ГЛАВА 3. ПРОЦЕССОРЫ С АРХИТЕКТУРОЙ VLIW

3.1. Архитектура процессора Itanium (Intel)

Архитектура IA-64, предложенная в последние годы компанией Intel, включает ряд новых свойств. Процессор Itanium, разрабатываемый в соответствии с IA-64, обладает следующими особенностями [12]:

- *Масштабируемость архитектуры* путем изменения количества функциональных устройств (в основном АЛУ различного типа) без переделки написанных программ;
- *Явное описание параллелизма* в машинном коде (EPIC - Explicitly Parallel Instruction Computing). Поиск зависимостей между командами производит не процессор, а компилятор;
- *Предикация* (Predication). Команды из разных ветвей условного ветвления снабжаются предикатными полями (полями условий) и запускаются параллельно;
- *Загрузка по предположению* (Speculative loading). Данные из медленной основной памяти загружаются заранее в предположении их будущей необходимости;

3.1.1. *Структура процессора Itanium*

Структура процессора представлена на рис.3.1. В процессоре имеются файлы "сдублированных" исполнительных функциональных устройств и связанных с ними регистров. Последние содержат множество портов чтения и записи и связаны с памятью, под которой понимается вся иерархия от кэша верхнего уровня до оперативной памяти. Такой подход позволяет строить "внутренне масштабируемые" МП: число ФУ, и, следовательно, уровень параллелизма, может возрастать по мере развития технологии производства.

В архитектуре IA-64 значительно возросло количество ресурсов различного назначения, включая большой набор регистров общего (РОН) и специального назначения. Это значительно

сокращает частоту обращения к памяти для загрузки и выборки промежуточных данных.

В Itanium имеется 128 64-разрядных РОН, каждый из них содержит дополнительный разряд NaT (Not a Thing), который указывает, является ли значение в регистре достоверным**.** Установку этого разряда могут производить команды исполнения по предположению.

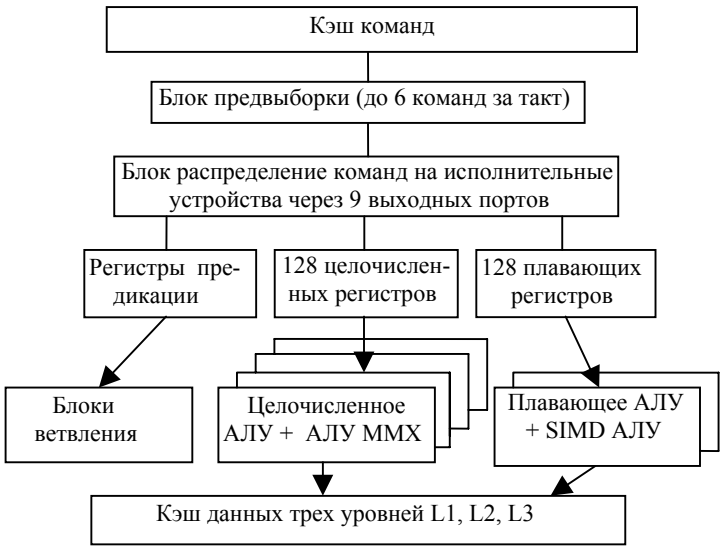

Рис. 3.1. Организация микропроцессора Itanium

Конвейер вычислительного ядра включает 10 ступеней, обрабатывает команды в порядке их расположения в программе и функционально разделен на 4 блока:

- Блок выборки команд обеспечивает выборку и предвыборку до 6 команд за такт, устанавливает иерархию ветвлений, связывает буфера и содержит три ступени;
- Блок доставки команд обеспечивает диспетчирование до 6 команд на 9 портах, переименование регистров, управление регистровым стеком, содержит две ступени;

• Блок доставки операндов обеспечивает связь блока регистров с памятью и АЛУ, наблюдение и управление состоянием регистров, предсказание зависимостей, содержит две ступени;

• Блок исполнения управляет работой нескольких комплектов однотактных АЛУ и устройств обращения к памяти. Один комплект содержит целочисленное АЛУ, плавающее АЛУ, АЛУ MMX для целочисленных вычислений и SIMD АЛУ для плавающей точки. Предполагается, что на первых этапах таких комплектов может быть до четырех. Кроме того блок исполнения обеспечивает упреждающую выборку данных, обработку предикатов, выполнение ветвлений, содержит три ступени.

Аппаратура для выполнения операций с плавающей точкой содержит 82-разрядное АЛУ, которое обеспечивает поддержку вычислений в широком диапазоне числовых применений. Два конвейеризованных вычислительных блока с расширенной точностью для умножения с накоплением FMAC обеспечивают до 4 плавающих операций за такт. Для 3D графики введены два дополнительных FMAC с одиночной точностью, которые в режиме ОКМД выполняют до 8 операций за такт. Имеется кэш уровня L3 на 4 Мбайта, кэш уровня L2 и 128 82-разрядных регистра общего назначения. Скорость обмена между кэш L2 и L3 составляет 2 операнда двойной точности за такт, а между L2 и регистрами – 4 операнда за такт.

3.1.2. Параллелизм

В IA-64 в ассемблерных программах, обычно создаваемых компилятором, группы команд, предназначенные для параллельного исполнения, должны отмечаться явно. Явный параллелизм – ключевая особенность IA-64. Эти группы в ассемблерном коде разделены признаком stop (s-разряд), и все команды одной группы могут исполняться параллельно без дополнительной проверки, но в зависимости от доступности ресурсов.

Команды IA-64 упаковываются (группируются) компилятором в *связку* длиною в 128 разрядов. Связка содержит 3 команды и шаблон, в котором указанываются зависимости между ко

мандами (можно ли с командой к1 запустить параллельно к2, или же к2 должна выполниться только после к1) , а также между другими связками (можно ли с командой к3 из связки с1 запустить параллельно команду к4 из связки с2). Заметим, что границы связок не имеют никакой корреляции с границами групп команд, поскольку группы команд могут расширяться на произвольное число связок. Группа команд начинается и заканчивается там, где в ассемблерном коде установлен s-разряд, а также динамически, когда выполнено ветвление.

Перечислим все варианты составления связки из 3-х команд (знак || обозначает возможность параллельного исполнения смежных команд, s соответствует stop-разряду):

i1 || i2 || i3 - все команды исполняются параллельно
i1 s i2 || i3 - сначала i1, затем исполняются параллельно i2 и i3
i1 || i2 s i3 - параллельно исполняются i1 и i2, после них - i3
i1 s i2 s i3 - последовательно исполняются i1, i2, i3

Одна такая связка, состоящая из трех команд, соответствует набору из трех функциональных устройств процессора. Процессоры IA-64 могут содержать разное количество таких блоков, оставаясь при этом совместимыми по коду. Ведь благодаря тому, что в шаблоне указана зависимость и между связками, процессору с N одинаковыми блоками из трех функциональных устройства будет соответствовать командное слово из N*3 команд (N связок). Таким образом должна обеспечиваться масштабируемость IA-64.

