Chapter I3
Architecture of the IBM-PC

Revised: Feb. 9, 2011

I3.1 INTRODUCTION

It is very difficult, if not impossible, to implement an operating system for a computer unless you understand the hardware architecture of that computer. Although you will be implementing your projects in Turbo C, you must understand what is happening at the machine level to be able to test and debug the operating system software you are implementing. Some of the software modules to be developed in the MPX-PC project must manipulate registers and other hardware resources in a very explicit manner. It is for this reason that an assembly language programming course is normally a prerequisite for an operating systems course.

In this chapter, we present an overview of the architecture of "IBM PC" computers. This is the architecture for the most widely-used personal computer today, generally called, simply, the "PC". We will generally use this designation as well, but do not forget there are other kinds of "Personal Computers"! The system-level architecture for the PC was designed and originally produced by IBM, but IBM no longer produces PCs. Other manufacturers, such as Dell, Gateway, HP, etc., have taken over this role.

Many extensions and additions have been made to the PC architecture over the years. These affect especially things like internal connections, input and output, networking and storage, and access to large memories. In spite of all this, the core architecture remains pretty much the same. This is necessary to ensure that software written for older PC versions will still run on newer ones.

Like any computer, the PC includes a processor, but the complete system is more than a processor. In the microcomputer world, the processor and the computer are considered two different things, and produced by different manufacturers. The processor used in the IBM-PC was designed and continues to be produced by Intel. Other manufacturers also produce equivalent processors used in some PCs. We will consider these identical and not be concerned with any internal differences among them.

The processor architecture we will be concerned with is the core Intel 8086 architecture used on Intel's earliest "16-bit" processors. This has been extended in many ways by later versions of these processors, but the core architecture is still present and this is all we need.

For our purposes, then, a PC is a computer using the "IBM-PC" architecture that uses an Intel 8086 or 80x86 microprocessor (where x = 2,3,4,...), including those with other names such as Pentium or Celeron made by Intel or other manufacturers. Such a computer should be capable of running "any" programs that will run using early versions of MS-DOS or Microsoft Windows, and usually later versions as well. We hereafter refer to any such computer generically as a PC, and to its processor generically as an 8086.

We need to examine both the architecture of the 8086 processor and the higher-level architecture of the PC itself. This discussion will *not* present *all* aspects of either architecture, nor will it make you an assembly language programmer. Instead we provide a selective overview, concentrating on information required for the MPX projects. Additional details required for specific projects will be presented in the appropriate project chapters.

The computer system hardware to be controlled by MPX-PC consists of the processor, the memory, the interrupt controller, and the I/O controllers and devices. All PCs use the same interrupt controller (Intel 8259A PIC or equivalent), which eliminates *most* incompatibility problems in servicing interrupts. The generation and servicing of interrupts will be covered in detail in Chapter R3.

The detailed hardware structure of I/O ports and devices, including the terminal screen and keyboard, varies greatly across different PC models. However, a high degree of compatibility is normally provided by a low-level software component called the Basic Input Output System (BIOS) that is permanently resident in ROM (read-only memory). Since the MPX-PC operating system must control I/O devices directly (at the hardware level, not via the BIOS), differences among models and devices can create problems. We have chosen to use I/O devices in MPX-PC that have "generic" characteristics across all PCs.

PCs access and control I/O devices using I/O *port addresses* in the *I/O address space*. In some cases, I/O port addresses will vary even for such generic devices. Whenever possible, such differences in addresses are pointed out to allow you to use the appropriate address for your particular PC. These issues are discussed in more detail as needed in later modules.

The architecture of any computer system is composed of processors, memory, I/O devices, and an interconnection system. The heart of such a system is the processor; its structure has the greatest impact on system programming, especially the low-level programming required to construct an operating system. The architecture of the 8086 family of processors is discussed in the following section.

I3.2 THE INTEL 8086 MICROPROCESSOR

The 8086 family

The 8086 family of microprocessors consists of a series of chips with essentially the same architecture and instruction set, although later versions of the processor add a number of new features, especially in connection with memory management. The initial processors, the 8086 and 8088, are referred to as "16-bit" architectures because their registers and internal data paths are 16 bits wide. The design of these processors was derived from the earlier successful 8080, an

-

"8-bit" chip. However, the architecture of the 8086 is in no sense "compatible" with that of the 8080.

