CHAPTER 28

Digital Signal Processors
Цифровые Сигнальные процессоры

При редактировании этой и последующих глав, следует обращать особое внимание на использование термина DSP: в одном случае речь идет о DSP как о Цифровой Обработке Сигналов (ЦОС) а в другом о DSP как Цифровом Сигнальном Процессоре (ЦСП)!

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.

Цифровая Обработка сигналов выполнена математическими операциями. Для сравнения, обработка слова и подобные программы просто перестраивает сохраненные данные. Это означает, что компьютеры, предназначенные для бизнеса и других общих приложений, не оптимизированы для алгоритмов типа цифровой фильтрации и анализа Фурье. *Цифровые Сигнальные Процессоры* - микропроцессоры, определенно предназначенные, чтобы обработать задачи Цифровой Обработки Сигналов. Эти устройства видели огромный рост в последнем десятилетии, находя использование во всем от ячеистых телефонов продвинуть научные приборы. Фактически, аппаратные инженеры используют "ЦОС", чтобы означать *Цифровой Сигнальный процессор*(ЦСП), так же, как разработчики алгоритма используют "ЦОС", чтобы означать *Цифровую Обработку сигналов*. Эта глава рассматривает, как ЦОС отличаются от других типов микропроцессоров, как решить, прав ли ЦОС для вашего приложения, и как начать в этом захватывающем новом поле. В следующей главе мы будем брать более детальный, смотрят на одно из этих сложных изделий(программ): семейство Аналоговых Устройств SHARC®.

How DSPs are Different from Other Microprocessors
Как ЦОС Отличаются от Других Микропроцессоров

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.

В 1960-ых было предсказано, что искусственный интеллект будет реконструировать путь, которым люди взаимодействуют с компьютерами и другими машинами. Полагалось, что к концу столетия мы будем иметь роботы, очищающие наши здания, компьютеры, ведущие наши автомобили, и интерфейсы голоса, управляющие хранением и поиском информации. Это не случилось; эти абстрактные задачи гораздо усложнены больше чем ожидаемый, и

очень трудный выполнить с постепенной логикой, обеспеченной цифровыми компьютерами.

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 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.

Однако, последние сорок лет показали, что компьютеры являются чрезвычайно способными в двух широких областях, (1) **манипуляции данных**, типа обработки текстов и управления базы данных, и (2) **математического вычисления**, используемых в науке, разработке, и Цифровой Обработке сигналов. Все микропроцессоры могут исполнять обеих задачи; однако, это трудно (дорого) делать устройство, которое *оптимизировано* для обоих. Имеются технические сделки в аппаратном проекте, типа размера системы команд и как прерывания обработаны. Даже что более важно, имеются вовлеченные проблемы маркетинга: развитие и производящий стоимость, конкурентоспособную позицию, срок службы(продолжительность жизни) изделия(программы), и так далее. Как широкое обобщение, эти факторы сделали традиционными микропроцессоры, типа Pentium ®, направленные, прежде всего на манипуляцию данными. Точно так же ЦОС(ЦСП) разработаны, чтобы исполнить математические вычисления, необходимые в Цифровой Обработке сигналов.

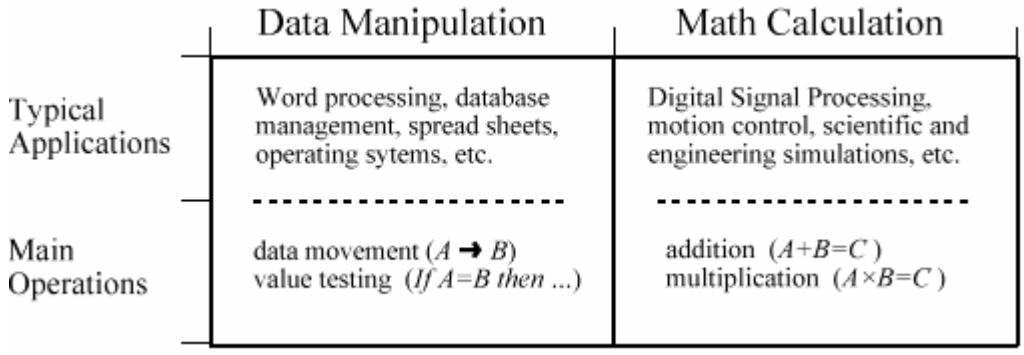


FIGURE 28-1
Data manipulation versus mathematical calculation. Digital computers are useful for two general tasks: *data manipulation* and *mathematical calculation*. Data manipulation is based on moving data and testing inequalities, while mathematical calculation uses multiplication and addition.

РИСУНОК 28-1
Манипуляция Данными против математического вычисления. Цифровые компьютеры полезны для двух общих задач: *манипуляция данными* и *математические вычисления*. Манипуляция Данными основана на перемещении данных и тестирования(проверки) неравенств(несоответствия), в то время как математическое вычисление использует умножение и сложение.

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**W**B). When this two step process is repeated many times on all adjacent pairs, the list will eventually become alphabetized.

Рисунок 28-1 список наиболее важных различий между этими двумя категориями. Манипуляция данными включает в себя(подразумевает) сортировку и сохранение информации. Например, рассмотрите программу обработки текстов. Основная задача состоит в том, чтобы сохранить информацию (напечатанную оператором), организовывать информацию ("вырезать и вставлять", проверку правильности написания, размещение страницы, и т.д.), и затем восстанавливать(отыскивать) информацию (типа сохранения документа относительно гибкого диска или вывода на печать на лазерном принтере). Эти задачи выполнены, перемещая данные от одного расположения в другое, и проверяя на неравенство (A = B, A < B, и т.д.). Как пример, вообразите сортировать список слов в алфавитном порядке. Каждое слово представлено номером 8 двоичных разрядов, значения ASCII первого символа в слове. При упорядочивании по алфавиту вовлечена реконструкция порядка слов, пока значения ASCII непрерывно не увеличиваются с начала к концу списка. Это может быть выполнено, повторяя два шага over-and-over(повторного вычитания), пока упорядочению по алфавиту не завершено. Во первых, проверьте два смежных входа на нахождение в алфавитном порядке IF A > B THEN … (ЕСЛИ A > B ТОГДА ...). Во вторых, если эти два входа не в алфавитном порядке, переключают их так, чтобы они были (A=B). Когда эти два шага процесса повторены много раз на всех смежных парах, список, в конечном счете станет упорядоченным по алфавиту.

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.

Как другой пример, рассмотрите, как документ напечатан в текстовом процессоре. Компьютер непрерывно проверяет устройство ввода данных (мышь или клавиатура) для двоичного кода, который указывает "печать документ." Когда этот код обнаружен, программа перемещает данные из памяти компьютера на принтер. Здесь мы имеем те же самые две основных операции: перемещение данных и проверка неравенства. В то время как математика иногда используется в этом типе приложения, это нечасто и знаменательно не затрагивает полное быстродействие выполнения.

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 ,… and then adding the products. In equation form, is found by:

Для сравнения, быстродействие выполнения большинства алгоритмов ЦОС ограничено почти полностью числом требуемых умножений и сложений. Например, рис. 28-2 показывает выполнение цифровой фильтрации с КИХ, наиболее обычной методики ЦОС. Используя стандартную систему обозначений, входной сигнал упомянут *x*[ ], в то время как сигнал выхода обозначен *y*[ ] Наша задача вычислить выборку в расположении *n* в сигнале выхода, то есть, *y*[*n*]. КИХ-ФИЛЬТР исполняет это вычисление, умножая соответствующие выборки от входного сигнала группой коэффициентов, обозначенных: *a*0 , *a*1 , *a*2 , *a*3 , … и затем складывая продукты. В форме уравнения, *y*[*n*] найден:

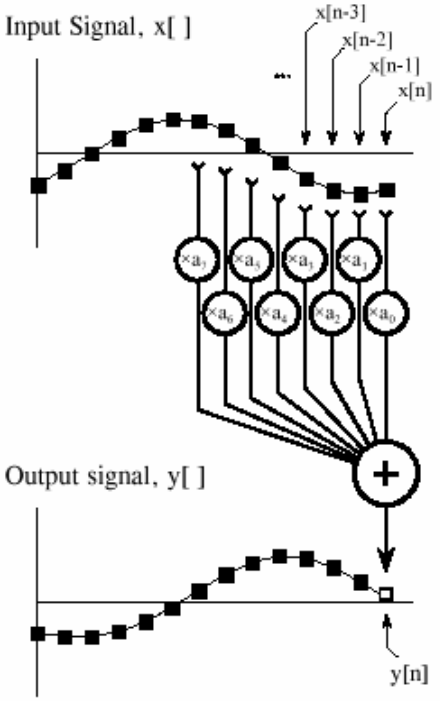
$$y[n] = a_0x[n] + a_1x[n\text{-}1] + a_2x[n\text{-}2] + a_3x[n\text{-}3] + a_4x[n\text{-}4] + \ldots$$

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 , … 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.

Это просто говорит, что входной сигнал был свернут с ядром фильтра (то есть, импульсной передаточной функцией) состоящей из: *a* 0 , *a* 1 , *a* 2 , *a* 3 , … . В зависимости от приложения, могут иметься только несколько коэффициентов в ядре фильтра, или многие тысячи. В то время как имеется некоторая передача(перемещение) данных и оценка неравенства в этом алгоритме, типа следить за промежуточными результатами и управлять циклами, математические операции доминируют над временем выполнения.

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.

РИСУНОК 28-2
Цифровая фильтрация с КИХ. В фильтрации с КИХ, каждая выборка в сигнале выхода, y[n], найдена, умножая выборки от входного сигнала, x [n], x [n-1], x [n-2], ..., коэффициентами ядра фильтра, a_0, a_1, a_2, a_3 ..., и суммируя продукты.



In addition to performing 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.

В дополнение к выполнению математических вычислений очень быстро, ЦСП должны также иметь предсказуемое время выполнения. Предположим, что Вы начинаете(запускаете) ваш настольный компьютер на некоторой задаче, скажем, преобразовывая документ обработки текстов от одной формы в другую. Это не имеет значение, если обработка берет десять миллисекунд или десять секунд; Вы просто ждете действие, которое будет закончено прежде, чем Вы даете компьютеру его следующее задание.

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.

Для сравнения, большинство ЦСП используется в приложениях, где обработка непрерывна, не имея определенное начала или конца. Например, рассмотрите инженера, разрабатывающего систему ЦОС для аудио-сигнала, типа слухового аппарата. Если цифровой сигнал получается при 20000 выборках в секунду, ЦСП должен быть способен обслужить(поддержать) длительную производительность 20000 выборок в секунду. Однако, имеются важные причины не делать это любой быстрее чем необходимый. Как увеличения быстродействия, так что делает стоимость, потребляемую мощность, трудность проекта, и так далее. Это делает точное знание из времени выполнения, критического для отбора надлежащего устройства, также как алгоритмов, которые могут применяться.

Circular Buffering
Кольцевая(Циклическая) Буферизация

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.

Цифровои сигнальный процессор разработан (предназначен), чтобы быстро выполнить КИХ-фильтры и подобные методы. Чтобы понимать *аппаратные средства*, мы должны сначала понять *алгоритмы*. В этом разделе мы будем делать детальный список из шагов, необходимых, чтобы осуществить КИХ-фильтр. В следующем разделе мы будем видеть, как ЦСП разработаны, чтобы исполнить эти шаги настолько эффективно насколько возможно.

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.

Чтобы начать, мы должны понимать различие между **автономной обработкой** и **обработкой в реальном времени**. В автономной обработке, *полный(целостный)* входной сигнал постоянно находится в компьютере в то же самое время. Например, геофизик мог бы использовать сейсмометр, чтобы делать запись наземного движения в течение землетрясения. После того, как колебание закончено, информация может читаться в компьютер, и проанализирована некоторым способом. Другой пример автономной обработки - медицинское отображение, типа вычисленной томографии и MRI. Набор данных приобретен, в то время как пациент - внутри машины, но реконструкция изображения может быть отсрочена до более позднего времени. Ключевой пункт - то, что *вся* информация является одновременно доступной программе обработки. Это обычно в научном исследовании и разработке, но не в изделиях(программах) потребителя. Автономная обработка - область персональных компьютеров и универсальных ЭВМ.

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, perform the algorithm, and output a sample, over-and-over. Alternatively, they may input a group of samples, perform the algorithm, and output a group of samples. This is the world of Digital Signal Processors.

В обработке в реальном времени, сигнал выхода произведен в то же самое время, что входной сигнал приобретается. Например, это необходимо в телефонной связи, слуховых аппаратах, и радаре. Эти приложения должны иметь информацию, немедленно доступную, хотя это может быть отсрочено коротким количеством. Например, задержка 10 миллисекунд в телефонном звонке не может быть обнаружена динамиком или слушателем. Аналогично, это не делает никакую разность, если радарный сигнал отсрочен к нескольким секундам перед отображением к оператору. Приложения анимации в реальном масштабе времени вводят выборку, исполняют алгоритм, и выводят выборку, "много раз". Альтернативно, они могут вводить группу выборок, исполнять алгоритм, и выводить группу выборок. Это - мир Цифровых Сигнальных Процессоров.

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 , …a_7. This means we must know the value of the eight most recent samples from the input signal, $x[n]$, $x[n - 1]$, … $x[n - 7]$. 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**.

Теперь оглянемся назад в рис. 28-2 и вообразим, что это - КИХ-фильтр, осуществляемый в анимации в реальном масштабе времени. Чтобы вычислять выборку выхода, мы должны иметь доступ к некоторому числу самых современных выборок от ввода. Например, предположите, что мы используем восемь коэффициентов в этом фильтре, a_0, a_1, … a_7. Это означает, что мы должны знать значение о восьми самых современных выборках от входного сигнала, $x[n]$, $x[n\text{-}1]$, … $x[n\text{-}7]$. Эти восемь выборок должны быть сохранены в памяти и непрерывно модифицированы, поскольку новые выборки приобретены. Какой лучший путь состоит в том, чтобы управлять этими сохраненными выборками? Ответ - кольцевая(циклическая) буферизация.

