Эффективное программирование с учетом архитектурных особенностей цифровых сигнальных процессоров

Современные цифровые сигнальные процессоры (ЦСП) обладают столь привлекательным сочетанием производительности, рассеиваемой мощности, набора периферийных узлов и цены, что многие разработчики склоняются к их применению вместо процессоров, на которых они традиционно создавали свои системы. Одно из потенциальных препятствий на пути перехода к ЦСП — это большое количество алгоритмов на языках C/C++, написанных разработчиками для своих приложений. Естественно, при этом у них возникает желание перенести существующее программное обеспечение (ПО), написанное на высокоуровневых языках, на платформу ЦСП, что позволит при правильном использовании архитектурных особенностей процессора обеспечить уровень производительности, недостижимый на платформах, с которыми они работали прежде. Более того, они предпочли бы использовать знакомую, интуитивно понятную среду разработки, имея, в то же время, возможность реализовывать отдельные части программы на языке ассемблера для повышения производительности. В этой статье обсуждаются методы и стратегии программирования ЦСП в современной среде разработки ПО.

Дэвид КАТЦ (David KATZ)
Томаш ЛУКАШЕК (Tomasz LUKASIAK)
Рик ДЖЕНТАЙЛ (Rick GENTILE)
Перевод: Александр СОТНИКОВ

Ассемблер и высокоуровневые языки программирования

Приступая к созданию проекта на базе ЦСП, разработчик должен определиться с используемой методикой программирования. Выбор обычно осуществляется между языком ассемблера и высокоуровневым языком, таким как C или C++. При принятии решения необходимо рассмотреть множество факторов, поэтому важно четко понимать преимущества и недостатки, которые таит в себе каждый из подходов.

Преимущества C/C++ заключаются в модульности, возможности переноса на другие платформы и простоте повторного использования кода. Большинство разработчиков ПО для встраиваемых систем имеют опыт программирования хотя бы на одном из этих высокоуровневых языков. Кроме того, существует огромная база доступного программного кода, который можно достаточно легко портировать с микроконтроллеров и ЦСП предыдущих поколений на современные ЦСП. Также, как правило, желательно, чтобы команда разработчиков была разбита на несколько групп, занимающихся написанием отдельных системных модулей. Высокоуровневые языки программирования позволяют таким группам заниматься созданием ПО без привязки к конкретной аппаратной базе.

Традиционные языки ассемблера долгое время не пользовались особой популярностью из-за их «таинственного» синтаксиса и странных аббревиатур. Однако в современных архитектурах с так называемым «алгебраическим синтаксисом» ассемблера эти факторы уже не столь явно выражены. В таблице приведены примеры записей типичных команд ЦСП в традиционном стиле и в алгебраическом формате. Очевидно, что вторая форма записи гораздо проще для понимания. В приведенных примерах *r* — это регистры данных, а *p* — регистры указателей.

Таблица. Сравнение традиционного синтаксиса ассемблера с современным алгебраическим синтаксисом

Тип операции	Традиционный синтаксис ассемблера	Алгебраический синтаксис ассемблера
Перемещение содержимого регистра	mov r7, r0	r7 = r0
Сложение	add r0, r1, r2	r0 = r1 + r2
Вычитание	sub r3, r3, r1	r3 = r3 - r1
Загрузка регистра из памяти	lw r5,p3	r5 = [p3]
Сохранение содержимого регистра в память	sw r1, p0	[p0] = r1
Условный переход к метке _equal при равенстве входных операндов (регистров), в противном случае — переход к метке _not_equal	beq r5, r6, _equal bne r5, r6, _not_equal	cc = r5 ==r6 if cc jump _equal if !ccjump _not_equal
Загрузка регистра из памяти с инкрементированием регистра указателя	lw r3, p5 addi p5, p5, 1	r3 = [p5++]

Другая причина, по которой на языке ассемблера достаточно сложно программировать, заключается в том, что операции с данными осуществляются в нем на уровне реальных регистров, вычислительных блоков и блоков памяти ЦСП. В языках C/C++ манипуляции данными обычно выполняются на гораздо более высоком уровне абстракции путем использования переменных и вызовов функций/процедур, что делает программу более понятной.

