00010	0000:0010
INT 05	00014	0000:0014
...	...	...
INT FF	003FC	0000:03FC

Example: Find the physical and logical addresses of the vector table associated with (a) INT 14H and (b) INT 38H.

Solution:

(a) The physical address for INT 14H is 00050H–00053H (4 × 14H = 50H). That gives the logical address of 0000:0050H–0000:0053H.

(b) The physical address for INT 38H is 000E0H–000E3H, making the physical address 0000:00E0H–0000:00E3H.

The difference between INTerrupt and CALL instructions

The following are some of the differences between the INT and CALL FAR instructions:

1. While a CALL can jump to any location within the 1-megabyte address range (00000–FFFFF) of the 8088/86 CPU, "INT nn" jumps to a fixed location in the vector table as discussed earlier.
2. While the CALL is used by the programmer at a predetermined point in a program, a hardware interrupt can come in at any time.
3. A CALL cannot be masked (disabled), but "INT nn" can be masked.
4. While a "CALL FAR" automatically saves on the stack only the CS:IP of the next instruction, "INT nn" saves the FR (flag register) in addition to the CS:IP.
5. While at the end of the procedure that has been CALLed the RETF (return FAR) is used, for "INT nn" the instruction IRET (interrupt return) is used.

The 256 interrupts can be categorized into two different groups: hardware and software interrupts.

Hardware interrupts

The 8088/86 microprocessors have two pins set aside for inputting hardware interrupts. They are INTR (interrupt request) and NMI (nonmaskable interrupt). Although INTR can be ignored through the use of software masking, NMI cannot be masked using software. These interrupts are activated externally by putting 5 volts on the hardware pins of NMI or INTR. Intel has assigned INT 02 to NMI. When it is activated it will jump to memory location 00008 to get the address (CS:IP) of the interrupt service routine (ISR). Memory locations 00008, 00009, 0000A, and 0000B contain the 4-byte CS:IP. There is no specific location in the vector table assigned to INTR because INTR is used to expand the number of hardware interrupts and should be allowed to use any "INT nn" instruction that has not been assigned previously. In the IBM PC, one Intel 8259 PIC (programmable interrupt controller) chip is connected to INTR to add a total of eight hardware interrupts to the microprocessor. IBM PC AT, PS/2 80286, and x86 computers use two 8259 chips to allow up to 15 hardware interrupts.

Software interrupts

These interrupts are called software interrupts since they are invoked as a result of the execution of an instruction and no external hardware is involved. In other words, these interrupts are invoked by executing an "INT nn" instruction such as the DOS function call "INT 21H" or video interrupt "INT 10H". These interrupts can be invoked by a program at any time, the same as any other instruction. Many of the interrupts in this category are used by the DOS operating system and IBM BIOS to perform the essential tasks that every computer must provide to the system and the user. Also within this group of interrupts are predefined functions associated with some of the interrupts. They are "INT 00" (divide error), "INT 01" (single step), "INT 03" (breakpoint), and "INT 04" (signed number overflow). Each one is described below. These interrupts are shown in Table B-2. Looking at Table B-2, one can say that aside from "INT 00" to "INT 04", which have predefined functions, the rest of the interrupts, from "INT 05" to "INT FF", can be used to implement either software or hardware interrupts.

Table B-2: IBM PC Interrupt System

Interrupt	Logical Address	Physical Address	Purpose
0	00E3:3072	03EA2	Divide error
1	0600:08ED	068ED	Single step (trace command in DEBUG)
2	F000:E2C3	FE2C3	Nonmaskable interrupt
3	0600:08E6	068E6	Breakpoint
4	0700:0147	00847	Signed number arithmetic overflow
5	F000:FF54	FFF54	Print screen (BIOS)
10	F000:F065	FF065	Video I/O (BIOS)
...	...	...	...
21	relocatable	---	DOS function calls
...	...	...	...

Functions associated with "INT 00" to "INT 03"

As mentioned earlier, interrupts "INT 00" to "INT 03" have predefined functions and cannot be used in any other way. The function of each is described next.

