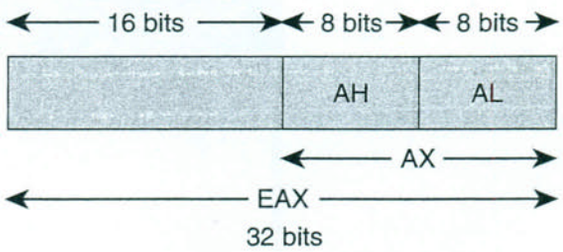

FIGURE 4.22 EAX Register, Broken into Parts

a **superscalar** design. This means the CPU had multiple ALUs and could issue more than one instruction per clock cycle (i.e., run instructions in parallel). The Pentium Pro added branch prediction, and the Pentium II added MMX technology (which most will agree was not a huge success) to deal with multimedia. The Pentium III added increased support for 3D graphics (using floating-point instructions). Historically, Intel used a classic CISC approach throughout its processor series. The more recent Pentium II and III used a combined approach, employing CISC architectures with RISC cores that could translate from CISC to RISC instructions. Intel was conforming to the current trend by moving away from CISC and toward RISC.

The seventh-generation family of Intel CPUs introduced the Intel **Pentium IV** (also known as the **Pentium 4**) processor. This processor differs from its predecessors in several different ways, many of which are beyond the scope of this text. Suffice it to say that the Pentium IV processor has clock rates of 1.4 and 1.7 GHz, uses no less than 42 million transistors for the CPU, and implements a **NetBurst microarchitecture**. (The processors in the Pentium family, up to this point, had all been based on the same **microarchitecture**, a term used to describe the architecture below the instruction set.) This type of architecture includes four salient improvements: hyper-pipelining, a wider instruction pipeline (pipelining is covered in Chapter 5) to handle more instructions concurrently; a rapid execution engine (the Pentium IV has two arithmetic logic units); an execution trace cache, a cache that holds decoded instructions so if they are used again, they do not have to be decoded again; and a 400 MHz bus. This has made the Pentium IV an extremely useful processor for multimedia applications.

The Pentium IV processor also introduced **hyperthreading (HT)**. **Threads** are tasks that can run independently of one another within the context of the same process. A thread shares code and data with the parent process but has its own resources, including a stack and instruction pointer. Because multiple child threads share with their parent, threads require fewer system resources than if each were a separate process. Systems with more than one processor take advantage of thread processing by splitting instructions so that multiple threads can execute on the processors in parallel. However, Intel's HT enables a single physical processor to simulate two logical

(or virtual) processors—the operating system actually sees two processors where only one exists. (To take advantage of HT, the operating system must recognize thread processing.) HT does this through a mix of shared, duplicated, and partitioned chip resources, including registers, math units, counters, and cache memory.

HT duplicates the architectural state of the processor but permits the threads to share main execution resources. This sharing allows the threads to utilize resources that might otherwise be idle (e.g., on a cache miss), resulting in up to a 40% improvement in resource utilization and potential performance gains as high as 25%. Performance gains depend on the application, with compute-intensive applications seeing the most significant gain. Commonplace programs, such as word processors and spreadsheets, are mostly unaffected by HT technology.

What's in a Name?

Intel Corporation makes approximately 80% of the CPUs used in today's microcomputers. It all started with the 4-bit 4004, which in 1971 was the first commercially available microprocessor, or "CPU on a chip." Four years later, Intel's 8-bit 8080 with 6000 transistors was put into the first personal computer, the Altair 8800. As technology allowed more transistors per chip, Intel kept pace by introducing the 16-bit 8086 in 1978 and the 8088 in 1979 (both with approximately 29,000 transistors). These two processors truly started the personal computer revolution, as they were used in the IBM personal computer (later dubbed the XT) and became the industry standard.

