Глава 18

Расширения традиционной архитектуры Intel

- **Два типа MMX-расширения**
- **Модель целочисленного MMX-расширения**
- **Система команд целочисленного MMX-расширения**
- **Модель XMM-расширения**
- **Система команд XMM-расширения**
- **Модельно-зависимые регистры**

С появлением процессоров пятого поколения (Pentium, Pentium MMX...) в программной модели процессора Intel следует различать два слоя архитектуры: базовый и модельно-зависимый.

Слой *базовой архитектуры* включает в себя элементы архитектуры, поддержку и неизменность которых производитель (Intel) гарантирует. Это набор и структура основных устройств процессора, системных регистров и регистров общего назначения, архитектура памяти и т. д.

Модельно-зависимый слой архитектуры включает в себя средства, поддержка которых зависит от конкретной модели процессора (как правило, снизу вверх). Для того чтобы программа, исполняемая на конкретном процессоре, могла получить сведения о его возможностях, в систему команд процессоров, поддерживающих IA-32 (Intel, AMD и т. д.), включена команда `CPUID`. Данная команда позволяет программе в любой момент времени получить сведения о классе, модели и архитектурных особенностях текущего процессора. Подробное описание данной

команды приведено в приложении А, а пример ее использования вы найдете в материале данной главы.

На этой главе рассмотрим следующие расширения базовой архитектуры процессора Intel:

- целочисленное MMX-расширение;
- XMM-расширение — потоковое MMX-расширение для работы с данными вещественного типа;
- поддержка модельно-зависимых особенностей процессоров IA-32.

MMX-технология процессоров Intel

В 1997 г. фирма Intel представила свой новый процессор — Pentium MMX. По сравнению с предыдущими моделями процессоров этой фирмы (i486 и Pentium) в его архитектуру были добавлены новые наборы регистров и команд. Pentium MMX стал своеобразной вехой в длинном ряду моделей процессоров семейства i80x86. С его появлением процессоры этого семейства естественно разбиваются на три группы, отражающие три этапа в истории развития процессоров:

- 16-разрядные процессоры i8086 и i80286 заложили основу популярности процессоров фирмы Intel. Их система команд содержала около 170 машинных инструкций, включая инструкции сопроцессора.
- Программная модель 32-разрядных процессоров i386, i486, Pentium, Pentium Pro практически одинакова и поддерживает порядка 220 команд.
- 32-разрядные процессоры Pentium MMX, Pentium II, Pentium III, Pentium IV поддерживают новые технологии обработки данных, которые обеспечиваются введением в их архитектуру дополнительных регистров и команд. Эти новые архитектурные элементы составляют так называемое *MMX-расширение*. Система команд процессоров Pentium MMX и Pentium II насчитывает около 300 команд, а процессора Pentium III(IV) — еще на 144 команды больше.

В этой главе мы обсудим MMX-расширение системы команд процессоров Pentium MMX, Pentium II и Pentium III(IV).

MMX-расширение процессора Pentium предназначено для поддержки приложений, ориентированных на работу с большими массивами данных целого и вещественного типов, над которыми выполняются одинаковые операции. С данными такого типа обычно работают мультимедийные, графические, коммуникационные программы. Именно по этой причине данное расширение архитектуры процессоров Intel и названо MMX (MultiMedia eXtensions — мультимедийные расширения).

С появлением процессора Pentium III следует различать MMX-расширение двух типов — целочисленное и с плавающей точкой. Каждое из этих MMX-расширений имеет свою программную модель и не зависит от другого. Чтобы различать эти типы расширений, введем следующие условные обозначения: целочисленное MMX-расширение будем называть *MMX-расширением*, а MMX-расширение с плавающей точкой — *XMM-расширением*.

MMX-расширение архитектуры процессора Pentium

Целочисленное MMX-расширение архитектуры процессора Pentium представляет собой программно-аппаратное решение, дополняющее архитектуру данного процессора новыми свойствами. Впервые это расширение было введено в процессоре Pentium MMX. В неизменном виде оно поддерживается в последующих моделях процессоров Pentium. В процессоре Pentium III целочисленное MMX-расширение было дополнено новыми командами. Знакомство с архитектурой целочисленного MMX-расширения удобно вести в рамках модели, основу которой составляют два компонента — *программный* и *аппаратный*.

Модель целочисленного MMX-расширения

Основа *программного* компонента — система команд MMX-расширения (57 команд) и четыре новых типа данных (рис. 18.1). MMX-команды являются естественным дополнением основной системы команд процессора. Главным принципом их работы является одновременная обработка нескольких единиц однотипных данных одной командой (Single Instruction Multiple Data, SIMD).

Рис. 18.1. Типы данных, поддерживаемые командами MMX-расширения

Из рисунка видно, что размер поля, занимаемого данными любого из этих типов, одинаков и составляет 64 бита. В них упаковываются и затем используются как отдельные объекты данные размером байт, слово и двойное слово (их мы будем называть *элементами* операндов). Именно с такими объектами работают MMX-команды. Такой подход приводит к существенному ускорению обработки данных, прежде всего, за счет экономии тактов процессора на передачу данных и выполнение самой команды. Кстати, здесь позитивно сказывается большая размерность шины данных процессора Pentium, которая равна 64 бита.

Основа *аппаратного* компонента — восемь новых регистров. Слово «новые» не совсем корректно. На самом деле в MMX-расширении используются регистры сопроцессора. Как известно, регистры сопроцессора стека имеют размерность 80 бит,

что касается регистров MMX-расширения, то их размерность — 64 бита. Поэтому, когда регистры сопроцессора играют роль MMX-регистров, то доступными являются лишь их младшие 64 бита. К тому же при работе стека сопроцессора в режиме MMX-расширения он рассматривается не как стек, а как обычный регистровый массив с произвольным доступом. Однако регистровый стек сопроцессора не может одновременно использоваться и по своему прямому назначению, и как MMX-расширение, поэтому забота о его разделении и корректной работе с ним ложится на программиста. Отображение MMX-регистров на регистры стека сопроцессора иллюстрирует рис. 18.2.

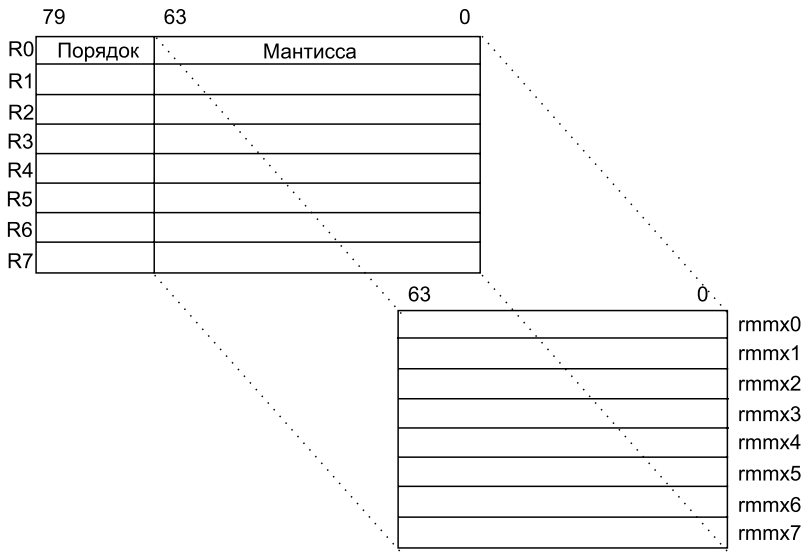

Рис. 18.2. Отображение MMX-регистров на регистры стека сопроцессора

При выполнении MMX-команд сопроцессор переводится в состояние, которое характеризуется следующими признаками:

- регистр тегов сопроцессора обнуляется;
- все регистры стека сопроцессора в MMX-режиме адресуются физически вне зависимости от значений поля `TOS` регистра состояния `SWR` (поле `TOS`, кстати, тоже обнуляется);
- MMX-регистру `RMMX0` соответствует физический регистр сопроцессора `R0`, MMX-регистру `RMMX1` — `R1` и т. д., при этом логическая нумерация регистров сопроцессора не имеет никакого значения;
- содержимое других регистров сопроцессора не изменяется (за исключением случая применения команды `EMMS`);
- при записи в MMX-регистр данных в младшие 64 бита заносятся сами записываемые данные, а в биты 64–79 — единицы, чтобы при попытке случайного или преднамеренного использования командой сопроцессора MMX-данных не возникло какого-либо исключения сопроцессора;
- при чтении данных из MMX-регистров их содержимое не изменяется.

Особенности команд MMX-расширения

Важное отличие MMX-команд от обычных команд процессора — в том, как они реагируют на ситуации переполнения и заема. В разделе «Арифметические операции над целыми двоичными числами» главы 8 нами разбирались ситуации, когда результат арифметической операции выходил за разрядную сетку исходных операндов. В этом случае производится усечение старших битов результата и возвращаются только те биты, которые умещаются в пределах исходного операнда. Этот принцип формирования результата называется *арифметикой с циклическим переносом* (wraparound arithmetic). Некоторые MMX-команды в подобной ситуации действуют иначе. В случае выхода значения результата за пределы операнда в нем фиксируется максимальное или минимальное значение. Такой принцип формирования результата называется *арифметикой с насыщением* (Saturation arithmetic). MMX-команды выполняют арифметические операции с использованием обоих принципов. При этом среди них есть команды, учитывающие знаки (значения старших битов) элементов операндов. Принцип формирования результатов арифметических операций с циклическим переносом вы можете вспомнить с помощью материала главы 8, здесь же рассмотрим на примерах, как формируются результаты в MMX-командах сложения и вычитания, работающих по принципу насыщения.

Сложение чисел (беззнаковое насыщение):

254 = 11111110
+
5 = 00000101
=
259 <> 11111111.

Результат MMX-сложения с беззнаковым насыщением равен 255. При сложении командами процессора `ADD` и `ADC`, использующими принцип циклического переноса, результат равен 00000011 = 3, а флаг `CF` устанавливается в 1. Это свидетельствует о факте переполнения.

Сложение чисел (знаковое насыщение):

+126 = 11111110	или	–126 = 10000010
+		+
+5 = 00000101		–5 = 11111011
=		=
131 <> 01111111		–131<> 10000000.

Результат MMX-сложения со знаковым насыщением двух положительных чисел равен 127. При сложении командами процессора `ADD` и `ADC`, использующими принцип циклического переноса, результат равен 00000011 = 3, а флаг `CF` установлен в 1. Это свидетельствует о факте переполнения.

Вычитание чисел (беззнаковое насыщение):

05 = 00000101
–
10 = 00001010
=
–5 <> 00000000.

Результат MMX-вычитания с беззнаковым насыщением двух чисел равен 00h. При вычитании командами процессора SUB и SBB, использующими принцип циклического переноса, результат равен 11111011 = –5 в дополнительном коде, а флаг CF устанавливается в 1. Это свидетельствует о факте воображаемого заема единицы из старшего разряда.

Вычитание чисел (знаковое насыщение):

+05 = 00000101	или	–5	= 11111011
–		–	
–126 = 100000010		+126	= 01111110
=		=	
–131<> 01111111		–131 <> 10000000.	

Результат MMX-вычитания со знаковым насыщением двух чисел равен 80h. Это минимально возможное отрицательное число размером в байт. При вычитании командами процессора SUB и SBB, использующими принцип циклического переноса, результат равен 11111011 = –5 в дополнительном коде, а флаг CF устанавливается в 1. Это свидетельствует о факте воображаемого заема единицы из старшего разряда.

Подобные рассуждения относятся и к некоторым другим MMX-командам, не являющимся арифметическими. В табл. 18.1 представлены граничные значения насыщения для всех четырех типов данных MMX-расширения.

Таблица 18.1. Граничные значения насыщения MMX-данных

Тип MMX-данных	Диапазон граничных значений (с насыщением)
Байт без знака	0...255 (00h...0ffh)
Слово без знака	0...65 535 (00h...0ffffh)
Двойное слово без знака	0...4 294 967295 (00000000...0ffffffffh)
Учетверенное слово без знака	(0000000000000000...0ffffffffffffffffh)
Байт со знаком	–128...127 (80h...7fh)
Слово со знаком	–32 768...327 67 (8000h...7fffh)
Двойное слово со знаком	–2 147 483 648...2 147 483 647 (80000000...7ffffffffh)
Учетверенное слово со знаком	(8000000000000000...7ffffffffffffffffh)

Отметим некоторые особенности практического использования команд MMX-расширения.

Не все трансляторы языка ассемблера поддерживают MMX-команды. Это относится к трансляторам TASM версии 5.0 и MASM версии 6.11. В качестве транслятора, поддерживающего MMX-команды, можно рекомендовать транслятор NASM от Netwide, информацию о котором можно найти в Интернете по адресу http://www.cryogen.com/nasm/. А что же делать тем, кто располагает транслятором TASM или MASM и не имеет возможности воспользоваться услугами NASM? Выход здесь один — писать программу в машинных кодах, и дело это не такое уж безнадежное. Фирма Intel предвидела возникновение подобной проблемы и предложила вариант включаемого файла iammx.inc для транслятора ассемблера от

Microsoft. Он содержит варианты макроопределений для всех мнемоник MMX-команд. Включив этот файл директивой include в начало вашей программы, вы можете использовать все MMX-команды в определенном формате. С транслятором TASM этот файл использовать нельзя. Для этого его нужно немного доработать. Ситуация усложняется тем, что в пакет ассемблера TASM входят два транслятора: 16-разрядный (tasm.exe) и 32-разрядный (tasm32.exe). Разрабатываемые с их помощью программы имеют свои особенности. Этих особенностей довольно много, поэтому в этой главе будет изложена методика, благодаря которой вы сможете разрабатывать программы с MMX-командами для обоих типов трансляторов. Если же в вашем распоряжении есть другие средства для разработки MMX-программ, вы можете использовать материал данной главы только в части, касающейся описания форматов MMX-команд и примеров их применения.

Перед началом работы доведем текст включаемого файла iammx.inc до рабочего состояния, пригодного для транслятора TASM 5.0. Для этого нам придется делать два варианта этого файла — 16- и 32-разрядный. Назовем их, соответственно, mmx16.inc и mmx32.inc. Они будут использоваться при разработке программ, обрабатываемых 16- и 32-разрядными трансляторами ассемблера (tasm.exe и tasm32.exe). Эти варианты включаемых файлов предназначены для работы в режиме MASM транслятора TASM 5.0. Объем исходных текстов включаемых файлов mmx16.inc и mmx32.inc довоьно велик, поэтому они находятся среди файлов, прилагаемых к книге, в каталоге данной главы[1]. Различий в этих файлах два:

- В mmx16.inc для моделирования MMX-команд используются 16-разрядные регистры общего назначения, а в mmx32.inc — 32-разрядные регистры.
- В mmx.inc для выяснения типа операнда используется оператор .TYPE. У автора при попытке указать эту директиву совместно с tasm32.exe возникала ошибка. Эту ошибку удалось преодолеть только при применении аналогичного оператора для режима Ideal — SYMTYPE.

Можно попытаться объединить эти два файла и препроцессорными средствами ассемблера распознавать текущую вычислительную ситуацию, но большого смысла в этом нет, так как 16- и 32-разрядные коды не пересекаются. Проще разработать два файла для каждой из технологий разработки и так избежать потенциальных ошибок. Поскольку содержательная часть этих файлов практически одинакова, есть смысл рассматривать только один из них, отмечая, при необходимости, имеющиеся отличия.

Рассмотрим структуру файла mmx16.inc. Этот файл содержит макроопределения, каждое из которых моделирует определенную MMX-команду. В качестве основы для моделирования выступает команда основного процессора, которая должна удовлетворять определенным требованиям. Каковы они? В поисках ответа рассмотрим машинные коды MMX-команд. Видно, что общими у них являются два момента:

- Поле кода операции MMX-команд состоит из двух байтов, первый из которых равен 0fh.

[1] Все прилагаемые к книге файлы можно найти по адресу http://www.piter.com/download. — *Примеч. ред.*

- Все MMX-команды, за исключением команды emms, используют форматы адресации с байтами modR/M и sib и, соответственно, допускают сочетание операндов как обычные двухоперандные команды целочисленного устройства — регистр — регистр или память — регистр.

Для моделирования MMX-команд нужно подобрать такую команду основного процессора, которая удовлетворяет этим двум условиям. Проанализировав с позиций этих требований машинные коды команд основного процессора, выберем для моделирования команды XADD и BTR. В процессе моделирования на место второго байта кода операции этих команд помещается байт со значением кода операции MMX-команды. Когда процессор «видит», что очередная команда является MMX-командой, то он начинает трактовать коды регистров в машинной команде как коды MMX-регистров и ссылки на память, размерностью соответствующей данной команде. В машинном формате команды нет символических названий регистров, которыми мы пользуемся при написании исходного текста программы, например, AX или BX. В этом формате они определенным образом кодируются. Например, регистр AX кодируется в поле reg машинной команды как 000 (см. главу 6). Если заменить код операции команды, в которой одним из операндов является регистр AX, кодом операции некоторой MMX-команды, то это же значение в поле reg процессор будет трактовать как регистр RMMX0. Таким образом, в MMX-командах коды регистров воспринимаются соответственно коду операции. В табл. 18.2 приведены коды регистров общего назначения и соответствующих им MMX-регистров. В правом столбце этой таблицы содержится условное обозначение MMX-регистров, принятое в файле mmx16.inc. Это же соответствие закреплено рядом следующих определений в этом файле (листинг 18.1).

Таблица 18.2. Кодировка MMX-регистров в машинном коде команды

Код в поле reg	Регистр целочисленного устройства	MMX-регистр
000	AX/EAX	RMMX0
001	CX/ECX	RMMX1
010	DX/EDX	RMMX2
011	BX/EBX	RMMX3
100	SP/ESP	RMMX4
101	BP/EBP	RMMX5
110	SI/ESI	RMMX6
111	DI/EDI	RMMX7

Листинг 18.1. Фрагмент включаемого файла mmx16.inc

```
rmmx0   equ <ax>
rmmx1   equ <cx>
rmmx2   equ <dx>
rmmx3   equ <bx>
rmmx4   equ <sp>
rmmx5   equ <bp>
rmmx6   equ <si>
rmmx7   equ <di>
```

В файле mmx32.inc эти же определения выглядят так, как показано в листинге 18.2.

Листинг 18.2. Фрагмент включаемого файла mmx32.inc

```
rmmx0   equ <eax>
rmmx1   equ <ecx>
rmmx2   equ <edx>
rmmx3   equ <ebx>
rmmx4   equ <esp>
rmmx5   equ <ebp>
rmmx6   equ <esi>
rmmx7   equ <edi>
```

Теперь в исходном тексте программы можно использовать символические имена MMX-регистров в качестве аргументов макрокоманд, моделирующих MMX-команды.

Рассмотрим, как в файле mmx16.inc описано макроопределение для моделирования MMX-команды PACKSSWB, предназначенной для упаковки с насыщением слов в байты (листинг 18.3).