Команды IA-64 имеют уникальный формат, который позволяет компилятору направлять аппаратное исполнение без особого увеличения объема программного обеспечения. Рис.3.2 показывает единую 128-разрядную связку, содержащую три команды вместе с шаблоном ("template"), содержащим информацию о связке. Каждая команда содержит: поле кода операции, предикатный регистр (6 разрядов), адрес операнда (7 разрядов), адрес другого операнда (7 разрядов), адрес результата (7 разрядов), код операции расширения. Поле «Шаблон» содержит информацию о параллелизме внутри связки и между связками.

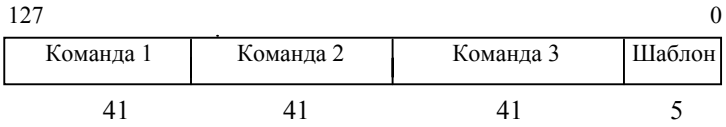

Рис.3.2. Структура связки

IA-64 определяет 6 типов команд и 4 типа исполнительных устройств (команды, память, плавающая точка и ветвление). Нижеследующая таблица представляет связь типов команд и устройств.

Тип команды	Описание	Тип устройства
A	Целочисленная операция в АЛУ	I или M
I	Целочисленная операция вне АЛУ	I
M	Обращение к памяти	M
F	Операция с плавающей точкой	F
B	Операция ветвления	B
L	Операция с непосредственным операндом	I

Первые четыре разряда шаблона описывают тип команд в связке и расположение s-разрядов в ней. Например, при кодировании MMI состав команд связки таков: память, память и целочисленная команда.

Поле шаблона используется процесором для быстрого декодирования связки и посылки команд на соответствующие устройства. Последний (пятый) разряд шаблона отмечает, может ли следующая связка команд выполняться параллельно с текущей связкой.

Связки могут быть сцеплены, создавая группы произвольной длины. Несколько примеров шаблонов представлено в нижеследующей таблице. Например, шаблон 00 и 04 не имеют stop'ов, а шаблон 03 имеет stop после slot 1 и другой после slot 2. Остальные шаблоны также имеют stop'ы.

Шаблон	Слот 0	Слот 1	Слот 2
00	M	I	I
01	M	I	I s
02	M	I s	I
03	M	I s	I s
04	M	L	X
05	M	L	X s

Команды media (аналогичны командам технологии MMX) параллельно обрабатывают содержимое 128-разрядных РОН как 8, 16 или 32-разрядные независимые элементы данных. Имеется 3 класса media команд: арифметические, сдвига и перестановки данных. На рис.3.3 приводится пример команды параллельного сложения элементов двух векторов. Более подробно этот вопрос рассматривается в главе 4 на примере технологии MMX для вычислений с фиксированной и плавающей точкой.

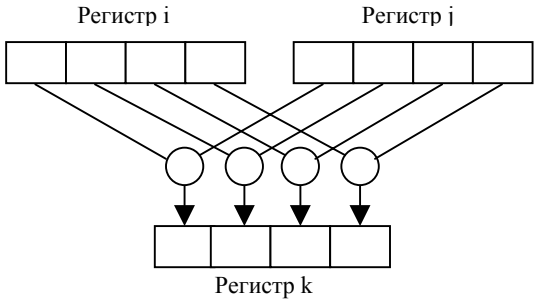

Рис.3.3. Команда сложения двух векторов, каждый вектор содержит четыре слова

Группу из 12 параллельных команд один процессор может выполнить за два такта, а более совершенный процессор – и за один такт. Тот же самый исполнительный код может исполняться на обоих процессорах без переделок. Примеры таких действий представлены на рис.3.4. Такая логика выборки делает процессор более сложным, чем чистый VLIW, однако возмож

ность строить семейство двоично-совместимых процессоров много важнее.

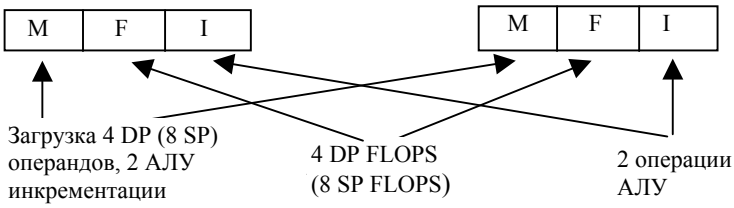

Рис.3.4. Две связки (6 команд) позволяют выполнить за 1 такт 12 операций двойной точности или 20 операций одинарной точности

3.1.3. *Предикация*

Вторая основная особенность IA-64 и, соответственно, процессора Itanium – кардинально новый подход к работе с переходами, названный предикацией (predication). Для этого в архитектуру IA-64 введены 64 одноразрядных регистра предикатов, каждый из них может находиться в состоянии ложь/истина. К командам IA-64 добавляется поле-предикат, говорящее аппаратуре, следует ли выполнять данную команду. Рассмотрим следующий код:

```
r1 = r2 + r3
```

Если использовать предикацию, то код будет иметь следующую форму:

```
If (p5) r1 = r2 + r3
```

В этом примере p5 – предикат, который решает, будет ли выполняться команда. Если p5 – истина, то команда будет выполняться, в противном случае команда пропускается, то есть она эквивалентна команде nop.

Более сложный пример выглядит так:

```
if (a > b) c = c +1
else d = d*e + f
```

Этот код может быть реализован в IA-64 с использованием предикатов, так что ветвления вследствие этого устраняются. В зависимости от того, будет ли предикат истина (pT) или ложь (pF), один из следующих операторов кода будет пропущен:

```
pT, pF = compare (a > b)
if (pT)  c = c + 1
if (pF) d = d*e + f
```

Благодаря большому числу функциональных устройств МП может выполнять сразу две небольшие ветви программы параллельно, и таким образом некоторые потенциальные команды перехода исключаются. Это позволяет существенно поднять общий уровень распараллеливания, поскольку исчезают затрудняющие распараллеливание маленькие блоки программных кодов, окаймленные командами перехода. Такой подход способствует и уменьшению числа неверных предсказаний перехода.

Технология "отмеченных команд" является наиболее характерным примером "дополнительной ноши", перекладываемой на компиляторы.

Обычно компилятор транслирует оператор ветвления (например, IF-THEN-ELSE) в блоки машинного кода, расположенные последовательно в потоке. В зависимости от условий ветвления процессор выполняет один из этих блоков и перескакивает через остальные.

Современные процессоры стараются предсказать результат вычисления условий ветвления и предварительно выполняют предсказанный блок (частично это делает и Itanium). При этом в случае ошибки много тактов тратится впустую. Сами блоки зачастую весьма малы – всего две или три команды, а ветвления встречаются в коде в среднем каждые шесть команд. Такая структура кода делает крайне сложным и неэффективным его параллельное выполнение.

Когда компилятор для IA-64 находит оператор ветвления в исходном коде, он исследует ветвление, определяя, стоит ли его "отмечать". Если такое решение принято, компилятор помечает все команды, относящиеся к одному пути ветвления, предикатом с уникальным идентификатором. Например, путь, соответствующий значению условия ветвления TRUE, помечается предикатом P1, а каждая команда пути, соответствующего значению условия ветвления FALSE - предикатом P2.

