point. There is a fixed overhead involved in moving data from memory to registers. The amount of control logic for the pipeline also increases in size proportional to the number of stages, thus slowing down total execution. In addition, there are several conditions that result in "pipeline conflicts," which keep us from reaching the goal of executing one instruction per clock cycle. These include:

- Resource conflicts
- Data dependencies
- Conditional branch statements

Resource conflicts are a major concern in instruction-level parallelism. For example, if one instruction is storing a value to memory while another is being fetched from memory, both need access to memory. Typically this is resolved by allowing the instruction executing to continue, while forcing the instruction fetch to wait. Certain conflicts can also be resolved by providing two separate pathways: one for data coming from memory and another for instructions coming from memory.

Data dependencies arise when the result of one instruction, not yet available, is to be used as an operand to a following instruction. There are several ways to handle these types of pipeline conflicts. Special hardware can be added to detect instructions whose source operands are destinations for instructions further up the pipeline. This hardware can insert a brief delay (typically a no-op instruction that does nothing) into the pipeline, allowing enough time to pass to resolve the conflict. Specialized hardware can also be used to detect these conflicts and route data through special paths that exist between various stages of the pipeline. This reduces the time necessary for the instruction to access the required operand. Some architectures address this problem by letting the compiler resolve the conflict. Compilers have been designed that reorder instructions, resulting in a delay of loading any conflicting data but having no effect on the program logic or output.

Branch instructions allow us to alter the flow of execution in a program, which, in terms of pipelining, causes major problems. If instructions are fetched one per clock cycle, several can be fetched and even decoded before a preceding instruction, indicating a branch, is executed. Conditional branching is particularly difficult to deal with. Many architectures offer **branch prediction**, using logic to make the best guess as to which instructions will be needed next (essentially, they are predicting the outcome of a conditional branch). Compilers try to resolve branching issues by rearranging the machine code to cause a **delayed branch**. An attempt is made to reorder and insert useful instructions, but if that is not possible, no-op instructions are inserted to keep the pipeline full. Another approach used by some machines given a conditional branch is to start fetches on both paths of the branch and save them until the branch is actually executed, at which time the "true" execution path will be known.

In an effort to squeeze even more performance out of the chip, modern CPUs employ superscalar design (introduced in Chapter 4), which is one step beyond pipelining. Superscalar chips have multiple ALUs and issue more than one

instruction in each clock cycle. The clock cycles per instruction can actually go below one. But the logic to keep track of hazards becomes even more complex; more logic is needed to schedule operations than to do them. But even with complex logic, it is hard to schedule parallel operations "on the fly."

The limits of dynamic scheduling have led machine designers to consider a very different architecture, **explicitly parallel instruction computers (EPIC)**, exemplified by the Itanium architecture discussed in Chapter 4. EPIC machines have very large instructions (recall the instructions for the Itanium are 128 bits), which specify several operations to be done in parallel. Because of the parallelism inherent in the design, the EPIC instruction set is heavily compiler dependent (which means a user needs a sophisticated compiler to take advantage of the parallelism to gain significant performance advantages). The burden of scheduling operations is shifted from the processor to the compiler, and much more time can be spent in developing a good schedule and analyzing potential pipeline conflicts.

To reduce the pipelining problems due to conditional branches, the IA-64 introduced **predicated** instructions. Comparison instructions set predicate bits, much like they set condition codes on the x86 machine (except that there are 64 predicate bits). Each operation specifies a predicate bit; it is executed only if the predicate bit equals 1. In practice, all operations are performed, but the result is stored into the register file only if the predicate bit equals 1. The result is that more instructions are executed, but we don't have to stall the pipeline waiting for a condition.

There are several levels of parallelism, varying from the simple to the more complex. All computers exploit parallelism to some degree. Instructions use words as operands (where words are typically 16, 32, or 64 bits in length), rather than acting on single bits at a time. More advanced types of parallelism require more specific and complex hardware and operating system support.

Although an in-depth study of parallelism is beyond the scope of this text, we would like to take a brief look at what we consider the two extremes of parallelism: program level parallelism (PLP) and instruction level parallelism (ILP). PLP actually allows parts of a program to run on more than one computer. This may sound simple, but it requires coding the algorithm correctly so that this parallelism is possible, in addition to providing careful synchronization between the various modules.

