CHAPTER 28

Digital Signal Processors

Digital Signal Processing is carried out by mathematical operations. In comparison, word processing and similar programs merely rearrange stored data. This means that computers designed for business and other general applications are not optimized for algorithms such as digital filtering and Fourier analysis. *Digital Signal Processors* are microprocessors specifically designed to handle Digital Signal Processing tasks. These devices have seen tremendous growth in the last decade, finding use in everything from cellular telephones to advanced scientific instruments. In fact, hardware engineers use "DSP" to mean *Digital Signal Processor*, just as algorithm developers use "DSP" to mean *Digital Signal Processing*. This chapter looks at how DSPs are different from other types of microprocessors, how to decide if a DSP is right for your application, and how to get started in this exciting new field. In the next chapter we will take a more detailed look at one of these sophisticated products: the Analog Devices SHARC® family.

How DSPs are Different from Other Microprocessors

In the 1960s it was predicted that artificial intelligence would revolutionize the way humans interact with computers and other machines. It was believed that by the end of the century we would have robots cleaning our houses, computers driving our cars, and voice interfaces controlling the storage and retrieval of information. This hasn't happened; these abstract tasks are far more complicated than expected, and very difficult to carry out with the step-by-step logic provided by digital computers.

However, the last forty years have shown that computers are extremely capable in two broad areas, (1) **data manipulation**, such as word processing and database management, and (2) **mathematical calculation**, used in science, engineering, and Digital Signal Processing. All microprocessors can perform both tasks; however, it is difficult (expensive) to make a device that is *optimized* for both. There are technical tradeoffs in the hardware design, such as the size of the instruction set and how interrupts are handled. Even

	Data Manipulation	Math Calculation
Typical Applications	Word processing, database management, spread sheets, operating sytems, etc.	Digital Signal Processing, motion control, scientific and engineering simulations, etc.
Main Operations	data movement ($A \rightarrow B$) value testing (*If* $A=B$ *then* ...)	addition ($A+B=C$) multiplication ($A \times B=C$)

FIGURE 28-1
Data manipulation versus mathematical calculation. Digital computers are useful for two general tasks: *data manipulation* and *mathematical calculation*. Data manipulation is based on moving data and testing inequalities, while mathematical calculation uses multiplication and addition.

more important, there are *marketing* issues involved: development and manufacturing cost, competitive position, product lifetime, and so on. As a broad generalization, these factors have made traditional microprocessors, such as the Pentium®, primarily directed at data manipulation. Similarly, DSPs are designed to perform the mathematical calculations needed in Digital Signal Processing.

Figure 28-1 lists the most important differences between these two categories. Data manipulation involves storing and sorting information. For instance, consider a word processing program. The basic task is to store the information (typed in by the operator), organize the information (cut and paste, spell checking, page layout, etc.), and then retrieve the information (such as saving the document on a floppy disk or printing it with a laser printer). These tasks are accomplished by *moving* data from one location to another, and *testing* for inequalities ($A=B$, $A<B$, etc.). As an example, imagine sorting a list of words into alphabetical order. Each word is represented by an 8 bit number, the ASCII value of the first letter in the word. Alphabetizing involved rearranging the order of the words until the ASCII values continually increase from the beginning to the end of the list. This can be accomplished by repeating two steps over-and-over until the alphabetization is complete. First, test two adjacent entries for being in alphabetical order (IF A>B THEN ...). Second, if the two entries are not in alphabetical order, switch them so that they are ($A \rightleftarrows B$). When this two step process is repeated many times on all adjacent pairs, the list will eventually become alphabetized.

As another example, consider how a document is printed from a word processor. The computer continually tests the input device (mouse or keyboard) for the binary code that indicates "print the document." When this code is detected, the program moves the data from the computer's memory to the printer. Here we have the same two basic operations: moving data and inequality testing. While mathematics is occasionally used in this type of application, it is infrequent and does not significantly affect the overall execution speed.

In comparison, the execution speed of most DSP algorithms is limited almost completely by the number of multiplications and additions required. For example, Fig. 28-2 shows the implementation of an FIR digital filter, the most common DSP technique. Using the standard notation, the input signal is referred to by *x*[ ], while the output signal is denoted by *y*[ ]. Our task is to calculate the sample at location *n* in the output signal, i.e., *y*[*n*]. An FIR filter performs this calculation by multiplying appropriate samples from the input signal by a group of coefficients, denoted by: $a_0, a_1, a_2, a_3, \cdots$, and then adding the products. In equation form, *y*[*n*] is found by:

$$y[n] = a_0 x[n] + a_1 x[n-1] + a_2 x[n-2] + a_3 x[n-3] + a_4 x[n-4] + \cdots$$

This is simply saying that the input signal has been *convolved* with a filter kernel (i.e., an impulse response) consisting of: $a_0, a_1, a_2, a_3, \cdots$. Depending on the application, there may only be a few coefficients in the filter kernel, or many thousands. While there is some data transfer and inequality evaluation in this algorithm, such as to keep track of the intermediate results and control the loops, the math operations dominate the execution time.

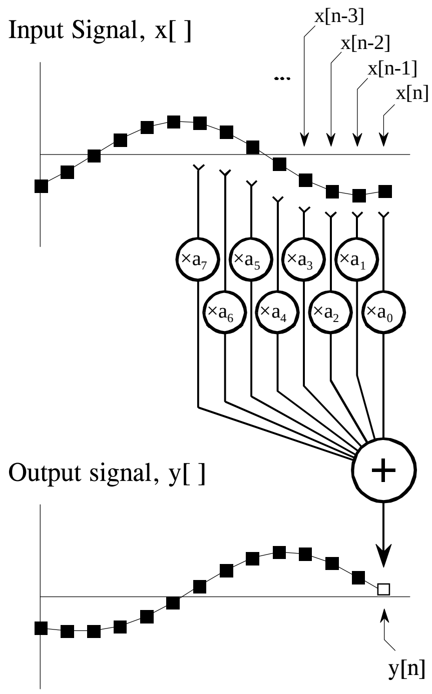

FIGURE 28-2
FIR digital filter. In FIR filtering, each sample in the output signal, y[n], is found by multiplying samples from the input signal, x[n], x[n-1], x[n-2], ..., by the filter kernel coefficients, a_0, a_1, a_2, a_3 ..., and summing the products.

In addition to preforming mathematical calculations very rapidly, DSPs must also have a *predictable* execution time. Suppose you launch your desktop computer on some task, say, converting a word-processing document from one form to another. It doesn't matter if the processing takes ten *milliseconds* or ten *seconds*; you simply wait for the action to be completed before you give the computer its next assignment.

In comparison, most DSPs are used in applications where the processing is *continuous*, not having a defined start or end. For instance, consider an engineer designing a DSP system for an audio signal, such as a hearing aid. If the digital signal is being received at 20,000 samples per second, the DSP must be able to maintain a sustained throughput of 20,000 samples per second. However, there are important reasons not to make it any faster than necessary. As the speed increases, so does the *cost*, the *power consumption*, the *design difficulty*, and so on. This makes an accurate knowledge of the execution time critical for selecting the proper device, as well as the algorithms that can be applied.

Circular Buffering

Digital Signal Processors are designed to quickly carry out FIR filters and similar techniques. To understand the *hardware*, we must first understand the *algorithms*. In this section we will make a detailed list of the steps needed to implement an FIR filter. In the next section we will see how DSPs are designed to perform these steps as efficiently as possible.

To start, we need to distinguish between **off-line processing** and **real-time processing**. In off-line processing, the *entire* input signal resides in the computer at the same time. For example, a geophysicist might use a seismometer to record the ground movement during an earthquake. After the shaking is over, the information may be read into a computer and analyzed in some way. Another example of off-line processing is medical imaging, such as computed tomography and MRI. The data set is acquired while the patient is inside the machine, but the image reconstruction may be delayed until a later time. The key point is that *all* of the information is simultaneously available to the processing program. This is common in scientific research and engineering, but not in consumer products. Off-line processing is the realm of personal computers and mainframes.

In real-time processing, the output signal is produced at the same time that the input signal is being acquired. For example, this is needed in telephone communication, hearing aids, and radar. These applications must have the information immediately available, although it can be delayed by a short amount. For instance, a 10 millisecond delay in a telephone call cannot be detected by the speaker or listener. Likewise, it makes no difference if a radar signal is delayed by a few seconds before being displayed to the operator. Real-time applications input a sample, preform the algorithm, and output a sample, over-and-over. Alternatively, they may input a group

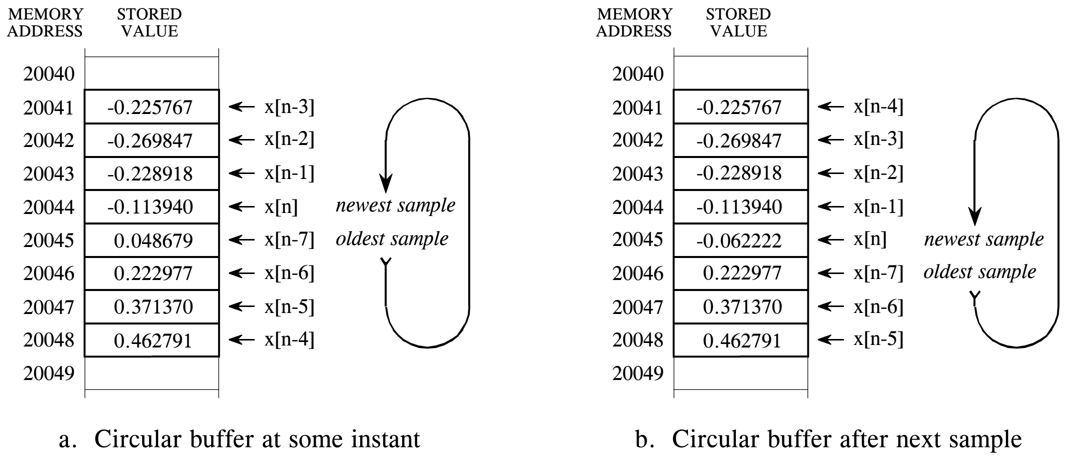

FIGURE 28-3
Circular buffer operation. Circular buffers are used to store the most recent values of a continually updated signal. This illustration shows how an eight sample circular buffer might appear at some instant in time (a), and how it would appear one sample later (b).

of samples, perform the algorithm, and output a group of samples. This is the world of Digital Signal Processors.