INT 00 (divide error)

This interrupt, sometimes referred to as a conditional or exception interrupt, is invoked by the microprocessor whenever there is a condition that it cannot take care of, such as an attempt to divide a number by zero. "INT 00" is invoked by the microprocessor whenever there is an attempt to divide a number by zero. In the IBM PC and compatibles, the service subroutine for this interrupt is responsible for displaying the message "DIVIDE ERROR" on the screen if a program such as the following is executed:

```
MOV   AL,25       ; put 25 into AL
MOV   BL,00       ; put 00 into BL
DIV   BL          ; divide 25 by 00
```

This interrupt is also invoked if the quotient is too large to fit into the assigned register when executing a DIV instruction.

INT 01 (single step)

There is often a need to execute a given program one instruction at a time and then inspect the registers (possibly memory as well) to see what is happening inside the CPU. This is commonly referred to as single-stepping. IBM and Microsoft call it TRACE in the DEBUG program. To allow the implementation of single-stepping, Intel has set aside "INT 01" specifically for that purpose. For the Trace command in DEBUG after execution of each instruction, the CPU jumps automatically to physical location 00004 to fetch the 4 bytes for CS:IP of the interrupt service routine. One of the functions of this ISR is to dump the contents of the registers onto the screen.

INT 02 (nonmaskable interrupt)

This interrupt is used in the PC to indicate memory errors, among other problems.

INT 03 (breakpoint)

While in single-step mode, one can inspect the CPU and system memory after execution of each instruction. A breakpoint allows one to do the same thing, after execution of a group of instructions rather than after each instruction. Breakpoints are put in at certain points of a program to monitor the flow of the program and to inspect the results after certain instructions. The CPU executes the program to the breakpoint and stops. One can proceed from breakpoint to breakpoint until the program is complete. With the help of single-step and breakpoints, programs can be debugged and tested more easily. The Intel 8088/86 CPUs have set aside "INT 03" for the sole purpose of implementing breakpoints. When the instruction "INT 03" is placed in a program the CPU will execute the program until it encounters "INT 03", and then it stops. One interesting point about this interrupt is that it is a one-byte instruction, in contrast to all other interrupt instructions, "INT nn", which are two-byte instructions. This allows the user to insert 1 byte of code and remove it to proceed with the execution of the program. The opcode for INT 03 is "CC".

IBM PC and DOS assignment of interrupts

When the IBM PC was being developed, the designers at IBM had to coordinate the assignment of the 256 available interrupts for the 8088/86 with Microsoft, the developer of the DOS operating system, lest a conflict occur between the BIOS and DOS interrupt designations. The result of cooperation in assigning interrupts to IBM BIOS subroutines and DOS function calls is shown in Table B-2. The table gives a partial listing of interrupt numbers from 00 to FF, the logical address of the service subroutine for each interrupt, their physical addresses, and the purpose of each interrupt. It must be mentioned that depending on the computer and the DOS version, some of the logical addresses could be different from Table B-2.

How to get the vector table of any PC

One can get the vector table of any x86 IBM-compatible computer and inspect the logical address assigned to each interrupt. To do that use DEBUG's DUMP command "-D 0000:0000", as shown next.

```
A>debug
-D 0000:0000
0000:0000  E8 56 2B 02 56 07 70 00-C3 E2 00 F0 56 07 70 00  ............
0000:0010  56 07 70 00 54 FF 00 F0-47 FF 00 F0 47 FF 00 F0  ............
```

Note: The contents of the memory locations could be different, depending on the DOS version.

Example:

From the dump above, find the CS:IP of the service routine associated with INT 5.

Solution:

To get the address of "INT 5", calculate the physical address of 00014H (5 × 4 = 00014H). The contents of these locations are 00014 = 54, 00015 = FF, 00016 = 00, and 00016 = F0. This gives CS = F000 and IP = FF54.

