8086 Microprocessor Architecture

- Architecture,
- Programming model,
- Instruction pipelining,
- Memory segmentation,
- 20-bit physical address generation and
- Physical memory organization

Course Material – EC 403

By

Er. Mrs. Saraswati Saha,
Assistant Professor, ECE Department,

RCCIIT, Kolkata

8086 μp :-

SARASWATI SAHA
ASST. PROF, ECE, RCCIIT

Introduction :- The Intel 8086 is a 16-bit microprocessor intended to be used as the CPU in a microcomputer. The term " 16-bit " means that its arithmetic logic unit (ALU), internal registers, and most of its instructions are designed to work with 16-bit binary words.

The 8086 has a 16-bit data bus, so it can read data from or write data to memory and ports (I/P/O/P) either 16 bits or 8 bits at a time.

The 8086 has a 20-bit address bus, so it can address any one of 2^{20} or 1,048,576 (1 MB) memory locations. Each of the 1MB memory addresses of the 8086 represents a byte-wide location. [i.e. each location can store one-byte]. words (16-bit wide or two-byte) will be stored in two consecutive memory locations(**)

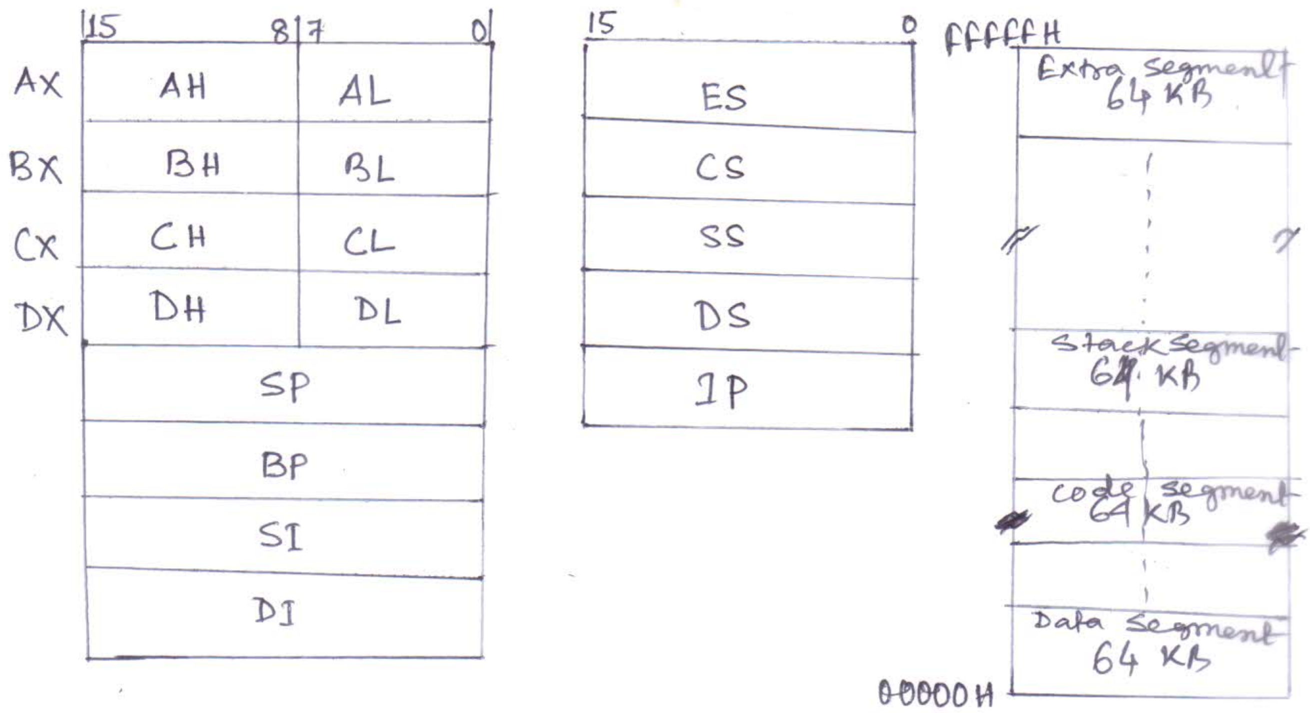
(**) → If the first byte of a word is at an even address, the 8086 can read the entire word in one operation. If the first byte of the word is at an odd address, the 8086 will read the first byte in one operation, and the second byte in another operation.

*** The Intel 8088 has the same arithmetic logic unit, the same registers, and the same instruction set as the 8086. The 8088 also has a 20-bit address bus but it has 8-bit data bus.

The Intel 80186 is an improved version of the 8086, and the 80188 is an improved version of the 8088. In addition to a 16-bit CPU the 80186 and 80188 each have programmable peripheral devices integrated in the same-package. The Intel 80286 is an advanced version of the 8086 specially designed for use as the CPU in a multiuser or multitasking microcomputer.

