УДК 681.3.06 (07)
У 912

№ 3451-2

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ
РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное агентство по образованию

Государственное образовательное учреждение
высшего профессионального образования

ТАГАНРОГСКИЙ ГОСУДАРСТВЕННЫЙ
РАДИОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

Учебно-методическое пособие

по курсу
АРХИТЕКТУРА И ПРОГРАММИРОВАНИЕ СИГНАЛЬНЫХ ПРОЦЕССОРОВ

РАЗРАБОТКА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ ПРОЦЕССОРОВ ЦИФРОВОЙ ОБРАБОТКИ СИГНАЛОВ

Часть 2

КАФЕДРА МАТЕМАТИЧЕСКОГО ОБЕСПЕЧЕНИЯ И ПРИМЕНЕНИЯ ЭВМ

ФАВТ

Для студентов специальностей 230105, 010503

Таганрог 2005

УДК 681.3.06 (076.5) + 681.325.5. (076.5)

Составитель Н.Ш. Хусаинов

Учебно-методическое пособие по курсу “Архитектура и программирование сигнальных процессоров". Разработка программного обеспечения процессоров цифровой обработки сигналов. Часть 2.-Таганрог: Изд-во ТРТУ, 2005.-76с.

Пособие посвящено вопросам разработки программного обеспечения процессоров цифровой обработки сигналов (ЦОС). Во второй части рассматриваются основы разработки программного обеспечения на языках высокого уровня C/C++ для сигнальных процессоров. Изучаются общие методики и приемы анализа и оптимизации программного кода для процессоров с RISC-архитектурой. Затрагиваются особенности стандарта Embedded C++ и отличительные особенности компилятора cc21k фирмы Analog Devices для процессоров семейства SHARC ADSP, его настройки и использование в среде VisualDSP++.
Значительное внимание уделяется описанию среды исполнения (run-time environment) программы, разработанной с использованием компилятора cc21k. Детально изучаются вопросы взаимодействия подпрограмм на C и на ассемблере. Рассматриваются приемы "ручной" и автоматической оптимизации исходного и объектного кода с использованием директив компилятора и результатов профилирования.
Пособие предназначено для студентов специальностей 220400, 351500, изучающих курс "Архитектура и программирование сигнальных процессоров". Представляет интерес для инженеров, разработчиков и программистов в области ЦОС, параллельного программирования, машинно-ориентированного и HLL-программирования для встроенных систем и систем реального времени, занимающихся проектированием систем ЦОС и разработкой программного обеспечения для них.

Ил.6. Библиогр.: 16 назв.

Рецензент В.Е.Золотовский, д-р техн. наук, профессор кафедры ВТ ТРТУ.

Хусаинов Наиль Шавкятович

Учебно-методическое пособие
по курсу
АРХИТЕКТУРА и программирование
сигнальных процессоров

РАЗРАБОТКА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ ПРОЦЕССОРОВ ЦИФРОВОЙ ОБРАБОТКИ СИГНАЛОВ

Часть 2

Для студентов специальностей 230105, 010503

Ответственный за выпуск Хусаинов Н.Ш.
Редактор Лунева Н.И.
Корректоры: Селезнева Л.И., Чиканенко Л.В.

ЛР № 020565 от 23.06.1997г. Подписано к печати . .2005г.
Формат 60x84 1/16 Бумага офсетная.
Офсетная печать. Усл.п.л. – 4,5. Уч.-изд. л. – 4,3.
Заказ № Тираж 150 экз.

"С"

Издательство Таганрогского государственного
радиотехнического университета
ГСП 17А, Таганрог, 28, Некрасовский, 44

Типография Таганрогского государственного
радиотехнического университета
ГСП 17А, Таганрог, 28, Энгельса, 1

1. Особенности оптимизации программного кода для RISC-процессоров

1.1. Задачи и возможности оптимизации программного кода для встроенных систем и RISC-процессоров

Трансляторы языков программирования прошли долгий путь эволюционного развития, реализуя самые различные способы повышения эффективности генерируемого кода, от использования различных методик низкоуровневой оптимизации до поддержки разнообразных модификаций и диалектов языков высокого уровня. Хотя появление стандартов языков программирования, таких как ANSI C, привело к тому, что компиляторы иногда становятся "предметами широкого потребления" со всеми присущими данному эпитету качественными характеристиками, можно выделить два фактора, которые являются определяющими в развитии технологий генерации и оптимизации кода в компиляторах. Один из них определяется различиями в архитектурах процессоров: наличие сложных многоступенчатых схем выполнения инструкций в конвейерах, кэши данных и команд, особенности распараллеливания потоков команд приводят к тому, что производительность современных процессоров гораздо сильнее, чем у их предшественников, зависит от структуры объектного кода, сгенерированного компилятором. Второй отражает современные требования рынка программного обеспечения. С появлением у разработчиков ПО новых запросов либо по повышению производительности кода, либо по сокращению его объема создатели компиляторов вынуждены разрабатывать и реализовывать новые, отвечающие этим требованиям, технологии и механизмы их использования.

Общими задачами, решаемыми компиляторами языков высокого уровня (ЯВУ) при разработке встроенных приложений, являются:

- повышение производительности исполняемого кода. Использование "тонких" настроек и методик оптимизации позволяет получить код, который выполняется на 20-30 % эффективнее программы, сгенерированной компилятором в режиме "по умолчанию". С учетом того, что во встроенных приложениях (в отличие от desktop-приложений) большую часть времени (более 90 %) процессор "выполняет" задачу пользователя, а не обслуживает операционную систему, такой выигрыш является весьма значительным;
- снижение стоимости системы в целом. Получение более быстродействующего кода позволяет использовать процессор с меньшей производительностью (и ценой). Уменьшение объема сгенерированного кода дает возможность снизить объем устанавливаемой в системе памяти. Вместе эти факторы могут оказать существенное влияние на стоимость системы;
- сокращение длительности цикла разработки программного обеспечения (ПО). Повторное использование процедур и модулей, разработанных на ЯВУ, более целесообразно, эффективно и просто, чем на ассемблере.

Исторически CISC-процессоры являются более широко распространенными по сравнению с RISC-процессорами и доминируют во многих приложениях, в том числе во встроенных системах. Соответственно можно говорить о том, что технология построения компиляторов с языков высокого уровня для CISC-процессоров в достаточной степени отлажена и позволяет получать приемлемый по эффективности исполняемый код.

В связи с постоянным повышением сложности задач, решаемых встроенными приложениями, разработчики подобных систем постепенно переходят от использования CISC-процессоров к использованию RISC-процессоров. Главным аргументом такого перехода является стремление повысить производительность системы. Но, как известно, производительность системы определяется не только быстродействием процессора, но и эффективностью исполняемого на нем программного кода. Компиляторы же для RISC-процессоров пока "отточены" в меньшей степени. Поэтому вполне вероятна генерация ими такого кода, который нивелирует все преимущества архитектуры процессора.

Следует также иметь в виду, что по сравнению с CISC-процессорами RISC-архитектура позволяет получить значительно больше преимуществ от использования определенного набора методик оптимизации кода компилятором.

Цели оптимизации кода и способы ее достижения для CISC- и RISC-процессоров существенно различаются вследствие различий в архитектурах процессоров. Наиболее очевидным различием (с точки зрения генерации кода) является различие в системе команд, которая у RISC-процессора обычно значительно беднее (что среди прочего приводит к "разрастанию" кода и повышает требования к объему памяти для хранения программы). Главным преимуществом сокращенной системы команд является возможность их более быстрого декодирования и выполнения. Зачастую все инструкции RISC-процессоров выполняются за один такт и длина всех инструкций одинакова.

В процессорах CISC-архитектуры различия в длительности выполнения инструкций во многом определяются местом размещения операндов. В отличие от RISC-процессоров, выполняющих вычисления обычно только над данными в регистровом файле, CISC-архитектура поддерживает возможность выполнения операций над операндом в памяти (естественно, при этом процессор загружает операнд, выполняет его модификацию и записывает обратно, но с точки зрения программиста для кодирования всех этих операций используется одна инструкция). В зависимости от размещения операндов и результатов в CISC-процессоре варьируется и длительность выполнения инструкции. Например, простейшая команда модификации операнда константой в процессоре Intel семейства 80x86 может занимать от 1 (операнд в регистре) до 3-х тактов (операнд в памяти).

Если сравнить длительности (в тактах) выполнения той же операции модификации в CISC-процессорах фирмы Intel и RISC-процессорах фирмы Analog Devices, то получается следующая картина:

R1 = dm(X); R2 = 10; R1 = R1 + R2; dm(X) = R1;	add X, 10

Несмотря на то, что команды в RISC-процессоре выполняются быстрее, обедненный набор режимов адресации к операндам в вычислительных инструкциях может свести на нет эти преимущества в случае необходимости интенсивных обменов данными между процессором и памятью. Поэтому одной из основных задач оптимизации кода для RISC-процессоров является сокращения числа инструкций чтения/записи памяти.

Для этого RISC-процессоры обычно содержат большее количество регистров общего назначения, что дает возможность хранить часто используемые переменные в регистровой памяти, а не обращаться за ними постоянно к внешней памяти или памяти на кристалле. Широко распространены также теневые (альтернативные) регистры, позволяющие быстро переключать контекст задачи без необходимости сохранения/восстановления из памяти большого числа регистров, например, при вызове или возврате из обработчика прерывания.

Еще одним элементом архитектуры (правда, уже общим для RISC- и CISC-процессоров) является многоступенчатый конвейер команд, позволяющий выполнять одновременно выборку, декодирование и выполнение инструкций. Для минимизации задержек, связанных с выборкой команд, компилятор должен оптимизировать загрузку конвейера путем возможного переупорядочивания инструкций.

1.2. Этапы компиляции и генерации программного кода и возможности оптимизации

Вспомним кратко этапы компиляции программы и возможности оптимизации генерируемого кода на каждом из них.

"Типичный" компилятор состоит из двух частей:

- парсер (сканер и синтаксический анализатор), представляющий собой в значительной мере процессорно-независимую часть компилятора (front-end). Этот блок выполняет семантический и синтаксический анализ исходной программы на языке высокого уровня, проверяет правильность конструкций программы. Если программа составлена корректно, генерируется машинно-независимый код на промежуточном языке, определяемом разработчиком компилятора. Одна из составляющих этого этапа – разделение кода на часть, зависящую только от конкретного языка программирования, и часть, зависящую от архитектуры процессора;

- генератор программного (исполняемого или объектного) кода (back-end) получает в качестве входных данных программный код на промежуточном языке и генерирует объектный код для конкретной архитектуры процессора. Обычно вначале выполняется целый ряд алгоритмов глобальной оптимизации

промежуточного кода с учетом особенностей процессора и параметров компиляции, например, числа доступных регистров. Затем генерируется объектный код, после которого вновь выполняется набор процедур оптимизации, но уже нацеленных на повышение эффективности сгенерированного ассемблерного кода. Эта постоптимизация может быть как глобальной, так и локальной. Back-end компилятор включает в себя несколько этапов, показанных на рис.1.

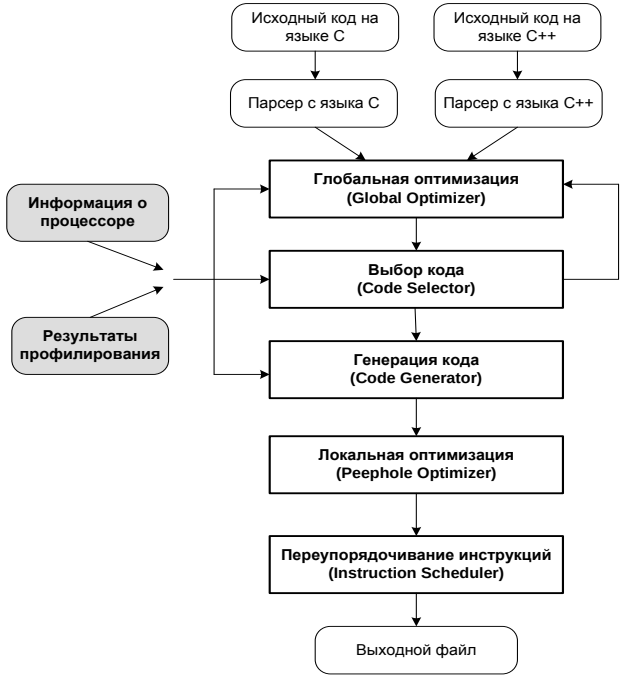

Рис.1. Этапы компиляции и генерации программного кода

Фактически оптимизация выполняется на каждом из этапов, но наибольшее значение имеет глобальная оптимизация. Global Optimizer выполняет "высокоуровневый" анализ программы и ряд общих оптимизаций. Эти общие оптимизации еще называют оптимизациями по базе данных (database-driven) или параметрическими (parameter-driven) оптимизациями: например, оптимизатор использует нечто вроде базы данных, чтобы определить количество доступных регистров для заданной модели процессора.

Блоки выбора и генерации кода (Code Selector и Code Generator) используют базу данных, похожую на базу данных "глобального оптимизатора" для выбора ассемблерных инструкций, реализующих ту или иную конструкцию промежуточного языка. На этапе выбора кода резервируются рабочие (Scratch) регистры для хранения временных переменных. Основная задача генератора кода – окончательно "утвердить" распределение регистров таким образом, чтобы

все неиспользуемые scratch-регистры хранили бы значения переменных (переменные, для которых не хватило места в регистрах, придется считывать из памяти).

После распределения регистров становятся очевидными зависимости между инструкциями. На этом этапе запускается локальный (peephole) оптимизатор, который сначала устраняет очевидные избыточности, а затем анализирует и оптимизирует явно неэффективные последовательности команд. Он работает совместно с блоком Instruction Scheduler для переупорядочивания инструкций таким образом, чтобы снизить вероятность появления "заторов" (run-time pipeline stalls) в конвейере (например, когда инструкции в одном конвейере не могут продвигаться дольше до окончания выполнения какой-то из команд в другом конвейере – типичная проблема для суперскалярных процессоров).

1.3. Обзор методик оптимизации кода для RISC-процессоров

Как упоминалось выше, преимущество по производительности при использовании RISC-процессоров за счет "быстрого" выполнения более простых инструкций может быть достигнуто лишь в случае постоянной загрузки исполнительного устройства. При частых простоях вследствие ожидания чтения инструкций или данных из памяти производительность процессора существенно снижается. Задача компилятора – минимизировать подобные нежелательные явления.

Рассмотрим некоторые способы оптимизации программного кода.

1.3.1. Анализ алиасов

Анализ алиасов (Alias Analysis) по своей сути не является методом оптимизации, однако позволяет повысить эффективность других алгоритмов оптимизации. Анализ предназначен для отслеживания обращения по указателю к той или иной переменной.

```
int *p;
int x,y;
…
y = 1;
*p = 10;
x = y;
```

Без выполнения анализа алиасов, отслеживающего адрес, на который указывает указатель `p`, компилятор не может знать, модифицирует ли команда `*p=10` переменную `y`? Соответственно, он не может заменить последнюю инструкцию на более простую команду `x=1` (для ее выполнения не нужно читать

из памяти переменную `y`). Если выполненный анализ позволяет сделать вывод о том, что p не указывает на `y`, то компилятор заменяет последнюю инструкцию.

Можно отметить, что для многих операций присваивания, которые очевидно не связаны по тексту исходной программы, аналогичный вывод по коду на промежуточном языке является менее очевидным. Хорошим примером является доступ к элементам массива.

```
int x, y, j;
x = 5;
a[j] = 0;
y = x;
```

Для программиста совершенно очевидно, что операция `a[j]=0` не может повлиять на значение переменной `x`. Но реализация массива обычно выполняется с использованием указателей, например, `*(a+j)=0`. Без использования анализа алиасов компилятор не сможет заменить последнюю инструкцию на `y=5` (кстати, использование анализа поможет только в том случае, если компилятор сможет однозначно определить диапазон изменения индекса `j`).

Результаты анализа используются при оптимизации порядка выполнения инструкций, для применения методик распространения констант, распространения копирования, удаления общих подвыражений.

Распространение констант (constant propagation) выполняется путем замены переменной в выражении ее константным значением. Например, фрагмент

```
int x, y, z;
x = 2;
y = x * z;
```

после распространения констант будет выглядеть так:

```
int x, y, z;
x = 2;
y = 2 * z;
```

Этот элемент оптимизации сокращает время вычислений и число обращений к памяти. В частности, он позволяет оценивать некоторые условные выражения на этапе компиляции и избежать использования команд проверки условия и особенно условных переходов (и потерь времени на их выполнение).

Распространение копии осуществляется путем использования в выражениях только одной переменной с данным значением, что позволяет облегчить компилятору решение задачи о том, какие переменные должны находиться в регистрах, а какие – храниться в памяти.

```
int x, y, z;
x = z;
y = x;
```

После применения

```
int x, y, z;
```

```
x = z;
y = z;
```

После первого чтения переменной z из памяти оно может быть помещено в регистр и использоваться в последней операции присваивания, исключая тем самым лишнюю операцию чтения переменной x из памяти.

При *устранении общих подвыражений* результат вычисления подвыражения сохраняется в регистре как временное значение и может быть использован при вычислении других выражений. Это позволяет сэкономить на отказе от повторных вычислений и чтений из памяти используемых в подвыражении переменных.

```
int a, b, c, x, y, z;
x = a * b;
y = a * b + 5;
z = a * b * c;
```

После оптимизации

```
int a, b, c, x, y, z;
register int tmp;
x = tmp = a * b;
y = tmp + 5;
z = tmp * c;
```

Понятно, что эффективность описанных выше методик зависит от того, сколько раз компилятор сможет их применить. Анализ алиасов позволяет сделать более "прозрачными" случаи, когда запись нового значения в ячейку памяти, на которую указывает указатель, изменяет какую-либо переменную из тех, что компилятор собирается "распространять". Это дает возможность более широко применять рассмотренные методики.

1.3.2. Передача параметров подпрограмм в регистрах

В CISC-процессорах параметры в процедуры обычно передаются через программный стек (в памяти) в порядке, согласованном между вызываемой и вызывающей процедурами. Наличие большого регистрового файла в большинстве RISC-процессоров позволяет передавать часть параметров через регистры (Passing Parameters in Registers, PPIR). Такой подход обычно повышает производительность программы как при передаче параметров (не нужно обращаться к памяти, чтобы затолкнуть операнды в стек), так и при обращении к ним в вызванной подпрограмме (не нужно обращаться к памяти, чтобы прочитать операнды из стека; к тому же операции могут выполняться непосредственно над полученными регистрами).

Обычно для передачи параметров используется не больше определенного количества регистров (например, трех). Все остальные операнды передаются через стек.

Есть ситуации, когда передача параметров через стек реально может привести к замедлению выполнения программы. Организация сложных вычислений требует использования большого числа регистров и для передачи параметров в подпрограмму может просто не хватить свободных регистров. В этом случае регистры будут сохранены в памяти и восстановлены после возврата из подпрограммы, что, естественно, потребует дополнительных операций доступа к памяти. К счастью, такой сценарий для большинства современных RISC-процессоров является нетипичным.