INTO Interrupt on Overflow

```
Flags:      Affected:  IF, TF.
Format:     INTO
```

Function: Transfers execution to an interrupt handler written for overflow if OF

(overflow flag) has been set. Intel has set aside INT 4 for this purpose. Therefore, if OF = 1 when INTO is executed, the CPU jumps to memory location 00010H (4 × 4 = 16 = 10H). The contents of memory locations 10H, 11H, 12H, and 13H are used as IP and CS of the interrupt handler procedure. This instruction is widely used to detect overflow in signed number addition. In signed number operations, OF becomes 1 in two cases:

1. Whenever there is a carry from D6 to D7 in 8-bit operations and no carry from D7 out (or in 16-bit operations when there is carry from D14 to D15 and CF = 0)
2. When there is carry from D7 out and no carry from D6 to D7 (or in the case of 16-bit operations when there is a carry from D15 out and no carry from D14 to D15)

IRET Interrupt Return

```
Flags:      Affected: OF, DF, IF, TF, SF, ZF, AF, PF, CF.
Format:     IRET
```

Function: Used at the end of an interrupt service routine (interrupt handler), this instruction restores all flags, CS, and IP to the values they had before the interrupt so that execution may continue at the next instruction following the INT instruction. While the RET instruction is used at the end of the subroutine associated with the CALL instruction, IRET must be used for the subroutine associated with the "INT XX" instruction or the hardware interrupt handler.

JUMP Instructions

The following instructions are associated with jumps (both conditional and unconditional). They are categorized according to their usage rather than alphabetically.

```
J condition

Flags:      Unchanged.
Format:     Jxx  target       ;jump to target upon condition
```

Function: Used to jump to a target address if certain conditions are met. The target address cannot be more than –128 to +127 bytes away. The conditions are indicated by the flag register. The conditions that determine whether the jump takes place can be categorized into three groups:

1. flag values,
2. the comparison of unsigned numbers, and
3. the comparison of signed numbers.

Each is explained next.

1. "J condition" where the condition refers to flag values. The status of each bit of the flag register has been decided by execution of instructions prior to the jump. The following "J condition" instructions check if a certain flag bit is raised or not.

```
JC    Jump Carry              jump if CF=1
JNC   Jump No Carry           jump if CF=0
JP    Jump Parity             jump if PF=1
JNP   Jump No Parity          jump if PF=0
JZ    Jump Zero               jump if ZF=1
JNZ   Jump No Zero            jump if ZF=0
JS    Jump Sign               jump if SF=1
JNS   Jump No Sign            jump if SF=0
JO    Jump Overflow           jump if OF=1
JNO   Jump No Overflow        jump if OF=0
```

Notice that there is no "J condition" instruction for AF.

2. "J condition" where the condition refers to the comparison of unsigned numbers. After a compare (CMP dest,source) instruction is executed, CF and ZF indicate the result of the comparison, as follows:

	CF	ZF
destination > source	0	0
destination = source	0	1
destination < source	1	0

Since the operands compared are viewed as unsigned numbers, the following "J condition" instructions are used.

```
JA    Jump Above              jump if CF=0 and ZF=0
JAE   Jump Above or Equal     jump if CF=0
JB    Jump Below              jump if CF=1
JBE   Jump Below or Equal     jump if CF=1 or ZF=1
JE    Jump Equal              jump if ZF=1
JNE   Jump Not Equal          jump if ZF=0
```

3. "J condition" where the condition refers to the comparison of signed numbers. In the case of the signed number comparison, although the same instruction, "CMP destination,source", is used, the flags used to check the result are as follows:

destination > source	OF=SF or ZF=0
destination = source	ZF=1
destination < source	OF inverse of SF

Consequently, the "J condition" instructions used are different. They are as follows:

```
JG    Jump Greater            jump if ZF=0 or OF=SF
JGE   Jump Greater or Equal   jump if OF=SF
JL    Jump Less               jump if OFSF
JLE   Jump Less or Equal      jump if ZF=1 or OFSF
JE    Jump if Equal           jump if ZF = 1
```

There is one more "J condition" instruction:

```
JCXZ        ;Jump if CX is Zero.  ZF is ignored.
```

