

CG2007/EE2007E Microprocessor Systems

Microprocessor (MuP) Age

First MuP - Intel **4004**; 4-bit processor; a programmable controller on a chip; Just 45 instructions; P-channel MOSFET technology
50KIPs[1971]

In 1972, 8-bit processor was released; About the same kind of limitations as the 4-bit predecessor. Consequently, they could not serve as general purpose MuPs.

- 1974 – Intel’s first general purpose 8-bit MuP **8080** was released
- Subsequently, **8085** came out with more features – kind of a functionally complete MuP!

Main limitations of the 8-bit MuPs were

Low speed of execution

Low memory addressing capabilities

Limited number of general purpose registers (GPRs)

Less powerful instruction set

About six months after Intel released 8080 MuP, Motorola corporation released its first MuP **MC6800**

From then onwards, competition started . . .

Manufacturer

MuP

Fairchild

F-8

Intel

8080

MOS Technology

6502

Motorola

MC6800

National Semiconductor

IMP-8

Rockwell Int'l

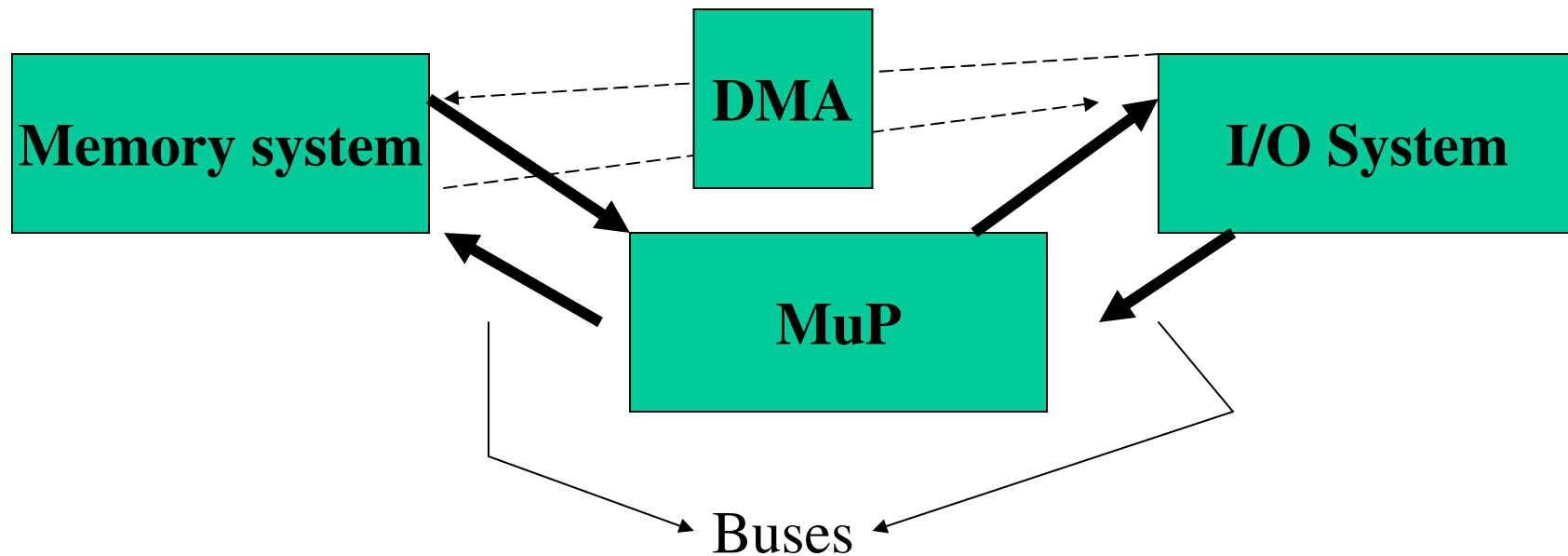
PPS-8

Zilog

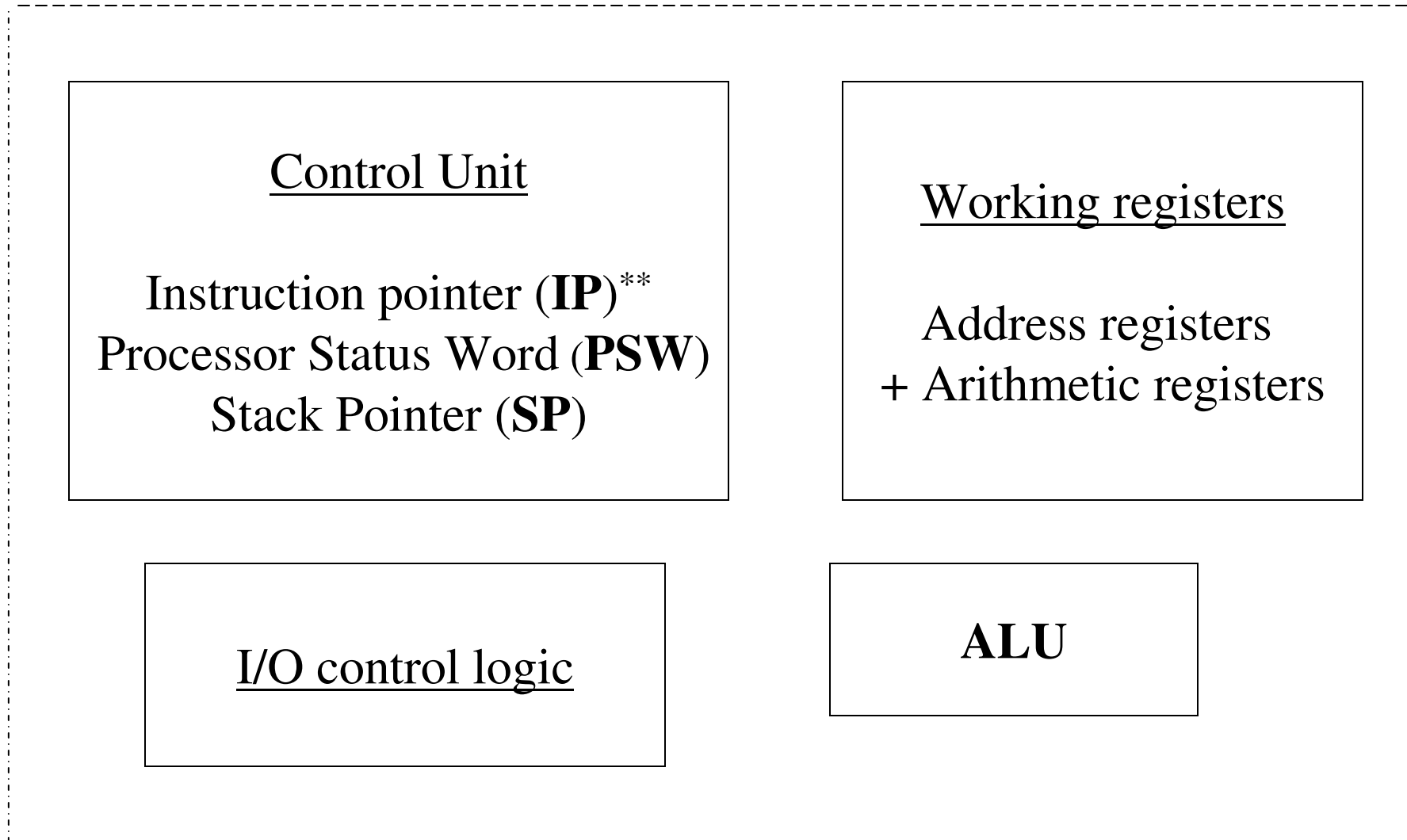
Z-8

MuP based Personal Computer System

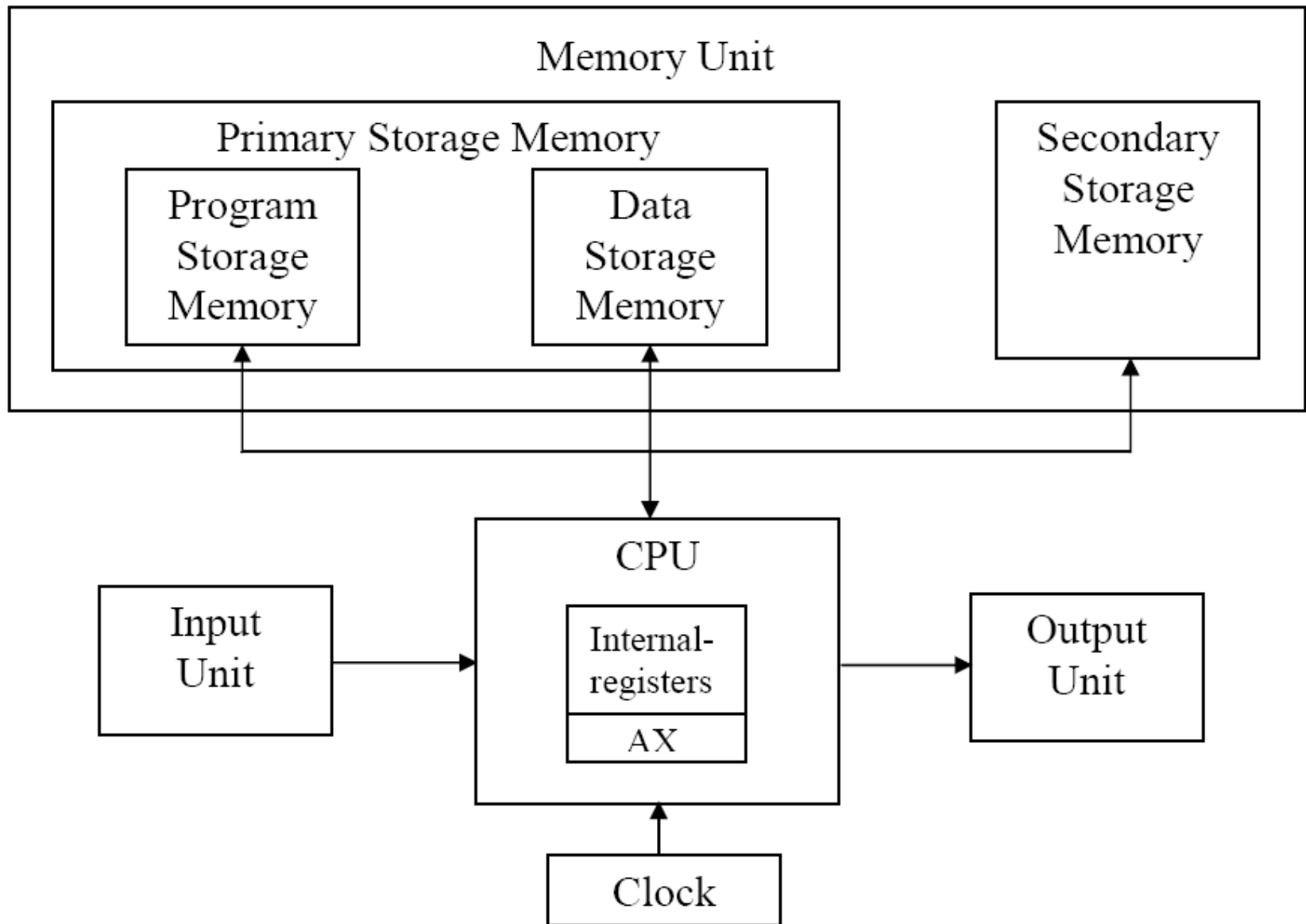
A block diagram of a PC can be as follows



Typical Architecture of a CPU



** IP is otherwise called as
Program counter (PC)



High-level Program

“Close to the
User/Application”

Object code

Until here time consumption may be
high; not preferable for certain
Time-and-Mission critical applications;

Assembly code

“Close to the
Actual hardware”

Executable (literally 0s and 1s as machine
binary understands)

High-level
Program



Set of “Instructions”

No 1-1 translation

1 line of HL language may translate to few instructions!!

For the time being, understand this....

Instruction captures the required actions to be performed, it carries the data to be processed and have specific formats; CPU understands the actions and datum via a “pre-coded” dictionary – the rules of an Instruction set designed by the CPU manufacturers.

Internal workings of a CPU

The function of a CPU is to execute information stored in memory. The CPU is connected to memory and I/O through strips of wires called as BUS. Bus is of three types – address bus, data bus, and control bus.

For a device to be recognized by CPU, the device must have an address. The address assigned must be unique. CPU puts the address on the address bus, and the decoding circuitry finds the device. Then the CPU uses data bus either to get or to send data to the device. The control buses are used to provide read/write signals to the device if the CPU is asking for info or sending the info. Address and Data busses determine the capability of a given CPU.

Data bus – Used to carry info in and out of CPU; the more the data buses available better will be the performance. A MuP is of type K-bit means that this MuP could work on only K-bits of data at a time. On this MuP, data larger than K-bits must be broken down into smaller data and then be processed. Thus,

8080/8085: 8-bit processor

8086: 16-bit processor

All the registers inside our K-bit MuP are of width K-bits, in our above example.

Address bus – If there are x lines on address bus, this means that we can access up to a maximum of 2^x memory byte locations. For example, if address bus is 16 bits wide then this means $2^{16} = 65,536 \approx 64\text{K}$ bytes of addressable locations. Each location can have a maximum of 1 byte data.

As another example, IBM PC AT uses 24 address lines and 16 data lines. In this case, the total accessible memory is $2^{24} = 16$ megabytes. Since there would be 2^{24} locations, and since each location is one byte, there would be 16 megabytes of memory

Inside CPUs – A program stored in memory provides instructions to the CPU to perform an action. This action could be simply adding data. It is the function of the CPU to *fetch* these instructions from memory and execute them. For CPU to process any information data must be available in RAM. To perform the actions of fetch and execute, all CPUs are equipped with resources such as following:

1. Registers - 8-bit, 16-bit, 32-bit, etc
2. ALU (arithmetic/logic unit) – responsible for performing arithmetic functions and logical operations (AND, OR, and NOT)
3. An Instruction Pointer (IP) – this pointer/counter points to the next instruction to be executed. As each instruction is executed, IP is incremented automatically to point to the address of the next instruction.

It is the contents of the IP that are placed on address bus to find and fetch the desired instruction.

4. Function of the instruction decoder is to interpret the instruction fetched into the CPU. We can think of the instruction decoder as a kind of dictionary, storing the meaning of each instruction and what steps CPU must take upon receiving that instruction.

Let us be specific on 8086 from now . . .

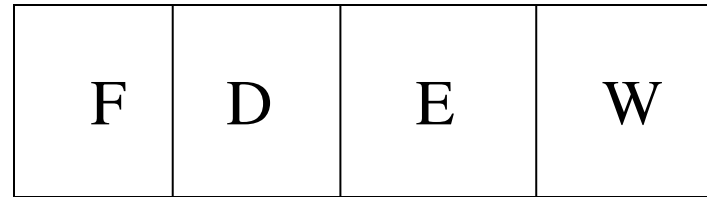
Inside 8086/8088

Two ways to make CPU faster to work

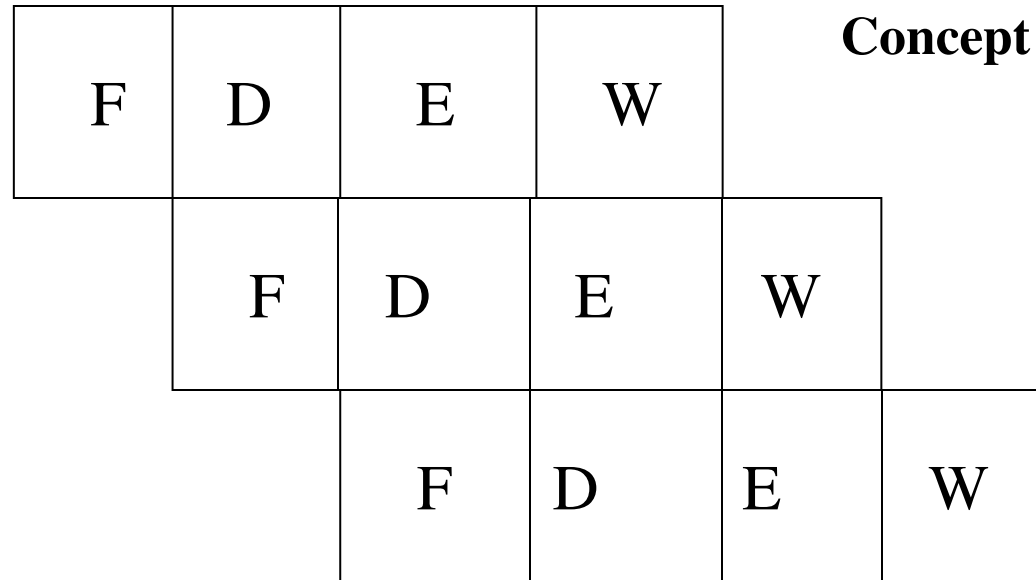
- (A) Increase the working frequency [8086 – 5MHz to 10MHz]
- (B) Planning and Using the internal architecture

- (A) Technology dependent – technology used in fabricating the ICs, # of transistors that can be packed per chip area, etc
- (B) Careful planning; Optimal utilization of available resources; design methodologies [example – pipelining]

8085 processor



80x86 processor



Concept of pipelining

F - Fetch
D - Decode
E - Execute
W - Write

Time in clock cycles →

Memory subsystem: It is an essential part of a computer to store initial and processed information. Typically microcomputers have two types of memories; **primary and secondary**.

Primary memory is normally smaller in size and temporarily stores active information (*such as the operating system (windows), programs that are currently being run, data that are currently being processed by computer etc*)

Secondary memory is normally large and used for long-term storage of information that is not currently being used by the PC. Typically this memory represents the external storage media and hard disks - *magnetic tapes like, Floppy disks (!!!), Zip disks, Optical media like, Read-only Compact Disk (CD), Read/write Compact Disk (CD-RW), Digital Video Disk (DVD)*

- Typically primary storage is implemented using CPU-registers, Caches, Random access memories (RAM), Read only memories (ROM) etc. *(Except ROM, information stored in primary memory is erased when the computer is switched off)*

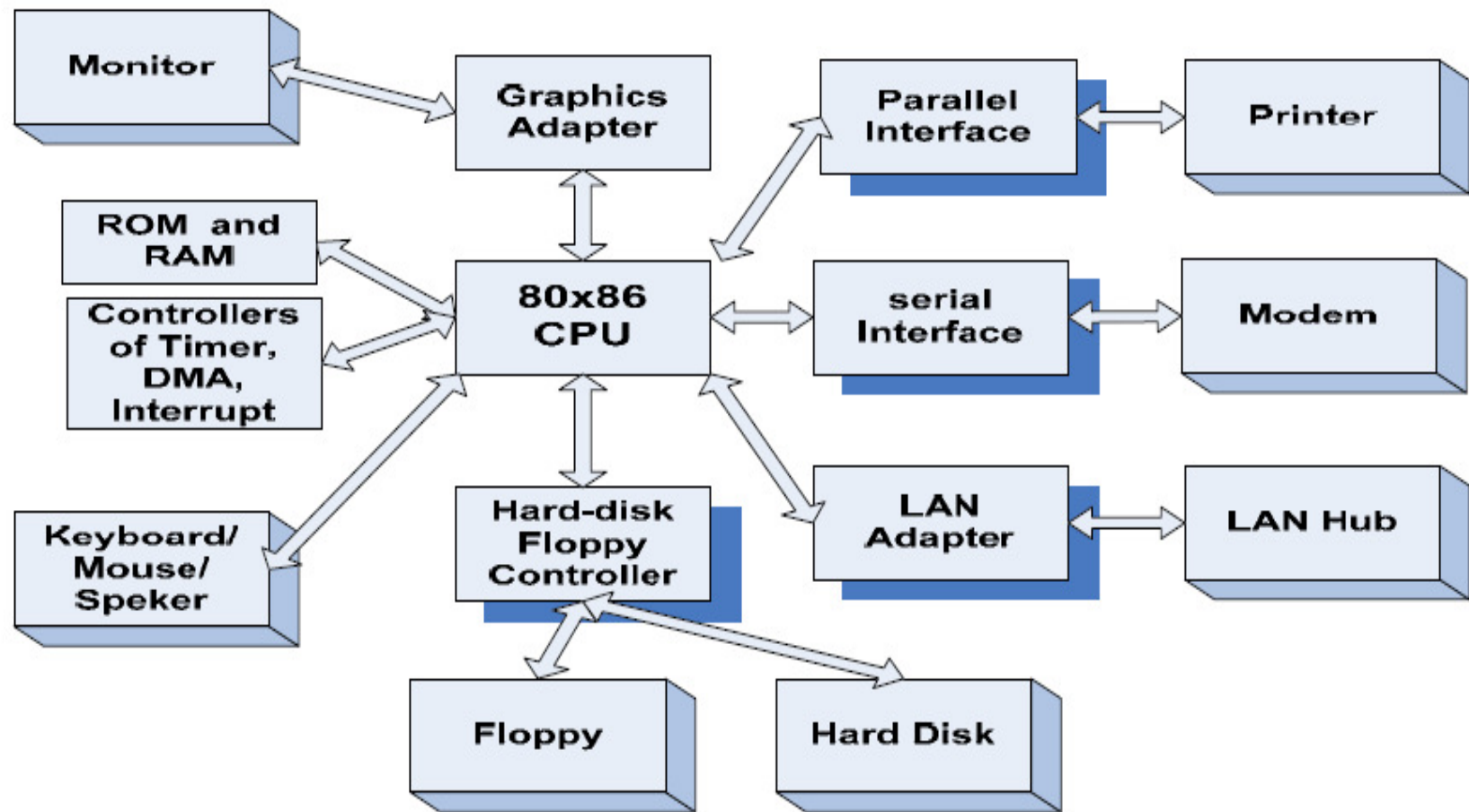
CPU Registers: Fastest storage elements, where frequently used **temporary data is stored** and accessed by specifying the register name (not addresses). Although general purpose registers can be accessed by the programmer, special purpose registers are only used by the computer.

Random-Access Memory (RAM): Usually **called the main memory of PC**, where addressed memory locations are used to temporarily store (write) information which can be accessed (read) in any order at equal time periods (hence the name random access). RAM is typically implemented using IC's, often called memory sticks (such as DIP 16-pin, SIPP, SIMM 30-pin, SIMM 72-pin, SDRAM DIMM, DDR DIMM)

Read-Only-Memory (ROM): stores information permanently and also can randomly access the stored information. ROM is usually used to store the information required by the computer to start-up or initiate the booting sequence. Recent types of ROMs that are used in micro-computers are also called FLASH or CMOS

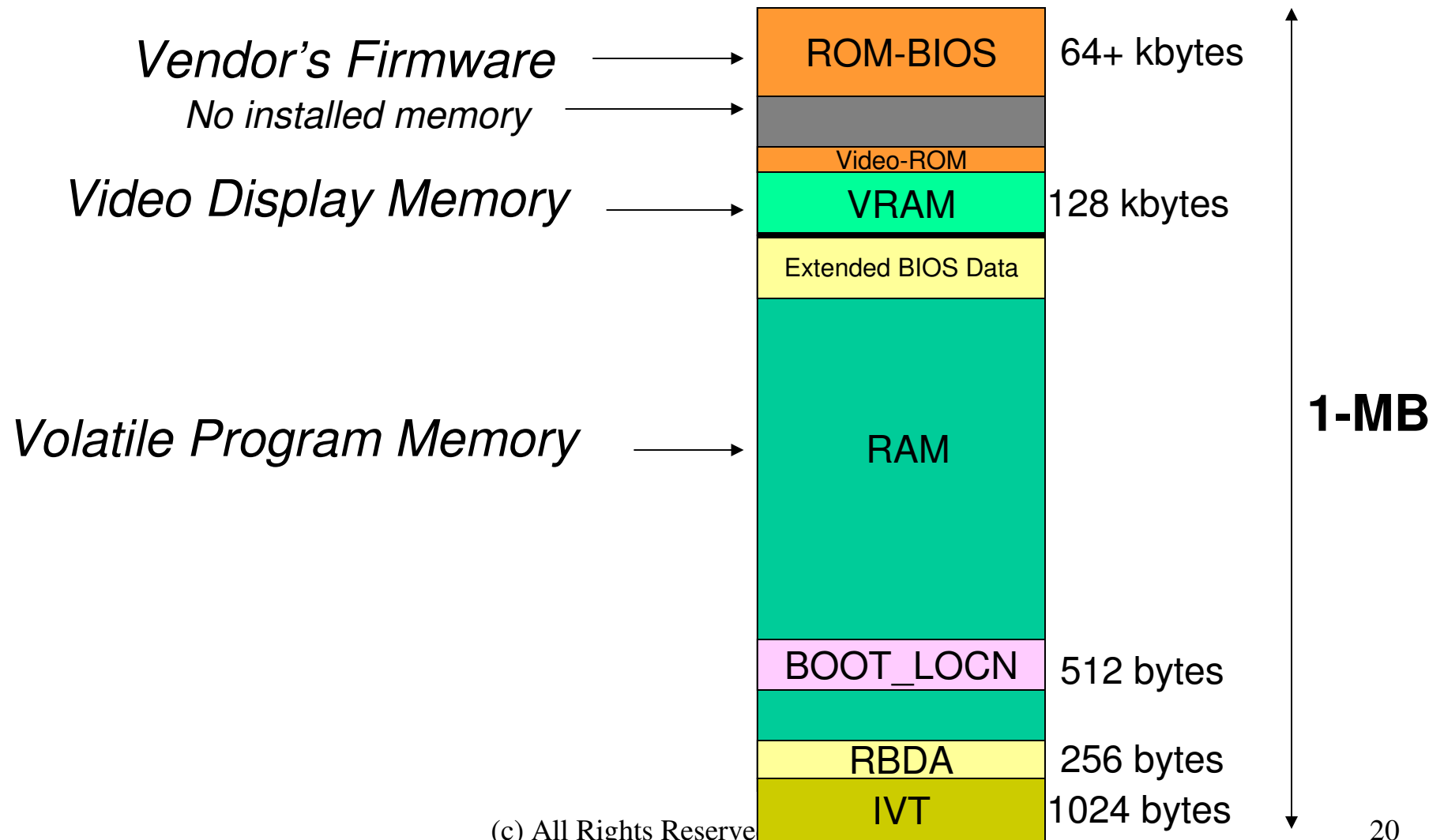
Cache: is a very fast type of RAM that is used to store information that is most frequently or recently used by the computer. Recent computers have 2-levels of cache; the first level is faster but smaller in size (usually called internal cache), and the second level is slower but larger in size (external cache)

Your PC has.....

