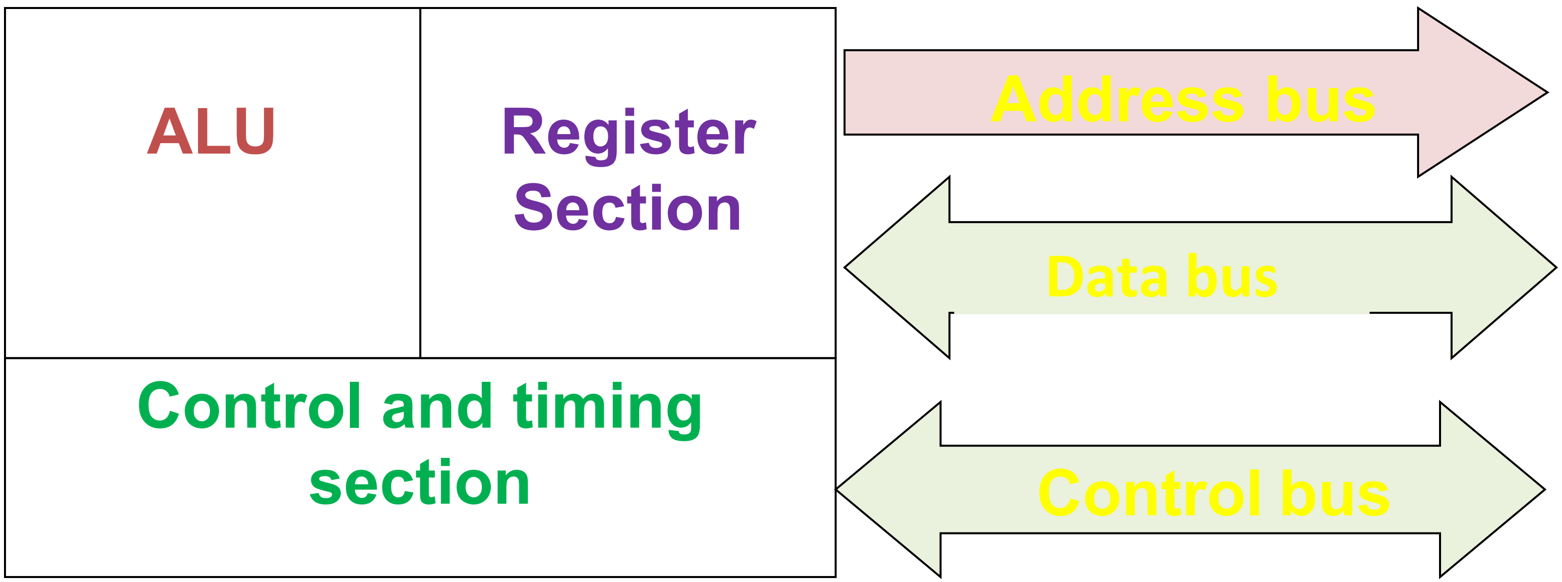
| Sl No | Microprocessor | Microcontroller |
|---|---|---|
| 1 | CPU is stand-alone, RAM, ROM, I/O, timer are separate | CP,RAM,ROM,I/O and timer are all on single chip |
| 2 | Designer can decide on the amount of ROM,RAM and I/O ports | Fix amount of on-chip ROM, RAM, I/O ports |
| 3 | Expansive | Not Expansive |
| 4 | General purpose | Single purpose |
| 5 | Microprocessor based system design is complex and expensive | Microcontroller based system design is rather simple and cost effective |
| 6 | The instruction set of microprocessor is complex with large number of instructions. | The instruction set of a Microcontroller is very simple with the less number of instructions. |

# Internal structure and basic operation of microprocessor




**Block diagram of a Microprocessor**

- Microprocessor performs three main tasks:
  - data transfer between itself and the memory or I/O systems
  - simple arithmetic and logic operations
  - program flow via simple decisions

# Microprocessor types

- Microprocessors can be characterized based on
  - the word size
    - 8 bit, 16 bit, 32 bit, etc. processors
  - Instruction set structure
    - RISC (Reduced Instruction Set Computer), CISC (Complex Instruction Set Computer)
  - Functions
    - General purpose, special purpose such image processing, floating point calculations
  - And more …

# Evolution of Microprocessors

- The first **microprocessor** was **introduced** in 1971 by Intel Corp.
- It was named Intel 4004 as it was a 4 bit processor.