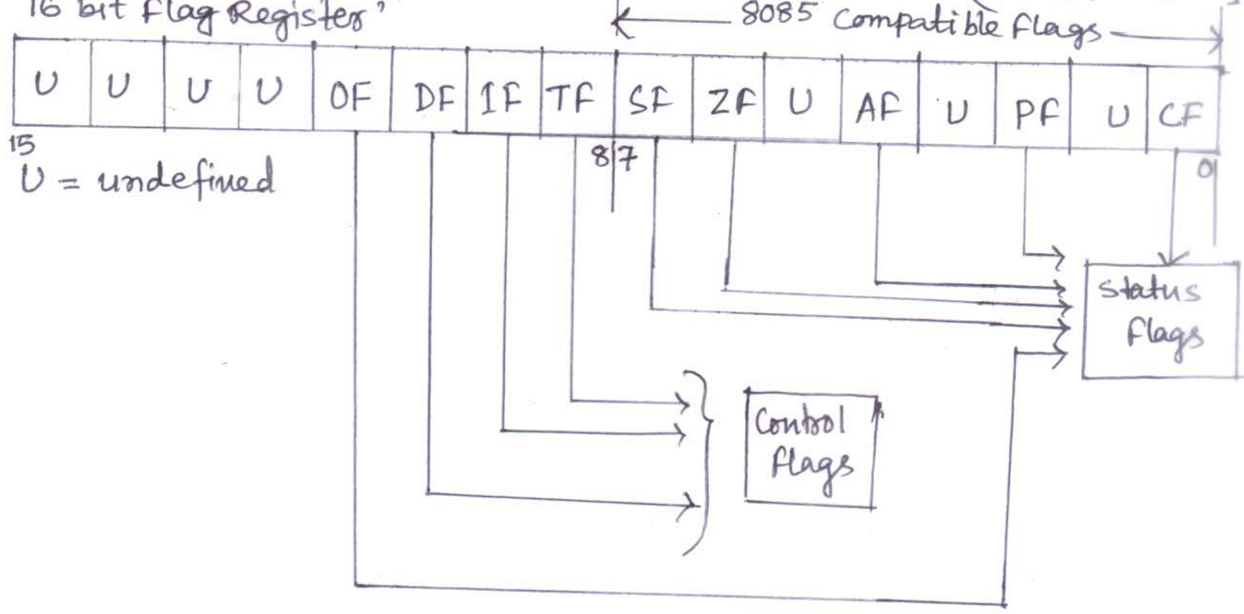
8086 programming model

SARASWATI SAHA
ASST. PROF, ECE, RCCIIT



(16 × 64 KB = 1MB)

"16 bit Flag Register"



- The BIU contains four 16-bit Segment registers. They are : the Code Segment (CS) register, the stack segment (SS) register, the extra segment (ES) register, and the data segment (DS) register.

These segment registers are used to hold the upper 16 bits of the starting addresses of four memory segments that the 8086 is working with at a particular time.

- The BIU contains another 16 bit register called Instruction pointer (IP). The IP register holds the 16-bit offset address of the next code byte within the Code Segment.

Contd...

- The EU has eight general purpose registers labeled AH, AL, BH, BL, CH, CL, DH and DL. These registers can be used individually for temporary storage of 8-bit data.

✓ The AL register is also called the accumulator for 8-bit operation. It has some features that the other general-purpose registers do not have.

Certain pairs of these general-purpose registers can be used together to store 16-bit data words. The acceptable register pairs are AH and AL, BH and BL, CH and CL, and DH and DL. These are referred to as AX, BX, CX and DX for 16-bit operations.

✓ AX is called the accumulator for 16-bit operation.

- The EU contains another 16-bit register called stack pointer (SP). This register is used to hold the 16-bit offset from the start of the stack segment to the memory location where a word was most recently stored on the stack.

- In addition to the SP register, the EU contains a 16-bit base pointer (BP) register. It also contins a 16-bit Source index (SI) register and a 16-bit destination index (DI) register. These three registers can be used for temporary storage of data just as the general purpose registers. However, their main use is to hold the 16-bit offset of a data word in one of the segments.

- EU contain a 16-bit flag register, in which 9 flag bits are active. six of the nine flags are used to indicate some condition produced by an instruction. The six conditional flags in this group are: the carry flag (CF), the parity flag (PF), the auxiliary carry flag (AF), the zero flag (ZF), the sign flag (SF), and the overflow flag. (OF)

The three remaining flags in the flag registers are used to control certain operations of the processor. The control flags are deliberately set or reset with specific instructions you put in the program. The three control flags are the trap flag (TF), which is used for single stepping through a program; the interrupt flag (IF), which is used to allow/prohibit the interruption of a prog, and the direction flag (DF), which is used with string instructions.

