32 parallel signal
the address of the
or read from. The
PU can address is
lines. If the CPU
ddress 2^N memory
address lines can
s, a CPU with 20
76 locations, and
2^{24} or 16,777,216
om or writes data
n the address bus.

rallel signal lines.
vs on the data bus
idirectional. This
from memory or
send data out to
Many devices in a
ed to the data bus,
s outputs enabled.
must have *three-*
lisabled when it is

rallel signal lines.
trol bus to enable
es or port devices.
ry Read, Memory
ead a byte of data
the CPU sends out
te on the address
ead signal on the
gnal enables the
data word onto the
travels along the

irmware

ou hear the terms
lmost constantly.
ysical devices and
rs to the programs
the term given to
er devices which
tion.

Summary of Important Points So Far

- A computer or microcomputer consists of memory, a CPU, and some input/output circuitry.
- These three parts are connected by the address bus, the data bus, and the control bus.
- The sequence of instructions or program for a computer is stored as binary numbers in successive memory locations.
- The CPU fetches an instruction from memory, decodes the instruction to determine what actions must be done for the instruction, and carries out these actions.

EXECUTION OF A THREE-INSTRUCTION PROGRAM

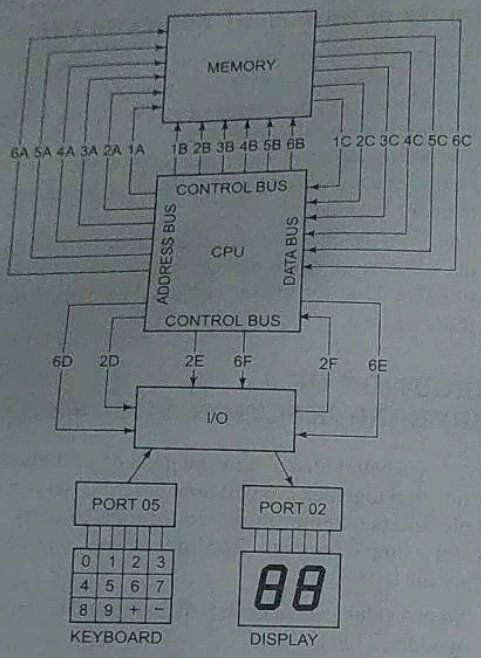
To give you a better idea of how the parts of a microcomputer function together, we will now describe the actions a simple microcomputer might go through to carry out (execute) a simple program. The three instructions of the program are

1. Input a value from a keyboard connected to the port at address 05H.
2. Add 7 to the value.
3. Output the result to a display connected to the port at address 02H.

Figure 2.6 shows in diagram form and sequential list form the actions that the computer will perform to execute these three instructions.

For this example, assume that the CPU fetches instructions and data from memory 1 byte at a time, as is done in the original IBM PC and its clones. Also assume that the binary codes for the instructions are in sequential memory locations starting at address 00100H. Fig. 2.6*b* shows the actual binary codes that would be required in successive memory locations to execute this program on an IBM PC-type microcomputer.

The CPU needs an instruction before it can do anything, so its first action is to fetch an instruction byte from memory. To do this, the CPU sends out the address of the first instruction byte, in this case 00100H, to memory on the address bus. This action is represented by line 1A in Fig. 2.6*a*. The CPU then sends out a Memory Read signal on the control bus (line 1B in the figure). The Memory Read signal enables the memory to output the addressed byte on the data bus. This action is represented by line 1C in the figure. The CPU reads in this first instruction byte (E4H) from the data bus and *decodes* it. By *decode* we mean that the CPU determines from the