Categories according to the generations or size

**First Generation (4 - bit Microprocessors)**

- could perform simple arithmetic such as addition, subtraction, and logical operations like Boolean OR and Boolean AND.
- had a control unit capable of performing control functions like
  - fetching an instruction from storage memory,
  - decoding it, and then
  - generating control pulses to execute it.

**Second Generation (8 - bit Microprocessor)**

- The second generation microprocessors were introduced in 1973 again by Intel.
- the first 8 - bit microprocessor which could perform arithmetic and logic operations on 8-bit words.

**Third Generation (16 - bit Microprocessor)**

- introduced in 1978
- represented by **Intel's 8086, Zilog Z800 and 80286**,
- 16 - bit processors with a performance like minicomputers.

**Fourth Generation (32 - bit Microprocessors)**

- Several different companies introduced the 32-bit microprocessors
- the most popular one is the **Intel 80386**

**Fifth Generation (64 - bit Microprocessors)**

- Introduced in 1995
- After 80856, Intel came out with a new processor namely Pentium processor followed by **Pentium Pro CPU**
- allows multiple CPUs in a single system to achieve multiprocessing.
- Other improved 64-bit processors are **Celeron, Dual, Quad, Octa Core processors**.

# Typical microprocessors

- Most commonly used
  - 68K
    - Motorola
  - x86
    - Intel
  - IA-64
    - Intel
  - MIPS
    - Microprocessor without interlocked pipeline stages
  - ARM
    - Advanced RISC Machine
  - PowerPC
    - Apple-IBM-Motorola alliance
  - Atmel AVR
- A brief summary will be given later

# 8086 Microprocessor

- designed by Intel in 1976
- 16-bit Microprocessor having
- 20 address lines
- 16 data lines
- provides up to 1MB storage
- consists of powerful instruction set, which provides operations like multiplication and division easily.

supports two modes of operation

Maximum mode :

suitable for system having multiple processors

Minimum mode :

suitable for system having a single processor.

# Features of 8086

- Has an instruction queue, which is capable of storing six instruction bytes
- First 16-bit processor having
  - 16-bit ALU
  - 16-bit registers
  - internal data bus
  - 16-bit external data bus