- The 8086 BIU sends out 20-bit addresses, so it can address any of 2^{20} or 1MB in memory. However, at any given time the 8086 only works with four, 64 KB segments within this 1MB range.

5

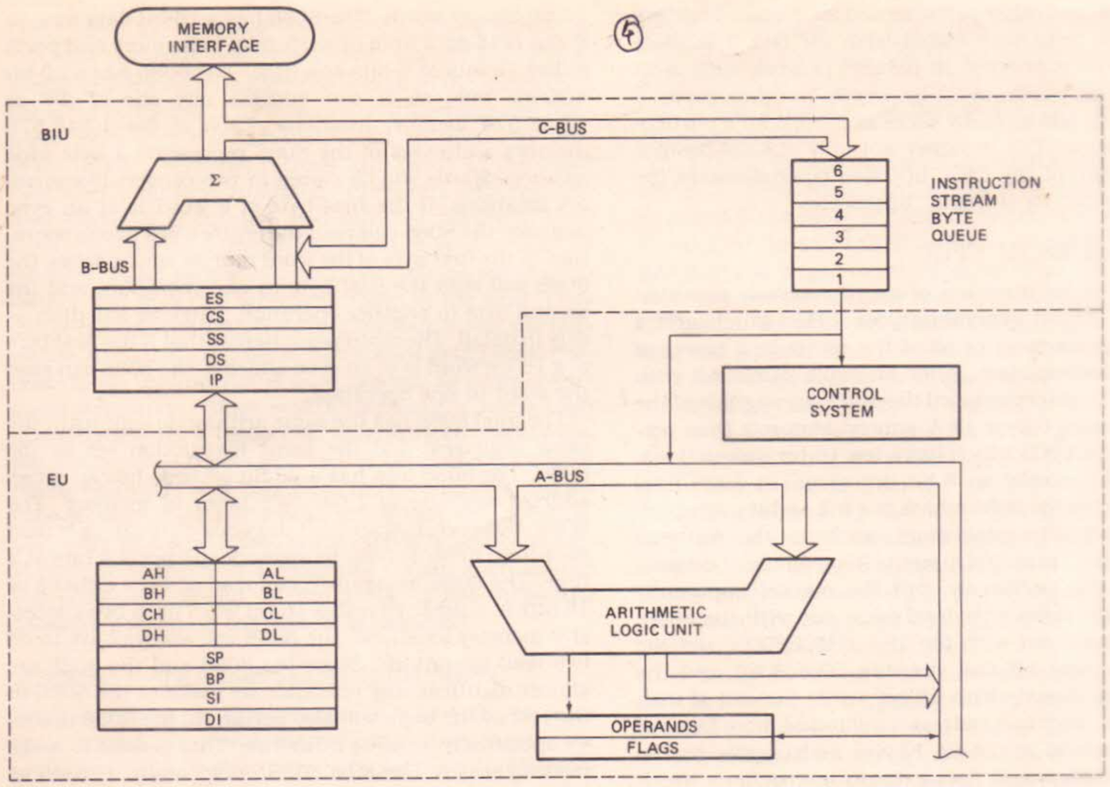


FIGURE 2-7 8086 internal block diagram. *(Intel Corp.)*

control signals, reads in instructions and data to internal registers, and sends out data to ports or memory. Before we can talk about how to write programs for the 8086, we need to discuss its specific internal features such as registers, instruction byte queue, and flags.

As shown by the block diagram in Figure 2-7, the 8086 CPU is divided into two independent functional parts, the *bus interface unit* or BIU, and the *execution unit* or EU. Dividing the work between these two units speeds up processing.

The Bus Interface Unit

The BIU sends out addresses, fetches instructions from memory, reads data from ports and memory, and writes data to ports and memory. In other words the BIU handles all transfers of data and addresses on the buses for the execution unit. The following sections describe the functional parts of the BIU.

THE QUEUE

To speed up program execution, the BIU fetches as many as six instruction bytes ahead of time from memory. The prefetched instruction bytes are held for the EU in a first-in-first-out group of registers called a *queue*. The BIU can be fetching instruction bytes while the EU is decoding an instruction or executing an instruction which does not require use of the buses. When the EU is ready for its next instruction, it simply reads the instruction from the queue in the BIU. This is much faster than sending out an address to the system memory and waiting for memory to send back the next instruction byte or bytes. The process is analogous to the way a bricklayer's assistant fetches bricks ahead of time and keeps a queue of bricks lined up so that the bricklayer can just reach out and grab a brick when necessary. Except in the cases of JUMP and CALL instructions where the queue must be dumped and then reloaded starting from a new address, this prefetch-and-queue scheme greatly speeds up processing. Fetching the next instruction while the current instruction executes is called *pipelining*.