PROGRAM

1. INPUT A VALUE FROM PORT 05.
2. ADD 7 TO THIS VALUE.
3. OUTPUT THE RESULT TO PORT 02.

SEQUENCE

1A CPU SENDS OUT ADDRESS OF FIRST INSTRUCTION TO MEMORY.
1B CPU SENDS OUT MEMORY READ CONTROL SIGNAL TO ENABLE MEMORY.
1C INSTRUCTION BYTE SENT FROM MEMORY TO CPU ON DATA BUS.
2A ADDRESS NEXT MEMORY LOCATION TO GET REST OF INSTRUCTION.
2B SEND MEMORY READ CONTROL SIGNAL TO ENABLE MEMORY.
2C PORT ADDRESS BYTE SENT FROM MEMORY TO CPU ON DATA BUS.
2D CPU SENDS OUT PORT ADDRESS ON ADDRESS BUS.
2E CPU SENDS OUT INPUT READ CONTROL SIGNAL TO ENABLE PORT.
2F DATA FROM PORT SENT TO CPU ON DATA BUS.
3A CPU SENDS ADDRESS OF NEXT INSTRUCTION TO MEMORY.
3B CPU SENDS MEMORY READ CONTROL SIGNAL TO ENABLE MEMORY.
3C INSTRUCTION BYTE FROM MEMORY SENT TO CPU ON DATA BUS.
4A CPU SENDS NEXT ADDRESS TO MEMORY TO GET REST OF INSTRUCTION.
4B CPU SENDS MEMORY READ CONTROL SIGNAL TO ENABLE MEMORY.
4C NUMBER 07H SENT FROM MEMORY TO CPU ON DATA BUS.
5A CPU SENDS ADDRESS OF NEXT INSTRUCTION TO MEMORY.
5B CPU SENDS MEMORY READ CONTROL SIGNAL TO ENABLE MEMORY.
5C INSTRUCTION BYTE FROM MEMORY SENT TO CPU ON DATA BUS.
6A CPU SENDS OUT NEXT ADDRESS TO GET REST OF INSTRUCTION.
6B CPU SENDS OUT MEMORY READ CONTROL SIGNAL TO ENABLE MEMORY.
6C PORT ADDRESS BYTE SENT FROM MEMORY TO CPU ON DATA BUS.
6D CPU SENDS OUT PORT ADDRESS ON ADDRESS BUS.
6E CPU SENDS OUT DATA TO PORT ON DATA BUS.
6F CPU SENDS OUT OUTPUT WRITE SIGNAL TO ENABLE PORT.

(a)

MEMORY ADDRESS	CONTENTS (BINARY)	CONTENTS (HEX)	OPERATION
00100H	11100100	E4	INPUT FROM
00101H	00000101	05	PORT 05H
00102H	00000100	04	ADD
00103H	00000111	07	07H
00104H	11100110	E6	OUTPUT TO
00105H	00000010	02	PORT 02

(b)

Fig. 2.6 *(a) Execution of a three-step computer program, (b) Memory addresses and memory contents for a three-step program.*

binary code read in, what actions it is supposed to take. If the CPU is a microprocessor, it selects the sequence of microinstructions needed to carry out the instruction read from memory. For the example instruction here, the CPU determines that the code read in represents an Input instruction. From decoding this instruction byte, the CPU also determines that it needs more information before it can carry out the instruction. The additional information the CPU needs is the address of the port that the data is to be input from. This port address part of the instruction is stored in the next memory location after the code for the Input instruction.

To fetch this second byte of the instruction, the CPU sends out the next sequential address (00101H) to memory, as shown by line 2A in the figure. To enable the addressed memory device, the CPU also sends out another Memory Read signal on the control bus (line 2B). The memory then outputs the addressed byte on the data bus (line 2C). When the CPU has read in this second byte, 05H in this case, it has all the information it needs to execute the instruction.

To execute the Input instruction, the CPU sends out the port address (05H) on the address bus (line 2D) and sends out an I/O Read signal on the control bus (line 2E). The I/O Read signal enables the addressed port device to put a byte of data on the data bus (line 2F). The CPU reads in the byte of data and stores it in an internal register. This completes the fetching and execution of the first instruction.

Having completed the first instruction, the CPU must now fetch its next instruction from memory. To do this, it sends out the next sequential address (00102H) on the address bus (line 3A) and sends out a Memory Read signal

on the control bus (line 3B). The Memory Read signal enables the memory device to put the addressed byte (04H) on the data bus (line 3C). The CPU reads in this instruction byte from the data bus and decodes it. From this instruction byte the CPU determines that it is supposed to add some number to the number stored in the internal register. The CPU also determines from decoding this instruction byte that it must go to memory again to get the next byte of the instruction, which contains the number that it is supposed to add. To get the required byte, the CPU will send out the next sequential address (00103H) on the address bus (line 4A) and another Memory Read signal on the control bus (line 4B). The memory will then output the contents of the addressed byte (the number 07H) on the data bus (line 4C). When the CPU receives this number, it will add it to the contents of the internal register. The result of the addition will be left in the internal register. This completes the fetching and executing of the second instruction.

