Chapter 1

Introduction

A computer system consists of a processor, memory, and input/output devices. Input/output devices are used to interact with the outside world. While memory of the system is the processor’s internal world. The processor has its own set of instructions to perform various operations like arithmetical, logical, and control activities. These set of instructions are called **'machine language instructions'**.

A computer processor only understands the machine language that consists of only **1’s** and **0’s**. But it was too difficult and complex for using in software development. So, we introduced the low-level assembly language which was designed for a specific family of processors that represents various instructions in symbolic code and a more understandable form.

Advantages of Assembly Language

1. Writing code in assembly language can help to deeply understand computer architecture and how instructions are executed at the hardware level.
2. Assembly language provides direct control over the hardware components of a computer, such as registers, memory addresses, and the instruction set.
3. Assembly programs is more efficient due to the direct correspondence between assembly instructions and machine code.
4. It is good for certain tasks, especially those requiring low-level manipulation of hardware or highly optimized algorithms

Instruction Cycle

The instruction cycle, also known as the **fetch-decode-execute cycle**, is the fundamental process that a computer's central processing unit (CPU) follows to execute a machine-level instruction. The cycle consists of several stages:

1. **Fetch**: The CPU fetches the next instruction from memory. The program counter (PC) holds the memory address of the next instruction to be fetched. The CPU reads the instruction from this address and increments the PC to point to the next instruction.
2. **Decode**: The fetched instruction is decoded by the CPU. This involves determining the operation to be performed and the operands involved.
3. **Execute**: The CPU performs the operation specified by the instruction.
4. **Write Back**: If the executed instruction produced a result that needs to be stored, the CPU writes the result back to the appropriate destination, such as a register or memory location.

This cycle continues until the program is finished or interrupted.

Computer Architecture

Memory

Memory is a vital component of a computer's architecture that provides storage for data and instructions that the central processing unit (CPU) actively uses during program execution. Memory is characterized by two main dimensions: cell width and the number of cells.

1. Cell Width:
The cell width, also referred to as the word size, is the number of bits that can be read from or written to a single memory cell in one operation. A common word size in modern computers is 8 bits (1 byte).

2. Number of Cells:
The number of cells refers to the total count of individual memory storage units (cells) available in the memory. Each cell can store a single unit of data, such as a byte or a word, based on the word size.

Caches

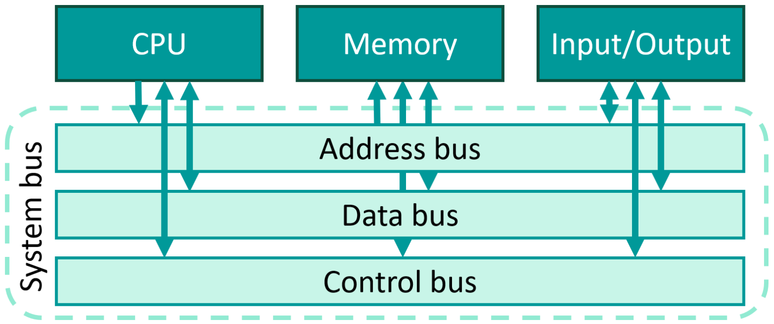
Caches are a type of high-speed volatile memory that serves as a bridge between the extremely fast CPU registers and the slower main memory. Caches store copies of frequently used data and instructions to reduce the time the CPU spends waiting for data to be fetched from memory.

Buses

Buses are a important component of computer architecture that helps to communicate between various hardware components, such as the central processing unit (CPU), memory, and input/output devices. Buses are pathways used for transmitting different types of information, such as addresses, data, and control signals, between these components.

1. **Address Bus**: It carries the memory address of the location the processor wants to read from or write to. The width of the address bus determines the maximum amount of memory that the processor can address directly. For example, a 32-bit address bus can address up to 2^32 (4 GB) memory locations.
2. **Data Bus**: The data bus is used to transfer actual data between the memory and the processor.
3. **Control Bus**: The control bus is a collection of lines that carries control signals between the CPU and other devices in the computer system. These signals convey information about the type of operation to be performed (read or write).

The control bus is bidirectional because it needs to facilitate communication in both directions. It not only carries signals from the CPU to the memory and other components but also conveys feedback or status information from these components back to the CPU. For example, a status line might indicate if a peripheral device is ready to receive data from the CPU.



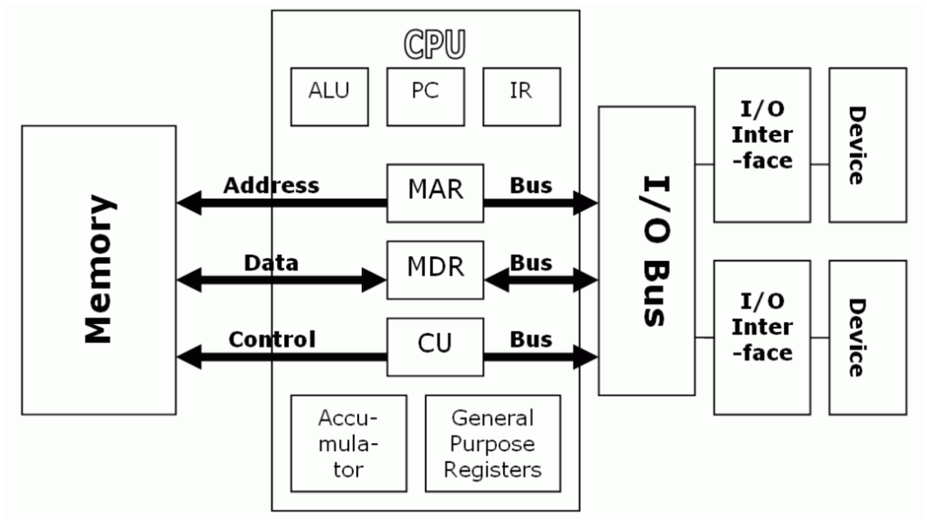
Registers

Registers are small, high-speed storage locations within a central processing unit (CPU) that are used for temporary storage and manipulation of data during program execution. Registers are vital for holding operands, intermediate results, and control information required for the CPU's operations.

