AMBO University Woliso Campus

School of Technology & Informatics

Department Of Computer Science

Microprocessor and Assembly Language Programming

Module Code CoSc M2041

Course Code CoSc 2043

Prepared By: Mr. Abdisa Lechisa(MSc)

Ambo University Woliso Campus, Woliso, Ethiopia 2012

Table of Contents

CHAPTER 1- INTRODUCTION TO MICROPROCESSOR AND COMPUTER

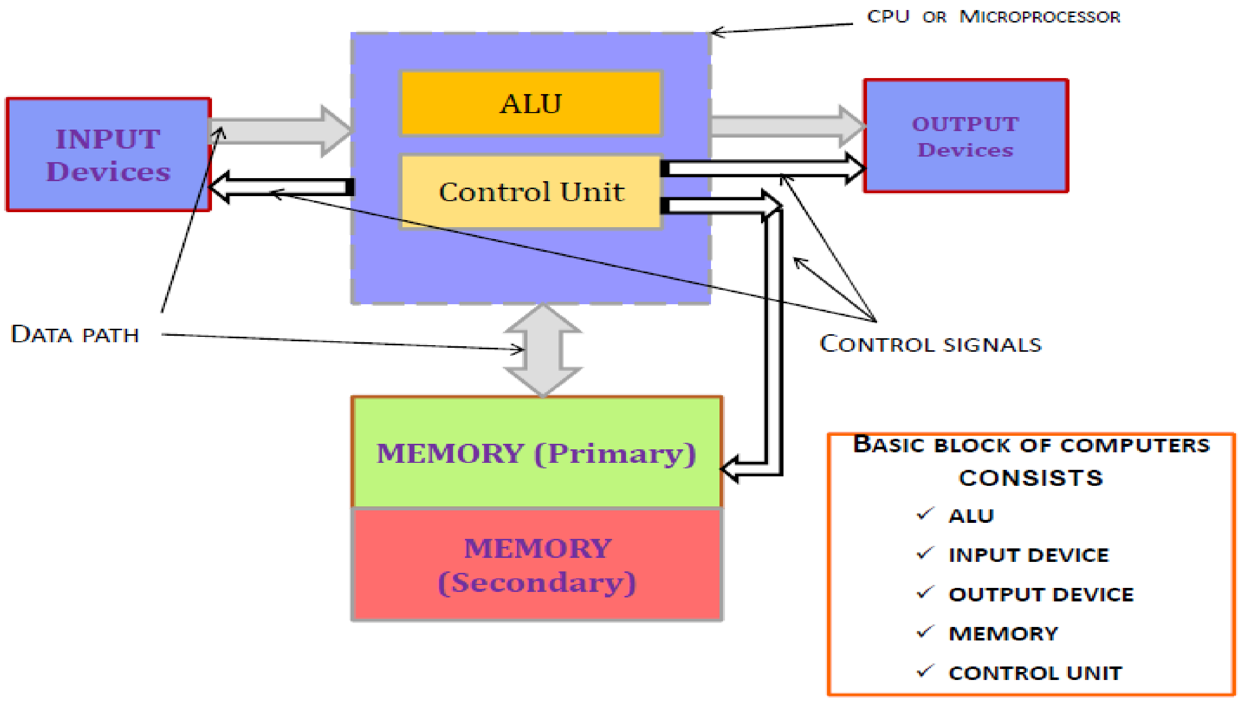
The objective of this chapter is the students will be able to:

- Understand Microprocessor and Computer
- Describe the microprocessor and its architecture
- List addressing modes Data movement instructions
- Arithmetic and Logical Instructions
- Program control instructions

A **Computer** is a programmable machine.

The principal characteristics of a computer are:

- It responds to a specific set of instructions in a well-defined manner.
- It can execute a prerecorded list of instructions (a program).
- The instructions and data are called software.
- The actual machinery wires, transistors, and circuits are called hardware.
- see computer system graphically below here.



Computer system consists of:

- CPU (central processing unit) which is the heart of the computer.
    - ✓ ALU (arithmetic-logic unit)

  - ✓ Control Logic
  - ✓ Registers, etc.…
- Memory
- Input / Output interfaces

The interconnections between these units are through 3 basic buses:

- Address Bus: - which select memory location or I/O device for CPU.
- Data Bus: - Move information b/n CPU & memory or I/O devices with size from 8 bits to 64 bits in microprocessor.
- Control Bus: - generate command signal to synchronize the CPU with I/O or memory.

Microprocessor: A microcomputer contains a CPU on a microchip (the microprocessor), a memory system (typically ROM and RAM), a bus system and I/O ports, typically housed in a motherboard. A silicon chip that contains a CPU. In the world of personal computers, the terms *microprocessor* and CPU are used interchangeably. A **microprocessor** (sometimes abbreviated **µP**) is a digital electronic component with miniaturized transistors on a single semiconductor integrated circuit (IC). One or more microprocessors typically serve as a central processing unit (CPU) in a computer system or handheld device. Microprocessors made possible the advent of the microcomputer. Microprocessors also control the logic of almost all digital devices, from clock radios to fuel-injection systems for automobiles.

Three basic characteristics differentiate microprocessors are:

- **Instruction set**: The set of instructions that the microprocessor can execute.
- **Bandwidth**: The number of bits processed in a single instruction.
- **Clock speed**: Given in megahertz (MHz), the clock speed determines how many instructions per second the processor can execute.

The evolutions of Microprocessors are: -

Processor	Date of launch	Clock speed	Data bus Width	Address Bus	Addressable Memory Size
4004	1971	740khz	4 bits	12	4 kb
8- Bit Processor					
8008	1972	800khz	8 bits	14	16 kb
8080	1974	2Mhz	8 bits	16	64 kb
8085	1976	3 MHz	8 bits	16	64 kb
16- Bit Processor					
8086	1978	5Mhz	16	20	1 M
80286	1982	16Mhz	16	24	16 M
32- Bit Processor					
80386	1985	33 MHz	32	32	4 G
80486	1989	40 MHz	32	32	4 G + 8 k cache
Pentium I	1993	100 MHz	32	32	4 G + 16 k cache
Pentium II	1997	233 MHz	32	32	4 G + 16 k cache + L2 256 cache
Pentium III	1999	1.4 Ghz	32	32	4 G + 32 k cache + L2 256 cache
Pentium IV	2000	2.66 Ghz	32 Internal 64 External	32	4 G + 32 k cache + L2 256 cache
64- Bit Processor					
Dual core	2006	2.66 GHz	64	36	64G + Independent L1 64 Kb + Common L2 256 kb Cache
Core 2 Duo	2006	3 GHz	64	36	64G + Independent L1 128 Kb + Common L2 4 Mb Cache
I7	2008	3.33 GHz	64	36	64G + Independent L1 64 Kb + Common L2 256 kb Cache + 8 Mb L3 Cache

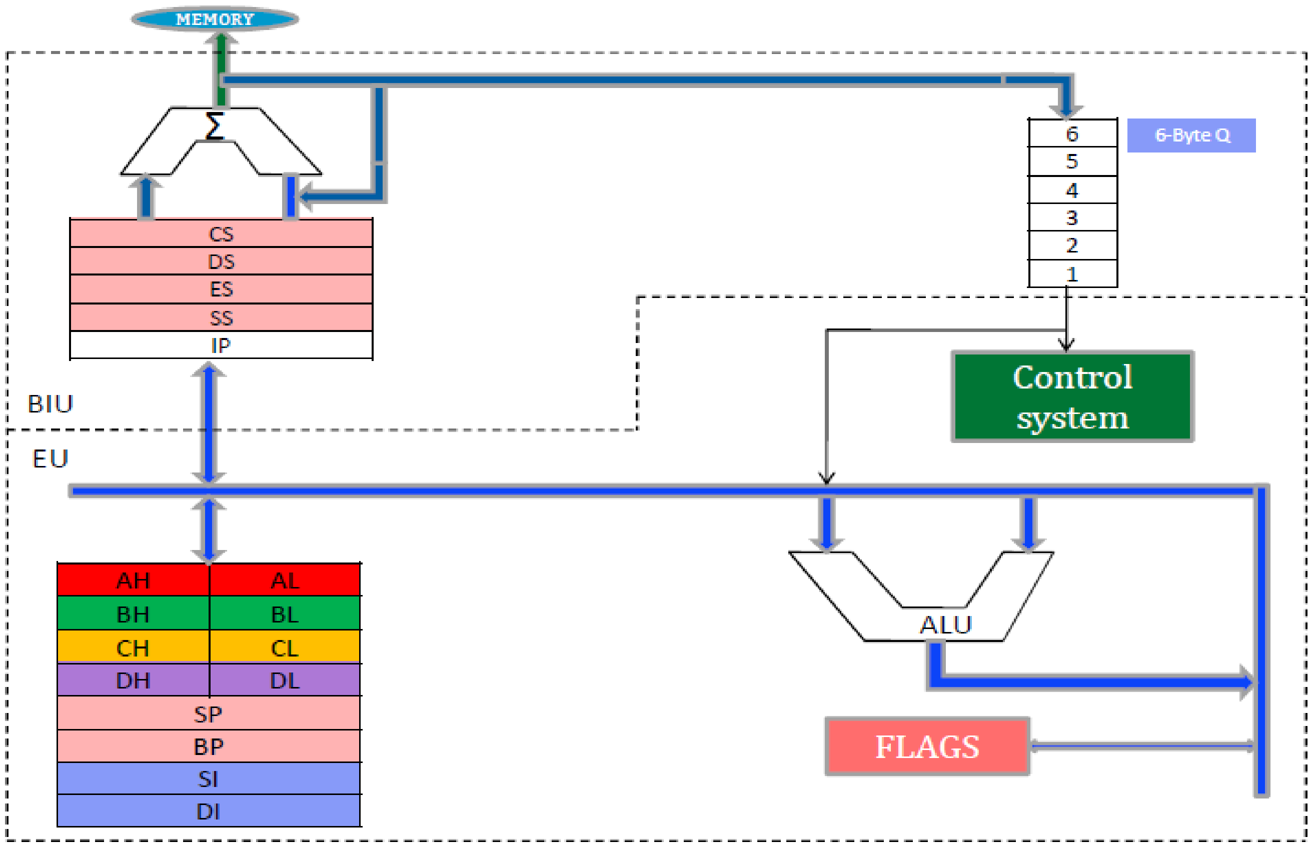
Microprocessor & Its Architecture

When we look microprocessor 8086: -

- It introduced on march 1978.
- It is implemented with 16-bit HMOS microprocessor with 29,000 transistors & operate with 5MHz clock frequency.
- Use HMOS technology & has 40 pins.
- It has 20-bit address lines hence it has 2^{20} = 1Mbytes memory locations.
- It can generate 16-bit address for I/O devices & can address 2^{16} = 64 k I/O ports.
- Can operate in 2 modes maximum or minimum mode & has 2 stage pipeline architecture.
- Has 135 number of instructions with eight 8-bit registers & eight 16-bit registers.
- Has +5v DC power supply.

Architecture Of 8086

Microprocessor architecture is different from one another. Hence, when we look intel 8086 microprocessor architecture: -



Architecture of 8086 consists of 2 units of Bus Interface Unit (BIU) & Execution Unit (EU).

Bus Interface Unit:

The BIU sends out address and fetches instructions from memory, reads data from ports and memory and return data to ports and memory. In other words, handles all transfers of data and address on the buses for the execution unit

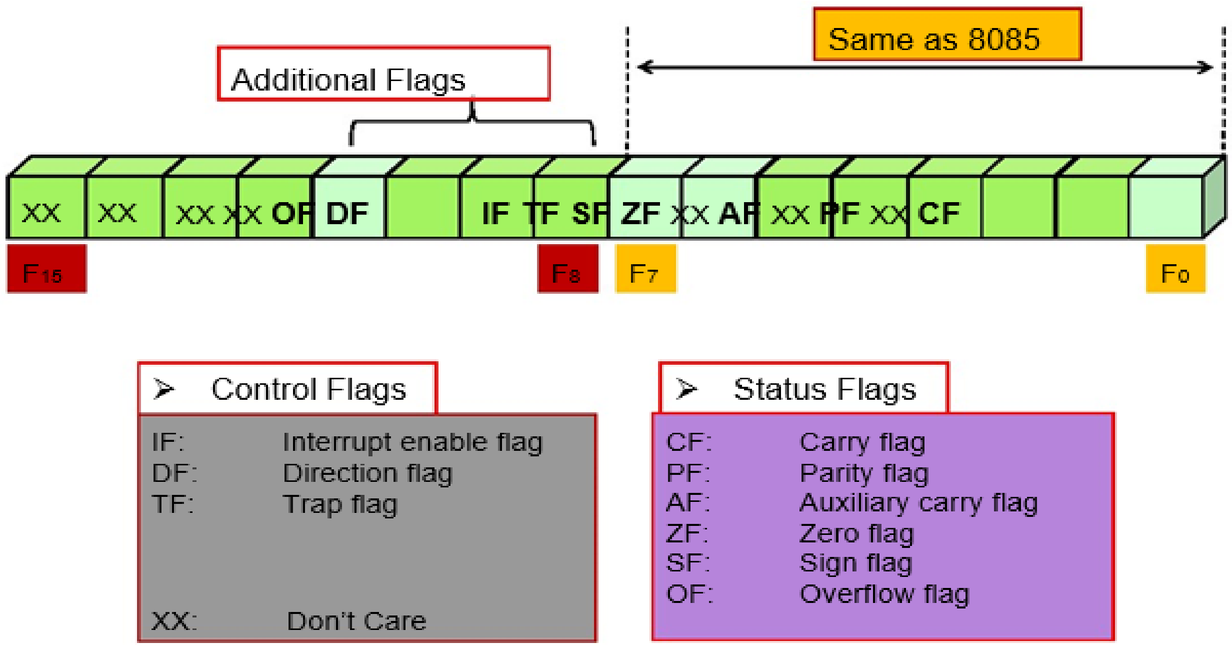
Execution Unit:

Execution of 8086 tells BIU where to fetch instructions or data from, decodes instructions and executes instructions

1. It contains control circuitry which directs internal operations
2. A decoder in the EU, translates instructions fetched from memory into a series of actions
3. It has 16-bit ALU which does all the arithmetic operations

Flag Register:

A flag is a flip-flop that indicates some status or condition produced by execution of an instruction. There are total 9 flags. Those are Status flags – 6 and Control flags- 3



Registers:

AX, BX, CX, DX are general purpose registers which are of 16-bit size which can be viewed as the combination 2 8bit size.

Register (16 bit)	8-bit	8-bit
AX	AH	AL
BX	BH	BL
CX	CH	CL
DX	DH	DL

Segment Registers: Unlike 8085, the 8086 addresses a segmented memory of 1MB, which the 8086 is able to address. The 1 MB is divided into 16 logical segments (16 X 64 KB = 1024 KB = 1 MB). Each segment thus contains 64 Kbytes of memory. There are four segment registers, viz. Code Segment Register (CS), Data Segment Register (DS), Extra Segment Register (ES) and Stack Segment Register (SS).
Code segment: It is used to store code in it.
Data segment: It is used to store data of the program.
Stack Segment: A stack is a set of memory set aside to store address and data while a sub programmed is executing. The upper 16 bits of the starting address are kept in stack segment register.
Extra segment: Strings are used in this.

Pointers and Index Registers: - The pointers contain offset within the particular segments. The pointers IP, BP and SP usually contain offsets within the code, data and stack segments respectively. The index registers are used as general-purpose registers as well as for offset storage in case of indexed, based indexed and relative based indexed addressing modes. The register SI is generally used to store the offset of source data in DMS while the register DI is used to store the offset of destination in DMS or EMS. The index registers are particularly useful for string manipulations.

Addressing Modes

The 8086 memory addressing modes provide flexible access to memory, allowing you to easily access variables, arrays, records, pointers, and other complex data types. The different ways in which the location of an operand is specified in an instruction is called as Addressing mode. These generic addressing modes are:

- Immediate mode
- Register mode

- Absolute mode
- Indirect mode
- Index mode
- Base with index
- Base with index and offset
- Relative mode
- Auto-increment mode
- Auto-decrement mode

Implementation of Variables and Constants

Variables:

The value can be changed as needed using the appropriate instructions. There are 2 accessing modes to access the variables. They are

- Register Mode and
- Absolute Mode

Register Mode:

The operand is the contents of the processor register. The name (address) of the register is given in the instruction.

Absolute Mode (Direct Mode):

The operand is in a memory location. The address of this location is given explicitly in the instruction. The various addressing modes and their assembler syntax and functions are as shown in figure below:

Name	**Assembler syntax**	**Addressing function**
Immediate	#value	Operand = value
Register	Ri	EA = Ri
Absolute(direct)	LOC	EA =LOC

Indirect	(Ri)
	(LOC)
		EA = [Ri]
		EA = [LOC]

Index	X(Ri)	EA = [Ri] + X
Base with index	(Ri, Rj)	EA = [Ri] + [Rj]
Base with index and offset	X (Ri, Rj)	EA = [Ri] + [Rj] + X
Relative	X(PC)	EA = [PC] + X

Autoincrement	(Ri)+	EA = [Ri]; Increment Ri
Autodecrement	-(Ri)	Decrement Ri EA = [Ri];

Figure of Generic addressing modes

Immediate Addressing Mode:

This addressing mode means the number that we are loading is part of the instruction itself, and is not found in the data memory.

e.g. MOV AX, 334 - will load the AX register with the number 344.

Direct Addressing Mode:

Operand resides in Memory and its address is given explicitly in the address field of an instruction.

e.g. MOV AX, my_variable - the accumulator AX is loaded with the contents of my_variable.

Register Addressing Mode:

All CPU's contain some internal registers, which act like local fast data stores. For example, an 8086 type CPU contains 4 general purpose registers called AX, BX, CX and DX. Programmers can access these registers as an integral part of their op-codes.

e.g. MOV AX, BX will move the contents of register BX into register AX.

Indirect Addressing:

This address of the variable that we wish to address is not included in the op-code at all, instead we are given the address of a different variable that contains the address the actual variable that we wish to access. At a low level this is achieved by means of either an INDEX or a POINTER register. In an 8086 one of these registers is called BP or base pointer. If we load BP with a number, then that number can then be used as the address of a variable.

```
e.g.  MOV  BP, #344;       Load the base pointer with the number 344
MOV  AX, [BP];        Load the AX register with the contents of the memory location pointed to by
                      BP.
```

Relative Addressing:

It is same as index mode. The difference is, instead of general-purpose register, here we can use program counter (PC).

Relative Mode:

The Effective Address is determined by the Index mode using the PC in place of the general-purpose register. This mode can be used to access the data operand. But it's most common use is to specify the target address in branch instruction. Eg. Branch>0 Loop. It causes the program execution to go to the branch target location. It is identified by the name loop if the branch condition is satisfied.

Additional Modes:

There are two additional modes. They are

- Auto-increment mode and
- Auto-decrement mode

Auto-increment mode:

The Effective Address of the operand is the contents of a register in the instruction. After accessing the operand, the contents of this register is automatically incremented to point to the next item in the list.

Auto-decrement mode:

The Effective Address of the operand is the contents of a register in the instruction. After accessing the operand, the contents of this register is automatically decremented to point to the next item in the list.

Data movement instructions

The **mov** instruction allows you to copy the contents of one register into another register. Each instruction can be used with different modes of addressing. Some of them are: MOV, MOVS, MOVSB, MOVSW etc.

Move Instruction

Purpose: Data transfer between memory cells, registers and the accumulator.

Syntax:

MOV Destiny, Source

Where Destiny is the place where the data will be moved and Source is the place where the data is.

The different movements of data allowed for this instruction are:

- Destiny: memory. Source: accumulator
- Destiny: accumulator. Source: memory
- Destiny: segment register. Source: memory/register
- Destiny: memory/register. Source: segment register
- Destiny: register. Source: register
- Destiny: register. Source: memory
- Destiny: memory. Source: register
- Destiny: register. Source: immediate data
- Destiny: memory. Source: immediate data

Example:

```
MOV      AX,     0006h
MOV      BX,     AX
MOV     AX,      4C00h
INT   21H
```

This small program moves the value of 0006H to the AX register, then it moves the content of AX (0006h) to the BX register, and lastly it moves the 4C00h value to the AX register to end the execution with the 4C option of the 21h interruption.

Loading instructions

They are specific register instructions. They are used to load bytes or chains of bytes onto a register.

LEA INSTRUCTION

Purpose: To load the address of the source operator

Syntax:

LEA destiny, source

The source operator must be located in memory, and its displacement is placed on the index register or specified pointer in destiny. To illustrate one of the facilities we have with this command let us write equivalence:

MOV SI, OFFSET VAR1

Is equivalent to:

LEA SI, VAR1

It is very probable that for the programmer it is much easier to create extensive programs by using this last format.

Stack instructions

These instructions allow the use of the stack to store or retrieve data. E.g. POP & PUSH.

POP INSTRUCTION

Purpose: It recovers a piece of information from the stack

Syntax:

POP destiny

This instruction transfers the last value stored on the stack to the destiny operator, it then increases by 2 the SP register. This increase is due to the fact that the stack grows from the highest memory segment address to the lowest, and the stack only works with words, 2 bytes, so then by increasing by two the SP register, in reality two are being subtracted from the real size of the stack.

PUSH INSTRUCTION

Purpose: It places a word on the stack.

Syntax:

PUSH source

The PUSH instruction decreases by two the value of SP and then transfers the content of the source operator to the new resulting address on the recently modified register.

The decrease on the address is due to the fact that when adding values to the stack, this one grows from the greater to the smaller segment address, therefore by subtracting 2 from the SP register what we do is to increase the size of the stack by two bytes, which is the only quantity of information the stack can handle on each input and output of information.

Arithmetic and Logical Instructions

Arithmetic instructions

They are used to perform arithmetic operations on the operators. Those instructions are: ADD, SUB, MUL & DIV

ADD INSTRUCTION

Purpose: Addition of the operators.

Syntax:

ADD destiny, source

It adds the two operators and stores the result on the destiny operator.

SUB INSTRUCTION

Purpose: Subtraction.

Syntax:

SUB destiny, source

It subtracts the source operator from the destiny.

MUL INSTRUCTION

Purpose: Multiplication with sign.

Syntax:

MUL source

The assembler assumes that the multiplicand will be of the same size as the multiplier, therefore it multiplies the value stored on the register given as operator by the one found to be contained in AH if the multiplier is 8 bits or by AX if the multiplier is 16 bits. When a multiplication is done with 8-bit values, the result is stored on the AX register and when the multiplication is with 16 bit values the result is stored on the even DX:AX register.

DIV INSTRUCTION

Purpose: Division without sign.

Syntax:

DIV source

The divider can be a byte or a word and it is the operator which is given the instruction.

If the divider is 8 bits, the 16 bits AX register is taken as dividend and if the divider is 16 bits the even DX:AX register will be taken as dividend, taking the DX high word and AX as the low.

If the divider was a byte then the quotient will be stored on the AL register and the residue on AH, if it was a word then the quotient is stored on AX and the residue on DX.

Logic instructions

They are used to perform logic operations on the operators. Those instructions are: AND, NEG, NOT, OR, & XOR

AND INSTRUCTION

Purpose: It performs the conjunction of the operators' bit by bit.

Syntax:

AND destiny, source

With this instruction the "y" logic operation for both operators is carried out:

```
Source Destiny | Destiny
-----------------------------
      1 1 | 1
      1 0 | 0
      0 1 | 0
      0 0 | 0
```

The result of this operation is stored on the destiny operator.

NEG INSTRUCTION

Purpose: It generates the complement to 2.

Syntax:

NEG destiny

This instruction generates the complement to 2 of the destiny operators and stores it on the same operator.

For example, if AX stores the value of 1234H, then:

NEG AX

This would leave the EDCCH value stored on the AX register.

NOT INSTRUCTION

Purpose: It carries out the negation of the destiny operator bit by bit.

Syntax:

NOT destiny

The result is stored on the same destiny operator.

OR INSTRUCTION

Purpose: Logic inclusive OR

Syntax:

OR destiny, source

The OR instruction carries out, bit by bit, the logic inclusive disjunction of the two operators:

```
Source Destiny | Destiny
----------------------------------
     1 1 | 1
     1 0 | 1
     0 1 | 1
     0 0 | 0
```

TEST INSTRUCTION

Purpose: It logically compares the operators

Syntax:

TEST destiny, source

It performs a conjunction, bit by bit, of the operators, but differing from AND, this instruction does not place the result on the destiny operator, it only has effect on the state of the flags.

XOR INSTRUCTION

Purpose: OR exclusive

Syntax:

XOR destiny, source Its function is to perform the logic exclusive disjunction of the two operators' bit by bit.

```
Source Destiny | Destiny
----------------------------------
    1   0 | 0
    0  0 | 1
    1   1 | 0
    0  1 | 1
```

Program control instructions

Jump instructions

They are used to transfer the flow of the process to the indicated operator. Some of these instructions are: JMP, JA (JNBE), JAE (JNBE), JB (JNAE), JBE (JNA), JE (JZ) and JNE (JNZ)

JMP INSTRUCTION

Purpose: Unconditional jump.

Syntax:

JMP destiny

This instruction is used to deviate the flow of a program without taking into account the actual conditions of the flags or of the data.

JA (JNBE) INSTRUCTION

Purpose: Conditional jump.

Syntax:

JA Label

After a comparison this command jumps if it is or jumps if it is not down or if not it is the equal.

This means that the jump is only done if the CF flag is deactivated or if the ZF flag is deactivated, that is that one of the two be equal to zero.

JAE (JNB) INSTRUCTION

Purpose: Conditional jump.

Syntax:

JAE label

It jumps if it is or it is the equal or if it is not down. The jump is done if CF is deactivated.

JB (JNAE) INSTRUCTION

Purpose: Conditional jump.

Syntax:

JB label

It jumps if it is down, if it is not or if it is the equal. The jump is done if CF is activated.

JBE (JNA) INSTRUCTION

Purpose: Conditional jump.

Syntax:

JBE label

It jumps if it is down, the equal, or if it is not. The jump is done if CF is activated or if ZF is activated, that any of them be equal to 1.

JE (JZ) INSTRUCTION

Purpose: Conditional jump.

Syntax:

JE label

It jumps if it is the equal or if it is zero. The jump is done if ZF is activated.

JNE (JNZ) INSTRUCTION

Purpose: Conditional jump.

Syntax:

JNE label

It jumps if it is not equal or zero. The jump will be done if ZF is deactivated.

Instructions for cycles: loop

They transfer the process flow, conditionally or unconditionally, to a destiny, repeating this action until the counter is zero. Some of these instructions are: LOOP, LOOPE & LOOPNE.

LOOP INSTRUCTION

Purpose: To generate a cycle in the program.

Syntax:

LOOP label

The loop instruction decreases CX on 1, and transfers the flow of the program to the label given as operator if CX is different than 1.

LOOPE INSTRUCTION

Purpose: To generate a cycle in the program considering the state of ZF.

Syntax:

LOOPE label

This instruction decreases CX by 1. If CX is different to zero and ZF is equal to 1, then the flow of the program is transferred to the label indicated as operator.

LOOPNE INSTRUCTION

Purpose: To generate a cycle in the program, considering the state of ZF.

