1

BASIC COMPUTING CONCEPTS

Modern computers come in all shapes and sizes, and they aid us in a million different types of tasks ranging from the serious, like air traffic control and cancer research, to the not-so-serious, like computer gaming and photograph retouching. But as diverse as computers are in their outward forms and in the uses to which they're put, they're all amazingly similar in basic function. All of them rely on a limited repertoire of technologies that enable them do the myriad kinds of miracles we've come to expect from them.

At the heart of the modern computer is the *microprocessor*—also commonly called the *central processing unit (CPU)*—a tiny, square sliver of silicon that's etched with a microscopic network of gates and channels through which electricity flows. This network of gates (*transistors*) and channels (*wires* or *lines*) is a very small version of the kind of circuitry that we've all seen when cracking open a television remote or an old radio. In short, the microprocessor isn't just the "heart" of a modern computer—it's a computer in and of itself. Once you understand how this tiny computer works, you'll have a thorough grasp of the fundamental concepts that underlie all of modern computing, from the aforementioned air traffic control system to the silicon brain that controls the brakes on a luxury car.

This chapter will introduce you to the microprocessor, and you'll begin to get a feel for just how straightforward computers really are. You need master only a few fundamental concepts before you explore the microprocessor technologies detailed in the later chapters of this book.

To that end, this chapter builds the general conceptual framework on which I'll hang the technical details covered in the rest of the book. Both newcomers to the study of computer architecture and more advanced readers are encouraged to read this chapter all the way through, because its abstractions and generalizations furnish the large conceptual "boxes" in which I'll later place the specifics of particular architectures.

The Calculator Model of Computing

Figure 1-1 is an abstract graphical representation of what a computer does. In a nutshell, a computer takes a stream of instructions (code) and a stream of data as input, and it produces a stream of results as output. For the purposes of our initial discussion, we can generalize by saying that the *code stream* consists of different types of arithmetic operations and the *data stream* consists of the data on which those operations operate. The *results stream*, then, is made up of the results of these operations. You could also say that the results stream begins to flow when the operators in the code stream are carried out on the operands in the data stream.

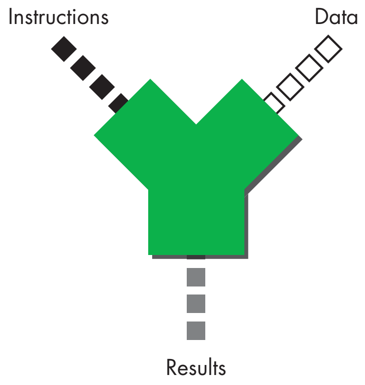


Figure 1-1: A simple representation of a general-purpose computer

NOTE *Figure 1-1 is my own variation on the traditional way of representing a processor's* arithmetic logic unit (ALU)*, which is the part of the processor that does the addition, subtraction, and so on, of numbers. However, instead of showing two operands entering the top ports and a result exiting the bottom port (as is the custom in the literature), I've depicted code and data streams entering the top ports and a results stream leaving the bottom port.*

To illustrate this point, imagine that one of those little black boxes in the code stream of Figure 1-1 is an addition operator (a + sign) and that two of the white data boxes contain two integers to be added together, as shown in Figure 1-2.

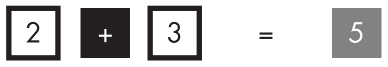


Figure 1-2: Instructions are combined with data to produce results

You might think of these black-and-white boxes as the keys on a calculator—with the white keys being numbers and the black keys being operators—the gray boxes are the results that appear on the calculator's screen. Thus the two input streams (the code stream and the data stream) represent sequences of key presses (arithmetic operator keys and number keys), while the output stream represents the resulting sequence of numbers displayed on the calculator's screen.

The kind of simple calculation described above represents the sort of thing that we intuitively think computers do: like a pocket calculator, the computer takes numbers and arithmetic operators (such as +, –, ÷, ×, etc.) as input, performs the requested operation, and then displays the results. These results might be in the form of pixel values that make up a rendered scene in a computer game, or they might be dollar values in a financial spreadsheet.

The File-Clerk Model of Computing

The "calculator" model of computing, while useful in many respects, isn't the only or even the best way to think about what computers do. As an alternative, consider the following definition of a computer:

> A *computer* is a device that shuffles numbers around from place to place, reading, writing, erasing, and rewriting different numbers in different locations according to a set of inputs, a fixed set of rules for processing those inputs, and the prior history of all the inputs that the computer has seen since it was last reset, until a predefined set of criteria are met that cause the computer to halt.

We might, after Richard Feynman, call this idea of a computer as a reader, writer, and modifier of numbers the "file-clerk" model of computing (as opposed to the aforementioned calculator model). In the file-clerk model, the computer accesses a large (theoretically infinite) store of sequentially arranged numbers for the purpose of altering that store to achieve a desired result. Once this desired result is achieved, the computer halts so that the now-modified store of numbers can be read and interpreted by humans.

The file-clerk model of computing might not initially strike you as all that useful, but as this chapter progresses, you'll begin to understand how important it is. This way of looking at computers is powerful because it emphasizes the end product of computation rather than the computation itself. After all, the purpose of computers isn't just to compute in the abstract, but to produce usable results from a given data set.

NOTE *Those who've studied computer science will recognize in the preceding description the beginnings of a discussion of a Turing machine. The Turing machine is, however, too abstract for our purposes here, so I won't actually describe one. The description that I develop here sticks closer to the classic Reduced Instruction Set Computing (RISC) load-store model, where the computer is "fixed" along with the storage. The Turing model of a computer as a movable read-write head (with a state table) traversing a linear "tape" is too far from real-life hardware organization to be anything but confusing in this discussion.*

In other words, what matters in computing is not that you did some math, but that you started with a body of numbers, applied a sequence of operations to it, and got a body of results. Those results could, again, represent pixel values for a rendered scene or an environmental snapshot in a weather simulation. Indeed, the idea that a computer is a device that transforms one set of numbers into another should be intuitively obvious to anyone who has ever used a Photoshop filter. Once we understand computers not in terms of the math they do, but in terms of the numbers they move and modify, we can begin to get a fuller picture of how they operate.

In a nutshell, a computer is a device that reads, modifies, and writes sequences of numbers. These three functions—read, modify, and write—are the three most fundamental functions that a computer performs, and all of the machine's components are designed to aid in carrying them out. This read-modify-write sequence is actually inherent in the three central bullet points of our initial file-clerk definition of a computer. Here is the sequence mapped explicitly onto the file-clerk definition:

> A computer is a device that shuffles numbers around from place to place, reading, writing, erasing, and rewriting different numbers in different locations according to a set of inputs [*read*], a fixed set of rules for processing those inputs [*modify*], and the prior history of all the inputs that the computer has seen since it was last reset [*write*], until a predefined set of criteria are met that cause the computer to halt.

That sums up what a computer does. And, in fact, that's *all* a computer does. Whether you're playing a game or listening to music, everything that's going on under the computer's hood fits into this model.

NOTE *All of this is fairly simple so far, and I've even been a bit repetitive with the explanations to drive home the basic read-modify-write structure of all computer operations. It's important to grasp this structure in its simplicity, because as we increase our computing model's level of complexity, we'll see this structure repeated at every level.*

The Stored-Program Computer

All computers consist of at least three fundamental types of structures needed to carry out the read-modify-write sequence:

Storage

To say that a computer "reads" and "writes" numbers implies that there is at least one number-holding structure that it reads from and writes to. All computers have a place to put numbers—a storage area that can be read from and written to.

Arithmetic logic unit (ALU)

Similarly, to say that a computer "modifies" numbers implies that the computer contains a device for performing operations on numbers. This device is the ALU, and it's the part of the computer that performs arithmetic operations (addition, subtraction, and so on), on numbers from the storage area. First, numbers are read from storage into the ALU's data input port. Once inside the ALU, they're modified by means of an arithmetic calculation, and then they're written back to storage via the ALU's output port.

The ALU is actually the green, three-port device at the center of Figure 1-1. Note that ALUs aren't normally understood as having a code input port along with their data input port and results output port. They do, of course, have command input lines that let the computer specify which operation the ALU is to carry out on the data arriving at its data input port, so while the depiction of a code input port on the ALU in Figure 1-1 is unique, it is not misleading.

Bus

In order to move numbers between the ALU and storage, some means of transmitting numbers is required. Thus, the ALU reads from and writes to the data storage area by means of the *data bus*, which is a network of transmission lines for shuttling numbers around inside the computer. Instructions travel into the ALU via the *instruction bus*, but we won't cover how instructions arrive at the ALU until Chapter 2. For now, the data bus is the only bus that concerns us.

The code stream in Figure 1-1 flows into the ALU in the form of a sequence of arithmetic instructions (add, subtract, multiply, and so on). The operands for these instructions make up the data stream, which flows over the data bus from the storage area into the ALU. As the ALU carries out operations on the incoming operands, the results stream flows out of the ALU and back into the storage area via the data bus. This process continues until the code stream stops coming into the ALU. Figure 1-3 expands on Figure 1-1 and shows the storage area.

The data enters the ALU from a special storage area, but where does the code stream come from? One might imagine that it comes from the keypad of some person standing at the computer and entering a sequence of instructions, each of which is then transmitted to the code input port of the ALU, or perhaps that the code stream is a prerecorded list of instructions that is fed into the ALU, one instruction at a time, by some manual or automated mechanism. Figure 1-3 depicts the code stream as a prerecorded list of instructions that is stored in a special storage area just like the data stream, and modern computers do store the code stream in just such a manner.

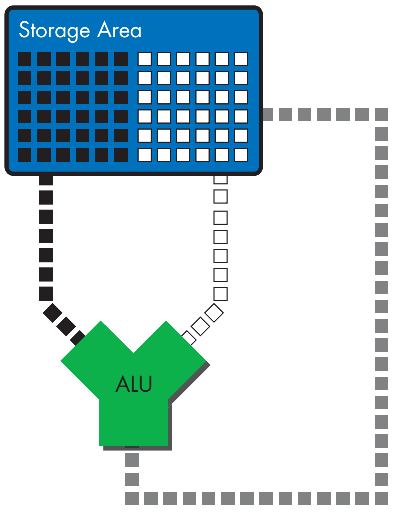


Figure 1-3: A simple computer, with an ALU and a region for storing instructions and data

NOTE *More advanced readers might notice that in Figure 1-3 (and in Figure 1-4 later) I've separated the code and data in main memory after the manner of a Harvard architecture level-one cache. In reality, blocks of code and data are mixed together in main memory, but for now I've chosen to illustrate them as logically separated.*

The modern computer's ability to store and reuse prerecorded sequences of commands makes it fundamentally different from the simpler calculating machines that preceded it. Prior to the invention of the first *stored-program computer*,[1] all computing devices, from the abacus to the earliest electronic computing machines, had to be manipulated by an operator or group of operators who manually entered a particular sequence of commands each time they wanted to make a particular calculation. In contrast, modern computers store and reuse such command sequences, and as such they have a level of flexibility and usefulness that sets them apart from everything that has come before. In the rest of this chapter, you'll get a first-hand look at the many ways that the stored-program concept affects the design and capabilities of the modern computer.

Refining the File-Clerk Model

Let's take a closer look at the relationship between the code, data, and results streams by means of a quick example. In this example, the code stream consists of a single instruction, an `add`, which tells the ALU to add two numbers together.

[1] In 1944 J. Presper Eckert, John Mauchly, and John von Neumann proposed the first stored-program computer, the EDVAC (Electronic Discreet Variable Automatic Computer), and in 1949 such a machine, the EDSAC, was built by Maurice Wilkes of Cambridge University.

The `add` instruction travels from code storage to the ALU. For now, let's not concern ourselves with how the instruction gets from code storage to the ALU; let's just assume that it shows up at the ALU's code input port announcing that there is an addition to be carried out immediately. The ALU goes through the following sequence of steps:

1. Obtain the two numbers to be added (the input operands) from data storage.
2. Add the numbers.
3. Place the results back into data storage.

The preceding example probably sounds simple, but it conveys the basic manner in which computers—*all* computers—operate. Computers are fed a sequence of instructions one by one, and in order to execute them, the computer must first obtain the necessary data, then perform the calculation specified by the instruction, and finally write the result into a place where the end user can find it. Those three steps are carried out billions of times per second on a modern CPU, again and again and again. It's only because the computer executes these steps so rapidly that it's able to present the illusion that something much more conceptually complex is going on.

To return to our file-clerk analogy, a computer is like a file clerk who sits at his desk all day waiting for messages from his boss. Eventually, the boss sends him a message telling him to perform a calculation on a pair of numbers. The message tells him which calculation to perform, and where in his personal filing cabinet the necessary numbers are located. So the clerk first retrieves the numbers from his filing cabinet, then performs the calculation, and finally places the results back into the filing cabinet. It's a boring, mindless, repetitive task that's repeated endlessly, day in and day out, which is precisely why we've invented a machine that can do it efficiently and not complain.

The Register File

Since numbers must first be fetched from storage before they can be added, we want our data storage space to be as fast as possible so that the operation can be carried out quickly. Since the ALU is the part of the processor that does the actual addition, we'd like to place the data storage as close as possible to the ALU so it can read the operands almost instantaneously. However, practical considerations, such as a CPU's limited surface area, constrain the size of the storage area that we can stick next to the ALU. This means that in real life, most computers have a relatively small number of very fast data storage locations attached to the ALU. These storage locations are called *registers*, and the first *x*86 computers only had eight of them to work with. These registers, which are arrayed in a storage structure called a *register file*, store only a small subset of the data that the code stream needs (and we'll talk about where the rest of that data lives shortly).

Building on our previous, three-step description of what goes on when a computer's ALU is commanded to add two numbers, we can modify it as follows. To execute an `add` instruction, the ALU must perform these steps:

1. Obtain the two numbers to be added (the *input operands*) from two *source registers.*
2. Add the numbers.
3. Place the results back in a *destination register.*

For a concrete example, let's look at addition on a simple computer with only four registers, named `A`, `B`, `C`, and `D`. Suppose each of these registers contains a number, and we want to add the contents of two registers together and overwrite the contents of a third register with the resulting sum, as in the following operation:

Code	**Comments**
`A + B = C`	Add the contents of registers `A` and `B`, and place the result in `C`, overwriting whatever was there.

Upon receiving an instruction commanding it to perform this addition operation, the ALU in our simple computer would carry out the following three familiar steps:

1. Read the contents of registers `A` and `B`.
2. Add the contents of `A` and `B`.
3. Write the result to register `C`.

NOTE *You should recognize these three steps as a more specific form of the read-modify-write sequence from earlier, where the generic modify step is replaced with an addition operation.*

This three-step sequence is quite simple, but it's at the very core of how a microprocessor really works. In fact, if you glance ahead to Chapter 10's discussion of the PowerPC 970's pipeline, you'll see that it actually has separate stages for each of these three operations: stage 12 is the register read step, stage 13 is the actual execute step, and stage 14 is the write-back step. (Don't worry if you don't know what a *pipeline* is, because that's a topic for Chapter 3.) So the 970's ALU reads two operands from the register file, adds them together, and writes the sum back to the register file. If we were to stop our discussion right here, you'd already understand the three core stages of the 970's main integer pipeline—all the other stages are either just preparation to get to this point or they're cleanup work after it.

RAM: When the Registers Alone Don't Cut It

Obviously, four (or even eight) registers aren't even close to the theoretically infinite storage space I mentioned earlier in this chapter. In order to make a viable computer that does useful work, you need to be able to store very large data sets. This is where the computer's *main memory* comes in. Main memory, which in modern computers is always some type of *random access memory (RAM)*, stores the data set on which the computer operates, and only a small portion of that data set at a time is moved to the registers for easy access from the ALU (as shown in Figure 1-4).

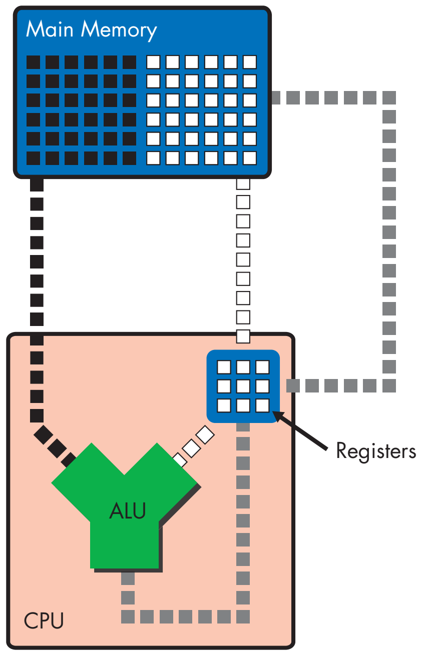


Figure 1-4: A computer with a register file

Figure 1-4 gives only the slightest indication of it, but main memory is situated quite a bit farther away from the ALU than are the registers. In fact, the ALU and the registers are internal parts of the microprocessor, but main memory is a completely separate component of the computer system that is connected to the processor via the *memory bus*. Transferring data between main memory and the registers via the memory bus takes a significant amount of time. Thus, if there were no registers and the ALU had to read data directly from main memory for each calculation, computers would run very slowly. However, because the registers enable the computer to store data near the ALU, where it can be accessed nearly instantaneously, the computer's computational speed is decoupled somewhat from the speed of main memory. (We'll discuss the problem of memory access speeds and computational performance in more detail in Chapter 11, when we talk about caches.)

The File-Clerk Model Revisited and Expanded

To return to our file-clerk metaphor, we can think of main memory as a document storage room located on another floor and the registers as a small, personal filing cabinet where the file clerk places the papers on which he's currently working. The clerk doesn't really know anything about the document storage room—what it is or where it's located—because his desk and his personal filing cabinet are all he concerns himself with. For documents that are in the storage room, there's another office worker, the office secretary, whose job it is to locate files in the storage room and retrieve them for the clerk.

This secretary represents a few different units within the processor, all of which we'll meet Chapter 4. For now, suffice it to say that when the boss wants the clerk to work on a file that's not in the clerk's personal filing cabinet, the secretary must first be ordered, via a message from the boss, to retrieve the file from the storage room and place it in the clerk's cabinet so that the clerk can access it when he gets the order to begin working on it.

An Example: Adding Two Numbers

To translate this office example into computing terms, let's look at how the computer uses main memory, the register file, and the ALU to add two numbers.

To add two numbers stored in main memory, the computer must perform these steps:

1. Load the two operands from main memory into the two source registers.
2. Add the contents of the source registers and place the results in the destination register, using the ALU. To do so, the ALU must perform these steps:
   a. Read the contents of registers `A` and `B` into the ALU's input ports.
   b. Add the contents of `A` and `B` in the ALU.
   c. Write the result to register `C` via the ALU's output port.
3. Store the contents of the destination register in main memory.

Since steps 2a, 2b, and 2c all take a trivial amount of time to complete, relative to steps 1 and 3, we can ignore them. Hence our addition looks like this:

1. Load the two operands from main memory into the two source registers.
2. Add the contents of the source registers, and place the results in the destination register, using the ALU.
3. Store the contents of the destination register in main memory.

The existence of main memory means that the user—the boss in our filing-clerk analogy—must manage the flow of information between main memory and the CPU's registers. This means that the user must issue instructions to more than just the processor's ALU; he or she must also issue instructions to the parts of the CPU that handle memory traffic. Thus, the preceding three steps are representative of the kinds of instructions you find when you take a close look at the code stream.

A Closer Look at the Code Stream: the Program

At the beginning of this chapter, I defined the code stream as consisting of "an ordered sequence of operations," and this definition is fine as far as it goes. But in order to dig deeper, we need a more detailed picture of what the code stream is and how it works.

The term *operations* suggests a series of simple arithmetic operations like addition or subtraction, but the code stream consists of more than just arithmetic operations. Therefore, it would be better to say that the code stream consists of an ordered sequence of *instructions*. Instructions, generally speaking, are commands that tell the whole computer—not just the ALU, but multiple parts of the machine—exactly what actions to perform. As we've seen, a computer's list of potential actions encompasses more than just simple arithmetic operations.

General Instruction Types

Instructions are grouped into ordered lists that, when taken as a whole, tell the different parts of the computer how to work together to perform a specific task, like grayscaling an image or playing a media file. These ordered lists of instructions are called *programs*, and they consist of a few basic types of instructions.

In modern RISC microprocessors, the act of moving data between memory and the registers is under the explicit control of the code stream, or program. So if a programmer wants to add two numbers that are located in main memory and then store the result back in main memory, he or she must write a list of instructions (a program) to tell the computer exactly what to do. The program must consist of

- a `load` instruction to move the two numbers from memory into the registers
- an `add` instruction to tell the ALU to add the two numbers
- a `store` instruction to tell the computer to place the result of the addition back into memory, overwriting whatever was previously there

These operations fall into two main categories:

Arithmetic instructions
These instructions tell the ALU to perform an arithmetic calculation (for example, `add`, `sub`, `mul`, `div`).

Memory-access instructions
These instructions tell the parts of the processor that deal with main memory to move data from and to main memory (for example, `load` and `store`).

NOTE *We'll discuss a third type of instruction, the branch instruction, shortly. Branch instructions are technically a special type of memory-access instruction, but they access code storage instead of data storage. Still, it's easier to treat branches as a third category of instruction.*

The *arithmetic instruction* fits with our calculator metaphor and is the type of instruction most familiar to anyone who's worked with computers. Instructions like integer and floating-point addition, subtraction, multiplication, and division all fall under this general category.

NOTE *In order to simplify the discussion and reduce the number of terms, I'm temporarily including logical operations like* `AND`, `OR`, `NOT`, `NOR`, *and so on, under the general heading of arithmetic instructions. The difference between arithmetic and logical operations will be introduced in Chapter 2.*

The *memory-access instruction* is just as important as the arithmetic instruction, because without access to main memory's data storage regions, the computer would have no way to get data into or out of the register file.

To show you how memory-access and arithmetic operations work together within the context of the code stream, the remainder of this chapter will use a series of increasingly detailed examples. All of the examples are based on a simple, hypothetical computer, which I'll call the DLW-1.[2]

The DLW-1's Basic Architecture and Arithmetic Instruction Format

The DLW-1 microprocessor consists of an ALU (along with a few other units that I'll describe later) attached to four registers, named `A`, `B`, `C`, and `D` for convenience. The DLW-1 is attached to a bank of main memory that's laid out as a line of 256 memory cells, numbered `#0` to `#255`. (The number that identifies an individual memory cell is called an *address*.)

The DLW-1's Arithmetic Instruction Format

All of the DLW-1's arithmetic instructions are in the following *instruction format*:

```
instruction source1, source2, destination
```

There are four parts to this instruction format, each of which is called a *field*. The *instruction* field specifies the type of operation being performed (for example, an addition, a subtraction, a multiplication, and so on). The two *source* fields tell the computer which registers hold the two numbers being operated on, or the *operands*. Finally, the *destination* field tells the computer which register to place the result in.

As a quick illustration, an addition instruction that adds the numbers in registers `A` and `B` (the two source registers) and places the result in register `C` (the destination register) would look like this:

Code	Comments
`add A, B, C`	Add the contents of registers `A` and `B` and place the result in `C`, overwriting whatever was previously there.

[2] "DLW" in honor of the DLX architecture used by Hennessy and Patterson in their books on computer architecture.

The DLW-1's Memory Instruction Format

In order to get the processor to move two operands from main memory into the source registers so they can be added, you need to tell the processor explicitly that you want to move the data in two specific memory cells to two specific registers. This "filing" operation is done via a memory-access instruction called the `load`.

As its name suggests, the `load` instruction loads the appropriate data from main memory into the appropriate registers so that the data will be available for subsequent arithmetic instructions. The `store` instruction is the reverse of the `load` instruction, and it takes data from a register and stores it in a location in main memory, overwriting whatever was there previously.

All of the memory-access instructions for the DLW-1 have the following instruction format:

```
instruction source, destination
```

For all memory accesses, the instruction field specifies the type of memory operation to be performed (either a load or a store). In the case of a load, the source field tells the computer which memory address to fetch the data from, while the destination field specifies which register to put it in. Conversely, in the case of a `store`, the source field tells the computer which register to take the data from, and the destination field specifies which memory address to write the data to.

An Example DLW-1 Program

Now consider Program 1-1, which is a piece of DLW-1 code. Each of the lines in the program must be executed in sequence to achieve the desired result.

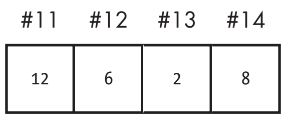
Line	Code	Comments
1	`load #12, A`	Read the contents of memory cell `#12` into register `A`.
2	`load #13, B`	Read the contents of memory cell `#13` into register `B`.
3	`add A, B, C`	Add the numbers in registers `A` and `B` and store the result in `C`.
4	`store C, #14`	Write the result of the addition from register `C` into memory cell `#14`.

Program 1-1: Program to add two numbers from main memory

Suppose the main memory looked like the following before running Program 1-1:

#11	#12	#13	#14
12	6	2	3

After doing our addition and storing the results, the memory would be changed so that the contents of cell `#14` would be overwritten by the sum of cells `#12` and `#13`, as shown here:



A Closer Look at Memory Accesses: Register vs. Immediate

The examples so far presume that the programmer knows the exact memory location of every number that he or she wants to load and store. In other words, it presumes that in composing each program, the programmer has at his or her disposal a list of the contents of memory cells `#0` through `#255`.

While such an accurate snapshot of the initial state of main memory may be feasible for a small example computer with only 256 memory locations, such snapshots almost never exist in the real world. Real computers have billions of possible locations in which data can be stored, so programmers need a more flexible way to access memory, a way that doesn't require each memory access to specify numerically an exact memory address.

Modern computers allow the *contents* of a register to be used as a memory address, a move that provides the programmer with the desired flexibility. But before discussing the effects of this move in more detail, let's take one more look at the basic `add` instruction.

Immediate Values

All of the arithmetic instructions so far have required two source registers as input. However, it's possible to replace one or both of the source registers with an explicit numerical value, called an *immediate value*. For instance, to increase whatever number is in register `A` by 2, we don't need to load the value 2 into a second source register, like `B`, from some cell in main memory that contains that value. Rather, we can just tell the computer to add 2 to `A` directly, as follows:

Code	**Comments**
`add A, 2, A`	Add 2 to the contents of register `A` and place the result back into `A`, overwriting whatever was there.

I've actually been using immediate values all along in my examples, but just not in any arithmetic instructions. In all of the preceding examples, each load and store uses an immediate value in order to specify a memory address. So the `#12` in the `load` instruction in line 1 of Program 1-1 is just an immediate value (a regular whole number) prefixed by a `#` sign to let the computer know that this particular immediate value is a memory address that designates a cell in memory.

Memory addresses are just regular whole numbers that are specially marked with the `#` sign. Because they're regular whole numbers, they can be stored in registers—and stored in memory—just like any other number. Thus, the whole-number contents of a register, like `D`, could be construed by the computer as representing a memory address.

For example, say that we've stored the number 12 in register `D`, and that we intend to use the contents of `D` as the address of a memory cell in Program 1-2.

Line	Code	Comments
1	`load #D, A`	Read the contents of the memory cell designated by the number stored in `D` (where `D = 12`) into register `A`.
2	`load #13, B`	Read the contents of memory cell `#13` into register `B`.
3	`add A, B, C`	Add the numbers in registers `A` and `B` and store the result in `C`.
4	`store C, #14`	Write the result of the addition from register `C` into memory cell `#14`.

Program 1-2: Program to add two numbers from main memory using an address stored in a register

Program 1-2 is essentially the same as Program 1-1, and given the same input, it yields the same results. The only difference is in line 1:

Program 1-1, Line 1	Program 1-2, Line 1
`load #12, A`	`load #D, A`

Since the content of `D` is the number 12, we can tell the computer to look in `D` for the memory cell address by substituting the register name (this time marked with a `#` sign for use as an address), for the actual memory cell number in line 1's `load` instruction. Thus, the first lines of Programs 1-1 and 1-2 are functionally equivalent.

This same trick works for `store` instructions, as well. For example, if we place the number 14 in `D` we can modify the `store` command in line 4 of Program 1-1 to read as follows: `store C, #D`. Again, this modification would not change the program's output.

Because memory addresses are just regular numbers, they can be stored in memory cells as well as in registers. Program 1-3 illustrates the use of a memory address that's stored in another memory cell. If we take the input for Program 1-1 and apply it to Program 1-3, we get the same output as if we'd just run Program 1-1 without modification:

Line	Code	Comments
1	`load #11, D`	Read the contents of memory cell `#11` into `D`.
2	`load #D, A`	Read the contents of the memory cell designated by the number in `D` (where `D = 12`) into register `A`.
3	`load #13, B`	Read the contents of memory cell `#13` into register `B`.
4	`add A, B, C`	Add the numbers in registers `A` and `B` and store the result in `C`.
5	`store C, #14`	Write the result of the addition from register `C` into memory cell `#14`.

Program 1-3: Program to add two numbers from memory using an address stored in a memory cell.

The first instruction in Program 1-3 loads the number 12 from memory cell `#11` into register `D`. The second instruction then uses the content of `D` (which is the value 12) as a memory address in order to load register `A` into memory location `#12`.

But why go to the trouble of storing memory addresses in memory cells and then loading the addresses from main memory into the registers before they're finally ready to be used to access memory again? Isn't this an overly complicated way to do things?

Actually, these capabilities are designed to make programmers' lives easier, because when used with the register-relative addressing technique described next they make managing code and data traffic between the processor and massive amounts of main memory much less complex.

Register-Relative Addressing

In real-world programs, `load`s and `store`s most often use *register-relative addressing*, which is a way of specifying memory addresses relative to a register that contains a fixed *base address*.

For example, we've been using `D` to store memory addresses, so let's say that on the DLW-1 we can assume that, unless it is explicitly told to otherwise, the operating system always loads the starting address (or base address) of a program's *data segment* into `D`. Remember that code and data are logically separated in main memory, and that data flows into the processor from a data storage area, while code flows into the processor from a special code storage area. Main memory itself is just one long row of undifferentiated memory cells, each one *byte* in width, that store numbers. The computer carves up this long row of bytes into multiple segments, some of which store code and some of which store data.

A *data segment* is a block of contiguous memory cells that a program stores all of its data in, so if a programmer knows a data segment's starting address (*base address*) in memory, he or she can access all of the other memory locations in that segment using this formula:

```
base address + offset
```

where *offset* is the distance in bytes of the desired memory location from the data segment's base address.

Thus, `load` and `store` instructions in DLW-1 assembly would normally look something like this:

Code	**Comments**
`load #(D + 108), A`	Read the contents of the memory cell at location `#(D + 108)` into `A`.
`store B, #(D + 108)`	Write the contents of `B` into the memory cell at location `#(D + 108)`.

In the case of the `load`, the processor takes the number in `D`, which is the base address of the data segment, adds 108 to it, and uses the result as the `load`'s destination memory address. The `store` works in the exact same way.

Of course, this technique requires that a quick addition operation (called an *address calculation*) be part of the execution of the `load` instruction, so this is why the *load-store units* on modern processors contain very fast integer addition hardware. (As we'll learn in Chapter 4, the load-store unit is the *execution unit* responsible for executing `load` and `store` instructions, just like the arithmetic-logic unit is responsible for executing arithmetic instructions.)

By using register-relative addressing instead of *absolute addressing* (in which memory addresses are given as immediate values), a programmer can write programs without knowing the exact location of data in memory. All the programmer needs to know is which register the operating system will place the data segment's base address in, and he or she can do all memory accesses relative to that base address. In situations where a programmer uses absolute addressing, when the operating system loads the program into memory, all of the program's immediate address values have to be changed to reflect the data segment's actual location in memory.

Because both memory addresses and regular integer numbers are stored in the same registers, these registers are called *general-purpose registers (GPRs)*. On the DLW-1, `A`, `B`, `C`, and `D` are all GPRs.

2

THE MECHANICS OF PROGRAM EXECUTION

Now that we understand the basics of computer organization, it’s time to take a closer look at the nuts and bolts of how stored programs are actually executed by the computer. To that end, this chapter will cover core programming concepts like machine language, the programming model, the instruction set architecture, branch instructions, and the fetch-execute loop.

Opcodes and Machine Language

If you’ve been following the discussion so far, it shouldn’t surprise you to learn that both memory addresses and instructions are ordinary numbers that can be stored in memory. All of the instructions in a program like Program 1-1 are represented inside the computer as strings of numbers. Indeed, a program is one long string of numbers stored in a series of memory locations.

How is a program like Program 1-1 rendered in numerical notation so that it can be stored in memory and executed by the computer? The answer is simpler than you might think.

As you may already know, a computer actually only understands 1s and 0s (or "high" and "low" electric voltages), not English words like *add*, *load*, and *store*, or letters and base-10 numbers like A, B, 12, and 13. In order for the computer to run a program, therefore, all of its instructions must be rendered in *binary notation*. Think of translating English words into Morse code's dots and dashes and you'll have some idea of what I'm talking about.

Machine Language on the DLW-1

The translation of programs of any complexity into this binary-based *machine language* is a massive undertaking that's meant to be done by a computer, but I'll show you the basics of how it works so you can understand what's going on. The following example is simplified, but useful nonetheless.

The English words in a program, like *add*, *load*, and *store*, are *mnemonics* (meaning they're easy for people to remember), and they're all mapped to strings of binary numbers, called *opcodes*, that the computer can understand. Each opcode designates a different operation that the processor can perform. Table 2-1 maps each of the mnemonics used in Chapter 1 to a 3-bit opcode for the hypothetical DLW-1 microprocessor. We can also map the four register names to 2-bit binary codes, as shown in Table 2-2.

Table 2-1: Mapping of Mnemonics to Opcodes for the DLW-1

Mnemonic	Opcode
`add`	000
`sub`	001
`load`	010
`store`	011

Table 2-2: Mapping of Registers to Binary Codes for the DLW-1

Register	Binary Code
`A`	00
`B`	01
`C`	10
`D`	11

The binary values representing both the opcodes and the register codes are arranged in one of a number of 16-bit (or 2-byte) formats to get a complete *machine language instruction,* which is a binary number that can be stored in RAM and used by the processor.

NOTE *Because programmer-written instructions must be translated into binary codes before a computer can read them, it is common to see programs in any format—binary, assembly, or a high-level language like BASIC or C, referred to generically as "code" or "codes." So programmers sometimes speak of "assembler code," "binary code," or "C code," when referring to programs written in assembly, binary, or C language. Programmers also will often describe the act of programming as "writing code" or "coding." I have adopted this terminology in this book, and will henceforth use the term "code" regularly to refer generically to instruction sequences and programs.*

Binary Encoding of Arithmetic Instructions

Arithmetic instructions have the simplest machine language instruction formats, so we'll start with them. Figure 2-1 shows the format for the machine language encoding of a *register-type* arithmetic instruction.

0	1	2	3	4	5	6	7
mode	opcode			source1		source2	

Byte 1

8	9	10	11	12	13	14	15
destination		000000					

Byte 2

Figure 2-1: Machine language format for a register-type instruction

In a register-type arithmetic instruction (that is, an arithmetic instruction that uses only registers and no immediate values), the first bit of the instruction is the *mode bit*. If the mode bit is set to 0, then the instruction is a register-type instruction; if it's set to 1, then the instruction is of the immediate type.

Bits 1–3 of the instruction specify the opcode, which tells the computer what type of operation the instruction represents. Bits 4–5 specify the instruction's first source register, 6–7 specify the second source register, and 8–9 specify the destination register. The last six bits are not needed by register-to-register arithmetic instructions, so they're padded with 0s (they're *zeroed out* in computer jargon) and ignored.

Now, let's use the binary values in Tables 2-1 and 2-2 to translate the `add` instruction in line 3 of Program 1-1 into a 2-byte (or 16-bit) machine language instruction:

Assembly Language Instruction	Machine Language Instruction
`add A, B, C`	00000001 10000000

Here are a few more examples of arithmetic instructions, just so you can get the hang of it:

Assembly Language Instruction	Machine Language Instruction
`add C, D, A`	00001011 00000000
`add D, B, C`	00001101 10000000
`sub A, D, C`	00010011 10000000

Increasing the number of binary digits in the opcode and register fields increases the total number of instructions the machine can use and the number of registers it can have. For example, if you know something about binary notation, then you probably know that a 3-bit opcode allows the processor to map up to 2^3 mnemonics, which means that it can have up to 2^3, or 8, instructions in its *instruction set*; increasing the opcode size to 8 bits would allow the processor's instruction set to contain up to 2^8, or 256, instructions. Similarly, increasing the number of bits in the register field increases the possible number of registers that the machine can have.

Arithmetic instructions containing an immediate value use an *immediate-type* instruction format, which is slightly different from the register-type format we just saw. In an immediate-type instruction, the first byte contains the opcode, the source register, and the destination register, while the second byte contains the immediate value, as shown in Figure 2-2.

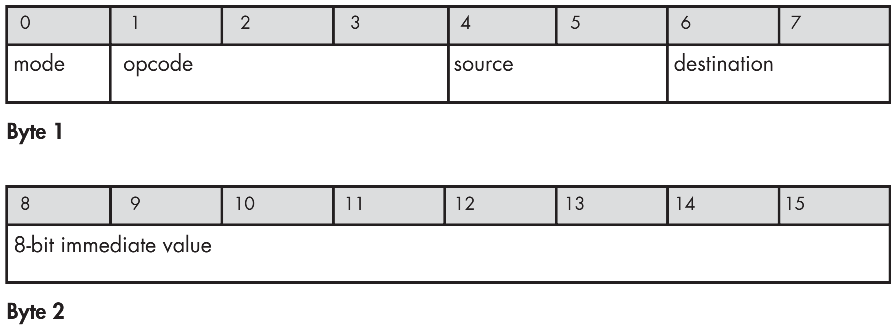


Figure 2-2: Machine language format for an immediate-type instruction

Here are a few immediate-type arithmetic instructions translated from assembly language to machine language:

Assembly Language Instruction	Machine Language Instruction
`add C, 8, A`	10001000 00001000
`add 5, A, C`	10000010 00000101
`sub 25, D, C`	10011110 00011001

Binary Encoding of Memory Access Instructions

Memory-access instructions use both register- and immediate-type instruction formats exactly like those shown for arithmetic instructions. The only difference lies in how they use them. Let's take the case of a load first.

The load Instruction

We've previously seen two types of `load`, the first of which was the immediate type. An immediate-type `load` (see Figure 2-3) uses the immediate-type instruction format, but because the `load`'s source is an immediate value (a memory address) and not a register, the source field is unneeded and must be zeroed out. (The source field is not ignored, though, and in a moment we'll see what happens if it isn't zeroed out.)

0	1	2	3	4	5	6	7
mode	opcode			00		destination	

Byte 1

8	9	10	11	12	13	14	15
8-bit immediate source address							

Byte 2

Figure 2-3: Machine language format for an immediate-type load

Now let's translate the immediate-type `load` in line 1 of Program 1-1 (12 is 1100 in binary notation):

Assembly Language Instruction	Machine Language Instruction
`load #12, A`	10100000 00001100

The 2-byte machine language instruction on the right is a binary representation of the assembly language instruction on the left. The first byte corresponds to an immediate-type `load` instruction that takes register `A` as its destination. The second byte is the binary representation of the number `12`, which is the source address in memory that the data is to be loaded from.

The second type of load we've seen is the register type. A register-type load uses the register-type instruction format, but with the source2 field zeroed out and ignored, as shown in Figure 2-4.

In Figure 2-4, the source1 field specifies the register containing the memory address that the processor is to load data from, and the destination field specifies the register that the loaded data is to be placed in.

0	1	2	3	4	5	6	7
mode	opcode			source1		00	

Byte 1

8	9	10	11	12	13	14	15
destination		000000					

Byte 2

Figure 2-4: Machine language format for a register-type load

For a register-relative addressed `load`, we use a version of the immediate-type instruction format, shown in Figure 2-5, with the base field specifying the register that contains the base address and the offset stored in the second byte of the instruction.

0	1	2	3	4	5	6	7
mode	opcode			base		destination	

Byte 1

8	9	10	11	12	13	14	15
8-bit immediate offset							

Byte 2

Figure 2-5: Machine language format for a register-relative load

Recall from Table 2-2 that 00 is the binary number that designates register `A`. Therefore, as a result of the DLW-1's particular machine language encoding scheme, any register but `A` could theoretically be used to store the base address for a register-relative `load`.

The store Instruction

The register-type binary format for a `store` instruction is the same as it is for a `load`, except that the destination field specifies a register containing a destination memory address, and the source1 field specifies the register containing the data to be stored to memory.

The immediate-type machine language format for a store, pictured in Figure 2-6, is also similar to the immediate-type format for a load, except that since the destination register is not needed (the destination is the immediate memory address) the destination field is zeroed out, while the source field specifies which register holds the data to be stored.

0	1	2	3	4	5	6	7
mode	011			source		00	

Byte 1

8	9	10	11	12	13	14	15
8-bit immediate destination address							

Byte 2

Figure 2-6: Machine language format for an immediate-type store

The register-relative `store`, on the other hand, uses the same immediate-type instruction format used for the register-relative `load` (Figure 2-5), but the destination field is set to a nonzero value, and the offset is stored in the second byte. Again, the base address for a register-relative `store` can theoretically be stored in any register other than `A`, although by convention it's stored in `D`.

Translating an Example Program into Machine Language

For our simple computer with four registers, three instructions, and 256 memory cells, it's tedious but trivial to translate Program 1-1 into machine-readable binary representation using the previous tables and instruction formats. Program 2-1 shows the translation.

Line	Assembly Language	Machine Language
1	`load #12, A`	10100000 00001100
2	`load #13, B`	10100001 00001101
3	`add A, B, C`	00000001 10000000
4	`store C, #14`	10111000 00001110

Program 2-1: A translation of Program 1-1 into machine language

The 1s and 0s in the rightmost column of Program 2-1 represent the high and low voltages that the computer "thinks" in.

Real machine language instructions are usually longer and more complex than the simple ones I've given here, but the basic idea is exactly the same. Program instructions are translated into machine language in a mechanical, predefined manner, and even in the case of a fully modern microprocessor, doing such translations by hand is merely a matter of knowing the instruction formats and having access to the right charts and tables.

Of course, for the most part the only people who do such translations by hand are computer engineering or computer science undergraduates who've been assigned them for homework. This wasn't always the case, though.

The Programming Model and the ISA

Back in the bad old days, programmers had to enter programs into the computer directly in machine language (after having walked five miles in the snow uphill to work). In the very early stages of computing, this was done by flipping switches. The programmer toggled strings of 1s and 0s into the computer's very limited memory, ran the program, and then pored over the resulting strings of 1s and 0s to decode the answer.

Once memory sizes and processing power increased to the point where programmer time and effort were valuable enough relative to computing time and memory space, computer scientists devised ways of allowing the computer to use a portion of its power and memory to take on some of the burden of making its cryptic input and output a little more human-friendly.

In short, the tedious task of converting human-readable programs into machine-readable binary code was automated; hence the birth of *assembly language* programming. Programs could now be written using mnemonics, register names, and memory locations, before being converted by an *assembler* into machine language for processing.

In order to write assembly language programs for a machine, you have to understand the machine's available resources: how many registers it has, what instructions it supports, and so on. In other words, you need a well-defined model of the machine you're trying to program.

The Programming Model

The *programming model* is the programmer's interface to the microprocessor. It hides all of the processor's complex implementation details behind a relatively simple, clean layer of abstraction that exposes to the programmer all of the processor's functionality. (See Chapter 4 for more on the history and development of the programming model.)

Figure 2-7 shows a diagram of a programming model for an eight-register machine. By now, most of the parts of the diagram should be familiar to you. The ALU performs arithmetic, the registers store numbers, and the *input-output unit (I/O unit)* is responsible for interacting with memory and the rest of the system (via loads and stores). The parts of the processor that we haven't yet met lie in the *control unit*. Of these, we'll cover the *program counter* and the *instruction register* now.

The Instruction Register and Program Counter

Because programs are stored in memory as ordered sequences of instructions and memory is arranged as a linear series of addresses, each instruction in a program lives at its own memory address. In order to step through and execute the lines of a program, the computer simply begins at the program's starting address and then steps through each successive memory location, fetching each successive instruction from memory, placing it in a special register, and executing it as shown in Figure 2-8.

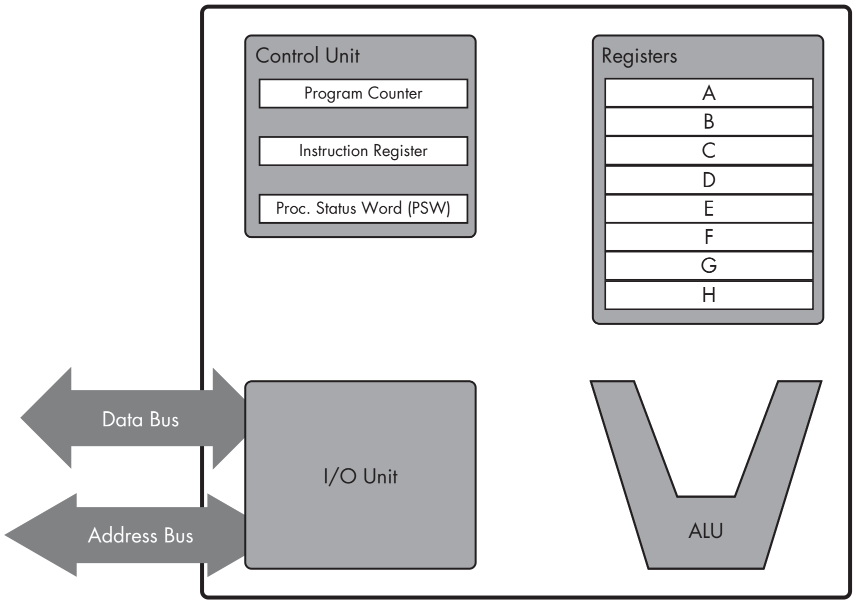


Figure 2-7: The programming model for a simple eight-register machine

The instructions in our DLW-1 computer are two bytes long. If we assume that each memory cell holds one byte, then the DLW-1 must step through memory by fetching instructions from two cells at a time.

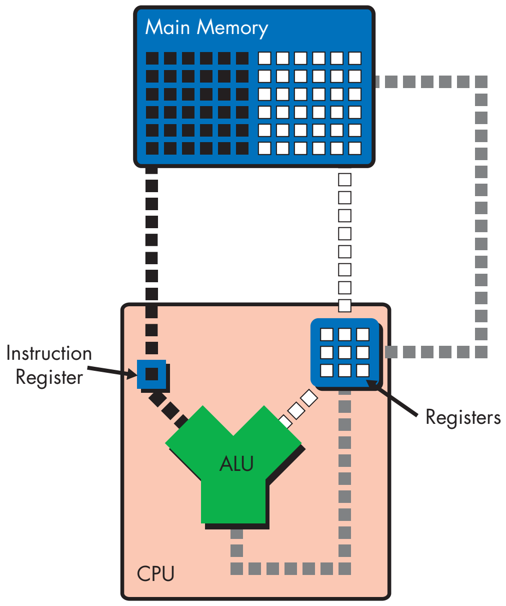


Figure 2-8: A simple computer with instruction and data registers

For example, if the starting address in Program 1-1 were #500, it would look like Figure 2-9 in memory (with the instructions rendered in machine language, not assembly language, of course).

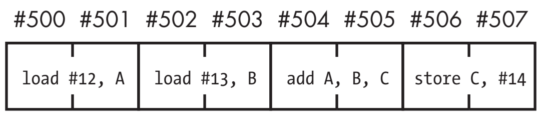


Figure 2-9: An illustration of Program 1-1 in memory, starting at address #500

The Instruction Fetch: Loading the Instruction Register

An *instruction fetch* is a special type of load that happens automatically for every instruction. It always takes the address that's currently in the program counter register as its source and the *instruction register* as its destination. The control unit uses a fetch to load each instruction of a program from memory into the instruction register, where that instruction is *decoded* before being executed; and while that instruction is being decoded, the processor places the address of the next instruction into the program counter by incrementing the address that's currently in the program counter, so that the newly incremented address points to the next instruction the sequence. In the case of our DLW-1, the program counter is incremented by two every time an instruction is fetched, because the two-byte instructions begin at every other byte in memory.

Running a Simple Program: the Fetch-Execute Loop

In Chapter 1 we discussed the steps a processor takes to perform calculations on numbers using the ALU in combination with a fetched arithmetic instruction. Now let's look at the steps the processor takes in order to fetch a series of instructions—a program—and feed them to either the ALU (in the case of arithmetic instructions) or the memory access hardware (in the case of loads and stores):

1. *Fetch* the next instruction from the address stored in the program counter, and load that instruction into the instruction register. Increment the program counter.
2. *Decode* the instruction in the instruction register.
3. *Execute* the instruction in the instruction register, using the following rules:
    a. If the instruction is an arithmetic instruction, execute it using the ALU and register file.
    b. If the instruction is a memory access instruction, execute it using the memory-access hardware.

These three steps are fairly straightforward, and with one modification they describe the way that microprocessors execute programs (as we'll see in the section "Branch Instructions" on page 30). Computer scientists often refer to these steps as the *fetch-execute loop* or the *fetch-execute cycle*. The fetch-execute loop is repeated for as long as the computer is powered on. The machine iterates through the entire loop, from step 1 to step 3, over and over again many millions or billions of times per second in order to run programs.

Let's run through the three steps with our example program as shown in Figure 2-9. (This example presumes that #500 is already in the program counter.) Here's what the processor does, in order:

1. Fetch the instruction beginning at #500, and load `load #12, A` into the instruction register. Increment the program counter to #502.
2. Decode `load #12, A` in the instruction register.
3. Execute `load #12, A` from the instruction register, using the memory-access hardware.
4. Fetch the instruction beginning at #502, and load `load #13, B` in the instruction register. Increment the program counter to #504.
5. Decode `load #13, B` in the instruction register.
6. Execute `load #13, B` from the instruction register, using the memory-access hardware.
7. Fetch the instruction beginning at #504, and load `add A, B, C` into the instruction register. Increment the program counter to #506.
8. Decode `add A, B, C` in the instruction register.
9. Execute `add A, B, C` from the instruction register, using the ALU and register file.
10. Fetch the instruction at #506, and load `store C, #14` in the instruction register. Increment the program counter to #508.
11. Decode `store C, #14` in the instruction register.
12. Execute `store C, #14` from the instruction register, using the memory-access hardware.

NOTE *To zoom in on the execute steps of the preceding sequence, revisit Chapter 1, and particularly the sections "Refining the File-Clerk Model" on page 6 and "RAM: When the Registers Alone Don't Cut It" on page 8. If you do, you'll gain a pretty good understanding of what's involved in executing a program on any machine. Sure, there are important machine-specific variations for most of what I've presented here, but the general outlines (and even a decent number of the specifics) are the same.*

The Clock

Steps 1 through 12 in the previous section don't take an arbitrary amount of time to complete. Rather, they're performed according to the pulse of the clock that governs every action the processor takes.

This clock pulse, which is generated by a *clock generator* module on the motherboard and is fed into the processor from the outside, times the functioning of the processor so that, on the DLW-1 at least, all three steps of the fetch-execute loop are completed in exactly one beat of the clock. Thus, the program in Figure 2-9, as I've traced its execution in the preceding section, takes exactly four clock beats to finish execution, because a new instruction is fetched on each beat of the clock.

One obvious way to speed up the execution of programs on the DLW-1 would be to speed up its clock generator so that each step takes less time to complete. This is generally true of all microprocessors, hence the race among microprocessor designers to build and market chips with ever-higher clock speeds. (We'll talk more about the relationship between clock speed and performance in Chapter 3.)

Branch Instructions

As I've presented it so far, the processor moves through each line in a program in sequence until it reaches the end of the program, at which point the program's output is available to the user.

There are certain instructions in the instruction stream, however, that allow the processor to jump to a program line that is out of sequence. For instance, by inserting a *branch instruction* into line 5 of a program, we could cause the processor's control unit to jump all the way down to line 20 and begin executing there (a *forward branch*), or we could cause it to jump back up to line 1 (a *backward branch*). Because a program is an ordered sequence of instructions, by including forward and backward branch instructions, we can arbitrarily move about in the program. This is a powerful ability, and branches are an essential part of computing.

Rather than thinking about forward or backward branches, it's more useful for our present purposes to categorize all branches as being one of the following two types: conditional branches or unconditional branches.

Unconditional Branch

An *unconditional branch* instruction consists of two parts: the branch instruction and the target address.

```
jump #target
```

For an unconditional branch, `#target` can be either an immediate value, like `#12`, or an address stored in a register, like `#D`.

Unconditional branches are fairly easy to execute, since all that the computer needs to do upon decoding such a branch in the instruction register is to have the control unit replace the address currently in the program counter with branch's target address. Then the next time the processor goes to fetch the instruction at the address given by the program counter, it'll fetch the address at the branch target instead.

Conditional Branch

Though it has the same basic instruction format as the unconditional branch (`instruction #target`), the *conditional branch* instruction is a little more complicated, because it involves jumping to the target address only if a certain condition is met.

For example, say we want to jump to a new line of the program only if the previous arithmetic instruction's result is zero; if the result is nonzero, we want to continue executing normally. We would use a conditional branch instruction that first checks to see if the previously executed arithmetic instruction yielded a zero result, and then writes the branch target into the program counter if it did.

Because of such conditional jumps, we need a special register or set of registers in which to store information about the results of arithmetic instructions—information such as whether the previous result was zero or nonzero, positive or negative, and so on.

Different architectures handle this in different ways, but in our DLW-1, this is the function of the *processor status word (PSW)* register. On the DLW-1, every arithmetic operation stores different types of data about its outcome in the PSW upon completion. To execute a conditional branch, the DLW-1 must first *evaluate* the condition on which the branch depends (e.g., "is the previous arithmetic instruction's result zero?" in the preceding example) by checking the appropriate bit in the PSW to see if that condition is true or false. If the branch condition evaluates to true, then the control unit replaces the address in the program counter with the branch target address. If the branch condition evaluates to false, then the program counter is left as-is, and the next instruction in the normal program sequence is fetched on the next cycle.

For example, suppose we had just subtracted the number in `A` from the number in `B`, and if the result was zero (that is, if the two numbers were equal), we want to jump to the instruction at memory address #106. Program 2-2 shows what assembler code for such a conditional branch might look like.

Line	**Code**	**Comments**
16	`sub A, B, C`	Subtract the number in register `A` from the number in register `B` and store the result in `C`.
17	`jumpz #106`	Check the PSW, and if the result of the previous instruction was zero, jump to the instruction at address #106. If the result was nonzero, continue on to line 18.
18	`add A, B, C`	Add the numbers in registers `A` and `B` and store the result in `C`.

Program 2-2: Assembler code for a conditional branch

The `jumpz` instruction causes the processor to check the PSW to determine whether a certain bit is 1 (true) or 0 (false). If the bit is 1, the result of the subtraction instruction was 0 and the program counter must be loaded with the branch target address. If the bit is 0, the program counter is incremented to point to the next instruction in sequence (which is the `add` instruction in line 18).

There are other bits in the PSW that specify other types of information about the result of the previous operation (whether it is positive or negative, is too large for the registers to hold, and so on). As such, there are also other types of conditional branch instructions that check these bits. For instance, the `jumpn` instruction jumps to the target address if the result of the preceding arithmetic operation was negative; the `jumpo` instruction jumps to the target address if the result of the previous operation was too large and overflowed the register. If the machine language instruction format of the DLW-1 could accommodate more than eight possible instructions, we could add more types of conditional jumps.

Branch Instructions and the Fetch-Execute Loop

Now that we have looked at the basics of branching, we can modify our three-step summary of program execution to include the possibility of a branch instruction:

1. *Fetch* the next instruction from the address stored in the program counter, and load that instruction into the instruction register. Increment the program counter.
2. *Decode* the instruction in the instruction register.
3. *Execute* the instruction in the instruction register, using the following rules:
   a. If the instruction is an arithmetic instruction, then execute it using the ALU and register file.
   b. If the instruction is a memory-access instruction, then execute it using the memory hardware.
   c. If the instruction is a branch instruction, then execute it using the control unit and the program counter. (For a taken branch, write the branch target address into the program counter.)

In short, you might say that branch instructions allow the programmer to redirect the processor as it travels through the instruction stream. Branches point the processor to different sections of the code stream by manipulating its control unit, which, because it contains the instruction register and program counter, is the rudder of the CPU.

The Branch Instruction as a Special Type of Load

Recall that an instruction fetch is a special type of load that happens automatically for every instruction and that always takes the address in the program counter as its source and the instruction register as its destination. With that in mind, you might think of a branch instruction as a similar kind of load, but under the control of the programmer instead of the CPU. The branch instruction is a load that takes the address specified by `#target` as its source and the instruction register as its destination.

Like a regular load, a branch instruction can take as its target an address stored in a register. In other words, branch instructions can use register-relative addressing just like regular `load` instructions. This capability is useful because it allows the computer to store blocks of code at arbitrary places in memory. The programmer doesn't need to know the address at which a block of code will wind up before writing a branch instruction that jumps to that particular block; all he or she needs is a way to get to the memory location where the operating system, which is responsible for managing memory, has stored the starting address of the desired block of code.

Consider Program 2-3, in which the programmer knows that the operating system has placed the address of the branch target in line 17 in register `C`. Upon reaching line 17, the computer jumps to the address stored in `C` by copying the contents of `C` into the instruction register.

Line	Code	Comments
16	`sub A, B, A`	Subtract the number in register `A` from the number in register `B` and store the result in `A`.
17	`jumpz #C`	Check the PSW, and if the result of the previous instruction was zero, jump to the instruction at the address stored in `C`. If the result was nonzero, continue on to line 18.
18	`add A, 15, A`	Add 15 to the number in `A` and store the result in `A`.

Program 2-3: A conditional branch that uses an address stored in a register

When a programmer uses register-relative addressing with a branch instruction, the operating system must load a certain register with the base address of the *code segment* in which the program resides. Like the data segment, the code segment is a contiguous block of memory cells, but its cells store instructions instead of data. So to jump to line 15 in the currently running program, assuming that the operating system has placed the base address of the code segment in `C`, the programmer could use the following instruction:

Code	Comments
`jump #(C + 30)`	Jump to the instruction located 30 bytes away from the start of the code segment. (Each instruction is 2 bytes in length, so this puts us at the 15 instruction.)

Branch Instructions and Labels

In programs written for real-world architectures, branch targets don't usually take the form of either immediate values or register-relative values. Rather, the programmer places a *label* on the line of code to which he or she wants to jump, and then puts that label in the branch's target field. Program 2-4 shows a portion of assembly language code that uses labels.

```
      sub A, B, A
      jumpz LBL1
      add A, 15, A
      store A, #(D + 16)
LBL1: add A, B, B
      store B, #(D + 16)
```

Program 2-4: Assembly language code that uses labels

In this example, if the contents of `A` and `B` are equal, the computer will jump to the instruction with the label LBL1 and begin executing there, skipping the instructions between the `jump` and the labeled `add`. Just as the absolute memory addresses used in `load` and `store` instructions are modified at load time to fit the location in memory of the program's data segment, labels like LBL1 are changed at load time into memory addresses that reflect the location in memory of the program's code segment.

Excursus: Booting Up

If you've been around computers for any length of time, you've heard the terms reboot or boot up used in connection with either resetting the computer to its initial state or powering it on initially. The term boot is a shortened version of the term bootstrap, which is itself a reference to the seemingly impossible task a computer must perform on start-up, namely, "pulling itself up by its own bootstraps."

I say "seemingly impossible," because when a computer is first powered on there is no program in memory, but programs contain the instructions that make the computer run. If the processor has no program running when it's first powered on, then how does it know where to fetch the first instruction from?

The solution to this dilemma is that the microprocessor, in its power-on default state, is hard-wired to fetch that first instruction from a predetermined address in memory. This first instruction, which is loaded into the processor's instruction register, is the first line of a program called the BIOS that lives in a special set of storage locations—a small read-only memory (ROM) module attached to the computer's motherboard. It's the job of the BIOS to perform basic tests of the RAM and peripherals in order to verify that everything is working properly. Then the boot process can continue.