Here are some key points about registers and their roles:

1. **Accumulator:** All mathematical and logical operations are performed on the accumulator. The word side of a processor is defined by the width of its accumulator.
2. **Pointer, Index, or Base Register:** Holds the address of data.
3. **Flags Register:** Stores condition codes or flags that indicate the outcome of arithmetic and logical operations.
4. **Program Counter or Instruction Pointer:** Keeps track of the address of the next instruction to be executed.
5. **Opcode:** Stores the instruction code that tells the CPU what operation to perform.

Registers are usually identified by names or mnemonics that represent their functions. These names make programming more readable and understandable, especially given that there are only a limited number of registers in a CPU.



Now we select a specific architecture to discuss assembly language in detail. We are going to use IBM PC based on Intel Architecture.

Intel “iAPX88” Architecture

The iAPX88 stands for **“Intel Advanced Processor Extensions 88”**. Intel 8088 is a 16-bit microprocessor introduced in 1979. It's a part of the 8086 family of processors and is often considered the brain of the original IBM PC (IBM Personal Computer) released in 1981.

Key Features:

- **16-Bit Architecture:** The 8088 is a 16-bit microprocessor, which means it can process data in 16-bit chunks at a time.
- **16-Bit Data Bus:** The 8088 has a 16-bit data bus, which means it can transfer 16 bits of data between the CPU and memory or peripherals in a single operation.
- **20-Bit Address Bus:** The 8088 features a 20-bit address bus, which theoretically allows it to address up to 1 MB of memory.
- **Instruction Set:** The 8088 uses the x86 instruction set architecture. It supports a variety of data manipulation, arithmetic, logical, and control operations.
- **Registers:** The 8088 includes several registers, including general-purpose registers, segment registers, the instruction pointer (IP), and status flags. These registers play a crucial role in executing instructions and managing data.

The Intel 8088 played a pivotal role in shaping the early days of personal computing and contributed to the widespread adoption of PCs.

Register Architecture

Here's an overview of the specific registers associated with the Intel x86 architecture:

General Registers (AX, BX, CX, DX):

- **AX (Accumulator):** Often used for arithmetic and logic operations. Also used for storing the result of these operations.
- **BX (Base Register):** Can be used as a base pointer for memory access.
- **CX (Counter):** Primarily used as a loop counter in loop operations.
- **DX (Data Register):** Used for data operations, I/O operations, and also stores results of certain operations.

X in their names stand for extended meaning 16bit registers. For example AX means we are referring to the extended 16 bit “A” register. Its upper and lower byte are separately accessible as AH (A high byte) and AL (A low byte). All general purpose registers can be accessed as one 16bit register or as two 8bit registers.

Index Registers (SI and DI):

- **SI (Source Index):** Often used as a pointer to the source data in memory operations.

- **DI (Destination Index):** Often used as a pointer to the destination data in memory operations.

These registers are commonly used in string operations, where data is moved or manipulated in memory. However, they're not strictly limited to string operations and can be used in other contexts as well. Also, SI and DI are 16bit and cannot be used as 8bit register pairs like AX, BX, CX, and DX.

Instruction Pointer (IP):

- Holds the memory address of the next instruction to be executed. It's automatically updated by the CPU as instructions are executed.

Stack Pointer (SP):

- Points to the top of the stack, which is a region of memory used for temporary storage of data. It's often used for function calls, local variables, and managing program flow.

Base Pointer (BP):

- It is also a memory pointer containing the address in a special area of memory.

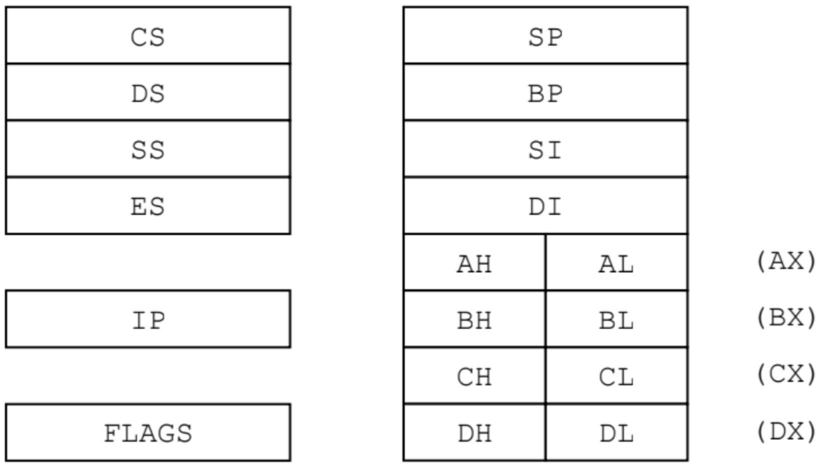
Flags Register:

- Stores various status flags that reflect the outcome of arithmetic and logical operations. Flags like carry, zero, overflow, and more influence program flow and decision-making.

Segment Registers (CS, DS, SS, ES):

- The code segment register, data segment register, stack segment register, and the extra segment register are special registers related to the Intel segmented memory model and will be discussed later.

The iAPX88 architecture consists of 14 registers.



Instruction Groups

Computers use small commands called **"instructions"** to do different tasks. These instructions are like building blocks that the computer follows to get things done. Here are some important things to know:

1. Data Movement Instructions:
These instructions help move information from one place to another inside the computer. This could be from one memory location to another or even between special storage areas. Examples are:

- Move data from one spot to another: `mov ax, bx`
- Load data from a location: `load 1234`

2. Arithmetic and Logic Instructions:
These instructions let the computer do math and logical operations. Math like adding, subtracting, multiplying, and dividing, as well as logical operations like comparing and combining data.

- Compare and do math: `and ax, 1234`
- Add numbers: `add bx, 0534`
- Add numbers from memory: `add bx, [1200]`

3. Program Control Instructions:
Instructions that help control the order in which tasks are done. Sometimes, we want the computer to do different things based on certain conditions.