All "J condition" instructions are short jumps, meaning that the target address cannot be more than –128 bytes backward or +127 bytes forward from the IP of the instruction following the jump. What happens if a programmer needs to use a "J condition" to go to a target address beyond the –128 to +127 range? The solution is to use the "J condition" along with the unconditional JMP instruction, as shown next.

```
            ADD   BX,[ SI]
            JNC   NEXT
            JMP   TARGET1
NEXT:       ....
            ...
TARGET1:    ADD   DI,10
            ...
```

JMP Unconditional Jump

```
Flags:      Unchanged.
Format:     JMP [ directives]  target       ;jump to target address
```

Function: This instruction is used to transfer control unconditionally to a new

address. The difference between JMP and CALL is that the CALL instruction will return and continue execution with the instruction following the CALL, whereas JMP will not return. The target address could be within the current code segment, which is called a near jump, or outside the current code segment, which is called a far jump. Within each category there are many ways to code the target address, as shown next.

1. Near jump

(a) direct short jump: In this jump the target address must be within –128 to +127 bytes of the IP of the instruction after the JMP. This is a 2-byte instruction. The first byte is the opcode EBH and the second byte is the signed number displacement, which is added to the IP of the instruction following the JMP to get the target address. The directive SHORT must be coded, as shown next:

```
      JMP   SHORT OVER
      ...
OVER: ...
```

If the target address is beyond the –128 to +127 byte range and the SHORT directive is coded, the assembler gives an error.

(b) Direct jump: This is a 3-byte instruction. The first byte is the opcode E9H and the next two bytes are the signed number displacement value.The displacement is added to the IP of the instruction following the JMP to get the target address. The displacement can be in the range –32,768 to +32,767. In the absence of the SHORT directive, the assembler in its first pass always uses this kind of JMP, and then in the second pass if the target address is within the –128 and +127 byte range, it uses the NOP opcode 90H for the third byte. This is the reason to code the directive SHORT if it is known that the target address of the JMP is within the short range.

(c) Register indirect jump: In this jump the target address is in a register as shown next:

```
      JMP   DI  ;jump to the address found in DI
```

Any nonsegment register can be used for this purpose.

(d) Memory indirect jump: In this jump the target address is in a memory location whose address is pointed at by a register:

```
      JMP   WORD PTR [SI] ;jump to the address found at the address in SI
```

The directive WORD PTR must be coded to indicate this is a near jump.

2. Far jump

In a far JMP, the target address is outside the present code segment; therefore, not only the offset value but also the segment value of the target address must be given. A far jump is a 5-byte instruction: the opcode EAH and 4 bytes for the offset and segment of the target address. The following shows the two methods of coding the far jump.

(a) Direct far jump: This requires that both CS and IP be updated. One way to do that is to use the LABEL directive:

```
            JMP   TARGET2
            ...
TARGET2     LABEL FAR
ENTRY:      ...
```

This is exactly what IBM did in the BIOS of the IBM PC/XT when the computer is booted. When the power to the PC is turned on, the 8088/86 CPU begins to execute at address FFFF:0000H. IBM uses a FAR jump to make it go to location F000:E05BH, as shown next:

```
                  ;CS=FFFF and IP=0000
```

```
0000 EA5BE000F0   JMP   RESET
                  ;CS=F000
                  ORG   0E05BH
E05B  RESET       LABEL   FAR
E05B  START:
E05B              CLI
E05C              ...
```

The EXTRN and PUBLIC directives also can be used for the same purpose.

(b) Memory indirect far jump: The target address (both CS:IP) is in a memory location pointed to by the register:

```
JMP   DWORD  PTR  [BX]
```

The DWORD and PTR directives must be used to indicate that it is a far jump.

LAHF Load AH from Flags

```
Flags:      Unchanged.
Format:     LAHF
```

Function: Loads the lower 8 bits of the flag register into AH.

LDS Load Data Segment Register

```
Flags:      Unchanged.
Format:     LDS  dest,source;load dest and DS starting at source
```

Function: Loads into destination (which is a register) the contents of two memory locations indicated by source and loads DS with the contents of the two succeeding memory locations. This is useful for accessing a new data segment and its offset.

