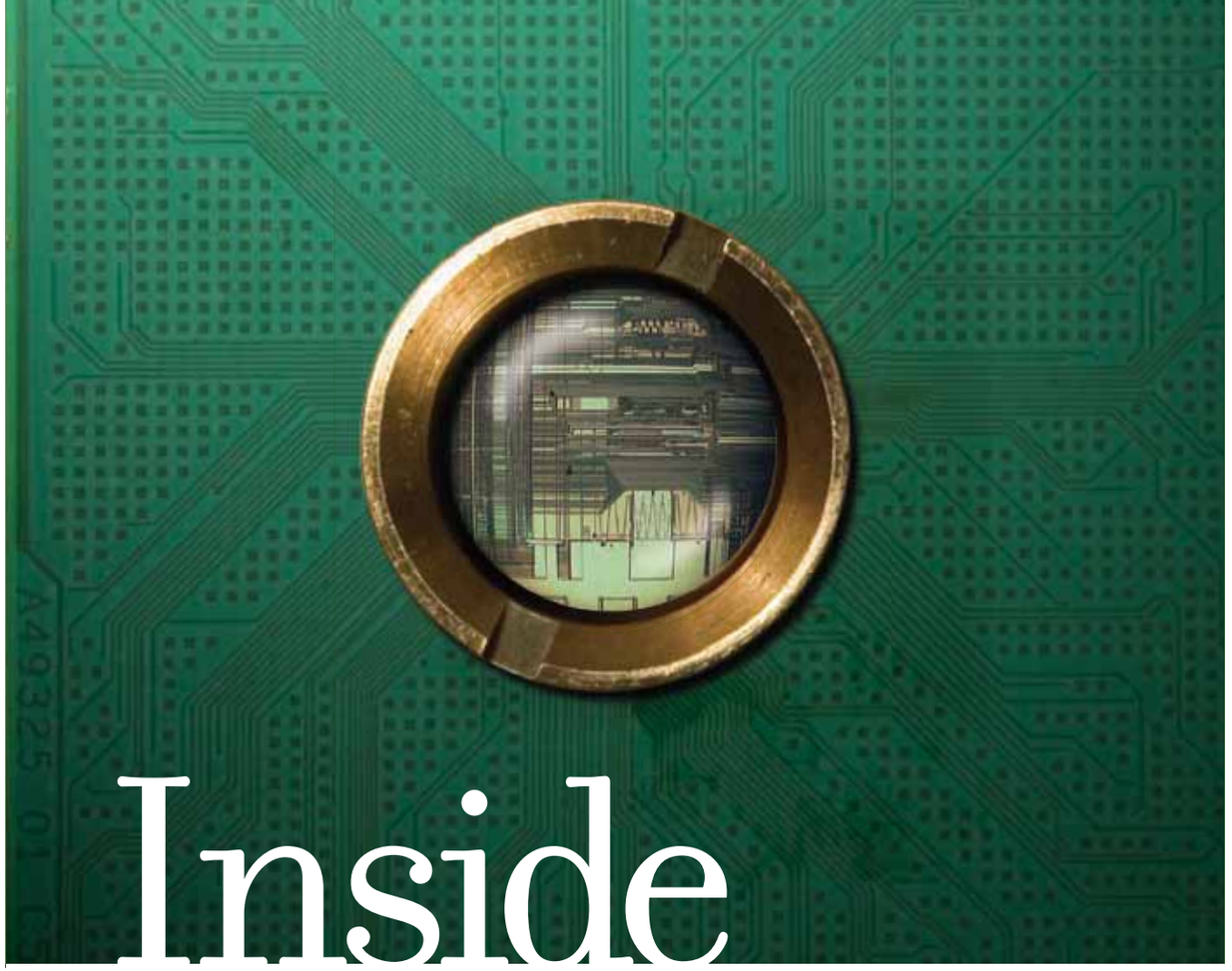

Inside the Machine

An Illustrated Introduction to Microprocessors and Computer Architecture

Jon Stokes

ars technica library

NO STARCH PRESS

BRIEF CONTENTS

CONTENTS IN DETAIL

PREFACE

"The purpose of computing is insight, not numbers."
—Richard W. Hamming (1915–1998)

When mathematician and computing pioneer Richard Hamming penned this maxim in 1962, the era of digital computing was still very much in its infancy. There were only about 10,000 computers in existence worldwide; each one was large and expensive, and each required teams of engineers for maintenance and operation. Getting results out of these mammoth machines was a matter of laboriously inputting long strings of numbers, waiting for the machine to perform its calculations, and then interpreting the resulting mass of ones and zeros. This tedious and painstaking process prompted Hamming to remind his colleagues that the reams of numbers they worked with on a daily basis were only a means to a much higher and often non-numerical end: keener insight into the world around them.

In today's post-Internet age, hundreds of millions of people regularly use computers not just to gain insight, but to book airline tickets, to play poker, to assemble photo albums, to find companionship, and to do every other sort of human activity from the mundane to the sublime. In stark contrast to the

way things were 40 years ago, the experience of using a computer to do math on large sets of numbers is fairly foreign to many users, who spend only a very small fraction of their computer time explicitly performing arithmetic operations. In popular operating systems from Microsoft and Apple, a small calculator application is tucked away somewhere in a folder and accessed only infrequently, if at all, by the majority of users. This small, seldom-used calculator application is the perfect metaphor for the modern computer's hidden identity as a shuffler of numbers.

This book is aimed at reintroducing the computer as a calculating device that performs layer upon layer of miraculous sleights of hand in order to hide from the user the rapid flow of numbers inside the machine. The first few chapters introduce basic computing concepts, and subsequent chapters work through a series of more advanced explanations, rooted in real-world hardware, that show how instructions, data, and numerical results move through the computers people use every day. In the end, *Inside the Machine* aims to give the reader an intermediate to advanced knowledge of how a variety of microprocessors function and how they stack up to each other from multiple design and performance perspectives.

Ultimately, I have tried to write the book that I would have wanted to read as an undergraduate computer engineering student: a book that puts the pieces together in a big-picture sort of way, while still containing enough detailed information to offer a firm grasp of the major design principles underlying modern microprocessors. It is my hope that *Inside the Machine*'s blend of exposition, history, and architectural "comparative anatomy" will accomplish that goal.