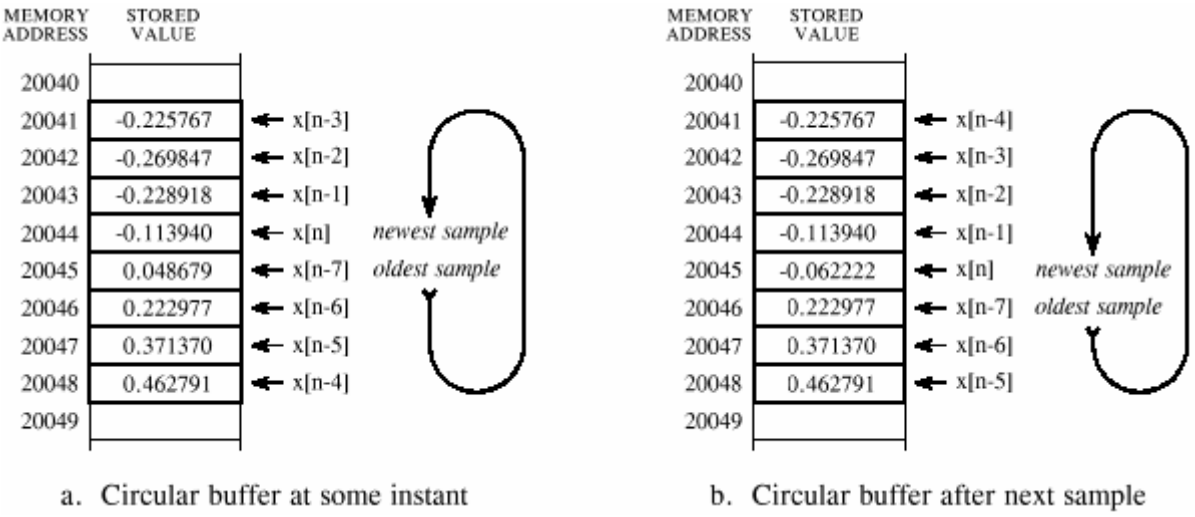


FIGURE 28-3
Circular buffer operation. Circular buffers are used to store the most recent values of a continually updated signal. This illustration shows how an eight sample circular buffer might appear at some instant in time (a), and how it would appear one sample later (b).

РИСУНОК 28-3
Кольцевая (Циклическая) буферная операция. Кольцевые буферы используются, чтобы сохранить самые современные значения непрерывно модифицируемого сигнала. Эта иллюстрация показывает, как восемь выборок циклического буфера могли бы появляться в некоторый момент во времени (a), и как это будет появляться на одну выборку позже (b).

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.

Рисунок 28-3 иллюстрирует восемь выборок кольцевого(циклического) буфера. Мы поместили этот круговой(циклический) буфер в восьми последовательных ячейках памяти, от 20041 до 20048. Рисунок (a) показывает, как восемь выборок от ввода могли бы быть сохранены в один специфический момент времени, в то время как (b) показывает изменения(замены) после того, как следующая выборка приобретена. Идея кольцевой буферизации состоит в том, что конец этого линейного массива связан с его началом; ячейка памя-

ти 20041 просмотрена как нахождение рядом с 20048, так же, как 20044 - рядом с 20045. Вы следите за указателем массива (переменная, чье значение - адрес) который указывает, где самая современная выборка постоянно находится. Например, в (a) указатель содержит адрес 20044, в то время как в (b) это содержит 20045. Когда новая выборка приобретена, это заменяет самую старую выборку в массиве, и указатель перемещен один адрес вперед. Кольцевые буферы эффективны, потому что только одно значение должно быть изменено(заменено), когда новая выборка приобретена.

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.

Четыре параметра необходимы, чтобы управлять кольцевым буфером. Во первых, должен иметься указатель, который указывает начало кольцевого буфера в памяти (в этом примере, 20041). Во вторых, должен иметься указатель, указывающий конец массива (например, 20048), или переменной, которая проводит(держит) ее длину (например, 8). Третье, размер шага адресования памяти должен быть определен. В рис. 28-3 размер шага *один*, например: адрес 20043 содержит одну выборку, адрес 20044 содержит следующую выборку, и так далее. Это - часто не случай(регистр). Например, адресование может обратиться(относиться) к байтам, и каждая выборка может требовать, чтобы два или четыре байта провели(держали) ее значение. В этих случаях, размер шага был бы должен быть два или четыре, соответственно.

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.

Эти три значения определяют размер и конфигурацию кольцевого буфера, и не будут изменяться в течение операции программы. Четвертое значение, указатель на самую современную выборку, должно измениться, поскольку каждая новая выборка приобретена. Другими словами, должна иметься логика программы, которая управляет, как это четвертое значение модифицировано основано на значении первых трех значений. В то время как эта логика весьма проста, это должно быть очень быстро. Это - целый пункт этого обсуждения; ЦСП должны быть оптимизированы в руководящих кольцевых буферах, чтобы достичь самого высокого возможного быстродействия выполнения.

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.

Как в стороне, кольцевая буферизация также полезна в *автономной обработке*. Рассмотрите программу, где и сигналы ввода и выхода полностью содержатся в памяти. Кольцевая буферизация не необходима для вычисления свертки, потому что к каждой выборке можно обращаться немедленно. Однако, много алгоритмов осуществлены постепенно, с промежуточным сигналом, создаваемым между каждой *стадией*. Например, рекурсивный фильтр, выполненный как ряд биквадратных операций таким образом. Метод решения "в лоб" состоит в том, чтобы сохранить полную длину каждого промежуточного сигнала в памяти. Кольцевой буфер обеспечивает другую опцию: сохраните только те промежуточные выборки, необходимые для вычисления под рукой. Это приводит требуемый объем памяти, за счет более сложного алгоритма. Важная идея - то кольцевые буферы, *полезны* для автономной обработки, но *критический* для приложений анимации в реальном масштабе времени.

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:

Теперь мы можем смотреть на шаги, необходимые, чтобы осуществить КИХ-фильр, используя кольцевые буферы, и для входного сигнала и коэффициентов. Этот список может казаться тривиальным и возбуждающим-, это не так! Эффективная обработка этих индивидуальных задач - то, что отделяет ЦСП от традиционного микропроцессора. Для каждой новой выборки, все следующие шаги должны быть приняты:

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

После перевода сделать картинку, там где перемычка с 12 на 6 поставить стрелки!

Таблица 28-1
FIR filter steps.
Шаги КИХ-фильтра.

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.

Цель состоит в том, чтобы заставить эти шаги выполниться быстро. Так как шаги 6-12 будут повторены много раз (однажды для каждого коэффициента в фильтре), специальное внимание нужно уделить этим операциям. Традиционные микропроцессоры должны вообще выполнять эти 14 шагов в *последовательный* (один за другим), в то время как цифровые микропроцессоры(ЦМП) разработаны(предназначены), чтобы исполнить их в *параллельном*. В некоторых случаях, все операции в пределах цикла (шаги 6-12) могут быть закончены в единственном тактовом цикле. Давайте рассмотрим на внутреннюю архитектуру, которая позволяет эту великолепную эффективность.

Architecture of the Digital Signal Processor
Архитектура Цифрового Сигнального процессора

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.

Одно из самых больших узких мест в выполняющихся алгоритмах ЦОС передает информацию к и от памяти. Это включает *данные*, типа выборок от входного сигнала и коэффициентов фильтра, также как *команд программы*, двоичные коды, которые входят в программу упорядочения программы. Например, предположите, что мы должны умножить два числа, которые постоянно находятся где-нибудь в памяти. Чтобы делать это, мы должны выбрать три двоичных значения от памяти - числа, которые будут умножены, плюс команда программы, описывающая, что делать.

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!

Рисунок 28-4a показывает, как эта по-видимому простая задача сделана в традиционном микропроцессоре. Это часто называется **Неймановой архитектурой**, после блестящего американского математика Джона Неймана (**Von Neumann**) (1903-1957). Нейман вел математику многих важных открытий раннего двадцатого столетия. Его многие достижения включают: разработка концепции сохраненного компьютера программы, формализация

математики квантовой механики, и работы по атомной бомбе. Если это было ново, и возбуждающе, Нейман был там!

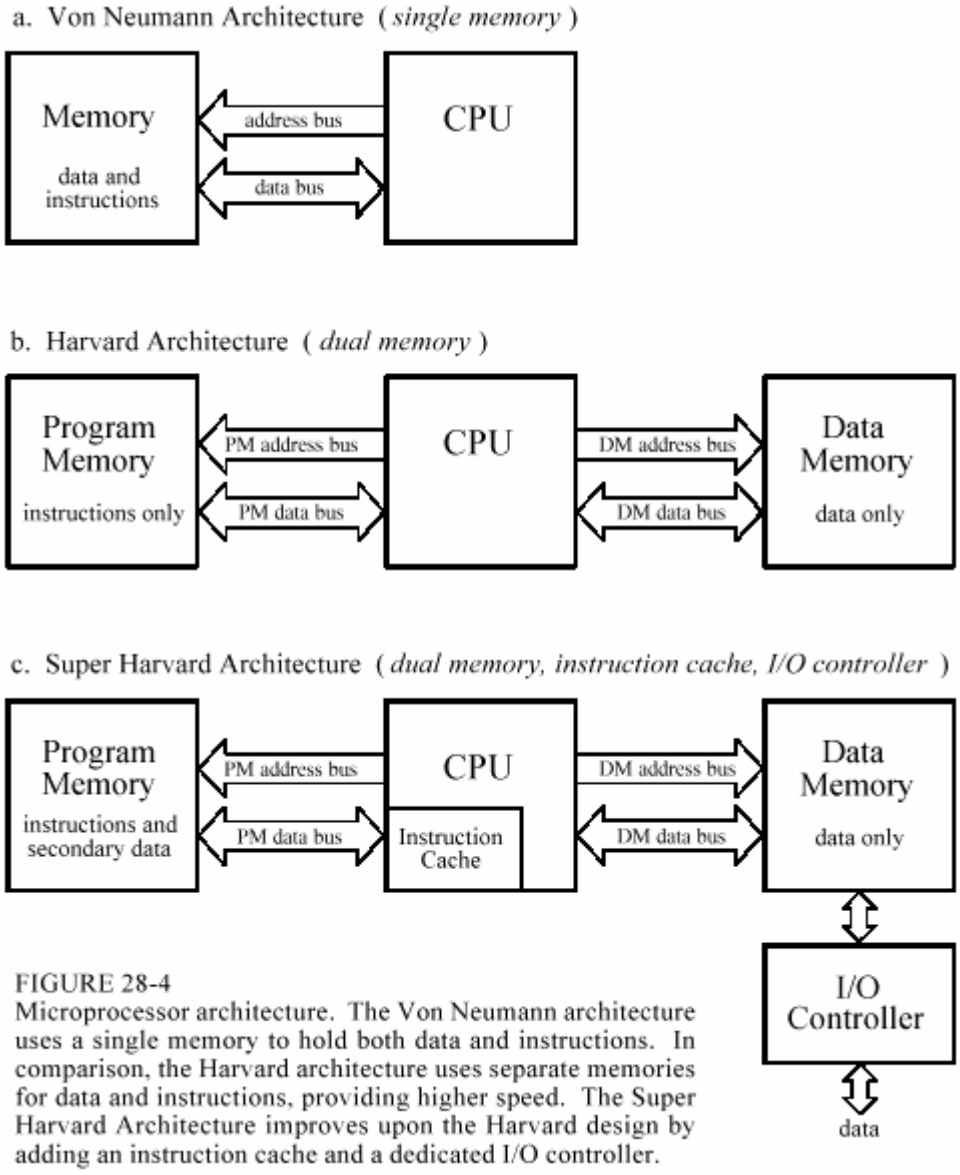


FIGURE 28-4
Microprocessor architecture. The Von Neumann architecture uses a single memory to hold both data and instructions. In comparison, the Harvard architecture uses separate memories for data and instructions, providing higher speed. The Super Harvard Architecture improves upon the Harvard design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

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.

РИСУНОК 28-4. Архитектура Микропроцессора.
Неймановa архитектура использует единственную память, чтобы провести(держать), и данные и команды. Для сравнения, использования архитектуры Гарварда отдельные блоки памяти для данных и команд, обеспечивая выше ускоряются. Архитектура Гарварда Высшего качества улучшается относительно проекта Гарварда, прибавляя кэш команды и специализированный контроллер Ввода - вывода.

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.

Как показано в (a), Нейманова архитектура содержит единственную память и единственную шину для передачи данных в и из центрального процессора (ЦПУ). Умножение двух чисел требует по крайней мере трех тактовых циклов, одного, чтобы передать(переместить) каждое из этих трех чисел по шине от памяти до ЦПУ. Мы не считаем время, чтобы передать(переместить) результат назад в память, потому что мы предполагаем, что это остается в ЦПУ для дополнительной манипуляции (типа суммы продуктов в КИХ-фильтре). Нейманов проект весьма удовлетворителен, когда Вы довольны, чтобы выполнить все требуемые задачи в последовательном. Фактически, большинство компьютеров сегодня имеет Нейманов проект. Мы только нуждаемся в другой архитектуре, когда очень быстро обработка требуется, и мы желаем оплачивать цену увеличенной сложности.

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.

Это ведет нас к **архитектуре Гарварда**, показанной в (b). Это названо по имени работы, сделанной в университете Гарварда в 1940-ых под руководством Говарда Аикена (1900-1973). Как показано в этой иллюстрации, Аикен настаивал на отдельных блоках памяти для данных и команд программы, с отдельными шинами для каждого. Так как шины оперируют независимо, программируют команды, и данные могут быть выбраны в то же самое время, улучшая быстродействие по единственному проекту шины. Большинство существующего дня ЦСП использует эту двойную архитектуру шины.

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*.

Рисунок (c) иллюстрирует следующий уровень сложности(изощренности), **Супер Архитектура Гарварда**(Высшего качества). Этот термин был сфабрикован(совпадает) Аналоговыми Устройствами, чтобы описать внутреннюю операцию их ADSP-2106-x и новых ADSP-211xx семейств Цифровых Сигнальных Процессоров. Они называются **SHARC®** DSPs (ЦСП), сокращение более длинного термина, Супер Архитектуры Гарварда. Идея состоит в том, чтобы формировать архитектуру Гарварда, прибавляя особенности, чтобы улучшить производительность. В то время как SHARC ЦСП оптимизированы способами путей, две области достаточно важны быть включенными в рис. 28-4c: кэш команды, и контроллер Ввода - вывода.

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 pro-

gram 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.