Now look back at Fig. 28-2 and imagine that this is an FIR filter being implemented in *real-time*. To calculate the output sample, we must have access to a certain number of the most recent samples from the input. For example, suppose we use eight coefficients in this filter, a_0, a_1, $\cdots a_7$. This means we must know the value of the eight most recent samples from the input signal, $x[n]$, $x[n-1]$, $\cdots x[n-7]$. These eight samples must be stored in memory and continually updated as new samples are acquired. What is the best way to manage these stored samples? The answer is **circular buffering**.

Figure 28-3 illustrates an eight sample circular buffer. We have placed this circular buffer in eight consecutive memory locations, 20041 to 20048. Figure (a) shows how the eight samples from the input might be stored at one particular instant in time, while (b) shows the changes after the next sample is acquired. The idea of circular buffering is that the end of this linear array is connected to its beginning; memory location 20041 is viewed as being next to 20048, just as 20044 is next to 20045. You keep track of the array by a **pointer** (a variable whose value is an *address*) that indicates where the most recent sample resides. For instance, in (a) the pointer contains the address 20044, while in (b) it contains 20045. When a new sample is acquired, it replaces the oldest sample in the array, and the pointer is moved one address ahead. Circular buffers are efficient because only one value needs to be changed when a new sample is acquired.

Four parameters are needed to manage a circular buffer. First, there must be a pointer that indicates the start of the circular buffer in memory (in this example, 20041). Second, there must be a pointer indicating the end of the array (e.g., 20048), or a variable that holds its length (e.g., 8). Third, the step size of the memory addressing must be specified. In Fig. 28-3 the step size is *one*, for example: address 20043 contains one sample, address 20044 contains the next sample, and so on. This is frequently not the case. For instance, the addressing may refer to bytes, and each sample may require two or four bytes to hold its value. In these cases, the step size would need to be two or four, respectively.

These three values define the size and configuration of the circular buffer, and will not change during the program operation. The fourth value, the pointer to the most recent sample, must be modified as each new sample is acquired. In other words, there must be program logic that controls how this fourth value is updated based on the value of the first three values. While this logic is quite simple, it must be very fast. This is the whole point of this discussion; DSPs should be optimized at managing circular buffers to achieve the highest possible execution speed.

As an aside, circular buffering is also useful in *off-line* processing. Consider a program where both the input and the output signals are completely contained in memory. Circular buffering isn't needed for a convolution calculation, because every sample can be immediately accessed. However, many algorithms are implemented in *stages*, with an intermediate signal being created between each stage. For instance, a recursive filter carried out as a series of biquads operates in this way. The brute force method is to store the entire length of each intermediate signal in memory. Circular buffering provides another option: store only those intermediate samples needed for the calculation at hand. This reduces the required amount of memory, at the expense of a more complicated algorithm. The important idea is that circular buffers are *useful* for off-line processing, but *critical* for real-time applications.

Now we can look at the steps needed to implement an FIR filter using circular buffers for both the input signal and the coefficients. This list may seem trivial and overexamined- it's not! The efficient handling of these individual tasks is what separates a DSP from a traditional microprocessor. For each new sample, all the following steps need to be taken:

TABLE 28-1
FIR filter steps.

```
    1. Obtain a sample with the ADC; generate an interrupt
    2. Detect and manage the interrupt
    3. Move the sample into the input signal's circular buffer
    4. Update the pointer for the input signal's circular buffer
    5. Zero the accumulator
┌─> 6. Control the loop through each of the coefficients
│     7. Fetch the coefficient from the coefficient's circular buffer
│     8. Update the pointer for the coefficient's circular buffer
│     9. Fetch the sample from the input signal's circular buffer
│    10. Update the pointer for the input signal's circular buffer
│    11. Multiply the coefficient by the sample
└─── 12. Add the product to the accumulator
   13. Move the output sample (accumulator) to a holding buffer
   14. Move the output sample from the holding buffer to the DAC
```

The goal is to make these steps execute quickly. Since steps 6-12 will be repeated many times (once for each coefficient in the filter), special attention must be given to these operations. Traditional microprocessors must generally carry out these 14 steps in *serial* (one after another), while DSPs are designed to perform them in *parallel*. In some cases, all of the operations within the loop (steps 6-12) can be completed in a *single clock cycle*. Let's look at the internal architecture that allows this magnificent performance.

Architecture of the Digital Signal Processor

One of the biggest bottlenecks in executing DSP algorithms is transferring information to and from memory. This includes *data*, such as samples from the input signal and the filter coefficients, as well as *program instructions*, the binary codes that go into the program sequencer. For example, suppose we need to multiply two numbers that reside somewhere in memory. To do this, we must fetch three binary values from memory, the numbers to be multiplied, plus the program instruction describing what to do.

Figure 28-4a shows how this seemingly simple task is done in a traditional microprocessor. This is often called a **Von Neumann architecture**, after the brilliant American mathematician John Von Neumann (1903-1957). Von Neumann guided the mathematics of many important discoveries of the early twentieth century. His many achievements include: developing the concept of a stored program computer, formalizing the mathematics of quantum mechanics, and work on the atomic bomb. If it was new and exciting, Von Neumann was there!

As shown in (a), a Von Neumann architecture contains a single memory and a single bus for transferring data into and out of the central processing unit (CPU). Multiplying two numbers requires at least three clock cycles, one to transfer each of the three numbers over the bus from the memory to the CPU. We don't count the time to transfer the result back to memory, because we assume that it remains in the CPU for additional manipulation (such as the sum of products in an FIR filter). The Von Neumann design is quite satisfactory when you are content to execute all of the required tasks in serial. In fact, most computers today are of the Von Neumann design. We only need other architectures when very fast processing is required, and we are willing to pay the price of increased complexity.

This leads us to the **Harvard architecture**, shown in (b). This is named for the work done at Harvard University in the 1940s under the leadership of Howard Aiken (1900-1973). As shown in this illustration, Aiken insisted on separate memories for data and program instructions, with separate buses for each. Since the buses operate independently, program instructions and data can be fetched at the same time, improving the speed over the single bus design. Most present day DSPs use this dual bus architecture.

Figure (c) illustrates the next level of sophistication, the **Super Harvard Architecture**. This term was coined by Analog Devices to describe the internal operation of their ADSP-2106x and new ADSP-211xx families of Digital Signal Processors. These are called **SHARC®** DSPs, a contraction of the longer term, Super Harvard ARChitecture. The idea is to build upon the Harvard architecture by adding features to improve the throughput. While the SHARC DSPs are optimized in dozens of ways, two areas are important enough to be included in Fig. 28-4c: an *instruction cache*, and an *I/O controller*.

First, let's look at how the instruction cache improves the performance of the Harvard architecture. A handicap of the basic Harvard design is that the data memory bus is busier than the program memory bus. When two numbers are multiplied, two binary values (the numbers) must be passed over the data memory bus, while only one binary value (the program instruction) is passed over the program memory bus. To improve upon this situation, we start by relocating part of the "data" to program memory. For instance, we might place the filter coefficients in program memory, while keeping the input signal in data memory. (This relocated data is called "secondary data" in the illustration). At first glance, this doesn't seem to help the situation; now we must transfer one value over the data memory bus (the input signal sample), but two values over the program memory bus (the program instruction and the coefficient). In fact, if we were executing random instructions, this situation would be no better at all.

However, DSP algorithms generally spend most of their execution time in loops, such as instructions 6-12 of Table 28-1. This means that the same set of program instructions will continually pass from program memory to the CPU. The Super Harvard architecture takes advantage of this situation by including an **instruction cache** in the CPU. This is a small memory that contains about 32 of the most recent program instructions. The first time through a loop, the program instructions must be passed over the program memory bus. This results in slower operation because of the conflict with the coefficients that must also be fetched along this path. However, on additional executions of the loop, the program instructions can be pulled from the instruction cache. This means that all of the memory to CPU information transfers can be accomplished in a single cycle: the sample from the input signal comes over the data memory bus, the coefficient comes over the program memory bus, and the program instruction comes from the instruction cache. In the jargon of the field, this efficient transfer of data is called a *high memory-access bandwidth*.

Figure 28-5 presents a more detailed view of the SHARC architecture, showing the **I/O controller** connected to data memory. This is how the signals enter and exit the system. For instance, the SHARC DSPs provides both serial and parallel communications ports. These are extremely high speed connections. For example, at a 40 MHz clock speed, there are two serial ports that operate at 40 Mbits/second each, while six parallel ports each provide a 40 Mbytes/second data transfer. When all six parallel ports are used together, the data transfer rate is an incredible 240 Mbytes/second.

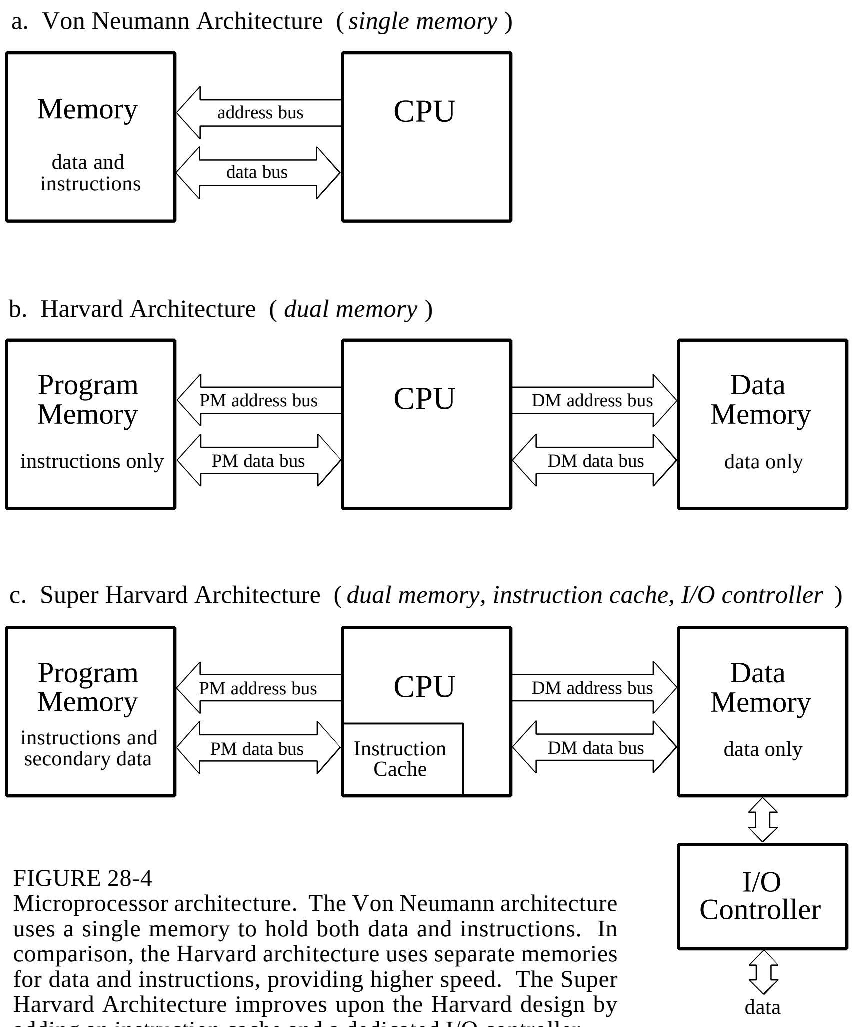

FIGURE 28-4
Microprocessor architecture. The Von Neumann architecture uses a single memory to hold both data and instructions. In comparison, the Harvard architecture uses separate memories for data and instructions, providing higher speed. The Super Harvard Architecture improves upon the Harvard design by adding an instruction cache and a dedicated I/O controller.