INTRODUCTION

Inside the Machine is an introduction to computers that is intended to fill the gap that exists between classic but more challenging introductions to computer architecture, like John L. Hennessy's and David A. Patterson's popular textbooks, and the growing mass of works that are simply too basic for motivated non-specialist readers. Readers with some experience using computers and with even the most minimal scripting or programming experience should finish *Inside the Machine* with a thorough and advanced understanding of the high-level organization of modern computers. Should they so choose, such readers would then be well equipped to tackle more advanced works like the aforementioned classics, either on their own or as part of formal curriculum.

The book's comparative approach, described below, introduces new design features by comparing them with earlier features intended to solve the same problem(s). Thus, beginning and intermediate readers are encouraged to read the chapters in order, because each chapter assumes a familiarity with the concepts and processor designs introduced in the chapters prior to it.

More advanced readers who are already familiar with some of the processors covered will find that the individual chapters can stand alone. The book's extensive use of headings and subheadings means that it can also be employed as a general reference for the processors described, though that is not the purpose for which it was designed.

The first four chapters of *Inside the Machine* are dedicated to laying the conceptual groundwork for later chapters' studies of real-world microprocessors. These chapters use a simplified example processor, the DLW, to illustrate basic and intermediate concepts like the instructions/data distinction, assembly language programming, superscalar execution, pipelining, the programming model, machine language, and so on.

The middle portion of the book consists of detailed studies of two popular desktop processor lines: the Pentium line from Intel and the PowerPC line from IBM and Motorola. These chapters walk the reader through the chronological development of each processor line, describing the evolution of the microarchitectures and instruction set architectures under discussion. Along the way, more advanced concepts like speculative execution, vector processing, and instruction set translation are introduced and explored via a discussion of one or more real-world processors.

Throughout the middle part of the book, the overall approach is what might be called "comparative anatomy," in which each new processor's novel features are explained in terms of how they differ from analogous features found in predecessors and/or competitors. The comparative part of the book culminates in Chapters 7 and 8, which consist of detailed comparisons of two starkly different and very important processors: Intel's Pentium 4 and Motorola's MPC7450 (popularly known as the G4e).

After a brief introduction to 64-bit computing and the 64-bit extensions to the popular *x*86 instruction set architecture in Chapter 9, the microarchitecture of the first mass-market 64-bit processor, the IBM PowerPC 970, is treated in Chapter 10. This study of the 970, the majority of which is also directly applicable to IBM's POWER4 mainframe processor, concludes the book's coverage of PowerPC processors.

Chapter 11 covers the organization and functioning of the memory hierarchy found in almost all modern computers.

Inside the Machine's concluding chapter is given over to an in-depth examination of the latest generation of processors from Intel: the Pentium M, Core Duo, and Core 2 Duo. This chapter contains the most detailed discussion of these processors available online or in print, and it includes some new information that has not been publicly released prior to the printing of this book.

4

SUPERSCALAR EXECUTION

Chapters 1 and 2 described the processor as it is visible to the programmer. The register files, the processor status word (PSW), the arithmetic logic unit (ALU), and other parts of the programming model are all there to provide a means for the programmer to manipulate the processor and make it do useful work. In other words, the programming model is essentially a user interface for the CPU.

Much like the graphical user interfaces on modern computer systems, there's a lot more going on under the hood of a microprocessor than the simplicity of the programming model would imply. In Chapter 12, I'll talk about the various ways in which the operating system and processor collaborate to fool the user into thinking that he or she is executing multiple programs at once. There's a similar sort of trickery that goes on beneath the programming model in a modern microprocessor, but it's intended to fool

the programmer into thinking that there's only one thing going on at a time, when really there are multiple things happening simultaneously. Let me explain.

Back in the days when computer designers could fit relatively few transistors on a single piece of silicon, many parts of the programming model actually resided on separate chips attached to a single circuit board. For instance, one chip contained the ALU, another chip contained the control unit, still another chip contained the registers, and so on. Such computers were relatively slow, and the fact that they were made of multiple chips made them expensive. Each chip had its own manufacturing and packaging costs, so the more chips you put on a board, the more expensive the overall system was. (Note that this is still true today. The cost of producing systems and components can be drastically reduced by packing the functionality of multiple chips into a single chip.)

With the advent of the Intel 4004 in 1971, all of that changed. The 4004 was the world's first microprocessor on a chip. Designed to be the brains of a calculator manufactured by a now defunct company named Busicom, the 4004 had 16 four-bit registers, an ALU, and decoding and control logic all packed onto a single, 2,300-transistor chip. The 4004 was quite a feat for its day, and it paved the way for the PC revolution. However, it wasn't until Intel released the 8080 four years later that the world saw the first true general-purpose CPU.

During the decades following the 8080, the number of transistors that could be packed onto a single chip increased at a stunning pace. As CPU designers had more and more transistors to work with when designing new chips, they began to think up novel ways for using those transistors to increase computing performance on application code. One of the first things that occurred to designers was that they could put more than one ALU on a chip and have both ALUs working in parallel to process code faster. Since these designs could do more than one scalar (or *integer*, for our purposes) operation at once, they were called *superscalar* computers. The RS6000 from IBM was released in 1990 and was the world's first commercially available superscalar CPU. Intel followed in 1993 with the Pentium, which, with its two ALUs, brought the *x*86 world into the superscalar era.

For illustrative purposes, I'll now introduce a *two-way superscalar* version of the DLW-1, called the DLW-2 and illustrated in Figure 4-1. The DLW-2 has two ALUs, so it's able to execute two arithmetic instructions in parallel (hence the term *two-way* superscalar). These two ALUs share a single register file, a situation that in terms of our file clerk analogy would correspond to the file clerk sharing his personal filing cabinet with a second file clerk.

As you can probably guess from looking at Figure 4-1, superscalar processing adds a bit of complexity to the DLW-2's design, because it needs new circuitry that enables it to reorder the linear instruction stream so that some of the stream's instructions can execute in parallel. This circuitry has to ensure that it's "safe" to dispatch two instructions in parallel to the two execution units. But before I go on to discuss some reasons why it might not be safe to execute two instructions in parallel, I should define the term I just used—*dispatch*.