Syntax:

LOOPNE label

This instruction decreases one from CX and transfers the flow of the program only if ZF is different to 0.

Counting instructions

They are used to decrease or increase the content of the counters. Those instructions are: DEC & INC.

DEC INSTRUCTION

Purpose: To decrease the operator.

Syntax:

DEC destiny

This operation subtracts 1 from the destiny operator and stores the new value in the same operator.

INC INSTRUCTION

Purpose: To increase the operator.

Syntax:

INC destiny the instruction adds 1 to the destiny operator and keeps the result in the same destiny operator.

Comparison instructions

They are used to compare operators, and they affect the content of the flags. This instruction is like CMP instruction.

CMP INSTRUCTION

Purpose: To compare the operators.

Syntax:

CMP destiny, source

This instruction subtracts the source operator from the destiny operator but without this one storing the result of the operation, and it only affects the state of the flags.

Flag instructions

They directly affect the content of the flags. Those instructions are: CLC, CLD, CLI, CMC, STC, STD & STI

CLC INSTRUCTION

Purpose: To clean the cartage flag.

Syntax:

CLC

This instruction turns off the bit corresponding to the cartage flag, or in other words it puts it on zero.

CLD INSTRUCTION

Purpose: To clean the address flag.

Syntax:

CLD

This instruction turns off the corresponding bit to the address flag.

CLI INSTRUCTION

Purpose: To clean the interruption flag.

Syntax:

CLI

This instruction turns off the interruptions flag, disabling this way those mask arable interruptions.

A mask arable interruptions is that one whose functions are deactivated when IF=0.

CMC INSTRUCTION

Purpose: To complement the cartage flag.

Syntax:

CMC

This instruction complements the state of the CF flag, if CF = 0 the instructions equals it to 1, and if the instruction is 1 it equals it to 0.

We could say that it only "inverts" the value of the flag.

STC INSTRUCTION

Purpose: To activate the cartage flag.

Syntax:

STC

This instruction puts the CF flag in 1.

STD INSTRUCTION

Purpose: To activate the address flag.

Syntax:

STD

The STD instruction puts the DF flag in 1.

STI INSTRUCTION

Purpose: To activate the interruption flag.

Syntax:

STI

The instruction activates the IF flag, and this enables the mask arable external interruptions (the ones which only function when IF = 1).

MOVS (MOVSB) (MOVSW) Instruction

Purpose: To move byte or word chains from the source, addressed by SI, to the destiny addressed by DI.

Syntax:

MOVS

This command does not need parameters since it takes as source address the content of the SI register and as destination the content of DI. The following sequence of instructions illustrates this:

```
MOV     SI,             OFFSET        VAR1
MOV      DI,             OFFSET    VAR2
MOVS
```

First we initialize the values of SI and DI with the addresses of the VAR1 and VAR2 variables respectively, then after executing MOVS the content of VAR1 is copied onto VAR2.

The MOVSB and MOVSW are used in the same way as MOVS, the first one moves one byte and the second one moves a word.

Summary

- The typical Computer system consists of: - ALU (arithmetic-logic unit), Control Logic, memory, input devices and Output devices
- The 8086 logically divided into: BIU & EU
- The flags of 8086 can be divided into two types: Conditional Flags and Control Flags
- Addressing modes is the way in which data is addressed in the operand part of the instruction.
- 8086 has 8 Addressing modes: - Immediate addressing, Register addressing, Direct addressing, Register Indirect addressing, Relative Based, Relative Indexed addressing, based indexed addressing & Relative Based indexed with displacement addressing
- 8086 Instructions cab be grouped as Data transfer instructions, Arithmetic instructions, String instructions, Loop and jump instructions etc.….

CHAPTER 2: PROGRAMMING THE MICROPROCESSOR

The objective of this chapter is the students will be able to:

- Programming the micro-processor
- Mention 8086/8088 Hardware Specification
- Know the arithmetic co-processor
- List Memory Interface ,Basic I/O Interface & Bus interface

Programming the micro-processor

Many programs are too large to be developed by one person. This means that programs are routinely developed by teams of programmers. The linker program is provided with Visual Studio so that programming modules can be linked together into a complete program. Linking is also available from the command prompt provided by Windows. This section describes the linker, the linking task, library files, EXTRN, and PUBLIC as they apply to program modules and modular programming.

The Assembler and Linker

The **assembler program** converts a symbolic **source module** (file) into a hexadecimal **object file**. The assembler dialog that appears as a source module named NEW.ASM is assembled. If a 16-bit assembler and linker are needed, they can be obtained in the Windows Driver Development Kit (DDK). Whenever you create a source file, it should have the extension of ASM, but, that is not always possible. Source files are created by using Notepad or almost any other word processor or editor capable of generating an ASCII file. The assembler program (ML) requires the source file name following ML. the /Fl switch is used to create a listing file named NEW.LST.

The source listing file (.LST) contains the assembled version of the source file and its hexadecimal machine language equivalent. The cross-reference file (.CRF), which is not generated in this example, lists all labels and pertinent information required for cross-referencing. An object file is also generated by ML as an input to the linker program. In many cases we only need to generate an object file, which is accomplished by using the /c switch.

The **linker program**, which executes as the second part of ML, reads the object files that are created by the assembler program and links them together into a single execution file. An **execution file** is created with the file name extension EXE. Execution files are selected by typing the file name at the DOS prompt (C:\). An example execution file is FROG.EXE, which is executed by typing FROG at the command prompt. If a file is short enough (less than 64K bytes long), it can be converted from an execution file to a **command file** (.COM).

The command file is slightly different from an execution file in that the program must be originated at location 0100H before it can execute. This means that the program must be no larger than 64K–100H in length. The ML program generates a command file if the tiny model is used with a starting address of 100H. Command files are only used with DOS or if a true binary version (for a EPROM/FLASH burner) is needed. The main advantage of a command file is that it loads off the disk into the computer much more quickly than an execution file. It also requires less disk storage space than the equivalent execution file.

PUBLIC and EXTRN

The PUBLIC and EXTRN directives are very important to modular programming because they allow communications between modules. We use PUBLIC to declare that labels of code, data, or entire segments are available to other program modules. EXTRN (external) declares that labels are external to a module. Without these statements, modules could not be linked together to create a program by using modular programming techniques. They might link, but one module would not be able to communicate to another. The PUBLIC directive is placed in the op-code field of an assembly language statement to define a label as public, so that the label can be used (seen by) by other modules. The label declared as public can be a jump address, a data address, or an entire segment.

The PUBLIC statement used to define some labels and make them public to other modules in a program fragment. When segments are made public, they are combined with other public segments that contain data with the same segment name.

The EXTRN statement appears in both data and code segments to define labels as external to the segment. If data are defined as external, their sizes must be defined as BYTE, WORD, or DWORD. If a jump or call address is external, it must be defined as NEAR or FAR.

Libraries

Library files are collections of procedures that are used by many different programs. These procedures are assembled and compiled into a library file by the LIB program that accompanies the MASM assembler program. Libraries allow common procedures to be collected into one place so they can be used by many different applications. The library file (FILENAME.LIB) is invoked when a program is linked with the linker program.

Macros

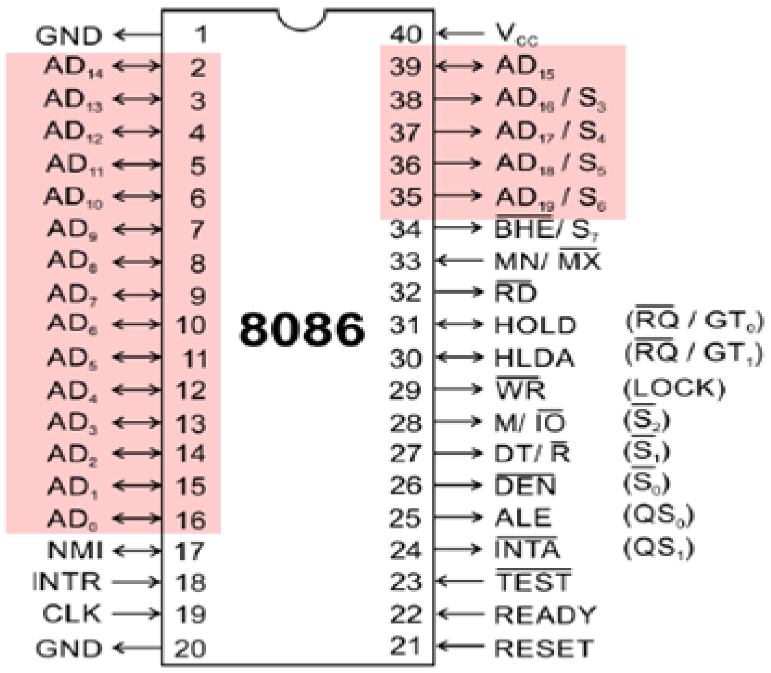
A **macro** is a group of instructions that perform one task, just as a procedure performs one task. The difference is that a procedure is accessed via a CALL instruction, whereas a macro, and all the instructions defined in the macro, is inserted in the program at the point of usage. Creating a macro is very similar to creating a new opcode, which is actually a sequence of instructions, in this case, that can be used in the program. You type the name of the macro and any parameters associated with it, and the assembler then inserts them into the program. Macro sequences execute faster than procedures because there is no CALL or RET instruction to execute. The instructions of the macro are placed in your program by the assembler at the point where they are invoked. Be aware that macros will not function using the inline assembler; they only function in external assembly language modules. The MACRO and ENDM directives delineate a macro sequence. The first statement of a macro is the MACRO instruction, which contains the name of the macro and any parameters associated with it. An example is MOVE MACRO A,B, which defines the macro name as MOVE. This new pseudo opcode uses two parameters: A and B. The last statement of a macro is the ENDM instruction, which is placed on a line by itself. Never place a label in front of the ENDM statement. If a label appears before ENDM, the macro will not assemble.

8086/8088 Hardware Specification

- 8086/8088 µ-p is packaged in a 40-pin Dual in-line Packages (DIP) and it requires a +5.0 V power Intel's supply.
- The 8086/8088 pin architectures use the combined address and data bus format commonly referred as a time multiplexed address and data bus.
- One advantage behind the multiplexed address/data bus is the maximum utilisation of processor pins (since the same pins carry address/data) and it facilitates the use of 40 pin standard DIP package.
- The bus can be de-multiplexed using a few latches and transceivers, whenever required.
- 8086 has a 20-bit address bus and can access up to 1 MB (2^{20}) memory locations.
- 8086 has a 16bit data bus, so it can read or write data to a memory/port either 16bits or 8 bit at a time.
- It provides 16 -bit registers.
- It can pre-fetch up to 6 instruction bytes from memory and queues them in order to speed up instruction execution.

- 8088 has a 20-bit address bus and can access up to 1 MB (2^{20}) memory locations.
- 8088 has a 8-bit external data bus, so it can read or write 8 bit data to a memory/port at a time.
- It provides 16 -bit registers.
- It can pre-fetch up to 4 instruction bytes from memory and queues them in order to speed up instruction execution.
- The 8086/8088 operates in single processor or multiprocessor configuration to achieve high performance.
- 8086/8088 is designed to operate in two modes, **Minimum mode** and **Maximum mode**.
- The pins serve a particular function in minimum mode (single processor mode) and other function in maximum mode configuration (multiprocessor mode).

Pin Configuration of 8086 Microprocessor



AD0-AD15 (Bidirectional)

Address/Data bus

Low order address bus; these are multiplexed with data. When AD lines are used to transmit memory address the symbol A is used instead of AD, for example A0-A15. When data are transmitted over AD lines the symbol D is used in place of AD, for example **D0-D7, D8-D15 or D0-D15.**

A16/S3, A17/S4, A18/S5, A19/S6

High order address bus. These are multiplexed with status signals

BHE (Active Low)/S7 (Output)

Bus High Enable/Status

It is used to enable data onto the most significant half of data bus, D8-D15. 8-bit device connected to upper half of the data bus use BHE (Active Low) signal. It is multiplexed with status signal S7.

MN/ MX
MINIMUM / MAXIMUM

This pin signal indicates what mode the processor is to operate in.

READY

This is the acknowledgement from the slow device or memory that they have completed the data transfer. The signal made available by the devices is synchronized by the 8284A clock generator to provide ready input to the 8086. The signal is active high.

RD (Read) (Active Low)

The signal is used for read operation. It is an output signal. It is active when low.

TEST

$\overline{Test}$ input is tested by the "wait" instruction. 8086 will enter wait state after execution of wait instruction and will resume execution only when the $\overline{Test}$ is made low by an active hardware.

RESET (Input)

Causes the processor to immediately terminate its present activity. The signal must be active HIGH for at least four clock cycles.

CLK

The clock input provides the basic timing for processor operation and bus control activity. It's an asymmetric square wave with 33% duty cycle.

INTR Interrupt Request

This is a triggered input. This is sampled during the last clock cycles of each instruction to determine the availability of the request. If any interrupt request is pending, the processor enters the interrupt acknowledge cycle. This signal is active high and internally synchronized.

MN/ MX (Active low)

The 8086 microprocessors can work in two modes of operations: Minimum mode and Maximum mode. In the minimum mode of operation, the microprocessors don't associate with any co-processors and cannot be used for multiprocessor systems. In the maximum mode the 8086 can work in multi-processor or co-processor configuration. Minimum or maximum mode operations are decided by the pin MN/ MX (Active low). When this pin is high 8086 operates in minimum mode otherwise it operates in Maximum mode. Pins 24-31 for minimum mode operation, the MN/MX is tied to Vcc (logic high). 8086 itself generates all the bus control signals.

DT/$\overline{R}$ => (Data Transmit/ Receive) Output signal from the processor to control the direction of data flow through the data transceivers.

$\overline{DEN}$ =>(Data Enable) Output signal from the processor used as output enable for the transceivers

M/$\overline{IO}$ =>(Add ess Latch Enable) Used to demultiplex the address and data lines using external latches

$\overline{WR}$ =>Write control signal; asserted low Whenever processor writes data to memory or I/O port

$\overline{INTA}$ =>(Interrupt Acknowledge) When the interrupt request is accepted by the processor, the output is low on this line

HOLD=>Input signal to the processor form the bus masters as a request to grant the control of the bus. Usually used by the DMA controller to get the control of the bus.

HLDA=> (Hold Acknowledge) Acknowledge signal by the processor to the bus master requesting the control of the bus through HOLD. The acknowledge is asserted high, when the processor accepts HOLD.

During maximum mode operation, the MN/MX is grounded (Logic low) Pins 24-31 are assigned.

$\overline{S0}$, $\overline{S1}$, $\overline{S2}$ => **Status signals**; used by the 8086-bus controller to generate bus timing and control signals. These are decoded as shown.

Status Signal			**Machine Cycle**
$\overline{S}_2$	$\overline{S}_1$	$\overline{S}_0$	
0	0	0	Interrupt acknowledgment
0	0	1	Read I/O port
0	1	0	Write I/O port
0	1	1	Halt
1	0	0	Code access
1	0	1	Read memory
1	1	0	Write memory
1	1	1	Passive/ inactive

$\overline{QS}_0$, $\overline{QS}_1$: The status processor provides the status of queue in these lines. The output of **QS_0 & QS_1** can be interpreted as shown in the table.

Queue Status		**Queue Operation**
QS_0	**QS_1**	
0	0	No operation
0	1	Frist byte of opcode from queue
1	0	Queue empty
1	1	Subsequent byte of opcode from queue

$\overline{RQ}/\overline{GT}_0$, $\overline{RQ}/\overline{GT}_1$ (bus request/bus grant) these requests are used by other local bus masters to force the processor to release the local bus at the end of the processor's current bus cycle. These pins are bidirectional. The request on $\overline{GT}0$ will have higher priority than$\overline{GT}1$.

$\overline{LOCK}$ an output signal activated by the LOCK prefix instruction. Remains active until the completion of the instruction prefixed by LOCK. The 8086 output low on the $\overline{LOCK}$ pin while executing an instruction prefixed by LOCK to prevent other bus masters from gaining control of the system bus.

The arithmetic co-processor

The Intel family of arithmetic coprocessors includes the 8087, 80287, 80387SX, 80387DX, and the 80487SX for use with the 80486SX microprocessor. The family of coprocessors, which is labeled the 80X87, is able to multiply, divide, add, subtract, find the square root, and calculate the partial tangent, partial arctangent, and logarithms.

Data types include 16-, 32-, and 64-bit signed integers; l8-digit BCD data; and 32-, 64-, and 80-bit floating-point numbers. The operations performed by the 80X87 generally execute many times faster than equivalent operations written with the most efficient programs that use the microprocessor’s normal instruction set.

Data Formats For The Arithmetic Coprocessor

These data types include signed integer, BCD, and floating-point. Each has a specific use in a system, and many systems require all three data types. Note that assembly language programming with the coprocessor is often limited to modifying the coding generated by a high level language such as C/C++.

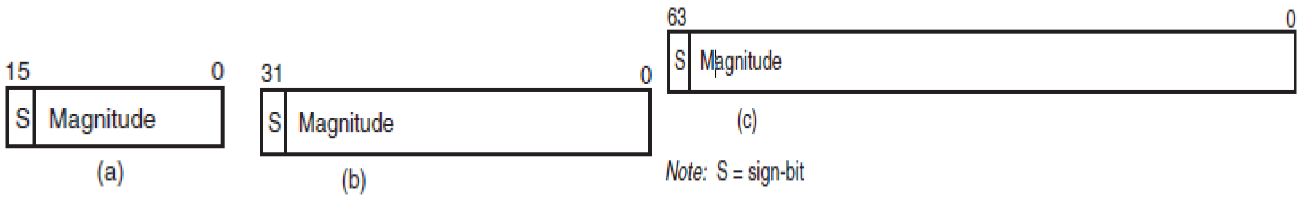
Signed Integers

Used with the arithmetic coprocessor, signed integers are 16- (word), 32- (double word integer), or 64-bits (quad word integer) wide. Positive numbers are stored in true form with a leftmost sign-bit of 0, and negative numbers are stored in two’s complement form with a leftmost sign-bit of 1. The word integers range in value from -32,768 to +32,767, the double word integer range is ±2 ×109, and the quad word integer range is ±9 × 1018. Integer data types are found in some applications that use the arithmetic coprocessor. The DW directive defines words, DD defines double word integers, and DQ defines quad word integers. Example sees how several different sizes of signed integers are defined for use by the assembler and arithmetic coprocessor.

```
0000 0002               DATA1   DW   2       ;16-bit integer
0002 FFDE               DATA2   DW   -34     ;16-bit integer
0004 000004D2           DATA3   DD   1234    ;32-bit integer
0008 FFFFFF9C           DATA4   DD   -100    ;32-bit integer
000C 0000000000005BA0   DATA5   DQ   23456   ;64-bit integer
0014 FFFFFFFFFFFFFF86   DATA6   DQ   -122    ;64-bit integer
```

Integer formats for the 80×87 family of arithmetic coprocessors:

a. Word b. Short and c. long.



Microprocessor	Coprocessor
8086/8088	8087
80186/80188	80187
80286	80287
80486SX	80487SX
80386	80387
80486DX–Core2	Built into the microprocessor

Table of showing microprocessor and coprocessor compatibility.

Binary-Coded Decimal (BCD)

The binary-coded decimal (BCD) form requires 80 bits of memory. Each number is stored as an 18-digit packed integer in nine bytes of memory as two digits per byte. The tenth byte contains only a sign-bit for the 18-digit signed BCD number.

Note that both positive and negative numbers are stored in true form and never in ten's complement form. The DT directive stores BCD data in the memory. This form is rarely used because it is unique to the Intel coprocessor.

EXAMPLE

```
0000 00000000000000000200     DATA1     DT     200       ; define 10 byte
000A 80000000000000000010     DATA2     DT     -10       ; define 10 byte
0014 00000000000000010020     DATA3     DT     10020     ; define 10 byte
```

Floating-Point

Floating-point numbers are often called *real numbers* because they hold signed integers, fractions, and mixed numbers. A floating-point number has three parts: a sign-bit, a biased exponent, and a significant. Floating-point numbers are written in scientific binary notation. The Intel family of arithmetic coprocessors supports three types of floating-point numbers: single (32 bits), double (64 bits), and temporary (80 bits).

79																		0
Sign byte	D17	D16	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0

Table of BCD data format for the 80X87 family of arithmetic coprocessors

Memory Interface

Memory: -Store Programs and Data. It includes Primary or Main Memory, processor memory and secondary memory.

Processor Memory

- Registers inside a microcomputer
- Store data and results temporarily
- No speed disparity
- Cost

Primary or Main Memory

- Storage area which can be directly accessed by microprocessor
- Store programs and data prior to execution
- Should not have speed disparity with processor => Semi-Conductor memories using CMOS technology
- Contain ROM, EPROM, Static RAM, DRAM

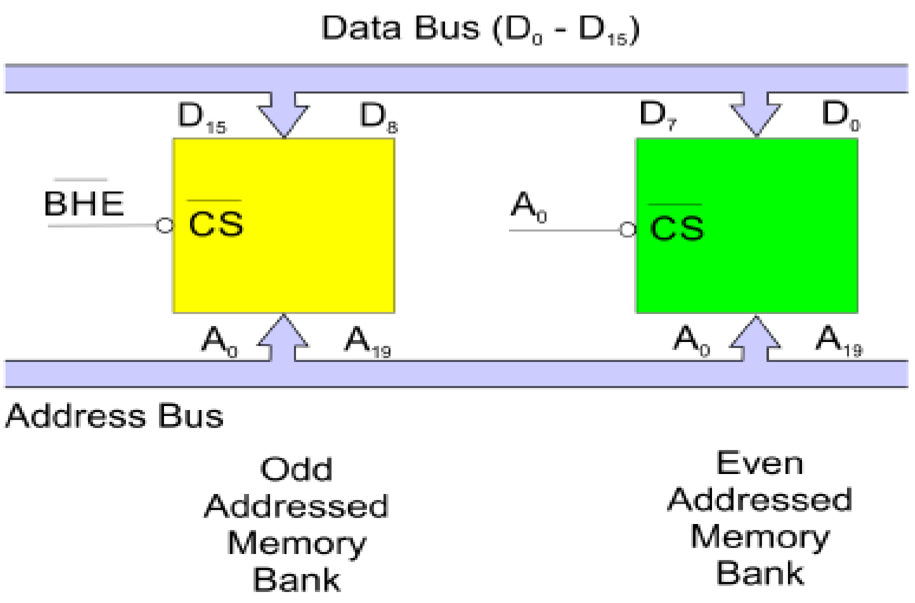
Secondary Memory

- Storage media comprising of slow devices such as magnetic tapes and disks
- Hold large data files and programs: Operating system, compilers, databases, permanent programs

Memory organization in 8086

- Memory IC's: Byte oriented
- 8086:16 bit

- Word: stored by two consecutive memory locations for LSB and MSB
- Address of word: Address of LSB
- Bank 0: A0 = 0 => Even addressed memory bank
- Bank 1: BHE = 0 => odd addressed memory bank



	Operation	BHE	A_0	Data Lines used
1	Read/ write byte at an even address	1	0	$D_7 – D_0$
2	Read/ write byte at an odd address	0	1	$D_{15} – D_8$
3	Read/ write word at an even address	0	0	$D_{15} – D_0$
4	Read/ write word at an odd address	0	1	$D_{15} – D_0$ in first operation byte from odd bank is transferred
		1	0	$D_7 – D_0$ in first operation byte from odd bank is transferred

- Available memory space = EPROM + RAM
- Allot equal address space in odd and even bank for both EPROM and RAM
- Can be implemented in two IC's (one for even and other for odd) or in multiple IC's

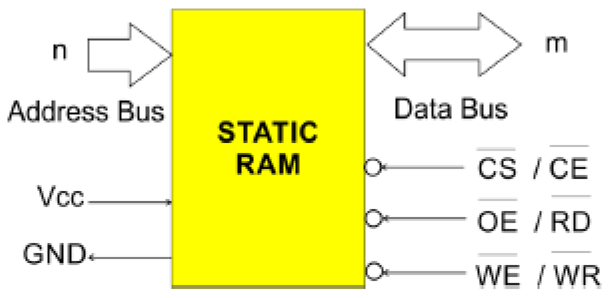
Interfacing SRAM and EPROM

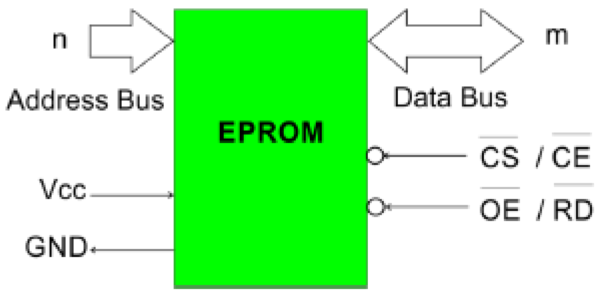
- Memory interface => Read from and write in to a set of semiconductor memory IC chip
- EPROM => Read operations
- RAM => Read and Write

In order to perform read/ write operations,

- Memory access time => read / write time of the processor
- Chip Select (CS) signal has to be generated
- Control signals for read / write operations
- Allot address for each memory location

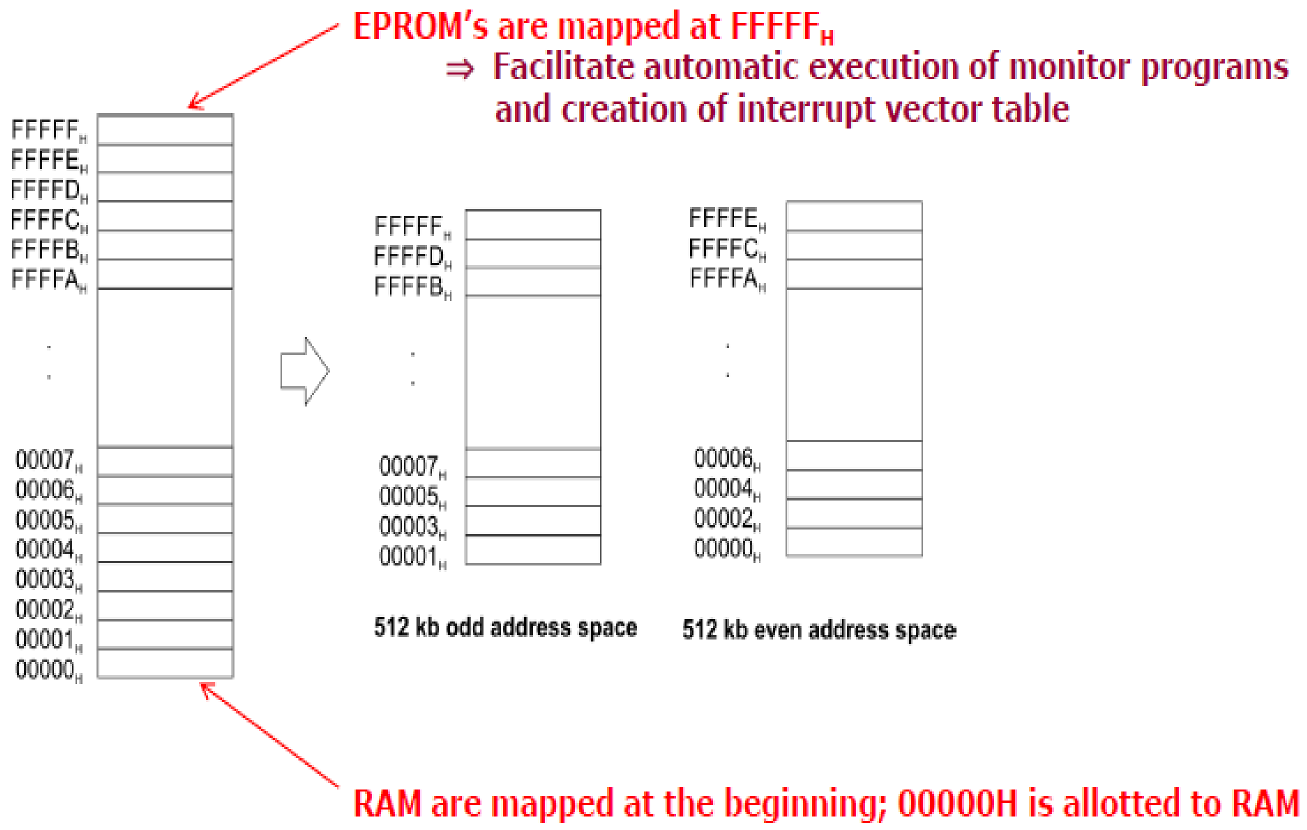
Its typical Semiconductor IC Chip for this is;


n
Address Bus
m
Data Bus
STATIC RAM
Vcc
GND
$\overline{CS}$ / $\overline{CE}$
$\overline{OE}$ / $\overline{RD}$
$\overline{WE}$ / $\overline{WR}$


n
Address Bus
m
Data Bus
EPROM
Vcc
GND
$\overline{CS}$ / $\overline{CE}$
$\overline{OE}$ / $\overline{RD}$

No of address Pin	Memory capacity			Range of address in Hexa
	In Decimal	In Kilo	In hexa	
20	2^{20} = 10,48,576	1024k = 1M	100000	00000 – FFFFF

Memory map of 8086



Monitor Programs

- Programing 8279 for keyboard scanning and display refreshing
- Programming peripheral IC's 8259, 8257, 8255, 8251, 8254 etc.
- Initialization of stack
- Display a message on display (output)
- Initializing interrupt vector table

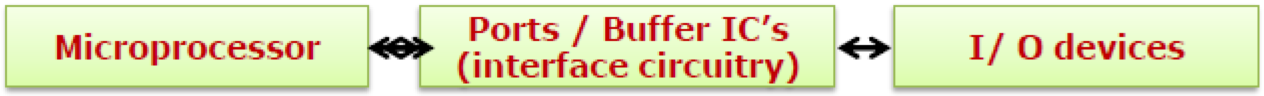
Note:

8279	Used for Programmable keyboard/ display controller
8257	Used for DMA controller
8259	Used for Programmable interrupt controller
8255	Used for Programmable peripheral interface

Basic I/O Interface

I/O devices are: -

- Used for communication between microprocessor and outside world
- Keyboards, CRT displays, Printers, Compact Discs etc.