SEGMENT REGISTERS

The BIU contains four 16-bit *segment registers*. They are: the *code segment* (CS) register, the *stack segment* (SS) register, the *extra segment* (ES) register, and the *data segment* (DS) register. These segment registers are used to hold the upper 16 bits of the starting addresses of four memory segments that the 8086 is working with at a particular time. The 8086 BIU sends out 20-bit addresses, so it can address any of 2^{20} or 1,048,576 bytes in memory. However, at any given time the 8086 only works with four, 65,536-byte (64 Kbyte) segments within this 1,048,576-byte (1 Mbyte) range. Figure 2-8

5

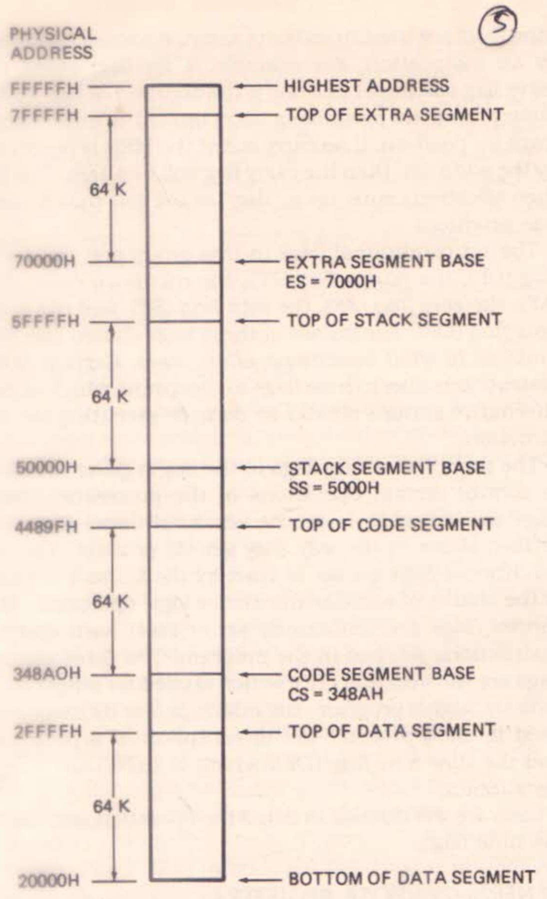


FIGURE 2-8 One way that four 64 Kbyte segments might be positioned within 1 Mbyte address space of 8086.

shows how these four segments might be positioned in memory at a given time. The four segments can be separated as shown, or, for small programs which do not need all 64 Kbytes in each segment, they can overlap. A minimum system, for example, might start all four segments at address 00000H.

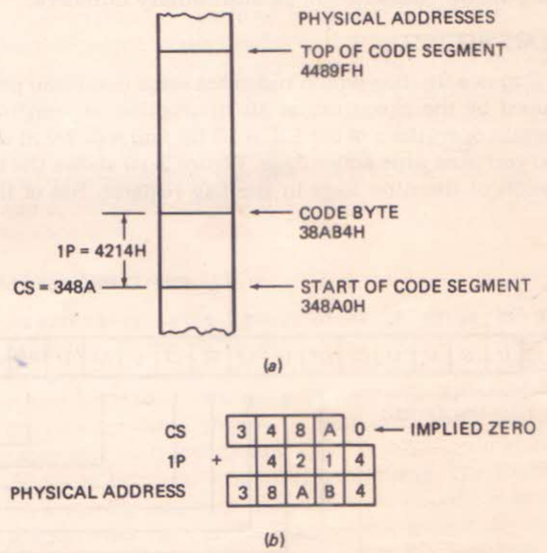
To repeat then, a segment register is used to hold the upper 16 bits of the starting address for each of the segments. The code segment register, for example, holds the upper 16 bits of the starting address for the segment from which the BIU is currently fetching instruction code bytes. The BIU always inserts zeros for the lowest four bits (nibble) of the 20-bit starting address for a segment. If the code segment register contains 348AH, for example, then the code segment will start at address 348A0H. In other words, a 64 Kbyte segment can be located anywhere within the 1 Mbyte address space, but the segment will always start at an address with zeros in the lowest 4 bits. This constraint was put on the location of segments so that it is only necessary to store and manipulate 16-bit numbers when working with the starting address of a segment. The part of a segment starting address stored in a segment register is often called the *segment base.*

A *stack* is a section of memory set aside to store addresses and data while a *subprogram* executes. The stack segment register is used for the upper 16 bits of the starting address for the program stack. We will discuss the use and operation of a stack in detail later.

The extra segment register and the data segment register are used to hold the upper 16 bits of the starting addresses of two memory segments that are used for data.