The CPU must now fetch the third instruction. To do this, it sends out the next sequential address (00104H) on the address bus (line 5A) and sends out a Memory Read signal on the control bus (line 5B). The memory then outputs the addressed byte (E6H) on the data bus (line 5C). From decoding this byte, the CPU determines that it is now supposed to do an Output operation to a port. The CPU also determines from decoding this byte that it must go to memory again to get the address of the output port. To do this, it sends out the next sequential address (00105H) on the address bus (line 6A), sends out a Memory Read signal on the control bus (line 6B), and reads in the byte (02H) put on the data bus by the memory (line 6C). The CPU now has all the information that it needs to execute the Output instruction.

To output a data byte to a port, the CPU first sends out the address of the desired port on the address bus (line 6D). Next it outputs the data byte from the internal register on the data bus (line 6E). The CPU then sends out an I/O Write signal on the control bus (line 6F). This signal enables the addressed output port device so that the data from the data bus lines can pass through it to the LED displays. When the CPU removes the I/O Write signal to proceed with the next instruction, the data will remain latched on the output pins of the port device. The data will remain latched on the port until the power is turned off or until a new data word is output to the port. This is important because it means that the computer does not have to keep outputting a value over and over in order for it to remain on the output.

All the steps described above may seem like a great deal of work just to input a value from a keyboard, add 7 to it, and output the result to a display. Even a simple microcomputer, however, can run through all these steps in a few microseconds.

Summary of Simple Microcomputer Bus Operation

1. A microcomputer fetches each program instruction in sequence, decodes the instruction, and executes it.
2. The CPU in a microcomputer fetches instructions or reads data from memory by sending out an address on the address bus and a Memory Read signal on the control bus. The memory outputs the addressed instruction or data word to the CPU on the data bus.
3. The CPU writes a data word to memory by sending out an address on the address bus, sending out the data word on the data bus, and sending a Memory Write signal to memory on the control bus.
4. To read data from a port, the CPU sends out the port address on the address bus and sends an I/O Read signal to the port device on the control bus. Data from the port comes into the CPU on the data bus.
5. To write data to a port, the CPU sends out the port address on the address bus, sends out the data to be written to the port on the data bus, and sends an I/O Write signal to the port device on the control bus.

MICROPROCESSOR EVOLUTION AND TYPES

As we told you in the preceding section, a microprocessor is used as the CPU in a microcomputer. There are now many different microprocessors available, so before we dig into the details of a specific device, we will give you a short microprocessor history lesson and an overview of the different types.

Microprocessor Evolution

A common way of categorizing microprocessors is by the number of bits that their ALU can work with at a time. In other words, a microprocessor with a 4-bit ALU will be referred to as a 4-bit microprocessor, regardless of the number of address lines or the number of data bus lines that it has. The first commercially available microprocessor was the Intel 4004, produced in 1971. It contained 2300 PMOS transistors. The 4004 was a 4-bit

device intended to be used with some other devices in making a calculator. Some logic designers, however, saw that this device could be used to replace PC boards full of combinational and sequential logic devices. Also, the ability to change the function of a system by just changing the programming, rather than redesigning the hardware, is very appealing. It was these factors that pushed the evolution of microprocessors.

In 1972 Intel came out with the 8008, which was capable of working with 8-bit words. The 8008, however, required 20 or more, additional devices to form a functional CPU. In 1974, Intel announced the 8080, which had a much larger instruction set than the 8008 and required only two additional devices to form a functional CPU. Also, the 8080 used NMOS transistors, so it operated much faster than the 8008. The 8080 is referred to as a *second-generation microprocessor.*

Soon after Intel produced the 8080, Motorola came out with the MC6800, another 8-bit general-purpose CPU. The 6800 had the advantage that it required only a + 5 – V supply rather than the –5-V, +5 – V, and +12-V supplies required by the 8080. For several years, the 8080 and the 6800 were the top-selling 8-bit microprocessors. Some of their competitors were the MOS Technology 6502, used as the CPU in the Apple II microcomputer, and the Zilog Z80, used as the CPU in the Radio Shack TRS-80 microcomputer.

As designers found more and more applications for microprocessors, they pressured microprocessor manufacturers to develop devices with architectures and features optimized for doing certain types of tasks. In response to the expressed needs, microprocessors have evolved in three major directions during the last 15 years.