- Used for data transfer
  - ✓ data transfer types are:
    - ♣ **Programmed I/O: -** Data transfer is accomplished through an I/O port controlled by software.
  - ❖ Programmed I/O can be memory mapped or I/O mapped.
    - ♣ **Interrupt driven I/O: -** I/O device interrupts the processor and initiate data transfer.
    - ♣ **Direct memory access: -** Data transfer is achieved by bypassing the microprocessor.

Memory Mapping	**I/O Mapping**
20-bit address are provided for I/O devices	8-bit or 16-bit address are provided for I/O devices
The I/O ports or peripherals can be treated like memory locations and so all instructions related to memory can be used for data transmission between I/O device and processor.	Only IN and OUT instructions can be used for data transfer between I/O device and processor
Data can be moved from any register to ports and vice versa	Data transfer takes place only between accumulator and ports
When memory mapping is used for I/O devices, Full memory address space cannot be used for addressing memory.	Full memory space can be used for addressing memory.
Useful only for small systems where memory requirement is low.	Suitable for system which require large memory capacity.
For accessing the memory mapped devices, the processor executes memory read or write cycle.	For accessing the I/O mapped devices, the processor executes memory read or write cycle.
M/IO is asserted high	M/IO is asserted low

Comparison of 8086 and 8088 Microprocessor

8086 Microprocessor	**8088 Microprocessor**
Similar EU and instruction set	Dissimilar BIU
16-bit data bus lines obtained by demultiplexing AD_0 - AD_{15}	8-bit data bus lines obtained by demultiplexing AD_0 - AD_7
20-bit address bus	8-bit address bus
6-bit instruction queue	4-bit instruction queue
Two banks of memory each of 512kb	Single memory bank
Clock speeds; - 5 or 8or 10 MHz	5 or 8 MHz
In MIN mode, pin 28 is assigned the signal M/IO	In MIN mode, pin 28 is assigned the signal IO/M
To access higher byte, BHE signal is used	No such signal required, since the data width is only 1-byte.

The 8087-Intel co-processor

Multiprocessor system

- A microprocessor system comprising of two or more processors.
- Distributed processing: Entire task is divided in to subtasks.

Advantages

- Better system throughput by having more than one processor
- Each processor has a local bus to access local memory or I/O devices so that a greater degree of parallel processing can be achieved.
- System structure is more flexible. One can easily add or remove modules to change the system configuration without affecting the other modules in the system.

8087 Coprocessors

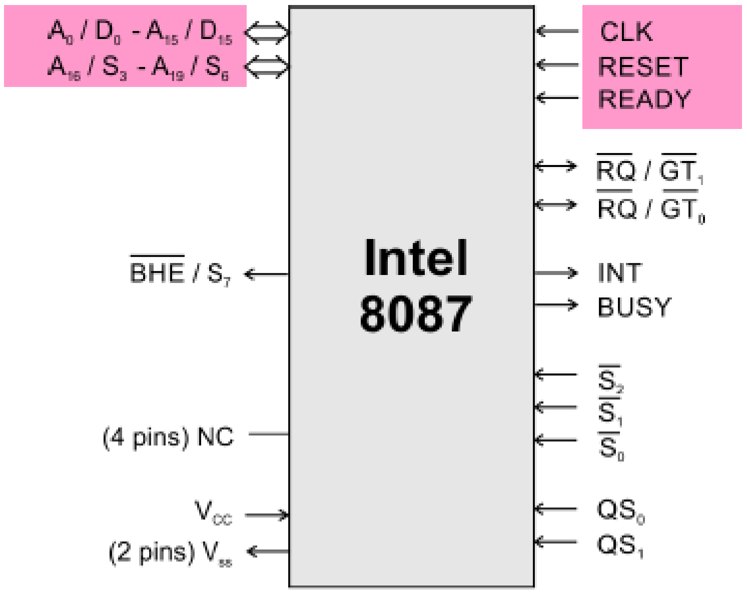
- Specially designed to take care of mathematical calculations involving integer and floating-point data
- “Math coprocessor” or “Numeric Data Processor (NDP)”

- Works in parallel with a 8086 in the maximum mode

Features

- Can operate on data of the integer, decimal and real types with lengths ranging from 2 to 10 bytes
- Instruction set involves square root, exponential, tangent etc. in addition to addition, subtraction, multiplication and division.
- High performance numeric data processor => it can multiply two 64-bit real numbers in about 27µs and calculate square root in about 36 µs
- Follows IEEE floating point standard
- It is multi bus compatible
- 16 multiplexed address / data pins and 4 multiplexed address / status pins.
- Hence it can have 16-bit external data bus and 20-bit external address bus like 8086.
- Processor clock, ready and reset signals are applied as clock, ready and reset signals for coprocessor.

  BUSY
- BUSY signal from 8087 is connected to the Test input of 8088.
- If the 8086 needs the result of some computation that the 8087 is doing before it can execute the next instruction in the program, a user can tell 8086 with WAIT instruction to keep looking at its TEST pin until it finds the pin low.
- A low on the BUSY output indicates that the 8087 has completed the computation.



RQ/GT$_1$

- The request /grant signal from the 8087 is usually connected to the request/ grant pin of the independent processor such as 8089

RQ/GT$_0$

- The request /grant signal from the 8087 is usually connected to the request/ grant (**RQ/GT$_0$ or RQ/GT$_1$**) pin of the 8086.

INT

- The interrupt pin is connected to the interrupt management logic.
- The 8087 can interrupt the 8086 through this interrupt management logic at the time error condition exists.

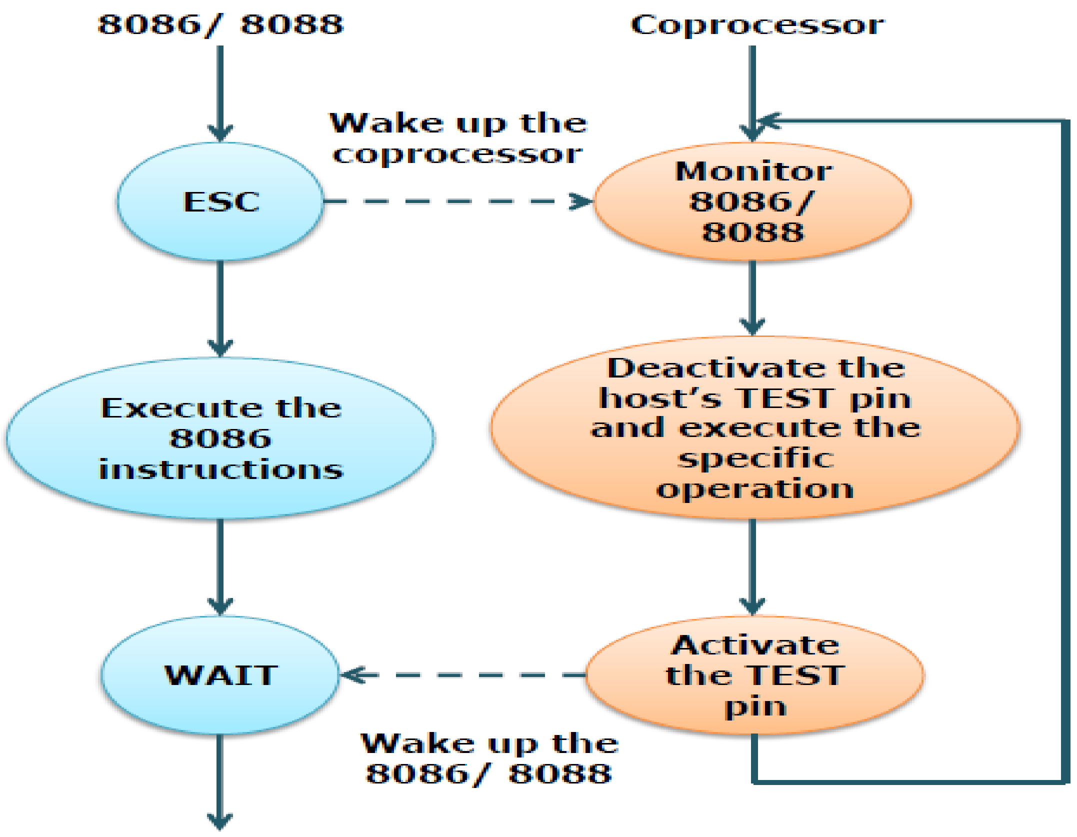
S$_0$- S$_2$

S$_2$	**S$_1$**	**S$_0$**	**Status**
1	0	0	Unused
1	0	1	Read memory
1	1	0	Write memory
1	1	1	Passive

QS_0 -QS_1

QS_0	QS_1	Status
0	0	No operation
0	1	Frist byte of opcode from queue
1	0	Queue empty
1	1	Subsequent byte of opcode from queue

Generally,



Bus Interfaces

A **bus** is a parallel data communication path over which information is transferred a byte or word at a time. The buses contain logic that the CPU controls. The items controlled are the transfer of data, instruction, and commands between the functional areas of the computer (CPU, Memory, and I/O). The name of the bus or its operation usually implies the type of signal it carries or method of operation. The direction of signal flow for the different buses. The direction may be **unidirectional** or **bidirectional** depending on the type of bus and type of computer. All computers use those three types of basic buses.

1. Data Bus (also called a memory bus)
2. Control Bus (also called timing and control bus) and
3. Address Bus

1. Data Bus

The bidirectional data bus, sometimes called the memory bus, handles the transfer of all **data** and **instructions** between functional areas of the computer. The bidirectional data bus can only transmit in one direction at a time. The data bus is used to transfer instructions from memory to the CPU for execution. It carries data (operands) to and from the CPU and memory as required by instruction translation. The data bus is also used to transfer data between memory and the I/O section during input/output operations. The size of this bus varies widely in the 8086 family.

The size of the data bus affects **the performance of the system** more than the size of any other bus. (This bus defines the size of the processor) On typical 8086systems, the data bus contains eight 16, 32 or 64 lines. Having an 8-bit data bus does not limit the processor to 8-bit data type. It is simply meaning that the processor can only access one byte of data per memory cycle. The 8-bit bus can only transmit half information per unit time (memory cycle) as the 16-bit bus. The processor with 32-bit bus is naturally faster than processors with a 16-bit data bus.

2. Address Bus

The address bus consists of all the signals necessary to define any of the possible memory address locations within the computer, or for modular memories any of the possible memory addresses locations within a module. An address is defined as a label, symbol, or other set of characters used to designate a location or register where information is stored. Before data or instructions can be written into or read from memory by the CPU or I/O sections, an address must be transmitted to memory over the address bus.

The system designer assigns a unique memory address to each memory elements and I/O device. When software wants to access some particular memory location or I/O address, it places the corresponding address on the address bus. With single address line, a processor could create exactly 2 unique addresses: Zero and One, with **n** address lines, the processor can provide **2^n** unique addresses. The number of bits on the address bus will determine the **maximum number of addressable memory and I/O locations**. The 8086, for example, have 20 bits address busses.

3. Control Bus

The control bus is used by the CPU to direct and monitor the actions of the other functional areas of the computer. It is used to transmit a variety of individual signals (read, write, interrupt, acknowledge, and so forth) necessary to control and coordinate the operations of the computer.

The individual signals transmitted over the control bus and their functions are covered in the appropriate functional area description.

The CPU sends data to memory and receives data from memory on the data bus. "Is it sending/ receiving?" there are 2 lines on the control bus, read and write, which specify the direction of dataflow. Other signals include system clocks; interrupt lines, status lines and so on.

Summary

- Programming the microprocessor describes the linker, the linking task, library files, EXTRN, and PUBLIC as they apply to program modules and modular programming.
- The **assembler program** converts a symbolic **source module** (file) into a hexadecimal **object file**.
- The Intel family of arithmetic coprocessors includes the 8087, 80287, 80387SX, 80387DX, and the 80487SX for use with the 80486SX microprocessor.
- The family of coprocessors, which is labeled the 80X87, is able to multiply, divide, add, subtract, find the square root, and calculate the partial tangent, partial arctangent, and logarithms.
- Data types include 16-, 32-, and 64-bit signed integers; l8-digit BCD data; and 32-, 64-, and 80-bit floating-point numbers.
- 8086/8088 µ-p is packaged in a 40-pin Dual in-line Packages (DIP) and it requires a +5.0 V power Intel’s supply.
- 8086 has a 20-bit address bus, 16-bit data bus and can access up to 1 MB (2^{20}) memory locations.
- All computers use those three types of basic buses like Data Bus (also called a memory bus), Control Bus (also called timing and control bus) and Address Bus.
- Memory is used for storing Programs and Data. It includes Primary or Main Memory, processor memory and secondary memory.

CHAPTER 3: COMPUTER ORGANIZATION

The objective of this chapter is the students will be able to:

- Understand about Memory & CPU
- List and compare 80X86 family of CPUS
- Identify about Interrupts
- Describe Registers

The 80X86 Family of CPU's

The term **x86** refers to a family of instruction set architectures based on the Intel 8086 CPU. The 8086 was launched in 1978 as a fully 16-bit extension of Intel's 8-bit based 8080 microprocessor and also introduced segmentation to overcome the 16-bit addressing barrier of such designs. The term x86 derived from the fact that early successors to the 8086 also had names ending in "86". The architecture has been implemented in processors from Intel, Cyrix, AMD, VIA, and many others.

As the term became common *after* the introduction of the 80386, it usually implies binary compatibility with the 32-bit instruction set of the 80386. This may sometimes be emphasized as x86-32 to distinguish it either from the original 16-bit "x86-16" or from the 64-bit x86-64. Although most x86 processors used in *new* personal computers and servers have 64-bit capabilities, to avoid compatibility problems with older computers or systems, the term *x86-64* (or *x64*) is often used to denote 64-bit software, with the term *x86* implying only 32-bit.

Although the 8086 was primarily developed for embedded systems and small single-user computers, largely as a response to the successful 8080-compatible Zilog Z80, the x86 line soon grew in features and processing power. Today, x86 is ubiquitous in both stationary and portable personal computers and has replaced midrange computers and RISC-based processors in a majority of servers and workstations as well. A large amount of software, including operating systems (OSs) such as DOS, Windows, Linux, BSD, Solaris, and Mac OS X supports x86-based hardware.

Modern x86 is relatively uncommon in embedded systems, however, and small low power applications (using tiny batteries) as well as low-cost microprocessor markets, such as home appliances and toys, lack any significant x86 presence. Simple 8-bit and 16-bit based architectures are common here, although the x86-compatible VIA C7, VIA Nano, AMD's Geode, Athlon Neo, and Intel Atom are examples of 32- and 64-bit designs used in some *relatively* low power and low cost segments.

Chronology

The table below lists brands of common consumer targeted processors implementing the x86 instruction set, grouped by generations that highlight important points in x86 history. Note: CPU generations are not strict: each generation is roughly marked by significantly improved or commercially successful processor micro architecture designs.

Generation	**First introduced**	**Prominent consumer CPU brands**	**Linear/physical address space**	**Notable (new) features**
1	1978	Intel 8086, Intel 8088 and clones	**16-bit** / 20-bit (segmented)	first x86 microprocessors
	1982	Intel 80186, Intel 80188 and clones, NEC V20/V30		hardware for fast address calculations, fast mul/div etc.
2		Intel 80286 and clones	**16-bit** (30-bit virtual) / 24-bit (segmented)	MMU, for protected mode and a larger address space
3 (IA-32)	1985	Intel 80386 and clones, AMD Am386	32-bit(46-bit virtual) / 32-bit	32-bit instruction set, MMU with paging
4 (FPU)	1989	Intel486 and clones, AMD Am486/Am5x86		risc-like pipelining, integrated x87 FPU (80-bit), on-chip cache
4/5	1997	IDT/Centaur-C6, Cyrix III-Samuel, VIA C3-Samuel2 / VIA C3-Ezra (2001), VIA C7 (2005)		In-order, integrated FPU, some models with on-chip L2 cache, MMX, SSE
5	1993	Pentium, Pentium MMX, Cyrix 5x86, Rise mP6		superscalar, 64-bit data bus, faster FPU, MMX (2x 32-bit)

5/6	1996	AMD K5, Nx586 (1994)		μ-op translation, conditional move instructions
6	1995	Pentium Pro, Cyrix 6x86, Cyrix MII, Cyrix III-Joshua (2000)	as above / **36**-bit physical (PAE)	μ-op translation, conditional move instructions, Out-of-order, register renaming, speculative execution, PAE (Pentium Pro), in-package L2 cache (Pentium Pro)
	1997	AMD K6/-2/3, Pentium II/III		L3-cache support, 3DNow!, SSE (2x 64-bit)
	2003	Pentium M, Intel Core (2006)		optimized for low power
7	1999	Athlon, Athlon XP		superscalar FPU, wide design (up to three x86 instr./clock)
	2000 `	Pentium 4		deeply pipelined, high frequency, SSE2, hyper-threading
7/8	2000	Transmeta Crusoe, Efficeon		VLIW design with x86 emulator, on-die memory controller
	2004	Pentium 4 Prescott	**64-bit** / 40-bit physical in first AMD implementation.	very deeply pipelined, very high frequency, SSE3, 64-bit capability (integer CPU) is available only in LGA 775 sockets

	2006	Intel Core 2		64-bit (integer CPU), low power, multi-core, lower clock frequency, SSE4 (Penryn)
	2008	VIA Nano		Out-of-order, superscalar, 64-bit (integer CPU), hardware-based encryption, very low power, adaptive power management
8 (x86-64)	2003	Athlon 64, Opteron		x86-64 instruction set (CPU main integer core), on-die memory controller, hyper transport
8/9	2007	AMD Phenom	as above / 48-bit physical for AMD Phenom	monolithic quad-core, SSE4a, Hyper Transport 3 or Quick Path, native memory controller, on-die L3 cache, modular design, in-package GPU (some Core i3/i5 models)
	2008	Intel Core i3, Intel Core i5, Intel Core i7, AMD Phenom II		
		Intel Atom		In-order but highly pipelined, very-low-power, on some models: 64-bit (integer CPU), on-die GPU
	2011	AMD Bobcat, Llano		Out-of-order, 64-bit (integer CPU), on-die GPU, low power (Bobcat)

9 (GPU)	2011	Intel Sandy Bridge, AMD Bulldozer and Trinity		SSE5/AVX (4x 64-bit), highly modular design, integrated on-die GPU
	2012	Intel Larrabee		very wide vector unit, LRBni instructions (8x 64-bit)
	2013	Intel Haswell		FMA3 instructions, DDR4

The processor intel 8086 is most important processor as it introduces many new functionalities. We must compare it with two nearby processor to get the better idea.

Comparison of 8086 with 8085

8085 Microprocessor	8086 Microprocessor
It is 8-bit Microprocessor	It is 16-bit Microprocessor
It has 16-bit address line.	It has 20-bit address line.
It has 8-bit data bus.	It has 16-bit data bus.
The clock speed of 8085 microprocessor is 3 MHz	The clock speed of 8086 can vary between 5, 8 and 10 MHz for three different microprocessors.
It has 5 flags.	It has 9 flags.
It does not support pipe-lining.	It supports pipe-lining.
It operates on clock cycle with 50% duty cycle.	It operates on clock cycle with 33% duty cycle.
It does not support memory segmentation	It supports memory segmentation
The 8085 microprocessor has 6500 transistors.	The 8086 microprocessor has 29000 transistors.
The 8085 microprocessor is Accumulator based processor.	The 8086 microprocessor is general purpose register-based processor.
The 8085 has no minimum or maximum mode.	The 8086 has minimum and maximum mode.
In 8085 microprocessors only one processor is being used.	In 8086 more than one processor is being used (additional external processor can be used).

In 8085 microprocessors only 64 KB memory is used together.	In 8086 microprocessor 1 MB is used.

Comparison of 8086 with 8088

8086 Microprocessor	8088 Microprocessor
8086 has 16-bit data lines.	8088 has 8-bit data lines.
8086 is available in three clock speed 5 MHz, 8 MHz and 10 MHz	Whereas 8088 is available in two clock speed 5 MHz and 8 MHz
The memory space of 8086 is organized as two 512KB banks.	The memory space of 8088 is implemented as single 1M*8 Memory bank.
8086 has 6-byte instruction queue.	8088 has 4-byte instruction queue.
The 8086 has BHE (Bank high enable)	The 8088 has SSO status signal.
The 8086 can read or write 8-bit or 16-bit data at a time.	The 8088 can read/write 8-bit data at a time.
The I/O voltages level for 8086 is measured at 2.5 mA.	The I/O voltages level for 8088 is measured at 2 mA.
The 8086 draws maximum supply current of 360mA.	The 8086 draws maximum supply current of 340mA.

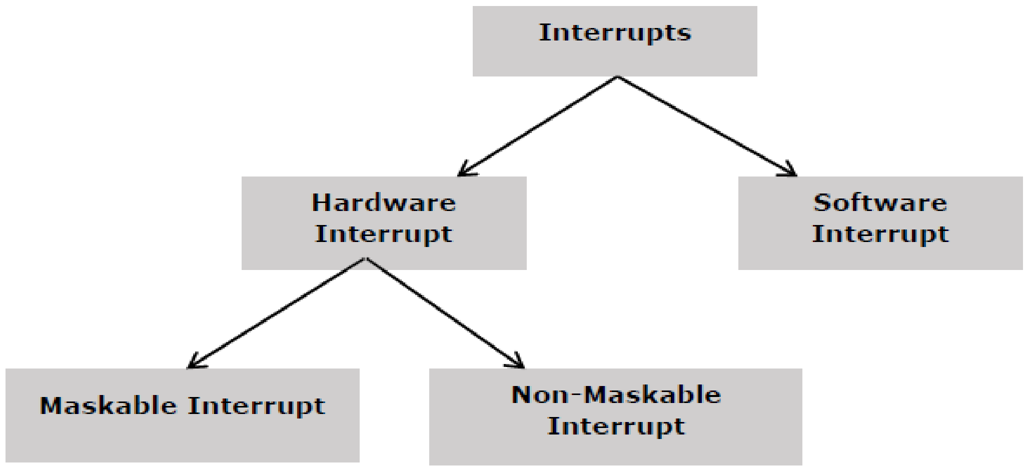
Interrupts in 8086

What is an interrupt?

Interrupt is the method of creating a temporary halt during program execution and allows peripheral devices to access the microprocessor. The microprocessor responds to that interrupt with an **ISR** (Interrupt Service Routine), which is a short program to instruct the microprocessor on how to handle the interrupt.

When the 8086 is executing a program, it can get interrupted because of one the following: -

1. Due to an interrupt getting activated. This is called as hardware interrupt.
2. Due to an exceptional happening during an instruction execution, such as division of a number by zero. This is generally termed as exception or traps
3. Due to the execution of an interrupt instruction like "INT 21H". This is called a software interrupt. The action taken by the 8086 is similar for all the three cases, except for minor differences.



Interrupt Processing in 8086

Priority Interrupt Controller

- It is used to determine priorities among devices when they request for interrupt service simultaneously.
- Priorities are determined by encoder.
- It responds to higher level input ignoring lower level input.
- The interrupting device connected at it always has highest priority.
- It includes status register, priority comparator and priority encoder.

Programmable Interrupt Controller 8259A:

- When 8259A receives interrupt signal, it sends interrupt request signal to INTR of microprocessor and INTA pulses cause the PIC to release vector information on data bus.
- It requires two internal address i.e. A=0 or A=1.
- Low order data buses D0 to D7 are connected to D0 to D7 of 8259.
- The address line A0 of microprocessor is connected to A0 of 8259 to provide internal address.
- 3 to 8 decoder generates chip select signal for 8259.
- Address lines A4, A5 and A6 are used as input to encoder.

- Control signal IO/M’ is used as logic high enables for decoder and address line A7 as logic low enable for decoder.

Interrupt Instructions of 8086

1. DI:
    - It means Disable Interrupt.
    - It is a 1 byte instruction.
    - It does not affect any flags.
    - All the interrupts except TRAP are disabled.
2. EI:
    - It means Enable Interrupt.
    - It is a 1 byte instruction.
    - It does not affect any flags.
    - All interrupts are enabled.
3. SIM:
    - It provides additional masking for RST 7.5, RST 6.5 and RST 5.5
    - Has 8 bit data format.
4. RIM:
    - The status of pending interrupts can be read from accumulator.
    - When RIM is executed, 8 bits data is loaded in accumulator.
    - Has 8 bit data format.