uses two stages of pipelining

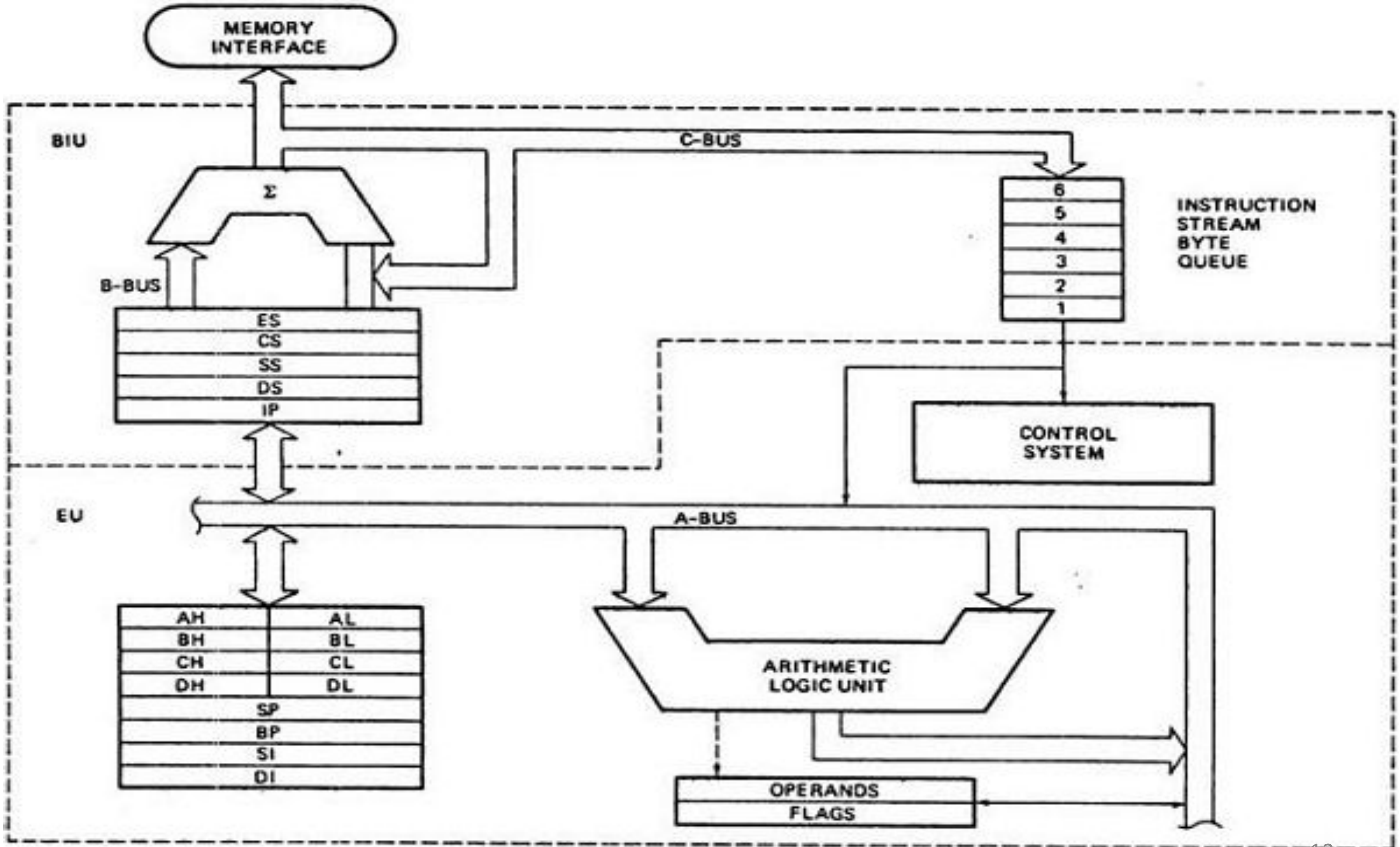
1. Fetch Stage and
2. Execute Stage

which improves performance.

Fetch stage : can pre-fetch up to 6 bytes of instructions and stores them in the queue.

Execute stage : executes these instructions.

# Architecture of 8086



# Segments in 8086

memory is divided into various sections called segments

**Code segment** : where you store the program.

**Data segment** : where the data is stored.

**Extra segment** : mostly used for string operations.

**Stack segment** : used to push/pop

# General purpose registers

used to store temporary data within the microprocessor

**AX –** Accumulator

16 bit register

divided into two 8-bit registers AH and AL to perform 8-bit instructions also

generally used for arithmetical and logical instructions

**BX –** Base register

16 bit register

divided into two 8-bit registers BH and BL to perform 8-bit instructions also

Used to store the value of the offset.

**CX –** Counter register

16 bit register

divided into two 8-bit registers CH and CL to perform 8-bit instructions also

Used in looping and rotation

**DX –** Data register

16 bit register

divided into two 8-bit registers DH and DL to perform 8-bit instructions also

Used in multiplication an input/output port addressing

# Pointers and Index Registers

**SP – S**tack pointer

16 bit register

points to the topmost item of the stack

If the stack is empty the stack pointer will be (FFFE)H

It's offset address relative to stack segment

**BP –B**ase pointer

16 bit register

used in accessing parameters passed by the stack

It's offset address relative to stack segment

**SI – S**ource index register

16 bit register

used in the pointer addressing of data and

as a source in some string related operations

It's offset is relative to data segment

**DI – D**estination index register

16 bit register

used in the pointer addressing of data and

as a destination in string related operations

It's offset is relative to extra segment.