At the end of the BIOS program lies a jump instruction, the target of which is the location of a *bootloader* program. By using a jump, the BIOS hands off control of the system to this second program, whose job it is to search for and load the computer's operating system from the hard disk. The operating system (OS) loads and unloads all of the other programs that run on the computer, so once the OS is up and running the computer is ready to interact with the user.

3

PIPELINED EXECUTION

All of the processor architectures that you've looked at so far are relatively simple, and they reflect the earliest stages of computer evolution. This chapter will bring you closer to the modern computing era by introducing one of the key innovations that underlies the rapid performance increases that have characterized the past few decades of microprocessor development: *pipelined execution.*

Pipelined execution is a technique that enables microprocessor designers to increase the speed at which a processor operates, thereby decreasing the amount of time that the processor takes to execute a program. This chapter will first introduce the concept of pipelining by means of a factory analogy, and it will then apply the analogy to microprocessors. You'll then learn how to evaluate the benefits of pipelining, before I conclude with a discussion of the technique's limitations and costs.

NOTE *This chapter's discussion of pipelined execution focuses solely on the execution of arithmetic instructions. Memory instructions and branch instructions are pipelined using the same fundamental principles as arithmetic instructions, and later chapters will cover the peculiarities of the actual execution process of each of these two types of instruction.*

The Lifecycle of an Instruction

In the previous chapter, you learned that a computer repeats three basic steps over and over again in order to execute a program:

1. *Fetch* the next instruction from the address stored in the program counter and load that instruction into the instruction register. Increment the program counter.
2. *Decode* the instruction in the instruction register.
3. *Execute* the instruction in the instruction register.

You should also recall that step 3, the execute step, itself can consist of multiple sub-steps, depending on the type of instruction being executed (arithmetic, memory access, or branch). In the case of the arithmetic instruction `add A, B, C`, the example we used last time, the three sub-steps are as follows:

1. *Read* the contents of registers `A` and `B`.
2. *Add* the contents of `A` and `B`.
3. *Write* the result back to register `C`.

Thus the expanded list of actions required to execute an arithmetic instruction is as follows (substitute any other arithmetic instruction for *add* in the following list to see how it's executed):

1. *Fetch* the next instruction from the address stored in the program counter and load that instruction into the instruction register. Increment the program counter.
2. *Decode* the instruction in the instruction register.
3. *Execute* the instruction in the instruction register. Because the instruction is not a branch instruction but an arithmetic instruction, send it to the arithmetic logic unit (ALU).
   a. *Read* the contents of registers `A` and `B`.
   b. *Add* the contents of `A` and `B`.
   c. *Write* the result back to register `C`.

At this point, I need to make a modification to the preceding list. For reasons we'll discuss in detail when we talk about the instruction window in Chapter 5, most modern microprocessors treat sub-steps 3a and 3b as a group, while they treat step 3c, the register write, separately. To reflect this conceptual and architectural division, this list should be modified to look as follows:

1. *Fetch* the next instruction from the address stored in the program counter, and load that instruction into the instruction register. Increment the program counter.
2. *Decode* the instruction in the instruction register.
3. *Execute* the instruction in the instruction register. Because the instruction is not a branch instruction but an arithmetic instruction, send it to the ALU.
    a. *Read* the contents of registers `A` and `B`.
    b. *Add* the contents of `A` and `B`.
4. *Write* the result back to register `C`.

In a modern processor, these four steps are repeated over and over again until the program is finished executing. These are, in fact, the four stages in a classic RISC[1] pipeline. (I'll define the term *pipeline* shortly; for now, just think of a pipeline as a series of stages that each instruction in the code stream must pass through when the code stream is being executed.) Here are the four stages in their abbreviated form, the form in which you'll most often see them:

1. Fetch
2. Decode
3. Execute
4. Write (or "write-back")

Each of these stages could be said to represent one *phase* in the *lifecycle* of an instruction. An instruction starts out in the *fetch phase*, moves to the *decode phase*, then to the *execute phase*, and finally to the *write phase*. As I mentioned in "The Clock" on page 29, each phase takes a fixed, but by no means equal, amount of time. In most of the example processors with which you'll be working in this chapter, all four phases take an equal amount of time; this is not usually the case in real-world processors. In any case, if the DLW-1 takes exactly 1 nanosecond (ns) to complete each phase, then the DLW-1 can finish one instruction every 4 ns.

[1] The term *RISC* is an acronym for *Reduced Instruction Set Computing*. I'll cover this term in more detail in Chapter 5.

Basic Instruction Flow

One useful division that computer architects often employ when talking about CPUs is that of *front end* versus *back end*. As you already know, when instructions are fetched from main memory, they must be decoded for execution. This fetching and decoding takes place in the processor's front end.

You can see in Figure 3-1 that the front end roughly corresponds to the control and I/O units in the previous chapter's diagram of the DLW-1's programming model. The ALU and registers constitute the back end of the DLW-1. Instructions make their way from the front end down through the back end, where the work of number crunching gets done.

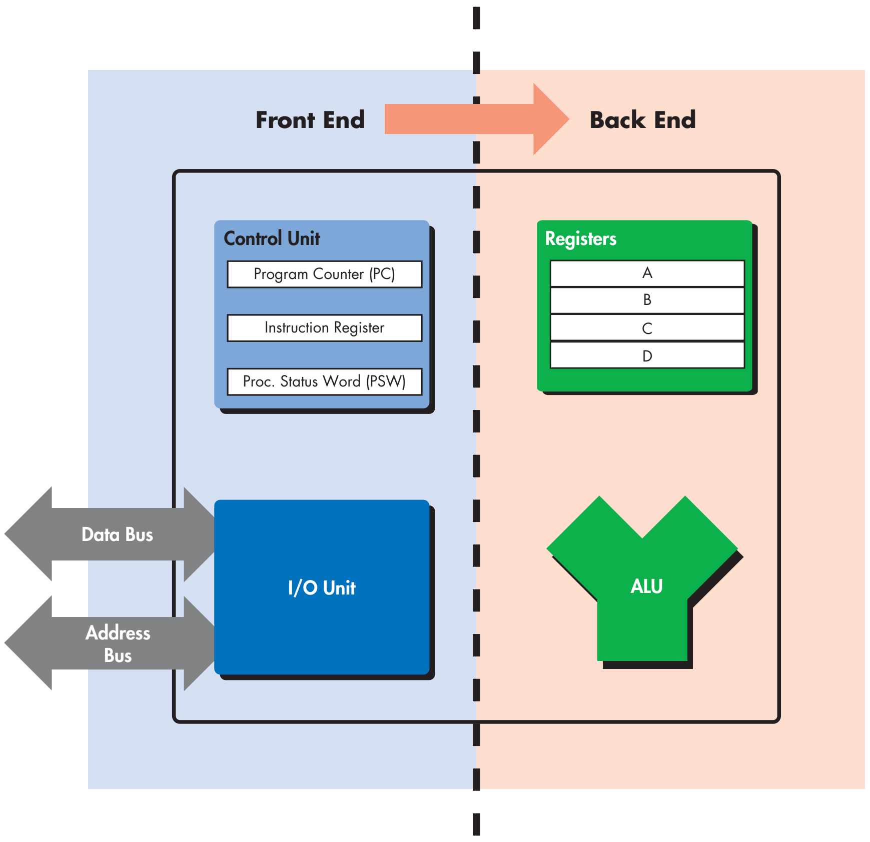


Figure 3-1: Front end versus back end

We can now modify Figure 1-4 to show all four phases of execution (see Figure 3-2).

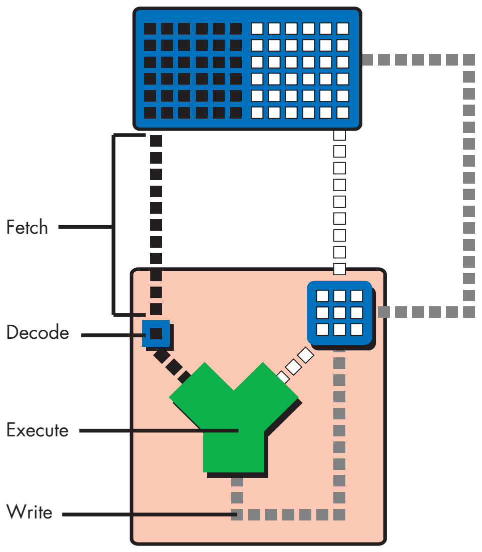


Figure 3-2: Four phases of execution

From here on out, we're going to focus primarily on the code stream, and more specifically, on how instructions enter and flow through the microprocessor, so the diagrams will need to leave out the data and results streams entirely. Figure 3-3 presents a microprocessor's basic instruction flow in a manner that's straightforward, yet easily elaborated upon.

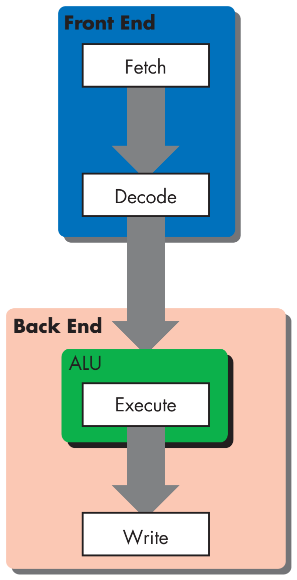


Figure 3-3: Basic instruction flow

In Figure 3-3, instructions flow from the front end's fetch and decode phases into the back end's execute and write phases. (Don't worry if this seems too simple. As the level of complexity of the architectures under discussion increases, so will the complexity of the diagrams.)

Pipelining Explained

Let's say my friends and I have decided to go into the automotive manufacturing business and that our first product is to be a sport utility vehicle (SUV). After some research, we determine that there are five stages in the SUV-building process:

Stage 1: Build the chassis.

Stage 2: Drop the engine into the chassis.

Stage 3: Put the doors, a hood, and coverings on the chassis.

Stage 4: Attach the wheels.

Stage 5: Paint the SUV.

Each of these stages requires the use of highly trained workers with very specialized skill sets—workers who are good at building chasses don't know much about engines, bodywork, wheels, or painting, and likewise for engine builders, painters, and the other crews. So when we make our first attempt to put together an SUV factory, we hire and train five crews of specialists, one for each stage of the SUV-building process. There's one crew to build the chassis, one to drop the engines, one to put the doors, hood, and coverings on the chassis, another for the wheels, and a painting crew. Finally, because the crews are so specialized and efficient, each stage of the SUV-building process takes a crew exactly one hour to complete.

Now, since my friends and I are computer types and not industrial engineers, we had a lot to learn about making efficient use of factory resources. We based the functioning of our first factory on the following plan: Place all five crews in a line on the factory floor, and have the first crew start an SUV at Stage 1. After Stage 1 is complete, the Stage 1 crew passes the partially finished SUV off to the Stage 2 crew and then hits the break room to play some foosball, while the Stage 2 crew builds the engine and drops it in. Once the Stage 2 crew is done, the SUV moves down to Stage 3, and the Stage 3 crew takes over, while the Stage 2 crew joins the Stage 1 crew in the break room.

The SUV moves on down the line through all five stages in this way, with only one crew working on one stage at any given time while the rest of the crews sit idle. Once the completed SUV finishes Stage 5, the crew at Stage 1 starts on another SUV. At this rate, it takes exactly five hours to finish a single SUV, and our factory completes one SUV every five hours.

In Figure 3-4, you can see the SUV pass through all five stages. The SUV enters the factory floor at the beginning of the first hour, where the Stage 1 crew begins work on it. Notice that all of the other crews are sitting idle while the Stage 1 crew does its work. At the beginning of the second hour, the Stage 2 crew takes over, and the other four crews sit idle while waiting on

Stage 2. This process continues as the SUV moves down the line, until at the beginning of the sixth hour, one SUV stands completed and while another has entered Stage 1.

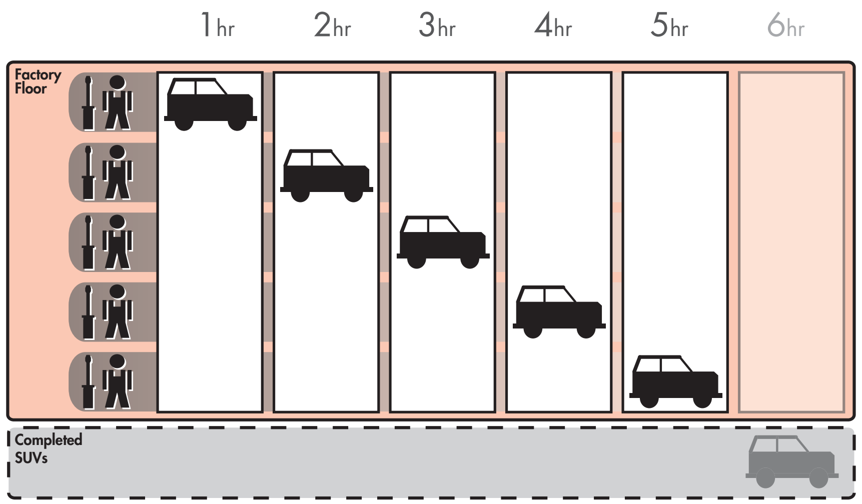


Figure 3-4: The lifecycle of an SUV in a non-pipelined factory

Fast-forward one year. Our SUV, the Extinction LE, is selling like . . . well, it's selling like an SUV, which means it's doing pretty well. In fact, our SUV is selling so well that we've attracted the attention of the military and have been offered a contract to provide SUVs to the U.S. Army on an ongoing basis. The Army likes to order multiple SUVs at a time; one order might come in for 10 SUVs, and another order might come in for 500 SUVs. The more of these orders that we can fill each fiscal year, the more money we can make during that same period and the better our balance sheet looks. This, of course, means that we need to find a way to increase the number of SUVs that our factory can complete per hour, known as our factory's *SUV completion rate*. By completing more SUVs per hour, we can fill the Army's orders faster and make more money each year.

The most intuitive way to go about increasing our factory's SUV completion rate is to try and decrease the production time of each SUV. If we can get the crews to work twice as fast, our factory can produce twice as many SUVs in the same amount of time. Our crews are already working as hard as they can, though, so unless there's a technological breakthrough that increases their productivity, this option is off the table for now.

Since we can't speed up our crews, we can always use the brute-force approach and just throw money at the problem by building a second assembly line. If we hire and train five new crews to form a second assembly line, also capable of producing one car every five hours, we can complete a grand total of two SUVs every five hours from the factory floor—double the SUV completion rate of our present factory. This doesn't seem like a very efficient use of factory resources, though, since not only do we have twice as many crews working at once but we also have twice as many crews in the break room at once. There has to be a better way.

Faced with a lack of options, we hire a team of consultants to figure out a clever way to increase overall factory productivity without either doubling the number of crews or increasing each individual crew's productivity. One year and thousands of billable hours later, the consultants hit upon a solution. Why let our crews spend four-fifths of their work day in the break room, when they could be doing useful work during that time? With proper scheduling of the existing five crews, our factory can complete *one SUV each hour*, thus drastically improving both the efficiency and the output of our assembly line. The revised workflow would look as follows:

1. The Stage 1 crew builds a chassis.
2. Once the chassis is complete, they send it on to the Stage 2 crew.
3. The Stage 2 crew receives the chassis and begins dropping the engine in, while the Stage 1 crew starts on a new chassis.
4. When both Stage 1 and Stage 2 crews are finished, the Stage 2 crew's work advances to Stage 3, the Stage 1 crew's work advances to Stage 2, and the Stage 1 crew starts on a new chassis.

Figure 3-5 illustrates this workflow in action. Notice that multiple crews have multiple SUVs simultaneously in progress on the factory floor. Compare this figure to Figure 3-4, where only one crew is active at a time and only one SUV is on the factory floor at a time.

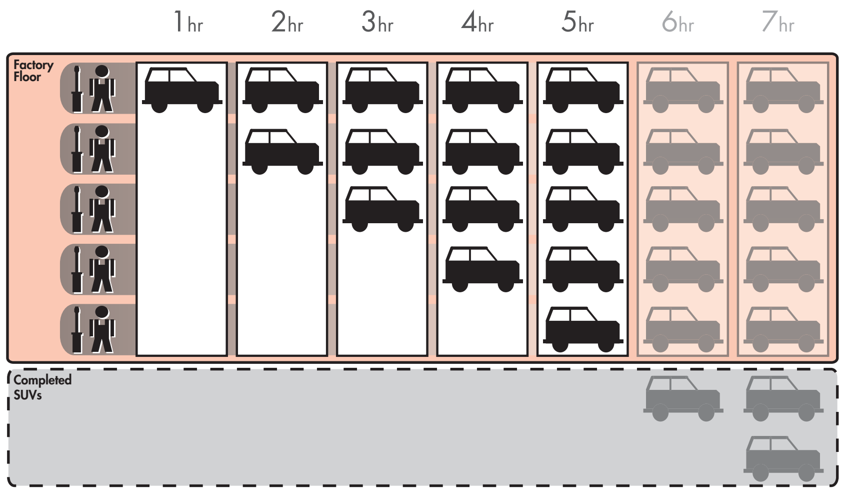


Figure 3-5: The lifecycle of an SUV in a pipelined factory

So as the assembly line begins to fill up with SUVs in various stages of production, more of the crews are put to work simultaneously until all of the crews are working on a different vehicle in a different stage of production. (Of course, this is how most of us nowadays in the post-Ford era expect a good, efficient assembly line to work.) If we can keep the assembly line full and keep all five crews working at once, we can produce one SUV every hour: a fivefold improvement in SUV completion rate over the previous completion rate of one SUV every five hours. That, in a nutshell, is pipelining.

While the total amount of time that each individual SUV spends in production has not changed from the original five hours, the rate at which the factory as a whole completes SUVs has increased drastically. Furthermore, the rate at which the factory can fulfill the Army's orders for batches of SUVs has increased drastically, as well. Pipelining works its magic by making optimal use of already existing resources. We don't need to speed up each individual stage of the production process, nor do we need to drastically increase the amount of resources that we throw at the problem; all that's necessary is that we get more work out of resources that are already there.

WHY THE SUV FACTORY?

The preceding discussion uses a factory analogy to explain pipelining. Other books use simpler analogies, like doing laundry, for instance, to explain this technique, but there are a few reasons why I chose a more elaborate and lengthy analogy to illustrate what is a relatively simple concept. First, I use factory analogies throughout this book, because assembly line-based factories are easy for readers to visualize and there's plenty of room for filling out the mental image in interesting ways in order to make a variety of related points. Second, and perhaps even more importantly, the many scheduling-, queuing- and resource management–related problems that factory designers face have direct analogies in computer architecture. In many cases, the problems and solutions are exactly the same, simply translated into a different domain. (Similar queuing-related problem/solution pairs also crop up in the service industry, which is why analogies involving supermarkets and fast food restaurants are also favorites of mine.)

Applying the Analogy

Bringing our discussion back to microprocessors, it should be easy to see how this concept applies to the four phases of an instruction's lifecycle. Just as the owners of the factory in our analogy wanted to increase the number of SUVs that the factory could finish in a given period of time, microprocessor designers are always looking for ways to increase the number of instructions that a CPU can complete in a given period of time. When you recall that a program is an ordered sequence of instructions, it becomes clear that increasing the number of instructions executed per unit time is one way to decrease the total amount of time that it takes to execute a program. (The other way to decrease a program's execution time is to decrease the number of instructions in the program, but this chapter won't address that approach until later.) In terms of our analogy, a program is like an order of SUVs from the military; just like increasing our factory's output of SUVs per hour enabled us to fill orders faster, increasing a processor's *instruction completion rate* (the number of instructions completed per unit time) enables it to run programs faster.

A Non-Pipelined Processor

The previous chapter briefly described how the simple processors described so far (e.g., the DLW-1) use the clock to time its internal operations. These non-pipelined processors work on one instruction at a time, moving each instruction through all four phases of its lifecycle during the course of one clock cycle. Thus non-pipelined processors are also called *single-cycle* processors, because all instructions take exactly one clock cycle to execute fully (i.e., to pass through all four phases of their lifecycles).

Because the processor completes instructions at a rate of one per clock cycle, you want the CPU's clock to run as fast as possible so that the processor's instruction completion rate can be as high as possible.

Thus you need to calculate the maximum amount of time that it takes to complete an instruction and make the clock cycle time equivalent to that length of time. It just so happens that on the hypothetical example CPU, the four phases of the instruction's lifecycle take a total of 4 ns to complete. Therefore, you should set the duration of the CPU clock cycle to 4 ns, so that the CPU can complete the instruction's lifecycle—from *fetch* to *write-back*—in a single clock. (A CPU clock cycle is often just called a *clock* for short.)

In Figure 3-6, the blue instruction leaves the code storage area, enters the processor, and then advances through the phases of its lifecycle over the course of the 4 ns clock period, until at the end of the fourth nanosecond, it completes the last phase and its lifecycle is over. The end of the fourth nanosecond is also the end of the first clock cycle, so now that the first clock cycle is finished and the blue instruction has completed its execution, the red instruction can enter the processor at the start of a new clock cycle and go through the same process. This 4 ns sequence of steps is repeated until, after a total of 16 ns (or four clock cycles), the processor has completed all four instructions at a completion rate of 0.25 instructions/ns (= 4 instructions/16 ns).

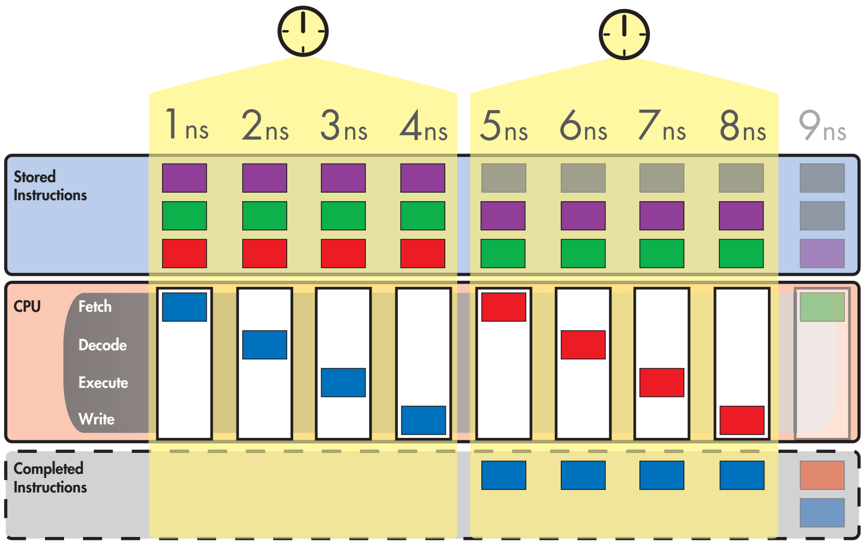


Figure 3-6: A single-cycle processor

Single-cycle processors like the one in Figure 3-6 are simple to design, but they waste a lot of hardware resources. All of that white space in the diagram represents processor hardware that's sitting idle while it waits for the instruction that's currently in the processor to finish executing. By pipelining the processor in this figure, you can put more of that hardware to work every nanosecond, thereby increasing the processor's efficiency and its performance on executing programs.

Before moving on, I should clarify a few concepts illustrated in Figure 3-6. At the bottom is a region labeled "Completed Instructions." Completed instructions don't actually go anywhere when they're finished executing; once they've done their job of telling the processor how to modify the data stream, they're simply deleted from the processor. So the "Completed Instructions" box does not represent a real part of the computer, which is why I've placed a dotted line around it. This area is just a place for you to keep track of how many instructions the processor has completed in a certain amount of time, or the processor's *instruction completion rate* (or *completion rate*, for short), so that when you compare different types of processors, you'll have a place where you can quickly see which processor performs better. The more instructions a processor completes in a set amount of time, the better it performs on programs, which are an ordered sequence of instructions. Think of the "Completed Instructions" box as a sort of scoreboard for tracking each processor's completion rate, and check the box in each of the subsequent figures to see how long it takes for the processor to populate this box.

Following on the preceding point, you may be curious as to why the blue instruction that has completed in the fourth nanosecond does not appear in the "Completed Instructions" box until the fifth nanosecond. The reason is straightforward and stems from the nature of the diagram. Because an instruction spends *one complete nanosecond*, from start to finish, in each stage of execution, the blue instruction enters the write phase at the *beginning* of the fourth nanosecond and exits the write phase at the *end* of the fourth nanosecond. This means that the fifth nanosecond is the first full nanosecond in which the blue instruction stands completed. Thus at the beginning of the fifth nanosecond (which coincides with the end of the fourth nanosecond), the processor has completed one instruction.

A Pipelined Processor

Pipelining a processor means breaking down its instruction execution process—what I've been calling the instruction's *lifecycle*—into a series of discrete *pipeline stages* that can be completed in sequence by specialized hardware. Recall the way that we broke down the SUV assembly process into five discrete steps—with one dedicated crew assigned to complete each step—and you'll get the idea.

Because an instruction's lifecycle consists of four fairly distinct phases, you can start by breaking down the single-cycle processor's instruction execution process into a sequence of four discrete pipeline stages, where each pipeline stage corresponds to a phase in the standard instruction lifecycle:

Stage 1: Fetch the instruction from code storage.

Stage 2: Decode the instruction.

Stage 3: Execute the instruction.

Stage 4: Write the results of the instruction back to the register file.

Note that the number of pipeline stages is called the *pipeline depth*. So the four-stage pipeline has a pipeline depth of four.

For convenience's sake, let's say that each of these four pipeline stages takes exactly 1 ns to finish its work on an instruction, just like each crew in our assembly line analogy takes one hour to finish its portion of the work on an SUV. So the original single-cycle processor's 4 ns execution process is now broken down into four discrete, sequential pipeline stages of 1 ns each in length.

Now let's step through another diagram together to see how a pipelined CPU would execute the four instructions depicted in Figure 3-7.

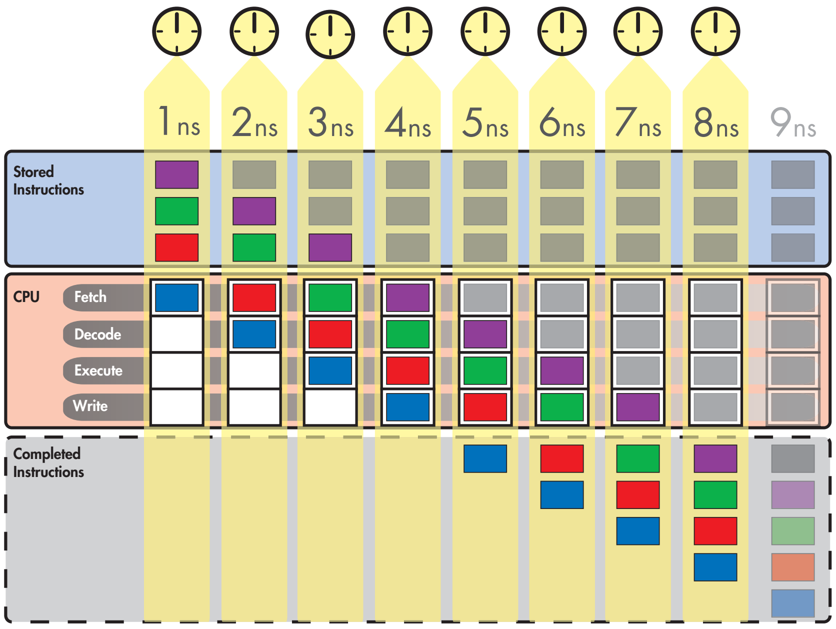


Figure 3-7: A four-stage pipeline

At the beginning of the first nanosecond, the blue instruction enters the fetch stage. After that nanosecond is complete, the second nanosecond begins and the blue instruction moves on to the decode stage, while the next instruction, the red one, starts to make its way from code storage to the processor (i.e., it enters the fetch stage). At the start of the third nanosecond, the blue instruction advances to the execute stage, the red instruction advances to the decode stage, and the green instruction enters the fetch stage. At the fourth nanosecond, the blue instruction advances to the write stage, the red instruction advances to the execute stage, the green instruction advances to the decode stage, and the purple instruction advances to the fetch stage. After the fourth nanosecond has fully elapsed and the fifth nanosecond starts, the blue instruction has passed from the pipeline and is now finished executing. Thus we can say that at the end of 4 ns (= four clock cycles), the pipelined processor depicted in Figure 3-7 has completed one instruction.

At start of the fifth nanosecond, the pipeline is now full and the processor can begin completing instructions at a rate of one instruction per nanosecond. This one instruction/ns completion rate is a fourfold improvement over the single-cycle processor's completion rate of 0.25 instructions/ns (or four instructions every 16 ns).

Shrinking the Clock

You can see from Figure 3-7 that the role of the CPU clock changes slightly in a pipelined processor, compared to the single-cycle processor shown in Figure 3-6. Because all of the pipeline stages must now work together simultaneously and be ready at the start of each new nanosecond to hand over the results of their work to the next pipeline stage, the clock is needed to coordinate the activity of the whole pipeline. The way this is done is simple: Shrink the clock cycle time to match the time it takes each stage to complete its work so that at the start of each clock cycle, each pipeline stage hands off the instruction it was working on to the next stage in the pipeline. Because each pipeline stage in the example processor takes 1 ns to complete its work, you can set the clock cycle to be 1 ns in duration.

This new method of clocking the processor means that a new instruction will not necessarily be completed at the close of each clock cycle, as was the case with the single-cycle processor. Instead, a new instruction will be completed at the close of only those clock cycles in which the write stage has been working on an instruction. Any clock cycle with an empty write stage will add no new instructions to the "Completed Instructions" box, and any clock cycle with an active write stage will add one new instruction to the box. Of course, this means that when the pipeline first starts to work on a program, there will be a few clock cycles—three to be exact—during which no instructions are completed. But once the fourth clock cycle starts, the first instruction enters the write stage and the pipeline can then begin completing new instructions on each clock cycle, which, because each clock cycle is 1 ns, translates into a completion rate of one instruction per nanosecond.

Shrinking Program Execution Time

Note that the total execution time for each individual instruction is not changed by pipelining. It still takes an instruction 4 ns to make it all the way through the processor; that 4 ns can be split up into four clock cycles of 1 ns each, or it can cover one longer clock cycle, but it's still the same 4 ns. Thus pipelining doesn't speed up instruction execution time, but it does speed up *program execution time* (the number of nanoseconds that it takes to execute an entire program) by increasing the number of instructions finished per unit of time. Just like pipelining our hypothetical SUV assembly line allowed us to fill the Army's orders in a shorter span of time, even though each individual SUV still spent a total of five hours in the assembly line, so does pipelining allow a processor to execute programs in a shorter amount of time, even though each individual instruction still spends the same amount of time traveling through the CPU. Pipelining makes more efficient use of the CPU's existing resources by putting all of its units to work simultaneously, thereby allowing it to do more total work each nanosecond.

The Speedup from Pipelining

In general, the speedup in completion rate versus a single-cycle implementation that's gained from pipelining is ideally equal to the number of pipeline stages. A four-stage pipeline yields a fourfold speedup in the completion rate versus a single-cycle pipeline, a five-stage pipeline yields a fivefold speedup, a twelve-stage pipeline yields a twelvefold speedup, and so on. This speedup is possible because the more pipeline stages there are in a processor, the more instructions the processor can work on simultaneously, and the more instructions it can complete in a given period of time. So the more finely you can slice those four phases of the instruction's lifecycle, the more of the hardware that's used to implement those phases you can put to work at any given moment.

To return to our assembly line analogy, let's say that each crew is made up of six workers, and that each of the hour-long tasks that each crew performs can be readily subdivided into two shorter, 30-minute tasks. So we can double our factory's throughput by splitting each crew into two smaller, more specialized crews of three workers each, and then having each smaller crew perform one of the shorter tasks on one SUV per 30 minutes.

Stage 1: Build the chassis.

- **Crew 1a:** Fit the parts of the chassis together and spot-weld the joints.
- **Crew 1b:** Fully weld all the parts of the chassis.

Stage 2: Drop the engine into the chassis.

- **Crew 2a:** Place the engine into the chassis and mount it in place.
- **Crew 2b:** Connect the engine to the moving parts of the car.

Stage 3: Put the doors, a hood, and coverings on the chassis.

- **Crew 3a:** Put the doors and hood on the chassis.
- **Crew 3b:** Put the other coverings on the chassis.

Stage 4: Attach the wheels.

- **Crew 4a:** Attach the two front wheels.
- **Crew 4b:** Attach the two rear wheels.

Stage 5: Paint the SUV.

- **Crew 5a:** Paint the sides of the SUV.
- **Crew 5b:** Paint the top of the SUV.

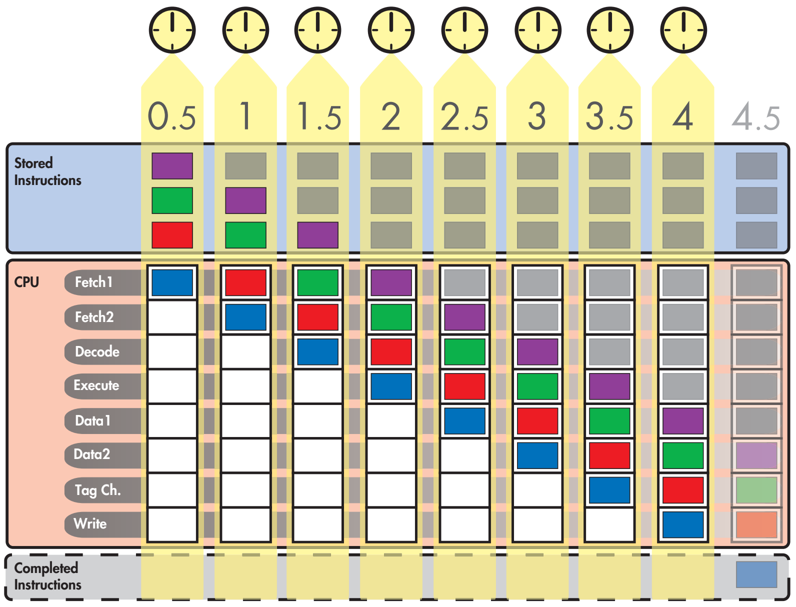
After the modifications described here, the 10 smaller crews in our factory now have a collective total of 10 SUVs in progress during the course of any given 30-minute period. Furthermore, our factory can now complete a new SUV every 30 minutes, a tenfold improvement over our first factory's completion rate of one SUV every five hours. So by pipelining our assembly line even more deeply, we've put even more of its workers to work concurrently, thereby increasing the number of SUVs that can be worked on simultaneously and increasing the number of SUVs that can be completed within a given period of time.

Deepening the pipeline of the four-stage processor works on similar principles and has a similar effect on completion rates. Just as the five stages in our SUV assembly line could be broken down further into a longer sequence of more specialized stages, the execution process that each instruction goes through can be broken down into a series of much more than just four discrete stages. By breaking the processor's four-stage pipeline down into a longer series of shorter, more specialized stages, even more of the processor's specialized hardware can work simultaneously on more instructions and thereby increase the number of instructions that the pipeline completes each nanosecond.

We first moved from a single-cycle processor to a pipelined processor by taking the 4 ns time period that the instruction spent traveling through the processor and slicing it into four discrete pipeline stages of 1 ns each in length. These four discrete pipeline stages corresponded to the four phases of an instruction's lifecycle. A processor's pipeline stages aren't always going to correspond exactly to the four phases of a processor's lifecycle, though. Some processors have a five-stage pipeline, some have a six-stage pipeline, and many have pipelines deeper than 10 or 20 stages. In such cases, the CPU designer must slice up the instruction's lifecycle into the desired number of stages in such a way that all the stages are equal in length.

Now let's take that 4 ns execution process and slice it into eight discrete stages. Because all eight pipeline stages must be of exactly the same duration for pipelining to work, the eight pipeline stages must each be 4 ns ÷ 8 = 0.5 ns in length. Since we're presently working with an idealized example, let's pretend that splitting up the processor's four-phase lifecycle into eight equally long (0.5 ns) pipeline stages is a trivial matter, and that the results look like what you see in Figure 3-8. (In reality, this task is not trivial and involves a number of trade-offs. As a concession to that reality, I've chosen to use the eight stages of a real-world pipeline—the MIPS pipeline—in Figure 3-8, instead of just splitting each of the four traditional stages in two.)

Because pipelining requires that each pipeline stage take exactly one clock cycle to complete, the clock cycle can now be shortened to 0.5 ns in order to fit the lengths of the eight pipeline stages. At the bottom of Figure 3-8, you can see the impact that this increased number of pipeline stages has on the number of instructions completed per unit time.



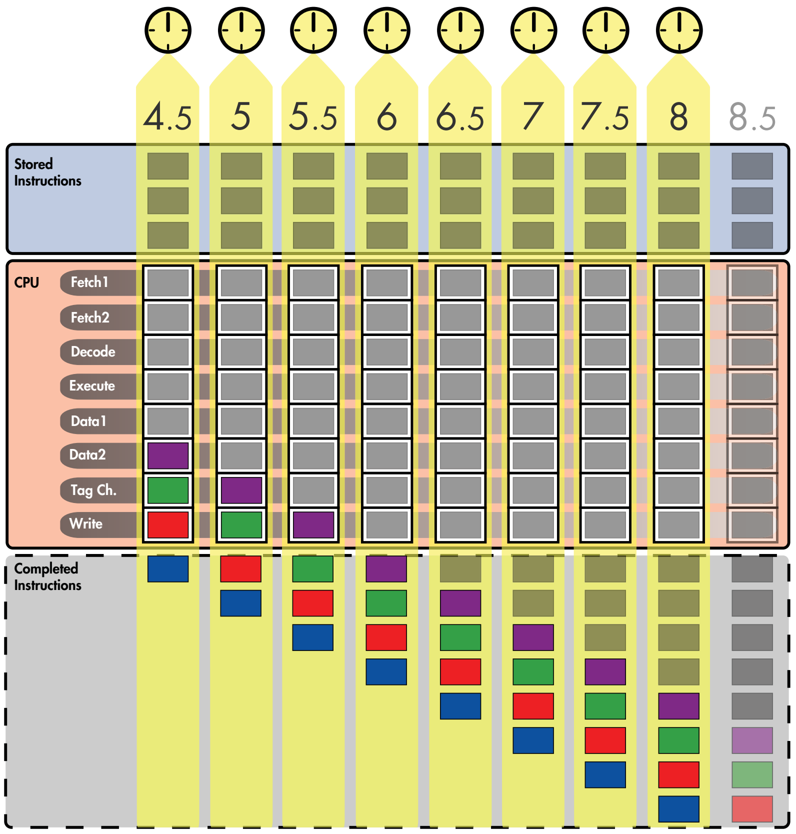


Figure 3-8: An eight-stage pipeline

The single-cycle processor can complete one instruction every 4 ns, for a completion rate of 0.25 instructions/ns, and the four-stage pipelined processor can complete one instruction every nanosecond for a completion rate of one instructions/ns. The eight-stage processor depicted in Figure 3-8 improves on both of these by completing one instruction every 0.5 ns, for a completion rate of two instructions/ns. Note that because each instruction still takes 4 ns to execute, the first 4 ns of the eight-stage processor are still dedicated to filling up the pipeline. But once the pipeline is full, the processor can begin completing instructions twice as fast as the four-stage processor and eight times as fast as the single-stage processor.

This eightfold increase in completion rate versus a single-cycle design means that the eight-stage processor can execute programs much faster than either a single-cycle or a four-stage processor. But does the eightfold increase in completion rate translate into an eightfold increase in processor performance? Not exactly.

Program Execution Time and Completion Rate

If the program that the single-cycle processor in Figure 3-6 is running consisted of only the four instructions depicted, that program would have a *program execution time* of 16 ns, or 4 instructions ÷ 0.25 instructions/ns. If the program consisted of, say, seven instructions, it would have a program execution time of 7 instructions ÷ 0.25 instructions/ns = 28 ns. In general, a program's execution time is equal to the total number of instructions in the program divided by the processor's instruction completion rate (number of instructions completed per nanosecond), as in the following equation:

```
program execution time = number of instructions in program / instruction
completion rate
```

Most of the time, when I talk about processor *performance* in this book, I'm talking about program execution time. One processor performs better than another if it executes all of a program's instructions in a shorter amount of time, so reducing program execution time is the key to increasing processor performance.

From the preceding equation, it should be clear that program execution time can be reduced in one of two ways: by a reduction in the number of instructions per program or by an increase in the processor's completion rate. For now, let's assume that the number of instructions in a program is fixed and that there's nothing that can be done about this term of the equation. As such, our focus in this chapter will be on increasing instruction completion rates.

In the case of a non-pipelined, single-cycle processor, the instruction completion rate (*x* instructions per 1 ns) is simply the inverse of the instruction execution time (*y* ns per 1 instruction), where *x* and *y* have different numerical values. Because the relationship between completion rate and instruction execution time is simple and direct in a single-cycle processor, an *n*fold improvement in one is an *n*fold improvement in the other. So improving the performance of a single-cycle processor is really about reducing instruction execution times.

With pipelined processors, the relationship between instruction execution time and completion rate is more complex. As discussed previously, pipelined processors allow you to increase the processor's completion rate without altering the instruction execution time. Of course, a reduction in instruction execution time still translates into a completion rate improvement, but the reverse is not necessarily true. In fact, as you'll learn later on, pipelining's improvements to completion rate often come at the price of *increased* instruction execution times. This means that for pipelining to improve performance, the processor's completion rate must be as high as possible over the course of a program's execution.

The Relationship Between Completion Rate and Program Execution Time

If you look at the "Completed Instructions" box of the four-stage processor back in Figure 3-7, you'll see that a total of five instructions have been completed at the start of the ninth nanosecond. In contrast, the non-pipelined processor illustrated in Figure 3-6 sports two completed instructions at the start of the ninth nanosecond. Five completed instructions in the span of 8 ns is obviously not a fourfold improvement over two completed instructions in the same time period, so what gives?

Remember that it took the pipelined processor 4 ns initially to fill up with instructions; the pipelined processor did not complete its first instruction until the end of the fourth nanosecond. Therefore, it completed fewer instructions over the first 8 ns of that program's execution than it would have had the pipeline been full for the entire 8 ns.

When the processor is executing programs that consist of thousands of instructions, then as the number of nanoseconds stretches into the thousands, the impact on program execution time of those four initial nanoseconds, during which only one instruction was completed, begins to vanish and the pipelined processor's advantage begins to approach the fourfold mark. For example, after 1,000 ns, the non-pipelined processor will have completed 250 instructions (1000 ns ÷ 0.25 instructions/ns = 250 instructions), while the pipelined processor will have completed 996 instructions [(1000 ns – 4 ns) ÷ 1 instructions/ns]—a 3.984-fold improvement.

What I've just described using this concrete example is the difference between a pipeline's *maximum theoretical completion rate* and its real-world *average completion rate*. In the previous example, the four-stage processor's maximum theoretical completion rate, i.e., its completion rate on cycles when its entire pipeline is full, is one instruction/ns. However, the processor's average completion rate during its first 8 ns is 5 instructions/8 ns = 0.625 instructions/ns. The processor's average completion rate improves as it passes more clock cycles with its pipeline full, until at 1,000 ns, its average completion rate is 996 instructions/1000 ns = 0.996 instructions/ns.

At this point, it might help to look at a graph of the four-stage pipeline's average completion rate as the number of nanoseconds increases, illustrated in Figure 3-9.

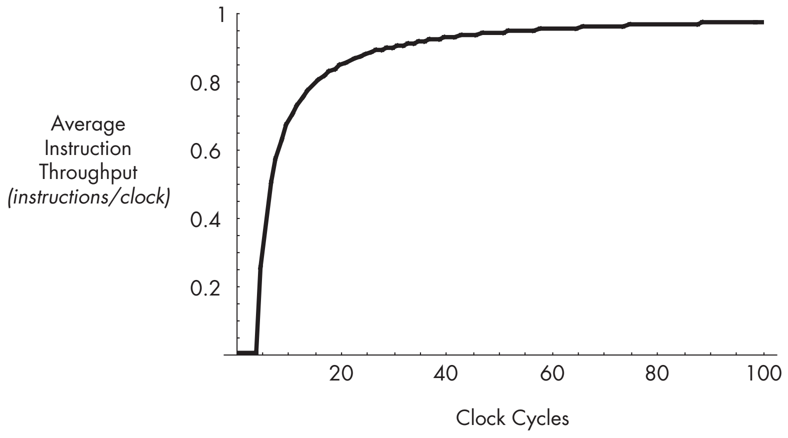


Figure 3-9: Average completion rate of a four-stage pipeline

You can see how the processor's average completion rate stays at zero until the 4 ns mark, after which point the pipeline is full and the processor can begin completing a new instruction on each nanosecond, causing the average completion rate for the entire program to curve upward and eventually to approach the maximum completion rate of one instruction/ns.

So in conclusion, a pipelined processor can only approach its ideal completion rate if it can go for long stretches with its pipeline full on every clock cycle.

Instruction Throughput and Pipeline Stalls

Pipelining isn't totally "free," however. Pipelining adds some complexity to the microprocessor's control logic, because all of these stages have to be kept in sync. Even more important for the present discussion, though, is the fact that pipelining adds some complexity to the ways in which you assess the processor's performance.

Instruction Throughput

Up until now, we've talked about microprocessor performance mainly in terms of instruction completion rate, or the number of instructions that the processor's pipeline can complete each nanosecond. A more common performance metric in the real world is a pipeline's *instruction throughput*, or the number of instructions that the processor completes *each clock cycle*. You might be thinking that a pipeline's instruction throughput should always be one instruction/clock, because I stated previously that a pipelined processor completes a new instruction at the end of each clock cycle *in which the write stage has been active*. But notice how the emphasized part of that definition qualifies it a bit; you've already seen that the write stage is inactive during clock cycles in which the pipeline is being filled, so on those clock cycles, the processor's instruction throughput is 0 instructions/clock. In contrast, when the instruction's pipeline is full and the write stage is active, the pipelined processor has an instruction throughput of 1 instruction/clock.

So just like there was a difference between a processor's maximum theoretical completion rate and its average completion rate, there's also a difference between a processor's maximum theoretical instruction throughput and its average instruction throughput:

Instruction throughput
The number of instructions that the processor finishes executing on each clock cycle. You'll also see instruction throughput referred to as instructions per clock (IPC).

Maximum theoretical instruction throughput
The theoretical maximum number of instructions that the processor can finish executing on each clock cycle. For the simple kinds of pipelined and non-pipelined processors described so far, this number is always one instruction per cycle (one instruction/clock or one IPC).

Average instruction throughput
The average number of instructions per clock (IPC) that the processor has actually completed over a certain number of cycles.

A processor's instruction throughput is closely tied to its instruction completion rate—the more instructions that the processor completes each clock cycle (instructions/clock), the more instructions it also completes over a given period of time (instructions/ns).

We'll talk more about the relationship between these two metrics in a moment, but for now just remember that a higher instruction throughput translates into a higher instruction completion rate, and hence better performance.

Pipeline Stalls

In the real world, a processor's pipeline can be found in more conditions than just the two described so far—i.e., a full pipeline or a pipeline that's being filled. Sometimes, instructions get hung up in one pipeline stage for multiple cycles. There are a number of reasons why this might happen—we'll discuss many of them throughout this book—but when it happens, the pipeline is said to *stall*. When the pipeline stalls, or gets hung in a certain stage, all of the instructions in the stages below the one where the stall happened continue advancing normally, while the stalled instruction just sits in its stage, and all the instructions behind it back up.

In Figure 3-10, the orange instruction is stalled for two extra cycles in the fetch stage. Because the instruction is stalled, a new gap opens ahead of it in the pipeline for each cycle that it stalls. Once the instruction starts advancing through the pipeline again, the gaps in the pipeline that were created by the stall—gaps that are commonly called "pipeline bubbles"—travel down the pipeline ahead of the formerly stalled instruction until they eventually leave the pipeline.

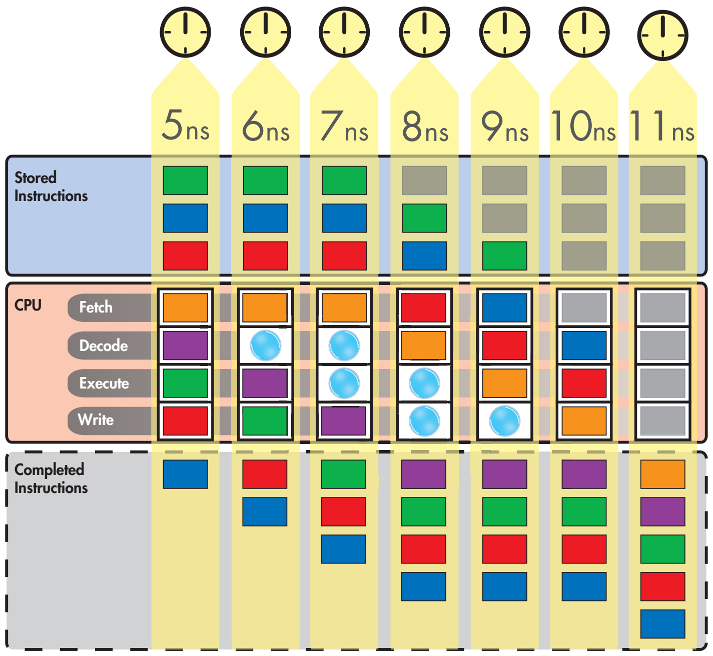


Figure 3-10: Pipeline stalls in a four-stage pipeline would look different without the effect of the "bubbles."

Pipeline stalls—or bubbles—reduce a pipeline's average instruction throughput, because they prevent the pipeline from attaining the maximum throughput of one finished instruction per cycle. In Figure 3-10, the orange instruction has stalled in the fetch stage for two extra cycles, creating two bubbles that will propagate through the pipeline. (Again, the bubble is simply a way of signifying that the pipeline stage in which the bubble sits is doing no work during that cycle.) Once the instructions below the bubble have completed, the processor will complete no new instructions until the bubbles move out of the pipeline. So at the ends of clock cycles 9 and 10, no new instructions are added to the "Completed Instructions" region; normally, two new instructions would be added to the region at the ends of these two cycles. Because of the bubbles, though, the processor is two instructions behind schedule when it hits the 11th clock cycle and begins racking up completed instructions again.

The more of these bubbles that crop up in the pipeline, the farther away the processor's actual instruction throughput is from its maximum instruction throughput. In the preceding example, the processor should ideally have completed seven instructions by the time it finishes the 10th clock cycle, for an average instruction throughput of 0.7 instructions per clock. (Remember, the maximum instruction throughput possible under ideal conditions is one instruction per clock, but many more cycles with no bubbles would be needed to approach that maximum.) But because of the pipeline stall, the processor only completes five instructions in 10 clocks, for an average instruction throughput of 0.5 instructions per clock. 0.5 instructions per clock is half the theoretical maximum instruction throughput, but of course the processor spent a few clocks filling the pipeline, so it couldn't have achieved that after 10 clocks, even under ideal conditions. More important is the fact that 0.5 instructions per clock is only 71 percent of the throughput that it could have achieved were there no stall (i.e., 0.7 instructions per clock). Because pipeline stalls decrease the processor's average instruction throughput, they increase the amount of time that it takes to execute the currently running program. If the program in the preceding example consisted of only the seven instructions pictured, then the pipeline stall would have resulted in a 29 percent program execution time increase.

Look at the graph in Figure 3-11; it shows what that two-cycle stall does to the average instruction throughput.

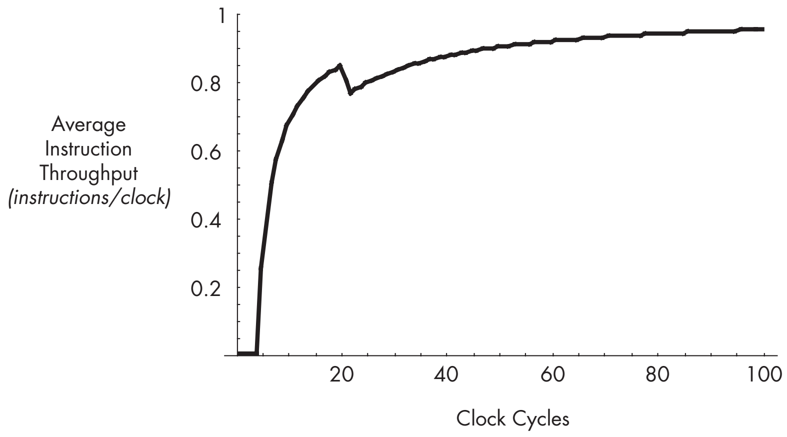


Figure 3-11: Average instruction throughput of a four-stage pipeline with a two-cycle stall

The processor's average instruction throughput stops rising and begins to plummet when the first bubble hits the write stage, and it doesn't recover until the bubbles have left the pipeline.

To get an even better picture of the impact that stalls can have on a pipeline's average instruction throughput, let's now look at the impact that a stall of 10 cycles (starting in the fetch stage of the 18th cycle) would have over the course of 100 cycles in the four-stage pipeline described so far. Look at the graph in Figure 3-12.

After the first bubble of the stall hits the write stage in the 20th clock, the average instruction throughput stops increasing and begins to decrease. For each clock in which there's a bubble in the write stage, the pipeline's instruction throughput is 0 instructions/clock, so its average instruction throughput for the whole period continues to decline. After the last bubble has worked its way out of the write stage, then the pipeline begins completing new instructions again at a rate of one instruction/cycle and its average instruction throughput begins to climb. And when the processor's instruction throughput begins to climb, so does its completion rate and its performance on programs.

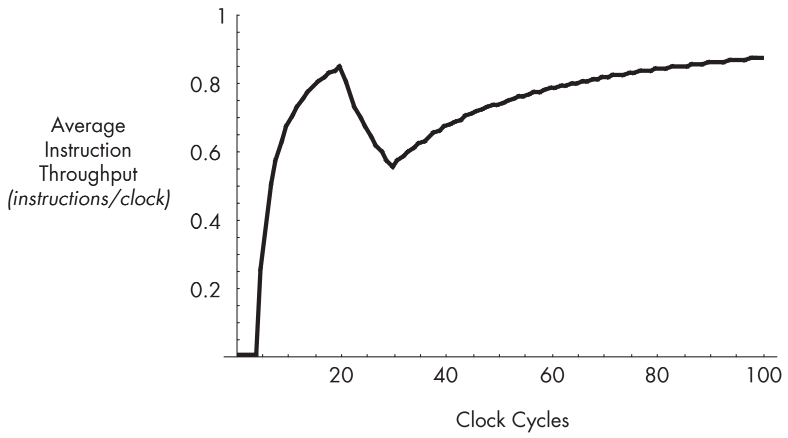


Figure 3-12: Average instruction throughput of a four-stage pipeline with a 10-cycle stall

Now, 10 or 20 cycles worth of stalls here and there may not seem like much, but they do add up. Even more important, though, is the fact that the numbers in the preceding examples would be increased by a factor of 30 or more in real-world execution scenarios. As of this writing, a processor can spend from 50 to 120 nanoseconds waiting on data from main memory. For a 3 GHz processor that has a clock cycle time of a fraction of a nanosecond, a 100 ns main memory access translates into a few thousand clock cycles worth of bubbles—and that's just for one main memory access out of the many millions that a program might make over the course of its execution.

In later chapters, we'll look at the causes of pipeline stalls and the many tricks that computer architects use to overcome them.

Instruction Latency and Pipeline Stalls

Before closing out our discussion of pipeline stalls, I should introduce another term that you'll be seeing periodically throughout the rest of the book: *instruction latency*. An instruction's latency is the number of clock cycles it takes for the instruction to pass through the pipeline. For a single-cycle processor, all instructions have a latency of one clock cycle. In contrast, for the simple four-stage pipeline described so far, all instructions have a latency of four cycles. To get a visual image of this, take one more look at the blue instruction in Figure 3-6 earlier in this chapter; this instruction takes four clock cycles to advance, at a rate of one clock cycle per stage, through each of the four stages of the pipeline. Likewise, instructions have a latency of eight cycles on an eight-stage pipeline, 12 cycles on a 12-stage pipeline, and so on.

In real-world processors, instruction latency is not necessarily a fixed number that's equal to the number of pipeline stages. Because instructions can get hung up in one or more pipeline stages for multiple cycles, each extra cycle that they spend waiting in a pipeline stage adds one more cycle to their latency. So the instruction latencies given in the previous paragraph (i.e., four cycles for a four-stage pipeline, eight cycles for an eight-stage pipeline, and so on) represent *minimum* instruction latencies. Actual instruction latencies in pipelines of any length can be longer than the depth of the pipeline, depending on whether or not the instruction stalls in one or more stages.

Limits to Pipelining

As you can probably guess, there are some practical limits to how deeply you can pipeline an assembly line or a processor before the actual speedup in completion rate that you gain from pipelining starts to become significantly less than the ideal speedup that you might expect. In the real world, the different phases of an instruction's lifecycle don't easily break down into an arbitrarily high number of shorter stages of perfectly equal duration. Some stages are inherently more complex and take longer than others.

But because each pipeline stage must take exactly one clock cycle to complete, the clock pulse that coordinates all the stages can be no faster than the pipeline's slowest stage. In other words, the amount of time it takes for the slowest stage in the pipeline to complete will determine the length of the CPU's clock cycle and thus the length of every pipeline stage. This means that the pipeline's slowest stage will spend the entire clock cycle working, while the faster stages will spend part of the clock cycle idle. Not only does this waste resources, but it increases each instruction's overall execution time by dragging out some phases of the lifecycle to take up more time than they would if the processor was not pipelined—all of the other stages must wait a little extra time each cycle while the slowest stage plays catch-up.

So, as you slice the pipeline more finely in order to add stages and increase throughput, the individual stages get less and less uniform in length and complexity, with the result that the processor's overall instruction execution time gets longer. Because of this feature of pipelining, one of the most difficult and important challenges that the CPU designer faces is that of balancing the pipeline so that no one stage has to do more work to do than any other. The designer must distribute the work of processing an instruction evenly to each stage, so that no one stage takes up too much time and thus slows down the entire pipeline.

Clock Period and Completion Rate

If the pipelined processor's clock cycle time, or *clock period*, is longer than its ideal length (i.e., non-pipelined instruction execution time/pipeline depth), and it always is, then the processor's completion rate will suffer. If the instruction throughput stays fixed at, say, one instruction/clock, then as the clock period increases, the completion rate decreases. Because new instructions can be completed only at the end of each clock cycle, a longer clock cycle translates into fewer instructions completed per nanosecond, which in turn translates into longer program execution times.

To get a better feel for the relationship between completion rate, instruction throughput, and clock cycle time, let's take the eight-stage pipeline from Figure 3-8 and increase its clock cycle time to 1 ns instead of 0.5 ns. Its first 9 ns of execution would then look as in Figure 3-13.

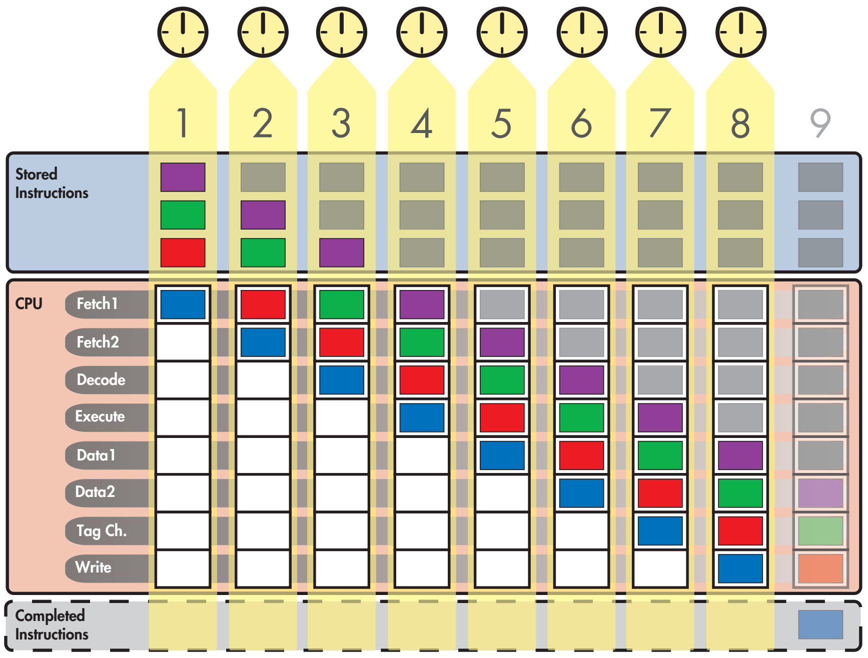


Figure 3-13: An eight-stage pipeline with a 1 ns clock period

As you can see, the instruction execution time has now increased from an original time of 4 ns to a new time of 8 ns, which means that the eight-stage pipeline does not complete its first instruction until the end of the eighth nanosecond. Once the pipeline is full, the processor pictured in Figure 3-13 begins completing instructions at a rate of one instruction per nanosecond. This completion rate is half the completion rate of the ideal eight-stage pipeline with the 0.5 ns clock cycle time. It's also the exact same completion rate as the one instruction/ns completion rate of the ideal four-stage pipeline. In short, the longer clock cycle time of the new eight-stage pipeline has robbed the deeper pipeline of its completion rate advantage. Furthermore, the eight-stage pipeline now takes twice as long to fill.