Во первых, давайте посмотрим, как кэш команды улучшает эффективность архитектуры Гарварда. Препятствие основного проекта Гарварда - то, что шина памяти данных является более занятой, чем шина памяти программы. Когда два числа умножены, два двоичных значения числа должны быть пропущены шиной памяти данных, в то время, как только одно двоичное значение (команда программы) пропускается по шине памяти программы. Чтобы улучшить эту ситуацию, мы запускаем(начинаем), перемещая часть "данных" к памяти программы. Например, мы могли бы размещать коэффициенты фильтра в память программы, при хранении входного сигнала в памяти данных. (Эти перемещенные данные в иллюстрации называются "вторичные данные"). На первый взгляд, это кажется, не помогает положению(ситуации); теперь мы должны передать(переместить) одно значение по шине памяти данных (выборка входного сигнала), но два значения по шине памяти программы (команда программы и коэффициент). Фактически, если бы мы выполняли случайные команды, это положение(ситуация) было бы вообще не лучше.

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*.

Однако, алгоритмы ЦОС вообще тратят большинство их времени выполнения в циклах, типа команд 6-12 в таблице 28-1. Это означает, что тот же самый набор команд программы будет непрерывно проходить от памяти программы до ЦПУ. Супер Архитектура Гарварда воспользуется преимуществом этого положения(ситуации) включением **команд кэша** в ЦПУ. Это - маленькая память, которая содержит приблизительно 32 из самых современных команд программы. Первый раз через цикл, команды программы должны быть пропущены шиной памяти программы. Это приводит к более медленной операции из-за конфликта с коэффициентами, которые должны также быть выбраны по этому пути. Однако, на дополнительных выполнении цикла, команды программы могут быть перемещены от кэша команды. Это означает, что вся память передачи информации ЦПУ может быть выполнена в единственном цикле: Выборка от входного сигнала спутник по шине памяти данных, коэффициент спутник по шине памяти программы, и команда программы спутник от кэша команды. На жаргоне поля, эта эффективная передача данных называется *шириной полосы частот доступа к области верхней памяти*.

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.

Рисунок 28-5 представляет более детальное представление SHARC архитектуры, показывая **контролер Ввода-Вывода** связанный с памятью данных. Это - то, как сигналы вводят и выходят из системы. Например, SHARC ЦСП обеспечивают, и последовательные и параллельные порты связи. Они - чрезвычайно высокоскоростные подключения(связи). Например, в 40 MHz тактовая частота, имеются два последовательных порта, которые оперируют в 40 Mbits/second каждый, в то время как шесть параллельных портов каждый обеспечивают передачу данных 40 Мегабайт в секунду. Когда все шесть параллельных портов используются вместе, скорость передачи данных – невероятна, 240 Мегабайт в секунду.

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!

Это достаточно быстро, чтобы передать текст целой этой книги всего за 2 миллисекунды! Также, как важный, специализированные(выделенные) аппаратные средства позволяют этим потокам данных быть переданными непосредственно в память (Прямой доступ к память или (DMA), без того, чтобы иметь необходимость проходить через регистры ЦПУ. Другими словами, задачи 1 и 14 на нашем списке случаются независимо и одновременно с другими задачами; никакие циклы не захвачены от ЦПУ. Основные шины (шина памяти программы и шина памяти данных) также доступны снаружи чипа, обеспечивая дополнительный интерфейс к памяти вне кристалла и периферийным устройствам. Это позволяет SHARC ЦСП использовать четыре Gigaword (16 Gbyte) память, доступную в 40 Mwords/second (160 Мегабайт в секунду), для данных 32 двоичных разрядов. Ничего себе!

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.

Этот тип высокоскоростного Ввода - вывода - ключевая характеристика ЦСП. Главная задача состоит в том, чтобы переместить данные в, выполнить математику, и переместить данные из, прежде, чем следующая выборка доступна. Все остальное вторично. Некоторые ЦСП имеют автономные аналого-цифровые и цифро-аналоговые преобразователи, особенность называемая **смешанным сигналом**. Однако, все ЦСП могут связываться с помощью интерфейса со внешними конвертерами через последовательные или параллельные порты.

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.

Теперь давайте заглянем внутрь ЦПУ. Наверху диаграммы - два блока помеченные **Генератор Адреса Данных** (DAG), один для каждого из этих двух блоков памяти(памяти программы(PM) и памяти данных(DM)). Они управляют адресами, посланными программе и блокам памяти данных, определяя, где информация должна читаться от или записана в. В более простых микропроцессорах эта задача обработана как собственная часть программы установления последовательности выполнения программы и весьма прозрачна для программиста. Однако, ЦПС разработаны(предназначены), чтобы оперировать с кольцевыми буферами, и выгодой от дополнительных аппаратных средств, чтобы управлять ими эффективно. Это избегает нуждаться использовать драгоценные тактовые циклы ЦПУ, чтобы следить, как данные сохранены. Например, в SHARC ЦСП, каждый из двух DAGs может управлять восьмью кольцевыми буферами. Это означает, что каждый DAG проводит(держит) 32 переменных (4 в буфере), плюс требуемой логикой.

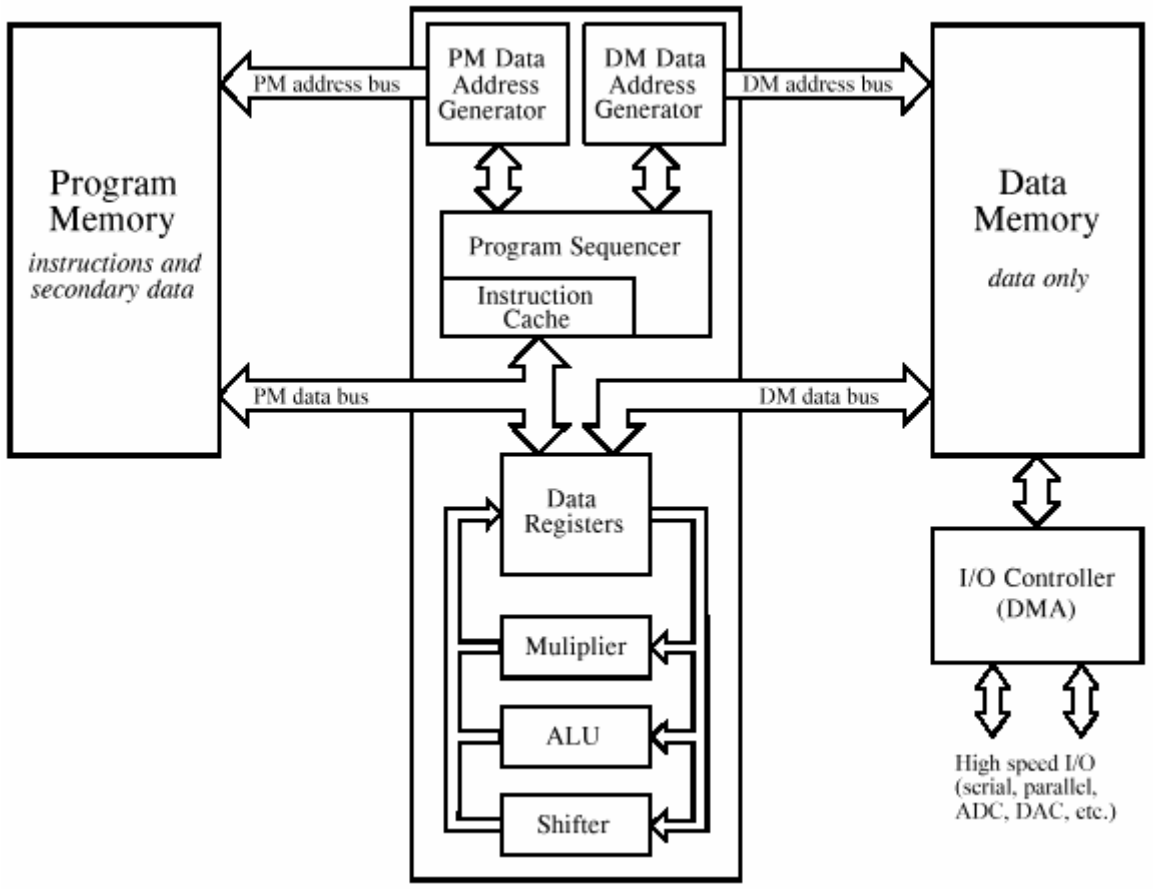


FIGURE 28-5
Typical DSP architecture. Digital Signal Processors are designed to implement tasks in parallel. This simplified diagram is of the Analog Devices SHARC DSP. Compare this architecture with the tasks needed to implement an FIR filter, as listed in Table 28-1. All of the steps within the loop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

РИСУНОК 28-5
Типичная архитектура ЦСП. Цифровой сигнальный процессор разработан(предназначен), чтобы осуществить задачи в параллельном. Эта упрощенная диаграмма имеет Аналоговые Устройства SHARC ЦСП. Сравните эту архитектуру с задачами, необходимыми, чтобы осуществить КИХ-фильтр, как перечислено в таблице 28-1. Все шаги в пределах цикла могут быть выполнены в единственном тактовом цикле.

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.

Почему так много кольцевых буферов? Некоторые алгоритмы ЦОС лучшие выполнены постепенно. Например, БИХ-фильтры более устойчивы, если осуществлено как биквадратный(биквад) каскад (стадия(каскад), содержащая два полюса и до двух нулей). Множественные стадии требуют множественных кольцевых буферов для самой быстрой операции. DAGs в SHARC ЦСG также предназначены, чтобы эффективно выполнить Быструю трансформанту Фурье. В этом режиме, DAGs конфигурированы, чтобы генерировать **двоичные перевернутые адреса** в кольцевых буферах, необходимая часть алгоритма БПФ. Кроме того, распространенность кольцевых буферов очень упрощает генерацию объектного кода ЦОС - оба для человеческого программиста также как интенсивных компиляторов языка, типа С.

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.

Раздел регистра данных ЦПУ используется таким же образом как в традиционных микропроцессорах. В ADSP-2106x SHARC ЦСП-ов, имеются 16 универсальных регистраторов 40 битов каждый. Они могут проводить(держать) промежуточные вычисления, готовить данные для математического процессора, служить как буфер для передачи данных, флажки хранения для программного управления, и так далее. Если необходимо, эти регистраторы могут также использоваться, чтобы управлять циклами и счетчиками; однако, SHARC ЦСП имеют дополнительные аппаратные регистры, чтобы выполнить многие из этих функций.

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.

Математическая обработка разбита на три раздела, **умножитель**, **арифметико-логическое устройство (АЛУ)**, и **многорегистровая схема циклического сдвига**. Множитель берет значения от двух регистров, умножает их, и размещает результат в другой регистр. АЛУ исполняет сложение, вычитание, абсолютное значение, логические операции (AND, OR, XOR, NOT), преобразование между установленными и форматами плавающей запятой, и подобными функциями. Элементарные двоичные операции выполнены многорегистровой схемой циклического сдвига, типа смещения, вращения, извлечения и внесения сегментов, и так далее. Мощная особенность семейства SHARC - то, что к множителю и АЛУ можно обращаться параллельно. В единственном тактовом цикле, данные от регистров 0-7 можно пропускать к множителю, данные от регистров 8-15 можно пропускать к АЛУ, и двум результатам, возвращенным любому из 16 регистров.

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 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.

Имеются также много важных особенностей семейства архитектуры SHARC, которые не показываются в этой упрощенной иллюстрации. Например, сумматор 80 двоичных разрядов сформирован в множитель, чтобы привести ошибку округления, связанную с множественными математическими операциями с фиксированной точкой. Другая интересная особенность - использование **теневых регистров** для всех ключевых регистров ЦПУ. Они - дубликаты регистров, которые могут быть переключены с их дубликатами в единственном тактовом цикле. Они используются для *быстрого переключения контекста*, способность обработать прерывания быстро. Когда прерывание происходит в традиционных микропроцессорах, все внутренние данные должны быть сохранены прежде, чем прерывание может быть обработано. Это обычно включает в себя выталкивание всех занятых регистров на стек, по одному. Для сравнении, прерывание в семействе SHARC обработано, перемещая внутренние данные в теневых регистрах в *единственном тактовом цикле*. Когда подпрограмма прерывания закончена, регистры так же, как быстро восстановлены. Эта особенность позволяет шагу 4 в нашем списке (управление sample-ready прерыванием) быть обработанным очень быстро и эффективно.

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.

Теперь мы прибываем в критическую эффективность архитектуры, сколько из операций в пределах цикла (шаги 6-12 таблицы 28-1) может быть выполнено одновременно. Из-за его высоко параллельного характера(природы), SHARC ЦСП может одновременно выполнять все эти задачи. Определенно, в пределах единственного тактового цикла, это может исполнять умножающийся (шаг 11), сложение (шаг 12), два шага данных (шаги 7 и 9), модифицировать два кольцевых буферных указателя (шаги 8 и 10), и управлять циклом (шаг 6). Будут иметься дополнительные тактовые циклы, связанные с началом и окончанием цикла (шаги 3, 4, 5 и 13, плюс, перемещение первоначальных значений на место); однако, эти задачи также обработаны очень эффективно. Если цикл выполнен, больше чем несколько раз, этот верхний будут незначительны. Как пример, предположите, что Вы записываете эффективную программу КИХ фильтра, используя 100 коэффициентов. Вы можете ожидать, что это будет требовать выполнения приблизительно от 105 до 110 тактовых циклов на выборку (то есть, 100 коэффициентов циклов плюс служебные). Это очень внушительно; традиционный микропроцессор требует многих тысяч тактовых циклов для этого алгоритма.

Fixed versus Floating Point
Фиксированный против Плавающей Запятой

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.

Цифровая Обработка сигналов может быть разделена на две категории, фиксированной точкой и с плавающей запятой. Они обращаются(относятся) к формату, имел обыкновение сохранять и управлять числами в пределах устройств. Фиксированная точка ЦОС обычно представляют каждое число минимумом 16-ю битами, хотя различная длина может использоваться. Например, Фирма Motorola производит семейство ЦСП с фиксированной точкой используя 24 бита. Имеются четыре обычных пути, которыми эти 2^{16} = 65,536 возможных двоичных кодов(битовых комбинаций) могут представлять число. В **целом числе без знака**, сохраненное число может принимать любое целочисленное значение от 0 до 65,535. Точно так же **целое число со знаком** использует дополнение до двух, чтобы заставить диапазон включить отрицательные числа, от -32,768 до 32,767. С системой обозначений **дроби без знака**, 65,536 уровней распространены равномерно между 0 и 1. Наконец, формат **подписанной дроби** позволяет отрицательные числа, одинаково раздельные между -1 и 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.