**IP -** Instruction Pointer

16 bit register

stores the address of the next instruction to be executed

also acts as an offset for CS register.

# Segment Registers

**CS - Code Segment Register:**

user cannot modify the content of these registers

Only the microprocessor's compiler can do this

**DS - Data Segment Register:**

The user can modify the content of the data segment.

**SS - Stack Segment Registers:**

used to store the information about the memory segment.

operations of the SS are mainly Push and Pop.

**ES - Extra Segment Register:**

By default, the control of the compiler remains in the DS where the user can add and modify the instructions

If there is less space in that segment, then ES is used

Also used for copying purpose.

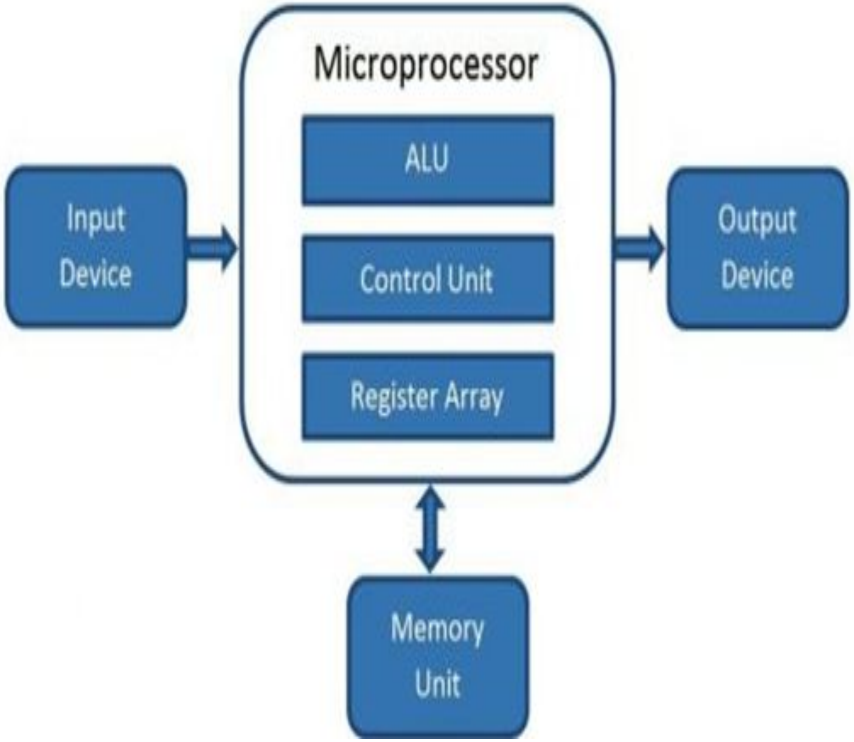
# Flag or Status Register

- 16-bit register
- contains 9 flags
- remaining 7 bits are idle in this register
- These flags tell about the status of the processor after any arithmetic or logical operation
- IF the flag value is 1, the flag is set, and if it is 0, it is said to be reset.