1.3.3. Разворачивание циклов

Циклы представляют собой структуры, трудно поддающиеся оптимизации. В зависимости от архитектуры процессора, компилятор может сократить количество итераций цикла, выполнив его "разворачивание" (Loop Unrolling). Такой прием дает два преимущества:

- приводит к уменьшению числа проверок условия окончания цикла и выполнения условных переходов по сравнению с числом "полезных" команд тела цикла;
- увеличивает размер блока инструкций, доступных для локальной оптимизации путем переупорядочения потока команд.

Разворачивание цикла – это способ оптимизации, при котором эффективность выполнения программы повышается за счет увеличения объема программного кода.

Количество повторений тела цикла называется фактором (коэффициентом) разворачивания цикла (unroll factor).

```
int x;
func (int n)
{
  int j;
  for (j=0; j<n; j++)
    x += x;
}
```

После разворачивания цикл будет выглядеть так:

```
for (j=0; j<n; j+=4)
{
  x += x;
  x += x;
  x += x;
  x += x;
}
```

Большинство компиляторов выполняют разворачивание циклов с фактором 2 или 4 (4 всегда лучше!). В случае, если количество итераций цикла n неизвестно на этапе компиляции, то необходима дополнительная проверка,

гарантирующая выполнение точно заданного числа итераций. В конечном итоге разворачивание цикла должно быть выполнено компилятором с учетом как выигрыша от разворачивания цикла, так и проигрыша от вставки дополнительных проверок.

1.3.4. Межпроцедурный анализ

Межпроцедурный анализ (Interprocedural Analysis, IPA) представляет собой более "агрессивную" методику оптимизации кода, обрамляющего вызов подпрограммы. При этом используется технология, похожая на анализ алиасов и позволяющая компилятору максимально "быть в курсе" того, какие переменные и регистры модифицируются в конкретном участке кода. Если анализ алиасов предоставляет сведения, касающиеся возможной адресации указателя на ту или иную область памяти, IPA анализирует возможность модификации регистра или переменной в результате вызова функции.

```
int a[10], b[10], c;
void bar()
{
  c++;
}
void foo()
{
  int i;
  for (i=1; i<10; i++)
    {
       b[0] =b[0] + a[i];
       bar();
}   }
```

Результат компиляции цикла без IPA-оптимизации (на промежуточном языке):

```
                …
                r5 = 1
loop_start:
                r6 = r5 * 2
                load r1, b
                load r2, a[r6]
                r1 = r1 + r2
                save b, r1
                call bar
                r5 = r5 + 1
                cmp r5, 10
                if less jump loop_start
                …
```


Результат компиляции цикла с IPA-оптимизацией (на промежуточном языке):

```
                …
                load r1, b
                r5 = 1
loop_start:
                r6 = r5 * 2
                load r2, a[r6]
                r1 = r1 + r2
                call bar
                r5 = r5 + 1
                      cmp r5, 10
                if less jump loop_start
                save b, r1
                …
```

Использование IPA-анализа позволило отказаться от постоянного сохранения значения `b[0]` в память, поскольку компилятор имеет информацию о том, что функция `bar()` использует только глобальную переменную `c` и не работает с `b[0]`. Это привело к повышению производительности программы и сокращению программного кода.

Залогом эффективности IPA является способность просматривать содержимое всех файлов проекта и библиотек, поскольку зачастую вызов функции и ее описание может находиться в различных файлах.

В компиляторах различных разработчиков "акценты" IPA-оптимизации могут различаться. Например, для рассматриваемого ниже компилятора cc21k одними из основных задач IPA-оптимизации являются исключение из программы операций работы с неиспользуемыми переменными, а также анализ возможностей генерации кода с аппаратной поддержкой циклов.

1.3.5. Оптимизации на уровне архитектуры процессора или машинных команд

Каждый компилятор разрабатывается под конкретный процессор (или семейство процессоров) и его неотъемлемой чертой является завершающая оптимизация кода с учетом особенностей системы команд конкретного процессора (Machine Specific Optimizations). Вариантов такой оптимизации может быть очень много, они определяются особенностями архитектуры процессоров и системы команд, но их общими основными направлениями являются:

- сокращение накладных расходов на условные переходы: отказ от специальных команд проверки условий (типа `cmp` в ассемблере Intel), анализ флагов и выполнение условных переходов непосредственно по результатам

арифметических и логических команд; выполнение операций по условию вместо выполнения условных переходов;

До машинной оптимизации на уровне команд	После машинной оптимизации на уровне команд
a = a - 1; if ZERO jump 3: r1=2; r2 = 11; jump 4; 3: r1 = 3; 4: …	a = a - 1; if ZERO r1 = 3 if NOTZERO r1=2 if NOTZERO r2=11 …

- замена доступа к элементам массива по индексам на доступ по указателям;

До машинной оптимизации на уровне команд	После машинной оптимизации на уровне команд
p[0] = 0; p[1] = 1; p += 2;	*p++ = 0; *p++ = 1;

Оптимизированный код более эффективен, поскольку не требует при каждом доступе к памяти каждый раз вычислять смещение относительно базового адреса начала массива элемента, соответствующего текущему индексу;

- замена условий в циклах по счетчику. Если переменная цикла не используется в теле цикла в качестве операнда, то для проверки условия завершения цикла вместо кода с проверкой переменной на равенство заданной константе выполнять сравнение ее с нулем:

До машинной оптимизации на уровне команд	После машинной оптимизации на уровне команд
for (i=0; i<100; i++) a += random();	for (i=100; i>0; i--) a += random();

Реализация второго варианта цикла всегда будет более эффективна, поскольку не требуется отдельная команда для сравнения с нулем. Каким является значение счетчика цикла после декрементирования можно определить сразу же по флагам. Это типичный пример Code Selection-оптимизации, когда две инструкции проверки и перехода заменяются на одну;

- попытка одновременного выполнения нескольких вычислительных операций в различных вычислительных устройствах (пример – многофункциональные инструкции в процессорах SHARC ADSP).

1.3.6. Управление порядком выполнения инструкций

Планировка инструкций (Instruction Scheduling) является элементом peephole-оптимизации. Поскольку для ее реализации необходима информация о зависимостях между инструкциями, такая планировка выполняется после всех других видов оптимизации.

Главная задача планирования потока команд заключается в обеспечении постоянной и непрерывной работы конвейера инструкций. Поскольку каждый процессор имеет собственную архитектуру конвейера, зачастую включающую сложную логику для предсказания ветвлений или когерентности данных, рассмотрим лишь базовые принципы планирования потока команд.

Конвейеризация эффективна только тогда, когда загрузка конвейера близка к полной, а скорость подачи новых команд и операндов соответствует максимальной производительности конвейера. Если произойдет задержка, то параллельно будет выполняться меньшее количество операций и суммарная производительность снизится.

Поскольку в конвейере выборка, декодирование и выполнение нескольких инструкций осуществляется одновременно, проблемы могут возникнуть, например, когда операнд второй инструкции (регистр или ячейка памяти) является результатом первой инструкции. Эта проблема актуальна для процессоров, архитектура которых предполагает выполнение выборки данных и выполнение инструкции с записью результата на различных ступенях конвейера. В этом случае следующая инструкция не может перейти на следующую стадию, поскольку должна ожидать появления правильного значения в регистре или ячейке памяти. Для избежания этой ситуации компилятор может переупорядочивать поток инструкций, например, "разнося" во времени выполнение зависимых команд.

До peephole-оптимизации	После peephole-оптимизации
int x, y, z, a, *p; p = &a; x = 1; y = x; *p = z;	int x, y, z, a, *p; p = &a; x = 1; *p = z; y = x;

Вторая реализация является более эффективной, поскольку к моменту декодирования и обращения за данными инструкции `y=x`; значение `x` уже наверняка будет правильным.

Планирование потока команд может быть локальным или глобальным. При локальном планировании переупорядочиваются только инструкции внутри какого-либо фрагмента кода (например, тела цикла). При глобальном планировании планировщик может смотреть за пределы анализируемого блока. Глобальное планирование более важно для суперскалярных архитектур, в которых имеются несколько конвейеров с несколькими вычислительными устройствами и более актуальна задача расширения области, в пределах которой компилятор может изменять порядок выполнения инструкций.

Ключевым моментом для эффективного переупорядочивания инструкций является возможность компилятора отслеживать зависимость по данным между различными инструкциями (зависимости могут быть трех основных типов: RAW, WAW, WAR). Приведенный выше пример наглядно демонстрирует необходимость использования анализа алиасов. Если компилятор считает, что `p`

к моменту выполнения операции `*p=z;` может указывать на переменную `x`, то он не сможет выполнить переупорядочивание инструкций, аналогичное приведенному выше, поскольку это приведет к нарушению логики работы программы.

1.3.7. Оптимизация ветвлений

Одним из приемов оптимизации кода является организация сложной системы переходов для выполнения команд в теле цикла и/или выхода из цикла. Например, приведенный ниже программный код на языке С может быть преобразован компилятором следующим образом

Исходный код	Код после оптимизации
`flag = 0;` `while (!flag)` `{` `  ...aaa...` `  stml ();` `  ...bbb...` `  if (e1)` `  break;` `  ...ccc...` `  if (e2)` `  {` `  ...ddd...` `  flag = 1 ;` `  ...eee...` `  }` `}` `if (flag)` `{` `  ...ttt...` `  stm2 ();` `}`	`L1:` `  ...aaa...` `  stml ();` `  ...bbb...` `  if (e1)` `  goto L2;  /* через stm2,т.к. flag1=0 */` `  ...ccc...` `  if (!e2)  /* if !e2 в начало */` `  goto L1;` `  ...ddd...` `  ...eee...` `  ...ttt...` `  stm2 ();` `L2:`

Компилятор определяет, что вход в цикл выполняется в любом случае, и поэтому функция `stml()` будет вызываться всегда. Компилятор также определяет, что `stm2()` никогда не будет выполнена, если `flag=0` и если `e1=1` – тогда `flag` "навсегда останется" равным "0". Эти сведения позволяют компилятору сгенерировать более эффективный код (естественно, при разработке программы приведенным справа "стилем" программирования пользоваться не рекомендуется).

1.3.8. Inline- функции

Inline-функции нашли широкое применение и поддерживаются всеми компиляторами, например языка программирования С/С++. Как известно, они

позволяют повысить производительность программы за счет замены вызова функции самим телом функции. Это может выглядеть, например, так:

Исходный код	Код после оптимизации
`void swap (int *a, int *b)` `{` `  int t = *a;` `  *a = *b;` `  *b = t;` `}` `main()` `{` `  swap(&x,&y) ;` `}`	`main()` `{` `  int t=x;` `  x = y;` `  y = t;` `}`

Обычно такой подход может приводить к увеличению объема кода, особенно при многочисленных вставках. Однако в программах на объектно-ориентированных языках зачастую встречаются функции, состоящие из одной-двух строк и команды возврата. В этом случае, даже если функция в программе не описана как `inline`, компилятор (при установке соответствующих опций) самостоятельно пытается "внедрить" ее тело вместо вызова, но только в том случае, если это не приведет к увеличению объема кода.

1.3.9. Оптимизация по результатам профилирования

При разработке встроенных приложений широко используется такой подход, как оптимизация по результатам профилирования (Profile-Guide Optimization) или прикладная оптимизация (Application Specific Optimization).

Типичной областью "приложения" такой оптимизации может служить обычный оператор вида `if ( ) goto`. Дело в том, что вследствие конвейерной обработки данных при проверке условия в команде if наилучшим вариантом (с точки зрения быстродействия) является невыполнение этого условия, так как в конвейере уже находятся инструкции, непосредственно следующие за `if`, и выполнение условия приведет к необходимости очистки конвейера при выполнении перехода и, как следствие, к снижению производительности.

Чтобы оптимизировать код, желательно записать в операторе `if ()` такое условие (прямое или обратное), которое менее вероятно при выполнении программы. Но как компилятор может оценить более вероятную ветвь программы? Поскольку в состав большинства современных средств разработки входит утилита профилирования, современные компиляторы "научились" анализировать данные, получаемые в результате профилирования. После запуска программы на наборах реальных данных с профилированием компилятор анализирует результаты профилирования и выполняет повторную компиляцию, оптимизированную под данный набор исходных данных.

Приведенный в данном разделе обзор методик анализа кода для повышения его эффективности и быстродействия, безусловно, является неполным и отражает лишь некоторые основные практические приемы, получившие широкое распространение в современных компиляторах, в частности, для процессоров фирмы Analog Devices.

2. Компилятор cc21k для процессоров SHARC ADSP

2.1. Общие сведения о компиляторе и его использовании

Основные задачи, решаемые компилятором C/C++ в интегрированной среде разработки VisualDSP++:

- обработка исходных C/C++ файлов и генерация кода в виде исполняемых машинных команд;
- размещение перемещаемого кода и отладочной информации в объектных файлах;
- формирование сегментов кода и данных для последующего размещения компоновщиком в физической памяти процессора.

Компилятор cc21k поддерживает стандарт Embedded C++, являющийся подмножеством стандарта C++ ANSI ISO/IEC 14882:1998 и предназначенный для разработки встроенных приложений (в том числе приложений ЦОС). Область применения разрабатываемых программ накладывает определенные требования к генерируемому коду: он должен быть компактным, максимально быстродействующим и детерминированным по времени исполнения. Фактически EC++ представляет собой "сокращенный" вариант стандарта ANSI C++, лишенный некоторых функциональных возможностей языка C++, которые могут существенно снизить быстродействие программы и/или увеличить объем занимаемой памяти (например, обработка исключений, шаблоны, таблицы виртуальных функций, множественное наследование и др.).

С другой стороны, как и любой компилятор, компилятор cc21k поддерживает собственные расширения стандарта EC++ (которые реализуют практически все возможности полного стандарта ANSI C++, в том числе обработку исключений и идентификацию системы во время выполнения, причем разработчики компилятора утверждают, что эта поддержка не приводит к снижению производительности генерируемого кода), а также специфические расширения стандарта ANSI C++ и языка C, связанные с архитектурой SHARC-процессоров.

Компилятор может вызываться как из интегрированной среды разработки VisualDSP++, так и из командной строки. Синтаксис командной строки для запуска компилятора:

```
cc21k [-ключ [-ключ …] исходный_файл [исходный_файл …]]
```


Компилятор обрабатывает исходные файлы программ на языке C/C++ и транслирует их в программы на языке ассемблера для SHARC-процессора. Файлы на ассемблере затем обрабатываются (автоматически) ассемблером easm21k с целью получения объектных файлов в стандартном формате ELF (Executable and Linkable Format), которые затем могут быть либо скомпонованы линкером в исполняемую программу, либо добавлены архиватором в архивную библиотеку. Выбор утилит обработки файлов и параметров их обработки осуществляется в зависимости от типа исходных файлов и опций проекта. Весь процесс получения исполняемого кода из программы на C/C++ контролируется компилятором (и в определенной степени компоновщиком) и зависит от типа исходных модулей (на каком языке написаны) и опций компилятора, задаваемых во вкладке Compile окна Project Options.

При компиляции файлов есть возможность задавать собственные параметры компиляции для каждого файла. В окне Project, если кликнуть на имени файла правой кнопкой мыши, то в pop-up меню надо выбрать File Options... Поддерживаются следующие варианты задания параметров компиляции:

- project-wide settings – используются глобальные параметры компиляции проекта;
- file-specific setting – можно на появившейся вкладке задать собственные параметры компиляции для данного модуля;
- custom build command – следует задать вызов компилятора посредством командной строки с явным указанием ключей в виде командной строки, выбора модулей для компиляции и указания имени выходного файла (фактически командная строка "с дружественным интерфейсом").

2.2. Поддерживаемые типы данных

Компилятор поддерживает обычные для C/C++ простые типы данных, однако их реализация в объектном коде выполняется с учетом архитектуры процессора, т.е. 32-разрядных регистров. Компилятор не поддерживает типы данных меньшие, чем адресуемая ячейка памяти процессора. Хотя это приводит к необычной эквивалентности размерности переменных типа `short int`, `int`, `long int` и `char` (все по 32 бита), но не противоречит стандарту ISO/IEC.

Следует сказать особо о представлении дробных операндов и операндов в формате с плавающей запятой (ПЗ). Тип данных `float` представляет собой 32-битовое значение в формате IEEE-1985. Тип double может быть либо 32-х, либо 64-битовым. Обработка 64-битовых ПЗ-данных выполняется с использованием программной эмуляции ПЗ-арифметики, что приводит к существенному снижению производительности вычислений. Поэтому более предпочтительным является использование данных в формате `float` или 32-битовых `double`,

поскольку в этом случае обработка ведется в "родном" для SHARC ADSP формате.

Тип	Размер в битах	Результат, возвращаемый оператором sizeof
int	32 (ФЗ знаковый)	1
unsigned int	32 (ФЗ беззнаковый)	1
long	32 (ФЗ знаковый)	1
unsigned long	32 (ФЗ беззнаковый)	1
char	32 (ФЗ знаковый)	1
unsigned char	32 (ФЗ беззнаковый)	1
short	32 (ФЗ знаковый)	1
unsigned short	32 (ФЗ беззнаковый)	1
pointer	32	1
float	32 (ПЗ)	1
double	32 или 64 (ПЗ)	1 или 2 (по умолчанию 1)
long double	64 (ПЗ)	2
fract	32 (в C++ - ФЗ дробный знаковый, в C – эмуляция через ПЗ)	1

2.3. Расширения языка C/C++

Компилятор cc21k поддерживает ряд расширений стандарта ANSI C++ и языка C. Эти расширения введены как с учетом архитектуры DSP-процессоров, так и для того, чтобы позволить программисту использовать некоторые возможности C++ при программировании на C.

2.3.1. Поддержка inline-функций

Inline-функции в C++ используются для вставки тела функции вместо ее вызова и исключения ее вызова/возврата, что повышает производительность вычислений. Поддержка inline-функций реализована в компиляторе и для программ на языке C. Объявление такой функции стандартно:

```
inline int add_values(int a, int b)
{
  return a+b;
}
```

Inline-функции имеют особое значение при генерации кода тела цикла. В связи с особенностями архитектуры процессора код для аппаратной поддержки цикла (обеспечивающей максимальную производительность) обычно может быть сгенерирован компилятором лишь в том случае, если внутри цикла нет инструкций вызова функции. Поэтому опциональным режимом работы

компилятора является автоматическая вставка функций, которые не объявлены как inline, если это может привести к повышению производительности программы.

2.3.2. Ассемблерные вставки

Конструкция вида asm( ) полезна при необходимости реализации низкоуровневых операций и достижения максимальной эффективности кода. Она присутствует практически во всех языках программирования, например:

```
asm ("R2 = 0;");
```

или

```
asm (    "R1 = 2;     \
         R3 = 4;"
    );
```

Компилятор не анализирует код внутри конструкции `asm()` и передает его напрямую ассемблеру. Единственное, что делает компилятор – выполняет макроподстановки параметров вместо `%0,...,%9`, если внутри `asm( )`-конструкции используется так называемый шаблон.