Для сравнения, плавающая запятая ЦСП типично используют минимум 32 бита, чтобы сохранить каждое значение. Это приводит намного больше к двоичным кодам(битовым комбинациям) чем для фиксированной точки, 2^{32} = 4,294,967,296 быть точным. Главная особенность системы обозначений плавающей запятой - то, что представленные числа равномерно не располагаются. В наиболее обычном формате (ANSI/IEEE Std. 754-1985), самые большие и самые маленькие числа являются $\pm 3.4 \times 10^{38}$ и $\pm 1.2 \times 10^{-38}$ соответственно. Представленные значения неравноценно раздельные между этими двумя крайностями, такой, что промежуток между любыми двумя числами является в десять миллионов раз меньше чем значение чисел. Это важно, потому что это размещает большие промежутки между большими числами, но маленькими промежутками между маленькими числами. Система обозначений плавающей запятой обсуждена более подробно в главе 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."

Все ЦСП с плавающей запятой могутт также обрабатывать числа с фиксированной точкой, потребность, чтобы осуществить счетчики, циклы, и сигналы, исходящие из АЦП и идущие на ЦАП. Однако, это не подразумевает, что математика фиксированной точки будет выполнена так быстро как операции с плавающей запятой; это зависит от внутренней архитектуры. Например, SHARC ЦСП оптимизированы и для операций с плавающей запятой и операций с фиксированной точкой, и выполняет их с равной эффективностью. По этой причине, SHARC устройства часто упоминаются как " 32-разрядные ЦСП, " скорее чем только " Плавающая Запятая. "

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 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.

Рисунок 28-6 иллюстрирует первичные компромиссные решения между ЦСП с фиксированной точкой и с плавающей запятой. В главе 3 мы подчеркнули, что арифметика фиксированной точки - намного быстрее, чем плавающая запятая в компьютерах общего назначения. Однако, с ЦСП быстродействие - относительно тот же самый, результат аппаратных средств, высоко оптимизируемых для математических операций. Внутренняя архитектура плавающей запятая ЦСП больше усложнена, чем для устройств с фиксированной точкой. Все регистры и шины данных должны быть шириной 32 бита вместо только 16;

множитель и АЛУ должны быть способны быстро исполнить арифметику плавающей запятой, система команд должна быть большая (так, чтобы они могли обрабатывать числа и с плавающей запятой и с фиксированной точкой), и так далее. Плавающая запятая (32 двоичных разряда) имеет лучшую прецизионность и более высокий динамический диапазон чем фиксированная точка (16 двоичных разрядов). Кроме того, программы плавающей запятой часто имеют более короткий период развития, так как программист вообще не должен волноваться относительно проблем типа переполнения, антипереполнения, и ошибки округления.

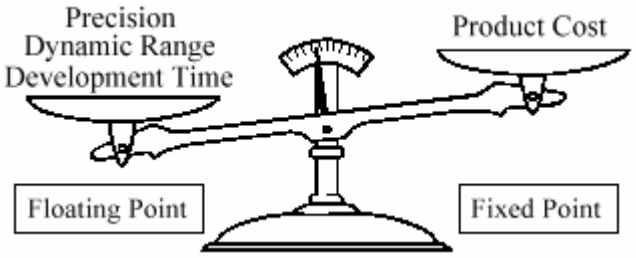


FIGURE 28-6
Fixed versus floating point. Fixed point DSPs are generally cheaper, while floating point devices have better precision, higher dynamic range, and a shorter development cycle.
РИСУНОК 28-6
Фиксированный против плавающей запятой. ЦСП с фиксированной точкой являются вообще более дешевыми, в то время как устройства с плавающей запятой имеют лучшую прецизионность, более высокий динамический диапазон, и более короткий цикл(период) развития.

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 1999, 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*.

С другой стороны, ЦСП с фиксированной точкой традиционно были более дешевые, чем устройства с плавающей запятой. Ничто не изменяется более быстро, чем цена электроники; что – ни будь, что Вы находите в книге, устареет прежде, чем это напечатано. Однако, стоимость - главный критерий в понимании, как ЦСП развиваются, и мы должны дать Вам общую идею. Когда эта книга была закончена в 1999, ЦСП с фиксированной точкой, проданные за между $ 5 и $ 100, в то время как устройства с плавающей запятой были в диапазоне $ 10 до $ 300. Эта разность в стоимости может быть просмотрена как мера относительной сложности между устройствами. Если Вы хотите выяснить то, чем цены являются *сегодня*, Вы должны смотреть *сегодня*.

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.

Теперь давайте повернем наше внимание к *эффективности*; что может делать 32-разрядная система с плавающей запятой, а 16-разрядная система с фиксированной точкой этого не делает? Ответ на этот вопрос - **отношение сигнал-шум**. Предположим, что мы сохраняем число в формате с плавающей запятой 32 двоичными разрядами. Как предварительно упомянуто, промежуток между этим числом и его смежным соседом приблизительно одна десяти миллионная из значения числа. Чтобы сохранить число, это должна быть округление в большую сторону или в меньшую сторону максимумом половины раз-

мера промежутка. Другими словами, каждый раз, как мы сохраняем число в системе записи с плавающей запятой, мы прибавляем *шум* к сигналу.

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.

Та же самая вещь случается, когда число сохранено как 16-разрядное значение с фиксированной точкой, за исключением того, что добавленный шум - намного хуже. Это - то, потому что промежутки между смежными числами намного большие. Например, предположите, что мы сохраняем число 10000 как целое число со знаком (выполняющееся от -32,768 до 32,767). Промежуток между числами - одна десятитысячная значения числа, которое мы сохраняем. Если мы хотим сохранить число 1000, промежуток между числами - только 1 (единица) одна тысячная значения.

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.

Шум в сигналах обычно представляется его среднеквадратичным отклонением. Это было обсуждено подробно в главе 2. Для здесь, важный факт то, что среднеквадратичное отклонение этого **шума квантования** является приблизительно одной третьей частью размера промежутка. Это означает, что отношение сигнал-шум для сохранения числа с плавающей запятой - приблизительно 30 миллионов к одному, в то время как для числа с фиксированной точкой это - только десять тысяч к одному. Другими словами, плавающая запятая имеет грубо в 30000 раз меньшее количество шума квантования чем фиксированная точка.

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.

Это поднимает важный путь, которым ЦСП отличаются от традиционных микропроцессоров. Предположите, что мы осуществляем КИХ-фильтр с фиксированной точкой. Чтобы делать это, мы, цикл через каждый коэффициент, умножает это на соответствующую выборку от входного сигнала, и прибавляет продукт к сумматору. Имеется проблема. В традиционных микропроцессорах, этот сумматор - только переменная фиксированной точка другой 16 двоичных разрядов. Чтобы избегать переполнения, мы должны масштабировать

добавляемые значения, и соответственно прибавим шум квантования на каждом шаге. В самом плохом случае, этот шум квантования просто добавится, очень понижая отношение сигнал-шум системы. Например, в 500 коэффициенте КИХ-фильтр, шум на каждой выборке выхода может быть в 500 раз выше шума на каждой входной выборке. Отношение сигнал-шум *десять тысяч к одному(единице)* понизился к ужасному *двадцать к одному*. Хотя это - критический случай, это иллюстрирует основной пункт: Когда много операций выполнено на каждой выборке, это плохо, действительно плохо. См. главу 3 для большего количества подробностей.

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.

ЦСП обрабатывают эту проблему, используя расширенный сумматор прецизионности. Это - специальный регистр, который имеет 2-3 раза так много битов как другие ячейки памяти. Например, в ЦСП 16 двоичных разрядов это может иметь 32 до 40 битов, в то время как в SHARC ЦСП это содержит 80 битов для использования фиксированной точки. Этот расширенный диапазон фактически устраняет шум округления, в то время как накопление происходит. Единственная перенесенная ошибка округления - то, когда сумматор масштабируется и сохранен в памяти 16 двоичных разрядов. Эта стратегия работает очень хорошо, хотя это ограничивает, как некоторые алгоритмы должны быть выполнены. Для сравнения, плавающая запятая имеет такой низкий шум квантования, что эти методы являются обычно не необходимыми.

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).

В дополнение к наличию более низкого шума квантования, системы с плавающей запятой также проще, чтобы разработать алгоритмы для. Большинство методов ЦОС основано на повторных умножениях и сложениях. В фиксированной точке, возможность переполнения или антипереполнения должна рассмотреться после каждой операции. Программист должен непрерывно понять амплитуду чисел, как ошибки квантования накапливаются, и какие масштабирующие потребности иметь место. На сравнении, эти проблемы не возникают в плавающей запятой; числа заботятся о себе (кроме редких случаев).

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 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.

Чтобы дать Вам лучшее понимание этой проблемы, рис. 28-7 показывает таблицу от SHARC руководства по эксплуатации. Это описывает пути, которыми умножение может быть выполнено для и для фиксированных форматов и форматов с плавающей запятой. Во первых, смотрите, как числа с плавающей запятой могут быть умножены; имеется только один путь! То есть Fn = Fx * Fy, где Fn, Fx, и Fy - любой из этих 16 регистров данных. Это не могло быть любой более простое. Для сравнения, смотрите на все возможные команды для умножения чисел с фиксированной точкой. Они - многие параметры, необходимые, чтобы эффективно обработать проблемы округления, масштабирования, и формата.

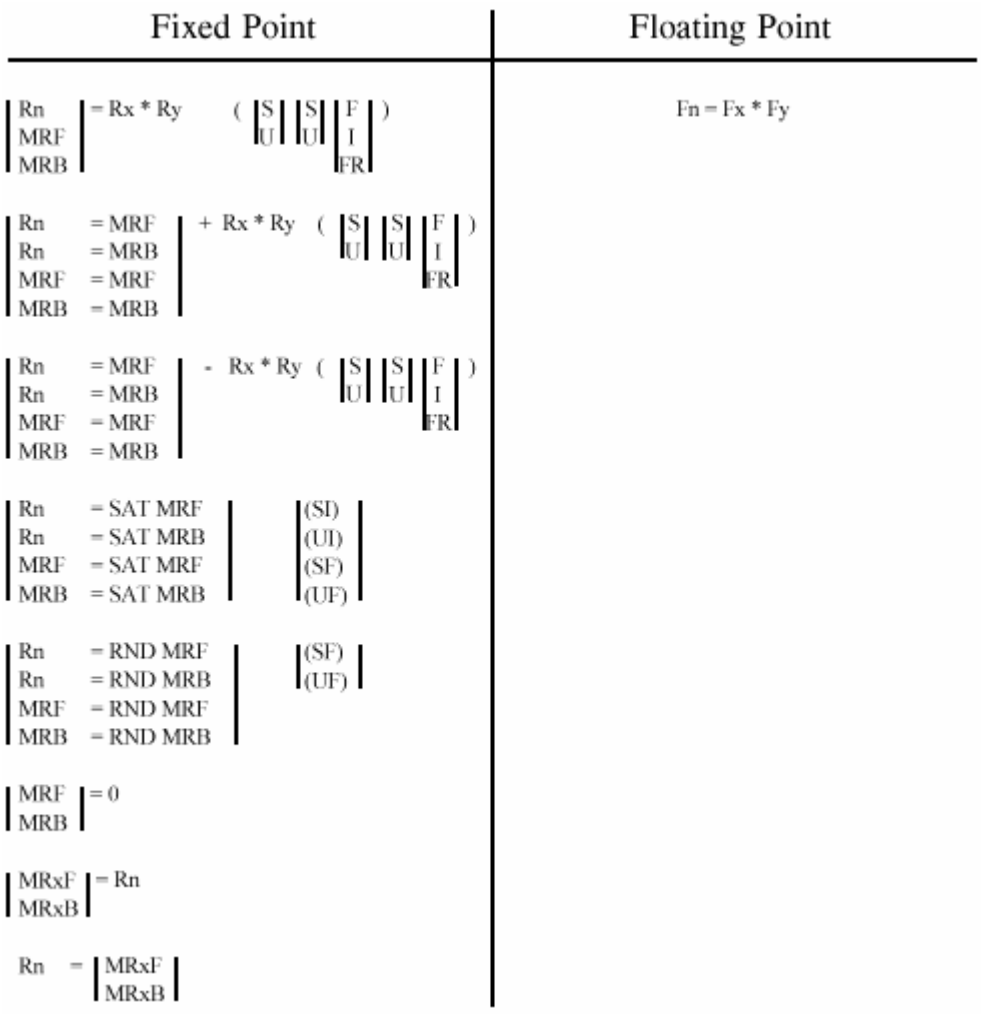


FIGURE 28-7
Fixed versus floating point instructions. These are the multiplication instructions used in the SHARC DSPs. While only a single command is needed for floating point, many options are needed for fixed point. See the text for an explanation of these options.
РИСУНОК 28-7
Фиксированный против команд с плавающей запятой. Они - команды умножения, используемые в SHARC ЦСП. В то время как только единственная команда необходима для плавающей запятой, много параметров необходимы для фиксированной точки. См. текст для объяснения этих параметров.

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.

В рис. 28-7, Rn, Rx, и Ry относятся к любому из этих 16 регистров данных, и MRF, и MRB - сумматоры 80 двоичных разрядов. Вертикальные строки указывают параметры. Например, верхний левый вход в этой таблице означает, что все следующее имеет силу команды: Rn = Rx * Ry, MRF = Rx * Ry, и MRB = Rx * Ry. Другими словами, значение из любых двух регистров может быть умножено и помещено в другой регистр, или в один из расширенных сумматоров прецизионности. Эта таблица также показывает, что числа могут быть или подписаны или без знака (S или U), и могут быть дробны или целочисленные (F или I). RND и SAT, параметры - пути управления переполнением регистра и округлением.

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.

Имеются другие подробности и параметры в таблице, но они не важны для нашего существующего обсуждения. Важная идея состоит в том, что программист фиксированной точки должен понять множества способов выполнить самую основную задачу умножения. Напротив, программист плавающей запятой может тратить свое время, концентрируясь на алгоритме.

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.