This is fast enough to transfer the entire text of this book in only 2 milliseconds! Just as important, dedicated hardware allows these data streams to be transferred directly into memory (Direct Memory Access, or DMA), without having to pass through the CPU's registers. In other words, tasks 1 & 14 on our list happen independently and simultaneously with the other tasks; no cycles are stolen from the CPU. The main buses (program memory bus and data memory bus) are also accessible from outside the chip, providing an additional interface to off-chip memory and peripherals. This allows the SHARC DSPs to use a four Gigaword (16 Gbyte) memory, accessible at 40 Mwords/second (160 Mbytes/second), for 32 bit data. Wow!

This type of high speed I/O is a key characteristic of DSPs. The overriding goal is to move the data in, perform the math, and move the data out before the next sample is available. Everything else is secondary. Some DSPs have on-board analog-to-digital and digital-to-analog converters, a feature called **mixed signal**. However, all DSPs can interface with external converters through serial or parallel ports.

Now let's look inside the CPU. At the top of the diagram are two blocks labeled **Data Address Generator** (DAG), one for each of the two memories. These control the addresses sent to the program and data memories, specifying where the information is to be read from or written to. In simpler microprocessors this task is handled as an inherent part of the program sequencer, and is quite transparent to the programmer. However, DSPs are designed to operate with *circular buffers*, and benefit from the extra hardware to manage them efficiently. This avoids needing to use precious CPU clock cycles to keep track of how the data are stored. For instance, in the SHARC DSPs, each of the two DAGs can control *eight* circular buffers. This means that each DAG holds 32 variables (4 per buffer), plus the required logic.

Why so many circular buffers? Some DSP algorithms are best carried out in stages. For instance, IIR filters are more stable if implemented as a cascade of biquads (a stage containing two poles and up to two zeros). Multiple stages require multiple circular buffers for the fastest operation. The DAGs in the SHARC DSPs are also designed to efficiently carry out the *Fast Fourier transform*. In this mode, the DAGs are configured to generate **bit-reversed addresses** into the circular buffers, a necessary part of the FFT algorithm. In addition, an abundance of circular buffers greatly simplifies DSP code generation- both for the human programmer as well as high-level language compilers, such as C.

The data register section of the CPU is used in the same way as in traditional microprocessors. In the ADSP-2106x SHARC DSPs, there are 16 general purpose registers of 40 bits each. These can hold intermediate calculations, prepare data for the math processor, serve as a buffer for data transfer, hold flags for program control, and so on. If needed, these registers can also be used to control loops and counters; however, the SHARC DSPs have extra hardware registers to carry out many of these functions.

The math processing is broken into three sections, a **multiplier**, an **arithmetic logic unit (ALU)**, and a **barrel shifter**. The multiplier takes the values from two registers, multiplies them, and places the result into another register. The ALU performs addition, subtraction, absolute value, logical operations (AND, OR, XOR, NOT), conversion between fixed and floating point formats, and similar functions. Elementary binary operations are carried out by the barrel shifter, such as shifting, rotating, extracting and depositing segments, and so on. A powerful feature of the SHARC family is that the multiplier and the ALU can be accessed in parallel. In a single clock cycle, data from registers 0-7 can be passed to the multiplier, data from registers 8-15 can be passed to the ALU, and the two results returned to any of the 16 registers.

There are also many important features of the SHARC family architecture that aren't shown in this simplified illustration. For instance, an 80 bit accumulator is built into the multiplier to reduce the round-off error associated with multiple fixed-point math operations. Another interesting

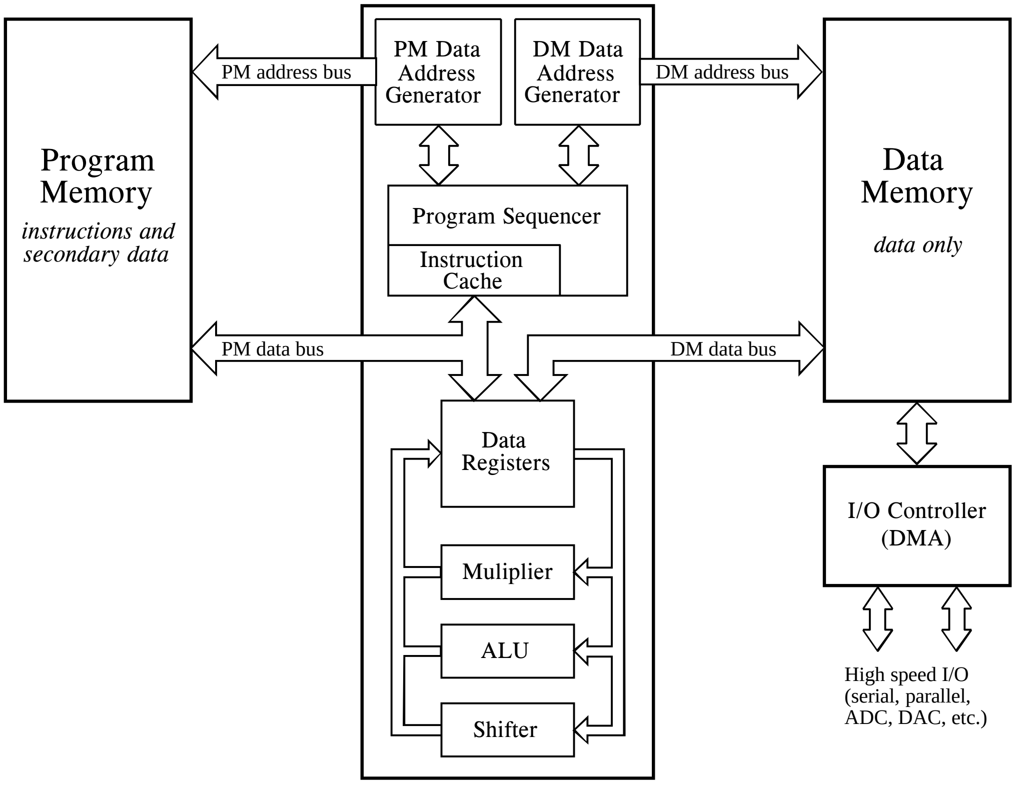

FIGURE 28-5
Typical DSP architecture. Digital Signal Processors are designed to implement tasks in parallel. This simplified diagram is of the Analog Devices SHARC DSP. Compare this architecture with the tasks needed to implement an FIR filter, as listed in Table 28-1. All of the steps within the loop can be executed in a single clock cycle.

feature is the use of **shadow registers** for all the CPU's key registers. These are duplicate registers that can be switched with their counterparts in a single clock cycle. They are used for *fast context switching*, the ability to handle interrupts quickly. When an interrupt occurs in traditional microprocessors, all the internal data must be saved before the interrupt can be handled. This usually involves pushing all of the occupied registers onto the stack, one at a time. In comparison, an interrupt in the SHARC family is handled by moving the internal data into the shadow registers in a *single clock cycle*. When the interrupt routine is completed, the registers are just as quickly restored. This feature allows step 4 on our list (managing the sample-ready interrupt) to be handled very quickly and efficiently.

Now we come to the critical performance of the architecture, how many of the operations within the loop (steps 6-12 of Table 28-1) can be carried out at the same time. Because of its highly parallel nature, the SHARC DSP can simultaneously carry out *all* of these tasks. Specifically, within a single clock cycle, it can perform a multiply (step 11), an addition (step 12), two data moves (steps 7 and 9), update two circular buffer pointers (steps 8 and 10), and control the loop (step 6). There will be extra clock cycles associated with beginning and ending the loop (steps 3, 4, 5 and 13, plus moving initial values into place); however, these tasks are also handled very efficiently. If the loop is executed more than a few times, this overhead will be negligible. As an example, suppose you write an efficient FIR filter program using 100 coefficients. You can expect it to require about 105 to 110 clock cycles per sample to execute (i.e., 100 coefficient loops plus overhead). This is very impressive; a traditional microprocessor requires many thousands of clock cycles for this algorithm.

Fixed versus Floating Point

Digital Signal Processing can be divided into two categories, **fixed point** and **floating point**. These refer to the format used to store and manipulate numbers within the devices. Fixed point DSPs usually represent each number with a minimum of 16 bits, although a different length can be used. For instance, Motorola manufactures a family of fixed point DSPs that use 24 bits. There are four common ways that these $2^{16} = 65{,}536$ possible bit patterns can represent a number. In **unsigned integer**, the stored number can take on any integer value from 0 to 65,535. Similarly, **signed integer** uses two's complement to make the range include negative numbers, from -32,768 to 32,767. With **unsigned fraction** notation, the 65,536 levels are spread uniformly between 0 and 1. Lastly, the **signed fraction** format allows negative numbers, equally spaced between -1 and 1.

In comparison, floating point DSPs typically use a minimum of 32 bits to store each value. This results in many more bit patterns than for fixed point, $2^{32} = 4{,}294{,}967{,}296$ to be exact. A key feature of floating point notation is that the represented numbers are *not* uniformly spaced. In the most common format (ANSI/IEEE Std. 754-1985), the largest and smallest numbers are $\pm 3.4 \times 10^{38}$ and $\pm 1.2 \times 10^{-38}$, respectively. The represented values are unequally spaced between these two extremes, such that the gap between any two numbers is about ten-million times smaller than the value of the numbers. This is important because it places large gaps between large numbers, but small gaps between small numbers. Floating point notation is discussed in more detail in Chapter 4.