The 80186 was introduced in 1980, and although buyers could choose from an 8-bit or a 16-bit version, the 80186 was never used in personal computers. In 1982, Intel introduced the 80286, a 16-bit processor with 134,000 transistors. In fewer than 5 years, over 14 million personal computers were using the 80286 (which most people shortened to simply "286"). In 1985, Intel came out with the first 32-bit microprocessor, the 80386. The 386 multitasking chip was an immediate success, with its 275,000 transistors and 5 million instructions-per-second operating speed. Four years later, Intel introduced the 80486, which had an amazing 1.2 million transistors per chip and operated at 16.9 million instructions per second! The 486, with its built-in math coprocessor, was the first microprocessor to truly rival mainframe computers.

With such huge success and name recognition, why then, did Intel suddenly stop using the 80x86 moniker and switch to *Pentium* in 1993? By this time, many companies were copying Intel's designs and using the same numbering scheme. One of the most successful of these was Advanced Micro Device (AMD). The AMD486 processor had already found its way into many portable and desktop computers. Another was Cyrix with its 486SLC chip. Before introducing its next processor, Intel asked the U.S. Patent and Trademark Office if the company could trademark the name "586." In the United States, numbers

cannot be trademarked. (Other countries do allow numbers as trademarks, such as Peugeot's trademark three-digit model numbers with a central zero.) Intel was denied its trademark request and switched the name to *Pentium*. (The astute reader will recognize that *pent* means five, as in *pentagon*.)

It is interesting to note that all of this happened at about the same time as Intel began using its ubiquitous "Intel inside" stickers. It is also interesting that AMD introduced what it called the PR rating system, a method of comparing their x86 processor to Intel's processor. PR stands for "Performance Rating" (not "Pentium Rating" as many people believe) and was intended to guide consumers regarding a particular processor's performance as compared to that of a Pentium.

Intel has continued to manufacture chips using the Pentium naming scheme. The first Pentium chip had 3 million transistors, operated at 25 million instructions per second, and had clock speeds from 60 to 200 MHz. Intel produced many different name variations of the Pentium, including the Pentium MMX in 1997, which improved multimedia performance using the MMX instruction set.

Other manufacturers have also continued to design chips to compete with the Pentium line. AMD introduced the AMD5x86, and later the K5 and K6, to compete with Pentium MMX technology. AMD gave its 5x86 processor a "PR75" rating, meaning this processor was as fast as a Pentium running at 75 MHz. Cyrix introduced the 6x86 chip (or M1) and MediaGX, followed by the Cyrix 6x86MX (M2), to compete with the Pentium MMX.

Intel moved on to the Pentium Pro in 1995. This processor had 5.5 million transistors but had only a slightly larger die than the 4004 which was introduced almost 25 years earlier. The Pentium II (1997) was a cross between the Pentium MMX and the Pentium Pro and contained 7.5 million transistors. AMD continued to keep pace and introduced the K6-2 in 1998, followed by the K6-3. In an attempt to capture more of the low-end market, Intel introduced the Celeron, an entry-level version of the Pentium II with less cache memory.

Intel released the Pentium III in 1999. This chip, housing 9.5 million transistors, used the SSE instruction set (which is an extension to MMX). Intel continued with improvements to this processor by placing cache directly on the core, making caching considerably faster. AMD released the Athlon line of chips in 1999 to compete with the Pentium III. (AMD continues to manufacture the Athlon line to this day.) In 2000, Intel released the Pentium IV, and depending on the particular core, this chip has from 42 to 55 million transistors!

Clearly, changing the name of its processors from the x86 designation to a Pentium-based series has had no negative effects on Intel's success. However, because Pentium is one of the most recognized trademarks in the processor world, industry watchers were surprised when Intel introduced its 64-bit Itanium processor without including *Pentium* as part of the name. Some people believe that this chip name has backfired and their comparison of this chip to a sinking ship has prompted some to call it the *Itanic*.

Intel recently submitted a patent bid to trademark "Intel VIIV." There is considerable speculation as to what VIIV could mean. VI and IV are the Roman

numerals for 6 and 4, which could reference 64-bit technology. VIIV might also be representative of Intel's new dual-core chips and could mean 5–2–5, or two Pentium 5 cores.