ILP involves the use of techniques to allow the execution of overlapping instructions. Essentially, we want to allow more than one instruction within a single program to execute concurrently. There are two kinds of ILP. The first type decomposes an instruction into stages and overlaps these stages. This is exactly what pipelining does. The second kind of ILP allows individual instructions to overlap (that is, instructions can be executed at the same time by the processor itself).

In addition to pipelined architectures, superscalar, superpipelining, and very long instruction word (VLIW) architectures exhibit ILP. Superscalar architectures (as you may recall from Chapter 4) perform multiple operations at the same time by employing parallel pipelines. Examples of superscalar architectures include IBM's PowerPC, Sun's UltraSparc, and DEC's Alpha. **Superpipelining** architectures combine superscalar concepts with pipelining, by dividing the pipeline

stages into smaller pieces. The IA-64 architecture exhibits a **VLIW** architecture, which means each instruction can specify multiple scalar operations (the compiler puts multiple operations into a single instruction). Superscalar and VLIW machines fetch and execute more than one instruction per cycle.

5.6 REAL-WORLD EXAMPLES OF ISAs

Let's return to the two architectures we discussed in Chapter 4, Intel and MIPS, to see how the designers of these processors chose to deal with the issues introduced in this chapter: instruction formats, instruction types, number of operands, addressing, and pipelining. We'll also introduce the Java Virtual Machine to illustrate how software can create an ISA abstraction that completely hides the real ISA of the machine.

5.6.1 Intel

Intel uses a little endian, two-address architecture, with variable-length instructions. Intel processors use a register-memory architecture, which means all instructions can operate on a memory location, but the other operand must be a register. This ISA allows variable-length operations, operating on data with lengths of 1, 2, or 4 bytes.

The 8086 through the 80486 are single-stage pipeline architectures. The architects reasoned that if one pipeline was good, two would be better. The Pentium had two parallel five-stage pipelines, called the U pipe and the V pipe, to execute instructions. Stages for these pipelines include Prefetch, Instruction Decode, Address Generation, Execute, and Write Back. To be effective, these pipelines must be kept filled, which requires instructions that can be issued in parallel. It is the compiler's responsibility to make sure this parallelism happens. The Pentium II increased the number of stages to 12, including Prefetch, Length Decode, Instruction Decode, Rename/Resource Allocation, UOP Scheduling/Dispatch, Execution, Write Back, and Retirement. Most of the new stages were added to address Intel's MMX technology, an extension to the architecture that handles multimedia data. The Pentium III increased the stages to 14, and the Pentium IV to 24. Additional stages (beyond those introduced in this chapter) included stages for determining the length of the instruction, stages for creating microoperations, and stages to "commit" the instruction (make sure it executes and the results become permanent). The Itanium contains only a 10-stage instruction pipeline.

Intel processors allow for the basic addressing modes introduced in this chapter, in addition to many combinations of those modes. The 8086 provided 17 different ways to access memory, most of which were variants of the basic modes. Intel's more current Pentium architectures include the same addressing modes as their predecessors, but also introduce new modes, mostly to help with maintaining backward compatibility. The IA-64 is surprisingly lacking in memory-addressing modes. It has only one: register indirect (with optional post-increment). This seems unusually limiting but follows the RISC philosophy. Addresses are calcu-

lated and stored in general-purpose registers. The more complex addressing modes require specialized hardware; by limiting the number of addressing modes, the IA-64 architecture minimizes the need for this specialized hardware.

5.6.2 MIPS

The MIPS architecture (which originally stood for "Microprocessor without Interlocked Pipeline Stages") is a little endian, word-addressable, three-address, fixed-length ISA. This is a load and store architecture, which means only the load and store instructions can access memory. All other instructions must use registers for operands, which implies that this ISA needs a large register set. MIPS is also limited to fixed-length operations (those that operate on data with the same number of bytes).

Some MIPS processors (such as the R2000 and R3000) have five-stage pipelines. The R4000 and R4400 have 8-stage superpipelines. The R10000 is quite interesting in that the number of stages in the pipeline depends on the functional unit through which the instruction must pass: there are five stages for integer instructions, six for load/store instructions, and seven for floating-point instructions. Both the MIPS 5000 and 10000 are superscalar.