All floating point DSPs can also handle fixed point numbers, a necessity to implement counters, loops, and signals coming from the ADC and going to the DAC. However, this doesn't mean that fixed point math will be carried out as quickly as the floating point operations; it depends on the internal architecture. For instance, the SHARC DSPs are optimized for both floating point and fixed point operations, and executes them with equal efficiency. For this reason, the SHARC devices are often referred to as "32-bit DSPs," rather than just "Floating Point."

Figure 28-6 illustrates the primary trade-offs between fixed and floating point DSPs. In Chapter 3 we stressed that fixed point arithmetic is much

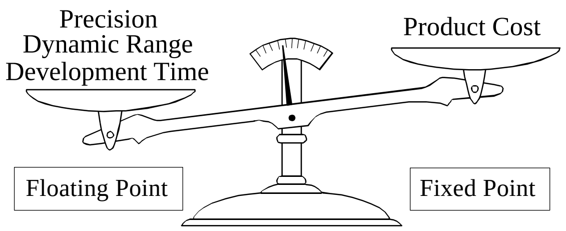

FIGURE 28-6
Fixed versus floating point. Fixed point DSPs are generally cheaper, while floating point devices have better precision, higher dynamic range, and a shorter development cycle.

faster than floating point in general purpose computers. However, with DSPs the speed is about the same, a result of the hardware being highly optimized for math operations. The internal architecture of a floating point DSP is more complicated than for a fixed point device. All the registers and data buses must be 32 bits wide instead of only 16; the multiplier and ALU must be able to quickly perform floating point arithmetic, the instruction set must be larger (so that they can handle both floating and fixed point numbers), and so on. Floating point (32 bit) has better precision and a higher dynamic range than fixed point (16 bit) . In addition, floating point programs often have a shorter development cycle, since the programmer doesn't generally need to worry about issues such as overflow, underflow, and round-off error.

On the other hand, fixed point DSPs have traditionally been cheaper than floating point devices. Nothing changes more rapidly than the price of electronics; anything you find in a book will be out-of-date before it is printed. Nevertheless, cost is a key factor in understanding how DSPs are evolving, and we need to give you a general idea. When this book was completed in early 1998, fixed point DSPs sold for between \$5 and \$100, while floating point devices were in the range of \$10 to \$300. This difference in cost can be viewed as a measure of the relative complexity between the devices. If you want to find out what the prices are *today*, you need to look *today*.

Now let's turn our attention to *performance*; what can a 32-bit floating point system do that a 16-bit fixed point can't? The answer to this question is **signal-to-noise ratio**. Suppose we store a number in a 32 bit floating point format. As previously mentioned, the gap between this number and its adjacent neighbor is about one ten-millionth of the value of the number. To store the number, it must be round up or down by a maximum of one-half the gap size. In other words, each time we store a number in floating point notation, we add *noise* to the signal.

The same thing happens when a number is stored as a 16-bit fixed point value, except that the added noise is much worse. This is because the gaps between adjacent numbers are much larger. For instance, suppose we store the number 10,000 as a signed integer (running from -32,768 to 32,767). The gap between numbers is one ten-thousandth of the value of the number we are storing. If we want to store the number 1000, the gap between numbers is only one one-thousandth of the value.

Noise in signals is usually represented by its *standard deviation*. This was discussed in detail in Chapter 2. For here, the important fact is that the standard deviation of this **quantization noise** is about one-third of the gap size. This means that the signal-to-noise ratio for storing a floating point number is about 30 million to one, while for a fixed point number it is only about ten-thousand to one. In other words, floating point has roughly 30,000 times less quantization noise than fixed point.

This brings up an important way that DSPs are different from traditional microprocessors. Suppose we implement an FIR filter in fixed point. To do this, we loop through each coefficient, multiply it by the appropriate sample from the input signal, and add the product to an accumulator. Here's the problem. In traditional microprocessors, this accumulator is just another 16 bit fixed point variable. To avoid overflow, we need to scale the values being added, and will correspondingly add quantization noise on each step. In the worst case, this quantization noise will simply add, greatly lowering the signal-to-noise ratio of the system. For instance, in a 500 coefficient FIR filter, the noise on each output sample may be 500 times the noise on each input sample. The signal-to-noise ratio of *ten-thousand to one* has dropped to a ghastly *twenty to one*. Although this is an extreme case, it illustrates the main point: when many operations are carried out on each sample, it's bad, really bad. See Chapter 3 for more details.

DSPs handle this problem by using an **extended precision accumulator**. This is a special register that has 2-3 times as many bits as the other memory locations. For example, in a 16 bit DSP it may have 32 to 40 bits, while in the SHARC DSPs it contains 80 bits for fixed point use. This extended range virtually eliminates round-off noise while the accumulation is in progress. The only round-off error suffered is when the accumulator is scaled and stored in the 16 bit memory. This strategy works very well, although it does limit how some algorithms must be carried out. In comparison, floating point has such low quantization noise that these techniques are usually not necessary.

In addition to having lower quantization noise, floating point systems are also easier to develop algorithms for. Most DSP techniques are based on repeated multiplications and additions. In fixed point, the possibility of an overflow or underflow needs to be considered after each operation. The programmer needs to continually understand the amplitude of the numbers, how the quantization errors are accumulating, and what scaling needs to take place. In comparison, these issues do not arise in floating point; the numbers take care of themselves (except in rare cases).

To give you a better understanding of this issue, Fig. 28-7 shows a table from the SHARC user manual. This describes the ways that multiplication can be carried out for both fixed and floating point formats. First, look at how floating point numbers can be multiplied; there is only one way! That

```
              Fixed Point                        |        Floating Point
-------------------------------------------------|-------------------------------
                                                 |
| Rn   | = Rx * Ry     ( |S| |S| | F | )          |        Fn = Fx * Fy
| MRF  |                 |U| |U| | I |            |
| MRB  |                         |FR |            |
                                                 |
| Rn    = MRF |  + Rx * Ry  ( |S| |S| | F | )     |
| Rn    = MRB |                |U| |U| | I |      |
| MRF   = MRF |                        |FR |      |
| MRB   = MRB |                                   |
                                                 |
| Rn    = MRF |  -  Rx * Ry ( |S| |S| | F | )     |
| Rn    = MRB |                |U| |U| | I |      |
| MRF   = MRF |                        |FR |      |
| MRB   = MRB |                                   |
                                                 |
| Rn    = SAT MRF |     | (SI) |                  |
| Rn    = SAT MRB |     | (UI) |                  |
| MRF   = SAT MRF |     | (SF) |                  |
| MRB   = SAT MRB |     | (UF) |                  |
                                                 |
| Rn    = RND MRF |     | (SF) |                  |
| Rn    = RND MRB |     | (UF) |                  |
| MRF   = RND MRF |                             |
| MRB   = RND MRB |                             |
                                                 |
| MRF | = 0                                      |
| MRB |                                          |
                                                 |
| MRxF | = Rn                                    |
| MRxB |                                         |
                                                 |
  Rn  = | MRxF |                                 |
        | MRxB |                                 |
```

FIGURE 28-7
Fixed versus floating point instructions. These are the multiplication instructions used in the SHARC DSPs. While only a single command is needed for floating point, many options are needed for fixed point. See the text for an explanation of these options.

is, Fn = Fx * Fy, where Fn, Fx, and Fy are any of the 16 data registers. It could not be any simpler. In comparison, look at all the possible commands for fixed point multiplication. These are the many options needed to efficiently handle the problems of round-off, scaling, and format.

In Fig. 28-7, Rn, Rx, and Ry refer to any of the 16 data registers, and MRF and MRB are 80 bit accumulators. The vertical lines indicate *options*. For instance, the top-left entry in this table means that all the following are valid commands: Rn = Rx * Ry, MRF = Rx * Ry, and MRB = Rx * Ry. In other words, the value of any two registers can be multiplied and placed into another register, or into one of the extended precision accumulators. This table also shows that the numbers may be either signed or unsigned (S or U), and may be fractional or integer (F or I). The RND and SAT options are ways of controlling rounding and register overflow.

There are other details and options in the table, but they are not important for our present discussion. The important idea is that the fixed point programmer must understand *dozens* of ways to carry out the very basic task of multiplication. In contrast, the floating point programmer can spend his time concentrating on the algorithm.

Given these tradeoffs between fixed and floating point, how do you choose which to use? Here are some things to consider. First, look at how many bits are used in the ADC and DAC. In many applications, 12-14 bits per sample is the crossover for using fixed versus floating point. For instance, television and other video signals typically use 8 bit ADC and DAC, and the precision of fixed point is acceptable. In comparison, professional audio applications can sample with as high as 20 or 24 bits, and almost certainly need floating point to capture the large dynamic range.

The next thing to look at is the complexity of the algorithm that will be run. If it is relatively simple, think fixed point; if it is more complicated, think floating point. For example, FIR filtering and other operations in the time domain only require a few dozen lines of code, making them suitable for fixed point. In contrast, frequency domain algorithms, such as spectral analysis and FFT convolution, are very detailed and can be much more difficult to program. While they can be written in fixed point, the development time will be greatly reduced if floating point is used.

Lastly, think about the money: how important is the cost of the product, and how important is the cost of the development? When fixed point is chosen, the cost of the product will be reduced, but the development cost will probably be higher due to the more difficult algorithms. In the reverse manner, floating point will generally result in a quicker and cheaper development cycle, but a more expensive final product.

Figure 28-8 shows some of the major trends in DSPs. Figure (a) illustrates the impact that Digital Signal Processors have had on the *embedded* market. These are applications that use a microprocessor to directly operate and control some larger system, such as a cellular telephone, microwave oven, or automotive instrument display panel. The name "microcontroller" is often used in referring to these devices, to distinguish them from the microprocessors used in personal computers. As shown in (a), about 38% of embedded designers have already started using DSPs, and another 49% are considering the switch. The high throughput and computational power of DSPs often makes them an ideal choice for embedded designs.