Одной из трудностей использования `asm`-вставок являются тонкости использования в ассемблерном коде различных классов регистров: как будет показано ниже, некоторые регистры компилятор использует монопольно и их нельзя модифицировать, некоторые доступны для использования с последующим восстановлением, а на использование остальных вообще нет ограничений.

Поскольку компилятор не анализирует код внутри `asm`-вставки, то вполне возможно "испортить" те регистры, которые задействованы компилятором для других целей, и он предполагает, что они находятся под его исключительным контролем. Поэтому реализацию `asm`-вставок целесообразно выполнять на основе шаблонов. При этом для программиста во многих случаях вообще отпадает необходимость явного указания имен регистров в ассемблерных инструкциях: ему достаточно указать процессорные регистры какого типа следует использовать и компилятор сам выберет "свободные" регистры и выполнит их подстановку.

Синтаксис шаблона для asm-конструкции выглядит следующим образом:

```
asm (
       template
       [ : [constraint (output_operand) [,constraint (output_operand)...]]
       [ : [constraint (input_operand) [,constraint (input_operand)...]]
       [ : clobber ]]]
     );
```

где `template` – строка в кавычках, содержащая ассемблерные инструкции с символами `%ЧИСЛО` вместо наименований регистров в тех позициях, куда компилятор должен сам подставить регистры (операнды нумеруются в порядке

появления слева направо, от 0 до 9); `constraint` – строка специального вида (в кавычках), указывающая компилятору на то, процессорные регистры какого типа следует использовать для каждого из входных/выходных операндов; `output_operand` – имя переменной программы C/C++, в которую следует записать результат ассемблерной инструкции; `input_operand` – имя переменной программы C/C++, из которой берется значение для выполнения ассемблерной инструкции; `clobber` – список регистров (каждый в кавычках, маленькими буквами), которые явно использованы (модифицированы) программистом в данной ассемблерной вставке, чтобы компилятор при необходимости сгенерировал дополнительный код по их сохранению и восстановлению.

Типы регистров

Литера	Тип регистров	Номера регистров
a	Регистр Bx из DAG2	b8 — b15
b	Регистр Rx 2-й группы регистрового файла (РФ)	r4 — r7
c	Регистр Rx 3-й группы РФ	r8 — r11
d	Rx-регистр	r0 — r15
e	Регистр Lx из DAG2	l8 — l15
F	Fx-регистр	F0 — F15
f	Регистр-аккумулятор умножителя	mrf, mrb
h	Регистр Bx из DAG1	b0 — b7
j	Регистр Lx из DAG1	l0 — l7
k	Регистр Rx 1-й группы РФ	r0 - r3
l	Регистр Rx 4-й группы РФ	r12 - r15
r	Регистр общего назначения	r0 — r15, i0 — i15, l0 — l15, m0 — m15, b0 — b15, ustat1, ustat2
u	Регистр пользователя	ustat1, ustat2 (+ ustat3, ustat4 для ADSP-2116x)
w	Регистр Ix из DAG1	I0 — I7
x	Регистр Mx из DAG1	M0 — M7
y	Регистр Ix из DAG2	I8 — I15
z	Регистр Mx из DAG2	M8 — M15
=& constraint	Данный операнд является результатом и не может перекрываться с каким-либо входным операндом	
=constraint	Данный операнд является результатом	

Использование других букв может привести к непредсказуемому поведению компилятора при выборе регистров.

Примеры

Исходный код на C	Сгенерированный код на ассемблере
`{` `int result, x, y;` `asm (` `"%0 = %1 + %2;"` `: "=d" (result) /* %0 → result */` `: "d" (x), "d" (y) /* %1 ← x, %2 ←y */` `:` `);` `}`	`r2=dm(_x);` `r1=dm(_y);` `r0 = r2 + r1;` `dm(_result)=r0;`
`{` `int result, x, y;` `asm (` `"r9 = %1; \` `r10 = %2; \` `%0 = r9+r10;"` `: "=d" (result) /* %0 → result */` `: "d" (x), "d" (y) /* %1 ← x, %2 ←y */` `: "r9", "r10"` `);` `}`	`modify(i7,-2);` `dm(-3,i6)=r9;` `dm(-2,i6)=r10;` `r2=dm(_x);` `r1=dm(_y);` `r9 = r2;` `r10 = r1;` `r0 = r9+r10;` `dm(_result)=r0;` `r9=dm(-3,i6);` `r10=dm(-2,i6);`

Особым случаем при использовании ассемблерных вставок на основе шаблонов является модификация исходных операндов. Если в качестве входного и выходного операнда просто указать одну и ту же переменную, то не гарантируется, что для чтения ее из памяти и записи значения в память будет использован один и тот же регистр:

```
asm ( "modify(%0,%2);"
      : "=w" (ptr_A)
      : "w" (ptr_A), "x" (a)
    );
```

Во избежание потенциальных проблем рекомендуется в таких случаях использовать номер выходного операнда вместо указания типа регистра для хранения входного операнда. В приведенной ниже модификации компилятор всегда будет размещать входной операнд `%1(ptr_A)` и выходной операнд `%0(ptr_A)` в одном и том же регистре:

```
asm ( "modify(%0,%2);"
      : "=w" (ptr_A)
      : "0" (ptr_A), "x" (a)
    );
```

Тем не менее это не решает проблему, возникающую, например, при выполнении операции доступа к памяти с постмодификацией указателя.

```
asm("%0=dm(%1,3);"
    : "=d" (res)
    : "w" (my_ptr)
   );
```

В самой ассемблерной вставке все корректно, но при генерации последующего кода компилятор "не может знать", что произошло изменение не только переменной `res`, но и указателя `my_ptr`. Значение указателя `my_ptr`, хранящегося в памяти, останется неизменным (будет модифицирован только регистр, использованный компилятором при загрузке `my_ptr` из памяти) и это может привести к неверному функционированию программы. Поэтому не рекомендуется использовать ассемблерные вставки с "неявным" изменением входных операндов.

Ограничения при использовании ассемблерных вставок:

- нельзя никаким образом передавать управление внутри asm-вставки (изменять ход выполнения программы);
- переменные C/C++ программы доступны только путем указания их в списке операндов/результатов;
- все изменяемые в явном виде регистры должны быть обязательно внесены в список clobber.

Препроцессор запускается перед компилятором и не просматривает вставки `asm()`. Поэтому, к сожалению, внутри вставок проблематично использовать названия регистров IOP-процессора, определенных в файле def21x60.h как константы адресов памяти.

2.3.3. Поддержка пространств памяти данных и памяти команд

Поскольку одной из ключевых особенностей архитектуры процессоров SHARC ADSP является разделение памяти на память данных и память команд, на этапе объявления всех имен (функций, переменных/объектов и указателей) компилятор должен знать, в каком пространстве памяти следует разместить эту переменную. Если объявляется указатель на область памяти, то необходимо задать дополнительно и пространство, на которое он указывает. По умолчанию компилятор придерживается следующих правил:

- все переменные размещаются в памяти данных;
- все функции всегда размещаются в памяти программ. Указатели на функции всегда указывают на адрес в памяти программ;
- локальные переменные всегда автоматически размещаются в стеке, который всегда располагается в памяти данных;
- для многоуровневых указателей возможно явное задание всех уровней памяти.

Компилятор поддерживает явное указание типа памяти, в которой размещается переменная или объект, если это не противоречит рассмотренным выше правилам, например:

```
int varstst;          // размещается по умолчанию в DM
int pm varb1;         // размещается в PM
float dm buf2[200];   // размещается в DM
```



```
int *xy;              // xy размещается в DM и указывает
                      // на DM
int pm * dm ddd;      // ddd размещается в DM и
                      // указывает на PM
int pm * asd;         // asd размещается в DM и
                      // указывает на PM
float pm * pm xxy;    // xxy размещается в PM и
                      // указывает на PM
int dm * pm uty;      // uty размещается в PM и
                      // указывает на DM
```

Компилятор отслеживает недопустимое с его точки зрение смешивание адресов и не позволяет присвоить указателю на DM-ячейку адрес, находящийся в PM.

```
int pm x;
int dm y;
x = y;                // все нормально

int pm *x;
int dm * y;
int dm a;
x = y;                // недопустимо
x = &a;               // недопустимо
```

Для глобальных статических данных возможно указание секции, в которую компилятор должен поместить переменную, например:

```
static section("sec_pmda") int x;
```

переменная x будет размещена в секции `sec_pmda`.

2.3.4. Доступ к круговым буферам

Для доступа к массиву в режиме циклического буфера может использоваться конструкция вида `A[i%N]`, где `i` – текущий индекс, а `N` – длина буфера. При этом компилятор пытается использовать аппаратную поддержку круговых буферов для вычисления адреса элемента массива. Если же аппаратная поддержка круговых буферов отключена опциями компилятора, то вызывается специальная встроенная функция для вычисления адреса элемента кругового буфера, что, естественно, занимает гораздо больше процессорного времени.

2.3.5. Специальные описания переменных и указателей

При описании переменных и указателей могут использоваться модификаторы `volatile` и `restrict`, указывающие компилятору правила

обращения с этими переменными с точки зрения возможностей оптимизации кода.

Модификатор `volatile` используется при описании переменных, которые могут изменяться в произвольный момент времени, например, в процедуре обработки прерывания. Это приводит к тому, что при каждом обращении к этой переменной в тексте программы компилятор будет вынужден генерировать код для ее чтения из памяти, чтобы обеспечить использование реального текущего значения переменной, например:

```
int A[N];
int B[N];
int C[N];
volatile int sss=1;
...
for (i=0; i<N; i++)
   C[i] = sss*(A[i] + B[i]);
```

Без модификатора volatile	С модификатором volatile
r2=dm(_sss); i1=_B; i0=_A; i4=_C; lcntr=10, do(pc,.P1L2-1)until lce; r12=dm(i1,m6); r8=dm(i0,m6); r12=r12+r8; r12=r2*r12 (SSI); dm(i4,m6)=r12; .P1L2: ...	i1=_B; i2=_A; i0=_C; **i4=_sss;** lcntr=10, do(pc,.P1L2-1)until lce; **r2=dm(i4,m5);** r12=dm(i2,m6); r8=dm(i1,m6); r12=r8+r12; r2=r2*r12 (SSI); dm(i0,m6)=r2; .P1L2: ...

Как видно из примера, использование расширения `volatile` приводит к тому, что для каждой итерации компилятор гарантирует использование действительного значения переменной `sss`, которое может изменяться от итерации к итерации каким-либо внешним воздействием (в обработке прерывания).

Аналогичное действие модификатор `volatile` оказывает при использовании с описанием ассемблерной вставки по шаблону:

```
asm volatile ("idle;" : : :);
```

запрещая компилятору, например, перемещать вставку или объединять несколько вставок в одну.

Ключевое слово `restrict` представляет собой расширение для использования при описании указателей. Этот модификатор приводит к тому, что программист "сообщает" компилятору, что область памяти, на которую указывает `restrict`-указатель, может быть изменена только посредством данного указателя и никаким другим образом. Это позволяет компилятору выполнять определенные оптимизации кода с учетом того, что некоторый объект

не может быть изменен через другой указатель. (Действие модификатора `restrict` противоположно действию модификатора `const`. При использовании последнего объект не может быть изменен с использованием самого указателя, объявленного как `const`, но может быть изменен через другой указатель.)

Гарантируется, что в область памяти, куда указывает указатель, объявленный с модификатором `restrict`, он будет получать доступ в первую очередь (по сравнению с другими указателями).

В обычных ситуациях это не гарантируется и в некоторых ситуациях может привести к непредсказуемым последствиям:

```
void fir(short *n, short *c, short *restrict out, int n);
```

Если `out` не объявить как `restrict`, то компилятор предполагает, что он может совпадать с `n` или `c` и возможности оптимизации кода могут быть несколько ограниченны, т.к. приемник совпадает с одним из операндов. Объявление `restrict` говорит компилятору о том, что на область памяти, куда помещается результат вычислений, указывает только указатель `out`. Это позволит, например, перемешивать операции чтения данных из `n` и `c` и записи в `out` и выполнять их одновременно с вычислительными операциями.

Поведение программы будет непредсказуемым, если в программе присутствует присваивание между двумя `restrict`-указателями (т.е. принудительное создание алиасов), за исключением передачи такого указателя в качестве аргумента функции.

2.3.6. Массивы переменной длины

Компилятор cc21k поддерживает массивы переменной длины, которые всегда размещаются в стеке (в памяти данных). Обычно массивы переменной длины представляют собой локальные переменные процедур с длиной, которая передается в функцию в виде ее аргумента вызова. Область видимости такого массива и время его жизни – сама функция. Размер вычисляется при передаче управления в функцию для резервирования места в стеке. Для доступа к массивам (особенно многомерным) компилятор должен сгенерировать код для хранения и использования размерности по каждой переменной. Компилятор генерирует код для отслеживания выхода за границу массива и, если такая ситуация встречается, то происходит перераспределение памяти для хранения массива. Поэтому работа с массивами переменной длины может выполняться достаточно медленно.

2.3.7. Инициализация с индексацией

Стандарт ANSI/ISO C/C++ предполагает размещение элементов массива при его инициализации последовательно в фиксированном порядке. Это неудобно,

особенно когда необходимо установить лишь некоторые значения массива, а остальные оставить равными нулю.

стандарт	cc21k
`int a[6] = {0, 0, 15, 0, 29, 0};`	`int a[6] = { [4] 29, [2] 15 };`

2.3.8. Встроенные (built-in) функции

Компилятор поддерживает специальные встроенные функции двух видов:
- предназначенные для доступа и модификации битов системных регистров;
- работы с индексами круговых буферов.

Использование встроенных функций в принципе идентично использованию ассемблерных инструкций чтений/модификации системных регистров и указателей буферов данных, однако более предпочтительно в программе на C/C++, чем ассемблерные вставки.

Для работы со встроенными функциями необходимо подключить заголовочный файл `sysreg.h`.

Функции доступа к системным регистрам аналогичны основным инструкциям ассемблера, предназначенным для этих же целей, например:

builtin - функция	Инструкция на ассемблере	Назначение
`int sysreg_read(const int SR_number)`	`= sysreg`	Чтение системного регистра sysreg
`sysreg_write(const int SR_number, const int new_value)`	`sysreg =`	Запись системного регистра sysreg
`sysreg_bit_clr(const int SR_number, const int bit_mask)`	`bit clr sysreg bitmask`	Очистка всех битов системного регистра, которые установлены в битовой маске
`sysreg_bit_set(const int SR_number, const int bit_mask)`	`bit set sysreg bitmask`	Установка всех битов системного регистра, которые установлены в битовой маске
`sysreg_bit_tgl(const int SR_number, const int bit_mask)`	`bit tgl sysreg bitmask`	Инверсия всех битов системного регистра, которые установлены в битовой маске
`int sysreg_bit_tst(const int SR_number, const int bit_mask)`	`bit tst sysreg bitmask`	Возвращает ненулевое значение, если все биты системного регистра, которые установлены в битовой маске, равны единице в системном регистре

Для всех функций есть варианты с _nop на конце функции. Использование такого варианта приводит к генерации команды nop после ассемблерной

инструкции (запись практически во все системные регистры требует для 1-2 тактов для вступления в силу).

В качестве наименований системных регистров используются:

в ADSP-21060	в ADSP-21160
sysreg_IMASK	sysreg_LIRPTL
sysreg_IMASKP	sysreg_MMASK
sysreg_ASTAT	sysreg_ASTATY
sysreg_STKY	sysreg_FLAGS
sysreg_USTAT1	sysreg_STKYY
sysreg_USTAT2	sysreg_USTAT3
	sysreg_USTAT4

В файлах def21060.h и def21160.h содержатся константы для указания номеров битов в системных регистров (полезно для установки маски прерываний).

Функций работы с адресами в циклических буферах всего две: одна предполагает модификацию индекса, другая – указателя.

builtin-функция	Код на ассемблере
`int circindex(int index, int incr, int num_items)`	`index += incr;` `if (index<0)` `  index += num_items;` `else if (index>=num_items)` `  index -= num_items;`
`int circptr(void *ptr, size_t incr, void *base, size_t buflen)`	`ptr += incr;` `if (ptr<base)` `  ptr += buflen;` `else if (ptr>= (base+buflen))` `  ptr -= buflen;`

При использовании встроенных функций в теле программы ее имя (вызов) должно начинаться с `__builtin_`.

2.3.9. Директивы компилятора (#pragmas)

Компиляторы языка C обычно поддерживают ряд директив, позволяющих локально устанавливать параметры генерации кода и размещения данных в памяти. Стандартно данные возможности реализуются с использованием директив #pragma. Формат записи директивы выглядит следующим образом:

```
#pragma  имя_директивы  [параметры_директивы]
```

Компилятор cc21k поддерживает следующие типы директив:
- директивы выравнивания данных;
- директивы описания обработчиков прерываний;
- директивы глобальной оптимизации;
- директивы оптимизации циклов;
- директивы описания поведения функций;
- директивы контроля экземпляров шаблонов в C++;
- директивы контроля заголовочных файлов;

- директивы задания видимости имен при компоновке.

В прил. 1 кратко рассматриваются некоторые директивы, имеющие наиболее широкое практическое значение. Более подробную информацию о директивах можно найти в руководстве по компилятору.

2.3.10. Поддержка дробной арифметики

Компилятор поддерживает использование арифметики дробных чисел с фиксированной запятой в программах на C++. Для работы с подобными данными на аппаратном уровне следует использовать переменные типа `fract` (необходимо, чтобы был подключен заголовочный файл `<fract>`). Для объявления дробных констант с фиксированной запятой используется суффикс "r" после записи числа, например:

```
#include <fract>
...
fract a=0.5, b=-0.5;
```

При преобразовании чисел между форматами `float/double` и `fract` возможны потери точности, связанные с меньшим числом разрядов, отводимых на представление мантиссы в форматах с плавающей запятой. Преобразование чисел между целочисленными и дробными форматами нецелесообразно, так как результатом такого преобразования могут быть только значения "0" и "-1".

При работе с дробными ФЗ-данными может использоваться режим арифметики с насыщением или с переполнением, который переключается путем вставки в программный код функций `set_saturate_mode()` и `reset_saturate_mode()`. Поддерживаются операции преобразования типов между дробным и целыми типами и между дробным и ПЗ-типами, а также стандартный набор арифметических операций: сложение, вычитание, умножение, сдвиги, сравнения. Следует соблюдать осторожность и при смешивании в процессе вычислений значений ПЗ-формата и дробного ФЗ-формата.