- Check and change direction: `cmp ax, 0`
- Jump to a different instruction if needed: `jne 1234`

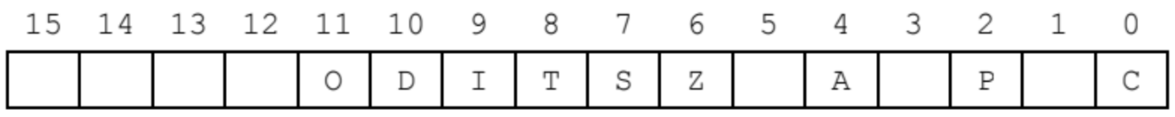
4. Special Instructions:
These are like special tools that make the computer behave differently for certain tasks. They're not used all the time, but they're important.

- Close ears from outside noise: `cli`
- Listen again after work: `sti`

Remember, this is just a simple introduction. As you learn more, you'll understand these concepts better and be able to do more complex things with the iAPX88 architecture. Don't worry if it sounds a bit confusing now – we'll dive into each part more deeply later on!

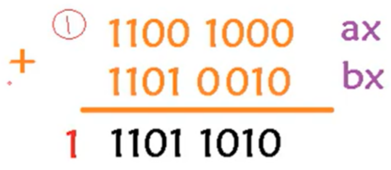
Flag registers

Flag registers are an essential part of computer organization and assembly language programming. They are like tiny status indicators that help the computer keep track of what's happening during calculations and operations.



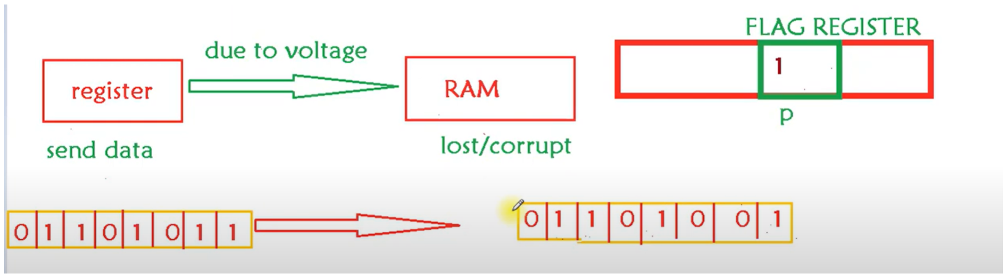
C Carry Flag:

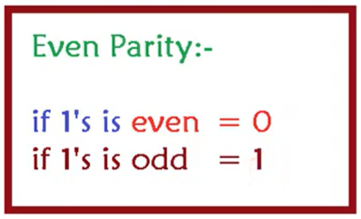
- When you add two numbers, like 16-bit or 8-bit values, the result might be larger than what can fit in the destination register. In such cases, the extra bit that can't fit is stored in the Carry Flag.



P Parity Flag:

- Parity is about counting the number of "one" bits in a binary number. It can either be an odd or even count. Parity checking is often used in communication to ensure data's reliability during transmission.

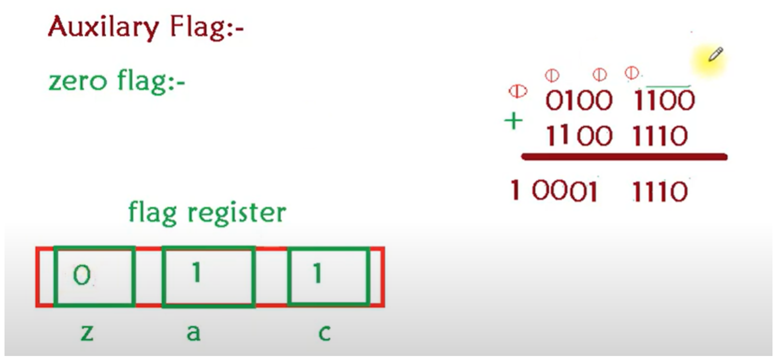



Even Parity:-

if 1's is even = 0
if 1's is odd = 1

A Auxiliary Carry Flag:

- In the world of hexadecimal (base 16) numbers, a 4-bit chunk is called a nibble. When you perform addition or subtraction and there's a carry from one nibble to the next, the Auxiliary Carry Flag gets set. It's like tracking the carry within a number's digits.

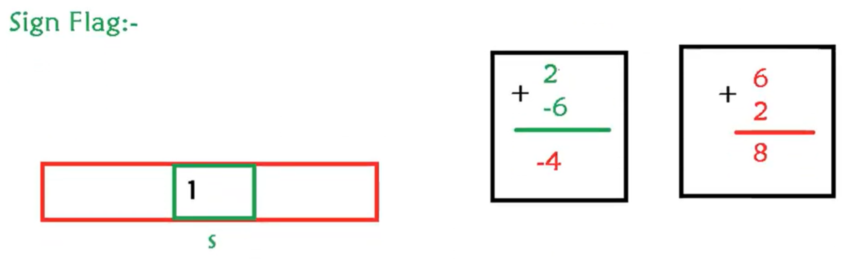


Z Zero Flag:

- The Zero Flag is a signal that tells you if the result of a math or logic operation is zero.

S Sign Flag:

- In computers, signed numbers are represented using two's complement notation. The Sign Flag keeps track of the most significant bit (MSB), which indicates whether a number is positive (MSB = 0) or negative (MSB = 1).

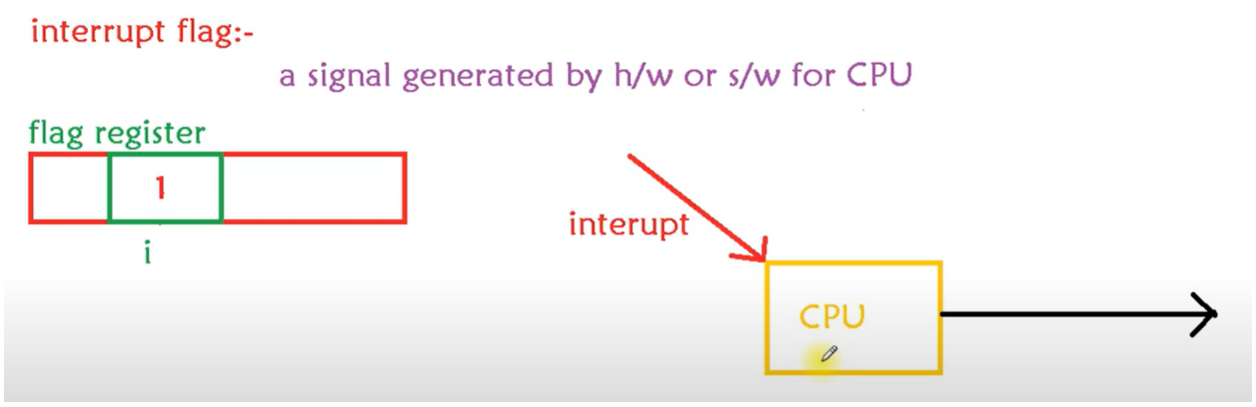


T Trap Flag:

- The Trap Flag is mainly used in debugging, and we'll explore it further later.