Система команд IA-64 определяет для каждой команды 6-битное поле для хранения этого предиката. Таким образом, одновременно могут быть использованы 64 различных предиката. После того, как команды "отмечены", компилятор определяет, какие из них могут выполняться параллельно. Это опять требует от компилятора знания архитектуры конкретного процессора, поскольку различные кристаллы архитектуры IA-64 могут иметь различное число и тип функциональных узлов.

Кроме того, компилятор, естественно, должен учитывать зависимости по данным (две команды, одна из которых использует результат другой, не могут выполняться параллельно). Поскольку каждый путь ветвления заведомо не зависит от других, почти всегда будет найдено какое-то количество параллелизма.

Заметим, что не все ветвления могут быть отмечены, так в некоторых случаях применение этой технологии может привести к тому, что будет затрачено больше тактов, чем сэкономлено.

После этого компилятор транслирует исходный код в машинный и упаковывает команды в 128-битные связки. Команды в связках не обязательно должны быть расположены в том же порядке, что и в машинном коде, и могут принадлежать к различным путям ветвления. Компилятор может также помещать в одной связке зависимые и независимые команды, поскольку возможность параллельного выполнения определяется шаблоном связки. В отличие от некоторых архитектур со сверхдлинными словами команд (VLIW), IA-64 не добавляет команд "нет операции" (NOPS) для дополнения связок.

Во время выполнения программы IA-64 просматривает шаблоны, выбирает взаимно независимые команды и распределяет их по функциональным узлам. После этого производится распределение зависимых команд. Когда процессор обнаруживает "отмеченное" ветвление, вместо попытки предсказать значение условия ветвления и перехода к блоку, соответствующему предсказанному пути, процессор начинает параллельно выполнять блоки, соответствующие всем возможным путям ветвления. Таким образом, на машинном уровне ветвления нет.

Разумеется, в какой-то момент процессор наконец вычислит значение условия ветвления в нашем операторе IF-THEN-ELSE. Предположим, оно равно TRUE, следовательно, правильный путь отмечен предикатом P1. В процессоре 6-разрядному полю предиката соответствует набор из 64 предикатных регистров длиной 1 бит. Процессор записывает 1 в регистр P1 и 0 во все остальные.

К этому времени процессор, возможно, уже выполнил некоторое количество команд, соответствующих обоим возможным путям, но до сих пор не сохранил результат. Перед тем, как сделать это, процессор проверяет соответствующий предикатный регистр. Если в нём “1” – команда верна и процессор завершает её выполнение и сохраняет результат. Если “0” – результат сбрасывается.

Если компилятор не "отметил" ветвление, IA-64 действует практически так же, как и современные процессоры: пытается предсказать путь ветвления и т.д. Испытания показали, что описанная технология позволяет устранить более половины ветвлений в типичной программе, и, следовательно, уменьшить более чем в два раза число возможных ошибок в предсказаниях. Программа для IA-64 обладает также большим параллелизмом.

3.1.4. *Загрузка по предположению*

Третья ключевая особенность IA-64 – применение загрузки по предположению (speculation). Известно, что задержки при выполнении команд загрузки load в регистр приводят к сущест

венному понижению производительности МП, даже если операнд находится в одном из кэшей, а не в оперативной памяти.

Особенностью IA-64 является предварительная загрузка данных. Она позволяет не только загружать данные из памяти до того, как они понадобятся программе, но и генерировать исключение только в случае, если загрузка прошла неудачно. Цель предварительной загрузки – разделить собственно загрузку и использование данных, что позволяет избежать простоя процессора. Как и в технологии "отмеченных команд" здесь также сочетается оптимизация на этапе компиляции и на этапе выполнения.

Сначала компилятор просматривает код программы, определяя команды, использующие данные из памяти. Везде, где это возможно, добавляется команда предварительной загрузки на достаточно большом расстоянии перед командой, использующей данные и команда проверки загрузки непосредственно перед командой, использующей данные.

На этапе выполнения программы процессор сначала обнаруживает команду предварительной загрузки и, соответственно, пытается загрузить данные из памяти. Иногда попытка оказывается неудачной - например, команда, требующая данные, находится после ветвления, условия которого ещё не вычислены. "Обычный" процессор тут же генерирует исключение. IA-64 откладывает генерацию исключения до того момента, когда встретит соответствующую команду.

IA-64 реализует загрузки по предположению или безошибочные загрузки. Такая загрузка, отмеченная суффиксом .s, не запускает исключения. Более того, если исключение имеет место, целевой регистр будет маркирован недостоверным. Это простой механизм, требующий только, чтобы регистр имел разряд достоверности. При использовании этого механизма загрузка в коде может быть размещена на самой ранней позиции, как только адрес станет доступным для вычисления. Если данные никогда не проверяются, никакого исключения не будет запущено. Если исключение происходит, когда необходимы данные, исключение будет распознано в "home block" оригинала загруз

ки. Загрузка по предположению обеспечивает компилятору максимальную гибкость по отношению к скрытым задержкам кэша.

Есть два вида загрузки по предположению: по управлению и по данным.

Загрузка по управлению. Она позволяет перемещать загрузки выше ветвлений. Для этого используются специальные NaT-разряды, которые являются частью целочисленного регистра. Когда загрузки по предположению вызывает исключение, оно не устанавливается немедленно. Вместо этого на регистре назначения устанавливается NaT-разряд. Следующие команды загрузки по предположению, которые используют регистр с установленным NaT-разрядом, распространяют установку до тех пор, пока не встретится команда проверки (checks for). Например, в отсутствии другой информации компилятор для традиционной архитектуры не может безопасно переместить загрузку выше ветвления в нижеприведенной последовательности:

```
(p1)   br.cond.dptk L1          // cycle 0
       id   r3 = r[5];;         // cycle 1
       shr  r7 = r3, r87        // cycle 3
```

Полагая, что задержка загрузки составляет два такта, команда правого сдвига остановится на 1 такт. Однако, использование загрузки по предположению и проверки позволяют исключить два такта, если переписать этот код по другому:

```
       Id.s    r3 = [r5]        //Earlier cycle
        ……..                    //Other instruction
(p1)   br.cond.dptk;;           //Cycle 0
       chk.s   r3,              //Cycle 1
       shr     r7 = r3, r87     //Cycle 1
```

Загрузка по данным. Она позволяет переместить загрузку выше возможного конфликта при обращении к памяти. Предварительные загрузки относятся исключительно к загрузкам данных. Рассмотрим загрузку и сохранение в следующем примере:

```
st      [r55] = r45              //Cycle 0
ld      r3 = [r5];;              //Cycle 0
shr     r7= r3, r87              //Cycle 2
```

IA-64 позволяет программисту или компилятору перемещать загрузку выше операции сохранения, даже если неизвестно, будут ли эти операции перекрываться при обращении к памяти. Это выполняется с помощью специальной команды предварительной загрузки и команды проверки:

```
ld.a     r3 = [r5]          //Advanced load
  …….                       //Other instruction (possibly hundreds)
st       [r55] = r45        //Cycle 0
ld.c     r3 = [r5]          //Cycle 0-check
shr      r7 = r3,r87        //Cycle 0
```

Заметим, что команда shr в этом расписании могла бы запускаться в такте 0, если бы не было никаких конфликтов между предварительной загрузкой и вмешивающимся сохранением. Если конфликт был, команда проверки (ld.c) определила его и повторила загрузку.