# Microcomputer

- A digital computer with one microprocessor which acts as a CPU
- A  complete computer on a small scale, designed for use by one person at a time
- called a personal computer (PC)
- a device based on a single-chip microprocessor
- includes laptops and desktops

**Block Diagram**



# Introduction to 8086 Assembly Language

*Assembly Language Programming*

# Program Statements

- **Program consist of statement, one per line.**
- **Each statement is either an instruction, which the assembler translate into machine code, or assembler directive, which instructs the assembler to perform some specific task, such as allocating memory space for a variable or creating a procedure.**
- **Both instructions and directives have up to four fields:**
- **At least one blank or tab character must separate the fields**

**name operation operand(s) comment**

# Program Statements

- An example of an instruction is

    START: MOV CX,5 ; initialize counter

    name operation operand(s) comment

    - The name field consists of the label START:
    - The operation is MOV, the operands are CX and 5
    - And the comment is ; initialize counter

- An example of an assembler directive is

    MAIN PROC

    name operation operand(s) comment

    - MAIN is the name, and the operation field contains PROC
    - This particular directive creates a procedure called MAIN

# Program Statements

## name operation operand(s) comment

- A Name field identifies a label, variable, or symbol.
  It may contain any of the following character :
  A,B…..Z ; a,b….z ; 0,1….9 ; ? ;
  _ (underline) ; @ ; $ ; . (period)

- Only the first 31 characters are recognized
- There is no distinction between uppercase and lower case letters.
- The first character may not be a digit
- If it is used, the period ( . ) may be used only as the first character.
- A programmer-chosen name may not be the same as an assembler reserved word.

# Program Statements

name operation operand(s) comment

- Operation field is a predefined or reserved word
  - mnemonic - symbolic operation code.
    - The assembler translates a symbolic opcode into a machine language opcode.
    - Opcode symbols often discribe the operation’s function; for example, MOV, ADD, SUB
  - assemler directive - pseudo-operation code.
    - In an assembler directive, the operation field contains a pseudo-operation code (pseudo-op)
    - Pseudo-op are not translated into machine code; for example the PROC pseudo-op is used to create a procedure

# Program Statements

name operation operand(s) comment

- An operand field specifies the data that are to be acted on by the operation.
- An instruction may have zero, one, or two operands. For example:
  - NOP No operands; does nothing
  - INC AX one operand; adds 1 to the contents of AX
  - ADD WORD1,2 two operands; adds 2 to the contents of memory word WORD1

# Program Statements

**name operation operand(s) comment**

- **The comment field is used by the programmer to say something about what the statement does.**

- **A semicolon marks the beginning of this field, and the assembler ignores anything typed after semicolon.**

- **Comments are optional, but because assembly language is low level, it is almost impossible to understand an assembly language program without comments.**

# Program Data and Storage

- Pseudo-ops to define data or reserve storage
  - DB - byte(s)
  - DW - word(s)
  - DD - doubleword(s)
  - DQ - quadword(s)
  - DT - tenbyte(s)
- These directives require one or more operands
  - define memory contents
  - specify amount of storage to reserve for run-time data

# Defining Data

- Numeric data values
  - 100 - decimal
  - 100B - binary
  - 100H - hexadecimal
  - '100' - ASCII
  - "100" - ASCII
- Use the appropriate DEFINE directive (byte, word, etc.)
- A list of values may be used - the following creates 4 consecutive words

```
DW 40CH,10B,-13,0
```

- A ? represents an uninitialized storage location

```
DB 255,?,-128,'X'
```

# Naming Storage Locations

- Names can be associated with storage locations

```
ANum DB -4
 DW 17
ONE
UNO DW 1
X DD ?
```

- These names are called variables

- ANum refers to a byte storage location, initialized to FCh
- The next word has no associated name
- ONE and UNO refer to the same word
- X is an unitialized doubleword

- Multiple definitions can be abbreviated

  Example:

```
message  DB 'B'
         DB 'y'
         DB 'e'
         DB 0DH
         DB 0AH
```

  can be written as

```
message  DB 'B','y','e',0DH,0AH
```

- More compactly as

```
message DB 'Bye',0DH,0AH
```

# Arrays

- Any consecutive storage locations of the same size can be called an array

```
X DW 40CH,10B,-13,0
Y DB 'This is an array'
Z DD -109236, FFFFFFFFH, -1, 100B
```

- Components of X are at X, X+2, X+4, X+6
- Components of Y are at Y, Y+1, …, Y+15
- Components of Z are at Z, Z+4, Z+8, Z+12