MIPS has a straightforward ISA with five basic types of instructions: simple arithmetic (add, XOR, NAND, shift), data movement (load, store, move), control (branch, jump), multi-cycle (multiply, divide), and miscellaneous instructions (save PC, save register on condition). MIPS programmers can use immediate, register, direct, indirect register, base, and indexed addressing modes. However, the ISA itself provides for only one (base addressing). The remaining modes are provided by the assembler. The MIPS64 has two additional addressing modes for use in embedded systems optimizations.

The MIPS instructions in Chapter 4 had up to four fields: an opcode, two operand addresses, and one result address. Essentially three instruction formats are available: the I type (immediate), the R type (register), and the J type (jump).

R type instructions have a 6-bit opcode, a 5-bit source register, a 5-bit target register, a 5-bit shift amount, and a 6-bit function. I type instructions have a 6-bit operand, a 5-bit source register, a 5-bit target register or branch condition, and a 16-bit immediate branch displacement or address displacement. J type instructions have a 6-bit opcode and a 26-bit target address.

5.6.3 Java Virtual Machine

Java, a language that is becoming quite popular, is very interesting in that it is platform independent. This means that if you compile code on one architecture (say a Pentium) and you wish to run your program on a different architecture (say a Sun workstation), you can do so without modifying or even recompiling your code.

The Java compiler makes no assumptions about the underlying architecture of the machine on which the program will run, such as the number of registers, memory size, or I/O ports, when you first compile your code. After compilation, [illegible] will need a **Java Virtual Machine**

is a software emulation of a real machine.) The JVM is essentially a "wrapper" that goes around the hardware architecture, and is very platform dependent. The JVM for a Pentium is different from the JVM for a Sun workstation, which is different from the JVM for a Macintosh, and so on. But once the JVM exists on a particular architecture, that JVM can execute any Java program compiled on any ISA platform. It is the JVM's responsibility to load, check, find, and execute bytecodes at run time. The JVM, although virtual, is a nice example of a well-designed ISA.

The JVM for a particular architecture is written in that architecture's native instruction set. It acts as an interpreter, taking Java bytecodes and interpreting them into explicit underlying machine instructions. **Bytecodes** are produced when a Java program is compiled. These bytecodes then become input for the JVM. The JVM can be compared to a giant switch (or case) statement, analyzing one bytecode instruction at a time. Each bytecode instruction causes a jump to a specific block of code, which implements the given bytecode instruction.

This differs significantly from other high-level languages with which you may be familiar. For example, when you compile a C++ program, the object code produced is for that particular architecture. (Compiling a C++ program results in an assembly language program that is translated to machine code.) If you want to run your C++ program on a different platform, you must recompile it for the target architecture. Compiled languages are translated into runnable files of the binary machine code by the compiler. Once this code has been generated, it can be run only on the target architecture. Compiled languages typically exhibit excellent performance and give very good access to the operating system. Examples of compiled languages include C, C++, Ada, FORTRAN, and COBOL.

Some languages, such as LISP, PhP, Perl, Python, Tcl, and most BASIC languages, are interpreted. The source must be reinterpreted each time the program is run. The trade-off for the platform independence of interpreted languages is slower performance—usually by a factor of 100 times. (We will have more to say on this topic in Chapter 8.)

Languages that are a bit of both (compiled and interpreted) exist as well. These are often called **P-code languages**. The source code written in these languages is compiled into an intermediate form, called P-code, and the P-code is then interpreted. P-code languages typically execute from 5 to 10 times more slowly than compiled languages. Python, Perl, and Java are actually P-code languages, even though they are typically referred to as interpreted languages.

Figure 5.6 presents an overview of the Java programming environment.

Perhaps more interesting than Java's platform independence, particularly in relationship to the topics covered in this chapter, is the fact that Java's bytecode is a stack-based language, partially composed of zero address instructions. Each instruction consists of a one-byte opcode followed by zero or more operands. The opcode itself indicates whether it is followed by operands and the form the operands (if any) take. Many of these instructions require zero operands.