2.3.11. Поддержка SIMD-архитектуры (ADSP-2116x +)

Процессоры ADSP-2116x и старше поддерживают выполнение инструкций в SIMD-режиме, позволяющем выполнять за один такт операцию параллельно над двумя наборами данных, находящихся в разных регистровых файлах процессорного ядра. Понимание логики процессов, выполняемых при работе в SIMD-режиме, связано со знанием архитектуры процессора, режимов доступа к памяти и правил размещения в ней исходных данных.

В стандартном SISD-режиме процессор выбирает значение из памяти, выполняет арифметическую операцию и записывает результат обратно в память. В SIMD-режиме при каждой выборке из памяти за один такт читаются сразу две

соседние ячейки: одна попадает в первый процессорный элемент (PEx), другая – во второй процессорный элемент (PEy). Одна и та же инструкция выполняется одновременно над одноименными регистрами двух процессорных элементов. Результат в виде пары значений также записывается за один такт из PEx и PEy в две смежные ячейки памяти.

Обычно SIMD-режим используется при обработке с использованием одного и того же алгоритма двух независимых (или частично независимых) векторов данных, например, для одновременного суммирования двух пар векторов (A+B и C+D), обработки двух независимых каналов данных, вычисления одной и той же функции для двух разных значений. Ключевым моментом при организации вычислений в SIMD-режиме является перемешивание данных двух независимых каналов при их хранении в памяти: данные одного канала должны быть расположены по четным адресам, а данные другого канала – по нечетным. Размер общего массива при этом увеличивается в два раза, а модификация адреса при доступе к памяти выполняется на 2 адреса за одно обращение. Если используются данные, одинаковые для обоих каналов (например, одни и те же коэффициенты фильтра), то они могут быть загружены либо с использованием широковещательной загрузки (broadcast load) на языке ассемблера, либо просто продублированы в памяти. Организация обработки многоканальных данных в SIMD-режиме в программе на языке C/C++ осуществляется исключительно программистом.

Было бы неверно считать, что SIMD-архитектура процессора может использоваться только при многоканальной обработке данных. Рассмотрим, например, тело цикла для поэлементного сложения двух векторов при обработке данных одного канала:

```
c[i] = a[i] + b[i]
i = i+1;
```

Поскольку операция над каждым *i*-м элементом массивов выполняется независимо от предыдущих итераций и индексы изменяются синхронно, перепишем данную команду следующим образом:

```
c[i] = a[i] + b[i]
c[i+1] = a[i+1] + b[i+1]
i = i + 2
```

Таким образом, мы можем получить двукратное повышение производительности при обработке даже одной пары векторов в SIMD-режиме. Именно подобные оптимизации поддерживает компилятор. Однако такая векторизация (распараллеливание) вычислений сопряжена с необходимостью выполнения одного или нескольких следующих действий, снижающих эффект от оптимизации:

- изменение режима работы процессора на уровне аппаратуры для перехода в SIMD-режим и возврата из него;

- принудительное дублирование скалярных значений для использования одних и тех же величин в параллельных ветвях алгоритма;

- объединение промежуточных результатов при необходимости получения итогового скалярного результата (после выхода из SIMD-режима), например:

Без векторизации кода	**С векторизацией кода**
Только SISD-режим (N итераций)	**SIMD-режим (N/2 итераций)**
`S = S + a[i]*b[i]`	`S = S+a[i]*b[i]  ||  S' = S'+a[i+1]*b[i+1]`
`i = i+1`	`i = i+2`
	SISD-режим
	`S = S + S'`

При организации вычислений (как многоканальных, так и одноканальных) в SIMD-режиме программисту необходимо учитывать следующие требования и ограничения:

- для использования возможности векторизации цикла следует явно указать это путем вставки соответствующей директивы перед заголовком цикла. Желательно векторизовать только циклы наибольшего уровня вложенности;

- данные, которые должны обрабатываться параллельно, следует размещать в памяти последовательно с чередованием. Так, задача обработки элементов столбца массива, хранящегося в памяти как развертка строк, не подходит для ее решения в SIMD-режиме из-за невозможности считывания за один такт следующего элемента столбца;

- массивы, которые предполагается обрабатывать в SIMD-режиме, должны быть выровнены по четной границе (например, с использованием директивы вида `#pragma align 2`). Это достаточно трудно при динамическом выделении памяти под массив. Следует особо осторожно требовать SIMD-реализации внутри функций, которые в качестве параметра получают указатель на произвольный элемент массива. Если указатель будет указывать на нечетный элемент массива, то результат обработки может оказаться ошибочным;

- не следует использовать SIMD-обработку для данных, размещаемых во внешней памяти, так как для всех процессоров ADSP-2116x+ внешняя шина является 32-разрядной (для чтения двух операндов за 1 обращение требуется 64-разрядная внутренняя шина данных);

- не рекомендуется использовать SIMD-оптимизацию для циклов, которые в текущей итерации используют результат предыдущей итерации. Так, выполнение команды `a[i+1]=a[i]+2` в SIMD-режиме приведет к неправильному результату, так как данные для (`i+1`)-й итерации читаются из памяти раньше, чем в нее записываются результаты `i`-й итерации;

- не рекомендуется использовать SIMD-оптимизацию для циклов, внутри которых доступ к элементам массива осуществляется через модифицированный индекс, например:

```
for (i=0; i<N; i++)
      tmp += ArrayX[j + (i*t)];
```

- следует проявлять осторожность при SIMD-оптимизации циклов с обработкой многомерных массивов, особенно с нечетными размерностями;

- внутри циклов с SIMD-оптимизацией не должно быть вызовов функций;

- внутри циклов с SIMD-оптимизацией может осуществляться доступ только к данным базовых типов с размером в 1 слово (нельзя обращаться к переменным типа long double, struct и т.п.).

При организации вычислений с векторизацией циклов особое значение приобретает число итераций цикла. Если оно известно заранее, то компилятор может эффективно оптимизировать код даже при нечетном числе итераций, например, выполняя последнюю итерацию в SISD-режиме, например:

Исходный код на С	**Сгенерированный код на ассемблере**
`void sum(int *a, int *b, int *c)`	`...`
`{`	**`// → SIMD`**
`int i;`	`bit set mode1 0x200000; nop;`
	`m4=2;`
`#pragma SIMD_for`	`lcntr=49, do(pc,.P1L4-1)until lce;`
`for (i=0; i<99; i++)`	`r2=dm(i4,m4);`
`c[i] = a[i] + b[i];`	`r12=dm(i1,m4);`
`}`	`r2=r2+r12;`
	`dm(i0,2)=r2;`
	`.P1L4:`
	`// → SISD`
	`bit clr mode1 0x200000; nop;`
	`// последняя итерация`
	`r2=dm(m5,i4);`
	`r12=dm(m5,i1);`
	`r2=r2+r12;`
	`dm(m5,i0)=r2;`
	`...`

Если число итераций на этапе компиляции неизвестно, то сгенерированный код существенно усложняется:

Исходный код на С		**Сгенерированный код на ассемблере**
`void sum(int *a, int *b, int *c, int num)`		`...`
`{`		`r2=dm(1,i6);`
		`// run-time проверка числа итераций`
`int i;`	*	`r1=pass r2, i4=r12;`
`int s=2;`	*	`if le jump(pc,.P1L2) (DB);`
		`// если ни одной итерации –`
`#pragma SIMD_for`	*	**`// ничего не делать`**
`#pragma loop_count(2,100,2)`	*	`i0=r4;`
`for (i=0; i<num; i++)`		`i1=r8;`
`c[i] = a[i] + b[i];`	*	**`// если одна итерация – обойти`**
`}`	*	**`//  SIMD-цикл`**
	*	`r2=r1-1;`
		`r2=pass r2;`
		`if le jump(pc,.P1L4);`
		`// уменьшить число итераций в 2раза`
		`r2=ashift r1 by -1;`
		`...`
		`// → SIMD`
		`bit set mode1 0x200000; nop;`
		`m4=2;`
		`lcntr=r2, do(pc,.P1L12-1)until lce;`
		`r2=dm(i1,m4);`

```
        r12=dm(i0,m4);
        r2=r2+r12;
        dm(i4,2)=r2;
.P1L12:
// → SISD
*   bit clr mode1 0x200000; nop;
*   // число ли число итераций было
*   // "заказано"
*     r2=1;
      r12=r1 and r2;
      comp(r12, r2);
      if ne jump(pc,.P1L2);
*   .P1L4:
*   // если нечетное число итераций -
*   сделать последнюю в SISD-режиме
*     r2=dm(m5,i1);
      r12=dm(m5,i0);
      r2=r2+r12;
      dm(m5,i4)=r2;
.P1L2:
      ...
```

В данном примере использование уточняющей директивы `loop_count(2,100,2)` перед объявлением цикла позволило бы компилятору не включать в генерируемый код инструкции, отмеченные "*", так как он получает информацию о том, что тело цикла будет выполняться не менее 2-х раз и количество итераций всегда будет четным числом. Как видно из листинга, такая подсказка (если она возможна) позволяет существенно повысить производительность программы.

Тем не менее разработчики компилятора cc21k указывают, что возможны ситуации, когда компилятор не в состоянии корректно оценить опасность векторизации цикла, что в конечном итоге может повлечь за собой неверный результат вычислений. Это в первую очередь связано с такой аппаратной особенностью организации доступа к внутренней памяти, как наличие теневого FIFO-буфера записи в ADSP-21160. Если в момент наличия в буфере незаписанных данных выполняется чтение 2-х соседних ячеек, пересекающих границу длинного слова (в SIMD-режиме), то результат "неявной" пересылки будет непредсказуем. Именно этой аномалией объясняется требование выравнивания по четной границе данных, к которым осуществляется доступ в SIMD-режиме в программах на C/C++ (в программе на ассемблере программист может самостоятельно избежать подобной ситуации, но компилятор cc21k не всегда может контролировать возможность возникновения подобной последовательности операций доступа к памяти).

Такая опасность может возникать во вложенных циклах, если число итераций вложенного цикла зависит от номера итерации внешнего цикла, и при этом выполняется доступ к данным, например:

```
for (k=0; k<N; ++k)
 {
```



```
 #pragma SIMD_for
  for (i=0; i<N-k; ++i)
   output[i] += (input[i] + input[i+k]);
 }
```

В данном случае при каждой нечетной итерации по `k` (`k`=1, `k`=3,...) адрес `input[i+k]` не будет выровнен по границе двойного слова, что в определенной ситуации может привести к неверно прочитанному значению при неявной пересылке данных. Поэтому разработчики рекомендуют подобные циклы оформлять таким образом, чтобы гарантированно избежать доступа к памяти в SIMD-режиме по нечетному адресу (предполагается, что базовые адреса всех участвующих в обработке массивов выровнены по границе длинного слова):

```
for (k=0; k<N; ++k)
 {
  if (k % 2)
   for (i=0; i<N-k; ++i)
         output[i] += (input[i] + input[i+k]);
else
#pragma SIMD_for
     for (i=0; i<N-k; ++i)
         output[i] += (input[i] + input[i+k]);
 }
```

2.4. Модель времени выполнения и окружение C/C++

Среда выполнения (run-time environment, RTE) компилятора cc21k представляет собой набор соглашений, касающихся правил выполнения программы на процессоре SHARC ADSP. Если в проекте присутствуют как C/C++-файлы, так и файлы на языке ассемблера, то ассемблерные подпрограммы должны поддерживать эти соглашения.

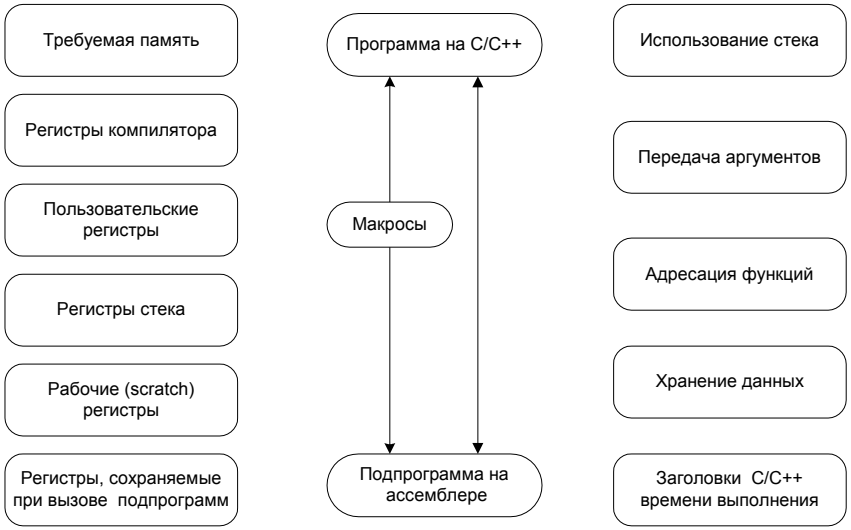

Рис. 2. Интерфейс подпрограммы на ассемблере

2.4.1. Использование памяти

Среда выполнения C/C++ требует использования определенных секций для размещения кода и данных в памяти процессора. Эти секции задаются в LDF-файле в виде выходных (Output) секций. Естественно, что проецирование секций на физическую память процессора должно выполняться с учетом модели процессора и структуры его памяти. Одним из наиболее простых вариантов для небольших проектов является отказ от разработки собственного LDF-файла и использование LDF-файла по умолчанию. В этом случае LDF-файл вообще не следует добавлять в проект: компоновщик будет использовать файл описания архитектуры, соответствующий выбранному в опциях типу процессора и языку программирования. Если же необходимы какие-либо модификации, например, в размерах сегментов, то следует скопировать стандартный ldf-файл в свою директорию, подключить его к проекту и затем вносить в него изменения.

Стандартными для компилятора C/C++ выходными секциями являются:

Имя секции	Для чего используется
seg_pmco	Размещается в памяти программ, содержит код программы, а также требуется для нормальной работы некоторых библиотечных функций C/C++
seg_dmda	Размещается в памяти данных. По умолчанию содержит все глобальные и статические переменные, а также символьные строки, например в этой секции будет размещен массив data: static int datas[10]; Требуется для работы некоторых библиотечных функций C/C++
seg_pmda	Размещается в памяти программ и содержит переменные, которые должны быть явно помещены в PM-память, например: static int pm coeffs[10]; Требуется для работы некоторых библиотечных функций C/C++
seg_stak	Размещается в памяти данных и используется для хранения локальных переменных и адресов возврата из подпрограмм. Обязательна для функционирования программы на C/C++. Обычно размер стека составляет 4K 32-битовых слов
seg_heap	Размещается в памяти данных и представляет собой "кучу", из которой динамически выделяется память при необходимости во время работы программы (при выполнении операций malloc, calloc и т.п.). Стандартный размер кучи – 60K 32-битовых слов. Компилятор поддерживает возможность использования нескольких куч, причем размещенных в различных пространствах памяти
seg_init	Размещается в памяти программ и содержит данные, необходимые для инициализации программы, т.е. ее загрузки во внутреннюю память процессора для выполнения (или настройки параметров внешней памяти при выполнении команды из внешнего ОЗУ)
seg_rth	Размещается в памяти программ и содержит код для выполнения инициализационных действий и таблицу векторов прерываний. Инициализационный (заголовочный) код, run-time header, устанавливает начальные значения переменных среды выполнения C/C++, инициализирует таблицу векторов прерывания, вызывает функцию main(). Для этого по умолчанию используется объектный файл, выходящий в состав среды VisualDSP++ (например, 060_hdr.doj). Механизм обработки прерываний в программах на C/C++ основан на использовании такого файла инициализации среды выполнения

2.4.2. Регистры компилятора

Компилятор cc21k предполагает использование регистров процессора для различных целей. В соответствии с этими целями и правилами модификации этих регистров подпрограммами пользователя (на ассемблере) регистры можно разделить на следующие категории:

- регистры компилятора (Compiler Registers), которые используются компилятором исключительно для собственных целей и должны содержать указанные в таблице значения. Не следует изменять значения этих регистров, так как компилятор предполагает, что эти регистры всегда имеют значения,

приведенные в таблице. Дополнительно следует отметить, что все L-регистры (кроме L6 и L7) должны быть равны 0 в любой точке вызова процедуры и возврата из процедуры. Если какой-либо из L-регистров был модифицирован, то при вызове функции или возврате управления в вызывающую подпрограмму, его следует обнулить. Подпрограммы – обработчики прерываний должны сохранять (в памяти) и сбрасывать L-регистры в 0 до использования соответствующих им I-регистров для любых инструкций с пост-модификацией;

Регистры компилятора (Compiler Registers)	Значение	Правило использования
M5, M13	0	Не модифицировать
M6, M14	1	Не модифицировать
M7, M15	-1	Не модифицировать
B6, B7	база стека	Не модифицировать
L6, L7	длина стека	Не модифицировать
L0, L1, L2, L3, L4, L5, L8, L9, L10, L11, L12, L13, L14, L15	0	Доступны для временного использования. После использования – восстановить

- регистры, зарегистрированные пользователем (User Registers), задание (перечисление) которых в опциях компилятора вынуждает компилятор отказаться от использования этих регистров. Если же компилятору не будет хватать регистров, он проигнорирует запрет на их использование. Следует заметить, что "заказ" L-регистра требует резервирования соответствующего I-регистра и наоборот, в противном случае возможны ошибки на этапе выполнения. Чем больше зарезервировано регистров для пользовательских целей, тем ниже может оказаться "качество" кода, сгенерированного компилятором;

Пользовательские регистры (User Registers)	Значение	Правило использования
i0, n0, l0, m0, i1, b1, l1, m1, i8, b8, l8, m8, i9, b9, l9, m9, mrb, ustat1, ustat2, ustat3, ustat4	Задается пользователем	Если не зарезервированы явно, то можно использовать для временного пользования, а по окончании восстановить прежние значения. Если явно зарезервированы под пользовательские цели в опциях компилятора, то можно использовать без ограничения

- регистры, сохраняемые при вызове (Call Preserved Registers), представляют собой набор регистров, которые должны быть сохранены в "прологе" функции и затем восстановлены в "эпилоге" функции, если эта функция написана на ассемблере. Естественно, что если функция не изменяет регистры данной группы, то сохранять и восстанавливать их необязательно. При этом, если используется сохраняемый при вызове I-регистр, то необходимо сохранить (и затем восстановить) не только этот I-регистр, но и соответствующий ему L-регистр. Значительная часть библиотечных функций выполняется в

предположении, что процессор работает в определенном режиме, определяемом, в частности, регистрами параметров и управления. Если эти значения будут изменены и затем выполнен вызов стандартной функции, то результат ее работы будет непредсказуем. Поэтому при необходимости изменения режимов работы процессора (регистры MODE1 и MODE2) "хорошим тоном" считается сохранение их старых значений и их восстановление при вызове другой функции и при возврате в вызывающую функцию.