# DUP

- Allows a sequence of storage locations to be defined or reserved
- Only used as an operand of a define directive

**DB 40 DUP (?) ; 40 words, uninitialized**

**DW 10h DUP (0) ; 16 words, initialized as 0**

**Table1 DW 10 DUP (?) ; 10 words, uninitialized**

**message DB 3 DUP ('Baby') ; 12 bytes, initialized**
**; as BabyBabyBaby**

**Name1 DB 30 DUP ('?') ; 30 bytes, each**
**; initialized to ?**

# DUP

- **The DUP directive may also be nested**

Example

stars  DB  4  DUP(3 DUP ('*'),2 DUP ('?'),5 DUP ('!'))

Reserves 40-bytes space and initializes it as

***??!!!!!***??!!!!!***??!!!!!***??!!!!!

matrix  DW  10 DUP (5 DUP (0))

defines a 10X5 matrix and initializes its elements to zero.

This declaration can also be done by

matrix  DW  50 DUP (0)

# Word Storage

- Word, doubleword, and quadword data are stored in reverse byte order (in memory)

| Directive | Bytes in Storage |
|---|---|
| DW 256 | 00 01 |
| DD 1234567H | 67 45 23 01 |
| DQ 10 | 0A 00 00 00 00 00 00 00 |
| X DW 35DAh | DA 35 |

Low byte of X is at X, high byte of X is at X+1

# Word Storage

## Symbol Table

- Assembler builds a symbol table so we can refer to the allocated storage space by the associated label

**Example**

```
.DATA
value     DW   0
sum       DD   0
marks     DW   10 DUP (?)
message   DB   'The grade is:',0
char1     DB   ?
```

| name | offset |
|---|---|
| value | 0 |
| sum | 2 |
| marks | 6 |
| message | 26 |
| char1 | 40 |

# Named Constants

- Symbolic names associated with storage locations represent addresses
- Named constants are symbols created to represent specific values determined by an expression
- Named constants can be numeric or string
- Some named constants can be redefined
- No storage is allocated for these values

# Equal Sign Directive

- name = expression
  - expression must be numeric
  - these symbols may be redefined at any time

```
maxint = 7FFFh
count = 1
DW count
count = count * 2
DW count
```

# EQU Directive

- name EQU expression
  - expression can be string or numeric
  - Use < and > to specify a string EQU
  - these symbols cannot be redefined later in the program

```
sample EQU 7Fh
aString EQU <1.234>
message EQU <This is a message>
```

# Data Transfer Instructions

- **MOV *target, source***
  - **reg, reg**
  - **mem, reg**
  - **reg, mem**
  - **mem, immed**
  - **reg, immed**
- Sizes of both operands must be the same

- reg can be any non-segment register except IP cannot be the target register
- MOV's between a segment register and memory or a 16-bit register are possible

# Sample MOV Instructions

```
b db 4Fh
w dw 2048

mov bl,dh
mov ax,w
mov ch,b
mov al,255
mov w,-100
mov b,0
```

- When a variable is created with a define directive, it is assigned a default size attribute (byte, word, etc)
- You can assign a size attribute using LABEL

```
LoByte LABEL BYTE
aWord DW 97F2h
```

# Program Segment Structure

- Data Segments
  - Storage for variables
  - Variable addresses are computed as offsets from start of this segment
- Code Segment
  - contains executable instructions
- Stack Segment
  - used to set aside storage for the stack
  - Stack addresses are computed as offsets into this segment
- Segment directives

```
.data
.code
.stack size
```

# Instruction types

## Data transfer instructions

### 8086 instruction set

IN Input byte or word from port
LAHF Load AH from flags
LDS Load pointer using data segment
LEA Load effective address
LES Load pointer using extra segment
MOV Move to/from register/memory
OUT Output byte or word to port
POP Pop word off stack
POPF Pop flags off stack
PUSH Push word onto stack
PUSHF Push flags onto stack
SAHF Store AH into flags
XCHG Exchange byte or word
XLAT Translate byte