Although this discussion has given a timeline of Intel's processors, it also shows that, for the past 30 years, Moore's law has held with remarkable accuracy. And we have looked at only Intel and Intel clone processors. There are many other microprocessors we have not mentioned, including those made by Motorola, Zilog, TI, and RCA, to name only a few. With continually increasing power and decreasing costs, there is little wonder that microprocessors have become the most prevalent type of processor in the computer market. Even more amazing is that there is no sign of this trend changing at any time in the near future.

The introduction of the **Itanium** processor in 2001 marked Intel's first 64-bit chip (IA-64). Itanium includes a register-based programming language and a very rich instruction set. It also employs a hardware emulator to maintain backward compatibility with IA-32/x86 instruction sets. This processor has four integer units, two floating-point units, a significant amount of cache memory at four different levels (we study cache levels in Chapter 6), 128 floating-point registers, 128 integer registers, and multiple miscellaneous registers for dealing with efficient loading of instructions in branching situations. Itanium can address up to 16 GB of main memory.

The assembly language of an architecture reveals significant information about that architecture. To compare MARIE's architecture to Intel's architecture, let's return to Example 4.1, the MARIE program that used a loop to add five numbers. Let's rewrite the program in x86 assembly language, as seen in Example 4.4. Note the addition of a `Data` segment directive and a `Code` segment directive.

EXAMPLE 4.4 A program using a loop to add five numbers written to run on a Pentium.

```
        .DATA
Num1    EQU  10                ; Num1 is initialized to 10
        EQU  15                ; Each word following Num1 is initialized
        EQU  20
        EQU  25
        EQU  30
Num     DB   5                 ; Initialize the loop counter
Sum     DB   0                 ; Initialize the Sum

        .CODE
        LEA  EBX, Num1         ; Load the address of Num1 into EBX
        MOV  ECX, Num          ; Set the loop counter
        MOV  EAX, 0            ; Initialize the sum
```

```
        MOV  EDI, 0              ; Initialize the offset (of which number to add)
Start:  ADD  EAX, [EBX+EDI*4]    ; Add the EBXth number to EAX
        INC  EDI                 ; Increment the offset by 1
        DEC  ECX                 ; Decrement the loop counter by 1
        JG   Start               ; If counter is greater than 0, return to Start
        MOV  Sum, EAX            ; Store the result in Sum
```

We can make this program easier to read (which also makes it look less like MARIE's assembly language) by using the loop statement. Syntactically, the loop instruction resembles a jump instruction, in that it requires a label. This loop can be rewritten as follows:

```
        MOV  ECX, Num                ; Set the counter
Start:  ADD  EAX, [EBX + EDI + 4]
        INC  EDI
        LOOP Start
        MOV  Sum, EAX
```

The loop statement in x86 assembly is similar to the `do...while` construct in C, C++, or Java. The difference is that there is no explicit loop variable—the ECX register is assumed to hold the loop counter. Upon execution of the loop instruction, the processor decreases ECX by one, and then tests ECX to see if it is equal to 0. If it is not 0, control jumps to `Start`; if it is 0, the loop terminates. The loop statement is an example of the types of instructions that can be added to make the programmer's job easier, but which aren't necessary for getting the job done.

4.14.2 MIPS Architectures

The MIPS family of CPUs has been one of the most successful and flexible designs of its class. The MIPS R3000, R4000, R5000, R8000, and R10000 are some of the many registered trademarks belonging to MIPS Technologies, Inc. MIPS chips are used in embedded systems, in addition to computers (such as Silicon Graphics machines) and various computerized toys (Nintendo and Sony use the MIPS CPU in many of their products). Cisco, a very successful manufacturer of Internet routers, uses MIPS CPUs as well.

The first MIPS ISA was MIPS I, followed by MIPS II through MIPS V. The current ISAs are referred to as MIPS32 (for the 32-bit architecture) and MIPS64 (for the 64-bit architecture). Our discussion in this section focuses on MIPS32. It is important to note that MIPS Technologies made a decision similar to that of Intel—as the ISA evolved, backward compatibility was maintained. And, like Intel, each new version of the ISA included operations and instructions to

Naming Convention	Register Number	Value Put in Register
$v0-$v1	2–3	Results, expressions
$a0-$a3	4–7	Arguments
$t0-$t7	8–15	Temporary values
$s0-$s7	16–23	Saved values
$t8-$t9	24–25	More temporary values

TABLE 4.9 MIPS32 Register Naming Convention

improve efficiency and handle floating-point values. The new MIPS32 and MIPS64 architectures have significant improvements in VLSI technology and CPU organization. The end result is notable cost and performance benefits over traditional architectures.