"Сохраняемые при вызове" регистры (Call Preserved Registers)
B0, B1, B2, B3, B5, B8, B9, B10, B11, B14, B15
I0, I1, I2, I3, I5, I8, I9, I10, I11, I14, I15
MODE1, MODE2
MRB, MRF
M0, M1, M2, M3, M8, M9, M10, M11
R3, R5, R6, R7, R9, R10, R11, R13, R14, R15
USTAT1, USTAT2

По умолчанию run-time среда C/C++ предполагает, что режим процессора задается следующими установками (задача обеспечить эти установки ложится на инициализационный код):

а) не используется бит-реверсная адресация;

б) используется основной (не теневой) набор регистров;

в) используется точность .PRESICION=32 (32-битовые ПЗ-числа) и режим округления до ближайшего целого;

г) запрещено насыщение АЛУ (бит ALUSAT=0);

д) для ADSP-2116x разрешены циклические буферы (бит CBUFEN=1 в регистре MODE1);

- временные или рабочие регистры (Scratch Registers) не требуют сохранения и восстановления, изменение их содержимого при вызове функций или вставке ассемблерного кода не "волнует" компилятор. Для процессора ADSP-2116x все регистры данных процессорного элемента PEy являются Scratch-регистрами;

Рабочие регистры (Scratch Registers)
B4, B12, B13
R0, R1, R2, R4, R8, R12
I4, I12, I13
M4, M12
PX

- отдельный набор регистров (Stack Registers) резервируется и всегда используется для работы с программным стеком. Подпрограммы на ассемблере должны строго придерживаться правил работы с ними;

Регистры стека (Stack Registers)	Значение	Правило использования
I7	Указатель стека (Stack Pointer)	Модифицируется при работе со стеком, восстановить при возврате из подпрограммы
I6	Указатель фрейма стека (Frame Pointer)	Модифицируется при работе со стеком, восстановить при возврате из подпрограммы
I12	Адрес возврата	Загружается адресом возврата при выходе из функции

- альтернативные регистры (Alternate Registers) не используются средой выполнения и доступны для использования в ассемблерных подпрограммах. Однако при их использовании следует учитывать, что при переключении регистров I6 и I7 (DAG1) на теневой набор может быть нарушена работа со стеком. Поэтому при использовании теневых регистров, особенно I6 и I7, прерывания должны быть запрещены. Кроме того, самый быстрый диспетчер прерываний (super fast interrupt dispatcher) для переключения контекста сам использует теневые регистры вместо сохранения регистров в стеке среды выполнения. Поэтому во избежание конфликтов желательно не применять данный вид диспетчера прерываний при использовании теневых регистров в ассемблерных вставках.

2.4.3. Работа со стеком

Среда выполнения C/C++ использует программный стек для хранения автоматических (локальных) переменных и адресов возврата. Необходимо помнить, что этот стек размещается в памяти данных и не имеет ничего общего со стеком программного секвенсора (PC Stack), используемого при программировании процессоров SHARC ADSP на низком уровне.

Как и любой run-time стек, он растет в сторону младших адресов памяти. Для понимания работы стека вспомним понятие кадра стека: это фрагмент стека, используемый для хранения информации о текущем контексте программы (локальных переменных, временных переменных компилятора, фактических параметрах вызова следующей функции).

Для работы со стеком используются два указателя:

- указатель стека, содержащий адрес вершины стека – верхней (т.е. расположенной по наибольшему адресу) *незаполненной* ячейки стека (в отличие от Intel-процессоров указатель run-time стека ADSP указывает на первую свободную ячейку);

- указатель кадра стека – базовый адрес, относительно которого выполняется доступ к кадру стека (обычно в режиме предмодификации).

Кадр стека для функции можно определить следующим образом:

Адреса	Хранимое значение	Когда и кем помещается в стек и выталкивается
Младшие адреса памяти	Сохраненный указатель на кадр "текущей" функции	Автоматически при выполнении стандартного кода для вызова/возврата "вложенной" функции из "текущей" (только если есть вызов "вложенной" функции)
. . . . .	Параметры вызова "вложенной" функции	Вручную программистом (компилятором), только если есть вызов "вложенной" функции
	Локальные переменные "текущей" функции	Вручную программистом (компилятором), если есть локальные параметры. Можно поменять местами с сохраняемыми регистрами
	Сохраненные регистры Call Preserved	Вручную программистом (компилятором), если Scratch-регистров недостаточно. Можно поменять местами с локальными параметрами
Старшие адреса памяти	Адрес возврата в вызывающую подпрограмму	Автоматически при выполнении стандартного кода вызова/возврата "текущей" функции из "вызывающей"

Действие приведенных фрагментов кода при вызове и возврате из подпрограмм проиллюстрировано на рис.3.

На рис.4 приведен пример структуры стека при вызове функции Test.

При генерации кода для вызова функции компилятор cc21k формирует код, выполняющий следующую последовательность действий (аналогичная последовательность должна выполняться в подпрограмме на ассемблере, если она выполняется в рамках среды RTE и вызывает C/C++ – функцию):

- загрузка регистра r2 значением указателя кадра: r2 = i6;

- установка указателя кадра i6 на вершину стека: i6 = i7;

- использование задержанного перехода (db) для передачи управления вызываемой функции;

- во время выполнения первой инструкции, находящейся в конвейере до перехода, проталкивание в стек прежнего указателя кадра, находящегося в регистре r2;

- во время выполнения второй инструкции, находящейся в конвейере до перехода, проталкивание в стек адреса возврата (регистр PC).

Система команд процессора содержит специальную инструкцию cjump xxx (DB), совмещающую выполнение первых трех действий, что позволяет не запрещать прерывания при изменениях указателей стеков. Таким образом, код вызова функции, будет выглядеть следующим образом:

```
cjump my_function (DB);
  dm(i7, m7) = r2;
  dm(i7, m7) = pc;
```

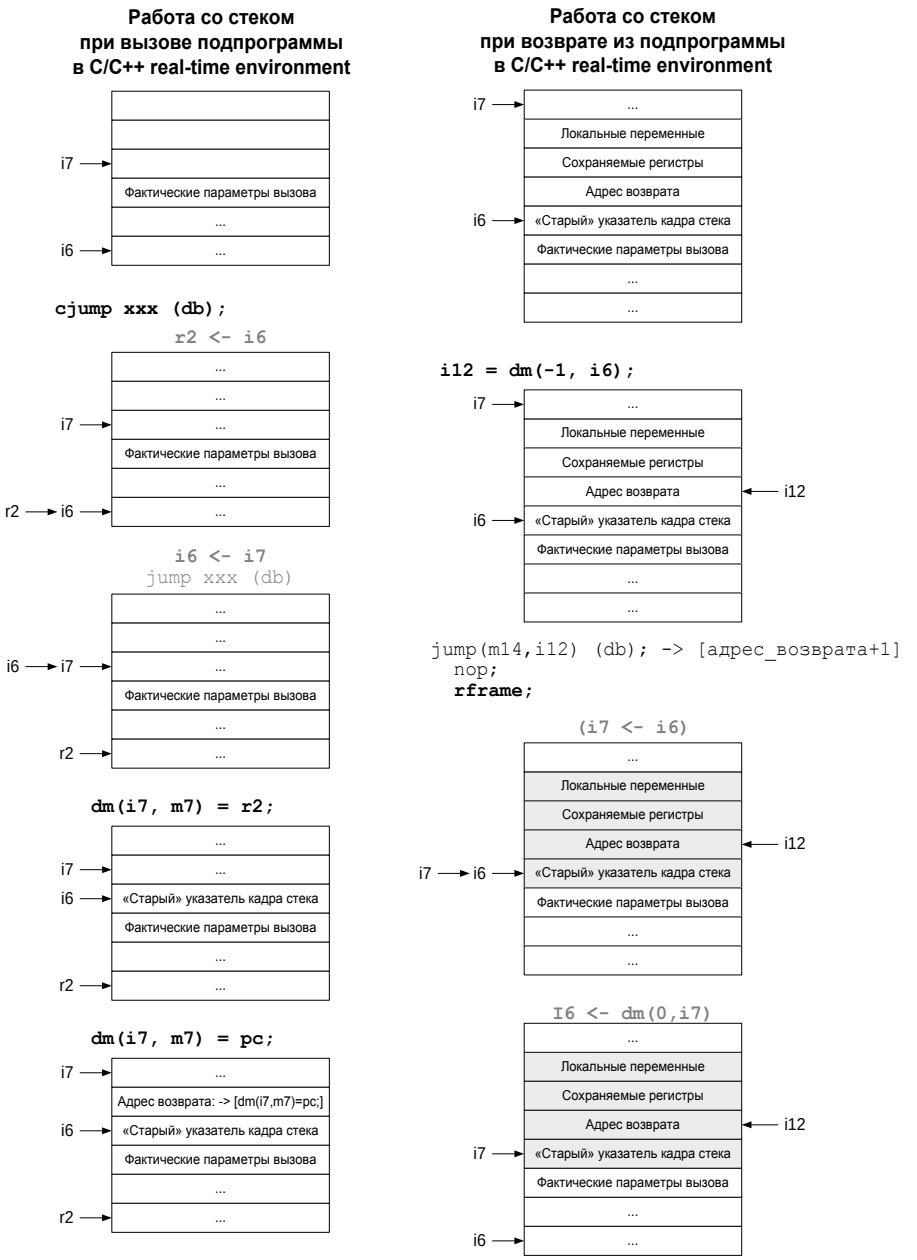

Рис.3. Работа со стеком при использовании команд cjump и rframe

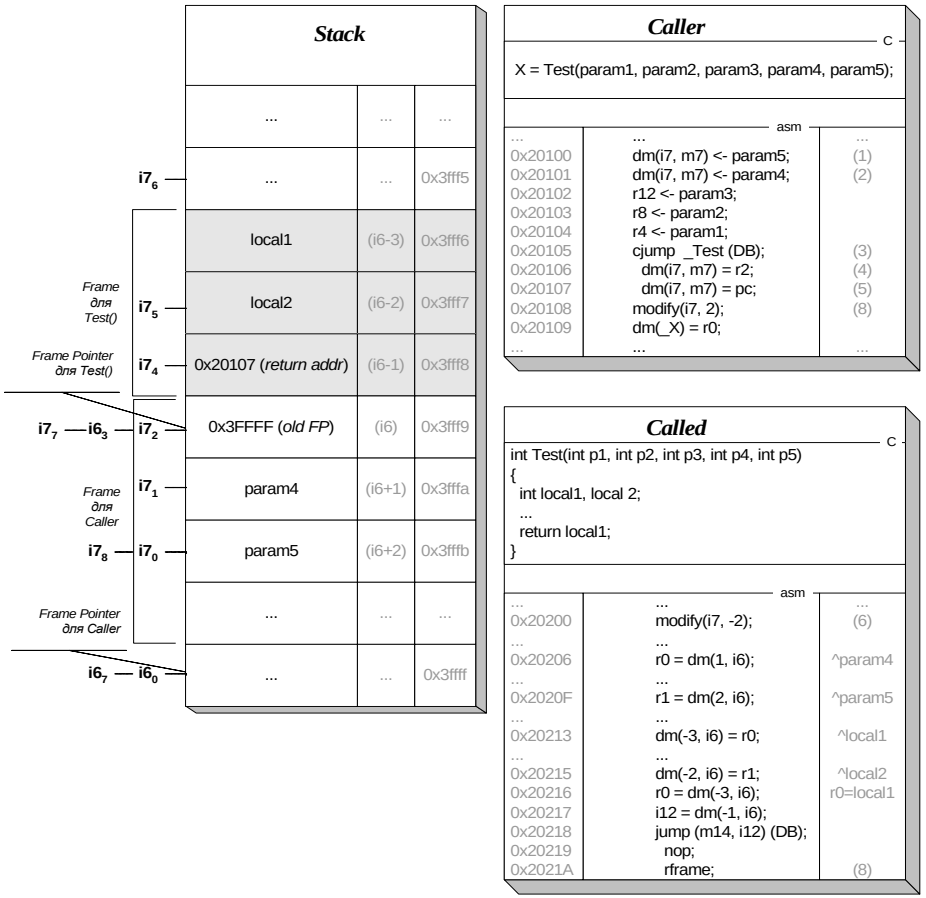

Рис.4. Структура стека при вызове функции Test

При генерации кода для выхода из функции компилятор формирует следующую последовательность действий (аналогичные правила для подпрограммы на ассемблере, если она вызвана из C/C++ – функции):

- выталкивание адреса возврата из стека и загрузка его в регистр `i12`;
- использование задержанного перехода для возврата управления в вызывающую подпрограмму и переход по адресу (`i12+1`);
- восстановление указателя стека вызывающей функции в регистре `i7` путем установки равным указателю стека кадра (регистр `i6`) во время выполнения первой отложенной инструкции при переходе;
- восстановление указателя кадра стека вызывающей функции в регистре `i6` путем выталкивания из стека ранее сохраненного указателя кадра стека и

загрузки его в регистр i6 во время выполнения второй отложенной инструкции при переходе.

В системе команд процессора есть специальная инструкция rframe, автоматизирующая выполнение последних двух шагов:

```
i12 = dm(-1, i6);
jump (m14, i12) (DB);
  nop;
  rframe;
```

2.4.4. Передача параметров в функции и возврат значений

В связи с большим размером регистрового файла процессоров SHARC ADSP компилятор по возможности старается передавать параметры в функции через регистры. Через регистры в общем случае передаются три первых параметра, остальные передаются через стек. В стек параметры помещаются в порядке справа-налево (т.е. самым нижним должен оказаться аргумент, соответствующий самому правому параметру в описании функции).

Первый параметр, указанный в списке фактических параметров при вызове функции, передается через регистр r4, второй – через регистр r8, третий – через регистр r12. Остальные параметры передаются через стек. Из этого правила есть 2 исключения:

- если в подпрограмму передается параметр, занимающий больше 32-х битов, то он и все остальные параметры передаются через стек;

- если C++ – функция объявлена как принимающая переменное количество параметров (в прототипе функции имеется многоточие), то через стек передаются последний именованный параметр и все последующие.

Возврат результата в вызывающую функцию всегда выполняется через регистр. Если возвращаемое значение является 32-битовым (слово, адрес, структура длиной в 1 слово), то оно возвращается в регистре r0; если 64-битовым (например, тип long double или структура из двух слов), то в регистрах r0 (старшее слово) и r1 (младшее слово). Если результат имеет длину более 2-х слов, то в регистре r1 возвращается адрес области памяти, где хранится результат.

Прототип функции	**Передача параметров**	**Доступ к параметрам в функции**
pass(int a, float b, char c, float d);	a – r4 b – r8 c – r12 d – через стек	- - - dm(1, i6)
count(int w, long double x, char y, float z)	w – r4 x – через стек y – через стек z – через стек	- dm(1, i6)-MSW(x), dm(2, i6)-LSW(x) dm(3, i6) dm(4, i6)
compute(float k, int l, char m,…)	k – r4 l – r8 m и остальные – через стек	- - dm(1, i6) ...

В приведенном ниже примере показан фрагмент кода, сгенерированного компилятором для вызова/возврата функции с двумя параметрами (и доступа к ним в теле функции).

Исходный код на языке C	**Программный код на ассемблере для SHARC ADSP-2106x, сгенерированный компилятором**
int my_func(int, int); int arg_a, return_c; int my_func(int arg_1, int arg_2) { return (arg_1 + arg_2)/2; } main() { static int arg_b; arg_b = 0; return_c = my_func(arg_a, arg_b); }	**/* Фрагмент функции main */** _main: ... r4=dm(_arg_a); /* r4 ← перв. аргумент */ r8=0; /* r8 ← втор. аргумент */ dm(arg_b)=r8; **/* Вызов функции my_func */** cjump (pc,_my_func) (DB); dm(i7,m7)=r2; dm(i7,m7)=pc; **/* Сохранение результата после возврата*/** dm(_return_c)=r0; ... **/* C-функция my_func */** _my_func: r0=(r4+r8)/2; **/* Возврат в main */** i12=dm(-1,i6); jump (m14, i12) (DB); nop; rframe;

2.5. Взаимодействие подпрограмм на C и ассемблере

2.5.1. Описания имен и области их видимости

Компилятор языка C/C++ предваряет любое внешнее имя знаком подчеркивания. Поэтому чтобы имя метки или переменной из ассемблерного модуля было видно в функциях C/C++, следует:

- либо метку в ассемблерном модуле начинать с подчеркивания;

- либо в программе на C при описании внешней метки использовать конструкцию вида:

```
extern "asm" {}
```

Не следует забывать, что в модуле, где метка описана, ее имя следует объявлять как global, а в модуле, в котором к ней идет обращение – как extern, например:

Описание функции в asm-модуле	**Объявление функции в C-модуле**
.global _asmfunc; …	void asmfunc(void); …

`_asmfunc:` `…`	`asmfunc();` `…`
`.global asmfunc;` `…` `asmfunc:` `…`	`extern "asm" void asmfunc(void);` `…` `asmfunc();` `…`

2.5.2. Макросы для работы со стеком при вызове и возврате из функций на ассемблере

Для облегчения работы со стеком при вызове и возврате из подпрограмм, сохранении/восстановлении регистров и очистки стека от передаваемых в подпрограмму параметров может быть использован ряд макросов, описанных в файле asm_sprt.h. Основные из них приведены в таблице.

Макрос	**Программный код для ADSP-21060**	**Описание применения макроса**
`leaf_exit`	`i12=dm(m7,i6);` `jump (m14,i12) (db);` `nop;` `RFRAME`	Эпилог функции – последняя строка ассемблерной программы
`ccall(x)`	`cjump (x) (DB);` `dm(i7,m7)=r2;` `dm(i7,m7)=PC`	Вызов функции с именем x
`reads(x)`	`dm(x, i6)`	Читает значение из стека, которое находится со смещением x относительно указателя кадра стека (I6), например: `R0=reads(1) -> R0=dm(1,I6);`
`puts`	`dm(i7, m7)`	Помещает в стек значение регистра, например: `puts=R0 -> dm(I7,M7)=R0;`
`gets(x)`	`dm(x, i7)`	Читает значение из стека со смещением x относительно его вершины (I7): `R0=gets(3) -> R0=dm(3,I7);`
`alter(x)`	`modify(i7, x)`	Изменяет значение указателя на вершину стека (I7) на число x, позволяя таким образом (при x>0) очистить стек от x верхних слов. Это может быть полезно при очистке стека от параметров процедуры в *вызывающей* функции
`save_reg`	`puts=r0;\` `puts=r1;\` `...` `puts=r15`	Последовательно сохраняет в стеке все регистры регистрового файла
`restore_reg`	`r15=gets(1);\` `r14=gets(2);\` `...` `r0=gets(16);\` `alter(16)`	Последовательно восстанавливает из стека все регистры регистрового файла и очищает стек

Пример подпрограммы на ассемблере с использованием макросов.

Вызов: `my_func(int a, int b, int c, int d, int e);`

```
#include <asm_sprt.h>
...
my_func:  puts=R3;        // сохранить R3
          R3 = reads(1); // R3 = d (доступ к парам. отн. I6)
          ...
          R3 = gets(1);  // восстановить R3
          alter(1);      // и выталкнуть его из стека - необязательно
          leaf_exit;
my_func.end: nop;
```

2.5.3. Вызов подпрограммы на ассемблере из программы на С

Хорошим стилем программирования считается описание прототипа ассемблерной функции для определения аргументов функции и интерфейса между программой на С и подпрограммой на ассемблере.