Dedicated or Embedded Controllers

One direction in the *dedicated or embedded controllers.* These devices are used to control "smart" machines, such as microwave ovens, clothes washers, sewing machines, auto ignition systems, and metal lathes. Texas Instruments has produced millions of their TMS-1000 family of 4-bit microprocessors for this type of application. In 1976 Intel introduced the 8048, which contains an 8-bit CPU, RAM, ROM, and some I/O ports all in one 40-pin package. Other manufacturers have followed with similar products. These devices are often referred to as *microcontrollers.* Some currently available devices in this category—the Intel 8051 and the Motorola MC6801, for example—contain programmable counters and a serial port (UART) as well as a CPU, ROM, RAM, and parallel I/O ports. A more recently introduced single-chip microcontroller, the Intel 8096, contains a 16-bit CPU, ROM, RAM, a UART, ports, timers, and a 10-bit analog-to-digital converter.

Bit-Slice Processors

A second direction of microprocessor evolution has been *bit-slice processors.* For some applications, general-purpose CPUs such as the 8080 and 6800 are not fast enough or do not have suitable instruction sets. For these applications, several manufacturers produce *devices* which can be used to build a custom CPU. An example is the Advanced Micro Devices 2900 family of devices. This family includes 4-bit ALUs, multiplexers, sequencers, and other parts needed for custom-building a CPU. The term *slice* comes from the fact that these parts can be connected in parallel to work with 8-bit words, 16-bit words, or 32-bit words. In other words, a designer can add as many slices as needed for a particular application. The designer not only custom-designs the hardware of the CPU, but also custom-makes the instruction set for it, using "microcode."

General-Purpose CPUs

The third major direction of microprocessor evolution has been towards general-purpose CPUs which give a microcomputer most or all of the computing power of earlier minicomputers. After Motorola came out with the MC6800, Intel produced the 8085, an upgrade of the 8080 that required only a +5-V supply. Motorola then produced the MC6809, which has a few 16-bit instructions, but is still basically an 8-bit processor. In 1978 Intel came out with the 8086, which is a full 16-bit processor. Some 16-bit microprocessors, such as the National PACE and the Texas Instruments 9900 family of devices, had been available previously, but the market apparently wasn't ready. Soon after Intel came out with the 8086, Motorola came out with the 16-bit MC68000, and the 16-bit race was off and running. The 8086 and the 68000 work directly with 16-bit words instead of with 8-bit words, they can address a million or more bytes of memory instead of the 64 Kbytes addressable by the 8-bit processors, and they execute instructions much faster than the 8-bit processors. Also, these 16-bit processors have single instructions for functions such as *multiply* and *divide*, which required a lengthy sequence of instructions on the 8-bit processors.

The evolution along this last path has continued on to 32-bit processors that work with gigabytes (10^9 bytes) or terabytes (10^{12} bytes) of memory. Examples of these devices are the Intel 80386, the Motorola MC68020, and the National 32032.

Since we could not possibly describe in this book the operation and programming of even a few of the available processors, we confine our discussions primarily to one

group of related microprocessors. The family we have chosen is the Intel 8086, 8088, 80186, 80188, 80286, 80386, 80486 family. Members of this family are very widely used in personal computers, business computer systems, and industrial control systems. Our experience has shown that learning the programming and operation of one family of microcomputers very thoroughly is much more useful than looking at many processors superficially. If you learn one processor family well, you will most likely find it quite easy to learn another when you have to.

THE 8086 MICROPROCESSOR FAMILY—OVERVIEW

The Intel 8086 is a 16-bit microprocessor that is intended to be used as the CPU in a microcomputer. The term *16-bit* means that its arithmetic logic unit, its 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 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, memory locations.

Each of the 1,048,576 memory addresses of the 8086 represents a byte-wide location. Sixteen-bit words 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 with one bus operation and the second byte with another bus operation. Later we will discuss this in detail. The main point here is that if the first byte of a 16-bit word is at an even address, the 8086 can read the entire word in one operation.

The Intel 8088 has the same ALU, the same registers, and the same instruction set as the 8086. The 8088 also has a 20-bit address bus, so it can address any one of 1,048,576 bytes in memory. The 8088, however, has an 8-bit data bus, so it can only read data from or write data to memory and ports, 8 bits at a time. The 8086, remember, can read or write either 8 or 16 bits at a time. To read a 16-bit word from two successive memory locations, the 8088 will always have to do two read operations. Since the 8086 and the 8088 are almost identical, any reference we make to the 8086 in the rest of the book will also pertain to the 8088 unless we specifically indicate otherwise. This is done to make reading easier. The Intel 8088, incidentally, is used as the CPU in the original IBM Personal Computer, the IBM PC/XT, and several compatible personal computers.