Листинг 18.3. Фрагмент включаемого файла mmx16.inc

```
<1>  packsswb   macro   dest:req,src:req
<2>     local   pre,post
<3>  pre:
<4>     xadd    src,dest
<5>  post:
<6>     org     pre+1
<7>     db      _opc_packsswb
<8>     org     post
<9>     endm
```

Понимание структуры приведенного макроопределения не должно вызвать у читателя трудностей. Центральное место в макроопределении занимают команда целочисленного устройства (в данном случае XADD) и директива ORG. Директива ORG предназначена для изменения значения счетчика адреса. С этой директивой мы уже встречались в главе 10. В строке 6 директива ORG устанавливает значение счетчика адреса равным адресу pre + 1. Адрес метки pre является адресом первого байта машинного кода команды XADD. Соответственно, если этот адрес увеличить на 1, то получим адрес второго байта кода операции. Таким образом, значением текущего счетчика адреса в строке 7 будет адрес pre + 1. Директива DB в строке 7 размещает по этому адресу байтовое значение opc_packsswb, которое соответствует второму байту кода операции MMX-команды PACKSSWB (см. листинг 18.1). Директива ORG в строке 8 устанавливает значение счетчика адреса равным значению адреса следующей после XADD команды. Для дотошных читателей отметим еще один характерный момент. Для его полного понимания необходимо хорошо представлять себе формат машинной команды и назначение его полей. Довольно полная информация об этом приведена в материалах главы 3 и приложения А. Обратите внимание на порядок следования операндов в заголовке макрокоманды, который построен по обычной схеме: коп приемник, источник. В команде XADD порядок обратный, как требует синтаксис команды. Это хорошо поясняет назначение бита d во втором байте кода операции, который характеризует направление передачи дан-

ных в процессор (в регистр) или в память из процессора (регистра). Вы можете провести эксперимент. Проанализируйте машинные коды команды MOV:

```
.data
p   dw  0
.code
...
mov p,bx    ;машинный код: 89 1e 00 00, d=1, w=1
mov bx,p    ;машинный код: 8b 1e 00 00, d=0, w=1
```

Из приведенного фрагмента видно, что изменение типов источника и приемника на обратный влияет только на изменение поля кода операции, а именно на его второй бит. Это бит d (Directory — направление), который определяет направление передачи. Если приемник — регистр, то бит d = 1, а это означает передачу из памяти в процессор (регистр). И наоборот, если приемник — адрес памяти, то бит d = 0, а это означает передачу из процессора (регистра) в память. Таким образом, значения полей в остальных байтах машинного кода операции лишь определяют местоположение операндов и не зависят от реального направления передачи. Например, значение второго байта кода операции opc_packsswb равно 63h (0110 0011b). Он имеет значение бита d = 1, то есть данные передаются из регистра в память. Это нам и позволило в команде XADD изменить порядок следования операндов (иначе транслятор ассемблера эту команду не пропустит — подумайте, почему?).

В связи с обсуждаемым моментом интересно посмотреть макроопределение другой MMX-команды — MOVD (листинг 18.4).

Листинг 18.4. Фрагмент включаемого файла mmx16.inc

```
<1> movd    macro   dest:req, src:req
<2>     local   pre, post
<3>  if (.type(dest)) and 10h
<4>  pre:
<5>     xadd    src, dest
<6>  post:
<7>     org     pre+1
<8>     db      _opc_movd_ld
<9>     org     post
<10>    else
<11>    pre:
<12>        xadd dest, src
<13>    post:
<14>        org pre+1
<15>        db  _opc_movd_st
<16>        org post
<17>    endif
<18>        endm
```

Команда MOVD позволяет производить передачу в обоих направлениях: память-процессор и процессор — память. Для того чтобы правильно смоделировать машинное представление MMX-команды, необходимо определить, каким объектом является операнд dest (приемник): ячейкой памяти или регистром. После этого будет ясен тип второго операнда, так как реализованы могут быть лишь две схемы расположения операндов: регистр — регистр и память — регистр. Для выяснения типа операнда ассемблер предоставляет оператор .type, который имеет следующий синтаксис:

```
.type выражение
```

Оператор .type в зависимости от типа операнда выражение возвращает байтовые значения, которые приведены в табл. 18.3. В строке 3 листинга 18.4 определяется тип операнда dest, и если это регистр, то формируется один код операции (строки 4–10), если нет, то другой (строки 11–17). Кстати, MOVD — это единственная MMX-команда, которая выпадает из общей схемы построения MMX-команд, так как только она позволяет производить обмен с 32-разрядным регистром общего назначения.

Таблица 18.3. Операнды и возвращаемые значения оператора .type

Значения битов	Тип объекта
0000 0000b	Указатель адреса памяти в сегменте кода
0000 0010b	Указатель адреса памяти в сегменте данных
0000 0100b	Константа
0000 1000b	Выражение использует режим прямой адресации
0001 0000b	Регистр
0010 0000b	Идентификатор
1000 0000b	Внешний идентификатор
???? ?00?b	Выражение использует режим косвенной адресации

Адреса операндов в памяти формируются таким же образом, как и для целочисленных команд, а их трактовка зависит от конкретной MMX-команды. Более детально с моделированием остальных MMX-команд вы можете познакомиться, изучив тексты включаемых файлов mmx16.inc и mmx32.inc, которые находятся среди файлов, прилагаемых к книге.

Далее мы рассмотрим MMX-команды, сопровождая описание характерными примерами, демонстрирующими механизм работы команд. В конце этого раздела рассмотрим типовой пример использования MMX-команд.

Система команд

MMX-команды по функциональному признаку делятся на группы (рис. 18.3).

Рассмотрим выделенные классы команд, укажем особенности их применения на примерах.

Команды передачи данных

MMX-команды пересылки подобно их целочисленным аналогам являются наиболее часто используемыми. Эти команды осуществляют доставку информации в MMX-регистры и из регистров. MMX-команды пересылки работают с 32- и 64-разрядными операндами. Далее перечислены команды, входящие в данную группу.

- MOVD приемник, источник — пересылка 32 битов из источника в приемник. Один из операндов, источник или приемник, но не одновременно, должен быть MMX-регистром. Другой операнд должен быть 32-разрядным регистром или 32-разрядной ячейкой памяти.

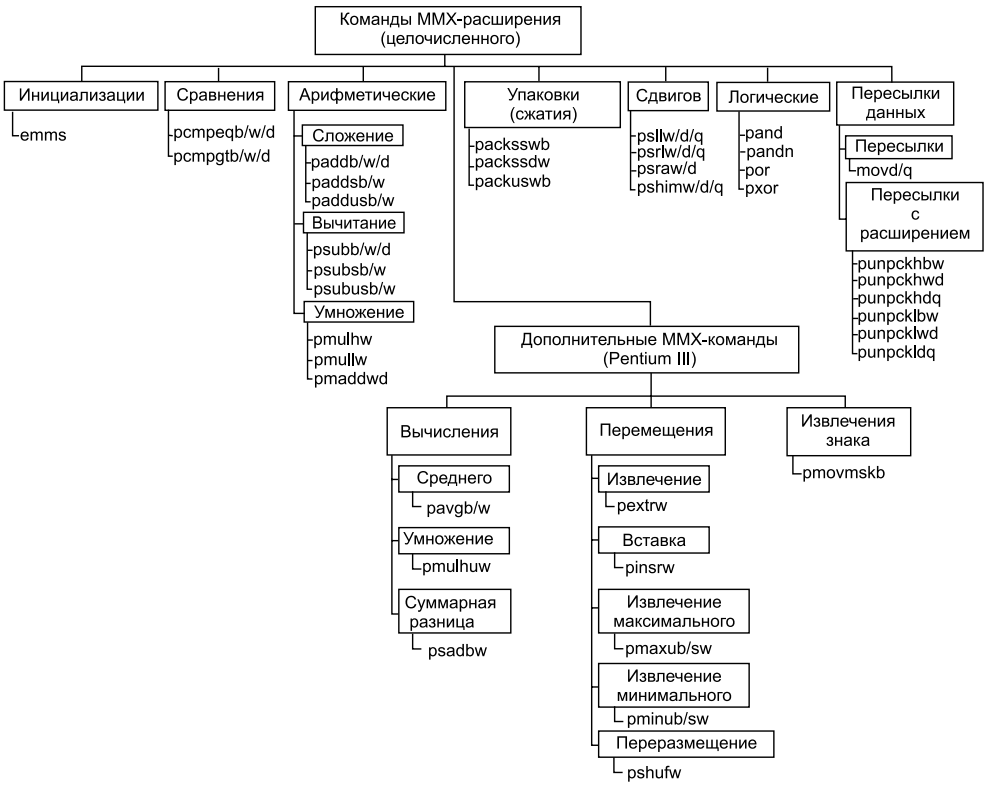

Рис. 18.3. Классификация целочисленных ММХ-команд

- `MOVQ приемник, источник` — пересылка 64 битов из источника в приемник. В отличие от команды `MOVD`, оба операнда команды `MOVQ` могут быть ММХ-регистрами. Если же операнды смешанные, то один из операндов, `источник` или `приемник`, должен быть ММХ-регистром, а другой — адресом 64-разрядной ячейки памяти.

Команда `MOVD` работает только с младшей половиной ММХ-регистра. Для доступа к старшей половине ММХ-регистра необходим либо сдвиг, либо команда `MOVQ`. Команда `MOVD` является единственной ММХ-командой, допускающей использование в качестве операндов 32-разрядных регистров общего назначения. Это же обстоятельство является причиной того, что при применении в качестве приемника регистра общего назначения макрокоманда `MOVD` будет работать неправильно. Для того чтобы понять, в чем здесь дело, посмотрим еще раз на макроопределение, моделирующее эту ММХ-команду:

```
movd    macro   dest:req, src:req
    local   pre, post
if (.type(dest)) and 10h
pre:
    xadd    src, dest
post:
    org     pre+1
    db      _opc_movd_ld
```

```
    org     post
else
pre:
    xadd    dest, src
post:
    org     pre+1
    db      _opc_movd_st
    org     post
endif
    endm
```

Допустимые сочетания операндов для команды MOVD следующие:

```
movd    mem32, rmmx
movd    rmmx, mem32
movd    rmmx, r32
movd    r32, rmmx
```

Приведенное макроопределение хорошо работает при первых трех сочетаниях операндов, а в последнем случае оно перестает понимать, что от него хотят, и дает сбой. Здесь две причины. Во-первых, оператор .type (SYMTYPE в режиме Ideal) одинаково реагирует на сочетания MOVD rmmx,r32 и movd r32,rmmx, так как в результате макроподстановки операнд rmmx заменяется одним из регистров r32. Поэтому в обоих случаях генерируется команда XADD rmmx,r32 со вторым байтом кода операции равным 6eh. MMX-команда с этим кодом операции предполагает пересылку 32 битов из памяти или регистра общего назначения, но не наоборот. Для пересылки из регистра общего назначения (32-разрядной ячейки памяти) в MMX-регистр требуется машинная команда со вторым байтом кода операции, равным 7eh. Вторая причина является следствием первой. Так как MMX-регистры и регистры общего назначения кодируются в машинной команде одинаково, то конкретная машинная команда по заложенной в ней схеме различает, какой из операндов является MMX-регистром, а какой регистром общего назначения, и для указанного ранее сочетания операндов делает это неправильно. Как выйти из положения? Можно, конечно, выполнить дополнительный анализ передаваемых в макрокоманду аргументов, но мы, учитывая единичность этого случая, пошли по самому легкому пути — включили в файл mmx.inc дополнительное макроопределение MOVDR32. Это макроопределение специально предназначено для случая пересылки данных из MMX-регистра в 32-разрядный регистр общего назначения.

Следующий важный вопрос связан с тем, как описывать данные для работы с MMX-командами. При этом также приходится учитывать то обстоятельство, что при использовании пакета TASM (как, впрочем, и MASM) MMX-команды приходится моделировать. Главное здесь — понять последовательность этого моделирования: на этапе трансляции моделируются операнды целочисленной команды (в нашем случае — XADD) и формируется поле кода операции в машинной команде (ее второй байт). Далее на этапе исполнения программы выполняется должная интерпретация операндов команды. Если посмотреть на описание команды XADD, то легко увидеть, что она может работать максимум с 32-разрядными операндами. А как же быть, если требуется моделировать MMX-команду, манипулирующую 64-разрядными операндами? В этом случае приходится в очередной раз заниматься обманом. Например, следующий фрагмент программы будет ошибочным:

```
include mmx.inc
.data
mem64   dq  1111222233334444h
```

```
...
.code
...
    movqrmmx0, mem64
...
```

Ошибка возникнет из-за того, что на этапе трансляции будет смоделирована команда XADD ax, mem64, которую транслятор не пропустит из-за несовпадения типов операндов. Суть «обмана» состоит в следующем описании данных:

```
include mmx.inc
.data
mem64   dw  4444h
df  111122223333h
...
.code
...
    movqrmmx0, mem64
...
```

Помните, что при описании операнда в памяти продолжает действовать принцип размещения данных в памяти «младший байт по младшему адресу». При работе с 32-разрядным ассемблером tasm32.exe описание данных должно быть выполнено так:

```
include mmx32.inc
.data
mem64   dd  33334444h
dd  11112222h
...
.code
...
    movqrmmx0, mem64
...
```

В листинге 18.5 приведены примеры команд пересылки данных. Текст программы листинга используйте в качестве основы для разработки других программ главы. Далее для экономии места мы будем приводить только фрагменты соответствующих программ.

Листинг 18.5. Использование MMX-команд пересылки данных

```
<1> ;---------mmx16.asm---------------------------------
<2>  .586p
<3>  model  use16 small ; use16 обязательно
<4>  %NOINCL;запретить вывод текста включаемых файлов
<5>  include mmx16.inc
<6>  .stack 100h
<7>  .data  ;сегмент данных
<8>  mem    dw  4444h
<9>         df  111122223333h
<10> mem1   dw  3h
<11>        df  0002cccc0004h
<12> mem2   dw  0h
<13>        df  0001dddd0f03h
<14> mem3   dw  7fffh
<15>        df  0001dddd0f03h
<16> .code
<17> main   proc;начало процедуры main
<18>        mov ax, @data
<19>        mov ds, ax
<20>
<21>        movd    rmmx0, mem  ;rmmx0=0000 0000 3333 4444
```

продолжение ⇨

Листинг 18.5 *(продолжение)*

```
<22>        movd r32 ax, rmmx0  ; rmmx0=0000 0000 3333 4444, eax=3333 4444
<23>        movq    rmmx0, mem1 ;rmmx0=0002 cccc 0004 0003
<24>        movd    mem2, rmmx0 ;mem2=0300 0400 cccc 0200, rmmx0=0002
                                 cccc 0004 0003
<25>        pxor    rmmx0, rmmx0;rmmx0=0000 0000 0000 0000
<26>        movd    mem3, rmmx0 ;mem3=0000 0000 0000 0000
<27>        emms
<28>        mov ax, 4c00h   ;пересылка 4c00h в регистр ax
<29>        int 21h         ;вызов прерывания с номером 21h
<30>    main endp           ;конец процедуры main
<31>    end main            ;конец программы с точкой входа main
```

Обратите внимание на строку 22, в которой команда MOVD (ранее мы уже обсуждали эту команду и соответствующую ей макрокоманду MOVDR32) пересылает 32 бита в регистр общего назначения, но при этом указывается 16-разрядный регистр AX. Указать впрямую EAX нельзя, так как для 16-разрядного ассемблера вместо RMMX0 в ходе моделирования подставляется AX (см. листинг 18.1). На этапе выполнения все расставляется на свои места, так как требуемую пересылку (из RMMX0 в EAX) производит микропрограмма, реализующая MMX-команду MOVD по полю кода операции и коду регистра, который, как видно из табл. 18.3, одинаков для регистров AX и EAX.

Здесь уместно уделить внимание еще одной проблеме, которая неизбежно встает на определенном этапе разработки программы, — какие отладочные средства можно использовать для просмотра результатов работы MMX-команд? Можно попробовать подыскать отладчик, поддерживающий работу с MMX-расширением, но это делать совсем необязательно. Понимание того, что MMX-расширение архитектурно создано на базе сопроцессора, дает вам возможность использовать практически любой отладчик для работы с MMX-приложениями, в том числе Turbo Debuger. При этом, конечно, необходимо будет соблюдать определенные ограничения, источник возникновения которых — в том, что Turbo Debuger ничего не знает о MMX-командах. Это обстоятельство может приводить к его неадекватной работе. Но если следовать некоторым рекомендациям, то можно «пройти над подводными камнями», не задев их.

В пакете TASM версии 5.0 есть два варианта отладчика Turbo Debuger:

- td.exe — отладчик для разработки 16-разрядных программ;
- td32.exe — отладчик для разработки 32-разрядных программ.

В основном, все сказанное далее касается любого из этих отладчиков, так как проблема у них общая — непонимание мнемоники MMX-команд.

Чтобы создать исполняемый файл, пригодный для отладки, задайте в качестве параметра транслятора (tasm.exe или tasm32.exe) ключ /zi, а в качестве компоновщика (tlink.exe или tlink32.exe) — ключ /v. Загрузите исполняемый файл в отладчик (исходный текст появится в окне Module).

Теперь «в уме» разделите программу на фрагменты двух типов: содержащие MMX-команды и не содержащие их. С фрагментами, не содержащими MMX-команд, вы можете работать в обычном режиме. Для отладки программы на фрагментах, содержащих MMX-команды, перейдите в окно CPU (View ▸ CPU), а также откройте окно Numeric processor (View ▸ Numeric processor). После этого окно отладчика должно выглядеть так, как показано на рис. 18.4.

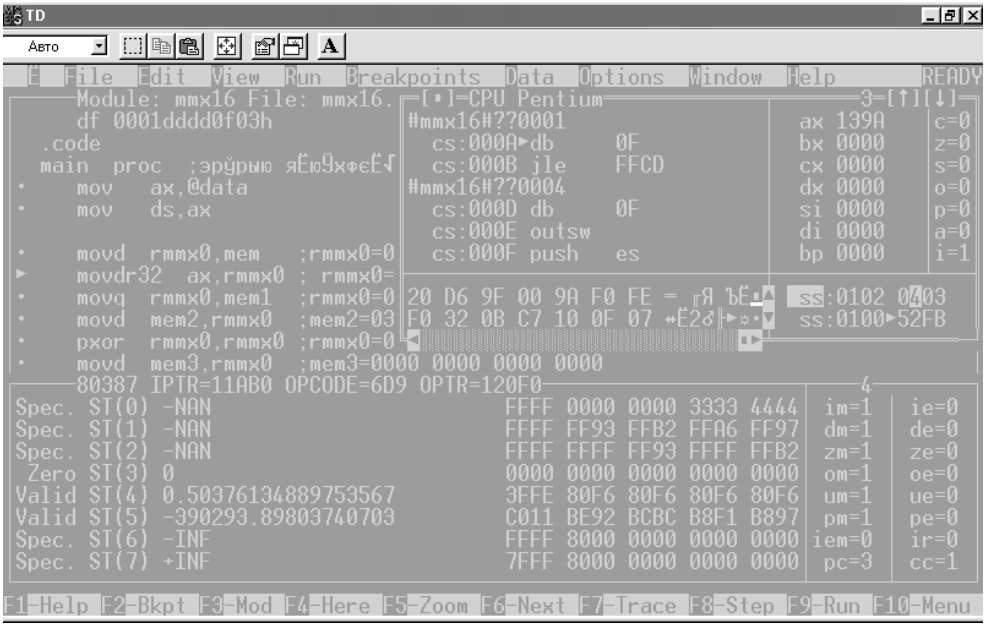

Рис. 18.4. Вид отладчика при работе с MMX-командами

Сам процесс отладки традиционен, и вы можете управлять им, используя привычные средства Turbo Debuger.