Take a look at the graph in Figure 3-14 to see what this doubled execution time does to the eight-stage pipeline's average completion rate curve versus the same curve for a four-stage pipeline.

It takes longer for the slower eight-stage pipeline to fill up, which means that its average completion rate—and hence its performance—ramps up more slowly when the pipeline is first filled with instructions. There are many situations in which a processor that's executing a program must flush its pipeline entirely and then begin refilling it from a different point in the code stream. In such instances, that slower-ramping completion rate curve causes a performance hit.

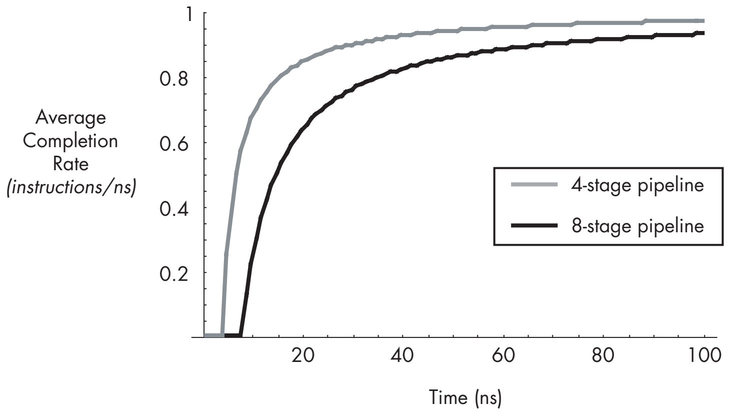


Figure 3-14: Average instruction completion rate for four- and eight-stage pipelines with a 1 ns clock period

In the end, the performance gains brought about by pipelining depend on two things:

1. Pipeline stalls must be avoided. As you've seen earlier, pipeline stalls cause the processor's completion rate and performance to drop. Main memory accesses are a major cause of pipeline stalls, but this problem can be alleviated significantly by the use of caching. We'll cover caching in detail in Chapter 11. Other causes of stalls, like the various types of hazards, will be covered at the end of Chapter 4.
2. Pipeline refills must be avoided. Flushing the pipeline and refilling it again takes a serious toll on both completion rate and performance. Once the pipeline is full, it must remain full for as long as possible if the average completion rate is to be kept up.

When we look more closely at real-world pipelined processors in later chapters, you'll see these two issues come up again and again. In fact, much of the rest of the book will be about how the different architectures under discussion work to keep their pipelines full by preventing stalls and ensuring a continuous and uninterrupted flow of instructions into the processor from the code storage area.

The Cost of Pipelining

In addition to the inherent limits to performance improvement that we've just looked at, pipelining requires a nontrivial amount of extra bookkeeping and buffering logic to implement, so it incurs an overhead cost in transistors and die space. Furthermore, this overhead cost increases with pipeline depth, so that a processor with a very deep pipeline (for example, Intel's Pentium 4) spends a significant amount of its transistor budget on pipeline-related logic. These overhead costs place some practical constraints on how deeply you can pipeline a processor. I'll say a lot more about such constraints in the chapters covering the Pentium line and the Pentium 4.

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*.

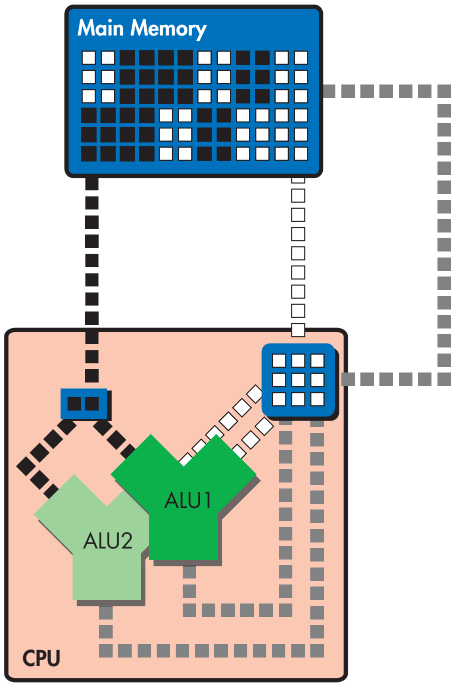


Figure 4-1: The superscalar DLW-2

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.

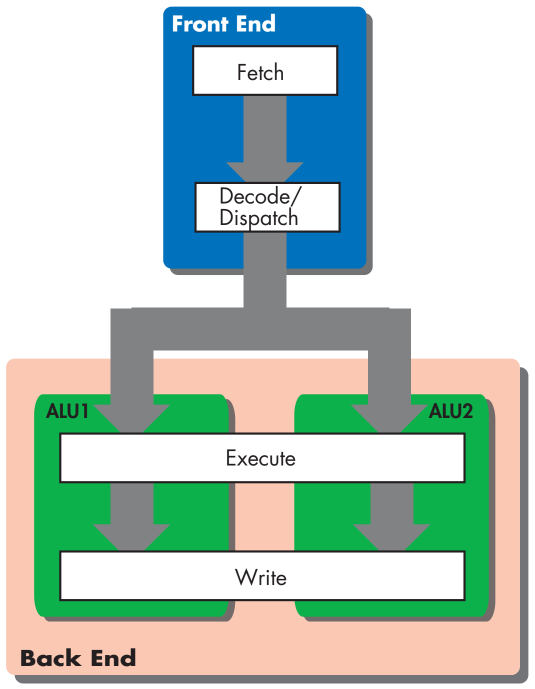


Figure 4-2: The pipeline of the superscalar DLW-2

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/clock. Recall that one instruction/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.

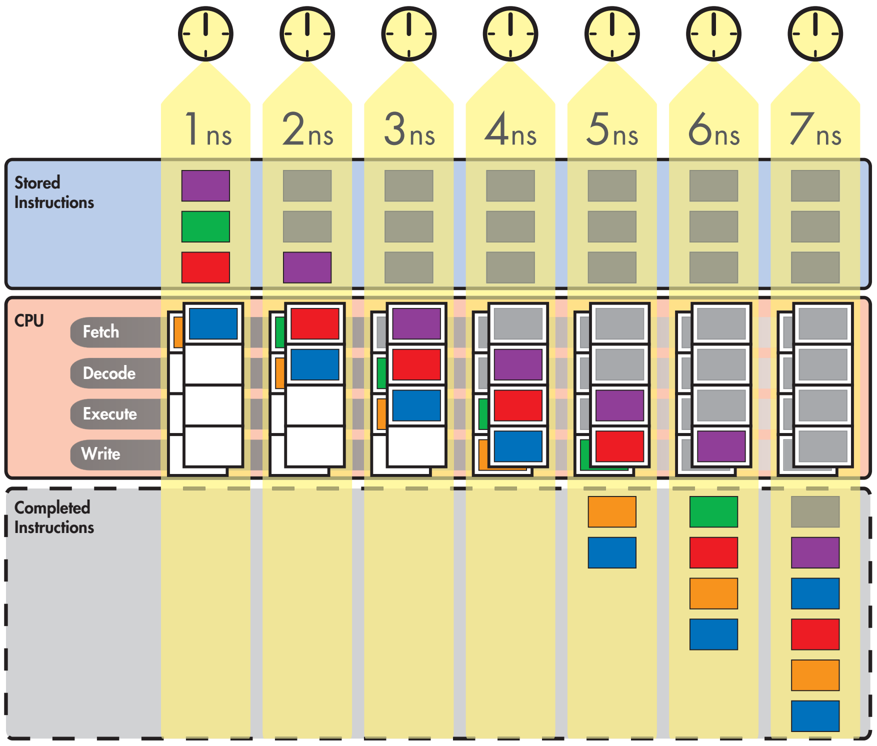


Figure 4-3: Superscalar execution and pipelining combined

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	1.01
	–500	15.234
	37	–0.0023
Vector	{5, –7, –9, 8}	{0.99, –1.1, 3.31}
	{1,003, 42, 97, 86, 97}	{50.01, 0.002, –1.4, 1.4}
	{234, 7, 6, 1, 3, 10, 11}	{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.

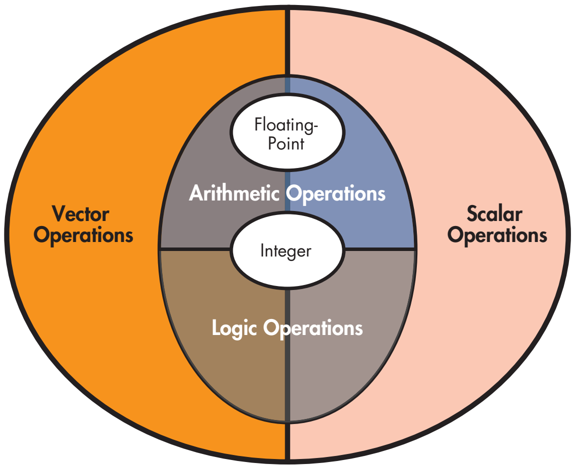


Figure 4-4: Number formats and operation types

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.

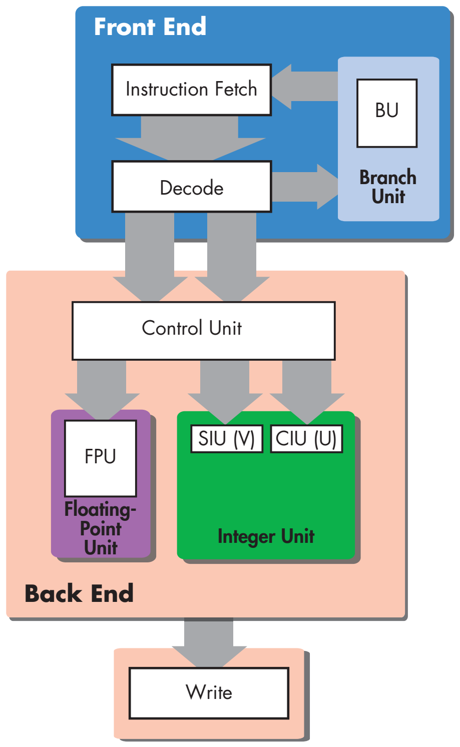


Figure 4-5: The Intel Pentium

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 (IUs) 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 (BU)* is responsible for executing conditional and unconditional branch instructions. The BU 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 BU 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've 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).

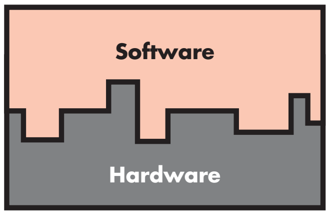


Figure 4-6: Software was custom-fitted to each generation of hardware

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 *instruction set architecture (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.

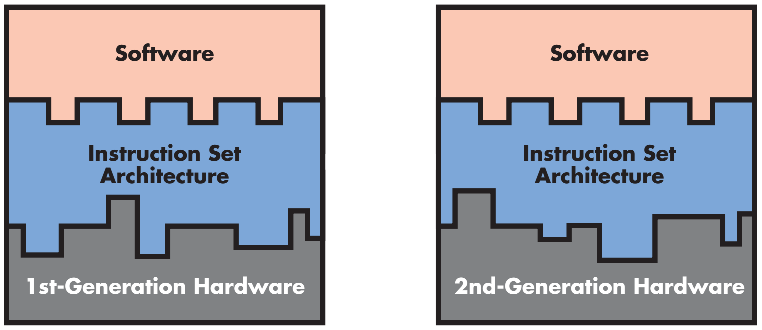


Figure 4-7: The ISA sits between the software and the hardware, providing a consistent interface to the software across hardware generations.

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* x86 *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 the following piece of code:

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.

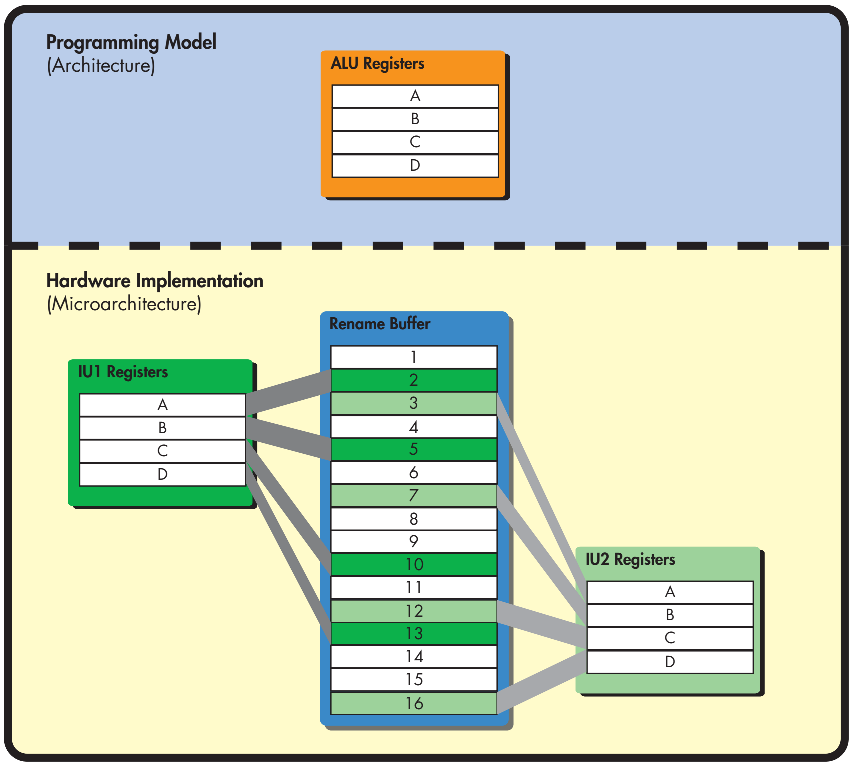


Figure 4-8: Register renaming

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 `adds` 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—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.

5

THE INTEL PENTIUM AND PENTIUM PRO

Now that you've got the basics of microprocessor architecture down, let's look at some real hardware to see how manufacturers implement the two main concepts covered in the previous two chapters—pipelining and superscalar execution—and introduce an entirely new concept: the *instruction window*. First, we'll wrap up our discussion of the fundamentals of microprocessors by taking a look at the Pentium. Then we'll explore in detail the P6 microarchitecture that forms the heart of the Pentium Pro, Pentium II, and Pentium III. The P6 microarchitecture represents a fundamental departure from the microprocessor designs we've studied so far, and an understanding of how it works will give you a solid grasp of the most important concepts in modern microprocessor architecture.

The Original Pentium

The original Pentium is an extremely modest design by today's standards. Transistor budgets were smaller when the chip was introduced in 1993, so the Pentium doesn't pack nearly as much hardware onto its die as a modern microprocessor. Table 5-1 summarizes its features.

Table 5-1: Summary of Pentium Features

Introduction Date	March 22, 1993
Manufacturing Process	0.8 micron
Transistor Count	3.1 million
Clock Speed at Introduction	60 and 66 MHz
Cache Sizes	L1: 8KB instruction, 8KB data
x86 ISA Extensions	MMX added in 1997

A glance at a diagram of the Pentium (see Figure 5-1) shows that it has two integer ALUs and a floating-point ALU, along with some other units that I'll describe later. The Pentium also has a *level 1 cache*—a component of the microprocessor that you haven't yet seen. Before moving on, let's take a moment to look in more detail at this new component, which acts as a code and data storage area for the processor.

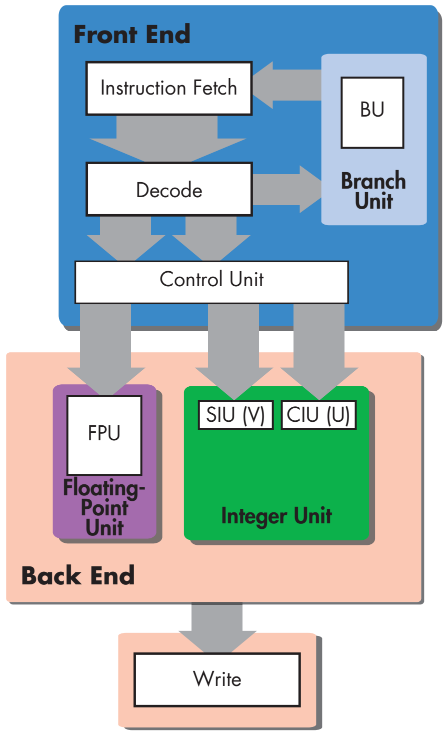


Figure 5-1: The basic microarchitecture of the original Intel Pentium

Caches

So far, I've talked about code and data as if they were all stored in main memory. While this may be true in a limited sense, it doesn't tell the whole story. Though processor speeds have increased dramatically over the past two decades, the speed of main memory has not been able to keep pace. In every computer system currently on the market, there's a yawning speed gap between the processor and main memory. It takes such a huge number of processor clock cycles to transfer code and data between main memory and the registers and execution units that if no solution were available to alleviate this bottleneck, it would kill most of the performance gains brought on by the increase in processor clock speeds.

Very fast memory that could close some of the speed gap is indeed available, but it's too expensive to be widely used in a PC system's main memory. In fact, as a general rule of thumb, the faster the memory technology, the more it costs per unit of storage. As a result, computer system designers fill the speed gap by placing smaller amounts of faster, more expensive memory, called *cache memory*, in between main memory and the registers. These caches, which are depicted in Figure 5-2, hold chunks of frequently used code and data, keeping them within easy reach of the processor's front end.

In most systems, there are multiple levels of cache between main memory and the registers. The *level 1 cache* (called *L1 cache* or just *L1* for short) is the smallest, most expensive bit of cache, so it's located the closest to the processor's back end. Most PC systems have another level of cache, called *level 2 cache* (*L2 cache* or just *L2*), located between the L1 cache and main memory, and some systems even have a third cache level, *L3 cache*, located between the L2 cache and main memory. In fact, as Figure 5-2 shows, main memory itself is really just a cache for the hard disk drive.

When the processor needs a particular piece of code or data, it first checks the L1 cache to see if the desired item is present. If it is—a situation called a *cache hit*—it moves that item directly to either the fetch stage (in the case of code) or the register file (in the case of data). If the item is not present—a *cache miss*—the processor checks the slower but larger L2 cache. If the item is present in the L2, it's copied into the L1 and passed along to the front end or back end. If there's a cache miss in the L2, the processor checks the L3, and so on, until there's either a cache hit, or the cache miss propagates all the way out to main memory.

One popular way of laying out the L1 cache is to have code and data stored in separate halves of the cache. The code half of the cache is often referred to as the *instruction cache* or *I-cache*, and the data half of the cache is referred to as the *data cache* or *D-cache*. This kind of split cache design has certain performance advantages and is used in the all of the processors discussed in this book.

NOTE *The split L1 cache design is often called the* Harvard architecture *as an homage to the Harvard Mark I. The Mark I was a relay-based computer designed by IBM and shipped to Harvard in 1944, and it was the first machine to incorporate the conceptual split between code and data explicitly into its architecture.*

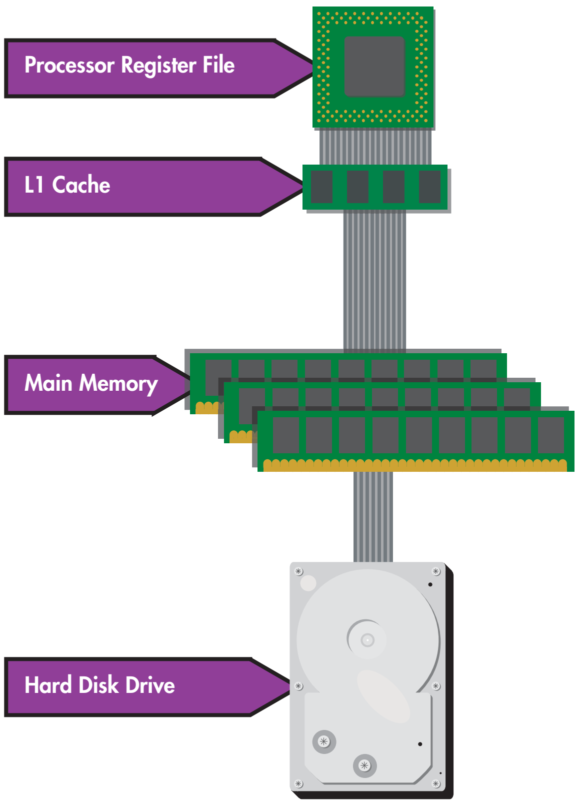


Figure 5-2: The memory hierarchy of a computer system, from the smallest, fastest, and most expensive memory (the register file) to the largest, slowest, and least expensive (the hard disk)

Back when transistor budgets were much tighter than they are today, all caches were located somewhere on the computer's system bus between the CPU and main memory. Today, however, the L1 and L2 caches are commonly integrated onto the CPU die itself, along with the rest of the CPU's circuitry. An on-die cache has significant performance advantages over an off-die cache and is essential for keeping today's deeply pipelined superscalar machines full of code and data.

The Pentium's Pipeline

As you've probably already guessed, a superscalar processor doesn't have just one *pipeline*. Because its execute stage is split up among multiple execution units that operate in parallel, a processor like the Pentium can be said to have multiple pipelines—one for each execution unit. Figure 5-3 illustrates the Pentium's multiple pipelines.

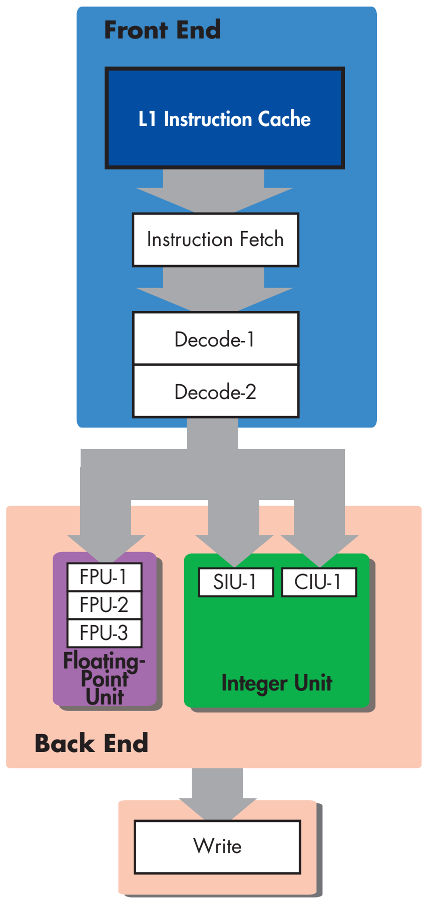


Figure 5-3: The Pentium's pipelines

As you can see, each of the Pentium's pipelines shares four stages in common:

- Fetch
- Decode-1
- Decode-2
- Write

It's when an instruction reaches the execute phase of its lifecycle that it enters a more specialized pipeline, specific to the execution unit.

A processor's various execution units can have different pipeline depths, with the integer pipeline usually being the shortest and the floating-point pipeline usually being the longest. In Figure 5-3, you can see that the Pentium's two integer ALUs have single-stage pipelines, while the floating-point unit has a three-stage pipeline.

Because the integer pipeline is the shortest, it's normally taken to be the default pipeline in discussions of a processor's microarchitecture. So when you see a reference, either in this text or in another text, to a superscalar processor's *pipeline* or *basic pipeline*, you should assume it's referring to the processor's integer pipeline.

The Pentium's basic integer pipeline is five stages long, with the stages broken down as follows:

1. **Prefetch/Fetch** Instructions are fetched from the instruction cache and aligned in prefetch buffers for decoding.
2. **Decode-1** Instructions are decoded into the Pentium's internal instruction format using a fast set of hardware-based rules. Branch prediction also takes place at this stage.
3. **Decode-2** Instructions that require the microcode ROM are decoded here. Also, address computations take place at this stage.
4. **Execute** The integer hardware ALU executes the instruction.
5. **Write-back** The results of the computation are written back to the register file.

These stages should be familiar to you, although the first three stages are slightly different from those of the simple four-stage pipelines described so far. Let's take a quick trip through the Pentium's pipeline stages, so that you can examine each one in a bit more detail.

The *prefetch/fetch* stage corresponds to the fetch phase of the standard instruction lifecycle. Unlike the simple, uniformly sized two-byte instructions of our example DLW architecture, *x*86 instructions can range in size from 1 to 17 bytes in length, though the average instruction length is a little under 3 bytes. *x*86's widely variable instruction length complicates the Pentium's fetch stage, because instructions cannot simply be fetched and then fed directly into the decode stage. Instead, *x*86 instructions are first fetched into a buffer, where each instruction's boundaries are detected and marked. From this buffer, the marked instructions are then aligned and sent to the Pentium's decode hardware.

The decode phase of the Pentium's execution process is split into two pipeline stages, the first of which corresponds most closely to the decode pipeline stage with which you're already familiar. The *decode-1* stage takes the newly fetched instruction and decodes it into the Pentium's internal instruction format, so that it can be used to tell the processor's execution units how to manipulate the data stream. The decode-1 stage also involves the Pentium's branch unit, which checks the currently decoding instruction to see if it's a branch, and if it is a branch, to determine its type. It's at this point that *branch prediction* takes place, but we'll cover branch prediction in more detail in a moment.

The main difference between the Pentium's five-stage pipeline and the four-stage pipelines discussed in Chapter 3 lies in the second decode stage. RISC ISAs, like the primitive DLW ISA of Chapters 1 and 2, support only a few simple, load-store addressing modes. In contrast, the *x*86 ISA supports multiple complex addressing modes, which were originally designed to make assembly language programmers' lives easier but ended up making everyone's lives more difficult. These addressing modes require extra address computations, and these computations are relegated to the *decode-2* stage, where dedicated address computation hardware handles them before dispatching the instruction to the execution units.

The decode-2 stage is also where the Pentium's microcode ROM kicks in. The Pentium decodes many *x*86 instructions directly in its decoding hardware, but the longer instructions are decoded by means of a microcode ROM, as described in the section "A Brief History of the ISA" on page 71.

Once instructions have been decoded, the Pentium's *control unit* determines when they can be dispatched to the back end and to which execution unit. So the control unit's job is to coordinate the movement of instructions from the processor's front end to its back end, so that they can enter the execute and write-back phases of the execution process.

The last two pipeline stages—execute and write-back—should be familiar to you by now. The next major section will describe the Pentium's execution units, so think of it as a more detailed discussion of the execute stage.

The Branch Unit and Branch Prediction

Before we take a closer look at the back end of the Pentium, let's look at one aspect of the Pentium's front end in a bit more detail: the *branch unit (BU)*.

On the right side of the Pentium's front end, shown earlier in Figure 5-1, notice a branch unit attached to the instruction fetch and decode/dispatch pipeline stages. I've depicted the branch unit as part of the front end of the machine—even though it technically still counts as a *memory access unit*—because the BU works closely with the instruction fetcher, steering it by means of the program counter to different sections of the code stream.

The branch unit contains the *branch execution unit (BEU)* and the *branch prediction unit (BPU)*, and whenever the front end's decoder encounters a conditional branch instruction, it sends it to the BU to be executed. The BU in turn usually needs to send it off to one of the other execution units to have the instruction's branch condition evaluated, so that the BU can determine if the branch is *taken* or *not taken*. Once the BU determines that the branch has been taken, it has to get the starting address of the next block of code to be executed. This address, the *branch target*, must be calculated, and the front end must be told to begin fetching code at the new address.

In older processors, the entire processor just sort of sat idle and waited for the branch condition to be evaluated, a wait that could be quite long if the evaluation involved a complex calculation of some sort. Modern processors use a technique called *speculative execution*, which involves making an educated guess at which direction the branch will ultimately take and then beginning execution at the new branch target *before* the branch's condition is actually evaluated. This educated guess is made using one of a variety of *branch prediction* techniques, about which I'll talk more in a moment. Speculative execution is used to keep the delays associated with evaluating branches from introducing bubbles into the pipeline.

Instructions that are speculatively executed cannot write their results back to the register file until the branch condition is evaluated. If the BPU predicted the branch correctly, those speculative instructions can then be marked as non-speculative and have their results written back just like regular instructions.

Branch prediction can backfire when the processor incorrectly predicts a branch. Such mispredictions are bad, because if all of those instructions that the processor has loaded into the pipeline and begun speculatively executing turn out to be from the wrong branch target, the pipeline must be flushed of the erroneous, speculative instructions and their attendant results. After this pipeline flush, the front end must then fetch the correct branch target address so that the processor can begin executing at the right place in the code stream.

As you learned in Chapter 3, flushing the pipeline of instructions and then refilling it again causes a huge hit to the processor's average completion rate and overall performance. Furthermore, there's a delay (and therefore a few cycles worth of pipeline bubbles) associated with calculating the correct branch target and loading the new instruction stream into the front end. This delay can degrade performance significantly, especially on branch-intensive code. It is therefore imperative that a processor's branch prediction hardware be as accurate as possible.

There are two main types of branch prediction: static prediction and dynamic prediction. *Static branch prediction* is simple and relies on the assumption that the majority of backward-pointing branches occur in the context of repetitive loops, where the branch instruction is used to determine whether or not to repeat the loop again. Most of the time, a loop's exit condition will be false, which means that the loop's branch will evaluate to taken, thereby instructing the machine to repeat the loop's code one more time. This being the case, static branch prediction merely assumes that all backward branches are taken. For a branch that points forward to a block of code that comes later in the program, the static predictor assumes that the branch is not taken.

Static prediction is very fast because it doesn't involve any table lookups or calculations, but its success rate varies widely with the program's instruction mix. If the program is full of loops, static prediction works fairly well; if it's not full of loops, static branch prediction performs quite poorly.

To get around the problems associated with static prediction, computer architects use a variety of algorithms for predicting branches based on a program's past behavior. These *dynamic branch prediction* algorithms usually involve the use of both of two types of tables—the *branch history table (BHT)* and the *branch target buffer (BTB)*—to record information about the outcomes of branches that have already been executed. The BHT creates an entry for each conditional branch that the BU has encountered on its last few cycles. This entry includes some bits that indicate the likelihood that the branch will be taken based on its past history. When the front end encounters a branch instruction that has an entry in its BHT, the branch predictor uses this branch history information to decide whether or not to speculatively execute the branch.

Should the branch predictor decide to execute the branch speculatively, it needs to know exactly where in memory the branch is pointing—in other words, it needs a branch target. The BTB stores the branch targets of previously executed branches, so when a branch is taken, the BPU grabs the speculative branch target from the BTB and tells the front end to begin fetching instructions from that address. Hopefully, the BTB contains an entry for the branch you're trying to execute, and hopefully that entry is correct. If the branch target either isn't there or is wrong, you've got a problem. I won't get into the issues surrounding BTB performance, but suffice it to say that a larger BTB is usually better, because it can store more branch targets and thus lower the chances of a BTB miss.

The Pentium uses both static and dynamic branch prediction techniques to prevent mispredictions and branch delays. If a branch instruction does not have an entry in the BHT, the Pentium uses static prediction to decide which path to take. If the instruction does have a BHT entry, dynamic prediction is used. The Pentium's BHT holds 256 entries, which means that it does not have enough space to store information on most of the branches in an average program. Nonetheless, the BHT allows the Pentium to predict branches with a much higher success rate, from 75 to 85 percent, according to Intel, than if it used static prediction alone. The Pentium also uses a BTB to store predicted branch targets. In most of Intel's literature and diagrams, the BTB and BHT are combined under the label the *front-end BTB* and are considered a single structure.

The Pentium's Back End

The Pentium's superscalar back end is fairly straightforward. It has two five-stage integer pipelines, which Intel has designated U and V, and one six-stage floating-point pipeline. This section will take a closer look at each of these ALUs.

The Integer ALUs

The Pentium's U and V integer pipes are not fully symmetric. U, as the default pipe, is slightly more capable and contains a shifter, which V lacks. For this reason, in Figure 5-1, I've labeled the U pipe *simple integer unit (SIU)* and the V pipe *complex integer unit (CIU)*. Most of the designs that we'll study throughout this book have asymmetrical integer units, where one integer unit is more complex and capable of handling more types of instructions than the other, simpler unit.

The Pentium's integer units aren't fully independent. There is a set of restrictions, which I won't take time to outline, that places limits on which combinations of integer instructions can be issued in parallel. All told, though, the Pentium's two integer units initially provided solid enough integer performance for it to be competitive in its day, especially for integer-intensive office apps.

The final thing worth noting about the Pentium's two integer ALUs is that they are responsible for many of the processor's address calculations.

More recently designed processors have specialized hardware for handling the address calculations associated with loads and stores, but on the Pentium these calculations are done in the integer ALUs.

The Floating-Point ALU

Floating-point operations are usually more complex to implement than integer operations, so floating-point pipelines often feature more stages than integer pipelines. The Pentium's six-stage floating-point pipeline is no exception to this rule. The Pentium's floating-point performance is limited by two main factors. First, the processor can only dispatch both a floating-point and an integer operation simultaneously under extremely restrictive circumstances. This isn't too bad, though, because floating-point and integer code are rarely mixed. The second factor, the unfortunate design of the *x*87 floating-point architecture, is more important.

In contrast to the average RISC ISA's *flat* floating-point register file, the *x*87 register file contains eight 80-bit registers arranged in the form of a stack. A *stack* is a simple data storage structure commonly used by programmers and some scientific calculators to perform arithmetic.

NOTE Flat *is an adjective that programmers use to describe an array of elements that is logically laid out so that any element is accessible via a simple address. For instance, all of the register files that we've seen so far are flat, because a programmer needs to know only the name of the register in order to access that register. Contrast the flat file with the stack structure described next, in which elements that are inside the data structure are not immediately and directly accessible to the programmer.*

As Figure 5-4 illustrates, a programmer writes data to the stack by *pushing* it onto the top of the stack via the `push` instruction. The stack therefore grows with each new piece of data that is pushed onto its top. To read data from the stack, the programmer issues a `pop` instruction, which returns the topmost piece of data and removes that data from the stack, causing the stack to shrink.

As the stack grows and shrinks, the variable ST, which stands for the *stack top*, always points to the top element of the stack. In the most basic type of stack, ST is the only element of the stack that can be directly accessed by the programmer—it is read using the `pop` command, and it is written to using the `push` command. This being the case, if you want to read the blue element from the stack in Figure 5-4, you have to pop all of the elements above it, and then you have to pop the blue element itself. Similarly, if you want to alter the blue element, you first have to pop all of the elements above it. Then you pop the blue element itself, alter it, and then push the modified element back onto the stack.

Because the first item that you place in a stack is not accessible until you've removed all the items above it, a stack is often called a *FILO (first in, last out)* data structure. Contrast this with a traditional queue structure, like a supermarket checkout line, which is a *FIFO (first in, first out)* structure.

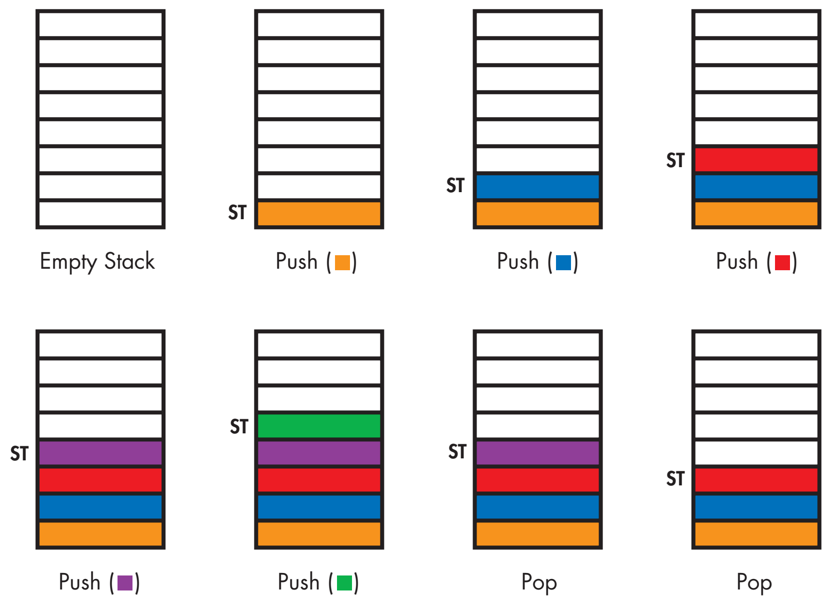


Figure 5-4: Pushing and popping data on a simple stack

All of this pushing and popping sounds like a lot of work, and you might wonder why anyone would use such a data structure. As it turns out, a stack is ideal for certain specialized types of applications, like parsing natural language, keeping track of nested procedure calls, and evaluating postfix arithmetic expressions. It was the stack's utility for evaluating postfix arithmetic expressions that recommended it to the designers of the *x*87 floating-point unit (FPU), so they arranged the FPU's eight floating-point registers as a stack.

NOTE *Normal arithmetic expressions, like 5 + 2 – 1 = 6, are called* infix *expressions, because the arithmetic operators (+ and –) are situated in between the numbers on which they operate.* Postfix *expressions, in contrast, have the operators affixed to the end of the expression, e.g. 521–+ = 6. You could evaluate this expression from left to right using a stack by pushing the numbers 5, 2, and 1 onto the stack (in that order), and then popping them back off (first 1, then 2, and finally 5) as the operators at the end of the expression are encountered. The operators would be applied to the popped numbers as they appear, and the running result would be stored in the top of the stack.*

The *x*87 register file is a little different than the simple stack described two paragraphs ago, because ST is not the only variable through which the stack elements can be accessed. Instead, the programmer can read and write the lower elements of the stack by using ST with an index value that designates the desired element's position relative to the top of the stack. For example, in Figure 5-5, the stack is at its tallest when the green value has just been pushed onto it. This green value is accessed via the variable ST(0), because it occupies the top of the stack. The blue value, because it is three elements down from the top of the stack, is accessed via ST(3).

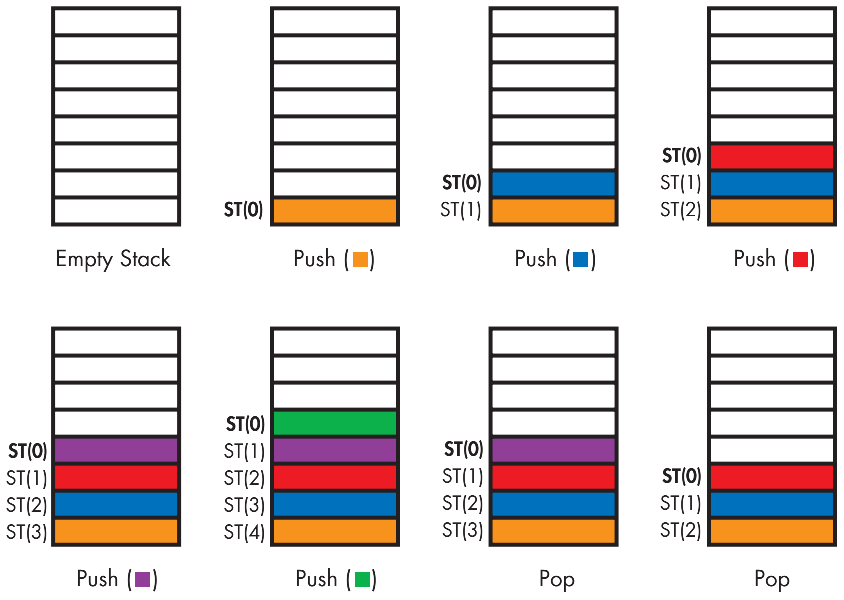


Figure 5-5: Pushing and popping data on the x87 floating-point register stack

In general, to read from or write to a specific register in the stack, you can just use the form ST(*i*), where *i* is the number of registers from the top of the stack.

Programming purists might suggest that since you can access its stack elements arbitrarily, it's kind of pointless to still call the *x*87 register file a *stack*. This would be true except for one catch: For every floating-point arithmetic instruction, at least one of the operands must be the stack top. For instance, if you want to add two floating-point numbers, one of the numbers must be in the stack top and the other can be in any of the other registers. For example, the instruction

```
fadd ST, ST(5)
```

performs the operation

```
ST = ST + ST(5)
```

Though the stack-based nature of *x*87's floating-point register file was originally a boon to assembly language programmers, it soon began to become an obstacle to floating-point performance as compilers saw more widespread use. A flat register file is easier for a compiler to manage, and the newer RISC ISAs featured not only large, flat register files but also three-operand floating-point instructions.

While compiler tricks are arguably enough to make up for *x*87's two-operand limit under most circumstances, they're not quite able to overcome both the two-operand limit and the stack-based limit. So compiler tricks alone won't eliminate the performance penalties associated with both of these quirks combined. The stack-based register file is bad enough that a microarchitectural hack is needed in order simulate a flat register file and thereby keep the *x*87's design from hobbling floating-point performance.

This microarchitectural hack involves turbocharging a single instruction: `fxch`. The `fxch` instruction is an ordinary *x*87 instruction that allows you to swap any element of the stack with the stack top. For example, if you wanted to calculate `ST(2) = ST(2) + ST(6)`, you might execute the following sequence of instructions:

Line #	Code	Comments
1	`fxch ST(2)`	Place the contents of `ST(2)` into `ST` and the contents of `ST` into `ST(2)`.
2	`fadd ST, ST(6)`	Add the contents of `ST` to `ST(6)`.
3	`fxch ST(2)`	Place the contents of `ST(2)` into `ST` and the contents of `ST` into `ST(2)`.

Program 5-1: Using the fxch instruction

Now, here's where the microarchitectural hack comes in. On all modern *x*86 designs, from the original Pentium up to but not including the Pentium 4, the `fxch` instruction can be executed in zero cycles. This means that for all intents and purposes, `fxch` is "free of charge" and can therefore be used when needed without a performance hit. (Note, however, that the `fxch` instruction still takes up decode bandwidth, so even when it's "free," it's not entirely "free.") If you stop and think about the fact that, before executing any floating-point instruction (which has to involve the stack top), you can instantaneously swap `ST` with any other register, you'll realize that a zero-cycle `fxch` instruction gives programmers the functional equivalent of a flat register file.

To revisit the previous example, the fact that the first instruction in Program 5-1 executes "instantaneously," as it were, means that the series of operations effectively looks as follows:

```
fadd ST(2), ST(6)
```

There are in fact some limitations on the use of the "free" `fxch` instruction, but the overall result is that by using this trick, both the Pentium and its successors get the effective benefits of a flat register file, but with the aforementioned hit to decode bandwidth.

x86 Overhead on the Pentium

There are a number of places, like the Pentium's decode-2 stage, where legacy *x*86 support adds significant overhead to the Pentium's design. Intel has estimated that a whopping 30 percent of the Pentium's transistors are dedicated solely to providing *x*86 legacy support. When you consider the fact that the Pentium's RISC competitors with comparable transistor counts could spend those transistors on performance-enhancing hardware like execution units and cache, it's no wonder that the Pentium lagged behind some of its contemporaries when it was first introduced.

A large chunk of the Pentium's legacy-supporting transistors are eaten up by its microcode ROM. Chapter 4 explained that one of the big benefits of RISC processors is that they don't need the microcode ROMs that CISC designs require for decoding large, complex instructions. (For more on *x*86 as a CISC ISA, see the section "CISC, RISC, and Instruction Set Translation" on page 103.)

The front end of the Pentium also suffers from *x*86-related bloat, in that its prefetch logic has to take account of the fact that *x*86 instructions are not a uniform size and hence can straddle cache lines. The Pentium's decode logic also has to support *x*86's segmented memory model, which means checking for and enforcing code segment limits; such checking requires its own dedicated address calculation hardware, in addition to the Pentium's other address hardware.

Summary: The Pentium in Historical Context

The primary factor constraining the Pentium's performance versus its RISC competitors was the fact that its entire front end was bloated with hardware that was there solely to support *x*86 features which, even at the time of the processor's introduction, were rapidly falling out of use. With transistor budgets as tight as they were in 1993, each of those extra address adders and prefetch buffers—not to mention the microcode ROM—represented a painful expenditure of scarce resources that did nothing to enhance the Pentium's performance.

Fortunately for Intel, Pentium's legacy support headaches weren't the end of the story. There were a few facts and trends working in the favor of Intel and the *x*86 ISA. If we momentarily forget about ISA extensions like MMX, SSE, and so on, and the odd handful of special-purpose instructions, like Intel's CPU identifier instruction, that get added to the *x*86 ISA every so often, the core legacy *x*86 ISA is fixed in size and has not grown over the years. Similarly, with one exception (the P6, covered next), the amount of hardware that it takes to support such instructions has not tended to grow either.

Transistors, on the other hand, have shrunk rapidly since the Pentium was introduced. When you put these two facts together, this means that the relative cost (in transistors) of *x*86 support, a cost that is mostly concentrated in an *x*86 CPU's front end, has dropped as CPU transistor counts have increased.

*x*86 support accounts for well under 10 percent of the transistors on the Pentium 4, and this percentage is even smaller for the very latest Intel processors. This steady and dramatic decrease in the relative cost of legacy support has contributed significantly to the ability of *x*86 hardware to catch up to and even surpass its RISC competitors in both integer and floating-point performance. In other words, Moore's Curves have been extremely kind to the *x*86 ISA.

In spite of the high price it paid for *x*86 support, the Pentium was commercially successful, and it furthered Intel's dominance in the *x*86 market that the company had invented. But for Intel to take *x*86 performance to the next level, it needed to take a series of radical steps with the follow-on to the Pentium, the Pentium Pro.

NOTE *Here and throughout this book, I use the term* Moore's Curves *in place of the more popular phrase* Moore's Law. *For a detailed explanation of the phenomenon referred to by both of these terms, see my article at Ars Technica entitled "Understanding Moore's Law" (http://arstechnica.com/paedia/m/moore/moore-1.html).*

The Intel P6 Microarchitecture: The Pentium Pro

Intel's P6 microarchitecture, first implemented in the Pentium Pro, was by any reasonable metric a resounding success. Its performance was significantly better than that of the Pentium, and the market rewarded Intel handsomely for it. The microarchitecture also proved extremely scalable, furnishing Intel with a good half-decade of desktop dominance and paving the way for *x*86 systems to compete with RISC in the workstation and server markets. Table 5-2 summarizes the evolution of the P6 microarchitecture's features.

Table 5-2: The Evolution of the P6

	Pentium Pro Vitals	Pentium II Vitals	Pentium III Vitals
Introduction Date	November 1, 1995	May 7, 1997	February 26, 1999
Process	0.60/0.35 micron	0.35 micron	0.25 micron
Transistor Count	5.5 million	7.5 million	9.5 million
Clock Speed at Introduction	150, 166, 180, and 200 MHz	233, 266, and 300 MHz	450 and 500 MHz
L1 Cache Size	8KB instruction, 8KB data	16KB instruction, 16KB data	16KB instruction, 16KB data
L2 Cache Size	256KB or 512KB (on-die)	512KB (off-die)	512KB (on-die)
***x*86 ISA Extensions**		MMX	SSE added in 1999

What was the P6's secret, and how did it offer such a quantum leap in performance? The answer is complex and involves the contribution of numerous technologies and techniques, the most important of which had already been introduced into the *x*86 world by Intel's smaller *x*86 competitors (most notably, AMD's K5): the decoupling of the front end's fetching and decoding functions from the back end's execution function by means of an *instruction window.*

Figure 5-6 illustrates the basic P6 microarchitecture. As you can see, this microarchitecture sports a quite a few prominent features that distinguish it fundamentally from the designs we've studied thus far.

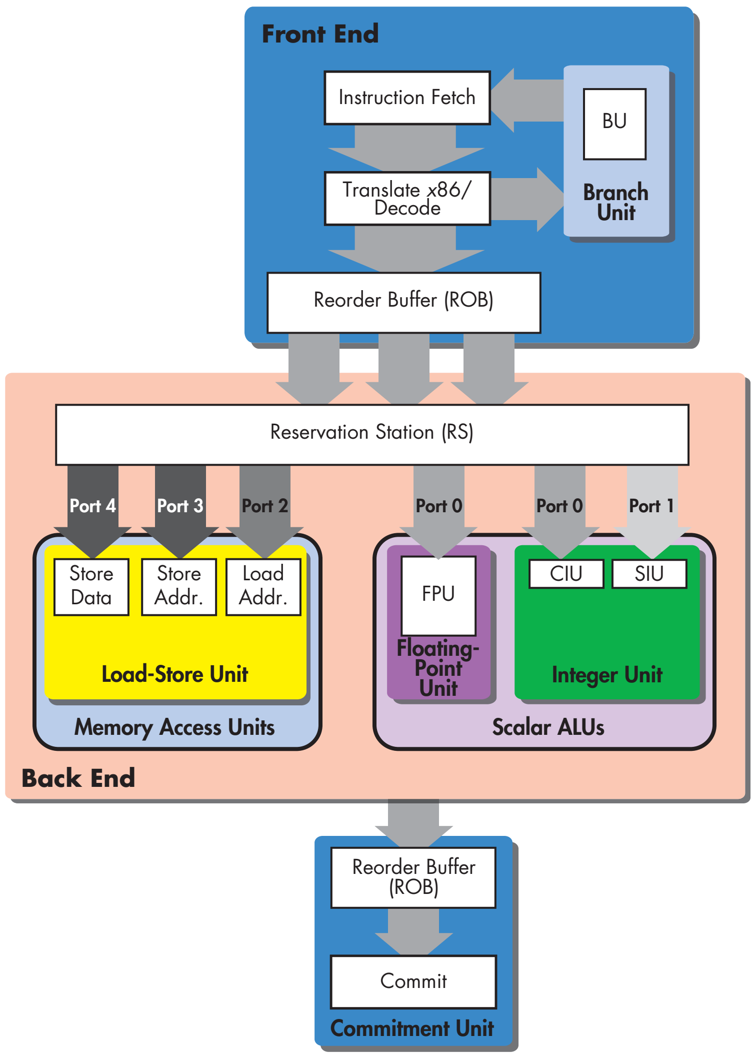


Figure 5-6: The Pentium Pro

Decoupling the Front End from the Back End

In the Pentium and its predecessors, instructions travel directly from the decoding hardware to the execution hardware, as depicted in Figure 5-7. In this simple processor, instructions are *statically scheduled* by the dispatch logic for execution by the two ALUs. First, instructions are fetched and decoded. Next, the control unit's dispatch logic examines a pair of instructions using a set of hardwired rules to determine whether or not they can be executed in parallel. If the two instructions can be executed in parallel, the control unit sends them to the two ALUs, where they're simultaneously executed on the same clock cycle. When the two instructions have *completed* their execution phase (i.e., their results are available on the data bus), they're put back in program order, and their results are written back to the register file in the proper sequence.

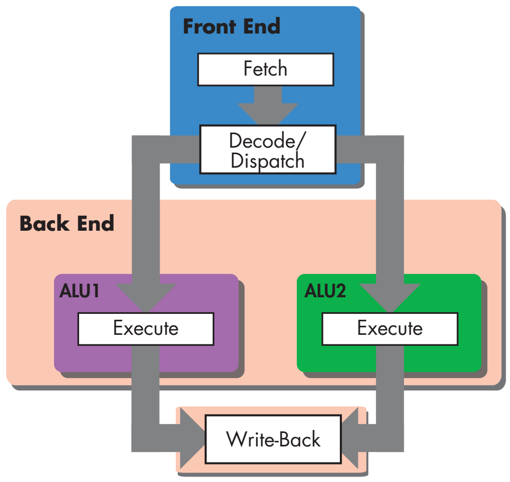


Figure 5-7: Static scheduling in the original Pentium

This static, rules-based approach to dispatching instructions is rigid and simplistic, and it has two major drawbacks, both stemming from the fact that although the code stream is inherently sequential, a superscalar processor attempts to execute parts of it in parallel. Specifically, static scheduling

- adapts poorly to the dynamic and ever-changing code stream;
- makes poor use of wider superscalar hardware.

Because the Pentium can dispatch at most two operations simultaneously from its decode hardware to its execution hardware on each clock cycle, its dispatch rules look at only two instructions at a time to see if they can or cannot be dispatched simultaneously. If more execution hardware were added to the Pentium, and the dispatch width were increased to three instructions per cycle (as it is in the P6), the rules for determining which instructions go where would need to be able to account for various possible combinations of two and three instructions at a time in order to get those instructions to the right execution unit at the right time. Furthermore, such rules would inevitably be difficult for programmers to optimize for, and if they weren't overly complex, there would necessarily exist many common instruction sequences that would perform suboptimally under the default rule set. In plain English, the makeup of the code stream would change from application to application and from moment to moment, but the rules responsible for scheduling the code stream's execution on the Pentium's back end would be forever fixed.

The Issue Phase

The solution to the dilemma posed by static execution is to dispatch newly decoded instructions into a special buffer that sits between the front end and the execution units. Once this buffer collects a handful of instructions that are waiting to execute, the processor's *dynamic scheduling* logic can examine the instructions and, after taking into account the state of the processor and the resources currently available for execution, *issue* instructions from the buffer to the execution units at the most opportune time and in the optimal order. The dynamic scheduling logic has quite a bit of freedom to reorder the code stream so that instructions execute optimally, even if it means that two (or more) instructions must be executed not just in parallel but in reverse order. With dynamic scheduling, the current context in which a particular instruction finds itself executing can have much more of an impact on when and how it's executed. In replacing the Pentium's control unit with the combination of a buffer and a dynamic scheduler, the P6 microarchitecture replaces fixed rules with flexibility.

Of course, instructions that have been issued from the buffer to the execution units out of program order must be put back in program order once they've completed their execution phase, so another buffer is needed to catch the instructions that have completed execution and to place them back in program order. We'll discuss that second buffer more in a moment.

Figure 5-8 shows the two new buffers, both of which work together to decouple the execute phase from the rest of the instruction's lifecycle.

In the processor depicted in Figure 5-8, instructions flow in program order from the decode stage into the first buffer, the *issue buffer*, where they sit until the processor's dynamic scheduler determines that they're ready to execute. Once the instructions are ready to execute, they flow from the issue buffer into the execution unit. This move, when instructions travel from the issue buffer where they're scheduled for optimal execution into the execution units themselves, is called *issuing*.

There are a number of factors that can prevent an instruction from executing out of order in the manner described earlier. The instruction may depend for input on the results of an as-yet-unexecuted instruction, or it may be waiting on data to be loaded from memory, or it may be waiting for a busy execution unit to become available, or any one of a number of other conditions may need to be met before the decoded instruction is ready to be sent off to the proper execution unit. But once the instruction is ready, the scheduler sees that it is issued to the execution unit, where it will be executed.

This new twist on the standard instruction lifecycle is called *out-of-order execution*, or *dynamic execution*, and it requires the addition of two new phases to our instruction's lifecycle, as shown in Table 5-3. The first new phase is the *issue phase*, and it encompasses the buffering and reordering of the code stream that I've just described.

The issue phase is implemented in different ways by different processors. It may take multiple pipeline stages, and it may involve the use of multiple buffers arranged in different configurations. What all of the different implementations have in common, though, is that instructions enter the issue phase and then wait there for an unspecified amount of time until the moment is right for them to execute. When they execute, they may do so out of program order.

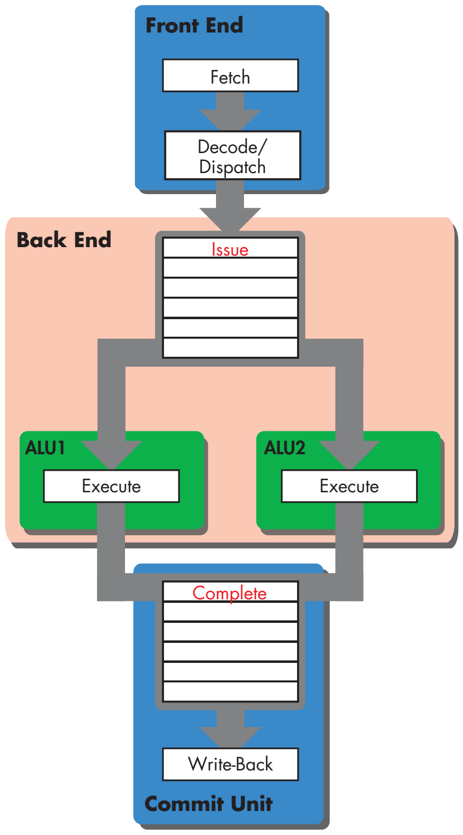


Figure 5-8: Dynamic scheduling using buffers

Aside from its use in dynamic scheduling, another important function of the issue buffer is that it allows the processor to "squeeze" bubbles out of the pipeline prior to the execution phase. The buffer is a queue, and instructions that enter it drop down into the bottommost available entry.

Table 5-3: Phases of a Dynamically Scheduled Instruction's Lifecycle

1	Fetch	In order
2	Decode/dispatch	
3	Issue	Reorder
4	Execute	Out of order
5	Complete	Reorder
6	Write-back (commit)	In order

So if an instruction is preceded by a pipeline bubble, when it enters the issue buffer, it will drop down into the empty space directly behind the instruction ahead of it, thereby eliminating the bubble.

Of course, the issue buffer's ability to squeeze out pipeline bubbles depends on the front end's ability to produce more instructions per cycle than the back end can consume. If the back end and front end move in lock step, the pipeline bubbles will propagate through the issue queues into the back end.

The Completion Phase

The second phase that out-of-order execution adds to an instruction's lifecycle is the *completion phase*. In this phase, instructions that have finished executing, or *completed execution*, wait in a second buffer to have their results written back to the register file *in program order*. When an instruction's results are written back to the register file and the programmer-visible machine state is permanently altered, that instruction is said to *commit*. Instructions must commit in program order if the illusion of sequential execution is to be maintained. This means that no instruction can commit until all of the instructions that were originally ahead of it in the code stream have committed.

The requirement that all instructions must commit in their original program order is what necessitates the second buffer shown in Figure 5-8. The processor needs a place to collect instructions as they complete the out-of-order execution phase of their lifecycle, so that they can be put back in their original order before being sent to the final write stage, where they're committed. Like the issue buffer described earlier, this completion buffer can take a number of forms. We'll look at the form that this buffer takes in the P6 shortly.

I stated previously that an instruction sits in the completion phase's buffer, which I'll call the *completion buffer* for now, and waits to have its result written back to the register file. But where does the instruction's result wait during this interim period? When an instruction is executed out of order, its result goes into a special rename register that has been allocated especially for use by that instruction. Note that this rename register is part of the processor's internal bookkeeping apparatus, which means it is not a part of the programming model and is therefore not visible to the programmer. The result waits in this hidden rename register until the instruction commits, at which time the result is written from the rename register into the programmer-visible architectural register file. After the instruction's result is committed, the rename register then goes back into the pool of available rename registers, where it can be assigned to another instruction on a later cycle.

The P6's Issue Phase: The Reservation Station

The P6 microarchitecture feeds each newly decoded instruction into a buffer called the *reservation station (RS)*, where it waits until all of its execution requirements are met. Once they've been met, the instruction then moves out of the reservation station into an execution unit (i.e., it is issued), where it executes.

A glance at the P6 diagram (Figure 5-6) shows that up to three instructions per cycle can be dispatched from the decoders into the reservation station. And as you'll see shortly, up to five instructions per cycle can be issued from the reservation station into the execution units. Thus the Pentium's original superscalar design, in which two instructions per cycle could dispatch from the decoders directly into the back end, has been replaced with a buffered design in which three instructions can dispatch into the buffer and five instructions can issue out of it on any given cycle.