The 8088 is identical to the 8086 except for one detail: the 8086 can input and output data in 16-bit units, while the 8088 performs input and output in 8-bit units only. Since MPX performs all I/O in bytes, this difference will not be of concern to us.

Later members of the 8086 family received the designations 80286, 80386, and 80486 (an 80186 was briefly produced but never used). The chip that started development under the designation 80586 is now called the Pentium. Later versions of the Pentium have a variety of designations. These chips differ chiefly by the introduction of increasingly sophisticated mechanisms for managing larger memory spaces, as well as by increasing speed. In addition, the 80386 and later designs include 32-bit data paths throughout. The 80286 was used in the PC/XT architecture, the 80386 in the PC/AT and PS/2 architectures, and the 80486 and Pentium in most of the newer PCs since the later 1990s. Still newer processors are considered to be evolutions of the Pentium (e.g., the Pentium 2, Pentium 4, or “P6”).

Despite the increasing capabilities of the newer family members, all of these chips retain full compatibility with the basic architecture of the 8086. MPX-PC relies only on this basic architecture, and should therefore operate correctly with any processor in this family.

Addressing

The Intel 8086 microprocessors have internal data paths at least 16 bits wide. This means that most data registers are 16 bits, and operations such as addition process up to 16 bits in a single step.

As in most modern architectures, the PC memory is treated as a collection of individually addressable 8-bit *bytes*. Two bytes (16 bits) make up a *word*. A word is addressed by its right-most (least significant) byte address, with the left-most (most significant) byte being at an address 1 greater than its right-most byte. This is sometimes called *little-endian* addressing.

Unlike many computers, the 8086 does *not* require that 16-bit words begin on even boundaries (but memory access is faster when this is the case). Note that the above "right-to-left" addressing of bytes in a word differs from the addressing used in many other computers (including IBM mainframes!) in which the low address of a 16-bit (half) word is the left-most (most significant) byte.

Although 8086 data paths are 16 bits, internal addresses are 20 bits. This makes the total address space one megabyte (2**20). Present-day processors can, of course, work with much larger memories; this relies on advanced mechanisms that will not be of concern to us.

Since data paths and registers in the core architecture are only 16-bits, the processor does not manipulate or store 20-bit addresses as a single unit. Instead, it constructs a 20-bit address as needed from two distinct 16-bit values; this scheme is known as **segmented addressing**.

The pair of values used to create an address in the 8086 consists of a **segment number** and an **offset.** The segment number specifies the starting location of a 64K-byte region called a **segment**, and the offset specifies a relative address *within* that segment. In principle segments may begin anywhere within the 20-bit address space, but the segment number must be stored within 16 bits. Therefore, the segment number is actually taken as the 16 *high-order* bits of the

-

20-bit segment address. The 4 low-order bits are assumed to be zero. This convention requires that all segments begin at an address that is an even multiple of 16, called a **paragraph** boundary.

The procedure for converting a segment number and offset to a complete address is straightforward:

1. Shift the segment number four bits to the left (i.e., multiply by 16), producing the segment address.

2. Add the offset to the segment address to get the complete address.

This can be illustrated by a few examples:

a) Segment number 7F4216; Offset 167C16:

7F4216 --> 7F42016 + 167C16 = 80A9C16

b) Segment number: 39A216; Offset: 000016:

39A216 --> 39A2016 + 000016 = 39A2016

c) Segment number: 39A016; Offset: 002016:

39A016 --> 39A0016 + 002016 = 39A2016

The last two of these examples demonstrate an important observation: the representation of an address as a segment number and offset is *not unique*. Many such pairs correspond to the same address; in particular, increasing the offset by 16 and decreasing the segment number by 1 would result in the same address.

The usual convention is to represent a segmented address in the form *xxxx:yyyy* where *xxxx* is the segment number, *yyyy* is the offset, and both values are assumed to be hexadecimal. Using this notation, the values in example *a* would be represented as 7F42:167C.