As illustrated in (b), about twice as many engineers currently use fixed point as use floating point DSPs. However, this depends greatly on the application. Fixed point is more popular in competitive consumer products where the cost of the electronics must be kept very low. A good example of this is cellular telephones. When you are in competition to sell millions of your product, a cost difference of only a few dollars can be the difference between success and failure. In comparison, floating point is more common when greater performance is needed and cost is not important. For

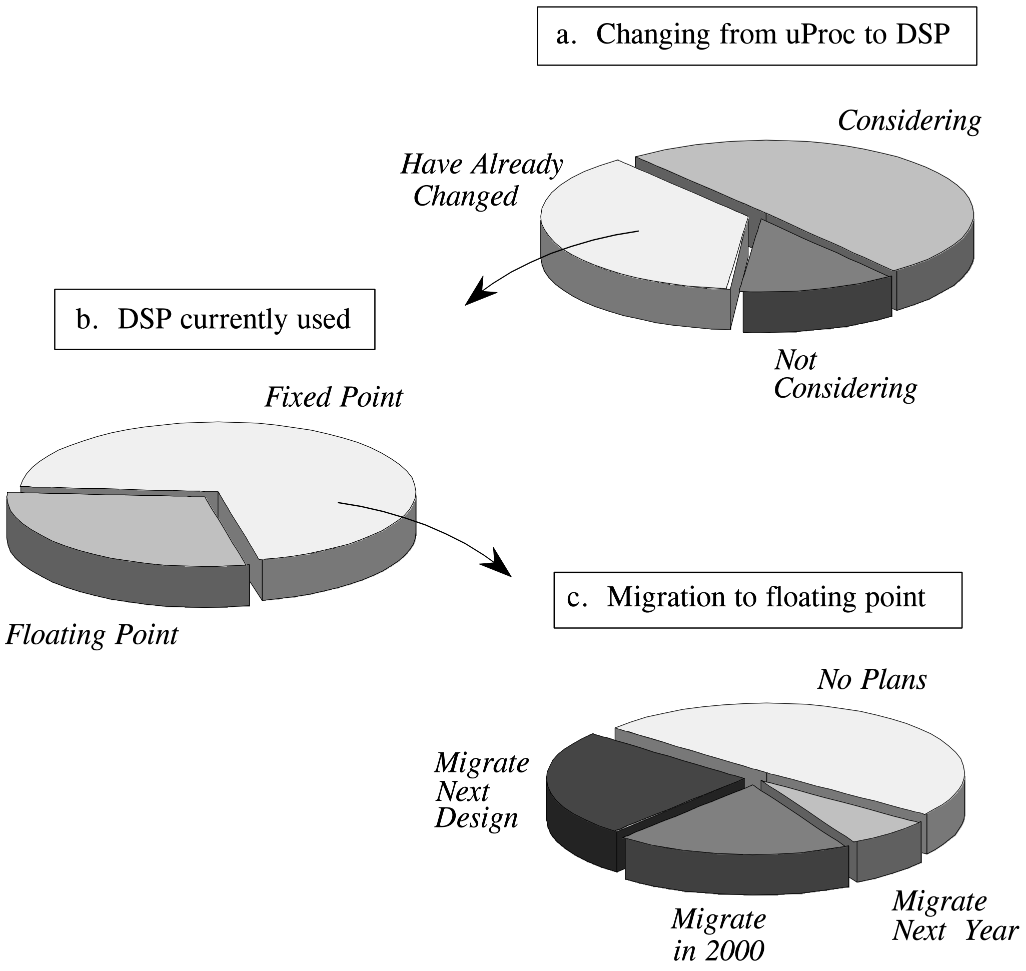

FIGURE 28-8
Major trends in DSPs. As illustrated in (a), about 38% of embedded designers have already switched from conventional microprocessors to DSPs, and another 49% are considering the change. In (b), about twice as many engineers use fixed point as use floating point DSPs. This is mainly driven by consumer products that must have low cost electronics, such as cellular telephones. However, as shown in (c), floating point is the fastest growing segment; over one-half of engineers currently using 16 bit devices plan to migrate to floating point DSPs

instance, suppose you are designing a medical imaging system, such a computed tomography scanner. Only a few hundred of the model will ever be sold, at a price of several hundred-thousand dollars each. For this application, the cost of the DSP is insignificant, but the performance is critical. In spite of the larger number of fixed point DSPs being used, the floating point market is the fastest growing segment. As shown in (c), over one-half of engineers using 16-bits devices plan to migrate to floating point at some time in the near future.

Before leaving this topic, we should reemphasize that floating point and fixed point usually use 32 bits and 16 bits, respectively, *but not always*. For instance, the SHARC family can represent numbers in 32-bit fixed point, a mode that is common in digital audio applications. This makes the 2^{32} quantization levels spaced uniformly over a relatively small range, say, between -1 and 1. In comparison, floating point notation places the 2^{32} quantization levels logarithmically over a huge range, typically $\pm 3.4 \times 10^{38}$. This gives 32-bit fixed point better *precision*, that is, the quantization error on any one sample will be lower. However, 32-bit floating point has a higher *dynamic range*, meaning there is a greater difference between the largest number and the smallest number that can be represented.

C versus Assembly

DSPs are programmed in the same languages as other scientific and engineering applications, usually *assembly* or *C*. Programs written in assembly can execute faster, while programs written in C are easier to develop and maintain. In traditional applications, such as programs run on personal computers and mainframes, C is almost always the first choice. If assembly is used at all, it is restricted to short subroutines that must run with the utmost speed. This is shown graphically in Fig. 28-9a; for every traditional programmer that works in assembly, there are approximately *ten* that use C.

However, DSP programs are different from traditional software tasks in two important respects. First, the programs are usually much shorter, say, one-hundred lines versus ten-thousand lines. Second, the execution speed is often a critical part of the application. After all, that's why someone uses a DSP in the first place, for its blinding speed. These two factors motivate many software engineers to switch from C to assembly for programming Digital Signal Processors. This is illustrated in (b); nearly as many DSP programmers use assembly as use C.

Figure (c) takes this further by looking at the revenue produced by DSP products. For every dollar made with a DSP programmed in C, two dollars are made with a DSP programmed in assembly. The reason for this is simple; money is made by outperforming the competition. From a pure performance standpoint, such as execution speed and manufacturing cost, assembly almost always has the advantage over C. For instance, C code usually requires a larger memory than assembly, resulting in more expensive hardware. However, the DSP market is continually changing. As the market grows, manufacturers will respond by designing DSPs that are *optimized* for programming in C. For instance, C is much more efficient when there is a large, general purpose register set and a unified memory space. These future improvements will minimize the difference in execution time between C and assembly, and allow C to be used in more applications.

To better understand this decision between C and assembly, let's look at a typical DSP task programmed in each language. The example we will use is the calculation of the *dot product* of the two arrays, $x[\,]$ and $y[\,]$. This is a simple mathematical operation, we multiply each coefficient in one

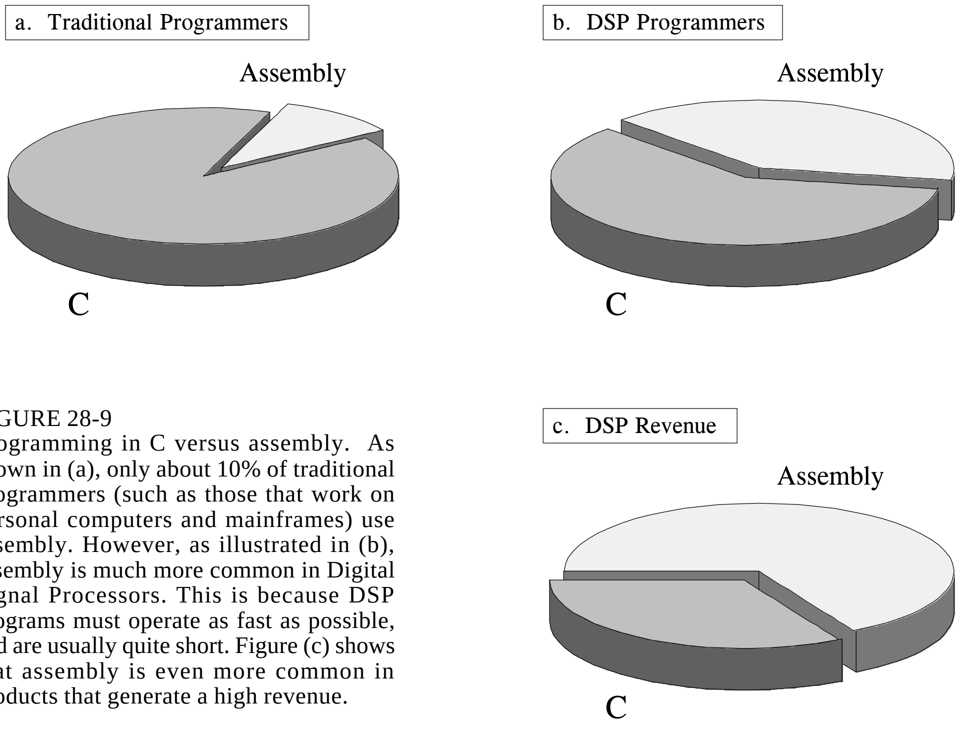

FIGURE 28-9
Programming in C versus assembly. As shown in (a), only about 10% of traditional programmers (such as those that work on personal computers and mainframes) use assembly. However, as illustrated in (b), assembly is much more common in Digital Signal Processors. This is because DSP programs must operate as fast as possible, and are usually quite short. Figure (c) shows that assembly is even more common in products that generate a high revenue.

array by the corresponding coefficient in the other array, and sum the products, i.e. $x[0]\times y[0] + x[1]\times y[1] + x[2]\times y[2] + \cdots$. This should look very familiar; it is the fundamental operation in an FIR filter. That is, each sample in the output signal is found by multiplying stored samples from the input signal (in one array) by the filter coefficients (in the other array), and summing the products.

Table 28-2 shows how the dot product is calculated in a C program. In lines 001-004 we define the two arrays, *x*[ ] and *y*[ ], to be 20 elements long. We also define *result*, the variable that holds the calculated dot

TABLE 28-2
Dot product in C. This progam calculates the dot product of two arrays, x[ ] and y[ ], and stores the result in the variable, *result*.

```
001    #define LEN 20
002      float dm x[LEN];
003      float pm y[LEN];
004      float result;
005
006    main()
007
008    {
009      int n;
010      float s;
011      for (n=0;n<LEN;n++)
012        s += x[n]*y[n];
013      result = s
014    }
```

product at the completion of the program. Line 011 controls the 20 loops needed for the calculation, using the variable *n* as a loop counter. The only statement within the loop is line 012, which multiplies the corresponding coefficients from the two arrays, and adds the product to the accumulator variable, *s*. (If you are not familiar with C, the statement: $s += x[n] * y[n]$ means the same as: $s = s + x[n] * y[n]$). After the loop, the value in the accumulator, *s*, is transferred to the output variable, *result*, in line 013.