This buffering action, and the decoupling of the front end's fetch/decode bandwidth from the back end's execution bandwidth that it enables, are at the heart of the P6's performance gains.

The P6's Completion Phase: The Reorder Buffer

Because the P6 microarchitecture must commit its instructions in order, it needs a place to keep track of the original program order of each instruction that enters the reservation station. Therefore, after the instructions are decoded, they must travel through the *reorder buffer (ROB)* before flowing into the reservation station. The ROB is like a large logbook in which the P6 can record all the essential information about each instruction that enters the out-of-order back end. The primary function of the ROB is to ensure that instructions come out the other side of the out-of-order back end in the same order in which they entered it. In other words, it's the reservation station's job to see that instructions are executed in the most optimal order, even if that means executing them out of program order. It's the reorder buffer's job to ensure that the finished instructions get put back in program order and that their results are written to the architectural register file in the proper sequence. To this end, the ROB stores data about each instruction's status, operands, register needs, original place in the program, and so on.

So newly decoded instructions flow into the ROB, where their relevant information is logged in one of 40 available entries. From there, they pass on to the reservation station, and then on to the back end. Once they're done executing, they wait in the ROB until they're ready to be committed.

The role I've just described for the reorder buffer should be familiar to you at this point. The reorder buffer corresponds to the structure that I called the *completion buffer* earlier, but with a few extra duties assigned to it.

If you look at my diagram of the P6 microarchitecture, you'll notice that the reorder buffer is depicted in two spots: the front end and the commit unit. This is because the ROB is active in both of these phases of the instruction's lifecycle. The ROB is tasked with tracking instructions as they move through the phases of their lifecycle and with putting the instructions back in program order at the end of their lifecycle. So newly decoded instructions must be given a tracking entry in the ROB and have a temporary rename register allocated for their private use. Similarly, newly executed instructions must wait in the ROB before they can commit by having the contents of the temporary rename register that holds their result permanently written to the architectural register file.

As implied in the previous sentence, not only does the P6's ROB act as a completion buffer and an instruction tracker, but it also handles register renaming. Each of the P6 microarchitecture's 40 ROB entries has a *data field* that holds program data just like an *x*86 register. These fields give the P6's back end 40 microarchitectural rename registers to work with, and they're used in combination with the P6's *register allocation table (RAT)* to implement register renaming in the P6 microarchitecture.

The Instruction Window

The reservation station and the reorder buffer together make up the heart of the P6's out-of-order back end, and they account for its drastic clock-for-clock performance advantage over the original Pentium. These two buffers—the one for reshuffling and optimizing the code stream (the RS) and the other for unshuffling and reordering the code stream (the ROB)—enable the P6 processor to dynamically and intelligently adapt its operation to fit the needs of the ever-changing code stream.

A common metaphor for thinking about and talking about the P6's RS + ROB combination, or analogous structures on other processors, is that of an *instruction window*. The P6's ROB can track up to 40 instructions in various stages of execution, and its reservation station can hold and examine up to 20 instructions to determine the optimal time for them to execute. Think of the reservation station's 20-instruction buffer as a window that moves along the sequentially ordered code stream; on any given cycle, the P6 is looking through this window at that visible segment of the code stream and thinking about how its hardware can optimally execute the 20 or so instructions that it sees there.

A good analogy for this is the game of Tetris, where a small preview window shows you the next piece that will come your way while you're deciding how best to place the currently falling piece. Thus at any given moment, you can see a total of two Tetris pieces and think about how those two should fit with the pieces that have gone before and those that might come after.

The P6 microarchitecture's job is a little harder than the average Tetris player's, because it must maneuver and optimally place as many as three falling pieces at a time; hence it needs to be able to see farther ahead into the future in order to make the best decisions about what to place where and when. The P6's wider instruction window allows the processor to look further ahead in the code stream and to juggle its instructions so that they fit together with the currently available execution resources in the optimal manner.

The P6 Pipeline

The P6 has a 12-stage pipeline that's considerably longer than the Pentium's five-stage pipeline. I won't enumerate and describe all 12 stages individually, but I will give a general overview of the phases that the P6's pipeline passes through.

BTB access and instruction fetch

The first three-and-a-half pipeline stages are dedicated to accessing the branch target buffer and fetching the next instruction. The P6's two-cycle instruction fetch phase is longer than the Pentium's one-cycle fetch phase, but it keeps the L1 cache access latency from holding back the clock speed of the processor as a whole.

Decode

The next two-and-a-half stages are dedicated to decoding *x*86 instructions and breaking them down into the P6's internal, RISC-like instruction format. We'll discuss this instruction set translation, which takes place in all modern *x*86 processors and even in some RISC processors, in more detail shortly.

Register rename

This stage takes care of register renaming and logging instructions in the ROB.

Write to RS

Writing instructions from the ROB into the RS takes one cycle, and it occurs in this stage.

Read from RS

At this point, the issue phase of the instruction's lifecycle is under way. Instructions can sit in the RS for an unspecified number of cycles before being read from the RS. Even if they're read from the RS immediately after entering it, it takes one cycle to move instructions out of the RS, through the *issue ports* and into the execution units.

Execute

Instruction execution can take one cycle, as in the case of simple integer instructions, or multiple cycles, as in the case of floating-point instructions.

Commit

These two final cycles are dedicated to writing the results of the instruction execution back into the ROB, and then committing the instructions by writing their results from the ROB into the architectural register file.

Lengthening the P6's pipeline as described in this chapter has two primary beneficial effects. First, it allows Intel to crank up the processor's clock speed, since each of the stages is shorter and simpler and can be completed quicker. The second effect is a little more subtle and less widely appreciated.

The P6's longer pipeline, when combined with its buffered decoupling of fetch/decode bandwidth from execution bandwidth, allows the processor to hide hiccups in the fetch and decode stages. In short, the nine pipeline stages that lie ahead of the execute stage combine with the RS to form a deep buffer for instructions. This buffer can hide gaps and hang-ups in the flow of instructions in much the same way that a large water reservoir can hide interruptions in the flow of water to a facility.

But on the downside (to continue the water reservoir example), when one dead animal is spotted floating in the reservoir, the whole thing has to be flushed. This is sort of the case with the P6 and a branch misprediction.

Branch Prediction on the P6

The P6's architects expended considerably more resources than its predecessor on branch prediction and managed to boost dynamic branch prediction accuracy from the Pentium's approximately 75 percent rate to upwards of 90 percent. The P6 has a 512-entry BHT + BTB, and it uses four bits to record branch history information (compared to the Pentium's two-bit predictor). The four-bit prediction scheme allows the Pentium to store more of each branch's history, thereby increasing its ability to correctly predict branch outcomes.

As you learned in Chapter 2, branch prediction gets more important as pipelines get longer, because a pipeline flush due to a misprediction means more lost cycles and a longer recovery time for the processor's instruction throughput and completion rate.

Consider the case of a conditional branch whose outcome depends on the result of an integer calculation. On the original Pentium, the calculation happens in the fourth pipeline stage, and if the branch prediction unit (BPU) has guessed incorrectly, only three cycles worth of work would be lost in the pipeline flush. On the P6, though, the conditional calculation isn't performed until stage 10, which means 10 cycles worth of work get flushed if the BPU guesses incorrectly.

When a dynamically scheduled processor executes instructions speculatively, those speculative instructions and their results are stored in the ROB just like non-speculative instructions. However, the ROB entries for the speculative instructions are marked as speculative and prevented from committing until the branch condition is evaluated. When the branch condition has been evaluated, if the BPU guessed correctly, the speculative instructions' ROB entries are marked as non-speculative, and the instructions are committed in order. If the BPU guessed incorrectly, the speculative instructions and their results are deleted from the ROB without being committed.

The P6 Back End

The P6's back end (illustrated in Figure 5-9) is significantly wider than that of the Pentium. Like the Pentium, it contains two asymmetrical integer ALUs and a separate floating-point unit, but its load-store capabilities have been beefed up to include three execution units devoted solely to memory accesses: a *load address unit*, a *store address unit*, and a *store data unit*. The load address and store address units each contain a pair of four-input adders for calculating addresses and checking segment limits; these are the adders that show up in the decode-1 stage of the original Pentium.

The asymmetrical integer ALUs on the P6 have single-cycle throughput and latency for most operations, with multiplication having single-cycle throughput but four-cycle latency. Thus, multiply instructions execute faster on the P6 than on the Pentium.

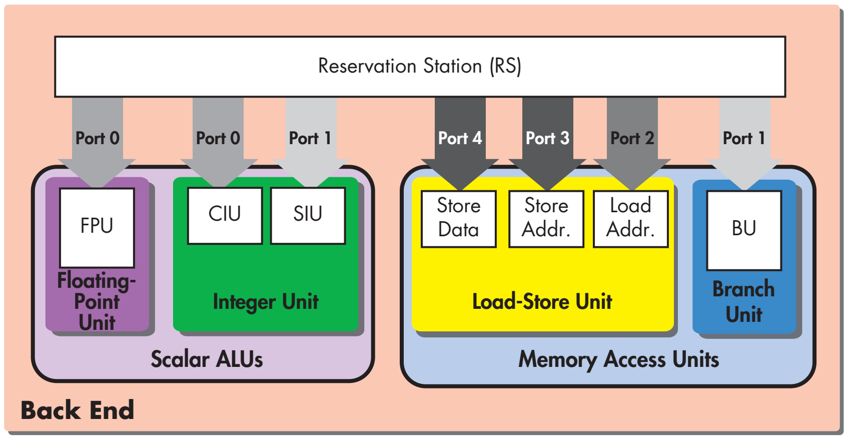


Figure 5-9: The P6 back end

The P6's floating-point unit executes most single- and double-precision operations in three cycles, with five cycles needed for multiply instructions. The FPU is fully pipelined for most instructions, so that most instructions execute with a single-cycle throughput. Some instructions, like floating-point division and square root, are not pipelined and take 18 to 38 and 29 to 69 cycles, respectively.

From the present overview's perspective, the most noteworthy feature of the P6's back end is that its execution units are attached to the reservation station via five *issue ports*, as shown in Figure 5-9.

This means that up to five instructions per cycle can pass from the reservation station through the issue ports and into the execution units. This five–issue port structure is one of the most recognizable features of the P6, and when later designs (like the PII) added execution units to the microarchitecture (like MMX units), they had to be added on the existing issue ports.

If you looked closely at the Pentium Pro diagram, you probably noticed that there were already two units that shared a single port in the original Pentium Pro: the simple integer unit and the floating-point unit. This means that there are some restrictions on issuing a second integer computation and a floating-point computation, but these restrictions rarely affect performance.

CISC, RISC, and Instruction Set Translation

Like the original Pentium, the P6 spends extra time in the decode phase, but this time, the extra cycle and a half goes not to address calculations but to *instruction set translation*. ISA translation is an important technique used in many modern processors, but before you can understand how it works you must first become acquainted with two terms that often show up in computer architecture discussions: RISC and CISC.

One of the most important ways in which the *x*86 ISA differs from that of both the PowerPC ISA (described in the next chapter) and the hypothetical DLW ISA presented in Chapter 2 is that it supports register-to-memory and memory-to-memory format arithmetic instructions. On the DLW architecture, source and destination operands for every arithmetic instruction had to be either registers or immediate values, and it was the programmer's responsibility to include in the program the `load` and `store` instructions necessary to ensure that the arithmetic instructions' source registers were populated with the correct values from memory and their results written back to memory. On the *x*86 architecture, the programmer can voluntarily surrender control of most load-store traffic to the processor by using source and/or destination operands that are memory locations. If the DLW architecture supported such operations, they might look as follows:

Line #	Code	Comments
1	`add #12, #13, A`	Add the contents of memory locations `#12` and `#13` and place the result in register `A`.
2	`sub A, #15, #16`	Subtract the contents of register `A` from the contents of memory location `#15` and store the result in memory location `#16`.
3	`sub A, #B, #100`	Subtract the contents of register `A` from the contents of the memory location pointed to by `#B` and store the result in memory location `#100`.

Program 5-2: Arithmetic instructions using memory-to-memory and memory-to-register formats

Adding the contents of two memory locations, as in line `1` of Program 5-2, still requires the processor to load the necessary values into registers and store the results. However, in memory-to-register and memory-to-memory format instructions, these load and store operations are *implicit* in the instruction. The processor must look at the instruction and figure out that it needs to perform the necessary memory accesses; then it must perform them before and/or after it executes the arithmetic part of the instruction. So for the `add` in line `1` of Program 5-2, the processor would have to perform two loads before executing the addition. Similarly, for the subtractions in lines `2` and `3`, the processor would have to perform one load before executing the subtraction and one store afterwards.

The use of such register-to-memory and memory-to-memory format instructions shifts the burden of scheduling memory traffic from the programmer to the processor, freeing the programmer to focus on other aspects of coding. It also has the effect of reducing the number of instructions that a programmer must write (or *code density*) in order to perform most tasks. In the days when programmers programmed primarily in assembly language, compilers for *high-level languages (HLLs)* like C and FORTRAN were primitive, and main memories were small and expensive, ISA qualities like programmer ease-of-use and high code density were very attractive.

A further technique that ISAs like *x*86 use to lessen the burden on programmers and increase code density is the inclusion of ISA-level support for complex data types like *strings*. A string is simply a series, or "string," of contiguous memory locations of a certain length. Strings are often used to store ASCII text, so a short string might store a word, or a longer string might store a whole sentence. If an ISA includes instructions for working with strings—

and the *x*86 ISA does—assembly language programmers can write programs like text editors and terminal applications in a much shorter length of time and with significantly fewer instructions than they could if the ISA lacked such support.

Complex instructions, like the *x*86 string manipulation instructions, carry out complex, multistep tasks and therefore stand in for what would otherwise be many lines of RISC assembler code. These types of instructions have serious drawbacks, though, when it comes to performing the kind of dynamic scheduling and out-of-order execution described earlier in this chapter. String instructions, for instance, have latencies that can vary with the length of the string being manipulated—the longer the string, the more cycles the instruction takes to execute. Because their latencies are not predictable, it's difficult for the processor to schedule them optimally using the dynamic scheduling mechanisms described previously.

Finally, complex instructions often vary in the number of bytes they need in order to be rendered in machine language. Such variable-length instructions are more difficult to fetch and decode, and once they're decoded, they're more difficult to schedule.

Because of its use of multiple instruction formats (register-to-memory and memory-to-memory) and complex, variable-length instructions, *x*86 is an example of an approach to processor and ISA design called *complex instruction set computing (CISC)*. Both DLW and PowerPC, in contrast, represent an approach called *reduced instruction set computing (RISC)*, in which all machine language instructions are the same length, fewer instruction formats are supported, and complex instructions are eliminated entirely. RISC ISAs are harder to program for in assembly language, so they assume the existence and widespread use of high-level languages and sophisticated compilers. For RISC programmers who use a high-level language like C, the burden of scheduling memory traffic and handling complex data types shifts from the processor to the compiler. By shifting the burden of scheduling memory accesses and other types of code to the compiler, processors that implement RISC ISAs can be made less complex, and because they're less complex, they can be faster and more efficient.

It would be nice if *x*86, which is far and away the world's most popular ISA, were RISC, but it isn't. The *x*86 ISA is a textbook example of a CISC ISA, and that means processors that implement *x*86 require more complicated microarchitectures. At some point, *x*86 processor designers realized that in order to use the latest RISC-oriented dynamic scheduling techniques to speed *x*86-based architectures without the processor's complexity spinning out of control, they'd have to limit the added complexity to the front end by translating *x*86 CISC operations into smaller, faster, more uniform RISC-like operations for use in the back end. AMD's K6 and Intel's P6 were two early *x*86 designs that used this type of *instruction set translation* to great advantage. The technique was so successful that all subsequent *x*86 processors from both Intel and AMD have used instruction set translation, as have some RISC processors like IBM's PowerPC 970.

The P6 Microarchitecture's Instruction Decoding Unit

The P6 microarchitecture breaks down complex, variable-length *x*86 instructions into one or more smaller, fixed-length *micro-operations* (aka *micro-ops*, *µops*, or *uops*), using a decoding unit that consists of three separate decoders, depicted in Figure 5-10: two simple/fast decoders, which handle simple *x*86 instructions and can produce one decoded micro-op per cycle; and one complex/slow decoder, which handles the more complex *x*86 instructions and can produce up to four decoded micro-ops per cycle.

Sixteen-byte groups of architected *x*86 instructions are fetched from the I-cache into the front end's 32-byte instruction queue, where predecoding logic first identifies each instruction's boundaries and type before aligning the instructions for entry into the decoding hardware. Up to three *x*86 instructions per cycle can then move from the instruction queue into the decoders, where they're converted to micro-ops and passed into a micro-op queue before going to the ROB. Together, the P6's three decoders are capable of producing up to six decoded micro-ops per cycle (four from the complex/slow decoder plus one from each of the two simple/fast decoders) for consumption by the micro-op queue. The micro-op queue, in turn, is capable of passing up to three micro-ops per cycle into the P6's instruction window.

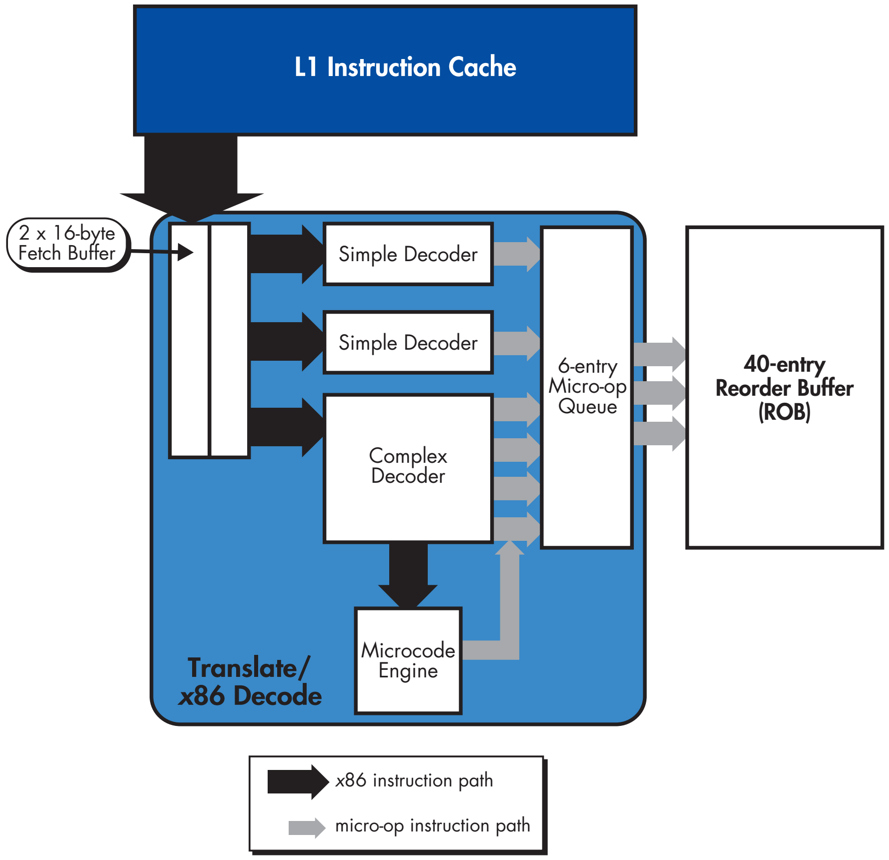


Figure 5-10: The P6 microarchitecture's decoding hardware

Simple *x*86 instructions, which can be decoded very rapidly and which break down into only one or two micro-ops, are by far the most common type of instruction found in an average *x*86 program. So the P6 dedicates most of its decoding hardware to these types of instructions. More complex *x*86 instructions, like string manipulation instructions, are less common and take longer to decode. The P6's complex/slow decoder works in conjunction with the microcode ROM to handle the really complex legacy instructions, which are translated into sequences of micro-ops that are read directly from the ROM.

The Cost of x86 Legacy Support on the P6

All of this decoding and translation hardware takes up a lot of transistors. MDR estimates that close to 40 percent of the P6's transistor budget is spent on *x*86 legacy support. If correct, that's even higher than the astonishing 30 percent estimate for the original Pentium, and even if it's incorrect, it still suggests that the cost of legacy support is quite high.

At this point, you're probably thinking back to the conclusion of the first part of this chapter, in which I suggested that the relative cost of *x*86 support has decreased with successive generations of the Pentium. This is still true, but the trend didn't hold for the first instantiation of the P6 microarchitecture: the original 133 MHz Pentium Pro. The Pentium Pro's L1 cache was a modest 16KB, which was small even by 1995 standards. The chip's designers had to skimp on on-die cache, because they'd spent so much of their transistor budget on the decoding and translation hardware. Comparable RISC processors had two to four times that amount of cache, because less of the die was taken up with front-end logic, so they could use the space for cache.

When the P6 microarchitecture was originally launched in its Pentium Pro incarnation, transistor counts were still relatively low by today's standards. But as Moore's Curves marched on, microprocessor designers went from thinking, "How do we squeeze all the hardware that we'd like to put on the chip into our transistor budget?" to "Now our generous transistor budget will let us do some really nice things!" to "How on earth do we get this many transistors to do useful, performance-enhancing work?"

What really drove the decrease in subsequent generations' costs for *x*86 support was the increase in L1 cache sizes and the L2 cache's move onto the die, because the answer to that last question has—until recently—been, "Let's add cache."

Summary: The P6 Microarchitecture in Historical Context

This concluding section provides an overview of the P6 microarchitecture in its various incarnations. The main focus here is on fitting everything together and giving you a sense of the big picture of how the P6 evolved. The historical narrative outlined in this section seems, in retrospect, to have unfolded over a much longer length of time than the seven years that it actually took to go from the Pentium Pro to the Pentium 4, but seven years is an eternity in computer time.

The Pentium Pro

The processor described in the preceding section under the name *P6* is the original, 133 MHz Pentium Pro. As you can see from the processor comparison in Table 5-2, the Pentium Pro was relatively short on transistors, short on cache, and short on features. In fact, the original Pentium eventually got rudimentary SIMD computing support in the form of Intel's MMX (Multimedia Extensions), but the Pentium Pro didn't have enough room for that, so SIMD got jettisoned in favor of all that fancy decoding logic described earlier.

In spite of all its shortcomings, though, the Pentium Pro did manage to raise the *x*86 performance bar significantly. Its out-of-order execution engine, dual integer pipelines, and improved floating-point unit gave it enough oomph to get the *x*86 ISA into the commodity server market.

The Pentium II

MMX didn't make a return to the Intel product line until the Pentium II. Introduced in 1997, this next generation of the P6 microarchitecture debuted at speeds ranging from 233 to 300 MHz and sported a number of performance-enhancing improvements over its predecessor.

First among these improvements was an on-die, split L1 cache that was doubled in size to 32KB. This larger L1 helped boost performance across the board by keeping the PII's lengthy pipeline full of code and data.

The P6's basic pipeline stayed the same in the PII, but Intel widened the back end as depicted in Figure 5-11 by adding the aforementioned MMX support in the form of two new MMX execution units: one on issue port 0 and the other on issue port 1. MMX provided vector support for integers only, though. It wasn't until the introduction of Streaming SIMD Extensions (SSE) with the PIII that the P6 microarchitecture got support for floating-point vector processing.

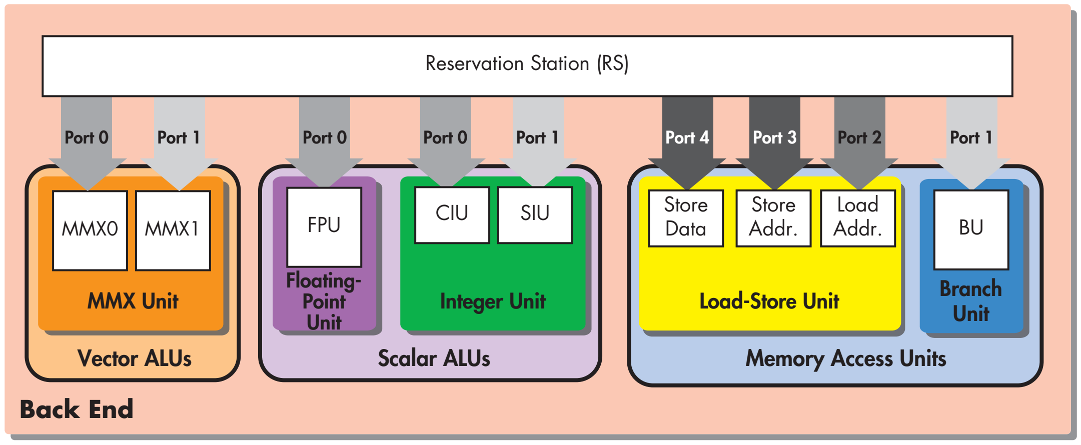


Figure 5-11: The Pentium II's back end

The Pentium II's integer and floating-point performance was relatively good compared to its CISC competitors, and it helped further the trend, started by the Pentium Pro, of *x*86 commodity hardware's migration into the server and workstation realms. However, the PII still couldn't stand up to RISC designs built on the same process with similar transistor counts. Its main advantage was in bang for the buck, whereas the more expensive RISC chips specialized in pure bang.

The Pentium III

Intel introduced its next P6 derivative, the Pentium III (PIII), in 1999 at 450 MHz on a 0.25 micron manufacturing process. The first version of the Pentium III, code-named *Katmai*, had a 512KB off-die L2 cache that shared a small piece of circuit board (called a *daughtercard*) with the PIII. While this design offered fair performance, the PIII didn't really begin to take off from a performance standpoint until the introduction of the next version of the PIII, code-named *Coppermine*, in early 2000.

Coppermine was produced on a 0.18 micron manufacturing process, which means that Intel could pack more transistors onto the processor die. Intel took advantage of this capability by reducing the PIII's L2 cache size to 256KB and moving the cache onto the CPU die itself. Having the L2 on the same die as both the CPU and the L1 cache dramatically reduced the L2 cache's access time, a fact that more than made up for the reduction in cache size. Coppermine's performance scaled well with increases in clock speed, eventually passing the 1 GHz milestone shortly after AMD's Athlon.

The Pentium III processor introduced two significant additions to the *x*86 ISA, the most important of which was a set of floating-point SIMD extensions to the *x*86 architecture called Streaming SIMD Extensions (SSE). With the addition of SSE's 70 new instructions, the *x*86 architecture completed much more of what had been lacking in its support for vector computing, making it more attractive for applications like games and digital signal processing. I'll cover the MMX and SSE extensions in more detail in Chapter 8, but for now it's necessary to say a word about how the extensions were implemented in hardware.

The Pentium III's designers added the majority of the new SSE hardware on issue port 1 (see the back end in Figure 5-12). The new SSE units attached to port 1 handle vector SIMD addition, shuffle, and reciprocal arithmetic functions. Intel also modified the FPU on port 0 to handle SSE multiplies. Thus the Pentium III's main FPU functional block is responsible for both scalar and vector operations.

The PIII also introduced the infamous *processor serial number (PSN)*, along with new *x*86 instructions aimed at reading the number. The PSN was a unique serial number that marked each processor, and it was intended for use in securing online commercial transactions. However, due to concerns from privacy advocates, the PSN was eventually dropped from the Pentium line.

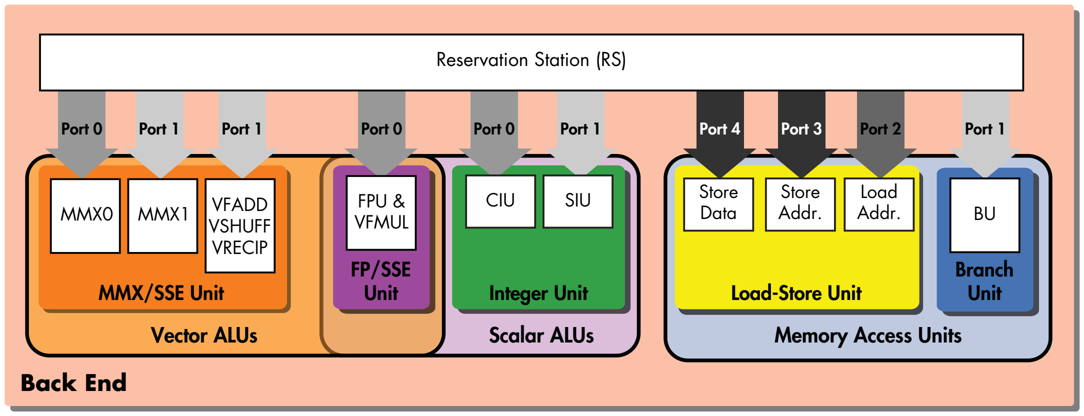


Figure 5-12: The Pentium III's back end

Conclusion

The Pentium may not have outperformed its RISC contemporaries, but it was superior enough to its *x*86-based competition to keep Intel comfortably in command of the commodity PC market. Indeed, prior to the rise of Advanced Micro Devices (AMD) as a serious competitor, Intel had the luxury of setting the pace of progress in the *x*86 PC space. Products were released when Intel was ready to release them, and clock speeds climbed when Intel was ready for them to climb. Intel's competitors were left to respond to what the larger chipmaker was doing, with their own *x*86 products always lagging significantly behind Intel's in performance and popularity.

AMD's Athlon was the first *x*86 processor to pose any sort of threat to Intel's technical dominance, and by the time the PIII made its debut in 1999, it was clear that Intel and AMD were locked in a "gigahertz race" to see who would be the first to introduce a processor with a 1 GHz clock speed. The P6 microarchitecture in its PIII incarnation was Intel's horse in this race, and that basic design eventually reached the 1 GHz mark shortly after AMD's Athlon. Thus a microarchitecture that started out at 150 MHz eventually carried *x*86 beyond 1 GHz and into the lucrative server and workstation markets that RISC architectures had traditionally dominated.

The gigahertz race had a profound effect not only on the commodity PC market but also on the Pentium line itself, insofar as the next chip to bear the Pentium name—the Pentium 4—bore the marks of the gigahertz race stamped into its very architecture. If Intel learned anything in those last few years of the P6's life, it learned that clock speed sells, and it kept that lesson foremost in its mind when it designed the Pentium 4's NetBurst microarchitecture. (For more on the Pentium 4, see Chapters 7 and 8.)

6

POWERPC PROCESSORS: 600 SERIES, 700 SERIES, AND 7400

Now that you've been introduced to the first half of Intel's Pentium line in the previous chapter, this chapter will focus on the origins and development of another popular family of microprocessors: the PowerPC (or PPC) line of processors produced from the joint efforts of Apple, IBM, and Motorola. Because the PowerPC family of processors is extremely large and can be found in an array of applications that ranges from mainframes to desktop PCs to routers to game consoles, this chapter's coverage of PowerPC will present only a small and limited sample of the processors that implement the PowerPC ISA. Specifically, this chapter will focus exclusively on a subset of the PowerPC chips that have been shipped in Apple products, because these chips are the most directly comparable to the Pentium line in that they're aimed at the "personal computer" market.

A Brief History of PowerPC

The PowerPC architecture has its roots in two separate architectures. The first of these is an architecture called POWER (Performance Optimization With Enhanced RISC), IBM's RISC architecture developed for use in mainframes and servers. The second is Motorola's 68000 (aka the 68K) processor, which prior to PowerPC, formed the core of Apple's desktop computing line.

To make a long story very short, IBM needed a way to turn POWER into a wider range of computing products for use outside the server closet, Motorola needed a high-end RISC microprocessor in order to compete in the RISC workstation market, and Apple needed a CPU for its personal computers that would be both cutting-edge and backward compatible with the 68K.

Thus the AIM (Apple, IBM, Motorola) alliance was born, and with it was also born a subset of the POWER architecture dubbed PowerPC. PowerPC processors were to be jointly designed and produced by IBM and Motorola with input from Apple, and were to be used in Apple computers and in the embedded market. The AIM alliance has since passed into history, but PowerPC lives on, not only in Apple computers but in a whole host of different products that use PowerPC-based chips from Motorola and IBM.

The PowerPC 601

In 1993, AIM kicked off the PowerPC party by releasing the 32-bit PowerPC 601 at an initial speed of 66 MHz. The 601, which was based on IBM's older RISC Single Chip (RSC) processor and was originally designed to serve as a "bridge" between POWER and PowerPC, combines parts of IBM's POWER architecture with the 60*x* bus developed by Motorola for use with their 88000. As a bridge, the 601 supports a union of the POWER and PowerPC instruction sets, and it enabled the first PowerPC application writers to easily make the transition from the older ISA to the newer.

NOTE *The term* 32-bit *may be unfamiliar to you at this point. If you're curious about what it means, you might want to skip ahead and skim the chapter on 64-bit computing, Chapter 9.*

Table 6-1 summarizes the features of the PowerPC 601.

Table 6-1: Features of the PowerPC 601

Introduction Date	March 14, 1994
Process	0.60 micron
Transistor Count	2.8 million
Die Size	121 mm^2
Clock Speed at Introduction	60–80 MHz
Cache Sizes	32KB unified L1
First Appeared In	Power Macintosh 6100/60

Even though the joint IBM-Motorola team in Austin, Texas had only 12 months to get this chip off the ground, it was a very nice and full-featured RISC design for its time.

The 601's Pipeline and Front End

In the previous chapter, you learned how complex the different Pentiums' front ends and pipelines tend to be. There is none of that with the 601, which has a classic four-stage RISC integer pipeline:

1. Fetch
2. Decode/dispatch
3. Execute
4. Write-back

The fact that PowerPC's RISC instructions are all the same size means that the 601's instruction fetch logic doesn't have the instruction alignment headaches that plague *x*86 designs, and thus the fetch hardware is simpler and faster. Back when transistor budgets were tight, this kind of thing could make a big difference in performance, power consumption, and cost.

The PowerPC Instruction Queue

As you can see in Figure 6-1, up to eight instructions per cycle can be fetched directly into an eight-entry *instruction queue (IQ)*, where they are decoded before being dispatched to the back end. Get used to seeing the instruction queue, because it shows up in some form in every single PPC model that we'll discuss in this book, all the way down to the PPC 970.

The instruction queue is used mainly for detecting and dealing with branches. The 601's branch unit scans the bottom four entries of the queue, identifying branch instructions and determining what type they are (conditional, unconditional, etc.). In cases where the branch unit has enough information to resolve the branch immediately (e.g., in the case of an unconditional branch, or a conditional branch whose condition depends on information that's already in the condition register), the branch instruction is simply deleted from the instruction queue and replaced with the instruction located at the branch target.

NOTE *The PowerPC condition register is the analog of the processor status word on the Pentium. We'll discuss the condition register in more detail in Chapter 10.*

This branch-elimination technique, called *branch folding*, speeds performance in two ways. First, it eliminates an instruction (the branch) from the code stream, which frees up dispatch bandwidth for other instructions. Second, it eliminates the single-cycle pipeline bubble that usually occurs immediately after a branch. All of the PowerPC processors covered in this chapter perform branch folding.

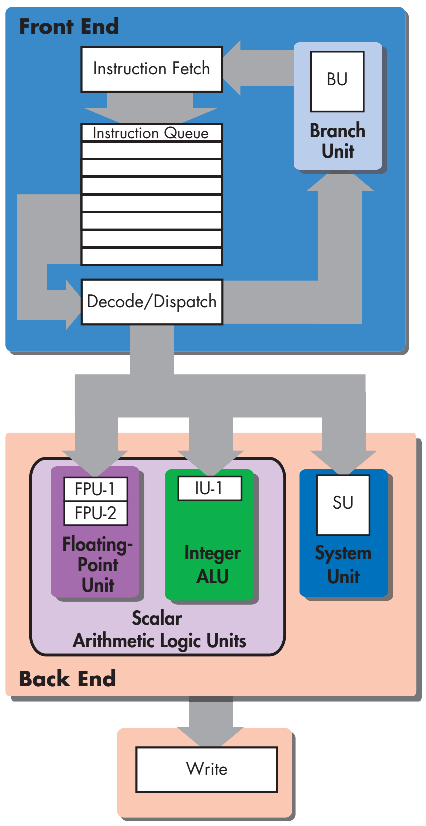


Figure 6-1: PowerPC 601 microarchitecture

If the branch unit determines that the branch is not taken, it allows the branch to propagate to the bottom of the queue, where the dispatch logic simply deletes it from the code stream. The act of allowing not-taken branches to fall out of the instruction queue is called *fall-through*, and it happens on all the PowerPC processors covered in this book.

Non-branch instructions and branch instructions that are not folded sit in the instruction queue while the dispatch logic examines the four bottommost entries to see which three of them it can send off to the back end on the next cycle. The dispatch logic can dispatch up to three instructions per cycle out of order from the bottom four queue entries, with a few restrictions, of which one is the most important for our immediate purposes: Integer instructions can be dispatched only from the bottommost queue entry.

Instruction Scheduling on the 601

Notice that the 601 has no equivalent to the Pentium Pro's reorder buffer (ROB) for keeping track of the original program order. Instead, instructions are tagged with what amounts to metadata so that the write-back logic can commit the results to the register file in program order. This technique of tagging instructions with program-order metadata works fine for a simple, statically scheduled design like the 601 with a very small number of in-flight instructions. But later, dynamically scheduled PPC designs would require dedicated structures for tracking larger numbers of in-flight instructions and making sure that they commit their results in order.

The 601's Back End

From the dispatch stage, instructions go into the 601's back end, where they're executed by one of three different execution units: the integer unit, the floating-point unit, and the branch unit. Let's take a look at each of these units in turn.

The Integer Unit

The 601's 32-bit integer unit is a straightforward fixed-point ALU that is responsible for all of the integer math—including address calculations—on the chip. While *x*86 designs, like the original Pentium, need extra address adders to keep all of the address calculations associated with *x*86's multiplicity of addressing modes from tying up the back end's integer hardware, the 601's RISC, load-store memory model means that it can feasibly handle memory traffic and regular ALU traffic with a single integer execution unit.

So the 601's integer unit handles the following memory-related functions, most of which are moved off into a dedicated load-store unit in subsequent PPC designs:

- Integer and floating-point load-address calculations
- Integer and floating-point store-address calculations
- Integer and floating-point load-data operations
- Integer store-data operations

Cramming all of these load-store functions into the 601's single integer ALU doesn't exactly help the chip's integer performance, but it is good enough to keep up with the Pentium in this area, even though the Pentium has two integer ALUs. Most of this integer performance parity probably comes from the 601's huge 32KB unified L1 cache (compare that to the Pentium's 8KB split L1), a luxury afforded the 601 by the relative simplicity of its front-end decoding hardware.

A final point worth noting about the 601's integer unit is that multi-cycle integer instructions (e.g., integer multiplies and divides) are not fully pipelined. When an instruction that takes, say, five cycles to execute entered the IU, it ties up the entire IU for the whole five cycles. Thankfully, the most common integer instructions are single-cycle instructions.

The Floating-Point Unit

With its single floating-point unit, which handles all floating-point calculations and store-address operations, the 601 was a very strong performer when it was first launched.

The 601's floating-point pipeline is six stages long, and includes the four basic stages outlined earlier in this chapter, but with an extra decode stage and an extra execute stage. What really sets the chip's floating-point hardware apart when compared to its contemporaries is the fact that not only are almost all single-precision operations fully pipelined, but most double-precision (64-bit) floating-point operations are as well. This means that for single-precision operations (with the exception of divides) and most double-precision operations, the 601's floating-point hardware can turn out one instruction per cycle with a two-cycle latency.

Another great feature of the 601's FPU is its ability to do single-precision fused multiply-add (`fmadd`) instructions with single-cycle throughput. The `fmadd` is a core digital signal processing (DSP) and scientific computing function, so the 601's fast `fmadd` capabilities make it well suited to these types of applications. This single-cycle `fmadd` capability is actually a significant feature of the entire PowerPC computing line, from the 601 on down to the present day, and it is one reason why these processors have been so popular for media and scientific applications.

Another factor in the 601's floating-point dominance is that its integer unit handles all of the memory traffic (with the FPU providing the data for floating-point stores). This means that during long stretches of floating-point–only code, the integer unit acts like a dedicated load-store unit (LSU), whose sole purpose is to keep the FPU fed with data.

Such an FPU + LSU combination performs well for two reasons: First, integer and floating-point code are rarely mixed, so it doesn't matter for performance if the integer unit is tied up with floating-point–related memory traffic. Second, floating-point code is often data-intensive, with lots of loads and stores, and thus high levels of memory traffic to keep a dedicated LSU busy.

When you combine both of these factors with the 601's hefty 32KB L1 cache and its ability to do single-cycle fused multiply-adds at a rate of one per clock, you have a floating-point force to be reckoned with in 1994 terms.

The Branch Execution Unit

The 601's branch unit (BU) works in combination with the instruction fetcher and the instruction queue to steer the front end of the processor through the code stream by executing branch instructions and predicting branches. Regarding the latter function, the 601's BU uses a simple static branch predictor to predict conditional branches. I'll talk a bit more about branch prediction and speculative execution in covering the 603e in "The PowerPC 603 and 603e" on page 118.

The Sequencer Unit

The 601 contains a peculiar holdover from the IBM RSC called the sequencer unit. The *sequencer unit*, which I'll admit is a bit of a mystery to me, appears to be a small, CISC-like processor with its own 18-bit instruction set, 32-word RAM, microcode ROM, register file, and execution unit, all of which are embedded on the 601. Its purpose is to execute some legacy instructions particular to the older RSC; to take care of housekeeping chores like self-test, reset, and initialization functions; and to handle exceptions, interrupts, and errors.

The inclusion of the sequencer unit on the 601 is quite obviously the result of the time crunch that the 601 team faced in bringing the first PowerPC chip to market; IBM admitted this much in its 601 white paper. The team started with IBM's RSC as its basis and began redesigning it to implement the PowerPC ISA. Instead of throwing out the sequencer unit, a component that played a major role in the functioning of the original RSC, IBM simply scaled back its size and functionality for use in the 601.

I don't have any exact figures, but I think it's safe to say that this embedded subprocessor unit took up a decent amount of die space on the 601 and that the design team would have thrown it out if it had had more time. Subsequent PowerPC processors, which didn't have to worry about RSC legacy support, implemented all of the (non–RSC-related) functions of the 601's sequencer unit by spreading them out into other functional blocks.

Latency and Throughput Revisited

On superscalar processors like the 601 and its more modern counterparts, different instructions take different numbers of cycles to pass through the processor. Different execution units often have different pipeline depths, and even within one execution unit, different instructions sometimes take different numbers of cycles. Regarding this latter case, one instruction can take longer to pass through an ALU than another instruction, either because the instruction has a mandatory stall in a certain stage or because the particular *subunit* that is handling the instruction has a longer pipeline than the other subunits that together make up the ALU. This being the case, it no longer makes sense for us to simplistically treat instruction latency as a property of the processor as a whole. Rather, instruction latency is actually a matter of the individual instruction, so our discussion will reflect that from now on.

Earlier, we defined an instruction's latency as the minimum number of cycles that an instruction must spend in the execution phase. Here are the latencies of some commonly used PowerPC instructions on the PowerPC G4, a processor that we'll discuss in "The PowerPC 7400 (aka the G4)" on page 133:

Mnemonic	Cycles to Execute
`add`	1
`and`	1
`cmp`	1
`divw`	19
`mul`	6

Notice that most of the instructions take only one cycle to execute, while a few, like division and multiplication, take more. An integer division for a full word, for example, takes 19 cycles to execute, while a 32-bit multiply takes 6 cycles. This means that the division instruction sits in IU1's integer pipeline for 19 cycles, and during this time, no other instruction can execute in IU1.

Now let's look at the floating-point instruction latencies for the G4:

Mnemonic	Cycles to Execute
`fabs`	1-1-1
`fadd`	1-1-1
`fdiv`	32
`fmadd`	1-1-1
`fmul`	1-1-1
`fsub`	1-1-1

These latencies are listed a bit differently than the integer instruction latencies. The numbers separated by dashes tell how long the instruction spends in each of the FPU's three pipeline stages. Most of the instructions listed spend one cycle in each stage, for a total of three cycles in the G4's FPU pipeline, so if a program is using only these instructions, the FPU can start and finish one instruction on each cycle.

A few instructions, like floating-point division, have only a single number in the latency column. This is because an `fdiv` ties up the entire floating-point pipeline when executing. While an `fdiv` is grinding out its 32 cycles in the FPU, no other instructions can execute along with it in the floating-point pipeline. This means that any floating-point instructions that come immediately after the `fdiv` in the code stream must wait in the instruction queue because they cannot be dispatched while the `fdiv` is executing.

Summary: The 601 in Historical Context

The 601 could spend a ton of transistors (at least, a ton for its day) on a 32KB cache, because its front end was so much simpler than that of its *x*86 counterpart, the Intel Pentium. This was a significant advantage to using RISC at that time. The chip made its debut in the PowerMac 6100 to good reviews, and it put Apple in the performance lead over its *x*86 competition. The 601 was definitive in firmly establishing the cult of Apple as a high-end computer maker.

Nonetheless, the 601 did leave some room for improvement. The sequencer unit that it inherited from its mainframe ancestor took up valuable die space that could have been put to better use. With a little more time to tweak it, the 601 could have been closer to perfect. But near perfection would have to wait for the one-two punch of the 603e and 604.

The PowerPC 603 and 603e

While one team was putting the finishing touches on the 601, another team at IBM's Sommerset Design Center in Austin had already begun working on the 601's successor—the 603. The 603 was a significantly different design than the 601, so it was less of an evolutionary shift than it was a completely different processor. Table 6-2 summarizes the features of the PowerPC 603 and 603e.

Table 6-2: Features of the PowerPC 603 and 603e

	PowerPC 603 Vitals	**PowerPC 603e Vitals**
Introduction Date	May 1, 1995	October 16, 1995
Process	0.50 micron	0.50 micron
Transistor Count	1.6 million	2.6 million
Die Size	81 mm^2	98 mm^2
Clock Speed at Introduction	75 MHz	100 MHz
L1 Cache Size	16KB split L1	32KB split L1
First Appeared In	Macintosh Performa 5200CD	Macintosh Performa 6300CD

The 603 was designed to run on very little power, because Apple needed a chip for its PowerBook line of laptop computers. As a result, the processor had a very good performance-per-watt ratio on native PowerPC code, and in fact was able to match the 601 in clock-for-clock performance even though it had about half the number of transistors as the older processor. But the 603's smaller 16KB split L1 cache meant that it was pretty bad at emulating the legacy 68K code that formed a large part of Apple's OS and application base.

As a result, the 603 was relegated to the very lowest end of Apple's product line (the Performas, beginning with the 6200, and the all-in-ones designed for the education market, beginning with the 5200), until a tweaked version (the 603e) with an enlarged, 32KB split cache was released. The 603e performed better on emulated 68K code, so it saw widespread use in the PowerBook line.

This section will take a quick look at the microarchitecture of the 603e, illustrated in Figure 6-2, because it was the version of the 603 that saw the most widespread use.

NOTE *The 604 was also released at the same time as the original 603. The 604, which was intended for Apple's high-end products just like the 603e was intended for its low-end products, was yet another brand new design. We'll cover the 604 in "The PowerPC 604" on page 123.*

The 603e's Back End

Like the 601, the 603e sports the classic RISC four-stage pipeline. But unlike the 601, which can decode and dispatch up to three instructions per cycle to any of its execution units—including its branch unit—the 603e has one important restriction that constrains how it uses its dispatch bandwidth of three instructions per cycle.

On the 603e, and on all processors derived from it (the 750 and the 7400/7410), branches that aren't folded or don't fall through are dispatched from the instruction queue to the branch unit over a *dispatch bus* that isn't connected to any of the other execution units. This way, branch instructions don't take up any of the available dispatch bandwidth that feeds the main part of the back end. The 603e and its derivatives can dispatch one branch instruction per cycle to the branch unit over this particular bus.

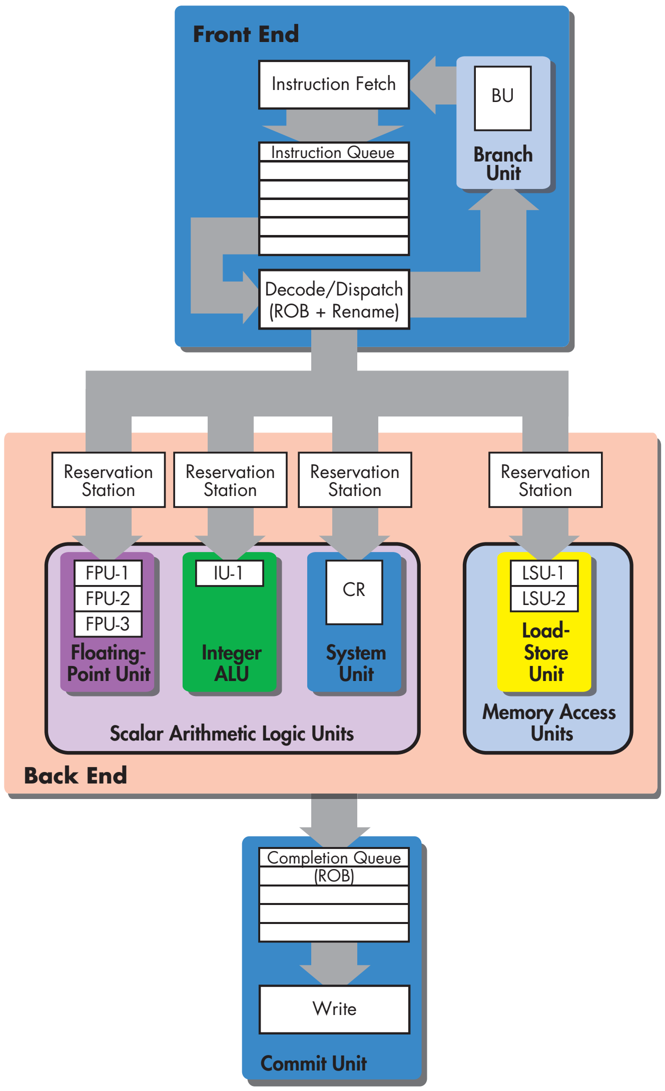


Figure 6-2: Microarchitecture of the PowerPC 603e

Non-branch instructions can dispatch at a rate of up to two instructions per cycle to the back end, which means that the 603 has a maximum dispatch rate of three instructions per cycle (two non-branch + one branch). However, because two non-branch instructions per cycle can dispatch, branch instructions are often ignored when discussing the dispatch rate of the 603 and its successors. Therefore, these processors are often said to have a dispatch rate of up to two instructions per cycle, even though the dispatch rate is *technically* three instructions per cycle.

The 603e's dispatch logic takes a maximum of two non-branch instructions per cycle from the bottom of the instruction queue and passes them to the back end, where they are executed by one of five execution units:

- Integer unit
- Floating-point unit
- Branch unit
- Load-store unit
- System unit

Notice that this list contains two more units than the analogous list for the 601: the *load-store unit (LSU)* and the *system unit*. The 603e's load-store unit takes over all of the address-calculating labors that the older 601 foisted onto its lone integer ALU. Because the 603e has a dedicated LSU for performing address calculations and executing store-data operations, its integer unit is freed up from having to handle memory traffic and can therefore focus solely on integer arithmetic. This helps improve the 603e's performance on integer code.

The 603e's dedicated system unit also takes over some of the functions of the 601's integer unit, in that it handles updates to the PowerPC condition register. We'll talk more about the condition register in Chapter 10, so don't worry if you don't know what it is. The 603e's system unit also contains a limited integer adder, which can take some of the burden off the integer ALU by doing certain types of addition. (The original 603's system unit lacked this feature.)

The 603e's basic floating-point pipeline differs from that of the 601 in that it has one more execute stage and one less decode stage. Most floating-point instructions have a three-cycle latency (and a one-cycle throughput) on the 603e, compared to a two-cycle latency on the 601. This three-cycle latency/one-cycle throughput design wouldn't be bad at all if it weren't for one serious problem: At its very fastest, the 603e's FPU can only execute three instructions every four cycles. In other words, after every third single-cycle floating-point instruction, there is a mandatory pipeline bubble. I won't get into the reason for this, but the 603e's FPU took a nontrivial hit to performance for this three-instruction/four-cycle design.

The other, perhaps more serious, flaw in the 603e's FPU is that it is not fully pipelined for multiply operations. Double-precision multiplies—and this includes double-precision `fmadds`—spend two cycles in the execute stage, which means that the 603e's FPU can complete only one double-precision multiply every two cycles.

603e's floating-point unit isn't all bad news, though. It still has the standard PPC ability to do single-precision `fmadd` operations, with a four-cycle latency and a one-cycle throughput. This fast `fmadd` ability helped the architecture retain much of its usefulness for DSP, scientific, and media applications, in spite of the aforementioned drawbacks.

The 603e's Front End, Instruction Window, and Branch Prediction

Up to two instructions per cycle can be fetched into the 603e's six-entry instruction queue. From there, a maximum of two instructions per cycle (one fewer than the 601) can be dispatched from the two bottom entries in the IQ to the reservation stations in the 603e's back end.

Not only does the 603e dispatch one fewer instruction per cycle to its back end than the 601 does, but its overall approach to superscalar and out-of-order execution differs from that of the 601 in another way, as well. The 603e uses a dedicated *commit unit*, which contains a five-entry *completion queue* (analogous to the P6's ROB) for keeping track of the program order of in-flight instructions. When instructions execute out of order, the commit unit refers to the information stored in the completion queue and puts the instructions back in program order before committing them.

To use a term that figured prominently in our discussion of the Pentium, the 603 is the first PowerPC processor to feature dynamic scheduling via a full-blown *instruction window*, complete with a ROB and reservation stations. We'll talk more about the concept of the instruction window and about the structures that make it up (the ROB and the reservation stations) in the next section on the 604. For now, it suffices to say that the 603's instruction window is quite small compared to that of its successors—three of its four reservation stations are only single-entry, and one is double-entry (the one attached to the load-store unit). Because the 603's instruction window is so small, it needs relatively few rename registers to temporarily hold execution results prior to commitment. The 603 has five general-purpose rename registers, four floating-point rename registers, and one rename register each for the condition register (CR), link register (LR), and count register (CTR).

The 603 and 603e follow the 601 in their ability to do speculative execution by means of a simple, static branch predictor. Like the static predictor on the 601, the 603e's predictor marks forward branches as not taken and backward branches as taken. This static branch predictor is simple and fast, but it is only mildly effective compared to even a weakly designed dynamic branch predictor. If PPC users in the 603e/604 era wanted dynamic branch prediction, they had to upgrade to the 604.

Summary: The 603 and 603e in Historical Context

With its stellar performance-per-watt ratio, the 603 was a great little processor, and it would have made a good low- to midrange desktop processor as well if it weren't for Apple's legacy 68K code base. The 603e's tweaks and larger cache size helped with the legacy problems somewhat, but the updated chip still played second fiddle in Apple's product line to the larger, much more powerful 604.

You haven't seen the last of the 603e, though. The 603e's design formed the basis for what would eventually become Motorola's PowerPC 7400—aka the G4—which we'll cover in "The PowerPC 7400 (aka the G4)" on page 133.

The PowerPC 604

At the same time the 603 was making its way toward the market, the 604 was in the works as well. The 604 was to be Apple's high-end PPC desktop processor, so its power and transistor budgets were much higher than that of the 603. Table 6-3 summarizes the 604's features, and a quick glance at a diagram of the 604 (see Figure 6-3) shows some obvious ways that it differs from its lower-end sibling. For example, in the front end, the length of the instruction queue has been increased by two entries. In the back end, two more integer units have been added, and the CR logical unit has been removed. These changes reflect some important differences in the overall approach of the 604, differences that will be examined in greater detail shortly.

Table 6-3: Features of the PowerPC 604 and 604e

	PowerPC 604	PowerPC 604e
Introduction Date	May 1, 1995	July 19, 1996
Process	0.50 micron	0.35 micron
Transistor Count	3.6 million	5.1 million
Die Size	197 mm^2	148 mm^2
Clock Speed at Introduction	120 MHz	180–200 MHz
L1 Cache Size	32KB split L1	64KB split L1
First Appeared In	PowerMac 9500/120	Power Computing PowerTower Pro 200 (PowerMac 9500/180 on August 7, 1996)

The 604's Pipeline and Back End

The 604's pipeline is deeper than that of the 601 and the 603, and it consists of the following six stages:

Four Phases of the Standard RISC Pipeline	Six Stages of the 604's Pipeline
Fetch	1. Fetch
Decode/dispatch	2. Decode
	3. Dispatch (ROB and rename)
Execute	4. Execute
Write-back	5. Complete
	6. Write-back

In the 604, the standard RISC decode/dispatch phase is split into two stages, as is the write-back phase. I'll explain just how these two new pipeline stages work in the section on the instruction window, but for now all you need to understand is that this lengthened pipeline enables the 604 to reach higher clock speeds than its predecessors. Because each pipeline stage is simpler, it takes less time to complete, which means that the CPU's clock cycle time can be shortened.

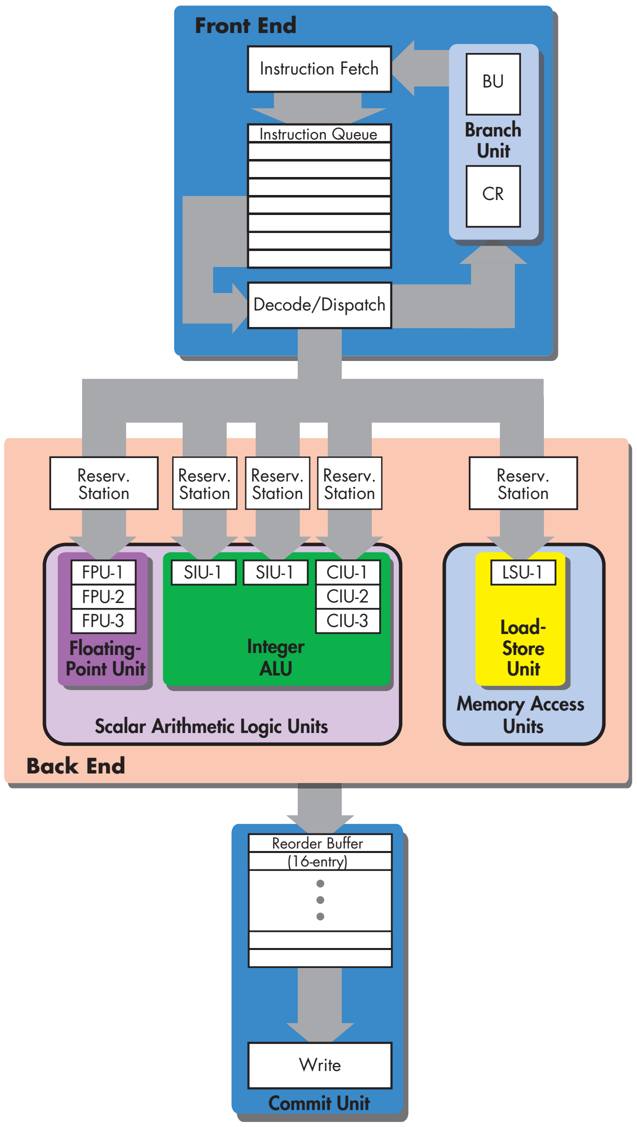


Figure 6-3: PowerPC 604 microarchitecture

Aside from the longer pipeline, another factor that really sets the 604 apart from the other 600-series PPC designs discussed so far is its wider back end. The 604 can execute up to six instructions per clock cycle in the following six execution units:

- Branch unit (BU)/condition register unit (CRU)
- Load-store unit (LSU)
- Floating-point unit (FPU)

- Three integer units (IU)
    - Two simple integer units (SIUs)
    - One complex integer unit (CIU)

Unlike the other 600-series processors, the 604 has multiple integer units. This division of labor, where multiple fast integer units executed simple integer instructions and one slower integer unit execute complex integer instructions, will be discussed in more detail in Chapter 8. Any integer instruction that takes only a single cycle to execute can pass through one of the two SIUs. On the other hand, integer instructions that take multiple cycles to execute, like integer divides, have to pass through the slower CIU.