How does the processor actually work with segmented addresses, without requiring every instruction to specify two separate 16-bit address components? The answer is that in most cases, only the offset is specified, while the segment number is implied by the operation being performed.

The 8086 architecture includes four 16-bit *segment registers* which contain segment numbers. These registers can be loaded explicitly by machine instructions. Whenever the processor needs a segment number, it fetches it from the appropriate segment register; the choice depends on the operation being performed. There are therefore four different segments that can be addressed by a program at any one time:

- **Data Segment** - contains the (static) data (variables) used by a program. The segment number is stored in the DS register. This value is automatically used for memory references that fetch and store data.
- **Code Segment** - contains the instructions of the program being executed. The segment number is stored in the CS register. This value is automatically used when *instructions* are being fetched.
- **Stack Segment** - the storage area for the current stack. The segment number is stored in the SS register. The stack is used for temporarily saving addresses and data, both explicitly via push and pop instructions, and implicitly during subroutine calls and interrupts.
- **Extra Segment** - This is a spare segment; its number is stored in the ES register. The ES is not used automatically but may be specified explicitly for certain data references.

Note that this scheme separates instructions, data, and stack into three separate 64K segments rather than having these types of information intermixed. Thus a program typically can make use of up to 64K for each of these purposes without special arrangements. However, there is nothing to prevent two segments from being defined to partially overlap, or even to coincide exactly. All that is required is to load the segment registers with values that differ by less than 1000 (base16).

Each memory access operation normally uses the appropriate segment register by default. In addition, data reference instructions may explicitly specify any one of the four registers to override the default choice.

While this at times can be confusing, you should usually be concerned with addresses that are computed using a given offset and a fixed segment number found in one of the CS, DS, SS, or ES segment registers.

You may wish to think of the segment register as being shifted 4 bits to the left or multiplied by 16 prior to being added to the offset. It is extremely important that you understand segment addressing, because in implementing MPX-PC there will be many times when you must verify that segment registers, and other registers used in calculating effective addresses, are correctly saved and restored. Without understanding segment addressing, this verification would be impossible.

Although most address references rely on the current value of a segment register, it is sometimes necessary to specify a full segmented address directly. As an example, the 8086 architecture supports two types of call instructions to invoke subprograms: *intrasegment* calls, which specify only the new offset value, and *intersegment* calls, which specify both the offset and the segment (CS) value. In the MPX-PC project, addresses will generally be represented in their full segmented form, using special 32-bit values known as *far pointers*. Turbo C provides macros to convert between a full segmented address and its segment and offset parts. These facilities are discussed further in Chapter I4.

The location of the offset within the 8086 environment varies. For determining the address of the next instruction to be executed, a 16-bit register called the **instruction pointer** (IP) is used. In addressing data, 16-bit pointer and index registers can be used. Finally, in some

-

instructions, the offset is determined by the sum of one or more 16-bit registers and a displacement found in an instruction. Since we will not be programming in assembler language, the precise details of all these calculations are not really needed. What will be important will be the organization of the hardware stack and the calling conventions for procedures in Turbo C.

The Stack

The 8086, like many modern processors, is a "stack-based architecture." This means that the concept of a pushdown, or Last-In, First-Out, stack is a central element of processor operation.

Essentially a stack is a data buffer. At any time, a set of data occupies a contiguous region within the buffer. One end of this region, called the bottom, is fixed; the other end, called the top, is movable. Generally data is added and removed only at the top of the stack. An operation to add a data item is called a *push*, and an operation to remove a data item is called a *pop*. A pointer must be maintained to the top of the stack; this pointer will change with each push and pop operation.

In the 8086, stacks are designed to "grow" from *higher* addresses to *lower* addresses. In addition, stack pointers are adjusted *before* a push, and *after* a pop. Finally, the normal amount of data pushed or popped is *two bytes* (one 16-bit word). It is important to keep these conventions in mind; they may be different from those of other stack-based architectures. A push operation will:

- Decrement the stack pointer by two.
- Store a data word at the new address given by the stack pointer.

A pop operation will:

- Fetch a data word from the address given by the stack pointer.
- Increment the stack pointer by two.