INSTRUCTION POINTER

The next feature to look at in the BIU is the *instruction pointer* (IP) register. As discussed previously, the code segment register holds the upper 16 bits of the starting address of the segment from which the BIU is fetching instruction code bytes. The instruction pointer register holds the 16-bit address of the next code byte *within* this code segment. The value contained in the IP is often referred to as an *offset*, because this value must be offset from (added to) the segment base address in CS to produce the required 20-bit physical address. Figure 2-9*a* shows in diagram form how this works. The CS register points to the *base* or start of the current code segment. The IP contains the distance or offset from this base address to the next instruction byte to be fetched. Figure 2-9*b* shows how the 16-bit offset in IP is added to the 16-bit segment base address in CS to produce the 20-bit *physical* address. Notice that the two 16-bit numbers are not added directly in line. One way to describe this process is to say that the contents of the CS register are shifted left four bit positions before the contents of the IP are added to it. CS contains 348AH. When shifted left by four bit positions this produces 348A0H as the starting address of the code segment. The offset of 4214H in the IP is added to this base to give a 20-bit physical address of 38AB4H.



PHYSICAL ADDRESSES
TOP OF CODE SEGMENT 4489FH
CODE BYTE 38AB4H
IP = 4214H
CS = 348A
START OF CODE SEGMENT 348A0H
(a)
CS 3 4 8 A 0 ← IMPLIED ZERO
IP + 4 2 1 4
PHYSICAL ADDRESS 3 8 A B 4
(b)

FIGURE 2-9 Addition of IP to CS to produce physical address of code byte. *(a)* Diagram. *(b)* Computation.

6

The 8086 20-bit physical addresses are often represented in a *segment base:offset form* rather than in the single number form. For the address of a code byte the alternative form will be CS:IP. For example, the address constructed in the preceding paragraph, 38AB4H, can also be represented as 348A:4214.

To summarize, then, the CS register contains the upper 16 bits of the starting address of the code segment in the 1 Mbyte address range of the 8086. The instruction pointer register contains a 16-bit offset which tells where in that 64 Kbyte code segment the next instruction byte will be fetched from. The actual physical address sent to memory is produced by shifting the contents of the CS register four bit positions left and adding the offset contained in IP.

As you will see in later sections, any time the 8086 accesses memory, the BIU produces the required 20-bit physical address by shifting the contents of one of the segment registers left four bit positions and adding to it a displacement or offset.

The Execution Unit

The execution unit of the 8086 tells the BIU where to fetch instructions or data from, decodes instructions, and executes instructions. The following sections describe the functional parts of the execution unit.

CONTROL CIRCUITRY, INSTRUCTION DECODER, AND ALU

Now take another look at the 8086 block diagram in Figure 2-7 to see what is contained in the execution unit. The EU contains *control circuitry* which directs internal operations. A *decoder* in the EU translates instructions fetched from memory into a series of actions which the EU carries out. The EU has a 16-bit *arithmetic logic unit* which can add, subtract, AND, OR, XOR, increment, decrement, complement, or shift binary numbers.

FLAG REGISTER

A *flag* is a flip-flop which indicates some condition produced by the execution of an instruction, or controls certain operations of the EU. A 16-bit *flag register* in the EU contains nine active flags. Figure 2-10 shows the location of the nine flags in the flag register. Six of the nine flags are used to indicate some *condition* produced by an instruction. For example, a flip-flop called the carry flag will be set to a one if the addition of two 16-bit binary numbers produces a carry out of the most significant bit position. If no carry out of the MSB is produced by the addition, then the carry flag will be a zero. The EU then effectively runs up a "flag" to tell you that a carry was produced.

The six conditional flags in this group are: the *carry flag* (CF), the *parity flag* (PF), the *auxiliary carry flag* (AF), the *zero flag* (ZF), the *sign flag* (SF), and the *overflow flag* (OF). The names of these flags should give you hints as to what conditions affect them. Certain 8086 instructions check these flags to determine which of two alternative actions should be done in executing the instruction.

The three remaining flags in the flag register are used to *control* certain operations of the processor. These flags are different from the six conditional flags described above in the way they get set or reset. The six conditional flags are set or reset by the EU on the basis of the results of some arithmetic or logic operation. The *control flags* are deliberately set or reset with specific instructions you put in the program. The three control flags are the *trap flag* (TF), which is used for single stepping through a program; the *interrupt flag* (IF), which is used to allow/prohibit the interruption of a program, and the *direction flag* (DF), which is used with string instructions.

Later we will discuss in detail the operation and use of the nine flags.

GENERAL-PURPOSE REGISTERS

Observe in Figure 2-7 that the EU has eight *general purpose registers* labeled AH, AL, BH, BL, CH, CL, DH, and DL. These registers can be used individually for temporary storage of 8-bit data. The AL register is also called the *accumulator*. It has some features that the other general-purpose registers do not have.