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. In a later chapter we will discuss these integrated peripherals. The instruction set of the 80186 and 80188 is a *superset* of the instruction set of the 8086. The term *superset* means that all the 8086 and 8088 instructions will execute properly on an 80186 or an 80188, but the 80186 and the 80188 have a few additional instructions. In other words, a program written for an 8086 or an 8088 is *upward-compatible* to an 80186 or an 80188, but a program written for an 80186 or an 80188 may not execute correctly on an 8086 or an 8088. In the instruction set descriptions in Chapter 6, we specifically indicate which instructions work only with the 80186 or 80188.

The Intel 80286 is a 16 bit, advanced version of the 8086 which was specifically designed for use as the CPU in a multiuser or multitasking microcomputer. When operating in its *real address mode*, the 80286 functions mostly as a fast 8086. Most programs written for an 8086 can be run on an 80286 operating in its real address mode. When operating in its *virtual address mode*, an 80286 has features which make it easy to keep users' programs separate from one another and to protect the system program from destruction by users' programs. In Chapter 15, we discuss the operation and use of the 80286. The 80286 is the CPU used in the IBM PC/AT personal computer.

With the 80386 processor, Intel started the 32-bit processor architecture, known as the IA-32 architecture. This architecture extended all the address and general purpose registers to 32-bits, which gave the processor the capability to handle 32-bit address (4 GB of memory addressing), with 32-bit data, and yet accommodating all the software designed for the earlier 16-bit processors, 8086,8088, 80186, 80188 and 80286. It contains more sophisticated features for use in multiuser and multitasking environments.

Intel 80486 is the next member of the IA-32 architecture. This processor has the floating point processor (80387) integrated into the CPU chip itself. These processors are then followed by different versions of the Pentium Processors, with different additional capabilities such as multimedia (MMX, SSE, SSE2 etc.), system power saving modes, hyper thread technology etc.

In Chapter 15, we will discuss the 80286, 386, and 486 processors, and in Chapter 16, we shall take up some major versions of the Pentium processors for discussion.

8086 INTERNAL ARCHITECTURE

Before we can talk about how to write programs for the 8086, we need to discuss its specific internal features, such as its ALU, flags, registers, instruction byte queue, and segment registers.

As shown by the block diagram in Fig. 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 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 execution unit of the 8086 tells the BIU where to fetch instructions or data from, decodes instructions, and executes instructions. Let's take a look at some of the parts of the execution unit.

The Execution Unit

CONTROL CIRCUITRY, INSTRUCTION DECODER, AND ALU

As shown in Fig. 2.7, 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 that 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. Fig. 2.8 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 1 if the addition of two 16-bit binary