The structure of an 8086 stack is illustrated below. The figure shows a stack with a capacity of six words, located at (hex) addresses 100 through 10A. Currently the stack contains three items located at addresses 10A, 108, and 106. The top item is at 106. If the stack pointer has the value 106, a push operation will store a data word at location 104, and change the pointer to this value. A pop operation will fetch the word at location 106, and change the pointer to 108.

```
                      ____________________
BOTTOM -->           |                    |  10A
                     |        DATA        |  108
TOP (SP) -->         |____________________|  106
                     |                    |  104
                     |                    |  102
LIMIT  -->           |____________________|  100
```

One important usage of a stack is for explicitly storing certain types of data. For example, in the C language, function parameters are pushed onto a stack before calling the function, and popped off the stack afterwards. The 8086 provides machine instructions to explicitly push and pop words on the stack, using a selected register as the stack pointer.

In a stack-based architecture, however, a critical usage of the stack mechanism is in the automatic saving and restoring of return addresses, and other related information, during subprogram calls and returns. Pushing and popping of return addresses is performed as an integral part of the call and return machine instructions. For this usage a special, dedicated stack pointer register is used, which is *always* expected to point to a valid stack. As noted earlier, the processor supports both intrasegment calls, in which the CS does not change, and intersegment calls, in which the subprogram has a different CS than its caller. In the latter case, the CS must also be pushed onto the stack. Turbo C refers to these cases as *near* calls and *far* calls, respectively.

In the 8086, the SP register is reserved as the stack pointer for subprogram calls and returns. Moreover, this stack is assigned a special segment register, SS. Interrupts are also considered special cases of calls that push information using the SP. Depending on the situation, either one or two words are automatically pushed during a subprogram call, and three words are pushed during an interrupt.

One difficulty shared by most stack-based architectures is detecting stack overflow. The 8086 has no hardware mechanism to guard against storing data beyond the low address boundary of a stack. Software checks are costly and not always feasible, so it is a good idea to ensure that all stacks have a more than adequate capacity. Nevertheless, stack overflow is a very frequent source of troubling software bugs in 8086 programming.

Because of the implicit use of the current stack during interrupts, a problem arises when the SS and SP are both to be changed. If an interrupt occurs *after one* of these registers is changed but *before the other* is also loaded, the SS-SP pair will point to an invalid stack, and the interrupt mechanism will store information at improper addresses.

In the earliest 8086 chips this was a serious problem, requiring a software discipline to disable interrupts whenever these registers were changed. The chip design was then revised to provide a special interlock, so *no interrupt can occur immediately after the SS is modified*. This mechanism is used on all current models, but it still requires a strict discipline: If the SS and SP are both changed, they must always be modified by *two consecutive machine instructions, in that order*. In Chapter I4 we will discuss how to accomplish this in the Turbo C environment.

Addressing Registers

As we explained previously, pointer and index registers can be used in calculating the 16-bit **offset** for eventually determining a 20-bit effective address. There are four such registers that can be used for offset calculation: the *Stack Pointer* (*SP*); the *Base Pointer* (*BP*); the *Source Index* (*SI*); and the *Destination Index* (*DI*).

-

Stack Pointer

The stack pointer (`SP`) is used primarily in pushing values and addresses onto the stack and popping values and addresses from the stack, as discussed above. The use of the stack pointer will be discussed in more detail in Chapter R3.

Base Pointer

The 16-bit base pointer (`BP`) can be used to compute offsets for data in both the stack segment and the data segment. In Turbo C it is often used as a temporary register for saving the value of the stack pointer. This usage is discussed in Chapter I4. It may be necessary to directly access the BP to save the correct stack pointer value in implementing some components of MPX-PC.

Source and Destination Index Registers

The source and destination index registers (`SI` and `DI` respectively) are combined with other 16-bit entities in computing offsets of data. They can be used in **base plus displacement addressing** and in **base plus index plus displacement addressing** to compute 16-bit offsets that are combined with some segment register to create the 20-bit effective address.

Addressing Modes

Addressing in the 8086 is quite variable; offsets are computed by one of many methods, and one of four segment registers can be used as the segment portion in computing the effective address. Since we will not be working with assembly language, we will not present the details of the various modes of 8086 addressing.