Like the 603e, the 604 has *register renaming*, a technique that is facilitated by the 12-entry register rename file attached to the 32-entry general-purpose register file. These rename buffers allow the 604's execution units more options for avoiding false dependencies and register-related stalls.

The 604's floating-point unit does most single- and double-precision operations with a three-cycle latency, just like the 603e. Unlike the 603e, though, the 604's floating-point unit is fully pipelined for double-precision multiplies. Floating-point division and two other instructions take from 18 to 33 cycles on the 604, as on the 603e. Finally, the 604's 32-entry floating-point register file is attached to an 8-entry floating-point rename register buffer.

The 604's load-store unit (LSU) is also similar to that of the 603e. Like the 603e's LSU, it contains an adder for doing address calculations and handles all load-store traffic, but unlike the 603e, it's connected to deeper load and store queues and allows a little more flexibility for the optimal reordering of memory operations.

The 604's branch unit also features a dynamic branch prediction scheme that's a vast improvement over the 603e's static branch predictor. The 604 has a large, 512-entry *branch history table (BHT)* with two bits per entry for tracking branches, coupled with a 64-entry *branch target address cache (BTAC)*, which is the equivalent of the Pentium's BTB.

As always, the more transistors you spend on branch prediction, the better performance is, so the 604's more advanced branch unit helps it quite a bit. Still, in the case of a misprediction, the 604's longer pipeline has to pay a higher price than its shorter-pipelined predecessors in terms of performance. Of course, the bigger performance loss associated with a misprediction is also the reason the 604 needs to spend those extra resources on branch prediction.

Notice that the list of execution units on page 124 is missing a unit that is present on the 603e: the system unit. The 603e's system unit handled updates to the PPC condition register, a function that was handled by the integer execution unit on the older 601. The 604 moves the responsibility of dealing with the condition register onto the branch unit. So the 604's branch unit contains a separate execution unit that handles all logical operations that involve the PowerPC condition register. This condition register unit (CRU) shares a dispatch bus and some other resources with the branch execution unit, so it's not a fully independent execution unit like the 603e's system unit. What does this BU/CRU combination do for performance? It probably doesn't have a huge impact, but whatever impact it does have is significant enough to where the 604's immediate successor—the 604e—adds an independent execution unit to the back end for CR logical operations.

The 604's Front End and Instruction Window

The 604's front end and instruction window look like a combination of the best features of the 601 and the 603e. Like the 601, the 604's instruction queue is eight entries deep. Instructions are fetched from the L1 cache into the instruction queue, where they're decoded before being dispatched to the back end. Branches that can be folded are folded, and the 604's dispatch logic can dispatch up to four instructions per cycle (up from two on the 603e and three on the 601) from the bottom four entries of the instruction queue to the back end's execution units.

During the 604's dispatch stage, rename registers and a reorder buffer entry are assigned to each dispatching instruction. When the instruction is ready to dispatch, it's sent either directly to an execution unit or to an execution unit's reservation station, depending on whether or not its operands are available at the time of dispatch. Note that the 604 can dispatch at most one instruction to each execution unit, and there are certain rules that govern when the dispatch logic can dispatch an instruction to the back end. We'll cover these rules in more detail in a moment, but for now you need to be aware of one of the rules: An instruction cannot dispatch if the execution unit that it needs is not available.

The Issue Phase: The 604's Reservation Stations

In Figure 6-3, you probably noticed that each of the 604's execution units has a reservation station attached to it; this includes a reservation station each (not depicted) for the branch execution and condition register units that make up the branch unit. The 604's reservation stations are relatively small, two-entry (the CIU's reservation station is single-entry), first-in first-out (FIFO) affairs, but they make up the heart of the 604's instruction window, because they allow the instructions assigned to one execution unit to issue out of program order with respect to the instructions that are assigned to the other execution units.

This works as follows: The dispatch stage sends instructions into the reservation stations (i.e., the issue phase) in program order, and, with one important exception (described in the next paragraph), the instructions pass through their respective reservation stations in order. An instruction enters the top of a reservation station, and as the instructions ahead of it issue, it moves down the queue, until it eventually exits through the bottom (i.e., it issues).

Therefore, we can say each instruction issues in order with respect to the other instructions in its same reservation station. However, the various reservation stations can issue instructions at different times, with the result that instructions issue out of order from the perspective of the overall program flow.

The simple integer units function a little differently than described earlier, because they allow instructions to issue from their two-entry reservation stations out of order with respect to the other instructions in their own execution unit. So unlike other types of instructions described previously, integer instructions can move through their respective reservation stations and pipelines out of program order, not just with respect to the overall program flow, but with respect to the other instructions in their own reservation station.

The reservation stations in the 604 and its architectural successors exist to keep instructions that lack their input operand data but are otherwise ready to dispatch from tying up the instruction queue. If an instruction meets all of the other dispatch requirements (see "The Four Rules of Instruction Dispatch"), and if its assigned execution unit is available but it just doesn't yet have access to the part of the data stream that it needs, it dispatches to the appropriate execution unit's reservation station so that the instructions behind it in the instruction queue can move up and be dispatched.

The small size of the 604's reservation stations compared to similar structures on the P6 is due to the fact that the 604's pipeline is relatively short. Pipeline stalls aren't quite as devastating for performance on a machine with a 6-stage pipeline as they are on a machine with a 12-stage pipeline, so the 604 doesn't need as large of an instruction window as its super-pipelined counterparts.

The Four Rules of Instruction Dispatch

Here are the four most important rules governing instruction dispatch on the 604:

The In-Order Dispatch Rule

Before an instruction can dispatch, all of the instructions preceding that instruction must have dispatched. In other words, instructions dispatch from the instruction queue in program order. It is not until instructions have arrived at the reservation stations, where they may issue out of order to the execution units, that the original program order is disrupted.

The Issue Buffer/Execution Unit Availability Rule

Before the dispatch logic can send an instruction to an execution unit's reservation station, that reservation station must have an entry available. If an instruction doesn't need to go to a reservation station because its inputs are available at the time of dispatch, the required execution unit must have a pipeline slot available, and the unit's reservation station must be empty (i.e., there are no older instructions waiting to execute) before the instruction can be sent to the execution unit. (This rule is modified on the PowerPC 7450—aka G4e—and we'll cover the modification in "The PowerPC 7400 (aka the G4)" on page 133.)

The Completion Buffer Availability Rule

For an instruction to dispatch, there must be space available in the completion queue so that a new entry can be created for the instruction. Remember, the completion queue (or ROB) keeps track of the program order of each in-flight instruction, so any instruction that enters the out-of-order back end must be logged in the completion queue first.

The Rename Register Availability Rule

There must be enough rename registers available to temporarily store the results for each register that the instruction will modify.

If a dispatched instruction meets the requirements imposed by these rules, and if it meets the other more instruction-specific dispatch rules not listed here, it can dispatch from the instruction queue to the back end.

All of the PowerPC processors discussed in this chapter that have reservation stations are subject to (at least) these four dispatch rules, so keep these rules in mind as we talk about instruction dispatch throughout the rest of this chapter. Note that all of the processors—including the 604—have additional rules that govern the dispatch of specific types of instructions, but these four general dispatch rules are the most important.

The Completion Phase: The 604's Reorder Buffer

As with the P6 microarchitecture, the reservation stations aren't the only structures that make up the 604's instruction window. The 604 has a 16-entry reorder buffer (ROB) that performs the same function as the P6 microarchitecture's much larger 40-entry ROB.

The ROB corresponds to the simpler completion queue on older PPC processors. In the dispatch stage, not only are instructions sent to the back end's reservation stations, but entries for the dispatched instructions are allocated an entry in the ROB and a set of rename registers. In the completion stage, the instructions are put back in program order so that their results can be written back to the register file in the subsequent write-back stage. The completion stage corresponds to what I've called the *completion phase* of an instruction's lifecycle, and the write-back stage corresponds to what I've called the *commit phase*.

The 604's ROB is much smaller than the P6's ROB for the same reason that the 604's reservation stations are fewer: the 604 has a much shallower pipeline, which means that it needs a much smaller instruction window for tracking fewer in-flight instructions in order to achieve the same performance.

The trade-off for this lack of complexity and lower pipeline depth is a lower clock speed. The 6-stage 604 debuted in May 1995 at 120 MHz, while the 12-stage Pentium Pro debuted later that year (November 1995) at speeds ranging from 150 to 200 MHz.

Summary: The 604 in Historical Context

With a 32KB split L1 cache, the 604 had a much heftier cache than its predecessors, which it needed to help keep its deeper pipeline fed. The larger cache, higher dispatch and issue rate, wider back end, and deeper pipeline made for a solid RISC performer that was easily able to keep pace with its *x*86 competitors.

Still, the Pentium Pro was no slouch, and its performance was scaling well with improvements in processor manufacturing techniques. Apple needed more power from AIM to keep the pace, and more power is what they got with a minor microarchitectural revision that came to be called the 604e.

The PowerPC 604e

The 604e built on gains made by the 604 with a few core changes that included a doubling of the L1 cache size (to 32KB instruction/32KB data) and the addition of a new independent execution unit: the *condition register unit (CRU)*.

The previous 600-series processors had moved the responsibility for handling condition register logical operations back and forth among various units (the integer unit in the 601, the system unit in the 603/603e, and the branch unit in the 604). Now with the 604e, these operations got an execution unit of their own. The 604e sported a functional block in its back end that was dedicated to handling condition register logical operations, which meant that these not uncommon operations didn't tie up other execution units—like the integer unit or the branch unit—that had more serious work to do.

The 604e's branch unit, now that it was free from having to handle CR logical operations, got a few expanded capabilities that I won't detail here. The 604e's caches, in addition to being enlarged, also got additional copyback buffers and a handful of other enhancements.

The 604e was ultimately able to scale up to 350 MHz once it moved from a 0.35 to a 0.25 micron manufacturing process, making it a successful part for Apple's budding RISC media workstation line.

The PowerPC 750 (aka the G3)

The PowerPC 750—known to Apple users as the G3—is a design based heavily on the 603/603e. Its four-stage pipeline is the same as that of the 603/603e, and many of the features of its front end and back end will be familiar to you from our discussion of the older processor. Nonetheless, the 750 sports a few very powerful improvements over the 603e that make it faster than even the 604e, as you can see in Table 6-4.

Table 6-4: Features of the PowerPC 750

Introduction Date	September 1997
Process	0.25 micron
Transistor Count	6.35 million
Die Size	67 mm^2
Clock Speed at Introduction	200–300 MHz
Cache Sizes	64KB split L1, 1MB L2
First Appeared In	Power Macintosh G3

The 750's significant improvement in performance over the 603/603e is the result of a number of factors, not the least of which are the improvements that IBM made to the 750's integer and floating-point capabilities.

A quick glance at the 750's layout (see Figure 6-4) reveals that its back end is wider than that of the 603. More specifically, where the 603 has a single integer unit, the 750 has two—a simple integer unit (SIU) and complex integer unit (CIU). The 750's complex integer unit handles all integer instructions, while the simple integer unit handles all integer instructions except multiply and divide. Most of the integer instructions that execute in the SIU are single-cycle instructions.

Like the 603 (and the 604), the 750's floating-point unit can execute all single-precision floating-point operations—including multiply—with a latency of three cycles. And like the 603, early versions of the 750 had to insert a pipeline bubble after every third floating-point instruction in its pipeline; this is fixed in later IBM-produced versions of the 750. Double-precision floating-point operations, with the exception of operations involving multiplication, also take three cycles on the 750. Double-precision multiply and multiply-add operations take four cycles, because the 750 doesn't have a full double-precision FPU.

The 750's load-store unit and system register unit perform the same functions described in the preceding section for the 603, so they don't merit further comment.

The 750's Front End, Instruction Window, and Branch Instruction

The 750 fetches up to four instructions per cycle into its six-entry instruction queue, and it dispatches up to two non-branch instructions per cycle from the IQ's two bottom entries. The dispatch logic follows the four dispatch rules described earlier when deciding when an instruction is eligible to dispatch, and each dispatched instruction is assigned an entry in the 750's six-entry ROB (compare the 603's five-entry ROB).

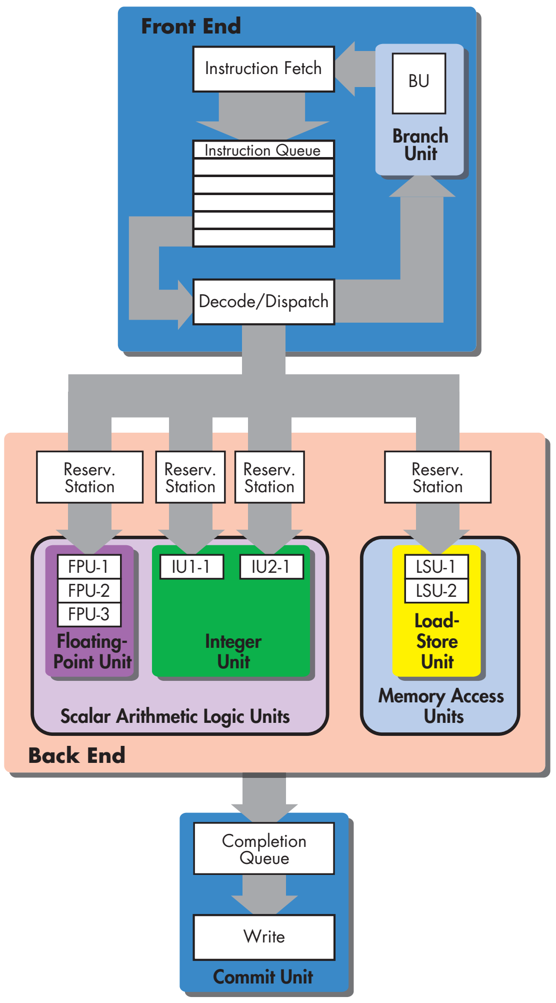


Figure 6-4: Microarchitecture of the PowerPC 750

As on the 603 and 604, newly dispatched instructions enter the reservation station of the execution unit to which they have been dispatched, where they wait for their operands to become available so that they can issue. The 750's reservation station configuration is similar to that of the 603 in that, with the exception of the two-entry reservation station attached to the 750's LSU, all of the execution units have single-entry reservation stations. And like the 603, the 750's branch unit has no reservation station.

Because the 750's instruction window is so small, it has half the rename registers of the 604. Nonetheless, the 750's six general-purpose and six floating-point rename registers still put it ahead of the 603's number of rename registers (five GPRs and four FPRs). Like the 603, the 750 has one rename register each for the CR, LR, and CTR.

You would think that the 750's smaller reservation stations and shorter ROB would put it at a disadvantage with respect to the 604, which has a larger instruction window. But the 750's pipeline is shorter than that of the 604, so it needs fewer buffers to track fewer in-flight instructions. More importantly, though, the 750 has one very clever trick up its sleeve that it uses to keep its pipeline full.

Recall that standard dynamic branch prediction schemes generally use a branch history table (BHT) in combination with a branch target buffer (BTB) to speculate on the outcome of branch instructions and to redirect the processor's front end to a different point in the code stream based on this speculation. The BHT stores information on the past behavior (taken or not taken) of the most recently executes branch instructions, so that the processor can determine whether or not it should take these branches if it encounters them again. The target addresses of recently taken branches are stored in the BTB, so that when the branch prediction hardware decides to speculatively take a branch, it has immediate access to that branch's target address without having to recalculate it. The target address of the speculatively taken branch is loaded from the BTB into the instruction register, so that on the next fetch cycle, the processor can begin fetching and speculatively executing instructions from the target address.

The 750 improves on this standard scheme in a very clever way. Instead of storing only the target addresses of recently taken branches in a BTB, the 750's 64-entry *branch target instruction cache (BTIC)* stores the *instruction* that is located at the branch's target address. When the 750's branch prediction unit examines the 512-entry BHT and decides to speculatively take a branch, it doesn't have to go to code storage to fetch the first instruction from that branch's target address. Instead, the BPU loads the branch's target instruction directly from the BTIC into the instruction queue, which means that the processor doesn't have to wait around for the fetch logic to go out and fetch the target instruction from code storage. This scheme saves valuable cycles, and it helps keep performance-killing bubbles out of the 750's pipeline.

Summary: The PowerPC 750 in Historical Context

In spite of its short pipeline and small instruction window, the 750 packed quite a punch. It managed to outperform the 604, partially because of a dedicated back-side L2 cache interface that allowed it to offload L2 traffic from the front-side bus. It was so successful that a 604 derivative was scrapped in favor of just building on the 750. The 750 and its immediate successors, all of which went under the name of *G3*, eventually found widespread use both as embedded devices and across Apple's entire product line, from its portables to its workstations.

The G3 lacked one important feature that separated it from the *x*86 competition, though: vector computing capabilities. While comparable PC processors supported SIMD in the form of Intel's and AMD's vector extensions to the *x*86 instruction set, the G3 was stuck in the world of scalar computing. So when Motorola decided to develop the G3 into an even more capable embedded and media workstation chip, this lack was the first thing it addressed.

The PowerPC 7400 (aka the G4)

The Motorola MPC7400 (aka the G4) was designed as a media processing powerhouse for desktops and portables. Apple Computer used the 7400 as the CPU in the first version of their G4 workstation line, and this processor was later replaced by a lower-power version—the 7410—before the 7450 (aka the G4+ or G4e) was introduced. Today, the successors to the 7400/7410 have seen widespread use as *embedded processors*, which means that they're used in routers and other non-PC devices that need a microprocessor with low power consumption and strong DSP capabilities. Table 6-5 lists the features of the PowerPC 7400.

Table 6-5: Features of the PowerPC 7400

Introduction Date	September 1999
Process	0.20 micron
Transistor Count	10.5 million
Die Size	83 mm^2
Clock Speed at Introduction	400–600 MHz
Cache Sizes	64KB split L1, 2MB L2 supported via on-chip tags
First Appeared In	Power Macintosh G4

Figure 6-5 illustrates the PowerPC 7400 microarchitecture.

Except for the addition of SIMD capabilities, which we'll discuss in the next chapter, the G4 is essentially the same as the 750. Motorola's technical summary of the G4 has this to say about the G4 compared to the 750:

> The design philosophy on the MPC7410 (and the MPC7400) is to change from the MPC750 base only where required to gain compelling multimedia and multiprocessor performance. The MPC7410's core is essentially the same as the MPC750's, except that whereas the MPC750 has a 6-entry completion queue and has slower performance on some floating-point double-precision operations, the MPC7410 has an 8-entry completion queue and a full double-precision FPU. The MPC7410 also adds the AltiVec instruction set, has a new memory subsystem, and can interface to the improved MPX bus.
>
> —*MPC7410 RISC Microprocessor Technical Summary, section 3.11.*

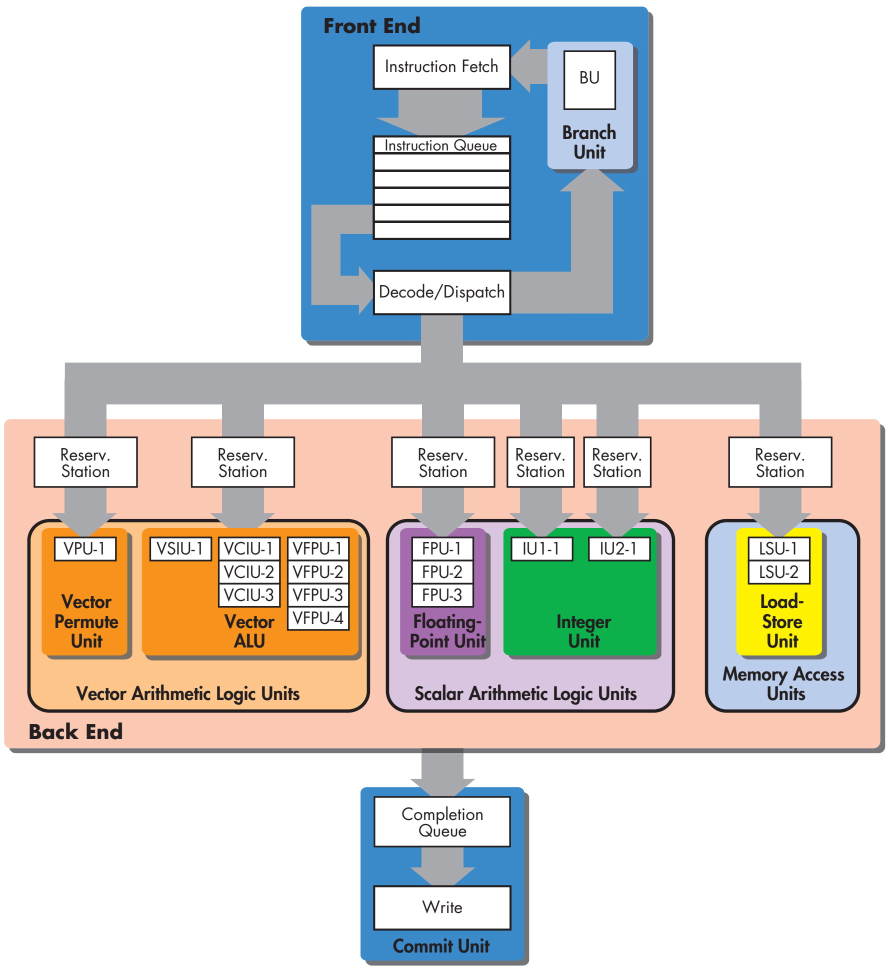


Figure 6-5: Microarchitecture of the PowerPC 7400

Aside from the vector execution unit, the most important difference in the back ends of the two units lies in the G4's improved FPU. The G4's FPU is a full-blown double-precision FPU, and it does single- and double-precision floating-point operations, including multiply and multiply-add, in three fully-pipelined cycles.

With respect to the instruction window, the G4 has the same number and configuration of reservation stations as the 750. (Note that the G4's two vector execution units, which were not present on the 750, each have a one-entry reservation station.) The only difference is that the G4's instruction queue has been lengthened to eight entries from the 750's original six as a way of reducing dispatch bottlenecks.

The G4's Vector Unit

In the late 1990s, Apple, Motorola, and IBM jointly developed a set of SIMD extensions to the PowerPC instruction set for use in the PowerPC processor series. These SIMD extensions went by different names: IBM called them VMX, and Motorola called them AltiVec. This book will refer to these extensions using Motorola's AltiVec label.

The new AltiVec instructions, which I'll cover in detail in Chapter 8, were first introduced in the G4. The G4 executes these instructions in its vector unit, which consists of two vector execution units: the *vector ALU (VALU)* and the *vector permute unit (VPU)*. The VALU performs vector arithmetic and logical operations, while the VPU performs permute and shift operations on vectors.

To support the AltiVec instructions, which can operate on up to 128 bits of data at a time, 32 new 128-bit vector registers were added to the PowerPC ISA. On the G4, these 32 architectural registers are accompanied by 6 vector rename registers.

Summary: The PowerPC G4 in Historical Context

The G4's AltiVec instruction set was a hit, and it began to see widespread use by Apple and by Motorola's embedded customers. But there was still much room for improvement to the G4's AltiVec implementation. In particular, the vector unit's single VALU was tasked with handling all integer and floating-point vector operations. Just like scalar code benefits from the presence of multiple specialized scalar ALUs, vector performance could be improved by splitting the burden of vector computation among multiple specialized VALUs operating in parallel. Such an improvement would have to wait for the successor to the G4—the G4e.

The major problem with the G4 was that its short, four-stage pipeline severely limited the upward scalability of its clock rate. While Intel and AMD were locked in the gigahertz race, Motorola's G4 was stuck around the 500 MHz mark for quite a long time. As a result, Apple's *x*86 competitors soon surpassed it in both clock speed and performance, leaving what was once the most powerful commodity RISC workstation line in serious trouble with the market.

Conclusion

The 600 series saw the PPC line go from the new kid on the block to a mature RISC alternative that brought Apple's PowerMac workstation to the forefront of personal computing performance. While the initial 601 had a few teething problems, the line was in great shape after the 603e and 604e made it to market. The 603e was a superb mobile chip that worked well in Apple's laptops, and even though it had a more limited instruction dispatch/commit bandwidth and a smaller cache than the 601, it still managed to beat its predecessor because of its more efficient use of transistors.

The 604 doubled the 603's instruction dispatch and commit bandwidth, and it sported a wider back end and a larger instruction window that enabled its back end to grind through more instructions per clock. Furthermore, its pipeline was deepened in order to increase the number of instructions per clock and to allow for better clock speed scaling. The end result was that the 604 was a strong enough desktop chip to keep the PowerMac comfortably in the performance game.

It's important to remember, though, that the 600 series reigned at a time when transistor budgets were still relatively small by today's standards, so the PowerPC architecture's RISC nature gave it a definite cost, performance, and power consumption edge over the *x*86 competition. This is not to say that the 600 series was always in the performance lead; it wasn't. The performance crown changed hands a number of time during this period.

During the heyday of the 600 series and into the dawn of the G3 era, the fact that PowerPC was a RISC ISA was a strong mark in the platform's favor. But as Moore's Curves drove transistor counts and MHz numbers ever higher, the relative cost of legacy *x*86 support began to go down and the PowerPC ISA's RISC advantage started to wane. By the time the 7400 hit the market, *x*86 processors from Intel and AMD were already catching up to it in performance, and by the time the gigahertz race was over, Apple's flagship workstation line was in trouble. The 7400's clock speed and performance had stagnated for too long during a period when Intel and AMD were locked in a heated price/performance competition.

Apple's stop-gap solution to this problem was to turn to *symmetric multiprocessing (SMP)* in order to increase the performance of its desktop line. (See Chapter 12 for a more detailed discussion of SMP.) By offering computers in which two G4s worked together to execute code and process data, Apple hoped to pack more processing power into its computers in a way that didn't rely on Motorola to ramp up clock speeds. The dual G4 met with mixed success in the market, and it wasn't until the debut of the significantly redesigned PowerPC 7450 (aka G4+ or G4e) that Apple saw the per-processor performance of its workstations improve. The introduction of the G4e into its workstation line enabled Apple to recover some ground in its race with its primary competitor in the PC space—systems based on Intel's Pentium 4.

7

INTEL'S PENTIUM 4 VS. MOTOROLA'S G4E: APPROACHES AND DESIGN PHILOSOPHIES

Now that we've covered not only the microprocessor basics but also the development of two popular *x*86 and PowerPC processor lines, you're equipped to compare and to understand two of the processors that have been among the most popular examples of these two lines: Intel's Pentium 4 and Motorola's G4e.

When the Pentium 4 hit the market in November 2000, it was the first major new *x*86 microarchitecture from Intel since the 1995 introduction of the Pentium Pro. In the years prior to the Pentium 4's launch, the Pentium Pro's P6 core dominated the market in its incarnations as the Pentium II and Pentium III, and anyone who was paying attention during that time learned at least one major lesson: Clock speed sells. Intel was definitely paying attention, and as the Willamette team members labored away in Hillsboro, Oregon, they kept MHz foremost in their minds. This singular focus is evident in everything from Intel's Pentium 4 promotional and technical literature down to the very last detail of the processor's design. As this chapter will show, the successor to the most successful *x*86 microarchitecture of all time was a machine built from the ground up for stratospheric clock speed.

NOTE Willamette *was Intel's code name for the Pentium 4 while the project was in development. Intel's projects are usually code-named after rivers in Oregon. Many companies use code names that follow a certain convention, like Apple's use of the names of large cats for versions of OS X.*

Motorola introduced MPC7450 in January 2001, and Apple quickly adopted it under the *G4* moniker. Because the 7450 represented a significant departure from the 7400, the 7450 was often referred to as the G4e or the G4+, so throughout this chapter we'll call it the G4e. The new processor had a slightly deeper pipeline, which allowed it to scale to higher clock speeds, and both its front end and back ends boasted a whole host of improvements that set it apart from the original G4. It also continued the excellent performance/power consumption ratio of its predecessors. These features combined to make it an excellent chip for portables, and Apple has exploited derivatives of this basic architecture under the G4 name in a series of innovative desktop enclosure designs and portables. The G4e also brought enhanced vector computing performance to the table, which made it a great platform for DSP and media applications.

This chapter will examine the trade-offs and design decisions that the Pentium 4's architects made in their effort to build a MHz monster, paying special attention to the innovative features that the Pentium 4 sported and the ways that those features fit with the processor's overall design philosophy and target application domain. We'll cover the Pentium 4's ultradeep pipeline, its trace cache, its double-pumped ALUs, and a host of other aspects of its design, all with an eye to their impact on performance. As a point of comparison, we'll also look at the microarchitecture of Motorola's G4e. By examining two microprocessor designs side by side, you'll gain a deeper understanding of how the concepts outlined in the previous chapters play out in a pair of popular, real-world designs.

The Pentium 4's Speed Addiction

Table 7-1 lists the features of the Pentium 4.

Table 7-1: Features of the Pentium 4

Introduction Date	April 23, 2001
Process	0.18 micron
Transistor Count	42 million
Clock Speed at Introduction	1.7 GHz
Cache Sizes	L1: Approximately 16KB instruction, 16KB data
Features	Simultaneous Multithreading (SMT, aka "hyperthreading") added in 2003. 64-bit support (EM64T) and SSE3 added in 2004. Virtualization Technology (VT) added in 2005.

While some processors still have the classic, four-stage pipeline, described in Chapter 1, most modern CPUs are more complicated. You've already seen how the original Pentium had a second decode stage, and the P6 core tripled the standard four-stage pipeline to 12 stages. The Pentium 4, with a whopping 20 stages in its basic pipeline, takes this tactic to the extreme. Take a look at Figure 7-1. The chart shows the relative clock frequencies of Intel's last six *x*86 designs. (This picture assumes the same manufacturing process for all six cores.) The vertical axis shows the relative clock frequency, and the horizontal axis shows the various processors relative to each other.

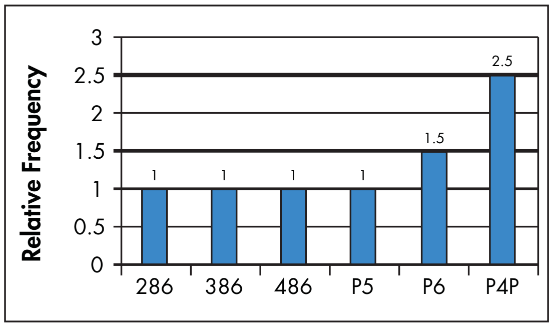


Figure 7-1: The relative frequencies of Intel's processors

Intel's explanation of this diagram and the history it illustrates is enlightening, as it shows where their design priorities were:

> Figure [3.2] shows that the 286, Intel386™, Intel486™, and Pentium® (P5) processors had similar pipeline depths—they would run at similar clock rates if they were all implemented on the same silicon process technology. They all have a similar number of gates of logic per clock cycle. The P6 microarchitecture lengthened the processor pipelines, allowing fewer gates of logic per pipeline stage, which delivered significantly higher frequency and performance. The P6 microarchitecture approximately doubled the number of pipeline stages compared to the earlier processors and was able to achieve about a 1.5 times higher frequency on the same process technology. The NetBurst microarchitecture was designed to have an even deeper pipeline (about two times the P6 microarchitecture) with even fewer gates of logic per clock cycle to allow an industry-leading clock rate.
>
> —*The Microarchitecture of the Pentium 4 Processor, p. 3.*

As you learned in Chapter 2, there are limits to how deeply you can pipeline an architecture before you begin to reach a point of diminishing returns. Deeper pipelining results in an increase in instruction execution time; this increase can be quite damaging to instruction completion rates if the pipeline has to be flushed and refilled often. Furthermore, in order to realize the throughput gains that deep pipelining promises, the processor's clock speed must increase in proportion to its pipeline depth. But in the real world, speeding up the clock of a deeply pipelined processor to match its pipeline depth is not all that easy.

Because of these drawbacks to deep pipelining, many critics of the Pentium 4's microarchitecture, dubbed NetBurst by Intel, have suggested that its staggeringly long pipeline was a gimmick—a poor design choice made for reasons of marketing and not performance and scalability. Intel knew that the public naïvely equated higher MHz numbers with higher performance, or so the argument went, so they designed the Pentium 4 to run at stratospheric clock speeds and in the process, made design trade-offs that would prove detrimental to real-world performance and long-term scalability.

As it turns out, the Pentium 4's critics were both wrong and right. In spite of the predictions of its most ardent detractors, the Pentium 4's performance has scaled fairly well with its clock rate, a phenomenon that readers of this book would expect given the section "Pipelining Explained" on page 40. But though they were wrong about its performance, the Pentium 4's critics were right about the origins of the processor's deeply pipelined approach. Revelations from former members of the Pentium 4's design team, as well as my own off-the-record conversations with Intel folks, all indicate that the Pentium 4's design was the result of a marketing-driven focus on clock speeds at the expense of actual performance and long-term scalability.

It's my understanding that this fact was widely known within Intel, even though it was not, and probably never will be, publicly acknowledged. We now know that during the course of the Pentium 4's design, the design team was under pressure from the marketing folks to turn out a chip that would give Intel a massive MHz lead over its rivals. The reasoning apparently went that MHz was a single number that the general public understood, and they knew that, just like with everything in the world—except for golf scores—higher numbers are somehow better.

When it comes to processor clock speeds, higher numbers are indeed better, but industry-wide problems with the transition to a 90-nanometer process caused problems for NetBurst, which on the whole relied on ever-increasing clock rates and ever-rising power consumption to maintain a performance edge over its rivals. As Intel ran into difficulties keeping up the regularly scheduled increases in the Pentium 4's clock rate, the processor's performance increases began to level off, even as its power consumption continued to rise.

Regardless of the drawbacks of the NetBurst architecture and its long-term prospects, the Pentium 4 line of processors has been successful from both commercial and performance standpoints. This is because the Pentium 4's way of doing things has advantages for certain types of applications—especially 3D and streaming media applications—even though it carries with it serious risks.

The General Approaches and Design Philosophies of the Pentium 4 and G4e

The drastic difference in pipeline depth between the G4e and the Pentium 4 reflects some very important differences in the design philosophies and goals of the two processors. Both processors try to execute as many instructions as quickly as possible, but they attack this problem in two different ways.

The G4e's approach to performance can be summarized as "wide and shallow." Its designers added more functional units to its back end for executing instructions, and its front end tries to fill up these units by issuing instructions to each functional unit in parallel. In order to extract the maximum amount of *instruction-level parallelism (ILP)* from the linear code stream, the G4e's front end first moves a small batch of instructions onto the chip. Then, its out-of-order (OOO) execution logic examines them for hazard-causing dependencies, spreads them out to execute in parallel, and then pushes them through the back end's nine execution units. Each of the G4e's execution units has a fairly short pipeline, so the instructions take very few cycles to move through and finish executing. Finally, in the G4e's final pipeline stages, the instructions are put back in their original program order before the results are written back to memory.

At any given moment, the G4e can have up to 16 instructions simultaneously spread throughout the chip in various stages of execution. As you'll see when we look at the Pentium 4, this instruction window is quite small. The end result is that the G4e focuses on getting a small number of instructions onto the chip at once, spreading them out widely to execute in parallel, and then getting them off the chip in as few cycles as possible. This "wide and shallow" approach is illustrated in Figure 7-2.

The Pentium 4 takes a "narrow and deep" approach to moving through the instruction stream, as illustrated in Figure 7-3. The fact that the Pentium 4's pipeline is so deep means that it can hold and work on quite a few instructions at once, but instead of spreading these instructions out more widely to execute in parallel, it pushes them through its narrower back end at a higher rate.

It's important to note that in order to keep the Pentium 4's fast back end fed with instructions, the processor needs deep buffers that can hold and schedule an enormous number of instructions. The Pentium 4 can have up to 126 instructions in various stages of execution simultaneously. This way, the processor can have many more instructions on chip for the OOO execution logic to examine for dependencies and then rearrange to be rapidly fired to the execution units. Or, another way of putting this is to say that the Pentium 4's instruction window is very large.

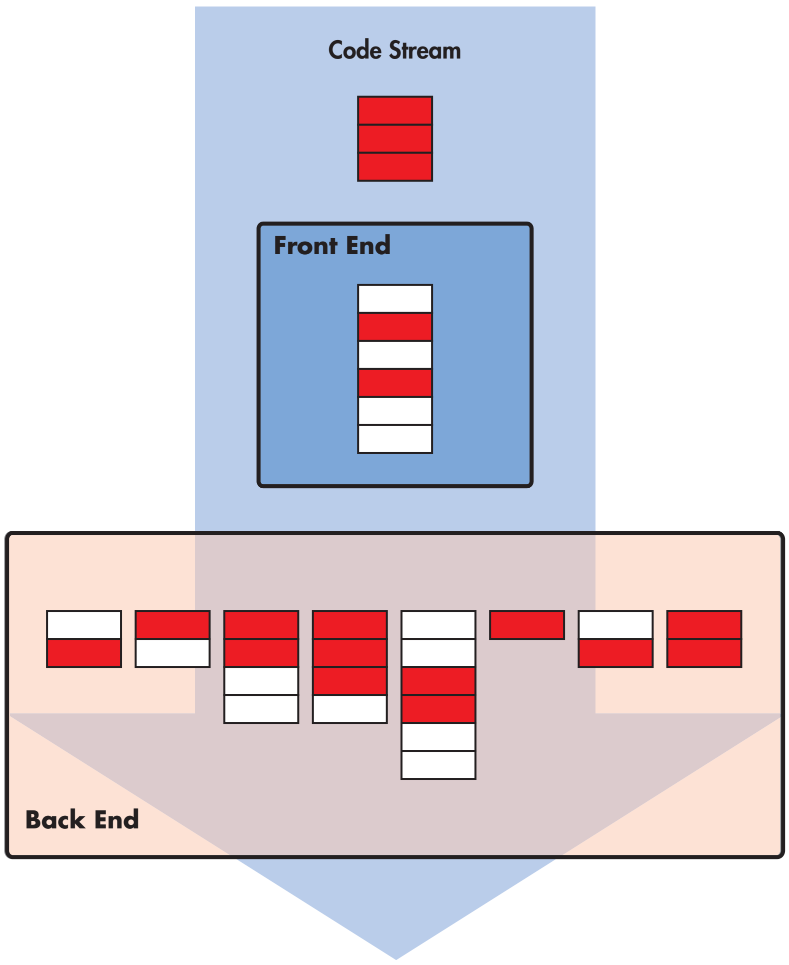


Figure 7-2: The G4e's approach to performance

It might help you to think about these two approaches in terms of a fast food drive-through analogy. At most fast food restaurants, you can either walk in or drive through. If you walk in, there are five or six short lines that you can get in and wait to have your order processed by a single server in one long step. If you choose to drive through, you'll wind up on a single long line, but that line is geared to move faster because more servers process your order in more, quicker steps. In other words:

1. You pull up to the speaker and tell them what you want.
2. You pull up to a window and pay a cashier.
3. You drive around and pick up your order.

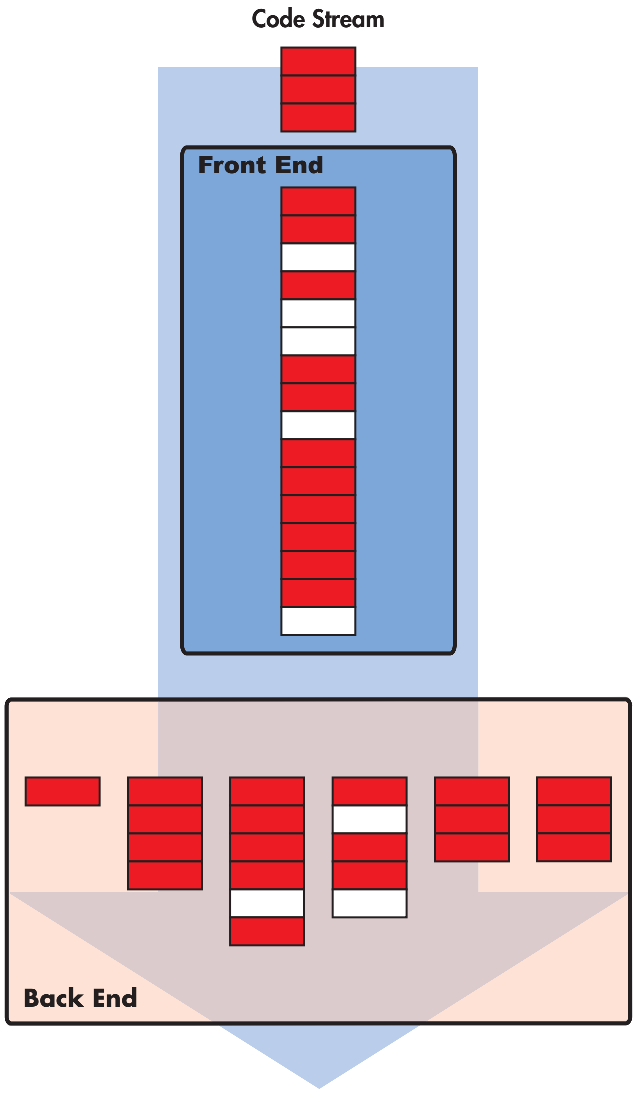


Figure 7-3: The Pentium 4's approach to performance

Because the drive-through approach splits the ordering process up into multiple, shorter stages, more customers can be waited on in a single line because there are more stages of the ordering process for different customers to find themselves in. The G4e takes the multiline, walk-in approach, while the Pentium 4 takes the single-line, drive-through approach.

As we've already discussed, the more deeply pipelined a machine is, the more severe a problem pipeline bubbles and pipeline fills become. When the Pentium 4's designers set high clock speeds as their primary goal in crafting the new microarchitecture, they had to do a lot of work to keep the pipeline from stalling and to keep branches from being mispredicted. The Pentium 4's enormous branch prediction resources and deep buffers represent a place where the Pentium 4 spends a large number of transistors to alleviate the negative effects of its long pipeline, transistors that the G4e spends instead on added execution units.

An Overview of the G4e's Architecture and Pipeline

The diagram in Figure 7-4 shows the basics of the G4e's microarchitecture, with an emphasis on representing the pipeline stages of the front end and back end. You might want to mark this page so you can refer to it throughout this section.

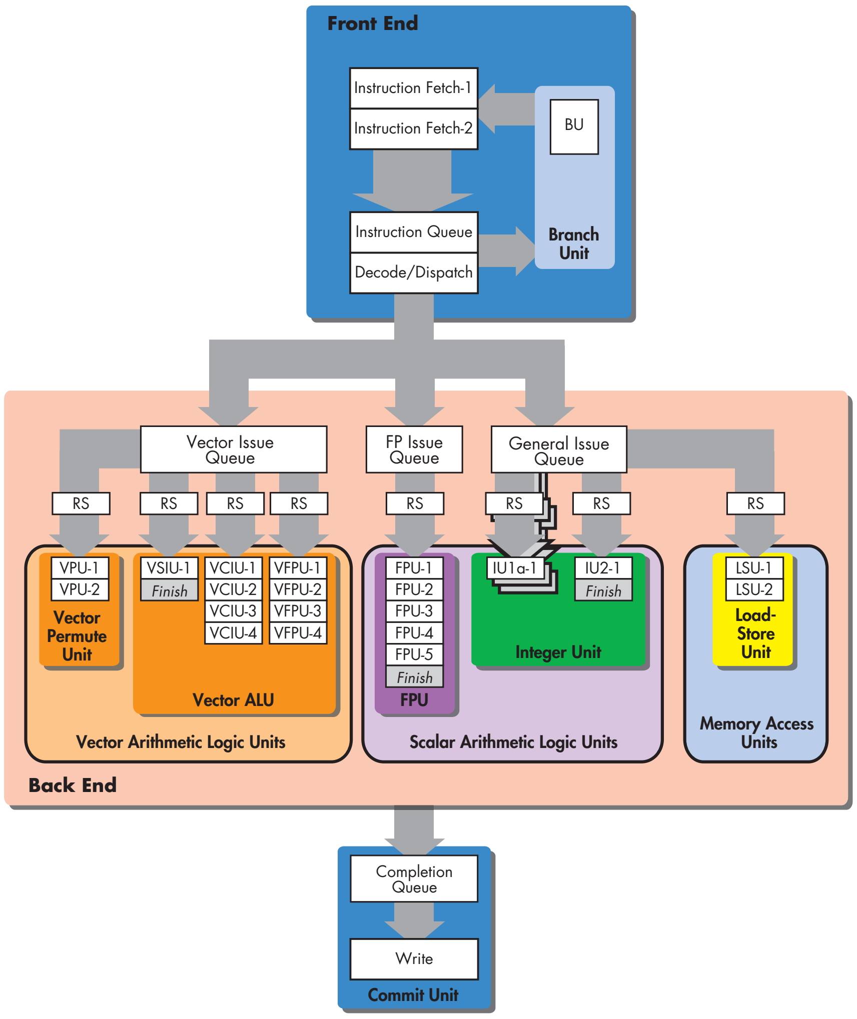


Figure 7-4: The basic microarchitecture of the G4e

Before instructions can enter the G4e's pipeline, they have to be available in its 32KB instruction cache. This instruction cache, together with the 32KB data cache, makes up the G4e's 64KB L1 cache. An instruction leaves the L1 and goes down through the various front-end stages until it hits the back end, at which point it's executed by one of the G4e's eight execution units (not counting the branch execution unit, which we'll talk about in a second).

As I've already noted, the G4e breaks down the G4's classic, four-stage pipeline into seven, shorter stages:

G4		G4e	
1	Fetch	1	Fetch-1
		2	Fetch-2
2	Decode/dispatch	3	Decode/dispatch
		4	Issue
3	Execute	5	Execute
		6	Complete
4	Write-back	7	Write-back (Commit)

Notice that the G4e dedicates one pipeline stage each to the characteristic issue and complete phases that bracket the out-of-order execution phase of a dynamically scheduled instruction's lifecycle.

Let's take a quick look at the basic pipeline stages of the G4e, because this will highlight some of the ways in which the G4e differs from the original G4. Also, an understanding of the G4e's more classic RISC pipeline will provide you with a good foundation for our upcoming discussion of the Pentium 4's much longer, more peculiar pipeline.

Stages 1 and 2: Instruction Fetch

These two stages are both dedicated primarily to grabbing an instruction from the L1 cache. Like its predecessor, the G4, the G4e can fetch up to four instructions per clock cycle from the L1 cache and send them on to the next stage. Hopefully, the needed instructions are in the L1 cache. If they aren't in the L1 cache, the G4e has to hit the much slower L2 cache to find them, which can add up to nine cycles of delay into the instruction pipeline.

Stage 3: Decode/Dispatch

Once an instruction has been fetched, it goes into the G4e's 12-entry instruction queue to be decoded. Once instructions are decoded, they're dispatched at a rate of up to three non-branch instructions per cycle to the proper *issue queue*.

Note that the G4e's dispatch logic dispatches instructions to the issue queues in accordance with "The Four Rules of Instruction Dispatch" on page 127. The only modification to the rules is in the issue buffer rule; instead of requiring that the proper execution unit and reservation station be available before an instruction can be dispatched, the G4e requires that there be space in one of the three issue queues.

Stage 4: Issue

The issue stage is the place where the G4e differs the most from the G4. Specifically, the presence of the G4e's three issue queues endows it with power and flexibility that the G4 lacks.

As you learned in Chapter 6, instructions can stall in the original G4's dispatch stage if there is no execution unit available to take them. The G4e eliminates this potential dispatch stall condition by placing a set of buffers, called *issue queues*, in between the dispatch stage and the reservation stations. On the G4e, it doesn't matter if the execution units are busy and their reservation stations are full; an instruction can still dispatch to the back end if there is space in the proper issue queue.

The six-entry *general issue queue (GIQ)* feeds the integer ALUs and can accept up to three instructions per cycle from the dispatch unit. It can also issue up to three instructions per cycle *out of order* from its bottommost three entries to any of the G4e's three integer units or to its LSU.

The four-entry *vector issue queue (VIQ)* can accept up to two instructions per cycle from the dispatch unit, and it can issue up to two instructions per cycle from its bottommost two entries to any two of the four vector execution units. But note that unlike the GIQ, instructions must issue *in order* from the bottom of the VIQ.

Finally, the single-entry *floating-point issue queue (FIQ)* can accept one instruction per cycle from the dispatch unit, and it can issue one instruction per cycle to the FPU.

With the help of the issue queues, the G4e's dispatcher can keep dispatching instructions and clearing the instruction queue, even if the execution units and their attached reservation stations are full. Furthermore, the GIQ's out-of-order issue ability allows integer and memory instructions in the code stream to flow around instructions that are stalled in the execute phase, so that a stalled instruction doesn't back up the pipeline and cause pipeline bubbles. For example, if a multicycle integer instruction is stalled in the bottom GIQ entry because the complex integer unit is busy, single-cycle integer instructions and load/store instructions can continue to issue to the simple integer units and the LSU from the two slots behind the stalled instruction.

Stage 5: Execute

The execute stage is pretty straightforward. Here, the instructions pass from the reservation stations into the execution units to be executed. Floating-point instructions move into the floating-point execution unit, vector instructions move into one of the four AltiVec units, integer instructions move into one of the G4e's four integer execution units, and memory accesses move into the LSU. We'll talk about these units in a bit more detail when we discuss the G4e's back end.

Stages 6 and 7: Complete and Write-Back

In these two stages, the instructions enter the completion queue to be put back into program order, and their results are written back to the register file. It's important that the instructions are rearranged to reflect their original ordering so that the illusion of in-order execution is maintained. The user needs to think that the program's commands were executed one after the other, the way they were written.

Branch Prediction on the G4e and Pentium 4

The G4e and the Pentium 4 each use both static and dynamic branch prediction techniques to prevent mispredictions and branch delays. If a branch instruction does not have an entry in the BHT, both processors will use static prediction to decide which path to take. If the instruction does have a BHT entry, dynamic prediction is used. The Pentium 4's BHT is quite large; at 4,000 entries, it has enough space to store information on most of the branches in an average program.

The earlier PIII's branch predictor had a success rate of around 91 percent, and the Pentium 4 allegedly uses an even more advanced algorithm to predict branches, so it should perform even better. The Pentium 4 also uses a BTB to store predicted branch targets. Note that in most of Intel's literature and diagrams, the BTB and BHT are combined under the label *the front-end BTB*.

The G4e has a BHT size of 2,000 entries, up from 512 entries in the original G4. I don't have any data on the G4e's branch prediction success rate, but I'm sure it's fairly good. The G4e has a 128-entry BTIC, which is twice as large as the original G4's 64-entry BTIC. The G4e's BTIC stores the first four instructions in the code stream starting at each branch target, so it goes even further than the original G4 in preventing branch-related pipeline bubbles.

Because of its long pipeline, the Pentium 4 has a *minimum misprediction penalty* of 20 clock cycles for code that's in the L1 cache—that's the minimum, but the damage can be much worse, especially if the correct branch can't be found in the L1 cache. (In such a scenario, the penalty is upward of 30 cycles.) The G4e's seven-stage pipeline doesn't pay nearly as high of a price for misprediction as the Pentium 4, but it does take more of a hit than its four-stage predecessor, the G4. The G4e has a minimum misprediction penalty of six clock cycles, as opposed to the G4's minimum misprediction penalty of only four cycles.

In conclusion, both the Pentium 4 and the G4e spend more resources than their predecessors on branch prediction, because their deeper pipelines make mispredicted branches a major performance killer.

The Pentium 4 and G4e do actually have one more branch prediction trick up their sleeves that's worth at least noting, even though I won't discuss it in any detail. That trick comes in the form of *software branch hints*, or extra information that a compiler or programmer can attach to conditional branch instructions. This information gives the branch predictor clues as to the expected behavior of the branch, whether the compiler or programmer expects it to be taken or not taken. There doesn't seem to be much information available on how big of a help these hints are, and Intel at least recommends that they be used sparingly since they can increase code size.

An Overview of the Pentium 4's Architecture

Even though the Pentium 4's pipeline is much longer than that of the G4e, it still performs most of the same functions. Figure 7-5 illustrates the Pentium 4's basic architecture so that you can compare it to the picture of the G4e presented in Figure 7-4. Due to space and complexity constraints, I haven't attempted to show each pipeline stage individually like I did with the G4e. Rather, I've grouped the related ones together so you can get a more general feel for the Pentium 4's layout and instruction flow.

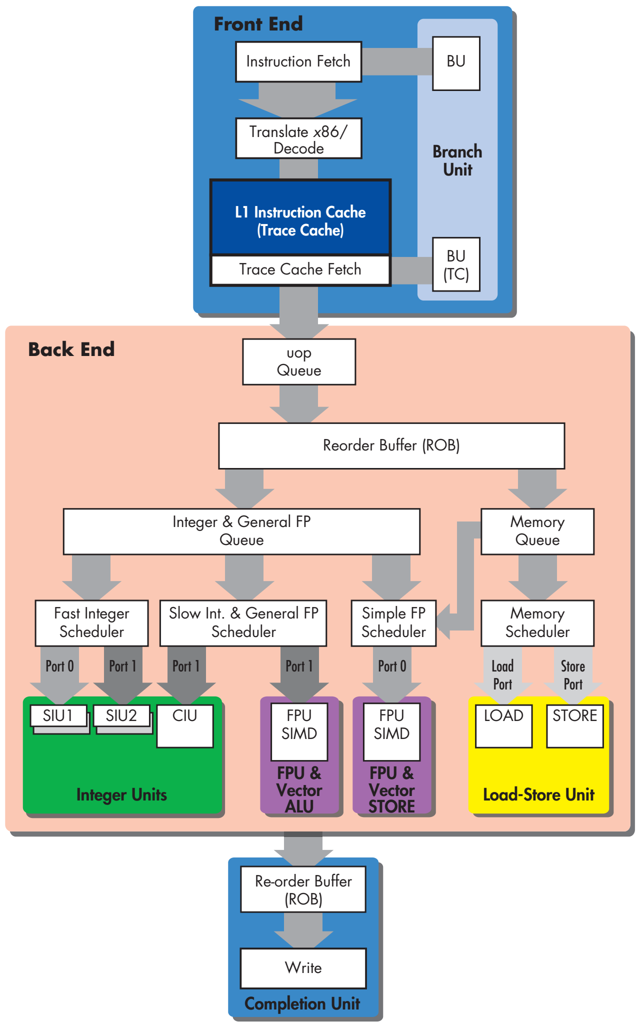


Figure 7-5: Basic architecture of the Pentium 4

The first thing to notice about Figure 7-5 is that the L1 instruction cache is actually sitting after the fetch and decode stages in the Pentium 4's front end. This oddly located instruction cache—called the *trace cache*—is one of the Pentium 4's most innovative and important features. It also greatly affects the Pentium 4's pipeline and basic instruction flow, so you have to understand it before we can talk about the Pentium 4's pipeline in detail.

Expanding the Instruction Window

Chapter 5 talked about the buffering effect of deeper pipelining on the P6, and how it allows the processor to smooth out gaps and hiccups in the code stream. The analogy I used was that of a reservoir, which can smooth out interruptions in the flow of water from a central source.

One of the innovations that makes this reservoir approach effective is the decoupling of the back end from the front end by means of the reservation station (RS). The RS is really the heart of the reservoir approach, a place where instructions can collect in a pool and then issue when their data become available. This *instruction pool* is what decouples the P6's fetch/decode bandwidth from its execution bandwidth by enabling the P6 to continue executing instructions during short periods when the front end gets hung up in either fetching or decoding the next instruction.

With the advent of the Pentium 4's much longer pipeline, the reservation station's decoupling just isn't enough. The Pentium 4's performance plummets when the front end cannot keep feeding instructions to the back end in extremely rapid succession. There's no extra time to wait for a complex instruction to decode or for a branch delay—the high-speed back end needs the instructions to flow quickly.

One route that Intel could have taken would have been to increase the size of the code reservoir, and in doing so, increase the size of the instruction window. Intel actually did do this—the Pentium 4 can track up to 126 instructions in various stages of execution—but that's not all they did. More drastic measures were required to keep the high-speed back end from depleting the reservoir before the front end could fill it.

The answer that Intel settled on was to take the costly and time-consuming *x*86 decode stage out of the basic pipeline. They did this by the clever trick of converting the L1 cache—a structure that was already on the die and therefore already taking up transistors—into a cache for decoded micro-ops.

The Trace Cache

As the previous chapter mentioned, modern *x*86 chips convert complex *x*86 instructions into a simple internal instruction format called a *micro-operation* (aka *micro-op*, *μop*, or *uop*). These micro-ops are uniform, and thus it's easier for the processor to manage them dynamically. To return to the previous chapter's Tetris analogy, converting all of the *x*86 instructions into micro-ops is kind of like converting all of the falling Tetris pieces into one or two types of simple piece, like the T and the block pieces. This makes everything easier to place, because there's less complexity to manage on the fly.

The older P6 fetches *x*86 instructions from the L1 instruction cache and translates them into micro-ops before passing them on to the reservation station to be scheduled for execution. The Pentium 4, in contrast, fetches groups of *x*86 instructions from the L2 cache, decodes them into strings of micro-ops called *traces*, and then fits these traces into its modified L1 instruction cache (the trace cache). This way, the instructions are already decoded, so when it comes time to execute them, they need only to be fetched from the trace cache and passed directly into the back end's buffers.

So the trace cache is a reservoir for a reservoir; it builds up a large pool of already decoded micro-ops that can be piped directly into the back end's smaller instruction pool. This helps keep the high-speed back end from draining that pool dry.

Shortening Instruction Execution Time

As noted earlier, on a conventional *x*86 processor like the PIII or the Athlon, *x*86 instructions make their way from the instruction cache into the decoder, where they're broken down into multiple smaller, more uniform, more easily managed instructions called micro-ops. (See the section on instruction set translation in Chapter 3.) These micro-ops are actually what the out-of-order back end rearranges, executes, and commits.

This instruction translation happens each time an instruction is executed, so it adds a few pipeline stages to the beginning of the processor's basic pipeline. Notice in Figures 7-6 and 7-7 that multiple pipeline stages have been collapsed into each other—instruction fetch takes multiple stages, translate takes multiple stages, decode takes multiple stages, and so on.

For a block of code that's executed only a few times over the course of a single program run, this loss of a few cycles to retranslation each time isn't that big of a deal. But for a block of code that's executed thousands and thousands of times (e.g., a loop in a media application that applies a series of operations to a large file), the number of cycles spent repeatedly translating and decoding the same group of instructions can add up quickly. The Pentium 4 reclaims those lost cycles by removing the need to translate those *x*86 instructions into micro-ops each time they're executed.

The Pentium 4's instruction cache takes translated, decoded micro-ops that are primed and ready to be sent straight out to the back end and arranges them into little mini-programs called traces. These traces, and not the *x*86 code that was produced by the compiler, are what the Pentium 4 executes whenever there's a trace cache hit, which is over 90 percent of the time. As long as the needed code is in the trace cache, the Pentium 4's execution path looks as in Figure 7-7.

As the front end executes the stored traces, the trace cache sends up to three micro-ops per cycle directly to the back end, without the need for them to pass through any translation or decoding stages. Only when there's a trace cache miss does that top part of the front end kick in order to fetch and decode instructions from the L2 cache. The decoding and translating steps brought on by a trace cache miss add another eight pipeline stages onto the

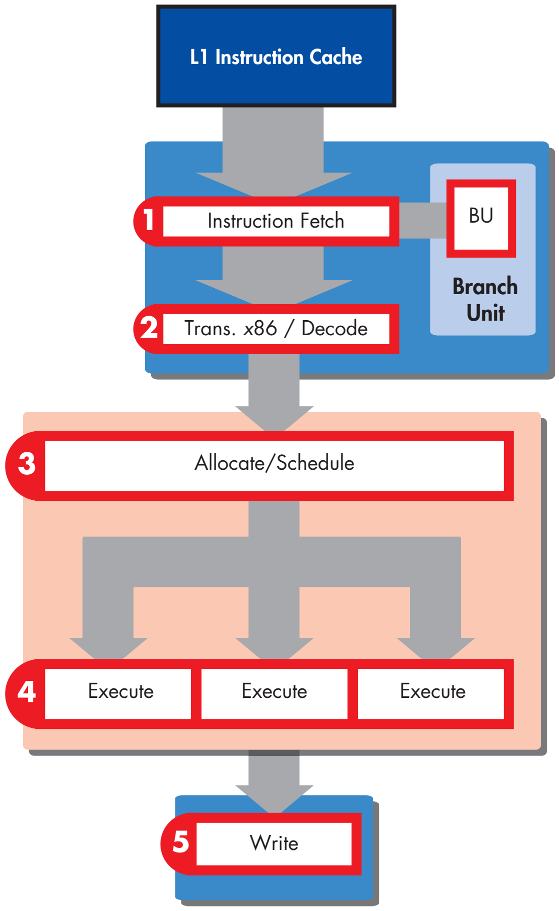


Figure 7-6: Normal x86 processor's critical execution path

beginning of the Pentium 4's pipeline. You can see that the trace cache saves quite a few cycles over the course of a program's execution, thereby shortening the average instruction execution time and average instruction latency.