Модель run-time среды выполнения определяет некоторые регистры как рабочие (Scratch), а некоторые – как "сохраняемые при вызове" (Call Preserved). Scratch-регистры могут использоваться в ассемблерной подпрограмме без ограничений, не нужно заботиться о восстановлении их старых значений при выходе из подпрограммы. Если рабочих регистров недостаточно, то можно использовать CallPreserved-регистры, однако при входе в функцию следует сохранить их значения, а при выходе – восстановить. Следует стремиться к тому, чтобы использовать выделенные регистры по их прямому назначению (например, для работы со стеком), поскольку компилятор, библиотечные функции и прерывания предполагают именно такое их использование.

Компилятор предполагает, что режимы работы процессора во время выполнения `asm`-функции не изменяются.

Пример взаимодействия подпрограмм на С и ассемблере.

ТЕКСТ МОДУЛЯ НА ЯЗЫКЕ С

```
#include <stdio.h>
int a, b, c, d, e;
          // прототипы функций
int SumC(int u, int v, int x, int y, int z);
extern int SumAsm(int u, int v, int x,
                  int y, int z);

int SumC(int u, int v, int x, int y, int z)
{
 int temp1;
 temp1 = u + v + x + y + z;
 return temp1;
}

int main()
{
  a = 1;  b = 5;  c = 8;  d = 3;  e = 10;
  e = SumAsm(a, b, c, d, e);
  e = SumC(a, b, c, d, e);
  return 0;
}
```

Код процедуры main() с оптимизацией:

```
   …
   r2=10;              // r2 = e
   r1=3;               // r1 = d
   dm(i7,m7)=r2;       // push (e)
   r8=5;               // r8 = b
   r12=8;              // r12 = c
   r4=m14;             // r4 = 1
   dm(i7,m7)=r1;       // push (d)
//сохранить a,b,c,d,e
   dm(_a)=r4;  dm(_b)=r8; dm(_c)=r12; dm(_d)=r1; dm(_e)=r2;
// вызов
   cjump _SumAsm (DB);
            dm(i7,m7)=r2;
            dm(i7,m7)=pc;
// возврат, результат в r0 - это e
   modify(i7,2);       //очистка стека от параметров
// подготовка к следующему вызову
   r2=dm(_d);          // r2 = d
   dm(i7,m7)=r0;       // push (e)
   r12=dm(_c);         // r12 = c
   r8=dm(_b);          // r8 = b
   r4=dm(_a);          // r4 = a
   dm(i7,m7)=r2;       // push (d)
   dm(_e)=r0;          // сохранить e=SumC()
// вызов
   cjump _SumC (DB);
            dm(i7,m7)=r2;
            dm(i7,m7)=pc;
   dm(_e)=r0;          // сохранить результат в e
   modify(i7,2);       // очистить стек от параметров
   …
```

Код процедуры SumC без оптимизации (DeBug)

```
_SumC:
// место под локальную temp и автопеременные
    modify(i7,-5);
 // полученные параметры -> в автопеременные
    dm(-4,i6)=r4;
    dm(-3,i6)=r8;
    dm(-2,i6)=r12;
// вычисления (без оптимизации)
    r2=r4+r8;           // r2 = u+v
    r1=r2+r12;          // r2 = u+v+x
    r0=dm(1,i6);        // r0 = y  (i6+1)
    r2=r1+r0;           // r2 = u+v+x+y
    r1=dm(2,i6);        // r1 = z (i6+2)
    r0=r2+r1;           // r0 = u+v+x+y+z
    dm(-6,i6)=r0;       // temp = r0
// возврат
    i12=dm(m7,i6);
    jump(m14,i12) (DB);
            rframe;
            nop;
```

Код процедуры SumC с оптимизацией (Release)

```
_SumC:
   modify(i7,-4);       //место под локальную temp (лишнее)
   r2=dm(2,i6);
   r2=r2+r4,  r1=dm(1,i6);
   r2=r2+r1,  i12=dm(m7,i6);
   jump(m14,i12) (DB), r2=r2+r12;
            r0=r2+r8;
            rframe;
```

Модуль с кодом процедуры SumAsm, написанный вручную

```
#include <def21060.h>
#include <asm_sprt.h>

.section/pm seg_pmco;
.global _SumAsm;

//int SumAsm(int u, int v, int x, int y, int z)

_SumAsm:
   r0 = r4+r8, r4 = dm(0x1,i6);     // r0=u+v, r4=y;
   r0 = r0 + r4, r4 = dm(0x2,i6);   // r0=u+v+y, r4=z;
   r0 = r0 + r4, i12=dm(m7, i6);    // r0=u+v+y+z
   jump (m14, i12) (DB),  r0 = r0 + r12; // r0=Sum, возврат
     nop;
     rframe;
```

2.5.4. Вызов подпрограммы на C из подпрограммы на ассемблере

Вызов подпрограммы на C не отличается от вызова подпрограммы на ассемблере. Однако при вызове C/C++-функции следует учитывать, что она выполняется в рамках среды. Поэтому, если `asm`-процедура в некоторых случаях (в частности, при запрещенных прерываниях) может вызывать другую вложенную `asm`-процедуру таким же способом, как в программах на одном "чистом" ассемблере (где RTE-модель не поддерживается), то для C-функции необходим код вызова/возврата, отвечающий всем без исключения требованиям run-time среды:

- вызов C-функции выполняется с использованием инструкции `cjump` (и связанных с ней);

- передача параметров осуществляется через регистры и через программный стек в соответствии с правилами передачи параметров в функции;

- значения Scratch-регистров могут быть свободно изменены C-функцией. Поэтому, если они содержат важные значения, то перед вызовом "вложенной" функции следует их сохранить в стеке, а после возврата из подпрограммы – восстановить;

- C-функции не изменяют значения CallPreserved–регистров. Тем не менее, если CallPreserved-регистры модифицировались ассемблерной процедурой, целесообразно восстановить их исходные значения перед вызовом вложенной C-функции.

2.6. Использование библиотечных функций. Библиотеки C/C++ and DSP

В комплект поставки среды VisualDSP++ входят следующие основные run-time библиотеки:

- библиотеки C и C++ run time library, содержащие набор функций, ориентированных на базовые, низкоуровневые, системные операции и простейшие вычисления: работа с динамической памятью, работа со строками, базовые математические функции, диспетчер прерываний;

- библиотека DSP run time library, содержащая реализации базовых типовых алгоритмов цифровой обработки сигналов (преобразование Фурье, компандирование по A- и m-законам, свертка, работа с векторами и матрицами) и настройки некоторых системных параметров (управление таймером, настройка линий флагов);

- библиотека I/O library, поддерживающая стандартные возможности C по вводу/выводу данных (на консоль).

Описания функций каждой из этих библиотек содержится в нескольких заголовочных файлах. Для их использования достаточно подключить (#include) к программе соответствующий заголовочный файл: стандартный ldf-файл уже содержит пути и параметры поиска самих объектных библиотечных модулей. Каждая библиотека имеет собственную реализацию для ADSP21020, ADSP21060, ADSP21160 и выше.

Подробный перечень библиотечных функций приведен в руководстве по компилятору.

2.7. Разработка обработчиков прерываний на языке высокого уровня

2.7.1. Диспетчер прерываний

Особое значение при разработке обработчиков прерываний и драйверов устройств (портов ввода/вывода) имеет библиотека подпрограмм `signal.h`. В

ней содержатся процедуры для инициализации, разрешения, запрещения и прерываний и код программного диспетчера прерываний.

Программный секвенсор процессора SHARC ADSP поддерживает аппаратную схему (модель) обработки прерываний, предполагающую использование процессорных регистров защелки и маски прерываний, аппаратных стеков программного секвенсора и состояния для вызова обработчика и возврата из прерывания. Подобная схема является единственно возможной при отработке аппаратных и внешних прерываний (прерываний от таймера, от портов ввода/вывода, от внешних устройств).

С другой стороны, машинный код, сгенерированный компилятором C/C++, выполняется в рамках совершенно другой модели – среды run-time environment, с ее программным стеком, логическими сегментами и различными (по степени доступности) категориями регистров.

Поэтому написание обработчика прерываний (в первую очередь – аппаратных, так как именно аппаратные прерывания являются основными при вводе/выводе данных в системах ЦОС) на языке высокого уровня требует наличия определенного интерфейса между аппаратной схемой защелкивания и распознавания прерывания и программной моделью его обработки в рамках RTE.

Именно такой интерфейс предоставляет диспетчер прерываний. С точки зрения программиста он является подобием драйвера, реагирующим на некоторые аппаратные события (сигналы прерывания) и вызывающим связанные с данными событиями соответствующие С-функции. Вдобавок к этому перед вызовом обработчика диспетчер сохраняет контекст задачи (регистры процессора) в памяти, а после возврата из обработчика восстанавливает их. С точки зрения процессора диспетчер – это обычный обработчик прерывания, размещенный в таблице векторов прерываний (частично), получающий управление при переходе по вектору прерывания и полностью отвечающий за его обработку и возврат из прерывания.

Для настройки диспетчера прерываний используются две функции: `interrupt()` и `signal()`.

```
int interrupt(int SIG, void (*func)(int)) (int);
```

Функция `interrupt()` определяет способ обработки (вызываемую функцию-обработчик) сигнала прерывания с номером `SIG`, получаемого каждый раз в ходе выполнения программы, например:

```
#include <signal.h>
...
void SPORT0_Receive_Handler(int sig)
...
   // прерывание по приему от SPORT0
interrupt(SIG_SPR0I, SPORT0_Receive_Handler);
```


Второй аргумент функции может быть не только адресом функции, но и одной из констант: `SIG_DFL` (выполнять обработку по умолчанию) или `SIG_IGN` (игнорировать прерывание).

Функция возвращает значение `SIG_ERR=0x02`, если условный код переданного прерывания не существует. В противном случае – возвращает 0. Перечень условных кодов прерываний приведен в файле `signal.h`.

Следует помнить, что при настройке диспетчера заданное прерывание размаскируется (разрешается) в регистре `IMASK`, но переход на адрес функции обработки не подставляется напрямую в таблицу векторов прерываний: переход на заданный обработчик (`SPORT0_Receive_Handler`) осуществляется через диспетчер прерываний.

Функция `signal()` отличается от функции `interrupt()` тем, что позволяет обработать только первое прерывание. После этого диспетчер прерываний автоматически запрещает (маскирует) данное прерывание в регистре IMASK.

```
int signal(int SIG, void (*func)(int)) (int);
```

Функция `raise()` "посылает" сигнал определенного прерывания в исполняемую программу (в диспетчер прерываний), т.е. фактически устанавливает соответствующий бит в регистре `IRPTL`.

```
int raise(int SIG);
```

Функция `clear_interrupt()` сбрасывает бит соответствующего прерывания в регистре защелки IRPTL.

```
int clear_interrupt(int SIG);
```

Эта функция не может использоваться в случае, когда прерывание защелкивается по состоянию липких битов (например, по некорректной математической операции и т.п.), т.к. источник прерывания в регистре `STKY` не будет устранен.

Компилятор cc21k поддерживает 4 типа диспетчеров прерываний, отличающихся набором функциональных возможностей и производительностью (временем реакции на прерывание, т.е. задержкой вызова обработчика):

1) стандартный (normal) диспетчер прерываний (функции `interrupt()` и `signal()`). Сохраняет все Scratch-регистры и стек цикла. Вложенные прерывания разрешены. Нет ограничений на вложенность циклов (программная поддержка). Передает номер прерывания в подпрограмму-обработчик в качестве параметра. Переход на обработчик занимает от 125 до 160 тактов. Вызов обработчика выполняется как вызов обычной С-процедуры – полностью в рамках RTE-модели (командой `cjump` или `jump`). Выход из обработчика также осуществляется как из обычной подпрограммы на С (т.е. не по `rti` и не по `rts`, а командой `rframe` или `jump`);

2) быстрый (fast) диспетчер прерываний (функции `interruptf()` и `signalf()`). Количество вложенных циклов не должно превышать 6 уровней (т.е. аппаратная поддержка циклов). Стек цикла не сохраняется. Вложенность прерываний не запрещается (до 20 уровней). Диспетчер не передает обработчику

номер прерывания. Переход на обработчик занимает приблизительно 50-60 тактов. Код самого обработчика представляет собой обычную С-подпрограмму с присущими ей правилами вызова и возврата. Диспетчер выполняет сохранение/восстановление меньшего числа регистров, чем стандартный диспетчер;

3) "очень быстрый" (super fast) диспетчер прерываний (функции `interrupts()` и `signals()`). Количество вложенных циклов не должно превышать 6 уровней (т.е. аппаратная поддержка циклов). Стек цикла не сохраняется. Вложенные прерывания запрещены путем запрещения в диспетчере бита глобального разрешения прерываний `IRPTEN` (в регистре `MODE1`). Диспетчер не передает обработчику номер прерывания. Код самого обработчика представляет собой обычную С-подпрограмму с присущими ей правилами вызова и возврата. Перед вызовом обработчика диспетчер для сохранения контекста задачи не сохраняет регистры в стеке, а просто переключается на альтернативный набор регистров (при этом в регистрах работы со стеком поддерживаются правильные значения – копируются через `R`-регистры). Поэтому переход на обработчик занимает всего 30-40 тактов;

4) диспетчер прерываний для обработчиков на ассемблере (функции `interruptss()` и `signalss()`) или С-функций, скомпилированных с использованием директивы `#pragma interrupt`. Диспетчер сохраняет только 5-6 ключевых регистров, определяющих параметры run-time среды выполнения и режим работы процессора (`MODE1`, `ASTAT`). Сохранение/восстановление регистров возлагается на ассемблерную процедуру–обработчик (или компилятор, если обработчиком является С-функция, сгенерированная с директивой `#pragma interrupt`). Количество вложенных циклов не должно превышать 6 уровней (т.е. аппаратная поддержка циклов). Стек цикла не сохраняется. Вложенные прерывания не запрещаются (до 20 уровней вложенности). Диспетчер не передает обработчику номер прерывания. Переход на обработчик занимает около 30 тактов. Вызывается как `asm`-процедура (т.е. вызов/возврат идет через аппаратный стек программного секвенсора, а не в рамках модели RTE).

Реализованы также варианты стандартного диспетчера `interruptcb()` и `signalcb()` с поддержкой явного запрещения круговых буферов (путем обнуления) регистров `L0-L5` и `L8-L15`. Для работы диспетчера требуется примерно на 50 тактов больше для сохранения и восстановления `L`-регистров.

2.7.2. Определение обработчика прерываний

Директива `#pragma interrupt` должна *непосредственно* предшествовать реализации функции, например:

```
#pragma interrupt void SPORT0_Receive_Handler(int sig);
void SPORT0_Receive_Handler(int sig)
{
 int a;
...
return;
}
```


Эта директива указывает компилятору на то, что функция должна быть скомпилирована не как подпрограмма, а как обработчик прерывания. В частности, это приводит к тому, что все регистры (в том числе Scratch-регистры) должны быть сохранены и восстановлены при выходе из обработчика – это отличие от обычной процедуры. Выход из процедуры осуществляется по команде `rti`, а не как из С-функции по `jump`.

Эту директиву необходимо использовать только с диспетчерами, зарегистрированными функциями `interruptss()` и `signalss()`.

2.7.3. Особенности организации доступа к данным в С-модулях в программах с прерываниями

При описании данных, которые могут модифицироваться в обработчиках прерываний, обычно используется модификатор volatile, который сообщает компилятору, что значение этой переменной может изменяться в любой момент времени внешним (по отношению к исполняемой в данный момент функции) кодом. Это предотвращает выполнение компилятором обычных оптимизаций. Например:

```
int a, b, c, d;
...
c = x+a+b;
... // код не меняющий ни x, ни a
d = x+a;
```

Здесь `(x+a)` – общее подвыражение, и при вычислении переменной *c* компилятор сохраняет его в каком-либо временном регистре, чтобы затем подставить при вычислении переменной *d*, поскольку "видит", что между этими двумя выражениями *x* не изменялся. Если же переменная *x* объявлена как `volatile`, то компилятор скорее всего не будет его оптимизировать.

Другой пример.

```
while (x<5) {
          // что-то, не меняющее x
};
```

Здесь очевидно, что условие цикла, не будь *x* – `volatile`, не меняется. То есть его можно "соптимизировать", вынеся лишние вычисления из цикла

```
if (x<5)
while(1) {
          // ....
```

```
}
```

Если же *x* – `volatile`, то компилятор такой оптимизации делать не будет.

Исходный текст программы	Код, сгенерированный компилятором
`int x=1;` `int y=2;` `int z=3;` `int a,b;` `...` `  a = x+y+z;` `  b = x+y;`	`r2=6;` `r12=3;` `dm(_a)=r2;` `dm(_b)=r12;`
`volatile` `int x=1;` `int y=2;` `int z=3;` `int a,b;` `...` `  a = x+y+z;` `  b = x+y;`	`r2=dm(_x);` `r12=5;` `r2=r2+r12;` `dm(_a)=r2;` `r12=dm(_x);` `r8=2;` `r12=r12+r8;` `dm(_b)=r12;`

Отсутствие модификатора `volatile` в некоторых случаях может привести к неверной работе программы. С другой стороны, излишне частое его использование не позволит компилятору выполнить эффективную оптимизацию кода.

2.7.4. Пример программы обработки прерываний с модулями на С и ассемблере

Задание. Написать программу, которая читает по одному 32-битовому целому знаковому числу из последовательного порта и для последних N введенных чисел вычисляет скалярное произведение введенного вектора и массива с постоянными коэффициентами фильтра. Результат выдавать с той же частотой дискретизации через другой последовательный порт в виде 32-битовых целых значений.

Дополнительные требования. Программа должна быть реализована на С и содержать ассемблерные подпрограммы. На С реализуется инициализационная часть и обработчик прерывания, который выполняет чтение данных из последовательного порта, вызов ассемблерной процедуры обработки и вывод полученного результата в последовательный порт. На ассемблере удобнее реализовать процедуру инициализации и включения последовательного порта.

Для вычисления скалярного произведения двух векторов следует использовать библиотечную С-функцию, например `vecdot`. Поскольку она работает с двумя массивами ПЗ-данных, расположенными в памяти данных,

следует предварительно перевести введенные числа в ПЗ-формат, а после возврата из функции – привести результат обратно к целому типу.