Addressing Models

There are many possible strategies for defining code, data, stack, and extra segments. Each register could contain a different value. For MPX we will use a standard model defined by Turbo C known as the **large memory model.** This model is based on the assumption that code, data and stack segments are distinct, and allows code (though not static data) to span multiple 64K segments. In this model, all pointers and addresses are expressed in their complete segmented form (**far pointers**). Memory models are discussed further in Chapter I4.

Programs that are executed under MPX-PC will have their own code segment and data segment values based on where these programs *dynamically* load into free memory space.

Data Registers

The 8086 has a group of four 16-bit **data registers** (also called **general registers** or **arithmetic registers)** that can alternately be treated as eight 8-bit registers. The names of the 16-bit registers are `AX`, `BX`, `CX` and `DX`; the names of the 8-bit registers are `AH`, `AL`, `BH`, `BL`, `CH`, `CL`, `DH`, and `DL`. The "L" and "H" registers are a part of the "X" registers, with the "L" registers being

-

the low order bytes and "H" registers being the high order bytes of the "X" registers as shown below:

AX register:	AH	AL
BX register:	BH	BL
CX register:	CH	CL
DX register:	DH	DL

The letters A, B, C, and D stand for **accumulator, base, count,** and **data,** respectively. These data registers are the *only* registers in the 8086 that you may reference as either 8-bit or 16-bit registers. Moving data to the "L" portion of these registers does not affect the "H" portion and vice versa. All these registers are available for general programming use. However, there are some special implicit uses of these registers as the names themselves might imply. The `AX` register corresponds to the accumulator of early microprocessors and can be used for immediate operations in which one of the operands is contained within the instruction. `AL` is used in `IN` instructions to transfer 8-bit data from an I/O device to `AL` and in `OUT` instructions to transfer data from `AL` to the I/O device register.

The `BX` register can serve as an addressing register as well as a data register. When used for addressing, it is one of the components that may be added when determining an effective address. The `CX` register is used as a count in many instructions that determine the number of times a loop is to be executed. `CL` contains shift counts for multi-bit shift and rotate instructions. Finally, the `DX` register is used as an extension of the accumulator in operations requiring 32-bit calculations (e.g., in multiplication and division).

Summary of 8086 Registers

As we have seen, the 8086 microprocessor contains twelve 16 bit registers in three groups: data registers, pointer/index registers, and segment registers:

Data Registers	Pointer/Index Registers	Segment Registers
AX = AH, AL	SI	CS
BX = BH, BL	BI	DS
CX = CH, CL	BP	SS
DX = DH, DL	SP	ES

If you are familiar with other processor architectures, you know there must be one more internal register in the 8086, the register that is often referred to as the **program status word (PSW).**

This register typically contains condition codes (that specify the results of comparisons or the values moved to registers) and bits that determine whether or not certain interrupts can be

-

generated. This register is called the **flag register** or simply the **flags** in the 8086. The format of the flag register is:

```
   15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
  ------------------------------------------------
 | xx xx xx xx OF DF IF TF SF ZF xx AF xx PF xx CF|
  ------------------------------------------------

  CF = CARRY FLAG                PF = PARITY FLAG
  AF = AUXILIARY CARRY FLAG      ZF = ZERO FLAG
  SF = SIGN FLAG                 TF = TRAP FLAG
  IF = INTERRUPT ENABLE FLAG     DF = DIRECTION FLAG
  OF = OVERFLOW FLAG             xx = not used
```

The flag register is very critical in the MPX-PC environment because programs (processes) will temporarily lose control of the CPU and later regain control. The flags must be restored to the exact values they had when control of the CPU was taken away, if the program is to continue execution correctly. This will be discussed in detail in Chapter R3.

I3.3 PC INPUT/OUTPUT

The 8086 performs I/O using a *separate I/O space* rather than *memory-mapped I/O* (see text). There are two input/output instructions, `IN` and `OUT`. Every device has one or more device registers that are used to control or perform I/O. Devices are controlled by manipulating these registers using these `IN` and `OUT` instructions. Each device register is assigned a unique address in the PC I/O address space (0 to 65,535).