Современные компиляторы C/C++ достаточно мощны, и многие из них могут весьма эффективно преобразовывать программы, написанные на высокоуровневом языке программирования, в код на языке ассемблера. Зачастую лучше всего просто предоставить всю работу оптимизатору компилятора. Однако факт остается фактом: производительность компилятора подстроена под специфический набор функций, которые его разработчики

посчитали наиболее важными. Поэтому ни при каких условиях производительность скомпилированного кода не может превзойти производительность кода, оптимизированного вручную на языке ассемблера.

В результате разработчики используют язык ассемблера только при необходимости оптимизировать блоки программы, отвечающие за интенсивную обработку данных. Применение ключей оптимизации компилятора высокоуровневого языка может дать хорошие результаты, однако ничто не сравнится с продуманным, непосредственным управлением вычислениями и перемещением данных в ЦСП. Именно поэтому разработчики зачастую используют комбинацию языков C/C++ и ассемблера. Высокоуровневые языки отлично подходят для задач управления и базовых манипуляций данными, а ассемблер идеален для интенсивных числовых расчетов.

Для эффективного написания программ на языке ассемблера от программиста требуется понимание архитектурных особенностей, которые отличают ЦСП от процессоров, не оптимизированных для супербыстрого «перемалывания» чисел. К ним относятся:

- специализированные режимы адресации;
- аппаратная поддержка циклов;
- кэшируемая память;
- многофункциональные команды;
- конвейер с блокировкой;
- гибкий регистровый файл данных.

Использование перечисленных архитектурных особенностей процессора может привести к значительному увеличению скорости вычислений. Обсудим каждую из них более подробно.

Специализированные режимы адресации

Для того чтобы процессор мог в одном цикле обращаться к нескольким словам данных, необходима полная гибкость в формировании адресов. Так как в ряде распространенных приложений, например, при обработке видеоизображений, требуется работа с 8 разрядными данными, для достижения максимальной эффективности процессор должен, в дополнение к более характерной для ЦСП адресации со смещением по 16 или по 32 бита, поддерживать и побайтовую адресацию. Если обращения к памяти ограничены одним фиксированным шагом смещения, то для маскирования необходимых битов могут понадобиться дополнительные процессорные циклы.

Другой полезный режим адресации — это «циклическая адресация». Это свойство процессор должен поддерживать аппаратно, без издержек на программное управление. Поддержка циклической адресации позволяет программисту настраивать в памяти процессора циклические (кольцевые) буферы и автоматически, без дополнительного программного вмешательства, выполнять адресацию их элементов со смещением указателя.