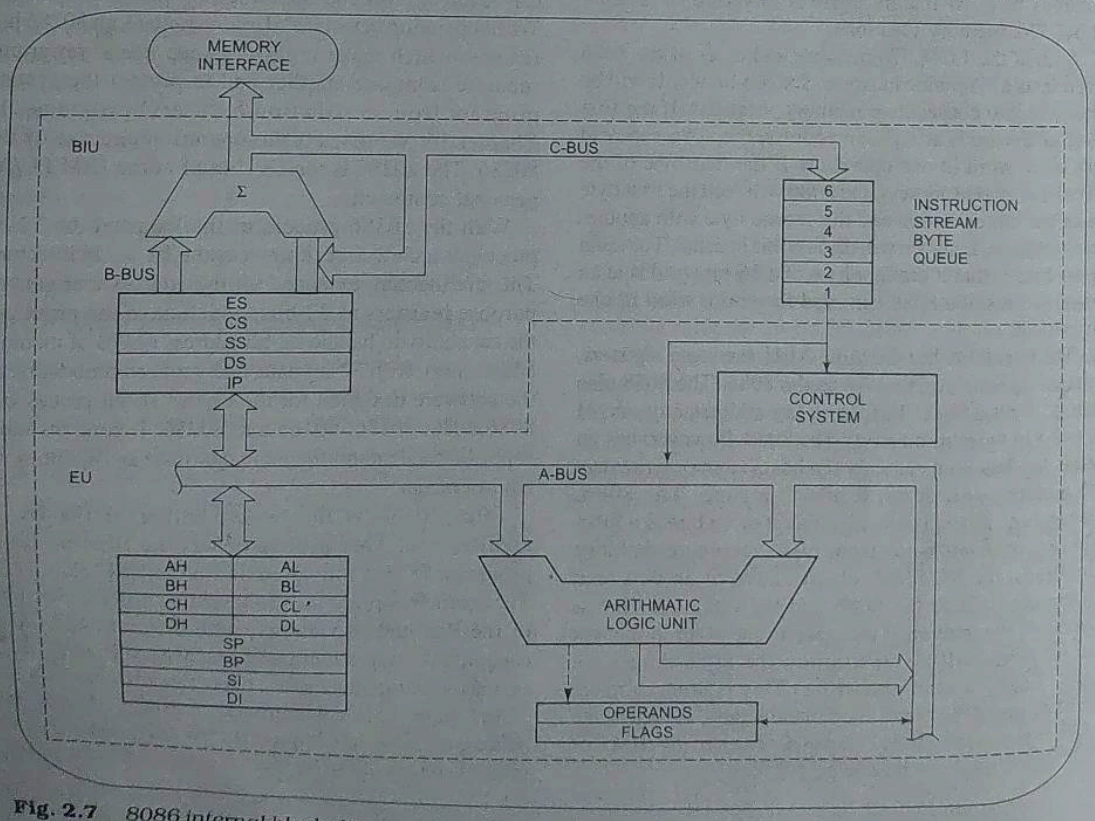


Fig. 2.7 *8086 internal block diagram. (Intel Corp.)*

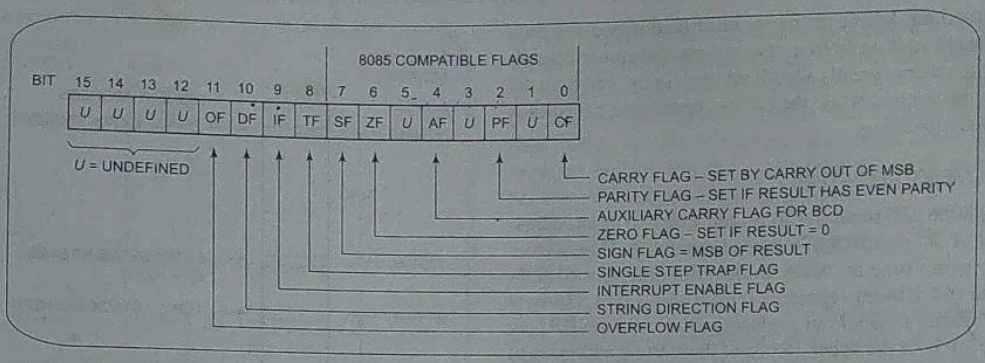


Fig. 2.8 *8086 flag register format. (Intel Corp.)*

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 0. The EU, thus 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 are 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 your 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 or 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 Fig. 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*, and the DH–DL pair is referred to as the *DX register*.

The 8086 general-purpose 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. Now let's look at the features of the BIU.

The BIU

THE QUEUE

While the EU is decoding an instruction or executing an instruction which does not require use of the buses, the BIU fetches up to six instruction bytes for the following instructions. The BIU stores these prefetched bytes in a first-in-first-out register set called a *queue*. When the EU is ready for its next instruction, it simply reads the instruction byte(s) for 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 JMP 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 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 works with only four 65,536-byte (64-Kbyte) segments within this 1,048,576-byte (1-Mbyte) range. Four *segment registers* in the BIU 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 four segment registers are the *code segment* (CS) register, the *stack segment* (SS) register, the *extra segment* (ES) register, and the *data segment* (DS) register.

Figure 2.9 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.