Hardware Interrupts in 8086

Hardware interrupt is caused by any peripheral device by sending a signal through a specified pin to the microprocessor.

The 8086 has two hardware interrupt pins, i.e. NMI and INTR. NMI is a non-maskable interrupt and INTR is a maskable interrupt having lower priority. One more interrupt pin associated is INTA called interrupt acknowledge.

1. INTR:
    - The INTR is a maskable interrupt because the microprocessor will be interrupted only if interrupts are enabled using set interrupt flag instruction.
    - It should not be enabled using clear interrupt Flag instruction.

- The INTR interrupt is activated by an I/O port.
- If the interrupt is enabled and NMI is disabled, then the microprocessor first completes the current execution and sends ‘0’ on INTA pin twice.
- The first ‘0’ means INTA informs the external device to get ready and during the second ‘0’ the microprocessor receives the 8 bit, say X, from the programmable interrupt controller. These actions are taken by the microprocessor:-
- First completes the current instruction.
- Activates INTA output and receives the interrupt type, say X.
- Flag register value; CS value of the return address and IP value of the return address are pushed on to the stack.
- IP value is loaded from the contents of word location X × 4
- CS is loaded from the contents of the next word location.
- Interrupt flag and trap flag is reset to 0
- Maskable hardware interrupt

2. NMI (non-maskable interrupt):
    - It indicates the non maskable interrupts.
    - Requires an immediate response by the MPU.
    - It is usually used for serious circumstances like power failure.
    - It is of type 2 interrupt. When this interrupt is activated, these actions take place.
    - Completes the current instruction that is in progress.
    - Pushes the Flag register values on to the stack.
    - Pushes the CS (code segment) value and IP (instruction pointer) value of the return address on to the stack.
    - IP is loaded from the contents of the word location 00008H.
    - CS is loaded from the contents of the next word location 0000AH.
    - Interrupt flag and trap flag are reset to 0.

Interrupt Vector Table of 8086

- IVT is a memory area where all the interrupts are mapped.
- It is located in memory page 00.
- It holds the vector that redirect the microprocessor to right place when interrupt arises.

- IVT is a 1024 bytes sized table consisting addresses of interrupts.
- Each address is of 4 bytes of form- offset: segment, representing address of ISR to be called when microprocessor receives interrupt.
- The interrupt number (0 to 255) is used as index into the table to get address of ISR.
- When interrupt number is passed as an argument to IVT, it points to required ISR.
- ISR executes its code and finally returns to original statement.
- The model of 4 byte entry is as below:

Software Interrupts Types of 8086

Some instructions are inserted at the desired position into the program to create interrupts. These interrupt instructions can be used to test the working of various interrupt handlers. It includes −

INT- Interrupt instruction with type number. It is 2-byte instruction. First byte provides the op-code and the second byte provides the interrupt type number. There are 256 interrupt types under this group.

- The lowest five types are dedicated to specific interrupts such as the divide by zero interrupt and the non maskable interrupt.
- The next 27 interrupt types, from 5 to 31 are reserved by Intel for use in future microprocessors.
- The upper 224 interrupt types, from 32 to 255 are available to use for hardware and software interrupts.

Its execution includes the following steps −

- Flag register value is pushed on to the stack.
- CS value of the return address and IP value of the return address are pushed on to the stack.
- IP is loaded from the contents of the word location 'type number' × 4
- CS is loaded from the contents of the next word location.
- Interrupt Flag and Trap Flag are reset to 0

The starting address for type0 interrupt is 000000H, for type1 interrupt is 00004H similarly for type2 is 00008H and ……so on. The first five pointers are dedicated interrupt pointers. i.e.

1. Type 0 (INT 00):
   - It is invoked by microprocessor whenever there is an attempt to divide a number by zero.
   - ISR displays message "Divide Error".
2. Type 1 (INT 01):
   - For single stepping, the trap flag must be 1.

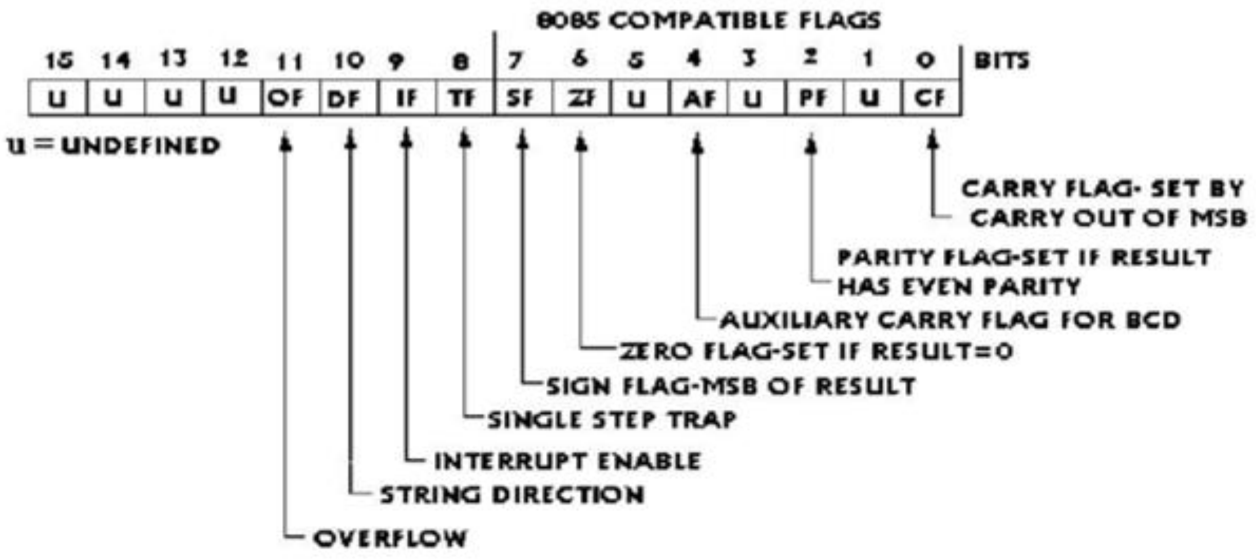
- After each instruction, 8086 jumps to 00004H to fetch 4 bytes for CS: IP of ISR.
- ISR dump registers on to the screen.

3. Type 2 (INT 02):
   - Whenever NMI pin is activated, CPU jumps to 00008H to fetch CS : IP of ISR associated with NMI.
4. Type 3 (INT 03):
   - Breakpoint is used to examine CPU and memory after execution of a group of instructions.
5. Type 4 (INT 04):
   - It is invoked when signed number overflows.
   - It interrupts on overflow.

Registers for 8086

Flag Register

- A flag is a flip-flop which indicates some condition produced by the execution of an instruction or controls operations of the EU.
- The flag register is a special register associated with the ALU.
- A 16-bit flag register in the EU contains 9 active flags. See the location of the 9 flags in the flag register.


8085 COMPATIBLE FLAGS
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 BITS
U U U U OF DF IF TF SF ZF U AF U PF U CF
u = UNDEFINED
CARRY FLAG- SET BY CARRY OUT OF MSB
PARITY FLAG-SET IF RESULT HAS EVEN PARITY
AUXILIARY CARRY FLAG FOR BCD
ZERO FLAG-SET IF RESULT=0
SIGN FLAG-MSB OF RESULT
SINGLE STEP TRAP
INTERRUPT ENABLE
STRING DIRECTION
OVERFLOW

- 6 flags are status flags: - AF, CF, OF, SF, PF and ZF.

- The remaining 3 flags are control flags: - DF, IF and TF. See the summary of these flags below.

Status Flags	Description
AF (Auxiliary flag)	Indicates if the instruction generated a carry out the 4 LSBs.
CF (carry flag)	Indicates if the instruction generated a carry out the MSB.
OF (Overflow flag)	Indicates if the instruction generated a signed result that is out of range.
SF (Signal flag)	Indicates if the instruction generated a negative result.
PF (parity flag)	Indicates if the instruction generated a result having an even number of 1s.
ZF (zero flag)	Indicates if the instruction generated a zero result
DF (direction flag)	Controls the direction of the string manipulation instructions.
IF (interrupt enable flag)	Enables or disables external interrupts.
TP (trap flag)	Puts the processor into a single step mode for program debugging.

- **AF ->**If this flag is set, there has been a carry out or borrows of the 4 least significant bits. This flag is used during decimal arithmetic instructions.
- **CF ->** If this flag is set, there has been a carry out or overflows the most significant bits. This flag is used by instructions that add and subtract multi byte numbers.
- **OF ->** If this flag is set, an arithmetic overflow has occurred; that is a significant digit has been lost because of the size of the result exceeded the capacity of its destination location.
- **SF ->** Since negative binary numbers are represented in the 80886/8088 in standard 2s complement notation. SF indicates the sign of the result (0 = positive, 1 = negative).
- **ZF** ->If this flag is set, the result of the operation is 0.
- **PF ->** If this flag is set, the result has even number of 1s. This flag can be used to check for transmission errors.
- **DF ->**setting DF causes string instructions to auto-decrement (count down); that is to process strings from the high address to the low address, or from right to left. Clearing DF causes string instruction to auto-increment (count up); that is to process strings from left to right.

- **IF ->** setting IF allows the MPU to recognize external or maskable interrupt requests. Clearing IF disables these interrupts. IF has no effect on either non maskable external or internally generated interrupts.
- **TP ->**setting TP puts the processor into single step mode for debugging. In this mode the MPU automatically generates an internal interrupt after each instruction. Allowing a program to be inspected as it executes instruction by instruction.

General Purpose Registers

EU has eight general purpose registers labeled AH, AL, BH, BL, CH, CL, DH and DL. These registers are a set of data registers which are used to hold intermediate results. The H represents the high order or most significant byte and L represents the low order or least-significant byte. Each of these registers may be used separately as 8-bit storage areas or combined to form one 16-bit(one word) storage area.

The acceptable register pairs are AH and AL, BH and BL, CH and CL & DH and DL. The AH-AL pair is referred to as the AX, the BH-BL pair is referred to as the BX, the CH-CL pair is referred to as the CX, & DH-DL pair is referred to as the DX register. The AL register is also called as the Accumulator. For 16-bit operations AX is called the accumulator. The 8086-register set is very similar to those of earlier generations 8080 and 8085 microprocessors. Many programs written for the 8080 and 8085 could easily be translated to run on the 8086.

Summary

- Simple 8-bit and 16-bit based architectures are common here, although the x86-compatible VIA C7, VIA Nano, AMD's Geode, Athlon Neo, and Intel Atom are examples of 32- and 64-bit designs used in some *relatively* low power and low cost segments.
- Interrupt can be classified as hardware interrupt and software interrupt.
- A 16-bit flag register in the EU contains 9 active flags like AF, CF, OF, SF, PF, ZF, DF, IF and TF.
- EU has eight general purpose registers labeled AH, AL, BH, BL, CH, CL, DH and DL.

CHAPTER-4: ASSEMBLY LANGUAGES

Machine Language

Machine languages are the lowest level of computer languages. Programs written in machine language consist of entirely of 1s and 0s. Programs in machine language can control directly to the computer's hardware. A machine language instruction consists of two parts: an instruction part and an address part. The instruction part (opcode) is the leftmost group of bits in the instruction and tells the computer the operation to be performed. The address part specifies the memory address of the data to be used in the instruction.

Machine language is the lowest-level computer language and the only language that computers directly understand, a program written in a more sophisticated language (e.g., C, Pascal) must be converted to machine language prior to execution. This is done via a compiler or assembler. The resulting binary file (also called an executable file) can then be executed by the CPU.

Originally, programs were written in machine language. But now programs are written in special programming languages, but these programs must be translated in to the machine language of the computer before the program can be executed. Machine language instructions are represented by binary numbers i.e., sequence consisting of zero's and one's.

For e.g:001010001110 could represent a 12-bit machine language instruction. This instruction is divided into two parts an operation code (or op code) and an operand, e.g.: Op code 001, Operand 010001110

The op code specifies the operation (add, multiply, move.....) and the operand is the address of the data item that is to be operated on. Besides remembering the dozens of code numbers for the operations, the programmer also has to keep track of the addresses of all the data items.

Machine code is extremely difficult for humans to read because it consists merely of patterns of bits (i.e., zeros and ones). Programmers who want to work at the machine code level instead usually use *assembly language*, which is a human-readable. In contrast to high-level languages (e.g., C, C++, Java, Perl and Python), there is a nearly one to one correspondence between a simple assembly language and its corresponding machine language.

Assembly language

What is Assembly Language?

Assembly language is a language like any other language. Assembly language is a low level language. Basically there are two types of programming language, low level and high level language.

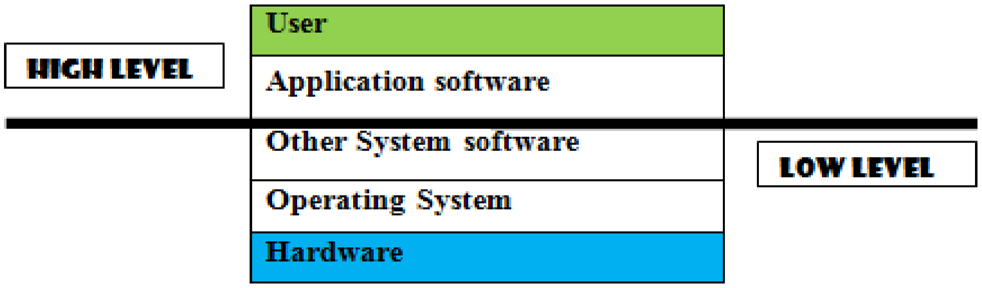


Fig of User view of Computer System

In the above figure Low level means something that can easily understand by the **hardware**. And High level means something that can easily understand by the **User**. So assembly language is languages that can be easily understand by the processor. Assembly language provides us directly what the processor provides you. The processor does not provide such function like **Cin and Cout**. So how do we communicate with the user? It is simple we can use **system call** (or interrupt calls in DOS). The only solution is code them explicitly in our programs.

Mov al, 2h

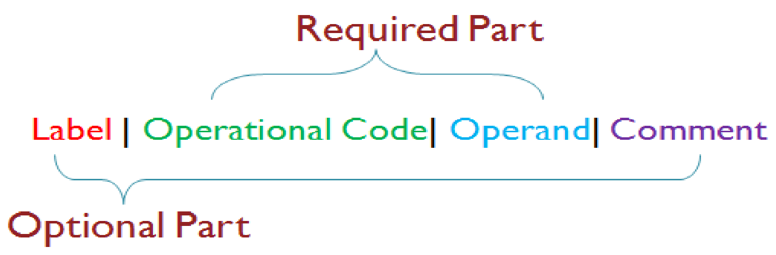
Mov dl, 'a'

Int 21 <--------- *coding interrupt explicitly*

Assembly language is the most basic programming language available for any processor. With assembly language, a programmer works only with operations implemented directly on the physical CPU. Assembly language is the most powerful computer programming language available, and it gives programmers the insight required to write effective code in high-level languages. Assembly language operations are expressed by using **mnemonics or symbolic abbreviations.**

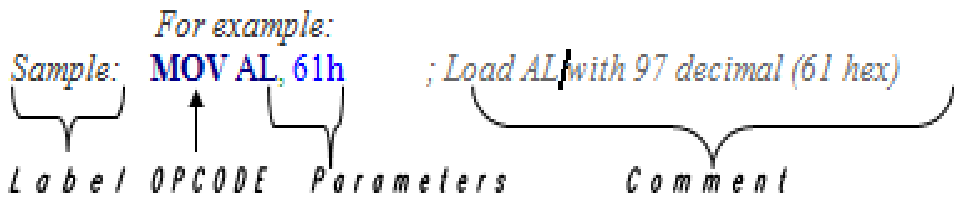
Assembly language structure

A program written in assembly language consists of a series of processor instructions and meta-statements known as directives, pseudo-instructions and pseudo-ops, comments and data. Assembly language instructions usually consist of an **OPCODE** mnemonic followed by a comma-separated list of data, arguments or parameters. Basically assembly language code lines have 4 parts or 2 required and 2 optional.



When we describe the above:

1. **Label**: symbolic names for memory addresses.
2. **Operation code**: the name of the instruction which is to be executed.
3. **Operand**: consists of additional information or data that the OPCODE requires.
4. **Comment**: provide a space for documentation to explain what has been done for the purpose of debugging and maintenance.



Sometimes instructions are used as follows:

add al, [170]

The brackets in the second parameter indicate to us that we are going to work with the content of the memory cell number 170 and not with the 170 value; this is known as **direct addressing**.

Intel assembly language provides the mnemonic MOV (an abbreviation of *move*) for instructions such as this, complete with an explanatory comment if required, after the semicolon. This is much easier to read and to remember.

MOV AL, 61h *; Load AL with 97 decimal (61 hex)*

At one time many assembly language mnemonics were three letter abbreviations, such as JMP for *jump*, INC for *increment*, etc.

Modern processors have a much larger instruction set and many mnemonics are now longer, for example FPATAN for "*floating point particular tangent"* and BOUND for "*check array index against bounds*".

In some assembly languages the same mnemonic such as MOV may refer to a family of related OPCODES for loading, copying and moving data, whether these are immediate values, values in registers, or memory locations pointed to by values in registers. Other assemblers may use separate OPCODES such as L for "move memory to register", ST for "move register to memory", LR for "move register to register", MVI for "move immediate operand to memory", etc.

Comparison of assembly and high level languages

- Assembly language is low level and High level language is high level
- AL can use registers and main memory and HLL uses main memory.
- AL is machine oriented and HLL is human oriented (problem oriented).

Assembly languages are close to a one to one correspondence between symbolic instructions and executable machine codes. Assembly languages also include directives to the assembler, directives to the linker, directives for organizing data space, and macros.

Macros can be used to combine several assembly language instructions into a high level language-like construct. There are cases where a symbolic instruction is translated into more than one machine instruction. But in general, symbolic assembly language instructions correspond to individual executable machine instructions.

High level languages are abstract. Typically a single high level instruction is translated into several executable machine language instructions. Some early high level languages had a close correspondence between high level instructions and machine language instructions. For example, COBOL.

Assembly language is much harder to program than high level languages. The programmer must pay attention to far more detail and must have an intimate knowledge of the processor in use. But high quality hand crafted assembly language programs can run much faster and use much less memory and other resources than a similar program written in a high level language. Speed increases of two to 20 times faster are fairly common, and increases of hundreds of times faster are occasionally possible. Assembly language programming also gives direct access to key machine features essential for implementing certain kinds of low level routines, such as an operating system kernel or microkernel, device drivers, and machine control.

High level programming languages are much easier for less skilled programmers to work in and for semi-technical managers to supervise. And high level languages allow **faster development times** than work in assembly language, even with highly skilled programmers.

Instruction operands

An x86 instruction can have zero to three operands. Operands are separated by commas (,) (ASCII 0x2C). For instructions with two operands, the first (left-hand) operand is the source operand, and the second (right-hand) operand is the destination operand (that is, source→destination). The Intel assembler uses the opposite order (destination←source) for operands.

Operands can be immediate (that is, constant expressions that evaluate to an inline value), register (a value in the processor number registers), or memory (a value stored in memory). An indirect operand contains the address of the actual operand value. Indirect operands are specified by prefixing the operand with an asterisk (*) (ASCII 0x2A). Only jump and call instructions can use indirect operands.

- Immediate operands are prefixed with a dollar sign ($) (ASCII 0x24)
- Register names are prefixed with a percent sign (%) (ASCII 0x25)
- Memory operands are specified either by the name of a variable or by a register that contains the address of a variable.
- A variable name implies the address of a variable and instructs the computer to reference the contents of memory at that address.
- Memory references have the following syntax: segment: offset (base, index, and scale).
- Segment is any of the x86 architecture segment registers.
- Segment is optional: if specified, it must be separated from offset by a colon (:).
- If segment is omitted, the value of %ds (the default segment register) is assumed.
- Offset is the displacement from segment of the desired memory value.
- Offset is optional.
- Base and index can be any of the general 32–bit number registers.
- Scale is a factor by which index is to be multipled before being added to base to specify the address of the operand.
- Scale can have the value of 1, 2, 4, or 8. If scale is not specified, the default value is 1. So see it each and every operands below.

1. Indirect Memory Operands

- Like *direct memory operands*, **indirect memory operands** specify the contents of a given address.
- However, the processor calculates the address at run time by referring to the contents of registers.
- Since values in the registers can change at run time, indirect memory operands provide *dynamic access to memory*.
- Indirect memory operands make possible run-time operations such as pointer indirection and dynamic indexing of array elements, including indexing of multidimensional arrays.
- For example, the following instruction moves into **AX** the word value found at the address in **DS: BX**.
- mov ax, WORD PTR [bx] where
    - ✓ WORD specifies the data size
    - ✓ PTR *re-casts* memory location pointed by **[BX]** into the WORD-sized value.
- When you specify more than one register, the processor adds the contents of the two addresses together to determine the *effective address* (the address of the data to operate on):
- mov ax, [bx+si]

2. Address Displacements

- Address displacement is a *constant value* added to the *effective address*.
- A direct memory specifier is the most common type of displacement:
- table WORD 100 DUP (0)

```
        .
        .
        .
```

- mov ax, table[esi]
- In relocatable expression **table[esi]** , the displacement is **table**, providing the base address of an array
- **ESI** holds an index to an array element. The **ESI** value is calculated at run time, often in a loop.

3. Multiple Address Displacements

- Each displacement can be an address or numeric constant.

- If there is more than one displacement, the assembler totals them at assembly time and encodes the total displacement.
- For example, in the statement
- table WORD 100 DUP (0)

  .

  .

  .
- mov ax, table[bx][di]+6
- Both **table** and **6** are displacements.

4. Indirect Syntax Options

- The assembler allows a variety of syntaxes for *indirect memory operands*.
- However, all registers must be inside brackets.
- Each register can be enclosed in its own pair of brackets, or in the same pair of brackets separated by a **plus** operator (+).
- The following variations are legal and assemble the same way:
  - ✓ mov ax, table[bx][di]
  - ✓ mov ax, table[di][bx]
  - ✓ mov ax, table[bx+di]
  - ✓ mov ax, [table+bx+di]
  - ✓ mov ax, [bx][di]+table

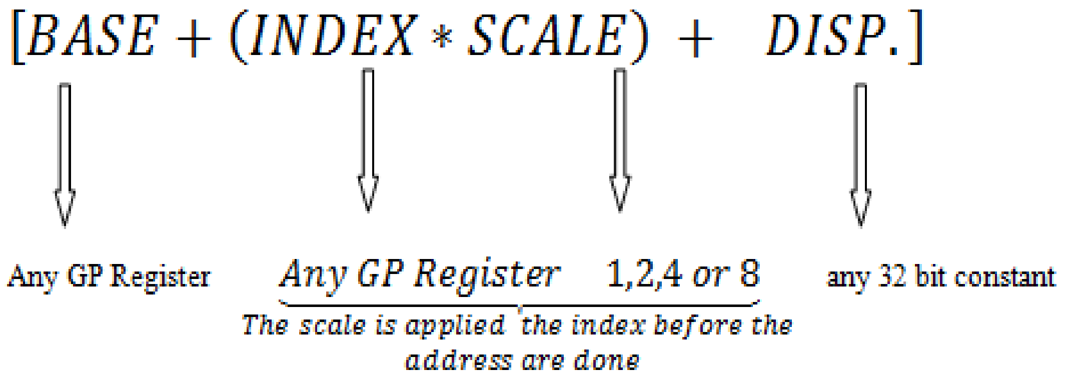
- All of these statements move the value in table indexed by BX+DI into AX

The following instructions are also equivalent:

- ✓ add ax, Table[bx]
- ✓ add ax, [Table + bx]
- ✓ add ax, Table + [bx]
- ✓ add ax, [bx] + Table

1. Scaling Factors

- You can use scaling to index into arrays with different sizes of elements.
- For example, the scaling factor is 1 for **byte** arrays (no scaling needed), 2 for **word** arrays, 4 for **double word** arrays, and 8 for **quad word** arrays.


[BASE + (INDEX * SCALE) + DISP.]
Any GP Register
Any GP Register
1,2,4 or 8
any 32 bit constant
The scale is applied the index before the address are done

2. Relation of Base Registers to Memory Segments

- In indirect memory addressing the *base register* identifies which segment register will be used to calculate the actual memory location.
- Therefore, we need to understand the rules that define which register is the base register in indirect memory addressing mode.
- The default segment register is SS if the base register is EBP or ESP.
- However, if EBP is scaled, the processor treats it as an *index register* with a value relative to DS, not SS.
- All other base registers are relative to DS.
- If two registers are used, only one can have a scaling factor.
- The register with the scaling factor is defined as the *index register*.
- The other register is defined as the *base register*.
- If scaling is not used, the first register is the base.
- If only one register is used, it is considered the base for deciding the default segment, unless it is scaled.
- The following examples illustrate how to determine the base register:

```
✓ mov   eax, [edx][ebp*4] ; EDX base (not scaled - seg DS)
✓ mov   eax, [edx*1][ebp] ; EBP base (not scaled - seg SS)
✓ mov   eax, [edx][ebp]   ; EDX base (first - seg DS)
✓ mov   eax, [ebp][edx]   ; EBP base (first - seg SS)
✓ mov   eax, [ebp]        ; EBP base (only - seg SS)
```

- ✓ mov   eax, [ebp*2]   ; EBP*2 index (seg DS)

3. Addressing Instruction Operands Summary

- **Immediate Mode** (memory is not accessed) - operand is part of the instruction. For example, a constant encoded in the instruction:

```
mov eax,567
mov ah, 09h
mov dx, offset Prompt
```

- **Register Addressing** (memory is not accessed) - operand contained in register:

```
add ax, bx
```

- **Direct Mode** (memory accessed once) - operand field of instruction contains address of the operand:

```
value dword 0
      ..
add eax, value       ; Either notation does the
add eax, [value]     ; same thing
```

❖ Assembly code

```
tbl DW 20 DUP (0)
   ..
mov [tbl], 56
```

is equivalent to C statement

tbl[0] = 56; // **C code**

- **Register indirect addressing** (aka *indirect addressing mode*) often used for addressing data arrays inside programming loops:

❖ Effective address of operand contained in a register.

❖ For 32-bit addressing, all 32-bit registers can be used.

- For 16-bit addressing, the offset value can be in one of the three registers: BX, SI, or DI:

```
mov bx, offset Table  ; Load address
add ax, [bx]          ; Register indirect addressing
```

- Square brackets [BX] indicate that BX is holding a memory offset.
- Operand [BX] serves as a *pointer* to data in memory.
- Register indirect can be used to implement arrays. For example, to sum an array of word-length integers,

```
mov cx, size          ; set up size of Table
mov bx, offset Table  ; BX <- address of Table
xor ax, ax            ; zero out Sum
Loop1:
 add ax, [bx]
inc bx     ; each word is 2 bytes long, so
inc bx      ; need to increment BX twice
loop Loop1
```

- **Indexing**: constant base + register.