Эти данные сделки между фиксированной и плавающей запятой, как Вы выбираете, чтобы использовать? Имеются некоторые вещи рассмотреть. Во первых, смотрите, сколько битов используются в АЦП и ЦАП. В многих приложениях, 12-14 бит в секунду - пересечение для использования фиксированного против плавающей запятой. Для образца, телевидение и другие видеосигналы типично использует 8 разрядные АЦП и ЦАП и прецизионность фиксированной точки приемлема. Для сравнения, профессиональные звуковые приложения могут производить выборку с столь же высоко как 20 или 24 бита, и почти конечно нуждаться в плавающей запятой, чтобы фиксировать большой динамический диапазон.

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.

Следующая вещь смотреть - сложность алгоритма, который будет выполнен. Если это относительно просто, думайте фиксированная точка; если это больше усложнено, думайте плавающая запятая. Например, КИХ-фильтрация и другие операции в домене времени требует только нескольких строк программы, делая их подходящими для фиксированной точки. Напротив, алгоритмы частотного домена, типа спектрального анализа и свертки БПФ, очень детализированы и могут быть намного более трудны программировать. В то

время как они могут быть написаны в фиксированной точке, время на разработку будет очень сокращено, если плавающая запятая используется.

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.

Наконец, думайте относительно денег: насколько важна - стоимость изделия, и насколько важна - стоимость развития? Когда фиксированная точка выбрана, стоимость изделия будет сокращена, но стоимость развития будет вероятно выше из-за более трудных алгоритмов. Обратным способом, плавающая запятая будет вообще приводить к более быстрому и более дешевому периоду развития, но более дорогому конечному продукту.

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.

Рисунок 28-8 показывает некоторые из главных тенденций в ЦСП. Рисунок (a) иллюстрирует влияние, которое Цифровые сигнальные процессоры имели на *внедренном* рынке. Они - приложения, которые используют микропроцессор, чтобы непосредственно оперировать и управлять несколько большей системой, типа ячеистого телефона, микроволновой печи, или автомобильной инструментальной панели дисплея. Название "микроконтроллер" часто используется в что касается этих устройств, отличайте их от микропроцессоров, используемых в персональных компьютерах. Как показано в (a), приблизительно 38% внутренних проектировщиков уже начали использовать ЦСП, и другие 49 % рассматривают переключение на них. Высокая производительность и вычислительная мощь ЦСП часто делают их идеальным выбором для внедренных проектов.

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 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.

Столь же иллюстрировано в (b), вдвое больше инженеров в настоящее время используют ЦСП с фиксированной точкой, чем инженеров использующих ЦСП с плавающей запятой. Однако, это зависит очень от приложения. Фиксированная точка более популярна в конкурентоспособных изделиях потребителя, где стоимость электроники должна сохраниться

очень низкой. Хороший пример этого - ячеистые телефоны. Когда Вы находитесь на соревновании, чтобы продать миллионы вашего изделия(программы), разность стоимости, только несколько долларов могут быть разностью между успехом и неудачей. Для сравнения, плавающая запятая более обычна, когда большая эффективность необходима, и стоимость не важна. Например, предположите, что Вы разрабатываете медицинскую систему отображения, такую как сканер компъютерной томографии. Только несколько сотен моделей будет когда-либо продано, по цене нескольких сотен тысяч долларов за каждый. Для этого приложения, стоимость ЦСП незначащая, но эффективность критическая. Несмотря на больший число используемых ЦСП с фиксированной точкой, рынок плавающей запятой - самый быстрый сегмент роста. Как показано в (c), до половины инженеров, использующих 16-разрядные устройства планируют перейти к плавающей запятой в ближайшем будущем.

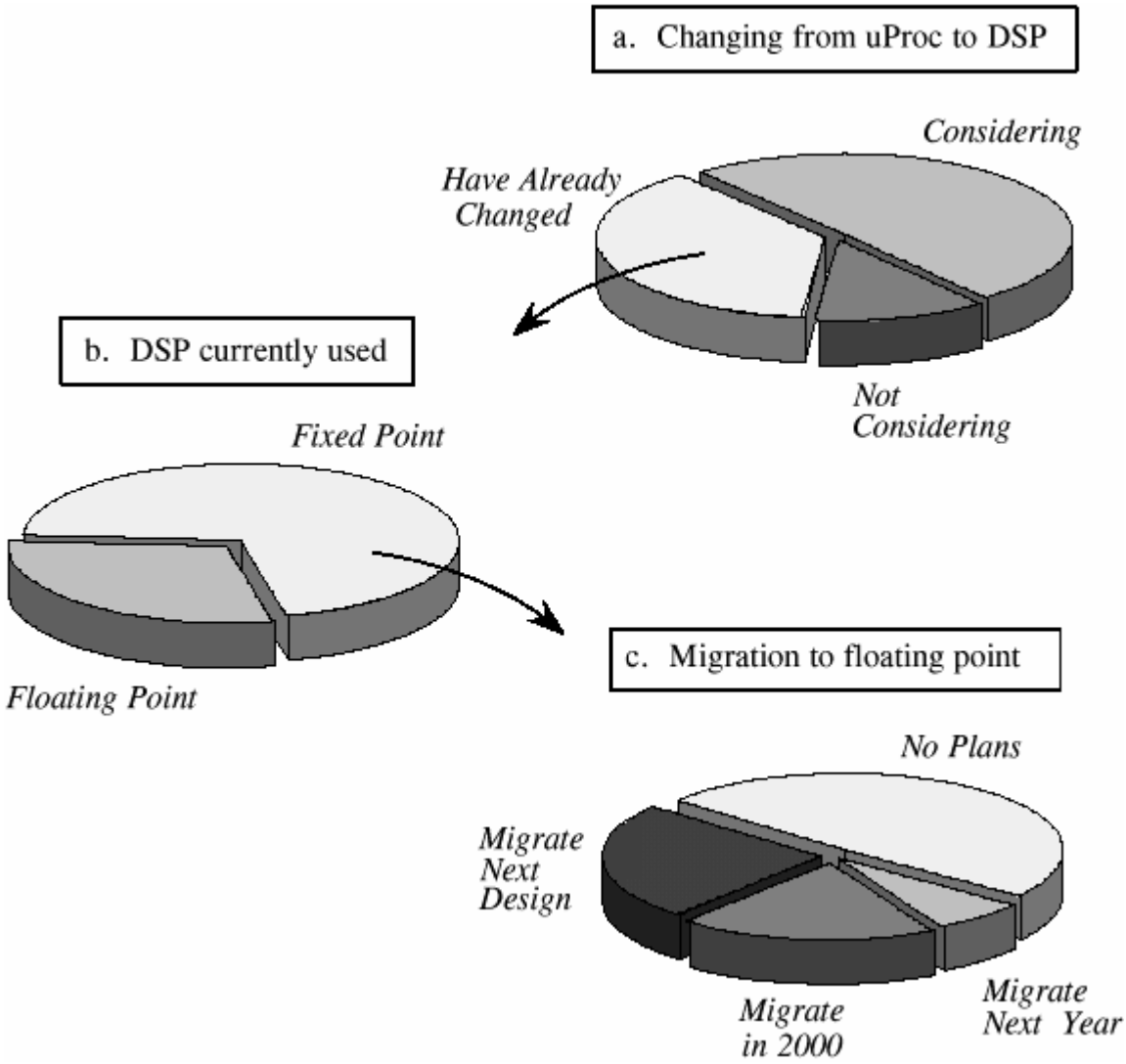


FIGURE 28-8
Major trends in DSPs. As illustrated in (a), about 38% of embedded designers have already switched from conventional microprocessors to DSPs, and another 49% are considering the change. In (b), about twice as many engineers use fixed point as use floating point DSPs. This is mainly driven 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

РИСУНОК 28-8
Главные тенденции в ЦСП. Как иллюстрировано в (a), приблизительно 38 % внедренных проектировщиков уже заменили обычные микропроцессоры на ЦСП, и другие 49 % рассматривает замену. В (b), вдвое больше инженеров используют ЦСП с фиксированной точкой, чем инженеров, которые использует ЦСП с плаваю-

щей запятой. Это главным образом управляется изделиями потребителя, которые должны иметь дешевую электронику, типа ячеистых телефонов. Однако, как показано в (c), плавающая запятая - самый быстрый сегмент роста; половина инженеров, в настоящее время использующих 16-разрядные устройства планируют перейти к ЦСП плавающей запятой

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.

Перед оставлением этой темы, мы должны повторно подчеркнуть, что плавающая запятая и фиксированная точка обычно используют 32 бита и 16 битов, соответственно, *но не всегда*. Например, SHARC семейство может представлять числа в режиме 32-разрядной фиксированной точки, режим, который является обычным в цифровых звуковых приложениях. Это делает 2^{32} уровня квантования раздельными равномерно по относительно узкому диапазону, скажем, между -1 и 1. Для сравнения, система с плавающей запятой размещает 2^{32} уровня квантования, логарифмически по огромному диапазону, типично $\pm 3.4 \times 10^{38}$. Это дает 32-разрядной фиксированной точке, *лучшую* прецизионность, то есть ошибка квантования на любой выборке будет более низкая. Однако, 32-разрядная плавающая запятая имеет более высокий *динамический диапазон*, что означает что имеется большая разность между самым большим числом и самым маленьким числом, которое может быть представлено.

C versus Assembly
Си против Трансляции(ассемблирования)

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.

ЦСП запрограммированы на тех же самых языках как другие научные и технические приложения, обычно ассемблер или СИ. Программы, написанные на ассемблере, могут выполняться быстрее, в то время как программы, написанные в СИ проще, чтобы разрабатывать и обслужить. В традиционных приложениях, типа программ, выполненных на персональных компьютерах и универсальных ЭВМ, СИ - почти всегда первый выбор. Если ассемблирование используется вообще, это ограничено короткими подпрограммам, которые должны работать с предельным быстродействием. Это показано графически на рис. 28-9a; на каждого программиста, который традиционно работает на ассемблировании, имеется приблизительно десять тех, кто использует СИ.

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 soft-

ware 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.

Однако, программы ЦОС отличаются от традиционных программных задач в двух важных отношениях. Во первых, программы - обычно намного короче, скажем, строки с одной сотней против строк с десятью тысячами. Во вторых, быстродействие выполнения - часто критическая часть приложения. В конце концов, именно поэтому кто - то использует ЦСП во-первых, для его ослепляющего быстродействия. Эти два фактора мотивируют много программных инженеров, чтобы переключить СИ на ассемблирование для программирования Цифровых Сигнальных Процессоров. Это иллюстрировано в (b); почти так много программистов ЦОС использует ассемблирование, как используют СИ.

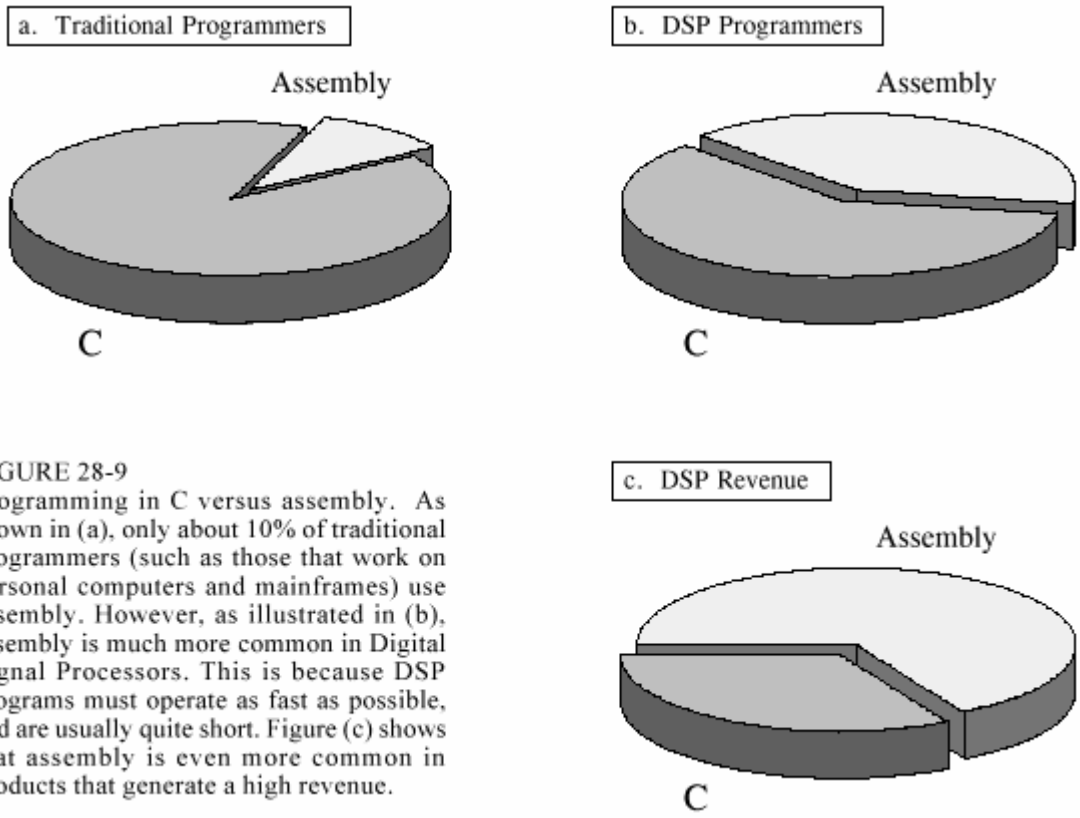


FIGURE 28-9
Programming in C versus assembly. As shown in (a), only about 10% of traditional programmers (such as those that work on personal computers and mainframes) use assembly. However, as illustrated in (b), assembly is much more common in Digital Signal Processors. This is because DSP programs 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.

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.

РИСУНОК 28-9
Программирование в С против трансляции(ассемблирования). Как показано в (a), только приблизительно 10 % традиционных программистов (типа тех, что работа на персональных компьютерах и универсальных ЭВМ) использует трансляцию(ассемблирование). Однако, как иллюстрировано в (b), трансляция(ассемблирование) намного более обычна в Цифровоях сигнальный процессор. Это - то, потому что программы DSP должны оперировать с такой скоростью как возможный, и обычно весьма коротки. Рисунок (c) показывает, что трансляция(ассемблирование) четная более обычна в изделиях(программах), которые генерируют высокий доход.

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 chang-

ing. 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.