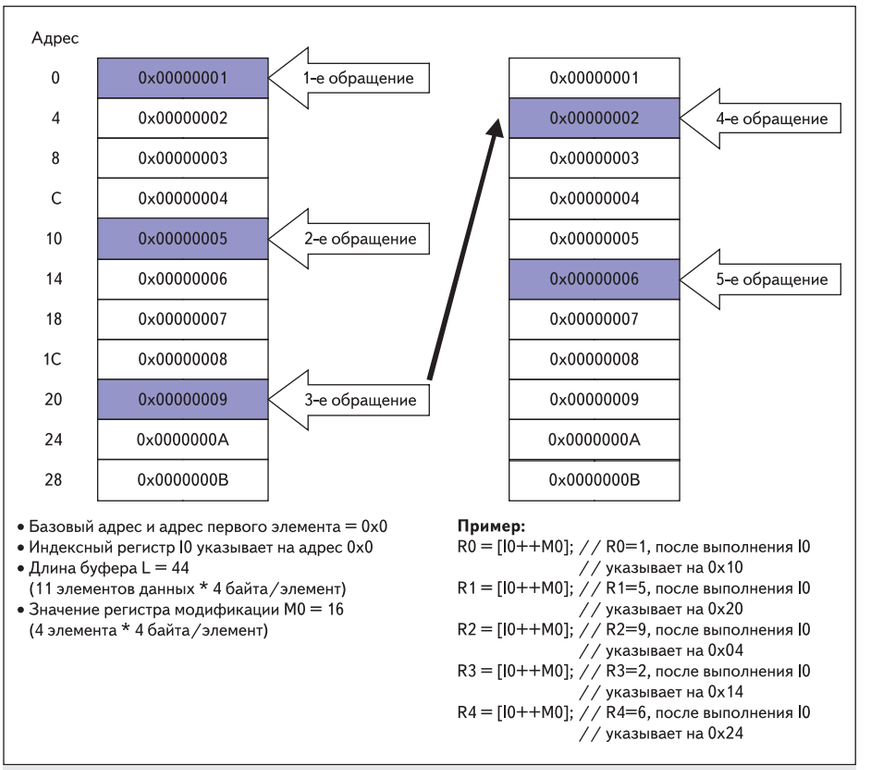

Рис. 1. Пример циклической буферизации

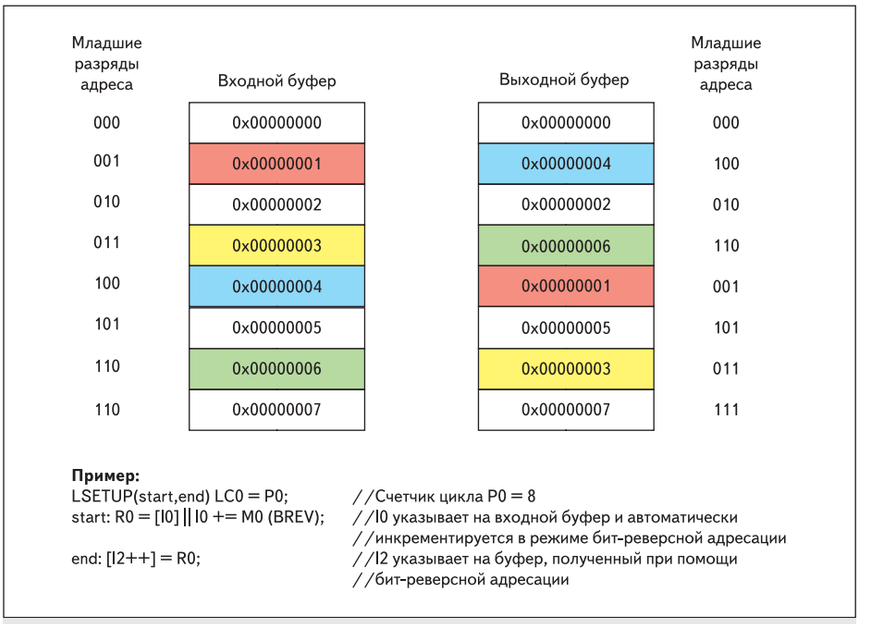

Рис. 2. Механизм аппаратной бит-реверсной адресации

Генератор адреса поддерживает значения шага по индексу, отличные от единицы, и, что более важно, автоматически выполняет «циклический возврат» (wrap around) к началу буфера, как показано на рис. 1. Без такой возможности автоматического формирования адреса программисту понадобилось бы вручную отслеживать перемещение указателя по буферу, тратя на это ценные процессорные циклы.

Еще одним важным с точки зрения эффективного выполнения алгоритмов обработки сигналов, таких как БПФ и дискретное косинусное преобразование (DCT), режимом адресации является бит-реверсная адресация (bit reversal). Как следует из названия, при та-

кой адресации порядок следования битов в двоичном адресе изменяется на обратный. То есть младшие разряды меняются местами со старшими разрядами адреса. Подобное упорядочивание данных необходимо для выполнения операции «бабочка» по основанию 2, поэтому бит-реверсная адресация применяется для связи отдельных каскадов БПФ. Организовать вычисление указателя с реверсированием битов можно и программно, однако это будет очень неэффективно. Пример, иллюстрирующий адресацию в режиме с бит-реверсией, показан на рис. 2.