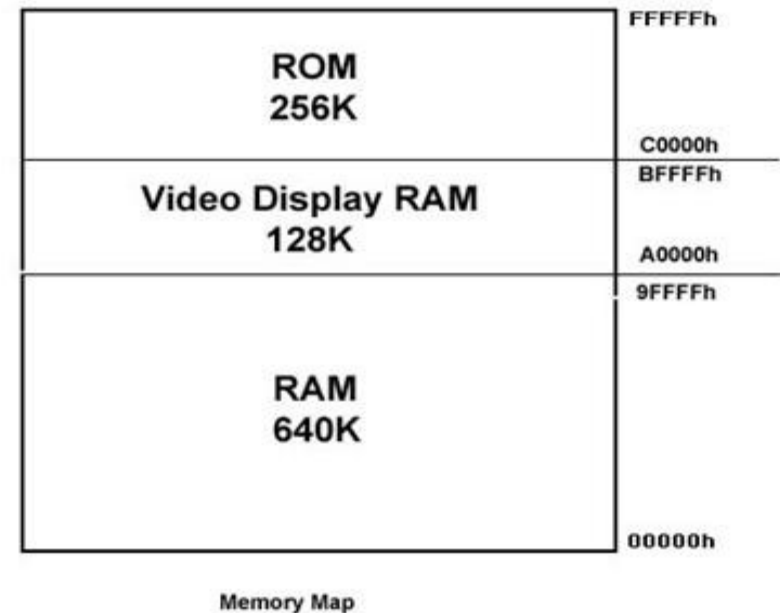


Memory Map of IBM PC

'Real-Mode' Memory Map



- It is a guide showing how the system memory is allocated
- 16 64K-byte blocks
- RAM is called as Conventional Memory
- Video RAM – 128K, set aside for Video
- ROM – First 64K for BIOS ROM
 - Eg: POST – Powe On Self Test
- Remaining ROM space for adapter cards



Memory map of IBM PC – RAM 640K (00000 H – 9FFFF H); Video Display 128K (A0000 H – BFFFF H); ROM 256K (C0000 H – FFFFF H);

The task of managing the RAM is left to the DOS as different versions of DOS consume different amounts of memory and different PCs may have different RAM sizes. Plus, memory needs of applications differ *(This is the reason why we do not assign any values to CS, DS and SS as it is beyond the knowledge of the user to take control)* This RAM space is called *Conventional Memory*.

ROM – Of this 256K only 64K is used by Basic Input-Output System (BIOS) programs/routines; BIOS routines contain programs to test RAM and other components connected to the CPU. It also contains programs that allow DOS to communicate with peripheral devices such as keyboards, mouse, printer, and disk. It is the function of the BIOS to test all the devices connected to PC when the computer is turned on and report any errors. It is only after setting up the peripherals that BIOS will load DOS from disk into RAM and hand over the control of the PC to DOS. The remaining space (256K-64K) is used by various adapter cards (such as cards for hard disks) and the rest is usually kept free.

Some facts:

8086 can handle 1 megabytes of memory

8085 can handle a maximum of 64K

8086 – 16-bit processor

8080/8085 – 8-bit processor

8086 – pipelined processor; first step towards “parallel processing”

8080/8085 – non-pipelined processor

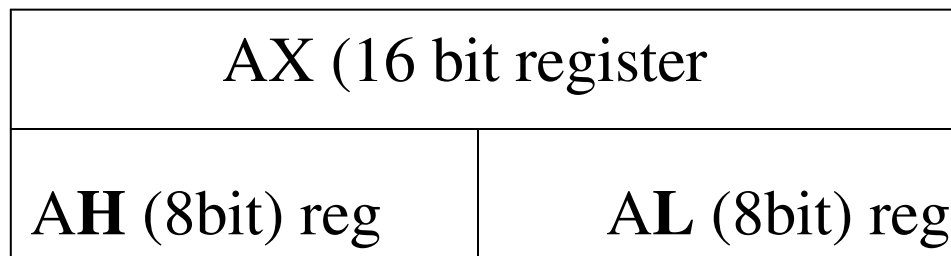
8088: Identical to 8086, however it has 8-bit data bus. 8086 has 16-bit data bus which means all the registers internally are 16-bit wide. But, in those days peripherals were built around 8-bit MuPs. Plus, PCB with 16-bit data bus was expensive in those days => Intel came out with 8088!

Address bus of 8086 is of 20 bits wide and hence the amount of physical memory that can be spanned is 2^{20} locations = 1Megabyte. These are from 00000H to FFFFFH, respectively.

Let us see the architecture first.

<u>General purpose registers</u> (1)	<u>Segment registers</u> (2)	<u>Pointers and index registers</u> (3)
AX, BX [AH,AL,BH,BL] CX,DX [CH,CL,DH,DL]	CS, SS, DS,ES	SP, BP SI, DI IP
+ <u>FLAGS/PSW</u> (4)		

GPRs are used to store the information temporarily. That info could be either 1 byte or 2 bytes of data to be processed. The GPRs can be accessed as either 8 bit or 16 bits



D15-D8

D7-D0

Lower order byte

Higher order byte

NOTE: All other registers can be accessed only as full 16-bits

Different registers are used for different purposes since some instructions use specific registers to perform certain tasks.

How to remember? The first letter of each register tells its purpose

AX – **a**ccumulator

BX – **b**ase addressing register

CX – **c**ounter by default for certain string manipulations

DX – (used to point to) **d**ata in I/O operations

Segment registers – What are these?

8086 addresses segmented memory. We have 1 Megabyte of physical memory on the whole and this entire memory can be accessed by 8086, as it has 20 address bits.

This 1MByte of space is divided evenly into 16 logical segments. Each segment, therefore, has

$$(2^{20} / 2^4) = 2^{16} = 2^6 \cdot 2^{10} = 64\text{Kbytes} \Rightarrow \text{each segment's length}$$

Why this segmented structure in 8086?

The answer is due to traditional reasons: 64K comes from 8085's architecture and has been carried over to the design of 8086 too.

For this reason, 8088/86 can handle a maximum of 64K bytes of code, 64K bytes of data, and 64K bytes of stack at any time, although it has a range of 1Megabyte space

Logical and physical address

Three types of address exist in Intel's literature – physical address, Offset address, and the logical address

Physical address → 1Meg ⇒ 00000H to FFFFFH (86 to 2/3/486 processors
(*Effective address*) in real-mode)

{ **Offset** – is a location within a 64K segment range ⇒ 0000H to FFFFH
Logical address consists of a (segment value + an offset address)

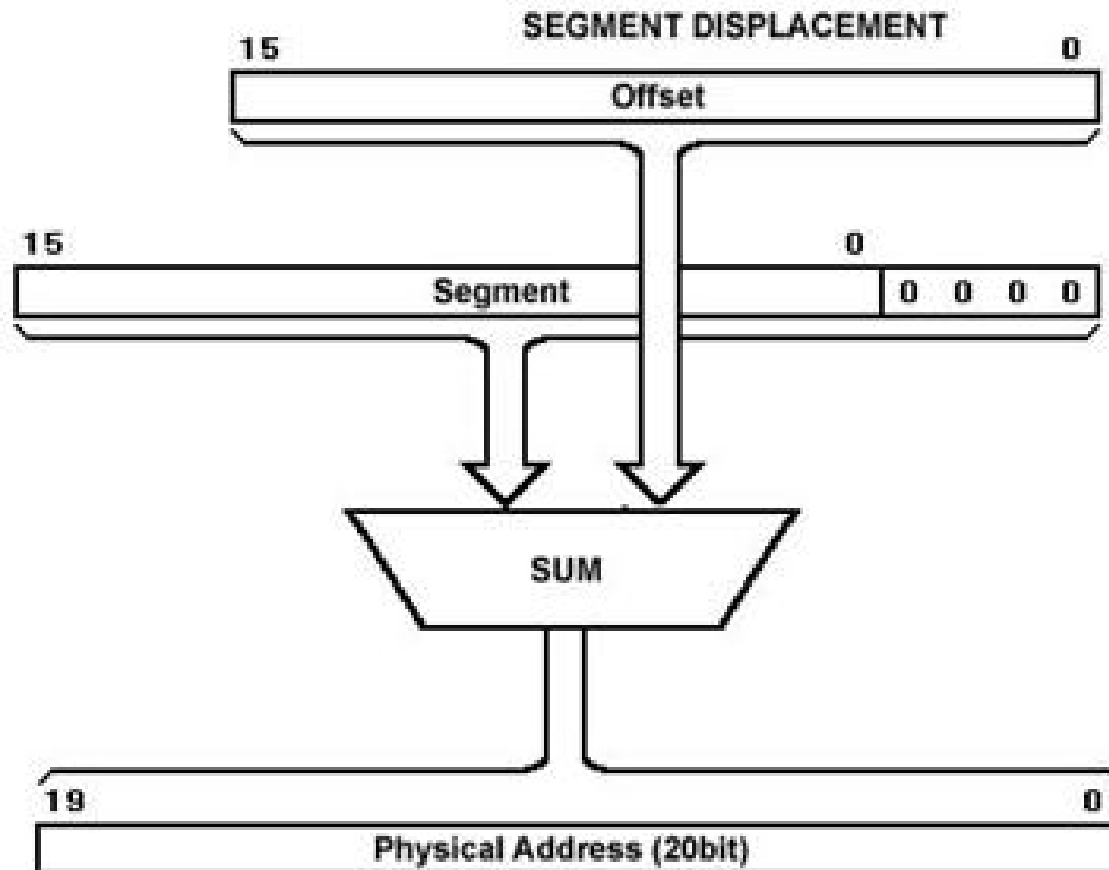
So when the loader loads the segments to the memory it makes sure that the *least significant nibble starts with a '0'*; Thus, if **CS = 1503** then, actual physical address from where this CS segment starts is **15030H** (first location) and **15030 + FFFF** will be the address of the last location in this segment.

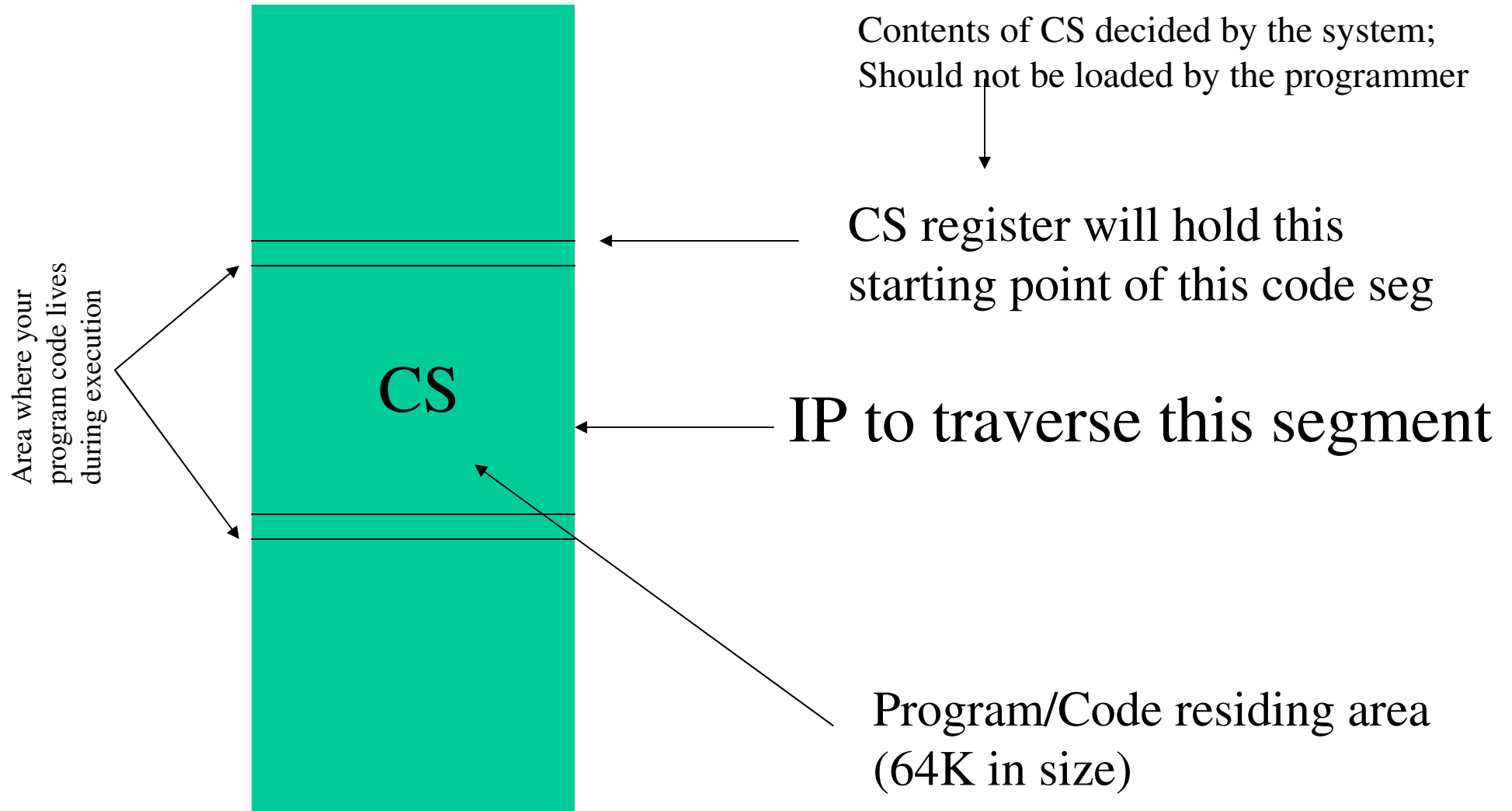
What system does is that:

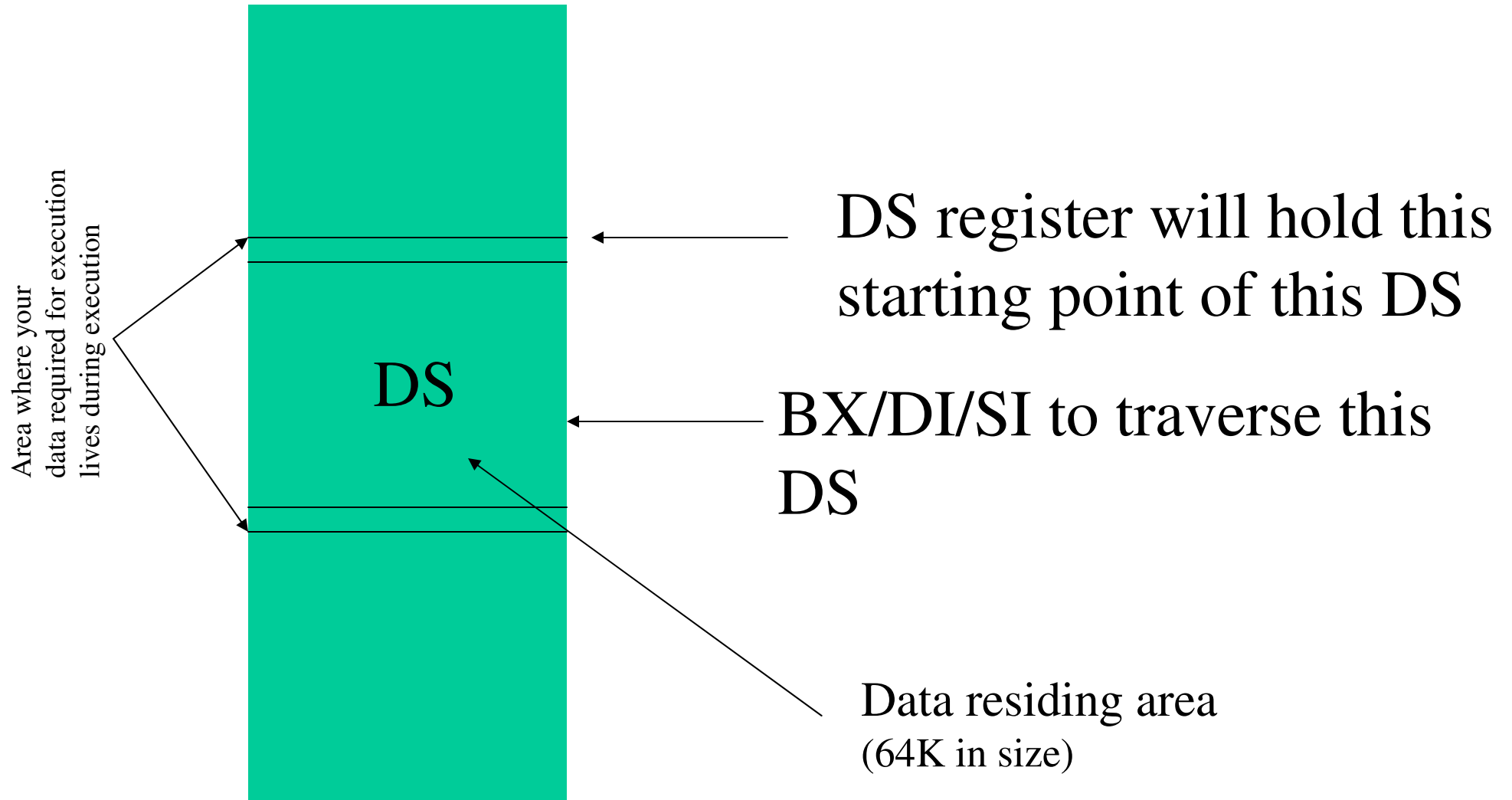
- (1) $1503 * 10H = 15030H$; multiplying by 10H is equal to left shift by 4-bits
- (2) $15030 + \text{the offset} = \text{exact physical address where the current IP is pointing to!}$

The above is applicable for all other segments too...

Physical address generation....







Convention – In 8086, how the data will be stored when it is of length more than 1 byte?

We use little endian convention:

Lower byte to lower address

Higher byte to higher address

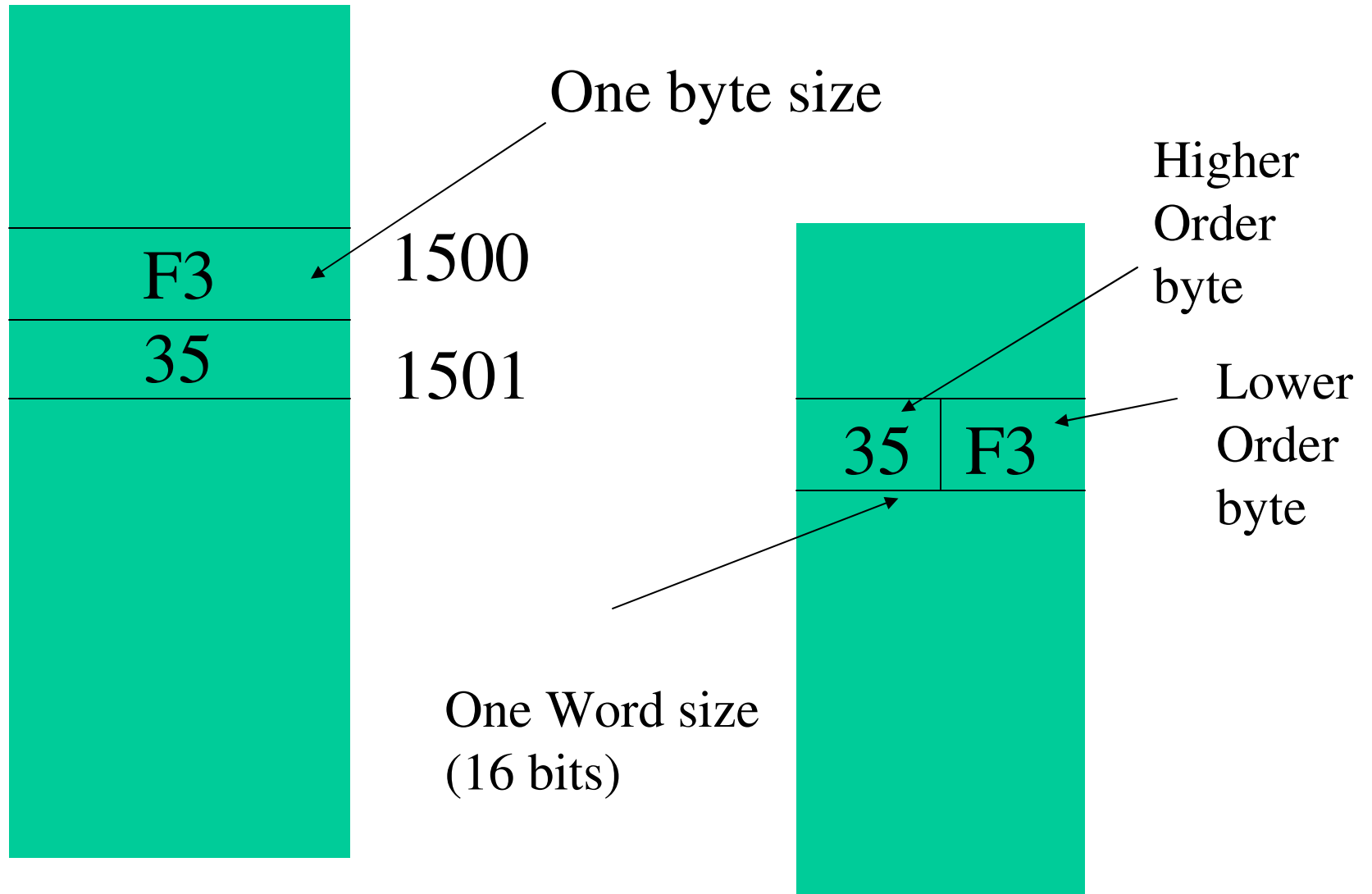
00000 (Lower address)

↓
FFFFFF (Higher address)

Example

MOV AX, 35F3H Then, DS:1500 = F3 and DS:1501 = 35
MOV [1500], AX

Representations

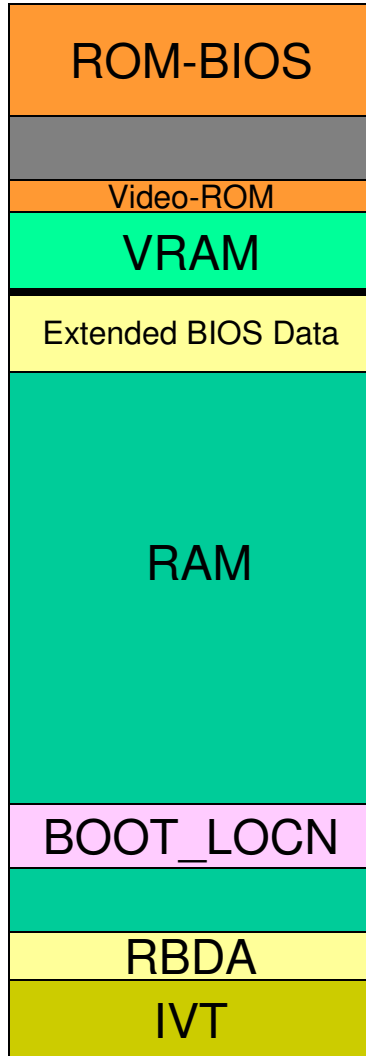


Therefore to surf through a given segment, we need two pointers – one pointer should tell us where to start from (segment address) and using the other pointer we will surf within that segment (offset address)

Extra Segment (ES, BX/DI/SI)

Stack Segment (SS, SP)

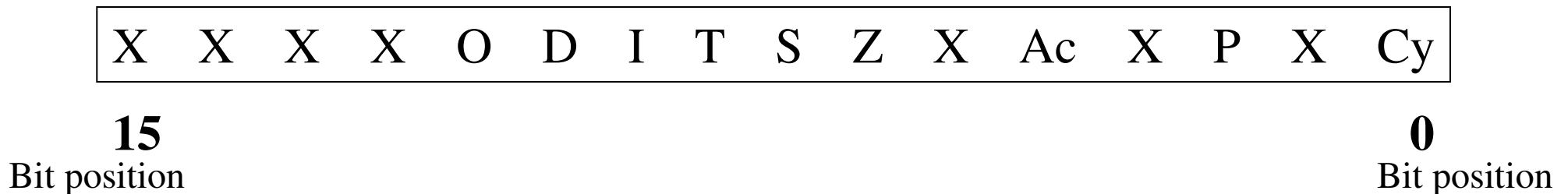
These two segments will be defined and used if needed in the program by the programmer



This is the zone where the segments (CS, DS, SS, & ES) live during execution of your program!