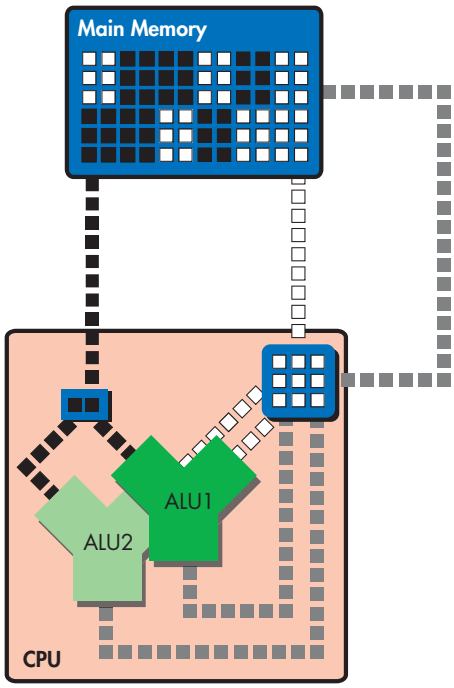

Figure 4-1: The superscalar DLW-2

Notice that in Figure 4-2 I've renamed the second pipeline stage *decode/dispatch*. This is because attached to the latter part of the decode stage is a bit of dispatch circuitry whose job it is to determine whether or not two instructions can be executed in parallel, in other words, on the same clock cycle. If they can be executed in parallel, the dispatch unit sends one instruction to the first integer ALU and one to the second integer ALU. If they can't be dispatched in parallel, the dispatch unit sends them in program order to the first of the two ALUs. There are a few reasons why the dispatcher might decide that two instructions can't be executed in parallel, and we'll cover those in the following sections.

It's important to note that even though the processor has multiple ALUs, the programming model does not change. The programmer still writes to the same interface, even though that interface now represents a fundamentally different type of machine than the processor actually is; the interface represents a sequential execution machine, but the processor is actually a parallel execution machine. So even though the superscalar CPU executes instructions in parallel, the illusion of sequential execution absolutely must be maintained for the sake of the programmer. We'll see some reasons why this is so later on, but for now the important thing to remember is that main memory still sees one sequential code stream, one data stream, and one results stream, even though the code and data streams are carved up inside the computer and pushed through the two ALUs in parallel.

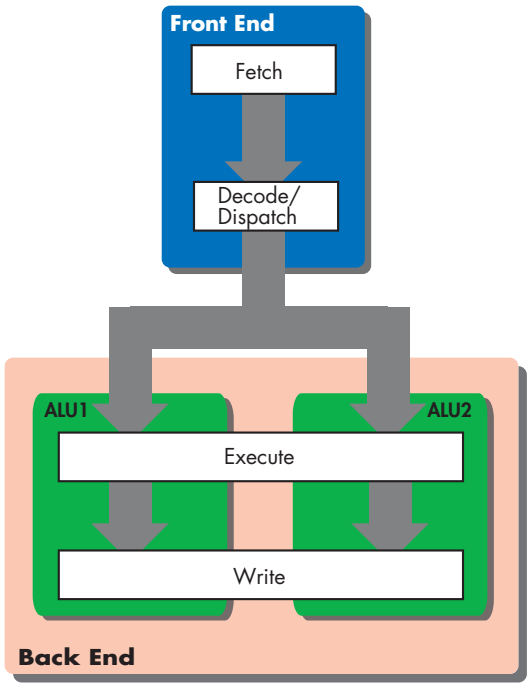

Figure 4-2: The pipeline of the superscalar DLW-2

If the processor is to execute multiple instructions at once, it must be able to fetch and decode multiple instructions at once. A two-way superscalar processor like the DLW-2 can fetch two instructions at once from memory on each clock cycle, and it can also decode and dispatch two instructions each clock cycle. So the DLW-2 fetches instructions from memory in groups of two, starting at the memory address that marks the beginning of the current program's code segment and incrementing the program counter to point four bytes ahead each time a new instruction is fetched. (Remember, the DLW-2's instructions are two bytes wide.)

As you might guess, fetching and decoding two instructions at a time complicates the way the DLW-2 deals with branch instructions. What if the first instruction in a fetched pair happens to be a branch instruction that has the processor jump directly to another part of memory? In this case, the second instruction in the pair has to be discarded. This wastes fetch bandwidth and introduces a bubble into the pipeline. There are other issues relating to superscalar execution and branch instructions, and I'll say more about them in the section on control hazards.

Superscalar Computing and IPC

Superscalar computing allows a microprocessor to increase the number of instructions per clock that it completes beyond one instruction per clock. Recall that one instruction per clock was the maximum theoretical instruction throughput for a pipelined processor, as described in "Instruction Throughput" on page 53. Because a superscalar machine can have multiple instructions

in multiple write stages on each clock cycle, the superscalar machine can complete multiple instructions per cycle. If we adapt Chapter 3's pipeline diagrams to take account of superscalar execution, they look like Figure 4-3.

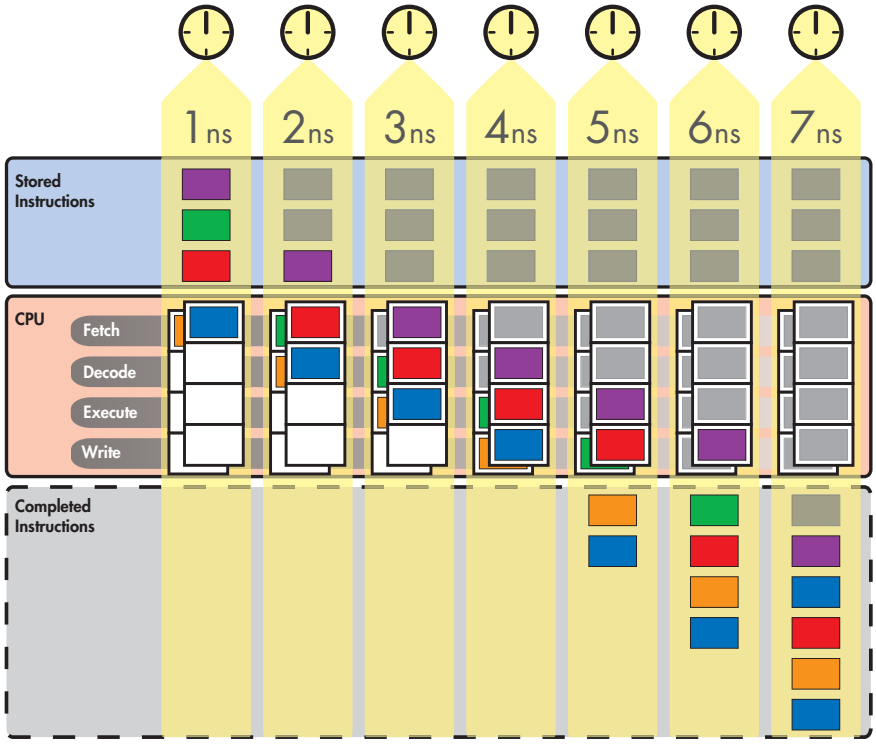

Figure 4-3: Superscalar execution and pipelining combined

In Figure 4-3, two instructions are added to the *Completed Instructions* box on each cycle once the pipeline is full. The more ALU pipelines that a processor has operating in parallel, the more instructions it can add to that box on each cycle. Thus superscalar computing allows you to increase a processor's IPC by adding more hardware. There are some practical limits to how many instructions can be executed in parallel, and we'll discuss those later.