Аппаратная поддержка циклов

Циклы — это одна из основных конструкций алгоритмов обработки данных в системах цифровой обработки сигналов. Существует два ключевых аспекта организации циклов, с помощью которых во многих алгоритмах достигается увеличение быстродействия. Первый из них — это поддержка «аппаратных циклов с нулевыми непроизводительными издержками». Как и в случае с режимами адресации, такие циклы реализуются в процессоре на аппаратном уровне. И, опять же, несмотря на то, что подобная функция может быть реализована программно, связанные с этим непроизводительные издержки скажутся на запасе производительности, необходимом для обработки в режиме реального времени. Поддержка циклов с нулевыми непроизводительными издержками позволяет программистам организовывать циклы, просто указав требуемое количество итераций и границы цикла. При этом процессор будет исполнять команды цикла до достижения счетчиком заданного значения.

Циклы с нулевыми непроизводительными издержками есть в большинстве процессоров, однако действительно существенного прироста производительности при выполнении циклов позволяют достичь «аппаратные буферы циклов». Эти буферы представляют собой подобие кэша для команд, исполняемых в теле цикла. То есть при первом прохождении цикла выполняемые в нем команды могут быть помещены в буфер, что устраняет необходимость повторной выборки (“re-fetch”) тех же самых команд из памяти на последующих итерациях. Это может привести к значительному увеличению скорости исполнения цикла, так как обращение к буферу осуществляется за один такт процессора. Никаких дополнительных программных настроек для работы буфера цикла не требуется — важно лишь знать его объем, поскольку максимальная эффективность будет достигнута, когда число команд в цикле меньше этого значения.

Кэшируемая память

В типичных ЦСП объем быстрой внутренней памяти обычно мал. В свою очередь, микроконтроллеры обычно имеют доступ к бóльшим областям внешней памяти. Иерархическая архитектура памяти совмещает в себе лучшие стороны обоих подходов за счет реализации нескольких уровней памяти с различной производительностью. Для приложений, требующих наибольшей степени детерминизма, используется внутренняя SRAM, обращение к которой может быть выполнено за один процессорный цикл ядра. Для систем с большими объемами программного кода есть внутренняя и внешняя памяти большой емкости, обладающие большими временами доступа.

Сама по себе такая иерархия приносит мало пользы, поскольку программы большого объема поместятся только в медленную внешнюю память, и, следовательно, современные высокоскоростные процессоры будут работать на гораздо меньших скоростях, чем потенциально достижимая. Кроме того, программистам придется вручную перемещать критические части кода во внутреннюю память и из нее. Однако если в архитектуре присутствуют кэш-памяти данных и команд, организация взаимодействия с внешней памятью значительно упрощается. Применение кэш-памяти сокращает необходимость программных пересылок данных и команд в ядро процессора.

На рис. 3 показана типичная конфигурация, в которой команды извлекаются из внешней памяти по мере необходимости. Кэш команд обычно работает по принципу LRU (Least Recently Used), когда наиболее часто исполняемые команды замещаются реже других. В ряде случаев оптимальной производительности системы можно добиться, настроив, как показано на рис. 3, часть внутренней памяти данных как кэш, а часть — как SRAM. В такой конфигурации контроллеры DMA пересылают данные в ядро напрямую, а табличные данные подгружаются в кэш данных по мере необходимости.