A key advantage of using a high-level language (such as C, Fortran, or Basic) is that the programmer does not need to understand the architecture of the microprocessor being used; knowledge of the architecture is left to the compiler. For instance, this short C program uses several variables: *n*, *s*, *result*, plus the arrays: *x*[ ] and *y*[ ]. All of these variables must be assigned a "home" in hardware to keep track of their value. Depending on the microprocessor, these storage locations can be the general purpose data registers, locations in the main memory, or special registers dedicated to particular functions. However, the person writing a high-level program knows little or nothing about this memory management; this task has been delegated to the software engineer who wrote the compiler. The problem is, these two people have never met; they only communicate through a set of predefined rules. High-level languages are easier than assembly because you give half the work to someone else. However, they are less efficient because you aren't quite sure how the delegated work is being carried out.

In comparison, Table 28-3 shows the dot product program written in assembly for the SHARC DSP. The assembly language for the Analog Devices DSPs (both their 16 bit fixed-point and 32 bit SHARC devices) are known for their simple algebraic-like syntax. While we won't go through all the details, here is the general operation. Notice that *everything* relates to hardware; there are no abstract variables in this code, only data registers and memory locations.

Each semicolon represents a clock cycle. The arrays *x*[ ] and *y*[ ] are held in circular buffers in the main memory. In lines 001 and 002, registers i4

```
001    i12 = _y;                          /* i12 points to beginning of y[ ] */
002    i4 = _x;                           /* i4 points to beginning of x[ ] */
003
004    lcntr = 20, do (pc,4) until lce;   /* loop for the 20 array entries */
005      f2 = dm(i4,m6);                  /* load the x[ ] value into register f2 */
006      f4 = pm(i12,m14);                /* load the y[ ] value into register f4 */
007      f8 = f2*f4;                      /* multiply the two values, store in f8 */
008      f12 = f8 + f12;                  /* add the product to the accumulator in f12 */
009
010    dm(_result) = f12;                 /* write the accumulator to memory */
```

TABLE 28-3
Dot product in assembly (unoptimized). This program calculates the dot product of the two arrays, x[ ] and y[ ], and stores the result in the variable, *result*. This is assembly code for the Analog Devices SHARC DSPs. See the text for details.

```
001      i12 = _y;                                          /* i12 points to beginning of y[ ] */
002      i4 = _x;                                           /* i4 points to beginning of x[ ] */
003
004      f2 = dm(i4,m6), f4 = pm(i12,m14)                   /* prime the registers */
005      f8 = f2*f4, f2 = dm(i4,m6), f4 = pm(i12,m14);
006
007      lcntr = 18, do (pc,1) until lce;                   /* highly efficient main loop */
008      f12 = f8 + f12, f8 = f2*f4, f2 = dm(i4,m6), f4 = pm(k12,m14);
009
010      f12 = f8 + f12, f8 = f2*f4;                        /* complete the last loop */
011      f12 = f8 + f12;
012
013      dm(_result) = f12;                                 /* store the result in memory */
```

TABLE 28-4
Dot product in assembly (optimized). This is an optimized version of the program in TABLE 28-2, designed to take advantage of the SHARC's highly parallel architecture.

and i12 are pointed to the starting locations of these arrays. Next, we execute 20 loop cycles, as controlled by line 004. The format for this statement takes advantage of the SHARC DSP's *zero-overhead looping* capability. In other words, all of the variables needed to control the loop are held in dedicated hardware registers that operate in parallel with the other operations going on inside the microprocessor. In this case, the register: *lcntr* (loop counter) is loaded with an initial value of 20, and decrements each time the loop is executed. The loop is terminated when *lcntr* reaches a value of zero (indicated by the statement: *lce*, for "loop counter expired"). The loop encompasses lines 004 to 008, as controlled by the statement (pc,4). That is, the loop ends four lines after the current program counter.

Inside the loop, line 005 loads the value from *x*[ ] into data register f2, while line 006 loads the value from *y*[ ] into data register f4. The symbols "dm" and "pm" indicate that the values are fetched over the "data memory" bus and "program memory" bus, respectively. The variables: i4, m6, i12, and m14 are registers in the data address generators that manage the circular buffers holding *x*[ ] and *y*[ ]. The two values in f2 and f4 are multiplied in line 007, and the product stored in data register f8. In line 008, the product in f8 is added to the accumulator, data register f12. After the loop is completed, the accumulator in f12 is transferred to memory.

This program correctly calculates the dot product, but it does not take advantage of the SHARC highly parallel architecture. Table 28-4 shows this program rewritten in a highly optimized form, with many operations being carried out in parallel. First notice that line 007 only executes 18 loops, rather than 20. Also notice that this loop only contains a single line (008), but that this line contains multiple instructions. The strategy is to make the loop as efficient as possible, in this case, a single line that can be executed in a single clock cycle. To do this, we need to have a small amount of code to "prime" the registers on the first loop (lines 004 and 005), and another small section of code to finish the last loop (lines 010 and 011).

To understand how this works, study line 008, the only statement inside the loop. In this single statement, *four* operations are being carried out in parallel: (1) the value for *x*[ ] is moved from a circular buffer in program memory and placed in f2; (2) the value for *y*[ ] is being moved from a circular buffer in data memory and placed in f4; (3) the previous values of f2 and f4 are multiplied and placed in f8; and (4) the previous value in f8 is added to the accumulator in f12.

For example, the fifth time that line 008 is executed, *x*[7] and *y*[7] are fetched from memory and stored in f2 and f4. At the same time, the values for *x*[6] and *y*[6] (that were in f2 and f4 at the start of this cycle) are multiplied and placed in f8. In addition, the value of $x[5] \times y[5]$ (that was in f8 at the start of this cycle) is added to the value of f12.

Let's compare the number of clock cycles required by the unoptimized and the optimized programs. Keep in mind that there are 20 loops, with four actions being required in each loop. The unoptimized program requires 80 clock cycles to carry out the actions within the loops, plus 5 clock cycles of overhead, for a total of 85 clock cycles. In comparison, the optimized program conducts 18 loops in 18 clock cycles, but requires 11 clock cycles of overhead to prime the registers and complete the last loop. This results in a total execution time of 29 clock cycles, or about three times faster than the brute force method.

Here is the big question: How fast does the C program execute relative to the assembly code? When the program in Table 28-2 is compiled, does the executable code resemble our *efficient* or *inefficient* assembly example? The answer is that the compiler generates the *efficient* code. However, it is important to realize that the dot product is a very simple example. The compiler has a much more difficult time producing optimized code when the program becomes more complicated, such as multiple nested loops and erratic jumps to subroutines. If you are doing something straightforward, expect the compiler to provide you a nearly optimal solution. If you are doing something strange or complicated, expect that an assembly program will execute significantly faster than one written in C. In the worst case, think a factor of 2-3. As previously mentioned, the efficiency of C versus assembly depends greatly on the particular DSP being used. Floating point architectures can generally be programmed more efficiently than fixed-point devices when using high-level languages such as C. Of course, the proper software tools are important for this, such as a debugger with profiling features that help you understand how long different code segments take to execute.

There is also a way you can get the best of both worlds: write the program in C, but use assembly for the critical sections that must execute quickly. This is one reason that C is so popular in science and engineering. It operates as a high-level language, but also allows you to directly manipulate

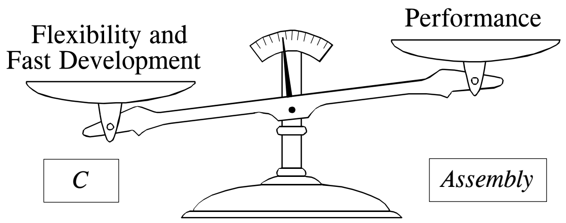

FIGURE 28-10
Assembly versus C. Programs in C are more flexible and quicker to develop. In comparison, programs in assembly often have better performance; they run faster and use less memory, resulting in lower cost.

the hardware if you so desire. Even if you intend to program only in C, you will probably need some knowledge of the architecture of the DSP and the assembly instruction set. For instance, look back at lines 002 and 003 in Table 28-2, the dot product program in C. The "dm" means that x[ ] is to be stored in data memory, while the "pm" indicates that y[ ] will reside in program memory. Even though the program is written in a high level language, a basic knowledge of the hardware is still required to get the best performance from the device.

Which language is best for *your* application? It depends on what is more important to you. If you need flexibility and fast development, choose C. On the other hand, use assembly if you need the best possible performance. As illustrated in Fig. 28-10, this is a tradeoff you are forced to make. Here are some things you should consider.

❏ How complicated is the program? If it is large and intricate, you will probably want to use C. If it is small and simple, assembly may be a good choice.

❏ Are you pushing the maximum speed of the DSP? If so, assembly will give you the last drop of performance from the device. For less demanding applications, assembly has little advantage, and you should consider using C.

❏ How many programmers will be working together? If the project is large enough for more than one programmer, lean toward C and use in-line assembly only for time critical segments.

❏ Which is more important, *product cost* or *development cost*? If it is product cost, choose assembly; if it is development cost, choose C.

❏ What is your background? If you are experienced in assembly (on other microprocessors), choose assembly for your DSP. If your previous work is in C, choose C for your DSP.

❏ What does the DSP's manufacturer suggest you use?

This last item is very important. Suppose you ask a DSP manufacturer which language to use, and they tell you: *"Either C or assembly can be used, but we* *recommend C."* You had better take their advice! What they are really saying is: *"Our DSP is so difficult to program in assembly that you will need 6 months of training to use it."* On the other hand, some DSPs are easy to program in assembly. For instance, the Analog Devices products are in this category. Just ask their engineers; they are very proud of this.

One of the best ways to make decisions about DSP products and software is to speak with engineers who have used them. Ask the manufacturers for references of companies using their products, or search the web for people you can e-mail. Don't be shy; engineers love to give their opinions on products they have used. They will be flattered that you asked.