Арифметические команды

В группу арифметических входят команды для реализации трех основных арифметических операций: сложения, вычитания, умножения. При этом, как мы уже отметили, для операций сложения и вычитания предусмотрены команды, учитывающие знак операндов (значение старшего бита).

Сложение

Команды сложения делятся на две подгруппы, исходя из того, как формируется результат при возникновении переполнения — по принципу насыщения или по принципу циклического переноса.

- PADDB | PADDW | PADDD приемник, источник — сложение беззнаковых упакованных байтов, слов, двойных слов. Результат помещается в приемник, который является одним из MMX-регистров. Источник — либо MMX-регистр, либо 64-разрядная ячейка памяти. При переполнении результат формируется по принципу циклического переноса (см. ранее подраздел «Особенности команд MMX-расширения»). Перенос теряется и нигде не учитывается:

```
 .data   ;сегмент данных
 mem dw  4444h
     df  111122223333h
 mem1dw  0fffeh
     df  0fffcfffdffffh
 .code
 ...
```

```
    movq    rmmx0, mem  ;rmmx0=11 11 22 22 33 33 44 44
                        ;mem1=ff fc ff fd ff ff ff fe
    paddb   rmmx0, mem1 ;rmmx0=10 0d 21 1f 32 32 43 42
...
```

- PADDSB | PADDSW приемник, источник — сложение упакованных байтов и слов со знаком. Результат помещается в приемник, который является одним из MMX-регистров. Источник — либо MMX-регистр, либо 64-разрядная ячейка памяти. При возникновении переполнения результат формируется по принципу знакового насыщения (см. ранее подраздел «Особенности команд MMX-расширения»).

```
.data   ;сегмент данных
mem dw  4444h
    df  111122223382h
mem1    dw  0157eh
    df  0717c3f7d7ffah
.code
...
    movq    rmmx0, mem  ;rmmx0=11 11 22 22 33 82 44 44
                        ;mem1 =71 7c 3f 7d 7f fa 15 7e
    paddsb  rmmx0, mem1 ;rmmx0=7f 7f 61 7f 7f 80 59 7f
...
```

Исходные данные этого примера подобраны так, чтобы было видно, как при переполнении формируется результат по принципу знакового насыщения. Сравните их с теми результатами, которые получаются в результате работы следующих команд.

- PADDUSB | PADDUSW приемник, источник — сложение беззнаковых упакованных байтов и слов. Результат помещается в приемник, который является одним из MMX-регистров. Источник — либо MMX-регистр, либо 64-разрядная ячейка памяти. При возникновении переполнения результат формируется по принципу беззнакового насыщения:

```
.data   ;сегмент данных
mem     dw  4444h
    df  111122223382h
mem1    dw  0157eh
    df  0717c3f7d7ffah
.code
...
    movq    rmmx0, mem  ;rmmx0=11 11 22 22 33 82 44 44
                        ;mem1 =71 7c 3f 7d 7f fa 15 7e
    paddusb rmmx0, mem1 ;rmmx0=82 8d 61 9f b2 ff 59 c2
...
```

Вычитание

Набор команд вычитания аналогичен командам сложения и содержит три группы. Эти команды вычитают беззнаковые элементы традиционным для целочисленного устройства способом — по принципу циклического переноса, знаковые элементы — по принципу знакового насыщения и беззнаковые элементы, по принципу беззнакового насыщения:

- PSUBB | PSUBW | PSUBD приемник, источник — вычитание беззнаковых упакованных байтов, слов, двойных слов. Результат помещается в приемник, который является одним из MMX-регистров. Источник — либо MMX-регистр, либо 64-разрядная ячейка памяти. При переполнении результат формируется по

принципу циклического переноса, то есть так, как это делается командами SUB и SBB процессора. Заем из старшего разряда, естественно, теряется и нигде не учитывается:

```
.data   ;сегмент данных
mem     dw  4444h
    df  111122223382h
mem1    dw  157eh
    df  1123f7d7ffah
.code
...
    movq    rmmx0, mem   ;rmmx0=1111 2222 3382 4444
                         ;mem1 =1123 f7d7 ffah 157e
    psubw   rmmx0, mem1  ;rmmx0=0fff e2a5 b388 2ec6
...
```

- PSUBSB | PSUBSW приемник, источник — вычитание упакованных байтов и слов со знаком. Результат помещается в приемник, который является одним из MMX-регистров. Источник — либо MMX-регистр, либо 64-разрядная ячейка памяти. При возникновении ситуации, когда результат вычитания получается меньше 80h (8000h), поле соответствующего байта (слова) формируется по принципу знакового насыщения (в нем остается значение 80h (8000h)). Если результат больше 7fh (7fffh), то результат насыщается до значения 7fh (7fffh):

```
.data   ;сегмент данных
mem     dw  5h
    df  0fffb22223382h
mem1    dw  8003h
    df  7ffedf7d7ffah
.code
...
    movq    rmmx0, mem   ;rmmx0=fffb 2222 3382 0005
                         ;mem1=7ffe df7d 7ffa 8003
    psubsw  rmmx0, mem1  ;rmmx0=8000 42a5 b388 7fff
...
```

Первая и последняя шестнадцатеричные тетрады показывают результат, сформированный в соответствии с принципом знакового насыщения.

- PSUBUSB | PSUBUSW приемник, источник — вычитание беззнаковых упакованных байтов и слов. Результат помещается в приемник, который является одним из MMX-регистров. Источник — либо MMX-регистр, либо 64-разрядная ячейка памяти. При возникновении ситуации, когда результат вычитания получается меньше 00h (0000h), поле соответствующего байта или слова формируется по принципу беззнакового насыщения (в нем остается значение 00h (0000h)):

```
.data   ;сегмент данных
mem     dw  5h
    df  0fffb22223382h
mem1    dw  8003h
    df  7ffe0f7d0ffah
.code
...
    movq    rmmx0, mem   ;rmmx0=fffb 2222 3382 0005
                         ;mem1 =7ffe 0f7d 0ffa 8003
    psubusw rmmx0, mem1  ;rmmx0=7ffd 12a5 2388 0000
...
```

Обратите внимание на последнюю шестнадцатеричную тетраду, которая соответствует результату вычитания (5 – 32771).

Умножение

Команды MMX-умножения предназначены только для умножения 16-разрядных элементов, причем реализация этих команд сделана несколько непривычно. Команды умножения целочисленного устройства формируют результат, размер которого вдвое превышает размер исходных операндов. Команды умножения MMX-расширения реализуют это действие несколько иначе. Во первых, умножению подвергаются одновременно 4 слова со знаком. Во-вторых, для получения полного результата умножения (размером в двойное слово) необходимо применение двух команд PMULHW и PMULLW. С их помощью формируются, соответственно, старшая и младшая части произведения. Для их объединения в единое двойное слово можно использовать команды расширения PUNPCKHWD или PUNPCKLWD:

- PMULHW приемник, источник — умножение четырех знаковых упакованных слов. Результат помещается в приемник, который является одним из MMX-регистров. Источник — либо MMX-регистр, либо 64-разрядная ячейка памяти. В приемник записываются не все 32 бита произведения, а только старшие 16 бит. Младшие 16 бит можно получить, используя команду PMULLW.
- PMULLW приемник, источник — умножение знаковых упакованных слов. Результат помещается в приемник, который является одним из MMX-регистров. Источник — либо MMX-регистр, либо 64-разрядная ячейка памяти. В приемник записываются не все 32 бита произведения, а только младшие 16 бит. Старшие 16 бит можно получить, используя команду PMULHW.
- PMADDWD приемник, источник — умножение четырех знаковых упакованных слов. Результат помещается в приемник, который является одним из MMX-регистров. Источник — либо MMX-регистр, либо 64-разрядная ячейка памяти. Формирование результата осуществляется по схеме, представленной на рис. 18.5.

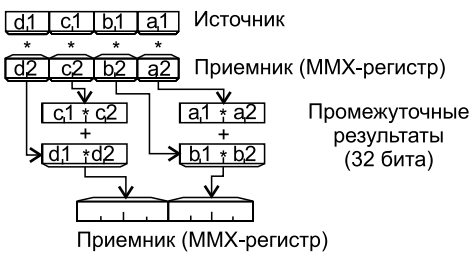

Рис. 18.5. Схема работы команды PMADDWD

Для иллюстрации работы команд PMULHW и PMULLW рассмотрим пример умножения четырех знаковых слов:

```
.data   ;сегмент данных
mem dw  5h
    df  7ffb22223382h
mem1    dw  8008h
    df  7ffe0f7d0ffah
mem2    dw  0
    df  0
mem3    dw  0
    df  0
.code
```

```
...
    movq    rmmx0, mem      ;rmmx0=7ffb 2222 3382 0005
                            ;mem1 =7ffe 0f7d 0ffa 8008
;получим младшие части произведений
    pmullw  rmmx0, mem1     ;rmmx0=800a a89a eaf4 8028
    movq    rmmx1, mem      ;rmmx1=7ffb 2222 3382 0005
                            ;mem1 =7ffe 0f7d 0ffa 8003
;получим старшие части произведений
    pmulhw  rmmx1, mem1     ;rmmx1=3ffc 0210 0336 fffd
;сохраним rmmx0 для последующих действий
    movq    rmmx2, rmmx0    ;rmmx2=800a a89a eaf4 8028
;в поле mem2 содержится результат произведений операндов
;младших половин исходных операндов
    punpcklwd   rmmx0, rmmx1;rmmx0=0336 eaf4 fffd 8028
    movq    mem2, rmmx0     ;mem2=0336 eaf4 fffd 8028
;в поле mem3 полный результат произведений операндов
;старших половин исходных операндов
    punpckhwd   rmmx2, rmmx1;rmmx2=3ffc 800a 0210 a89a
    movq    mem3, rmmx2     ;mem3=3ffc 800a 0210 a89a
...
```

С помощью программы-калькулятора Windows вы можете проверить полученные произведения. Вычислите с его помощью произведение 7ffb · 7ffe = 3ffc 800a. Такое же значение мы получили в поле mem3. Аналогично можно проверить и другие результаты.

Результат работы команды PMADDWD представляет собой сумму произведений двух старших и двух младших слов операндов приемник и источник (см. рис. 18.5). Результат получается в виде двух двойных упакованных слов. Далее показан пример использования команды PMADDWD:

```
.data   ;сегмент данных
mem dw  5h
    df  7ffb22223382h
mem1    dw  8008h
    df  7ffe0f7d0ffah
mem2    dw  0
    df  0
mem3    dw  0
    df  0
.code
...
    movq    rmmx0, mem  ;rmmx0=7ffb 2222 3382 0005
                        ;mem1 =7ffe 0f7d 0ffa 8008
    pmaddwd rmmx0, mem1 ;rmmx0=420d 28a4 0334 6b1c
...
```

Команды сравнения

Группа команд сравнения MMX-расширения содержит команды двух типов. Команды простого сравнения (PCMPEQB | PCMPEQW | PCMPEQD) характеризуются тем, что устанавливают только факт равенства операндов (равно — не равно). Команды сравнения по величине (CMPGTB | PCMPGTW | PCMPGTD) устанавливают соотношение операндов по величине:

- PCMPEQB | PCMPEQW | PCMPEQD операнд_1,операнд_2 — сравнение упакованных байтов, слов или двойных слов. Результат формируется в первом операнде, который является одним из MMX-регистров. Второй операнд — либо MMX-регистр, либо 64-разрядная ячейка памяти. Элементы формируемого результата представляются в виде единичных или нулевых байтов, слов или двойных слов.

Единичные байты, слова или двойные слова формируются, если соответствующие байты, слова или двойные слова исходных операндов равны. Нулевые байты, слова, двойные слова формируются, если соответствующие байты, слова или двойные слова исходных операндов не равны:

```
 .data   ;сегмент данных
 mem     dw  5h
     df  7fff22223382h
 mem1    dw  5h
     df  7ffe0f7d0ffah
 .code
 ...
     movq    rmmx0, mem  ;rmmx0=7fff 2222 3382 0005
                         ;mem1 =7ffe 0f7d 0ffa 0005
     pcmpeqw rmmx0, mem1 ;rmmx0=0000 0000 0000 ffff
 ...
```

- PCMPGTB | PCMPGTW | PCMPGTD операнд_1,операнд_2 — сравнение по величине упакованных байтов, слов или двойных слов. Результат формируется в первом операнде, который является одним из MMX-регистров. Второй операнд — либо MMX-регистр, либо 64-разрядная ячейка памяти. Элементы формируемого результата представляются в виде единичных или нулевых байтов, слов или двойных слов. Единичные байты, слова, двойные слова формируются в случае, если байты, слова или двойные слова исходного операнда операнд_1 были больше соответствующих байтов, слов или двойных слов операнда операнд_2. Иначе формируются нулевые байты, слова или двойные слова.

В примере ниже наглядно показано, что единичные элементы получаются только в случае, если элементы первого операнда (всегда находящегося в MMX-регистре) больше соответствующих элементов второго операнда:

```
.data   ;сегмент данных
mem dw  5h
    df  7fff22223382h
mem1    dw  5h
    df  7ffe0f7d0ffah
.code
...
    movq    rmmx0, mem  ;rmmx0=7fff 2222 3382 0005
                        ;mem1 =7ffe 0f7d f0fa 0005
    pcmpgtw rmmx0, mem1 ;rmmx0=ffff ffff 0000 0000
...
```

Команды логических операций

Команды логических операций предназначены для поразрядной обработки содержимого двух MMX-регистров или MMX-регистра и 64-разрядного операнда в памяти. Эти команды реализуют логические операции И, И-НЕ, ИЛИ, исключающее ИЛИ. Заметьте, что появилась новая логическая операция И-НЕ, которой нет среди логических команд основного процессора. Другая особенность логических команд заключается в том, что они выполняются над всеми шестьюдесятью четырьмя разрядами MMX-операндов. Для структурирования результатов их работы нужно применять другие команды MMX-расширения:

- PAND приемник, источник — выполнение поразрядной операции И над операндами. Результат помещается в приемник, который является одним из MMX-регистров. Источник — либо MMX-регистр, либо 64-разрядная ячейка памяти. Биты результата в приемнике формируются в соответствии с табл. 18.4.

Таблица 18.4. Таблица истинности для операции И

Приемник	0	0	1	1
Источник	0	1	0	1
Результат	0	0	0	1

Принцип работы команды PAND иллюстрирует пример:

```
.data   ;сегмент данных
mem     dw  0505h
    df  7ff002203080h
mem1    dw  0005h
    df  7ff002000f80h
.code
...
    movq    rmmx0, mem   ;rmmx0=7ff0 0220 3080 0505
                         ;mem1 =7ff0 0200 0f80 0005
    pand    rmmx0, mem1  ;rmmx0=7ff0 0200 0080 0005
...
```

- PANDN приемник, источник — выполнение поразрядной операции И-НЕ над операндами. Результат помещается в приемник, который является одним из MMX-регистров. Источник — либо MMX-регистр, либо 64-разрядная ячейка памяти. Биты результата в приемнике формируются в соответствии с табл. 18.5. Из таблицы видно, что единичные биты в результате могут появиться только в одном случае, когда единичные биты источника совпадают с нулевыми битами приемника. Команда позволяет определить положение единичных битов источника, которым не соответствуют единичные биты приемника.

Таблица 18.5. Таблица истинности для операции И-НЕ

Приемник	0	0	1	1
Источник	0	1	0	1
Результат	0	1	0	0

Принцип работы команды PANDN иллюстрирует пример:

```
.data   ;сегмент данных
mem     dw  0505h
    df  7ff002203080h
mem1    dw  0005h
    df  7ff003000f80h
.code
...
    movq    rmmx0, mem   ;rmmx0=7ff0 0220 3080 0505
                         ;mem1 =7ff0 0300 0f80 0005
    pandn   rmmx0, mem1  ;rmmx0=0000 0100 0f00 0000
...
```

- POR приемник, источник — выполнение поразрядной операции ИЛИ над операндами. Результат помещается в приемник, который является одним из MMX-регистров. Источник — либо MMX-регистр, либо 64-разрядная ячейка памяти. Биты результата в приемнике формируются в соответствии с табл. 18.6.

Принцип работы команды POR иллюстрирует пример:

```
.data   ;сегмент данных
mem     dw  0505h
    df  7ff002203080h
```

```
mem1    dw  0005h
    df  7ff003000f80h
.code
...
    movq    rmmx0, mem   ;rmmx0=7ff0 0220 3080 0505
                         ;mem1=7ff0 0300 0f80 0005
    por     rmmx0, mem1  ;rmmx0=7ff0 0320 3f80 0505
...
```

Таблица 18.6. Таблица истинности для операции ИЛИ

Приемник	0	0	1	1
Источник	0	1	0	1
Результат	0	1	1	1

- PXOR приемник, источник — выполнение поразрядной операции исключающее ИЛИ над операндами. Результат помещается в приемник, который является одним из MMX-регистров. Источник — либо MMX-регистр, либо 64-разрядная ячейка памяти. Биты результата в приемнике формируются в соответствии с табл. 18.7.

Таблица 18.7. Таблица истинности для операции исключающее ИЛИ

Приемник	0	0	1	1
Источник	0	1	0	1
Результат	0	1	1	0

Принцип работы команды PXOR иллюстрирует пример:

```
.data   ;сегмент данных
mem     dw  0505h
    df  7ff002203080h
mem1    dw  0005h
    df  7ff003000f80h
.code
...
    movq    rmmx0, mem   ;rmmx0=7ff0 0220 3080 0505
                         ;mem1 =7ff0 0300 0f80 0005
    pxor    rmmx0, mem1  ;rmmx0=0000 0120 3f00 0500
...
```

Команды сдвига

В главе 9 мы знакомились с командами арифметического и логического сдвига основного процессора. Различие этих двух типов команд сдвига — в способе интерпретации знакового бита операнда. Среди MMX-команд сдвига также существуют команды арифметического и логического сдвига. Обратите внимание на то обстоятельство, что MMX-команд сдвига упакованных байтов нет. Сдвигать можно только упакованные слова, двойные слова и учетверенные слова (целиком весь MMX-регистр):

- PSLLW | PSLLD | PSLLQ приемник, источник — команды логического сдвига влево упакованных слов, двойных слов или учетверенных слов в приемнике на количество разрядов, указанное значением источника. Результат помещается в приемник, который является одним из MMX-регистров. Источник — либо MMX-

регистр, либо 64-разрядная ячейка памяти. Освобождающиеся в результате сдвига младшие биты упакованных элементов приемника заполняются нулями.

- PSLLW | PSLLD | PSLLQ приемник, количество_сдвигов — команды логического сдвига влево аналогичны рассмотренным ранее командам, за исключением того, что все упакованные слова, двойные слова и учетверенные слова в приемнике сдвигаются на количество разрядов, указанное значением непосредственного операнда количество_сдвигов. Освобождающиеся в результате сдвига младшие биты упакованных элементов приемника заполняются нулями.
- PSRLW | PSRLD | PSRLQ приемник, источник — команды логического сдвига вправо упакованных слов, двойных слов или учетверенных слов в приемнике на количество разрядов, указанное значением в источнике. Результат помещается в приемник, который является одним из MMX-регистров. Источник — либо MMX-регистр, либо 64-разрядная ячейка памяти. Освобождающиеся в результате сдвига старшие биты упакованных элементов приемника заполняются нулями.
- PSRLW | PSRLD | PSRLQ приемник, количество_сдвигов — команды логического сдвига вправо аналогичны рассмотренным выше командам, за исключением того, что все упакованные слова, двойные слова или учетверенные слова в приемнике сдвигаются на количество разрядов, указанное значением непосредственного операнда количество_сдвигов. Освобождающиеся в результате сдвига старшие биты упакованных элементов приемника заполняются нулями:

```
.data   ;сегмент данных
mem     dw  0ffffh
    df  0ffffffffffffh
mem1    dw  4
    df  0
.code
...
    movq    rmmx0, mem   ;rmmx0=ffff ffff ffff ffff
                         ;mem1 =0000 0000 0000 0004
    psllw   rmmx0, mem1  ;rmmx0=fff0 fff0 fff0 fff0
    psrlw   rmmx0, 4     ;rmmx0=0fff 0fff 0fff 0fff
...
```

Следующие команды являются командами арифметического сдвига. Эти команды сдвигают значение операнда вправо. Команд арифметического сдвига влево нет, так как они аналогичны командам логического сдвига, не сохраняющим значения знакового разряда.