Рисунок (c) берет этот дальнейший смотрящий на доход, произведенный программами ЦОС. Для каждого доллара, сделанного с ЦОС, программируя на СИ, два доллара сделаны с ЦОС, программированием на ассемблере. Причина для этого проста; деньги сделаны, превосходя по быстродействию соревнование. С чистой точки зрения эффективности, типа быстродействия выполнения и производящий стоимость, ассемблирование почти всегда имеет преимущество перед СИ. Например, код Си обычно требует большей памяти чем ассемблирование, приводя к более дорогим аппаратным средствам. Однако, рынок ЦОС непрерывно изменяется. Поскольку рынок растет, изготовители ответят, разрабатывая ЦСП, которые оптимизированы для программирования в СИ. Например, Си намного более эффективен, когда имеется большой, универсальный набор регистров и объединенное пространство памяти. Эти будущие уточнения минимизируют разность во времени выполнения между СИ и ассемблированием, и позволят СИ использоваться в большем количестве приложений.

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 array by the corresponding coefficient in the other array, and sum the products, i.e. *x*[0] x *y*[0] + *x*[1] x *y*[1] + *x*[2] x *y*[2] + …. 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.

Чтобы лучше понимать это решение между СИ и ассемблированием, давайте рассмотрим типичную задачу ЦОС, запрограммированную на каждом языке. Пример, который мы будем использовать - вычисление *скалярного произведения* двух массивов, *x*[ ] и *y*[ ]. Это - простая математическая операция, мы умножаем каждый коэффициент в одном массиве на соответствующий коэффициентом в другом массиве, и суммируем произведения, то есть *x*[0] x *y*[0] + *x*[1] x *y*[1] + *x*[2] x *y*[2] + … . Это должно выглядеть очень знакомым; это - фундаментальная операция в программе фильтрации с КИХ. То есть каждая выборка в сигнале выхода найдена, умножая сохраненные выборки от входного сигнала (в одном массиве) коэффициентами фильтра (в другом массиве), и подводя сумму произведений.

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 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.

Таблица 28-2 показывает, как скалярное произведение рассчитано в программе СИ. В строках 001-004 мы определяем два массива, *x*[ ] и *y*[ ], по 20 элементов длинной. Мы также определяем переменную *result*, которая содержит расчетное скалярное произведение при завершении программы. Строки 011 управляют этими 20 циклами необходимые для вычисления, используя переменную *n* как счетчик цикла. Единственная инструкция в пре-

делах цикла - строка 012, которая умножает соответствующие коэффициенты от двух массивов, и прибавляет произведение к переменной сумматора, *s*. (Если Вы не знакомы с СИ, инструкция: *s* + = *x*[*n*] * *y*[*n*] означает тот же самое как: *s* = *s* + *x*[*n*] * *y*[*n*]). После цикла, значение в сумматоре, *s*, передано переменной *result*, в строке 013.

.
ТАБЛИЦА 28-2 Скалярное произведение в СИ. Эта программа вычисляет скалярное произведение двух массивов, *x*[ ] и *y*[ ], и сохраняет результат в переменной, *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  }
```

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.

Ключевое преимущество использования языков высокого уровня (типа СИ, ФОРТРАН, или БЭЙСИК) состоит в том, что программист не должен понимать архитектуру используемого микропроцессора; знание архитектуры оставлено компилятору. Например, эта короткая программа СИ использует несколько переменных: *n*, *s*, *result*, плюс массивы: *x*[ ] и *y*[ ]. Все эти переменные должны быть назначены "home" в аппаратных средствах следить за их значением. В зависимости от микропроцессора, эти расположения памяти(хранения) могут быть универсальные регистры данных, расположения в основной памяти, или специальные регистры, выделенные специфическим функциям. Однако, человек, пишущий интенсивную программу знает немного или ничто относительно этого управления памятью; эта задача была делегирована программному инженеру, кто записал компилятор. Проблема, эти двое людей никогда не встречались; они связываются только через набор предопределенных правил. Языки высокого уровня проще чем ассемблирование, потому что Вы даете половину работы кому - то еще. Однако, они менее эффективны, потому что Вы не весьма уверены, как делегированная работа выполняется.

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.

Для сравнения, таблица 28-3 показывает программу скалярного произведения, написанную на ассемблере для ЦСП SHARC. Язык Ассемблер для Аналоговых Устройств ЦСП (и для 16 разрядных с фиксированной точкой и 32 разрядных SHARC устройств) известен за их простой алгебраическо-подобный синтаксис. В то время как мы не будем проходить все подробности, имеется общая операция. Обратите внимание, что *все* касается аппаратных средств; не имеется никаких абстрактных переменных в этом коде, только регистры данных и ячейки памяти.

```
001  i12 = _y;                          /* i12 points to beginning of y[ ] */
002  i4 = _x;                           /* i4 points to beginning of x[ ] */
003
004  cntr = 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.