Certain pairs of these general-purpose registers can be used together to store 16-bit data words. The acceptable register pairs are AH and AL, BH and BL, CH and CL, and DH and DL. The AH-AL pair is referred to as the *AX register*, the BH-BL pair is referred to as the *BX register*, the CH-CL pair is referred to as the *CX register*,

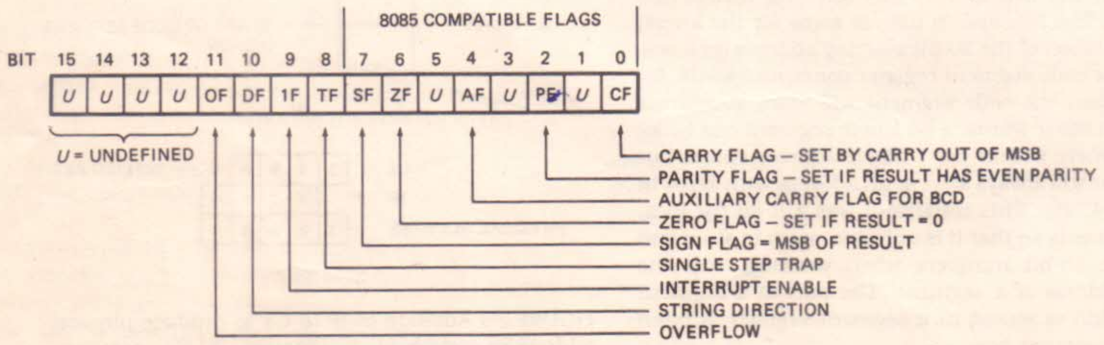

8085 COMPATIBLE FLAGS
BIT 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
U U U U OF DF IF TF SF ZF U AF U PF U CF
U = UNDEFINED
CARRY FLAG – SET BY CARRY OUT OF MSB
PARITY FLAG – SET IF RESULT HAS EVEN PARITY
AUXILIARY CARRY FLAG FOR BCD
ZERO FLAG – SET IF RESULT = 0
SIGN FLAG = MSB OF RESULT
SINGLE STEP TRAP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

FIGURE 2-10 8086 flag register format. *(Intel Corp.)*

7

and the DH-DL pair is referred to as the *DX register*. For 16-bit operations, AX is called the accumulator.

The 8086 register set is very similar to those of the earlier generation 8080 and 8085 microprocessors. It was designed this way so that the many programs written for the 8080 and 8085 could easily be translated to run on the 8086 or the 8088. The advantage of using internal registers for the temporary storage of data is that, since the data is already in the EU, it can be accessed much more quickly than it could be accessed in external memory.

STACK POINTER REGISTER

A stack, remember, is a section of memory set aside to store addresses and data while a subprogram is executing. The 8086 allows you to set aside an entire 64 Kbyte segment as a stack. The upper 16 bits of the starting address for this segment is kept in the stack segment register. The *stack pointer* (SP) register contains the 16-bit offset from the start of the segment to the memory location where a word was most recently stored on the stack. The memory location where a word was most recently stored is called the *top of stack*. Figure 2-11*a* shows this in diagram form.

The physical address for a stack read or for a stack write is produced by adding the contents of the stack pointer register to the segment base address in SS. To do this the contents of the stack segment register are shifted four bit positions left and the contents of SP are added to the shifted result. Figure 2-11*b* shows an example. The 5000H in SS is shifted left four bit positions to give 50000H. When FFE0H in the SP is added to this, the resultant physical address for the top of the stack will be 5FFE0H. The physical address can be represented either as a single number, 5FFE0H, or it can be represented in SS:SP form as 5000:FFE0H.

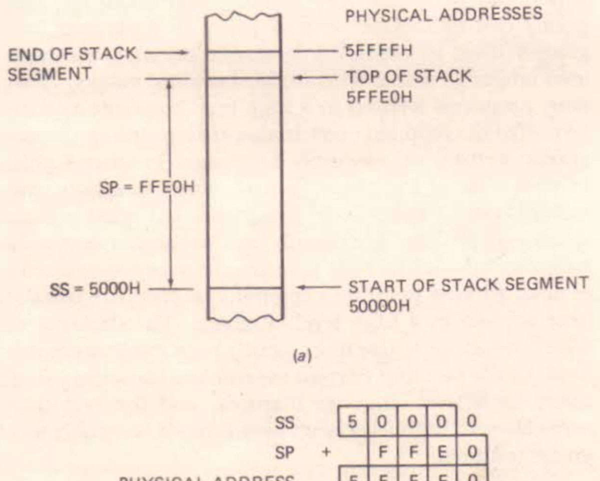


FIGURE 2-11 Addition of SS and SP to produce physical address of top of stack. *(a)* Diagram. *(b)* Computation.

The operation and use of the stack will be discussed in detail later as need arises.