I Interrupt Flag:

- This flag controls whether the computer can be interrupted from the outside. Programmers can set or clear the Interrupt Flag to decide if certain tasks should or shouldn't be interrupted. It's like a "Do Not Disturb" sign for the computer.

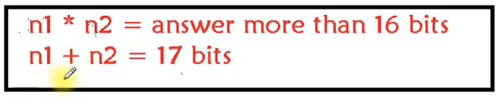


D Direction Flag:

- Specifically related to string operations, the Direction Flag indicates whether the current operation should work from the bottom to the top of a block (D = 0) or from the top to the bottom (D = 1). It's like telling the computer the direction to move data in a list.

O Overflow Flag:

- The Overflow Flag helps you avoid errors when dealing with very large numbers. If an operation generates a result too large to fit in the allotted space, the Overflow Flag lights up to warn you.


n1 * n2 = answer more than 16 bits
n1 + n2 = 17 bits

flag register
1

- Another indication to see the OF will be set is when the MSB of the destination first operand is not same as result.

Ex:

0000 0001 - 1111 1111 = 0000 0010

OF= 0

0000 0001 + 0111 1111 = 1000 0000

OF=1

Our First Program

The first program that we will write will only add three numbers. We will start with writing our algorithm in English and then moving on to convert it into assembly language.

English Language Version

Our first program will be instructions manipulating AX and BX in plain English.

```
move 5 to ax
move 10 to bx
add bx to ax
move 15 to bx
add bx to ax
```

There are some key things to observe. One is the concept of destination as every instruction has a **“to destination”** part and there is a **source** before it as well. For example the second line has a constant **10** as its source and the register **BX** as its destination.

Assembly Language Version

Intel could have made their assembly language exactly identical to our program in plain English but they have abbreviated a lot of symbols to avoid unnecessarily lengthy program when the meaning could be conveyed with less effort. For example Intel has named their move instruction **“mov”** instead of **“move”**. Similarly the Intel order of placing source and destination is opposite to what we have used in our English program, just a change of interpretation. So the Intel way of writing things is:

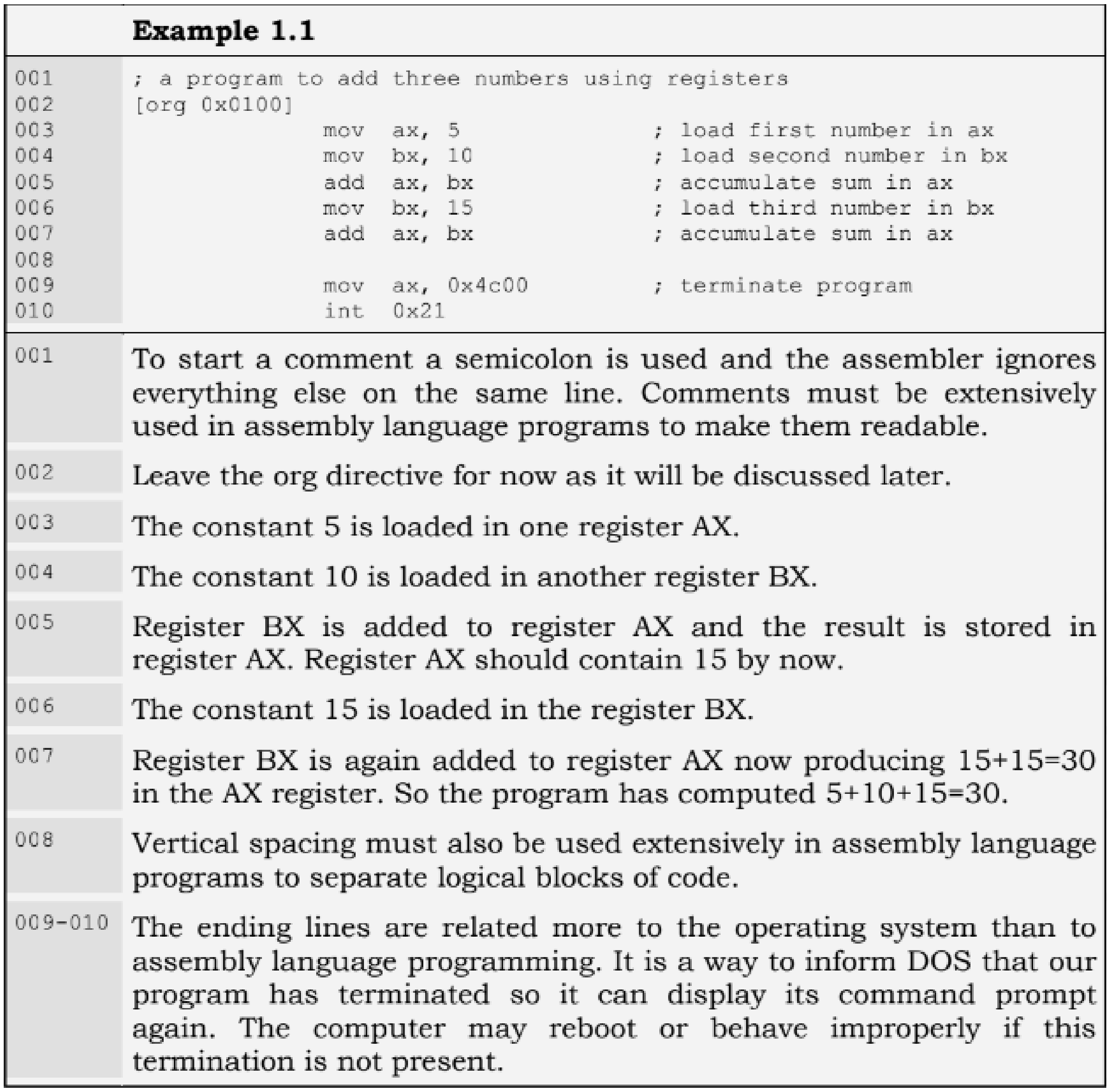
operation destination, source

operation destination

operation source

operation

Now we attempt to write our program in actual assembly language of the iapx88.