Expanding Superscalar Processing with Execution Units

Most modern processors do more with superscalar execution than just adding a second ALU. Rather, they distribute the work of handling different types of instructions among different types of execution units. An *execution unit* is a block of circuitry in the processor's back end that executes a certain category of instruction. For instance, you've already met the arithmetic logic unit (ALU), an execution unit that performs arithmetic and logical operations on integers. In this section we'll take a closer look at the ALU, and you'll learn about some other types of execution units for non-integer arithmetic operations, memory accesses, and branch instructions.

Basic Number Formats and Computer Arithmetic

The kinds of numbers on which modern microprocessors operate can be divided into two main types: integers (aka fixed-point numbers) and floating-point numbers. *Integers* are simply whole numbers of the type with which you first learn to count in grade school. An integer can be positive, negative, or zero, but it cannot, of course, be a fraction. Integers are also called *fixed-point numbers* because an integer's decimal point does not move. Examples of integers are 1, 0, 500, 27, and 42. Arithmetic and logical operations involving integers are among the simplest and fastest operations that a microprocessor performs. Applications like compilers, databases, and word processors make heavy use of integer operations, because the numbers they deal with are usually whole numbers.

A *floating-point number* is a decimal number that represents a fraction. Examples of floating-point numbers are 56.5, 901.688, and 41.9999. As you can see from these three numbers, the decimal point "floats" around and isn't fixed in once place, hence the name. The number of places behind the decimal point determines a floating-point number's accuracy, so floating-point numbers are often *approximations* of fractional values. Arithmetic and logical operations performed on floating-point numbers are more complex and, hence, slower than their integer counterparts. Because floating-point numbers are approximations of fractional values, and the real world is kind of approximate and fractional, floating-point arithmetic is commonly found in real world–oriented applications like simulations, games, and signal-processing applications.

Both integer and floating-point numbers can themselves be divided into one of two types: scalars and vectors. *Scalars* are values that have only one numerical component, and they're best understood in contrast with *vectors*. Briefly, a vector is a multicomponent value, most often seen as an ordered sequence or array of numbers. (Vectors are covered in detail in "The Vector Execution Units" on page 168.) Here are some examples of different types of vectors and scalars:

	Integer	Floating-Point
Scalar	14 –500 37	1.01 15.234 –0.0023
Vector	{5, –7, –9, 8} {1,003, 42, 97, 86, 97} {234, 7, 6, 1, 3, 10, 11}	{0.99, –1.1, 3.31} {50.01, 0.002, –1.4, 1.4} {5.6, 22.3, 44.444, 76.01, 9.9}

Returning to the code/data distinction, we can say that the data stream consists of four types of numbers: scalar integers, scalar floating-point numbers, vector integers, and vector floating-point numbers. (Note that even memory addresses fall into one of these four categories—scalar integers.) The code stream, then, consists of instructions that operate on all four types of numbers.

The kinds of operations that can be performed on the four types of numbers fall into two main categories: arithmetic operations and logical operations. When I first introduced arithmetic operations in Chapter 1, I lumped them together with logical operations for the sake of convenience. At this point, though, it's useful to distinguish the two types of operations from one another:

- Arithmetic operations are operations like addition, subtraction, multiplication, and division, all of which can be performed on any type of number.
- Logical operations are Boolean operations like AND, OR, NOT, and XOR, along with bit shifts and rotates. Such operations are performed on scalar and vector integers, as well as on the contents of special-purpose registers like the processor status word (PSW).

The types of operations performed on these types of numbers can be broken down as illustrated in Figure 4-4.

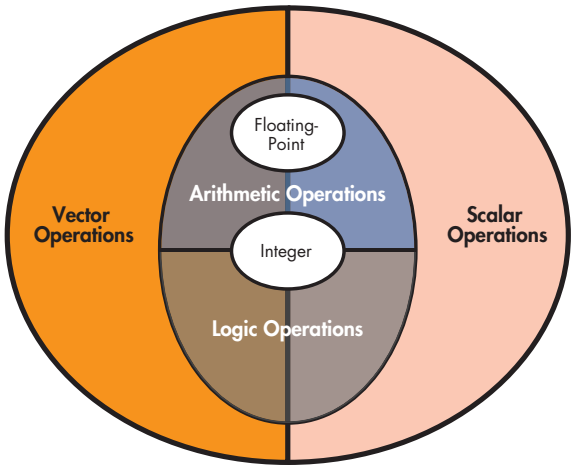

Figure 4-4: Number formats and operation types

As you make your way through the rest of the book, you may want to refer back to this section occasionally. Different microprocessors divide these operations among different execution units in a variety of ways, and things can easily get confusing.

Arithmetic Logic Units

On early microprocessors, as on the DLW-1 and DLW-2, all integer arithmetic and logical operations were handled by the ALU. Floating-point operations were executed by a companion chip, commonly called an *arithmetic coprocessor*, that was attached to the motherboard and designed to work in conjunction with the microprocessor. Eventually, floating-point capabilities were integrated onto the CPU as a separate execution unit alongside the ALU.

Consider the Intel Pentium processor depicted in Figure 4-5, which contains two integer ALUs and a floating-point ALU, along with some other units that we'll describe shortly.

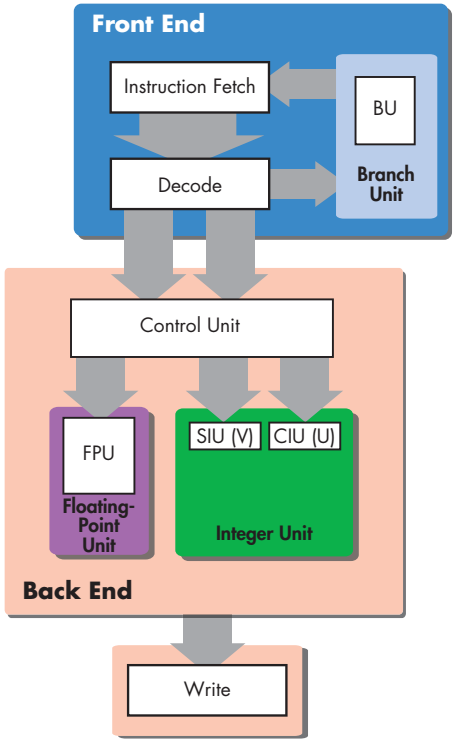

Figure 4-5: The Intel Pentium

This diagram is a variation on Figure 4-2, with the execute stage replaced by labeled white boxes (SIU, CIU, FPU, BU, etc.) that designate the type of execution unit that's modifying the code stream during the execution phase. Notice also that the figure contains a slight shift in terminology that I should clarify before we move on.