- PSRAW | PSRAD приемник, источник — команды арифметического сдвига вправо упакованных слов или двойных слов в приемнике на количество разрядов, указанное в источнике. Результат помещается в приемник, который является одним из MMX-регистров. Источник — либо MMX-регистр, либо 64-разрядная ячейка памяти. Освобождающиеся в результате сдвига старшие биты упакованных элементов приемника заполняются значением знаковых (старших) разрядов этих элементов:

```
.data   ;сегмент данных
mem     dw  0fff0h
    df  0fff0fff0fff0h
mem1    dw  4
    df  0
```

```
.code
...
    movq    rmmx0, mem   ;rmmx0=fff0 fff0 fff0 fff0
                         ;mem1 =0000 0000 0000 0004
    psraw   rmmx0, 4     ;rmmx0=ffff ffff ffff ffff
...
```

Команды упаковки и распаковки

Команды упаковки и распаковки предназначены для изменения размерности элементов операндов с учетом их значений. Команды упаковки позволяют уменьшить размерность элементов в два раза. При этом, если значение сжимаемого элемента больше максимального допустимого значения, которое может содержаться в элементе меньшего размера, то результат формируется по принципу знакового насыщения:

- PACKSSDW приемник, источник — команда упаковки со знаковым насыщением двух двойных слов в приемнике и двух двойных слов в источнике в четыре слова в приемнике. Схема выполнения команды показана на рис. 18.6. Результат помещается в приемник, который является одним из MMX-регистров. Источник — либо MMX-регистр, либо 64-разрядная ячейка памяти:

packssdw *приемник, источник*

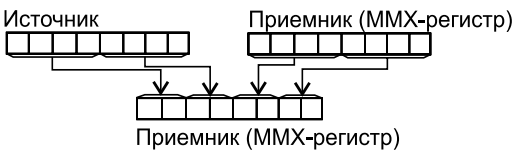

Рис. 18.6. Схема работы команды PACKSSDW

- PACKSSWB приемник, источник — команда упаковки со знаковым насыщением четырех слов в приемнике и четырех слов в источнике в четыре слова в приемнике. Схема выполнения команды показана на рис. 18.7. Результат помещается в приемник, который является одним из MMX-регистров. Источник — либо MMX-регистр, либо 64-разрядная ячейка памяти.

```
.data   ;сегмент данных
mem dw  0fe00h
    df  00457ffe00f0h
.code
...
    packsswb    rmmx0, mem   ;rmmx0=45 7f 7f 80 00 00 00 00
...
```

packsswb *приемник, источник*

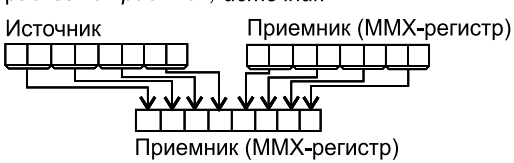

Рис. 18.7. Схема работы команды PACKSSWB

Пример показывает, как выполняется принцип знакового насыщения результата до значений 7fh и 80h байт. Подобная ситуация возникает каждый раз, когда значение в исходном слове превышает максимально возможное.

Следующая группа MMX-команд позволяет выполнить обратную операцию — расширить размер элементов операнда в два раза. При этом недостающая половина вновь формируемого элемента извлекается из второго операнда:

- PUNPCKHBW приемник, источник — команда распаковки байтов из старшей половины приемника в слова с использованием в качестве старшей половины этих слов байтов из источника. Формирование результата происходит путем поочередной выборки байтов из приемника и источника (рис. 18.8). Результат помещается в приемник, который является одним из MMX-регистров. Источник — либо MMX-регистр, либо 64-разрядная ячейка памяти:

```
.data   ;сегмент данных
mem     dw  0
    df  01020304ffffh
mem1    dw  0
    df  0f0f0f0feeeeh
.code
...
    movqr       mmx0, mem   ;rmmx0=01 02 03 04 ff ff 00 00
                            ;mem1 =0f 0f 0f 0f ee ee 00 00
    punpckhbw   rmmx0, mem1 ;rmmx0=0f01 0f02 0f03 0f04
...
```

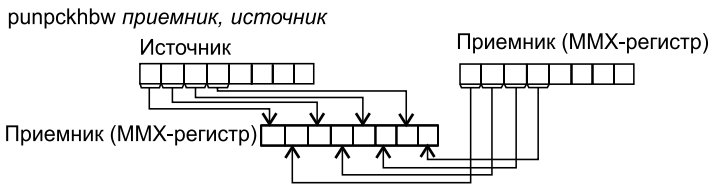

Рис. 18.8. Схема работы команды PUNPCKHBW

- PUNPCKHWD приемник, источник — команда распаковки слов из старшей половины приемника в двойные слова с использованием слов из источника в качестве старшей половины этих двойных слов. Формирование результата происходит путем поочередной выборки слов из приемника и источника (рис. 18.9). Результат формируется в приемнике, который является одним из MMX-регистров, в то же время источник может быть либо MMX-регистром, либо 64-разрядной ячейкой памяти.

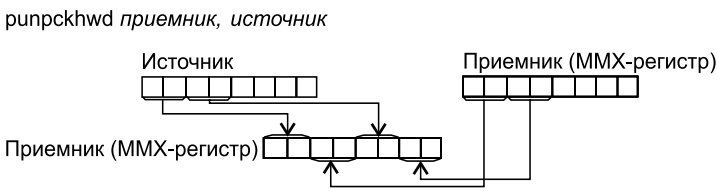

Рис. 18.9. Схема работы команды PUNPCKHWD

- PUNPCKHDQ приемник, источник — команда распаковки двойных слов из старшей половины приемника в учетверенные слова с использованием в качестве старшей половины этих учетверенных слов двойных слов из источника. Формирование результата происходит путем поочередной выборки двойных слов из приемника и источника (рис. 18.10). Результат помещается в приемник, который является одним из MMX-регистров. Источник — либо MMX-регистр, либо 64-разрядная ячейка памяти.

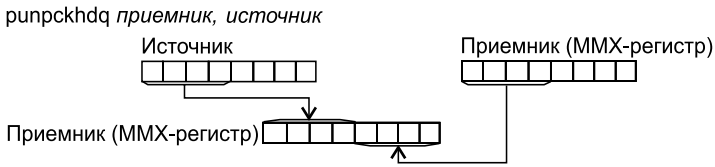

Рис. 18.10. Схема работы команды PUNPCKHDQ

Вы, наверное, обратили внимание на то, что предыдущие три команды работают со старшими половинами операндов. Следующие три команды, наоборот, работают с младшими половинами операндов:

- PUNPCKLBW приемник, источник — команда распаковки байтов из младшей половины приемника в слова с использованием в качестве младшей половины этих слов байтов из источника. Формирование результата осуществляется путем поочередной выборки байтов из приемника и источника (рис. 18.11). Результат помещается в приемник, который является одним из MMX-регистров. Источник — MMX-регистр или 64-разрядная ячейка памяти:

```
.data   ;сегмент данных
mem     dw  0304h
    df  0ffffeeee0102h
mem1    dw  0f0fh
    df  0c0c0c0c0f0fh
.code
...
    movqr       mmx0, mem   ;rmmx0=ff ff ee ee 01 02 03 04
                            ;mem1 =0c 0c 0c 0c 0f 0f 0f 0f
    punpcklbw   rmmx0, mem1 ;rmmx0=0f01 0f02 0f03 0f04
...
```

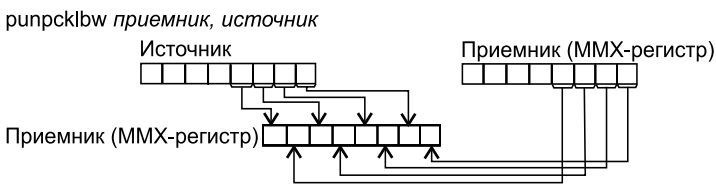

Рис. 18.11. Схема работы команды PUNPCKLBW

- PUNPCKLWD приемник, источник — команда распаковки слов из младшей половины приемника в двойные слова с использованием в качестве их младшей половины слов из источника. Формирование результата осуществляется путем поочередной выборки слов из приемника и источника (рис. 18.12). Результат помещается в приемник, который является одним из MMX-регистров, в то же

время источник может быть либо MMX-регистром, либо 64-разрядной ячейкой памяти.

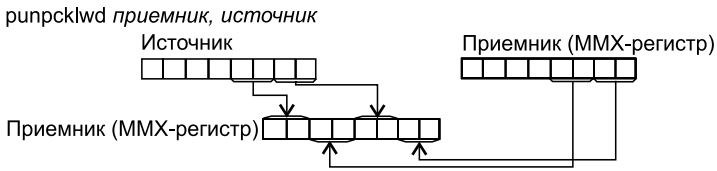

Рис. 18.12. Схема работы команды PUNPCKLWD

- `PUNPCKLDQ приемник, источник` — команда распаковки двойных слов из младшей половины приемника в учетверенные слова с использованием в качестве младшей половины этих учетверенных слов двойных слов из источника. Формирование результата осуществляется путем поочередной выборки двойных слов из приемника и источника (рис. 18.13). Результат помещается в приемник, который является одним из MMX-регистров. Источник — либо MMX-регистр, либо 64-разрядная ячейка памяти.

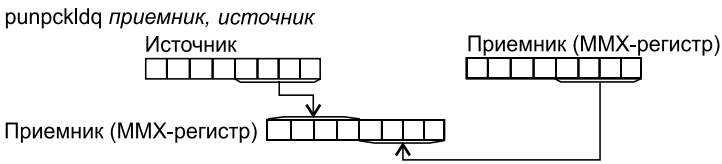

Рис. 18.13. Схема работы команды PUNPCKLDQ

Команда очистки стека регистров сопроцессора

В конце блока команд, содержащего MMX-инструкции, должна обязательно присутствовать команда `EMMS`, в функции которой входит очистка стека регистров и установка единичных значение в регистре тегов. Каждому регистру стека сопроцессора соответствует двухразрядное поле в этом регистре, единичное значение которого говорит о том, что регистр пуст.

Наличие команды `EMMS` обязательно, если MMX-команды комбинируются с командами сопроцессора.

В настоящее время еще довольно большое количество компьютеров работает на процессорах, не поддерживающих MMX-технологию. Чтобы количество пользователей вашей программы было как можно большим, это обстоятельство необходимо учитывать. Для этого при разработке программы должны предусматриваться альтернативные фрагменты кода, в которых функционирование MMX-команд будет эмулироваться с помощью целочисленных команд. Перед началом работы такая программа должна выполнять проверку текущего процессора на возможность поддержки им MMX-команд, по результатам которой передавать управление на соответствующий фрагмент кода. На основе чего выполняется подобная проверка?

В систему команд процессоров, начиная с i586 (Pentium), входит команда `CPUID` (см. приложение А), с помощью которой можно получить информацию о текущем

процессоре. Листинг 18.6 содержит пример программы, которая определяет факт поддержки процессором MMX-технологии.

Листинг 18.6. Определение факта поддержки процессором MMX-технологии

```
;test_cpuid.asm определение факта поддержки процессором MMX-технологии
.586p
model   use16 small
.stack  100h
.data   ;сегмент данных
mmx_mes db  "Процессор поддерживает MMX-технологию", "$"
no_mmx_mes db   "Процессор не поддерживает MMX-технологию", "$"
.code
main    proc;начало процедуры main
    mov ax, @data
    mov ds, ax
    xor edx, edx
    mov eax, 1
    cpuid
    bt  edx, 23
    jnc no_mmx
    mov ah, 9
    mov dx, offset  mmx_mes
    int 21h
    jmp exit
no_mmx: mov ah, 9
    mov dx, offset  no_mmx_mes
    int 21h
exit:
    mov ax, 4c00h   ;пересылка 4c00h в регистр ax
    int 21h         ;вызов прерывания с номером 21h
main    endp    ;конец процедуры main
end  main           ;конец программы с точкой входа main
```

Пример применения MMX-технологии

В качестве примера применения MMX-технологии рассмотрим преобразование графического изображения. Как правило, большинство мультимедийных файлов, к которым можно отнести и графические файлы, представляют собой массивы однородных элементов. Такому массиву предшествует некоторая описательная информация (заголовок), в котором содержится общая информация о файле. Так как в подобных массивах размер элементов обычно одинаков, то их удобно обрабатывать группами. Наибольшая эффективность работы программы достигается, если использовать MMX-команды.

В качестве типового мультимедийного файла можно рассматривать файл растрового изображения. Среди типов файлов, предназначенных для хранения растровых, изображений, наиболее простую структуру имеют bmp-файлы.

На практике, часто возникает необходимость преобразования цветного изображения в полутоновое. Подобную возможность, в частности, поддерживают многие популярные редакторы растровой графики. Попробуем самостоятельно решить эту задачу, используя возможности команд MMX-расширения.

Структурно файл растрового изображения в формате BMP содержит два или три блока:

- Обязательный заголовок с информацией, характеризующий растровое изображение. По отношению к содержащимся в этом заголовке данным его можно разделить на две части:

- первая часть, имеющая размер 14 байтов, предназначена для идентификации файла как растрового графического и хранения общей информации о нем;
- вторая часть, имеющая размер 40 байтов, содержит характеристики самого растрового изображения.

- Необязательный массив с информацией о цвете. Элементы этого массива представляют собой структуру, содержащую четыре поля. Размерность каждого поля составляет один байт. Три поля этой структуры содержат значения интенсивности красного, зеленого и синего цветов (RGB-модель цвета), четвертое поле всегда равно нулю и предназначено для выравнивания начала следующего элемента на четную границу. Для процессоров архитектуры Intel это означает существенное повышение производительности, так как выровненные данные выбираются из памяти быстрее. Более того, это обстоятельство дает хорошие предпосылки к эффективному использованию команд MMX-расширения, так как данные во время обработки можно группировать. Массив с информацией о цвете отсутствует для растровых изображений с 24-разрядным представлением цвета (True Color). В случае с нашей программой мы имеем на входе именно такой файл. На выходе мы должны получить файл с полутоновым изображением, который уже будет содержать подобный массив цветов.
- Обязательный массив с описанием пикселов растрового изображения. Формат элементов этого массива зависит от типа растрового изображения. В нашем случае мы имеем два типа растрового изображения: 24-разрядное (True Color) и 8-разрядное (полутоновое). Как отмечено ранее, файл с описанием 24-разрядного изображения не содержит массива с информацией о цвете. В этом файле непосредственно за заголовком (54 бита) следует информация о цвете пикселов. Каждый пиксел описывается тремя байтами, которые содержат значения (из диапазона 0...255) интенсивностей красного, зеленого и синего цветов. Для других типов изображений информация о цвете каждого пиксела предоставляется по другому принципу. К примеру, для восьмиразрядного изображения (которое мы должны сформировать в нашей задаче) информация о цвете конкретного пиксела получается следующим образом. Каждый пиксел в массиве с описанием пикселов растрового изображения описывается одним байтом. Значение этого байта является индексом в массиве с информацией о цвете, рассмотренном в предыдущем пункте. Соответствующий элемент массива с информацией о цвете содержит значения красной, синей и зеленой составляющих цвета пиксела.

Рассмотрим суть алгоритма обработки, который реализует программа преобразования растрового изображения. На рис. 18.14 и 18.15 приведены фрагменты исходного bmp-файла (24-разрядное изображение) и соответствующего выходного bmp-файла (8-разрядное полутоновое изображение), сформированного в результате обработки созданной программой.

На рис. 18.14 с адреса 0000:0036 и до конца файла (адрес 0008:DCF5) располагается описание пикселов — три последовательных байта описывают красную, синюю и зеленую составляющие одного пиксела.

```
0000:0000  42 4D F6 DC 08 00 00 00 00 00 36 00 00 00 28 00  BM........6...(.
0000:0010  00 00 B8 01 00 00 B8 01 00 00 01 00 18 00 00 00  ................
0000:0020  00 00 C0 DC 08 00 C2 1E 00 00 C2 1E 00 00 00 00  ................
0000:0030  00 00 00 00 00 00 49 7A A6 4A 7B A7 4D 7E AA 4F  ......Iz.J{.M~.O
0000:0040  80 AC 50 81 AD 50 81 AD 4F 80 AC 4F 80 AC 4B 7C  Ђ.Р..Р..ОЂ.ОЂ.K|
...
0008:DCE0  A5 7A 93 A3 79 93 A3 79 92 A6 7D 99 B1 92 AD C8  .z..y..y..}.....
0008:DCF0  AD CA E5 C2 DE FC    ...
```

Рис. 18.14. Фрагмент файла 24-разрядного растрового изображения в формате BMP

На рис. 18.15 с адреса 0000:0436 и до конца файла (адрес 0002:F875) располагается описание пикселов. Каждый байт в этом описании является индексом массива цветовых значений.

```
0000:0000  42 4D 76 F8 02 00 00 00 00 00 36 04 00 00 28 00  BMv.......6...(.
0000:0010  00 00 B8 01 00 00 B8 01 00 00 01 00 08 00 00 00  ................
0000:0020  00 00 40 F4 02 00 C2 1E 00 00 C2 1E 00 00 00 01  ..@.............
0000:0030  00 00 00 00 00 00 00 00 00 00 01 01 01 00 02 02  ................
0000:0040  02 00 03 03 03 00 04 04 04 00 05 05 05 00 06 06  ................
0000:0050  06 00 07 07 07 00 08 08 08 00 09 09 09 00 0A 0A  ................
0000:0060  0A 00 0B 0B 0B 00 0C 0C 0C 00 0D 0D 0D 00 0E 0E  ................
0000:0070  0E 00 0F 0F 0F 00 10 10 10 00 11 11 11 00 12 12  ................
0000:0080  12 00 13 13 13 00 14 14 14 00 15 15 15 00 16 16  ................
0000:0090  16 00 17 17 17 00 18 18 18 00 19 19 19 00 1A 1A  ................
0000:00A0  1A 00 1B 1B 1B 00 1C 1C 1C 00 1D 1D 1D 00 1E 1E  ................
0000:00B0  1E 00 1F 1F 1F 00 20 20 20 00 21 21 21 00 22 22  ......  .!!!.""
0000:00C0  22 00 23 23 23 00 24 24 24 00 25 25 25 00 26 26  ".###.$$$.%%%.&&
...    массив цветовых значений
0000:0400  F2 00 F3 F3 F3 00 F4 F4 F4 00 F5 F5 F5 00 F6 F6  ................
0000:0410  F6 00 F7 F7 F7 00 F8 F8 F8 00 F9 F9 F9 00 FA FA  ................
0000:0420  FA 00 FB FB FB 00 FC FC FC 00 FD FD FD 00 FE FE  ................
0000:0430  FE 00 FF FF FF 00 70 71 74 76 77 77 76 76 72 75  ......pqtvwwvvru
0000:0440  78 77 72 69 5F 5A 61 62 63 66 69 6D 72 74 67 70  xwri_Zabcfimrtgp
0000:0450  77 77 74 72 70 6D 6E 6B 69 6A 6C 6C 68 64 63 6F  wwtrpmnkijllhdco
0000:0460  74 6F 6F 76 7A 75 68 73 75 6F 74 7E 78 69 76 71  toovzuhsuot~xivq
0000:0470  6A 68 6C 74 7A 7E 86 7B 78 7C 78 6A 67 6E 69 79  jhltz~.{x|xjgniy
...
0002:F860  73 71 6D 6C 70 74 77 75 75 79 7D 7E 7C 79 8B 8D  sqmlptwuuy}~|y..
0002:F870  8C 8C 93 A7 C4 D8    ...
```

Рис. 18.15. Фрагмент файла 8-разрядного растрового изображения в формате bmp

Программа преобразования 24-разрядного растрового изображения в 8-разрядное выполнена в виде консольного приложения Windows. Таким образом, этим примером демонстрируется решение двух задач: во-первых, показан практический пример использования возможностей MMX-технологии обработки данных и, во-вторых, приведен еще один пример (см. главу 16) использования консольных Windows-приложений в программах на языке ассемблера.