Example 1.1

```
001  ; a program to add three numbers using registers
002  [org 0x0100]
003                mov  ax, 5              ; load first number in ax
004                mov  bx, 10             ; load second number in bx
005                add  ax, bx             ; accumulate sum in ax
006                mov  bx, 15             ; load third number in bx
007                add  ax, bx             ; accumulate sum in ax
008
009                mov  ax, 0x4c00         ; terminate program
010                int  0x21
```

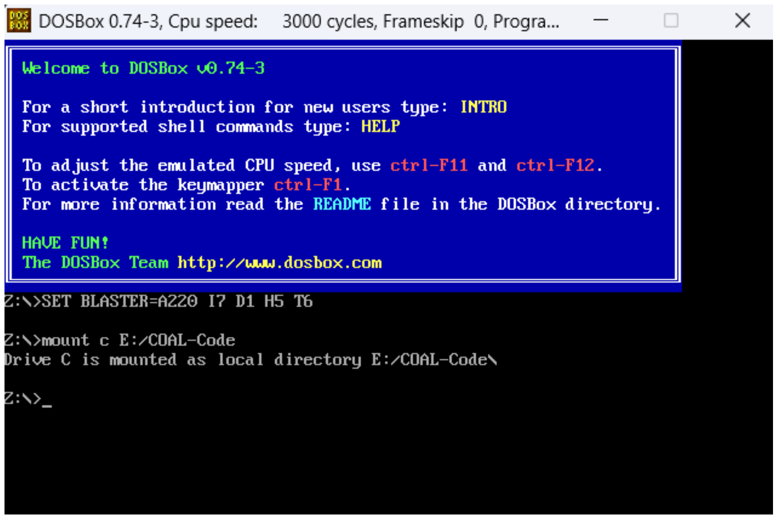
001	To start a comment a semicolon is used and the assembler ignores everything else on the same line. Comments must be extensively used in assembly language programs to make them readable.
002	Leave the org directive for now as it will be discussed later.
003	The constant 5 is loaded in one register AX.
004	The constant 10 is loaded in another register BX.
005	Register BX is added to register AX and the result is stored in register AX. Register AX should contain 15 by now.
006	The constant 15 is loaded in the register BX.
007	Register BX is again added to register AX now producing 15+15=30 in the AX register. So the program has computed 5+10+15=30.
008	Vertical spacing must also be used extensively in assembly language programs to separate logical blocks of code.
009-010	The ending lines are related more to the operating system than to assembly language programming. It is a way to inform DOS that our program has terminated so it can display its command prompt again. The computer may reboot or behave improperly if this termination is not present.

Assembler, Linker, and Debugger

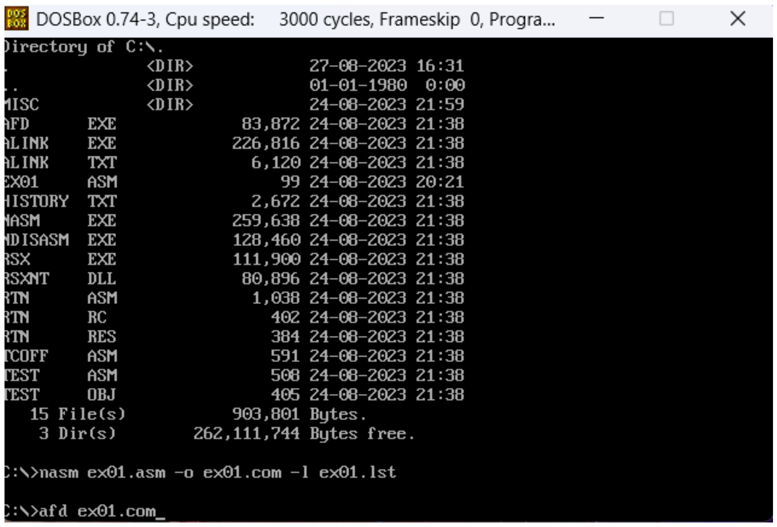
We need an assembler to assemble this program and convert this into executable binary code. The assembler that we will use during this course is **“Netwide Assembler”** or **NASM**. It is a free and open source assembler. And the tool that will be most used will be the debugger. We will use a free debugger called **“A full-screen debugger”** or **AFD**.

To assemble we will give the following command to the processor assuming that our input file is named **EX01.ASM**.