FLAGS/PSW

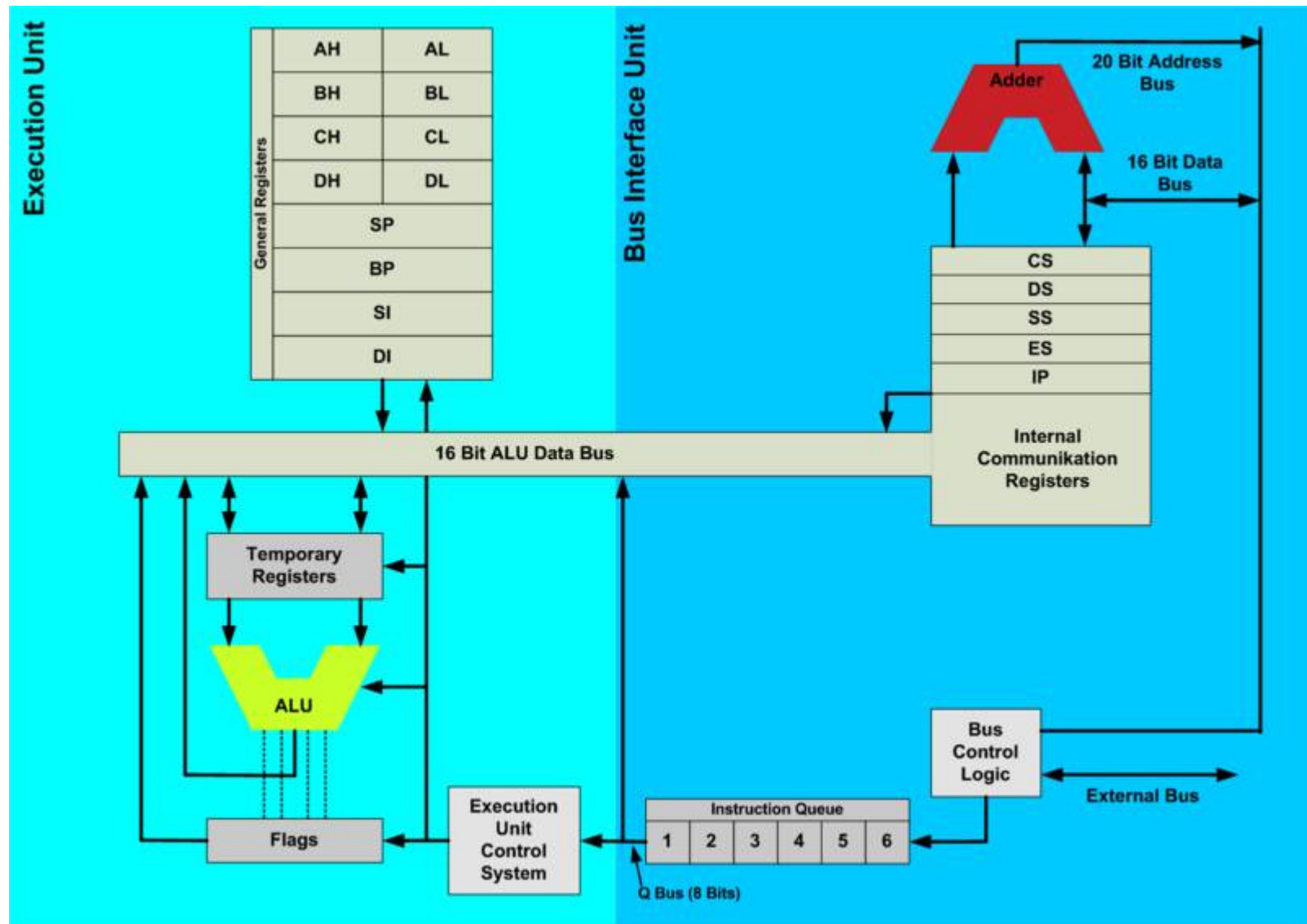
8086 flag register contents indicate the status of computations in the ALU. It also has some flag bits that can be used to control CPU Operations and useful for decision making purposes. It is of 16-bits in length and has the following.



X – unused; Flags- **O, D, I, T, S, Z, Ac, P, C**

{ Overflow, Direction, Interrupt, Trap, Sign, Zero, Auxiliary carry flag, Parity, Carry }

Now see the complete picture....

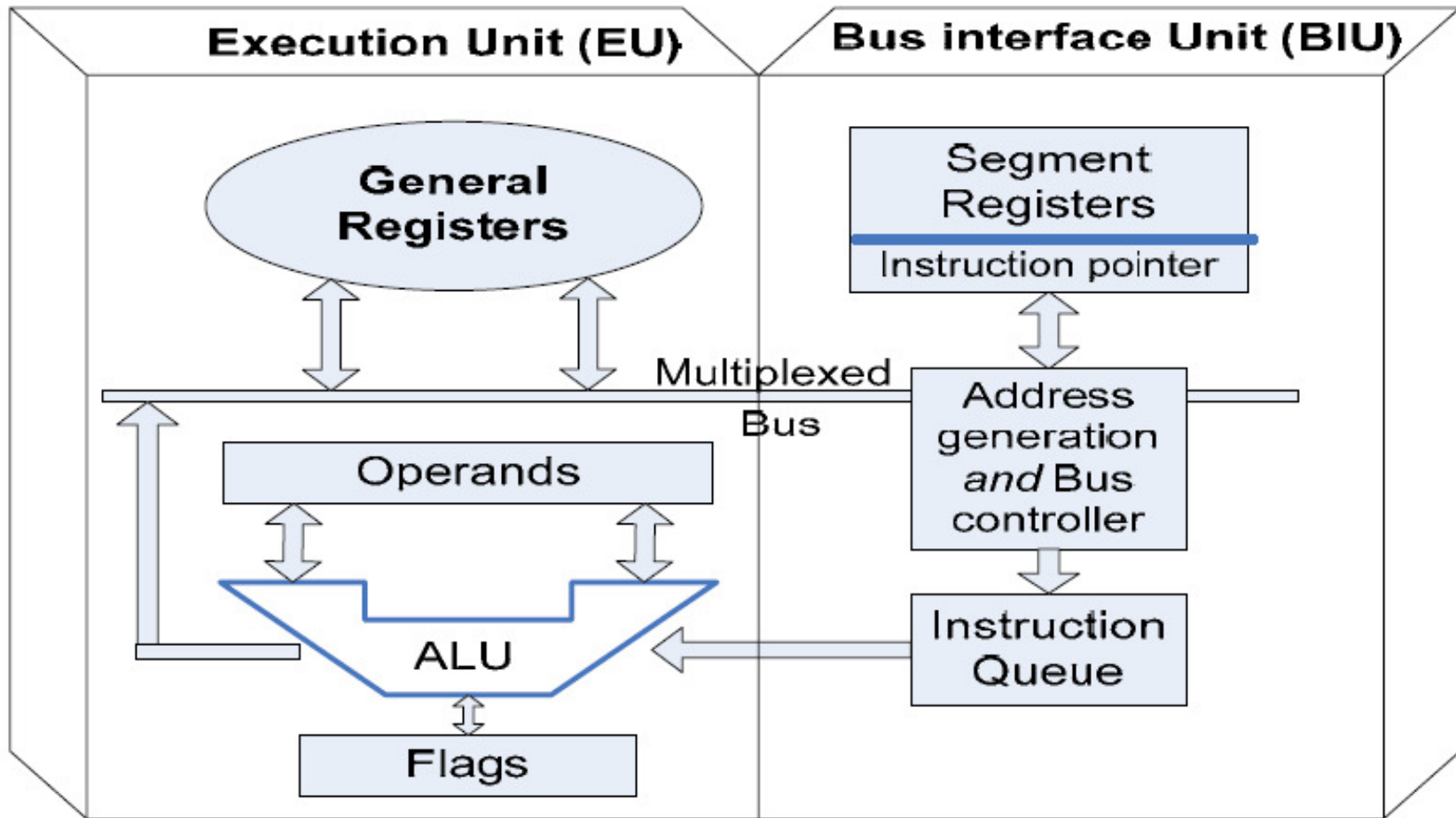


-Micro-architecture – What is it?

Implements the software and hardware architecture and specifies information about the **execution-unit**, **pipelining** and the **instruction-set** of the processor;

- Micro-architecture comprises - **Execution-unit (EU)** and **Bus-interface-unit (BIU)**

This is your CPU....



The **BIU uses multiplexed system** bus to fetch instruction, read/write data operands for memory or for input/output peripherals. It is also responsible for instruction queuing and physical-address generation.

- The pipelined architecture implemented by the instruction queue, allows 8086/8088 to pre-fetch up to 6/4 bytes of instructions;

- **Execution unit (EU)** is responsible to decoding and executing instructions. It accesses instructions from the instruction-queue and data from the general-purpose-registers or memory (with the help of BIU)

(The ALU of the execution unit (shown in previous figure) performs the arithmetic, logical and shift operation required by the instruction. During execution the EU tests and updates the status of the Flag registers based on the result of the execution.)

About BIU:

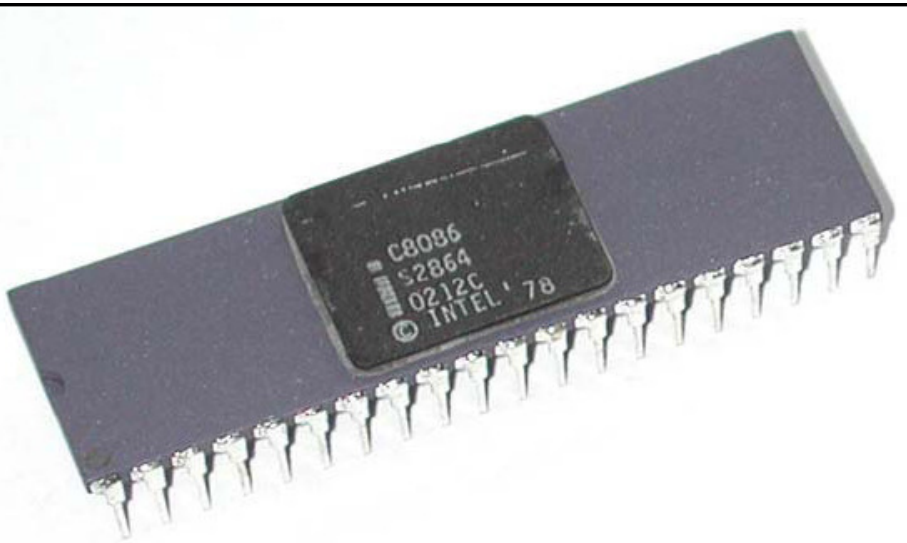
The BIU is made up of Address Generation and Bus-Control Unit, the instruction queue, and the instruction pointer. It has the task of making sure that the bus is fully and properly utilized in order to speed-up the operations. This function is carried out in 2 ways - (1) by fetching the instructions before they are needed (pre-fetching) by the EU and storing them in an instruction queue; (2) by taking care of bus control functions, the EU can be freed to carry on its work (instruction execution); The IP contains the location or address of the next instruction to be executed;

About EU:

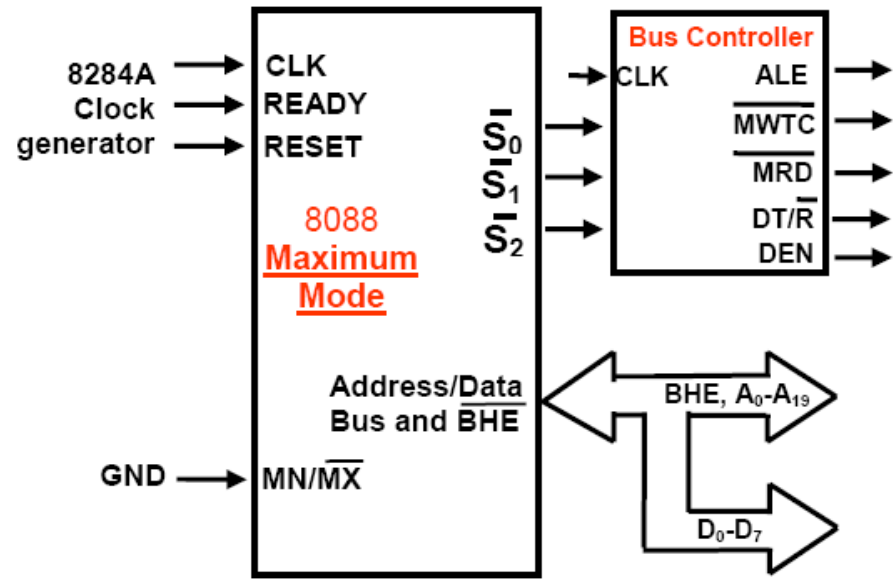
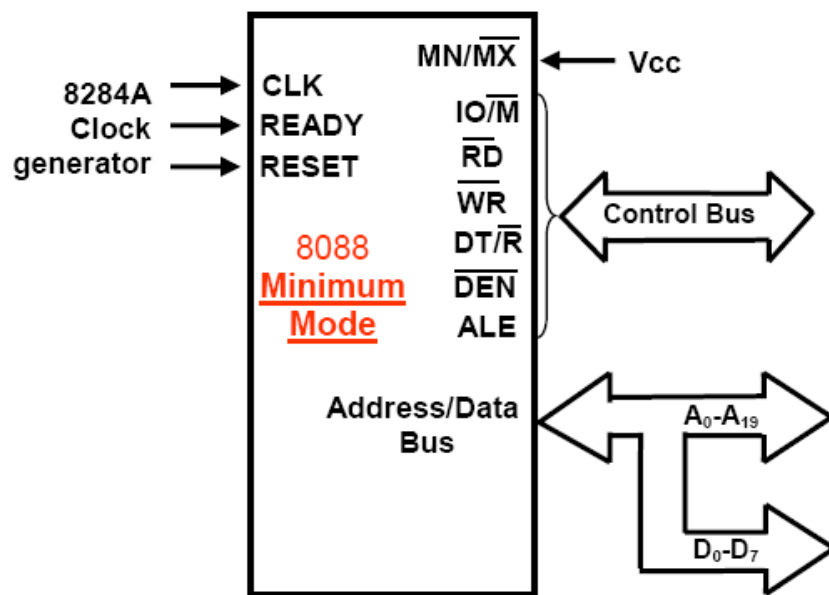
The EU is made of two parts - (1) ALU (2) GPRs; It is in EU that instructions are received, decoded, and executed from the instruction queue part of BIU; The instructions are taken from the top of the IQ FIFO basis; ALU is the *calculator part* of our CPU! It has electronic circuitry that performs the required arithmetic and logical operations. The control system can also be thought of as a part of EU, which provides a path for the flow of instructions into the ALU, GPRs and flag register;

The microprocessors 8086 and 8088 were both made of **HMOS** technology with an IC circuitry equivalent to ≈ 29000 transistors.
(*HMOS: high performance metal oxide semiconductor*)

- Unlike the software model, the hardware architecture of 8088 microprocessor is different from that of 8086 (Recall - **8086 has 16-bit data bus and 8088 has 8-bit data bus**)
- Both processors are housed in a **40-pin dual in-line package**, with many of the pins having multiple functions are *Multiplexed*.



- The microprocessors 8086 and 8088 can be configured to work in two modes: The Minimum mode and the Maximum mode.
- The Minimum mode is used for single processor system, where 8086/8088 directly generates all the necessary control signals.
- The Maximum mode is designed for multiprocessor systems, where an additional “Bus-controller” IC is required to generate the control signals. The processors control the Bus-controller using status-codes.



- A set of conductors, used for communicating information between the components in a computer system is called **System-Bus**.

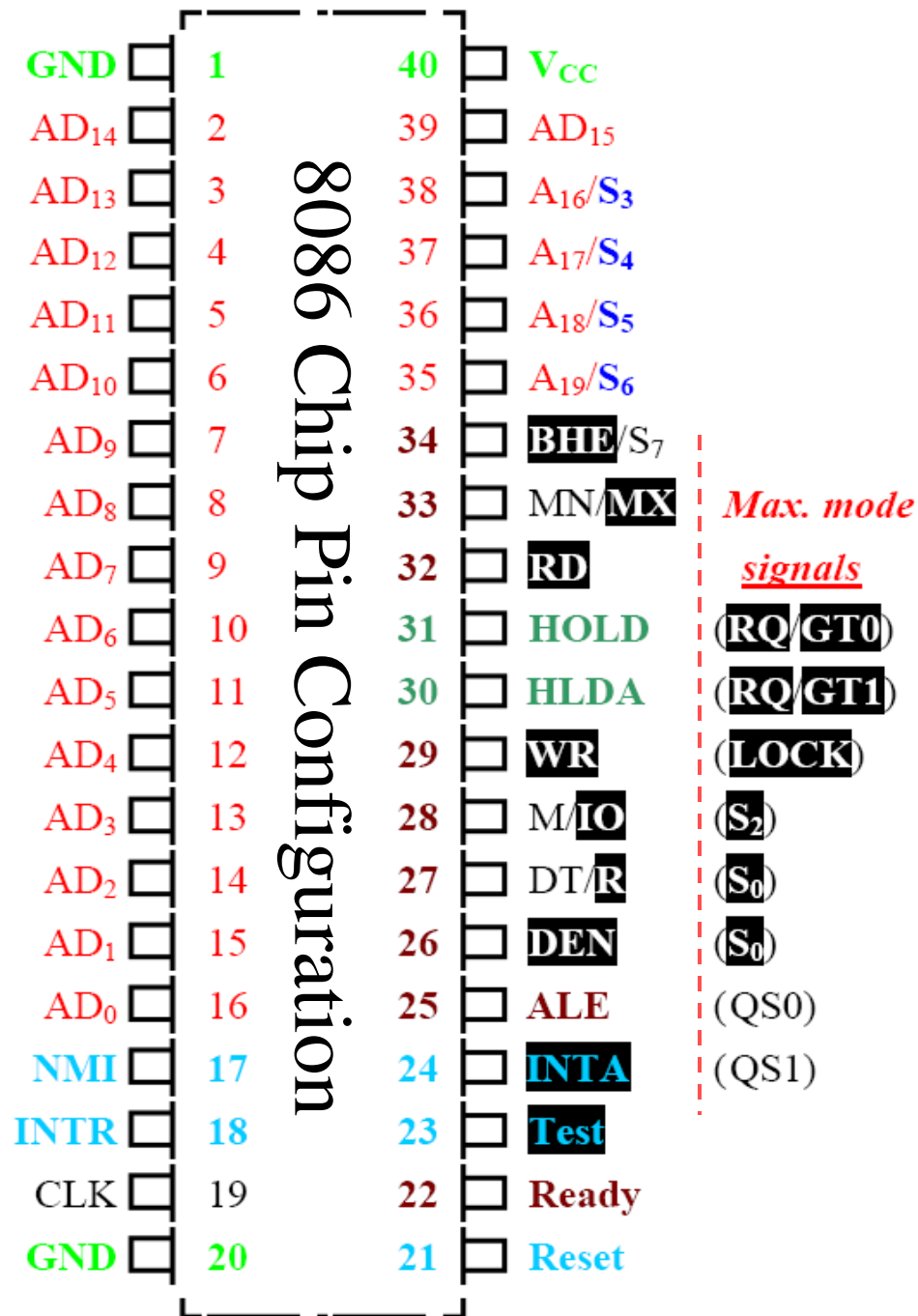
- **Internal Bus**: connects two minor components within a major component (or IC), such the connection between the control unit and internal registers of the CPU)

- **External Bus**: connects two major components, such as CPU and an interface (Memory or input/output). Although some systems include more than one external bus, 8086 and 8088 processors contain one bus called system-bus.

- Typical system-bus includes; **Address-bus** (*carries physical address of memory storages or input/output locations*), **Data-bus** (*carries data to be read or written into CPU registers*) and **Control-bus** (*carries information to control the read or write operation*).

- In addition to CPU, the bus-system is also used by other components of the computer, during which the address, data and control pins of the CPU remains logically disconnected or at high-impedance state.

On buses...



Note that signals can be divided into following groups:

Address & Data signals:

- Address BUS (A₀-A₁₉)
- Data BUS (D₀-D₇ & D₀-D₁₅)
(Note, **Multiplexed** pins)

Control Signals:

- MN/MX signal
- ALE signal
- (IO/M)₈₀₈₈
- RD and WT signals
- DT/R signal
- DEN signal (Min. ≠ Max.)
- SSO signal etc.....

Status Signals:

- S₃ to S₆ signals
(multiplex with address pins)

Interrupt Signals:

- INTR and INTA signals
- TEST signal

DMA interface Signals:

- HOLD/HLDA

Definition of signals from 8086/8088 IC pin's :

- Pins 2 to 16 and Pins 35 to 39 of 8086/8088 IC generate address signals (A19-A0). This 20-bit **address bus** allows the processor to access 1 Mega, Byte-wide, memory storage locations or 64 kilo, byte-wide, input/output ports.
- In 8086 IC, Pin 2 to 16 and Pin 39 generates/receives 16-bit data signals (D15-D0) to write/read data into the CPU registers. Thus, **Data bus** supports bi-directional data flow. But in 8088 IC, the Data bus is 8-bit wide and consists of data signals from Pin's 9 to 16.
- **Note:** Address-bus and Data-Bus in both 8088 and 8086 uses *Multiplexed pins*.

- Pin 33 of 8088 and 8086 IC accepts *Minimum and Maximum mode*(MN/MX) signal to select the processors operating mode. Thus, MN/MX='1' initiates minimum mode and MN/MX='0' initiates maximum mode of operation.

- Pin 25 controls *Address Latch Enable* (ALE) signal (pulse) to indicate that valid physical-address is present in the address-bus (A19-A0). During this pulsed period, address/data-multiplexed-pins carries Address-information to locate desired I/O or memory location.

Note: Since we have segmented memory system, the way in which we like to write programs (commands for a specific task) must also capture these definitions. Therefore, while programming 8086, we need to define “clearly” and “separately” the “portions” that are called as “code”, “data” and “stack” to the “assembler”.

With some examples this should be clear shortly. *Stay tuned!*

We are now set to learn

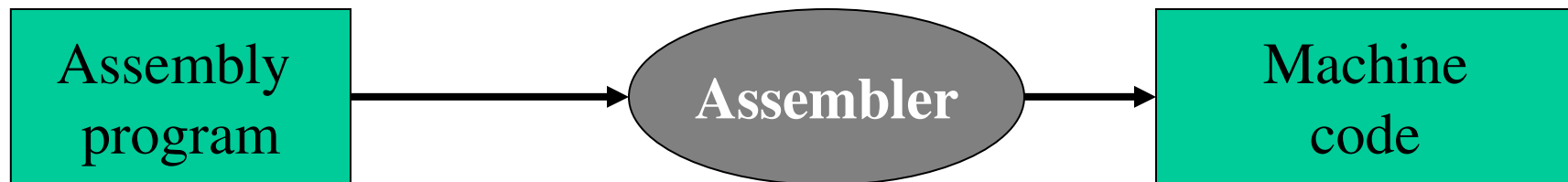
“ASSEMBLY PROGRAMMING”

Introduction to Assembly Programming

While CPU can work only in binary, it can do so at very high speeds. It is quite tedious and will be a slow process for humans to deal with 0s and 1s for programming.