Загрузка по предположению минимизует влияние задержки памяти. Следующая программа написана для традиционной архитектуры:

```
add  t1+1
compt t1 > t2
jump
load a[t1-t2]
load b[i]
add b[i] + 1
```

В программе для IA-64 загрузка перенесена как можно раньше:

```
add t1+1
load.s a[t1-t2]
comp t1> t2
jump
check.s
```

```
load b[i]
add b[i] + 1
```

В этой программе на второй команде определяется исключение, которое распространяется на дальнейшее выполнение программы. Загрузка по предположению позволяет поставить load.s выше ветвления.

Разработка микропроцессора, способного конкурировать с Itanium, ведется в России. Микропропроцессор E2k (Эльбрус-2000) также относят к архитектуре EPIC, но разработка архитектуры E2k является самостоятельной и восходит к разработке ЭВМ семейства Эльбрус. Предположительно E2k начнет выпускаться в конце 2001 года и по быстродействию и другим характеристикам превзойдет Itanium [23, Приложение 2].

3.2. Архитектура процессора C6 (Texas Instruments)

В этом разделе будет рассмотрена архитектура программируемых цифровых сигнальных процессоров корпорации Texas Instruments серии TMS320C6xx[24], выполненных по архитектуре VLIW. В серии имеются процессоры для работы с фиксированной (C62x) и плавающей (C67x) точкой. Для простоты в дальнейшем будет рассматриваться только процессор с фиксированной точкой, условно называемый C6.

3.2.1. *Организация процессора и система команд*

Особенностью процессора является наличие двух практически идентичных вычислительных блоков, которые могут работать параллельно и обмениваться данными. Каждый блок содержит четыре фунциональных устройства, которые также могут работать параллельно.

На рис.3.5 представлена структура процессора. Блоки выборки программ, диспетчирования и декодирования команд могут каждый такт доставлять функциональным блокам до восьми

32-разрядных команд. Обработка команд производится в в путях A и B, каждый из которых содержит 4 функциональных устройства (L, S, M, и D) и 16 32-разрядных регистра общего назначения. Каждое ФУ одного пути почти идентично соответствующему ФУ другого пути.

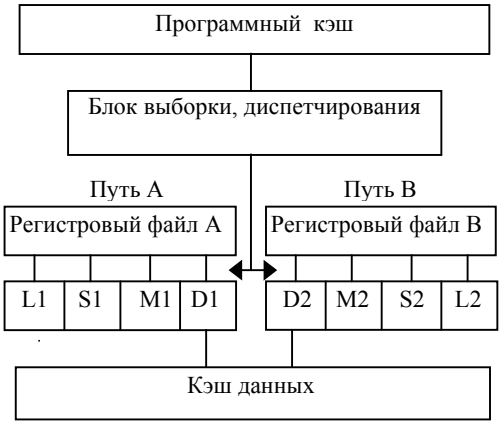

Рис.3.5. Структура микропроцессора C6

Каждое ФУ прямо обращается к регистровому файлу внутри своего пути, однако существуют дополнительные связи, позволяющие ФУ одного пути получать доступ к данным другого пути. Функциональные устройства описаны в таблице:

Функциональное устройство	Операции с фиксированной точкой
L	32/40-разрядные арифметические, логические и операции сравнения
S	32-разрядные арифметические, логические операции, сдвиги, ветвления,
M	16-разрядное умножение
D	32-разрядное сложение, вычитание, вычисление адресов, операции обращения к памяти

В этом параграфе приводится много примеров программ на ассемблере. Чтобы облегчить их понимание, опишем структуру ассемблерной команды. Команда содержит следующие поля:

метка:	||	[условие]	команда	устройство	операнды	комментарий

Поле «Метка» именует строку кода и используется, например, для организации переходов. Поле « || » отмечает команду, которая может выполняться параллельно с предыдущей. Поле «Условия» содержит имя одного их пяти регистров условий: A1, A2, B0, B1 и B2. Все команды C6 являются условными, при этом:

- Если условие не описано, команда всегда выполняется;
- Если условие описано и истинно, команда выполняется;
- Если условие описано и неверно, команда не выполняется.

Поле «Команды» содержит либо директивы ассемблера, либо коды выполняемых операций. Поле «Функциональные устройства» содержит имена нужных для данной команды ФУ, входящих в состав восьми ФУ процессора C6. Поле «Операнды» хранит имена операндов и результата.

В ассемблере все команды обязаны иметь адрес результата. Адрес результата должен находится в том же регистровом файле, что и исходный операнд. Один исходный операнд для каждого регистрового файла в одном исполняемом пакете может поступать из файла, противоположного тому, где расположен другой исходный операнд. Это отмечается символом X, как показано в нижеследующем примере:

```
ADD     .L1     A0, A1, A3
ADD     .L1X    A0, B1, A3
```

Команды C6 используют три типа операндов: регистровые; константы, описанные в ассемблере; операнды, размещенные в памяти и задаваемые их адресами.

Наконец, поле «Комментарии» содержит пояснительную информацию, необходимую для документирования программы.

Команды с фиксированной точкой процессора C6 являются обычными для большинства цифровых сигнальных процессоров, имеют общепринятую мнемонику и поэтому их коды опе

раций как правило не поясняются. В необходимых случаях для некоторых команд в примерах программ будут даны соответствующие пояснения.

3.2.2. *Параллельные операции*

За один такт из памяти всегда извлекается 8 команд. Они составляют пакет выборки длиною 256 разрядов. В нем содержится 8 32-разрядных команд.

Выполнение отдельной команды частично управляется *p*-разрядом, расположенным в этой команде. Этот разряд соответствует значку " || " в ассемблере и определяет, может ли команда выполняться параллельно с другими командами параллельной команды. *p* – разряды сканируюся слева направо (к более старшим адресам команд в пакете). Если в команде *i* *p*-разряд равен 1, то *i*+1 команда может выполняться параллельно с *i*-ой. В противном случае команда *i*+1 должна выполняться в следующем цикле, то есть после цикла, в котором выполнялась команда *i*. Все команды, которые будут выполниться в одном цикле, образуют исполнительный пакет. *Пакет выборки* и *исполнительный пакет* это различные объекты.

Исполнительный пакет может содержать до 8 команд. Каждая команда этого пакета должна использовать отдельное функциональное устройство. Исполнительный пакет не может пересекать границу 8 слов. Следовательно, последний *p*-разряд в пакете всегда устанавливается в 0 и каждый пакет выборки запускает новый исполнительный пакет. Имеется три типа пакетов выборки: полностью последовательные, полностью параллельные, и смешанные. В полностью последовательных пакетах *p*-разряды всех команд установлены в 0, следовательно, все команды выполняются строго последовательно. В полностью параллельных пакетах *p*-разряды всех команд установлены в 1 и все команды выполняются параллельно. Пример смешанного пакета выборки из восьми 32- разрядных команд приведен на рис.3.6.