  - Fixed Base (address) + Variable Register Offset (operand field contains a constant base)
  - Effective address is obtained by adding value of operand field to contents of register.
  - This is known as *array type addressing*, also called *displacement addressing*.

```
mov eax, [ ebx + 5 ]
mov eax, [ ebx + esi + 5 ]
```

  - There are restrictions on the combinations of registers allowed within the brackets.
  - You can have ESI or EDI, but not both, and you can have EBX or EBP, but not both.
  - Note for large operand E is added before Register name to get extended register with 32-bi.

- **Stack Addressing**: PUSH and POP, a variant of register indirect with auto-increment/decrement using the ESP register implicitly.

Basic Instructions

There is a large degree of diversity in the way the authors of assemblers categorize statements and in the nomenclature that they use. In particular, some describe anything other than a machine mnemonic or extended mnemonic as a pseudo-operation (pseudo-op). A typical assembly language consists of 3 types of instruction statements that are used to define program operations:

- Opcode mnemonics
- Data sections
- Assembly directives

Opcode mnemonics and extended mnemonics

Instructions (statements) in assembly language are generally very simple, unlike those in high-level language. Generally, a mnemonic is a symbolic name for a single executable machine language instruction (an opcode), and there is at least one opcode mnemonic defined for each machine language instruction. Each instruction typically consists of an *operation* or *opcode* plus zero or more *operands*. Most instructions refer to a single value, or a pair of values. Operands can be immediate (typically one byte values, coded in the instruction itself), registers specified in the instruction, implied or the addresses of data located elsewhere in storage. This is determined by the underlying processor architecture: We have the following segments in our program.

1. Assembly directives

Assembly directives, also called pseudo opcode, pseudo-operations or pseudo-ops, are instructions that are executed by an assembler at assembly time, not by a CPU at run time. They can make the assembly of the program dependent on parameters input by a programmer, so that one program can be assembled different ways, perhaps for different applications. They also can be used to manipulate presentation of a program to make it easier to read and maintain.

2. Data sections

There are instructions used to define data elements to hold data and variables. They define the type of data, the length and the alignment of data. These instructions can also define whether the data is available to outside programs (programs assembled separately) or only to the program in which the data section is defined. Some assemblers classify these as pseudo-ops.

3. Macros

Many assemblers support *predefined macros*, and others support *programmer-defined* (and repeatedly re-definable) macros involving sequences of text lines in which variables and constants are embedded. This sequence of text lines may include opcode or directives. Once a macro has been defined its name may be used in place of a mnemonic. When the assembler processes such a statement, it replaces the statement with the text lines associated with that macro, and then processes them as if they existed in the source code file.

Since macros can have 'short' names but expand to several or indeed many lines of code, they can be used to make assembly language programs appear to be far shorter, requiring fewer lines of source code, as with higher level languages. They can also be used to add higher levels of structure to assembly programs, optionally introduce embedded debugging code via parameters and other similar features.

4. Code section

In the code section we will write actual program containing instructions to perform the required task.

Compiler Dependencies

Emu8086 is an 8086 microprocessor emulator and disassembler. It permits to assemble, emulate and debug 8086 programs (16bit/DOS). Although this program was made for *Windows*, it works fine on GNU/Linux (with the help of *Wine*).

The problems are that Emu8086 can opens only one file at time, and that I dislike its text editor… I used another editor for writing my program and then I opened the main file of my project in Emu8086 every time I wanted to compile it. That's boring and repetitive, so I made a build environment that uses the standard make command for building my program. It is easy to use and efficient.

Runtime Dependencies

For running your application, you will need to install few supporting software.

- DOSBox (for 64-bit system)

Running your program with DOSBox, If the build was successful, two files are created in your source folder:

- myprog.com or myprog.exe - This is your application. It can be executed on **DOS* or on a 32 bit *Windows*.
- myprog.sh - This is a *Bash* shell script for GNU/Linux that launch your application in *DOSBox*.

Bugging Dependency

Debugging also depends on system to system and os to os. Different debuggers are required for debugging on the basis of OS.

Linking-Dependencies

- The Emu8086 Building repository

1. For launching emu8086.exe and for making the building
2. The extension depends of your code. If your code is made for being a flat binary, you have to set this variable to **com**; if you have different segments in your code, you have to set **exe**. For Linux and Ubuntu you have to set .**sh** for running on Linux shell, .boot for bootable program (which needs to be executing during booting process.

Assembling, Linking and Executing

1. **Assembling:**

   - Assembling converts source program into object program if syntactically correct and generates an intermediate **.obj** file or module.

   - It calculates the offset address for every data item in data segment and every instruction in code segment.

   - A header is created which contains the incomplete address in front of the generated **obj** module during the assembling.

   - Assembler complains about the syntax error if any and does not generate the object module.

   - Assembler creates **.obj .lst** and **.crf** files and last two are optional files that can be created at run time.

   - For short programs, assembling can be done manually where the programmer translates each mnemonic into the machine language using lookup table.

- Assembler reads each assembly instruction of a program as ASCII character and translates them into respective machine code.

2. **Linking:**

- This involves the converting of **.OBJ** module into **.EXE**(executable) module i.e. executable machine code.

- It completes the address left by the assembler.

-It combines separately assembled object files.

- Linking creates **.EXE**, **.LIB**, **.MAP** files among which last two are optional files.

3. **Loading and Executing:**

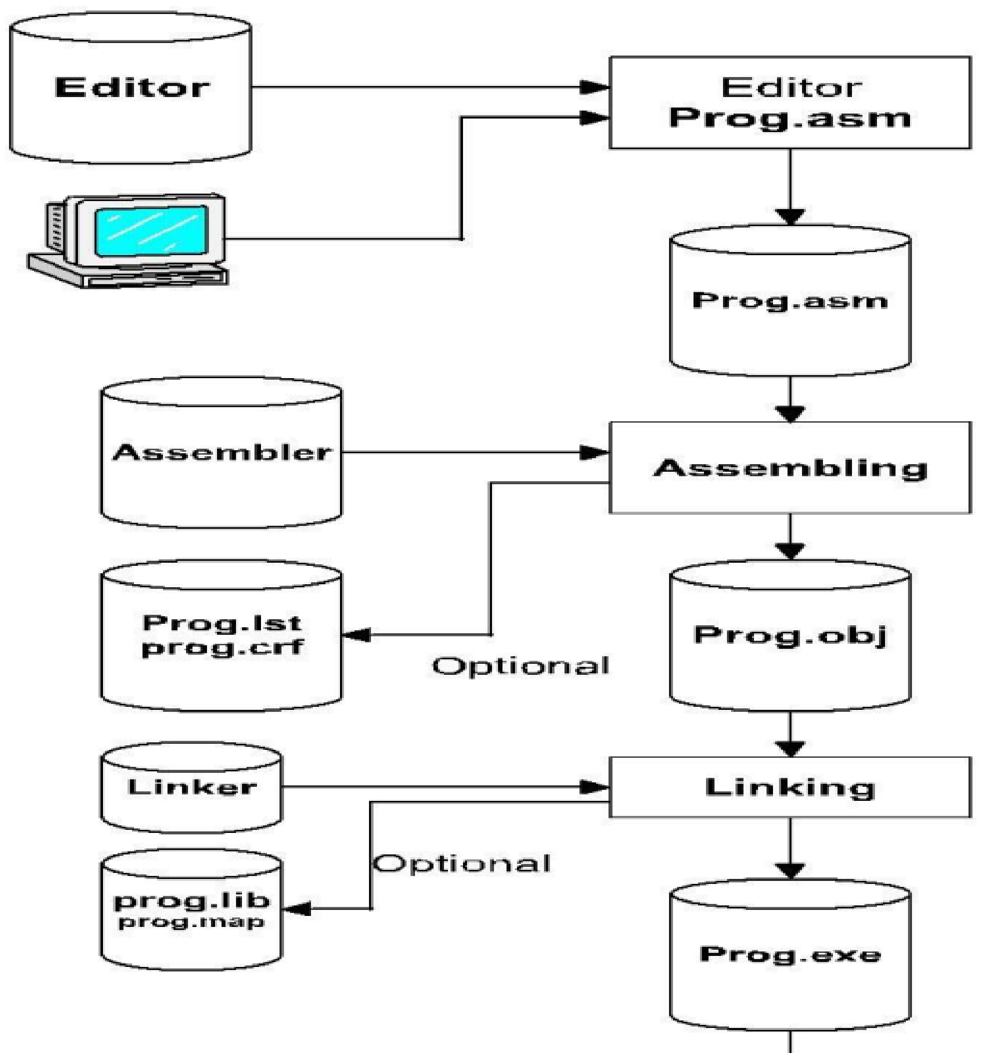
- It Loads the program in memory for execution.

- It resolves remaining address.

- This process creates the program segment prefix (PSP) before loading.

- It executes to generate the result.

$$\text{Sample program} \xrightarrow{\text{assembling}} \text{object Program} \xrightarrow{\text{linking}} \text{executable program}$$



Assembly Directive

Directives are commands that are part of the assembler syntax but are not related to the x86 processor instruction set. All assembler directives begin with a period (.) (ASCII 0x2E). Some of them are:

.align integer, pad

- The .align directive causes the next data generated to be aligned modulo integer bytes.
- Integer must be a positive integer expression and must be a power of 2.
- If specified, pad is an integer bye value used for padding.
- The default value of pad for the text section is 0x90 (nop); for other sections, the default value of pad is zero (0).

.ascii "string"

- The .ascii directive places the characters in string into the object module at the current location but does not terminate the string with a null byte (\0).
- String must be enclosed in double quotes (") (ASCII 0x22).
- The .ascii directive is not valid for the .bss section.

.bcd integer

- The .bcd directive generates a packed decimal (80-bit) value into the current section.
- The .bcd directive is not valid for the .bss section.

.bss

- The .bss directive changes the current section to .bss.

.bss symbol, integer

- Define symbol in the .bss section and add integer bytes to the value of the location counter for .bss.
- When issued with arguments, the .bss directive does not change the current section to .bss. Integer must be positive.

.byte byte1, byte2... byteN

- The .byte directive generates initialized bytes into the current section.
- The .byte directive is not valid for the .bss section.
- Each byte must be an 8-bit value.

.2byte expression1, expression2... expressionN

- Refer to the description of the .value directive.

.4byte expression1, expression2, ..., expressionN

- Refer to the description of the .long directive.

.8byte expression1, expression2, ..., expressionN

- Refer to the description of the .quad directive.

.comm name, size, alignment

- The .comm directive allocates storage in the data section.
- The storage is referenced by the identifier name.
- Size is measured in bytes and must be a positive integer.
- Name cannot be predefined. Alignment is optional.
- If alignment is specified, the address of name is aligned to a multiple of alignment.

.data

- The .data directive changes the current section to .data.

.double float

- The .double directive generates a double-precision floating-point constant into the current section.
- The .double directive is not valid for the .bss section.

.even

- The .even directive aligns the current program counter (.) to an even boundary.

.ext expression1, expression2, ..., expressionN

- The .ext directive generates an 80387 80–bit floating point constant for each expression into the current section.
- The .ext directive is not valid for the .bss section.

.file "string"

- The .file directive creates a symbol table entry where string is the symbol name and STT_FILE is the symbol table type.
- String specifies the name of the source file associated with the object file.

.float float

- The .float directive generates a single-precision floating-point constant into the current section.
- The .float directive is not valid in the .bss section.

.globl symbol1, symbol2, ..., symbolN

- The .globl directive declares each symbol in the list to be global.
- Each symbol is either defined externally or defined in the input file and accessible in other files.
- Default bindings for the symbol are overridden.

- A global symbol definition in one file satisfies an undefined reference to the same global symbol in another file.
- Multiple definitions of a defined global symbol are not allowed.
- If a defined global symbol has more than one definition, an error occurs.
- The .globl directive only declares the symbol to be global in scope, it does not define the symbol.

.group group, section, #comdat

- The .group directive adds section to a COMDATgroup.

.hidden symbol1, symbol2, ..., symbolN

- The .hidden directive declares each symbol in the list to have hidden linker scoping.
- All references to symbol within a dynamic module bind to the definition within that module.
- Symbol is not visible outside of the module.

.ident "string"

- The .ident directive creates an entry in the .comment section containing string.
- String is any sequence of characters, not including the double quote (").
- To include the double quote character within a string, precede the double quote character with a backslash (\) (ASCII 0x5C).

.lcomm name, size, alignment

- The .lcomm directive allocates storage in the .bss section.
- The storage is referenced by the symbol name, and has a size of size bytes.
- Name cannot be predefined, and size must be a positive integer.
- If alignment is specified, the address of name is aligned to a multiple of alignment bytes.
- If alignment is not specified, the default alignment is 4 bytes.

.local symbol1, symbol2, ..., symbolN

- The .local directive declares each symbol in the list to be local.
- Each symbol is defined in the input file and not accessible to other files.
- Default bindings for the symbols are overridden.
- Symbols declared with the .local directive take precedence over weak and global symbols.
- Because local symbols are not accessible to other files, local symbols of the same name may exist in multiple files.
- The .local directive only declares the symbol to be local in scope, it does not define the symbol.

.long expression1, expression2, ..., expressionN

- The .long directive generates a long integer (32-bit, two's complement value) for each expression into the current section.
- Each expression must be a 32–bit value and must evaluate to an integer value.
- The .long directive is not valid for the .bss section.

.popsection

- The .popsection directive pops the top of the section stack and continues processing of the popped section.

.previous

- The .previous directive continues processing of the previous section.

.pushsection section

- The .pushsection directive pushes the specified section onto the section stack and switches to another section.

.quad expression1, expression2, ..., expressionN

- The .quad directive generates an initialized word (64-bit, two's complement value) for each expression into the current section.
- Each expression must be a 64-bit value, and must evaluate to an integer value.
- The .quad directive is not valid for the .bss section.

.rel symbol@ type

- The .rel directive generates the specified relocation entry type for the specified symbol.
- The .lit directive supports TLS (thread-local storage

.section section attributes

- The .section directive makes section the current section.
- If section does not exist, a new section with the specified name and attributes is created.
- If section is a non-reserved section, attributes must be included the first time section is specified by the .section directive.

.set symbol, expression

- The .set directive assigns the value of expression to symbol.
- Expression can be any legal expression that evaluates to a numerical value.

.skip integer, value

- While generating values for any data section, the .skip directive causes integer bytes to be skipped over, or, optionally, filled with the specified value.

.sleb128 expression

- The .sleb128 directive generates a signed, little-endian, base 128 number from expression.

.string "string"

- The .string directive places the characters in string into the object module at the current location and terminates the string with a null byte (\0).
- String must be enclosed in double quotes (") (ASCII 0x22).
- The .string directive is not valid for the .bss section.

.symbolic symbol1, symbol2... symbolN

- The .symbolic directive declares each symbol in the list to have symbolic linker scoping.
- All references to symbol within a dynamic module bind to the definition within that module.
- Outside of the module, symbol is treated as global.

.tbss

- The .tbss directive changes the current section to .tbss.
- The .tbss section contains uninitialized TLS data objects that will be initialized to zero by the runtime linker.

.tcomm

- The .tcomm directive defines a TLS common block.

.tdata

- The .tdata directive changes the current section to .tdata.
- The .tdata section contains the initialization image for initialized TLS data objects.

.text

- The .text directive defines the current section as .text.

.uleb128 expression

- The .uleb128 directive generates an unsigned, little-endian, base 128 number from expression.

.value expression1, expression2, ..., expressionN

- The .value directive generates an initialized word (16-bit, two's complement value) for each expression into the current section.
- Each expression must be a 16-bit integer value.
- The .value directive is not valid for the .bss section.

.weak symbol1, symbol2, ..., symbolN

- The .weak directive declares each symbol in the argument list to be defined either externally or in the input file and accessible to other files.
- Default bindings of the symbol are overridden by the .weak directive.
- A weak symbol definition in one file satisfies an undefined reference to a global symbol of the same name in another file.
- Unresolved weak symbols have a default value of zero.
- The link editor does not resolve these symbols.
- If a weak symbol has the same name as a defined global symbol, the weak symbol is ignored and no error results.
- The .weak directive does not define the symbol.

.zero expression

- While filling a data section, the .zero directive fills the number of bytes specified by expression with zero (0).

Summary

- Machine languages are the lowest level of computer languages & consist of entirely of 1s and 0s.
- Machine language instruction is divided into two parts: - an operation code (or op code) and an operand.
- Assembly language is a low level language which is easily understandable by hardware.
- A program written in assembly language consists of a series of processor instructions and meta-statements known as directives, pseudo-instructions and pseudo-ops, comments and data.
- Assembly language instructions usually consist of an **OPCODE** mnemonic followed by a comma-separated list of data, arguments or parameters.
- Basically assembly language code lines have 4 parts or 2 required and 2 optional. Those are: label, Operation code, Operand & Comment.
- Directives are commands that are part of the assembler syntax but are not related to the x86 processor instruction set & they begins with a period (.) (ASCII 0x2E).
- An x86 instruction can have zero to three operands.

CHAPTER-5- WRITING ASSEMBLY LANGUAGE PROGRAMS

Assembly programming language is a language closer to what machines can understand. Assembly language is an example of low level language. A short code (mnemonics) is written for each instruction in assembly language programming. So, instead of having to remember a string of 0's and 1's, the programmer would only need to remember short codes like ADD, SUB, DIV, JMP, MOV called mnemonics.

Assembly language programming is just abbreviation of machine language, so it also not user friendly. Programmers yet have to write long codes for small program. But many programs are written in assembly language as it is closer to machine language and execution time is faster.

Again, assembly language is also processor dependent or incompatible for different machine. Program written for MOTOROLA processor won't work in INTEL processor.

Computers or other electronic devices doesn't understand Assembly Language or mnemonics on its own. We know computer only understands instruction in forms of 0's and 1's. Assemblers translates menomics into machine language or binary codes. Assembler is a program which converts "ASSEMBLY level instruction" into "MACHINE level instruction".

Advantages of assembly language

Since mnemonics replace machine instruction it is easy to write, debug and understand is comparison to machine codes.

Useful to write lightweight application (in embedded system like traffic light) because it needs fewer codes than high level language.

Disadvantages of assembly language

Mnemonics are in abbreviated form and in large number, so they are hard to remember.

Program written in assembly language are machine dependent, so are incompatible for different type of machines.

A program written in assembly language is less efficient to same program in machine language.

Mnemonics can be different for different machines according to manufacturers. So assembly language suffers from the defect of non-standardization

Control Structures

High level languages provide high level control structures (e.g., the if and while statements) that control the thread of execution. Assembly language does not provide such complex control structures. It instead uses the infamous goto and used inappropriately can result in spaghetti code! How-ever, it is possible to write structured assembly language programs. The basic procedure is to design the program logic using the familiar high level control structures and translate the design into the appropriate assembly language (much like a compiler would do).

Comparisons

Control structures decide what to do based on comparisons of data. In assembly, the result of a comparison is stored in the FLAGS register to be used later. The 80x86 provides the CMP instruction to perform comparisons.

The FLAGS register is set based on the difference of the two operands of the CMP instruction. The operands are subtracted and the FLAGS are set based on the result, but the result is not stored anywhere. If you need the result use the SUB instead of the CMP instruction.

For unsigned integers, there are two flags (bits in the FLAGS register) that are important: the zero (ZF) and carry (CF) flags. The zero flag is set (1) if the resulting difference would be zero. The carry flag is used as a borrow flag for subtraction. Consider a comparison like:

```
cmp vleft, vright
```

The difference of vleft - vright is computed and the flags are set accord-ingly. If the difference of the of CMP is zero, vleft = vright, then ZF is set (i.e. 1) and the CF is unset (i.e. 0). If vleft > vright, then ZF is unset and CF is unset (no borrow). If vleft < vright, then ZF is unset and CF is set (borrow).

For signed integers, there are three flags that are important: the zero (ZF) flag, the overflow (OF) flag and the sign (SF) flag. The overflow flag is set if the result of an operation overflows (or underflows). The sign flag is set if the result of an operation is negative. If vleft = vright, the ZF is set (just as for unsigned integers). If vleft > vright, ZF is unset and SF = OF. If vleft < vright, ZF is unset and SF 6 = OF.

Do not forget that other instructions can also change the FLAGS register, not just CMP

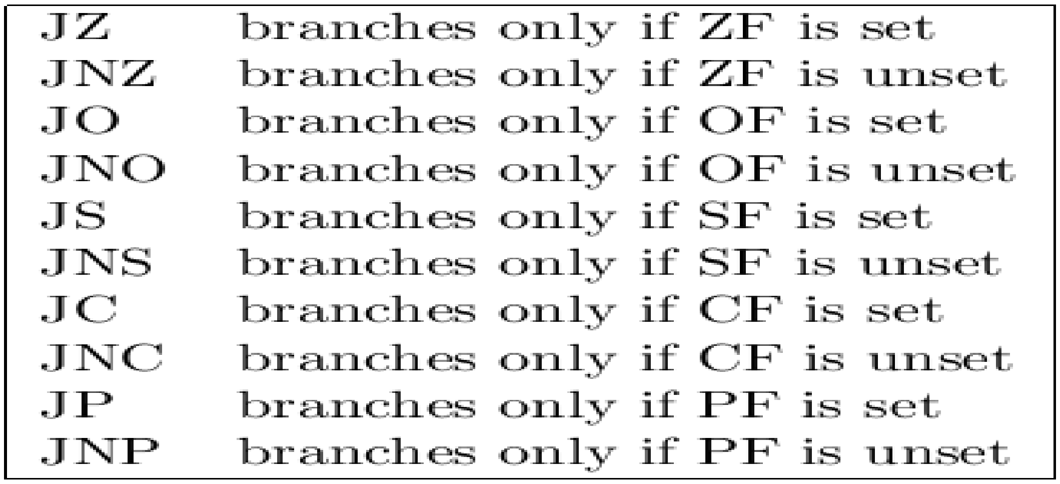
Branch instructions

Branch instructions can transfer execution to arbitrary points of a pro-gram. In other words, they act like a goto. There are two types of branches: unconditional and conditional. An unconditional branch is just like a goto, it always makes the branch. A conditional branch may or may not make the branch depending on the flags in the FLAGS register. If a conditional branch does not make the branch, control passes to the next instruction. The JMP (short for jump) instruction makes unconditional branches. Its single argument is usually a code label to the instruction to branch to. The assembler or linker will replace the label with correct address of the instruction. This is another one of the tedious operations that the assembler does to make the programmer's life easier. It is important to realize that the statement immediately after the JMP instruction will never be executed unless another instruction branches to it!

There are several variations of the jump instruction:

SHORT. This jump is very limited in range. It can only move up or down 128 bytes in memory. The advantage of this type is that it uses memory than the others. It uses a single signed byte to store the displacement of the jump.

The displacement is how many bytes to move ahead or behind. (The displacement is added to EIP). To specify a short jump, use the SHORT keyword immediately before the label in the JMP instruction.



JZ	branches only if ZF is set
JNZ	branches only if ZF is unset
JO	branches only if OF is set
JNO	branches only if OF is unset
JS	branches only if SF is set
JNS	branches only if SF is unset
JC	branches only if CF is set
JNC	branches only if CF is unset
JP	branches only if PF is set
JNP	branches only if PF is unset

Table: Simple Conditional Branches

NEAR. This jump is the default type for both unconditional and conditional branches; it can be used to jump to any location in a segment. Actually, the 80386 supports two types of near jumps. One uses two bytes for the displacement. This allows one to move up or down roughly 32,000 bytes. The other type uses four bytes for the displacement, which of course allows one to move to any location in the code segment. The four byte type is the default in 386 protected modes. The two byte type can be specified by putting the WORD keyword before the label in the JMP instruction.

FAR. This jump allows control to move to another code segment. This is a very rare thing to do in 386 protected modes. Valid code labels follow the same rules as data labels. Code labels are defined by placing them in the code segment in front of the statement they label. A colon is placed at the end of the label at its point of definition. The colon is not part of the name.

There are many different conditional branch instructions. They also take a code label as their single operand. The simplest ones just look at a single flag in the FLAGS register to determine whether to branch or not. See the above table for a list of these instructions. (PF is the parity flag which indicates the odd or evenness of the number of bits set in the lower 8-bits of the result.)

The following pseudo-code:

```
    if ( EAX == 0 )
        EBX = 1;
    else
        EBX = 2;
```

could be written in assembly as:

```
        cmp     eax, 0          ; set flags (ZF set if eax - 0 = 0)
        jz      then block      ; if ZF is set branch to then block
        mov     ebx, 2          ; ELSE part of IF
        jmp     next            ; jump over THEN part of IF
  then block:
         mov    ebx, 1          ; THEN part of IF
  next:
```

Other comparisons are not so easy using the conditional branches. To illustrate, consider the following pseudo-code:

```
    if ( EAX >= 5 )
        EBX = 1;
    else
        EBX = 2;
```

If EAX is greater than or equal to five, the ZF may be set or unset and
SF will equal OF. Here is assembly code that tests for these conditions
(assuming that EAX is signed):

```
        cmp     eax, 5
        js      sign on             ; goto sign on if SF = 1
        jo      else block      ; goto else block if OF = 1 and SF = 0
        jmp     then block      ; goto then block if SF = 0 and OF = 0
 sign on:
        jo      then block      ; goto then block if SF = 1 and OF = 1
 else block:
        mov     ebx, 2
        jmp     next
 then block:
        mov     ebx, 1
 next:
```

The above code is very awkward. Fortunately, the 80x86 provides additional branch instructions to make these type of tests much easier. There are signed and unsigned versions of each. The next shows these instructions. The equal and not equal branches (JE and JNE) are the same for both signed and unsigned integers. (In fact, JE and JNE are really identical to JZ and JNZ, respectively.) Each of the other branch instructions have two synonyms. For example, look at JL (jump less than) and JNGE (jump not greater than or equal to). These are the same instruction because:

$$x < y = \Rightarrow \text{not}(x \geq y)$$

The unsigned branches use A for above and B for below instead of L and G.

Signed		**Unsigned**	
JE	branches if `vleft` = `vright`	JE	branches if `vleft` = `vright`
JNE	branches if `vleft` ≠ `vright`	JNE	branches if `vleft` ≠ `vright`
JL, JNGE	branches if `vleft` < `vright`	JB, JNAE	branches if `vleft` < `vright`
JLE, JNG	branches if `vleft` ≤ `vright`	JBE, JNA	branches if `vleft` ≤ `vright`
JG, JNLE	branches if `vleft` > `vright`	JA, JNBE	branches if `vleft` > `vright`
JGE, JNL	branches if `vleft` ≥ `vright`	JAE, JNB	branches if `vleft` ≥ `vright`

Table: Signed and Unsigned Comparison Instructions

Using these new branch instructions, the pseudo-code above can be translated to assembly much easier.

```
        cmp     eax, 5
        jge     then block
        mov     ebx, 2
```

```
            jmp     next
  then block:
 mov     ebx, 1
  next:
```

The loop instructions

The 80x86 provides several instructions designed to implement for –like loops. Each of these instructions takes a code label as its single operand.