Until now, I've been using the term *ALU* as synonymous with *integer execution unit*. After the previous section, however, we know that a microprocessor does arithmetic and logical operations on more than just integer data, so we have to be more precise in our terminology. From now on, *ALU* is a general term for any execution unit that performs arithmetic and logical operations on any type of data. More specific labels will be used to identify the ALUs that handle specific types of instructions and numerical data. For instance, an *integer execution unit (IU)* is an ALU that executes integer arithmetic and logical instructions, a *floating-point execution unit (FPU)* is an ALU that executes floating-point arithmetic and logical instructions, and so on. Figure 4-5 shows that the Pentium has two IUs—a simple integer unit (SIU) and a complex integer unit (CIU)—and a single FPU.

Execution units can be organized logically into functional blocks for ease of reference, so the two integer execution units can be referred

to collectively as the Pentium's *integer unit*. The Pentium's *floating-point unit* consists of only a single FPU, but some processors have more than one FPU; likewise with the load-store unit (LSU). The floating-point unit can consist of two FPUs—FPU1 and FPU2—and the load-store unit can consist of LSU1 and LSU2. In both cases, we'll often refer to "the FPU" or "the LSU" when we mean all of the execution units in that functional block, taken as a group.

Many modern microprocessors also feature vector execution units, which perform arithmetic and logical operations on vectors. I won't describe vector computing in detail here, however, because that discussion belongs in another chapter.

Memory-Access Units

In almost all of the processors that we'll cover in later chapters, you'll see a pair of execution units that execute memory-access instructions: the load-store unit and the branch execution unit. The *load-store unit (LSU)* is responsible for the execution of load and store instructions, as well as for *address generation*. As mentioned in Chapter 1, LSUs have small, stripped-down integer addition hardware that can quickly perform the addition required to compute an address.

The *branch execution unit (BEU)* is responsible for executing conditional and unconditional branch instructions. The BEU of the DLW series reads the processor status word as described in Chapter 1 and decides whether or not to replace the program counter with the branch target. The BEU also often has its own address generation unit for performing quick address calculations as needed. We'll talk more about the branch units of real-world processors later on.

Microarchitecture and the ISA

In the preceding discussion of superscalar execution, I made a number of references to the discrepancy between the linear-execution, single-ALU programming model that the programmer sees and what the superscalar processor's hardware actually does. It's now time to flesh out that distinction between the programming model and the actual hardware by introducing some concepts and vocabulary that will allow us to talk with more precision about the divisions between the apparent and the actual in computer architecture.

Chapter 1 introduced the concept of the programming model as an abstract representation of the microprocessor that exposes to the programmer the microprocessor's functionality. The DLW-1's programming model consisted of a single, integer-only ALU, four general-purpose registers, a program counter, an instruction register, a processor status word, and a control unit. The DLW-1's *instruction set* consisted of a few instructions for working with different parts of the programming model: arithmetic instructions (e.g., `add` and `sub`) for the ALU and general-purpose registers (GPRs), `load` and `store` instructions for manipulating the control unit and filling the GPRs with data,

and branch instructions for checking the PSW and changing the PC. We can call this programmer-centric combination of programming model and instruction set an *instruction set architecture (ISA)*.

The DLW-1's ISA was a straightforward reflection of its hardware, which consisted of a single ALU, four GPRs, a PC, a PSW, and a control unit. In contrast, the successor to the DLW-1, the DLW-2, contained a second ALU that was invisible to the programmer and accessible only to the DLW-2's decode/dispatch logic. The DLW-2's decode/dispatch logic would examine pairs of integer arithmetic instructions to determine if they could safely be executed in parallel (and hence out of sequential program order). If they could, it would send them off to the two integer ALUs to be executed simultaneously. Now, the DLW-2 has the same instruction set architecture as the DLW-1—the instruction set and programming model remain unchanged—but the DLW-2's *hardware implementation* of that ISA is significantly different in that the DLW-2 is superscalar.

A particular processor's hardware implementation of an ISA is generally referred to as that processor's *microarchitecture*. We might call the ISA introduced with the DLW-1 the *DLW ISA*. Each successive iteration of our hypothetical DLW line of computers—the DLW-1 and DLW-2—implements the DLW ISA using a different microarchitecture. The DLW-1 has only one ALU, while the DLW-2 is a two-way superscalar implementation of the DLW-ISA.

Intel's *x*86 hardware followed the same sort of evolution, with each successive generation becoming more complex while the ISA stayed largely unchanged. Regarding the Pentium's inclusion of floating-point hardware, you might be wondering how the programmer was able to use the floating-point hardware (i.e., the FPU plus a floating-point register file) if the original *x*86 ISA didn't include any floating-point operations or specify any floating-point registers. The Pentium's designers had to make the following changes to the ISA to accommodate the new functionality:

- First, they had to modify the programming model by adding an FPU and floating-point–specific registers.
- Second, they had to extend the instruction set by adding a new group of floating-point arithmetic instructions.

These types of *ISA extensions* are fairly common in the computing world. Intel extended the original *x*86 instruction set to include the *x*87 floating-point extensions. The *x*87 included an FPU and a stack-based floating-point register file, but we'll talk in more detail about the *x*87's stack-based architecture in the next chapter. Intel later extended *x*86 again with the introduction of a vector-processing instruction set called *MMX (multimedia extensions)*, and again with the introduction of the *SSE (streaming SIMD extensions)* and SSE2 instruction sets. (*SIMD* stands for *single instruction, multiple data* and is another way of describing vector computing. We'll cover this in more detail in "The Vector Execution Units" on page 168.) Similarly, Apple, Motorola, and IBM added a set of vector extensions to the PowerPC ISA in the form of AltiVec, as the extensions are called by Motorola, or VMX, as they're called by IBM.