How Fast are DSPs?

The primary reason for using a DSP instead of a traditional microprocessor is *speed*, the ability to move samples into the device, carry out the needed mathematical operations, and output the processed data. This brings up the question: How fast are DSPs? The usual way of answering this question is **benchmarks**, methods for expressing the speed of a microprocessor as a number. For instance, fixed point systems are often quoted in **MIPS** (million integer operations per second). Likewise, floating point devices can be specified in **MFLOPS** (million floating point operations per second).

One hundred and fifty years ago, British Prime Minister Benjamin Disraeli declared that there are three types of lies: *lies*, *damn lies*, and *statistics*. If Disraeli were alive today and working with microprocessors, he would add *benchmarks* as a fourth category. The idea behind benchmarks is to provide a head-to-head comparison to show which is the best device. Unfortunately, this often fails in practicality, because different microprocessors excel in different areas. Imagine asking the question: Which is the better car, a Cadillac or a Ferrari? It depends on what you want it for!

Confusion about benchmarks is aggravated by the competitive nature of the electronics industry. Manufacturers want to show their products in the best light, and they will use any ambiguity in the testing procedure to their advantage. There is an old saying in electronics: "*A specification writer can get twice as much performance from a device as an engineer*." These people aren't being untruthful, they are just paid to have good imaginations. Benchmarks should be viewed as a *tool* for a complicated task. If you are inexperienced in using this tool, you may come to the wrong conclusion. A better approach is to look for specific information on the execution speed of the algorithms you plan to carry out. For instance, if your application calls for an FIR filter, look for the exact number of clock cycles it takes for the device to execute this particular task.

Using this strategy, let's look at the time required to execute various algorithms on our featured DSP, the Analog Devices SHARC family. Keep

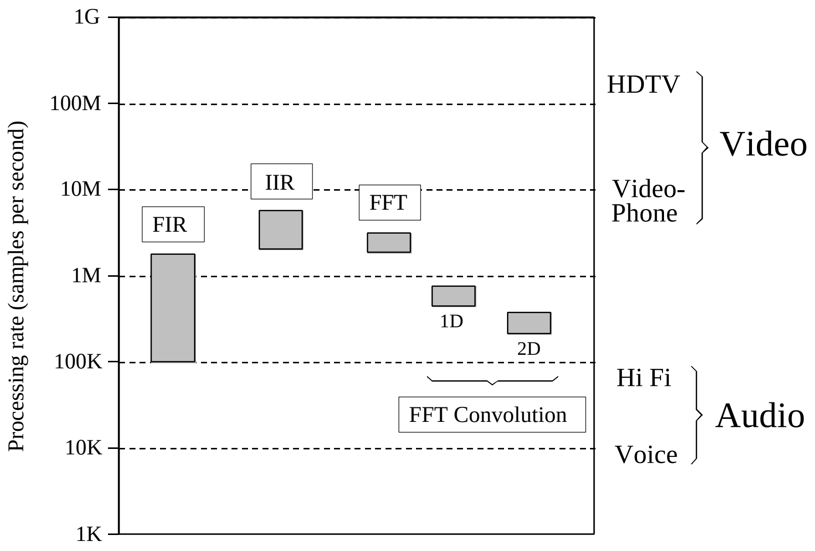

FIGURE 28-11
The speed of DSPs. The throughput of a particular DSP algorithm can be found by dividing the clock rate by the required number of clock cycles per sample. This illustration shows the range of throughput for four common algorithms, executed on a SHARC DSP at a clock speed of 40 MHz.

in mind that microprocessor speed is doubling about every three years. This means you should pay special attention to the *method* we use in this example. The actual numbers are always changing, and you will need to repeat the calculations every time you start a new project. In the world of twenty-first century technology, blink and you are out-of-date!

When it comes to understanding execution time, the SHARC family is one of the easiest DSP to work with. This is because it can carry out a multiply-accumulate operation in a single clock cycle. Since most FIR filters use 25 to 400 coefficients, 25 to 400 clock cycles are required, respectively, for each sample being processed. As previously described, there is a small amount of overhead needed to achieve this loop efficiency (priming the first loop and completing the last loop), but it is negligible when the number of loops is this large. To obtain the throughput of the filter, we can divide the SHARC clock rate (40 MHz at present) by the number of clock cycles required per sample. This gives us a maximum FIR data rate of about 100k to 1.6M samples/second. The calculations can't get much simpler than this! These FIR throughput values are shown in Fig. 28-11.

The calculations are just as easy for recursive filters. Typical IIR filters use about 5 to 17 coefficients. Since these loops are relatively short, we will add a small amount of overhead, say 3 cycles per sample. This results in 8 to 20 clock cycles being required per sample of processed data. For the

40 MHz clock rate, this provides a maximum IIR throughput of 1.8M to 3.1M samples/second. These IIR values are also shown in Fig. 28-11.

Next we come to the frequency domain techniques, based on the Fast Fourier Transform. FFT subroutines are almost always provided by the manufacturer of the DSP. These are highly-optimized routines written in assembly. The specification sheet of the ADSP-21062 SHARC DSP indicates that a 1024 sample complex FFT requires 18,221 clock cycles, or about 0.46 milliseconds at 40 MHz. To calculate the throughput, it is easier to view this as 17.8 clock cycles per sample. This "per-sample" value only changes slightly with longer or shorter FFTs. For instance, a 256 sample FFT requires about 14.2 clock cycles per sample, and a 4096 sample FFT requires 21.4 clock cycles per sample. Real FFTs can be calculated about 40% faster than these complex FFT values. This makes the overall range of all FFT routines about 10 to 22 clock cycles per sample, corresponding to a throughput of about 1.8M to 3.3M samples/second.

FFT convolution is a fast way to carry out FIR filters. In a typical case, a 512 sample segment is taken from the input, padded with an additional 512 zeros, and converted into its frequency spectrum by using a 1024 point FFT. After multiplying this spectrum by the desired frequency response, a 1024 point Inverse FFT is used to move back into the time domain. The resulting 1024 points are combined with the adjacent processed segments using the overlap-add method. This produces 512 points of the output signal.

How many clock cycles does this take? Each 512 sample segment requires two 1024 point FFTs, plus a small amount of overhead. In round terms, this is about a factor of five greater than for a single FFT of 512 points. Since the real FFT requires about 12 clock cycles per sample, FFT convolution can be carried out in about 60 clock cycles per sample. For a 2106x SHARC DSP at 40 MHz, this corresponds to a data throughput of approximately 660k samples/second.

Notice that this is about the same as a 60 coefficient FIR filter carried out by conventional convolution. In other words, if an FIR filter has less than 60 coefficients, it can be carried out faster by standard convolution. If it has greater than 60 coefficients, FFT convolution is quicker. A key advantage of FFT convolution is that the execution time only increases as the logarithm of the number of coefficients. For instance a 4,096 point filter kernel only requires about 30% longer to execute as one with only 512 points.

FFT convolution can also be applied in two-dimensions, such as for image processing. For instance, suppose we want to process an 800×600 pixel image in the frequency domain. First, pad the image with zeros to make it 1024×1024. The two-dimensional frequency spectrum is then calculated by taking the FFT of each of the rows, followed by taking the FFT of each of the resulting columns. After multiplying this 1024×1024 spectrum by the desired frequency response, the two-dimensional Inverse FFT is taken. This is carried out by taking the Inverse FFT of each of the rows, and then each of the resulting columns. Adding the number of clock cycles and dividing by the number of samples, we find that this entire procedure takes roughly 150 clock cycles per pixel. For a 40 MHz ADSP-2106, this corresponds to a data throughput of about 260k samples/second.

Comparing these different techniques in Fig. 28-11, we can make an important observation. *Nearly all DSP techniques require between 4 and 400 instructions (clock cycles in the SHARC family) to execute.* For a SHARC DSP operating at 40 MHz, we can immediately conclude that its data throughput will be between 100k and 10M samples per second, depending on how complex of algorithm is used.

Now that we understand how fast DSPs *can* process digitized signals, let's turn our attention to the other end; how fast do we *need* to process the data? Of course, this depends on the application. We will look at two of the most common, audio and video processing.

The data rate needed for an audio signal depends on the required quality of the reproduced sound. At the low end, telephone quality speech only requires capturing the frequencies between about 100 Hz and 3.2 kHz, dictating a sampling rate of about 8k samples/second. In comparison, high fidelity music must contain the full 20 Hz to 20 kHz range of human hearing. A 44.1 kHz sampling rate is often used for both the left and right channels, making the complete Hi Fi signal 88.2k samples/second. How does the SHARC family compare with these requirements? As shown in Fig. 28-11, it can easily handle high fidelity audio, or process several dozen voice signals at the same time.

Video signals are a different story; they require about *one-thousand* times the data rate of audio signals. A good example of low quality video is the the CIF (Common Interface Format) standard for videophones. This uses 352×288 pixels, with 3 colors per pixel, and 30 frames per second, for a total data rate of 9.1 million samples per second. At the high end of quality there is HDTV (high-definition television), using 1920×1080 pixels, with 3 colors per pixel, and 30 frames per second. This requires a data rate to over 186 million samples per second. These data rates are above the capabilities of a single SHARC DSP, as shown in Fig. 28-11. There are other applications that also require these very high data rates, for instance, radar, sonar, and military uses such as missile guidance.