Machine language – 0s and 1s; working with machine codes is cumbersome

Assembly languages – provided what are called as *mnemonics* for machine Code instructions + other features that made programming faster and efficient

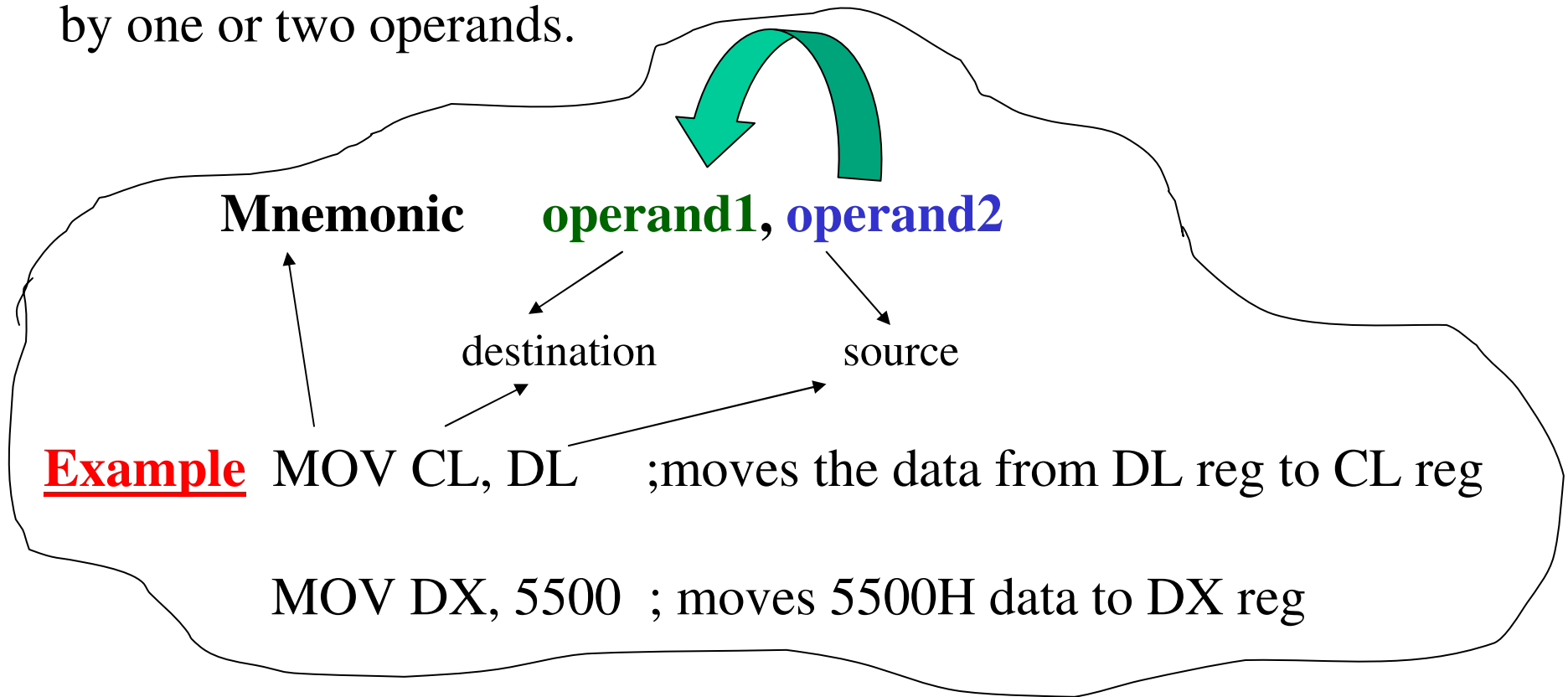


Assembly language programming is often called as low-level Programming due its proximity to CPU internal architectures.

So, if you are “programming in assembly language” means that you should be aware of what type of CPU you are using and what are the resources (registers, memory space range, I/O operation style, etc) it has. These resources and the way in which these are used are captured in the instruction set of the CPU. Thus, instruction set tells you how to use your CPU and what all you can do with your CPU.

Now, recall all you might have come across on several electronic devices and you must also have heard people saying “microprocessor controlled.....”

An assembly language programming consists of, among other things, a series of instructions (designed for that particular CPU). An assembly language instruction consists of a **mnemonic**, optionally followed by one or two operands.



Note: Though we say it “moves” it is actually copying the data from source to destination. MOV instruction does not affect the source operand

A machine language instruction usually has one or more fields associated with it.

First field – operation code field/ opcode field- type of operation to be performed

Other fields – operand fields

- Six (6) general formats of instructions exists for 8086 processor. Further, length of an instruction may vary between ONE BYTE to SIX BYTES.

Instruction set of 8086/8088

- Data copy/transfer instruction
- Arithmetic and logical instructions
- Branch instructions
- Loop instructions
- Machine control instructions
- Flag manipulation instructions
- Shift and rotate instructions
- String instructions


Addressing modes of 8086


Addressing modes indicate the ways of locating data or operands. Depending on data types used in instruction and the memory addressing Modes, any instruction may belong to one or more addressing modes. Sometimes, they may not belong to any type of addressing modes.

This means that addressing modes describe the types of operands and the Way they are accessed for executing an instruction.

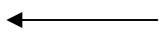
There are 8 types of addressing modes, as follows.

Learn through Examples

1. **Immediate** MOV AX, 0005 
 - Immediate data as a part of instruction
 - Data may be 8 or 16 bits

2. **Direct** MOV AX, [5000]  16-bit offset address is directly specified as a part of the instruction

So, in the above example, what happens is that, the contents of address = (left shift DS + 5000) are moved to AX. This is what is written as 10*DS+5000.

3. **Register** MOV AX, BX  All registers except IP may be used in this mode

4. **Register Indirect** MOV AX, [BX]

- Default segment is either DS or ES;
- BX, SI, or DI could be offset address holders;

This is same as the previous one, however, the difference is that, the contents pointed by BX is of our concern. Thus, the contents of the address $10*DS+[BX]$ are moved to reg AX

5. **Indexed** MOV AX, [SI]

Contents of $10*DS+[SI]$ are moved to AX

Special case of (4) above.
Offset of operand is stored in one of index registers. Default seg reg always is DS.

Special case - DS and ES are the default segment registers for the index registers SI and DI, in the case of string processing;

6. **Register Relative** MOV AX, 50[BX]

Data for processing is available at an effective (physical) address. This address can be reached by adding an 8-bit or 16-bit displacement (in the example 50H) with the contents of any one of the registers BX, BP, SI, and DI in the default segment (DS or ES). Thus, contents of $10*DS+[BX]+50$ are moved to AX.

7. **Based Indexed** MOV AX, [BX][SI]

We are now experts on this! Thus, the above means move the contents of $10*DS+[BX]+[SI]$ are moved to AX.

8. **Relative Based Indexed** MOV AX, 50[BX][SI]

Uhh! I know this too... Contents of $10*DS + 50 + [BX] + [SI]$ are moved to AX

What about control transfer instructions? The addressing mode depends on whether the destination location is within the same segment or a different segment. It also depends on the method by which the destination address is passed to the CPU.

Most commonly used instructions

- MOV, PUSH and POP, IN and OUT
- ADD, ADC, SUB, SBB, CMP, INC, DEC, Different flavors of ASCII Adjust after Addition (AAA), different flavors of decimal adjustment (DAA)
- AND, OR, NOT, XOR, SHR, SHL, SAR and SAL, different flavors of rotate right/left with/without carry
- REP, MOVSB, MOVSW, CMPS
- CALL, RET, INT N, JMP, LOOP

- Different flavors in JZ/JE, JNZ/JNE, LOOPZ, LOOPNZ
- CLC, CMC, STC, CLD, STD, CLI, STI
- WAIT, HLT, NOP, ESC, LOCK

Assembler directives and operators

Hints to the assembler using some predefined alphabetical strings are called *assembler directives*. These directives help the assembler to correctly understand the assembly language programs to prepare the machine codes. Use of directives and operators will become clear and familiar once we start looking out some examples.

Let us see how a typical ASM program structure looks like!!

Segment definition

```
data_seg1      SEGMENT
    directives
data_seg1      ENDS

data_seg2      SEGMENT
    directives
data_seg2      ENDS

code_seg SEGMENT
    start:
        instruction and directives
code_seg ENDS
END
```

Requires ASSUME and also explicit loading of DS, ES and SS registers. For example, the ASSUME will tell the assembler to use the contents of DS as the segment address when a variable in data_seg1 is referred to in an instruction.

```
data_seg1      SEGMENT
    directives
data_seg1      ENDS

data_seg2      SEGMENT
    directives
data_seg2      ENDS

code_seg SEGMENT
    ASSUME CS:code_seg, DS:data_seg1, ES:daga_seg2
    start:
        MOV     AX, data_seg1
        MOV     DS, AX
        MOV     AX, data_seg2
        MOV     ES, AX
        etc
code_seg ENDS
END
```

Before we introduce the concept of sub-routines we shall understand how we have to write programs in assembly language. We will use this knowledge to surf through the codes used for explaining sub-routines later.

Example Program for adding two numbers. Typically, an assembly program looks like this.

ASSUME CS:MYCODE, DS:MYDATA

MYDATA SEGMENT

```
    OPR1  DW  1234H
    OPR2  DW  0002H
    RESULT DW  01 DUP (?)
```

Data segment definition and its contents

MYDATA ENDS

MYCODE SEGMENT

```
START:  MOV AX, MYDATA
        MOV DS, AX
        MOV AX, OPR1
        MOV BX, OPR2
        CLC
        ADD AX, BX
```

```
        MOV DI, OFFSET RESULT
        MOV [DI], AX
        MOV AH, 4CH
        INT 21H
        MYCODE ENDS
```

Code segment definition and its contents

END START

Step 1 is to analyze the problem given, in terms of deciding the logic you want to use and to decide on what are the segments you need. In our above example, we can guess that some data are required for adding and we need some “space” to store the result. Consequently, data segment becomes an important segment to consider in this example. Thus, we need a logical space called data segment, as defined. Code segment is of course expected, as it will contain the actual set of instructions to be executed. Sometimes we need to use STACK facilities and hence, we can go for SS.

ASSUME directive declares that the label **MYCODE** is to be used as a logical name for CODE segment and the label **MYDATA** is to be used for DATA segment. These labels **MYCODE** & **MYDATA** are reserved by the assembler (MASM) for this purpose. Once these are defined, **MYDATA** refers to data segment and **MYCODE** refers to code segment throughout the program. Note that we can also use

DATA_X SEGMENT

....

DATA_X ENDS

CODE_Y SEGMENT

....

CODE_Y ENDS

Then, we may define **ASSUME CS:CODE_Y, DS:DATA_X**

MYDATA SEGMENT statement marks the start of logical data space MYDATA. Inside this data segment, OPR1 is the 1st operand, OPR2 is the 2nd operand. The final Result is stored in RESULT for which space has been already reserved. Statement MYDATA ENDS marks the end of DATA segment.

These labels OPR1 OPR2 and RESULT will be allocated physical memory locations whenever logical SEGMENT MYDATA is allocated memory or loaded to the memory. Assembler calculates that the above data segment requires 6 bytes (2+2+2).

Similarly, code segment starts with MYCODE SEGMENT statement. The label STARTS marks the start of the point of execution.

The assembler was just informed by the directive ASSUME that the label MYCODE is used for code segment and MYDATA is used for data segment and does not actually put the address corresponding in the code segment register CS and the address corresponding to the data segment register in DS. **Then, who puts them?**

A programmer has to carry out this operation of loading the DS, SS, and ES registers. But, CS is automatically initialized by the loader at the time of loading the EXE file into the memory for actual execution.

No segment register can be loaded with an immediate segment address directly. They should be loaded via a GPR. Further, CS should not be loaded at all and its contents can be changed whenever a program calls other routines for execution.

The logic of the above example is then straightforward to follow. Note that indexed addressing mode is used to store the result of addition in memory locations labeled RESULT.

A function value of 4CH is loaded to AH register for invoking INT 21H routine. This Value (can be obtained from manual) is for returning to the DOS prompt.

MYCODE ENDS marks the end of the procedures that started with a label **START**. At the end of each file, END statement is a must.

A typical ASM program looks like this....

data segment

msg db 'Hello World!', 10, 13, '\$'

data ends

code segment

assume cs:code, ds:data

start: mov ax, data

mov ds, ax

...

int 21h

exit: mov ah, 4ch

int 21h

code ends

end start

*We will see this coding part
now....*

Best way to start learning a programming language is by examples. As and when we go, we will learn more syntax and some rules in writing a program!

Let us start with examples . . .

Machine Level Programs: Examples

Let us first see how we can use the instruction set we have learnt in **Part 1** of this course so far.

Example 1

Add a data byte located at offset 0500H in 2000H segment to another data byte stored at 0600H in the same segment. Store your result at 0700H in the same segment.

```
MOV AX, 2000H ; Rule: DS cannot be loaded directly with 2000H, we do this  
MOV DS, AX      in two steps  
MOV AX, [0500H] ; contents of 0500H are moved to AX  
ADD AX,[0600H]  ; contents of 0600H are added to AX and result is in AX  
MOV [0700H],AX ; contents of AX are moved to the location 0700H  
HLT
```

Example 2

Move the contents of memory location 0500H to register BX and also to CX. Add immediate byte 05H to the data residing in memory location, whose address is computed using DS=2000H and offset=0600H. Store the result of the addition in 0700H. Assume that the data is located in the segment specified by DS which contains 2000H.

```
MOV AX,2000H  
MOV DS, AX      ; loading DS as before  
MOV BX,[0500H] ; contents of 0500H are moved to BX  
MOV CX,BX      ; loading CX too as asked  
MOV AX,[0600H] ; contents of 0600H are moved to AX  
ADD AX,05H     ; adding AX and 05H and the result in AX  
MOV [0700H],AX ; moving the result to location 0700H  
HLT
```

Note: MOV CX,BX & MOV CX,[0500H] are possible. Case 1-opcode is 2 bytes whereas in case 2 – opcode is of 4 bytes long, implying more execution time and memory.

Example 3

Add the contents of memory location 2000:0500H to contents of 3000:0600H and store the result in 5000:0700H.

The data are located in different data segments. The respective starting addresses and the offsets are given.

```
MOV AX,2000H    ; FIRST DATA SEG  
MOV DS,AX  
MOV CX,[0500H]  
MOV AX,3000H    ; SECOND DATA SEG  
MOV DS,AX  
MOV AX, [0600H]  
ADD AX,CX       ; RESULT IN AX  
MOV CX,5000H    ; THIRD DATA SEG  
MOV DS,CX  
MOV [0700H],AX  ; MOVE THE RESULT TO 0700H OF SEG3  
HLT
```


Example 4

Move a string of 16 bytes long from the offset address 0200H to 0300H in the segment 7000H.

MOV AX, 7000H

MOV DS, AX

MOV SI, 0200H ; SI is loaded with 0200H and serves as a pointer

MOV DI, 0300H ; DI is loaded with 0300H and serves as a pointer

; We need a counter to track the number of bytes so,

MOV CX, 0010H ; length of the string – 16 bytes

AGAIN: MOV AX,[SI] ; contents of current SI

MOV [DI],AX

INC SI

INC DI

DEC CX

JNZ AGAIN

HLT

Alternatively,

8086 instruction set provides an alternate way of handling the looping facility via LOOP instruction. Thus, DEC CX and JNZ AGAIN statements can be replaced with

-
-
-

```
INC SI  
INC DI  
LOOP AGAIN  
HLT
```

Thus, this saves memory space and time of execution. The LOOP instruction needs INC SI and INC DI instructions only and does not need to perform counter decrement and check if zero results.

Alternatively, for string manipulations and handling, 8086 has something special...

```
MOV AX, 7000H  
MOV DS,AX  
MOV ES,AX      ; DESTINATION SEGMENT INITIALIZATION  
MOV CX, 0010H ; COUNTER INITIALIZATION  
MOV SI, 0200H  
MOV DI, 0300H  
CLD           ; CLEAR DF FLAG  
REP MOVSB      ; MOVE THE COMPLETE STRING FROM SRC TO DST  
HLT
```

MOVSB – MOVING THE STRING BYTE FROM SRC TO DST

MOVSW – MOVING THE STRING WORD FROM SRC TO DST

In the above program, CX specifies the length, DS and ES are initialized to SRC and DST segment addresses, respectively. SI and DI are to point these SRC and DST addresses. Before the use of string instructions all these registers must be initialized properly.

Example 5

Determine the largest number from an unordered array of 16 8-bit numbers stored sequentially in the memory locations starting at offset 0500H in the segment 2000H.

Program : Refer to **Page 81** of your prescribed text book by Ray.

Logic: A number is taken in a register. It is compared with another number. If the second one is greater than the first one (in the register) then the second number is replaced in the register. If not, we consider another number from the memory. The process is repeated until all numbers are exhausted (checked) and the largest number is eventually stored in that register.

Use of instructions **JNC** and **CMP** are demonstrated in this example.

Macro assembly language programming

- Fairly sophisticated, almost C like
 - Executable statements
 - non-executable statements
 - reserve storage commands
 - pre-assigned values
 - define names to constant
 - assumed content of segment registers
- Scope
 - data definition and storage allocation
 - structures
 - segment definition
 - alignment directive
 - assembly process

Data definition and storage allocation

DB **define byte**
DW **define word (2 bytes)**
DD **define double word (4 bytes)**

Examples

data_byte	DB	10, 4, 10H	
data_word	DW	100, 100H, -5	
data_dw	DD	3*20, 0FFFDH	
message_b	DB	'HELLO'	
message_w	DW	'AB'	;order of storage is reversed
ABC	DB	0, ?, ? ?, 0	;? ==>space reserved but not initialized

DUP operator

array1	DB	2 DUP (0, 1, 2, ?)
array1	DB	0,1,2,?,0,1,2,?
len	EQU	2
array2	DB	len DUP (0, 3 DUP(1 ,2), 3)
array2	DB	0, 1, 2, 1, 2, 1, 2, 3, 0, 1, 2, 1, 2, 1, 2, 3

Initialize address

para_table	DW	PARA1 ;offset of PARA1 stored
int_seg_add	DD	SERIAL ;offset & segment address of SERIAL stored

Data structures

- Similar to the structure in C language
- Facilitates programming
- Defines a pattern
- We can then define many variables of that pattern
- **Example: to define a pattern of 11 bytes**

```
personal_data  STRUC
    initial          DB      'XX'
    last_name        DB      5 DUP(?)
    id                DB      0, 0
    age               DB      ?
    Weight            DB      ?
personal_data  ENDS
```

- **Example: to reserve space of this pattern**

```
employee_1      personal_data    'JR', , , 35
employee_2      personal_data
employee         personal_data
```

- **Example: usage**

```
mov al, employee_1.last_name[SI]
```

Assigning names

- Not essential to make a program work but meaningful
- Adds to readability and convenience
- Example: to define

```
port0      EQU      80H
port1      EQU      81H
index_char EQU      char_array[SI+10]
```

- Example: to use

```
OUT port0, AL
IN  AL, port1
```

```
mov AL, index_char <====> mov AL, char_array[SI+10]
```

- if expression contains var, label, constant names, etc
- then it must constitute entire expression
- or it must be predefined

- Allows easy modification of program

```
num EQU 6
```

- follow by many statements involving num
- Changing num will automatically change all statements involving num

Using Procedures

Procedure or otherwise referred to as **subroutine** (in the case of high-level languages) is an important part of computer system's software architecture.

In general, a procedure is nothing but a group of instructions that can be used any number of times as per the requirements of the computation demanded by the program. Thus, the significant advantage is that, memory space is well-utilized and makes the programme modular in its design.

What is the disadvantage, if any? The only disadvantage is in the time taken by the CPU to link the procedure and to return from it to the main program. That is, execution of the program may take little more time in using the procedure, however, there is a lot of “gain” in using a procedure, especially if the set of instructions under the procedure need to be used several times.

How does the main program know where to come back after executing the procedure? The stack stores the address to which the execution by the CPU should return after the procedure.

So, who fills the stack with the return address? The answer is as follows.

The **CALL** pushes the address of the instruction following the CALL onto the stack while the **RET** instruction removes (retrieves) this address from the stack and so that the program continues its execution after the CALL instruction.

Rules for writing a procedure:

With every assembler, there are some specific rules to be followed in writing a procedure. In general, a procedure begins with a directive **PROC** and ends with a directive **ENDP**. Each directive appears with a name of the procedure. This sort of programming style makes it easier to locate the procedure in a program listing. Then, PROC directive is followed by the type of procedure we are interested in. The two types are, **NEAR** or **FAR**. Following example clarifies these aspects.

```
SUMS  PROC NEAR
      ADD AX,BX
      ADD AX, CX
      RET
SUMS  ENDP
```

```
SUMS1  PROC FAR
      ADD AX,BX
      ADD AX,CX
      RET
SUMS1  ENDP
```

The only difference in the above codes is in the opcode of RET (for NEAR AND FAR) instructions. The NEAR RET instruction uses the opcode C3H while the FAR RET instruction uses CBH. The significance is that, the NEAR RET retrieves a 16-bit number from the stack and places it in the IP to return from the procedure in the current code segment while the FAR RET retrieves a 32-bit number from the stack and places it in the IP and CS to return from the procedure to any memory location.

Intuitively it should be clear why this is done this way. For the NEAR procedure, predominantly the call is within the same segment (intra-segment) and hence, only IP needs to be modified, however, a FAR call will have a different CS as well. Hence, while returning to the main program, CS and IP have to be loaded appropriately.

Detailed example in **Tutorial 3** clarifies the effect of NEAR and FAR CALLS.

A procedure may be in :

1. Same code segment as the statement that calls it
2. A code segment that is different from the one containing the statement that calls it, but in the same source module as the calling statement
3. A different source module and segment from the calling statement.

In the first case, the attribute is NEAR and all the calls are from the same segment as the Procedure.

In the other two cases, the attribute must be FAR . If a procedure is of FAR type, then all calls to it must be inter-segment calls even if the call is from the same code segment. Why? Otherwise, the RET would pop two words from the stack even though only one word was pushed onto the stack by the call. In case 3, the procedure name must be declared in EXTRN and PUBLIC statements.

EXAMPLE

SEGX SEGMENT

.
. .
. .

SUBT PROC FAR

.....

SUBT ENDP

.
. .
. .

CALL FAR PTR SUBT

.
. .
. .

SEGX ENDS

SEGY

SEGMENT

.
. .
. .

CALL FAR PTR SUBT

.....

SEGY

ENDS

PROC SUBT is called both from SEGX and SEGY, and hence the attribute given is FAR. Since the attribute given is FAR, the calls from within SEGX must push both IP and CS onto the stack as well as for the call from SEGY.

Concept of Push and Pop – Stack operations

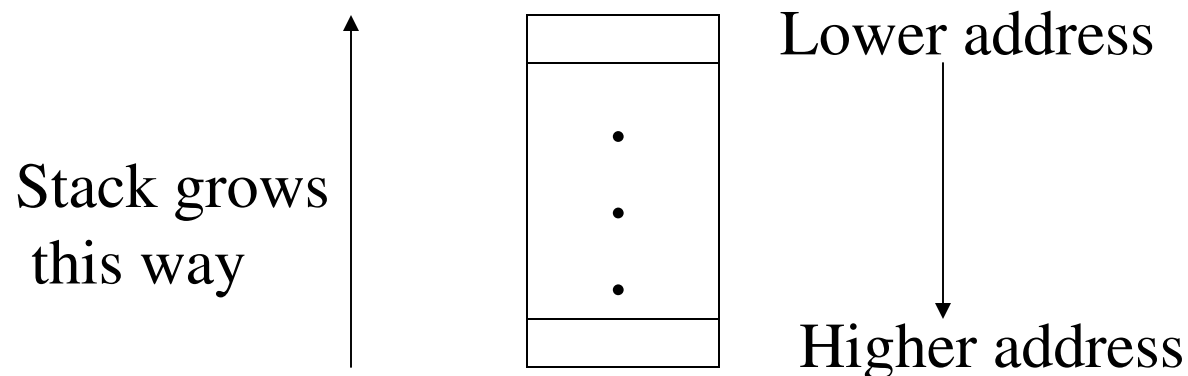
Since there are limited number of registers and all the codes must use the same set of registers, the contents of the registers must be saved before allowing other procedures to use them. The way in which the old data is saved is through stack. See the following Example.

<u>EXAMPLE</u>	SUM	PROC NEAR
		PUSH AX
		PUSH BX
		PUSH CX
		PUSH DX
	
		POP DX
		POP CX
		POP BX
		POP AX
	SUM	ENDP

Note: The code presented here does the saving and restoring within the procedure. alternatively, these can be done before the procedure is called and restored after just after the return, but this is not done usually in the practice.

Some rules have to be followed and these come from instruction set design of a computer. One of the rules, which is a “must be followed” says that “*When we push or pop a GPR, it must be the entire 16-bit contents*”. This means that, there are no instructions such as “push AL”, “push BH” in the instruction set.

SP is decremented after the “push”. This is to make sure that’s tack is growing “downwards” from “higher” to “lower” memory addresses.