The `IN` instruction causes a byte to be transferred from the addressed device register to the `AL` register. The port (device register address) at the machine level is usually specified either as an immediate operand in the `IN` instruction or by specifying `DX` as a register containing the port address. Similarly, the `OUT` instruction transfers a byte from the `AL` register to the port specified by an immediate operand in the `OUT` instruction or in the `DX` register. The following four instructions illustrate the use of the `IN` and `OUT` instructions:

```
IN    AL,020H    ; Input byte from port 20
IN    AL,DX      ; Input byte from port specified in DX
OUT   021H,AL    ; Output byte in AL to port 21
OUT   DX,AL      ; Output byte in AL to port specified in DX
```

Most processor versions support word I/O operations, in which case the `AX` register is specified rather than the `AL` register. In MPX we shall use only byte I/O operations in controlling I/O devices.

The assignment of port addresses and techniques for accessing these ports (device registers) in Turbo C will be discussed in Chapter R5.

-

I3.4 INTERRUPTS

Like most computers, the IBM-PC provides an interrupt mechanism, as explained in the text, for efficient handling of I/O transfers and external events. The same mechanism is also used for sytem calls. The 8086 architecture provides for only two distinct external interrupt signals. The PC extends this mechanism to support multiple prioritized interrupts using an external device called the Programmable Interrupt Controller (PIC).

A major task in MPX-PC is the implementation of various interrupt handlers to deal with service requests and to handle I/O devices. Chapter R3 focuses on the concept of interrupts and provides the details of the PC interrupt structure. Chapter R5 discusses the PIC and the use of interrupts in device drivers.

I3.5 MEMORY USAGE

The segmented addressing scheme of the 8086 processors provides reasonably straightforward access to a 20-bit address space that can contain a maximum of one megabyte of memory. However, the PC sets aside a significant amount of this address space for special purposes. Moreover, even if the full 20-bit space could be freely used, many present-day applications require still more memory. Several mechanisms are provided by newer PC models to support memory beyond the one megabyte limit. MPX-PC requires only the basic memory provided by almost all models, but some versions of Turbo C require additional memory. This section briefly overviews PC memory usage.

The range of addresses available in the PC, expressed in 20-bit hexadecimal form, runs from 00000 to FFFFF. The PC reserves a small portion of the lowest addresses and a large portion of the highest addresses.

First, addresses 00000 through 003FF are set aside strictly for interrupt vectors. After this, several hundred bytes are reserved for use as a working area by the BIOS routines (see below).

The PC further reserves all addresses from A0000 to FFFFF for special use. The PC uses a bit-mapped video system, in which the image displayed on the screen is stored directly in main memory. Addresses from A0000 to BFFFF are used for video display memory. Also, a portion of the operating system itself is permanently stored in a read-only memory (ROM). This portion, called the BIOS, provides control routines for various I/O devices. Locations beginning at C0000 are reserved for this read-only memory.

This scheme leaves a bit less than 640K bytes for the rest of the operating system and for the application programs. In early PCs this was considered generous, and most machines actually had less physical memory than 640K. For newer applications, though, 640K is not sufficient. For example, even Turbo C++ V3.0 requires an additional megabyte; applications and systems such as Windows or Microsoft Office require dozens to hundreds of megabytes.

Beginning with the 80286, processor models included support for some type of higher-level memory management, generally using a separate chip. This provides a separate level of address translation, usually transparent to programs, which can be used to access larger amounts of memory. Each new processor version introduced different, more sophisticated mechanisms. Beginning with the 80386 it has been possible to support true virtual memory as described in the text.

-

Two important models have emerged for convenient use of additional memory through memory management. The first mechanism, *expanded memory*, allows programs to recover the missing portions of the basic one megabyte space by mapping higher addresses to real memory during normal operation. The second mechanism, *extended memory*, supports access to memory above the 1-MB limit. We will not discuss the details of these mechanisms; it is enough here to point out that it is increasingly common for programs to require more memory than the basic PC architectures support, and the newer processors provide mechanisms to make this possible.

Most recent PC models also provide support for *virtual memory*, in which additional (apparent) memory space is obtained by swapping unused data to a disk. We will not require this mechanism for MPX and we will not discuss it further.

-