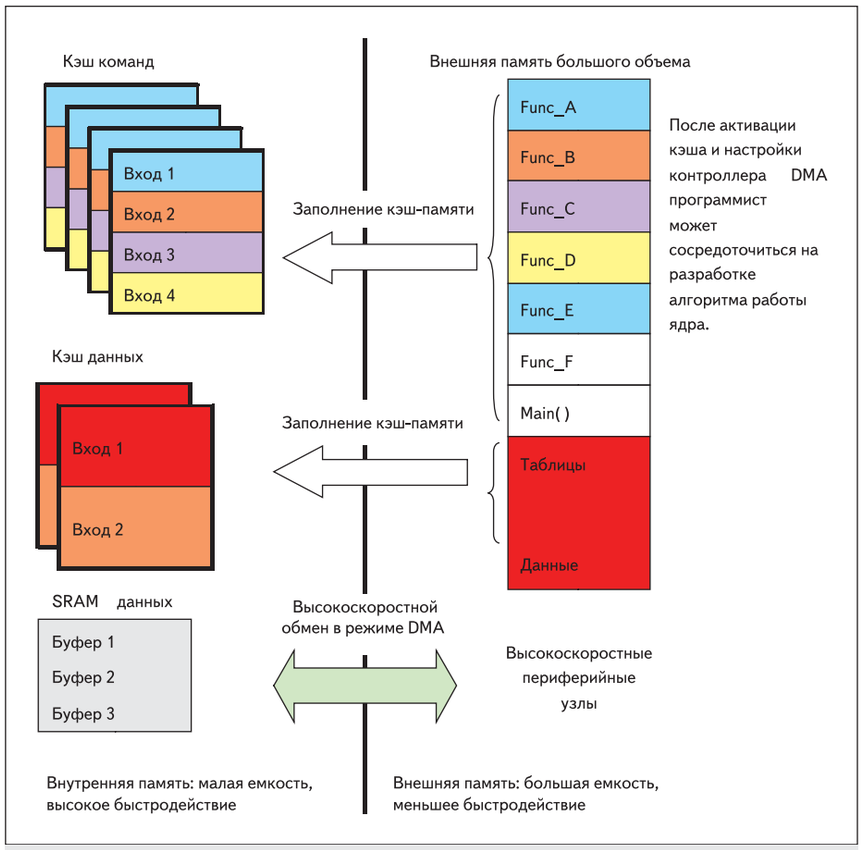

Рис. 3. Оптимизация процесса перемещения данных за счет применения конфигурируемой структуры кэша и памяти

Многофункциональные команды

Производительность процессоров обычно оценивается в миллионах команд, которые они могут выполнить в секунду (MIPS, millions instructions per second). Однако из-за разночтений в определении того, что представляет собой команда, для современных процессоров такой способ оценки не очень подходит. Например, многофункциональные команды, которые раньше можно было встретить только в дорогих параллельных процессорах, теперь доступны и в недорогих процессорах с фиксированной точкой. С помощью таких команд процессоры могут в одном процессорном цикле выполнять несколько операций АЛУ/умножителя-накопителя (MAC), а также операции загрузки данных из памяти и сохранения данных в память. Память в процессоре обычно разбита на «суббанки»,

и к любому из них в одном процессорном цикле возможны два обращения ядра, а также обращение контроллера DMA. Если добавить к этому описанные ранее возможности аппаратного вычисления адресов, то становится очевидным, что за один процессорный цикл процессор может выполнить достаточно много действий.

Пример многофункциональной команды приведен на рис. 4. Как показано на нем, в одном процессорном цикле помимо двух независимых операций умножения-накопления также может выполняться выборка данных из памяти и сохранение данных в память.

Конвейер с внутренней блокировкой

По мере увеличения скорости процессора неизбежно возрастает глубина (количество уровней) конвейера. Это очень важно понимать, поскольку эффекты конвейера могут сильно усложнить написание программ на языке ассемблера. Некоторые процессоры имеют конвейер с внутренней блокировкой ("interlocked" pipeline). В этом случае при написании программы на языке ассемблера программисту не потребуется заниматься планированием или отслеживанием продвижения данных и команд по конвейеру — процессор автоматически отрабатывает остановы конвейера и другие ситуации, при которых последовательный процесс исполнения программы нарушается.

Гибкий регистровый файл данных

И, наконец, еще одной особенностью современных ЦСП является наличие универсальных регистров данных. В традиционных ЦСП с фиксированной точкой размер слов обычно фиксирован. В то же время, возможность работы с регистрами данных, в зависимости от задачи, как с одним 32-разрядным словом (например, R0), либо как с двумя 16-разрядными словами (R0.L и R0.H — младшая и старшая половины регистра соответственно) дает свои преимущества. В системах с двумя MAC это свойство позволяет в одном процессорном цикле выполнять операции над четырьмя 16-разрядными операндами.