Модуль constants.h определяет используемые в С-модуле константы.

```
#define N 10

#define cnSerialPort0         0
#define cnSerialPort1         1
#define cnReceive             0
#define cnTransmit            1
```

Модуль `module1.c` содержит описание буфера данных и массива коэффициентов, содержит код обработчика прерывания по приему от последовательного порта SPORT0, а также содержит инициализационную часть.

```
/*************************************************************/
/* Пример обработки данных на С с I/O через порты            */
/*************************************************************/
#include <def21060.h>
#include <signal.h>
#include "constants.h"


// Процедура установки и включения последовательного порта
extern void SetupSPORT(       int NumSPORT,  // номер порта
                              int Direction, // направление
                              int LenWord,   // длина слова
                              int Divisor);  // делитель частоты

// Процедура обработки буфера
extern "asm" int ProcessBuffer(int *InputBuffer, // входной буфер
                              int FirstIndex,  // номер первого эл-нта
                              int *CoefBuffer, // второй массив (const)
                              int Num);        // размер буфера

int SrcBuffer[N];             // буфер для накопления элементов массива
int cur_index;                // первое свободное значение
              // коэффициенты (второй массив)
float  Coefs[N]  = {1.0,  1.0,  1.0,  1.0,  1.0,  1.0,  1.0,  1.0,  1.0,  1.0};


//---------------------------------------
// Обработчик прерывания от SPORT0-прием
//---------------------------------------
//#pragma interrupt void SPORT0_Receive_Handler(int sig);
void SPORT0_Receive_Handler(int sig)
{
 int a;
       // прочитать значение из порта RX0
 asm (  "R0 = dm(0xE3);"
               "%0 = R0;"
               : "=d" (a)
               :
               : "r0" );
```

```
 SrcBuffer[cur_index] = a;                  // записать его в буфер
 cur_index = __builtin_circindex(cur_index, 1, N);  // передвинуть индекс
              // вызвать функцию обработки
 a = ProcessBuffer(SrcBuffer, cur_index, (int *)Coefs, N);
             // записать значение в порт TX1
 asm ( "R0 = %0;"
              "dm(0xf2) = R0;"
              :
              : "d" (a)
              : "r0" );
 return;
}
//----------------------------------------


//------------------------------------------------
// Инициализация
//------------------------------------------------
int main()
{
       // установить обработчик прерывания по приему через SPORT0
  interrupt(SIG_SPR0I, SPORT0_Receive_Handler);
       // сконфигурировать и включить порт SPORT0 на прием
  SetupSPORT(cnSerialPort0, cnReceive, 32, 0x10);
  SetupSPORT(cnSerialPort1, cnTransmit, 32, 0x10);
       // завершить main(). Диспетчер остается работать
  return 0;
}
```

Модуль `processing.asm` содержит процедуру инициализации последовательного порта (как на прием, так и на передачу) и процедуру обработки последних *N* принятых отсчетов, а также массив временных данных.

```
/*************************************************************/
// Обработка данных: модуль на asm
//*************************************************************/
#include <def21060.h>
#include <asm_sprt.h>
#include "constants.h"

.global _SetupSPORT;
.global ProcessBuffer;

.extern _vecdot;
// Прототип для вызова:
// float *vecdotf(    const float dm a[],
//                          const float dm b[],
//                          int samples);


//************************************************
.section/dm seg_dmda;
.var TempBuffer[N];
```



```
//************************************************
.section/pm seg_pmco;

//---------------------------------------------------------------
/* Установка и включение последовательно порта. Прототип:
//void SetupSport(    int NumSPORT,  // номер порта
//                    int Direction, // направление (прием-0/передача-1)
//                    int LenWord,   // длина слова
//                    int Divisor);  // делитель частоты       */
//---------------------------------------------------------------
// Доступ к аргументам:
// r4 - NumSport,  r8 - Direction,  r12 - LenWord
// dm(+1,i6) - Divisor
//---------------------------------------------------------------
_SetupSPORT:
                    // регистры сохранять не нужно - используем только
Scratch
              r0 = 0x00000401;      // r0 - SxCTLx spen=1, iclk=1
              r12 = r12-1;                  // slen = длина - 1
              r0 = r0 OR FDEP r12 by 4:5;   // ставим на место slen
       r12 = dm(+1,i6);             // r12 - делитель частоты - xDIVx

              r4 = pass r4;         // проверяем номер порта (0/1)
              if NE jump SPORT1_setup;
                    // для SPORT0
SPORT0_setup:
              r8 = pass r8;         // проверяем прием или передача
              if NE jump SPORT0_transmit;
              jump end_setup (db);
                    dm(0xe6) = r12;       // RDIV0 = Divisor;
                    dm(0xe1) = r0;        // SRCTL0 = r0;
SPORT0_transmit:
              jump end_setup (db);
                    dm(0xe4) = r12;       // TDIV0 = Divisor;
                    dm(0xe0) = r0;        // STCTL0 = r0;
                    // для SPORT1
SPORT1_setup:
              r8 = pass r8;         // проверяем прием или передача
              if NE jump SPORT1_transmit;
              jump end_setup (db);
                    dm(0xf6) = r12;       // RDIV0 = Divisor;
                    dm(0xf1) = r0;        // SRCTL0 = r0;
SPORT1_transmit:
              jump end_setup (db);
                    dm(0xf4) = r12;       // TDIV0 = Divisor;
                    dm(0xf0) = r0;        // STCTL0 = r0;
end_setup:
              // возврат из функции
              I12 = dm(-1,I6);
              jump (M14, I12) (db);
              nop;
              rframe;
_SetupSPORT.end: nop; //метка  завершения  функции  (для  отлад.  инф.)

//-------------------------------------------------
```

```
//---------------------------------------------------------------
/* Обработка буфера. Прототип:
int ProcessBuffer(    int *InputBuffer,      // входной буфер
                      int FirstIndex,        // номер первого эл-нта
                      int *CoefBuffer,       // второй массив (const)
                      int Num);              // размер буфера         */
//---------------------------------------------------------------
// Доступ к аргументам:
// r4 - ^InputBuffer,  r8 - FirstIndex,  r12 - ^CoefBuffer
// dm(+1,i6) - Num
//---------------------------------------------------------------
ProcessBuffer:
       // сохраняем регистры - переключаемся на теневые только i0-i3
              bit set MODE1 SRD1L;
              // разворачиваем массив
              r1 = dm(+1,i6);
              b3 = r4;
              r8 = r4+r8;           // текущий адрес в массиве
              i3 = r8;              // откуда начинаем по InputBuffer
              l3 = r1;              // длина массива
              i1 = TempBuffer;
          LCNTR = r1, DO xxx UNTIL LCE;
                     r0 = dm(i3,m6);
                     f0 = FLOAT r0;        // в ПЗ-формат
xxx:          dm(i1, m6) = r0;
              // восстанавливаем основные регистры только i0-i3
              bit clr MODE1 SRD1L;
              // вызов библиотечной функции vecdot()
              r4 = TempBuffer;
              r8 = r12;                    // r8 - ^CoefBuffer
              r12 = r1;
              cjump _vecdot (db);
                     dm(I7,M7) = R2;
                     dm(I7,M7) = PC;
              // результат - в F0
              r0 = FIX f0;                 // преобразовать в ФЗ
              // восстановить регистры
              // возврат из функции
              leaf_exit;
ProcessBuffer.end: nop;
//---------------------------------------------------------------
```

Приведенная реализация не является максимально оптимизированной по быстродействию или размеру кода и предназначена лишь для иллюстрации использования рассмотренных выше приемов программирования на С для решения реальных задач ЦОС.

3. Управление режимами оптимизации при использовании компилятора CC21K

3.1. Уровни оптимизации программного кода компилятором

Компилятор обеспечивает генерацию программного кода с различной степенью "оптимизированности". Степень оптимизации кода определяется флажками в окне проекта Project Options на вкладке опций компилятора (Compiler) в среде VisualDSP++.

Уровень отладки (**Debug**). Компилятор генерирует и записывает в объектный файл отладочную информацию для того, чтобы объектный код был сопоставим с исходным текстом программы (при пошаговом выполнении в отладочном режиме это приводит к синхронному перемещению строки подсветки в окне с текстом программы и в окне исполняемого дизассемблированного кода). Этот уровень соответствует включенному флагу "Generate debug information". Если больше никакие флаги и опции оптимизации не включены, то компилятор не выполняет даже вставку inline-функций, т.е. данный режим по существу не является режимом оптимизации.

Следующие два уровня оптимизации включаются путем установки флага "Enable Optimization" на вкладке компилятора. Программист имеет возможность выбирать стратегии оптимизации (по размеру (Size) кода или по быстродействию (Speed) кода), которая определяет приоритеты и степень "агрессивности" компилятора при выполнении оптимизационных процедур:

- уровень оптимизации "по умолчанию" (**Default**). Компилятор выполняет простейшую высокоуровневую оптимизацию, такую как вставка функций, явно описанных как inline, или оптимизация циклов. При включенном флаге "Generate Debug Information" компилятор также формирует таблицу символов и другую отладочную информацию. Однако, когда эти два режима используются совместно, в некоторых случаях точное соответствие между исполняемым кодом и исходной программой может быть нарушено. Это стандартный режим оптимизации, позволяющий получать наиболее быстрый исполняемый код, отвечающий стандартной интерпретации языков C/C++ и "осторожному консервативному" взгляду на связи между переменными;

- уровень локальной внутрипроцедурной оптимизации (**Procedural Optimization**). Компилятор выполняет более "агрессивную" оптимизацию каждой компилируемой процедуры. При включенном флаге "Generate Debug Information" возможности пошаговой отладки кода могут быть существенно ограничены.

Уровень межпроцедурной оптимизации (**Interprocedural Optimization, IPA**). В дополнение к локальной внутрипроцедурной оптимизации компилятор выполняет оптимизацию кода в масштабе всей программы. Данная опция включается установкой флага "Interprocedural Optimization" на вкладке компилятора.

Автоматическое встраивание inline-функций (**Automatic Inlining**). При включенном флаге компилятор может самостоятельно принимать решение о необходимости вставки тела функции вместо ее вызова в том случае, если это согласуется со стратегией оптимизации по размеру/скорости кода.

Дополнительные возможности по оптимизации программы и ее отдельных фрагментов могут быть получены при использовании директив оптимизации кода и циклов.

3.2. Особенности использования IPA-оптимизации

При генерации программного кода с IPA-оптимизацией компилятор на этапе компиляции программы создает особые (временные) объектные файлы с расширением `.ipa` или `.opa`, куда он записывает информацию об использовании переменных и вызовах функций. Особенностью IPA-оптимизации является то, что она выполняется после стадии начальной компоновки программы: после завершения первичного связывания модулей запускается специальная программа предкомпоновки (prelinker), которая повторно вызывает компилятор, "заставляет" его выполнить новую, окончательную оптимизацию временных объектных модулей и создает исполняемый модуль. Поэтому понятно, что использование IPA-оптимизации не дает преимуществ, если генерируется библиотечный файл, а не исполняемая программа (библиотечный файл представляет собой только объектный модуль).

При необходимости использования межпроцедурного анализа все модули проекта должны быть скомпилированы с поддержкой IPA, в противном случае поведение программы будет непредсказуемым.

Одним из действий, выполняемых компилятором при IPA-оптимизации, является удаление из проекта неиспользуемых переменных и невызываемых функций, что может существенно уменьшить объем кода.

При разработке собственных библиотечных модулей и проектов и их использовании может возникнуть ситуация, когда библиотечная функция обращается к подпрограмме с заранее предопределенным именем в пользовательском модуле. Если пользовательский проект будет генерироваться с IPA-опцией, то компилятор удалит функцию, на которую идет ссылка только из библиотечного модуля, поскольку он не видит вызов данной функции. Чтобы этого не произошло, следует использовать директиву `#pragma retain_name("имя_функции")`, чтобы оставить функцию в объектном коде.

Аналогичные проблемы возможны в случае, когда вызов функции выполняется через указатель, а не по ее имени.

Следует иметь в виду, что при использовании IPA-оптимизации значительная часть директив оптимизации компилятором игнорируется.

3.3. Дополнительные средства анализа и оптимизации кода

3.3.1. Профилирование исполняемого кода

Рассмотренные выше подходы к оптимизации кода предназначены для использования на этапе генерации объектных модулей и исполняемого файла. Фактически они представляют собой некоторые приемы для локального повышения эффективности выполнения тех или иных операций. Тем не менее, нередки ситуации, когда более существенный выигрыш может быть достигнут при некоторой модификации исходного алгоритма самим программистом (модернизация логики обработки данных, изменение порядка обхода массивов и структур, наконец, переход на язык низкого уровня). Было бы нерационально заставлять программиста доводить до совершенства весь программный код, поскольку различные его фрагменты выполняются с разной частотой и имеют различную трудоемкость. Например, инициализационная часть выполняется лишь один раз при загрузке и запуске программы, тогда как обработчик прерывания вызывается регулярно и должен отрабатываться как можно быстрее. К тому же алгоритмы обработки данных обычно весьма неоднородны по своей сложности: они могут содержать как относительно "простые" управляющие инструкции (для установки режима работы процессора, конфигурирования портов данных и т.п.), так и итерационные последовательности сложной математической обработки данных, занимающих наибольшую часть процессорного времени. Естественно, что наибольшее внимание при "доведении" программного кода следует уделить тем фрагментам программы, которые на этапе выполнения оказываются наиболее ресурсоемкими.

Для решения данной задачи используются интегрированные в отладчик механизмы профилирования – сбора информации о числе процессорных тактов, "проведенных" программой в том или ином фрагменте кода. Получаемые в результате профилирования данные в графическом и числовом виде характеризуют (обычно в процентах) длительность пребывания программы в заданном фрагменте кода (внутри тела функции, в цикле, на отдельной инструкции). Эта информация позволяет программисту выявить наиболее "узкие" места программы и заняться их модификацией в первую очередь (или игнорировать, если оптимизация не представляется возможной).

В среде VisualDSP++ профилирование включается путем выбора пункта меню Tools→Linear Profiling→New Profile. Статистические данные в окне профиля обновляются при каждой остановке программы на точке останова.

3.3.2. Автоматическая оптимизация по результатам профилирования

В большинстве реальных приложений в зависимости от характеристик наборов исходных данных разные фрагменты кода выполняются с различной частотой. Например, при кодировании речи на основе вокодерных методов,

первым шагом анализа кадра входного сигнала является оценка того, представляет ли он собой вокализованный или шумоподобный фрагмент сигнала. В зависимости от проведенной классификации, выбирается одна из двух ветвей алгоритма обработки. Поскольку подобный переход должен быть условным, то целесообразно организовать проверку условия таким образом, чтобы для более вероятного случая не приходилось очищать конвейер и тратить на этот переход дополнительные такты времени, т.е. выбрать оптимальное условие проверки, например:

`переход, если равно нулю` | `переход, если не равно нулю`

Установлено, что для речевого сигнала, передаваемого, скажем, по телефонной линии, доля вокализованных кадров составляет всего лишь порядка 20-30 %. Очевидно, что в данном случае выбор условия для программиста не представляет затруднения.

При автоматической генерации кода программист не может быть уверен, что выберется наиболее оптимальная форма условного перехода, поскольку компилятор изначально не располагает подобной статистикой. Однако, если он будет "знать" результаты профилирования программы для какого-либо "типового" набора исходных данных, то подобные оптимизации могут быть легко реализованы в автоматическом режиме без участия программиста. Такой режим оптимизации называется оптимизацией по результатам профилирования или оптимизацией "по профилю" (profile-guided optimization, PGO).

Для использования PGO-оптимизации в VisualDSP++ следует выполнить следующие действия.

Шаг 1. Подготовить один или несколько типовых наборов исходных данных в виде текстовых файлов, характеризующих реальный сигнал (пункт меню "Tools→PGO→Manage Data Sets"). Задать наборы данных (один набор данных соответствует одному эксперименту и оптимизации кода по его итогам). На рис. 5 заданы три набора данных `Data Set`. Для каждого набора данных следует указать файл для размещения результатов профилирования (с расширением `*.pgo`) и связать файл данных с портом или переменной в памяти (аналогично заданию потоков при работе с портами).

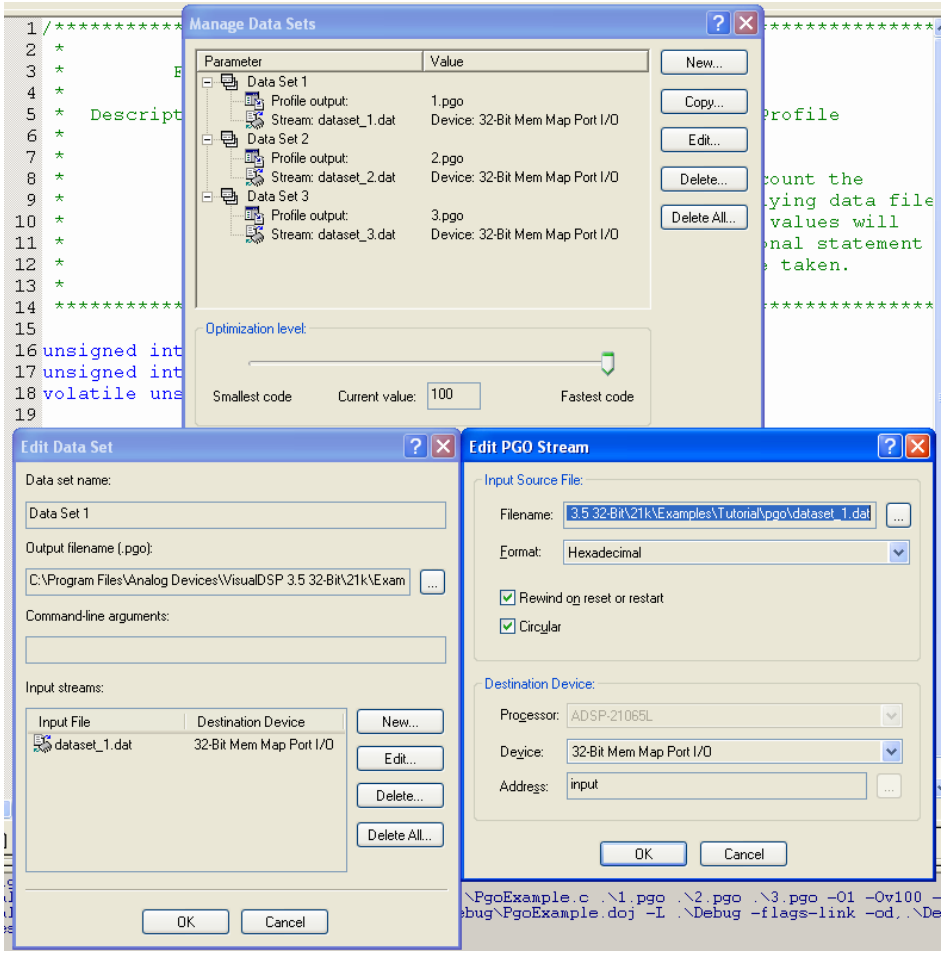

Рис. 5. Установка наборов данных для PGO-оптимизации

В соответствии с листингом программы, ключевым фактором оптимизации является значение переменной `input`. В зависимости от того, как часто ее значение оказывается четным или нечетным различные ветви условного оператора могут выполняться более или менее часто.

```
unsigned int odds = 0;
unsigned int evens = 0;
volatile unsigned int input = 0;

int main( int argc, char* argv[] )
{
   int i;
```

```
    for( i=0; i<256; i++ )
    {
        if( (input & 0xff) % 2 != 0 )
        {
            odds++;
        }
        else
        {
            evens++;
        }
    }
    return 0;
}
```