ТАБЛИЦА 28-3

Скалярное произведение на ассемблировании (неоптимизированное). Эта программа вычисляет скалярное произведение двух массивов, *x*[ ] и *y*[ ], и сохраняет результат в переменной, *result*. Это - ассемблерный код для Аналоговых Устройств SHARC ЦСП. См. текст для подробностей.

```
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.

ТАБЛИЦА 28-4

Скалярное произведение на (оптимизированной) трансляции(ассемблировании). Это - оптимизированная версия программы в ТАБЛИЦЕ 28-2, разработанное, чтобы воспользоваться преимуществом высоко параллельной архитектуры SHARC's.

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 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.

Каждая точка с запятой представляет тактовый цикл. Массивы *x*[ ] и *y*[ ] хранятся в кольцевых буферах в основной памяти. В строках 001 и 002, регистры i4, и i12 указаны стартовые расположения этих массивов. Затем, мы выполняем 20 циклов, как управляется строкой 004. Формат для этой инструкции воспользуется преимуществом(способностью) SHARC ЦСП *zero-overhead looping* (нулевое - верхней возможности выполнения цикла). Другими словами, все переменные, необходимые для управления циклом содержаться в специализированных аппаратных регистрах, которые оперируют параллельно с другими операциями, продолжающими внутри микропроцессора. В этом случае, регистр: lcntr (счетчик цикла) загружен первоначальным значением 20, и декрементами, каждый раз цикл выполнен. Цикл закончен, когда lcntr достигает значения нуля (обозначенный инструкцией: lce, для "счетчик цикла истек"). Цикл охватывает строки от 004 до 008, как управляется инструкцией (pc, 4). То есть цикл заканчивает четыре строки после текущего счетчика программы.

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.

Внутри цикла, строка 005 загружает значение из *x*[ ] в данные регистра f2, в то время как строка 006 загружает, значение из *y*[ ] в данные регистра f4. Символы "dm" и "pm" указывают, что значения загружены(вызваны) по шине "памяти(хранения) данных ", и шине "памяти(хранения)программы", соответственно. Переменные: i4, m6, i12, и m14 – регистры в генераторах адреса данных, которые управляют кольцевым(циклическим) проведением буферов *x*[ ] и *y*[ ]. Два значения в f2 и f4 умножены в строке 007, и произведение сохранено в регистре данных f8. В строке 008, произведение в f8 добавлено к сумматору, регистра данных f12. После того, как цикл закончен, сумматор в f12 передан в память.

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).

Эта программа корректно вычисляет скалярное произведение, но это не воспользуется преимуществом высоко параллельная архитектура SHARC. Таблица 28-4 показывает эту программу, перезаписанная в высоко оптимизированной форме, с многими операциями, выполняемыми параллельно. Во первых обратите внимание, что строка 007 выполняет только 18 циклов, скорее чем 20. Также обратите внимание, что этот цикл содержит только одиночную строку (008), но что эта строка содержит многочисленные команды. Стратегия состоит в том, чтобы сделать цикл настолько эффективным, чтобы, в этом случае, одиночная строка, которая могла быть выполнена в единственном тактовом цикле. Чтобы делать это, мы должны иметь маленькое количество кода, чтобы "заполнить" регистры на первом цикле (строки 004 и 005), и другой маленький раздел кода, чтобы закончить последний(прошлый) цикл (строки 010 и 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.

Чтобы понимать, как это работает, изучите строку 008, единственная инструкция внутри цикла. В этой единственной инструкции, четыре операции выполняются параллельно: (1) значение для перемещено от кольцевого буфера в памяти программы и помещено в f2; (2) значение для *y*[ ] передвигается от кольцевого буфера в памяти данных и помещено в f4; (3) предыдущие значения f2 и f4 умножены и помещены в f8; и (4) предыдущее значение в f8 добавлено к сумматору в 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 *x*[5] x *y*[5] of (that was in f8 at the start of this cycle) is added to the value of f12.

Например, когда строка 008 выполнена пятый раз, *x*[7] и *y*[7] выбраны от памяти и сохранен в f2 и f4. В то же самое время, значения для *x*[6] и *y*[6] (которые были в f2 и f4 в начале этого цикла) умножены и помещены в f8. Кроме того, значение *x*[5] x *y*[5] (которое было в f8 в начале этого цикла) добавлено к значению 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.

Давайте сравним число тактовых циклов, требуемых неоптимизированной и оптимизированной программами. Имейте в виду, что имеются 20 циклов, с четырьмя действиями, требуемыми в каждом цикле. Неоптимизированная программа требует, чтобы 80 тактовых циклов выполнили действия в пределах циклов, плюс 5 тактовых циклов верхних, для общего количества 85 тактовых циклов. Для сравнения, оптимизированная программа проводит 18 циклов в 18 тактовых циклах, но требует 11 тактовых циклов наверху, чтобы заполнить регистры и заканчивать последний(прошлый) цикл. Это приводит к полному времени выполнения 29 тактовых циклов, или приблизительно три раза быстрее, чем метод решения "в лоб".

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.

Имеется большой вопрос: Как быстро программа СИ выполняется относительно ассемблерного кода? Когда программа в таблице 28-2 откомпилирована, выполнимый код походит на наш эффективный или неэффективный пример ассемблирования? Ответ - то, что компилятор генерирует эффективный код. Однако, важно понять, что скалярное произведение - очень простой пример. Компилятор имеет намного более трудное время, производя оптимизированную программу, когда программа становится более сложной, типа множителя вложенные циклы и ошибочные переходы к подпрограммам. Если Вы делаете кое-что прямо, ожидаете, что компилятор обеспечит Вас почти оптимальным решением. Если Вы делаете кое-что странный или сложный, ожидаете, что программа ассемблирования выполнится знаменательно быстрее чем один написанный в СИ. В самом плохом случае, думайте о коэффициенте(факторе) 2-3. Как предварительно упомянуто, эффективность СИ против ассемблирования зависит очень от частности используемый ЦСП. Архитектура с плавающей запятой может вообще запрограммироваться более эффективно чем устройства с фиксированной точкой при использовании интенсивных языков типа СИ. Конечно, надлежащие программные инструментальные средства важны для этого, типа отладчика с профилированием особенностей, которые помогают Вам понять, как долго различные сегменты кода берут, чтобы выполниться.

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 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 is to be *x* [ ] stored in data memory, while the "pm" indicates that will reside in *y* [ ] 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.

Имеется также путь, которым Вы можете получить лучшее из обоих миров: запишите программу в СИ, но используйте ассемблирование для критических разделов, которые должны выполниться быстро. Это - одна причина, что СИ настолько популярен в науке и разработке. Это оперирует как интенсивный язык, но также и позволяет Вам непосредственно управлять аппаратными средствами, если Вы, так желаете. Даже если Вы намереваетесь программировать только в СИ, Вы будете вероятно нуждаться в некотором знании архитектуры ЦСП и системы команд ассемблирования. Например, оглянитесь назад, в строках 002 и 003 в таблице 28-2, скалярное произведение программируется в СИ. "dm" означает, что *x*[ ] должен быть сохранен в памяти данных, в то время как "pm" указывает, что *y*[ ] постоянно находится в памяти программы. Даже притом, что программа написана на языке высоких уровней, основное знание аппаратных средств, все же требуется, чтобы получить лучшую эффективность от устройства.

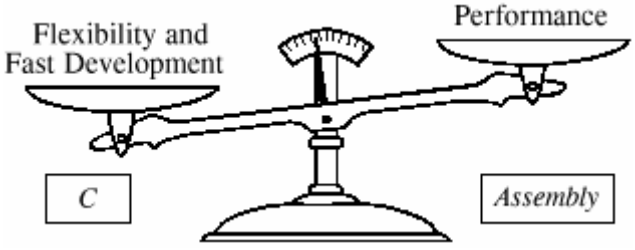


РИСУНОК 28-10. Ассемблирование против СИ. Программы в СИ более гибки и более быстрые, чтобы развиться. Для сравнения, программы на ассемблере часто имеют лучшую эффективность; они выполняются быстрее и используют меньшее количество памяти, снижая их стоимость.

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.

Который язык является лучшим для *вашего* приложения? Это зависит от того, что для Вас является более важным. Если Вы нуждаетесь в гибкости и быстром развитии, выберите СИ. С другой стороны, используйте ассемблирование, если Вы нуждаетесь в лучшей возможной эффективности. Как иллюстрировано на рис. 28-10, это - сделка, на которую Вы вынуждены пойти. Имеются некоторые вещи, которые Вы должны рассмотреть.

□ 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.
□ Какая зависимость от сложности программы? Если программа большая и сложная, Вы, вероятно, захотите использовать СИ. Если программа маленькая и простая, ассемблирование может быть хороший выбором.

□ 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.
□ Вы делаете упор на максимальное быстродействие ЦОС? Если так, ассемблирование даст Вам снижение эффективности используемого устройства. Для менее взыскательных приложений(с небольшими требованиями), ассемблирование имеет небольшое преимущество, и Вы должны рассмотреть использование СИ.

□ 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.
□ Сколько программистов будут работать вместе? Если проект достаточно большой для больше чем одного программиста, к СИ и используя подключение ассемблирования только в течение времени критические сегменты

□ Which is more important, *product cost* or *development cost*? If it is product cost, choose assembly; if it is development cost, choose C.
□ Что более важно, *стоимость изделия(программы)* или *стоимость развития*? Если это - стоимость изделия(программы), выберите ассемблирование; если это - стоимость развития, выберите СИ.

□ 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.
□ Какова ваша подготовка? Если Вы опытны в трансляции(ассемблировании) (на других микропроцессорах), выберите ассемблирование для вашего ЦСП. Если ваша предыдущая работа находится в СИ, выберите СИ для вашего ЦСП.

□ What does the DSP's manufacturer suggest you use?
□ Что изготовитель ЦСП предлагает, чтобы Вы использовали?

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.

Этот последний пункт очень важен. Предположим, что Вы спрашиваете изготовителя ЦСП, который язык использовать, и они сообщают Вам: "*Или СИ или ассемблирование могут использоваться, но мы рекомендуем СИ*." Вы имели, лучше берут их совет! Что они действительно говорят: " *Наш ЦСП настолько труден, программировать на ассемблере, что Вы будете нуждаться в 6 месяцах обучения использовать это*. " С другой стороны, некоторые ЦСП просты программировать на ассемблировании. Например, Аналоговые устройства находятся в этой категории. Только спросите их инженеров; они очень гордятся этим.

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.

Один из лучших способов для выбора решения относительно изделий(программ) ЦОС и программного обеспечения должен говорить с инженерами, кто использовали их. Спросите изготовителей относительно рекомендаций компаний, использующих их изделия(программы), или ищите сеть людей, которых Вы можете посылать по электронной почте. Не будьте застенчивы; проектирует любят давать их мнения относительно изделий(программ), которые они использовали. Они будут польщены, что Вы спросили.

How Fast are DSPs?
Как Быстры – ЦСП-ы?

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).

Первичная причина для использования ЦСП вместо традиционного микропроцессора - быстродействие, способность переместить выборки в устройство, выполнять необходимые математические операции, и вывод обработанных данных. Это поднимает вопрос: Как быстр - ЦСП? Обычный путь ответа на этот вопрос - **эталонные тесты**, методы для выражения быстродействия микропроцессора как число. Например, системы с фиксированной точкой часто цитируются в **MIPS** (миллион целочисленных операций в секунду). Аналогично, устройства плавающей запятой, могут быть определены в **MFLOPS** (миллион операций плавающей запятой в секунду).

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!

Сто пятьдесят лет назад, Британский премьер-министр Бенджамин Дисраели объявлял, что имеются три типа лжи: *ложь*, *проклятая ложь*, и *статистика*. Если бы Дисраели был жив сегодня и работал с микропроцессорами, он прибавил бы *эталонные тесты* как четвертую категорию. Идея эталонных тестов состоит в том, чтобы обеспечить сравнение голова в голову, чтобы показать, которое устройство является лучшим. К сожалению, это часто терпит неудачу в практичности, потому что различные микропроцессоры превосходят других в различных областях. Вообразите спрашивать вопрос: Который лучший автомобиль, является ли Кадиллак или Феррари? Это зависит от того, что Вы хотите это для!

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.

Путаница относительно эталонных тестов ухудшена по конкурентоспособному характеру промышленности электроники. Изготовители хотят показать их изделия в лучшем свете, и они будут использовать любую неоднозначность в процедуре испытания к их преимуществу. Имеется старое высказывание в электронике: "*запись спецификации может получить вдвое больше эффективность от устройства от инженера*". (“*В русской транскрипции это звучит “неважно что представить, важно как представить*”). Эти люди не неправдивы, платят только за то, что они имеют хорошее воображение. Эталонные тесты должны быть просмотрены как инструмент для сложной задачи. Если Вы неопытны в использовании этого инструмента, Вы можете сделать неправильное заключение. Лучший подход состоит в том, чтобы искать специфическую информацию относительно быстродействия выполнения алгоритмов, которые Вы планируете выполнять. Например, если ваше приложение вызывает алгоритм КИХ-фильтра, ищите точное число тактовых циклов, которые требуется для устройства, чтобы выполнить эту специфическую задачу.

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

При использовании этой стратегии, давайте смотреть на время, требуемое, чтобы выполнить различные алгоритмы на нашем показанном ЦСП, аналоговых устройств семейства SHARC. Имейте в виду, что быстродействие микропроцессора удваивается каждые три года. Это означает, что Вы должны обратить особое внимание на метод, который мы используем в этом примере. Фактические числа(номера) всегда изменяются, и Вы будете должны повторить вычисления, каждый раз, как Вы начинаете новый проект. В мире технологий двадцать первого столетия, мигните, и Вы устарели!

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.

Когда это спутник к понимающему времени выполнения, семейство SHARC - один из самого простого ЦСП, чтобы работать с. Это - то, потому что это может выполнять операцию суммирования умножений в единственном тактовом цикле. Так как большинство КИХ-фильтров использует от 25 до 400 коэффициентов, от 25 до 400 тактовых циклов требуются, соответственно, для каждой обрабатываемой выборки. Как предварительно описано, имеется маленькое количество сверх необходимого, чтобы достичь этой эффективности цикла (заправка первый цикла и завершение последнего(прошлого) цикла), но незначительно, когда число циклов этот большой. Чтобы получить производительность фильтра, мы можем делить тактовую частоту SHARC (40 мгЦ в настоящее время) числом тактовых циклов, требуемых в выборку. Это дает нам максимальную скорость передачи данных КИХ около 100КБ к 1.6М выборок. Вычисления не могут становиться намного более простыми, чем это! Эти значения производительности КИХ показываются в рис. 28-11.

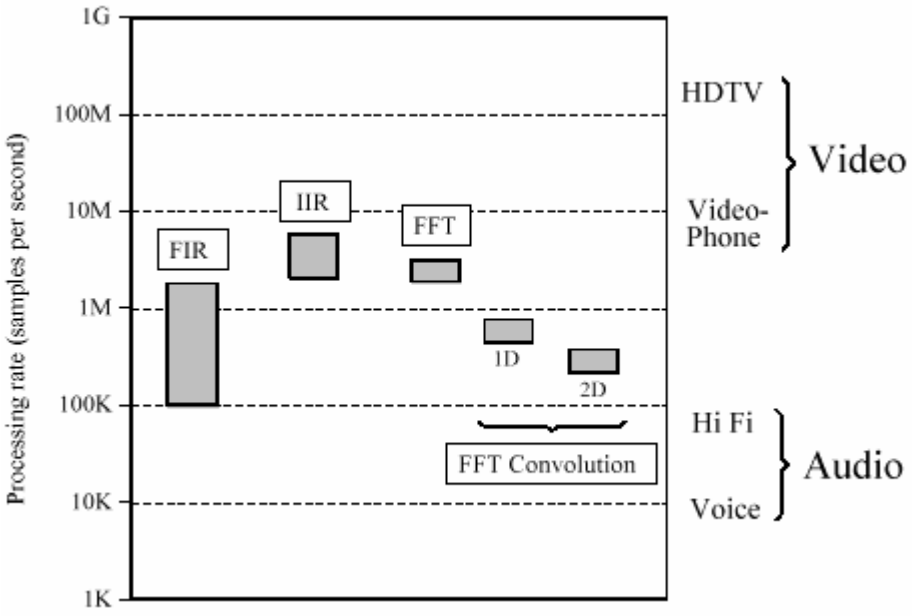


FIGURE 28-11
The speed of DSPs. The throughput of a particular DSP algorithm can be found by dividing the clock rate by the required number of clock cycles per sample. This illustration shows the range of throughput for four common algorithms, executed on a SHARC DSP at a clock speed of 40 MHz.
РИСУНОК 28-11
Быстродействие ЦОС. Производительность алгоритм частного ЦСП может быть найдена, разделяя тактовую частоту требуемым числом тактовых циклов в выборку. Эта иллюстрация показывает диапазон производительности для четырех обычных алгоритмов, выполненных на SHARC ЦСП с тактовой частотой 40 МГц.

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.