Example:

Assume the following memory locations with the contents:

```
;DS:1200=(46)
;DS:1201=(10)
;DS:1202=(38)
;DS:1203=(82)
LDS   DI,[1200]   ;now DI=1046 and DS=8238.
```

LEA Load Effective Address

```
Flags:      Unchanged.
Format:     LEA dest,source   ;dest = OFFSET source
```

Function: Loads into the destination (a 16-bit register) the effective address of a direct memory operand.

Example 1:

```
      ORG   0100H
DATA  DB    34,56,87,90,76,54,13,29
      ...
                        ;to access the sixth element:
      LEA   SI,DATA+5   ;SI=100H+5=105 THE EFFECTIVE ADDRESS
      MOV   AL,[SI]     ;GET THE SIXTH ELEMENT
```

Example 2:

```
                        ;if BX=2000H and SI=3500H
      LEA   DX,[BX][SI]+100H
                        ;DX=effective address=2000+3500+100=5600H
```

The following two instructions show two different ways to accomplish the same

thing:

```
      MOV   SI,OFFSET DATA    ;advantage: executes faster
      LEA   SI,DATA
```

LES Load Extra Segment Register

```
Flags:      Unchanged.
Format:     LES  dest,source;load dest and ES starting at source
```

Function: Loads into destination (a register) the contents of two memory locations indicated by the source and loads ES with the contents of the two succeeding memory locations. Useful for accessing a new extra segment and its offset. This instruction is similar to LDS except that the ES and its offset are being loaded.

LOCK Lock System Bus Prefix

```
Flags:      Unchanged.
Format:     LOCK        ;used as a prefix before instructions
```

Function: Used in microcomputer systems with more than one processor to prevent another processor from gaining control over the system bus during execution of an instruction.

LODS/LODSB/LODSW Load Byte or Word String

```
Flags:      Unchanged.
Format:     LODSx
```

Function: Loads AL or AX with a byte or word from the memory location pointed to by DS:SI. If DF = 0, SI will be incremented to point to the next location. If DF = 1, SI will be decremented to point to the next location. SI is incremented/decremented by 1 or 2, depending on whether it is a byte or word string.

LOOP Loop until CX = 0

```
Flags:      Unchanged.
Format:     LOOP  target;DEC CX, then jump to target if CX not 0
```

Function: Decrements CX by 1, then jumps to the offset indicated by the operand if CX is not zero; otherwise continues with the next instruction below the LOOP. This instruction is equivalent to

```
      DEC  CX
      JNZ  target
```

LOOPE/LOOPZ LOOP if Equal / Loop if Zero

```
Flags:  Unchanged.
Format: LOOPx target   ;DEC CX, jump to target if CX  0 and ZF=1
```

Function: Decrements CX by 1, then jumps to location indicated by the operand if CX is not zero and ZF is 1; otherwise continues with the next instruction after the LOOP. In other words, it gets out of the loop only when CX becomes zero or when ZF = 0.

```
Example: Assume that 200H memory locations from offset 1680H
should contain 55H.  LOOPE can be used to see if any of these
locations does not contain 55H:

      MOV   CX,200      ;SET UP THE COUNTER
      MOV   SI,1680H    ;SET UP THE POINTER
```

```
BACK: CMP   [SI],55H    ;COMPARE THE 55H WITH MEM LOCATION
                        ;POINTED AT BY SI
      INC   SI          ;INCREMENT THE POINTER
      LOOPE BACK        ;CONTINUE THE PROCESS UNTIL CX=0 OR
                        ;ZF=0.  IN OTHER WORDS EXIT IF ONE
                        ;LOCATION DOES NOT HAVE 55H
```

LOOPNE/LOOPNZ LOOP While CF Not Zero and ZF Equal Zero

```
Flags:  Unchanged.
Format: LOOPxx target   ;DEC CX, then jump if CX and ZF not zero
```

Function: Decrements CX by 1, then jumps to location indicated by the operand if CX and ZF are not zero; otherwise continues with the next instruction below the LOOP. In other words it will exit the loop if CX becomes 0 or ZF = 1.