Программа преобразования последовательно выполняет следующие действия:

1. Ввод имен исходного и выходного файлов. Следует отметить следующее требование к исходному файлу. Этот файл должен быть отсканирован с разрешением не менее 200 dpi, в противном случае возможны искажения. В нашем случае борьба за качество не является основной задачей, поэтому для исправления возможных искажений в программе не предпринимается никаких действий. Если сканера у вас нет, то нужно подобрать файл соответствующего качества.

Если у вас нет ни того, ни другого, то среди файлов, прилагаемых к книге, есть файл, с которым вы можете проводить необходимые эксперименты.

2. Открытие исходного и создание выходного файла. Для работы с этими файлами используется еще один полезный механизм Windows — *отображение файлов на память*. Этот механизм позволяет открыть файлы специальным образом и работать с ними далее как с обычными массивами. При рассмотрении программы обратите внимание на эту возможность и на то, как ею пользоваться в программах на языке ассемблера.
3. Определение корректности исходного файла. При этом проверяются два обстоятельства: первое — является ли он bmp-файлом, и второе — является ли он 24-разрядным файлом растрового изображения.
4. Формирование заголовка выходного файла на основании информации исходного файла. Основными при этом являются две задачи — определение размера выходного файла и заполнение полей, определяющих особенности данного растрового файла.
5. Формирование массива цветов. Массив содержит 256 элементов, что позволяет формировать 256-цветное растровое изображение. Файл с 8-разрядным полутоновым растровым изображением структурно ничем не отличается от 8-разрядного цветного файла. Единственное отличие можно увидеть при сравнении массивов цветов. Для массива 8-разрядного полутонового изображения значения красной, зеленой и синей составляющих одинаковы. Это особенность формирования полутонового изображения. Для цветных изображений подобной закономерности не существует, так как оттенки цветного изображения формируются комбинациями различающихся значений красной, зеленой и синей составляющих.
6. Формирование значений пикселов. Как мы уже упоминали ранее, каждый пиксел описывается одним байтом, который на самом деле является лишь индексом, обозначающим цвет в массиве цветов.
7. Закрытие файлов. В конце текста программы необходимо с помощью соответствующих функций Windows корректно закрыть файлы, в противном случае результаты работы программы будут потеряны.

Более детально алгоритм работы программы поясним по ходу рассмотрения ее текста, представленного в листинге 18.7. При этом мы не будем обсуждать функций Windows, обеспечивающих функционирование программы. Эту информацию вы можете почерпнуть из многочисленных источников. В частности программа разработана с использованием упоминавшегося ранее механизма отображения файлов на память [10]. Основное внимание будет уделено MMX-командам.

Листинг 18.7. Преобразование цветного изображения в полутоновое

```
;prg18.asm
;Пример консольного приложения для Win32
;(с использованием команд MMX-расширения и файлов, проецируемых в память)
.486
.model flat, STDCALL    ;модель памяти flat
;STDCALL - передача параметров в стиле C (справа налево)
;   вызываемая процедура чистит за собой стек
```

продолжение ⇨

Листинг 18.7 *(продолжение)*

```
%NOINCL ;запретить вывод текста включаемых файлов
include mmx32.inc
include WindowConA.inc
;Объявление внешними используемых в данной программе функций Win32 (ASCII):
extrn   AllocConsole:PROC
...
extrn   WriteConsoleA:PROC
;структура bmp-файла
;макроопределения типов
SSHORT  equ <dw 0>
UINT    equ <dw 0>
DDWORD  equ <dd 0>
LONG    equ <dd 0>
WWORD   equ <dw 0>
BBYTE   equ <db 0>
;структура для установки положения курсора в консоли:
Coord   struc
xx  SSHORT
yy  SSHORT
Coord   ends
;заголовок bmp-файла
BitMapFileHeader    struc
bfType  UINT        ;символы "B" и "M"
bfSize  DDWORD          ;размер файла в байтах
bfReserved1 UINT        ;резерв
bfReserved2 UINT        ;резерв
bfOffBits   DDWORD      ;смещение в байтах к началу растрового изображения
;характеристики растрового изображения
biSize  DDWORD          ;размер данной структуры в байтах
                        ;(должно быть равно 00000028h)
biWidth LONG        ;ширина изображения в пикселах
biHeight    LONG        ;высота изображения в пикселах
biPlanes    WWORD       ;число цветовых плоскостей (должно быть равно 1)
biBitCount  WWORD       ;число бит на пиксел (должно быть равно 1, 4, 8, 24)
biCompression   DDWORD  ;метод сжатия
biSizeImage DDWORD      ;размер собственно растрового изображения в байтах
biXPelsPerMeter LONG;разрешение по горизонтали в пикселах на метр
biYPelsPerMeter LONG;разрешение по вертикали в пикселах на метр
biClrUsed   DDWORD      ;число цветов в изображении
biClrImportant  DDWORD  ;число важных цветов изображения
BitMapFileHeader    ends
.data
NumWri  dd  0
inFile  db  80 dup (20)
outFile db  80 dup (20)
con Coord   <>
hinFile dd  0
houtFile    dd  0
hMapinFile  dd  0
hMapoutFile dd  0
TitleText   db  'MMX-преобразование bmp-файла', 0
mes1    db  'Введите путь к исходному файлу TrueColor:'
len_mes1=$-mes1
mes2    db  'Введите путь к выходному файлу :'
len_mes2=$-mes2
mesErr1 db  'Это не bmp-файл:'
len_mesErr1=$-mesErr1
mesErr2 db  'Это не TrueColor-файл:'
len_mesErr2=$-mesErr2
mesRet  db  'Нажмите любую клавишу для выхода'
len_mesRet=$-mesRet
dOut    dd  0
dIn dd  0
```

```
porog   label   dword
    dw  0000h
    dw  004dh
    dw  0097h
    dw  001ch   ;00 77 151 28
RabObl  dd  0
    dd  0
pixSizeLow  dd  0
pixSizeHi   dd  0
i8  dd  8
outFileSize dd  0
initRMMX0   label   dword
    db  00, 00, 00, 00, 01, 01, 01, 00
initRMMX1   label   dword
    db  02, 02, 02, 00, 02, 02, 02, 00
PointOutRegion  dd  0
PointInRegion   dd  0
nWrByte dd  0
temp    db  "B"
.code
start   proc near
;точка входа в программу:
;запрос консоли
    call    AllocConsole
;проверить успех запроса консоли
    test    ax, ax
    jz      exit;неудача
;текст окна заголовка
    push    offset TitleText
    call    SetConsoleTitleA
;вывод строки текста
;вначале получим дескрипторы ввода и вывода консоли
    push    STD_OUTPUT_HANDLE
    call    GetStdHandle
    mov     dOut, eax   ;dOut - дескриптор ввода-вывода консоли
    push    STD_INPUT_HANDLE
    call    GetStdHandle
    mov     dIn, eax;dIn - дескриптор ввода-вывода консоли
;установим курсор в позицию (2, 5)
    mov     con.xx, 2
    mov     con.yy, 5
    push    con
    push    dOut
    call    SetConsoleCursorPosition
    cmp     eax, 0
    jz      exit;если неуспех
;вывести приглашение на ввод имени исходного файла
    push    0
    push    offset NumWri
    push    len_mes1
    push    offset mes1
    push    dOut
    call    WriteConsoleA
;установим курсор в позицию (2, 6)
    mov     con.xx, 2
    mov     con.yy, 6
    push    con
    push    dOut
    call    SetConsoleCursorPosition
    cmp     eax, 0
    jz      exit;если неуспех
    push    0
    push    offset NumWri
    push    80
    push    offset inFile
```

продолжение ☞

Листинг 18.7 *(продолжение)*

```
    push    dIn
    call    ReadConsoleA
    lea     eax, inFile
    sub     NumWri, 2
    add     eax, NumWri
    mov     [eax], byte ptr 0
;установим курсор в позицию (2, 7)
    mov     con.xx, 2
    mov     con.yy, 7
    push    con
    push    dOut
    call    SetConsoleCursorPosition
    cmp     eax, 0
    jz      exit;если неуспех
;вывести приглашение на ввод имени выходного файла
    push    0
    push    offset NumWri
    push    len_mes2
    push    offset mes2
    push    dOut
    call    WriteConsoleA
;установим курсор в позицию (2, 8)
    mov     con.xx, 2
    mov     con.yy, 8
    push    con
    push    dOut
    call    SetConsoleCursorPosition
    cmp     eax, 0
    jz      exit;если неуспех
    push    0
    push    offset NumWri
    push    80
    push    offset outFile
    push    dIn
    call    ReadConsoleA
    lea     eax, outFile
    sub     NumWri, 2
    add     eax, NumWri
    mov     [eax], byte ptr 0
;------------------ inFile ----------------------------------------------
;открытие объекта ядра "файл" для исходного файла inFile
    push    NULL
    push    FILE_ATTRIBUTE_NORMAL
    push    OPEN_ALWAYS
    push    NULL
    push    0
    push    GENERIC_READ+GENERIC_WRITE  ;разрешены чтение и запись в файл
    push    offset inFile
    call    CreateFileA
;проверить успех
    cmp     eax, 0ffffffffh
    jz      exit;неудача
    mov     hinFile, eax
;создание объекта ядра "проецируемый файл" для исходного файла inFile
    push    NULL
    push    0   ;размер файла текущий
    push    0
    push    PAGE_READWRITE
    push    NULL
    push    hinFile
    call    CreateFileMappingA
;проверить успех
    cmp     eax, 0
```

```
    jz      exit;неудача
    mov     hMapinFile, eax
;проецирование файловых данных для исходного файла inFile на адресное
;пространство процесса
    push    NULL
    push    0   ;смещение первого считываемого байта в файле
    push    0
    push    FILE_MAP_WRITE
    push    hMapinFile
    call    MapViewOfFile
;проверить успех открытия файла
    cmp     eax, 0
    jz      exit;неудача
    mov     PointInRegion, eax
    mov     ebx, eax;адрес начала исходного файла в памяти в ebx
;------------------ inFile ---------------------------------------------
    mov     ebx, eax;адрес файла в памяти в ebx
    cmp     [ebx].biSize, 28h
    jne     exit_err1   ;это не bmp-файл
    cmp     [ebx].biBitCount, 18h
    jne     exit_err2   ;это не TrueColor файл
;преобразование True -> Gray
;------------------ outFile --------------------------------------------
;открытие объекта ядра "файл" для выходного файла outFile
    push    NULL
    push    FILE_ATTRIBUTE_NORMAL
    push    OPEN_ALWAYS
    push    NULL
    push    0
    push    GENERIC_READ+GENERIC_WRITE  ;разрешены чтение и запись в файл
    push    offset outFile
    call    CreateFileA
;проверить успех
    cmp     eax, 0ffffffffh
    jz      exit;неудача
    mov     houtFile, eax
;создание объекта ядра "проецируемый файл" для выходного файла outFile
;;вычисляем длину выходного файла TrueColor в пикселах
;(умножаем длину изображения на его высоту (в пикселах))
    mov     eax, [ebx].biWidth
    mul     dword ptr [ebx].biHeight
    mov     pixSizeLow, eax ;младшая половина размера растра в пикселах
    mov     pixSizeHi, edx  ;старшая половина размера растра в пикселах
;в eax размер собственно растрового изображения в байтах
    add     eax, 54+4*256   ;54 – размер фиксированного заголовка
                            ;4*256 – размер массива цветовых значений
;в eax размер выходного файла в байтах
    mov     outFileSize, eax
    push    NULL
    push    eax ;размер выходного файла
    push    NULL
    push    PAGE_READWRITE  ;PAGE_WRITECOPY
    push    NULL
    push    houtFile
    call    CreateFileMappingA
;проверить успех
    cmp     eax, 0
    jz      exit;неудача
    mov     hMapoutFile, eax
;проецирование файловых данных для выходного файла outFile на адресное
;пространство процесса
    push    outFileSize
    push    0   ;смещение первого считываемого байта в файле
    push    0
    push    FILE_MAP_WRITE  ;FileMapAllAccess
```

продолжение ⇨

Листинг 18.7 *(продолжение)*

```
    push    hMapoutFile
    call    MapViewOfFile
;проверить успех открытия файла
    cmp     eax, 0
    jz      exit;неудача
    mov     PointOutRegion, eax
    mov     edi, eax        ;адрес начала выходного файла в памяти в edi
```

Представленными в этом листинге действиями мы подготовили приложение к выполнению собственно преобразования. Для этого средствами Windows для работы с консолью были введены имена файлов. Затем эти файлы были открыты как файлы, отображаемые на память. Адреса файлов в памяти были помещены в ячейки PointInRegion и PointOutRegion. Для удобства работы эти адреса были помещены также в регистры EBX и EDI соответственно. Для того чтобы иметь возможность просматривать выходной файл стандартными средствами (графическим редактором Paint), необходимо сформировать корректный заголовок для выходного файла. При этом следует обратить внимание на формирование полей, характеризующих общий размер файла и размер собственно растрового изображения. Методика здесь простая. Из заголовка исходного файла извлекаются значения высоты и ширины растра в пикселах. Их произведение и дает размер всего растра в пикселах. Для 8-разрядного изображения это значение представляет собой длину растра в байтах. Для получения размера всего файла остается прибавить размер массива цветов (256 · 4 байт) и размер заголовка bmp-файла (54 байта). Эти действия мы выполняли, когда создавали выходной файл и определяли его размер. Результат этих действий мы будем использовать далее для формирования соответствующих полей заголовка.

Листинг 18.7 *(продолжение)*

```
;формируем заголовок выходного файла:
    mov ax, [ebx].bfType
    mov [edi].bfType, ax
    mov eax, outFileSize
    mov [edi].bfSize, eax       ;общий размер файла
    mov [edi].bfReserved1, 0
    mov [edi].bfReserved2, 0
    mov [edi].bfOffBits, 54+4*256
    mov [edi].biSize, 28h
    mov eax, [ebx].biWidth
    mov [edi].biWidth, eax      ;ширина растра в пикселах
    mov eax, [ebx].biHeight
    mov [edi].biHeight, eax     ;высота растра в пикселах
    mov [edi].biPlanes, 1       ;число цветовых плоскостей
    mov [edi].biBitCount, 8     ;число битов на пиксел
    mov [edi].biCompression, 0  ;метод сжатия
    mov eax, outFileSize
    sub eax, 54+4*256
    mov [edi].biSizeImage, eax  ;размер собственно растрового изображения
    mov eax, [ebx].biXPelsPerMeter
    mov [edi].biXPelsPerMeter, eax  ;ширина растра в пикселах
    mov eax, [ebx].biYPelsPerMeter
    mov [edi].biYPelsPerMeter, eax  ;высота растра в пикселах
    mov [edi].biClrUsed, 100h   ;число цветов в изображении
    mov [edi].biClrImportant, 0 ;число важных цветов в изображении
```

Далее формируем массив цветов. Особенности его содержимого мы обсуждали выше. Теперь нас интересуют особенности процесса его формирования. Каждый

элемент этого массива занимает 4 байта, поэтому для его формирования удобно использовать MMX-команды. Содержание исполняемых действий очень простое, поэтому попробуйте разобраться в них самостоятельно, обращаясь для справки к описанию команд MMX-расширения, приведенных в тексте этой главы (см. ранее подраздел «Система команд») и в приложении А.

Листинг 18.7 *(продолжение)*

```
;формируем массив цветов
    add     edi, 54
    mov     ecx, 128
    movq    rmmx0, initRMMX0
    movq    rmmx1, initRMMX1
    xor     esi, esi
m3:
    movq    [edi+esi*8], rmmx0
    padd    usb rmmx0, rmmx1
    dec     ecx
    inc     esi
    jcxz    m1
    jmp     m3
```

После формирования массива цветов все готово для выполнения собственно преобразования растрового изображения. Оно выполняется по определенному алгоритму, суть которого заключается в следующем. Каждый пиксел цветного изображения преобразуется в эквивалентный оттенок серого цвета. Для получения такого оттенка необходимо красную, синюю и зеленую составляющие цвета исходного пиксела умножить на определенные коэффициенты, после чего сложить. Полученный результат делится на 256 (можно сдвигом на 8 разрядов вправо). Частное от деления (размером в один байт) и является эквивалентным оттенком серого цвета, полученным из RGB-составляющих исходного цветного пиксела. Схематично процесс преобразования цвета пиксела показан на рис. 18.16.

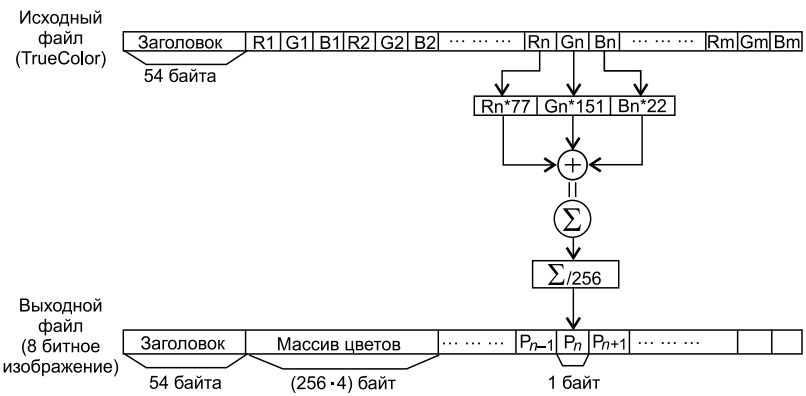

Рис. 18.16. Схема преобразования цветного пиксела (True Color) в эквивалентный серый

Листинг 18.7 *(продолжение)*

```
;----------------- outFile ---------------------------------------------
m1:
    mov ecx, pixSizeLow
    add edi, 4*256  ;[edi] - на начало собственно растрового изображения
                    ;в выходном файле
```

продолжение ⇨

Листинг 18.7 *(продолжение)*

```
    add ebx, [ebx]+0ah
    dec ebx ;в ebx адрес в файле начала изображения - 1
    xor esi, esi
;загрузка в регистры цветовых значений для преобразования
m2:
    punpcklbw   rmmx0, [ebx][esi*4]
    psrlw   rmmx0, 8
    pmullw  rmmx0, porog
    movqr   mmx1, rmmx0
    psllq   rmmx1, 16
    paddusw rmmx0, rmmx1
    psllq   rmmx1, 16
    paddusw rmmx0, rmmx1
    psrlw   rmmx0, 8;делим на 256
    movq    RabObl, rmmx0
    mov     al, byte ptr RabObl+6
    mov     [edi], al
    add     ebx, 3
    dec     ecx
    inc     edi
    jecxz   m5
    jmp     m2
m5: jmp     exit
```

В данном варианте программы обрабатываются только две ошибочные ситуации: ввод некорректного имени файла (или пути к нему) и выбор в качестве исходного файла с типом, отличным от TrueColor. В случае возникновения других ошибочных ситуаций производится переход на метку exit с последующим выходом из программы.