Вычисления – так же, как простой для рекурсивных фильтров. Типично БИХ-фильтры используют приблизительно от 5 до 17 коэффициентов. Так как эти циклы относительно коротки, мы прибавим маленькое количество сверх них, будем говорить 3 цикла на выборку. Это приводит от 8 до 20 тактовых циклов, требуемых на выборку обработанных данных. Для тактовой частоты 40 МГЦ, это обеспечивает максимальную производительность БИХ 1.8M - 3.1M выборки в секунду. Эти значения БИХ также показываются в рис. 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.

Затем мы прибываем в методы частотного домена, основанные на Быстром Преобразовании Фурье. Подпрограммы БПФ почти всегда обеспечиваются изготовителем ЦСП. Они - высокооптимизированные подпрограммы, написанные на ассемблировании. Полость спецификации ADSP-21062 SHARC DSP указывает, что 1024 выборки комплексных БПФ требует 18221 тактовых циклов, или приблизительно 0.46 миллисекунд в 40 МГЦ. Чтобы вычислять производительность, проще рассмотреть это как 17.8 тактовых циклов в выборку. Это значение " на выбрку " только изменяется слегка с дольше или более короткими БПФ. Для образца, 256 выборок БПФ требуют приблизительно 14.2 тактовых цикла на выборку, и 4096 выборок БПФ требуют 21.4 тактовых цикла на выборку. Реальные БПФ могут быть рассчитаны приблизительно на 40% быстрее, чем эти значения комплексного БПФ. Это делает полный диапазон из всех подпрограмм БПФ приблизительно от 10 до 22 тактовых циклов на выборку, соответствуя производительности около 1.8M - 3.3M выборки в секунду.

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.

Свертка БПФ - быстрый способ выполнить КИХ-фильтры. В типичном случае 512 выборок сегмента принятые от ввода, дополняются 512 дополнительными нулями, преобразовывая его в спектр частот 1024 точки БПФ. После умножения этого спектра желательной частотной характеристикой, 1024 точки инверсии БПФ используется, чтобы двигаться обратно во домен времени. Заканчивающийся 1024 точки объединены с смежными обрабо-

танными сегментами, используя перекрытие - добавленный метод. Это производит 512 точек сигнала выхода.

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.

Сколько тактовых циклов это занимает? Каждые 512 выборок сегмента требуют 1024 точки двух БПФ, плюс маленькое количество сверх. В вокруг терминов, это около коэффициента пять больше чем для единственного БПФ 512 точек. Так как реальное БПФ требует приблизительно 12 тактовых циклов на выборку, свертка БПФ может быть выполнена приблизительно за 60 тактовых циклов на выборку. Для 2106x SHARC DSP 40 MHz, это соответствует производительности данных приблизительно 660k выборок в секунду.

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.

Обратите внимание, что это - относительно то же самое как 60 коэффициентов КИХ-фильтра, выполненных обычной сверткой. Другими словами, если КИХ-фильтр имеет меньше чем 60 коэффициентов, это может быть выполнено быстрее стандартной сверткой. Если это имеет больше чем 60 коэффициентов, свертка БПФ более быстрая. Ключевое преимущество свертки БПФ состоит в том, что время выполнения увеличивается только как логарифм числа коэффициентов. Например 4096 точек ядра фильтра, требует приблизительно только, чтобы 30% дольше выполнил как один с только 512 точек.

FFT convolution can also be applied in two-dimensions, such as for image processing. For instance, suppose we want to process an 800x600 pixel image in the frequency domain. First, pad the image with zeros to make it 1024x1024. 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 1024x1024 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.

Свертка БПФ может также применяться в с двумя измерениями, типа для обработки изображения. Для образца, предположите, что мы хотим обработать изображение 800x600 пикселов в частотном домене. Во первых, дополните изображение с нулями, чтобы делать это 1024x1024. Двумерный спектр частот тогда рассчитан, беря БПФ каждой из строк, сопровождается, беря БПФ каждого из заканчивающихся столбцов. После умножения этого спектра 1024x1024 желательной частотной характеристикой, двумерное Обратное БПФ принято. Это выполнено, беря Обратное БПФ каждой из строк, и затем каждый из заканчивающихся столбцов. Прибавляя число тактовых циклов и деления числом выборок, мы находим, что эта полная процедура берет грубо 150 циклов на пиксел. Для ADSP-

2106 40 MHz, это соответствует производительности данных относительно 260k выборок в секунду.

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.

Сравнивая эти различные методы в рис. 28-11, мы можем делать важное наблюдение. *Почти все методы ЦОС требуют от 4 до 400 команд (тактовых циклов в семействе SHARC) для выполнения*. Для SHARC ЦСП, работающего в 40 МГЦ, мы можем немедленно заключить, что его производительность данных будет между 100КБ и 10М выборок в секунду, в зависимости от того, как комплекс алгоритма используется.

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.

Теперь, когда мы понимаем, как быстрые ЦСП *могут* обрабатывать цифровые сигналы, давайте поворачивать наше внимание к другому концу; как быстро мы должны обработать данные? Конечно, это зависит от приложения. Мы будем смотреть на два из наиболее общих, обработку звука и обработку видео.

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.

Скорость передачи данных, необходимая для аудио-сигнала зависит от требуемого качества воспроизведенного звука. В низком конце, качество телефонной речи требует только фиксации частот приблизительно от 100 Hz до 3.2 кГц, диктовка частоты выборки относительно 8КБ выборок в секунду. Для сравнении, высококачественная музыка должна содержать полный диапазон, от 20 Hz до 20 кГц, человеческого слуха. Частота выборки 44.1 кГц часто используется, и для левых и правых каналов, делая полный сигнал высокой верности 88.2k выборками в секунду. Как семейство SHARC сравнивается с этими требованиями? Как показано на рис. 28-11, это может легко обрабатывать высокую точность аудио, или обрабатывать несколько дюжин голосовых сигналов одновременно.

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 352x288 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 1920x1080 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.

Видеосигналы - другая история; они требуют относительно времен с одной тысячей скорости передачи данных аудиосигналов. Хороший пример низкокачественного видео - CIF (Общий Формат Интерфейса) стандарт для видеофонов. Это использует 352x288 пикселов, с 3 цветами на пиксел, и 30 кадров в секунду, для полной скорости передачи данных 9.1 миллионов выборок в секунду. В высоком конце качества имеется HDTV (телевидение высокой четкости), используя 1920x1080 пикселов, с 3 цветами на пиксел, и 30 кадров в секунду. Это требует скорости передачи данных к более чем 186 миллионам выборок в секунду. Эти скорости передачи данных - выше возможностей единственного(отдельного) SHARC ЦСП, как показано в рис. 28-11. Имеются другие приложения, которые также требуют их очень высокой скорости передачи данных, например, радар, гидролокатор, и военные использования типа управления ракетой.

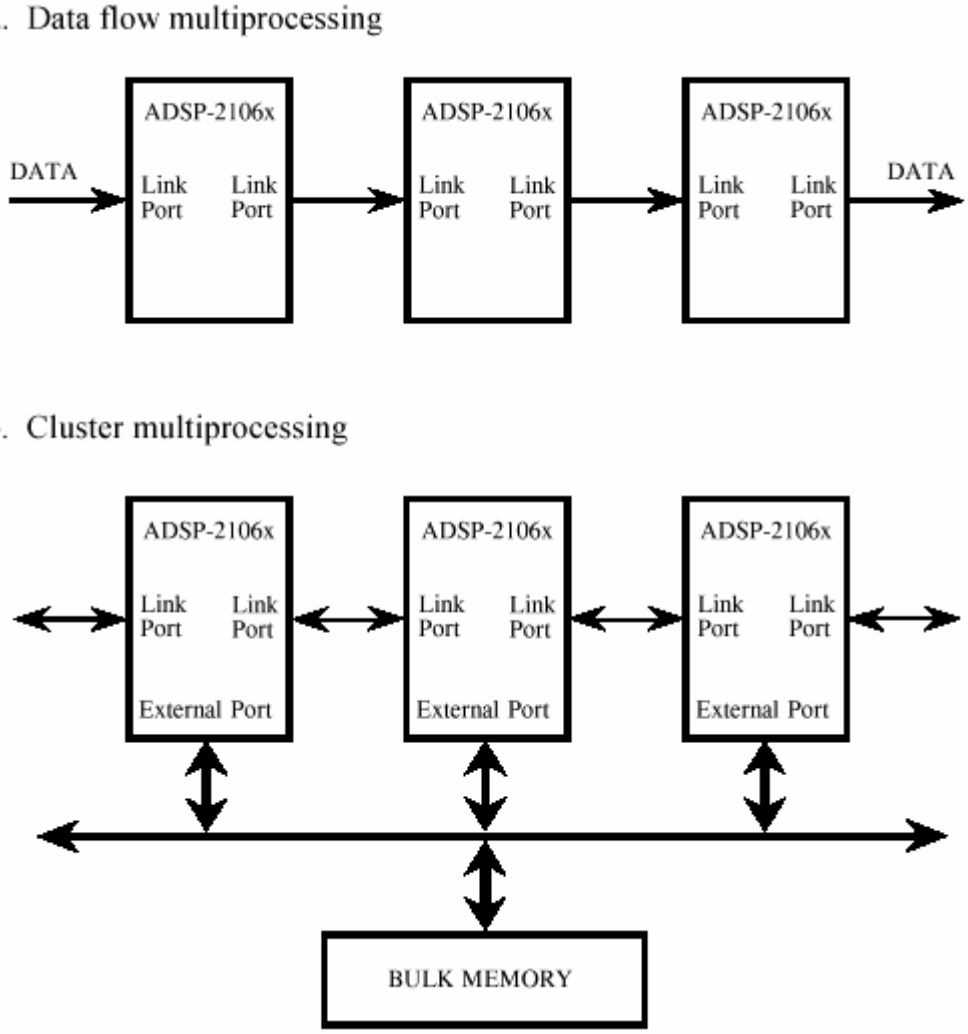


FIGURE 28-12
Multiprocessing configurations. Multiprocessor systems typically use one of two schemes to communicate between processor nodes, (a) dedicated point-to-point communication channels, or (b) a shared global memory accessed over a parallel bus.

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 mul-

tiprocessing systems. In Fig. (a), the algorithm is broken into sequential steps, with each processor performing one of the steps in an "assembly line" 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.

Чтобы обрабатывать эти мощные задачи, несколько ЦСП могут быть объединены в единственную систему. Это называется **мультипроцессорной обработкой** (**многопроцессорной обработкой**) или **параллельной обработкой**. SHARC ЦСП были разработаны с этим типом многопроцессорной обработки в памяти, и включают специальные особенности, чтобы делать это настолько просто насколько возможно. Для образца, никакая внешняя аппаратная логика не требуется, чтобы подключить внешние шины множителя SHARC ЦСП вместе; вся логика организации доступа к общей шине уже содержится в пределах каждого устройства. Как альтернатива, порты связи (4 разряда, параллельных) могут использоваться, чтобы подключить многочисленные процессоры в различных конфигурациях. Рисунок 28-12 показывает типичные пути, которыми SHARC ЦСП могут размещаться в многопроцессорных системах. В рис. (a), алгоритм разбит на последовательные шаги, с каждым процессором, выполняющим один из шагов в стратегию "сборочной линии". В (b), процессоры взаимодействуют через отдельную общедоступную глобальную память, обратился по параллельной шине, (то есть, внешний порт). Рисунок 28-13 показывает другой путь, которым большое количество процессоров может быть объединено в отдельную систему, 2-ую или трехмерную "сеть". Каждая из этой конфигурации будет иметь относительные преимущества и недостатки для специфической задачи.

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).

Чтобы делать жизнь программиста проще, семейство SHARC использует объединенное адресное пространство. Это означает, что 4 Gigaword адресное пространство, обратился адресной шиной 32 двоичных разрядов, разделено среди различных процессоров, которые работают вместе. К пересылке данных от одного процессора до другого, просто читают от или записывают в соответствующие ячейки памяти. Внутренняя логика SHARC заботится о остальном, передавая данные между процессорами в со скоростью столь же высокой как 240 Мегабайтов в секунду (40 МГЦ).

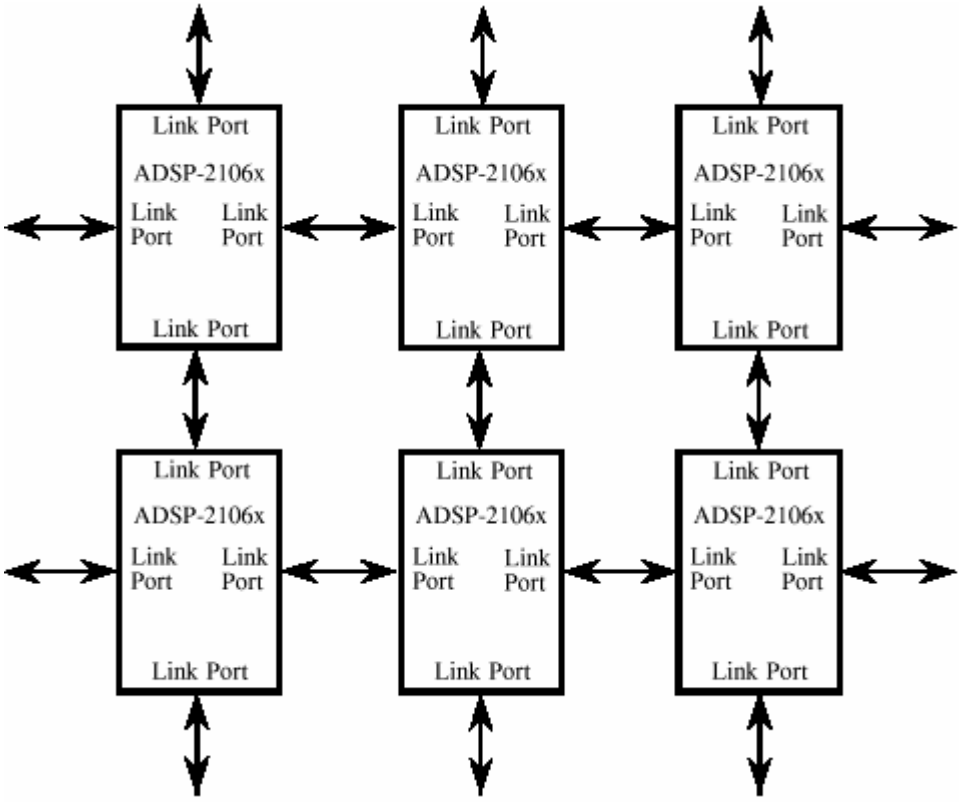


FIGURE 28-13
Multiprocessing "mesh" configuration. For applications such as radar imaging, a 2D or 3D array may be the most efficient way to coordinate a large number of processors.
РИСУНОК 28-13
Многопроцессорная конфигурация "сети". Для приложений типа отображения радара, 2-ый или трехмерный массив может быть наиболее эффективным способом координировать большое количество процессоров.

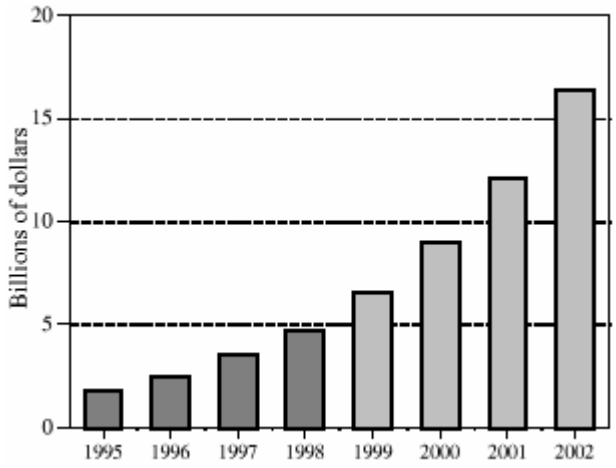
The Digital Signal Processor Market
Рынок Цифровых Сигнальных процессоров

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 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.

Рынок ЦОС очень большой и быстро растет. Как показано в рис. 28-14, это будет приблизительно 8-10 миллиардов долларов/года в повороте столетия, и возрастающий в скорости 30-40% каждый год. Это подпитывается непрерывным запросом на лучшие и более дешевые бытовые изделия, типа: ячеистый телефоны, мультимедийные компьютеры, и воспроизводство музыки высокой верности воспроизведения. Эти приложения с высоким доходом формируют поле, в то время как менее выгодные области, типа оснащения научного аппаратурой, только дополнительная статья(пункт) на волне технологии.

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. Billions of dollars
РИСУНОК 28-14
Рынок ЦСП. В повороте столетия, рынок ЦОС будет 8-10 миллиардов долларов в год, и разворачивающий в скорости приблизительно 30-40% в год. Миллиарды долларов



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.

ЦСП могут быть куплены в трех формах, как **ядро**, как **процессор**, и как изделие **board level** (уровня платы). В ЦСП, термин "ядро" относится к разделу процессора, где ключевые задачи выполнены, включая регистры данных, множитель, АЛУ, адресный генератор, и установления последовательности(упорядочения) выполнения программ. Полный процессор требует объединения ядра с памятью и связи, с помощью интерфейса, с внешним миром. В то время как ядро и эти периферийные разделы разработаны отдельно, они будут изготовлены на той же самом кристалле кремния, делая *процессор* отдельной интегральной схемой.

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 35x35x4 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.

Предположим, что Вы формируете ячеистые телефоны и хотите включить ЦСП в проект. Вы будете вероятно хотеть купить ЦСП как *процессор*, то есть интегральную схему

("чип") который содержит ядро, память и другие внутренние особенности. Например, SHARC ADSP-21060 спутник в пакете "240 lead Metric PQFP", только 35x35x4 мм в размере. Чтобы включать этот ИС в ваше изделие, Вы проектируете печатную монтажную схему, где это будет спаяно рядом с вашей другой электроникой. Это - наиболее обычный путь, которым ЦСП используются.

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.

Теперь, предположите компанию, Вы работаете для производства ее собственные интегральные схемы. В этом случае, Вы не могли бы хотеть полный процессор, только проект ядра. После завершения соответствующего соглашения лицензирования, Вы можете запускать делать чипы, которые высоко настроены к вашему специфическому приложению. Это дает Вам гибкость отбора, сколько памяти включена, как чип получает и пересылает данные, как это упаковано, и так далее. Заказные устройства этого типа - все более и более важный сегмент рынка ЦСП.

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.

Наконец, имеются несколько компаний дюжины, которые продадут Вам ЦСП, уже установленные на напечатанной монтажной схеме. Они имеют такие особенности как дополнительная память, Аналого-цифровые и Цифро-аналоговые преобразователи, разъемы EPROM (ПРОГРАММИРУЕМОГО ПЗУ), многочисленные процессоры на той же самой плате, и так далее. В то время как некоторые из этих плат предназначены, чтобы использоваться как автономные компьютеры, больше всего конфигурированы, чтобы быть подключенными в главный компьютер, типа персонального компьютера. Компании, которые делают эти типы плат, называются **Разработчиками Третьего лица**. Лучший способ находить их состоит в том, чтобы спросить изготовителя ЦСП, который Вы хотите использовать. Смотрите на website изготовителя ЦСП; если Вы не находите список там, пошлите им электронную почту. Они будут больше чем счастливы сообщить Вам, кто использует их изделия и как войти с ними в контакт.

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:

На сегодняшний день рынок Цифровых Сигнальных процессоров (1998) - во власти четырех компаний. Имеется список, и общая схема, которую они используют для нумерации их изделий:

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:

Имейте в виду, что различие между ЦСП и другими микропроцессорами - не всегда чистая строка. Например, смотрите, как Intel описывает MMX добавление технологии его процессору Pentium:

> *"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."*
>
> *" Инженеры Intel прибавили 57 мощных новых команд определенно разработанных(предназначенных), чтобы управлять и обрабатывать видео, звуковые и графические данные эффективно. Эти команды ориентируются к высоко параллельным, повторным последовательностям, часто находимым в операции мультимедиа. "*

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.

В будущем, мы будем несомненно видеть большее количество функций подобных ЦСП, объединенных в традиционные микропроцессоры и микроконтроллеры. Internet и другие приложения мультимедиа - сильная движущая сила для этих изменений. Эти приложения расширяются так быстро, через двадцать лет, очень возможно, что Цифровой сигнальный процессор может стать "традиционным" микропроцессором.

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 ex-

ploring the manufacturers' websites, and then sending them e-mail requesting specific information.

Как Вы не отстаете от этого быстро изменения поля? Лучший путь состоит в том, чтобы читать каталоги, которые охватывают рынок ЦОС, типа EDN (Electronic Design News, *www.ednmag.com*), и ECN (Electronic Component News, *www.ecnmag.com*). Они распределены свободно, и содержат современную информацию относительно того, что является доступным и где промышленность идет. Каталоги - "чтение" для любого серьезно относящегося к полю. Вы будете также хотеть быть в списке адресатов нескольких изготовителей ЦСП. Это позволит Вам получать новые объявления изделий, оценивая информацию, и специальные предложения (типа свободных программных и дешевых комплектов оценки). Некоторые изготовители также распределяют периодические информационные бюллетени. Например, Analog Devices издают *Analog Dialogue* четыре раза в год, содержа статьи и информацию относительно текущих тем(разделов) в обработке сигналов. Со всеми этими ресурсами, и намного больше, можно входить в контакт по internet. Начало, исследуя websites изготовителей, и затем посылая им требование электронной почтой специфической информации.