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 4 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 to hold 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

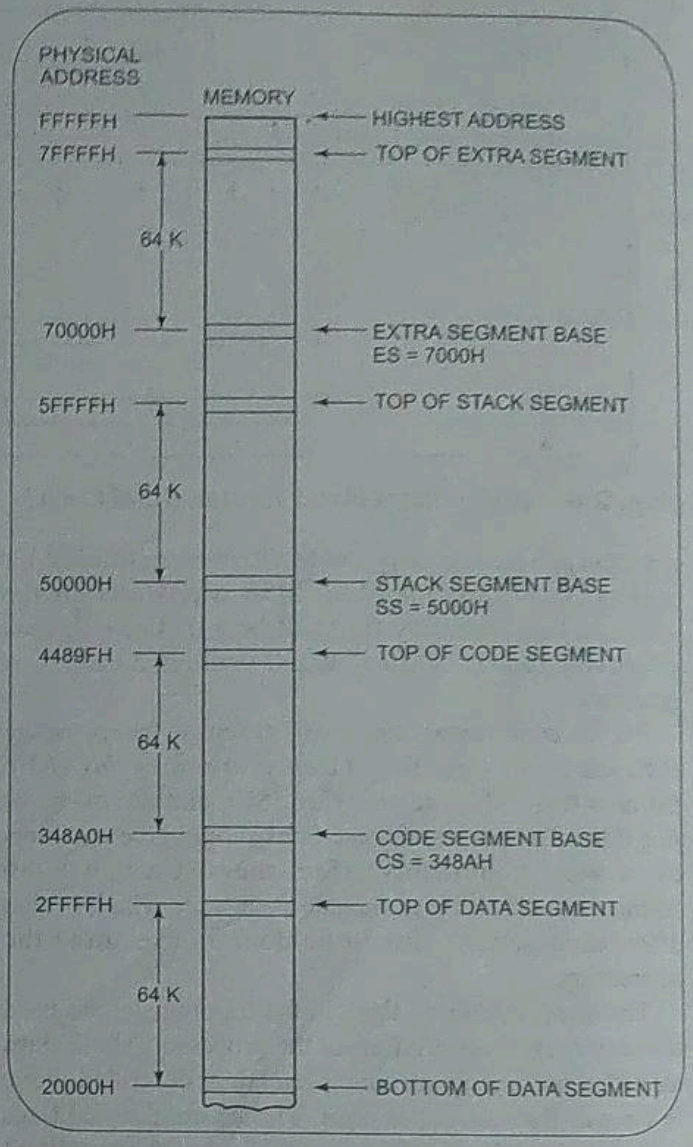


Fig. 2.9 *One way four 64-Kbyte segments might be positioned within the 1-Mbyte address space of an 8086.*

segment register holds the upper 16 bits of the starting address of the segment from which the BIU is currently fetching instruction code bytes. The instruction pointer register holds the 16-bit address, or *offset*, of the next code byte *within* this code segment. 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 sent out by the BIU. Fig. 2.10*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. Fig. 2.10*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

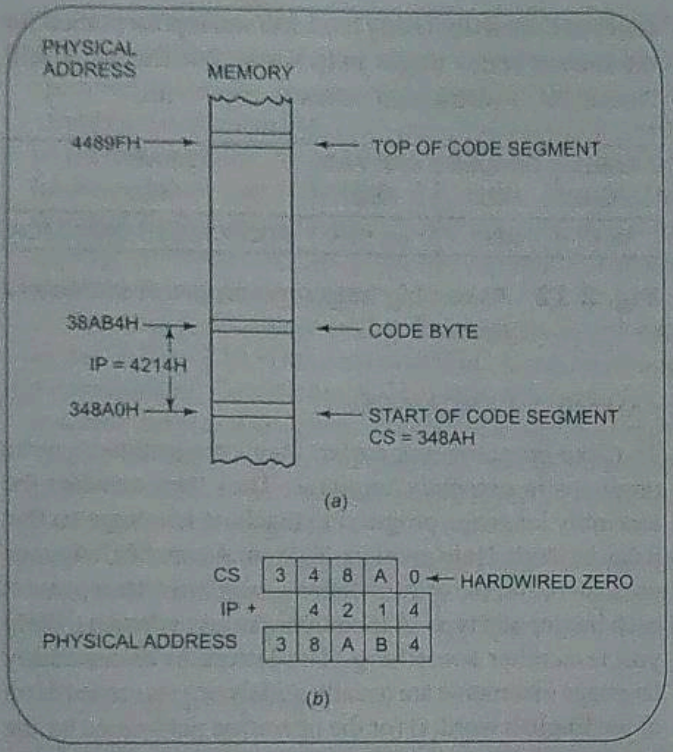


Fig. 2.10 Addition of IP to CS to produce the physical address of the code byte. (a) Diagram, (b) Computation.