The Trace Cache's Operation

The trace cache operates in two modes. *Execute mode* is the mode pictured above, where the trace cache feeds stored traces to the execution logic to be executed. This is the mode that the trace cache normally runs in. When there's an L1 cache miss, the trace cache goes into *trace segment build mode*. In this mode, the front end fetches *x*86 code from the L2 cache, translates it into micro-ops, builds a *trace segment* with it, and loads that segment into the trace cache to be executed.

Notice in Figure 7-7 that the trace cache execution path knocks the BPU out of the picture along with the instruction fetch and translate/decode stages. This is because a trace segment is much more than just a translated, decoded, predigested slice of the same *x*86 code that compiler originally produced. The trace cache actually uses branch prediction when it builds a

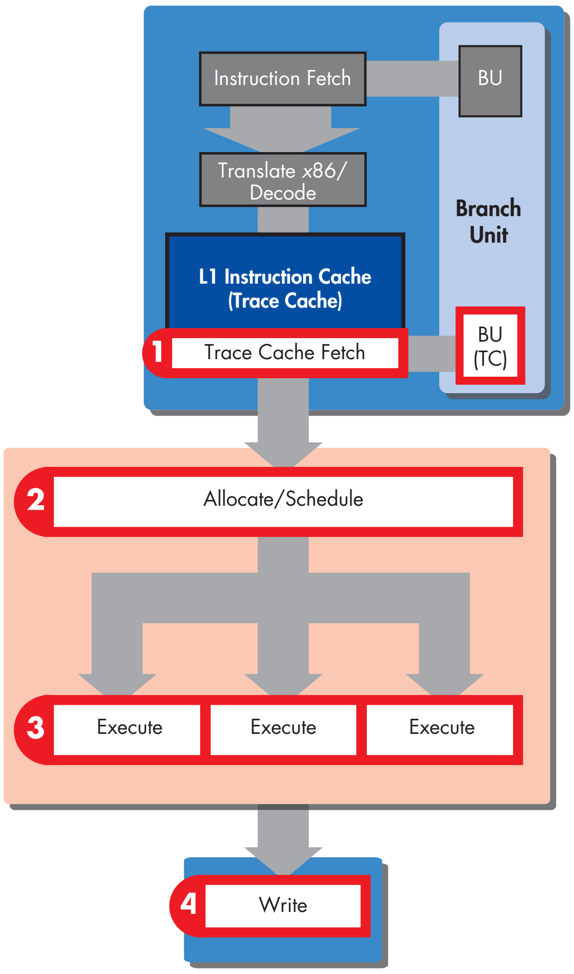


Figure 7-7: The Pentium 4's critical execution path

trace. As shown in Figure 7-8, the trace cache's branch prediction hardware splices code from the branch that it speculates the program will take right into the trace behind the code that it knows the program will take. So if you have a chunk of *x*86 code with a branch in it, the trace cache builds a trace from the instructions up to and including the branch instruction. Then, it picks which branch it thinks the program will take, and it continues building the trace along that speculative branch.

Having the speculative execution path spliced in right after the branch instruction confers on the trace cache two big advantages over a normal instruction cache. First, in a normal machine, it takes the branch predictor and BPU some time to do their thing when they come across a conditional branch instruction—they have to figure out which branch to speculatively execute, load up the proper branch target, and so on. This whole process usually

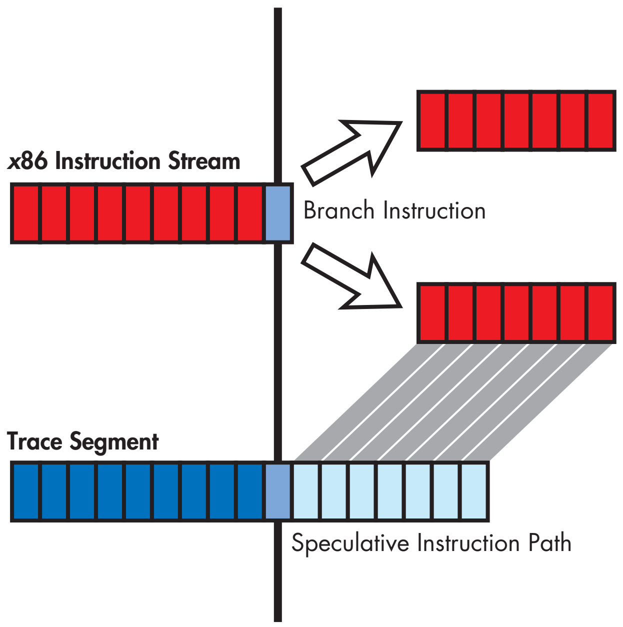


Figure 7-8: Speculative execution using the trace cache

adds at least one cycle of delay after every conditional branch instruction, a delay that often can't be filled with other code and therefore results in a pipeline bubble. With the trace cache, however, the code from the branch target is already sitting there right after the branch instruction, so there's no delay associated with looking it up and hence no pipeline bubble. In other words, the Pentium 4's trace cache implements a sort of branch folding, like what we previously saw implemented in the instruction queues of PowerPC processors.

The other advantage that the trace cache offers is also related to its ability to store speculative branches. When a normal L1 instruction cache fetches a cache line from memory, it stops fetching when it hits a branch instruction and leaves the rest of the line blank. If the branch instruction is the first instruction in an L1 cache line, then it's the only instruction in that line and the rest of the line goes to waste. Trace cache lines, on the other hand, can contain both branch instructions and the speculative code after the branch instruction. This way, no space in the trace cache's six–micro-op line goes to waste.

Most compilers take steps to deal with the two problems I've outlined (the delay after the branch and the wasted cache line space). As you saw, though, the trace cache solves these problems in its own way, so programs that are optimized to exploit these abilities see some advantages from them.

One interesting effect that the trace cache has on the Pentium 4's front end is that *x*86 translation/decode bandwidth is for the most part decoupled from dispatch bandwidth. You saw previously how the P6, for instance, spends a lot of transistor resources on a three different *x*86 decoders so that it can translate enough clunky *x*86 instructions each cycle into micro-ops to keep the back end fed. With the Pentium 4, the fact that most of the time program code is fetched from the trace cache in the form of predigested micro-ops means that a high bandwidth translator/decoder isn't necessary. The Pentium 4's decoding logic only has to kick on whenever there's an L1 cache miss, so it was designed to decode only one *x*86 instruction per clock cycle. This is one-third the maximum theoretical decode bandwidth of the P6, but the Pentium 4's trace cache allows it to meet or exceed the P6's real-world average dispatch rate.

The trace cache's handling of very long, multi-cycle *x*86 instructions is worth taking a look at, because it's quite clever. While most *x*86 instructions decode into around two or three micro-ops, there are some exceedingly long (and thankfully rare) *x*86 instructions (e.g., the string manipulation instructions) that decode into hundreds of micro-ops. Like the P6, the Pentium 4 has a special microcode ROM that decodes these longer instructions so that the regular hardware decoder can concentrate on decoding the smaller, faster instructions. For each long instruction, the microcode ROM stores a canned sequence of micro-ops, which it spits out when fed that instruction.

To keep these long, prepackaged sequences of micro-ops from polluting the trace cache, the Pentium 4's designers devised the following solution: Whenever the trace cache is building a trace segment and it encounters one of the long *x*86 instructions, instead of breaking it down and storing it as a micro-op sequence, the trace cache inserts into the trace segment a tag that points to the section of the microcode ROM containing the micro-op sequence for that particular instruction. Later, in execute mode, when the trace cache is streaming instructions out to the back end and it encounters one of these tags, it stops and temporarily hands control of the instruction stream over to the microcode ROM. The microcode ROM spits out the proper sequence of micro-ops (as designated by the tag) into the instruction stream, and then hands control back over to the trace cache, which resumes putting out instructions. The back end, which is on the other end of this instruction stream, doesn't know or care if the instructions are coming from the trace cache or the microcode ROM. All it sees is a constant, uninterrupted stream of instructions.

Intel hasn't said exactly how big the trace cache is—only that it holds 12,000 micro-ops. Intel claims this is roughly equivalent to a 16KB to 18KB I-cache.

By way of finishing up our discussion of the trace cache and introducing our detailed walk-through of the Pentium 4's pipeline, I should note two final aspects of the trace cache's effect on the pipeline. First, the trace cache still needs a short instruction fetch stage so that micro-ops can be fetched from it and sent to the allocation and scheduling logic. When we look at the Pentium 4's basic execution pipeline, you'll see this stage. Second, the trace cache actually has its own little BPU for predicting the directions and return addresses of branches within the trace cache itself. So the trace cache doesn't eliminate branch processing and prediction entirely from the picture; it just alleviates their effects on performance.

An Overview of the Pentium 4's Pipeline

Now let's step back and take a look at the Pentium 4's basic execution pipeline. Here's a breakdown of the various pipeline stages.

Stages 1 and 2: Trace Cache Next Instruction Pointer

In these stages, the Pentium 4's trace cache fetch logic gets a pointer to the next instruction in the trace cache.

Stages 3 and 4: Trace Cache Fetch

These two stages fetch an instruction from the trace cache to be sent to the back end.

Stage 5: Drive

This is the first of two special *drive stages* in the Pentium 4's pipeline, each of which is dedicated to driving signals from one part of the processor to the next. The Pentium 4 runs so fast that sometimes a signal can't make it all the way to where it needs to be in a single clock pulse, so the processor dedicates some pipeline stages to letting these signals propagate across the chip. These drive stages are there because the Pentium 4's designers intend for the chip to reach such stratospheric clock speeds that stages like this are absolutely necessary.

At the end of these first five stages, the Pentium 4's trace cache sends up to three micro-ops per cycle into a large, FIFO *micro-op queue*. This in-order queue, which sits in between the Pentium 4's front end and back end, smoothes out the flow of instructions to the back end by squeezing out any fetch- or decode-related bubbles. Micro-ops enter the top of the queue and fall down to rest at the lowest available entry, directly above the most recent micro-op to enter the queue. Thus, any bubbles that may have been ahead of the micro-ops disappear from the pipeline at this point. The micro-ops leave the bottom of the micro-op queue in program order and proceed to the next pipeline stage.

Stages 6 Through 8: Allocate and Rename (ROB)

In this group of stages, up to three instructions per cycle move from the bottom of the micro-op queue and are allocated entries in the Pentium 4's ROB and rename registers. With regard to the latter, the *x*86 ISA specifies only eight GPRs, eight FPRs, and eight VPRs, but the Pentium 4 has 128 of each type of register in its rename register files.

The allocator/renamer stages also allocates each micro-op an entry in one of the two micro-op queues detailed in the next section and can send up to three micro-ops per cycle into these queues.

Stage 9: Queue

To implement out-of-order execution, the Pentium 4 flows micro-ops from its trace cache through the ROB and into two deep micro-op queues that sit between its instructions' dispatch and execution phases. These two queues are the *memory micro-op queue* and the *arithmetic micro-op queue*. The memory micro-op queue holds memory operations (loads and stores) that are destined for the Pentium 4's LSU, while the arithmetic micro-op queue holds all other types of operations.

The two main micro-op queues are roughly analogous to the G4e's issue queues, but with one crucial difference: the Pentium 4's micro-op queues are FIFO queues, while the G4e's issue queues are not. For the Pentium 4, this means an instruction passes into and out of a micro-op queue *in program order* with respect to the other instructions in its own queue. However, instructions can still exit the bottom of each queue *out of program order* with respect to instructions in the other queue.

These two micro-op queues feed micro-ops into the scheduling logic in the next stage.

Stages 10 Through 12: Schedule

The micro-op queues described in the preceding section are only part of the Pentium 4's dynamic scheduling logic. The other half consists of a set of four micro-op schedulers whose job it is to schedule micro-ops for execution and to determine to which execution unit the micro-ops should be passed. Each of these schedulers consists of a smaller, 8- to 12-entry micro-op queue attached to a bit of scheduling logic. The scheduler's scheduling logic arbitrates with the other schedulers for access to the Pentium 4's four issue ports, and it removes micro-ops from its in-order scheduling queue and sends them through the right port at the right time.

An instruction cannot exit from a scheduling queue until its input operands are available and the appropriate execution unit is available. When the micro-op is ready to execute, it is removed from the bottom of its scheduling queue and passed to the proper execution unit through one of the Pentium 4's four issue ports, which are analogous to the P6 core's five issue ports in that they act as gateways to the back end's execution units.

Here's a breakdown of the four schedulers:

Memory scheduler

Schedules memory operations for the LSU.

Fast IU scheduler

Schedules arithmetic logic unit (ALU) operations (simple integer and logical instructions) for the Pentium 4's two double-speed integer execution units. As you'll see in the next chapter, the Pentium 4 contains two integer ALUs that run at twice the main core's clock speed.

Slow IU/general FPU scheduler

Schedules the rest of the integer instructions and most of the floating-point instructions.

Simple FP scheduler

Schedules simple FP instructions and FP memory operations.

These schedulers feed micro-ops through the four dispatch ports described in the next stage.

Stages 13 and 14: Issue

The P6 core's reservation station sends instructions to the back end via one of five issue ports. The Pentium 4 uses a similar scheme, but with four issue ports instead of five. There are two *memory ports* for memory instructions: the load port and the store port, for loads and stores, respectively. The remaining two ports, called *execution ports*, are for all the other instructions: execution port 0 and execution port 1. The Pentium 4 can send a total of six micro-ops per cycle through the four execution ports. This issue rate of six micro-ops per cycle is more micro-ops per cycle than the front end can fetch and decode (three per cycle) or the back end can complete (three per cycle), but that's okay because it gives the machine some headroom in its middle so that it can have bursts of activity.

You might be wondering how six micro-ops per cycle can move through four ports. The trick is that the Pentium 4's two execution ports are double-speed, meaning that they can dispatch instructions (integer only) on the rising and falling edges of the clock. But we'll talk more about this in the next chapter. For now, here's a breakdown of the two execution ports and which execution units are attached to them:

Execution port 0:

Fast integer ALU1 This unit performs integer addition, subtraction, and logical operations. It also evaluates branch conditionals and executes store-data micro-ops, which store data into the outgoing store buffer. This is the first of two double-speed integer units, which operate at twice the core clock frequency.

Floating-point/SSE move This unit performs floating-point and SSE moves and stores. It also executes the `FXCH` instruction, which means that it's no longer "free" on the Pentium 4.

Execution port 1:

Fast integer ALU2 This very simple integer ALU performs only integer addition and subtraction. It's the second of the two double-speed integer ALUs.

Slow integer ALU This integer unit handles all of the more time-consuming integer operations, like shift and rotate, that can't be completed in half a clock cycle by the two fast ALUs.

Floating-point/SEE/MMX ALU This unit handles floating-point and SSE addition, subtraction, multiplication, and division. It also handles all MMX instructions.

In Figure 7-5, I've labeled the instruction flow paths going into each execution unit to show which dispatch port instructions must pass through in order to reach which execution unit.

Stages 15 and 16: Register Files

This stage is where the execution units, upon receiving the instructions, read each instruction's input operands from the appropriate register file. To return to the discussion from Chapter 1, this step is the read step in the read-execute-write cycle of computation.

Stage 17: Execute

In this stage, the instructions are actually executed by the back end's execution units. We'll take a closer look at the Pentium 4's back end in the next chapter.

Stage 18: Flags

If an instruction's outcome stipulates that it needs to set any flags in the PSW, then it does so at this stage.

Stage 19: Branch Check

Here's the stage where the Pentium 4 checks the outcome of a conditional branch to see if it has just wasted 19 cycles of its time executing some code that it'll have to throw away. By stage 19, the branch condition has been evaluated, and the front end knows whether or not the branch predictor's guess was right or not.

Stage 20: Drive

You've already met the drive stage. Again, this stage is dedicated to propagating signals across the chip.

Stages 21 and Onward: Complete and Commit

Although Intel only lists the last part of the execution phase—stage 20—as part of the "normal Pentium 4 pipeline," for completeness, I'll include the write-back phase. Completed instructions file into their pre-assigned entries in the ROB, where they're put back in program order before having their results update the machine's architectural state.

As you can see, the Pentium 4's 20-stage pipeline does much of the same work in mostly the same order as the G4e's seven-stage pipeline. By dividing the pipeline into more stages, though, the Pentium 4 can reach higher clock rates. As I've already noted, this deeply pipelined approach fits with the Pentium 4's "narrow and deep" design philosophy.

The Pentium 4's Instruction Window

Before I discuss the nature of the Pentium 4's instruction window, note that I'm using the terms *instruction pool* and *instruction window* somewhat interchangeably. These two terms represent two slightly different metaphors for thinking about the set of queues and buffers positioned between a processor's front end and its back end. Instructions collect up in these queues and buffers—just like water collects in a pool or reservoir—before being drained away by the processor's back end. Because the instruction pool represents a small segment of the code stream, which the processor can examine for dependencies and reorder for optimal execution, this pool can also be said to function as a window on the code stream. Now that that's clear, let's take a look at the Pentium 4's instruction window.

As I explained in the previous chapter, the older P6 core's RS and ROB made up the heart of its instruction window. The Pentium 4 likewise has a ROB for tracking micro-ops, and in fact, its ROB is much larger than that of the P6. The buffer functions of the P6's reservation station, however, have been divided among multiple structures. The previous section's pipeline description explains how these structures are configured.

This partitioning of the instruction window into memory and arithmetic portions by means of the two scheduling queues has the effect of ensuring that both types of instructions will always have space in the window, and that an overabundance of one instruction type will not crowd the other type out of the window. The multiple schedulers provide fine-grained control over the instruction flow, so that it's optimally reordered for the fast execution units.

All of this deep buffering, scheduling, queuing, and optimizing is essential for keeping the Pentium 4's high-speed back end full. To return yet again to the Tetris analogy, imagine what would happen if someone were to double the speed at which the blocks fall; you'd hope that they would also double the size of the look-ahead window to compensate. The Pentium 4 greatly increases the size of the P6's instruction window as a way of compensating for the fact that the arrangement of instructions in its core is made so much more critical by the increased clock speed and pipeline depth.

The downside to all of this is that the schedulers and queues and the very large ROB all add complexity and cost to the Pentium 4's design. This complexity and cost are part of the price that the Pentium 4 pays for its deep pipeline and high clock speed.

8

INTEL'S PENTIUM 4 VS. MOTOROLA'S G4E: THE BACK END

In this chapter, I'll explain in greater detail the back end of both the Pentium 4 and the G4e. I'll talk about the execution resources that each processor uses for crunching code and data, and how those resources contribute to overall performance on specific types of applications.

Some Remarks About Operand Formats

Unlike the DLW and PowerPC ISAs described so far, the *x*86 ISA uses a two-operand format for both integer and floating-point instructions. If you want to add two numbers in registers `A` and `B`, the instruction would look as follows:

```
add A, B
```

This command adds the contents of `A` to the contents of `B` and places the result in `A`, overwriting whatever was previously in `A` in the process. Expressed mathematically, this would look as follows:

```
A = A + B
```

The problem with using a two-operand format is that it can be inconvenient for some sequences of operations. For instance, if you want to add `A` to `B` and store the result in `C`, you need two operations to do so:

Line #	Code	Comments
1	`mov A, C`	Copy the contents of register `A` to register `C`.
2	`add C, B`	Add the numbers in registers `C` and `B` and store the result in `C`, overwriting the previous contents of `C`.

Program 8-1

The first instruction in Program 8-1 copies the contents of `A` into `C` so that `A`'s value is not erased by the addition, and the second instruction adds the two numbers.

With a three-operand or more format, like many of the instructions in the PPC ISA, the programmer gets a little more flexibility and control. For instance, you saw earlier that the PPC ISA has a three-operand `add` instruction of the format

```
add destination, source1, source2
```

So if you want to add the contents of register `1` to the contents of register `2` and store the result in register `3` (i.e., `r3 = r1 + r2`), you just use the following instruction:

```
add 3, 2, 1
```

Some PPC instructions support even more than three operands, which can be a real boon to programmers and compiler writers. In "AltiVec Vector Operations" on page 170, we'll look in detail at the G4e AltiVec instruction set's use of a four-operand instruction format.

The PPC ISA's variety of multiple-operand formats are obviously more flexible than the one- and two-operand formats of *x*86. But nonetheless, modern *x*86 compilers are quite advanced and can overcome many of the aforementioned problems through the use of hidden microarchitectural rename registers and various scheduling algorithms. The problems with *x*86's two-operand format are much more of a liability for floating-point and vector code than for integer code. We'll talk more about this later, though.

The Integer Execution Units

Though the Pentium 4's *double-pumped* integer execution units got quite a bit of press when Netburst was first announced, you might be surprised to learn that both the G4e and the Pentium 4 embody approaches to enhancing integer performance that are very similar. As you'll see, this similarity arises from both processors' application of the computing design dictum: *Make the common case fast.*

For integer applications, the common case is easy to spot. As I outlined in Chapter 1, integer instructions generally fall into one of two categories:

Simple/fast integer instructions

Instructions like `add` and `sub` require very few steps to complete and are therefore easy to implement with little overhead. These simple instructions make up the majority of the integer instructions in an average program.

Complex/slow integer instructions

While addition and subtraction are fairly simple to implement, multiplication and division are complicated to implement and can take quite a few steps to complete. Such instructions involve a series of additions and bit shifts, all of which can take a while. These instructions represent only a fraction of the instruction mix for an average program.

Since simple integer instructions are by far the most common type of integer instruction, both the Pentium 4 and G4e devote most of their integer resources to executing these types of instructions very rapidly.

The G4e's IUs: Making the Common Case Fast

As was explained in the previous chapter, the G4e has a total of four IUs. The IUs are divided into two groups:

Three simple/fast integer execution units—SIUa, SIUb, SIUc

These three simple IUs handle only fast integer instructions. Most of the instructions executed by these IUs are single-cycle, but there are some multi-cycle exceptions to this rule. Each of the three fast IUs is fed by a single-entry reservation station.

One complex/slow integer execution unit—CIU

This single complex IU handles only complex integer instructions like multiply, divide, and some special-purpose register instructions, including condition register (CR) logical operations. Instructions sent to this IU generally take four cycles to complete, although some take longer. Note that divides, as well as some multiplication instructions, are not fully pipelined and thus can tie up the entire IU for multiple cycles. Also, instructions that update the PPC CR have an extra pipeline stage—called *finish*—to pass through before they leave the IU (more on this shortly). Finally, the CIU is fed by a two-entry reservation station.

By dedicating three of its four integer ALUs to the fastest, simplest, and most common instructions, the G4e is able to make the common case quite fast.

Before moving on, I should note that the finish stages attached to the ends of some of the execution unit pipelines are new in the G4e. These finish stages are dedicated to updating the condition register to reflect the results of any arithmetic operation that needs to do such updating (this happens infrequently). It's important to understand that at the end of the execute stage/start of the finish stage, an arithmetic instruction's results are available for use by dependent instructions, even though the CR has not yet been updated. Therefore, the finish stage doesn't affect the effective latency of arithmetic instructions. But for instructions that depend on the CR—like branch instructions—the finish stage adds an extra cycle of latency.

One nice thing that the PPC ISA has going for it is its large number of general-purpose registers (GPRs) for storing integers and addresses. This large number of architectural GPRs (32 to be exact) gives the compiler plenty of flexibility in scheduling integer operations and address calculations.

In addition to the PPC ISA's 32 GPRs, the G4e provides 16 microarchitectural general-purpose rename registers for use by the on-chip scheduling logic. These additional registers, not visible to the compiler or programmer, are used by the G4e to augment the 32 GPRs, thereby providing more flexibility for scheduling the processor's execution resources and keeping them supplied with data.

The Pentium 4's IUs: Make the Common Case Twice as Fast

The Pentium 4's integer functional block takes a very similar strategy to the G4e for speeding up integer performance. It contains one slow integer execution unit and two fast integer execution units. By just looking at the number of integer execution units, you might think that the Pentium 4 has less integer horsepower than the G4e. This isn't quite the case, though, because the Pentium 4's two fast IUs operate at *twice the core clock speed*, a trick that allows them to look to the outside world like four fast IUs.

The Pentium 4 can issue two integer instructions per cycle in rapid succession to each of the two fast IUs—one on the rising edge of the clock pulse and one on the falling edge. Each fast ALU can process an integer instruction in 0.5 cycles, which means it can process a total of two integer instructions per cycle. This gives the Pentium 4 a total peak throughput of four simple integer instructions per cycle for the two fast IUs combined.

Does this mean that the Pentium 4's two double-speed integer units are twice as powerful as two single-speed integer units? No, not quite. Integer performance is about much more than just a powerful integer functional block. You can't squeeze peak performance out of an integer unit if you can't keep it fed with code, and the Pentium 4 seems to have a weakness in this area when it comes to integer code.

We talked earlier in this book about how, due to the Pentium 4's "narrow and deep" design philosophy, branch mispredictions and cache misses can degrade performance by introducing pipeline bubbles into the instruction stream. This is especially a problem for integer performance, because integer-intensive applications often contain branch-intensive code that exhibits poor *locality of reference*. As a result, branch mispredictions in conjunction with cache latencies can kill integer performance. (For more information about these issues, see Chapter 11.)

As most benchmarks of the Pentium 4 bear out, in spite of its double-pumped ALUs, the Pentium 4's "narrow and deep" design is much more suited to floating-point applications than it is to integer applications. This floating-point bias seems to have been a deliberate choice on the part of the Pentium 4's designers—the Pentium 4 is designed to give not maximum but acceptable performance on integer applications. This strategy works because most modern processors (at least since the PIII, if not the PII or Pentium Pro) are able to offer perfectly workable performance levels on consumer-level integer-intensive applications like spreadsheets, word processors, and the like. Though server-oriented applications like databases require higher levels of integer performance, the demand for ever-increasing integer performance just isn't there in the consumer market. As a way of increasing the Pentium 4's integer performance for the server market, Intel sells a version of the Pentium 4 called the *Xeon*, which has a much larger cache.

Before moving on to the next topic, I should note that one characteristic of the Pentium 4 that bears mentioning is its large number of microarchitectural rename registers. The *x*86 ISA has only eight GPRs, but the Pentium 4 augments these with the addition of a large number of rename registers: 128 to be exact. Since the Pentium 4 keeps so many instructions on-chip for scheduling purposes, it needs these added rename resources to prevent the kinds of register-based resource conflicts that result in pipeline bubbles.

The Floating-Point Units (FPUs)

While the mass market's demand for integer performance may not be picking up, its demand for floating-point seems insatiable. Games, 3D rendering, audio processing, and almost all other forms of multimedia- and entertainment-oriented computing applications are extremely floating-point intensive. With floating-point applications perennially driving the home PC market, it's no wonder that the Pentium 4's designers made their design trade-offs in favor of FP performance over integer performance.

In terms of the way they use the processor and cache, floating-point applications are in many respects the exact opposite of the integer applications described in the preceding section. For instance, the branches in floating-point code are few and extremely predictable. Most of these branches occur as exit conditions in small chunks of loop code that iterate through a large dataset (e.g., a sound or image file) in order to modify it. Since these loops iterate many thousands of times as they work their way through a file, the branch instruction that is the exit condition evaluates to *taken* many thousands of times; it only evaluates to *not taken* once—when the program exits the loop. For such types of code, simple static branch prediction, where the branch prediction unit guesses that all branches will be taken every time, works quite well.

This description also points out two other ways in which floating-point code is the opposite of integer code. First, floating-point code has excellent locality of reference with respect to the instruction cache. A floating-point–intensive program, as I've noted, spends a large part of its execution time in relatively small loops, and these loops often fit in one of the processor caches. Second, because floating-point–intensive applications operate on large data files that are streamed into the processor from main memory, memory bandwidth is extremely important for floating-point performance. So while integer programs need good branch prediction and caching to keep the IUs fed with instructions, floating-point programs need good memory bandwidth to keep the FPUs fed with data.

Now that you understand how floating-point code tends to operate, let's look at the FPUs of the G4e and Pentium 4 to see how they tackle the problem.

The G4e's FPU

Since the G4e's designers have bet on the processor's vector execution units (described shortly) to do most of the serious floating-point heavy lifting, they made the G4e's FPU fairly simple and straightforward. It has a single pipeline for all floating-point instructions, and both single- and double-precision operations take the same number of cycles. (This ability to do double-precision FP is important mostly for scientific applications like simulations.) In addition, one single- or double-precision operation can be issued per cycle, with one important restriction (described later in this section). Finally, the G4e inherits the PPC line's ability to do single-cycle `fmadds`, and this time, both double- and single-precision `fmadds` have a single-cycle throughput.

Almost all of the G4e's floating-point instructions take five cycles to complete. There are a few instructions, however, that can take longer (`fdiv`, `fre`, and `fdiv`, for example). These longer instructions can take from 14 to 35 cycles to complete, and while they're executing, they hang up the floating-point pipeline, meaning that no other instructions can be done during that time.

One rarely discussed weakness of the G4e's FPU is the fact that it isn't *fully pipelined*, which means that it can't have five different instructions in five different stages of execution simultaneously. Motorola's software optimization manual for the 7450 states that if the 7450's first four stages of execution are occupied, the FPU will stall on the next cycle. This means that the FPU's peak theoretical instruction throughput is four instructions every five cycles.

It could plausibly be said that the PowerPC ISA gives the G4e a slight advantage over the Pentium 4 in terms of floating-point performance. However, a better way of phrasing that would be to say that the *x*86 ISA (or more specifically, the *x*87 floating-point extensions) puts the Pentium 4 at a slight disadvantage with respect to the rest of the world. In other words, the PPC's floating-point implementation is fairly normal and unremarkable, whereas the *x*87 has a few quirks that can make life difficult for FPU designers and compiler writers.

As mentioned at the outset of this chapter, the PPC ISA has instructions with one-, two-, three-, and four-operand formats. This puts a lot of power in the hands of the compiler or programmer as far as scheduling floating-point instructions to minimize dependencies and increase throughput and performance. Furthermore, this instruction format flexibility is augmented by a flat, 32-entry floating-point register file, which yields even more scheduling flexibility and even more performance.

In contrast, the instructions that make up the *x*87 floating-point extensions support two operands at most. You saw in Chapter 5 that the *x*87's very small eight-entry register file has a stack-based structure that limits it in certain ways. All Pentium processors up until the Pentium 4 get around this stack-related limitation with the "free" `fxch` instruction described in Chapter 5, but on the Pentium 4, `fxch` is no longer free.

So the Pentium 4's small, stack-based floating-point register file and two-operand floating-point instruction format put the processor at a disadvantage compared to the G4e's cleaner PowerPC floating-point specification. The Pentium 4's 128 floating-point rename registers help alleviate some of the false dependencies that arise from the low number of architectural registers, but they don't help much with the other problems.

The Pentium 4's FPU

There are two fully independent FPU pipelines on the Pentium 4, one of which is strictly for floating-point memory operations (loading and storing floating-point data). Since floating-point applications are extremely data- and memory-intensive, separating the floating-point memory operations and giving them their own execution unit helps a bit with performance.

The other FPU pipeline is for all floating-point arithmetic operations, and except for the fact that it doesn't execute memory instructions, it's very similar to the G4e's single FPU. Most simple floating-point operations take between five and seven cycles, with a few more complicated operations (like floating-point division) tying up the pipeline for a significantly longer time. Single- and double-precision operations take the same number of cycles, with both single- and double-precision floating-point numbers being converted into the *x*87's internal 80-bit temporary format. (This conversion is done for overflow reasons and doesn't concern us here.)

So the Pentium 4's FPU hardware executes instructions with slightly higher instruction latencies than the FPU of its predecessor, the P6 (for example, three to five cycles on the P6 for common instructions), but because the Pentium 4's clock speed is so much higher, it can still complete more floating-point instructions in a shorter period of time. The same is true of the Pentium 4's FPU in comparison with the G4e's FPU—the Pentium 4 takes more clock cycles than the G4e to execute floating-point instructions, but those clock cycles are much faster. So Pentium 4's clock-speed advantage and high-bandwidth front-side bus give it a distinct advantage over the Pentium III in floating-point–intensive benchmarks and enable it to be more than competitive with the G4e in spite of *x*87's drawbacks.

Concluding Remarks on the G4e's and Pentium 4's FPUs

The take-home message in the preceding discussion can be summed up as follows: While the G4e has fairly standard, unremarkable floating-point hardware, the PPC ISA does things the way they're supposed to be done—with three-operand or more instructions and a large, flat register file. The Pentium 4, on the other hand, has slightly better hardware but is hobbled by the legacy *x*87 ISA. The exact degree to which the *x*87's weaknesses affect performance has been debated for as long as the *x*87 has been around, but there seems to be a consensus that the situation is less than ideal.

The other thing that's extremely important to note is that when it comes to floating-point performance, a good memory subsystem is absolutely key. It doesn't matter how good a processor's floating-point hardware is—if you can't keep it fed, it won't be doing much work. Therefore, floating-point performance on both Pentium 4– and G4e-based systems depends on each system's available memory bandwidth.

The Vector Execution Units

One key technology on which both the Pentium 4 and the G4e rely for performance in their most important type of application—media applications (image processing, streaming media, 3D rendering, etc.)—is *Single Instruction, Multiple Data (SIMD) computing*, also known as *vector computing*. This section looks at SIMD on both the G4e and the Pentium 4.

A Brief Overview of Vector Computing

Chapter 1 discussed the movement of floating-point and vector capabilities from co-processors onto the CPU die. However, the addition of vector instructions and hardware to a modern, superscalar CPU is a bit more drastic than the addition of floating-point capability. A microprocessor is a *Single Instruction stream, Single Data stream (SISD)* device, and it has been since its inception, whereas vector computation represents a fundamentally different type of computing: SIMD. Figure 8-1 compares the SIMD and SISD in terms of a simple diagram that was introduced in Chapter 1.

As you can see in Figure 8-1, an SIMD machine exploits a property of the data stream called *data parallelism*. Data parallelism is said to be present in a dataset when its elements can be processed in parallel, a situation that most often occurs in large masses of data of a uniform type, like media files. Chapter 1 described media applications as applications that use small, repetitious chunks of code to operate on large, uniform datasets. Since these small chunks of code apply the same sequence of operations to every element of a large dataset, and these datasets can often be processed out of order, it makes sense to use SIMD to apply the same instructions to multiple elements at once.

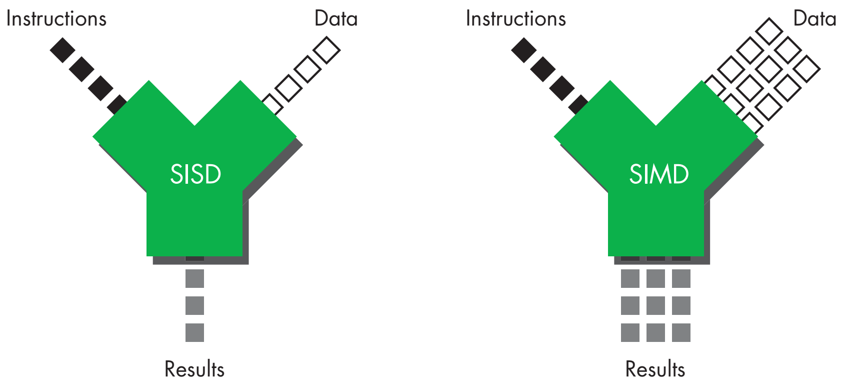


Figure 8-1: SISD versus SIMD

A classic example of a media application that exploits data parallelism is the inversion of a digital image to produce its negative. The image processing program must iterate through an array of uniform integer values (pixels) and perform the same operation (inversion) on each one. Consequently, there are multiple data points on which a single operation is performed, and the order in which that operation is performed on the data points doesn't affect the outcome. The program could start the inversion at the top of the image, the bottom of the image, or in the middle of the image—it doesn't matter as long as the entire image is inverted.

This technique of applying a single instruction to multiple data elements at once is quite effective in yielding significant speedups for many types of applications, especially streaming media, image processing, and 3D rendering. In fact, many of the floating-point–intensive applications described previously can benefit greatly from SIMD, which is why both the G4e and the Pentium 4 skimped on the traditional FPU in favor of strengthening their SIMD units.

There were some early, ill-fated attempts at making a purely SIMD machine, but the SIMD model is simply not flexible enough to accommodate general-purpose code. The only form in which SIMD is really feasible is as a part of a SISD host machine that can execute branch instructions and other types of code that SIMD doesn't handle well. This is, in fact, the situation with SIMD in today's market. Programs are written for a SISD machine and include SIMD instructions in their code.

Vectors Revisited: The AltiVec Instruction Set

The basic data unit of SIMD computation is the vector, which is why SIMD computing is also known as vector computing or *vector processing*. Vectors, which you met in Chapter 1, are nothing more than rows of individual numbers, or scalars. Figure 8-2 illustrates the differences between vectors and scalars.

A simple CPU operates on scalars one at a time. A superscalar CPU operates on multiple scalars at once, but it performs a different operation on each instruction. A vector processor lines up a whole row of scalars, all of the same type, and operates on them in parallel as a unit.

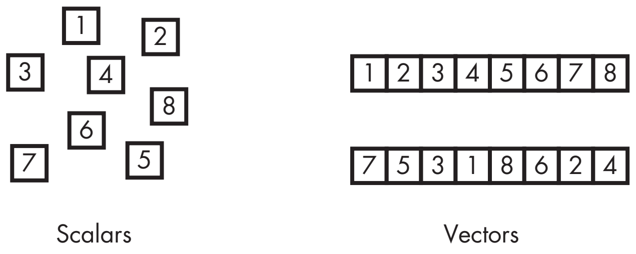


Figure 8-2: Scalars versus vectors

Vectors are represented in what is called a *packed data format*, in which data are grouped into bytes or words and *packed* into a vector of a certain length. To take Motorola's AltiVec, for example, each of the 32 AltiVec registers is 128 bits wide, which means that AltiVec can operate on vectors that are 128 bits wide. AltiVec's 128-bit wide vectors can be subdivided into

- 16 elements, where each element is either an 8-bit signed or unsigned integer or an 8-bit character;
- 8 elements, where each element is a 16-bit signed or unsigned integer;
- 4 elements, where each element is either a 32-bit signed or unsigned integer, or a single-precision (32-bit) IEEE floating-point number.

Figure 8-3 can help you visualize how these elements are packed together.

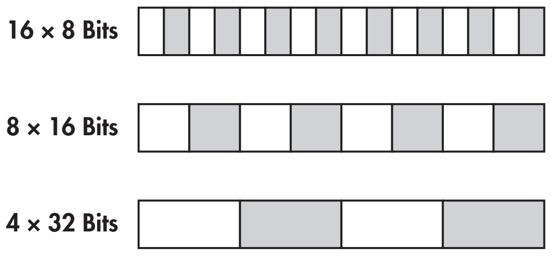


Figure 8-3: Vectors as packed data formats

AltiVec Vector Operations

Motorola's AltiVec literature divides the types of vector operations that AltiVec can perform into four useful and easily comprehensible categories. Because these categories offer a good way of dividing up the basic things you can do with vectors, I'll use Motorola's categories to break down the major types of vector operations. I'll also use pictures similar to those from Motorola's AltiVec literature, modifying them a bit when needed.

Before looking at the types of operations, note that AltiVec's instruction format supports up to four operands, laid out as follows:

```
AltiVec_instruction source1, source2, filter/mod, destination
```

The main difference to be noted is the presence of the `filter/mod` operand, also called a *control operand* or *control vector*. This operand specifies a register that holds either a bit mask or a control vector that somehow modifies or sets the terms of the operation. You'll see the control vector in action when we look at the vector permute.

AltiVec's four-operand format is much more flexible than the two-operand format available to Intel's SIMD instructions, making AltiVec a much more ideal vector processing instruction set than the Pentium line's SSE or SSE2 extensions. The *x*86 ISA's SIMD extensions are limited by their two-operand format in much the same way that the *x*86 floating-point extension is limited by its stack-based register file.

Motorola's categories for the vector operations that the AltiVec can perform are as follows:

- intra-element arithmetic
- intra-element non-arithmetic
- inter-element arithmetic
- inter-element non-arithmetic

Intra-Element Arithmetic and Non-Arithmetic Instructions

Intra-element arithmetic operation is one of the most basic and easy-to-grasp categories of vector operation because it closely parallels scalar arithmetic. Consider, for example, an intra-element addition. This involves lining up two or three vectors (`VA`, `VB`, and `VC`) and adding their individual elements together to produce a sum vector (`VT`). Figure 8-4 contains an example of intra-element arithmetic operation at work on three vectors, each of which consists of 16 eight-bit numbers. Other intra-element operations include multiplication, multiply-add, average, and minimum.

Intra-element non-arithmetic operations basically work the same way as intra-element arithmetic functions, except for the fact that the operations performed are different. Intra-element non-arithmetic operations include logical operations like AND, OR, and XOR.

Figure 8-4 shows an intra-element addition involving three vectors of pixel values: red, blue, and green. The individual elements of the three vectors are added to produce a target vector consisting of white pixels.

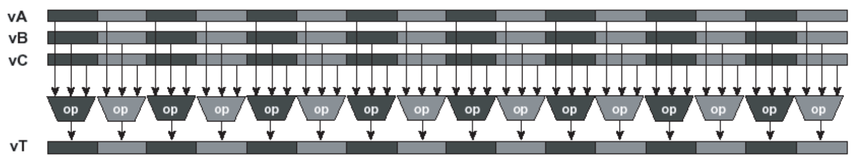


Figure 8-4: Intra-element arithmetic operations

The following list summarizes the types of vector intra-element instructions that AltiVec supports:

- integer logical instructions
- integer arithmetic instructions

- integer compare instructions
- integer rotate and shift instructions
- floating-point arithmetic instructions
- floating-point rounding and conversion instructions
- floating-point compare instructions
- floating-point estimate instructions
- memory access instructions

Inter-Element Arithmetic and Non-Arithmetic Instructions

Inter-element arithmetic operations are operations that happen between the elements in a single vector. An example of an inter-element arithmetic operation is shown in Figure 8-5, in which the elements in one vector are added together and the total is stored in an accumulation vector—VT.

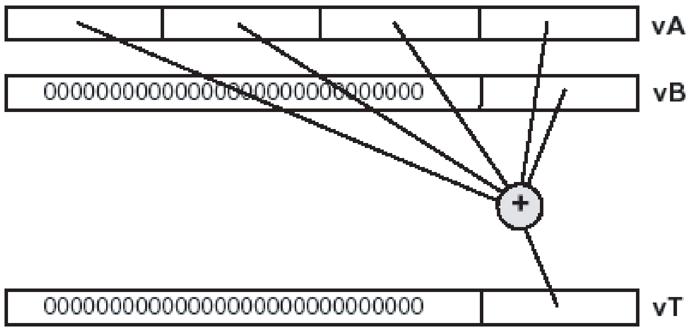


Figure 8-5: An inter-element sum across operation

Inter-element non-arithmetic operations are operations like vector permute, which rearrange the order of the elements in an individual vector. Figure 8-6 shows a vector permute instruction. VA and VB are the source registers that hold the two vectors to be permuted, VC contains the control vector that tells AltiVec which elements it should put where, and VT is the destination register.

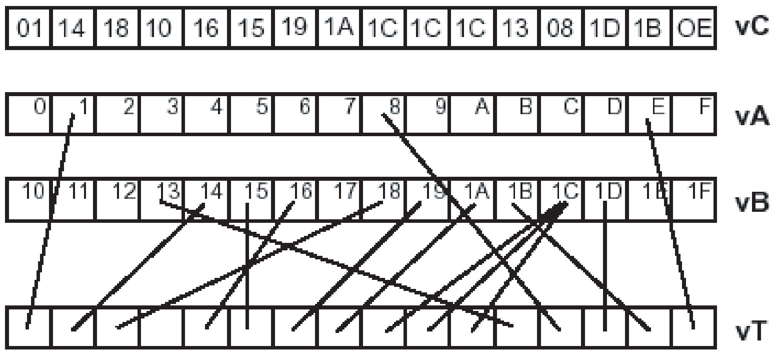

01 14 18 10 16 15 19 1A 1C 1C 1C 13 08 1D 1B 0E
vC
0 1 2 3 4 5 6 7 8 9 A B C D E F
vA
10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F
vB
vT

Figure 8-6: An inter-element permute operation

The following list summarizes the types of vector inter-element instructions that AltiVec supports:

- Alignment support instructions
- Permutation and formatting instructions
- Pack instructions
- Unpack instructions
- Merge instructions
- Splat instructions
- Shift left/right instructions

The G4e's VU: SIMD Done Right

The AltiVec extension to PowerPC adds 162 new instructions to the PowerPC instruction set. When Motorola first implemented AltiVec support in their PowerPC processor line with the MPC 7400, they added 32 new AltiVec registers to the G4's die, along with two dedicated AltiVec SIMD functional units. All of the AltiVec calculations were done by one of two fully-pipelined, independent AltiVec execution units.

The G4e improves significantly on the original G4's AltiVec implementation. The processor boasts four independent AltiVec units, three of which are fully pipelined and can operate on multiple instructions at once. These units are as follows:

Vector permute unit

This unit handles the instructions that rearrange the operands within a vector. Some examples are `permute`, `merge`, `splat`, `pack`, and `unpack`.

Vector simple integer unit

This unit handles all of the fast and simple vector integer instructions. It's basically just like one of the G4e's three fast IUs, except vectorized. This unit has only one pipeline stage, so most of the instructions it executes are single-cycle.

Vector complex integer unit

This is the vector equivalent of the G4e's one slow IU. It handles the slower vector instructions, like multiply, multiply-add, and so on.

Vector floating-point unit

This unit handles all vector floating-point instructions.

The G4e can issue up to two AltiVec instructions per clock cycle, with each instruction going to any one of the four vector execution units. All of the units, with the exception of the VSIU, are pipelined and can operate on multiple instructions at once.

All of this SIMD execution hardware is tied to a generous register file that consists of 32 128-bit architectural registers and 16 additional vector rename registers.

Intel's MMX

The story of MMX and SSE/KNI/MMX2 is quite a bit more complicated than that of AltiVec, and there are a number of reasons why this is so. To begin with, Intel first introduced MMX as an *integer-only* SIMD solution, so MMX doesn't support floating-point arithmetic at all. Another weakness of MMX is the fact that Intel jumped through some hoops to avoid adding a new processor state, hoops that complicated the implementation of MMX. I'll deal with this in more detail in "SSE and SSE2" on page 175.

Where AltiVec's vectors are 128 bits wide, MMX's are only 64 bits wide. These 64-bit vectors can be subdivided into:

- 8 elements (a packed byte), where each element is an 8-bit integer;
- 4 elements (a packed word), where each element is a 16-bit signed or unsigned integer;
- 2 elements (packed double word), where each element is a 32-bit signed or unsigned integer.

These vectors are stored in eight MMX registers, based on a flat-file model. These eight registers, MM0 to MM7, are *aliased* onto the *x*87's stack-based floating-point registers, FP0 to FP7. Intel did this in order to avoid imposing a costly *processor state switch* any time you want to use MMX instructions. The drawback to this approach is that floating-point operations and MMX operations must share a register space, so a programmer can't mix floating-point and MMX instructions in the same routine. Of course, since there's no *mode bit* for toggling the register file between MMX and floating-point usage, there's nothing to prevent a programmer from pulling such a stunt and corrupting his floating-point data by overwriting it with integer vector data.

The fact that a programmer can't mix floating-point and MMX instructions normally isn't a problem, though. In most programs, floating-point calculations are used for generating data, while SIMD calculations are used for displaying it.

In all, MMX added 57 new instructions to the *x*86 ISA. The MMX instruction format is pretty much like the conventional *x*86 instruction format:

```
MMX_instruction mmreg1, mmreg2
```

In this instruction, `mmreg1` is the both the destination and source operand, meaning that `mmreg1` gets overwritten by the result of the calculation. For the reasons outlined in the previous discussion of operand formats, this two-operand instruction format isn't nearly as optimal as AltiVec's four-operand format. Furthermore, MMX instructions lack that third filter/mod vector that AltiVec has. This means that you can't do those one-cycle, arbitrary two-vector permutes.

SSE and SSE2

Even as MMX was being rolled out, Intel knew that its 64-bit nature and integer-only limitation made it seriously deficient as a vector processing solution. An article in an issue of the *Intel Technology Journal* tells this story:

> In February 1996, the product definition team at Intel presented Intel's executive staff with a proposal for a single-instruction-multiple-data (SIMD) floating-point model as an extension to IA-32 architecture. In other words, the "Katmai" processor, later to be externally named the Pentium III processor, was being proposed. The meeting was inconclusive. At that time, the Pentium® processor with MMX instructions had not been introduced and hence was unproven in the market. Here the executive staff were being asked essentially to "double down" their bets on MMX instructions and then on SIMD floating-point extensions. Intel's executive staff gave the product team additional questions to answer and two weeks later, still in February 1996, they gave the OK for the "Katmai" processor project. During the later definition phase, the technology focus was refined beyond 3D to include other application areas such as audio, video, speech recognition, and even server application performance. In February 1999, the Pentium III processor was introduced.
>
> —*Intel Technology Journal, Second Quarter, 1999*

Intel's goal with SSE (Streaming SIMD Extensions, aka MMX2/KNI) was to add four-way, 128-bit SIMD single-precision floating-point computation to the *x*86 ISA. Intel went ahead and added an extra eight 128-bit architectural registers, called *XMM registers*, for holding vector floating-point instructions. These eight registers are in addition to the eight MMX/*x*87 registers that were already there. Since these registers are totally new and independent, Intel had to hold their nose and add an extra processor state to accommodate them. This means a state switch if you want to go from using *x*87 to MMX or SSE, and it also means that operating system code had to be rewritten to accommodate the new state.

SSE still had some shortcomings, though. SSE's vector floating-point operations were limited to single-precision, and vector integer operations were still limited to 64 bits, because they had to use the old MMX/*x*87 registers.

With the introduction of SSE2 on the Pentium 4, Intel finally got its SIMD act together. On the integer side, SSE2 finally allows the storage of 128-bit integer vectors in the XMM registers, and it modifies the ISA by extending old instructions and adding new ones to support 128-bit SIMD integer operations. For floating-point, SSE2 now supports double-precision SIMD floating-point operations. All told, SSE2 adds 144 new instructions, some of which are cache control instructions.

The Pentium 4's Vector Unit: Alphabet Soup Done Quickly

Now that you know something about SSE2, let's look at how it's implemented on the Pentium 4. In keeping with the Pentium 4's "narrow, deep, and fast" approach, the Pentium 4 does not sport any dedicated SSE2 pipelines. Rather, both FPU pipes double as VU pipes, meaning that the FPU memory unit also handles vector memory operations and the FPU arithmetic unit also handles vector arithmetic operations. So in contrast to the G4e's four vector arithmetic units, the Pentium 4 has one vector arithmetic unit that just does everything—integer, floating-point, permutes, and so on.

So, with only one execution pipeline to handle all vector and floating-point operations, you're probably wondering how the Pentium 4's designers expected it to perform competitively on the media applications for which it was obviously designed. The Pentium 4 is able to offer competitive SIMD performance based on a combination of three factors:

- relatively low instruction latencies
- extremely high clock speeds
- a high-bandwidth caching and memory subsystem

Let's take a look at these factors and how they work together.

The Pentium 4 optimization manual lists in Section C the average latencies of the most commonly used SIMD instructions. A look through the latency tables reveals that the majority of single- and double-precision arithmetic operations have latencies in the four- to six-cycle range. In other words, most vector floating-point instructions go through four to six pipeline stages before leaving the Pentium 4's VU. This number is relatively low for a 20-stage pipeline design like the Pentium 4, especially considering that vector floating-point instructions on the G4e go through four pipeline stages on average before leaving the G4e's vector FPU.

Now, considering the fact that at the time of this writing, the Pentium 4's clock speed is significantly higher than (roughly double) that of the G4e, the Pentium 4's ability to execute single- and double-precision vector floating-point operations in almost the same number of clock cycles as the G4e means that the Pentium 4 executes these operations almost three times as fast in real-world, "wall clock" time. So the Pentium 4 can get away with having only one VU, because that VU is able to grind through vector operations with a much higher instruction completion rate (instructions/ns) than the G4e. Furthermore, as the Pentium 4's clock speed increases, its vector crunching power grows.

Another piece that's crucial to the whole performance picture is the Pentium 4's high-bandwidth FSB, memory subsystem, and low-latency caching subsystem. I won't cover these features here, but I'll just note that the Pentium 4 has large amounts of bandwidth at its disposal. All this bandwidth is essential to keeping the very fast VU fed with data, and as the Pentium 4's clock speed has increased, bandwidth has played an even larger role in vector processing performance.

Increasing Floating-Point Performance with SSE2

I mentioned earlier that MMX uses a flat register file, and the same is true of both SSE and SSE2. The eight, 128-bit XMM registers are arranged as a flat file, which means that if you're able to replace an *x*87 FP operation with an SSE or SSE2 operation, you can use clever resource scheduling to avoid the performance hit brought on by the Pentium 4's combination of a stack-based FP register file and a non-free `fxch` instruction. Intel's highly advanced compiler has proven that converting large amounts of *x*87 code to SSE2 code can yield a significant performance boost.

Conclusions

The preceding discussion should make it clear that the overall design approaches outlined in the first half of this chapter can be seen in the back ends of each processor. The G4e continues its "wide and shallow" approach to performance, counting on instruction-level parallelism (ILP) to allow it to squeeze the most performance out of code. The Pentium 4's "narrow and deep" approach, on the other hand, uses fewer execution units, eschewing ILP and betting instead on increases in clock speed to increase performance.

Each of these approaches has its benefits and drawbacks, but as I've stressed repeatedly throughout this chapter, microarchitecture is by no means the only factor in the application performance equation. Certain properties of the ISA that a processor implements can influence performance.

9

64-BIT COMPUTING AND *x*86-64

On a number of occasions in previous chapters, I've discussed some of the more undesirable aspects of the PC market's most popular instruction set architecture—the *x*86. The *x*86 ISA's complex addressing modes, its inclusion of unwieldy and obscure instructions, its variable instruction lengths, its dearth of architectural registers, and its other quirks have vexed programmers, compiler writers, and microprocessor architects for years.

In spite of these drawbacks, the *x*86 ISA continues to enjoy widespread commercial success, and the number of markets in which it competes continues to expand. The reasons for this ongoing success are varied, but one factor stands out as by far the most important: inertia. The installed base of *x*86 application software is huge, and the costs of an industry-wide transition to a cleaner, more classically RISC ISA would enormously outweigh any benefits. Nonetheless, this hasn't stopped many folks, including the top brass at Intel, from dreaming of a post-*x*86 world.

Intel's IA-64 and AMD's *x*86-64

As of this writing, there have been two important attempts to move mainstream, commodity desktop and server computing beyond the 32-bit *x*86 ISA, the first by Intel and the second by Intel's chief rival, Advanced Micro Devices (AMD). In 1994, Intel and Hewlett-Packard began work on a completely new ISA, called *IA-64*. IA-64 is a 64-bit ISA that embodies a radically different approach to performance from anything the mainstream computing market has yet seen. This approach, which Intel has called *Explicitly Parallel Instruction Computing (EPIC)*, is a mix of a *very long instruction word (VLIW)* ISA, *predication*, *speculative loading*, and other compiler-oriented, performance-enhancing techniques, many of which had never been successfully implemented in a commercial product line prior to IA-64.

Because IA-64 represents a total departure from *x*86, IA-64–based processors cannot run legacy *x*86 code natively. The fact that IA-64 processors must run the very large mass of legacy *x*86 code in emulation has posed a serious problem for Intel as they try to persuade various segments of the market to adopt the new architecture.

Unfortunately for Intel, the lack of backward compatibility isn't the only obstacle that IA-64 has had to contend with. Since its inception, Intel's IA-64 program has met with an array of setbacks, including massive delays in meeting development and production milestones, lackluster integer performance, and difficulty in achieving high clock speeds. These and a number of other problems have prompted some wags to refer to Intel's first IA-64 implementation, called *Itanium* and released in 2001, as *Itanic*. The Itanium Processor Family (IPF) has since found a niche in the lucrative and growing high-end server and workstation segments. Nonetheless, in focusing on Itanium Intel left a large, 64-bit sized hole in the commodity workstation and server markets.

In 1999, with Itanium development beset by problems and clearly some distance away from commercial release, AMD saw an opening to score a major blow against Intel by using a 64-bit derivative of Intel's own ISA to jump-start and dominate the nascent commodity 64-bit workstation market. Following on the success of its Athlon line of desktop processors, AMD took a gamble and bet the company's future on a set of 64-bit extensions to the *x*86 ISA. Called x*86-64*, these extensions enabled AMD to produce a line of 64-bit microprocessors that are cost-competitive with existing high-end and midrange *x*86 processors and, most importantly, backward-compatible with existing *x*86 code. The new processor architecture, popularly referred to by its code name *Hammer*, has been a commercial and technical success.

Introduced in April 2003 after a long series of delays and production problems, the Hammer's strong benchmark performance and excellent adoption rate spelled trouble for any hopes that Intel may have had for the mainstream commercial adoption of its Itanium line. Intel conceded as much when in 2004, they took the unprecedented step of announcing support for AMD's extensions in their own *x*86 workstation and server processors. Intel calls these 64-bit extensions *IA-32e*, but in this book we'll refer to them by AMD's name—*x*86-64.

Because *x*86-64 represents the future of *x*86 for both Intel and AMD, this chapter will look in some detail at the new ISA. As you'll see, *x*86-64 is more than just a 64-bit extension to the 32-bit *x*86 ISA; it adds some new features as well, while getting rid of some obsolete ones.

Why 64 Bits?

The question of why we need 64-bit computing is often asked but rarely answered in a satisfactory manner. There are good reasons for the confusion surrounding the question, the first of which is the rarely acknowledged fact that "the 64-bit question" is actually two questions:

1. How does the existing 64-bit server and workstation market use 64-bit computing?
2. What use does the consumer market have for 64-bit computing?

People who ask the 64-bit question are usually asking for the answer to question 1 in order to deduce the answer to question 2. This being the case, let's first look at question 1 before tackling question 2.

What Is 64-Bit Computing?

Simply put, the labels *16-bit*, *32-bit*, or *64-bit*, when applied to a microprocessor, characterize the processor's data stream. You may have heard the term *64-bit code*; this designates code that operates on 64-bit data.

In more specific terms, the labels *64-bit*, *32-bit*, and so on designate the number of bits that each of the processor's general-purpose registers (GPRs) can hold. So when someone uses the term *64-bit processor*, what they mean is a processor with GPRs that store 64-bit numbers. And in the same vein, a *64-bit instruction* is an instruction that operates on 64-bit numbers that are stored in 64-bit GPRs.

Figure 9-1 shows two computers, one a 32-bit computer and the other a 64-bit computer.

In Figure 9-1, I've tried my best to modify Figure 1-3 on page 6 in order to make my point. Don't take the instruction and code sizes too literally, since they're intended to convey a general feel for what it means to "widen" a processor from 32 bits to 64 bits.

Notice that not all of the data in memory, the cache, or the registers is 64-bit data. Rather, the data sizes are mixed, with 64 bits being the widest. We'll discuss why this is and what it means shortly.

Note that in the 64-bit CPU pictured in Figure 9-1, the width of the code stream has not changed; the same-sized machine language instruction could theoretically represent an instruction that operates on 32-bit numbers or an instruction that operates on 64-bit numbers, depending on the instruction's default data size. On the other hand, the widths of some elements of the data and results streams have doubled. In order to accommodate the wider data stream, the sizes of the processor's registers and the sizes of the internal data paths that feed those registers must also be doubled.