Шаг 2. Выполнить программу на заданных наборах данных. При этом для каждого набора данных выполняются следующие действия:

- с определенным набором опций компилируется и запускается на выполнение исходная программа, в которую подается заданный набор тестовых данных;
- формируется pgo-файл с профилем выполнения программы;
- программа перекомпилируется с учетом имеющегося файла профиля и вновь запускается на выполнение с тем же набором данных;
- вычисляются сводные результаты повышения/понижения производительности в процентах, которые записывается в xml-файл в виде, приведенном на рис.6:

Profile Guided Optimization Results

Generated on:	Mon Jan 10 02:08:41 2005
Application:	C:\Program Files\Analog Devices\VisualDSP 3.5 32-Bit\21k\Examples\Tutorial\pgo\Debug\PgoExample.dxe
Project:	C:\Program Files\Analog Devices\VisualDSP 3.5 32-Bit\21k\Examples\Tutorial\pgo\PgoExample.dpj
Optimization level:	100
Average cycle reduction:	15.94%

Data Set: Data Set 1

Command line:	
Input stream:	File: C:\Program Files\Analog Devices\VisualDSP 3.5 32-Bit\21k\Examples\Tutorial\pgo\dataset_1.dat Device: 32-Bit Mem Map Port I/O
PGO output:	C:\Program Files\Analog Devices\VisualDSP 3.5 32-Bit\21k\Examples\Tutorial\pgo\1.pgo
Before optimization:	2790 cycles
After optimization:	2790 cycles
Cycle reduction:	0.00%

Data Set: Data Set 2

Command line:	
Input stream:	File: C:\Program Files\Analog Devices\VisualDSP 3.5 32-Bit\21k\Examples\Tutorial\pgo\dataset_2.dat Device: 32-Bit Mem Map Port I/O
PGO output:	C:\Program Files\Analog Devices\VisualDSP 3.5 32-Bit\21k\Examples\Tutorial\pgo\2.pgo
Before optimization:	3046 cycles
After optimization:	2534 cycles
Cycle reduction:	16.81%

Data Set: Data Set 3

Command line:	
Input stream:	File: C:\Program Files\Analog Devices\VisualDSP 3.5 32-Bit\21k\Examples\Tutorial\pgo\dataset_3.dat Device: 32-Bit Mem Map Port I/O
PGO output:	C:\Program Files\Analog Devices\VisualDSP 3.5 32-Bit\21k\Examples\Tutorial\pgo\3.pgo
Before optimization:	3302 cycles
After optimization:	2278 cycles
Cycle reduction:	31.01%

Рис.6. Результаты оптимизации

Результаты оптимизации, показанные в последней строке каждой таблицы (для каждого набора данных), соответствуют характеристикам исходных файлов: в первом эксперименте в файле содержалось одинаковое количество четных и нечетных значений; во втором файле четных чисел в 3 раза больше, чем нечетных; в третьем – только четные значения.

ПРИЛОЖЕНИЯ

1. КРАТКОЕ ОПИСАНИЕ ДИРЕКТИВ КОМПИЛЯТОРА cc21k

Директивы выравнивания данных. Директивы выравнивания данных (директивы `align`, `pack` и `pad`) предназначены для выравнивания начальных адресов переменных при их размещении в памяти. Например, при отсутствии принудительного выравнивания (выравнивании по умолчанию на границу 1) структура `st_UserTag` будет занимать в памяти 9 слов, а массив `UserTagArray` – 90 слов. Использование директивы `#pragma align 8` перед описанием поля `ID` структуры приведет к тому, что каждое поле `ID` должно начинаться в памяти с адреса, кратного 8, т.е. структура `st_UserTag` будет занимать уже не 9, а 16 слов (ближайшее сверху к 9 число, кратное 8). Соответственно размер массива `UserTagArray` при этом будет составлять 160 слов.

```
struct st_UserTag {
#pragma align 8
          int ID[3];
          int Name[5];
          float Code;
   };
...
#pragma align 32
struct st_UserTag UserTagArray[10];
```

Введение дополнительного выравнивания по 32-словной границе для массива `UserTagArray` указывает компилятору на необходимость размещения массива с начального адреса, кратного 32 словам. Вторая директива `#pragma align` в данном примере не влияет на размер массива `UserTagArray`.

Хотя эти директивы (в особенности `pack` и `pad`) и могут использоваться для более "плотной" упаковки в памяти структур, имеющих различные уровни гранулярности, их основное практическое назначение заключается в выравнивании данных в памяти с целью организации корректного доступа к данным при вычислениях в SIMD-режиме.

Директива описания обработчика прерываний. Директива `interrupt` используется для определения функции как обработчика прерывания, например:

```
#pragma interrupt
void Int_Handler(int sig);
```

Данная директива указывает компилятору, что для функции, имя которой следует непосредственно за директивой interrupt, необходимо сгенерировать дополнительный исполняемый код, позволяющий использовать ее в качестве обработчика прерывания, вызов/возврат которого выполняется через аппаратную схему прерывания программного секвенсора, а не через программный диспетчер прерываний, функционирующий в среде выполнения C/C++. "Дополнительный"

генерируемый компилятором код отвечает за восстановление всех регистров (в том числе scratch-регистров) по завершении выполнения функции, а также обнуление L-регистров для всех используемых I-регистров.

Директиву `#pragma interrupt` следует использовать только с обработчиками, регистрируемыми в процессе выполнения программы в диспетчере прерываний с использованием функций `interruptss()` или `signalss()`. Поскольку для достижения максимальной производительности вызов/возврат и выполнение данных обработчиков осуществляется частично в контексте среды выполнения, частично с использованием аппаратной схемы обработки прерывания процессора, то данные функции не должны содержать вызовы других C/C++ функций (которые, естественно, работают в рамках run-time environment).

Директивы задания стратегии глобальной оптимизации. Директивы уровня оптимизации `optimize_off`, `optimize_for_space`, `optimize_for_speed`, `optimize_as_cmd_line` позволяют включать/выключать оптимизацию и выбирать стратегию компиляции модуля, нацеленную либо на минимизацию размера кода, либо на максимальное быстродействие программы. Типичным примером повышения быстродействия за счет увеличения размера исполняемого модуля является "разворачивание" циклов, которое широко используется для отказа от инструкций проверки условия выхода из цикла и выполнения команд условного перехода, если на этапе компиляции известно количество итераций. Следует учитывать, что при включенном режиме IPA-анализа и оптимизации данные директивы "не имеют силы".

Директивы управления видимостью внешних имен при компоновке. Для контроля видимостью внешних имен используются директивы `linkage_name` и `retain_name`. Использование директивы `linkage_name` с указанием идентификатора связывает данный идентификатор с именем последующей функции для всех ее вызовов, например:

Модуль 1	Модуль 2
`extern void COU(void);` `int function(void)` `{` `...` `for (i=0; i<N; i++)` `  {` `  COU();` `...` `  }`	`#pragma linkage_name _COU` `void Counter();` `int CountCalls = 0;` `void Counter(void)` `{` ` CountCalls++;` `}` `void dasd(void)` `{` ` Counter();` ` COU();` `}`

После такого описания функция Counter() будет "видна" в Модуле1 только под именем "COU", а внутри собственного модуля в программе на C/C++ будет доступна как по имени "COU", так и по имени "Counter" (тем не менее, в объектном модуле, получаемом при компиляции Модуля2, будет встречаться только имя "COU").

Директива retain_name "заставляет" компилятор не удалять из объектного модуля переменную и функцию, даже если IPA-анализ показал, что в компилируемых модулях к этому имени нет обращения.

```
#pragma retain_name
int ABC=0;
```

```
#pragma retain_name
int func_1(void)
{
...
}
```

Использование данной директивы необходимо при включенной IPA-оптимизации, например, если доступ к функции может осуществляться через модифицируемый указатель, а также в том случае, если доступ к функции или переменной осуществляется из библиотечного модуля, который не просматривается IPA-анализатором.

Директива описания поведения функции. Директивы описания поведения функции (alloc, pure, const, regs_clobbered, result_alignment) записываются перед объявлением функции и дают компилятору дополнительную информацию о действиях, выполняемых в функции, что позволяет выполнить дополнительную оптимизацию.

Например, директива alloc "говорит" компилятору о том, что возвращаемое функцией значение является указателем на уникальный объект в памяти и не может совпадать с каким-либо другим указателем в вызывающей подпрограмме (т.е. функция ведет себя аналогично функции malloc/alloc, возвращающей указатель на вновь созданный объект). Для следующего фрагмента кода на языке C компилятор (в зависимости от наличия директивы alloc) может сгенерировать два различных кода.

```
#pragma alloc
int *CreateBuffer(void);

int *CreateBuffer(void)
{
 return C;              // C - массив
}

int function(void)
{
 unsigned int i;
 int *Res = CreateBuffer();
```



```
 for (i=0; i<N; i++)
   {
   Res[i] = A[i]*B[i];
   }
```

Без директивы alloc	С директивой alloc
i1=_B; i0=_A; i4=r0; lcntr=100, do(pc,.P2L2-1)until lce; r2=dm(i1,m6); r12=dm(i0,m6); r2=r2*r12 (SSI); dm(i4,m6)=r2; .P2L2:	i1=_B; i0=_A; r12=dm(i4,m6); r2=dm(i0,m6); i4=r0; lcntr=99, do(pc,.P2L14-1)until lce; r12=r12*r2 (SSI),r2=dm(i0,m6); dm(i4,m6)=r12; r12=dm(i1,m6); .P2L14:

В программном коде, сгенерированном без директивы alloc, сначала читаются оба значения из массивов A (i0) и B (i1), затем вычисляется результат, который записывается в массив Res (i4). Информация о том, что массивы Res и, например, A не пересекаются, позволяет в правом столбце таблицы прочитать следующее значение из массива A (i0) еще до того, как текущее значение будет записано в массив Res (i4). Обратите внимание, что в левом столбце нельзя совмещать операции вычисления и чтения следующих операндов, так как компилятор предполагает, что массив Res (i4) может совпадать с любым из массивов A (i0) или B (i1).

Директивы pure и const перед объявлением функции говорят компилятору о том, что функция не модифицирует (а в случае const – даже не читает) никакие глобальные переменные и не осуществляет доступ (ни на чтение, ни на запись) к volatile-переменным, а работает лишь с переданными параметрами и глобальными переменными. Если в функцию передаются указатели, то она может читать память по соответствующим адресам, но не должна ее модифицировать (а для const – не должна даже читать). Это приводит к тому, что при каждом вызове функции результат ее работы будет одним и тем же. Поэтому компилятор может вызвать pure- или const-функцию один раз перед циклом, а внутри цикла использовать лишь результат ее вызова.

Директивы оптимизации циклов. Директивы оптимизации циклов дают компилятору дополнительную информацию о конкретном цикле, позволяя использовать более агрессивную стратегию его оптимизации. Директива помещается непосредственно перед операторами for, while или do. Наибольший эффект достигается при использовании директив с вложенными циклами. Компилятор всегда выполняет попытку векторизовать цикл

(естественно, если это не нарушает логику выполнения программы) и может использовать для этого информацию, полученную на этапе IPA-анализа.

Директива `SIMD_for` работает при генерации кода для процессоров ADSP-2116x и старше и говорит компилятору о необходимости векторизации цикла (т.е. параллельного (одновременного) выполнения двух итераций цикла) с использованием преимуществ SIMD-архитектуры процессорного ядра.

Директива `no_vectorization` запрещает компилятору векторизовать следующий за ней цикл для процессоров ADSP-2116x и старше.

Директива `vector_for` сообщает компилятору о возможности векторизации цикла. В данном случае решение о векторизации цикла с использованием SIMD-возможностей архитектуры процессора ADSP-2116x принимается компилятором на основе оценки возможного выигрыша от векторизации с учетом дополнительных накладных расходов. Например, для фрагмента программы

```
void copy(short *a, short *b)
{
 int i;
 #pragma vector_for
 for (i=0; i<100; i++)
       a[i] = b[i];
 }
```

при отсутствии директивы `vector_for` компилятор не сможет выполнить векторизацию цикла для SIMD-режима, поскольку "не уверен", что и указатель `a` и указатель `b` выровнены по границе двойного слова. Наличие директивы позволит выполнить цикл не за 100, а за 50 итераций.

Директива `no_alias` означает, что в последующем за ней цикле не содержится операций чтения и записи одной и той же области памяти. При наличии подобных перекрытий указателей (алиасов) возможности оптимизации кода могут быть существенно ограничены: сначала должны быть завершены все операции на более ранней итерации и лишь затем могут выполняться операции доступа к памяти для следующей итерации цикла.

Директива `loop_count(min [, max [, modulo]])` содержит сведения о минимальном и максимальном числе итераций, а также о кратности числа итераций цикла.

2. Библиотеки макроопределений для работы с портами и DMA-контроллером

В комплект поставки VisualDSP++ и компилятора cc21k входят файлы библиотеки макроопределений для упрощения работы с регистрами IOP-процессора, в частности для настройки DMA-пересылок через внешний порт и управления последовательными портами (подключаются заголовочными файлами, соответственно `dma.h` и `sport.h`). Они позволяют существенно упростить разработку кода и повысить его наглядность для программиста,

предпочитающего язык высокого уровня "неэлегантным" ассемблерным вставкам.

Реализация на C (запись в битовые поля)

```
SPORT0_Receive.rdiv.rckdiv = 0x10;

SPORT0_Receive.rxc.slen = 31;

SPORT0_Receive.rxc.iclk = 1;

// Записать все регистры
sport_setup(0, SPORT0_Receive);

// Отдельно затем - включить порт
sport_enable_receive(0);
```

"Расшифровка" компилятора с оптимизацией по быстродействию

```
r2=dm(_SPORT0_Receive+6);
r12=0xffff0000;
r8=0x10;
r2=r2 and r12;
r2=r2 or r8;
dm(_SPORT0_Receive+6)=r2;

r4=dm(_SPORT0_Receive+1); r1=496;
r4=r4 or r1;

r12=0x400;
r12=r4 or r12;
dm(_SPORT0_Receive+1)=r12;

i4=_SPORT0_Receive; // источник
r2=11;              // сколько регистров SPORT0
i12=0xE0;           // =0xE0 - регистры SPORT0
R0=R0-R0, PM(i12,M14)=M5;
R0=DM(i4,M6), PM(i12,M15)=R0;
LCNTR=r2, DO (PC, 1) UNTIL LCE;
          R0=DM(i4,M6), PM(i12,M14)=R0;
PM(i12,M14)=R0;

i0=0xE1;            // SRCTL0
r2=1;
r12=dm(m5,i0);
r2=r12 or r2;
dm(m5,i0)=r2;
```

Реализация в виде ассемблерной вставки

```
asm(        "r0 = 0x10;"
            "dm(0xe6) = r0;"
            "r0 = 0x5F1;"
            "dm(0xe1) = r0;"
            :
            :
            : "r0");
```

Размещение основных регистров управления SPORT0 в памяти

```
#define STCTL0 0xe0  /* Transmit Control Register  */
#define SRCTL0 0xe1  /* Receive Control Register */
#define TX0    0xe2  /* Transmit Data Buffer */
#define RX0    0xe3  /* Receive Data Buffer  */
#define TDIV0  0xe4  /* Transmit Divisor */
#define TCNT0  0xe5  /* Transmit Count Reg */
#define RDIV0  0xe6  /* Receive Divisor  */
#define RCNT0  0xe7  /* Receive Count Reg  */
#define MTCS0  0xe8  /* Multichannel Transmit Sel. */
#define MRCS0  0xe9  /* Multichannel Receive Sel. */
```

При использовании данной библиотеки следует обратить внимание на отдельное включение последовательного порта функцией `sport_enable_receive()`. Это вызвано тем, что все параметры последовательного порта должны быть установлены обязательно до его включения. При записи же регистров IOP-процессора от младших адресов к старшим с использованием цикла (как это реализовано в h-файле) сначала записывается `SRCTL0`, а затем регистр делителя `RDIV0`. Поэтому получается, что значение `RDIV0` фактически игнорируется аппаратурой порта.

Более того, чтение значения из элемента структуры, например, из регистра RX0 не приводит к физическому чтению данных из порта процессора. Как показывает приведенный пример, при включении стандартных функций и макроопределений в свою программу следует четко понимать особенности их реализации и трудоемкость генерируемого компилятором кода.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Солонина А., Улахович Д., Яковлев Л. Алгоритмы и процессоры цифровой обработки сигналов. – СПб:БХВ-Петербург, 2001.

2. Сергиенко А.Б. Цифровая обработки сигналов. Учебник для вузов. – СПб.:Питер, 2002.

3. Куприянов М.С., Матюшкин Б.Д. Цифровая обработка сигналов: процессоры, алгоритмы, средства проектирования. – СПб.: Политехника, 1998.

4. Гончаров Ю. Технология разработки ExpressDSP //Серия статей в журнале ChipNews за 2000.

5. Ануфриев И.Е. Самоучитель MatLab 5.3/6.x. – СПб.: БХВ-Петербург, 2003.

6. Мак-Кракен, Дорн У. Численные методы и программирование на Фортране. – М.: Мир, 1977.

7. Bateman A., Peterson-Stephens I. The DSP Handbook. Prentice Hall, 2002.

8. Marvin C., Ewers G. A Simple Approach to Digital Signal Processing. John Wiley and Sons Inc., New York, 1996.

9. Tomarakos J. Advanced digital audio demands larger word widths in data converters and DSPs (Part 1 of 3). – www.chipcenter.com.

10. Tomarakos J. Consider a 24- or 32-bit DSP to Improve Digital Audio (Part 2 of 3). – www.chipcenter.com.

11. Tomarakos J. A 32-bit DSP ensures no impairment of 16-bit audio quality (Part 3 of 3) – www.chipcenter.com.

12. Tomarakos J. The Relationship of Data Word Size to Dynamic Range and Signal Quality in Digital Audio Processing Applications. – DSP Field Applications, Analog Devices Inc., Norwood, MA.

13. VisualDSP++ 3.0 User's Guide for SHARC DSPs. – Analog Devices Inc., Norwood, MA, 2003.

14. VisualDSP++ 3.0 Linker and Utilities Manual for SHARC DSPs. – Analog Devices Inc., Norwood, MA, 2003.

15. VisualDSP++ 3.0 Getting Started Guide for SHARC DSPs. – Analog Devices Inc., Norwood, MA, 2003.

16. Code Composer Studio Getting Started Guide. – Texas Instruments Inc., 2001.

ОГЛАВЛЕНИЕ