OTHER POINTER AND INDEX REGISTERS

In addition to the stack pointer register, SP, the EU contains a 16-bit *base pointer* (BP) register. It also contains a 16-bit *source index* (SI) register and a 16-bit *destination index* (DI) register. These three registers can be used for temporary storage of data just as the general-purpose registers described above. However, their main use is to hold the 16-bit offset of a data word in one of the segments. SI, for example, can be used to hold the offset of a data word in the data segment. The physical address of the data in memory will be generated in this case by shifting the contents of the data segment register four bit positions to the left and adding the contents of SI to the result. A later section on addressing modes will discuss and show many examples of the use of these base and index registers.

INTRODUCTION TO PROGRAMMING THE 8086

Programming Languages

Now that you have an overview of the 8086 CPU, it is time to start you thinking about how it is programmed. To run a program, a microcomputer must have the program stored in binary form in successive memory locations. There are three language levels that can be used to write a program for a microcomputer.

MACHINE LANGUAGE

You can write programs as simply a sequence of the binary codes for the instructions you want the microcomputer to execute. The three-instruction program in Figure 2-2*b* is an example. This binary form of the program is referred to as *machine language* because it is the form required by the machine. However, it is difficult, if not impossible, for a programmer to memorize the thousands of binary instruction codes for a CPU such as the 8086. Also, it is very easy for an error to occur when working with long series of 1's and 0's. Using hexadecimal representation for the binary codes might help some, but there are still thousands of instruction codes to cope with.

ASSEMBLY LANGUAGE

To make programming easier many programmers write programs in *assembly language*. They then translate the assembly language program to machine language so it can be loaded into memory and run. Assembly language uses two-, three-, or four-letter *mnemonics* to represent each instruction type. A mnemonic is just a device to help you remember something. The letters in an assembly language mnemonic are usually initials or a shortened form of the English word(s) for the operation performed by the instruction. For example, the mnemonic for subtract is SUB, the mnemonic for exclusive OR is XOR, and the mnemonic for the instruction to copy data from one location to another is MOV.

8086 Internal block diagram shows that 8086 CPU is divided into two independent functional parts, the bus interface unit or BIU, and the execution unit or EU. This type of architecture is known as pipelined architecture. Dividing the work between these two units speeds up processing.

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

The EU of the 8086 tells the BIU where to fetch instructions or data from, decodes instructions, and executes instructions.

Pipelining: To speed up program execution in 8086 up, the BIU fetches as many as six instruction bytes ahead of time from memory. The prefetched instruction bytes are held for the EU in a first-in-first-out (FIFO) group of registers called a Queue. The BIU can be fetching instruction bytes while the EU is decoding an instruction or executing an instruction which does not require use of the buses. When the EU is ready for its next instruction, it simply reads the instruction from the Queue in the BIU. This is much faster than sending out an address to the system memory and waiting for memory to send back the next instruction byte or bytes. Except in the cases of JUMP and CALL instructions where the queue must be dumped and then reloaded starting from a new address, this prefetch-and-queue scheme greatly speeds up processing.

Fetching the next instruction while the current instruction executes is called pipelining.

Memory segmentation :- The architecture of 8086 up supports memory segmentation. The BIU contains four 16-bit segment registers. These segment registers are used to hold the upper 16 bits of the starting addresses of four memory segments that the 8086 is working with at a particular time. These segments are code segment, stack segment, Data segment and extra segment memory.

Though the address of memory is 20-bit wide but in 8086 µp, there is no 20bit registers to hold the 20-bit address like 8085 µp. [SP and PC].

The CS register, for example, holds the upper 16 bits of the starting address for the segment from which the BIU is currently fetching instruction code bytes. The BIU always inserts zeros for the lowest four bits (nibble) of the 20-bit starting address for a segment. If the code segment register contains 348A H, for example, then the code segment memory will start at address 348A0 H. In other words, a 64 KB segment can be located anywhere within the 1MB address space, but the segment will always start at an address with zeros in the lowest 4-bits.

The part of a segment ~~starting~~ starting address stored in a segment register is often called the segment base.

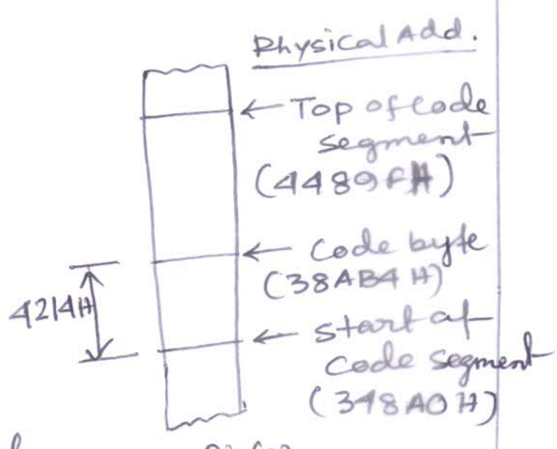
The value contained in the IP is referred to as an offset, because this value must be offset from (added to) the segment base address in CS to produce the required 20-bit physical address.