Example:

Assume that the daily temperatures for the last 30 days have been stored starting at memory location with offset 1200H. LOOPE can be used to find the first day that had a 90-degree temperature.

```
      MOV   CX,30       ;SET UP THE COUNTER
      MOV   DI,1200H    ;SET UP THE POINTER
AGAIN:CMP   [DI],90
      INC   DI
      LOOPNE AGAIN
```

MOV Move

```
Flags:      Unchanged.
Format:     MOV  dest,source        ;copy source to dest
```

Function: Copies a word or byte from a register, memory location, or immediate number to a register or memory location. Source and destination must be of the same size and cannot both be memory locations.

MOVS/MOVSB/MOVSW Move Byte or Word String

```
Flags:      Unchanged.
Format:     MOVSx
```

Function: Moves byte or word from memory location pointed to by DS:SI to memory location pointed to by ES:DI. If DF = 0, both pointers are incremented; otherwise, they are decremented. SI and DI are incremented/decremented by 1 or 2 depending on whether it is a byte or word string. When used with the REP prefix, CX is decremented each time until CX is zero.

MUL Unsigned Multiplication

```
Flags:  Affected: OF, CF.  Unpredictable:  SF, ZF, AF, PF.
Format: MUL source      ;AX = source × AL or DX:AX = source × AX
```

Function: Multiplies an unsigned byte or word indicated by the operand by a unsigned byte or word in AL or AX with the result placed in AX or DX:AX.

NEG Negate

```
Flags:      Affected: OF, SF, ZF, AF, PF, CF.
Format:     NEG  dest   ;negates operand
```

Function: Performs 2's complement of operand. Effectively reverses the sign bit

of the operand. This instruction should only be used on signed numbers.

NOP No Operation

```
Flags:      Unchanged.
Format:     NOP
```

Function: Performs no operation. Sometimes used for timing delays to waste clock cycles. Updates IP to point to next instruction following NOP.

NOT Logical NOT

```
Flags:      Unchanged.
Format:     NOT dest      ;dest = 1's complement of dest
```

Function: Replaces the operand with its negation (the 1's complement). Each bit is inverted.

OR Logical OR

```
Flags:      Affected: CF=0, OF=0, SF, ZF, PF. Unpredictable: AF.
Format:     OR dest,source    ;dest= dest OR source
```

Function: Performs logical OR on the bits of two operands, replacing the destination operand with the result. Often used to turn a bit on.

A	B	A OR B
0	0	0
0	1	1
1	0	1
1	1	1

OUT Output Byte or Word

```
Flags:      Unchanged.
Format:     OUT  dest,acc     ;transfer acc to port dest
```

Function: Transfers a byte or word from AL or AX to an output port specified by the first operand. Port address can be direct or register indirect as shown next:

1. Direct: port address is specified directly and cannot be larger than FFH.

```
Example 1:
      OUT   68H,AL      ;SEND OUT A BYTE FROM AL TO PORT 68H

            or

      OUT   34H,AX      ;SEND OUT A WORD FROM AX TO PORT
                        ;ADDRESSES 34H AND 35H. THE BYTE
                        ;FROM AL GOES TO PORT 34H AND
                        ;THE BYTE FROM AH GOES TO PORT 35H
```

2. Register indirect: port address is kept by the DX register. Therefore, it can be as high as FFFFH.

```
Example 2:
      MOV   DX,64B1H    ;DX=64B1H
      OUT   DX,AL       ;SENT OUT THE BYTE IN AL TO THE PORT
                        ;WHOSE ADDRESS IS POINTED TO BY DX
            or

      OUT   DX,AX       ;SEND OUT A WORD FROM AX TO PORT
                        ;ADDRESS POINTED TO DX.   THE BYTE
                        ;FROM AL GOES TO PORT DX AND AND BYTE
                        ;FROM AH GOES TO PORT DX+1.
```

POP POP Word

```
Flags:      Unchanged.
Format:     POP dest    ;dest = word off top of stack
```

Function: Copies the word pointed to by the stack pointer to the register or memory location indicated by the operand and increments the SP by 2.