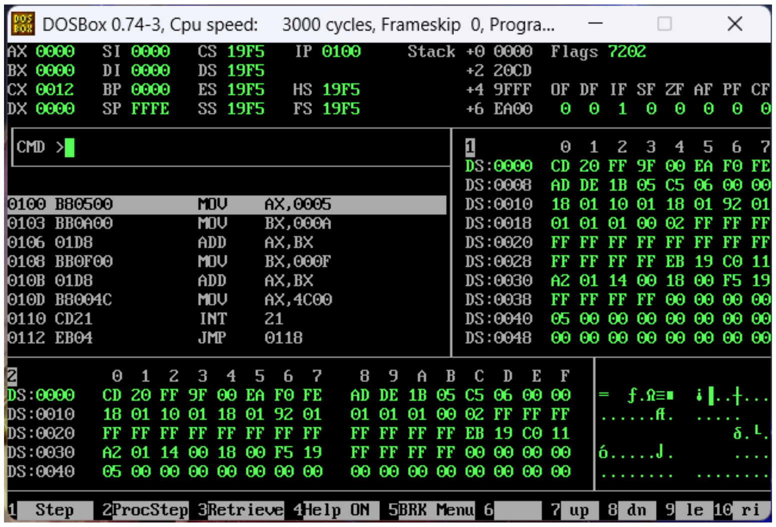
- **nasm ex01.asm –o ex01.com –l ex01.lst**



```
DOSBox 0.74-3, Cpu speed:    3000 cycles, Frameskip  0, Progra...

Welcome to DOSBox v0.74-3

For a short introduction for new users type: INTRO
For supported shell commands type: HELP

To adjust the emulated CPU speed, use ctrl-F11 and ctrl-F12.
To activate the keymapper ctrl-F1.
For more information read the README file in the DOSBox directory.

HAVE FUN!
The DOSBox Team http://www.dosbox.com
Z:\>SET BLASTER=A220 I7 D1 H5 T6

Z:\>mount c E:/COAL-Code
Drive C is mounted as local directory E:/COAL-Code\

Z:\>_
```



```
DOSBox 0.74-3, Cpu speed:    3000 cycles, Frameskip  0, Progra...
Directory of C:\.
.              <DIR>            27-08-2023 16:31
..             <DIR>            01-01-1980  0:00
MISC           <DIR>            24-08-2023 21:59
AFD      EXE               83,872 24-08-2023 21:38
ALINK    EXE              226,816 24-08-2023 21:38
ALINK    TXT                6,120 24-08-2023 21:38
EX01     ASM                   99 24-08-2023 20:21
HISTORY  TXT                2,672 24-08-2023 21:38
NASM     EXE              259,638 24-08-2023 21:38
NDISASM  EXE              128,460 24-08-2023 21:38
RSX      EXE              111,900 24-08-2023 21:38
RSXNT    DLL               80,896 24-08-2023 21:38
RTN      ASM                1,038 24-08-2023 21:38
RTN      RC                   402 24-08-2023 21:38
RTN      RES                  384 24-08-2023 21:38
TCOFF    ASM                  591 24-08-2023 21:38
TEST     ASM                  508 24-08-2023 21:38
TEST     OBJ                  405 24-08-2023 21:38
   15 File(s)            903,801 Bytes.
    3 Dir(s)         262,111,744 Bytes free.

C:\>nasm ex01.asm -o ex01.com -l ex01.lst

C:\>afd ex01.com_
```


DOSBox 0.74-3, Cpu speed: 3000 cycles, Frameskip 0, Progra...

```
AX 0000   SI 0000   CS 19F5   IP 0100     Stack +0 0000  Flags 7202
BX 0000   DI 0000   DS 19F5                     +2 20CD
CX 0012   BP 0000   ES 19F5   HS 19F5           +4 9FFF   OF DF IF SF ZF AF PF CF
DX 0000   SP FFFE   SS 19F5   FS 19F5           +6 EA00    0  0  1  0  0  0  0  0

 CMD >                                          1          0  1  2  3  4  5  6  7
                                                DS:0000  CD 20 FF 9F 00 EA F0 FE
                                                DS:0008  AD DE 1B 05 C5 06 00 00
0100 B80500         MOV     AX,0005             DS:0010  18 01 10 01 18 01 92 01
0103 BB0A00         MOV     BX,000A             DS:0018  01 01 01 00 02 FF FF FF
0106 01D8           ADD     AX,BX               DS:0020  FF FF FF FF FF FF FF FF
0108 BB0F00         MOV     BX,000F             DS:0028  FF FF FF FF EB 19 C0 11
010B 01D8           ADD     AX,BX               DS:0030  A2 01 14 00 18 00 F5 19
010D B8004C         MOV     AX,4C00             DS:0038  FF FF FF FF 00 00 00 00
0110 CD21           INT     21                  DS:0040  05 00 00 00 00 00 00 00
0112 EB04           JMP     0118                DS:0048  00 00 00 00 00 00 00 00

2              0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F
DS:0000       CD 20 FF 9F 00 EA F0 FE  AD DE 1B 05 C5 06 00 00  =  ƒ.Ω≡■  ¡▌..┼...
DS:0010       18 01 10 01 18 01 92 01  01 01 01 00 02 FF FF FF  ......Æ.  .....
DS:0020       FF FF FF FF FF FF FF FF  FF FF FF FF EB 19 C0 11              δ.└.
DS:0030       A2 01 14 00 18 00 F5 19  FF FF FF FF 00 00 00 00  ó.....⌡.      ....
DS:0040       05 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ........  ........

1 Step  2ProcStep 3Retrieve 4Help ON  5BRK Menu 6       7 up  8 dn  9 le 10 ri
```

This will produce two files **EX01.COM** that is our executable file and **EX01.LST** that is a special listing file that we will explore now.

```

                                      [org 0x0100]
   00000000 B80500                           mov   ax, 5
   00000003 BB0A00                           mov   bx, 10
   00000006 01D8                             add   ax, bx
   00000008 BB0F00                           mov   bx, 15
   0000000B 01D8                             add   ax, bx

   0000000D B8004C                           mov   ax, 0x4c00
   00000010 CD21                             int   0x21
```

- The listing file shows the program's code and how it's translated into machine code. Each line has:
  - An **"offset"**, which is where the instruction is in the program.
  - An **"opcode"**, a code for the instruction.
  - Numbers and bytes are stored in a specific order, called **"endian order"**. Some computers use **big-endian** (most significant first), but Intel uses **little-endian** (least significant first). For example, if we move the number 5 to the AX register, it becomes opcode B8 with 0500 as the data. This means the least significant byte (05) comes first, and when read as a word, it's 0005. The number 5 was turned into 0500. This is because numbers are stored in groups of 4 bits (hexadecimal digits), and 4 of them make a word (2 bytes).
- Instructions are made of bytes. The length of the first instruction is three bytes, so the next instruction is at offset 3. The "opcode" BB moves a number into the BX register, with 0A00 as the data (10 in little-endian).
- The last instruction is at offset 16, and the whole program is **18 bytes**.
- When we load the program in the debugger, the first instruction is at offset 0100 due to the **"org"** directive. This is a requirement for COM files.