A Brief History of the ISA

Back in the early days of computing, computer makers like IBM didn't build a whole line of software-compatible computer systems and aim each system at a different price/performance point. Instead, each of a manufacturer's systems was like each of today's game consoles, at least from a programmer's perspective—programmers wrote directly to the machine's unique hardware, with the result that a program written for one machine would run neither on competing machines nor on other machines from a different product line put out by the manufacturer's own company. Just like a Nintendo 64 will run neither PlayStation games nor older SNES games, programs written for one circa-1960 machine wouldn't run on any machine but that one particular product from that one particular manufacturer. The programming model was different for each machine, and the code was fitted directly to the hardware like a key fits a lock (see Figure 4-6 below).

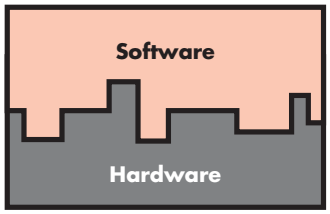

Figure 4-6: Software was custom-fitted to each generation of hardware

The problems this situation posed are obvious. Every time a new machine came out, software developers had to start from scratch. You couldn't reuse programs, and programmers had to learn the intricacies of each new piece of hardware in order to code for it. This cost quite a bit of time and money, making software development a very expensive undertaking. This situation presented computer system designers with the following problem: How do you *expose* (make available) the functionality of a range of related hardware systems in a way that allows software to be easily developed for and ported between those systems? IBM solved this problem in the 1960s with the launch of the IBM System/360, which ushered in the era of modern computer architecture. The System/360 introduced the concept of the ISA as a layer of abstraction—or an interface, if you will—separated from a particular processor's microarchitecture (see Figure 4-7). This means that the information the programmer needed to know to program the machine was abstracted from the actual hardware implementation of that machine. Once the design and specification of the instruction set, or the set of instructions available to a programmer for writing programs, was separated from the low-level details of a particular machine's design, programs written for a particular ISA could run on any machine that implemented that ISA.

Thus the ISA provided a standardized way to expose the features of a system's hardware that allowed manufacturers to innovate and fine-tune that hardware for performance without worrying about breaking the existing software base. You could release a first-generation product with a particular

ISA, and then work on speeding up the implementation of that same ISA for the second-generation product, which would be backward-compatible with the first generation. We take all this for granted now, but before the IBM System/360, binary compatibility between different machines of different generations didn't exist.

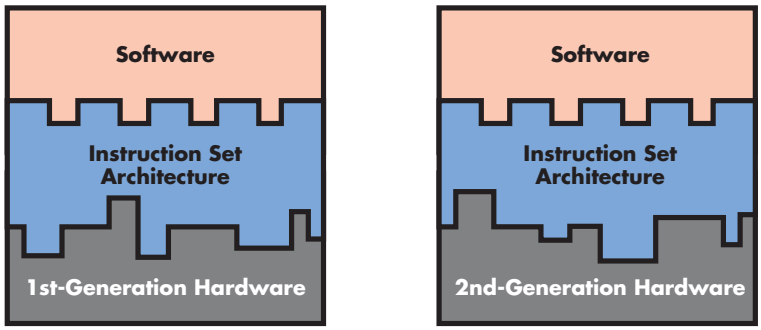

Figure 4-7: The ISA sits between the software and the hardware, providing a consistent interface to the software across hardware generations.

The blue layer in Figure 4-7 simply represents the ISA as an abstract model of a machine for which a programmer writes programs. As mentioned earlier, the technical innovation that made this abstract layer possible was something called the microcode engine. A *microcode engine* is sort of like a CPU within a CPU. It consists of a tiny bit of storage, the *microcode ROM*, which holds *microcode programs*, and an execution unit that executes those programs. The job of each of these microcode programs is to translate a particular instruction into a series of commands that controls the internal parts of the chip. When a System/360 instruction is executed, the microcode unit reads the instruction in, accesses the portion of the microcode ROM where that instruction's corresponding microcode program is located, and then produces a sequence of *machine instructions*, in the processor's internal instruction format, that orchestrates the dance of memory accesses and functional unit activations that actually does the number crunching (or whatever else) the architectural instruction has commanded the machine to do.

By decoding instructions this way, all programs are effectively running in *emulation*. This means that the ISA represents a sort of idealized model, emulated by the underlying hardware, on the basis of which programmers can design applications. This emulation means that between iterations of a product line, a vendor can change the way their CPU executes a program, and all they have to do is rewrite the microcode program each time so the programmer will never have to be aware of the hardware differences because the ISA hasn't changed a bit. Microcode engines still show up in modern CPUs. AMD's Athlon processor uses one for the part of its decoding path that decodes the larger *x*86 instructions, as do Intel's Pentium III and Pentium 4.

The key to understanding Figure 4-7 is that the blue layer represents a layer of abstraction that hides the complexity of the underlying hardware from the programmer. The blue layer is not a hardware layer (that's the gray one) and it's not a software layer (that's the peach one), but it's a *conceptual layer*. Think of it like a user interface that hides the complexity

of an operating system from the user. All the user needs to know to use the machine is how to close windows, launch programs, find files, and so on. The UI (and by this I mean the WIMP conceptual paradigm—windows, icons, menus, pointer—not the software that implements the UI) exposes the machine's power and functionality to the user in a way that he or she can understand and use. And whether that UI appears on a PDA or on a desktop machine, the user still knows how to use it to control the machine.

The main drawback to using microcode to implement an ISA is that the microcode engine was, in the beginning, slower than direct decoding. (Modern microcode engines are about 99 percent as fast as direct execution.) However, the ability to separate ISA design from microarchitectural implementation was so significant for the development of modern computing that the small speed hit incurred was well worth it.

The advent of the *reduced instruction set computing (RISC)* movement in the 1970s saw a couple of changes to the scheme described previously. First and foremost, RISC was all about throwing stuff overboard in the name of speed. So the first thing to go was the microcode engine. Microcode had allowed ISA designers to get elaborate with instruction sets, adding in all sorts of complex and specialized instructions that were intended to make programmers' lives easier but that were in reality rarely used. More instructions meant that you needed more microcode ROM, which in turn meant larger CPU die sizes, higher power consumption, and so on. Since RISC was more about less, the microcode engine got the ax. RISC reduced the number of instructions in the instruction set and reduced the size and complexity of each individual instruction so that this smaller, faster, and more lightweight instruction set could be more easily implemented directly in hardware, without a bulky microcode engine.

While RISC designs went back to the old method of direct execution of instructions, they kept the concept of the ISA intact. Computer architects had by this time learned the immense value of not breaking backward compatibility with old software, and they weren't about to go back to the bad old days of marrying software to a single product. So the ISA stayed, but in a stripped-down, much simplified form that enabled designers to implement directly in hardware the same lightweight ISA over a variety of different hardware types.