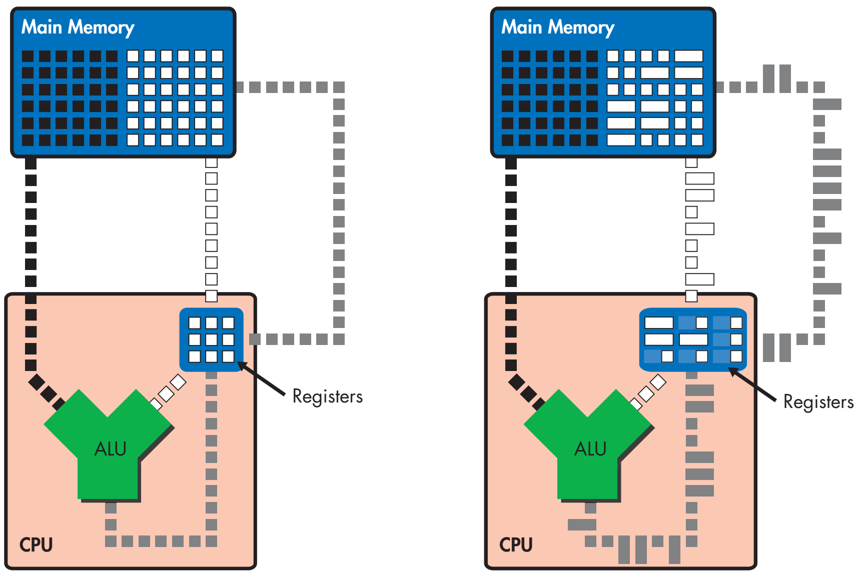


Figure 9-1: 32-bit versus 64-bit computing

Now look at the two programming models illustrated in Figure 9-2—one for a 32-bit processor and another for a 64-bit processor.

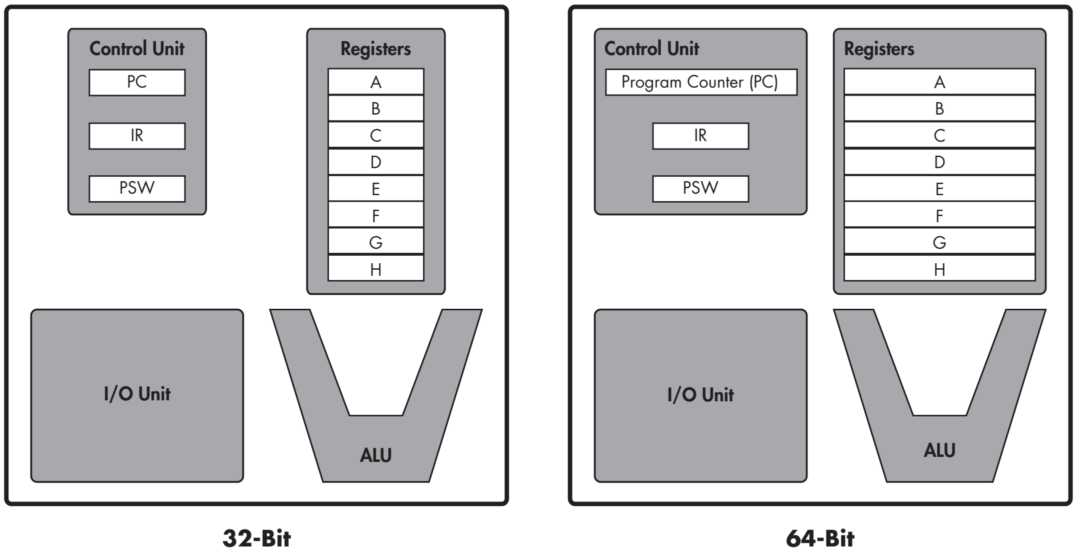


Figure 9-2: 32-bit versus 64-bit programming models

The registers in the 64-bit ISA pictured in Figure 9-2 are twice as wide as those in the 32-bit ISA, but the size of the instruction register (IR) that holds the currently executing instruction is the same in both processors. Again, the data stream has doubled in size, but the instruction stream has not. You might also note that the program counter (PC) has doubled in size. We'll talk about the reason for this in the next section.

The discussion up so far about widened instruction and data registers has laid out the simple answer to the question, “What is 64-bit computing?” If you take into account the fact that the data stream is made up of multiple types of data—a fact hinted at in Figure 9-1—the answer gets a bit more complicated.

For the simple processor pictured in Figure 9-2, the two types of data that can be processed are integer data and address data. As you’ll recall from Chapter 1, addresses are really just integers that designate a memory address, so address data is just a special type of integer data. Hence, both data types are stored in the GPRs, and both integer and address calculations are done by the arithmetic logic unit (ALU).

Many modern processors support two additional data types: floating-point data and vector data. Each of these two data types has its own set of registers and its own execution unit(s). The following table compares all four data types in 32-bit and 64-bit processors:

Data Type	Register Type	Execution Unit	x86 Width (in Bits)	x86-64 Width (in Bits)
Integer	GPR	ALU	32	64
Address	GPR	ALU or AGU	32	64
Floating-point	FPR	FPU	80	80
Vector	VR	VPU	128	128

You can see from this table that the difference the move to 64 bits makes is in the integer and address hardware. The floating-point and vector hardware stays the same.

Current 64-Bit Applications

Now that you know what 64-bit computing is, let’s look at the benefits of increased integer and address sizes.

Dynamic Range

The main thing that a wider integer gives you is increased *dynamic range*. Instead of giving a one-line definition of the term dynamic range, I’ll just explain how it works.

In the *base-10* number system which you’re all accustomed to, you can represent a maximum of 10 integers (0 to 9) with a single digit. This is because base-10 has 10 different symbols with which to represent numbers. To represent more than 10 integers, you need to add another digit, using a combination of two symbols chosen from among the set of 10 to represent any one of 100 integers (00 to 99). The formula you can use to compute the number of integers (dynamic range, or DR) you can represent with an n-digit base-10 number is

$$DR = 10^n$$

So a one-digit number gives you $10^1 = 10$ possible integers, a two-digit number $10^2 = 100$ integers, a three-digit number $10^3 = 1{,}000$ integers, and so on.

The base-2, or *binary*, number system that computers use has only two symbols with which to represent integers: 0 and 1. Thus, a single-digit binary number allows you to represent only two integers—0 and 1. With a two-digit (or two-bit) binary, you can represent four integers by combining the two symbols (0 and 1) in any of the following four ways:

Binary	Base-10
00	0
01	1
10	2
11	3

Similarly, a three-bit binary number gives you eight possible combinations, which you can use to represent eight different integers. As you increase the number of bits, you increase the number of integers you can represent. In general, n bits allow you to represent 2^n integers in binary. So a 4-bit binary number can represent $2^4 = 16$ integers, an 8-bit number gives you $2^8 = 256$ integers, and so on.

In moving from a 32-bit GPR to a 64-bit GPR, the range of integers that a processor can recognize and perform arithmetic calculations on goes from $2^{32} = 4.3e9$ to $2^{64} = 1.8e19$. The dynamic range, then, increases by a factor of 4.3 billion. Thus a 64-bit integer can represent a much larger range of numbers than a 32-bit integer.

The Benefits of Increased Dynamic Range, or, How the Existing 64-Bit Computing Market Uses 64-Bit Integers

Some applications, mostly in the realm of scientific computing and simulations, require 64-bit integers because they work with numbers outside the dynamic range of 32-bit integers. When the magnitude of the result of a calculation exceeds the range of possible integer values that the machine can represent, you get a situation called either *overflow* (the result was greater than the highest positive integer) or *underflow* (the result was less than the largest negative integer). When this happens, the number you get in the register isn't the right answer. There's a bit in the *x*86's processor status word that allows you to check to see if an integer arithmetic result has just exceeded the processor's dynamic range, so you know that the result is erroneous. Such situations are very, very rare in integer applications. I personally have never run into this problem, although I have run into the somewhat related problem of floating-point round-off error a few times.

Programmers who run into integer overflow or underflow problems on a 32-bit platform have the option of using a 64-bit integer construct provided by a high-level language like C. In such cases, the compiler uses two registers per integer—one for each half of the integer—to do 64-bit calculations in 32-bit hardware. This has obvious performance drawbacks, making it less desirable than a true 64-bit integer implementation.

In addition to scientific computing and simulations, there is another application domain for which 64-bit integers can offer real benefits: cryptography. Most popular encryption schemes rely on the multiplication and factoring of very large integers, and the larger the integers, the more secure the encryption. AMD and Intel are hoping that the growing demand for tighter security and more encryption in the mainstream business and consumer computing markets will make 64-bit processors attractive.

Larger GPRs don't just make for larger integers, but they make for larger addresses, as well. In fact, larger addresses are the primary advantage that 64-bit computers have over their 32-bit counterparts.

Virtual Address Space Versus Physical Address Space

Throughout this book, I've been referring to three different types of storage—the caches, the RAM, and the hard disk—without ever explicitly describing how all three of these types fit together from the point of view of a programmer. If you're not familiar with the concept of *virtual memory*, you might be wondering how a programmer makes use of the hard disk for code and data storage if load and store instructions take memory addresses as source and destination operands. Though this brief section only scratches the surface of the concept of virtual memory, it should give you a better idea of how this important concept works.

Every major component in a computer system must have an address. Just like having a phone number allows people to communicate with each other via the phone system, having an address allows a computer's components to communicate with each other via the computer's internal system of buses. If a component does not have an address, no other component in the system can communicate with it directly, either for the purpose of reading from it, writing to it, or controlling it. Video cards, SCSI cards, memory banks, chipsets, processors, and so on all have unique addresses. Even more importantly for our purposes, storage devices such as the hard disk and main memory also have unique addresses, and because such devices are intended for use as storage, they actually look to the CPU like a range of addresses that can be read from or written to.

Like a phone number, each component's address is a whole number (i.e., an integer), and as you learned earlier in our discussion of dynamic range, the width of an address (the number of digits in the address) determines the range of possible addresses that can be represented.

A range of addresses is called an *address space*, and in a modern 32-bit desktop computer system, the processor and operating system work together to provide each running program with the illusion that it has access to a flat address space of up to 4GB in size. (Remember the 2^{32} = 4.3 billion number? Those 4.3 billion bytes are 4GB, which is the number of bytes that a 32-bit computer can address.) One large portion of this *virtual address space* that a running program sees consists of addresses through which it can interact with the operating system, and through the OS with the other parts of the system. The other portion of the address space is set aside for the program to interact with main memory. This main memory portion of the address space always represents only a part of the total 4GB virtual address space, in many cases about 2GB.

This 2GB chunk of address space represents a kind of window through which the program can look at main memory. Note that the program can see and manipulate only the data it has placed in this special address space, so on a 32-bit machine, a program can see only about 2GB worth of addresses at any given time. (There are some ways of getting around this, but those don't concern us here.)

The Benefits of a 64-Bit Address

Because addresses are just special-purpose integers, an ALU and register combination that can handle more possible integer values can also handle that many more possible addresses. This means that each process's virtual address space is greatly increased on a 64-bit machine, allowing each process to "see" a much larger range of virtual addresses. In theory, each process on a 64-bit computer could have an 18 million–terabyte address space.

The specifics of microprocessor implementation dictate that there's a big difference between the amount of address space that a 64-bit address value could theoretically yield and the actual sizes of the virtual and physical address spaces that a given 64-bit architecture supports. In the case of *x*86-64, the actual size of the Hammer line's virtual addresses is 48 bits, which makes for about 282 terabytes of virtual address space. (I'm tempted to say about this number what Bill Gates is falsely alleged to have said about 640KB back in the DOS days: "282 terabytes ought to be enough for anybody." But don't quote me on that in 10 years when Doom 9 takes up three or four hundred terabytes of hard disk space.) *x*86-64's physical address space is 40 bits wide, which means that an *x*86-64 system can support about one terabyte of physical memory (RAM).

So, what can a program do with up to 282 terabytes of virtual address space and up to a terabyte of RAM? Well, caching a very large database in it is a start. Back-end servers for mammoth databases are one place where 64 bits have long been a requirement, so it's no surprise to see 64-bit offerings billed as capable database platforms.

On the media and content creation side of things, folks who work with very large 2D image files also appreciate the extra address space. A related application domain where large amounts of memory come in handy is in simulation and modeling. Under this heading you could put various CAD tools and 3D rendering programs, as well as things like weather and scientific simulations, and even, as I've already half-jokingly referred to, real-time 3D games. Though the current crop of 3D games (as of 2006) probably wouldn't benefit from greater than 4GB of address space, it's certain that you'll see a game that benefits from greater than 4GB of address space within the next few years.

There is one drawback to the increase in memory space that 64-bit addressing affords. Because memory address values (or *pointers*, in programmer lingo) are now twice as large, they take up twice as much cache space. Pointers normally make up only a fraction of all the data in the cache, but when that fraction doubles, it can squeeze other useful data out of the cache and degrade performance.

NOTE *Some of you who read the preceding discussion would no doubt point out that 32-bit Xeon systems are available with more than 4GB of RAM. Furthermore, Intel allegedly has a fairly simple hack that it could implement to allow its 32-bit systems to address up to 512GB of memory. Still, the cleanest and most future-proof way to address the 4GB ceiling is a 64-bit pointer.*

The 64-Bit Alternative: *x*86-64

When AMD set out alter the *x*86 ISA in order to bring it into the world of 64-bit computing, they took the opportunity to do more than just widen the GPRs. *x*86-64 makes a number of improvements to *x*86, and this section looks at some of them.

Extended Registers

I don't want to get into a historical discussion of the evolution of what eventually became the modern *x*86 ISA, as Intel's hardware went from 4-bit to 8-bit to 16-bit to 32-bit. You can find such discussions elsewhere, if you're interested. I'll only point out that what we now consider to be the "*x*86 ISA" was first introduced in 1978 with the release of the 8086. The 8086 had four 16-bit integer registers and four 16-bit registers that were intended to hold memory addresses but also could be used as integer registers. (The four integer registers, though, could not be used to store memory addresses in 16-bit addressing mode.) This gave the 8086 a total of eight integer registers, four of which could also be used to store addresses.

With the release of the 386, Intel extended the *x*86 ISA to support 32-bit integers by doubling the size of original eight 16-bit registers. In order to access the extended portion of these registers, assembly language programmers used a different set of register mnemonics.

With *x*86-64, AMD has done pretty much the same thing that Intel did to enable the 16-bit to 32-bit transition—it has doubled the sizes of the eight GPRs and assigned new mnemonics to the extended registers. However, extending the existing eight GPRs isn't the only change AMD made to the *x*86 register model.

More Registers

One of the oldest and longest-running gripes about *x*86 is that the programming model has only eight GPRs, eight FPRs, and eight SIMD registers. All newer RISC ISAs support many more architectural registers; the PowerPC ISA, for instance, specifies 32 of each type of register. Increasing the number of registers allows the processor to keep more data where the execution units can access it immediately; this translates into a reduced number of loads and stores, which means less memory subsystem traffic and less waiting for data to load. More registers also give the compiler or programmer more flexibility to schedule instructions so that dependencies are reduced and pipeline bubbles are kept to a minimum.

Modern *x*86 CPUs get around some of these limitations by means of a trick called *register renaming*, described in Chapter 4. Register renaming involves putting extra, "hidden," internal registers onto the die and then dynamically mapping the programmer-visible registers to these internal, machine-visible registers. The Pentium 4, for instance, has 128 of these microarchitectural rename registers, which allow it to store more data closer to the ALUs and reduce false dependencies.

In spite of the benefits of register renaming, it would still be nicer to have more registers directly accessible to the programmer via the *x*86 ISA. This would allow a compiler or an assembly language programmer more flexibility and control to statically optimize the code. It would also allow a decrease in the number of memory access instructions (`loads` and `stores`). In extending *x*86 to 64 bits, AMD has also taken the opportunity to double the number of programmer-visible GPRs and SIMD registers.

When running in 64-bit mode, *x*86-64 programmers have access to eight additional GPRs, for a total of 16 GPRs. Furthermore, there are eight new SIMD registers, added for use in SSE/SSE2 code. So the number of GPRs and SIMD registers available to *x*86-64 programmers has gone from eight each to 16 each. Take a look at Figure 9-3, which contains a diagram from AMD that shows the new programming model.

Notice that they've left the *x*87 floating-point stack alone. This is because both Intel and AMD are encouraging programmers to use SSE/SSE2 for floating-point code, instead of *x*87. I've discussed the reason for this before, so I won't recap it here.

Also notice that the PC is extended. This was done because the PC holds the address of the next instruction, and since addresses are now 64-bit, the PC must be widened to accommodate them.

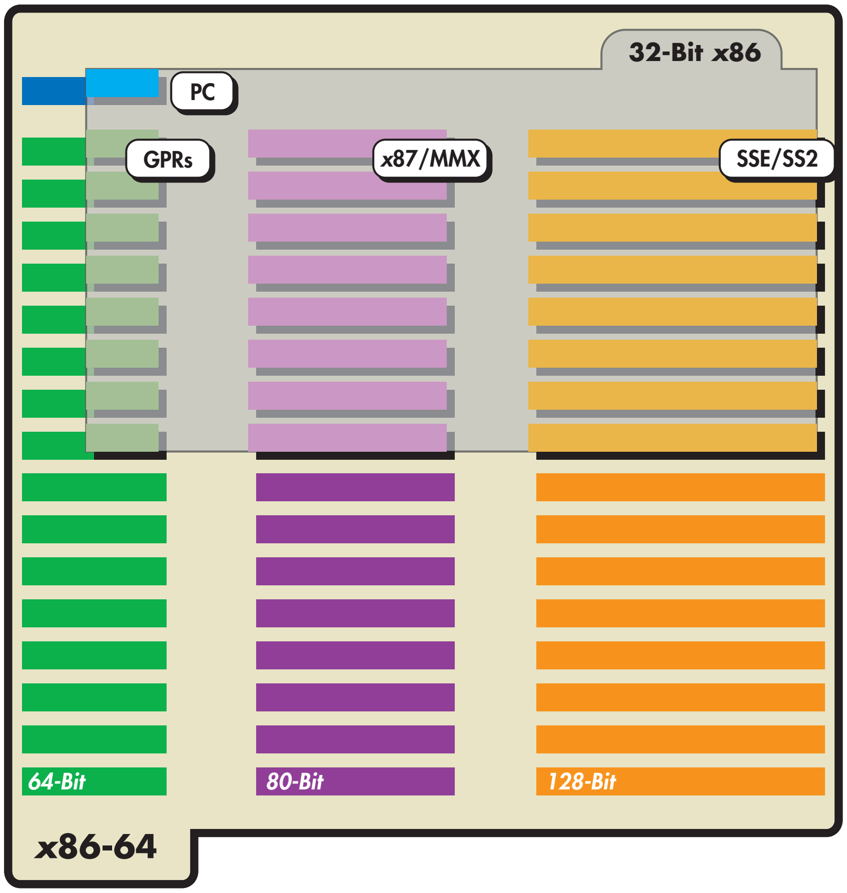


Figure 9-3: The x86-64 programming model

Switching Modes

Full binary compatibility with existing *x*86 code, both 32-bit and older 16-bit flavors, is one of *x*86-64's greatest strengths. *x*86-64 accomplishes this using a nested series of *modes*. The first and least interesting mode is *legacy mode*. When in legacy mode, the processor functions exactly like a standard *x*86 CPU—it runs a 32-bit operating system and 32-bit code exclusively, and none of *x*86-64's added capabilities are turned on. Figure 9-4 illustrates how legacy mode works.

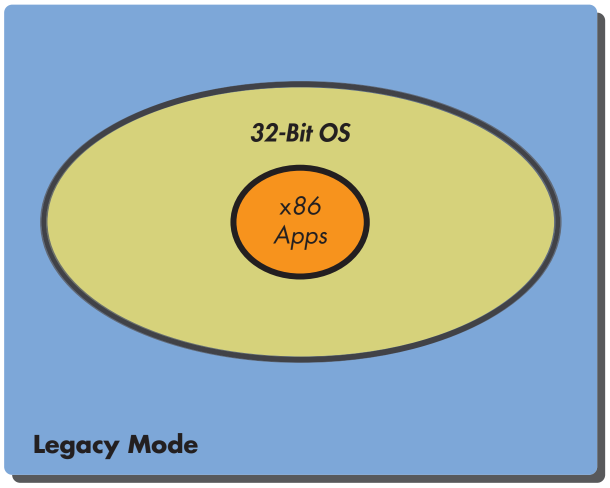


Figure 9-4: x86-64 legacy mode

In short, the Hammer in legacy mode looks like just another *x*86 processor.

It's in the 64-bit *long mode* that things start to get interesting. To run application software in long mode, you need a 64-bit operating system. Long mode provides two submodes—*64-bit mode* and *compatibility mode*—in which the OS can run either *x*86-64 or vanilla *x*86 code. Figure 9-5 should help you visualize how long mode works. (In this figure, x*86 Apps* includes both 32-bit and 16-bit *x*86 applications.)

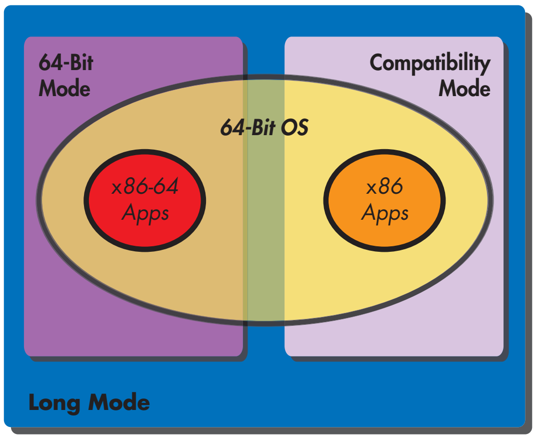


Figure 9-5: x86-64 long mode

So, legacy *x*86 code (both 32-bit and 16-bit) runs under a 64-bit OS in compatibility mode, and *x*86-64 code runs under a 64-bit OS in 64-bit mode. Only code running in long mode's 64-bit submode can take advantage of all the new features of *x*86-64. Legacy *x*86 code running in long mode's compatibility submode, for example, cannot see the extended parts of the registers, cannot use the eight extra registers, and is limited to the first 4GB of memory.

These modes are set for each segment of code on a per-segment basis by means of two bits in the segment's *code segment descriptor*. The chip examines these two bits so that it knows whether to treat a particular chunk of code as 32-bit or 64-bit. Table 9-1 (from AMD) shows the relevant features of each mode.

Table 9-1: *x*86-64 Modes

Mode		Operating System Required	Application Recompile Required	Defaults[1]			
				Address Size (Bits)	Operand Size (Bits)	Register Extensions[2]	GPR Width (Bits)
Long mode[3]	64-bit mode	New 64-bit OS	Yes	64	32	Yes	64
	Compatibility mode		No	32 16		No	32
Legacy mode[4]		Legacy 32-bit or16-bit OS	No	32 16	32 16	No	32

[1] Defaults can be overridden in most modes using an instruction prefix or system control bit.

[2] Register extensions includes eight new GPRs and eight new XMM registers (also called SSE registers).

[3] Long mode supports only *x*86 protected mode. It does not support *x*86 real mode or virtual-8086 mode. Also, it does not support task switching.

[4] Legacy mode supports *x*86 real mode, virtual-8086 mode, and protected mode.

Notice that Table 9-1 specifies 64-bit mode's default integer size as 32 bits. Let me explain.

We've already discussed how only the integer and address operations are really affected by the shift to 64 bits, so it makes sense that only those instructions would be affected by the change. If all the addresses are now 64-bit, there's no need to change anything about the address instructions apart from their default pointer size. If a `load` in 32-bit legacy mode takes a 32-bit address pointer, then a `load` in 64-bit mode takes a 64-bit address pointer.

Integer instructions, on the other hand, are a different matter. You don't always need to use 64-bit integers, and there's no need to take up cache space and memory bandwidth with 64-bit integers if your application needs only smaller 32- or 16-bit ones. So it's not in the programmer's best interest to have the default integer size be 64 bits. Hence, the default data size for integer instructions is 32 bits, and if you want to use a larger or smaller integer, you must add an optional *prefix* to the instruction that overrides the default. This prefix, which AMD calls the *REX prefix* (presumably for *register extension*), is one byte in length. This means that 64-bit instructions are one byte longer, a fact that makes for slightly increased code sizes.

Increased code size is bad, because bigger code takes up more cache and more bandwidth. However, the effect of this prefix scheme on real-world code size depends on the number of 64-bit integer instructions in a program's instruction mix. AMD estimates that the average increase in code size from *x*86 code to equivalent *x*86-64 code is less than 10 percent, mostly due to the prefixes.

It's essential to AMD's plans for *x*86-64 that there be no performance penalty for running in legacy or compatibility mode versus long mode. The two backward-compatibility modes don't give you the performance-enhancing benefits of *x*86-64 (specifically, more registers), but they don't incur any added overhead, either. A legacy 32-bit program simply ignores *x*86-64's added features, so they don't affect it one way or the other.

Out with the Old

In addition to beefing up the *x*86 ISA by increasing the number and sizes of its registers, *x*86-64 also slims it down by kicking out some of the older and less frequently used features that have been kept thus far in the name of backward compatibility.

When AMD's engineers started looking for legacy *x*86 features to jettison, the first thing to go was the segmented memory model. Programs written to the *x*86-64 ISA use a flat, 64-bit virtual address space. Furthermore, legacy *x*86 applications running in long mode's compatibility submode must run in protected mode. Support for real mode and virtual-8086 mode are absent in long mode and available only in legacy mode. This isn't too much of a hassle, though, since, except for a few fairly old legacy applications, modern *x*86 apps use protected mode.

Conclusion

*x*86 wasn't the only consumer ISA to make the 64-bit leap in the first few years of the 21st century. IBM's 970, more popularly known as the G5, brought the PowerPC ISA moved into the same commodity 64-bit desktop and server space as AMD's Hammer. The next chapter will take an in-depth look at the G5, comparing it to the processors we've studied so far.

10

THE G5: IBM'S POWERPC 970

The last PowerPC processor to succeed the Motorola 74*xx* family as the heart of Apple's workstation line is the IBM PowerPC 970—the processor in Apple's G5 computer. This chapter takes an in-depth look at this processor, comparing it to Motorola's 7455 and, where appropriate, Intel's Pentium 4.

I'll begin by taking a look at the 970's overall design philosophy, and then I'll step through the stages of the 970's pipeline, much as we did in the previous two chapters on the Pentium 4 and G4e. Then we'll talk about instruction fetching, decoding, dispatching, issuing, execution, and completion, and we'll end with a look at the 970's back end.

At the outset, I should note that one of the most significant features of the 970 is its 64-bit integer and address hardware. If you want to learn more about 64-bit computing—what it can and cannot do, and what makes a 64-bit processor like the 970 different from 32-bit processors like the G4e and Pentium 4—make sure to read Chapter 9.

NOTE *With the exception of the section on vector processing, most of the discussion below is relevant to IBM's POWER4 microarchitecture—the multiprocessor server microarchitecture on which the PowerPC 970 is based.*

Overview: Design Philosophy

In the previous chapters' comparison of the design philosophies behind the Pentium 4 and G4e, I tried to summarize each processor's overall approach to organizing its execution resources to squeeze the most performance out of today's software. I characterized the G4e's approach as *wide and shallow*, because the G4e moves a few instructions through its very wide back end in as few clock cycles as possible. I contrasted this approach to the Pentium 4's *narrow and deep* approach, which focuses on pushing large numbers of instructions through a narrow back end over the course of a many clock cycles.

Using similar language, the 970's approach could be characterized as *wide and deep*. In other words, the 970 wants to have it both ways: an extremely wide back end and a 14-stage (integer) pipeline that, while not as deep as the Pentium 4's, is nonetheless built for speed. Using a special technique, which we'll discuss shortly, the 970 can have a whopping 200 instructions on-chip in various stages of execution, a number that dwarfs not only the G4e's 16-instruction window but also the Pentium 4's 126-instruction one.

You can't have everything, though, and the 970 pays a price for its "more is better" design. When we discuss instruction dispatching and out-of-order execution on the 970, you'll see what trade-offs IBM made in choosing this design.

Figure 10-1 shows the microarchitecture's main functional blocks, and it should give you a preliminary feel for just how wide the 970 is.

Don't worry if all the parts of Figure 10-1 aren't immediately intelligible, because by the time this chapter is finished, you'll have a good understanding of everything depicted there.

Caches and Front End

Let's take a short look at the caches for the 970 and the G4e. Table 10-1 should give you a rough idea of how the two chips compare.

Table 10-1: The Caches of the PowerPC 970 and G4e

	L1 I-cache	L1 D-cache	L2 Cache
PowerPC 970	64KB, direct-mapped	32KB, two-way assoc.	512KB, eight-way assoc.
G4e	32KB, eight-way assoc.	32KB, eight-way assoc.	256KB, eight-way assoc.

Table 10-1 shows that the 970 sports a larger instruction cache than the G4e. This is because the 970's pipeline is roughly twice the length of the G4e's, which means that like the Pentium 4, the 970 pays a much higher performance penalty when its pipeline stalls due to a miss in the I-cache. In short, the 970's 64KB I-cache is intended to keep pipeline bubbles out of the chip's pipeline.

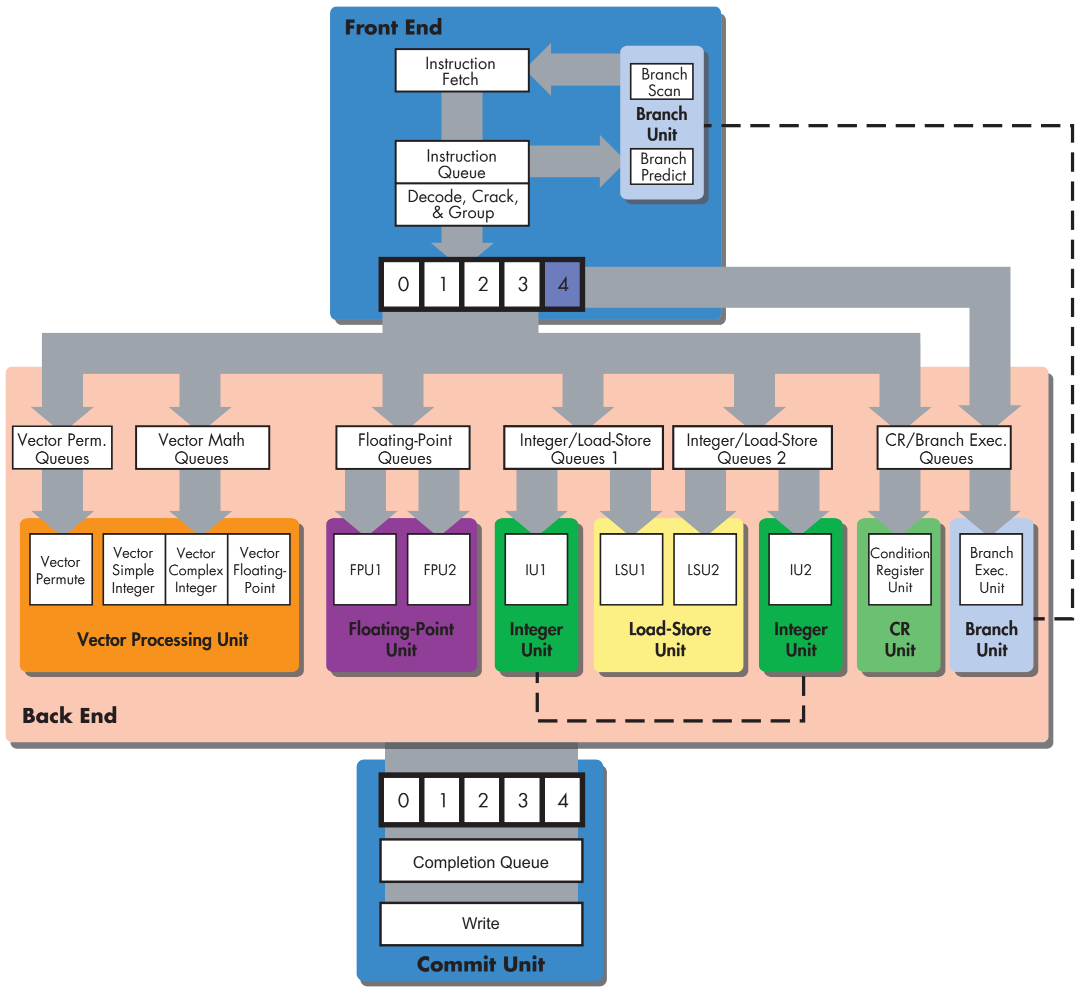


Figure 10-1: The IBM PowerPC 970

When you combine the 970's 32KB D-cache with its sizable 512KB L2, its high-speed double data rate (DDR) front-side bus and its support for up to eight data prefetch streams, you can see that this chip was made for floating-point- and SIMD-intensive media applications. This chip performs better on vector code than the G4e just based on these features alone.

Branch Prediction

Because of the depth of its pipeline and the width of its back end, the 970's designers spent a sizable chunk of the chip's resources on branch prediction. Like a high-hit-rate instruction cache, accurate branch prediction is essential if the 970 is to keep its pipeline full and its extensive execution resources in constant use. As such, the 970's extremely robust branch prediction unit (BPU) is one of its greatest strengths. This section takes a closer look at the top half of the 970's front end and at the role that branch prediction plays in steering that front end through the instruction stream.

The 970's instruction fetch logic fetches up to eight instructions per cycle from the L1 I-cache into an instruction queue, and on each fetch, the front end's branch unit scans these eight instructions in order to pick out up to two branches. If either of the two branches is conditional, the branch prediction unit predicts the condition's outcome (taken or not taken) and possibly its target address using one of two branch prediction schemes.

The first branch prediction scheme employed by the 970 is the standard BHT-based scheme first described in Chapter 5. The 970's BHT has 16 K entries—four times the number of entries in the Pentium 4's BHT and eight times the number of entries in the G4's BTIC. For each of these 16 K entries, a one-bit flag tells the branch predictor whether the branch should be taken or not taken.

The second scheme involves another 16 K entry table called the *global predictor table*. Each entry in this global predictor table is associated with an 11-bit vector that records the actual execution path taken by the previous 11 *fetch groups*. The processor uses this vector, which it constantly keeps up to date with the latest information, to set another one-bit flag for the global predictor table that specifies whether the branch should be taken or not taken.

Finally, there's a third 16 K entry table that's used by the 970's front end to keep track of which of the two schemes works best for each branch. When a branch is finally evaluated, the processor compares the success of both schemes and records in this selector table which scheme has done the best job so far of predicting the outcome of that particular branch.

Spending all those transistors on such a massive amount of branch prediction resources may seem like overkill right now, but when you've completed the next section, you'll see that the 970 can't afford to introduce any unnecessary bubbles into its pipeline.

The Trade-Off: Decode, Cracking, and Group Formation

As noted earlier, IBM's PowerPC 970 fetches eight instructions per cycle from the L1 cache into an instruction queue, from which the instructions are pulled for decoding at a rate of eight per cycle. This compares quite favorably to the G4e's four instructions per cycle fetch and decode rate.

Much like the Pentium 4 and its predecessor, the P6, the PowerPC 970 translates PowerPC instructions into an 86-bit internal instruction format that not only makes the instructions easier for the back end to schedule, but also explicitly encodes register dependency information. IBM calls these internal instructions *IOPs*, presumably short for *internal operations*. Like micro-ops on the Pentium 4, it is these IOPs that are actually executed out of order by the 970's back end. And also like micro-ops, cracking instructions down into multiple, more atomic and more strictly defined IOPs can help the back end squeeze out some extra instruction-level parallelism (ILP) by giving it more freedom to schedule code.

One important difference to note is that the architected PowerPC instructions are very close in form and function to the 970's IOPs, and in fact, the latter are probably just a restricted subset of the former. (That this is definitely true is not clear from the publicly available documentation.) The Pentium 4, in contrast, uses an internal instruction format that is significantly different in many important respects from the *x*86 ISA. So the process of "ISA translation" on the 970 is significantly less complex and less resource-intensive than the analogous process on the Pentium 4.

Almost all the PowerPC ISA instructions, with a few exceptions, translate into exactly one IOP on the 970. Of the instructions that translate into more than one IOP, IBM distinguishes two types:

- A *cracked* instruction is an instruction that splits into exactly two IOPs.
- A *millicoded* instruction is an instruction that splits into more than two IOPs.

This difference in the way instructions are classified is not arbitrary. Rather, it ties into a very important design decision that the POWER4's designers made regarding how the chip tracks instructions at various stages of execution.

Dispatching and Issuing Instructions on the PowerPC 970

If you take a look at the middle of the large PPC 970 diagram in Figure 10-1, notice that right below the *Decode, Crack, and Group* phase I've placed a group of five boxes. These five boxes represent what IBM calls a *dispatch group* (or *group*, for short), and each group consists of five IOPs arranged in program order according to certain rules and restrictions. It is these organized and packaged groups of five IOPs that the 970 dispatches in parallel to the six issue queues in its back end.

I probably shouldn't go any further in discussing how these groups work without first explaining the reason for their existence. By assembling IOPs together into specially ordered groups of five for dispatch and completion, the 970 can track these groups, and not individual IOPs, through the various stages of execution. So instead of tracking 100 individual IOPs in-flight as they work their way through the 100 or so execution slots available in the back end, the 970 need track only 20 groups. IOP grouping, then, significantly reduces the overhead associated with tracking and reordering the huge volume of instructions that can fit into the 970's deep and wide design.

As noted earlier, the 970's peculiar group dispatch scheme doesn't go into action until after the decode stage. Decoded PowerPC instructions flow from the bottom of the 970's instruction queue in order and fill up the five available dispatch slots in a group (four non-branch instructions and one branch instruction). Once the five slots have been filled from the instruction queue, the entire group is dispatched to the back end, where the individual instructions that constitute it (`loads`, `stores`, `adds`, `fadds`, etc.) enter the tops of their respective issue queues (i.e., an `add` goes into an integer issue queue, a `fadd` goes into a floating-point issue queue, etc.).

When the individual IOPs in a group reach their proper issue queues, they can then be issued out of program order to the execution units at a rate of eight IOPs/cycle for all the queues combined. Before they reach the completion stage, however, they need to be placed back into their group so that an entire group of five IOPs can be completed on each cycle. (Don't worry if this sounds a bit confusing right now. The group dispatch and formation scheme will become clearer when we discuss the 970's peculiar issue queue structure.)

The price the 970 pays for the reduced bookkeeping overhead afforded it by the dispatch grouping scheme is a loss of execution efficiency brought on by the diminished granularity of control that comes from being able to dispatch, schedule, issue, and complete instructions on an individual basis. Let me explain.

The 970's Dispatch Rules

When the 970's front end assembles an IOP group, there are certain rules it must follow. The first rule is that the group's five slots must be populated with IOPs in program order, starting with the oldest IOP in slot 0 and moving up to newest IOP in slot 4. Another rule is that all branch instructions must go in slot 4, and slot 4 is reserved for branch instructions only. This means that if the front end can't find a branch instruction to put in slot 4, it can issue one less instruction that cycle.

Similarly, there are some situations in which the front end must insert noops into the group's slots in order to force a branch instruction into slot 4. *Noop* (pronounced "no op") is short for *no operation*. It is a kind of non-instruction instruction that means "Do nothing." In other words, the front end must sometimes insert empty execution slots, or pipeline bubbles, into the instruction stream in order to make the groups comply with the rules.

The preceding rules aren't the only ones that must be adhered to when building groups. Another rule dictates that instructions destined for the conditional register unit (CRU) can go only in slots 0 and 1.

And then there are the rules dealing with cracked and millicoded instructions. Consider the following from IBM's POWER4 white paper:

> Cracked instructions flow into groups as any other instructions with one restriction. Both IOPs must be in the same group. If both IOPs cannot fit into the current group, the group is terminated and a new group is initiated. The instruction following the cracked instruction may be in the same group as the cracked instruction, assuming there is room in the group. Millicoded instructions always start a new group. The instruction following the millicoded instruction also initiates a new group.

And that's not all! A group has to have the following resources available before it can even dispatch to the back end. If just one of following resources is too tied up to accommodate the group or any of its instructions, then the entire group has to wait until that resource is freed up before it can dispatch:

Group completion table (GCT) entry

The group completion table is the 970's equivalent of a reorder buffer or completion queue. While a normal ROB keeps track of individual in-flight instructions, the GCT tracks whole dispatch groups. The GCT has 20 entries for keeping track of 20 active groups as the groups' constituent instructions make their way through the ~100 execution slots available in the back end's pipelines. Regardless of how few instructions are actually in the back end at a given moment, if those instructions are grouped so that all 20 GCT entries happen to be full, no new groups can be dispatched.

Issue queue slot

If there aren't enough slots available in the appropriate issue queues to accommodate all of a group's instructions, the group must wait to dispatch. (In a moment I'll elaborate on what I mean by "appropriate issue queues.")

Rename registers

There must be enough register rename resources available so that any instruction that requires register renaming can issue when it's dispatched to its issue queue.

Again, when it comes to the preceding restrictions, one bad instruction can keep the whole group from dispatching.

Because of its use of groups, the 970's dispatch bandwidth is sensitive to a complex host of factors, not the least of which is a sort of "internal fragmentation" of the group completion table that could potentially arise and needlessly choke dispatch bandwidth if too many of the groups in the GCT are partially or mostly empty.

In order to keep dispatch bottlenecks from stopping the fetch/decode portion of the pipeline, the 970 can buffer up to four dispatch groups in a four-entry *dispatch queue*. So if the preceding requirements are not met and there is space in the dispatch queue, a dispatch group can move into the queue and wait there for its dispatch requirements to be met.

Predecoding and Group Dispatch

The 970 uses a trick called *predecoding* in order to move some of the work of group formation higher up in the pipeline, thereby simplifying and speeding up the latter decode and group formation phases in the front end. As instructions are fetched from the L2 cache into the L1 I-cache, each instruction is predecoded and marked with a set of five predecode bits. These bits indicate how the instruction should be grouped—i.e., if it should be first in its group, last in its group, or unrestricted; if it will be a microcoded instruction; if it will trigger an exception; if it will be split or not; and so on. This information is used by the decode and group formation hardware to quickly route instructions for decoding and to group them for dispatch.

The predecode hardware also identifies branches and marks them for type—conditional or unconditional. This information is used by the 970's branch prediction hardware to implement branch folding, fall-through, and branch prediction with minimal latencies.

Some Preliminary Conclusions on the 970's Group Dispatch Scheme

In the preceding section, I went into some detail on the ins and outs of group formation and group dispatching in the 970. If you only breezed through the section and thought, "All of that seems like kind of a pain," then you got 90 percent of the point I wanted to make. Yes, it is indeed a pain, and that pain is the price the 970 pays for having both width and depth at the same time. The 970's big trade-off is that it needs less logic to support its long pipeline and extremely wide back end, but in return, it has to give up a measure of granularity, flexibility, and control over the dispatch and issuing of its instructions. Depending on the makeup of the instruction stream and how the IOPs end up being arranged, the 970 could possibly end up with quite a few groups that are either mostly empty, partially empty, or stalled waiting for execution resources.

So while the 970 may be theoretically able to accommodate 200 instructions in varying stages of fetch, decode, execution, and completion, the reality is probably that under most circumstances, a decent number of its execution slots will be empty on any given cycle due to dispatch, scheduling, and completion limitations. The 970 makes up for this with the fact that it just has so many available slots that it can afford to waste some on group-related pipeline bubbles.

The PowerPC 970's Back End

The PowerPC 970 sports a total of 12 execution units, depending on how you count them. Even a more conservative count that lumps together the three SIMD integer and floating-point units and doesn't count the branch execution unit would still give nine execution units.

In the following three sections, I'll discuss each of the 970's execution units, comparing them to the analogous units on the G4e, and in some cases the Pentium 4. As the discussion develops, keep in mind that a simple comparison of the types and numbers of execution units for each of the three processors is not at all adequate to the task of sorting out the real differences between the processors. Rather, there are complicating factors that make comparisons much more difficult than one might naïvely expect. Some of these factors will be evident in the sections dealing with each type of execution unit, but others won't turn up until we discuss the 970's issue queues in the last third of the chapter.

NOTE *As I cover each part of the 970's back end, I'll specify the number of rename registers of each type (integer, floating-point, vector) that the 970 has. If you compare these numbers to the equivalent numbers for the G4e, you'll see that 970 has many more rename registers than its predecessor. This increased number of rename registers is necessary because the 970's instruction window (up to 200 instructions in-flight) is significantly higher than that of the G4e (up to 16 instructions in-flight). The more instructions a processor can hold in-flight at once, the more rename registers it needs in order to pull off the kinds of tricks that a large instruction window enables you to do—i.e., dynamic scheduling, loop unrolling, speculative execution, and the like. In a nutshell, more instructions on the chip in various stages of execution means more data needs to be stored in more registers.*

Integer Units, Condition Register Unit, and Branch Unit

In the chapters on the Pentium 4 and G4e, I described how both of these processors embody a similar approach to integer computation in that they divide integer operations into two types: simple and complex. Simple integer instructions, like `add`, are the most common type of integer instruction and take only one cycle to execute on most hardware. Complex integer instructions (e.g., integer division) are rarer and take multiple cycles to execute.

In keeping with the quantitative approach to computer design's central dictum, "Make the common case fast,"[1] both the Pentium 4 and G4e split up their integer hardware into two specialized types of execution units: a group of units that handle only simple, single-cycle instructions and a single unit that handles complex, multi-cycle instructions. By dedicating the majority of their integer hardware solely to the rapid execution of the most common types of instructions (the simple, single-cycle ones), the Pentium 4 and the G4e are able to get increased integer performance out of a smaller amount of overall hardware.

Think of the multiple fast IUs (or SIUs) as express checkout lanes for one-item shoppers and the single slow IU as a general-purpose checkout lane for multiple-item shoppers in a supermarket where most of the shoppers only buy a single item. This kind of specialization keeps that one guy who's stocking up for Armageddon from slowing down the majority of shoppers who just want to duck in and grab eggs or milk on the way home from work.

The PPC 970 differs from both of these designs in that it has two general-purpose IUs that execute almost all integer instructions. To return to the supermarket analogy, the 970 has two general-purpose checkout lanes in a supermarket where most of the shoppers are one-item shoppers. The 970's two IUs are attached to 80 64-bit GPRs (32 architected and 48 rename).

Why doesn't the 970 have more specialized hardware (express checkout lanes) like the G4e and Pentium 4? The answer is complicated, and I'll take an initial stab at answering it in a moment, but first I should clear something up.

The Integer Units Are Not Fully Symmetric

I said that the 970's two IUs execute "almost all" integer instructions, because the units are not, in fact, fully symmetric. One of the IUs performs fixed-point divides, and the other handles SPR operations. So the 970's IUs are slightly specialized, but not in the same manner as the IUs of the G4e and Pentium 4. If the G4e and Pentium 4 have express checkout lanes, the 970 has something more akin to a rule that says, "All shoppers who bought something at the deli must go through line 1, and all shoppers who bought something at the bakery must go through line 2; everyone else is free to go through either line."

Thankfully, integer divides and SPR instructions are relatively rare, so the impact on performance of this type of forced segregation is minimized. In fact, if the 970 didn't have the group formation scheme, this seemingly

[1] See John L. Hennessy and David A. Patterson, *Computer Architecture: A Quantitative Approach*, Third Edition (Morgan Kauffman Publishers: 2003).

minor degree of specialization might hardly be worth commenting on. But as it stands, group-related scheduling issues turn this specialization into a potentially negative factor—albeit a minor one—for integer performance for reasons that we'll discuss later on in this chapter.

Integer Unit Latencies and Throughput

The vast majority of integer instructions take one cycle to execute on the 970, while a few more complex integer instructions can take more cycles. Note that this one-cycle number signifies integer throughput. As a result, simple, non-dependent integer IOPs can issue and finish at a rate of one per cycle. Dependent integer IOPs, on the other hand, must be separated by a dead cycle, giving a latency of two cycles. Note that this two-cycle latency applies to operations in the same IU or in different IUs.

In the end, this unfortunate increase in latency is somewhat than mitigated by other factors, which I'll discuss shortly. Nonetheless, the two-cycle latency issue has turned out to have a non-trivial impact on the 970's integer performance.

The CRU

I haven't said all there is to say about the 970's integer-processing capabilities, so the preceding summary isn't quite complete. As I mentioned, the 970 divides up its integer resources in a slightly different way from that of the Pentium 4 or G4e.

Some of the operations handled by the integer units on the Pentium 4 and G4e are instead handled in different places by the 970. Specifically, there's one type of fixed-point operation normally handled by an integer unit that in the 970's case gets its own separate execution unit. The 970 has a dedicated unit for handling logical operations related to the PPC's condition register: the *CRU*. On the G4e these condition register (CR) operations are handled by the complex integer unit. Thus the 970, in giving these operations their own separate unit, takes some of the processing load off of the two integer units.

The PowerPC Condition Register

The CR is a part of the PowerPC ISA that handles some of the functions of the *x*86's processor status word (PSW), but in a more flexible and programmer-friendly way. Almost all PowerPC arithmetic operations, when they finish executing, have the option of setting various bits (or flags) in the PPC's 32-bit condition register as a way of recording information about their outcome for future reference (in other words, Was the result positive or negative? Did it overflow or underflow the register?). So you might think of the CR as a place to store metadata for arithmetic results. Subsequent instructions, like conditional branch instructions, can then check the CR's flags and thereby use that metadata to decide what to do.

The flag combinations that instructions can set in the condition register are called *condition codes*, and the condition register has room enough to store up to eight separate condition codes, which can describe the outcome of eight different instructions. Enabling the programmer to manipulate those condition codes are a collection of PowerPC instructions that perform logical operations (AND, OR, NOT, NOR, etc.) directly on the flags in the CR. The CRU is the execution unit that executes those instructions.

Preliminary Conclusions About the 970's Integer Performance

To summarize, the G4e dedicates the majority of its integer hardware to the execution of simple, one-cycle instructions, with the remainder of the hardware dedicated to the execution of less common complex instructions. In contrast, with two rare exceptions, all of the PPC 970's integer hardware can execute almost any type of integer instruction. Table 10-2 shows a breakdown of how the three types of integer operations we've discussed—simple, complex, and CR logical—are handled by the G4e and the 970.

Table 10-2: Integer Operations on the PowerPC 970 and G4e

	Simple Int.	Complex Int.	CR Logical	SPR
PPC 970	IU1, IU2	IU1, IU2[1]	CRU	IU1
G4e	SIU1-SIU3	CIU	CIU	CIU

[1] IU2 handles all divides on the 970.

As you can see from Table 10-2, the G4e has more and more specialized integer hardware at its disposal than the 970. Also, in terms of instruction latencies, the G4e's integer hardware is slightly faster for common, simple integer operations than that of the 970. Finally, as I hinted at earlier and will develop in more detail shortly, the 970's integer performance—as well its performance on other types of code—is fairly sensitive to compiler optimization and scheduling. All of this adds up to make 32-bit integer computation the place where the 970 looks the weakest compared to the competition.

Load-Store Units and Front-Side Bus

Chapter 1 discussed the difference between instructions that access memory (`load`s and `store`s) and instructions that do actual computation (integer instructions, floating-point instructions, etc.). Just like integer instructions are executed in the IUs and floating-point instructions are executed in the FPUs, memory access instructions have their own specialized execution units in the form of one or more load-store units (LSUs).

Chapter 1 also discussed the fact that in order to access memory via a `load` or a `store`, it's usually necessary to perform an address calculation so the processor can figure out the location in memory that it should access. Even though such address calculations are just simple integer operations, they're usually not handled by the processor's integer hardware. Instead, all of the processors under discussion here have dedicated address generation hardware as part of their LSUs. Consequently, address generation takes place as part of the execution phase of the LSU's pipeline.

The G4e has one LSU that executes all of the `loads` and `stores` for the entire chip (integer, floating-point, and vector). As mentioned earlier, the G4e's LSU contains dedicated integer hardware for address calculations.

The 970 has two identical LSUs execute all of the `loads` and `stores` for the entire chip. This gives it literally twice the load-store hardware of the G4e, which it needs in order to keep all the instructions in its much larger instruction window fed with data. The 970's load-store hardware is more comparable to that of the Pentium 4, which also features a larger instruction window.

Front-Side Bus

A *bus* is an electrical conduit that connects two components in a computer system, allowing them to communicate and share data and/or code. If a computer system is like a large office building, then buses are like the phone lines that keep all the employees connected to each other and to the outside world.

The *front-side bus (FSB)* is the bus that connects the computer's CPU to the core logic chipset. If buses are the phone lines of a computer system, then the core logic chipset is the operator and switchboard. The *core logic chipset*, or *chipset* for short, opens and closes bus connections between components and routes data around the system. Figure 10-2 shows a simple computer system consisting of a CPU, RAM, a video card, a hard drive, and a chipset.

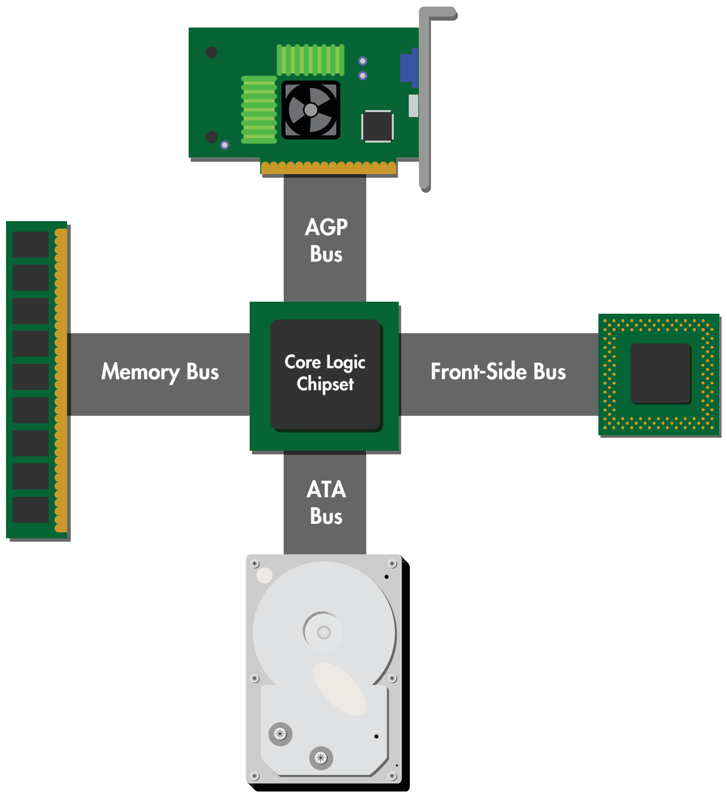


Figure 10-2: Core logic chipset

As you can see from Figure 10-2, the front-side bus is the processor's sole means of communication with the rest of the system, so it needs to be very fast.

A processor's front-side bus usually operates at a clock speed that is some fraction of the core clock speed of the CPU, and the 970 is no different. On the first release of Apple's G5 towers, the 970's front-side bus operates at half the clock speed of the 970. So for a 2 GHz 970, the FSB runs at 1 GHz DDR. (DDR stands for *double data rate*, which means that the bus physically runs at 500 MHz, but data is transferred on the rising and falling edges of the clock pulse.) The 970 can run at other multiples of the FSB clock, including three, four, and six times the FSB clock speed.

Because the 970's front-side bus is composed of two unidirectional channels, each of which is 32 bits wide, the total theoretical peak bandwidth for the 900 MHz bus is 7.2GB per second. Address and control information is multiplexed onto the bus along with data, so IBM estimates that the bus's total peak bandwidth for data alone (after subtracting the bandwidth used for address and control information) is somewhere around 6.4GB per second.

In the end, this high-bandwidth FSB is one of the 970's largest performance advantages over its competitors. The 970's two LSUs place high demands on the FSB in order to keep the 970's large instruction window full and its wide back end fed with data. When coupled with a high-bandwidth memory subsystem like dual-channel DDR400, the 970's fast FSB and dual LSUs make it a great media workstation platform.

The Floating-Point Units

The G4e's very straightforward floating-point implementation has a single FPU that takes a minimum of five cycles to finish executing the fastest floating-point instructions. (Some instructions take many more cycles.) The FPU is served by 48 microarchitectural floating-point registers (32 registers for the PPC ISA and 16 additional rename registers). Finally, single- and double-precision floating-point operations take the same amount of time.

The 970's floating-point implementation is almost exactly like the G4e's, except there's twice as much hardware. The 970 has two identical FPUs, each of which can execute the fastest floating-point instructions (like the `fadd`) in six cycles. As with the G4e, single- and double-precision operations take the same amount of time to execute. The 970's two FPUs are fully pipelined for all operations except floating-point divides, which are very costly in terms of cycles and stall both FPUs. And finally, the 970 has a larger number of FPRs: 80 total microarchitectural registers, where 32 are PowerPC architectural registers and the remaining 48 are rename registers.

Before moving on, I should note one peculiarity in the way that the 970 handles floating-point instructions. Some floating-point instructions—particularly the fused multiply-add series of instructions—are not translated into IOPs by the decode hardware, but instead are executed directly by the FPU. The reason for this is fairly straightforward.

Recall from Chapter 4 that the amount of die space taken up by the register file increases approximately with the square of the number of register file ports. The vast majority of PowerPC instructions specify at most two source registers and one destination register, which means that they use at most two register file read ports and one register file write port. However, there is a handful of PowerPC instructions that require more ports. In order to keep the size of the 970's register files to a minimum, the 970's designers opted not to add more ports to the register files in order to accommodate this small number of instructions. Instead, the 970 has two ways of keeping this small group of instructions from stalling due to the structural hazards that might be brought on by their larger-than-average port requirements:

- Non–floating-point instructions that require more than three ports total are dealt with at the decode stage. All of the 970's IOPs are restricted to read at most two registers and write at most one register, so instructions that do not obey this restriction are cracked into multiple IOPs that do obey it.
- There are a few types of floating-point instructions that do not fit the three-port requirement, the most common of which is the fused multiply-add series of instructions. For performance reasons, cracking a floating-point fused multiply-add instruction into multiple IOPs is neither

necessary nor desirable. Instead, such instructions are simply passed to the FPU as decoded PowerPC instructions—not as IOPs—where they execute by accessing the register file on more than one cycle. Since these instructions take multiple cycles to execute anyway, the extra register file read and/or write cycles are simply overlapped with computation cycles so that they don't add to the instruction's latency.

As the preceding discussion indicates, the 970's floating-point hardware compares quite favorably with that of the G4e. Twice the hardware does not necessarily equal twice the performance, but it's clear that the 970 performs significantly better, clock for clock, on floating-point code. This performance advantage is not only due to the doubled amount of floating-point hardware, but also to the 970's longer pipeline and clock-speed advantage. Furthermore, the 970's fast front-side bus (effectively half the core clock speed), when coupled with a high-bandwidth memory subsystem, gives it a distinct advantage over the G4e in bandwidth-intensive floating-point code. Note that this last point also applies to vector code, but more on that in a moment.

Vector Computing on the PowerPC 970

The G4e's AltiVec implementation is the strongest part of its design. With four fully pipelined vector processing units, it has plenty of hardware to go around. As a brief recap from the last chapter, here's a breakdown of the G4e's four AltiVec units:

- vector permute unit (VPU)
- vector simple integer unit (VSIU)
- vector complex integer unit (VCIU)
- vector floating-point unit (VFPU)

All of this vector execution hardware is tied to a generous register file that consists of thirty-two 128-bit architectural registers and sixteen additional vector rename registers. Furthermore, each of the units is attached to a four-entry vector issue queue that can issue two vector ops per cycle to any of the four vector units.