LOOP Decrements ECX, if ECX 6 = 0, branches to label

LOOPE, LOOPZ Decrements ECX (FLAGS register is not modified), if
    ECX 6 = 0 and ZF = 1, branches

LOOPNE, LOOPNZ Decrements ECX (FLAGS unchanged), if ECX 6 =
    0 and ZF = 0, branches

The last two loop instructions are useful for sequential search loops. The following pseudo-code:

```
        sum = 0;
        for ( i=10; i >0; i −− )
          sum += i;
```

could be translated into assembly as:

```
        mov     eax, 0          ; eax is sum
        mov     ecx, 10         ; ecx is i
 loop_start:
        add     eax, ecx
        loop    loop_start
```

Translating Standard Control Structures

This section looks at how the standard control structures of high level languages can be implemented in assembly language.

If statements

The following pseudo-code:

```
        if ( condition )
                then block ;
        else
                else block ;
```

could be implemented as:

```
        ; code to set FLAGS
        jxx else_block ; select xx so that branches if condition false
        ; code for then block
        jmp endif
  else_block:
        ; code for else block
  endif:
```

If there is no else, then the else_block branch can be replaced by a branch to endif.

```
        ; code to set FLAGS
        jxx endif ; select xx so that branches if condition false
        ; code for then block
         endif:
```

While loops

The while loop is a top tested loop:

```
        while( condition ) {
          body of loop;
        }
```

This could be translated into:

```
  while:
        ; code to set FLAGS based on condition
        jxx endwhile            ; select xx so that branches if false
;       body of loop
        jmp while
  endwhile:
```

Do while loops

The do while loop is a bottom tested loop:

```
        do {
```

```
  body of loop;
} while( condition );
```

This could be translated into:

```
do:
    ; body of loop
    ; code to set FLAGS based on condition
    jxx do ; select xx so that branches if true
```

CHAPTER-6- PROCEDURES AND FUNCTIONS

Procedures

In a procedural environment, the basic unit of code is the procedure. A procedure is a set of instructions that compute some value or take some action (such as printing or reading a character value). The definition of a procedure is very similar to the definition of an algorithm. A procedure is a set of rules to follow which, if they conclude, produce some result. An algorithm is also such a sequence, but an algorithm is guaranteed to terminate whereas a procedure offers no such guarantee.

Most procedural programming languages implement procedures using the call/return mechanism. That is, some code calls a procedure, the procedure does its thing, and then the procedure returns to the caller. The call and return instructions provide the 80x86's procedure invocation mechanism. The calling code calls a procedure with the call instruction, the procedure returns to the caller with the ret instruction. For example, the following 80x86 instruction calls the UCR Standard Library sl_putcr routine:

```
                call    sl_putcr
```

sl_putcr prints a carriage return/line feed sequence to the video display and returns control to the instruction immediately following the call sl_putcr instruction.

Alas, the UCR Standard Library does not supply all the routines you will need. Most of the time you'll have to write your own procedures. A simple procedure may consist of nothing more than a sequence of instructions ending with a ret instruction. For example, the following "procedure" zeros out the 256 bytes starting at the address in the bx register:

```
ZeroBytes:      xor     ax, ax
                mov     cx, 128
ZeroLoop:       mov     [bx], ax
                add      bx, 2
                loop    ZeroLoop
                ret
```

By loading the bx register with the address of some block of 256 bytes and issuing a call ZeroBytes instruction, you can zero out the specified block.

As a general rule, you won't define your own procedures in this manner. Instead, you should use MASM's proc and endp assembler directives. The ZeroBytes routine, using the proc and endp directives, is

```
ZeroBytes       proc
```

```
                xor         ax, ax
                mov         cx, 128
ZeroLoop:       mov         [bx], ax
                add         bx, 2
                loop        ZeroLoop
                ret
ZeroBytes       endp
```

Keep in mind that proc and endp are assembler directives. They do not generate any code. They're simply a mechanism to help make your programs easier to read. To the 80x86, the last two examples are identical; however, to a human being, latter is clearly a self-contained procedure, the other could simply be an arbitrary set of instructions within some other procedure. Consider now the following code:

```
ZeroBytes:      xor         ax, ax
                jcxz        DoFFs
ZeroLoop:       mov         [bx], ax
                add         bx, 2
                loop        ZeroLoop
                ret
DoFFs:          mov         cx, 128
                mov         ax, 0ffffh
FFLoop:         mov         [bx], ax
                sub         bx, 2
                loop        FFLoop
                ret
```

Are there two procedures here or just one? In other words, can a calling program enter this code at labels ZeroBytes and DoFFs or just at ZeroBytes? The use of the proc and endp directives can help remove this ambiguity:

Treated as a single subroutine:

```
ZeroBytes       proc
                xor         ax, ax
                jcxz        DoFFs
```

```
ZeroLoop:       mov         [bx], ax
                add         bx, 2
                loop        ZeroLoop
                ret
DoFFs:          mov         cx, 128
                mov         ax, 0ffffh
FFLoop:         mov         [bx], ax
                sub         bx, 2
                loop        FFLoop
                ret
ZeroBytes       endp
```

Treated as two separate routines:

```
ZeroBytes       proc
                xor         ax, ax
                jcxz        DoFFs
ZeroLoop:       mov         [bx], ax
                add         bx, 2
                loop        ZeroLoop
                ret
ZeroBytes       endp
DoFFs           proc
                mov         cx, 128
                mov         ax, 0ffffh
FFLoop:         mov         [bx], ax
                sub         bx, 2
                loop        FFLoop
                ret
DoFFs           endp
```

Always keep in mind that the proc and endp directives are logical procedure separators. The 80x86 microprocessor returns from a procedure by executing a ret instruction, not by encountering an endp directive. The following is not equivalent to the code above:

```
ZeroBytes       proc
                xor         ax, ax
                jcxz        DoFFs
ZeroLoop:       mov         [bx], ax
                add         bx, 2
                loop        ZeroLoop
;       Note missing RET instr.
ZeroBytes       endp
DoFFs           proc
                mov         cx, 128
                mov         ax, 0ffffh
FFLoop:         mov         [bx], ax
                sub         bx, 2
                loop        FFLoop
;       Note missing RET instr.
DoFFs           endp
```

Without the ret instruction at the end of each procedure, the 80x86 will fall into the next subroutine rather than return to the caller. After executing ZeroBytes above, the 80x86 will drop through to the DoFFs subroutine (beginning with the mov cx, 128instruction).

Once DoFFs is through, the 80x86 will continue execution with the next executable instruction following DoFFs' endp directive.

An 80x86 procedure takes the form:

```
ProcName        proc        {near|far}      ;Choose near, far, or neither.
        <Procedure instructions>
ProcName        endp
```

The near or far operand is optional, the next section will discuss its purpose. The procedure name must be on the both proc and endp lines. The procedure name must be unique in the program.

Every proc directive must have a matching endp directive. Failure to match the proc and endp directives will produce a block nesting error.

Near and Far Procedures

The 80x86 supports near and far subroutines. Near calls and returns transfer control between procedures in the same code segment. Far calls and returns pass control between different segments. The two calling and return mechanisms push and pop different return addresses. You generally do not use a near call instruction to call a far procedure or a far call instruction to call a near procedure. Given this little rule, the next question is "how do you control the emission of a near or far call or ret?"

Most of the time, the call instruction uses the following syntax:

```
                call        ProcName
```

and the ret instruction is either:

```
                ret
or              ret          disp
```

Unfortunately, these instructions do not tell MASM if you are calling a near or far procedure or if you are returning from a near or far procedure. The proc directive handles that chore. The proc directive has an optional operand that is either near or far. Near is the default if the operand field is empty. The assembler assigns the procedure type (near or far) to the symbol. Whenever MASM assembles a call instruction, it emits a near or far call depending on operand. Therefore, declaring a symbol with proc or proc near, forces a near call. Likewise, using proc far, forces a far call.

Besides controlling the generation of a near or far call, proc's operand also controls ret code generation. If a procedure has the near operand, then all return instructions inside that procedure will be near. MASM emits far returns inside far procedures.

Nested Procedures

MASM allows you to nest procedures. That is, one procedure definition may be totally enclosed inside another. The following is an example of such a pair of procedures:

```
OutsideProc     proc        near
                jmp         EndofOutside
InsideProc      proc        near
                mov         ax, 0
                ret
InsideProc      endp
EndofOutside:   call        InsideProc
                mov         bx, 0
```

```
                 ret
OutsideProc      endp
```

Unlike some high level languages, nesting procedures in 80x86 assembly language doesn't serve any useful purpose. If you nest a procedure (as with InsideProc above), you'll have to code an explicit jmp around the nested procedure. Placing the nested procedure after all the code in the outside procedure (but still between the outside proc/endp directives) doesn't accomplish anything. Therefore, there isn't a good reason to nest procedures in this manner.

Whenever you nest one procedure within another, it must be totally contained within the nesting procedure. That is, the proc and endp statements for the nested procedure must lie between the proc and endp directives of the outside, nesting, procedure. The following is not legal:

```
OutsideProc      proc      near
                   .
                   .
InsideProc       proc      near
                   .
                   .
Outside          Proc      endp
                   .
                   .
InsideProc       endp
```

The OutsideProc and InsideProc procedures overlap, they are not nested. If you attempt to create a set of procedures like this, MASM would report a "block nesting error".

Functions

The difference between functions and procedures in assembly language is mainly a matter of definition. The purpose for a function is to return some explicit value while the purpose for a procedure is to execute some action. To declare a function in assembly language, use the proc/endp directives. All the rules and techniques that apply to procedures apply to functions. From here on, procedure will mean procedure or function.

Saving the state of Machine

Take a look at this code:

```
                mov         cx, 10
Loop0:          call        PrintSpaces
                putcr
                loop        Loop0
                  .
                  .
PrintSpaces     proc        near
                mov         al, ‘ ‘
                mov         cx, 40
PSLoop:         putc
                loop        PSLoop
                ret
PrintSpaces     endp
```

This section of code attempts to print ten lines of 40 spaces each. Unfortunately, there is a subtle bug that causes it to print 40 spaces per line in an infinite loop. The main pro-gram uses the loop instruction to call PrintSpaces10 times. PrintSpaces uses cx to count off the 40 spaces it prints. PrintSpaces returns with cx containing zero. The main program then prints a carriage return/line feed, decrements cx, and then repeats because cx isn’t zero (it will always contain 0FFFFh at this point).

The problem here is that the PrintSpaces subroutine doesn’t preserve the cx register. Preserving a register means you save it upon entry into the subroutine and restore it before leaving. Had the PrintSpaces subroutine preserved the contents of the cx register, the program above would have functioned properly.

Use the 80x86’s push and pop instructions to preserve register values while you need to use them for something else. Consider the following code for PrintSpaces:

```
PrintSpaces     proc        near
                push        ax
                push        cx
                mov         al, ‘ ‘
                mov         cx, 40
PSLoop:         putc
```

```
                    loop        PSLoop
                    pop         cx
                    pop         ax
                    ret
PrintSpaces         endp
```

Note that PrintSpaces saves and restores ax and cx(since this procedure modifies these registers). Also, note that this code pops the registers off the stack in the reverse order that it pushed them. The operation of the stack imposes this ordering.

Either the caller (the code containing the call instruction) or the callee (the subroutine) can take responsibility for preserving the registers. In the example above, the callee pre-served the registers. The following example shows what this code might look like if the caller preserves the registers:

```
                    mov         cx, 10
Loop0:              push        ax
                    push        cx
                    call        PrintSpaces
                    pop         cx
                    pop         ax
                    putcr
                    loop        Loop0
                     .
                     .
PrintSpaces          proc       near
                    mov         al, ‘ ‘
                    mov         cx, 40
PSLoop:             putc
                    loop        PSLoop
                    ret
PrintSpaces         endp
```

There are two advantages to callee preservation: space and maintainability. If the callee preserves all affected registers, then there is only one copy of the push and pop instructions, those the procedure contains. If the caller saves the values in the registers, the program needs a set of push and pop instructions around every call. Not only does this make your programs longer, it also makes them harder to maintain. Remembering which registers to push and pop on each procedure call is not something easily done.

On the other hand, a subroutine may unnecessarily preserve some registers if it pre-serves all the registers it modifies. In the examples above, the code needn't save ax. Although PrintSpaces changes the al, this won't affect the program's operation. If the caller is preserving the registers, it doesn't have to save registers it doesn't care about:

```
                mov         cx, 10
Loop0:          push        cx
                call        PrintSpaces
                pop         cx
                putcr
                loop        Loop0
                putcr
                call        PrintSpaces
                mov         al, '*'
                mov         cx, 100
Loop1:          putc
                push        ax
                push        cx
                call        PrintSpaces
                pop         cx
                pop         ax
                putc
                putcr
                loop        Loop1
.
.
PrintSpaces     proc        near
                mov         al, ' '
                mov         cx, 40
PSLoop:         putc
```

```
                    loop          PSLoop
                    ret
PrintSpaces         endp
```

This example provides three different cases. The first loop (Loop0) only preserves the cx register. Modifying the al register won't affect the operation of this program. Immediately after the first loop, this code calls PrintSpaces again. However, this code doesn't save ax or cx because it doesn't care if PrintSpaces changes them. Since the final loop (Loop1) uses ax and cx, it saves them both.

One big problem with having the caller preserve registers is that your program may change. You may modify the calling code or the procedure so that they use additional registers. Such changes, of course, may change the set of registers that you must preserve. Worse still, if the modification is in the subroutine itself, you will need to locate every call to the routine and verify that the subroutine does not change any registers the calling code uses.

Preserving registers isn't all there is to preserving the environment. You can also push and pop variables and other values that a subroutine might change. Since the 80x86 allows you to push and pop memory locations, you can easily preserve these values as well.

Parameters

Although there is a large class of procedures that are totally self-contained, most procedures require some input data and return some data to the caller. Parameters are values that you pass to and from a procedure. There are many facets to parameters. Questions concerning parameters include:

- Where is the data coming from?
- How do you pass and return data?
- • What is the amount of data to pass?

There are six major mechanisms for passing data to and from a procedure, they are

- Pass by value,
- Pass by reference,
- Pass by result, and
- Pass by name.

You also have to worry about where you can pass parameters. Common places are

- In registers
- In global memory locations
- On the stack

- In the code stream, or
- In a parameter block referenced via a pointer.

Finally, the amount of data has a direct bearing on where and how to pass it. The following sections take up some these issues.

Pass by Value

A parameter passed by value is just that – the caller passes a value to the procedure. Pass by value parameters are input only parameters. That is, you can pass them to a procedure but the procedure cannot return them. In HLLs, like Pascal, the idea of a pass by value parameter being an input only parameter makes a lot of sense. Given the Pascal procedure call:

```
Call Proc(I);
```

If you pass I by value, the Call Proc does not change the value of I, regardless of what hap-pens to the parameter inside CallProc.

Since you must pass a copy of the data to the procedure, you should only use this method for passing small objects like bytes, words, and double words. Passing arrays and strings by value is very inefficient (since you must create and pass a copy of the structure to the procedure).

Pass by Reference

To pass a parameter by reference, you must pass the address of a variable rather than its value. In other words, you must pass a pointer to the data. The procedure must dereference this pointer to access the data. Passing parameters by reference is useful when you must modify the actual parameter or when you pass large data structures between procedures.

Passing parameters by reference can produce some peculiar results. The following Pascal procedure provides an example of one problem you might encounter:

```
program main(input, output);
var m:integer;
        procedure bletch(var i, j:integer);
        begin
                i := i+2;
                j := j-i;
                writeln(i,’ ‘,j);
        end;
          .
          .
        begin {main}
                m := 5;
                bletch(m, m);
        end.
```

This particular code sequence will print "00" regardless of m's value. This is because the parameters i and j are pointers to the actual data and they both point at the same object. Therefore, the statement j:=j-i; always produces zero since i and j refer to the same variable.

Pass by reference is usually less efficient than pass by value. You must dereference all pass by reference parameters on each access; this is slower than simply using a value. However, when passing a large data structure, pass by reference is faster because you do not have to copy a large data structure before calling the procedure.

Pass by Value-Returned

Pass by value-returned (also known as value-result) combines features from both the pass by value and pass by reference mechanisms. You pass a value-returned parameter by address, just like pass by reference parameters. However, upon entry, the procedure makes a temporary copy of this parameter and uses the copy while the procedure is executing. When the procedure finishes, it copies the temporary copy back to the original parameter.

The Pascal code presented in the previous section would operate properly with pass by value-returned parameters. Of course, when Bletch returns to the calling code, m could only contain one of the two values, but while Bletch is executing, i and j would contain distinct values.

In some instances, pass by value-returned is more efficient than pass by reference, in others it is less efficient. If a procedure only references the parameter a couple of times, copying the parameter's data is expensive. On the other hand, if the procedure uses this parameter often, the procedure amortizes the fixed cost of copying the data over many inexpensive accesses to the local copy.

Pass by Result

Pass by result is almost identical to pass by value-returned. You pass in a pointer to the desired object and the procedure uses a local copy of the variable and then stores the result through the pointer when returning. The only difference between pass by value-returned and pass by result is that when passing parameters by result you do not copy the data upon entering the procedure. Pass by result parameters are for returning values, not passing data to the procedure. Therefore, pass by result is slightly more efficient than pass by value-returned since you save the cost of copying the data into the local variable.

Pass by Name

Pass by name is the parameter passing mechanism used by macros, text equates, and the #define macro facility in the C programming language. This parameter passing mechanism uses textual substitution on the parameters. Consider the following MASM macro:

```
PassByName      macro       Parameter1, Parameter2
                Mov          ax, Parameter1
                add         ax, Parameter2
                endm
```

If you have a macro invocation of the form:

```
                PassByName  bx, I
```

MASM emits the following code, substituting bx for Parameter1andI for Parameter2:

```
                mov         ax, bx
                add         ax, I
```

Some high level languages, such as ALGOL-68 and Panacea, support pass by name parameters. However, implementing pass by name using textual substitution in a com-piled language (like ALGOL-68) is very difficult and inefficient. Basically, you would have to recompile a function every time you call it. So, compiled languages that support pass by name parameters generally use a different technique to pass those parameters. Consider the following Panacea procedure:

```
PassByName: procedure (name item: integer; var index: integer);
begin PassByName;
        foreach index in 0..10 do
                item := 0;
        endfor;
end PassByName;
```

Assume you call this routine with the statement PassByName(A[i], i); where A is an array of integers having (at least) the elements A[0]..A[10]. Were you to substitute the pass by name parameter item you would obtain the following code:

```
begin PassByName;
        foreach index in 0..10 do
```

```
                    A[I] := 0; (* Note that index and I are aliases *)
                endfor;
end PassByName;
```

This code zeros out elements 0..10 of array A.

High level languages like ALGOL-68 and Panacea compile pass by name parameters into functions that return the address of a given parameter. So in one respect, pass by name parameters are similar to pass by reference parameters insofar as you pass the address of an object. The major difference is that with pass by reference you compute the address of an object before calling a subroutine; with pass by name the subroutine itself calls some function to compute the address of the parameter.

So what difference does this make? Well, reconsider the code above. Had you passed A[I] by reference rather than by name, the calling code would compute the address of A[I] just before the call and passed in this address. Inside the PassByName procedure the variable item would have always referred to a single address, not an address that changes along with I. With pass by name parameters, item is really a function that computes the address of the parameter into which the procedure stores the value zero. Such a function might look like the following:

```
ItemThunk       proc        near
                mov         bx, I
                shl         bx, 1
                lea         bx, A[bx]
                ret
ItemThunk       endp
```

The compiled code inside the PassByName procedure might look something like the fol-lowing:

```
; item := 0;
                call        ItemThunk
                mov         word ptr [bx], 0
```

Thunk is the historical term for these functions that compute the address of a pass by name parameter. It is worth noting that most HLLs supporting pass by name parameters do not call thunks directly (like the call above). Generally, the caller passes the address of a thunk and the subroutine calls the thunk indirectly. This allows the same sequence of instructions to call several different thunks (corresponding to different calls to the subroutine).

Passing Parameters in Registers

Having touched on how to pass parameters to a procedure, the next thing to discuss is where to pass parameters. Where you pass parameters depends, to a great extent, on the size and number of those parameters. If you are passing a small number of bytes to a procedure, then the registers are an excellent place to pass parameters. The registers are an ideal place to pass value parameters to a procedure. If you are passing a single parameter to a procedure you should use the following registers for the accompanying data types:

Data Size Pass in this Register

Byte: al

Word: ax

Double Word: dx:axor eax(if 80386 or better)

This is, by no means, a hard and fast rule. If you find it more convenient to pass 16 bit values in the sior bx register, by all means do so. However, most programmers use the registers above to pass parameters.

If you are passing several parameters to a procedure in the 80x86's registers, you should probably use up the registers in the following order:

First Last

ax, dx, si, di, bx, cx

In general, you should avoid using bpregister. If you need more than six words, perhaps you should pass your values elsewhere.

The UCR Standard Library package provides several good examples of procedures that pass parameters by value in the registers. Putc, which outputs an ASCII character code to the video display, expects an ASCII value in the al register. Likewise, puti expects the value of a signed integer in the ax register. As another example, consider the following putsi(put short integer) routine that outputs the value in alas a signed integer:

```
putsi       proc
            push        ax      ;Save AH's value.
            cbw                 ;Sign extend AL -> AX.
            puti                ;Let puti do the real work.
            pop         ax      ;Restore AH.
            ret
putsi endp
```

The other four parameter passing mechanisms (pass by reference, value-returned, result, and name) generally require that you pass a pointer to the desired object (or to a thunk in the case of pass by name). When passing such parameters in registers, you have to consider whether you're passing an offset or a full segmented address. Sixteen bit off-sets can be passed in any of the 80x86's general purpose 16 bit registers. si, di, and bx are the best place to pass an offset since you'll probably need to load it into one of these registers anyway. You can pass 32 bit segmented addresses dx:ax like other double word parameters. However, you can also pass them in ds:bx, ds:si, ds:di, es:bx, es:si, or es:di and be able to use them without copying into a segment register.

The UCR Stdlib routine puts, which prints a string to the video display, is a good example of a subroutine that uses pass by reference. It wants the address of a string in the es:di register pair. It passes the parameter in this fashion, not because it modifies the parameter, but because strings are rather long and passing them some other way would be inefficient. As another example, consider the following strfill(str,c)that copies the character c(passed by value in al) to each character position in str(passed by reference in es:di) up to a zero terminating byte:

```
; strfill-          copies value in al to the string pointed at by es:di
;                   up to a zero terminating byte.
byp                 textequ     <byte ptr>
strfill     proc
                    pushf                   ;Save direction flag.
                    cld                     ;To increment D with STOS.
                    push    di              ;Save, because it's changed.
                    jmp     sfStart
sfLoop:             stosb                   ;es:[di] := al, di := di + 1;
sfStart:    cmp     byp es:[di], 0 ;Done yet?
                    jne     sfLoop
                    pop     di              ;Restore di.
                    popf                    ;Restore direction flag.
                    ret
strfill     endp
```

When passing parameters by value-returned or by result to a subroutine, you could pass in the address in a register. Inside the procedure you would copy the value pointed at by this register to a local variable (value-returned only). Just before the procedure returns to the caller, it could store the final result back to the address in the register.

The following code requires two parameters. The first is a pass by value-returned parameter and the subroutine expects the address of the actual parameter in bx. The sec-ond is a pass by result parameter whose address is in si. This routine increments the pass by value-result parameter and stores the previous result in the pass by result parameter:

```
; CopyAndInc-             BX contains the address of a variable. This routine
;                   copies that variable to the location specified in SI
;                   and then increments the variable BX points at.
;                   Note: AX and CX hold the local copies of these
;                   parameters during execution.
CopyAndInc          proc
                    push   ax            ;Preserve AX across call.
                    push   cx            ;Preserve CX across call.
                    mov    ax, [bx]      ;Get local copy of 1st parameter.
                    mov    cx, ax        ;Store into 2nd parm’s local var.
                    inc    ax            ;Increment 1st parameter.
                    mov    [si], cx      ;Store away pass by result parm.
                    mov    [bx], ax      ;Store away pass by value/ret parm.
                    pop    cx            ;Restore CX’s value.
                    pop    ax            ;Restore AX’s value.
                    ret
CopyAndInc          endp
```

To make the call CopyAndInc(I,J)you would use code like the following:

```
                    lea    bx, I
                    lea    si, J
                    call   CopyAndInc
```

This is, of course, a trivial example whose implementation is very inefficient. Neverthe-less, it shows how to pass value-returned and result parameters in the 80x86’s registers. If you are willing to trade a little space for some speed, there is another way to achieve the same results as pass by value-returned or pass by result when passing parameters in registers. Consider the following implementation of CopyAndInc:

```
CopyAndInc      proc
                mov   cx, ax          ;Make a copy of the 1st parameter,
                inc   ax              ; then increment it by one.
                ret
CopyAndInc      endp
```

To make the CopyAndInc(I,J)call, as before, you would use the following 80x86 code:

```
                mov   ax, I
                call  CopyAndInc
                mov   I, ax
                mov   J, cx
```

Note that this code does not pass any addresses at all; yet it has the same semantics (that is, performs the same operations) as the previous version. Both versions increment I and store the pre-incremented version into J. Clearly the latter version is faster, although your program will be slightly larger if there are many calls to CopyAndInc in your program (six or more).

You can pass a parameter by name or by lazy evaluation in a register by simply load-ing that register with the address of the thunk to call. Consider the Panacea PassByName procedure One implementation of this procedure could be the following:

```
;PassByName-          Expects a pass by reference parameter index
;               passed in si and a pass by name parameter, item,
;               passed in dx (the thunk returns the address in bx).
PassByName      proc
                push       ax                  ;Preserve AX across call
                mov        word ptr [si], 0    ;Index := 0;
ForLoop:        cmp        word ptr [si], 10   ;For loop ends at ten.
                jg         ForDone
                call       dx ;Call thunk item.
                mov        word ptr [bx], 0    ;Store zero into item.
                inc        word ptr [si]       ;Index := Index + 1;
                jmp        ForLoop
```

```
          ForDone:          pop         ax ;Restore AX.
                            ret                             ;All Done!
          PassByName        endp
```

You might call this routine with code that looks like the following:

```
                            lea         si, I
                            lea         dx, Thunk_A
                            call        PassByName
                            .
                            .
Thunk_A                     proc
                            mov          bx, I
                            shl         bx, 1
                            lea         bx, A[bx]
                            ret
Thunk_A                     endp
```

The advantage to this scheme, over the one presented in the earlier section, is that you can call different thunks, not just the ItemThunk routine appearing in the earlier example.