* Generation of 20-bit physical address (PA) in 8086 µp.

Let,
CS = 348A H
IP = 4214 H

One way to describe this process is to say that the contents of the CS register are shifted left four bit positions before the contents of the IP are added to it.

CS contains 348AH. When shifted left by four bit positions this produces 348A0 H as the starting address of the code segment. The offset of 4214H in the IP is then added to this base to give a 20-bit physical address of 38AB4H.


Physical Add.
← Top of code segment (4489F H)
← Code byte (38AB4 H)
4214H
← start of code segment (348A0 H)

Fig(a)

CS	3	4	8	A	0	← Implied zero
IP +		4	2	1	4	
PA→	3	8	A	B	4	

Fig.(b)

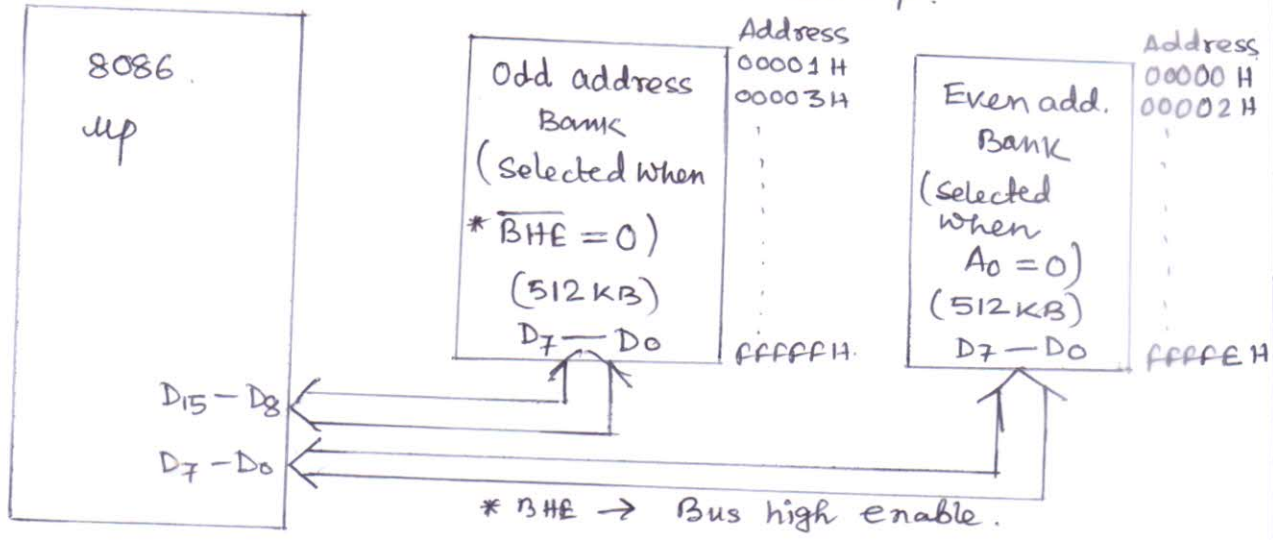
Fig(a) → Diagram , Fig(b) computation.

SARASWATI SAHA
ASST. PROF, ECE, RCCIIT

Segment registers and default offset registers in the 8086 µp

Segment registers	Default offset registers
CS	IP
DS	BX, SI, DI, 8-bit or 16-bit displacement
SS	SP, BP
ES	DI for string instructions

PHYSICAL MEMORY ORGANIZATION IN 8086 µp.



The 8086 has a 20-bit address bus, so it can address 2^{20} or 1,048,576 addresses. Each address represents a stored byte. when we write a word to memory, the word is actually written into two consecutive memory addresses. [for example the low byte of the word is written into the memory add. 0437A H, and the high byte of the word is written into the next higher add., 0437B H] To make it possible to read or write a word with one machine cycle, the memory for an 8086 is set up as two "banks" of up to 512 KB each. fig. shows this in diagram form.

One memory bank contains all the bytes which have even addresses such as 00000 H, 00002 H, and 00004 H. The data lines of this bank are connected to the lower eight data lines, $D_0 - D_7$, of the 8086. The other memory bank contains all of the bytes which have odd addresses such as 00001 H, 00003 H, and 00005 H. The data lines of this bank are connected to the upper eight data lines, $D_8 - D_{15}$, of the 8086. The 8086 provides two enable signals, $\overline{BHE}$ and A_0, for the selection of odd banks and even banks, respectively. Address lines $A_1 - A_{19}$ are used to select the desired memory location to access the byte. * [when $A_0 = 0$, it indicates even address
when $\overline{BHE} = 0$, it indicates odd address.]