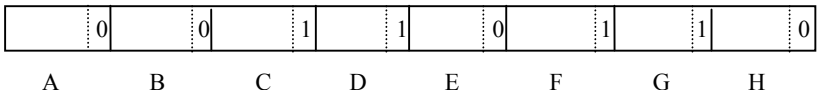

Рис.3.6. Частично последовательный пакет выборки

Состав исполнительных пакетов, соответствующих рис.3.6, представлен ниже:

Исполнительные пакеты (такты)	Команды		
1	A		
2	B		
3	C	D	E
4	F	G	H

Код примера 3.6 может быть представлен следующим образом:

```
    instruction A
    instruction B
    instruction C
||  instruction D
||  instruction E
    instruction F
||  instruction G
||  instruction H
```

Знак || как и прежде обозначает, что команда может выполняться параллельно с предыдущей.

Ветвление в середине исполнительного пакета. Если имеется ветвление в середине пакета, все команды с меньшими адресами игнорируются. В пакете 3.6, если команда D выполняет ветвление, то выполнятся только команды D и E. Хотя команда C в том же пакете, она игнорируется. Команды A и B также игнорируются, поскольку они принадлежат предыдущему исполнительному пакету. Если результат зависит от исполнения A, B или C, то ветвление в середине пакета приведет к ошибочному результату.

Условные операции. Все команды могут быть условными. Условие управляется 3-разрядным полем кода операции (creg), которое задает номер проверяемого регистра, и 1-разрядным полем (z), которое задает условие проверки. Ситуация z = 1 соответствует тесту на равенство 0; если же z = 0, то это тест на неравенство 0. Случай creg = 0 и z = 0 рассматривается как безусловное выполнение команды.

Условные команды представляются в коде с использованием квадратных скобок [ ], охватывающих название регистра условия. Ниже показаны исполнительные пакеты, которые содержат две команды ADD для параллельного исполнения. Первая команда ADD является условной по B0 на неравенство 0, вторая команда ADD является условной по B0 на равенство 0. Символ ! обозначает инверсию условия.

```
   [B0]   ADD   .L1    A1, A2, A3
|| [!B0]   ADD   .L2    B1, B2, B3
```

Приведенные команды являются взаимоисключающими. Это означает, что выполнится только одна из них.

При планировании вычислений следует учитывать не только ограничения на параллельное исполнение команд, вызываемое зависимостями по данным, но и ограничения, вызываемые доступностью ресурсов. Некоторые из этих ограничений рассмотрены ниже.

Ограничения по функциональным устройствам. Никакие две команды внутри любого исполнительного пакета не могут использовать тот же самый ресурс. Это создает ограничения по различным ресурсам. Две команды, использующие то же самое ФУ не могут быть запущены в одном и том же исполнительном пакете. Из-за этого нижеследующий исполнительный пакет неверен:

```
   ADD   .S1   A0, A1, A2 ;    \.S1 использовано для
|| SHR   .S1   A3, 15,  A4 ;    /обеих команд
```

А этот пакет верен:

```
   ADD   .L1   A0, A1, A2 ;     \ Испоьзуются два различных
|| SHR   .S1   A3, 15,  A4 ;    / функциональных устройства
```

Ограничения на кросс-пути. Для любого ФУ имеется возможность в процессе вычислений обращаться за данными через кросс-пути в регистры другого пути. Следовательно, кросс-пути являюся ресурсом, поэтому не допускается размещение в одном исполнительном пакете двух команд, использующих один и тот же кросс-путь.

Ограничения по загрузке и сохранению. Команды load/store могут использовать адресный указатель из одного регистрового файла, в то время как команды loading to или storing from – из другого регистрового файла. Две команды load/store, использующие регистры исходных операндов и результата (destination/sourse) из того же самого регистрового файла, не могут быть запущены в том же самом исполнительном пакете. Адресный регистр обязан быть на том пути, где используется устройство D. Следующий исполнительный пакет неверен:

```
   LDW   .D1    *A0, A1 ;   \ .D2 обязано использовать адресный
|| LDW   .D2    *A2, B2 ;   / регистр из регистровогофайла B
```

Этот пакет верен:

```
   LDW    .D1    *A0, A1 ;    \ Адресные регистры
|| LDW    .D2    *B0, B2 ;    / используются правильно
```

Две команды loads и/или stores, loading to и/или storing from, использующие тот же самый регистровый файл, не могут быть запущены в том же исполнительном пакете. Следующий исполнительный пакет неверен:

```
   LDW .D1 *A4,  A5 ;  \ Loading to и storing from
|| STW  .D2   A6,*B4 ;  / из того же регистрового файла
```

Эти пакеты верны:

```
   LDW .D1 *A4,  B5 ;  \ используются различные
|| STW  .D2   A6,*B4 ;  / регистровые файлы

   LDW .D1 *A0,  B2 ; \ используются различные
|| LDW .D2 *B0,  A1 ; / регистровые файлы
```

Ограничения по чтению регистров. В одном и том же такте из одного и того же регистра нельзя произвести более 4-ех чтений. Регистры условий не входят в их число. Следующие кодовые последовательности неверны:

```
   MPY   .M1    A1, A1, A4 ;    пять чтений регистра  A1
|| ADD   .L1     A1,  A1, A5
|| SUB    .D1     A1, A2, A3

   MPY   .M1    A1, A1, A4 ;    пять чтений регистра  A1
|| ADD   .L1     A1,  A1, A5
|| SUB    .D2x   A1, B2, B3
```

А этот– верен:

```
          MPY    .M1   A1, A1, A4 ;   только четыре
|| [A1]   ADD    .L1    A0,  A1,  A5    чтения A1
||        SUB    .D1    A1,  A2,  A3
```

Ограничения по записи в регистры. Две команды не могут записывать в том же цикле в один и тот же регистр. Две команды с тем же адресом назначения могут быть спланированы в один такт, если они не записывают в регистр назначения в том же такте. Например, MPY, запущенная в такте *i*, и следующая за ней команда ADD в такте *i*+1 не могут записывать в тот же самый регистр, поскольку обе команды записывают результат в такте *i*+1. Значит, следующий код неверен:

```
MPY   .M1   A0,  A1,  A2
ADD   .L1    A4,  A5,  A2
```

Но этот код верен:

```
    MPY   .M1   A0, A1, A2
|| ADD   .L1    A4, A5, A2
```

Следующий код представляет различные варианты конфликтов при выполнении операций умножения-записи.

```
L1:           ADD   .L2    B5, B6, B7 ;   \ определено: конфликт
||            SUB    .S2    B8, B9, B7 ;   /
L2:           MPY   .M2   B0, B1, B2 ;   \ нельзя  определить
L3:           ADD   .L2    B3, B4, B2 ;   /
L4:  [!B0]    ADD   .L2    B5, B6, B7 ;   \ определено: нет конфликта
||    [B0]    SUB    .S2    B8 ,B9, B7 ;  /
L5: [!B1]    ADD   .L2    B5, B6, B7 ;   \ нельзя определить
||    [B0]    SUB    .S2    B8, B9, B7 ;   /
```

Например, команды ADD и SUB в исполнительном пакете L1 записывают в тот же самый регистр. Этот конфликт легко распознается.