Following example clarifies the working style of SS and SP

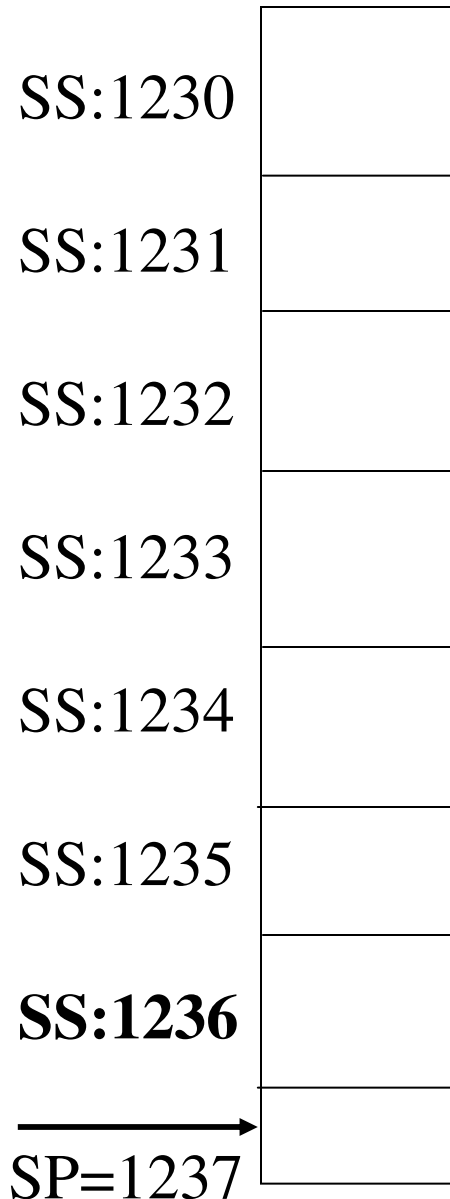
Example

SP = 1237, AX = 24B6, DI = 85C2, DX = 5F93

Instructions: Push AX, Push DI, Push DX

Initially SP points to 1237 (top of stack (TOS))

- After Push AX, SP = 1235; Contents of 1236 = 24, 1235 = B6
- After Push DI, SP = 1233; Contents of 1236 = 24, 1235 = B6, 1234 = 85, 1233 = C2
- After Push DX, SP = 1231; Contents of 1236 = 24, 1235 = B6, 1234 = 85, 1233 = C2, 1232 = 5F, 1231 = 93



Example

Let **SP = 18FA** and refer to the contents of the memory
Instructions: Pop CX, Pop DX, Pop BX

Initially SP = 18FA

SS:18FA	23
SS:18FB	14
SS:18FC	6B
SS:18FD	2C
SS:18FE	91
SS:18FF	F6
SS:1900	



- After Pop CX, CX = **1423**, SP = 18FC
- After Pop DX, DX = **2C6B** and SP = 18FE
- After Pop BX, BX = **F691**, and SP = 1900

Note the order of storing in the registers – our convention is always followed

Data actually remains in the memory after popping,
But cannot be accessed since SP is beyond that point!

Passing data to and from the procedures: Communication

Procedures may operate on the same set of data every time or they may process different set of data each time they are called.

EXAMPLE

```
DATA SEGMENT
    ARY    DW 100 DUP(?)
    CNT    DW ?
    SUM    DW ?
DATA ENDS
```

Note: In this example, the procedure refers to the variables directly as defined.

```
CODE SEGMENT
```

```
.....
    CALL NEAR PTR FUNCADD
.....
FUNCADD PROC NEAR
    PUSH AX
    PUSH CX
    PUSH SI
    LEA SI, ARY
    MOV CX, CNT
    XOR AX,AX
    NEXT:
        ADD AX,[SI]
        ADD SI,2
        LOOP NEXT
        MOV SUM,AX
        POP SI
        POPCX
        POPAX
        RET
FUNCADD ENDP
.....
CODE ENDS
```

The diagram illustrates the execution flow of the assembly code. An arrow originates from the `CALL NEAR PTR FUNCADD` instruction in the `CODE SEGMENT` and points to the `NEXT:` label, which marks the beginning of the procedure body. Another arrow starts from the `RET` instruction within the procedure body and points back to the `CODE ENDS` label, indicating the return from the procedure to the caller.

What if the number of array elements to be passed varies or need to be decided by the programmer every time he/she runs the code?

What if the number of parameters are also different every time?

There are two ways of passing parameter addresses:

Method 1: To construct a parameter address table (or array) and pass the address of the table via a register

Method 2: Push the parameter addresses onto the stack

We will see Method 1 in detail.

Method 1: Communication of data between procedures: *via a table of addresses; push parameter addresses onto stack before calling procedure*

```

PROG_SEG      SEGMENT
    ARY        DW      100 DUP(?)      ; RESERVE 100 WORDS FOR ARY.
    COUNT      DW      ?              ; AND 1 WORD FOR COUNT
    SUM         DW      ?              ; AND 1 WORD FOR RESULT
    TABLE     DW      3 DUP(?)        ; RESERVE 3 WORDS FOR
                                         ; PARAMETER ADDRESSES
                                         ; RESERVE SPACE
    TOS         DW      100 DUP(?)      ; FOR STACK
ASSUME  CS:PROG_SEG, DS:PROG_SEG, SS:PROG_SEG
START:  MOV     AX,CS
        MOV     DS,AX                ; INITIALIZE DS
        MOV     SS,AX                ; INITIALIZE SS
        MOV     SP,OFFSET TOS        ; INITIALIZE SP
        .
        .
        .
        MOV     TABLE,OFFSET ARY    ; PUT THE ADDRESSES
        MOV     TABLE+2,OFFSET COUNT; OF ARY, COUNT, AND SUM
        MOV     TABLE+4,OFFSET SUM  ; IN PARAMETER TABLE
        MOV     BX, OFFSET TABLE    ; PUT ADDR OF TABLE IN BX
        CALL    FAR PTR PROADD       ; CALL PROADD
        .
        .
        .
PROG_SEG      ENDS

```

(a) Calling program

```

PROADD      PROC    FAR
              PUSH    AX                ;SAVE REGISTERS
              PUSH    CX
              PUSH    SI
              PUSH    DI
              MOV     SI,[BX]           ;GET THE ADDR OF
              MOV     DI,[BX+2]         ;ARRAY, THE VALUE OF COUNT, AND
              MOV     CX,[DI]           ;THE ADDR OF
              MOV     DI,[BX+4]         ;RESULT
              XOR     AX,AX             ;CLEAR AX
NEXT:        ADD     AX,[SI]            ;ADD THE ELEMENTS
              ADD     SI,2              ;OF THE ARRAY
              LOOP    NEXT              ;TO AX
              MOV     [DI],AX           ;RETURN RESULT
              POP     DI                ;RESTORE REGISTERS
              POP     SI
              POP     CX
              POP     AX
              RET                       ;RETURN
PROADD      ENDP

```

(b) Procedure

Writing Macros

Definition

```
name MACRO [parameters,...]  
<instructions>  
ENDM
```

Macros are just like procedures, but not really!!!

Macros look like procedures, but they exist only until your code is compiled, after compilation all macros are replaced with real instructions.

If you declared a macro and never used it in your code, compiler will simply ignore it.

Example:

```
MyMacro MACRO p1, p2, p3  
    MOV AX, p1  
    MOV BX, p2  
    MOV CX, p3  
ENDM
```

```
MyMacro 1, 2, 3  
MyMacro 4, 5, DX  
RET
```

Note: Unlike procedures, macros should be defined above the code that uses it, for example:

The above code is expanded into:

```
MOV AX, 00001h  
MOV BX, 00002h  
MOV CX, 00003h  
MOV AX, 00004h  
MOV BX, 00005h  
MOV CX, DX
```

Some important facts about macros and procedures:

When you want to use a procedure you should use **CALL** instruction; For example: **CALL MyProc**

When you want to use a macro, you can just type its name. For example: **MyMacro**

Procedure is located at some specific address in memory, and if you use the same procedure 100 times, the CPU will transfer control to this part of the memory. The control will be returned back to the program by **RET** instruction. The **stack** is used to keep the return address. The **CALL** instruction takes about 3 bytes, so the size of the output executable file grows very insignificantly, no matter how many time the procedure is used.

Macro is expanded directly in program's code. So if you use the same macro 100 times, the compiler expands the macro 100 times, making the output executable file larger and larger, each time all instructions of a macro are inserted.

You should use **stack** or any general purpose registers to pass parameters to procedure.

To pass parameters to macro, you can just type them after the macro name.

To mark the end of the macro **ENDM** directive is enough.

To mark the end of the procedure, you should type the name of the procedure before the **ENDP** directive.

Interrupts

An **interrupt** is exactly similar to a procedure in the sense that it may be “called” from any other program and a return can be made to the called program, after the interrupt routine has been executed.

Since there is a branch to elsewhere, before executing the interrupt routine, the PSW and the registers used by the calling program must be saved and restored and the return must be made to the following instruction following the last instruction.

- Types

- Software: using INT command
- Internal: purely local depending on the state of CPU, example: divide by 0
- External: hardware at the interrupt pin

- Interrupt Sequence

- instruction completed
- push PSW, CS, IP
- load new CS, IP These new CS and IP addresses will tell where the interrupt routine is residing
- clear IF, TF

- **Interrupt routines**

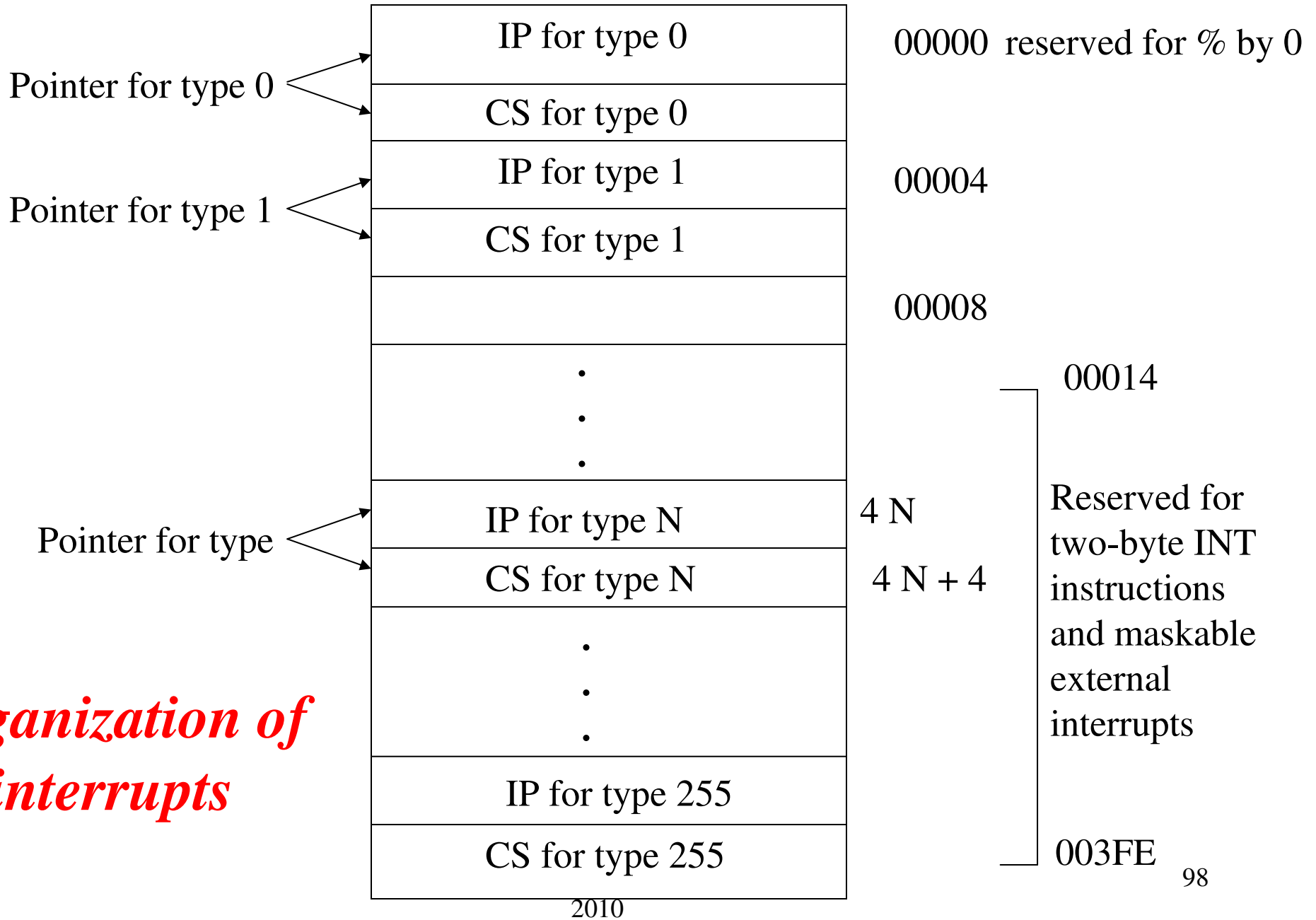
- written such that as if nothing happens to interrupted program except for a lapse in time
- usually absolutely located
- no parameters are passed, variables are directly accessible
- IF and TF must be properly set

The double word containing the new contents of IP and CS is called an **interrupt pointer (or vector)**. These CS and IP addresses give the locations of these interrupt routines. Each interrupt type is given a number, for identification, between 0 and 255 (thus, 256 types of interrupts can be handled).

If the interrupt type is 9, then the interrupt pointer will be in bytes 00024 through 00027.

Since it takes 4 bytes to store a double word, the interrupt pointers may occupy the first 1024 bytes of memory and these bytes should never be used for other purposes. Some of the 256 interrupts may be initialized by the OS when the system boots up and users may also customize other types as per their application requirements.

Organization of interrupts



Interrupts instructions

Mnemonic	Meaning	Format	Operation	Flags Affected
CLI	Clear interrupt flag	CLI	$0 \rightarrow (IF)$	IF
STI	Set interrupt flag	STI	$1 \rightarrow (IF)$	IF
INT n	Type n software interrupt	INT n	$(Flags) \rightarrow ((SP) - 2)$ $0 \rightarrow TF, IF$ $(CS) \rightarrow ((SP) - 4)$ $(2 + 4 \cdot n) \rightarrow (CS)$ $(IP) \rightarrow ((SP) - 6)$ $(4 \cdot n) \rightarrow (IP)$	TF, IF
IRET	Interrupt return	IRET	$((SP)) \rightarrow (IP)$ $((SP) + 2) \rightarrow (CS)$ $((SP) + 4) \rightarrow (Flags)$ $(SP) + 6 \rightarrow (SP)$	All
INTO	Interrupt on overflow	INTO	INT 4 steps	TF, IF
HLT	Halt	HLT	Wait for an external interrupt or reset to occur	None
WAIT	Wait	WAIT	Wait for \overline{TEST} input to go active	None

More interrupts

- software interrupts
 - equivalent to subroutine calls, except that the call address is global
- type 1 - reserved for single step
 - set TF, on next instruction goes into type 1 interrupt
 - with interrupt, CS, IP, PSW pushed onto stack, and TF is reset. The interrupt routine is then executed normally. Do what you want within the interrupt routine.
 - On exit interrupt routine, PSW is popped. The TF flag which was set earlier is now back. ==> program will execute one more step before going into the type 1 interrupt again.
- as a debugging tool.
 - insert INT at key points - goes to interrupt routine.
 - Can use to print out messages etc.

SERIAL DATA COMMUNICATIONS

Computers transfer data in 2 ways: **Serial and Parallel**

Serial transfer - one bit at a time;

Parallel Transfer - Byte or more at a time

We will study all about the serial data communications and the associated chips in this section.

Following are the terminology associated with data communications.

Channel behavior: Simplex; Half-Duplex; Duplex

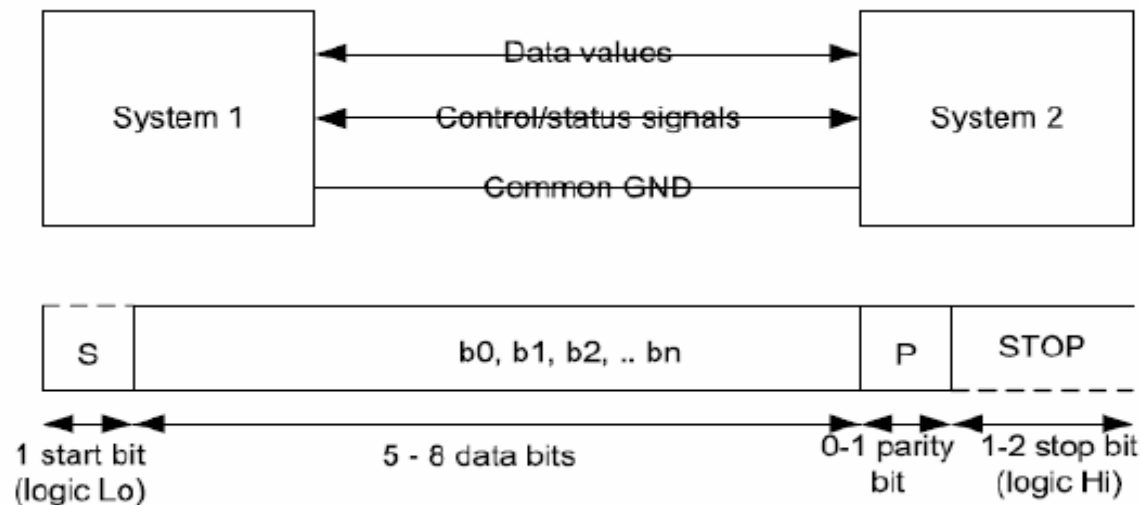
Type of communication: Synchronous; Asynchronous

- Simplex
 - Single channel, uni-direction for one-way communication.
- Half duplex
 - Single channel, time-shared bi-direction for two-way communication.
- Full duplex
 - Dual channels, bi-direction for simultaneous two-way communication.
- Synchronous communication
 - Requires a clock signal and synchronized characters.
 - An error detection code is needed.
- Asynchronous communication
 - Does not require a clock, does not have to be as responsive.
 - Error detection code is also needed.

Asynchronous Communications and Data Framing

Data are transferred in an organized way - the data are in the form of 0s and 1s; Transmitter and Receiver must agree to a set of rules (protocols) to correctly transfer, extract and interpret the data;

In Asynchronous communications, each character is put between Start and stop bits - this is called *framing*



Modern PCs use 1 stop bit. Now, assuming that we are transferring a file of ASCII characters using 11 bits for a character - 8 bits for ASCII code, 1 for start bit and 2 for stop bits, we have 30% overhead (extra 3 bits for every character);

With just 1 start and 1 stop bit, with 2400 bits per sec rate of transfer, the time taken to transfer 5 pages of ASCII data with each page having 80 x 25 characters is $100,000/2400 = 41.67$ seconds

Handshaking :

When two devices are involved, signals are to be sent to and forth between devices to reflect the status on either side;

Signals to indicate the device status;

Signals to indicate data transfer;

Actual Transmission and Reception

Bit rate/Baud rate

Bps - bits per second; Baud - Number of signal changes per second;

In modems, there are occasions when a single change of signal transfers several bits of data; For binary data bps = baud rate and we can use them interchangeably.

Common bps/ baud rates - 110, 150, 300, 600, 1200, 2400, 4800, 9600, 19200

RS232 Standards

To allow compatibility among data communication equipment made by Different manufacturers, an interfacing standard called RS232 Was set by Electronics Industries Association(EIA) in 1960. Refer to any manual for details on voltage levels and history of RS232 series.

RS232 - 25 pin connector, commonly referred to as DB-25 connector.

In labeling, DB-25P refers to plug connector (male) and DB-25S refers to socket connector (female). Your lab manual has a picture of this connector. Since not all pins are useful IBM introduced a reduced version DB-9 having 9 pins.

The electrical characteristics of the serial port as per the EIA (Electronics Industry Association) RS232C Standard specifies a maximum baud rate of 20,000bps, which is slow compared to today's standard speed. For this reason, we have chosen the new RS-232D Standard, which was recently released.

The RS-232D has existed in two types. i.e., DB-TYPE 25 pin connector and DB-TYPE 9 pin connector, which are male connectors on the back of the PC. You need a female connector on your communication from Host to Guest computer.

These 9 pins are: \DCD, RxD, TxD, DTR, GND, \DSR, \RTS, \CTS, RI

Modern terminology classifies data communication equipment as **DTE** (Data Terminal Equipment) or **DCE** (Data Communication Equipment);

DTE refers to terminals and PCs that send & receive data, while DCE refers to communications equipment such as modems that are responsible for transferring data. (All RS232 pin function definitions are from DTE point of view)

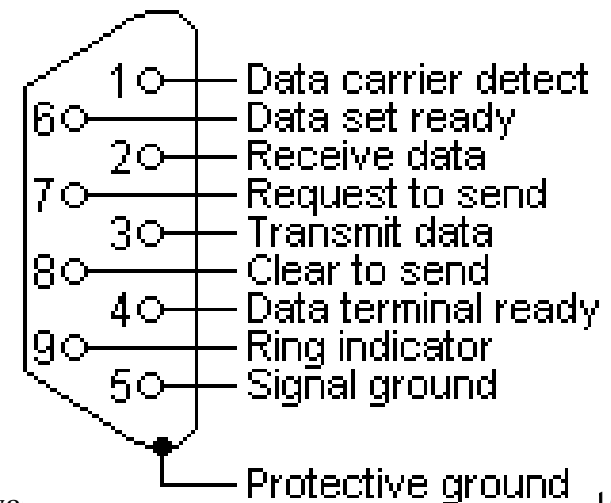
Handshaking is via two levels - Device-level & Data-level

Device-level - \DTR, \DSR

Data-level- \RTS, \CTS

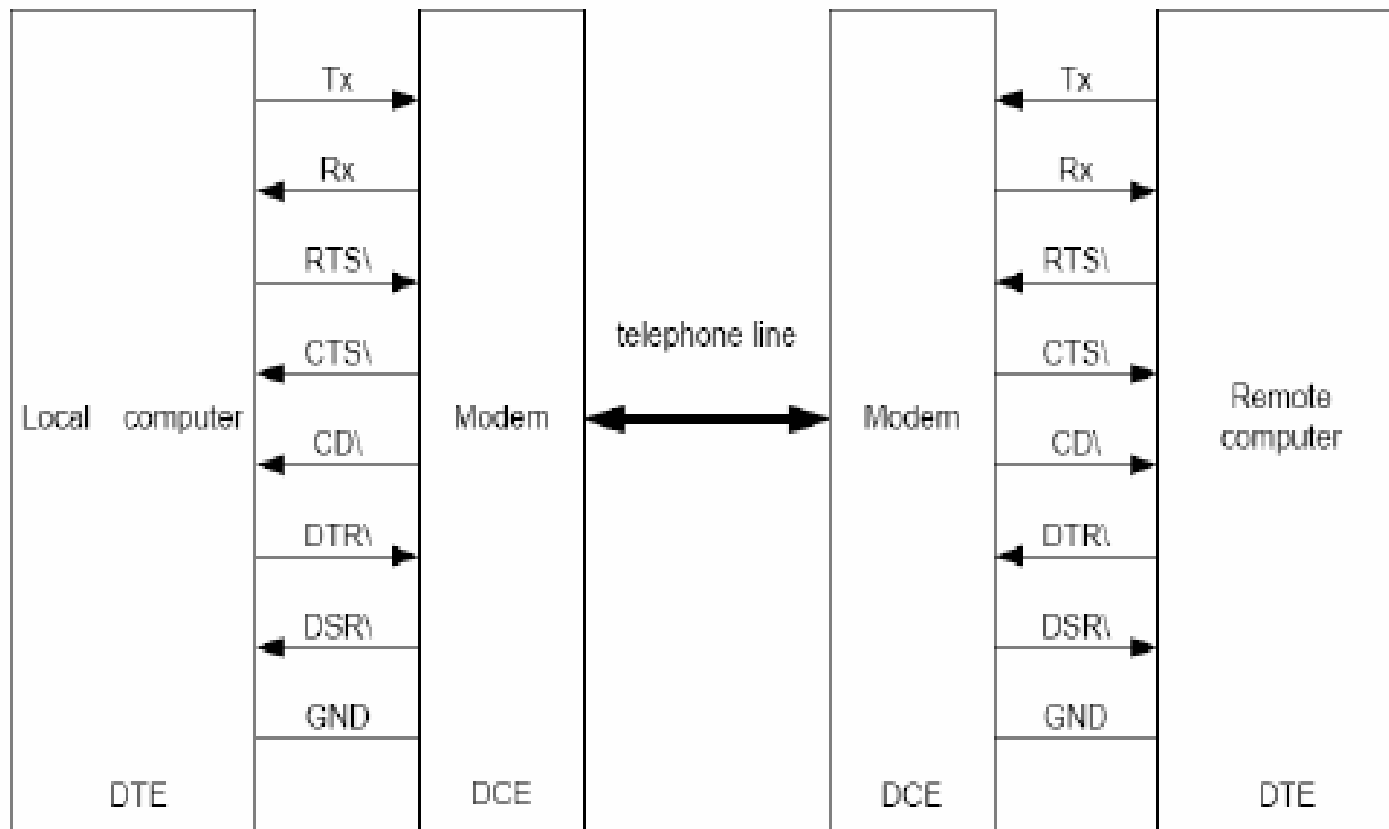
Output Pins: TxD, \DTR, \RTS

Input Pins: RxD, \CTS, \DCD, \DSR



The pin outs of both D-9 & D-25 are shown below.

