Inside the microprocessor, this math is performed by shifting the segment pointer (0x135F) left by four bits and then adding the offset pointer (0x0102) as shown below.

	1	3	5	F	0	Segment pointer
+		0	1	0	2	Offset pointer
	1	3	6	F	2	Effective address

This segmented addressing scheme has some awkward characteristics. First, programs must organize their instructions and data into 64-kB chunks and properly keep track of which portions are being accessed. If data outside of the current segments is desired, the appropriate segment register must be updated. Second, the same memory location can be represented by multiple combinations of segment and offset values, which can cause confusion in sorting out which instruction is accessing which location in memory. Nonetheless, programmers and the manufacturers of their development tools have figured out ways to avoid these traps and others like them.

Instructions that reference memory implicitly or explicitly determine which offset pointer is added to which segment register to yield the desired effective address. For example, a push or pop instruction inherently uses the stack pointer in combination with the stack segment register. However, an instruction to move data from memory to the accumulator can use one of multiple pointer registers relative to any of the segment registers.

The 8086's reset and interrupt vectors are located at opposite ends of the memory space. On reset, the instruction pointer is set to 0xFFFF0, and the microprocessor begins executing instructions from this address. Therefore, rather than being a true vector, the 16-byte reset region contains normal executable instructions. The interrupt vectors are located at the bottom of the memory space starting from address 0, and there are 256 vectors, one for each of the 256 interrupt types. Each interrupt vector is composed of a 2-byte segment address and a 2-byte offset address, from which a 20-bit effective address is calculated. When the 8086's INTR pin is driven high, an interrupt acknowledge process begins via the INTA* output pin. The 8086 pulses INTA* low twice and, on the second pulse, the interrupting peripheral drives an interrupt type, or vector number, onto the eight lower bits of the data bus. The vector number is used to index into the interrupt vector table by multiplying it by 4 (shifting left by two bits), because each vector consists of four bytes. For example, interrupt type 0x03 would cause the microprocessor to fetch four bytes from addresses 0x0C through 0x0F. Interrupts triggered by the INTR pin are all maskable via an internal control bit. Software can also trigger interrupts of various types via the `INT` instruction. A nonmaskable interrupt can be triggered by external hardware via the NMI pin. NMI initiates the type-2 interrupt service routine at the address indicated by the vector at 0x08-0x0B.

Locating the reset boot code at the top of memory and the interrupt vectors at the bottom often leads to an 8086 computer architecture with ROM at the top and some RAM at the bottom. ROM must be at the top, for obvious reasons. Placing the interrupt vector table in RAM enables a flexible system in which software applications can install their own ISRs to perform various tasks. On the original IBM PC platform, it was not uncommon for programs to insert their own ISR addresses into certain interrupt vectors located in RAM. The system timer and keyboard interrupts were common objects of this activity. Because the PC's operating system already implemented ISRs for these interrupts, the program could redirect the interrupt vector to its own ISR and then call the system's default ISR when its own ISR completed execution. If properly done, this interrupt chaining process could add new features to a PC without harming the existing housekeeping chores performed by the standard ISRs. Chaining the keyboard interrupt could enable a program that is normally dormant to pop up each time a particular key sequence is pressed.

Despite its complexity and 16-bit processing capability, the 8086 was originally housed in a 40-pin DIP—the same package used for most 8-bit processors of the time. Intel chose to use a multiplexed address/data scheme similar to that used on the 8051 microcontroller, thereby saving 16 pins. The 8086's 20-bit address bus is shared by the data bus on the lower 16 bits and by status flags on the upper 4 bits. Combined with additional signals, these status flags control the microprocessor's bus interface. As with Intel's other microprocessors, the 8086 contains separate address spaces for memory and I/O devices. A control pin on the chip indicates whether a transaction is memory or I/O. While the memory space is 1 MB in size, the I/O space is only 64 kB. The 8086 bus interface operates in one of two modes, minimum and maximum, determined by a control pin tied either high or low, respectively. In each of these two modes, many of the control and status pins take on different functions. In minimum mode, the control signals directly drive a standard "Intel-style" bus similar to that of the 8080 and 8051, with read and write strobes and address latch enable. Other signals include a READY signal for inserting wait states for slow peripherals and a bus grant/acknowledge mechanism for supporting DMA or similar bus-sharing peripherals. Minimum mode is designed for smaller systems in which little address decoding logic is necessary to interface the 8086 to memory and peripherals devices. Maximum mode is designed for larger systems where an Intel companion chip, the 8288 bus controller, integrates more complex bus control logic onto an off-the-shelf IC. In maximum mode, certain status and control pins communicate more information about what type of transaction is being performed at any given time, enabling the 8288 to take appropriate action.

The 8086's 16-bit data bus is capable of transacting a single byte at a time for purposes of accessing byte-wide peripherals. One early advantage of the 8086 was its backward bus compatibility with the 8080/8085. In the 1970s, Intel manufactured a variety of I/O peripherals such as timers and parallel I/O devices for their eight-bit microprocessors. The 8086's ability to perform byte-wide transactions enabled easy reuse of existing eight-bit peripheral products. Two signals, byte high enable (BHE*) and address bit zero (A[0]), communicate the width and active byte of each bus transaction as shown in Table 6.3.

TABLE 6.3 8086 Bus Sizing

BHE*	A[0]	Transaction Type
0	0	16-bit transaction
0	1	8-bit transaction: high byte (odd address)
1	0	8-bit transaction: low byte (even address)
1	1	Undefined

Intel's microprocessors follow the *little-endian* byte ordering convention. *Little-endian* refers to the practice of locating the LSB of a multibyte quantity in a lower address and the MSB in a higher address. In a little-endian 16-bit microprocessor, the value 0x1234 would be stored in memory by locating 0x12 into address 1 and 0x34 into address 0. *Big-endian* is the opposite: locating the LSB in the higher address and the MSB in the lower address. Therefore, a big-endian 16-bit microprocessor would store 0x12 into address 0 and 0x34 into address 1. To clarify the difference, Table 6.4 shows little-endian versus big-endian for 16- and 32-bit quantities as viewed from a memory chip's perspective. Here, ADDR represents the base address of a multibyte data element.

Proponents of little-endian argue that it makes better sense, because the low byte goes into the low address. Proponents of big-endian argue that it makes better sense, because data is stored in