Команда MPY в пакете L2 и ADD в пакете L3 могут одновременно обращаться в B2. Однако, если команда ветвления создает ситуацию, когда после L2 будет выполняться не пакет L3, а какой-либо другой, то конфликта не будет, поэтому потенциальный конфликт может быть и не определен ассемблером.

Команды в L4 не создают конфликт по записи, поскольку они взаимно исключают друг друга. По контрасту, поскольку команды в L5 могут или не могут быть взаимно исключающими, ассемблер не может определить конфликт. Если конвейер получает команды для выполнения умножения – записи в тот же регистр, результат будет неопределенным.

3.2.3. *Функционирование конвейера*

Особенности функционирования конвейера процессора таковы:

• Все команды процессора проходят через ступени выборки, декодирования и исполнения и требуют для своего выполнения одинакового числа фаз;

• Параллельные команды проходят одновременно через каждую ступень конвейера, а последовательные команды идут через ступени конвейера с фиксированным относительным фазовым сдвигом между командами;

• Для всех команд выборка имеет четыре, а декодирование – две фазы. Число фаз исполнения зависит от вида команды.

Ниже в таблице представлено описание функций всех фаз ступеней выборки, декодирования и исполнения.

Ступень	Фаза	Имя	Действия
Чтение команды	Генерация программного адреса	PG	Определяется адрес пакета
	Посылка программного адреса	PS	Адрес посылается в память
	Ожидание програмы	PK	Выполняется обращение в память
	Прием данных	PR	Пакет поступает в процессор
Декодирование программы	Диспетчирование	DP	Следующий исполнительный пакет в выбранном пакете выделяется и посылается на соответствующее исполнительное устройство
	Декодирование	DC	Команды декодируются в исполнительном устройсвах
Execute	Execute 1	E1	Для всех типов команд оцениваются условия и читаются операнды Для команд обращения к памяти генерируется адрес и его модификация записывается в регистровый файл. При ветвлении извлекается в фазе PG нужный пакет перехода
	Execute 2	E2	Для команд Load адрес посылается в память, для команд Store адрес и данные посылаются в память. Для команд умножения результат записывается в регистровый файл
	Execute 3	E3	Выполняется доступ в память
	Execute 4	E4	Для команд Load данные передаются в процессор
	Execute 5	E5	Для команд Load команд данные записываются в регистр

Когда конвейер заполнен, в каждом такте его работы выбирается новый пакет. Большинство команд процессора являются однотактными (имеется только фаза E1). Небольшое число команд требует более одной фазы.

Рассмотрим выполнение конвейерной операции с несколькими исполнительными пакетами в одном пакете выборки. Такой пакет представлен ниже.

FP	EP	1	2	3	4	5	6	7	8	9	10	11	12	13
n	k	pg	ps	pk	pr	dp	dc	e1	e2	e3	e4	e5		
n	k+1						dp	dc	e1	e2	e3	e4	e5	
n	k+2							dp	dc	e1	e2	e3	e4	e5
n+1	k+3		pg	ps	pk	pr	кон-		dp	dc	e1	e2	e3	e4
n+2	k+4			pg	ps	pk	вейер		pr	dp	dc	e1	e2	e3
n+3	k+5				pg	ps	стоит		pk	pr	dp	dc	e1	e2
n+4	k+6					pg			ps	pk	pr	dp	dc	e1
n+5	k+7								pg	ps	p	pr	dp	dc
n+6	k+8									pg	ps	p	pr	dp

Здесь извлечен пакет *n*, который содержит три исполнительных пакета. За ним следует 6 пакетов (от *n*+1 до *n*+6), каждый из которых содержит один исполнительный пакет. Пакет *n* проходит через фазу выборки за 4 такта. В такте 5 на фазе диспетчирования процессор сканирует *p*-биты и определяет, что имеются три исполнительных пакета (от *k* до *k*+2). Это приводит к остановке конвейера, которая позволяет фазе DP запустить на исполнение пакеты *k*+1 и *k*+2 в тактах 6 и 7. Как только пакет *k*+2 готов к доставке на фазу DC (такт 8), остановка конвейера прекращается. Выборка пакетов от *n*+1 до *n*+4 была приостановлена, так что процессор имел время запустить на фазе DC три исполнительных пакета. Пакет *n*+5 также был остановлен в тактах 6 и 7. Ему не разрешается перейти на фазу *PG*, пока приостановка не прекратится в такте 8. Далее операции продолжаются, как показано, пока не появится новый пакет с несколькими исполнительными пакетами или не произойдет прерывание.

При работе конвейера возможны остановки из-за памяти. Конфигурация памяти С6 характерна для ЦСП – имеются физически различные пространства для памяти программ и данных, но они разнесены по разным ступеням конвейера: выборка команд осуществляется на ступени fetch, а выборка данных – на ступени execute.

В случае, когда в однопортовую память производится несколько одновременных обращений, конвейер будет останавливаться, чтобы разрешить конфликты. Они имеют место, когда память не готова обслужить запрос процессора. Это происходит во время фазы PK для программного доступа и в течение фазы E3 при обращении за данными. Остановки памяти приводят к удлинению работы всех фаз за пределы одного такта. Таблица илюстрирует ситуацию:

FP	1	2	3	4	5	6	7	8-9	10	11	12-13-14	15	16
n	pg	ps	pk	pr	dp	dc	e1	Простой из-за программы	e2	e3	Простой из-за памяти	e4	e5
n+1		pg	ps	pk	pr	dp	dc		e1	e2		e3	e4
n+2			pg	ps	pk	pr	dp		dc	e1		e2	e3
n+3				pg	ps	pk	pr		dp	dc		e1	e2
n+4					pg	ps	pk		pr	dp		dc	e1
n+5						pg	ps		pk	pr		dp	dc
n+6							pg		ps	pk		pr	dp
n+7									pg	ps		pk	pr
n+8										pg		ps	pk
n+9												pg	ps
n+10													pg

3.2.4. *Обзор программного пакета разработчика*

На рис.3.7 представлен состав программного пакета разработчика и порядок разработки прикладных программ. Возможны два варианта разработки исходного кода. В первом случае пользователь создает программу на обычном последовательном языке С. Затем компилятор С распараллеливает эту программу, оптимизирует ее и представляет на параллельном ассемблере.

Другой вариант состоит в том, что программа пишется на линейном ассемблере микропроцессора С6. При этом не предпринимается никаких действий по распараллеливанию программы. Линейная программа передается на оптимизатор, который назначает регистры и производит оптимизацию циклов, чтобы преобразовать линейный код в высокопараллельный код на ассемблере, использующий все преимущества программного конвейера.

Для выполнения отладки поставляются: программный симулятор, работающий с точностью до одной команды и такта ее исполнения, и эмулятор.