Анализ и сравнение кода

Описанные в предыдущем разделе архитектурные особенности являются основой для эффективного программирования ЦСП. Многие распространенные алгоритмы, ориентированные на решение числовых задач, могут работать экстремально быстро, если программист полностью использует потенциальные возможности процессора. В этом разделе обсуждается реализация на базе ЦСП нескольких типовых алгоритмов обработки сигналов. Эффективность кода, естественно, следует оценивать на ассемблерных программах, однако в современных оптимизирующих компиляторах ЦСП применяются многие из

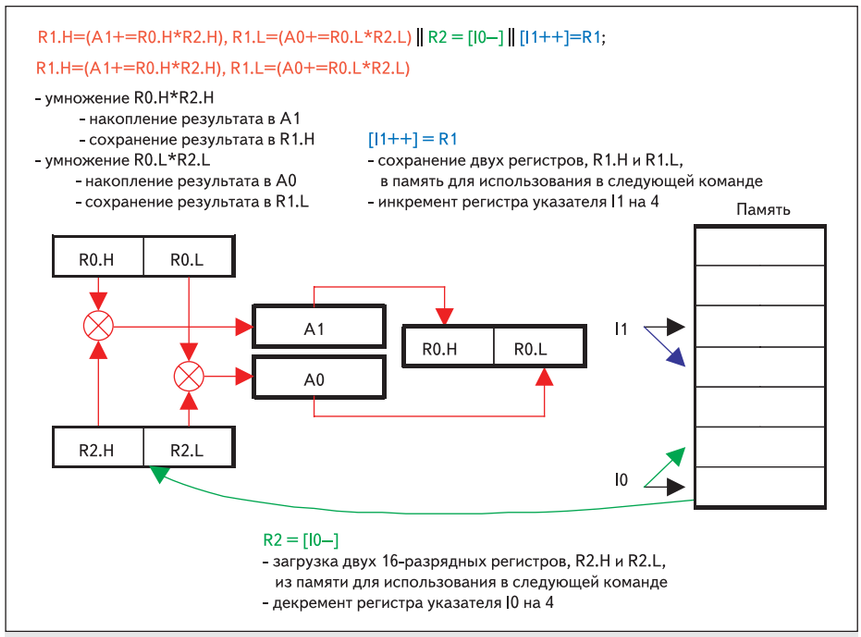

Рис. 4. Выполнение нескольких операций за один процессорный цикл при помощи многофункциональных команд процессора Blackfin

тех правил, которыми руководствуется программист, пишущий на языке ассемблера. В приводимых примерах для иллюстрации использовался язык ассемблера процессоров Blackfin.

Скалярное произведение

Скалярное произведение — это операция, позволяющая измерить степень ортогональности двух векторов. Большинству C-программистов знакома следующая реализация скалярного произведения:

```
short dot(short a[], short b[], int size)
{
int i; int output = 0;
for(i=0; i<size; i++)
{ output += (a[i] * b[i]); }
return output;
}
```

Основная часть программы на языке ассемблера выглядит следующим образом:

```
//P0 = счетчик циклов, I0 и P1 — регистры адреса
A1 = A0 = 0;                          //A0 и A1 — аккумуляторы
LSETUP(loop1,loop1) LC0 = P0;  //Настройка аппаратного цикла,
                                      //начинающегося с метки loop1
loop1: A1 += R1.H * R0.H, A0 += R1.L * R0.L || R1 = [P1 ++] || R0
= [I0 ++] ;
```

Отметим архитектурные особенности ЦСП, за счет которых достигается такой малый объем кода.

Аппаратные буферы и счетчики циклов устраняют необходимость применения команд перехода в конце каждой итерации. Поскольку скалярное произведение представляет собой сумму результатов отдельных произведений, оно реализуется с помощью цикла. Во многих RISC-микроконтроллерах для начала следующей итерации цикла в конце каждой предыдущей итерации необходима команда перехода. В приведенной программе на языке ассемблера для организации цикла требуется всего одна команда — LSETUP.

Многофункциональные команды позволяют в одном процессорном цикле выполнять две арифметические команды и два обращения к данным. На каждой итерации цикла необходимо извлечь из памяти значения a[i] и b[i], перемножить их и, наконец, прибавить результат к текущему значению суммы. На многих микроконтроллерных платформах для выполнения этих операций потребовалось бы четыре команды. Последняя строка программы на языке ассемблера показывает, что в ЦСП все эти операции могут быть выполнены за один процессорный цикл.