NOTE *The SIMD instruction set known as AltiVec was codeveloped by both IBM and Motorola, and it is co-owned by both of them. Motorola has a trademark on the AltiVec name, so in most of IBM's literature (but not all), the instruction set is referred to as VMX, presumably for Vector Multimedia Extensions. VMX and AltiVec are therefore two different names for the same group of 162 vector instructions added to the PowerPC ISA. This chapter uses the name AltiVec for both the G4e and the 970.*

The 970's AltiVec implementation looks pretty much like the original G4's—the MPC7400—except for the addition of issue queues for dynamic execution. The 970 has the following units:

- Vector permute unit (VPU)
- Vector arithmetic logic unit (VALU)

- Vector simple integer unit (VSIU)
- Vector complex integer unit (VCIU)
- Vector floating-point unit (VFPU)

These units are attached to 80 vector registers—32 architectural registers and 48 rename registers.

Notice that the vector execution units listed here are essentially the same units as in the G4e, but they're grouped differently. This grouping makes all the difference. Take a look at the simplified diagram in Figure 10-3.

The 970 can dispatch up to four vector IOPs per cycle total to its two vector *logical issue queues*—two IOPs per cycle in program order to the 16-entry VPU logical issue queue and two IOPs per cycle in program order to the 20-entry VALU logical issue queue. (Each of these logical issue queues consists physically of a pair of interleaved queues that operate together as a single queue, but we'll talk more about how the logical queues are actually implemented in a moment.) Each of the two logical queues can then issue one vector operation per cycle to any of the units attached to it. This means that the 970 can issue one IOP per cycle to the VPU and one IOP per cycle to any of the VALU's three subunits.

As you can see, if you place the 970's vector unit and the G4e's vector unit side by side, the 970 and the G4e can issue the same number of vector instructions per cycle to their AltiVec units, but the 970 is much more limited in the combinations of instructions it can issue, because one of its two instructions must be a vector permute. For instance, the G4e would have no problem issuing both a complex integer instruction and a simple integer instruction to its VCIU and VSIU in the same cycle, whereas the 970 would only be able to issue one of these in a cycle.

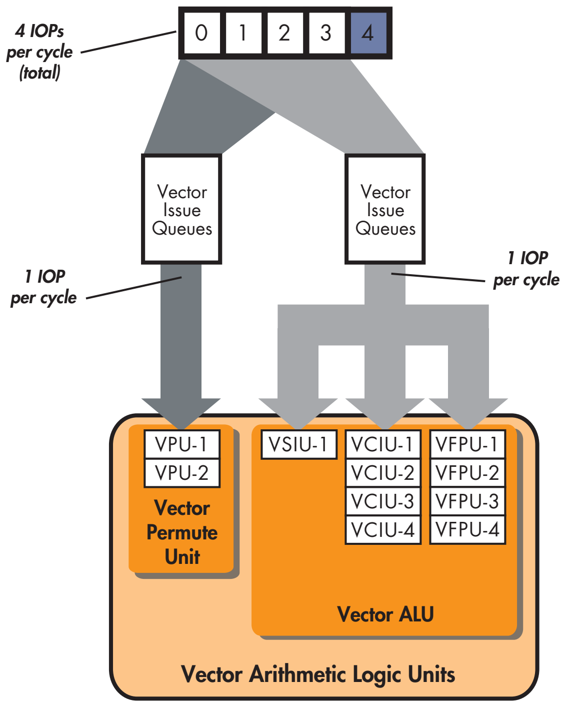


Figure 10-3: The 970's vector unit

Table 10-3 replicates a chart from Apple that compares the AltiVec execution unit latencies for the two G4s and the 970.[2]

Table 10-3: Vector Instruction Latencies for the G4, G4e, and 970

Hardware Unit	7400/7410	7450/7455	PPC 970
Vector simple integer unit (VSIU)	1	1	2[1]
Vector complex integer unit (VCIU)	3	4	5*
Vector floating-point unit (VFPU)	4 (5)[2]	4 (5)†[3]	8*
Vector permute unit (VPU)	1	2	2[4]

[1] An extra cycle latency is required if the data is next used in VPU.

[2] The VFPU takes an extra cycle if Java mode is turned on. (It is off by default on Mac OS X, but on by default on Mac OS 9.)

[3] Some FP-related VSIU instructions were moved to the VFPU for later G4. These only take two cycles instead of the usual 4 (5).

[4] An extra cycle latency is required if the data is next used in VCIU/VSIU/VFPU.

Notice that the 970's vector instruction latencies are close to those in the G4e, which is an excellent sign given the fact that the 970's pipeline is longer and its issue queues are much deeper (18 instructions on the 970 versus 4 instructions on the G4e). The 970's larger instruction window and deeper issue queues allow the processor to look farther ahead in the instruction

[2] This is a truncated version of Apple's *AltiVec Instruction Cross-Reference* (http://developer.apple.com/hardware/ve/instruction_crossref.html).

stream and extract more instruction-level parallelism (ILP) from the vector instruction stream. This means that the 970's vastly superior dynamic scheduling hardware can squeeze more performance out of slightly inferior vector execution hardware, a capability that, when coupled with a high-bandwidth memory subsystem and an ultrafast front-side bus, enable its vector performance to exceed that of the G4e.

Floating-Point Issue Queues

Figure 10-1 shows the five dispatch slots connected to issue queues for each of the functional units or groups of functional units. Figure 10-1 is actually oversimplified, since the true relationship between the dispatch slots, issue queues, and functional units is more complicated than depicted there. The actual physical issue queue configuration is a bit hard to explain in words, so Figure 10-4 shows how the floating-point issue queues really look.

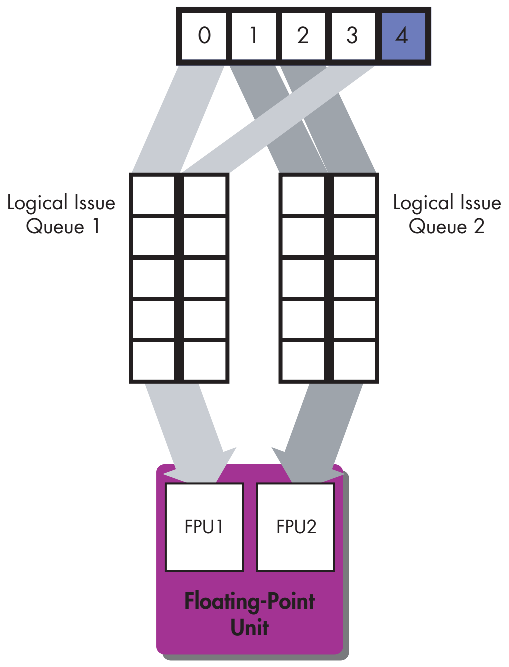


Figure 10-4: The 970's floating-point issue queues

Each of the floating-point execution units is fed by what I've called a logical issue queue. As you can see in Figure 10-4, each 10-entry logical issue queue actually consists of an interleaved pair of five-entry physical issue queues, which together feed a single floating-point execution unit.

Figure 10-4 also shows how the four non-branch dispatch slots each feed a specific physical issue queue. Floating-point IOPs that dispatch from slots 0 and 3 go into the tops of the two physical issue queues that are attached to them. Similarly, floating-point IOPs that dispatch from slots 1 and 2 go into the tops of their two attached physical issue queues. As IOPs issue from different places in the physical queues, the IOPs "above" them in the queue "fall down" to fill in the gaps.

Each pair of physical issue queues—the pair attached to slots 0 and 3, and the pair attached to slots 1 and 2—is interleaved and functions as a single "logical" issue queue. An IOP can issue from a logical issue queue to an execution unit when all of its sources are set, and the oldest IOP with all of its sources set in a logical queue is issued to the attached execution unit. This means that, as is the case with the G4e's issue queues, instructions are issued *in* program order from within each logical issue queue, but *out of* program order with respect to the overall code stream.

NOTE *The term* logical issue queue *is one that I've coined for the purposes of this chapter, and is not to my knowledge used in IBM's literature. IBM prefers the phrase* common interleaved queues.

It's important for me to emphasize that individual IOPs issue from their respective logical issue queues *completely independent of their dispatch group*. So the execution units' schedulers are blind to which group an IOP is in when it comes time to schedule the IOP and issue it to an execution unit. Thus the dispatch groups are allowed to break apart after they reach the level of the issue queue, and they're reassembled after execution in the group completion table (GCT).

Integer and Load-Store Issue Queues

The integer and load-store execution units are fed by issue queues in a similar manner to the floating-point units, but they're slightly more complex because they share issue queues. Take a look at Figure 10-5 to see how this works.

As with the floating-point issue queues, integer or memory access IOPs in dispatch slots 0 and 3 go into the two integer physical issue queues that are specifically intended for them. The twist is that this pair of queues is shared by two execution units: IU1 and LSU1. So integer or memory IOPs from slots 0 and 3 are sent into the appropriate issue queue pair—or logical issue queue—and these IOPs then issue to either IU1 or LSU1 as the integer scheduler sees fit. Similarly, integer or memory access IOPs from dispatch slots 1 and 2 go into their respective physical queues, both of which feed IU2 and LSU2. As with the floating-point issue queues, these four 9-entry interleaved queues should be thought of as being grouped into two 18-entry logical issue queues, where each logical issue queue works together with the appropriate scheduler to feed a pair of execution units.

BU and CRU Issue Queues

The branch unit and condition register unit issue queues work in a similar manner to what I've described previously, with the differences depending on grouping and issue restrictions. So the CRU has a single 10-entry logical issue queue comprised of two issue queues with five entries each, one for slot 0

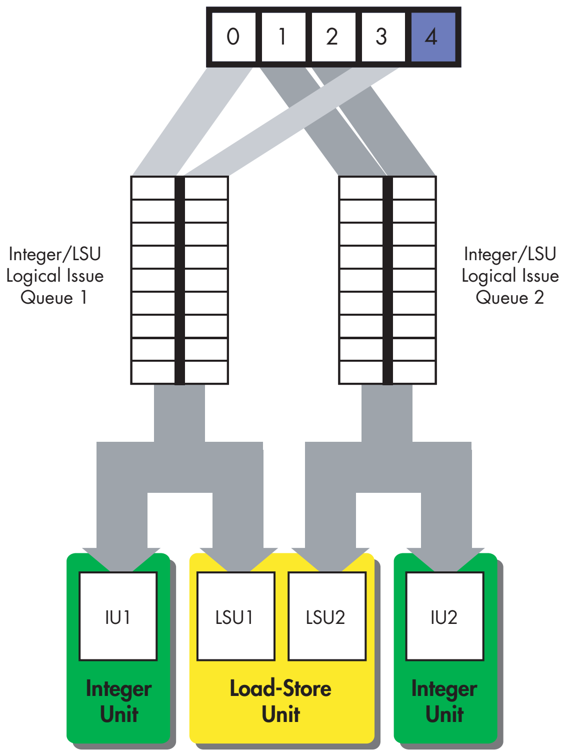

0
1
2
3
4
Integer/LSU Logical Issue Queue 1
Integer/LSU Logical Issue Queue 2
IU1
LSU1
LSU2
IU2
Integer Unit
Load-Store Unit
Integer Unit

Figure 10-5: The 970's IU and LSU issue queues

and one for slot 1 (because CR IOPs can be placed only in slots 0 and 1). The branch execution unit has a single logical issue queue comprised of one 12-entry issue queue for slot 4 (because branch IOPs can go only in slot 4).

Vector Issue Queues

The vector issue queues are laid out slightly different than the other issue queues. The vector issue queue configuration is depicted in Figure 10-6.

The vector permute unit is fed from one 16-entry (four entries × four queues) logical issue queue connected to all four non-branch dispatch slots, and the vector ALU is fed from a 20-entry (five entries × four queues) logical issue queue that's also connected to all four non-branch dispatch slots. As with all of the other issue queue pairs, one IOP per cycle can issue in order from each logical issue queue to any of the execution units that are attached to it.

The Performance Implications of the 970's Group Dispatch Scheme

Because of the way it affects the 970's issue queue design, the group formation scheme has some interesting performance implications. Specifically, proper code scheduling is important in ways that it wouldn't normally be for the other processors discussed here.

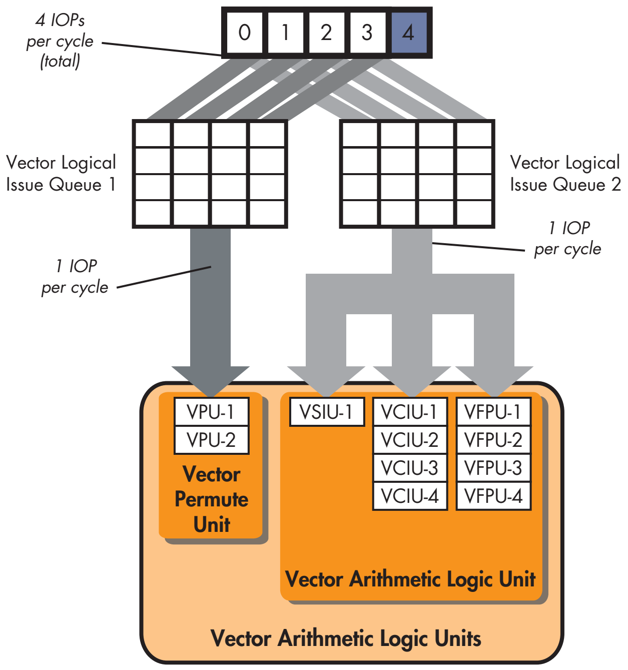


Figure 10-6: The 970's vector issue queues

Instead of trying to explain this point, I'll illustrate it with an example. Let's look at an instruction with few group restrictions—the floating-point add. The 970's group formation rules dictate that the `fadd` can go into any of the four dispatch slots, and which slot it goes into in turn dictates which of the 970's two identical FPUs executes it. As I explained in the previous section, if the `fadd` goes into slots 0 or 3, it is dispatched to the logical issue queue associated with FPU1; if it goes into dispatch slot 1 or 2, it is dispatched to the logical issue queue associated with FPU2. This means that the FPU instruction scheduling hardware is restricted in ways that it wouldn't be if both FPUs were fed from a common issue queue, because half the instructions are forced into one FPU and half the instructions are forced into the other FPU. Or at least this 50/50 split is how it's supposed to work out under optimal circumstances, when the code is scheduled properly so that it dispatches IOPs evenly into both logical issue queues.

Because of the grouping scheme and the two separate logical issue queues, it seems that keeping both FPUs busy by splitting the computation load between them is very much a matter of scheduling instructions for dispatch so that no single FPU happens to get overworked while the other goes underutilized. Normally, this kind of load balancing among execution units would happen at the issue queue level, but in the 970's case, it's constrained by the structure of the issue queues themselves.

This load balancing at the dispatch level isn't quite as simple as it may sound, because group formation takes place according to a specific set of rules that ultimately constrain dispatch bandwidth and subsequent instruction issue in very specific and peculiar ways. For instance, an integer instruction that's preceded by, say, a CR logical instruction, may have to move over a slot to make room because the CR logical instruction can go only in slots 0 and 1. Likewise, depending on whether an instruction near the integer IOP in the instruction stream is cracked or millicoded, the integer IOP may have to move over a certain number of slots; if the millicoded instruction breaks down into a long string of instructions, that integer IOP may even get bumped over into a later dispatch group. The overall result is that which queue an integer IOP goes into very much depends on the other (possibly non-integer) instructions that surround it.

The take-home message here is that PowerPC code that's optimized specifically for the 970 performs significantly better on the processor than legacy code that's optimized for other PowerPC processors like the G4e.

Of course, no one should get the impression that legacy code runs poorly on the 970. It's just that the full potential of the chip can't be unlocked without properly scheduled code. Furthermore, in addition to the mitigating factors mentioned in the section on integer performance (for example, deep OOOE capabilities, or high-bandwidth FSB), the fact that quantitative studies have shown the amount of ILP inherent in most RISC code to be around two instructions per clock means that the degenerate case described in the FPU example should be exceedingly rare.

Conclusions

While the 970's group dispatch scheme does suffer from some of the drawbacks described in the preceding section, it must be judged a success in terms of its impact on the processor's performance per watt. That this dispatch scheme has a significant positive impact on performance per watt is evidenced by the fact that Intel's Pentium M processor also uses a similar grouping mechanism to achieve greater power efficiency. Furthermore, Intel continues to employ this grouping mechanism more extensively with each successive revision of the Pentium M, as the company seeks to minimize power consumption without sacrificing number-crunching capabilities. Thus such grouping mechanisms will only become more widespread as microprocessor designers become ever more sensitive to the need to balance performance and power consumption.

Because the 970 can track more instructions with less power-hungry bookkeeping logic, it can spend more transistors on execution units, branch prediction resources, and cache. This last item—cache—is an especially important performance-enhancing element in modern processors, for reasons that will be covered in Chapter 11.

11

UNDERSTANDING CACHING AND PERFORMANCE

This chapter is intended as a general introduction to CPU caching and performance. Because cache is critical to keeping the processors described so far fed with code and data, you can't understand how computer systems function without first understanding the structure and functioning of the cache memory hierarchy. To that end, this chapter covers fundamental cache concepts like spatial and temporal locality, set associativity, how different types of applications use the cache, the general layout and function of the memory hierarchy, and other cache-related issues.

Caching Basics

In order to really understand the role of caching in system design, think of the CPU and memory subsystem as operating on a *consumer-producer model* (or *client-server model*): The CPU consumes information provided to it by the hard disks and RAM, which act as producers.

Driven by innovations in process technology and processor design, CPUs have increased their ability to consume at a significantly higher rate than the memory subsystem has increased its ability to produce. The problem is that CPU clock cycles have gotten shorter at a faster rate than memory and bus clock cycles, so the number of CPU clock cycles that the processor has to wait before main memory can fulfill its requests for data has increased. With each CPU clockspeed increase, memory is getting farther and farther away from the CPU in terms of the number of CPU clock cycles.

Figures 11-1 and 11-2 illustrate how CPU clock cycles have gotten shorter relative to memory clock cycles.

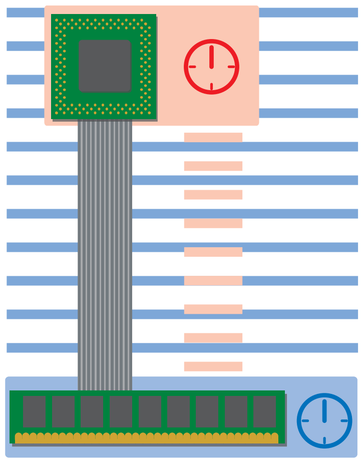


Figure 11-1: Slower CPU clock

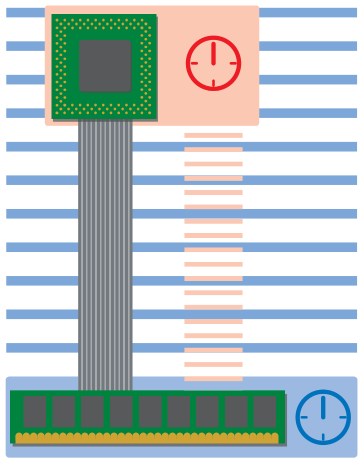


Figure 11-2: Faster CPU clock

To visualize the effect that this widening speed gap has on overall system performance, imagine the CPU as a downtown furniture maker's workshop and the main memory as a lumberyard that keeps getting moved farther and farther out into the suburbs. Even if you start using bigger trucks to cart all the wood, it's still going to take longer from the time the workshop places an order to the time that order gets filled.

NOTE *I'm not the first person to use a workshop and warehouse analogy to explain caching. The most famous example of such an analogy is the Thing King game, which is widely available on the Internet.*

Sticking with the furniture workshop analogy, one solution to this problem would be to rent out a small warehouse in town and store the most commonly requested types of lumber there. This smaller, closer warehouse would act as a *cache* that sits between the lumberyard and the workshop, and you could keep a driver on hand at the workshop who could run out at a moment's notice and quickly pick up whatever you need from the warehouse.

Of course, the bigger your warehouse, the better, because it allows you to store more types of wood, thereby increasing the likelihood that the raw materials for any particular order will be on hand when you need them. In the event that you need a type of wood that isn't in the nearby warehouse, you'll have to drive all the way out of town to get it from your big, suburban lumberyard. This is bad news, because unless your furniture workers have another task to work on while they're waiting for your driver to return with the lumber, they're going to sit around in the break room smoking and watching *The Oprah Winfrey Show*. And you hate paying people to watch *The Oprah Winfrey Show*.

The Level 1 Cache

I'm sure you've figured it out already, but the smaller, closer warehouse in this analogy is the *level 1 cache* (*L1 cache* or *L1*, for short). The L1 can be accessed very quickly by the CPU, so it's a good place to keep the code and data that the CPU is most likely to request. (In a moment, we'll talk in more detail about how the L1 can "predict" what the CPU will probably want.) The L1's quick access time is a result of the fact that it's made of the fastest and most expensive type of *static RAM*, or *SRAM*. Since each SRAM memory cell is made up of four to six transistors (compared to the one-transistor-per-cell configuration of DRAM), its cost per bit is quite high. This high cost per bit means that you generally can't afford to have a very large L1 unless you really want to drive up the total cost of the system.

In modern CPUs, the L1 sits on the same piece of silicon as the rest of the processor. In terms of the warehouse analogy, this is a bit like having the warehouse on the same block as the workshop. This has the advantage of giving the CPU some very fast, very close storage, but the disadvantage is that now the main memory (the suburban lumberyard) is just as far away from the L1 as it is from the processor. If data that the CPU needs is not in the L1—a situation called a *cache miss*—it's going to take quite a while to retrieve that data from memory. Furthermore, remember that as the processor gets faster, the main memory gets "farther" away all the time. So while your warehouse may be on the same block as your workshop, the lumberyard has now moved not just out of town but out of the state. For an ultra–high-clock-rate processor like the P4, being forced to wait for data to load from main memory in order to complete an operation is like your workshop having to wait a few days for lumber to ship in from out of state.

Check out Table 11-1, which shows common latency and size information for the various levels of the memory hierarchy. (The numbers in this table are shrinking all the time, so if they look a bit large to you, that's probably because by the time you read this, they're dated.)

Table 11-1: A Comparison of Different Types of Data Storage

Level	Access Time	Typical Size	Technology	Managed By
Registers	1–3 ns	1KB	Custom CMOS	Compiler
Level 1 Cache (on-chip)	2–8 ns	8KB–128KB	SRAM	Hardware
Level 2 Cache (off-chip)	5–12 ns	0.5MB–8MB	SRAM	Hardware
Main Memory	10–60 ns	64MB–1GB	DRAM	Operating system
Hard Disk	3,000,000–10,000,000 ns	20GB–100GB	Magnetic	Operating system/user

Notice the large gap in access times between the L1 and the main memory. For a 1 GHz CPU, a 50 ns wait means 50 wasted clock cycles. Ouch! To see the kind of effect such stalls have on a modern, hyperpipelined processor, see "Instruction Throughput and Pipeline Stalls" on page 53.

The Level 2 Cache

The solution to this dilemma is to add more cache. At first you might think you could get more cache by enlarging the L1, but as I said earlier, cost considerations are a major factor limiting L1 cache size. In terms of the workshop analogy, you could say that rents are much higher in town than in the suburbs, so you can't afford much in-town warehouse space without the cost of rent eating into your bottom line, to the point where the added costs of the warehouse space would outweigh the benefits of increased worker productivity. You have to fine-tune the amount of warehouse space that you rent by weighing all the costs and benefits so that you get the maximum output for the least cost.

A better solution than adding more in-town warehouse space would be to rent some cheaper, larger warehouse space right outside of town to act as a cache for the in-town warehouse. Similarly, processors like the P4 and G4e have a *level 2 cache* (*L2 cache* or *L2*) that sits between the L1 and main memory. The L2 usually contains all of the data that's in the L1 plus some extra. The common way to describe this situation is to say that the L1 *subsets* the L2, because the L1 contains a subset of the data in the L2.

A series of caches, starting with the page file on the hard disk (the lumberyard) and going all the way up to the registers on the CPU (the workshop's work benches), is called a *cache hierarchy*. As you go up the cache hierarchy towards the CPU, the caches get smaller, faster, and more expensive to implement; conversely, as you go down the cache hierarchy, the caches get larger, cheaper, and much slower. The data contained in each level of the hierarchy is usually mirrored in the level below it, so for a piece of data that's in the L1, there are usually copies of that same data in the L2, main memory, and hard disk's page file. Each level in the hierarchy subsets the level below it. We'll talk later about how all of those copies are kept in sync.

In Figure 11-3, the red cells are the code and data for the program that the CPU is currently running. The blue cells are unrelated to the currently running program. This figure, which will become even clearer once you read "Locality of Reference" on page 220, shows how each level of the cache hierarchy subsets the level below it.

As Table 11-1 indicates, each level of the hierarchy depicted in Figure 11-3 is controlled by a different part of the system. Data is promoted up the hierarchy or demoted down the hierarchy based on a number of different criteria; in the remainder of this chapter we'll concern ourselves only with the top levels of the hierarchy.

Example: A Byte's Brief Journey Through the Memory Hierarchy

For the sake of example, let's say the CPU issues a `load` instruction that tells the memory subsystem to load a piece of data (in this case, a single byte) into one of its registers. First, the request goes out to the L1, which is

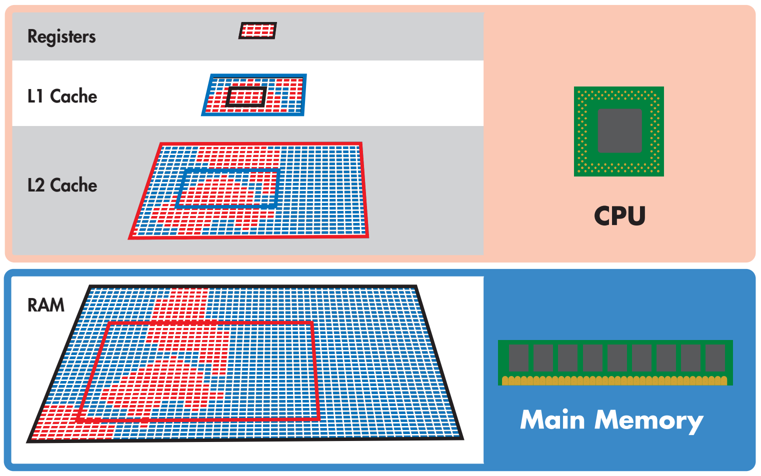


Figure 11-3: Code and data in the cache hierarchy

checked to see if it contains the requested data. If the L1 does not contain the data and therefore cannot fulfill the request—in other words, a cache miss occurs—the request propagates down to the L2. If the L2 does not contain the desired byte, the request begins the relatively long trip out to main memory. If main memory doesn't contain the data, you're in big trouble, because then it has to be *paged* in from the hard disk, an act that can take a relative eternity in CPU time.

Let's assume that the requested byte is found in main memory. Once located, the byte is copied from main memory, along with a bunch of its neighboring bytes in the form of a *cache block* or *cache line*, into the L2 and L1. When the CPU requests this same byte again, it will be waiting for it there in the L1, a situation called a *cache hit*.

Cache Misses

Computer architects usually divide cache misses up into three different types depending on the situation that brought about the miss. I'll introduce these three types of misses at appropriate points over the course of the chapter, but I can talk about the first one right now.

A *compulsory miss* is a cache miss that occurs because the desired data was never in the cache and therefore must be paged in for the first time in a program's execution. It's called a *compulsory* miss because, barring the use of certain specialized tricks like data prefetching, it's the one type of miss that just can't be avoided. All cached data must be brought into the cache for the very first time at some point, and the occasion for doing so is normally a compulsory miss.

The two other types of cache misses are misses that result when the CPU requests data that was previously in the cache but has been *evicted* for some reason or other. We'll discuss evictions in "Temporal and Spatial Locality Revisited: Replacement/Eviction Policies and Block Sizes" on page 230, and I'll cover the other two types of cache misses as they come up throughout the course of this chapter.

Locality of Reference

Caching works because of a very simple property exhibited to one degree or another by all types of code and data: *locality of reference*. We generally find it useful to talk about two types of locality of reference: *spatial locality* and *temporal locality*.

Spatial locality

Spatial locality is a fancy label for the general rule that *if the CPU needs an item from memory at any given moment, it's likely to need that item's neighbors next.*

Temporal locality

Temporal locality is the name we give to the general rule that *if an item in memory was accessed once, it's likely to be accessed again in the near future.*

Depending on the type of application, both code and data streams can exhibit spatial and temporal locality.

Spatial Locality of Data

Spatial locality of data is the easiest type of locality to understand, because most of you have used media applications like MP3 players, DVD players, and other types of applications whose datasets consist of large, ordered files. Consider an MP3 file, which has a bunch of blocks of data that are consumed by the processor in sequence from the file's beginning to its end. If the CPU is running iTunes and it has just requested second 1:23 of a five-minute MP3, you can be reasonably certain that next it's going to want seconds 1:24, 1:25, and so on. This is the same with a DVD file, and with many other types of media files like images, AutoCAD drawings, and 3D game levels. All of these applications operate on large arrays of sequentially ordered data that get ground through in sequence by the CPU again and again.

Business applications like word processors also have great spatial locality for data. If you think about it, few people open six or seven documents in a word processor and quickly alternate between them typing one or two words in each. Most of us just open up one or two relatively modest-sized files and work in them for a while without skipping around too much within the same file. These files are stored in contiguous regions of memory, where they can be brought quickly into the cache in a few large batches.

Ultimately, spatial locality is just a way of saying that related chunks of data tend to clump together in memory, and since they're related, they also tend to be processed together in batches by the CPU.

In Figure 11-4, as in Figure 11-3, the red cells are related chunks of data stored in the memory array. This figure shows a program with fairly good spatial locality, since the red cells are clumped closely together. In an application with poor spatial locality, the red cells would be more randomly distributed among the unrelated blue cells.

Spatial Locality of Code

Spatial locality applies to code just like it does to data—most well-written code tries to avoid jumps and branches so that the processor can execute through large contiguous blocks uninterrupted. Games, simulations, and media processing applications tend to have decent spatial locality for code, because such applications often feature small blocks of code (called *kernels*) operating repeatedly on very large datasets.

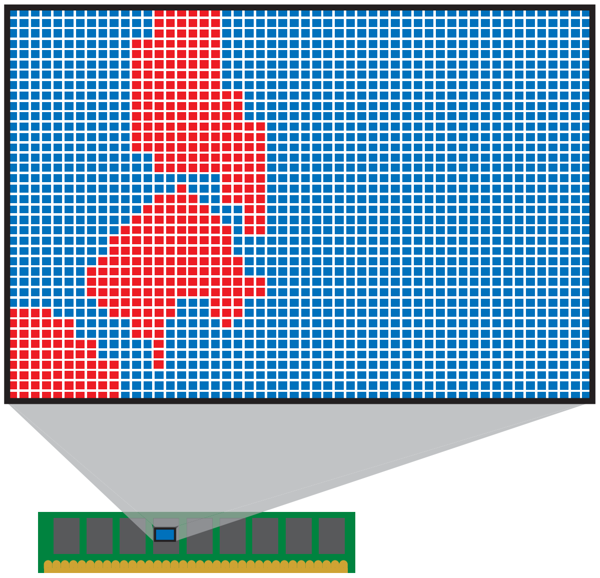


Figure 11-4: Spatial locality

When it comes to spatial locality of code for business applications, the picture is mixed. If you think about the way that you use a word processor, it's easy to see what's going on. As you create a document, most of you are constantly pressing different formatting buttons and invoking different menu options. For instance, you might format one word in italics, change the paragraph spacing, and then save the file, all in sequence. Each of these actions invokes a very different part of the code in a large application like Microsoft Word; it's not likely that the code for the File ▸ Save menu option is stored right next to the code that formats a font in italics. The way you use a word processor forces the CPU to jump around from place to place in memory in order to retrieve the correct code. However, the segment of the code stream that implements each individual action (i.e., saving a file, formatting a font, and so on) usually exists in memory as a fairly large, spatially localized chunk—very much like a little subprogram within the larger application. While the code for the File ▸ Save menu action might be quite far away from the code for the italics formatting option, both of these chunks of code exhibit good spatial locality as small programs in their own right.

What this means for a cache designer is that business applications need large instruction caches to be able to collect all of the most frequently used clumps of code that correspond to the most frequently executed actions and pack them together in the cache. If the instruction cache is too small, the different clumps get *swapped* out as different actions are performed. If the instruction cache is large enough, many of these subprograms will fit and there's little swapping needed. Incidentally, this is why business applications performed so poorly on Intel's original cacheless Celeron processor.

Temporal Locality of Code and Data

Consider a simple Photoshop filter that inverts an image to produce a negative; there's a small piece of code that performs the same inversion on each pixel, starting at one corner and going in sequence all the way across and down to the opposite corner. This code is just a small loop that gets executed repeatedly, once on each pixel, so it's an example of code that is reused again and again. Media applications, games, and simulations, because they use lots of small loops that iterate through very large datasets, have excellent temporal locality for code.

The same large, homogenous datasets that give media applications and the like good temporal locality for code also given them extremely poor temporal locality for data. Returning to the MP3 example, a music file is usually played through once in sequence and none of its parts are repeated. This being the case, it's actually a waste to store any of that file in the data cache, since it's only going to stop off there temporarily before passing through to the CPU.

When an application, like the aforementioned MP3 player, fills up the cache with data that doesn't really need to be cached because it won't be used again and as a result winds up bumping out of the cache data that will be reused, that application is said to "pollute" the cache. Media applications, games, and the like are big cache polluters, which is why they weren't too affected by the original Celeron's lack of cache. Because these applications' data wasn't going to be needed again any time soon, the fact that it wasn't in a readily accessible cache didn't really matter.

The primary way in which caches take advantage of temporal locality is probably obvious by this point: Caches provide a place to store code and data that the CPU is currently working with. By *working with*, I mean that the CPU has used it once and is likely to use it again. A group of related blocks of code and/or data that the CPU uses and reuses over a period of time in order to complete a task is called a *working set*. Depending on the size of a task's working set and the number of operations it takes the CPU to complete that task, spatial and temporal locality—and with them the cache hierarchy—will afford a greater or lesser performance increase on that task.

Locality: Conclusions

One point that should be apparent from the preceding discussion is that temporal locality implies spatial locality, but not vice versa. That is to say, data that is reused is always related (and therefore collects into spatially localized clumps in memory), but data that is related is not always reused. An open text file is an example of reusable data that occupies a localized region of memory, and an MP3 file is an example of non-reusable (or streaming) data that also occupies a localized region of memory.

The other thing that you probably noticed from this section is the fact that memory access patterns for code and memory access patterns for data are often very different within the same application. For instance, media applications have excellent temporal locality for code but poor temporal locality for data. This fact has inspired many cache designers to split the L1 into two regions—one for code and one for data. The code half of the cache is called the *instruction cache*, or *I-cache*, while the data half of the cache is called the *data cache*, or *D-cache*. This partitioning can result in significant performance gains, depending on the size of the cache, the types of applications normally run on the system, and a variety of other factors.

Cache Organization: Blocks and Block Frames

One way that caches take advantage of locality of reference is by loading data from memory in large chunks. When the CPU requests a particular piece of data from the memory subsystem, that piece gets fetched and loaded into the L1 along with some of its nearest neighbors. The actual piece of data that was requested is called the *critical word*, and the surrounding group of bytes that gets fetched along with it is called a cache line or cache block. By fetching not only the critical word but also a group of its neighbors and loading them into the cache, the CPU is prepared to take advantage of the fact that those neighboring bytes are the most likely to be the ones it will need to process next.

Cache blocks form the basic unit of cache organization, and RAM is also organized into blocks of the same size as the cache's blocks. When a block is moved from RAM to the cache, it is placed into a special slot in the cache called a *block frame*. Figure 11-5 shows a set of cache blocks stored in RAM and a cache with an empty set of block frames.

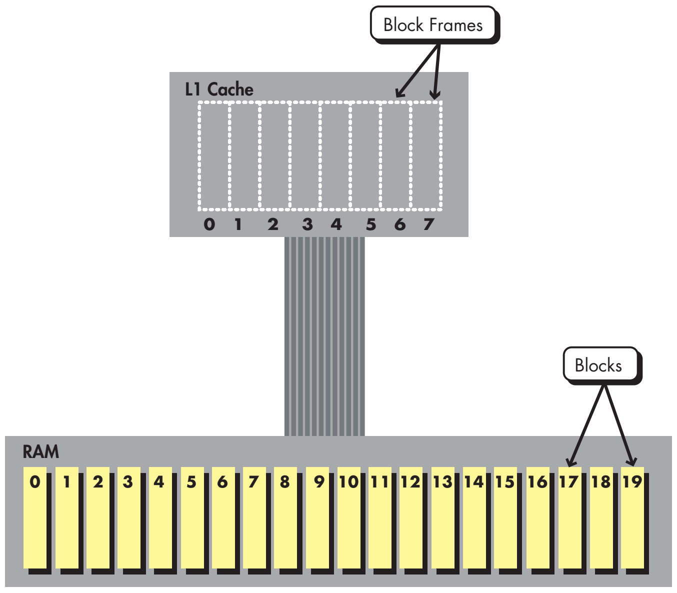


Figure 11-5: Blocks and block frames

Cache designers can choose from a few different schemes for governing which RAM blocks can be stored in which of the cache's block frames. Such a scheme is called a *cache placement policy*, because it dictates where in the cache a block from memory can be placed.

Tag RAM

When the CPU requests a byte from a particular block from RAM, it needs to be able to determine three things very quickly:

- whether or not the needed block is actually in the cache (i.e., whether there is a cache hit or a cache miss)
- the location of the block within the cache (in the case of a cache hit)
- the location of the desired byte (or critical word) within the block (again, in the case of a cache hit)

A cache accommodates all three needs by associating a special piece of memory—called a tag—with each block frame in the cache. The *tag* holds information about the blocks currently being stored in the frame, and that information allows the CPU to determine the answer to all three of the questions above. However, the speed with which that answer comes depends on a variety of factors.

Tags are stored in a special type of memory called the *tag RAM*. This memory has to be made of very fast SRAM because it can take some time to search it in order to locate the desired cache block. The larger the cache, the greater the number of blocks, and the greater the number of blocks, the more tag RAM you need to search and the longer it can take to locate the correct block. Thus the tag RAM can add unwanted access latency to the cache. As a result, you not only have to use the fastest RAM available for the tag RAM, but you also have to be smart about how you use tags to map RAM blocks to block frames. In the following section, I'll introduce the three general options for doing such mapping, and I'll discuss some of the pros and cons of each option.

Fully Associative Mapping

The most conceptually simple scheme for mapping RAM blocks to cache block frames is called *fully associative mapping*. Under this scheme, any RAM block can be stored in any available block frame. Fully associative mapping is depicted in Figure 11-6, where any of the red RAM blocks can go into any of the red cache block frames.

The problem with fully associative mapping is that if you want to retrieve a specific block from the cache, you have to check the tag of every single block frame in the entire cache because the desired block could be in any of the frames. Since large caches can have thousands of block frames, this tag searching can add considerable delay (latency) to a fetch. Furthermore, the larger the cache, the worse the delay gets, since there are more block frames and hence more block tags to check on each fetch.

Direct Mapping

Another, more popular way of organizing the cache is to use *direct mapping*. In a *direct-mapped cache*, each block frame can cache only a certain subset of the blocks in main memory.

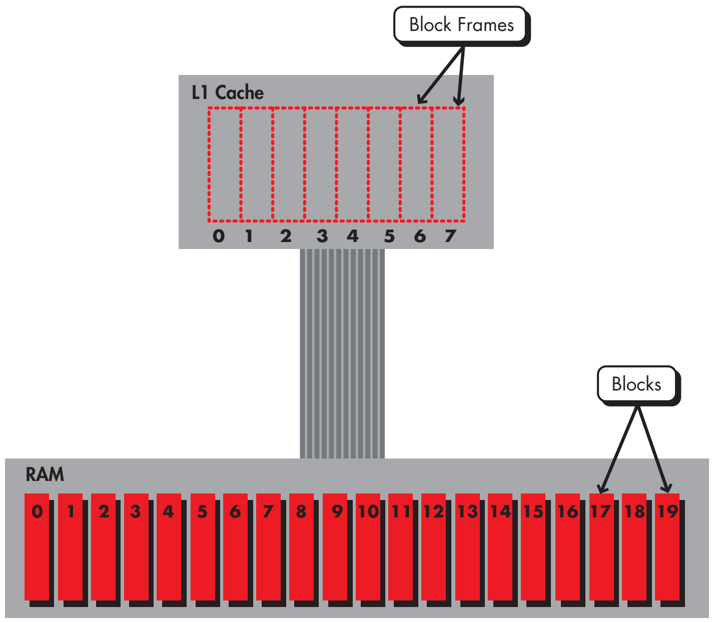


Figure 11-6: Fully associative mapping

In Figure 11-7, each of the red blocks (blocks 0, 8, and 16) can be cached only in the red block frame (frame 0). Likewise, blocks 1, 9, and 17 can be cached only in frame 1, blocks 2, 10, and 18 can be cached only in frame 2, and so on. Hopefully, the pattern here is apparent: Each frame caches every eighth block of main memory. As a result, the potential number of locations for any one block is greatly narrowed, and therefore the number of tags that must be checked on each fetch is greatly reduced. For example, if the CPU needs a byte from blocks 0, 8, or 16, it knows that it only has to check the tag associated with frame 0 to determine if the desired block is in the cache and to retrieve it if it is. This is much faster and more efficient than checking every frame in the cache.

There are some drawbacks to this scheme, though. For instance, what if blocks 0 to 3 and 8 to 11 combine to form an eight-block working set that the CPU wants to load into the cache and work on for a while? The cache is eight blocks long, but since it's direct-mapped, it can only store four of these particular blocks at a time. Remember, blocks 0 and 8 have to go in the same frame, as do blocks 1 and 9, 2 and 10, and 3 and 11. As a result, the CPU must load only four blocks of this eight-block set at a time, and swap them in and out as it works on different parts of the set. If the CPU wants to work on this whole eight-block set for a long time, that could mean a lot of swapping. Meanwhile, half of the cache is going completely unused! While direct-mapped caches are almost always faster than fully associative caches due to the shortened amount of time it takes to locate a cached block, they can still be inefficient under some circumstances.

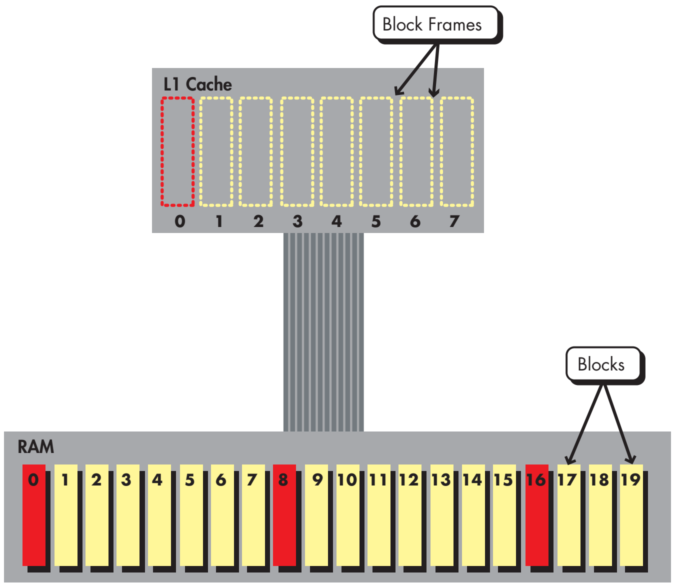


Figure 11-7: Direct mapping

Note that the kind of situation described here, where the CPU would like to store multiple blocks but it can't because they all require the same frame, is called a *collision*. In the preceding example, blocks 0 and 8 are said to collide, since they both want to fit into frame 0 but can't. Misses that result from such collisions are called *conflict misses*, the second of the three types of cache miss that I mentioned earlier.

N-Way Set Associative Mapping

One way to get some of the benefits of direct-mapped caches while lessening the amount of cache space wasted due to collisions is to restrict the caching of main memory blocks to a subset of the available cache frames. This technique is called *set associative mapping*, and a few popular implementations of it are described below.

Four-Way Set Associative Mapping

To see an example of what it means to restrict main memory blocks in a subset of available cache frames, take a look at the Figure 11-8, which illustrates *four-way set associative mapping*.

In Figure 11-8, any of the red blocks can go anywhere in the red set of frames (set 0) and any of the light yellow blocks can go anywhere in the light yellow set of frames (set 1). Think of the four-way associative cache like this: You took a fully associative cache and cut it in two, restricting half the main memory blocks to one side and half the main memory blocks to the other.

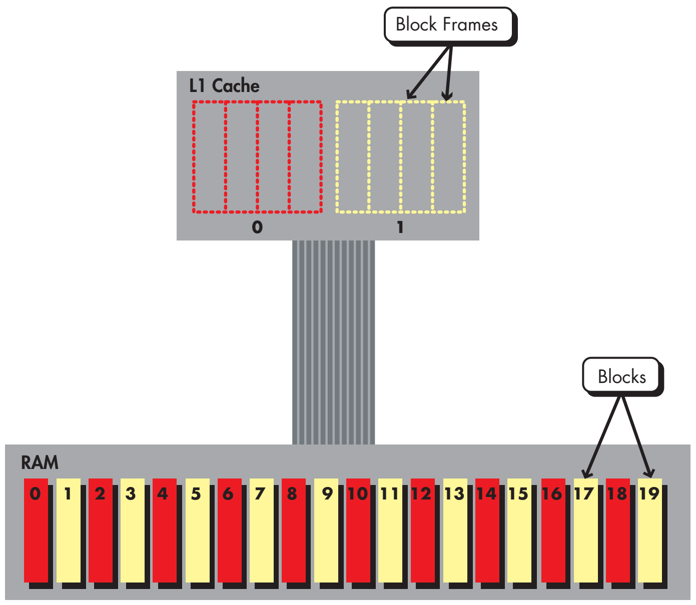


Figure 11-8: Four-way set associative mapping

This way, the odds of a collision are greatly reduced versus the direct-mapped cache, but you still don't have to search all the tags on every fetch like you did with the fully associative cache. For any given fetch, you need search only a single, four-block set to find the block you're looking for, which in this case amounts to half the cache.

The cache pictured in Figure 11-8 is said to be four-way *set associative* because the cache is divided into *sets* of four frames each. Since this cache has only eight frames, it can accommodate only two four-frame sets. A larger cache could accommodate more four-frame sets, reducing the odds of a collision even more.

Figure 11-9 shows a four-way set associative cache like the one in Figure 11-8, but with three sets instead of two. Notice that there are fewer red main memory blocks competing for space in set 0, which means lower odds of a collision.

In addition to its decreased odds of a collision, a four-way set associative cache has a low access latency because the number of frames that must be searched for a block is limited. Since all the sets consist of exactly four frames, no matter how large the cache gets, you'll only ever have to search through four frames (or one full set) to find any given block. This means that as the cache gets larger and the number of sets that it can accommodate increases, the tag searches become more efficient. Think about it. In a cache with three sets, only one-third of the cache (or one set) needs to be searched for a given block. In a cache with four sets, only one-fourth of the cache is searched. In a cache with one hundred four-block sets, only one-hundredth of the cache needs to be searched. The relative search efficiency scales with the cache size.

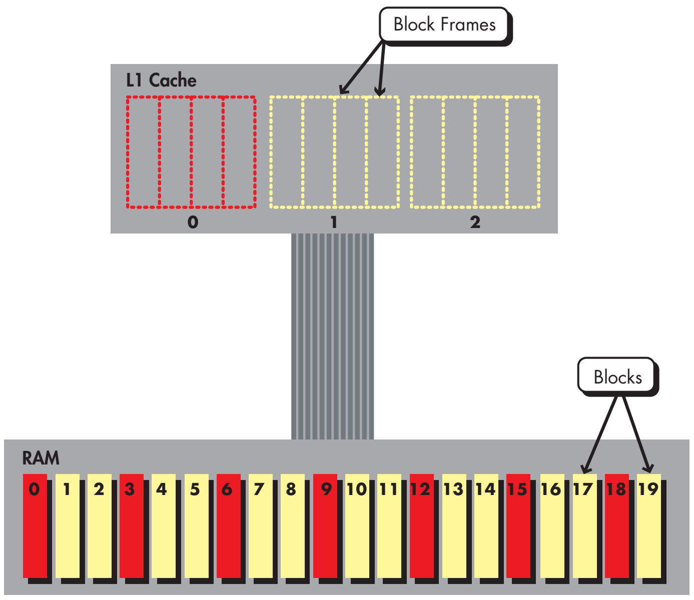


Figure 11-9: Four-way set associative mapping with three block frames

Two-Way Set Associative Mapping

Another way to increase the number of sets in the cache without actually increasing the cache size is to reduce the number of blocks in each set. Check out Figure 11-10, which shows a two-way set associative cache.

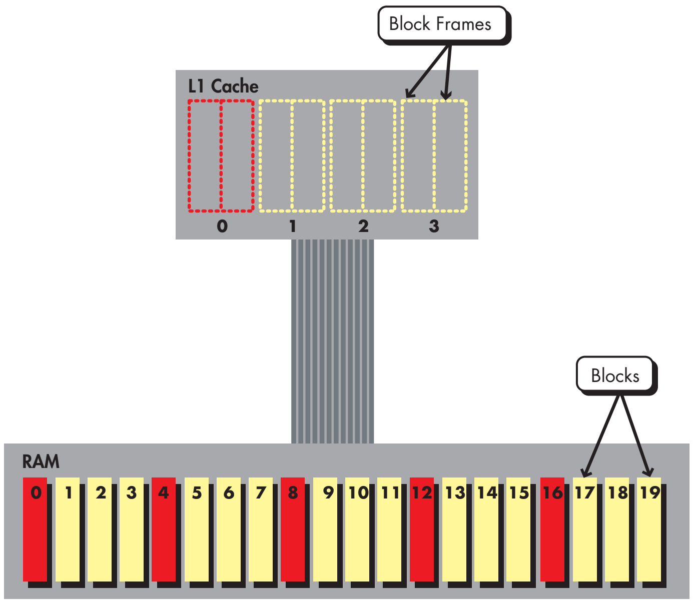


Figure 11-10: Two-way set associative mapping

To get a better idea of what's going on, let's compare the two-way associative cache to both the direct-mapped and the four-way cache. For the sake of comparison, assume that the cache size and the memory size both stay constant. And as you read the comparison, keep in mind that since each increase in the level of associativity (i.e., from two-way to four-way, or from four-way to eight-way) also increases the number of tags that must be checked in order to locate a specific block, an increase in associativity also means an increase in the cache's latency.

Two-Way vs. Direct-Mapped

With the two-way cache, the number of potential collisions (and hence the miss rate) is reduced compared to the direct-mapped scheme. However, the number of tags that must be searched for each fetch is twice as high. Depending on the relative sizes of the cache and main memory, this may or may not increase the cache's overall latency to the point that the decreased conflict miss rate is worth it.

Two-Way vs. Four-Way

Though a two-way cache's latency is less than that of a four-way cache, its number of potential collisions (and hence its miss rate) is higher. Just as with the preceding comparison, how a two-way associative cache compares to a four-way associative cache depends on just how much latency the four-way cache's increase in associativity ends up costing versus the decrease in conflict miss rate.

Associativity: Conclusions

In general, it turns out that when you factor in current technological conditions (the speed of tag RAM, the range of sizes that most caches fall into, and so on), some level of associativity less than or equal to eight-way turns out to be optimal for most caches. Any more than eight-way associativity and the cache's latency is increased so much that the decrease in miss rate isn't worth it. Any less than two-way associativity and the number of collisions often increases the miss rate to the point that the decrease in latency isn't worth it. There are some direct-mapped caches out there, though.

Before I conclude the discussion of associativity, there are two minor bits of information that I should include for the sake of completeness. First, though you've probably already figured it out, a direct-mapped cache is simply a one-way set associative cache, and a fully associative cache is an *n*-way set associative cache, where *n* is equal to the total number of blocks in the cache.

Second, the cache placement formula, which enables you to compute the set in which an arbitrary block of memory should be stored in an *n*-way associative cache, is as follows:

```
(block_address) MOD (number_of_sets_in_cache)
```

I recommend trying out this formula on some of the preceding examples. It might seem like a boring and pointless exercise, but if you take five minutes and place a few blocks using this simple formula in conjunction with the preceding diagrams, everything I've said in this section will really fall into place for you in a "big picture" kind of way. And it's actually more fun to do that it probably sounds.

Temporal and Spatial Locality Revisited: Replacement/Eviction Policies and Block Sizes

Caches can increase the amount of benefit they derive from temporal locality by implementing an intelligent *replacement policy* (also called, conversely, an *eviction policy*). A replacement policy dictates which of the blocks currently in the cache will be replaced by any new block that gets fetched in. (Or, another way of putting it is that an eviction policy dictates which of the blocks currently in the cache will be evicted in order to make room for any new blocks that are fetched in.)

Types of Replacement/Eviction Policies

One way to implement a replacement policy would be to pick a block at random to be replaced. Other possible replacement policies would be a FIFO policy, a LIFO policy, or some other such variation. However, none of these policies take into account the fact that any block that was recently used is likely to be used again soon. With these simple policies, you wind up evicting blocks that will be used again shortly, thereby increasing the cache miss rate and eating up valuable memory bus bandwidth with a bunch of unnecessary fetches.

The ideal replacement policy would be one that makes room for an incoming block by evicting the cache block that is destined to go unused for the longest period of time. Unfortunately, computer architects haven't yet devised a way of infallibly predicting the future, so there's no way to know for sure which of the blocks that are currently residing in the cache is the one that will go the longest time before being accessed again.

Even if you can't predict the future, you can make an educated guess based on what you know of the cache's past behavior. The optimal replacement policy that doesn't involve predicting the future is to evict the block that has gone the longest period of time without being used, or the *least recently used (LRU) block*. The logic here is that if a block hasn't been used in a while, it's less likely to be part of the current working set than a block that was more recently used.

An LRU algorithm, though ideal, isn't quite feasible to implement in real life. Control logic that checks each cache block to determine which one is the least recently used not only would add complexity to the cache design, but such a check would also take up valuable time and thereby add unwanted latency to each replacement. Most caches wind up implementing some type of *pseudo-LRU algorithm* that approximates true LRU by marking blocks as more and more *dirty* the longer they sit unused in the cache. When a new block is fetched into the cache, the dirtiest blocks are the first to be replaced.

Sometimes, blocks that aren't all that dirty get replaced by newer blocks, just because there isn't enough room in the cache to contain the entire working set. A miss that results when a block containing needed data has been evicted from the cache due to a lack of cache capacity is called a *capacity miss*. This is the third and final type of cache miss.

Block Sizes

In the section on spatial locality I mentioned that storing whole blocks is one way that caches take advantage of spatial locality of reference. Now that you know a little more about how caches are organized internally, you can look a closer at the issue of block size.

You might think that as cache sizes increase, you can take even better advantage of spatial locality by making block sizes even bigger. Surely fetching more bytes per block into the cache would decrease the odds that some part of the working set will be evicted because it resides in a different block. This is true to some extent, but you have to be careful. If you increase the block size while keeping the cache size the same, you decrease the number of blocks that the cache can hold. Fewer blocks in the cache means fewer sets, and fewer sets means that collisions and therefore misses are more likely. And of course, with fewer blocks in the cache, the likelihood decreases that any particular block that the CPU needs will be available in the cache.

The upshot of all this is that smaller block sizes allow you to exercise more fine-grained control of the cache. You can trace out the boundaries of a working set with a higher resolution by using smaller cache blocks. If your cache blocks are too large, you wind up with a lot of wasted cache space, because many of the blocks will contain only a few bytes from the working set while the rest is irrelevant data. If you think of this issue in terms of cache pollution, you can say that large cache blocks are more prone to pollute the cache with non-reusable data than small cache blocks.

Figure 11-11 shows the memory map we've been using, sitting in a cache with large block sizes.

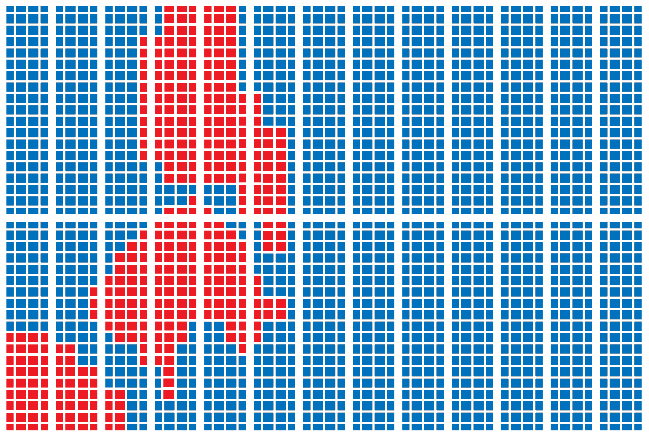


Figure 11-11: Memory map with large block sizes

Figure 11-12 shows the same map, but with the block sizes decreased. Notice how much more control the smaller blocks allow over cache pollution. The smaller cache blocks have a higher average ratio of red to blue in each block, which means that it's easier to keep the precise contours of the working set in the cache.

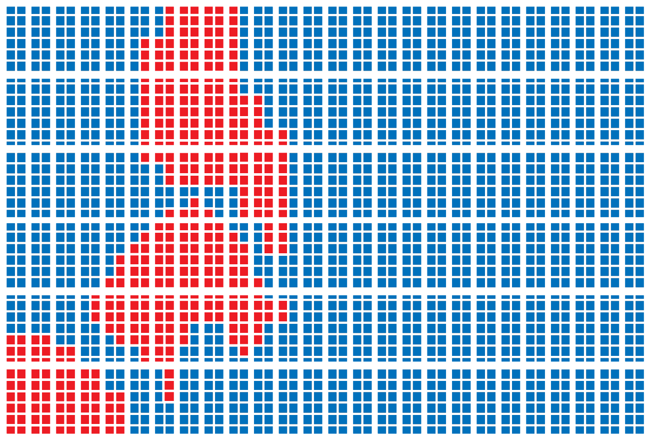


Figure 11-12: Memory map with small block sizes

The other problems with large block sizes are bandwidth related. The larger the block size, the more data is fetched with each load, so large block sizes can really eat up memory bus bandwidth, especially if the miss rate is high. A system therefore needs plenty of bandwidth if it's going to make good use of large cache blocks. Otherwise, the increase in bus traffic can increase the amount of time it takes to fetch a cache block from memory, thereby adding latency to the cache.

Write Policies: Write-Through vs. Write-Back

So far, this entire chapter has dealt with only one type of memory traffic: *loads*, or requests for data from memory. I've talked only about loads because they make up the vast majority of memory traffic. The remainder of memory traffic is made up of stores, which in simple uniprocessor systems are much easier to deal with. In this section, I'll explain how to handle stores in single-processor systems with just an L1. When you throw in more caches and multiple processors, things get more complicated than I want to go into here.

Once a retrieved piece of data is modified by the CPU, it must be stored or written back out to main memory so that the rest of the system has access to the most up-to-date version of it.

There are two ways to deal with such writes to memory. The first way is to immediately update all the copies of the modified data in each level of the cache hierarchy to reflect the latest changes. A piece of modified data would be written to both the L1 and main memory so that all of its copies are current. Such a policy for handling writes is called *write-through*, since it writes the modified data through to all levels of the hierarchy.

A write-through policy can be nice for multiprocessor and I/O–intensive system designs, since multiple clients are reading from memory at once and all need the most current data available. However, the multiple updates per write required by this policy can greatly increase memory traffic. For each `store`, the system must update multiple copies of the modified data. If there's a large amount of data that has been modified, this increased write activity could eat up quite a bit of memory bandwidth that could be used for the more important `load` traffic.

The alternative to write through is *write-back*, and it can potentially result in less memory traffic. With a write-back policy, changes propagate down to the lower levels of the hierarchy as cache blocks are evicted from the higher levels. An updated piece of data in an L1 cache block will not be updated in main memory until that block is evicted from the L1. The trade-off for write-back is that it's more complex to keep the multiple copies of the data in sync, especially in a multiprocessor system.

Conclusions

There is much, much more that can be said about caching; this chapter has covered only the basic concepts. As main memory moves farther away from the CPU in terms of CPU clock cycles, the importance of caching will only increase. For modern microprocessor systems, larger on-die caches have turned out to be one of the simplest and most effective uses for the increased transistor budgets that new manufacturing process technologies afford.