- During program execution, registers or memory can change. Our program only changes registers like AX and BX. Watch how AX, BX, and IP change after each instruction. IP points to the next instruction, and AX stores the result of addition.

`[org 0x0100]`:

- This tells the computer where the program should start running.
- `org` stands for "origin," and `0x0100` is the memory address where the program begins.
- It's like setting the starting point on a map before you begin a journey.

`mov ax, 0x4c00`:

- `0x4c00` is a number in hexadecimal (base-16) notation. It's like saying "put the number 4C00 in the `ax` box."
- This instruction is asking the computer to load a specific value into the `ax` storage space.

`int 0x21`:

- `int` means "interrupt". It's like raising your hand to get the teacher's attention in class.
- `0x21` is a special number that tells the computer what type of service we need.
- In this case, it's like asking the computer to do something for us using a specific service.
- It's often used for things like input and output, which let us communicate with the computer.

Memory Models and Addresses

Linear Memory Model (Used in 8080 and 8085):

In the linear memory model, the entire memory is treated as a single continuous array of data. This means that each memory location has a unique address that can be represented by a single value. For old processors like the 8080 and 8085, they could access a total of **64K** (64 kilobytes) of memory using the 16 lines of their address bus.

Segmented Memory Model (Introduced in 8088):

With the advent of the 8088 microprocessor, Intel wanted to maintain compatibility with the 8080 and 8085 while also addressing the limitation of the 64K memory size. To achieve this, they introduced the segmented memory model.

Concept of Segmented Memory Model:
In the segmented memory model, memory is divided into distinct segments. These segments can be thought of as separate functional windows into the main memory. The processor can see different parts of memory through these windows, such as a code window, a data window, and more. The size of one window is restricted to 64K. 8085 software fits in just one such window.

However the maximum memory **iAPX88** can access is **1MB** which can be accessed with **20 bits**. Compare this with the 64K of 8085 that were accessed using **16** bits. The idea is that the 64K window just discussed can be moved anywhere in the whole 1MB.

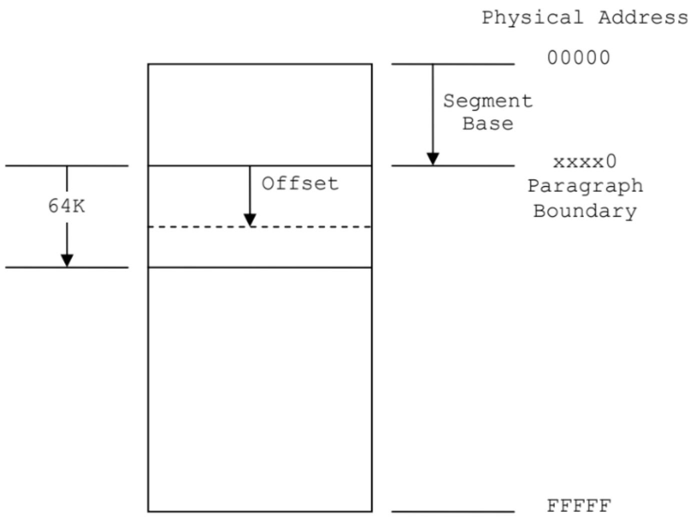
Segment Registers:
Intel introduced four segment registers: CS (Code Segment), DS (Data Segment), SS (Stack Segment), and ES (Extra Segment). These registers define the starting point (base) of each segment window. Think of it like the reference point on a graph – the floor is the base, and measurements are made from that base.

- CS: Holds the base of the code segment.
- DS: Holds the base of the data segment.
- SS: Holds the base of the stack segment.
- ES: Used as an extra data segment and has special roles in certain instructions.

Segmented Address Calculation:
To access memory, a segmented address is used, which consists of two parts: a segment value and an offset value. The segment value is stored in the segment register, and the offset is added to it. This allows you to access different parts of memory relative to the segment base.

IP Register and Window Movement:
The IP (Instruction Pointer) register works together with the CS register. The CS register determines the base of the segment window (64K), and the IP register selects specific code offsets within this window. IP can only work within the current window defined by CS.

Maximum Memory Access:
The 8088 processor can access up to 1MB of memory using 20 bits. The idea is that the 64K window can be moved anywhere within this 1MB range. The four segment registers allow you to set up and move these windows.



Physical Addresses and Logical Addresses:

A **logical address** refers to an address that is used within a program, and it typically consists of two parts: a **segment** and an **offset**. The segment specifies a segment of memory, and the offset specifies a location within that segment.

A **physical address** is the actual address that corresponds to a specific location in the physical memory of the computer. In a system with a segmented memory model like the Intel 8088, a logical address is used by the CPU to access data, and this logical address is then converted into a physical address for memory access.

Calculating Physical Addresses Using Segment-Offset Pairs:

In the Intel 8088 architecture, the segment value and the offset value are used to calculate the physical address for memory access. Both the CS (Code Segment) and IP (Instruction Pointer) registers are 16-bit, but the total addressable memory is 1MB, which requires 20 bits. To accommodate this, a mechanism is used to combine the segment value and the offset value to create a 20-bit physical address.

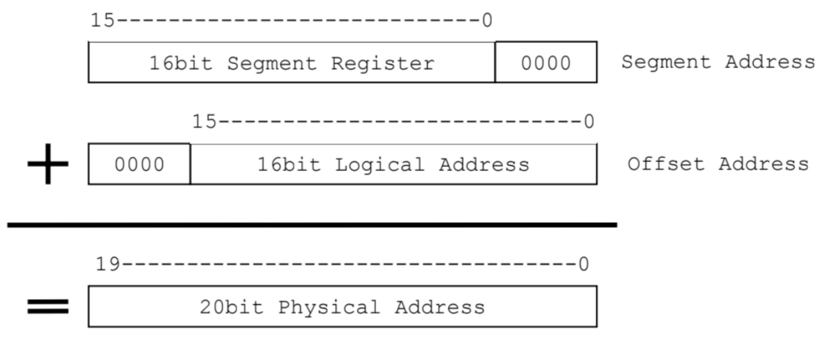
Here's how it works:

1. The segment value is treated as a 20-bit number by appending four zero bits at the lower end.
2. The offset value is also treated as a 20-bit number by appending four zero bits at the upper end.
3. These two 20-bit numbers (segment value and offset value) are then added together.
4. If a carry is generated during the addition, it is dropped without being stored anywhere. This is known as **address wraparound**. Hence, memory is like a circle, if you start from 0000 an keep on going to next byte by adding 1 then on reaching FFFF adding 1 will take you again to 0000.