POPF POP Flags off Stack

```
Flags:      OF, DF, IF, TF, SF, ZF, AF, PF, CF.
Format:     POPF
```

Function: Copies bits previously pushed onto the stack with the PUSHF instruction into the flag register. The stack pointer is then incremented by 2.

PUSH PUSH Word

```
Flags:      Unchanged.
Format:     PUSH  source      ;PUSH source onto stack
```

Function: Copies the source word to the stack and decrements SP by 2.

PUSHF PUSH Flags onto stack

```
Flags:      Unchanged.
Format:     PUSHF
```

Function: Decrements SP by 2 and copies the contents of the flag register to the stack.

RCL/RCR Rotate Left through Carry and Rotate Right through Carry

```
Flags:  Affected: OF, CF.
Format: RCx dest,n;dest = dest rotate right/left n bit positions
```

Function: Rotates the bits of the operand right or left. The bits rotated out of the operand are rotated into the CF and the CF is rotated into the opposite end of the word or byte. Note: "n" must be 1 or CL.

RET Return from a Procedure

```
Flags:      Unchanged.
Format:     RET  [n]    ;return from procedure
```

Function: Used to return from a procedure previously entered by a CALL instruction. The IP is restored from the stack and the SP is incremented by 2. If the procedure was FAR, then RETF (return FAR) is used, and in addition to restoring the IP, the CS is restored from the stack and SP is again incremented by 2. The RET instruction may be followed by a number that will be added to the SP after the SP has been incremented. This is done to skip over any parameters being passed back to the calling program segment.

ROL/ROR Rotate Left and Rotate Right

```
Flags:  Affected: OF, CF.
Format: ROx dest,n      ;rotate dest right/left n bit positions
```

Function: Rotates the bits of a word or byte indicated by the second operand right or left. The bits rotated out of the word or byte are rotated back into the word or byte at the opposite end. Note: "n" must be 1 or CL.

SAHF Store AH in Flag Register

```
Flags:      Affected: SF, ZF, AF, PF, CF.
Format:     SAHF
```

Function: Copies AH to the lower 8 bits of the flag register.

SAL/SAR Shift Arithmetic Left/ Shift Arithmetic Right

```
Flags:  Affected: OF, SF, ZF, PF, CF.  Unpredictable: AF.
Format: SAx dest,n ;shift signed dest left/right n bit positions
```

Function: Shifts a word or byte left /right. SAR/ SAL arithmetic shifts are used for signed number shifting. In SAL, as the operand is shifted left bit by bit, the LSB is filled with 0s and the MSB is copied to CF. In SAR, as each bit is shifted right, the LSB is copied to CF and the empty bits filled with the sign bit (the MSB). SAL/SAR essentially multiplies/divides destination by a power of 2 for each bit shift. Note: "n" must be 1 or CL.

SBB Subtract with Borrow

```
Flags:      Affected: OF, SF, ZF, AF, PF, CF.
Format:     SBB  dest,source        ;dest = dest - CF - source
```

Function: Subtracts the source operand from the destination, replacing the destination. If CF = 1, it subtracts 1 from the result; otherwise, it executes like SUB.

SCAS/SCASB/SCASW Scan Byte or Word String

```
Flags:      Affected: OF, SF, ZF, AF, PF, CF.
Format:     SCASx
```

Function: Scans a string of data pointed by ES:DI for a value that is in AL or AX. Often used with the REPE/REPNE prefix. If DF is zero, the address is incremented; otherwise, it is decremented.

SHL/SHR Shift Left/Shift Right

```
Flags:  Affected: OF, SF, ZF, PF, CF.  Unpredictable: AF.
Format: SHx dest,n ;shift unsigned dest left/right n bit positions
```

Function: These are logical shifts used for unsigned numbers, meaning that the sign bit is treated as data. In SHR, as the operand is shifted right bit by bit and copied into CF, the empty bits are filled with 0s instead of the sign bit as is the case for SAR. In the case of SHL, as the bits are shifted left, the MSB is copied to CF and empty bits are filled with 0, which is exactly the same as SAL. In reality, SAL and SHL are two different mnemonics for the same opcode. SHL/SHR essentially multiplies/divides the destination by a power of 2 for each bit position shifted. Note: "n" must be 1 or CL.