To handle these high-power tasks, several DSPs can be combined into a single system. This is called **multiprocessing** or **parallel processing**. The SHARC DSPs were designed with this type of multiprocessing in mind, and include special features to make it as easy as possible. For instance, no external hardware logic is required to connect the external busses of multiple SHARC DSPs together; all of the bus arbitration logic is already contained within each device. As an alternative, the link ports (4 bit, parallel) can be used to connect multiple processors in various configurations. Figure 28-12 shows typical ways that the SHARC DSPs can be arranged in multiprocessing systems. In Fig. (a), the algorithm is broken into sequential steps, with each processor performing one of the steps in an "assembly line"

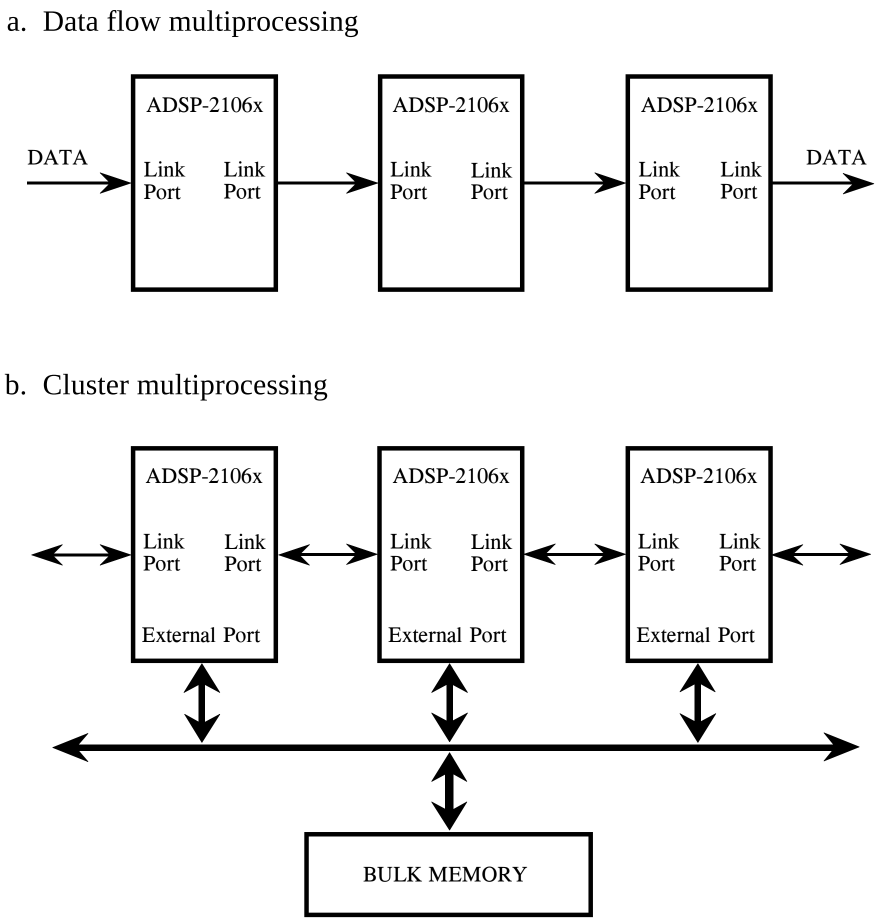

FIGURE 28-12
Multiprocessing configurations. Multiprocessor systems typically use one of two schemes to communicate between processor nodes, (a) dedicated point-to-point communication channels, or (b) a shared global memory accessed over a parallel bus.

strategy. In (b), the processors interact through a single shared global memory, accessed over a parallel bus (i.e., the external port). Figure 28-13 shows another way that a large number of processors can be combined into a single system, a 2D or 3D "mesh." Each of these configuration will have relative advantages and disadvantages for a particular task.

To make the programmer's life easier, the SHARC family uses a *unified address space*. This means that the 4 Gigaword address space, accessed by the 32 bit address bus, is divided among the various processors that are working together. To transfer data from one processor to another, simply read from or write to the appropriate memory locations. The SHARC internal logic takes care of the rest, transferring the data between processors at a rate as high as 240 Mbytes/sec (at 40 MHz).

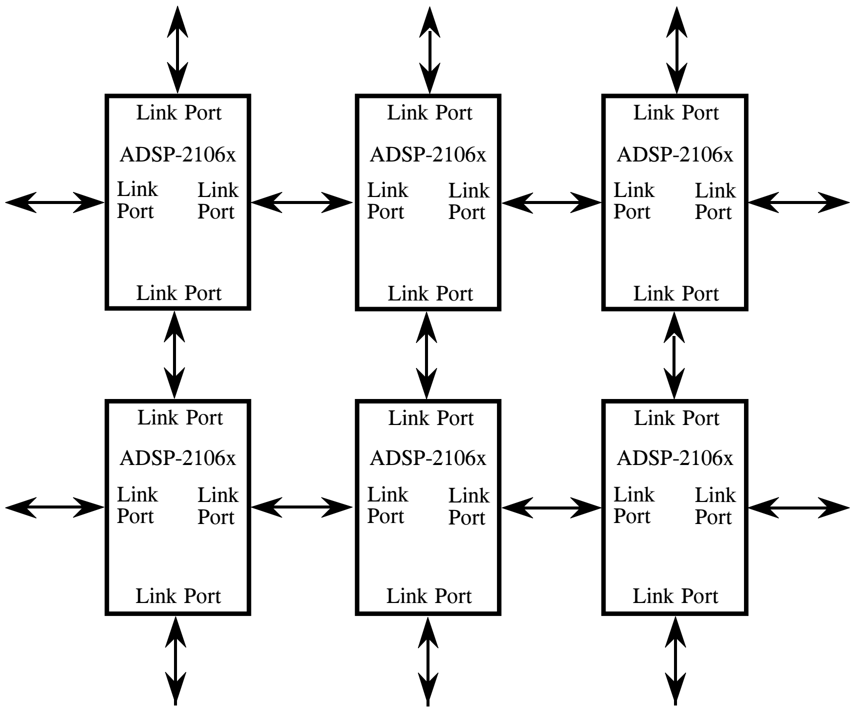

FIGURE 28-13
Multiprocessing "mesh" configuration. For applications such as radar imaging, a 2D or 3D array may be the most efficient way to coordinate a large number of processors.

The Digital Signal Processor Market

The DSP market is very large and growing rapidly. As shown in Fig. 28-14, it will be about 8-10 billion dollars/year at the turn of the century, and growing at a rate of 30-40% each year. This is being fueled by the incessant

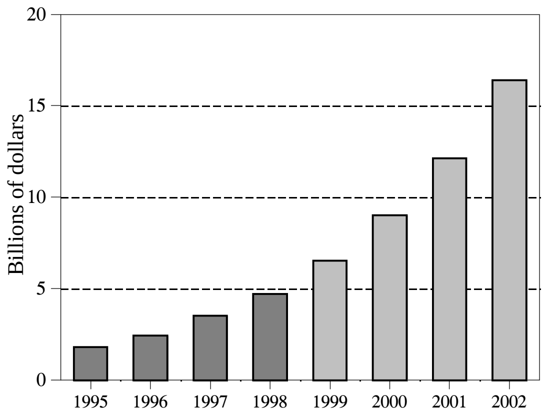

FIGURE 28-14
The DSP market. At the turn of the century, the DSP market will be 8-10 billion dollars per year, and expanding at a rate of about 30-40% per year.

demand for better and cheaper consumer products, such as: cellular telephones, multimedia computers, and high-fidelity music reproduction. These high-revenue applications are shaping the field, while less profitable areas, such as scientific instrumentation, are just riding the wave of technology.

DSPs can be purchased in three forms, as a **core**, as a **processor**, and as a **board level** product. In DSP, the term "core" refers to the section of the processor where the key tasks are carried out, including the data registers, multiplier, ALU, address generator, and program sequencer. A complete **processor** requires combining the core with memory and interfaces to the outside world. While the core and these peripheral sections are designed separately, they will be fabricated on the same piece of silicon, making the *processor* a single integrated circuit.

Suppose you build cellular telephones and want to include a DSP in the design. You will probably want to purchase the DSP as a *processor*, that is, an integrated circuit ("chip") that contains the core, memory and other internal features. For instance, the SHARC ADSP-21060 comes in a "240 lead Metric PQFP" package, only 35×35×4 mm in size. To incorporate this IC in your product, you design a printed circuit board where it will be soldered in next to your other electronics. This is the most common way that DSPs are used.

Now, suppose the company you work for manufactures its own integrated circuits. In this case, you might not want the entire *processor*, just the design of the *core*. After completing the appropriate licensing agreement, you can start making chips that are highly customized to your particular application. This gives you the flexibility of selecting how much memory is included, how the chip receives and transmits data, how it is packaged, and so on. Custom devices of this type are an increasingly important segment of the DSP marketplace.

Lastly, there are several dozen companies that will sell you DSPs already mounted on a printed circuit board. These have such features as extra memory, A/D and D/A converters, EPROM sockets, multiple processors on the same board, and so on. While some of these boards are intended to be used as stand alone computers, most are configured to be plugged into a host, such as a personal computer. Companies that make these types of boards are called **Third Party Developers**. The best way to find them is to ask the manufacturer of the DSP you want to use. Look at the DSP manufacturer's website; if you don't find a list there, send them an e-mail. They will be more than happy to tell you who is using their products and how to contact them.

The present day Digital Signal Processor market (1998) is dominated by four companies. Here is a list, and the general scheme they use for numbering their products:

Analog Devices (*www.analog.com/dsp*)

ADSP-21xx	16 bit, fixed point
ADSP-21xxx	32 bit, floating and fixed point

Lucent Technologies (*www.lucent.com*)

DSP16xxx	16 bit fixed point
DSP32xx	32 bit floating point

Motorola (*www.mot.com*)

DSP561xx	16 bit fixed point
DSP560xx	24 bit, fixed point
DSP96002	32 bit, floating point

Texas Instruments (*www.ti.com*)

TMS320Cxx	16 bit fixed point
TMS320Cxx	32 bit floating point

Keep in mind that the distinction between DSPs and other microprocessors is not always a clear line. For instance, look at how Intel describes the MMX technology addition to its Pentium processor:

> *"Intel engineers have added 57 powerful new instructions specifically designed to manipulate and process video, audio and graphical data efficiently. These instructions are oriented to the highly parallel, repetitive sequences often found in multimedia operations."*

In the future, we will undoubtedly see more DSP-like functions merged into traditional microprocessors and microcontrollers. The internet and other multimedia applications are a strong driving force for these changes. These applications are expanding so rapidly, in twenty years it is very possible that the Digital Signal Processor may *be* the "traditional" microprocessor.

How do you keep up with this rapidly changing field? The best way is to read trade journals that cover the DSP market, such as EDN (Electronic Design News, *www.ednmag.com*), and ECN (Electronic Component News, *www.ecnmag.com*). These are distributed free, and contain up-to-date information on what is available and where the industry is going. Trade journals are a "must-read" for anyone serious about the field. You will also want to be on the mailing list of several DSP manufacturers. This will allow you to receive new product announcements, pricing information, and special offers (such as free software and low-cost evaluation kits). Some manufacturers also distribute periodic newsletters. For instance, Analog Devices publishes *Analog Dialogue* four times a year, containing articles and information on current topics in signal processing. All of these resources, and much more, can be contacted over the internet. Start by exploring the manufacturers' websites, and then sending them e-mail requesting specific information.