16-bit numbers are not added directly in line, because the CS register contains only the upper 16 bits of the base address for the code segment. As we said before, the BIU automatically inserts zeros for the lowest 4 bits of the segment base address.

If the CS register, for example, contains 348AH, you know that the starting address for the code segment is 348A0H. When the BIU adds the offset of 4214H in the IP to this segment base address, the result is a 20-bit physical address of 38AB4H.

An alternative way of representing a 20-bit physical address is the *segment base:offset form*. For the address of a code byte, the format for this alternative form will be CS:IP. As an example of this, 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 is to be fetched from. The actual physical address sent to memory is produced by adding the offset contained in the IP register to the segment base represented by the upper 16 bits in the CS register.

Any time the 8086 accesses memory, the BIU produces the required 20-bit physical address by adding an offset to a segment base value represented by the contents of one of the segment registers. As another example of this, let's look at how the 8086 uses the contents of the stack segment register and the contents of the stack pointer register to produce a physical address.

STACK SEGMENT REGISTER AND 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 are kept in the stack segment register. The *stack pointer* (SP) register in the execution unit holds 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*. Fig. 2.11*a* shows this in diagram form.

The physical address for a stack read or a stack write is produced by adding the contents of the stack pointer register to the segment base address represented by the upper 16 bits of the base address in SS. Fig. 2.11*b* shows

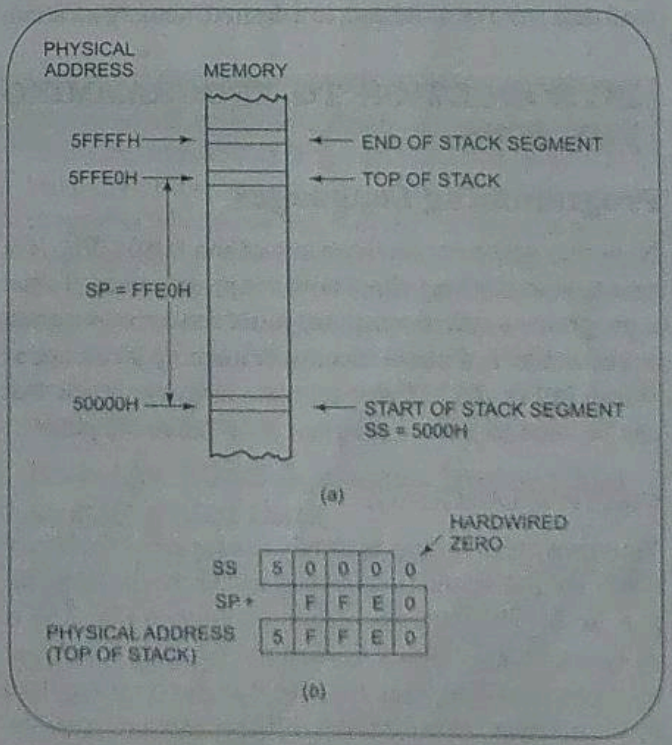


Fig. 2.11 *Addition of SS and SP to produce the physical address of the top of the stack. (a) Diagram, (b) Computation.*

an example. The 5000H in SS represents a segment base address of 50000H. When the 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 in SS:SP form as 5000:FFE0H.

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

POINTER AND INDEX REGISTERS IN THE EXECUTION UNIT

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 adding the contents of SI to the segment base address represented by the 16-bit number in the DS register. After we give you an overview of the different levels of languages used to program a microcomputer, we will show you some examples of how we tell the 8086 to read data from or write data to a desired memory location.

INTRODUCTION TO PROGRAMMING THE 8086

Programming Languages

Now that you have an overview of the 8086 CPU, it is time to start thinking about how it is programmed. To run a program, a microcomputer must have the program stored in binary form in successive memory locations, as shown in Fig. 2.12. There are three language levels that can be used to write

series of 1's a
the binary c
thousands o

LABEL FIELD	OP F
NEXT:	

Fig. 2.12

ASSEMBLY

To make prog
programs in a
assembly lang
it can be loade
uses two-, thr
each instructio
you remembe
language mnen
of the English
instruction. Fo
SUB, the mne
mnemonic fo
location to ano

Assembly la
standard form
The first field
label field. A la
to represent an
at the time the
followed by a co
they are just inse
later many uses

The *opcode*
mnemonic for th