### Additional 80286 instructions

INS Input string from port
OUTS Output string to port
POPA Pop all registers
PUSHA Push all registers

### Additional 80386 instructions

LFS Load pointer using FS
LGS Load pointer using GS
LSS Load pointer using SS
MOVSX Move with sign extended
MOVZX Move with zero extended
POPAD Pop all double (32 bit) registers
POPD Pop double register
POPFD Pop double flag register
PUSHAD Push all double registers
PUSHD Push double register
PUSHFD Push double flag register

### Additional 80486 instruction

BSWAP Byte swap

### Additional Pentium instruction

MOV Move to/from control register

# Arithmetic instructions

**8086 instruction set**

AAA ASCII adjust for addition
AAD ASCII adjust for division
AAM ASCII adjust for multiply
AAS ASCII adjust for subtraction
ADC Add byte or word plus carry
ADD Add byte or word
CBW Convert byte or word
CMP Compare byte or word
CWD Convert word to double-word
DAA Decimal adjust for addition
DAS Decimal adjust for subtraction
DEC Decrement byte or word by one
DIV Divide byte or word
IDIV Integer divide byte or word
IMUL Integer multiply byte or word
INC Increment byte or word by one
MUL Multiply byte or word (unsigned)
NEG Negate byte or word
SBB Subtract byte or word and carry (borrow)
SUB Subtract byte or word

**Additional 80386 instructions**

CDQ Convert double-word to quad-word
CWDE Convert word to double-word

**Additional 80486 instructions**

CMPXCHG Compare and exchange
XADD Exchange and add

**Additional Pentium instruction**

CMPXCHG8B Compare and exchange 8 bytes

# Bit manipulation instructions

**8086 instruction set**

**AND Logical AND of byte or word**
**NOT Logical NOT of byte or word**
**OR Logical OR of byte or word**
**RCL Rotate left trough carry byte or word**
**RCR Rotate right trough carry byte or word**
**ROL Rotate left byte or word**
**ROR Rotate right byte or word**
**SAL Arithmetic shift left byte or word**
**SAR Arithmetic shift right byte or word**
**SHL Logical shift left byte or word**
**SHR Logical shift right byte or word**
**TESTTest byte or word**
**XOR Logical exclusive-OR of byte or word**

**Additional 80386 instructions**

**BSF Bit scan forward**
**BSR Bit scan reverse**
**BT Bit test**
**BTC Bit test and complement**
**BTR Bit test and reset**
**BTS Bit test and set**
**SETcc Set byte on condition**
**SHLD Shift left double precision**
**SHRD Shift right double precision**

# String instructions

## 8086 instruction set

| | |
|---|---|
| CMPS | Compare byte or word string |
| LODS | Load byte or word string |
| MOVS | Move byte or word string |
| MOVSB(MOVSW) | Move byte string (word string) |
| REP | Repeat |
| REPE (REPZ) | Repeat while equal (zero) |
| REPNE (REPNZ) | Repeat while not equal (not zero) |
| SCAS | Scan byte or word string |
| STOS | Store byte or word string |

# Program Skeleton

```
.model small
.stack 100H
.data
   ;declarations
.code
main proc
   ;code
main endp
   ;other procs
end main
```

- Select a memory model
- Define the stack size
- Declare variables
- Write code
  - organize into procedures
- Mark the end of the source file
  - optionally, define the entry point

# EXAMPLE : Adding two 8 bit numbers

```
DATA SEGMENT                    ; Data Segment
N1
3n2 DB 12H
N2 DB 21H
RES DB ?
DATA ENDS
CODE SEGMENT                    ; Code segment
ASSUME CS: CODE, DS: DATA
START: MOV AX, DATA
MOV DS, AX
MOV AL, N1
MOV BL, N2
ADD AL, BL
MOV RES, AL
INT 21H
CODE ENDS
END START
```