Java uses two's complement to represent signed integers but does not allow for unsigned integers. Characters are coded using 16-bit Unicode. Java has four registers, which provide access to five different main memory regions. All references to

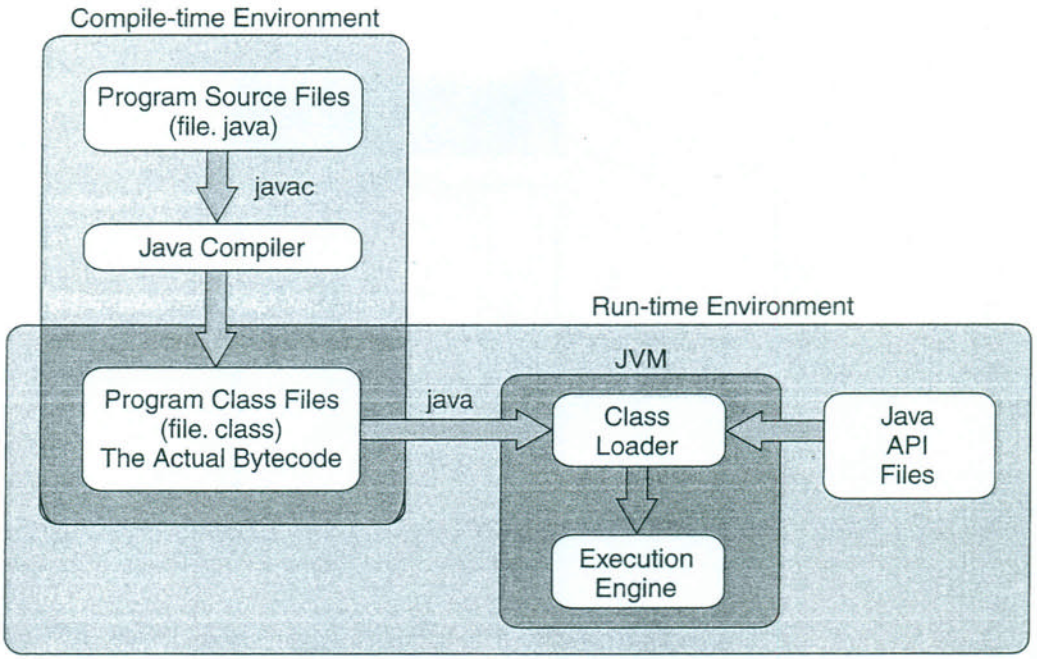

FIGURE 5.6 The Java Programming Environment

memory are based on offsets from these registers; pointers or absolute memory addresses are never used. Because the JVM is a stack machine, no general registers are provided. This lack of general registers is detrimental to performance, as more memory references are generated. We are trading performance for portability.

Let's take a look at a short Java program and its corresponding bytecode. Example 5.2 shows a Java program that finds the maximum of two numbers.

EXAMPLE 5.2 Here is a Java program to find the maximum of two numbers.

```
public class Maximum {

  public static void main (String[] Args)
  { int X,Y,Z;
    X = Integer.parseInt(Args[0]);
    Y = Integer.parseInt(Args[1]);
    Z = Max(X,Y);
    System.out.println(Z);
  }

  public static int Max (int A, int B)
  { int C;
    if (A > B)C = A;
    else C = B;
    return C;
  }
}
```

After we compile this program (using `javac`), we can disassemble it to examine the bytecode, by issuing the following command:

```
javap -c Maximum
```

You should see the following:

```
Compiled from Maximum.java
public class Maximum extends java.lang.Object {
    public Maximum();
    public static void main(java.lang.String[]);
    public static int Max(int, int);
}

Method Maximum()
   0 aload_0
   1 invokespecial #1 <Method java.lang.Object()>
   4 return
Method void main(java.lang.String[])
   0 aload_0
   1 iconst_0
   2 aaload
   3 invokestatic #2 <Method int parseInt(java.lang.String)>
   6 istore_1
   7 aload_0
   8 iconst_1
   9 aaload
  10 invokestatic #2 <Method int parseInt(java.lang.String)>
  13 istore_2
  14 iload_1
  15 iload_2
  16 invokestatic #3 <Method int Max(int, int)>
  19 istore_3
  20 getstatic #4 <Field java.io.PrintStream out>
  23 iload_3
  24 invokevirtual #5 <Method void println(int)>
  27 return

Method int Max(int, int)
   0 iload_0
   1 iload_1
   2 if_icmple 10
   5 iload_0
   6 istore_2
   7 goto 12
  10 iload_1
  11 istore_2
  12 iload_2
  13 ireturn
```