D-Type-9 pin no.	D-Type-25 pin no.	Pin outs	Function
3	2	RxD	Receive Data (Serial data input)
2	3	TxD	Transmit Data (Serial data output)
7	4	\RTS	Request to send (acknowledge to modem that UART is ready to exchange data)
8	5	\CTS	Clear to send (i.e.; modem is ready to exchange data)
6	6	\DSR	Data ready state (UART establishes a link)
5	7	GND	Signal ground
1	8	\DCD	Data Carrier detect (This line is active when modem detects a carrier)
4	20	DTR	Data Terminal Ready.
9	22	RI	Ring Indicator (Becomes active when modem detects ringing signal from PSTN)



DTE-DCE connection showing all the signal lines

Handshaking Signals and Definitions are explained below

Group 1: pure data transmission

- Tx: Transmit Data - used for sending data.
- Rx: Receive Data - used for receiving data.
- GND: Signal Ground - This pin is the same for DTE and DCE devices, and it provides the return path for both data and hand-shake signals.

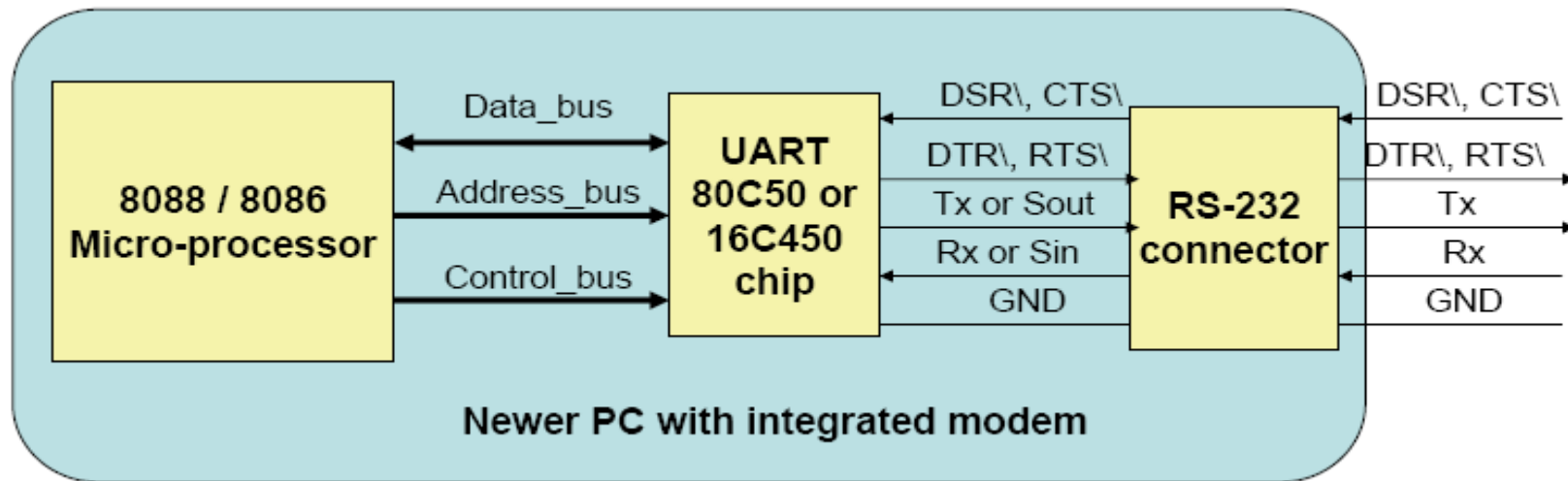
Group 3: others

- CD\: Carrier Detect - Used by a DCE to indicate to the DTE that it has received a carrier signal from the modem and that real data is being transmitted. (not used in our projects)
- RI: Ring Indicator - Used by DCE modem to tell the DTE that the phone is ringing and that data will be forthcoming. (not used in our projects)

Group 2: hand-shaking signals

- DTR\: Data Terminal Ready - Used by a DTE (Data Terminal Equipment) to signal that it is plugged in and available to begin communication.
- DSR\: Data Set Ready – ‘Sister’ signal to DTR, it is used by the DCE (Data Communication Equipment) to indicate it is also ready to begin communication, and used in response to a DTR.
- RTS\: Request to Send - Used by a DTE to indicate that it wants to send data. Also, in a multi-drop network, this signal is used to turn carrier on the modem on and off.
- CTS\: Clear to Send - Used by a DCE to signal it is also available to send data, and used in response to a RTS request for data.

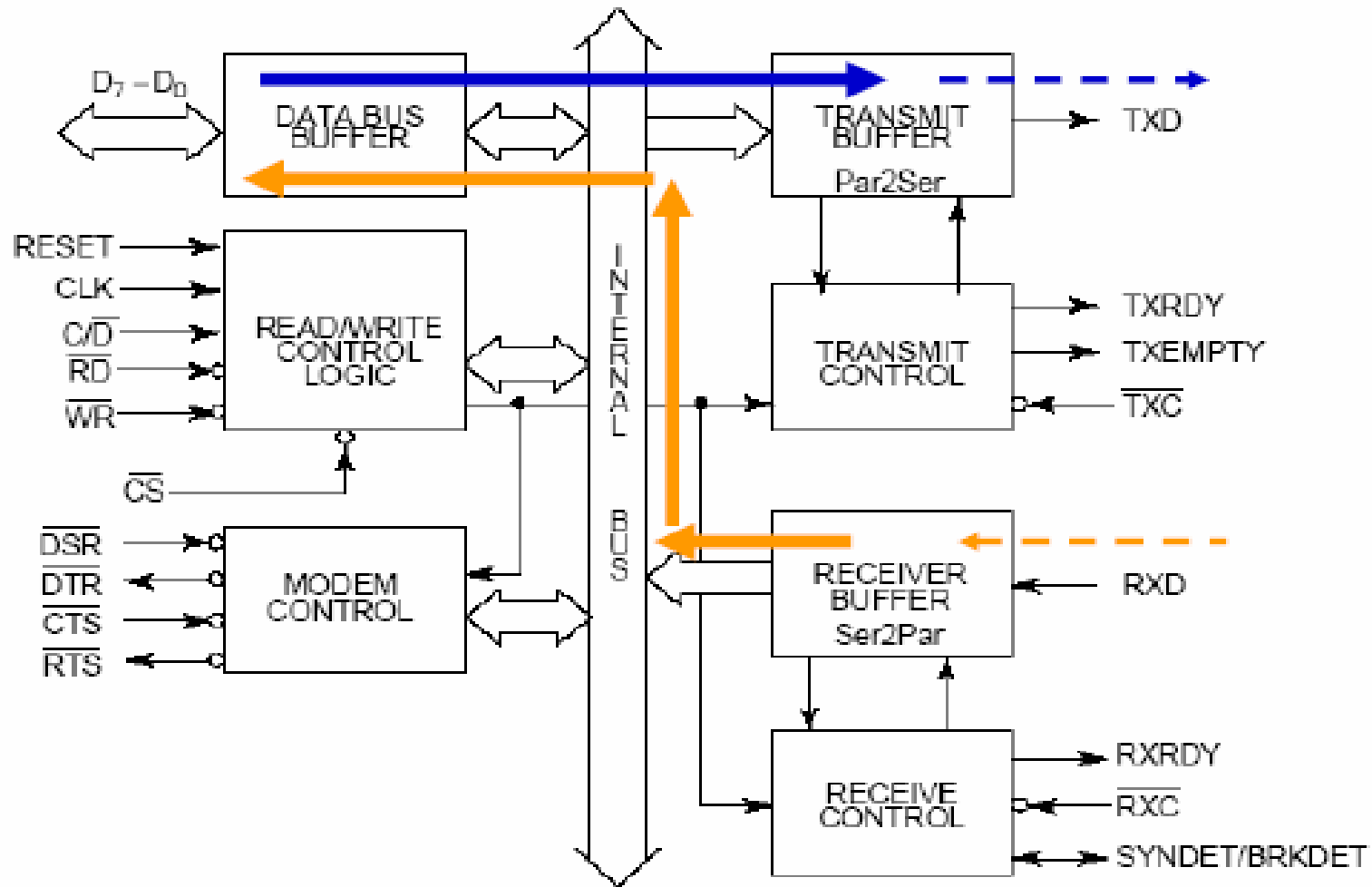
8086 - UART - RS232 Connection



Serial communications are carried out using special chips made by manufacturers. These are referred to as UART (Universal Asynchronous Receiver-Transmitter) and USART (Universal Synchronous-Asynchronous Receiver-Transmitter).

Your COM port in the IBM PC uses 8250 UART.

UART Block Diagram



How to connect

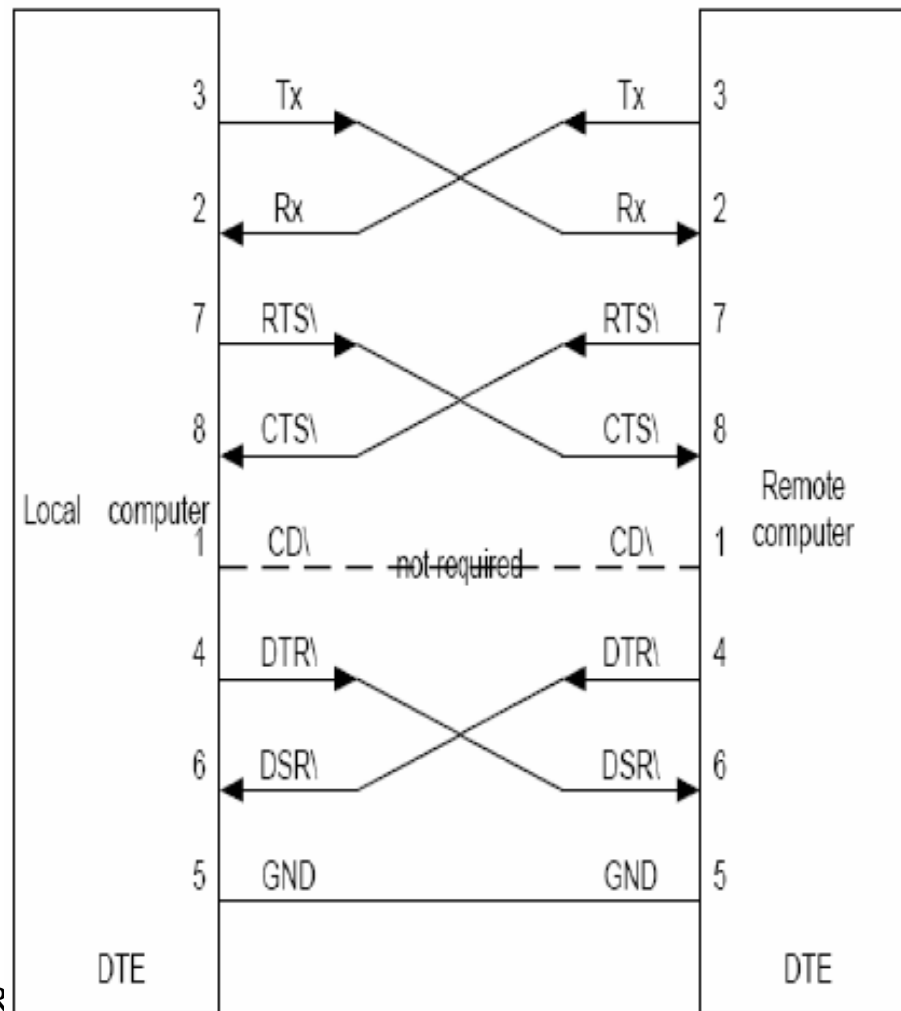
- In lab we will connect two PCs together, not a PC to a modem.

Handshaking is via two levels -

Device-level & Data-level
(see flow chart)

Device-level - \DTR, \DSR

Data-level- \RTS, \CTS



(c) All Rights Reserved

2010

UART needs to be programmed when you want to transmit and receive data. There are 10 registers that bear the following description.

Base Address COM1=3F8h COM2=2F8h	DLAB	Read/Write	Abbr.	Register Name
+ 0 = 3F8h/2F8h	=0	Write	THR	Transmitter Holding Register
	=0	Read	RBR	Receiver Buffer Register
	=1	Read/Write	DLL	<u>Divisor Latch Low Byte</u>
+ 1 = 3F9h/2F9h	=0	Read/Write	IER	Interrupt Enable Register
	=1	Read/Write	DLM	<u>Divisor Latch High Byte</u>
+ 2 = 3FAh/2FAh	-	Read	IIR	Interrupt Identification Register
	-	Write	FCR	FIFO Control Register
+ 3 = 3FBh/2FBh	-	Read/Write	LCR	Line Control Register
+ 4 = 3FCh/2FCh	-	Read/Write	MCR	Modem Control Register
+ 5 = 3FDh/2FDh	-	Read	LSR	Line Status Register
+ 6 = 3FEh/2FEh	-	Read	MSR	Modem Status Register
+ 7 = 3FFh/2FFh	-	Read/Write	-	Scratch Register

8250 UART is a **40-pin chip** with an 8-bit data bus. It receives a single character from CPU, frames it, then transmits it serially.

In the same way, it can receive serial data, strip away the start and Stop bits, make a character out of it and present it to the CPU.

It generates all necessary handshaking signals.

Data Registers - Receiver Buffer Register (RBR), Transmitter Holding Register (THR), Scratchpad Register (SCR);

Control Registers - Line Control Register (LCR), Modem Control Register (MCR), Interrupt Control Register (IER);

Status Registers - Line Status Register (LSR) and Modem Status Register (MSR)

THR (DLAB = 0; A2 A1 A0 = 000) : To transfer a byte serially, CPU must Write to this register; In this case, DLAB bit of the LCR must be 0; Transfer of data happens via Sout pin;

RBR (DLAB = 0; A2 A1 A0 = 000) : When 8250 receives the data through Sin pin, it strips away the headers and makes a byte; It holds this Byte in this register for CPU to read it.

LCR (A2 A1 A0 = 011) : Framing information is sent to this register; See the figure on the next slide;

Example:

Let us say we want to send 7 bits of data with 1 stop bit and with odd Parity case; DLAB = 0 with no break control; Let the Port address be 3FB H. We program like this:

0000 1010 = 0A H for the data format register;

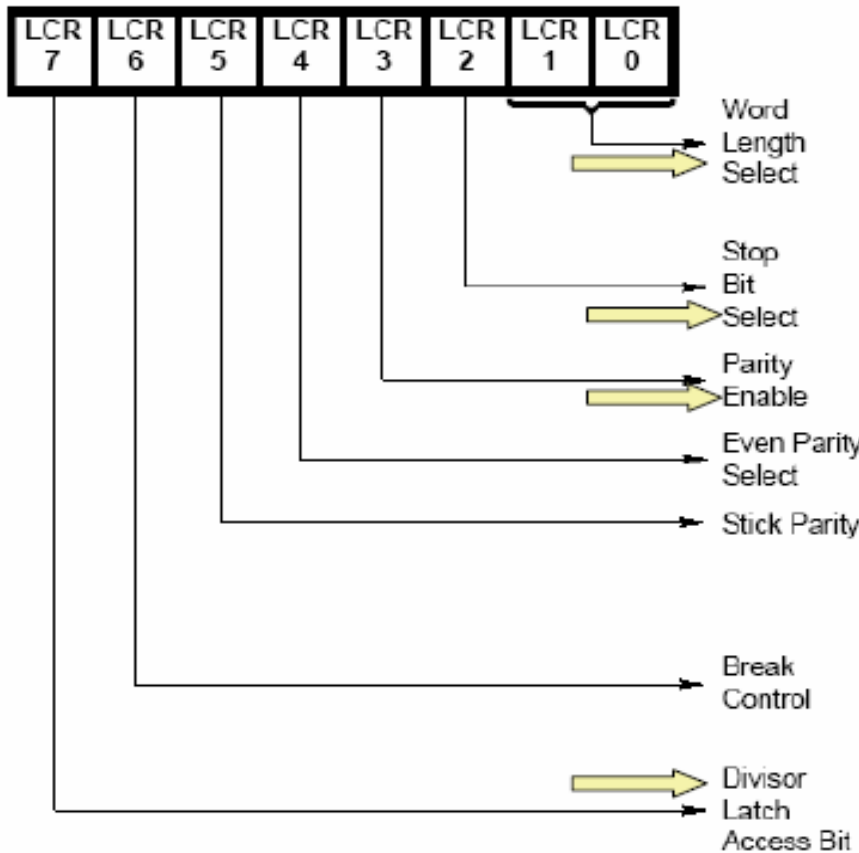
Mov DX,3FBH

Mov AL, 0AH

Out DX,AL

Line Control Register

Line Control Register (LCR)



- LCR controls the format of the data character.

0 0 = 5 Data Bits
 0 1 = 6 Data Bits
 1 0 = 7 Data Bits
 1 1 = 8 Data Bits

0 = 1 Stop Bit
 1 = 1.5 Stop Bits if 5 Data Bit Word Length is Selected
 2 Stop Bits if 6, 7, or 8 Data Bit Word Length is Selected

0 = Parity Disabled
 1 = Parity Enabled (Generated & Checked)

0 = Odd Parity When Parity is Enabled
 1 = Even Parity When Parity is Enabled

0 = Stick Parity Disabled
 1 = When Parity is Enabled Forces the Transmission and Checking of a Parity Bit of a Known State. Parity Bit Forced to a Logic 1 if LCR (4) = 0 or to a Logic 0 if LCR (4) = 1.

0 = Break Disabled
 1 = Break Enabled. The Serial Output (SOUT) is Forced to the Spacing (Logic 0) State.

0 = Must be Low to Access the Receiver Buffer, Transmitter Holding Register or the Interrupt Enable Register.
 1 = Must be High to Access the Divisor Latches DLL and DLM of the Baud Rate Generator During a Read or Write Operation.

LSR provides status indications - first register for CPU to read and to determine the cause of an interrupt;

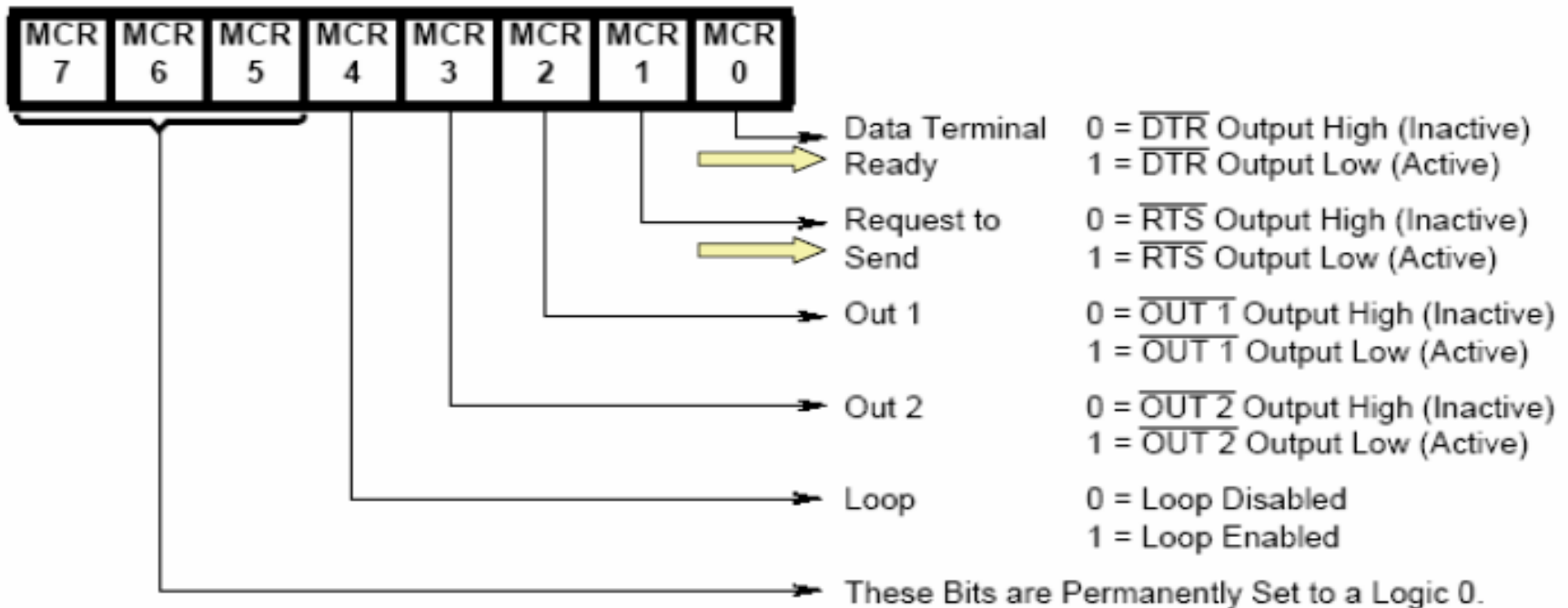
LSR BITS 0 THRU 7

	LOGIC 1	LOGIC 0
→ LSR (0) Data Ready (DR)	Ready	Not Ready
LSR (1) Overrun Error (OE)	Error	No Error
LSR (2) Parity Error (PE)	Error	No Error
LSR (3) Framing Error (FE)	Error	No Error
LSR (4) Break Interrupt (BI)	Break	No Break
→ LSR (5) Transmitter Holding Register Empty (THRE)	Empty	Not Empty
LSR (6) Transmitter Empty (TEMT)	Empty	Not Empty
LSR (7) Not Used		

Modem Control Register

- MCR controls the interface with the modem or data set.

Modem Control Register (MCR)



Divisor Latch Register (LSB & MSB)

Example: For a divisor latch for 300 baud determine the divisor value and send the divisor value; 3FB(address of LCR) ; 3F8(address of DLR)
 $X_{in} = 1.8432 \text{ MHz}$

; set the LCR to 1 for accessing DLAB first

```
MOV AL, 80H ; to access DLAB(1000 0000)
MOV DX, 3FB ; address of LCR
OUT DX, AL
```

; now send the divisor value

Divisor value = Clock Freq (X_{in}) / (16 × baud rate)

```
MOV AX, 384 ; 384 is the divisor value for 300 baud rate
MOV DX, 3F8 ; Address of DLR (LSB)
OUT DX, AL ; issue the lower byte
MOV AL, AH
INC DX ; Address of DLA (MSB)
OUT DX, AL
```


Modem Status Register - MSR provides the CPU the status of the modem input lines from the modem

MSR BITS 0 THRU 7

MSR BIT	MNEMONIC	DESCRIPTION
MSR (1)	DDSR	Delta Data Set Ready
MSR (2)	TERI	Trailing Edge of Ring Indicator
MSR (0)	DCTS	Delta Clear To Send
MSR (3)	DDCD	Delta Data Carrier Detect
MSR (4)	CTS	Clear To Send
MSR (5)	DSR	Data Set Ready
MSR (6)	RI	Ring Indicator
MSR (7)	DCD	Data Carrier Detect

APPENDIX - 2

All About Segments.....

Recall all the workings of segments and the role of the two important pointers in every segment....

Code Segment (CS, IP)

This is the space where (y)our code dwells physically. This is the space from where 8086 fetches instructions to execute.

CS holds code segment address and IP holds the offset

Physical memory is of 20 bits, and the offset and segment addresses are 16 bits each. We need to generate 20 bits address to go to the desired physical address. How do we do this? Here is an example.

Example

CS = 2500; IP = 95F3 [Often this combination is written as CS:IP]

Start with CS 2500

Shift left CS 25000

Add IP 2E5F3 → Physical address

This 20 bits address is what will be put on the address bus to be decoded by the memory decoding circuitry.

Example

Logical address (CS:IP)	Machine language code	Assembly language code
1132:0100	B057	MOV AL,57
1132:0102	B686	MOV DH,86
1132:0104	B272	MOV DL, 72

Data Segment (DS, BX / DI / SI)

This is where you can define your data. Data can be mixed with code, however, if the data changes then the code has to be changed frequently.

Therefore we can have,

Example

DS:0200 = 25 Program can now be written as
DS:0201 = 12
DS:0202 = 15 ADD AL, [0200] The contents of the memory locations
 ADD AL, [0201] ← are written like this in programming
 ADD AL, [0202]

Example

DS=5000; Offset=1950 Calculate the physical address **Ans: 51950**

Same idea here as we have seen before – left shift DS and add the offset

Extra Segment (ES, BX/DI/SI)

This is just an additional data segment that comes very handy while manipulating strings

Stack Segment (SS, SP)

Useful for stack operations. Stack is **LIFO** by its virtue of operation

Every register inside 80x86 (except segment registers and SP) can be stored (“pushed” is the right terminology) in stack They can then be brought back (“popped” is the right terminology) at any time we want in the program.

*Now that we have learnt the ASM, do we know **how our instructions actually look like** ?*

A machine language instruction usually has one or more fields associated With it.