Листинг 18.7 *(продолжение)*

```
exit_err1:
;установим курсор в позицию (5, 10)
    mov     con.xx, 5
    mov     con.yy, 10
    push    con
    push    dOut
    call    SetConsoleCursorPosition
    cmp     eax, 0
    jz      exit;если неуспех
;вывести сообщение об ошибке
    push    0
    push    offset NumWri
    push    len_mes2
    push    offset mesErr1
    push    dOut
    call    WriteConsoleA
    jmp     exit;return
exit_err2:
;установим курсор в позицию (5, 10)
    mov     con.xx, 5
    mov     con.yy, 10
    push    con
    push    dOut
    callSetConsoleCursorPosition
    cmp     eax, 0
    jz      exit;если неуспех
;вывести сообщение об ошибке
    push    0
    push    offset NumWri
    push    len_mes2
```

```
    push    offset mesErr2
    push    dOut
    call    WriteConsoleA
;   jmp     return
;закрываем файлы
exit:
    emms
;установим курсор в позицию (5, 12)
    mov     con.xx, 5
    mov     con.yy, 10
    push    con
    push    dOut
    callSetConsoleCursorPosition
;вывести сообщение о нажатии любой клавиши
    push    0
    push    offset NumWri
    push    len_mesRet
    push    offset mesRet
    push    dOut
    call    WriteConsoleA
    push    0
    push    offset NumWri
    push    80
    push    offset inFile
    push    dIn
    call    ReadConsoleA
```

Перед выходом из программы необходимо должным образом закрыть файлы. Если этого не сделать, то их содержимое не изменится. Для корректного закрытия файла, спроецированного на память, необходимо вызвать функции FlushViewOfFile и CloseHandle. С помощью функции FlushViewOfFile измененные данные из памяти копируются на диск. Функция CloseHandle вызывается дважды для закрытия объектов ядра «файл» и «проецируемый файл».

Листинг 18.7 *(продолжение)*

```
    push    NULL
    push    PointInRegion
    call    FlushViewOfFile
    push    hinFile
    call    CloseHandle
    push    hMapinFile
    call    CloseHandle
    push    outFileSize
    push    PointOutRegion
    call    FlushViewOfFile
    push    houtFile
    call    CloseHandle
    push    hMapoutFile
    call    CloseHandle
;выход из приложения
return:
;готовим вызов VOID ExitProcess(UINT uExitCode)
    push    0
    call    ExitProcess
start   endp
end start
```

Дополнительные целочисленные MMX-команды

Представленные здесь команды нужно рассматривать как дополнение к системе целочисленных MMX-команд. Впервые они появились в архитектуре процессора Pentium III. Общее их количество — 12:

- PAVGB | PAVGW приемник, источник — команды вычисления среднего двух значений. Значения представляют собой беззнаковые целочисленные упакованные элементы операндов приемник и источник размером байт/слово. Источник может быть MMX-регистром или 64-разрядной ячейкой памяти, и его содержимое не изменяется. Изменяется только значение приемника, который должен быть MMX-регистром. В нем формируется результат вычисления среднего арифметического. Сама операция выполняется над парными элементами в обоих операндах следующим образом:
  1) выполнить беззнаковое сложение парных элементов операндов приемника и источника;
  2) запомнить перенос в старшие разряды;
  3) сдвинуть вправо на один разряд результат сложения (без учета бита переноса), то есть разделить на 2;
  4) результат сдвига сложить со значением переноса.

  Дополнительные MMX-команды придают MMX-расширению ряд новых возможностей для организации более эффективной обработки данных. Так, следующие команды позволяют организовать доступ к отдельным элементам MMX-операнда.
- PEXTRW приемник, источник, маска — команда извлечения одного из четырех упакованных слов операнда источник. Источник должен быть MMX-регистром. Извлекаемое из источника слово локализуется с помощью значения, задаваемого маской. Значение в ней имеют два младших бита, которые численно определяют номер слова (от 0 до 3), извлекаемого из источника. Извлеченное слово помещается в младшее слово приемника, являющегося одним из 32-разрядных регистров общего назначения. Старшее слово приемника обнуляется.
- PINSRW приемник, источник, маска — команда вставки слова в одно из четырех упакованных слов операнда приемник. Приемник должен быть MMX-регистром. Источник может быть 32-разрядным регистром общего назначения или словом памяти. Место вставки слова локализуется в приемнике с помощью значения, задаваемого маской. Значение в ней имеют два младших бита, которые численно определяют номер слова (от 0 до 3) в приемнике, замещаемого новым значением из источника. Если источник — 32-разрядный регистр, то вставляемое из него слово должно быть младшим.
- PMAXUB | PMAXSW приемник, источник — команды извлечения максимального значения из каждой пары упакованных элементов в операндах. Элементы представляют собой беззнаковые байты (для команды PMAXUB) или знаковые слова (для команды PMAXSW). Приемник должен быть MMX-регистром. Источник может быть MMX- регистром или 64-разрядной ячейкой памяти. Результат из максимальных элементов каждой пары формируется в операнде приемник.
- PMINUB | PMINSW приемник, источник — команды для извлечения минимального значения из каждой пары упакованных элементов в операндах. Элементы представляют собой беззнаковые байты (для команды PMINUB) или знаковые слова (для команды PMINSW). Приемник должен быть MMX-регистром. Источник

может быть MMX-регистром или 64-разрядной ячейкой памяти. Результат из минимальных элементов каждой пары формируется в операнде приемник.

Следующая команда необычная — она позволяет выделить значения знака (старшего бита) упакованных байтовых элементов. После чего эти биты собираются в один байт и помещаются в младшие разряды регистра общего назначения. Подобная операция позволяет выполнять анализ знаков упакованных байтов в MMX-регистре и организовывать ветвление программы.

- PMOVMSKB приемник, источник — команда для формирования байтового значения, биты которого являются знаковыми для всех восьми байтов, упакованных в MMX-регистре. Источник должен быть MMX-регистром. Результат из минимальных элементов каждой пары формируется в операнде приемник. Приемник является 32-разрядным регистром общего назначения (формируемый из знаковых разрядов байт будет являться младшим).
- PMULHUW приемник, источник — команда умножения упакованных беззнаковых слов с формированием в качестве результата старших слов произведения. Источник может быть MMX-регистром или 64-разрядной ячейкой памяти. Результат из старших слов произведения каждой пары формируется в операнде приемник. Приемник является MMX-регистром. Ранее мы уже рассматривали подобную команду — PMULHW, которая также работала со словами, но с учетом знака.
- PSADBW приемник, источник — команда вычисления суммарной разницы значений каждой пары беззнаковых байтов упакованных байтовых значений в приемнике и источнике. Источник может быть MMX-регистром или 64-разрядной ячейкой памяти. Результат из старших слов произведения каждой пары формируется в операнде приемник. Приемник является MMX-регистром. Понимание алгоритма работы этой команды вызывает определенные трудности. Поэтому рассмотрим его более детально:
  1) для каждой пары байтовых элементов операндов приемник и источник вычислить модуль их разности;
  2) сложить модули разности всех пар байтовых элементов и записать полученный результат в младшее слово приемника;
  3) старшие три слова приемника обнулить.
- PSHUFW приемник, источник, маска — команда перераспределяет упакованные слова из операнда источник в операнд приемник в соответствии с маской, заданной операндом маска. Источник может быть MMX-регистром или 64-разрядной ячейкой памяти. Результат перераспределения слов второго операнда помещается в приемник, который является MMX-регистром. Исходное содержимое приемника значения не имеет. Маска представляет собой байт, в котором:
  - численное значение битов 0 и 1 определяют, какое из четырех слов источника необходимо разместить в байтах 0 и 1 приемника;
  - численное значение битов 2 и 3 определяют, какое из четырех слов источника необходимо разместить в байтах 2 и 3 приемника;

- численное значение битов 4 и 5 определяют, какое из четырех слов источника необходимо разместить в байтах 4 и 5 приемника;
- численное значение битов 6 и 7 определяют, какое из четырех слов источника необходимо разместить в байтах 6 и 7 приемника.

Например, если источник имеет значение 0012 950b 0054 0fd5 и задана маска 01101101b, то после применения команды `PSHUFW` в приемнике будет сформировано значение 0054 950b 0012 0054.

XMM-расширение архитектуры процессора Pentium

В 1999 г. семейство процессоров Pentium фирмы Intel пополнилось новой моделью — процессором Pentium III (Katmai). Основу его архитектуры составляет ядро процессора Pentium II, дополненное модулем SSE (Streaming SIMD Extensions — потоковое SIMD-расширение). Потоковое SIMD-расширение дополняет MMX-технологию средствами обработки данных с плавающей точкой. Ранее мы условились называть этот тип MMX-расширения *XMM-расширением*.

XMM-расширение является продолжением политики Intel на ввод в архитектуру процессоров средств, повышающих эффективность вычислений на больших массивах однотипных данных. Программно-аппаратная модель XMM-расширения имеет простую и гибкую структуру. В нее входит ряд новых регистров, новые команды и форматы данных. Программы, разработанные для ранних моделей процессоров, будут также выполняться на компьютерах, оснащенных процессором Pentium III, но они не смогут задействовать новшества, введенные XMM-расширением. Для этого программа должна быть изначально написана с учетом логики работы программно-аппаратных средств, входящих в XMM-расширение.

Модель XMM-расширения

XMM-расширение добавляет следующие новые элементы в архитектуру процессора:

- восемь 128-разрядных регистров с плавающей точкой — `xmm0...xmm7` (XMM-регистры);
- формат данных (формат XMM) размером 128 бит, представляющий собой совокупность из четырех упакованных 32-разрядных чисел с плавающей точкой в коротком формате — SPFP (Single Precision Floating Point);
- набор XMM-команд;
- 32-разрядный регистр управления/состояния.

Рассмотрим их подробнее.

На рис. 18.17 показаны восемь 128-разрядных XMM-регистров для хранения данных и один регистр управления/состояния. Каждый из этих регистров доступен соответствующими командами XMM-расширения.

XMM-регистры применяются только для хранения данных. Не имеет смысла задействовать их для хранения адресов памяти — для этого достаточно регистров

Рис. 18.17. XMM-регистры

общего назначения. При написании программы программист может смешивать команды обоих типов MMX-расширения, так как они используют физически разные наборы регистров. Вследствие этого для работы с ними не нужны команды, подобные `EMMS`.

Основной тип данных XMM-расширения — число с плавающей точкой в коротком формате. Операндом XMM-команды является XMM-регистр или 128-разрядная ячейка памяти. Операнд содержит четыре числа с плавающей точкой в коротком формате. Поэтому говорят, что числа упакованы, и соответственно, формат данных называется *упакованным форматом с плавающей точкой* (форматом XMM). В главе 17 нами были рассмотрены различные форматы данных сопроцессора, и указанный формат был одним из них. Вспомним основные его характеристики:

- длина числа — 32 бита;
- длины полей в формате: мантиссы — 24 бита, порядка — 7 бит, знака — 1 бит;
- диапазон значений:
  - двоичное числа — $2^{-126}...2^{127}$;
  - десятичное числа — $1{,}18 \cdot 10^{-38}...1{,}70 \cdot 10^{38}$.

Формат XMM-регистров или 128-разрядной ячейки памяти отражает структуру формата XMM. Числа в *упакованном коротком формате с плавающей точкой* в пределах этих объектов нумеруются от 0 до 3. Описание данных формата XMM в программе на языке ассемблера производится по тем же принципам, которыми мы руководствовались при описании данных целочисленного MMX-расширения. В памяти данные в формате XMM располагаются в соответствии с обычным для архитектуры Intel порядком — «младший байт по младшему адресу» (рис. 18.18).

Назначение регистра управления/состояния `MXCSR` аналогично соответствующим регистрам сопроцессора `CWR` и `SWR` (см. главу 17). Так, большая часть битов этого регистра используется для маскирования/демаскирования исключений, другая часть — для установления способов округления, просмотра флагов состояния (табл. 18.8). Работа с этим регистром осуществляется с помощью команд `LDMXCSR`

Рис. 18.18. Схема расположения в памяти данных формата XMM

и FXRSTOR (загрузка содержимым из памяти), а также STMXCSR и FXSAVE (сохранение содержимого в памяти).

Таблица 18.8. Назначение битов регистра MXCSR

Биты	Название	Назначение
0...5	IE, DE, ZE, OE, UE, PE	Фиксация исключений с плавающей точкой (см. главу 17). Для очистки этих битов используется команда LDMXCSR. Возникающие исключения фиксируются для всего упакованного формата, и средств для определения конкретного числа (из четырех), вызвавшего это исключение, не существует
7...12	IM, DM, ZM, OM, UM, PM	Маскирование определенных типов исключений с плавающей точкой (см. главу 17). Начальное состояние этих битов единичное, что означает маскирование всех исключений. Для установки этих битов используется команда LDMXCSR. Маскирование исключений действует для всего упакованного формата
13...14	RC (Rounding Control)	Задание режима округления. По умолчанию действует режим округления к ближайшему представимому значению числа в коротком формате с плавающей точкой. При желании можно установить следующие режимы округления: 00 — округление к ближайшему; 01 — округление в меньшую сторону; 10 — округление в большую сторону; 11 — округление к нулю с усечением, то есть отбрасыванием дробной части
15	FZ (Flush-to-Zero)	Установка режима «сдвиг к нулю». Используется в ситуации переполнение следующим образом: возвращается нулевое значение вместе с истинным знаком результата; устанавливаются биты UE и PE в регистре MXCSR. По умолчанию бит FZ установлен в 0. Наличие бита FZ не соответствует стандарту IEEE-754 на вычисления с плавающей точкой, так как этот стандарт требует при возникновении переполнения формировать денормализованный операнд. Этот режим введен из соображений эффективности. Ценой небольшой потери точности можно достичь более быстрой работы приложений, для которых ситуация переполнение обычна (так как не требуется вызова обработчика исключения). Демаскирование бита UE имеет приоритет перед режимом fz

Остальные биты регистра MXCSR (биты 31–16 и бит 6) зарезервированы и установлены в 0. При попытке записать ненулевое значение в эти биты будет возбуждено исключение общей защиты.

Система команд

XMM-расширение процессора Pentium III дополнительно вводит в систему команд IA-32 70 новых машинных инструкций (рис. 18.19). Процессор Pentium IV дополняет систему команд IA-32 еще 144 новыми командами. В основном это команды XMM- и MMX-расширений. Из-за большого их количества в данном разделе ограничимся рассмотрением команд XMM-расширения Pentium III. Этого достаточно для иллюстрации принципов и идей, заложенных в XMM-расширение. Информацию о XMM-командах Pentium IV можно получить из приложения А.

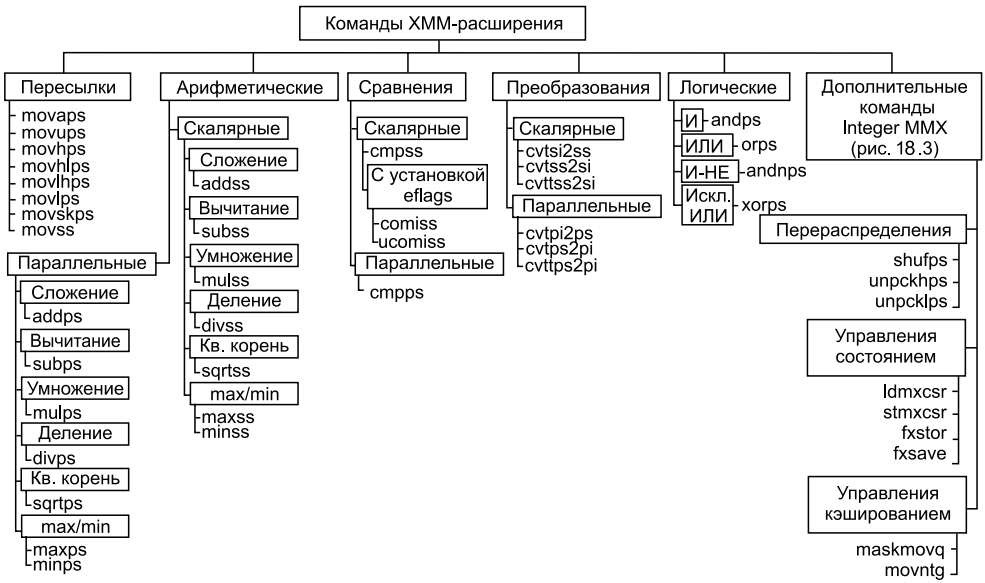

Рис. 18.19. Классификация команд XMM-расширения Pentium III

Команды перемещения данных

Далее перечислены команды перемещения данных:

- MOVAPS приемник, источник — пересылка выровненных 128 бит из источника в приемник. Один из операндов (любой) — XMM-регистр. Другой операнд должен быть XMM-регистром или 128-разрядной ячейкой памяти. Адрес 128-разрядной ячейки памяти должен быть выровнен по 16-разрядной границе, иначе произойдет исключение общей защиты.
- MOVUPS приемник, источник — пересылка не выровненных 128 бит из источника в приемник. Один из операндов (любой) — XMM-регистр. Другой операнд должен быть XMM-регистром или 128-разрядной ячейкой памяти. В отличие от

команды MOVAPS данная команда не требует выравнивания по 16-разрядной границе адреса 128-разрядной ячейки памяти.

- MOVHPS приемник, источник — пересылка не выровненных 64 бит из источника в приемник. Один из операндов, но не оба одновременно, должен быть XMM-регистром. Другой операнд — 64-разрядная ячейка памяти. Если пересылка производится из памяти, то 64 бита помещаются в старшие 64 бита XMM-регистра, если пересылка производится из регистра, то пересылке подлежат старшие 64 бита XMM-регистра. Младшие 64 бита XMM-регистра не изменяются. Данная команда не требует выравнивания по 16-разрядной границе адреса 64-разрядной ячейки памяти.
- MOVHLPS приемник, источник — пересылка 64 бит из источника в приемник. Оба операнда должны быть XMM-регистрами. В результате работы команды изменяется только содержимое младших 64 бит приемника в соответствии со схемой рис. 18.20.

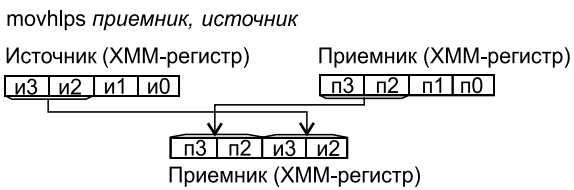

Рис. 18.20. Схема работы команды MOVHLPS

- MOVLHPS приемник, источник — пересылка 64 бит из источника в приемник. Оба операнда должны быть XMM-регистрами. В результате работы команды изменяются только содержимое старших 64 бит приемника в соответствии со схемой рис. 18.21.