STC Set Carry Flag

```
Flags:      Affected: CF.
Format:     STC
```

Function: Sets CF to 1.

STD Set Direction Flag

```
Flags:      Affected: DF.
Format:     STD
```

Function: Sets DF to 1. Used widely with string instructions. As explained in the string instructions, if DF = 1, the pointers are decremented.

STI Set Interrupt Flag

```
Flags:      Affected: IF.
Format:     STI
```

Function: Sets IF to 1, allowing the hardware interrupt to be recognized through the INTR pin of the CPU.

STOS/STOSB/STOSW Store Byte or Word String

```
Flags:      Unchanged.
Format:     STOSx
```

Function: Copies a byte or word from AX or AL to a location pointed to by ES:DI and updates DI to point to the next string element. The pointer DI is incremented if DF is zero; otherwise, it is decremented.

SUB Subtract

```
Flags:      Affected: OF, SF, ZF, AF, PF, CF.
Format:     SUB  dest,source        ;dest = dest - source
```

Function: Subtracts the source from the destination and puts the result in the destination. Sets the carry and zero flags according to the following:

	CF	ZF	
dest > source	0	0	the result is positive
dest = source	0	1	the result is 0
dest < source	1	0	the result is negative in 2's comp

The steps for subtraction performed by the internal hardware of the CPU are as follows:

1. Takes the 2's complement of the source
2. Adds this to the destination
3. Inverts the carry and changes the flags accordingly

The source operand remains unchanged by this instruction.

TEST Test Bits

```
Flags:      Affected: OF, SF, ZF, PF, CF.  Unpredictable: AF.
Format:     TEST  dest,source       ;performs dest AND source
```

Function: Performs a logical AND on two operands, setting flags but leaving the contents of both source and destination unchanged. While the AND instruction changes the contents of the destination and the flag bits, the TEST instruction changes only the flag bits.

Example: Assume that D0 and D1 of port 27 indicate conditions A and B, respectively, if they are high and only one of them can be high at a given time. The TEST instruction can be used as follows:

```
      IN     AL,PORT_27
      TEST AL,0000 0001B      ;CHECK THE CONDITION A
      JNZ    CASE_A           ;JUMP TO INDICATE CONDITION A
      TEST AL,0000 0010B      ;CHECK FOR CONDITION B
      JNZ    CASE_B           ;JUMP TO INDICATE CONDITION B
      ....                    ;THERE IS AN ERROR SINCE NEITHER
      ....                    ;    A NOR B HAS OCCURRED.
CASE_A:   ....
      ....
CASE_B:   ....
```

WAIT Puts Processor in WAIT State

```
Flags:      Unchanged.
Format:     WAIT
```

Function: Causes the microprocessor to enter an idle state until an external interrupt occurs. This is often done to synchronize it with another processor or with an external device.

XCHG Exchange

```
Flags:      Unchanged.
Format:     XCHG  dest,source       ;swaps dest and source
```

Function: Exchanges the contents of two registers or a register and a memory location.

XLAT Translate

```
Flags:      Unchanged.
Format:     XLAT
```

Function: Replaces contents of AL with the contents of a look-up table whose address is specified by AL. BX must be loaded with the start address of the look-up table and the element to be translated must be in AL prior to the execution of this instruction. AL is used as an offset within the conversion table. Often used to translate data from one format to another, such as ASCII to EBCDIC.

XOR Exclusive OR

```
Flags:      Affected: CF = 0, OF = 0, SF, ZF, PF.
            Unpredictable: AF.
Format:     XOR  dest,source
```

Function: Performs a logical exclusive OR on the bits of two operands and puts the result in the destination. "XOR AX,AX" can be used to clear AX.

A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0