Параллельные операции АЛУ позволяют одновременно выполнять две 16-разрядные команды. В приведенной программе на языке ассемблера на каждой итерации цикла используются два аккумулятора (A0 и A1). За счет этого число необходимых итераций уменьшается на 50%, и, следовательно, время, затрачиваемое на выполнение цикла, уменьшается вдвое.

Фильтр с конечной импульсной характеристикой

Фильтр с конечной импульсной характеристикой (КИХ-фильтр) — это очень распространенный тип фильтра, выходной сигнал которого представляет собой свертку отсчетов входного сигнала с коэффициентами фильтра. Прямая реализация фильтра на языке C очень похожа на программу для определения скалярного произведения:

```
//отсчет сигнала помещается в циклический буфер
x[cur] = sampling_function();
cur = (cur+1)%TAPS; // циклическое смещение указателя cur

//выполнение умножения-накопления
y = 0;
for(k=0;k<TAPS;k++)
{
y += h[k] * x[(cur+k)%TAPS];
}
```

Основная часть программы КИХ-фильтра на языке ассемблера похожа на программу, реализующую скалярное произведение. Действительно, в ней для достижения максимальной скорости выполнения применяются те же архитектурные особенности ЦСП. В данном конкретном примере регистр R0 используется для промежуточного хранения отсчетов сигнала, а регистр R1 — коэффициентов фильтра:

```
//P0 содержит количество звеньев фильтра
R0=[I0++] || R1=[I1++];          //Инициализация R0 и R1
A1=A0=0;                         //Обнуление аккумуляторов
LSETUP (loop1, loop1) LC0 = P0; //Настройка внутреннего цикла
loop1: A1+=R0. L*R1.L, A0+=R0.H*R1.H || R0 = [I0++] || R1 =
[I1++];                          //Вычисление
```

Помимо архитектурных особенностей процессора, отмеченных для программы скалярного произведения, в программе КИХ-фильтра также используется циклическая буферизация.

Поддержка циклических буферов устраняет необходимость явного применения арифметики по модулю. В языке С циклическая буферизация реализуется при помощи оператора % (оператор деления по модулю). Из приведенной программы на языке ассемблера следует, что этот оператор не транслируется в дополнительную команду в теле цикла. Вместо этого регистры I0 и I1 генератора адреса данных конфигурируются вне цикла таким образом, что при достижении границы буфера коэффициентов происходит автоматический возврат указателей к его началу.

Быстрое преобразование Фурье

Быстрое преобразование Фурье (БПФ) — это неотъемлемая часть многих алгоритмов обработки сигналов. Одна из его любопытных особенностей заключается в том, что при последовательном во времени порядке отсчетов входного массива выходной массив будет иметь бит-реверсный порядок. Для большинства традиционных процессоров общего назначения от программиста потребуется реализовать отдельную процедуру для переупорядочивания выходного массива. На платформе ЦСП поддержка бит-реверсии адреса встроена в генератор адресов.

Бит-реверсная адресация устраняет необходимость в отдельной процедуре бит-реверсии при реализации БПФ. Возможность автоматической аппаратной бит-реверсной адресации элементов выходного массива БПФ избавляет программиста от написания дополнительных подпрограмм, и, следовательно, приводит к росту производительности.

В дополнение к описанным в статье программным конструкциям некоторые процессоры имеют дополнительный набор специфических команд для поддержки широкого диапазона приложений. Назначение этих команд состоит в дальнейшем увеличении производительности таких алгоритмов, как кодирование Витерби, Хаффмэна, а также многих других процедур манипуляции битами.

Очевидно, что при выборе стратегии программирования для приложения на базе ЦСП необходимо учесть много факторов. В большинстве случаев использование языка C/C++ с эффективным компилятором/оптимизатором дает неплохие результаты, но максимальной производительности от процессора можно добиться, как правило, лишь при написании программы на языке ассемблера вручную. Однако делать это стоит только при условии хорошего понимания принципов работы архитектурных компонентов, обеспечивающих эффективное кодирование.