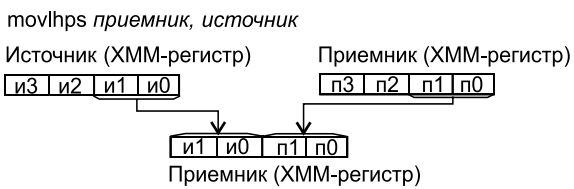

Рис. 18.21. Схема работы команды MOVLHPS

- MOVLPS приемник, источник — пересылка не выровненных 64 бит из источника в приемник. Один из операндов (любой) — XMM-регистр. Другой операнд должен быть 64-разрядной ячейкой памяти. Если пересылка производится из памяти, то 64 бита помещаются в младшие 64 бита XMM-регистра. Если пересылка производится из регистра, то пересылке подлежат младшие 64 бита регистра. Старшие 64 бита XMM-регистра не изменяются. Данная команда не требует выравнивания по 16-разрядной границе адреса 64-разрядной ячейки памяти.
- MOVMSKPS приемник, источник — пересылка знакового бита каждого из четырех упакованных чисел с плавающей точкой источника в младшие четыре бита

приемника (рис. 18.22). Источник должен быть XMM-регистром. Приемник должен быть 32-разрядным регистром общего назначения. В дальнейшем полученную четырехразрядную величину можно использовать для организации условных переходов.

Рис. 18.22. Схема работы команды MOVMSKPS

- MOVSS приемник, источник — пересылка 32 младших бит из источника в приемник. Один из операндов или все операнды должны быть XMM-регистрами. Если XMM-регистром является только один из операндов, другой операнд должен быть 32-разрядной ячейкой памяти. Если пересылка производится из памяти, то 32 бита помещаются в младшие 32 бита XMM-регистра. Если пересылка производится из регистра, то пересылке подлежат младшие 32 бита регистра XMM. Остальные биты XMM-регистра не изменяются.

Арифметические команды

Набор арифметических команд XMM-расширения включает обычные команды для выполнения арифметических операций. Особенность этого набора в том, что он содержит два типа арифметических операций: скалярные и параллельные. Различить эти команды можно по их мнемонике: скалярные команды имеют суффикс S, а параллельные — суффикс P. Команды *скалярных* арифметических операций обрабатывают только младшие 32-разрядные двойные слова упакованных операндов, остальные двойные слова в операции не участвуют и не изменяются. Команды *параллельных* арифметических операций обрабатывают одновременно четыре упакованных двойных слова. Заметьте также, что в наборе команд XMM-расширения присутствуют команды для операции деления, которых не было в целочисленном MMX-расширении.

Сложение и вычитание

- ADDPS (SUBPS) приемник, источник — параллельное сложение (вычитание) элементов операндов приемник и источник. Операнды должны иметь формат XMM. Результат операции помещается в приемник — XMM-регистр. Источник может быть XMM-регистром или 128-разрядной ячейкой памяти.
- ADDSS (SUBSS) приемник, источник — скалярное сложение (вычитание) операндов приемник и источник. Младшие двойные слова операндов должны быть числами с плавающей точкой в коротком формате. Результат помещается в операнд приемник. Приемник должен быть XMM-регистром. Источник может быть XMM-регистром или 32-разрядной ячейкой памяти. Содержимое старших битов операндов не имеет значения и не меняется.

Умножение и деление

- MULPS (DIVPS) приемник, источник — параллельное умножение (деление) операнда приемник на операнд источник. Операнды должны иметь формат XMM. Результат операции помещается в приемник — XMM-регистр. Источник может быть XMM-регистром или 128-разрядной ячейкой памяти.
- MULSS (DIVSS) приемник, источник — скалярное умножение (деление) операнда приемник на операнд источник. Младшие двойные слова операндов должны быть числами с плавающей точкой в коротком формате. Результат помещается в приемник. Приемник должен быть XMM-регистром. Источник может быть XMM-регистром или 32-разрядной ячейкой памяти. Содержимое старших битов операндов не имеет значения и не меняется.

Извлечение квадратного корня

- SQRTPS приемник, источник — параллельное извлечение квадратного корня из упакованных чисел с плавающей точкой операнда источник в формате XMM. Результат помещается в операнд приемник — XMM-регистр. Источник может быть XMM-регистром или 128-разрядной ячейкой памяти.
- SQRTSS приемник, источник — скалярное извлечение квадратного корня из упакованного числа с плавающей точкой операнда источник в формате XMM. Результат помещается в операнд приемник. Приемник должен быть XMM-регистром. Источник может быть XMM-регистром или 32-разрядной ячейкой памяти. Если источник — XMM-регистр, то в операции извлечения участвует только число с плавающей точкой в его младшем двойном слове. Содержимое старших битов операндов не имеет значения и не меняется.

Извлечение максимальных/минимальных значений операндов

- MAXPS (MINPS) приемник, источник — параллельное извлечение максимальных (минимальных) значений из каждой пары упакованных чисел с плавающей точкой операндов источник и приемник в формате XMM. Результат формируется из максимальных (минимальных) значений каждой пары и помещается в приемник — XMM-регистр. Источник может быть XMM-регистром или 128-разрядной ячейкой памяти, и его содержимое не меняется.
- MAXSS (MINSS) приемник, источник — скалярное извлечение максимального (минимального) значения из младшей пары упакованных чисел с плавающей точкой операндов источник и приемник в формате XMM. Результат формируется из максимального (минимального) значений каждой пары и помещается в приемник — XMM-регистр. Источник может быть XMM-регистром или 32-разрядной ячейкой памяти, и его содержимое не меняется. Если источник —XMM-регистр, то в операции сравнения и извлечения участвует только число с плавающей точкой в его младшем двойном слове.

Команды сравнения

Команды сравнения также делятся на два типа: скалярные и параллельные. Команды сравнения или их комбинации поддерживают полный набор условий срав-

нения. Обратите внимание на формат команд — он несколько необычен, так как условие сравнения задается непосредственно в операнде:

- CMPPS приемник, источник, условие — параллельное сравнение значений каждой пары упакованных чисел с плавающей точкой операндов источник и приемник в формате XMM. Условие, по которому производится сравнение, определяется третьим операндом, представляющим собой непосредственное значение. Результат сравнения формируется в приемнике — XMM-регистре. Источник может быть XMM-регистром или 128-разрядной ячейкой памяти, и его содержимое не меняется. Изменяется только значение приемника, которое замещается 32-разрядными значениями 00000000h или 0ffffffffh в зависимости от результата сравнения (00000000h, если условие не выполняется для данной пары значений упакованных чисел операндов источник и приемник, и 0ffffffffh, если условие выполняется).
- CMPSS приемник, источник, условие — скалярное сравнение младших пар чисел с плавающей точкой в коротком формате операндов приемник и источник. Условие, по которому производится сравнение, определяется третьим операндом, представляющим собой непосредственное значение. Результат сравнения формируется в приемнике — XMM-регистре. Источник может быть XMM-регистром или 32-разрядной ячейкой памяти, и его содержимое не меняется. Изменяется только значение приемника, младшее двойное слово которого замещается 32-разрядным значением 00000000h или 0ffffffffh в зависимости от результата сравнения (00000000h, если условие не выполняется для младшей пары чисел с плавающей точкой в коротком формате из операндов источник и приемник, и 0ffffffffh, если условие выполняется).

Предыдущие две команды сравнивали операнды, изменяя при этом содержимое операнда приемник. Следующие две команды позволяют выполнить операцию сравнения без изменения его содержимого, но с одним ограничением — эти команды являются скалярными. По результатам их работы устанавливаются флаги в регистре EFLAGS:

- COMISS приемник, источник, условие — скалярное сравнение младших пар чисел с плавающей точкой в коротком формате операндов приемник и источник. Условие, по которому производится сравнение, определяется третьим операндом, представляющим собой непосредственное значение. Результат сравнения формируется установкой флагов в регистре EFLAGS (устанавливаются флаги ZF, PF, CF и очищаются флаги OF, SF, AF). Приемник должен быть XMM-регистром. Источник может быть XMM-регистром или 32-разрядной ячейкой памяти. Содержимое приемника и источника не меняется.
- UCOMISS приемник, источник, условие — неупорядоченное скалярное сравнение младших пар чисел с плавающей точкой в коротком формате операндов приемник и источник. Данная команда отличается от команды COMISS тем, что позволяет обнаружить ситуацию, когда младшие 32 бита приемника являются тихим или сигнальным нечислом. Результат сравнения формируется установкой флагов в регистре EFLAGS (устанавливаются флаги ZF, PF, CF и очищаются флаги OF, SF, AF). Приемник должен быть XMM-регистром. Источник может быть XMM-

регистром или 32-разрядной ячейкой памяти. Содержимое приемника и источника не меняется.

Команды преобразования

Группа команд преобразования позволяет выполнить взаимное преобразование значений трех форматов: формата XMM, формата MMX и обычного целочисленного двоичного формата представления числа в одном из регистров общего назначения. Набор команд также делится на два класса: скалярные и параллельные:

- CVTPI2PS приемник, источник — параллельное преобразование двух 32-разрядных целочисленных значений в операнде источник, которым является MMX-регистр целочисленного MMX-расширения или 64-разрядная ячейка памяти. Операнд источник содержит два 32-разрядных целочисленных значения. Результат преобразования представляется в виде двух 32-разрядных чисел с плавающей точкой в коротком формате. Эти значения размещаются в двух младших двойных словах XMM-регистра, представляющего собой операнд приемник. Старшие два двойных слова в приемнике не изменяются. Если не удается получить точный результат преобразования, то производится его округление в соответствии с режимом, заданным в соответствующих битах регистра MXCSR.
- CVTPS2PI приемник, источник — параллельное преобразование двух 32-разрядных чисел с плавающей точкой в коротком формате, содержащихся в двух младших двойных словах операнда источник, в два 32-разрядных целых значения. Операнд источник представляет собой XMM-регистр или 64-разрядную ячейку памяти. Операнд приемник представляет собой адрес 64-разрядной ячейки памяти. Данная команда по своему действию является обратной команде CVTPI2PS. Если не удается получить точный результат преобразования, то производится его округление в соответствии с режимом, заданным в соответствующих битах регистра MXCSR.

  В системе команд XMM-расширения существует другой вариант последней команды — CVTTPS2PI, которая перед преобразованием отбрасывает дробную часть операнда источник.
- CVTSI2SS приемник, источник — скалярное преобразование одного 32-разрядного целочисленного значения из операнда источник в одно 32-разрядное число с плавающей точкой в коротком формате в операнде приемник. Операнд источник представляет собой один из 32-разрядных регистров общего назначения или 32-разрядную ячейку памяти. Результат преобразования в виде одного 32-разрядного числа с плавающей точкой в коротком формате помещается в младшее двойное слово 128-разрядного операнда приемник, представляющего собой XMM-регистр. Старшие три двойных слова в операнде приемник не меняются. Если не удается получить точный результат преобразования, то производится его округление в соответствии с режимом, заданным в соответствующих битах регистра MXCSR.
- CVTSS2SI приемник, источник — скалярное преобразование одного 32-разрядного числа с плавающей точкой в коротком формате из операнда источник в одно 32-разрядное целое число в операнде приемник. Операнд источник представляет собой XMM-регистр или 32-разрядную ячейку памяти. Результат преобразования

в виде одного 32-разрядного целочисленного значения помещается в операнд приемник, представляющий собой 32-разрядный регистр общего назначения. Старшие три двойных слова в операнде приемник не меняются. Если не удается получить точный результат преобразования, то производится его округление в соответствии с режимом, заданным в соответствующих битах регистра MXCSR.

В системе команд существует другой вариант последней команды — CVTTSS2SI, которая перед преобразованием отбрасывает дробную часть операнда источник.

Логические команды

Все логические команды являются параллельными и реализуют над парами битов в обоих операндах логические операции И, И-НЕ, ИЛИ, исключающее ИЛИ:

- ANDPS приемник, источник — параллельное выполнение логического умножения (операция логического И) над парами битов упакованных чисел с плавающей точкой операндов источник и приемник в формате XMM. Результат операции логического умножения формируется в операнде приемник — XMM-регистре. Источник может быть XMM-регистром или 128-разрядной ячейкой памяти, и его содержимое не меняется. Изменяется только значение приемника.
- ANDNPS приемник, источник — параллельное выполнение логической операции И-НЕ над парами битов упакованных чисел с плавающей точкой операндов источник и приемник в формате XMM. Результат выполнения логической операции И-НЕ формируется в операнде приемник. Приемник должен быть XMM-регистром. Источник может быть XMM-регистром или 128-разрядной ячейкой памяти, и его содержимое не меняется. Изменяется только значение приемника.
- ORPS приемник, источник — параллельное выполнение логического сложения (операция логического ИЛИ) над парами битов упакованных чисел с плавающей точкой операндов источник и приемник в формате XMM. Результат операции логического сложения формируется в операнде приемник. Приемник должен быть XMM-регистром. Источник может быть XMM-регистром или 128-разрядной ячейкой памяти, и его содержимое не меняется. Изменяется только значение приемника.
- XORPS приемник, источник — параллельное выполнение логической операции исключающего ИЛИ над парами битов упакованных чисел с плавающей точкой операндов источник и приемник в формате XMM. Результат логической операции «исключающего ИЛИ» формируется в операнде приемник. Приемник должен быть XMM-регистром. Источник может быть XMM-регистром или 128-разрядной ячейкой памяти, и его содержимое не меняется. Изменяется только значение приемника.

Команды управления состоянием

Далее перечислены команды управления состоянием:

- LDMXCSR источник — команда загрузки регистра MXCSR (регистра состояния и управления XMM-расширения) из памяти. Источник должен быть 32-разрядной ячейкой памяти.

- STMXCSR приемник — команда сохранения содержимого регистра MXCSR (регистра состояния и управления XMM-расширения) в памяти. Приемник должен быть 32-разрядной ячейкой памяти.
- FXRSTOR источник — команда загрузки состояния сопроцессора (целочисленного MMX-расширения) и XMM-расширения в области памяти размером 512 байт.
- FXSAVE приемник — команда сохранения состояния сопроцессора (целочисленного MMX-расширения) и XMM-расширения в области памяти размером 512 байт. Подобно команде FSAVE команда FXSAVE не инициализирует сопроцессор.

Команды перераспределения

Далее перечислены команды перераспределения:

- SHUFPS приемник, источник, маска — команда упакованной перестановки двойных слов из операндов источник и приемник в операнд приемник в соответствии с маской, заданной операндом маска. Источник может быть MMX-регистром или 128-разрядной ячейкой памяти. Результат перераспределения двойных слов обоих операндов помещается в приемник, который является MMX-регистром. Третий операнд является непосредственным операндом. Принцип формирования результата следующий. Младшие два двойных слова результата формируют два из четырех двойных слов приемника. Старшие два двойных слова результата формируют два из четырех двойных слов источника. Какие именно двойные слова будут включены в результат в качестве двойных слов, определяют биты маски. Значение первых двух битов маски определяют номер двойного слова источника, которое будет включено в результат в качестве первого двойного слова. Значение следующей пары битов (биты 2 и 3 маски) определяют номер двойного слова источника, которое будет включено в результат в качестве второго двойного слова. Значение следующей пары битов маски (4 и 5) определяют номер двойного слова приемника, которое будет включено в результат в качестве третьего двойного слова. Последняя пара битов маски (6 и 7) определяют номер двойного слова приемника, которое будет включено в результат в качестве четвертого двойного слова.
- UNPCKHPS приемник, источник — команда упакованного перемещения с чередованием старших двух двойных слов из операндов источник и приемник в операнд приемник. Источник может быть MMX-регистром или 128-разрядной ячейкой памяти. Результат перемещения двойных слов обоих операндов формируется в приемнике, который является MMX-регистром. Принцип формирования результата иллюстрирует рис. 18.23.

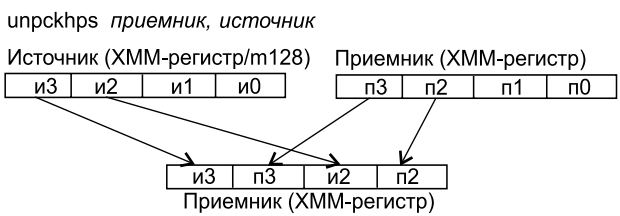

Рис. 18.23. Схема работы команды UNPCKHPS

- UNPCKLPS приемник, источник — команда упакованного перемещения с чередованием младших двух двойных слов из операндов источник и приемник в операнд приемник. Источник может быть MMX-регистром или 128-разрядной ячейкой памяти. Результат перемещения двойных слов обоих операндов формируется в операнде приемник, который является MMX-регистром. Принцип формирования результата иллюстрирует рис. 18.24.

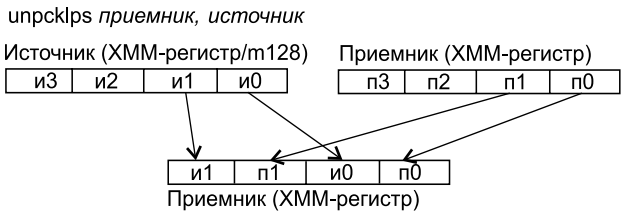

Рис. 18.24. Схема работы команды UNPCKLPS

Команды управления кэшированием

Команды XMM-расширения, рассмотренные в этом разделе, позволяют программисту управлять кэшированием данных. До появления моделей процессора с этим расширением в системе команд IA-32 было несколько команд, которые влияли на работу кэша, но они не отличались гибкостью и с их помощью тяжело было учесть особенности конкретной задачи. Потребность оптимизировать работу кэша возникла из-за большого объема данных, которыми манипулирует большинство мультимедийных приложений. Чтобы предотвратить загрузку в кэш данных, которые впоследствии могут вообще не понадобиться, были введены специальные команды. Более того, с помощью новых команд можно организовать загрузку данных заранее, то есть до того, как они реально будут востребованы. Понятно, что это поднимет скорость работы приложений.

Вначале будут рассмотрены три команды, которые обеспечивают программное управление записью данных в память из регистров целочисленного или потокового MMX-расширений, обеспечивая при этом минимальное «загрязнение» кэш-памяти:

- MASKMOVQ источник, маска — команда выборочного сохранения в памяти байтов упакованного целого числа из MMX-регистра. Источник является MMX-регистром целочисленного MMX-расширения. Местоположение приемника в памяти задано неявно содержимым пары DS:DI/EDI. Маска определяет, какие байты будут сохранены в памяти. Это делается следующим образом. Значение для данной команды имеет знаковый разряд каждого упакованного байта маски. Если он равен 1, то значение соответствующего байта из источника (MMX-регистра) переносится в соответствующий байт памяти. Если же он равен 0, то содержимое соответствующего байта из источника в память не переносится. Наглядно этот процесс иллюстрирует рис. 18.25. Особенностью команды MASKMOVQ (и, собственно, причиной, по которой она включена в данную группу команд) является то, что при записи информации из регистров в память не производится кэширования этой информации и, таким образом, отсутствует загрязнение кэш-памяти первого или второго уровня.