This process ensures that the logical address is translated into a physical address within the 1MB memory space.

Example:

- Let's say the CS register holds the segment value 1DDD (where D represents any digit), and the IP register holds the offset value 0100.
- The 20-bit segment value becomes 1DDD0, and the 20-bit offset value becomes 00100.
- Adding these two values (1DDD0 + 00100) results in the physical memory address 1DED0.



Overlapping Segments and Logical Addresses:

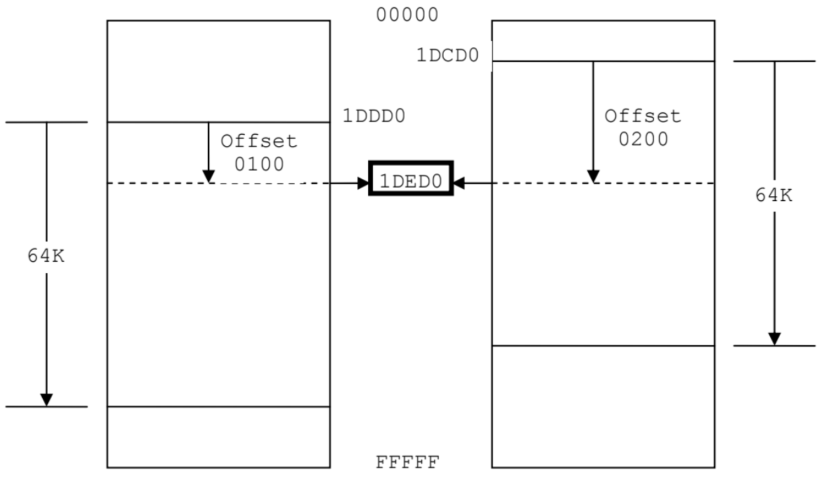
In certain cases, the same memory location can be accessed through different segments. When segments have the same value in them, it's referred to as **overlapping segments**. Overlapping segments allow you to see the same memory content from different functional windows.

For example, in the case of a COM file structure, where the CS (Code Segment), DS (Data Segment), SS (Stack Segment), and ES (Extra Segment) all have the same value, overlapping segments occur. This kind of arrangement is common in COM files and allows different parts of the program to access the same memory locations without having to change segment values.

By using overlapping segments, you can create multiple **segment-offset pairs** that all point to the same physical memory location. For instance, let's consider the following examples:

1. Logical address 1DDD:0100
2. Logical address 1DED:0000
3. Logical address IDCD:0200

All three logical addresses correspond to the same physical memory location. Even though these addresses are expressed differently, they access the same memory content.



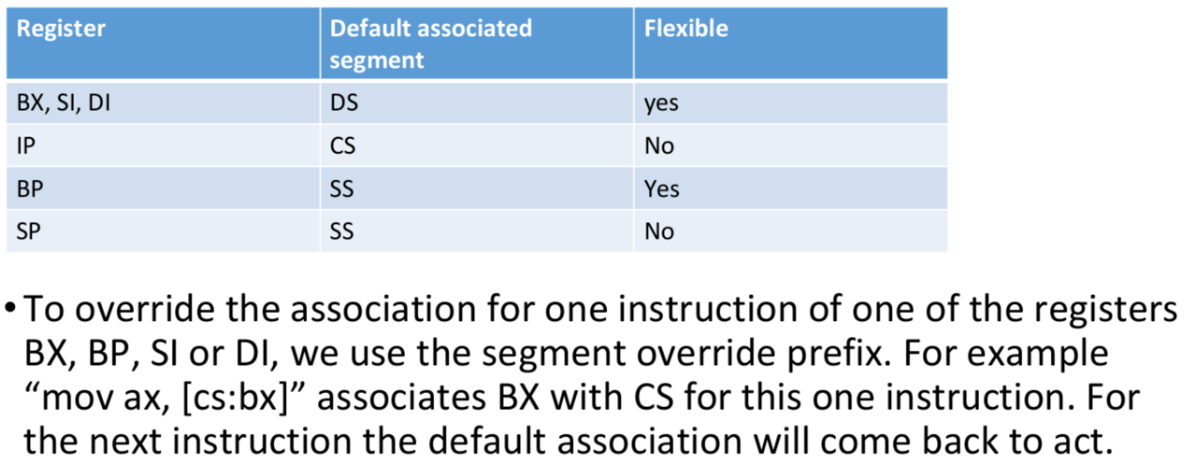
Segment Association:

In the Intel 8088 architecture, registers that access memory are associated with specific segments. This association helps define the base of the memory window from which these registers calculate physical addresses. Each register has a default segment associated with it, which determines the base of its memory window.

Default Segment Associations:

1. **CS (Code Segment) and IP (Instruction Pointer):**
   - CS is associated with IP by default. This means that the segment base of CS serves as the base for calculating the physical address of the instruction pointed to by IP. The association between CS and IP is unchangeable.
2. **DS (Data Segment) and BX (Base Register):**
   - DS is associated with BX by default. BX can be used as an offset register for memory access, and its association with DS allows it to work within the memory window defined by DS. The association between them is changeable.

The list mentioned likely provides a comprehensive overview of all the default segment associations for different registers.



Register	Default associated segment	Flexible
BX, SI, DI	DS	yes
IP	CS	No
BP	SS	Yes
SP	SS	No

- To override the association for one instruction of one of the registers BX, BP, SI or DI, we use the segment override prefix. For example “mov ax, [cs:bx]” associates BX with CS for this one instruction. For the next instruction the default association will come back to act.