Like IA-32 and IA-64, the MIPS ISA embodies a rich set of instructions, including arithmetic, logical, comparison, data transfer, branching, jumping, shifting, and multimedia instructions. MIPS is a **load/store architecture**, which means that all instructions (other than the load and store instructions) must use registers as operands (no memory operands are allowed). MIPS32 has 168 32-bit instructions, but many are similar. For example, there are six different add instructions, all of which add numbers, but they vary in the operands and registers used. This idea of having multiple instructions for the same operation is common in assembly language instruction sets. Another common instruction is the MIPS NOP instruction, which does nothing except eat up time (NOPs are used in pipelining as we see in Chapter 5).

The CPU in a MIPS32 architecture has thirty-two 32-bit general-purpose registers numbered r0 through r31. (Two of these have special functions: r0 is hardwired to a value of 0 and r31 is the default register for use with certain instructions, which means it does not have to be specified in the instruction itself.) In MIPS assembly, these 32 general-purpose registers are designated $0, $1, ... , $31. Register 1 is reserved, and registers 26 and 27 are used by the operating system kernel. Registers 28, 29, and 30 are pointer registers. The remaining registers can be referred to by number, using the naming convention shown in Table 4.9. For example, you can refer to register 8 as $8 or as $t0.

There are two special purpose registers, HI and LO, which hold the results of certain integer operations. Of course, there is a PC register as well, giving a total of three special-purpose registers.

MIPS32 has thirty two 32-bit floating-point registers that can be used in single-precision floating-point operations (with double-precision values being stored in even-odd pairs of these registers). There are four special-purpose floating-point control registers for use by the floating-point unit.

Let's continue our comparison by writing the programs from Examples 4.1 and 4.4 in MIPS32 assembly language.

EXAMPLE 4.5

```
        . . .
                .data
        # $t0 = sum
        # $t1 = loop counter Ctr
Value:  .word 10, 15,20,25,30
        Sum = 0
        Ctr = 5
        .text
        .global main          # Declaration of main as a global variable
main:   lw $t0, Sum           # Initialize register containing sum to zero
        lw $t1, Ctr           # Copy Ctr value to register
        la $t2, value         # $t2 is a pointer to current value
while:  blez $t1, end_while   # Done with loop if counter <= 0
        lw $t3, 0($t2)        # Load value offset of 0 from pointer
        add $t0, $t0, $t3     # Add value to sum
        addi $t2, $t2, 4      # Go to next data value
        sub $t1, $t1, 1       # Decrement Ctr
        b while               # Return to top of loop
        la $t4, sum           # Load the address of sum into register
        sw $t0, 0($t4)        # Write the sum into memory location sum
        . . .
```

This is similar to the Intel code in that the loop counter is copied into a register, decremented during each interation of the loop, and then checked to see if it is less than or equal to 0. The register names may look formidable, but they are actually easy to work with once you understand the naming conventions.

If you are interested in writing MIPS programs, but don't have a MIPS machine, there are several simulators that you can use. The most popular is **SPIM**, a self-contained simulator for running MIPS R2000/R3000 assembly language programs. SPIM provides a simple debugger and implements almost the entire set of MIPS assembly instructions. The SPIM package includes source code and a full set of documentation. It is available for many flavors of Unix (including Linux), Windows, and Windows (DOS), as well as Macintosh. For further information, see the references at the end of this chapter.

If you examine Examples 4.1, 4.4, and 4.5, you can see that the instructions are quite similar. Registers are referenced in different ways and have different names, but the underlying operations are basically the same. Some assembly languages have larger instruction sets, allowing the programmer more choices for coding various algorithms. But, as we have seen with MARIE, a large instruction set is not absolutely necessary to get the job done.