NOTE *Because the older, non-RISC ISAs featured richer, more complex instruction sets, they were labeled* complex instruction set computing (CISC) *ISAs in order to distinguish them from the new RISC ISAs. The* x*86 ISA is the most popular example of a CISC ISA, while PowerPC, MIPS, and Arm are all examples of popular RISC ISAs.*

Moving Complexity from Hardware to Software

RISC machines were able to get rid of the microcode engine and still retain the benefits of the ISA by moving complexity from hardware to software. Where the microcode engine made CISC programming easier by providing programmers with a rich variety of complex instructions, RISC programmers depended on high-level languages, like C, and on compilers to ease the burden of writing code for RISC ISAs' restricted instruction sets.

Because a RISC ISA's instruction set is more limited, it's harder to write long programs in assembly language for a RISC processor. (Imagine trying to write a novel while restricting yourself to a fifth grade vocabulary, and you'll get the idea.) A RISC assembly language programmer may have to use many instructions to achieve the same result that a CISC assembly language programmer can get with one or two instructions. The advent of high-level languages (HLLs), like C, and the increasing sophistication of compiler technology combined to effectively eliminate this programmer-unfriendly aspect of RISC computing.

The ISA was and is still the optimal solution to the problem of easily and consistently exposing hardware functionality to programmers so that software can be used across a wide range of machines. The greatest testament to the power and flexibility of the ISA is the longevity and ubiquity of the world's most popular and successful ISA: the *x*86 ISA. Programs written for the Intel 8086, a chip released in 1978, can run with relatively little modification on the latest Pentium 4. However, on a microarchitectural level, the 8086 and the Pentium 4 are as different as the Ford Model T and the Ford Mustang Cobra.

Challenges to Pipelining and Superscalar Design

I noted previously that there are conditions under which two arithmetic instructions cannot be "safely" dispatched in parallel for simultaneous execution by the DLW-2's two ALUs. Such conditions are called *hazards*, and they can all be placed in one of three categories:

- Data hazards
- Structural hazards
- Control hazards

Because pipelining is a form of parallel execution, these three types of hazards can also hinder pipelined execution, causing bubbles to occur in the pipeline. In the following three sections, I'll discuss each of these types of hazards. I won't go into a huge amount of detail about the tricks that computer architects use to eliminate them or alleviate their affects, because we'll discuss those when we look at specific microprocessors in the next few chapters.

Data Hazards

The best way to explain what a *data hazard* is to illustrate one. Consider Program 4-1:

Line #	Code	Comments
1	`add A, B, C`	Add the numbers in registers `A` and `B` and store the result in `C`.
2	`add C, D, D`	Add the numbers in registers `C` and `D` and store the result in `D`.

Program 4-1: A data hazard

Because the second instruction in Program 4-1 depends on the outcome of the first instruction, the two instructions cannot be executed simultaneously. Rather, the `add` in line `1` *must* finish first, so that the result is available in `C` for the `add` in line `2`.

Data hazards are a problem for both superscalar and pipelined execution. If Program 4-1 is run on a superscalar processor with two integer ALUs, the two `add` instructions cannot be executed simultaneously by the two ALUs. Rather, the ALU executing the `add` in line `1` has to finish first, and then the other ALU can execute the `add` in line `2`. Similarly, if Program 4-1 is run on a pipelined processor, the second `add` has to wait until the first `add` completes the write stage before it can enter the execute phase. Thus the dispatch circuitry has to recognize the `add` in line `2`'s dependence on the `add` in line `1`, and keep the `add` in line `2` from entering the execute stage until the `add` in line `1`'s result is available in register `C`.

Most pipelined processors can do a trick called *forwarding* that's aimed at alleviating the effects of this problem. With forwarding, the processor takes the result of the first `add` from the ALU's output port and feeds it directly back into the ALU's input port, bypassing the register-file write stage. Thus the second `add` has to wait for the first `add` to finish only the execute stage, and not the execute and write stages, before it's able to move into the execute stage itself.

Register renaming is a trick that helps overcome data hazards on superscalar machines. Since any given machine's programming model often specifies fewer registers than can be implemented in hardware, a given microprocessor implementation often has more registers than the number specified in the programming model. To get an idea of how this group of additional registers is used, take a look at Figure 4-8.

In Figure 4-8, the DLW-2's programmer thinks that he or she is using a single ALU with four architectural general-purpose registers—`A`, `B`, `C`, and `D`—attached to it, because four registers and one ALU are all that the DLW architecture's programming model specifies. However, the actual superscalar DLW-2 hardware has two ALUs and 16 microarchitectural GPRs implemented in hardware. Thus the DLW-2's register rename logic can map the four architectural registers to the available microarchitectural registers in such a way as to prevent false register name conflicts.

In Figure 4-8, an instruction that's being executed by IU1 might think that it's the only instruction executing and that it's using registers `A`, `B`, and `C`, but it's actually using rename registers 2, 5, and 10. Likewise, a second instruction executing simultaneously with the first instruction but in IU2 might also think that it's the only instruction executing and that it has a monopoly on the register file, but in reality, it's using registers 3, 7, 12, and 16. Once both IUs have finished executing their respective instructions, the DLW-2's write-back logic takes care of transferring the contents of the rename registers back to the four architectural registers in the proper order so that the program's state can be changed.