First field – operation code field/ opcode field- type of operation to be performed

Other fields – operand fields

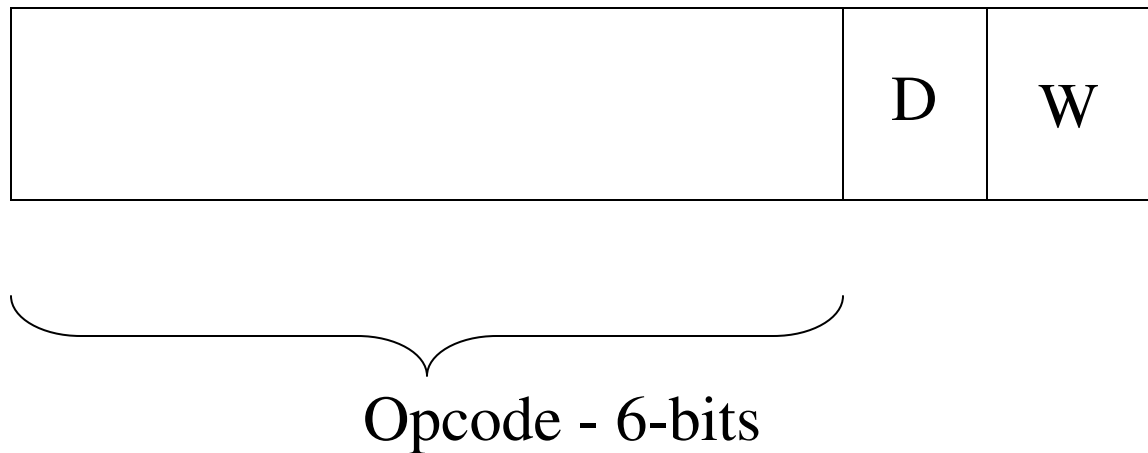
- Six (6) general formats of instructions exists for 8086 processor. Further, length of an instruction may vary between ONE BYTE to SIX BYTES.

Instruction set for 8086 takes the following form

Just an info: Compatibility exists between 8086 and 80X86 processors, if the latter is programmed to operate with 16-bits. 80386 and processors above, Operate in real mode and protected mode. In real mode, they use 16-bit and in protected mode, they use 32-bit.

Opcode 1-2 bytes	MOD-REG-R/M 0-1 bytes	DISP 0-1 bytes	Immediate 0-2 bytes
---------------------	--------------------------	-------------------	------------------------

Opcode: Following figure shows the first opcode byte for many, But not all, instructions.



D – direction bit, W- word/byte. If D=1, data flow to the REG field from R/M field located in the second byte. If D=0, it is the other way. If W = 1, data size is a double or a word and if W=0, the data size is a byte.

MOD field – specifies the addressing mode for the instruction; Selects The type of addressing and whether a displacement is present or not.

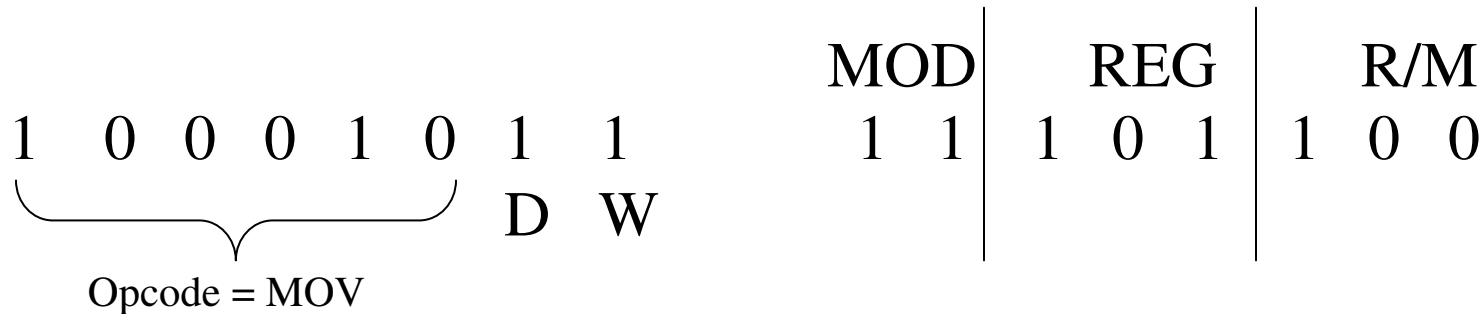
<u>MOD</u>	<u>Function</u>
00	No displacement
01	8-bit sign-extended displacement
10	16-bit displacement
11	R/M is a register

REG and R/M fields, when MOD = 11

<u>Code</u>	<u>W = 0(byte)</u>	<u>W = 1(word)</u>
000	AL	AX
001	CL	CX
010	DL	DX
011	BL	BX
100	AH	SP
101	CH	BP
110	DH	SI
111	BH	DI

Machine Codes

Example MOV BP, SP



Note that D and W bits are logic 1, which means that a word moves into the destination register specified by REG field. The REG field contains – 101, indicating that register is BP, so the MOV will move the data into register BP. Here, R/M = 100 (SP), therefore this instruction will move the data from SP to BP and is written in symbolic form MOV BP, SP instruction. This is equal to 8BEC.

Note: If MOD = 00, 01, or 10, R/M takes a new meaning

Take a look at Table 2.2 on Page 36 in your prescribed book.

Example MOV DL, [DI]

1	0	0	0	1	0		1	0		0	0		0	1	0		1	0	1	= 8A15
							d	w												

So, here $D = 1$ and hence, data flow is from R/M to REG. $W = 0$ implying that the data is a byte. $MOD = 00$, indicating that there is no displacement. $REG = DL = 010$, and $R/M = 101$

If instruction changes to MOV DL, [DI+1], MOD changes to 01 for an 8-bit displacement; Thus we have 8A5501 as the instruction code. Note that the 8-bit displacement appends to the two byte instruction to form a 3-byte instruction.

Useful exercise – Follow examples given on Pages 82 – 85 in Ray's book

Operands		Memory Operands		
		No Displacement	Displacement 8-bit	Displacement 16-bit
MOD	R/M	00	01	10
000		(BX) + (SI)	(BX) + (SI) + D8	(BX) + (SI) + D16
001		(BX) + (DI)	(BX) + (DI) + D8	(BX) + (DI) + D16
010		(BP) + (SI)	(BP) + (SI) + D8	(BP) + (SI) + D16
011		(BP) + (DI)	(BP) + (DI) + D8	
100		(SI)	(SI) + D8	(SI) + D16
101		(DI)	(DI) + D8	(DI) + D16
110		D16	(BP) + D8	(BP) + D16
111		(BX)	(BX) + D8	(BX) + D16

- Note:* 1. D8 & D16 represent 8 and 16 bit displacements respectively.
 2. The default segment for the addressing modes using BP and SP is SS
 addressing modes the default segments are DS or ES.

When a data is to be referred as an operand, DS is the default data segment register for storing program codes (executable code). SS is the default segment register for the stack data accesses and operations. ES is the default segment register for the destination data storage. All the segments available in the 8086 are DS, SS, ES, FS, GS and they are used to store data segments by newly defined.

Some more details on h/w...

APPENDIX - 4

- In minimum mode, Pin 32 outputs *Read* (RD) signal - data reading process of the microprocessor. Thus, RD='0' the data read operation and continues until RD='1'.
- In minimum mode, Pin 29 outputs *Write* (WR) signal to control the writing process of the microprocessor. Thus, to initiate and maintain the writing process, CPU needs to output WT='0' and RD='1' signals
- Pin 27 outputs *Data transmit/receive* (DT/R) signal to control the direction of data transfer mode (receive/transmit) in the 'Data-BusTransceiver-buffer IC'.

- In the 8086/8088 processors, the power is applied between Pin 40 (VCC) and Pins 1 (GND) or 2 (GND).
- At room temperature (25 °C), the value of VCC is specified to be +5V DC with a tolerance of $\pm 10\%$ and the maximum current drawn by the processor is 340 mA.
- The minimum (during stand-by mode) and maximum power dissipated by 8086/8088 processors are 0.0025 watt and 1.28 watt, per unit time interval, respectively.
- The operating temperature for 8088/8086 ranges from 0 - 80 °C

The clock speed supported by the 80X86 family are:

- Standard 8086 operate at 5 MHz clock speed

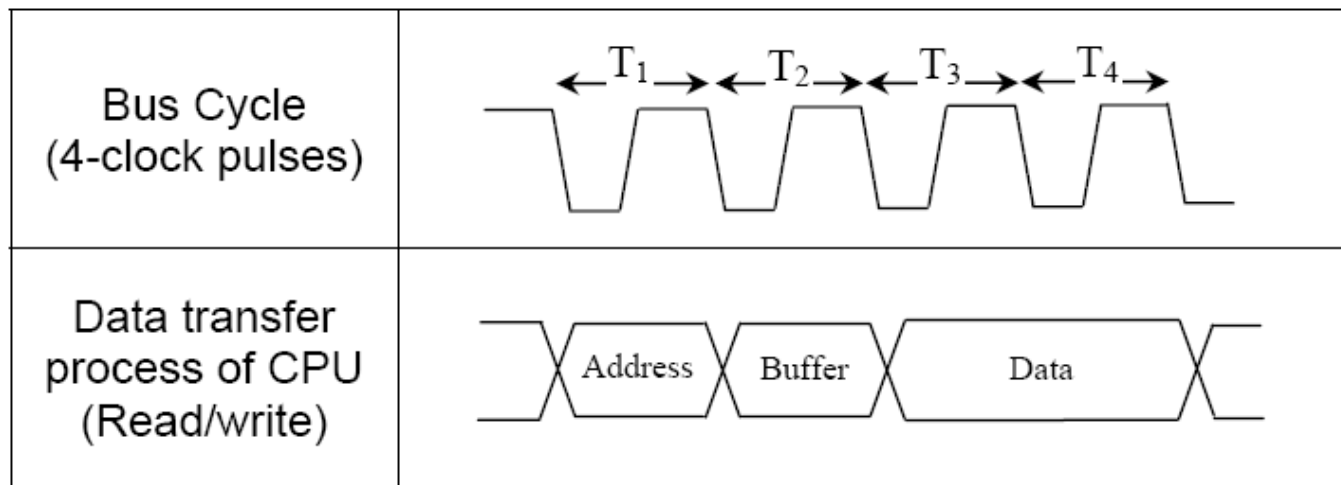
Standard 8088 operate at 5 MHz

The CLK signal is externally generated by 8284 clock generator and driver IC and fed into the processor via Pin number 19.

8284 outputs the crystal frequency (15-24MHz) as oscillator (OSC) frequency; One third of crystal frequency (5MHz) is supplied as clock (CLK) frequency to 8086 via Pin 19;

Bus-Cycle and Time States of 8086/8088 processor:

- A bus-cycle defines the basic operation process of a microprocessor to communicate with external devices, *such as, memory read bus-cycle, where data stored in main memory is read into the internal registers of the CPU (such as AX).*
- Typically, the bus-cycle of the 8086 and 8088 processors consist of four clock cycles or pulses. Thus, duration of a bus-cycle is = ' $4 \times T$ '
- A bus-cycle involving a data transfer operation between the CPU and external storage device (memory) is shown in the figure below.



Even and Odd addressed memory – What is it?

This means how the memory words are stored and accessed by the CPU!

In 8088, the entire memory is organized as a 1 piece, 1 MegaByte memory bank, with each location storing 1 byte;

In **8086**, the entire 1 MegaByte is organized into TWO 512K bytes memory banks;

“Even/Low” bank storing “even addresses” – 00000, 00002, 00004, ..., FFFFE;

“Odd/High” bank storing “odd addresses” – 00001, 00003, 00005, ..., FFFFF;

8088 – In one bus cycle one byte is accessed from an address;

8088 – To access 1 word, 2 bus cycles are consumed;

8086 – To access one byte, 8086 consumes 1 bus cycle (for even and odd banks)

8086 – for a word: If a word starts at a even address: 1 bus cycle is required to access;

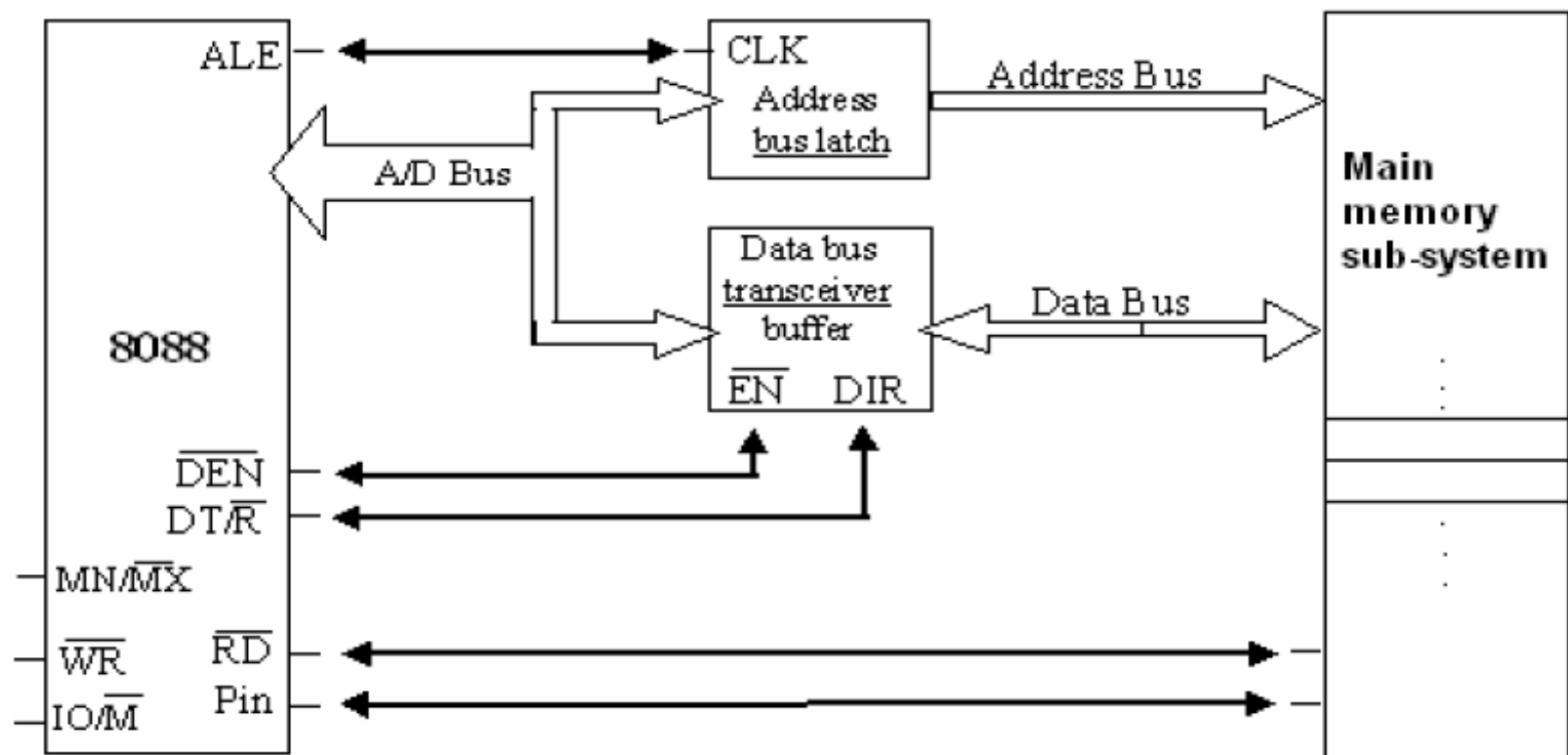
 If a word starts at an odd address: 2 bus cycles are required!!

So, in our programming, in our data segment, we need to put a Directive “EVEN” at the beginning of the data segment definition, to save some time during memory access!!

Processor-Memory Interface – *How does it look like?*

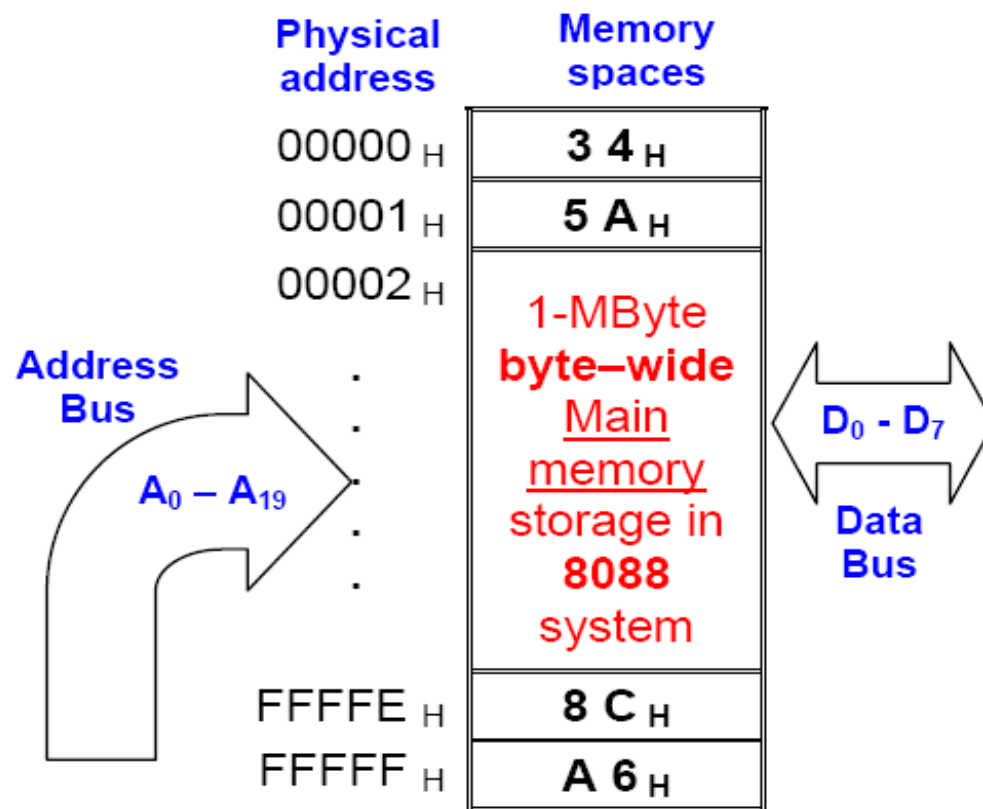
- The figure below shows the memory interface circuit of an 8088 based system operating in minimum mode.

-



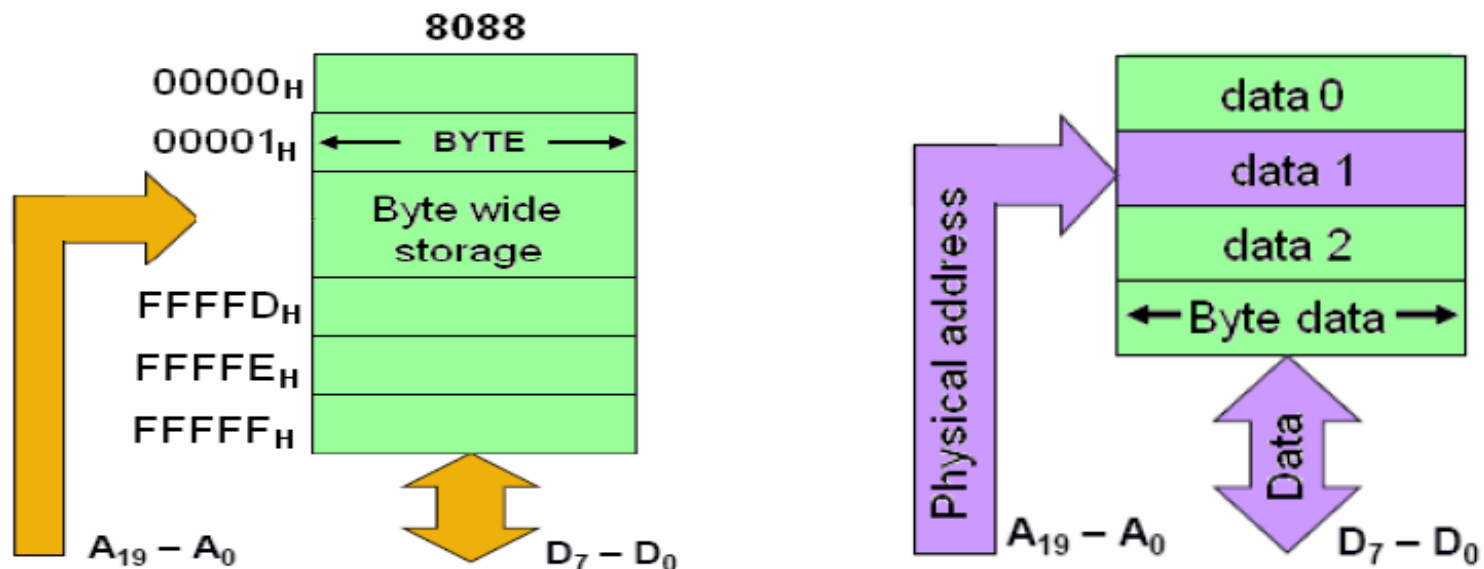
- The 8088's memory system uses a single '1-Mega X 8 bit' memory bank with Physical Address from 00000_H to FFFFF_H.

- The 20-bit (≈ 5 Hex digit) physical address supplied by the address-bus (A₀-A₁₉) is used to select the memory location required to be accessed via data-bus (D₀-D₇)



Case I - Accessing a Byte-data in a 8088-based Memory System

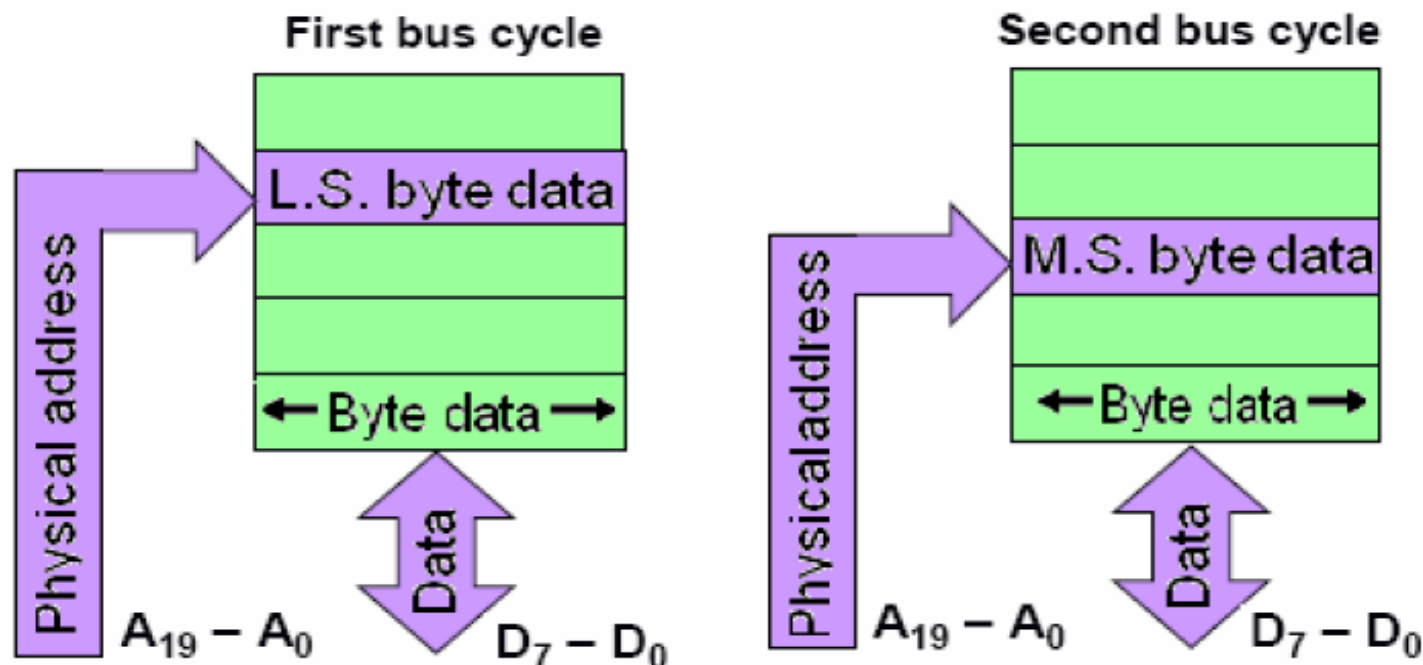
- In 8088 system, one bus cycle is required to complete the process of applying valid physical address and accessing (reading or writing) a byte data from that memory location.