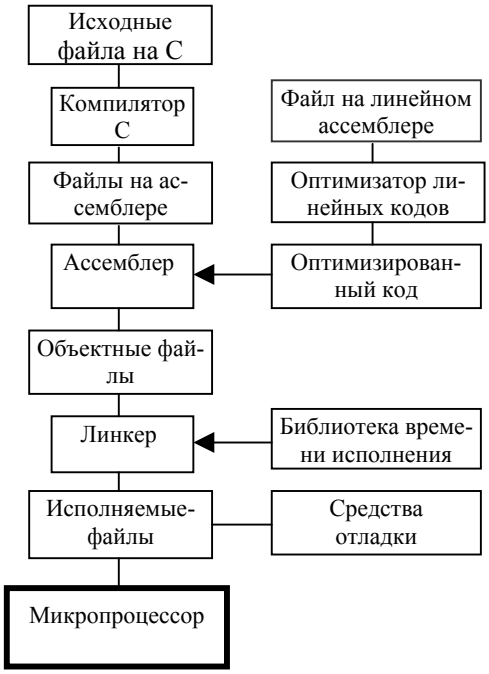

Рис.3.7. Порядок разработки прикладных программ

3.2.5. *Порядок создания кода*

Имеется три фазы создания оптимизированного кода для С6:

Фаза1 – написание кода на языке С. Код можно написать без всяких знаний о параллелизме процессора, а затем, используя программу Profiler, входящую в состав отладчика (Debugger) или симулятор, оценить время исполения прикладной программы и устранить неэффективные участки программы

Фаза 2 – совершенствование кода на С с помощью различных опций компилятора с последующей проверкой эффективности измененного кода на симуляторе. Если код еще неудовлетворителен, осуществляется переход к фазе 3

Фаза 3 - написание линейного кода на ассемблере для критических участков С кода. Затем используется оптимизатор ассемблерного уровня для улучшения этого кода. Однако внесение изменений в фазе 3 достаточно трудоемко, поэтому нужно постараться сделать все коррективы до фазы 3.

Рассмотрим весь порядок создания кода на примере программы demo1.c, которая в свою очередь использует три функции: mac1( ), vec_mpy1( ) и iir1( ).

Код головной программы demo1.c на языке С представлен ниже (пример 1):

```
main (int argc, char*argv [ ])
{
     const short coefs [150];
     short optr [150];
     short state [2];
     const short a [150];
     const short b [150];                                          (1)
     int c = 0;
     int dotp [1] = [0];
     int sum = 0;
     short y[150];
     short scalar = 3345;
     const short x [150];
     sum = mac1 (a, b, c, dotp);
     vec_mpu1 (y, x, scalar);
    iir1 (coefs, x, optr, state);
}
```

Код функции mac1.c на языке С (пример 2) выполняет умножения с накоплением:

```
int mac1 (const short*a, const short*b, int sqr, int*sum)
{
   int i;
   int dotp =  *sum;
       for (i = 0; i < 150; i++)
   {
       dotp + = b[i] * a[i]                                    (2)
       sqr   + = b[i] * b[i]
   }
  *sum = dotp;
  return sqr;
}
```

Код функции vec_mpu1.c на языке (пример 3) предназначен для векторного умножения:

```
void vec_mpu1 (short y [ ], const short x [ ] , short scalar)
{
    int i;                                                     (3)
    for (i = 0; i < 150; i++)
        y [i] + = ((scalar * x [i] ) >> 15 );
}
```

Наконец, код функции iir1.c (пример 4) представляет биквадратный фильтр:

```
void iir1 (const short * coefs, const short * input, short * optr, short * state)
{
    short x;  short t;
    int n;
    x = input [0];
    for (n = 0; n < 50; n++ )
    {
         t = x + ((coefs [2] * state [0] + coefs [3] * state [1]) >> 15);          (4)
         x = t + ((coefs [0] * state [0] + coefs [1] * state [1]) >> 15);
        state [1] = state [0];
        state [0] = t;
        coefs + = 4;        /* указатель на следующий coefs фильтра */
        state  + = 2;       /* указатель на следующее  state фильтра */
    }
    *optr ** = x;
}
```

Оценим с помощью программы Profiler число тактов, необходимое для выполнения программы demo1.c и всех ее функций. Результат представлен в таблице:

Тип	Имя	Число тактов
Функция C	iir1 ( )	270
Функция C	mac1 ( )	167
Функция C	main ( )	831
Функция C	vec mpu1 ( )	316

С помощью Profiler'а можно получить и более подробную информацию об особенностях выполнения программ. Так, столбец «Счетчик циклов» в следующей таблице показывает число тактов, необходимое для выполнения каждой функции. Заметим, что счетчик mac1( ) равен167, а для vec_mpy1( ) and iir1( ) гораздо больше – 316 и 270 соответственно. Чтобы понять, как производится подсчет числа тактов, приведем формулу для их вычисления:

Z = A*B + константа

где *z* – число тактов, *A* – число исполнительных пакетов в теле цикла (пакет включает до 8 команд, исполняемых параллельно), *B* – число итераций соответствующго цикла в программе на языке *C*, что и показано в таблице:

Функция	Исполнительные пакеты	Итерации цикла	Константы	Счетчик циклов
mac1 ( )	1	150	17	1x50 + 17 = 167
vec_mpu1 ( )	2	150	16	2x150 + 16 = 316
iir1 ( )	5	50	20	5x50 + 20 = 270

Фаза 1 создания высокопараллельного кода. Рассмотрев функции в demo1.c можно определить, нуждаются ли они в улучшении. Изучим первую функцию mac1( ). Пример 5 показывает ядро внутреннего цикла функции. Программа на параллельном ассемблере является стандартным выходом компилято

ра языка С. Ядро цикла есть область, обладающая максимальным потенциальным параллелизмом. Однако, визуальный анализ показывает, что все 8 команд исполняются параллельно и улучшить здесь ничего нельзя:

```
L3:             ;  PIPED LOOP KERNEL

                ADD     .L2     B4, B7, B7      ;
||              ADD     .L1     A5, A3, A3      ;
||               MPY    .M2X    A4, B5, B4      ; @@
||               MPY    .M1     A4, A4, A5      ; @@                    (5)
||  [ B0 ]      B       .S1      L3             ; @@@@@@
||  [ B0 ]      SUB     .S2      B0, 1, B0      ; @@@@@@@
||              LDH     .D1     *A0++, A4       ; @@@@@@@@
||              LDH     .D2     *B6++, B5       ; @@@@@@@@
```

Символ @ обозначает итерацию цикла, команда которой находится в программном конвейере. Эти символы автоматически создаются генератором кода. Первая итерация не имеет @, один такой символ есть во второй итерации, два – в третей итерации и так далее. Поскольку mac1( ) не нуждается в улучшении, ей не нужны другие фазы улучшения.