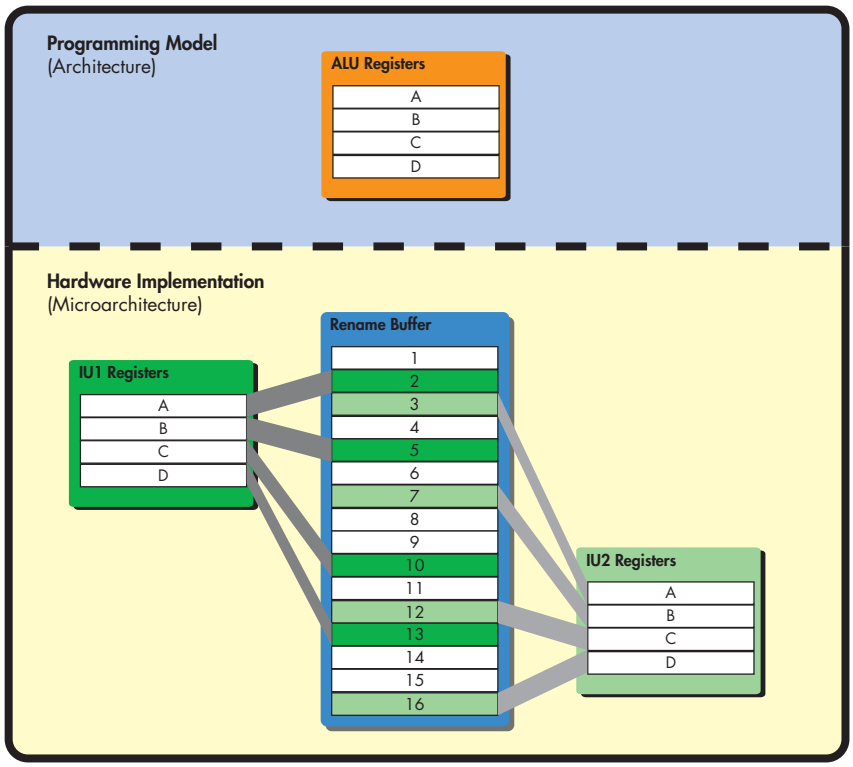

Figure 4-8: Register renaming

Let's take a quick look at a false register name conflict in Program 4-2.

Line #	Code	Comments
1	`add A, B, C`	Add the numbers in registers `A` and `B` and store the result in `C`.
2	`add D, B, A`	Add the numbers in registers `A` and `D` and store the result in `A`.

Program 4-2: A false register name conflict

In Program 4-2, there is no data dependency, and both `add` instructions can take place simultaneously except for one problem: the first `add` reads the contents of `A` for its input, while the second `add` writes a new value into `A` as its output. Therefore, the first `add`'s read absolutely must take place *before* the second `add`'s write. Register renaming solves this register name conflict by allowing the second `add` to write its output to a temporary register; after both `add`s have executed in parallel, the result of the second `add` is written from that temporary register into the architectural register `A` after the first `add` has finished executing and written back its own results.

Structural Hazards

Program 4-3 contains a short code example that shows superscalar execution in action. Assuming the programming model presented for the DLW-2, consider the following snippet of code.

Line #	Code	Comments
15	`add A, B, B`	Add the numbers in registers `A` and `B` and store the result in `B`.
16	`add C, D, D`	Add the numbers in registers `C` and `D` and store the result in `D`.

Program 4-3: A structural hazard

At first glance, there appears to be nothing wrong with Program 4-3. There's no data hazard, because the two instructions don't depend on each other. So it should be possible to execute them in parallel. However, this example presumes that both ALUs share the same group of four registers. But in order for the DLW-2's register file to accommodate multiple ALUs accessing it at once, it needs to be different from the DLW-1's register file in one important way: it must be able to accommodate two simultaneous writes. Otherwise, executing Program 4-3's two instructions in parallel would trigger what's called a *structural hazard*, where the processor doesn't have enough resources to execute both instructions at once.

The Register File

In a superscalar design with multiple ALUs, it would take an enormous number of wires to connect each register directly to each ALU. This problem gets worse as the number of registers and ALUs increases. Hence, in superscalar designs with a large number of registers, a CPU's registers are grouped together into a special unit called a *register file*. This unit is a memory array, much like the array of cells that makes up a computer's main memory, and it's accessed through a special interface that allows the ALU to read from or write to specific registers. This interface consists of a data bus and two types of ports: the *read ports* and the *write ports*. In order to read a value from a single register in the register file, the ALU accesses the register file's read port and requests that the data from a specific register be placed on the special internal data bus that the register file shares with the ALU. Likewise, writing to the register file is done through the file's write port.

A single read port allows the ALU to access a single register at a time, so in order for an ALU to read from two registers simultaneously (as in the case of a three-operand `add` instruction), the register file must have two read ports. Likewise, a write port allows the ALU to write to only one register at a time, so a single ALU needs a single write port in order to be able to write the results of an operation back to a register. Therefore, the register file needs two read ports and one write port for each ALU. So for the two-ALU superscalar design, the register file needs a total of four read ports and two write ports.

It so happens that the amount of die space that the register file takes up increases approximately with the square of the number of ports, so there is a practical limit on the number of ports that a given register file can support. This is one of the reasons why modern CPUs use separate register files to store integer, floating-point, and vector numbers. Since each type of math (integer, floating-point, and vector) uses a different type of execution unit, attaching multiple integer, floating-point, and vector execution units to a single register file would result in quite a large file.

There's also another reason for using multiple register files to accommodate different types of execution units. As the size of the register file increases, so does the amount of time it takes to access it. You might recall from "The File-Clerk Model Revisited and Expanded" on page 9 that we assume that register reads and writes happen instantaneously. If a register file gets too large and the register file access latency gets too high, this can slow down register accesses to the point where such access takes up a noticeable amount of time. So instead of using one massive register file for each type of numerical data, computer architects use two or three register files connected to a few different types of execution units.

Incidentally, if you'll recall "Opcodes and Machine Language" on page 19, the DLW-1 used a series of binary numbers to designate which of the four registers an instruction was accessing. Well, in the case of a register file read, these numbers are fed into the register file's interface in order to specify which of the registers should place its data on the data bus. Taking our two-bit register designations as an example, a port on our four-register file would have two lines that would be held at either high or low voltages (depending on whether the bit placed on each line was a 1 or a 0), and these lines would tell the file which of its registers should have its data placed on the data bus.

Control Hazards

Control hazards, also known as *branch hazards*, are hazards that arise when the processor arrives at a conditional branch and has to decide which instruction to fetch next. In more primitive processors, the pipeline stalls while the branch condition is evaluated and the branch target is calculated. This stall inserts a few cycles of bubbles into the pipeline, depending on how long it takes the processor to identify and locate the branch target instruction.

Modern processors use a technique called *branch prediction* to get around these branch-related stalls. We'll discuss branch prediction in more detail in the next chapter.

Another potential problem associated with branches lies in the fact that once the branch condition is evaluated and the address of the next instruction is loaded into the program counter, it then takes a number of cycles to actually fetch the next instruction from storage. This *instruction load latency* is added to the branch condition evaluation latency discussed earlier in this section. Depending on where the next instruction is located—such as in a nearby cache, in main memory, or on a hard disk—it can take anywhere from a few cycles to thousands of cycles to fetch the instruction. The cycles that the processor spends waiting on that instruction to show up are dead, wasted cycles that show up as bubbles in the processor's pipeline and kill performance. Computer architects use *instruction caching* to alleviate the effects of load latency, and we'll talk more about this technique in the next chapter.