In this figure, appropriate physical address is applied to access the memory location storing 'data 1'

Case II - Accessing a Word-data in a 8088-based Memory System

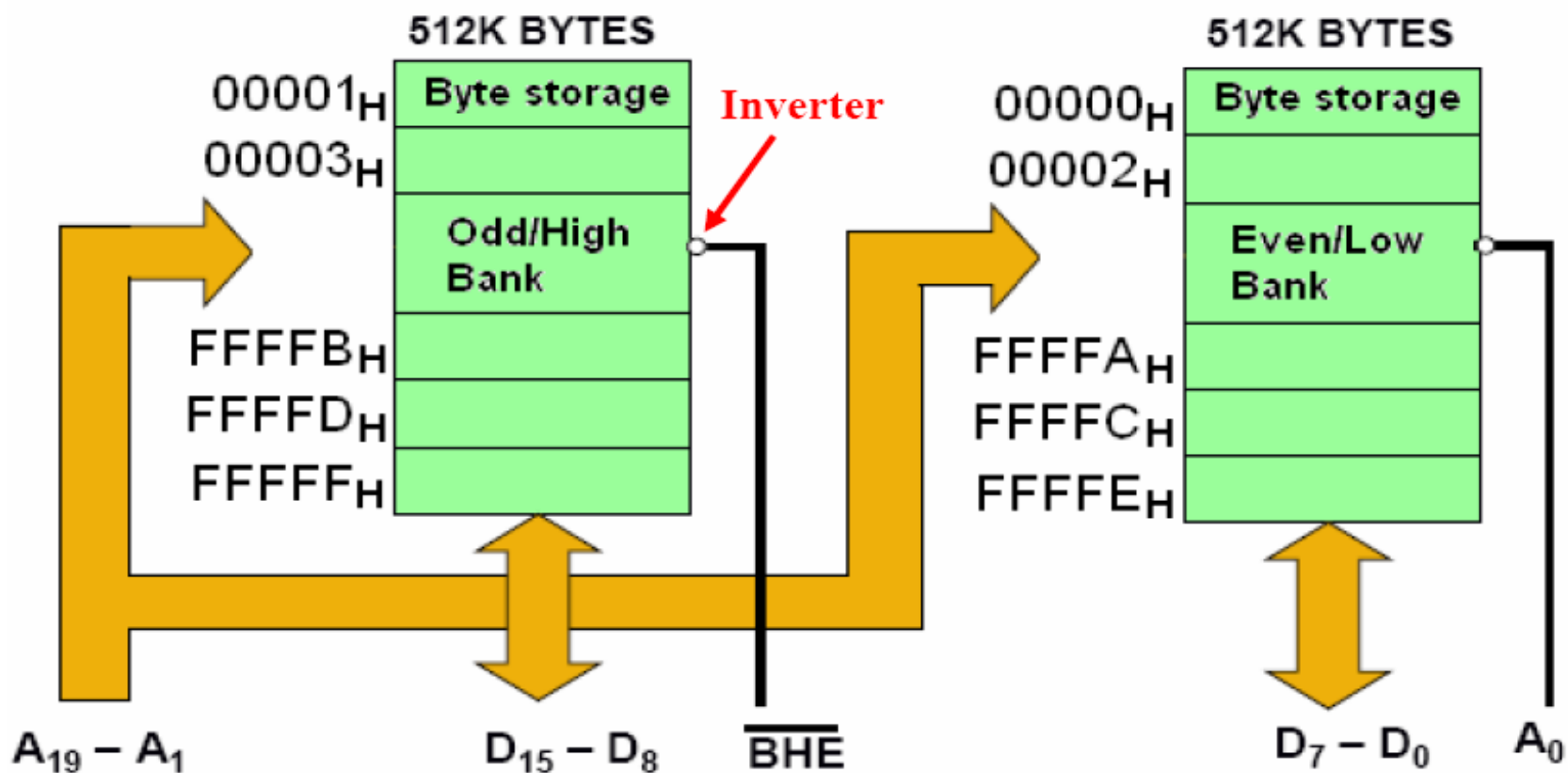
- Two bus cycles are required to access a word data in an 8088 memory system. The 1st bus cycle access the least significant byte (LSB) of the stored word, as show in figure below.
- During the 2nd bus cycle, the physical address is automatically incremented to access the most significant byte of the word.



What about 8086 ?

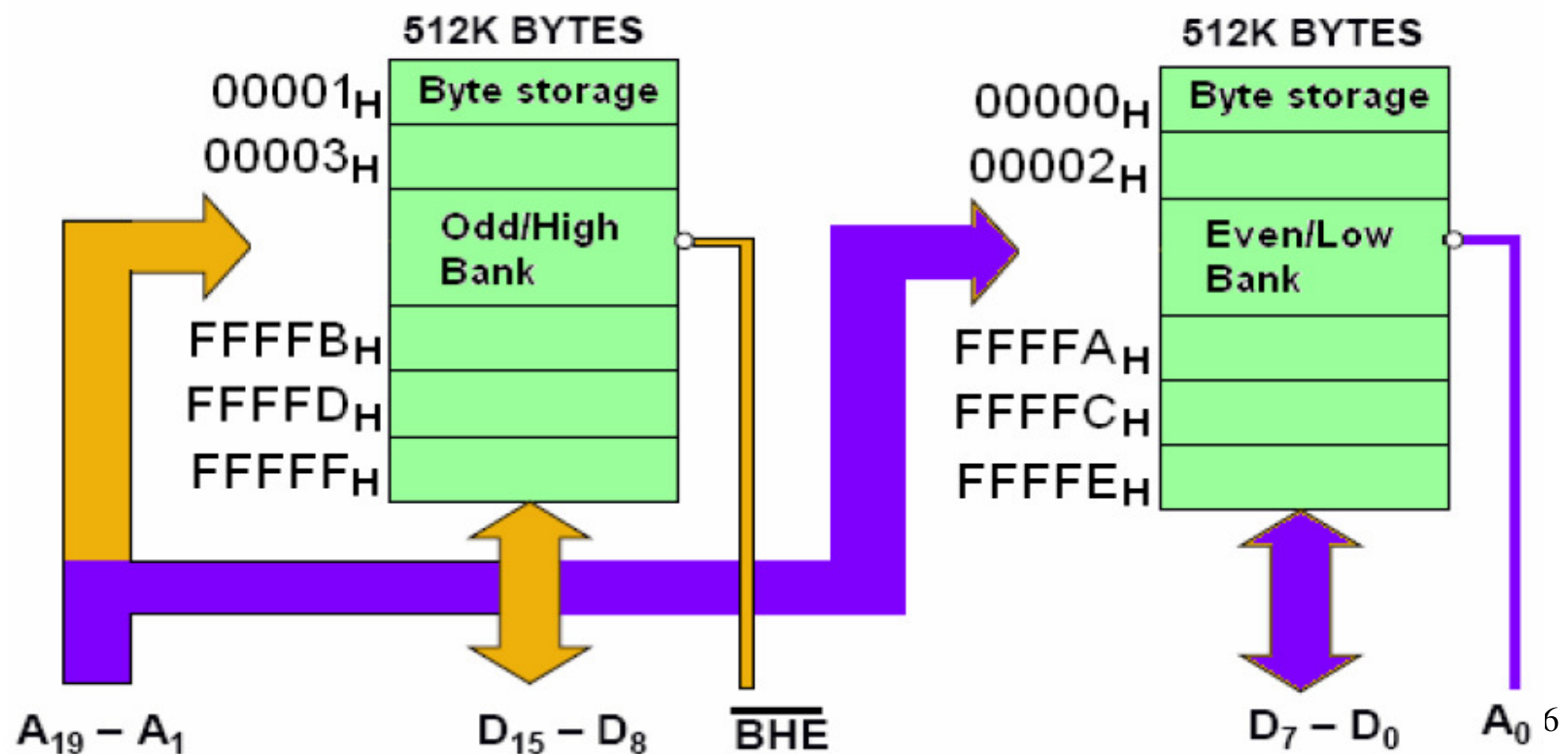
The 8086 systems have two '0.5 Mega X 8 bit' memory banks

Data-bytes associated with even-addresses reside in low bank and odd-address reside in high bank. Address pins A_1 to A_{19} selects the storage locations, whereas A_0 and \overline{BHE} pins are used to enable high or low memory banks.



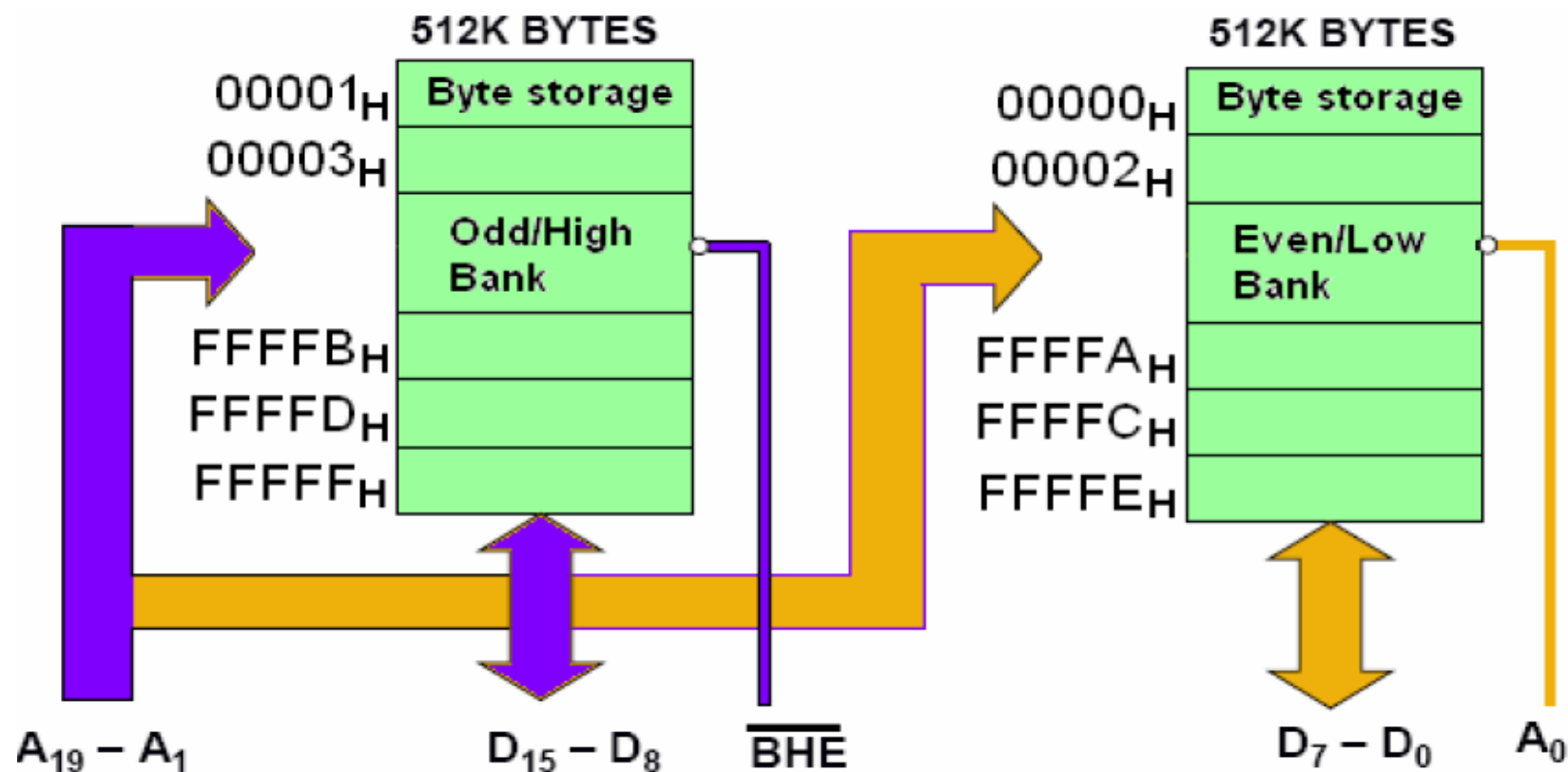
Case III - Accessing a Byte-data in a 8086-based Memory System

- The two bank memory modules of 8086 based storage system requires one bus-cycle to read/write a data-byte.
- To access a Byte of data in Low-bank, valid address is provided via address pins A_1 to A_{19} together with $A_0 = '0'$ and $\overline{BHE} = '1'$.



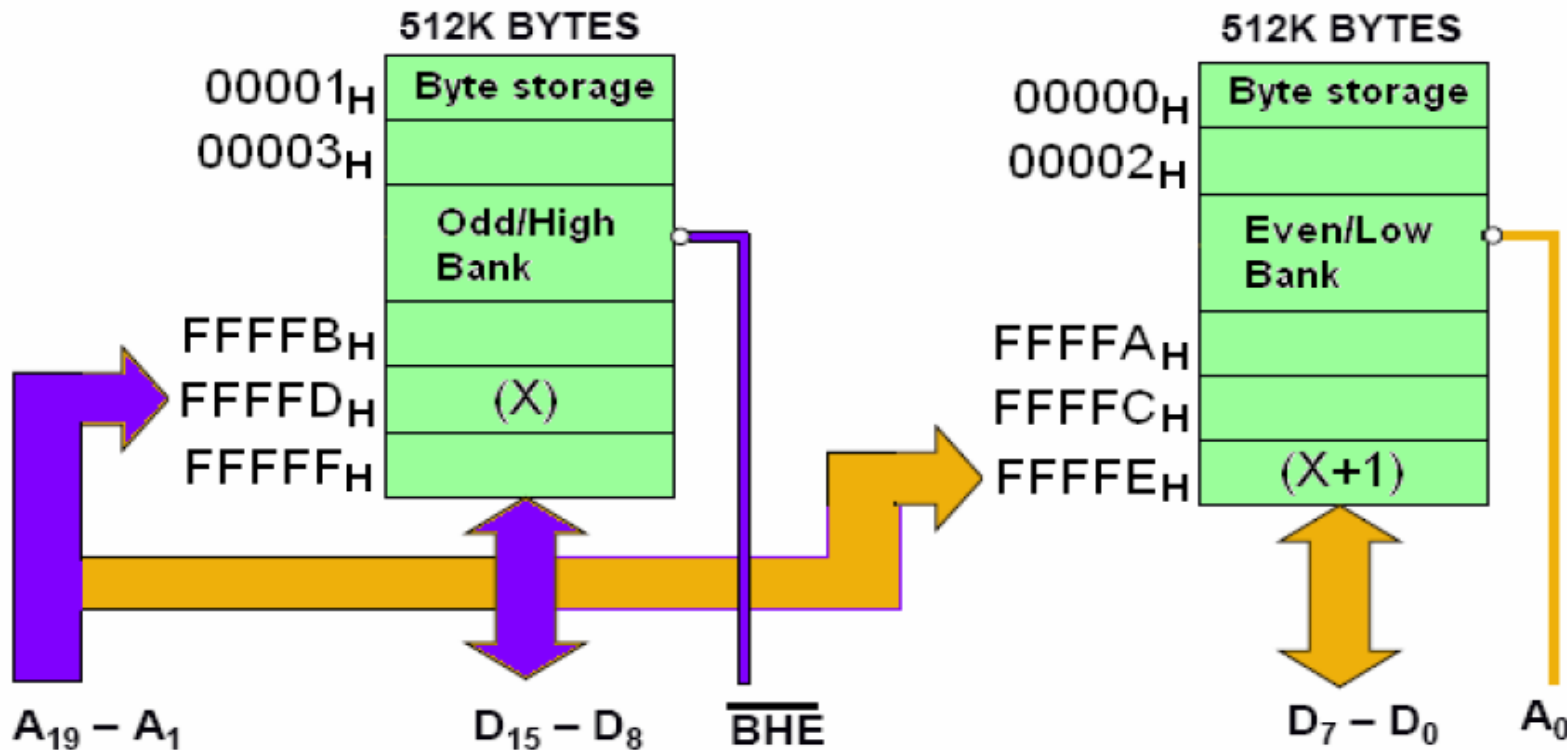
Accessing a Byte-data in a 8086 System (Cont'd):

- Similarly to access a Byte of data in High-bank, valid address in pins A_1 to A_{19} , $A_0=1$ and $\overline{BHE}=0$ are required to access the data through D_8 to D_{15} of the data-bus.
- These signals disable the Low bank and enable the High bank to transfer (in/out) data through D_8 to D_{15} of the data-bus.



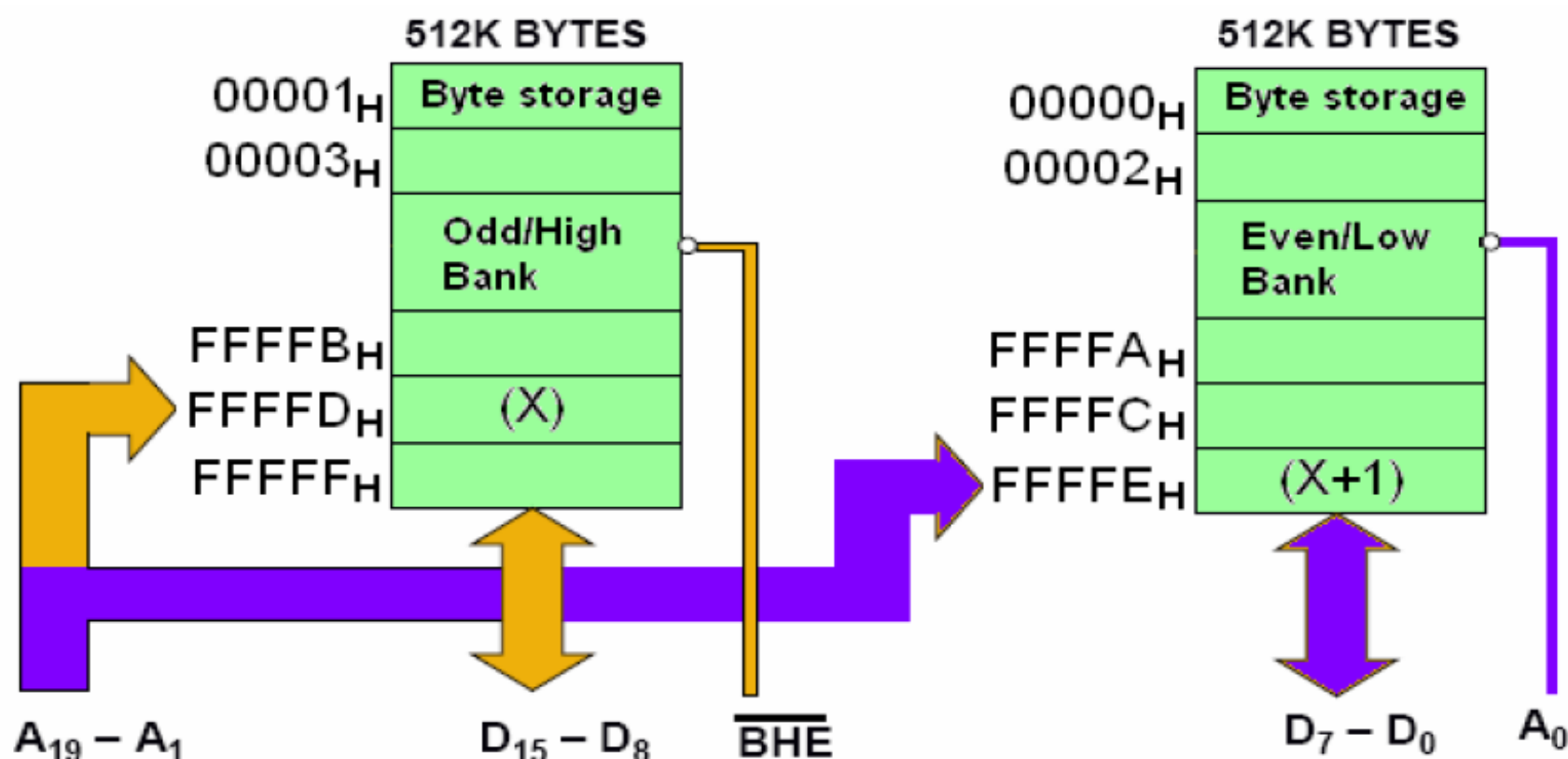
Case IV - Accessing an Unaligned Word-data in a 8086 System

- For odd-addressed (unaligned) words (with odd P.A of the LSB), two bus-cycles are required to access the Word-data.
- During the 1st bus-cycle, odd addressed LSB of the word is accessed from the High-memory -bank via D₈ to D₁₅ of data bus



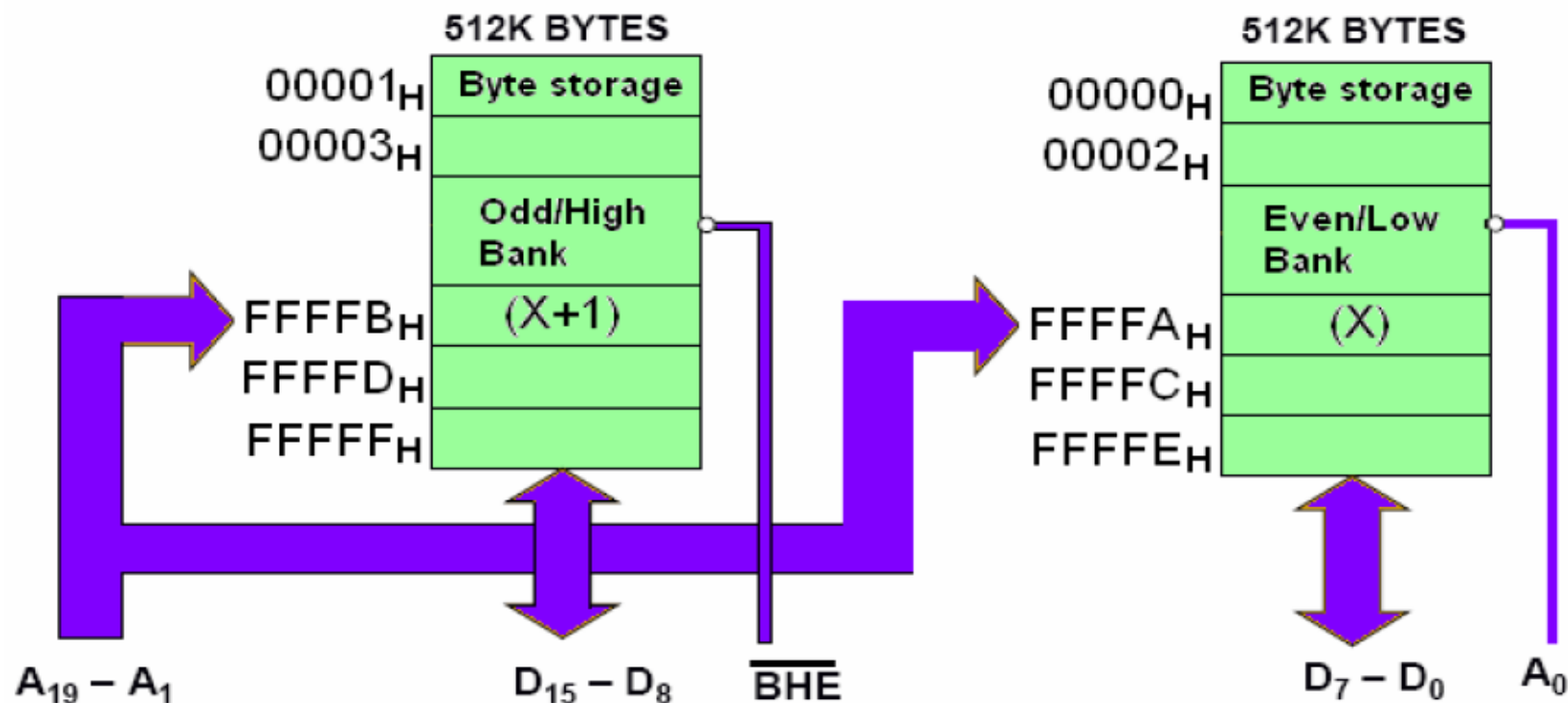
Accessing an Unaligned Word-data in a 8086 System (Cont'd)

- During 2nd bus-cycle, P.A. is auto-incremented to access the even address MSB of the word from the Low bank via D₀ to D₇.
- Note that **A₀** and **BHE** signals are reset (violet) accordingly to enable the required memory bank.



Case V - Accessing an Aligned Word-data in a 8086 System

- For even-addressed (aligned) words, only **one bus-cycle** is needed to access the word, as both low and high banks are activated at the same time using **$A_0='0'$** and **$\overline{BHE}='0'$**
- Note that during this bus-cycle, all 16-bit data is transferred via D_0 to D_{15} of the data bus.



Other ways of passing parameters... **APPENDIX - 5**

When the procedure is in a separate source module, it can still refer to the variables directly, however, in the calling program, we must use the directive PUBLIC as,

```
PUBLIC  ARY, CNT, SUM    [in the calling program]
```

and the procedure source module must contain the directive,

```
EXTRN  ARY:WORD, CNT:WORD, SUM:WORD    [in the procedure]
```

Alternatively, ARY, CNT, and SUM could be put in some common area by placing the following code

```
DATA SEGMENT COMMON
    ARY DW 100 DUP(?)
    CNT DW ?
    SUM DW ?
DATA ENDS
```

at the beginning of both the source modules. This would cause both the source modules to share the segment DATA.

Note: If the calling program uses different variable names as NUM for ARY, N for CNT, and TOTAL for SUM, then the DATA segment in its source module would need to be defined using NUM, N, and TOTAL, respectively.

In either case, the segments DATA would be overlaid by the linking process and the first 100 words of DATA would contain numbers to be added, the next word contains the count, and the last word would hold the result.

In the above examples, whether the variables are referred to directly or by means of a shared area, both the above approaches to communicating with the FUNCADD are restricted to adding numbers in the same area of memory whenever the procedure is called.

Q: How do we go about adding numbers in a different array, say WEIGHTS?

The solution is to pass the address of the array, the count, and the result to the procedure, thus allowing it to work directly with different memory locations each time it is called. Note that even though 1000 words may be involved for adding, we need to pass only three addresses!. The variables whose addresses are passed are called **parameters**.