Passing Parameters on the Stack

Most HLLs use the stack to pass parameters because this method is fairly efficient. To pass parameters on the stack, push them immediately before calling the subroutine. The subroutine then reads this data from the stack memory and operates on it appropriately. Consider the following Pascal procedure call:

```
CallProc(i,j,k+4);
```

Most Pascal compilers push their parameters onto the stack in the order that they appear in the parameter list. Therefore, the 80x86 code typically emitted for this subroutine call (assuming you're passing the parameters by value) is

```
push    i
push    j
mov     ax, k
add     ax, 4
push    ax
```

```
        call    CallProc
```

You could gain access to the parameters passed on the stack by removing the data from the stack (Assuming a near procedure call):

```
CallProc        proc    near
                pop     RtnAdrs
                pop     kParm
                pop     jParm
                pop     iParm
                push    RtnAdrs
                .
                .
                ret
CallProc        endp
```

There is, however, a better way. The 80x86's architecture allows you to use the bp(base pointer) register to access parameters passed on the stack. This is one of the reasons the disp[bp], [bp][di], [bp][si], disp[bp][si],and disp[bp][di] addressing modes use the stack segment rather than the data segment. The following code segment gives the standard procedure entry and exit code:

```
StdProc         proc    near
                push    bp
                mov     bp, sp
                .
                .
                pop      bp
                ret     ParmSize
StdProc         endp
```

ParmSize is the number of bytes of parameters pushed onto the stack before calling the procedure. In the CallProc procedure there were six bytes of parameters pushed onto the stack so ParmSize would be six.

Take a look at the stack immediately after the execution of mov bp, sp in StdProc. Assuming you've pushed three parameter words onto the stack.

Summary

- In an assembly language program, all you need is a call and ret instruction to implement procedures and functions.
- This chapter begins with a review of what a procedure is, how to implement procedures with MASM, and the difference between near and far procedures on the 80x86.
- Functions are a very important construct in high level languages like Pascal.
- How-ever, there really isn't a difference between a function and a procedure in an assembly language program. Logically, a function returns a result and a procedure does not; but you declare and call procedures and functions identically in an assembly language program.
- Procedures and functions often produce side effects.
- That is, they modify the values of registers and non-local variables.
- Often, these side effects are undesirable. For example, a procedure may modify a register that the caller needs preserved.
- There are two basic mechanisms for preserving such values: callee preservation and caller preservation.
- One of the major benefits to using a procedural language like Pascal or C++ is that you can easily pass parameters to and from procedures and functions.
- Although it is a little more work, you can pass parameters to your assembly language functions and procedures as well.
- This chapter discusses how and where to pass parameters.
- It also discusses how to access the parameters inside a procedure or function.

CHAPTER-7 INSTRUCTION TYPES

CHAPTER OBJECTIVES

Upon completion of this chapter, you will be able to:

1. Use arithmetic and logic instructions to accomplish simple binary, BCD, and ASCII arithmetic.
2. Use AND, OR, and Exclusive-OR to accomplish binary bit manipulation.
3. Use the shift and rotate instructions.
4. Explain the operation of the 80386 through the Core2 exchange and add, compare and exchange, double-precision shift, bit test, and bit scan instructions.
5. Check the contents of a table for a match with the string instructions.

Stack Instructions

PUSH/POP

The PUSH and POP instructions are important instructions that *store* and *retrieve* data from the LIFO (last-in, first-out) stack memory. The microprocessor has six forms of the PUSH and POP instructions: register, memory, immediate, segment register, flags, and all registers. The PUSH and POP immediate and the PUSHA and POPA (all registers) forms are not available in the earlier 8086/8088 microprocessors, but are available to the 80286 through the Core2.

Register addressing allows the contents of any 16-bit register to be transferred to or from the stack. In the 80386 and above, the 32-bit extended registers and flags (EFLAGS) can also be pushed or popped from the stack. Memory-addressing PUSH and POP instructions store the contents of a 16-bit memory location (or 32 bits in the 80386 and above) on the stack or stack data into a memory location. Immediate addressing allows immediate data to be pushed onto the stack, but not popped off the stack. Segment register addressing allows the contents of any segment register to be pushed onto the stack or removed from the stack (ES may be pushed, but data from the stack may never be popped into ES). The flags may be pushed or popped from that stack, and the contents of all the registers may be pushed or popped.

PUSH

The 8086–80286 PUSH instruction always transfers 2 bytes of data to the stack; the 80386 and above transfer 2 or 4 bytes, depending on the register or size of the memory location. The source of the data may be any internal 16- or 32-bit register, immediate data, any segment register, or any 2 bytes of memory data. There is also a PUSHA instruction that copies the contents of the internal register set, except the segment registers, to the stack. The PUSHA (**push all**) instruction copies the registers to the stack in the following order: AX, CX, DX, BX, SP, BP, SI, and DI. The value for SP that is pushed onto the stack is whatever it was before the PUSHA instruction executed. The PUSHF (push flags) instruction copies the contents of the flag register to the stack. The PUSHAD and POPAD instructions push and pop the contents of the 32-bit register set found in the 80386 through the Pentium 4. The PUSHA and POPA instructions do not function in the 64-bit mode of operation for the Pentium 4.

Whenever data are pushed onto the stack, the first (most-significant) data byte moves to the stack segment memory location addressed by SP-1 . The second (least-significant) data byte moves into the stack segment memory location addressed by SP-2. After the data are stored by a PUSH, the contents of the SP register decrement by 2. The same is true for a doubleword push, except that 4 bytes are moved to the stack memory (most-significant byte first), after which the stack pointer decrements by 4. The figure shows the operation of the PUSH AX instruction. This instruction copies the contents of AX onto the stack where address SS:[SP-1] = AH, SS:[SP-2] = AL, and afterwards SP=SP-2. In 64-bit mode,
8 bytes of the stack are used to store the number pushed onto the stack.

The PUSHA instruction pushes all the internal 16-bit registers onto the stack. The PUSHA instruction is very useful when the entire register set (microprocessor environment) of the 80286 and above must be saved during a task. The PUSHAD instruction places the 32-bit register set on the stack in the 80386 through the Core2. PUSHAD requires 32 bytes of stack storage space.

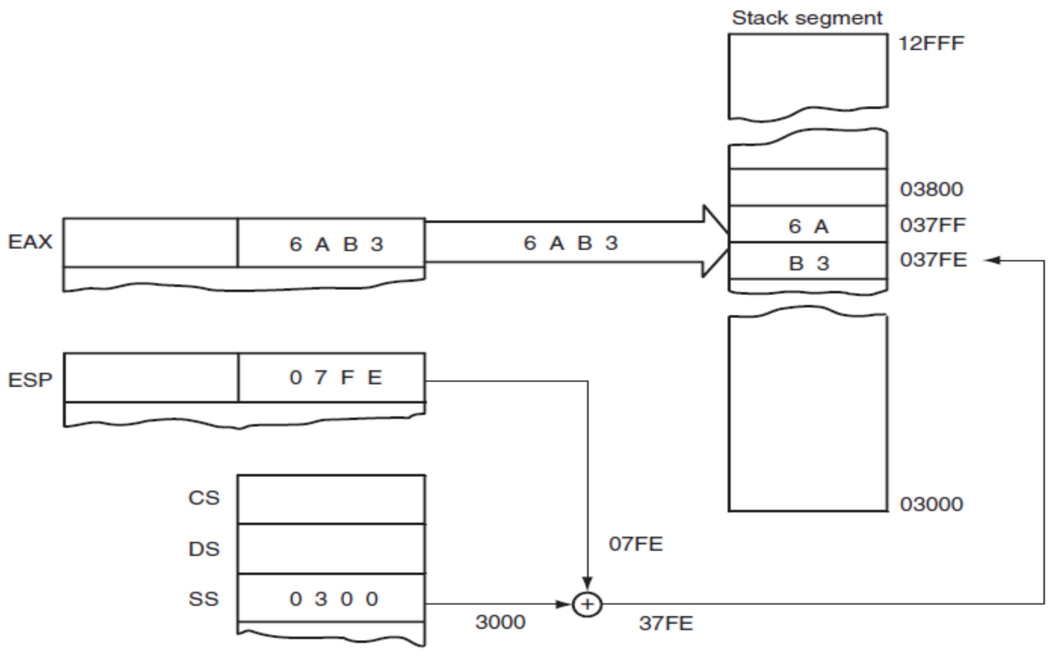


Figure: The effect of the PUSH AX instruction on ESP and stack memory locations 37FFH and 37FEH.

This instruction is shown at the point after execution.

The PUSH immediate data instruction has two different opcodes, but in both cases, a 16- bit immediate number moves onto the stack; if PUSHD is used, a 32-bit immediate datum is pushed. If the values of the immediate data are 00H–FFH, the opcode is a 6AH; if the data are 0100H–FFFFH, the opcode is 68H. The PUSH 8 instruction, which pushes 0008H onto the

Assembly Instructions	Example	Note
PUSH reg16	PUSH BX	16-bit register
PUSH reg32	PUSH EDX	32-bit register
PUSH mem16	PUSH WORD PTR[BX]	16-bit pointer
PUSH mem32	PUSH DWORD PTR[EBX]	32-bit pointer
PUSH mem64	PUSH QWORD PTR[RBX]	64-bit pointer (64-bit mode)
PUSH seg	PUSH DS	Segment register
PUSH imm8	PUSH ‘R’	8-bit immediate
PUSH imm16	PUSH 1000H	16-bit immediate
PUSHD imm32	PUSHD 20	32-bit immediate
PUSHA	PUSHA	Save all 16-bit registers
PUSHAD	PUSHAD	Save all 32-bit registers
PUSHF	PUSHF	Save flags
PUSHFD	PUSHFD	Save EFLAGS

TABLE of PUSH instruction.

stack, assembles as 6A08H. The PUSH 1000H instruction assembles as 680010H. Another example of PUSH immediate is the PUSH ‘A’ instruction, which pushes a 0041H onto the stack. Here, the 41H is the ASCII code for the letter A. Table lists the forms of the PUSH instruction that include PUSHA and PUSHF. Notice how the instruction set is used to specify different data sizes with the assembler.

POP

The POP instruction performs the inverse operation of a PUSH instruction. The POP instruction removes data from the stack and places it into the target 16-bit register, segment register, or a 16- bit memory location. In the 80386 and above, a POP can also remove 32-bit data from the stack and use a 32-bit address. The POP instruction is not available as an immediate POP. The POPF (pop flags) instruction removes a 16-bit number from the stack and places it into the flag register; the POPFD removes a 32-bit number from the stack and places it into the extended flag register. The POPA (pop all) instruction removes 16 bytes of data from the stack and places them into the following registers, in the order shown: DI, SI, BP, SP, BX, DX, CX, and AX. This is the reverse order from the way they were placed on the stack by the PUSHA instruction, causing the same data to return to the same registers. In the 80386 and above, a POPAD instruction reloads the 32-bit registers from the stack.

Suppose that a POP BX instruction executes. The first byte of data removed from the stack (the memory location addressed by SP in the stack segment) moves into register BL. The second byte is removed from stack segment memory location and is placed into register BH. After both bytes are removed from the stack, the SP register is incremented by 2. The figure shows how the POP BX instruction removes data from the stack and places them into register BX.

The opcodes used for the POP instruction and all of its variations appear in Table. Note that a POP CS instruction is not a valid instruction in the instruction set. If a POP CS instruction executes, only a portion of the address (CS) of the next instruction changes. This makes the POP CS instruction unpredictable and therefore not allowed.

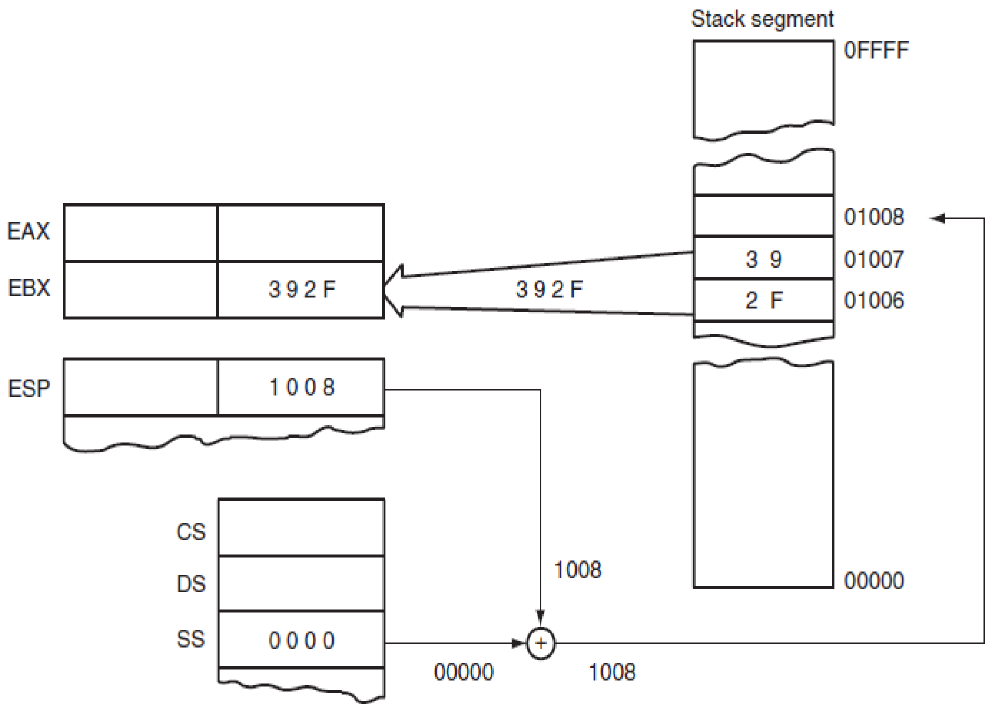


Figure of the POP BX instruction, showing how data are removed from the stack. This instruction is shown after execution.

Assembly Instructions	Example	Note
POP reg16	POP CX	16-bit register
POP reg32	POP EBP	32-bit register
POP mem16	POP WORD PTR[BX+1]	16-bit pointer
POP mem32	POP DATA3	32-bit memory address
POP mem64	POP FROG	64-bit memory address -64-bit mod

POP seg	POP FS	Segment register
POPA	POPA	Pops all 16-bit registers
POPAD	POPAD	Pops all 32-bit registers
POPF	POPF	Pops flags
POPFD	POPFD	Pops EFLAGS

TABLE of the POP instructions.

Integral ALU instructions

Arithmetic Instructions

- The ADD/SUB Instruction

Addition (ADD) appears in many forms in the microprocessor. This section details the use of the ADD instruction for 8-, 16-, and 32-bit binary addition. A second form of addition, called **add-with-carry**, is introduced with the ADC instruction.

Table 5–1 illustrates the addressing modes available to the ADD instruction. However, because there are more than 32,000 variations of the ADD instruction in the instruction set, it is impossible to list them all in this table. The only types of addition *not* allowed are memory-to-memory and segment register. The segment registers can only be moved, pushed, or popped. Note that, as with all other instructions, the 32-bit registers are available only with the 80386 through the Core2. In the 64-bit mode of the Pentium 4 and Core2, the 64-bit registers are also used for addition.

Assembly Instructions	Operations
ADD AL,BL	AL = AL + BL
ADD CX,DI	CX = CX + DI
ADD EBP,EAX	EBP = EBP + EAX
ADD CL,44H	CL = CL + 44H
ADD BX,245FH	BX = BX + 245FH
ADD EDX,12345H	EDX = EDX + 12345H
ADD [BX],AL	AL adds to the byte contents of the data segment memory location addressed by BX with the sum stored in the same memory location
ADD CL,[BP]	The byte contents of the stack segment memory location addressed by BP add to CL with the sum stored in CL
ADD AL,[EBX]	The byte contents of the data segment memory location addressed by EBX add to AL with the sum stored in AL

ADD BX,[SI+2]	The word contents of the data segment memory location addressed by SI + 2 add to BX with the sum stored in BX
ADD CL,TEMP	The byte contents of data segment memory location TEMP add to CL with the sum stored in CL
ADD BX,TEMP[DI]	The word contents of the data segment memory location addressed by TEMP + DI add to BX with the sum stored in BX
ADD [BX+D],DL	DL adds to the byte contents of the data segment memory location addressed by BX + DI with the sum stored in the same memory location
ADD BYTE PTR [DI],3	A 3 adds to the byte contents of the data segment memory location addressed by DI with the sum stored in the same location
ADD BX,[EAX+2*ECX]	The word contents of the data segment memory location addressed by EAX plus 2 times ECX add to BX with the sum stored in BX
ADD RAX,RBX	RBX adds to RAX with the sum stored in RAX (64-bit mode)
ADD EDX,[RAX+RCX]	The double word in EDX is added to the double word addressed by the sum of RAX and RCX and the sum is stored in EDX (64-bit mode)

Register Addition. Example+1 shows a simple sequence of instructions that uses register addition to add the contents of several registers. In this example, the contents of AX, BX, CX, and DX are added to form a 16-bit result stored in the AX register.

EXAMPLE +1

```
0000 03 C3 ADD AX,BX
0002 03 C1 ADD AX,CX
0004 03 C2 ADD AX,DX
```

Whenever arithmetic and logic instructions execute, the contents of the flag register change. Note that the contents of the interrupt, trap, and other flags do not change due to arithmetic and logic instructions. Only the flags located in the rightmost 8 bits of the flag register and the overflow flag change. These rightmost flags denote the result of the arithmetic or a logic operation. Any ADD instruction modifies the contents of the sign, zero, carry, auxiliary carry, parity, and overflow flags. The flag bits never change for most of the data transfer instructions.

Immediate Addition. Immediate addition is employed whenever constant or known data are added. An 8-bit immediate addition appears in Example 5-2. In this example, DL is first loaded with 12H by using an immediate move instruction. Next, 33H is added to the 12H in DL by an immediate addition instruction. After the addition, the sum (45H) moves into register DL and the flags change, as follows:

EXAMPLE +2

```
0000 B2 12 MOV DL,12H
0002 80 C2 33 ADD DL,33H
```

Memory-to-Register Addition. Suppose that an application requires memory data to be added to the AL register. Example +3 shows an example that adds two consecutive bytes of data, stored at the data segment offset locations NUMB and NUMB+1, to the AL register.

EXAMPLE +3

```
0000 BF 0000 R   MOV DI, OFFSET NUMB        ;address NUMB
0003 B0 00            MOV AL,0              ;clear sum
0005 02 05            ADD AL,[DI]                 ;add NUMB
0007 02 45 01         ADD AL,[DI+1]               ;add NUMB+1
```

The first instruction loads the destination index register (DI) with offset address NUMB. The DI register, used in this example, addresses data in the data segment beginning at memory location NUMB. After clearing the sum to zero, the ADD AL,[DI] instruction adds the contents of memory location NUMB to AL. Finally, the ADD AL,[DI+1] instruction adds the contents of memory location NUMB plus 1 byte to the AL register. After both ADD instructions execute, the result appears in the AL register as the sum of the contents of NUMB plus the contents of NUMB+1 .

Array Addition. Memory arrays are sequential lists of data. Suppose that an array of data (ARRAY) contains 10 bytes, numbered from element 0 through element 9. Example +4 shows how to add the contents of array elements 3, 5, and 7 together.

This example first clears AL to 0, so it can be used to accumulate the sum. Next, register SI is loaded with a 3 to initially address array element 3. The ADD AL,ARRAY[SI] instruction adds the contents of array element 3 to the sum in AL. The instructions that follow add array elements 5 and 7 to the sum in AL, using a 3 in SI plus a displacement of 2 to address element 5, and a displacement of 4 to address element 7.

EXAMPLE +4

```
0000 B0 00           MOV AL,0        ;clear sum
0002 BE 0003         MOV SI,3        ;address element 3
0005 02 84 0000 R    ADD AL,ARRAY[SI]      ;add element 3
0009 02 84 0002 R    ADD AL,ARRAY[SI+2]    ;add element 5
000D 02 84 0004 R    ADD AL,ARRAY[SI+4]    ;add element 7
```

Suppose that an array of data contains 16-bit numbers used to form a 16-bit sum in register AX. Example +5 shows a sequence of instructions written for the 80386 and above, showing the scaled-index form of addressing to add elements 3, 5, and 7 of an area of memory called ARRAY. In this example, EBX is loaded with the address ARRAY, and ECX holds the array element number. Note how the scaling factor is used to multiply the contents of the ECX register by 2 to address words of data. (Recall that words are 2 bytes long.)

EXAMPLE +5

```
0000 66|BB 00000000 R      MOV EBX,OFFSET ARRAY      ;address ARRAY
0006 66|B9 00000003        MOV ECX,3            ;address element 3
000C 67&8B 04 4B           MOV AX,[EBX+2*ECX]        ;get element 3
0010 66|B9 00000005        MOV ECX,5            ;address element 5
0016 67&03 04 4B           ADD AX,[EBX+2*ECX]        ;add element 5
001A 66|B0 00000007        MOV ECX,7            ;address element 7
0020 67&03 04 4B           ADD AX,[EBX+2*ECX]        ;add element 7
```

Addition-with-Carry. An addition-with-carry instruction (ADC) adds the bit in the carry flag (C) to the operand data. This instruction mainly appears in software that adds numbers that are wider than 16 bits in the 8086–80286 or wider than 32 bits in the 80386–Core2. Table +2 lists several add-with-carry instructions, with comments that explain their operation. Like the ADD instruction, ADC affects the flags after the addition.

TABLE +2 Example add-with-carry instructions.

Assembly Instructions	Operations
ADC AL,AH	AL = AL + AH + carry
ADC CX,BX	CX = CX + BX + carry
ADC EBX,EDX	EBX = EBX + EDX + carry
ADC RBX,0	RBX = RBX + 0 + carry (64-bit mode)

ADC DH,[BX]	The byte contents of the data segment memory location addressed by BX add to DH with the sum stored in DH
ADC BX,[BP+2]	The word contents of the stack segment memory location addressed by BP plus 2 add to BX with the sum stored in BX
ADC ECX,[EBX]	The double word contents of the data segment memory Location addressed by EBX add to ECX with the sum stored in ECX

The INC Instruction

Increment addition (INC) adds 1 to a register or a memory location. The INC instruction adds 1 to any register or memory location, except a segment register. Table +3 illustrates some of the possible forms of the increment instructions available to the 8086–Core2 processors. As with other instructions presented thus far, it is impossible to show all variations of the INC instruction because of the large number available. With indirect memory increments, the size of the data must be described by using the BYTE PTR, WORD PTR, DWORD PTR, or QWORD PTR directives. The reason is that the assembler program cannot determine if, for example, the INC [DI] instruction is a byte-, word-, or double word-sized increment. The INC BYTE PTR [DI] instruction clearly indicates byte sized memory data; the INC WORD PTR [DI] instruction unquestionably indicates a word-sized memory data; and the INC DWORD PTR [DI] instruction indicates double word-sized data. In 64-bit mode operation of the Pentium 4 and Core2, the INC QWORD PTR [RSI] instruction indicates quad word-sized data.

TABLE +3 Example increment instructions.

Assembly Instructions	Operations
INC BL	BL = BL + 1
INC SP	SP = SP + 1
INC EAX	EAX = EAX + 1
INC BYTE PTR [BX]	Adds 1 to the byte contents of the data segment memory location addressed by BX
INC WORD PTR[SI]	Adds 1 to the word contents of the data segment memory location addressed by SI
INC DWORD PTR [ECX]	Adds 1 to the double word contents of the data segment memory location addressed by ECX
INC DATA1	Adds 1 to the contents of data segment memory location DATA1
INC RCX	Adds 1 to RCX (64-bit mode)

Example +6 shows how to modify the Example +3 to use the increment instruction for addressing NUMB and NUMB+1. Here, an INC DI instruction changes the contents of register DI from offset address NUMB+1 to offset address. Both program sequences shown in
Examples +3 and +6 add the contents of NUMB and NUMB+1. The difference between them is the way that the address is formed through the contents of the DI register using the increment instruction.

EXAMPLE +6

```
0000 BF 0000 R MOV DI, OFFSET NUMB ;address NUMB
0003 B0 00 MOV AL,0 ;clear sum
0005 02 05 ADD AL, [DI] ;add NUMB
0007 47 INC DI  ;increment DI
0008 02 05 ADD AL,[DI] ;add NUMB+1
```

Increment instructions affect the flag bits, as do most other arithmetic and logic operations. The difference is that increment instructions do not affect the carry flag bit. Carry doesn't change because we often use increments in programs that depend upon the contents of the carry flag. Note that increment is used to point to the next memory element in a byte-sized array of data only. If word-sized data are addressed, it is better to use an ADD DI,2 instruction to modify the DI pointer in place of two INC DI instructions. For double word arrays, use the ADD DI,4 instruction to modify the DI pointer. In some cases, the carry flag must be preserved, which may mean that two or four INC instructions might appear in a program to modify a pointer.

- **The SUB Instruction**

Many forms of subtraction (SUB) appear in the instruction set. These forms use any addressing mode with 8-, 16-, or 32-bit data. A special form of subtraction (decrement, or DEC) subtracts 1 from any register or memory location. Section 5–3 shows how BCD and ASCII data subtract. As with addition, numbers that are wider than 16 bits or 32 bits must occasionally be subtracted. The **subtract-with-borrow instruction** (SBB) performs this type of subtraction. In the 80486 through the Core2 processors, the instruction set also includes a compare and exchange instruction. In the 64-bit mode for the Pentium 4 and Core2, a 64-bit subtraction is also available.

Table -4 lists some of the many addressing modes allowed with the subtract instruction (SUB). There are well over 1000 possible subtraction instructions, far too many to list here. About the only types of subtraction not allowed are memory-to-memory and segment register subtractions. Like other arithmetic instructions, the subtract instruction affects the flag bits.

Register Subtraction. Example -1 shows a sequence of instructions that perform register subtraction. This example subtracts the 16-bit contents of registers CX and DX from the contents of register BX. After each subtraction, the microprocessor modifies the contents of the flag register. The flags change for most arithmetic and logic operations.

EXAMPLE -1

```
0000 2B D9            SUB BX,CX
0002 DA              SUB BX,DX
```

Immediate Subtraction. As with addition, the microprocessor also allows immediate operands for the subtraction of constant data. Example -2 presents a short sequence of instructions that subtract 44H from 22H. Here, we first load the 22H into CH using an immediate move instruction.

TABLE +4 Example subtraction instructions.

Assembly Instructions	Operations
SUB CL, BL	CL = CL – BL
SUB AX, SP	AX = AX – SP
SUB ECX, EBP	ECX = ECX – EBP
SUB RDX, R8	RDX = RDX – R8 (64-bit mode)
SUB DH, 6FH	DH = DH – 6FH
SUB AX, 0CCCCH	AX = AX – 0CCCCH
SUB ESI, 2000300H	ESI = ESI – 2000300H
SUB [DI], CH	Subtracts CH from the byte contents of the data Segment memory addressed by DI and stores the difference in the same memory location
SUB CH, [BP]	subtracts the byte contents of the stack segment memory location addressed by BP from CH and stores the difference in CH
SUB AH, TEMP	subtracts the byte contents of memory location TEMP from AH and stores the difference in AH
SUB DI, TEMP [ESI]	Subtracts the word contents of the data segment memory location addressed by TEMP plus ESI from DI an stores the difference in DI
SUB ECX, DATA1	Subtracts the double word contents of memory location DATA1 from ECX and stores the difference in ECX
SUB RCX, 16	RCX = RCX – 18 (64-bit mode)

Next, the SUB instruction, using immediate data 44H, subtracts 44H from the 22H. After the subtraction, the difference (0DEH) moves into the CH register. The flags change as follows for this subtraction:

Z = 0 (result not zero)

C = 1 (borrow)

A = 1 (half-borrow)

S = 1 (result negative)

P = 1 (even parity)

O = 0 (no overflow)

EXAMPLE -2

```
0000 B5 22            MOV CH, 22H
0002 80 ED 44         SUB CH, 44H
```

Both carry flags (C and A) hold borrows after a subtraction instead of carries, as after an addition. Notice in this example that there is no overflow. This example subtracted 44H (+68) from 22H (+34), resulting in a 0DEH (-34). Because the correct 8-bit signed result is -34, there is no overflow in this example. An 8-bit overflow occurs only if the signed result is greater than +127 or less than -128.