Взглянем на пример 6, который представляет ассемблерный выход самого внутреннего цикла vec_mpy1( ) function. Время ее исполнения 316 тактов. Этот код не такой параллельный, как в функции mac1( ). Ассемблерный выход vec_mpy1( ) имеет два исполнительных пакета, каждый из четырех команд. Этот цикл может быть улучшен:

```
L3:             ;  PIPED LOOP KERNEL

                ADD     .L2X     A3, B6, B5     ;
||  [ A1]       B       .S1      L3             ; @@
||              LDH     .D2      *+B4(6), B6    ; @@@
||              LDH     .D1      *A0++, A4      ; @@@@                  (6)

                STH     .D2      B5, *B4++      ;
||              SHR     .S1      A3, 15, A3     ; @
||              MPU     .M1      A5, A4, A3     ; @@
||  [ A1]       SUB     .L1      A1, 1, A1      ; @@@
```

Пример 7 показывает ассемблерный выход внутреннего цикла функции iir1( ), которая исполняется за 270 тактов. В ней некоторые пакеты имеют пять параллельных команд, другие – меньше. Возможно эту функцию можно улучшить.

```
L3:             ; PIPED LOOP KERNEL

                SHR     .S2     B4, 15, B4      ;
||              SHR     .S1     A3, 15, A5      ;
||              MPY     .M2X    B6, A5, B6      ; @
||              LDH     .D1     *+A6(16), A4    ; @@
||              LDH     .D2     *+B7(10), B6    ; @@

                ADD     .L1     A0, A5, A0      ;
||              MPY     .M1X    B6, A3, A3      ; @
||              MPY     .M2X    B5, A4, B5      ; @
||              LDH     .D1     *+A6(22), A3    ; @@
||              LDH     .D2     *+B7(8), B5     ; @@

                EXT     .S1     A0, 16, 16, A0  ;
||              STH     .D2     B5, *+B7(6 )    ; @
||              MPY     .M1X    B5, A3, A4      ; @
||              LDH     .D1     *+A6(20), A3    ; @@                    (7)

                ADD     .S1     8, A6, A6       ;
||              STH     .D2     A0, *B7++(4 )   ;
||              ADD     .L1X    A0, B4, A0      ; @
||  [ B0 ]      SUB     .L2     B0, 1, B0       ; @
||              ADD     .S2     B6, B5, B4      ; @

                EXT     .S1     A0, 16, 16, A0  ;
||  [ B0 ]      B       .S2     L3              ; @
||              ADD     .L1     A3, A4, A3      ; @
||              LDH     .D1     *+A6(18), A5    ; @@@
```

Фаза 2 создания высокопараллельного кода. Рассмотрим векторную функцию vec_mpy1( ), представленную в примерах (3) и (6).

Пример (3) использует короткие типы данных по 16 разрядов. Они переводятся в команды для полуслов, такие как LDH and STH (Load Halfword и Store Halfword). Цикл в примере (6) использует две команды STH, чтобы загрузить $x[i]$ and $y[i]$ и запоминает результат обратно в y[i]. Поскольку в одном такте

возможны только два обращения в память, то самое быстрое, что можно сделать, это вычислить только один результат за каждые два такта. Характеристики этого цикла ограничены числом устройств D.

Поскольку x является массивом, $x[i]$ and $x[i + 1]$ являются смежными в памяти. Это позволяет обращаться к памяти с помощью команд LDW and STW (Load Word и Store Word), чтобы загрузить два элемента сразу за один такт. Два элемента $x[i]$ and $x[i + 1]$ будут загружены в 32-разрядный регистр.

Возникает естественный вопрос, как выполнять операции умножения или сложения над двумя операндами, находящимися в одном регистре. Этот вопрос легко решается в векторной архитектуре типа MMX, которая используется, например, в МП Pentium или Itanium, но в C6 такая возможность отсутствует. Для умножения в C6 заменой этому являются две функции языка C, соответствующие командам ассемблера: *mpy intrinsics* и *mpyth intrinsics*. Первая из них умножает младшие 16 разрядов двух 32-разрядных регистров, а вторая делает то же самое для старших 16 разрядов. Аналогичные функции есть для сложения и других операций.

Аналогичное усовершенствование проводится и для функции iir1, представленной в примере (4).

В нижеследующей таблице представлены улучшенные результаты, полученные в результате изменения способа обращения к памяти в функциях vec_mpy1 ( ) и iir1 ( ):

Тип	Имя	Число тактов
Функция C	vec mpu2 ( )	172
Функция C	iir2 ( )	220
Функция C	mac1 ( )	167
Функция C	main ( )	637

Фаза 3 создания высокопараллельного кода. Рассмотрим возможности фазы 3 на примере дальнейшего усовершенствования функции iir2 ( ). Чтобы улучшить характеристики этой функции, нам нужно переписать ее в код линейного ассемблера,

который является входом для оптимизатора линейного ассемблера. Выход оптимизатора является входом параллельного ассемблера. Линейный ассемблер отличается от обычного ассемблера тем, что в нем не используются параллельные команды, распределение регистров, информация о распределении и задержках функциональных устройств. Это предоставляет больше свободы для действий оптимизатора, при этом сохраняется возможность внесения любых изменений вручную.

Код на линейном ассемблере может быть написан либо вручную, либо как результат работы одной из опций компилятора.

Оптимизатор линейного кода обладает расширенными возможностями. Он может выполнять большую часть трансформаций, описанных в главе 2. К ним, например, относятся: выборка из памяти коротких данных с помощью обращения по словам и двойным словам; использование программного конвейера; планирование по модулю многотактных циклов; оптимальная реализация условных переходов; развертка циклов; исключение избыточных загрузок; оптимальное разрешение конфликтов при доступе к многоблочной памяти; программный конвейер внешнего цикла и многое другое

В таблице ниже показано, что в результате работы оптимизатора линейного кода число тактов, необходимое для выполнения функции iir3 ( ), значительно сократилось по сравнению с предыдушими вариантами iir1 ( ) и iir2 ( ):

Функция	Исполнительные пакеты	Итерации цикла	Константы	Счетчик циклов
iir1 ( )	5	50	20	5x50 + 20 = 270
iir2 ( )	4	50	20	4x50 + 20 = 220
iir3( )	3	50	27	3x50 + 27 = 177

В нижеприведенной таблице даны конечные результаты времени выполнения программы demo3 ( ) с учетом усовершенствований всех ее функций:

Тип	Имя	Число тактов
Функция С	vec_mpy2 ( )	172
Функция С	iir3 ( )	177
Функция С	Mac1 ( )	167
Функция С	Main ( )	594

Конечно, эти результаты лучше, чем исходные, поскольку часть функций оптимизирована.

Контрольные вопросы к главе 3

1. Опишите структуру процесора Itanium.
2. Каковы особенности архитектуры процессора Itanium?
3. Как реализуется параллельная обработка в процессоре Itanium?
4. Что такое “связка “ в процессоре Itanium, каков ее формат?
5. Имеются ли в процессоре Itanium операции типа SIMD?
6. Как выполняется предикация в процессоре Itanium?
7. Что такое загрузка по предположению?
8. Какова организация и система команд процессора С6?
9. Какова структура пакета выборки?
10. Какова структура исполнительного пакета для С6?
11. Как выполняются ветвления в середине исполнительного пакета?
12. Какие ограничения влияют на формирование исполнительного пакета?
13. Приведите примеры влияния ограничений из-за недостатка ФУ.
14. Приведите примеры влияния ограничений при записи в регистры.
15. Опишите функции этапов конвейера С6.
16. Как функционирует конвейер С6?
17. Чем вызваны остановки конвейера?
18. Каков порядок создания параллельной программы?
19. Назовите основные способы увеличения параллелизма в рассмотренном примере вычисления программы demo1.c