- **The DEC Instruction**

Decrement subtraction (DEC) subtracts 1 from a register or the contents of a memory location. Table +5 lists some decrement instructions that illustrate register and memory decrements.

The decrement indirect memory data instructions require BYTE PTR, WORD PTR, DWORD PTR, or QWORD PTR because the assembler cannot distinguish a byte from a word or double word when an index register addresses memory. For example, DEC [SI] is vague because the assembler cannot determine whether the location addressed by SI is a byte, word, or double word. Using DEC BYTE PTR[SI], DEC WORD PTR[DI], or DEC DWORD PTR[SI] reveals

TABLE +5 Example decrement instructions.

Assembly Instructions	Operations
DEC BH	BH = BH – 1
DEC CX	CX = CX – 1
DEC EDX	EDX = EDX – 1
DEC R14	R14 = R14 – 1 (64-bit mode)
DEC BYTE PTR [DI]	Subtracts 1 from the byte contents of the data segment memory location addressed by DI
DEC WORD PTR [BP]	Subtracts 1 from the word contents of the stack

	segment memory location addressed by BP
DEC DWORD PTR [EBX]	Subtracts 1 from the double word contents of the data Segment memory location addressed by EBX
DEC QWORD PTR[RSI]	Subtracts 1 from the quad word contents of the memory location addressed by RSI (64-bit mode)
DEC NUMB	Subtracts 1 from the contents of data segment Memory location NUMB

The size of the data to the assembler. In the 64-bit mode, a DEC QWORD PTR [RSI] decrement the 64-bit number stored at the address pointed to by the RSI register.

Subtraction-with-Borrow. A subtraction-with-borrow (SBB) instruction functions as a regular subtraction, except that the carry flag (C), which holds the borrow, also subtracts from the difference. The most common use for this instruction is for subtractions that are wider than 16 bits in the 8086–80286 microprocessors or wider than 32 bits in the 80386–Core2. Wide subtractions require that borrows propagate through the subtraction, just as wide additions propagate the carry.

Table +6 lists several SBB instructions with comments that define their operations. Like the SUB instruction, SBB affects the flags. Notice that the immediate subtract from memory instruction in this table requires a BYTE PTR, WORD PTR, DWORD PTR, or QWORD PTR directive.

When the 32-bit number held in BX and AX is subtracted from the 32-bit number held in SI and DI, the carry flag propagates the borrow between the two 16-bit subtractions. The carry flag holds the borrow for subtraction. Example -3 shows how this subtraction is performed by a program. With wide subtraction, the least significant 16- or 32-bit data are subtracted with the SUB instruction. All subsequent and more significant data are subtracted by using the SBB instruction. The example uses the SUB instruction to subtract DI from AX, and then uses SBB to subtract-with-borrow SI from BX.

EXAMPLE -4

```
0000 2B C7          SUB AX, DI
 0002 1B DE         SBB BX, SI
```

TABLE +6 Example subtraction-with-borrow instructions.

Assembly Instructions	Operations
SBB AH,AL	AH = AH – AL – carry
SBB AX,BX	AX = AX – BX – carry

SBB EAX,ECX	EAX = EAX – ECX – carry
SBB CL, 2	CL = CL – 2 – carry
SBB RBP, 8	RBP = RBP– 2 – carry (64-bit mode)
SBB BYTE PTR[DI], 3	Both 3 and carry subtract from the data segment memory location addressed by DI
SBB [DI], AL	Both AL and carry subtract from the data segment memory Location addressed by DI
SBB DI, [BP+2]	Both carry and the word contents of the stack segment Memory location addressed by BP plus 2 subtract from DI
SBB AL, [EBX+ECX]	Both carry and the byte contents of the data segment Memory location addressed by EBX plus ECX subtract from AL

MULTIPLICATION AND DIVISION

Only modern microprocessors contain multiplication and division instructions. Earlier 8-bit microprocessors could not multiply or divide without the use of a program that multiplied or divided by using a series of shifts and additions or subtractions. Because microprocessor manufacturers were aware of this inadequacy, they incorporated multiplication and division instructions into the instruction sets of the newer microprocessors. The Pentium–Core2 processors contain special circuitry that performs a multiplication in as little as one clocking period, whereas it took over 40 clocking periods to perform the same multiplication in earlier Intel microprocessors.

Multiplication

Multiplication is performed on bytes, words, or double words, and can be signed integer (IMUL) or unsigned integer (MUL). Note that only the 80386 through the Core2 processors multiply 32-bit double words. The product after a multiplication is always a double-width product. If two 8-bit numbers are multiplied, they generate a 16-bit product; if two 16-bit numbers are multiplied, they generate a 32-bit product; and if two 32-bit numbers are multiplied, a 64-bit product is generated. In the 64-bit mode of the Pentium 4, two 64-bit numbers are multiplied to generate a 128-bit product.

Some flag bits (overflow and carry) change when the multiply instruction executes and produce predictable outcomes. The other flags also change, but their results are unpredictable and therefore are unused. In an 8-bit multiplication, if the most significant 8 bits of the result are zero, both C and O flag bits equal zero. These flag bits show that the result is 8 bits wide (C = 0) or 16 bits wide (C = 1). In a 16-

bit multiplication, if the most significant 16-bits part of the product is 0, both C and O clear to zero. In a 32-bit multiplication, both C and O indicate that the most significant 32 bits of the product are zero.

TABLE +7 Example 8-bit multiplication instructions.

Assembly Instructions	Operations
MUL CL	AL is multiplied by CL; the unsigned product is in AX
IMUL DH	AL is multiplied by DH; the signed product is in AX
IMUL BYTE PTR[BX]	AL is multiplied by the byte contents of the data segment Memory location addressed by BX; the signed product is in AX
MUL TEMP	AL is multiplied by the byte contents of data segment Memory location TEMP; the unsigned product is in AX

8-Bit Multiplication. With 8-bit multiplication, the multiplicand is always in the AL register, whether signed or unsigned. The multiplier can be any 8-bit register or any memory location. Immediate multiplication is not allowed unless the special signed immediate multiplication instruction, discussed later in this section, appears in a program. The multiplication instruction contains one operand because it always multiplies the operand times the contents of register AL. An example is the MUL BL instruction, which multiplies the unsigned contents of AL by the unsigned contents of BL. After the multiplication, the unsigned product is placed in AX—a double-width product. Table 5–8 illustrates some 8-bit multiplication instructions.

Suppose that BL and CL each contain two 8-bit unsigned numbers, and these numbers must be multiplied to form a 16-bit product stored in DX. This procedure cannot be accomplished by a single instruction because we can only multiply a number times the AL register for an 8-bit multiplication. Example *1 shows a short program that generates DX=BL * CL . This example loads register BL and CL with example data 5 and 10. The product, a 50, moves into DX from AX after the multiplication by using the MOV DX, AX instruction.

EXAMPLE *1

```
0000 B3 05                    MOV BL, 5   ;load data
0002 B1 0A                    MOV CL, 10
0004 8A C1                    MOV AL, CL   ;position data
0006 F6 E3                    MUL BL       ;multiply
0008 8B D0                    MOV DX, AX   ;position product
```

For signed multiplication, the product is in binary form, if positive, and in two's complement form, if negative. These are the same forms used to store all positive and negative signed numbers used by the microprocessor. If the program of Example *1 multiplies two signed numbers, only the MUL instruction is changed to IMUL.

Division

As with multiplication, division occurs on 8- or 16-bit numbers in the 8086–80286 microprocessors, and on 32-bit numbers in the 80386 and above microprocessor. These numbers are signed (IDIV) or unsigned (DIV) integers. The dividend is always a double-width dividend that is divided by the operand. This means that an 8-bit division divides a 16-bit number by an 8-bit number; a 16-bit division divides a 32-bit number by a 16-bit number; and a 32-bit division divides a 64-bit number by a 32-bit number. There is no immediate division instruction available to any microprocessor. In the 64-bit mode of the Pentium 4 and Core2, a 64-bit division divides a 128-bit number by a 64-bit number.

None of the flag bits change predictably for a division. A division can result in two different types of errors; one is an attempt to divide by zero and the other is a divide overflow. A divide overflow occurs when a small number divides into a large number. For example, suppose that AX=3000 and that it is divided by 2. Because the quotient for an 8-bit division appears in AL, the result of 1500 causes a divide overflow because the 1500 does not fit into AL. In either case, the microprocessor generates an interrupt if a divide error occurs. In most systems, a divide error interrupt displays an error message on the video screen.

8-Bit Division. An 8-bit division uses the AX register to store the dividend that is divided by the contents of any 8-bit register or memory location. The quotient moves into AL after the division with AH containing a whole number remainder. For a signed division, the quotient is positive or negative; the remainder always assumes the sign of the dividend and is always an integer. For example, if AX= 0010H(+16) and BL= 0FDH(-3) and the IDIV BL instruction executes, AX= 01FBH. This represents a quotient -5(AL) of with a remainder of 1 (AH). If, on the other hand, a -16 is divided by +3, the result will be a quotient of -5(AL) with a remainder of -1(AH) . Table +7 lists some of the 8-bit division instructions.

With 8-bit division, the numbers are usually 8 bits wide. This means that one of them, the dividend, must be converted to a 16-bit wide number in AX. This is accomplished differently for signed and unsigned numbers. For the unsigned number, the most significant 8 bits must be cleared to zero (zero-extended). The MOVZX instruction described in Chapter 4 can be used to zero-extend a number in the 80386 through the Core2 processors. For signed numbers, the least significant 8 bits are sign-extended into the most significant 8 bits. In the microprocessor, a special instruction sign-extends AL into AH, or converts an 8-bit signed number in AL into a 16-bit signed number in AX. The CBW (**convert byte to word**) instruction performs this conversion. In the 80386 through the Core2, a MOVSX instruction sign-extends a number.

TABLE +8 Example 8-bit division instructions.

Assembly Instructions	Operations
DIV CL	AX is divided by CL; the unsigned quotient is in AL and The unsigned remainder is in AH
IDIV BL	AX is divided by BL; the signed quotient is in AL and the Signed remainder is in AH
DIV BYTE PTR [BP]	AX is divided by the byte contents of the stack segment Memory location addressed by BP; the unsigned quotient is in AL and the unsigned remainder is in AH

EXAMPLE /1

```
0000 A0 0000 R        MOV AL, NUMB          ;get NUMB
0003 B4 00            MOV AH, 0            ;zero-extend
0005 F6 36 0002 R     DIV NUMB1           ;divide by NUMB1
0009 A2 0003 R        MOV ANSQ, AL          ;save quotient
000C 88 26 0004 R     MOV ANSR, AH          ;save remainder
```

Example /1 illustrate a short program that divides the unsigned byte contents of memory location NUMB by the unsigned contents of memory location NUMB1. Here, the quotient is stored in location ANSQ and the remainder is stored in location ANSR. Notice how the contents of location NUMB are retrieved from memory and then zero-extended to form a 16-bit unsigned number for the dividend.

Logical Instruction

The basic logic instructions include AND, OR, Exclusive-OR, and NOT. Another logic instruction is TEST, which is explained in this section of the text because the operation of the TEST instruction is a special form of the AND instruction. Also explained is the NEG instruction, which is similar to the NOT instruction.

Logic operations provide binary bit control in low-level software. The logic instructions allow bits to be set, cleared, or complemented. Low-level software appears in machine language or assembly language form and often controls the I/O devices in a system. All logic instructions affect the flag bits. Logic operations always clear the carry and overflow flags, while the other flags change to reflect the condition of the result.

When binary data are manipulated in a register or a memory location, the rightmost bit position is always numbered bit 0. Bit position numbers increase from bit 0 toward the left, to bit 7 for a byte, and to bit 15 for a word. A double word (32 bits) uses bit position 31 as its leftmost bit and a quad word (64-bits) uses bit position 63 as it leftmost bit.

AND

The AND operation performs logical multiplication, as illustrated by the truth table in Table +8. Here, two bits, A and B, are ANDed to produce the result X. As indicated by the truth table, X is a logic 1 only when both A and B are logic 1s. For all other input combinations of A and B, X is a logic 0. It is important to remember that 0 AND anything is always 0, and 1 AND 1 is always 1.

A	B	T
0	0	0
0	1	0
1	0	0
1	1	1

The AND instruction uses any addressing mode except memory-to-memory and segment register addressing. Table +8 lists some AND instructions and comments about their operations. An ASCII-coded number can be converted to BCD by using the AND instruction to mask off the leftmost four binary bit positions. This converts the ASCII 30H to 39H to 0–9. Example &1 shows a short program that converts the ASCII contents of BX into BCD. The AND instruction in this example converts two digits from ASCII to BCD simultaneously.

EXAMPLE &1

```
0000 BB 3135          MOV BX, 3135H          ;load ASCII
```

```
0003 81 E3 0F0F      AND BX, 0F0FH        ;mask BX
```

TABLE +9 Example AND instructions.

Assembly Instructions	Operations
AND AL, BL	AL = AL and BL
AND CX, DX	CX = CX and DX
AND ECX, EDI	ECX = ECX and EDI
AND RDX, RBP	RDX = RDX and RBP 164-bit mode2
AND CL, 33H	CL = CL and 33H
AND DI, 4FFFH	DI = DI and 4FFFH
AND ESI, 34H	ESI = ESI and 34H
AND RAX, 1	RAX = RAX and 1 164-bit mode2
AND AX, [DI]	The word contents of the data segment memory location addressed by DI are ANDed with AX
AND ARRAY [SI], AL	The byte contents of the data segment memory location addressed by ARRAY plus SI are ANDed with AL
AND [EAX], CL	CL is ANDed with the byte contents of the data segment memory location addressed by ECX

OR

The **OR operation** performs logical addition and is often called the *Inclusive-OR* function. The OR function generates a logic 1 output if any inputs are 1. A 0 appears at the output only when all inputs are 0. The truth table for the OR function appears in Table +10. Here, the inputs A and B OR together to produce the X output. It is important to remember that 1 ORed with anything yields a 1.

The OR instruction uses any of the addressing modes allowed to any other instruction except segment register addressing. Table +11 illustrates several example OR instructions with comments about their operation.

A	B	T
0	0	0
0	1	0
1	0	0
1	1	1

TABLE +11 Example OR instructions.

Assembly Instructions	Operations
OR AH, BL	AL = AL or BL
OR SI, DX	SI = SI or DX
OR EAX, EBX	EAX = EAX or EBX
OR R9, R10	R9 = R9 or R10 164-bit mode2
OR DH, 0A3H	DH = DH or 0A3H
OR SP, 990DH	SP = SP or 990DH
OR EBP, 10	EBP = EBP or 10
OR RBP, 1000H	RBP = RBP or 1000H 164-bit mode2
OR DX, [BX]	DX is ORed with the word contents of data segment Memory location addressed by BX
OR DATES [DI + 2], AL	The byte contents of the data segment memory Location addressed by DI plus 2 are ORed with AL

Suppose that two BCD numbers are multiplied and adjusted with the AAM instruction. The result appears in AX as a two-digit unpacked BCD number. Example |1 illustrates this multiplication and shows how to change the result into a two-digit ASCII-coded number using the OR instruction. Here, OR AX, 3030H converts the 0305H found in AX to 3335H. The OR operation can be replaced with an ADD AX, 3030H to obtain the same results.

EXAMPLE ||1

```
0000 B0 05          MOV   AL, 5           ;load data
0002 B3 07          MOV   BL, 7
0004 F6 E3          MUL   BL
```

```
0006 D4 0A          AAM              ;adjust
0008 0D 3030        OR   AX, 3030H   ;convert to ASCII
```

Exclusive-OR

The **Exclusive-OR** instruction (XOR) differs from Inclusive-OR (OR). The difference is that a 1, 1 condition of the OR function produces a 1; the 1, 1 condition of the Exclusive-OR operation produces a 0. The Exclusive-OR operation excludes this condition; the Inclusive-OR includes it. Table+12 show the truth table of the Exclusive-OR function. If the inputs of the Exclusive-OR function are both 0 and both 1, the output is 0. If the inputs are different, the output is 1. Because of this, the Exclusive-OR is sometimes called a comparator.

A	B	T
0	0	0
0	1	1
1	0	1
1	1	0

The XOR instruction uses any addressing mode except segment register addressing. Table+13 lists several Exclusive-OR instructions and their operations.

TABLE +13 Example XOR instructions.

Assembly Instructions	Operations
XOR CH, DL	CH = CH xor DL
XOR SI, BX	SI = SI xor BX
XOR EBX, EDI	EBX = EBX xor EDI
XOR RAX, RBX	RAX = RAX xor RBX 164-bit mode2
XOR AH, 0EEH	AH = AH xor 0EEH
XOR DI, 00DDH	DI = DI xor 00DDH
XOR ESI, 100	ESI = ESI xor 100
XOR R12, 20	R12 = R12 xor 20 164-bit mode2
XOR DX, [SI]	DX is Exclusive-ORed with the word contents of the data segment memory location addressed by SI
XOR DEAL [BP+2], AH	AH is Exclusive-ORed with the byte contents of the stack segment memory location addressed by BP plus 2

Example 0+1 shows a short sequence of instructions that clears bits 0 and 1 of CX, sets bits 9 and 10 of CX, and inverts bit 12 of CX. The OR instruction is used to set bits, the AND instruction is used to clear bits, and the XOR instruction inverts bits.

EXAMPLE 0+1

```
0000 81 C9 0600       OR CX, 0600H          ;set bits 9 and 10
0004 83 E1 FC         AND CX, 0FFFCH        ;clear bits 0 and 1
0007 81 F1 1000       XOR CX, 1000H         ;invert bit 12
```

NOT and NEG

Logical inversion, or the one's complement (NOT), and arithmetic sign inversion, or the two's complement (NEG), are the last two logic functions presented (except for shift and rotate in the next section of the text). These are two of a few instructions that contain only one operand.

Table +14 lists some variations of the NOT and NEG instructions. As with most other instructions, NOT and NEG can use any addressing mode except segment register addressing.

TABLE +14 Example NOT instructions.

Assembly Instructions	Operations
NOT CH	CH is one's complemented
NEG CH	CH is two's complemented
NEG AX	AX is two's complemented
NOT EBX	EBX is one's complemented
NEG ECX	ECX is two's complemented
NOT RAX	RAX is one's complemented (64-bit mode)
NOT TEMP	The contents of data segment memory location TEMP is one's complemented
NOT BYTE PTR[BX]	The byte contents of the data segment memory Location addressed by BX are one's complemented

The NOT instruction inverts all bits of a byte, word, or double word. The NEG instruction two's complements a number, which means that the arithmetic sign of a signed number changes from positive to negative or from negative to positive. The NOT function is considered logical, and the NEG function is considered an arithmetic operation.

Floating Point instructions

A big problem with floating point arithmetic is that it does not follow the standard rules of algebra. Nevertheless, many programmers apply normal algebraic rules when using floating point arithmetic. This is a source of bugs in many programs. One of the primary goals of this section is to describe the limitations of floating point arithmetic so you will understand how to use it properly.

Normal algebraic rules apply only to *infinite precision* arithmetic. Consider the simple statement x:=x+1, x is an integer. On any modern computer this statement follows the normal rules of algebra *as long as overflow does not occur*. That is, this statement is valid only for certain values of x (min int <= x < max int). Most programmers do not have a problem with this because they are well aware of the fact that integers in a program do not follow the standard algebraic rules.

Integers do not follow the standard rules of algebra because the computer represents them with a finite number of bits. You cannot represent any of the (integer) values above the maximum integer or below the minimum integer. Floating point values suffer from this same problem, only worse. After all, the integers are a subset of the real numbers. Therefore, the floating point values must represent the same infinite set of integers. However, there are an infinite number of values between any two real values, so this problem is infinitely worse. Therefore, as well as having to limit your values between a maximum and minimum range, you cannot represent all the values between those two ranges, either.

To represent real numbers, most floating point formats employ scientific notation and use some number of bits to represent a *mantissa* and a smaller number of bits to represent an *exponent*. The end result is that floating point numbers can only represent numbers with a specific number of *significant* digits. This has a big impact on how floating point arithmetic operations. To easily see the impact of limited precision arithmetic, we will adopt a simplified decimal floating point format for our examples. Our floating point format will provide a mantissa with three significant digits and a decimal exponent with two digits. The mantissa and exponents are both signed values.

When adding and subtracting two numbers in scientific notation, you must adjust the two values so that their exponents are the same. For example, when adding 1.23e1 and 4.56e0, you must adjust the values so they have the same exponent. One way to do this is to to convert 4.56e0 to 0.456e1 and then add. This produces 1.686e1. Unfortunately, the result does not fit into three significant digits, so we must either *round* or *truncate* the result to three significant digits. Rounding generally produces the most accurate result, so let's round the result to obtain 1.69e1. As you can see, the lack of *precision* (the number of digits or bits we maintain in a computation) affects the accuracy (the correctness of the computation).

In the previous example, we were able to round the result because we maintained *four* significant digits *during* the calculation. If our floating point calculation is limited to three significant digits *during* computation, we would have had to truncate the last digit of the smaller number, obtaining 1.68e1 which is even less correct. Extra digits available during a computation are known as *guard digits* (or *guard bits* in the case of a binary format). They greatly enhance accuracy during a long chain of computations.

The accuracy loss during a single computation usually isn't enough to worry about unless you are greatly concerned about the accuracy of your computations. However, if you compute a value which is the result of a sequence of floating point operations, the error can *accumulate* and greatly affect the computation itself. For example, suppose we were to add 1.23e3 with 1.00e0. Adjusting the numbers so their exponents are the same before the addition produces 1.23e3 + 0.001e3. The sum of these two values, even after rounding, is 1.23e3. This might seem perfectly reasonable to you; after all, we can only maintain three significant digits, adding in a small value shouldn't affect the result at all.

However, suppose we were to add 1.00e0 1.23e3 *ten times*. The first time we add 1.00e0 to 1.23e3 we get 1.23e3. Likewise, we get this same result the second, third, fourth, ..., and tenth time we add 1.00e0 to 1.23e3. On the other hand, had we added 1.00e0 to itself ten times, then added the result (1.00e1) to 1.23e3, we would have gotten a different result, 1.24e3. This is the most important thing to know about limited precision arithmetic: The order of evaluation can affect the accuracy of the result. You will get more accurate results if the relative magnitudes (that is, the exponents) are close to one another. If you are performing a chain calculation involving addition and subtraction, you should attempt to group the values appropriately.

Another problem with addition and subtraction is that you can wind up with *false precision*. Consider the computation 1.23e0 - 1.22 e0. This produces 0.01e0. Although this is mathematically equivalent to 1.00e-2, this latter form suggests that the last two digits are exactly zero. Unfortunately, we've only got a single significant digit at this time. Indeed, some FPUs or floating point software packages might actually insert random digits (or bits) into the L.O. positions. This brings up a second important rule concerning limited precision arithmetic:

Whenever subtracting two numbers with the same signs or adding two numbers with different signs, the accuracy of the result may be less than the precision available in the floating point format.

Multiplication and division do not suffer from the same problems as addition and subtraction since you do not have to adjust the exponents before the operation; all you need to do is add the exponents and multiply the mantissas (or subtract the exponents and divide the mantissas). By themselves, multiplication and division do not produce particularly poor results. However, they tend to multiply any error which already exists in a value. For example, if you multiply 1.23e0 by two, when you should be multiplying 1.24e0 by two, the result is even less accurate. This brings up a third important rule when working with limited precision arithmetic:

When performing a chain of calculations involving addition, subtraction, multiplication, and division, try to perform the multiplication and division operations first.

Often, by applying normal algebraic transformations, you can arrange a calculation so the multiply and divide operations occur first. For example, suppose you want to compute x*(y+z). Normally you would add y and z together and multiply their sum by x. However, you will get a little more accuracy if you transform x*(y+z) to get x*y+x*z and compute the result by performing the multiplications first.

Multiplication and division are not without their own problems. When multiplying two very large or very small numbers, it is quite possible for *overflow* or *underflow* to occur. The same situation occurs when dividing a small number by a large number or dividing a large number by a small number. This brings up a fourth rule you should attempt to follow when multiplying or dividing values:

When multiplying and dividing sets of numbers, try to arrange the multiplications so that they multiply large and small numbers together; likewise, try to divide numbers that have the same relative magnitudes.

Comparing floating pointer numbers is very dangerous. Given the inaccuracies present in any computation (including converting an input string to a floating point value), you should *never* compare two floating point values to see if they are equal. In a binary floating point format, different computations which produce the same (mathematical) result may differ in their least significant bits. For example, adding 1.31e0+1.69e0 should produce 3.00e0. Likewise, adding 2.50e0+1.50e0 should produce 3.00e0. However, were you to compare (1.31e0+1.69e0) agains (2.50e0+1.50e0) you might find out that these sums are *not* equal to one another. The test for equality succeeds if and only if all bits (or digits) in the two operands are exactly the same. Since this is not necessarily true after two different floating point computations which should produce the same result, a straight test for equality may not work.

The standard way to test for equality between floating point numbers is to determine how much error (or tolerance) you will allow in a comparison and check to see if one value is within this error range of the other. The straight-forward way to do this is to use a test like the following:

```
    if Value1 >= (Value2-error) and Value1 <= (Value2+error) then …
```

Another common way to handle this same comparison is to use a statement of the form:

```
if abs(Value1-Value2) <= error then …
```

Most texts, when discussing floating point comparisons, stop immediately after discussing the problem with floating point equality, assuming that other forms of comparison are perfectly okay with floating point numbers. This isn't true! If we are assuming that x=y if x is within y ±error, then a simple bitwise comparison of x and y will claim that x<y if y is greater than x but less than y+error. However, in such a case x should really be treated as equal to y, not less than y. Therefore, we must always compare two floating point numbers using ranges, regardless of the actual comparison we want to perform. Trying to compare two floating point numbers directly can lead to an error. To compare two floating point numbers, x and y, against one another, you should use one of the following forms:

```
= if abs(x-y) <= error then …
≠ if abs(x-y) > error then …
< if (x-y) < error then …
≤ if (x-y) <= error then …
> if (x-y) > error then …
≥ if (x-y) >= error then …
```

You must exercise care when choosing the value for *error*. This should be a value slightly greater than the largest amount of error which will creep into your computations. The exact value will depend upon the particular floating point format you use, but more on that a little later. The final rule we will state in this section is

When comparing two floating point numbers, always compare one value to see if it is in the range given by the second value plus or minus some small error value.

There are many other little problems that can occur when using floating point values. This text can only point out some of the major problems and make you aware of the fact that you cannot treat floating point arithmetic like real arithmetic – the inaccuracies present in limited precision arithmetic can get you into trouble if you are not careful. A good text on numerical analysis or even scientific computing can help fill in the details which are beyond the scope of this text. If you are going to be working with floating point arithmetic, *in any language*, you should take the time to study the effects of limited precision arithmetic on your computations.