Рис. 18.25. Схема работы команды MASKMOVQ

- `MOVNTQ приемник, источник` — команда сохранения в памяти упакованных целых чисел в целочисленном формате MMX. Источник является MMX-регистром целочисленного MMX-расширения. Приемником является 64-разрядная ячейка памяти. Запись в память производится, минуя кэш-память.
- `MOVNPS приемник, источник` — команда сохранения в памяти упакованных чисел с плавающей точкой в формате XMM. Источник является XMM-регистром. Приемником является 128-разрядная ячейка памяти, адрес которой должен быть выровнен по границе параграфа (по значению, кратному 16). Запись в память производится, минуя кэш-память.

Таким образом, приведенные выше три команды отличаются от обычных команд пересылки политикой работы с кэш-памятью.

После общего обзора возможностей XMM-расширения рассмотрим несколько практических примеров программирования и пути решения специфических проблем, которые при этом могут возникать. Начнем с рассмотрения порядка описания данных, которыми манипулируют XMM-команды Pentium III(IV).

Описание упакованных и скалярных данных

Описание XMM-данных в приложении обычно производится в одном из двух форматов:

- в массиве структур;
- в структуре, элементами которой являются массивы.

Описание точек изображения в трехмерном пространстве принято задавать в виде четырехмерного вектора (x, y, z, w). Это связано с тем, что проекционные преобразования, необходимые для показа изображения с различных точек зрения, проще всего описываются матрицами 4×4. Используя перечисленные ранее форматы задания XMM-данных, совокупность точек в трехмерном пространстве можно описать двумя способами:

- Первый способ — для каждой точки определить свой экземпляр структуры:

```
point_3D struc
x   dd  0.0
y   dd  0.0
z   dd  0.0
w   dd  0.0
    ends
.data
```

```
p1  point_3D4 dup (<>)  ;описание пирамиды массивом структур,
                        ;каждая из которых описывает
                        ;одну из 4 вершин
;...
```

- Второй способ — все точки описать одной структурой, элементами которой являются массивы координат x, y, z, w:

```
prism_point_3D  struc
x   dd  4 dup (0.0)
y   dd  4 dup (0.0)
z   dd  4 dup (0.0)
w   dd  4 dup (0.0)
    ends
.data
prism   prism_point_3D  <>  ;структура, описывающая
                            ;треугольную пирамиду (4 вершины)
;...
```

Приведенные ранее примеры описания пирамиды иллюстрирует рис. 18.26.

Рис. 18.26. Расположение в памяти описания вершин пирамиды

В большинстве приложений используется первый способ представления XMM-данных, хотя он и считается менее эффективным. При необходимости можно произвести преобразование представления данных из одного способа в другой. Вариант такого преобразования показан в программе из листинга 18.8.

Листинг 18.8. Смена представления XMM-данных

```
;prg18_01.asm - программа преобразования XMM-данных
;из одного способа представления в другой.
;...
prizm       struc
union
struc           ;структура, описывающая треугольную пирамиду (1-й способ)
xyzw1   dd  0.0
xyzw2   dd  0.0
xyzw3   dd  0.0
xyzw4   dd  0.0
    ends
struc           ;структура, описывающая треугольную пирамиду (2-й способ)
x   dd  4 dup (0.0)
y   dd  4 dup (0.0)
z   dd  4 dup (0.0)
w   dd  4 dup (0.0)
    ends
    ends                ;конец объединения
```

продолжение ⇨

Листинг 18.8 *(продолжение)*

```
    ends
.data
prizm_1     prizm   <>              ;экземпляр объединения
.code
;...
;преобразование представлений вершин пирамиды (на месте)
    lea     si,prizm_1
    movlps  rxmm0,[si]          ;rxmm0=? ? y0 x0
    movhps  rxmm0,[si+16]       ;rxmm0= y1 x1 y0 x0
    movlps  rxmm2,[si+32]       ;rxmm2= ? ? y2 x2
    movhps  rxmm2,[si+48]       ;rxmm2= y3 x3 y2 x2
    movaps  rxmm1,rxmm0         ;rxmm1= y1 x1 y0 x0
    shufps  rxmm0,rxmm2,88h     ;rxmm0= x3 x2 x1 x0
    shufps  rxmm1,rxmm2,0ddh    ;rxmm1= y3 y2 y1 y0
    movlps  rxmm2,[si+8]        ;rxmm2=? ? w0 z0
    movhps  rxmm2,[si+24]       ;rxmm2= w1 z1 w0 z0
    movlps  rxmm4,[si+40]       ;rxmm4= ? ? w2 z2
    movhps  rxmm4,[si+56]       ;rxmm4= w3 z3 w2 z2
    movaps  rxmm3,rxmm2         ;rxmm3= w1 z1 w0 z0
    shufps  rxmm2,rxmm4,88h     ;rxmm2= = z3 z2 z1 z0
    shufps  rxmm3,rxmm4,0ddh    ;rxmm3= w3 w2 w1 w0
;на выходе получим следующее состояние XMM-регистров:
;RXMM0= x3 x2 x1 x0, RXMM1= y3 y2 y1 y0,
;RXMM2= = z3 z2 z1 z0, RXMM3= w3 w2 w1 w0
;теперь их необходимо сохранить в памяти:
    movups  [si],rxmm0
    movups  [si+16],rxmm1
    movups  [si+32],rxmm2
    movups  [si+48],rxmm3
;...
```

Описание скалярных данных намного проще — это обычные значения с плавающей точкой в коротком формате:

```
.data
scal_real   dd  1.0     ;пример описания скалярного XMM-значения;
```

Примеры использования команд XMM-расширения

Далее будут рассмотрены несколько типовых примеров использования команд XMM-расширения. Основная цель — демонстрация методики работы с основными группами команд XMM-расширения. Начнем с реализации простейших операций — сложения и умножения.

Сложение и умножение двух упакованных XMM-значений

Задача — вычислить скалярное произведение двух векторов, каждый из которых состоит из четырех вещественных чисел в коротком формате. Если в качестве таких векторов взять два вектора **A** и **B**, то их произведение вычисляется по формуле

$$\mathbf{A} \times \mathbf{B} = a_0 \cdot b_0 + a_1 \cdot b_1 + a_2 \cdot b_2 + a_3 \cdot b_3.$$

В программной реализации с использованием XMM-команд это выглядит так, как показано далее.

Листинг 18.9. Программа вычисления скалярного произведения двух векторов

```
;prg18_02.asm
;...
.data
```

```
xmm_pack_1  dd  1.0, 2.0, 3.0, 4.0
xmm_pack_2  dd  5.0, 6.0, 7.0, 8.0
rez_sum     dd  0.0 ;результат сложения
.code
;...
    movaps  rxmm0,xmm_pack_1;RXMM0= 4.0, 3.0, 2.0, 1.0
    mulps   rxmm0,xmm_pack_2;RXMM0= 4.0*8.0, 3.0*7.0, 2.0*6.0, 1.0*5.0
    movaps  rxmm1, rxmm0    ;RXMM1= 4.0*8.0, 3.0*7.0, 2.0*6.0, 1.0*5.0
    shufps  rxmm1,rxmm1,4eh ;RXMM1= 2.0*6.0, 1.0*5.0, 4.0*8.0, 3.0*7.0
    addps   rxmm0, rxmm1    ;складываем:
;RXMM0= 4.0*8.0, 3.0*7.0, 2.0*6.0, 1.0*5.0
;+
;RXMM1= 2.0*6.0, 1.0*5.0, 4.0*8.0, 3.0*7.0
;=
;RXMM0= 4.0*8.0+2.0*6.0, 3.0*7.0+1.0*5.0, 2.0*6.0+4.0*8.0, 1.0*5.0+3.0*7.0
;или
;RXMM0= 44.0, 26.0, 44.0, 26.0
    movaps  rxmm1, rxmm0    ;RXMM1= 44.0, 26.0, 44.0, 26.0
    shufps  rxmm1,rxmm1,11h ;RXMM1= 26.0, 44.0, 26.0, 44.0
    addps   rxmm0, rxmm1    ;складываем:
;RXMM0= 44.0, 26.0, 44.0, 26.0
;+
;RXMM1= 26.0, 44.0, 26.0, 44.0
;=
;RXMM0= 70.0, 70.0, 70.0, 70.0
;сохраняем результат
    movss   rez_sum,rxmm0
;...
```

Умножение матрицы на вектор

Умножение матрицы на вектор — наиболее характерная операция для вычислений в области машинной графики. Существуют различные способы формирования трехмерного изображения. В наиболее простом случае изображение на экране задается в виде опорных точек. К примеру, рассмотрим случай, когда на экране дисплея находится трехмерное изображение, состоящее из отрезков прямых. Необходимая для его формирования информация хранится в памяти как список опорных точек — концов отрезков. Если изображение дается в трехмерном изображении, то описание каждой точки удобно задать в виде четырехмерного вектора (x, y, z, w). Включение в трехмерный вектор (x, y, z) дополнительной координаты w объясняется тем, что проекционные преобразования, необходимые для показа изображения с различных точек зрения, описываются матрицами 4×4. Поэтому для удобства реализации проекционных преобразований, зачастую сопровождаемых операциями умножения матриц и векторов, трехмерный вектор (x, y, z) представляют в виде четырехмерного вектора (x, y, z, w), где значение w обычно принимается равным 1. Для выполнения самого преобразования подготовленная заранее матрица преобразования умножается на этот вектор, в результате чего получается четырехмерный вектор (x', y', z', w'). Для обратного перехода к требуемому трехмерному вектору необходимо разделить координаты x', y', z' на w', после чего удалить четвертую координату w'.

Матрица **M** преобразования и вектор **V** имеют следующий вид:

$$m_{00}\ m_{01}\ m_{02}\ m_{03}x;$$
$$m_{10}\ m_{11}\ m_{12}\ m_{13}y;$$
$$m_{20}\ m_{21}\ m_{22}\ m_{23}z;$$
$$m_{30}\ m_{31}\ m_{32}\ m_{33}w = 1.$$

Преобразования координат выполняются по формулам

$$x' = x \cdot m_{00} + y \cdot m_{01} + z \cdot m_{02} + 1 \cdot m_{03};$$
$$y' = x \cdot m_{10} + y \cdot m_{11} + z \cdot m_{12} + 1 \cdot m_{13};$$
$$z' = x \cdot m_{20} + y \cdot m_{21} + z \cdot m_{22} + 1 \cdot m_{23};$$
$$w' = x \cdot m_{30} + y \cdot m_{31} + z \cdot m_{32} + 1 \cdot m_{33}.$$

Для получения преобразованных координат в виде трехмерного вектора (x,y,z) делим x', y', z' на w':

$$x = x'/w' = (x \cdot m_{00}+y \cdot m_{01}+z \cdot m_{02}+1 \cdot m_{03})/(x \cdot m_{30}+y \cdot m_{31}+z \cdot m_{32}+1 \cdot m_{33});$$
$$y = y'/w' = (x \cdot m_{10}+y \cdot m_{11}+z \cdot m_{12}+1 \cdot m_{13})/(x \cdot m_{30}+y \cdot m_{31}+z \cdot m_{32}+1 \cdot m_{33});$$
$$z = z'/w' = (x \cdot m_{20}+y \cdot m_{21}+z \cdot m_{22}+1 \cdot m_{23})/(x \cdot m_{30}+y \cdot m_{31}+z \cdot m_{32}+1 \cdot m_{33}).$$

Элементы матрицы и векторов представлены числами с плавающей точкой в коротком вещественном формате (4 байта).

Далее приведены два варианта программы умножения матрицы на вектор — один с использованием стандартного сопроцессора, другой с использованием команд XMM-расширения (с помощью профайлера можно сравнить скорость преобразования). Результирующий трехмерный вектор замещает исходный.

Подпрограмма умножения матрицы 4×4 на четырехмерный вектор с использованием стандартного сопроцессора состоит из двух частей. В первой части вычисляется собственно произведение четырехмерной матрицы на четырехмерный вектор. Во второй части полученный четырехмерный вектор преобразуется в трехмерный делением его компонентов на его четвертую координату.

Листинг 18.10. Умножение матрицы на вектор (стандартный сопроцессор)

```
;prg18_03.asm
;...
.data
xyzw    dd  3 dup (0.0), 1.0     ;исходные координаты точки
                                 ;(необходимо инициализировать)
                                 ;и w=1.0
M       dd  16 dup (0.0)         ;матрица преобразования расположена
                                 ;по строкам (необходимо инициализировать)
.code
;...
    lea     di,xyzw              ;адрес вектора
    lea     bx,M                 ;адрес матрицы M
    finit                        ;инициализировать сопроцессор
    fld     dword ptr [di+12]    ;включить в стек w
    fld     dword ptr [di+8]     ;включить в стек z
    fld     dword ptr [di+4]     ;включить в стек y
    fld     dword ptr[di]        ;включить в стек x
    mov     cx,4                 ;счетчик строк матрицы M
;умножаем очередную строку M на (x,y,z,w)
row:    fld dword ptr[bx]        ;первый элемент очередной строки
    fmul    st,st(1)             ;умножить на x
    fld     dword ptr [bx+4]     ;второй элемент строки M
    fmul    st,st(3)             ;умножить на y
    fadd                         ;накопление суммы
    fld     dword ptr [bx+8]     ;третий элемент строки M
    fmul    st,st(4)             ;умножить на z
    fadd                         ;накопление суммы
```

```
    fld     dword ptr [bx+12]   ;четвертый элемент строки M
    fmul    st,st(5)            ;умножить на w
    fadd                        ;накопление суммы
    fstp    dword ptr [di]      ;запомнить элемент вектора
    add     di,4                ;на следующую координату в xyzw
    add     bx,16               ;к следующей строке M
    loop    row                 ;повторить (всего 3 раза)
;теперь необходимо разделить полученные x', y', z', w' на w':
    lea     di,xyzw             ;адрес вектора
    fld     dword ptr [di+12]   ;w' в стек
    mov     cx,4
cycl:   fld dword ptr [di]      ;очередную координату в стек
    fdiv ;st(0)/st(1)->st(0)    ;делим на w'
    fstp    dword ptr [di]          ;запоминаем очередную координату
    add     di,4                ;к следующей координате
    loop    cycl
;получили (x, y, z, 1)=> убираем w=1
;и получаем результат преобразования (x,y,z)
;...
```

Подпрограмму умножения матрицы 4×4 на четырехмерный вектор с использованием XMM-расширения иллюстрирует листинг 18.11.

Листинг 18.11. Умножение матрицы на вектор (XMM-расширение)

```
;prg18_04.asm
;...
.data
;ALIGN 16
xyzw    dd  3 dup (2.0), 1.0    ;исходные координаты точки
                                ;(необходимо инициализировать)
                                ;и w=1.0
M       dd  16 dup (0.0)        ;матрица преобразования расположена
                                ;по строкам (необходимо инициализировать)
.code
;...
    lea     edi,xyzw            ;адрес вектора
    lea     ebx,M               ;адрес матрицы M
    mov     ecx,4               ;повторений цикла
    movups  rxmm0,xyzw          ;загружаем вектор
row:    movups  rxmm1,rxmm0     ;копируем вектор
    movups  rxmm2,[ebx]         ;загружаем очередную строку M
    mulps       rxmm2,rxmm1     ;умножаем вектор на очередную строку M
;складываем частичные умножения:
    shufps  rxmm1, rxmm2,0ah
    addps   rxmm2, rxmm1
    shufps  rxmm3, rxmm2,1
    addss   rxmm3, rxmm2        ;скалярное сложение
    movss   [edi],rxmm3         ;скалярная пересылка
;готовимся к вычислению следующей координаты
    add     edi,4
    loop    row
;теперь необходимо разделить полученные x', y', z', w' на w':
    lea     edi,xyzw            ;адрес вектора
    movups  rxmm0,xyzw          ;загружаем вектор (x',y',z',w')
    movss   rxmm1,rxmm0         ;пересылка w'
    shufps  rxmm1, rxmm1,0      ;размножаем w'
    divps   rxmm0,rxmm1
    movups  xyzw, rxmm0         ;сохраняем новое значение вектора
;получили (x, y, z, 1)=> убираем w=1
;и получаем результат преобразования (x,y,z)
;...
```

Поворот изображения

Следующая проблема, постоянно решаемая в задачах двухмерной графики, — поворот изображения на определенный угол. Преобразование координат (x_i, y_i) опорных точек при повороте на угол α может выполняться по следующим формулам:

$$x'_i = x_i \cdot \cos\alpha + y_i \cdot \sin\alpha;$$
$$y'_i = -x_i \cdot \sin\alpha + y_i \cdot \cos\alpha.$$

Здесь x_i и y_i — координаты опорных точек, $i = 0...n-1$; n — число опорных точек изображения.

Аналогично программе умножения матрицы на вектор приведем два варианта программы поворота двухмерного изображения (пусть это будет прямоугольник) — с использованием сопроцессора и с использованием XMM-расширения.

В приведенной в листинге 18.12 программе поворота изображения с использованием стандартного сопроцессора предполагается, что координаты прямоугольника хранятся в виде массива вещественных чисел короткого формата. Элементы массива логически представляют собой последовательность x0, y0, x1, y1, x2, y2, x3, y3. Угол поворота в радианах находится в переменной a.

Листинг 18.12. Поворота изображения на месте (стандартный сопроцессор)

```
;prg18_05.asm
;...
.data
;координаты квадрата (необходимо инициализировать) x0,y0,x1,y1,x2,y2,x3,y3
mas_xy  dd  8 dup (0.0)
a       dd  0.0 ;угол (необходимо инициализировать)
.code
;...
    lea     si,mas_xy
    mov     cx,4;цикл 4 раза - по количеству вершин
    finit
;вычисляем sin a и cos a;
    fld     a                   ;включаем в стек угол
    fsin                        ;вычисляем sin a
    fxch                        ;меняем st(0)<->st(1)
    fcos                        ;вычисляем cos a
    fxch                        ;меняем st(0)<->st(1)
    fstpa                       ;выталкиваем a
;поворот изображения
cycl:   fild word ptr [si]      ;включить в стек координату x элемента
    fild    word ptr [si+2]     ;включить в стек координату y элемента
    fld     st(1)               ;дублируем их
    fld     st(1)
    fmul    st,st(5)            ;вычислить y*sin
    fxch                        ;меняем st(0)<->st(1)
    fmul    st,st(4)            ;вычислить x*cos
    fadd                        ;новая координата x
    fistp   word ptr[si]        ;передать новую координату x в память
    fmul    st,st(2)            ;вычислить y*cos
    fxch                        ;меняем st(0)<->st(1)
    fmul    st,st(3)            ;вычислить x*sin
    fsubr                       ;новая координата y
    fistp   word ptr[si+4]      ;передать ее в память
    add     si,8                ;продвинуть указатель массива mas_xy
    loop    cycl                ;повторить еще 3 раза
;в mas_xy преобразованные для поворота координаты квадрата
;...
```

Программа поворота изображения с использованием XMM-расширения представлена в листинге 18.13.

Листинг 18.13. Поворот изображения на месте (XMM-расширение)

```
;prg18_06.asm
;...
.data
;ALIGN 16
;координаты квадрата (необходимо инициализировать) x0,y0,x1,y1,x2,y2,x3,y3
mas_xy  dd  8 dup (0.0)
a       dd  0.0             ;угол (необходимо инициализировать)
sin_a   dd  0.0
cos_a   dd  0.0
null    dd  0.0
.code
;...
    lea     esi,mas_xy
    mov     ecx,4           ;цикл 4 раза - по количеству вершин
    finit
;вычисляем sin a и cos a;
    fld a                   ;включаем в стек угол
    fsin                    ;вычисляем sin a
    fxch                    ;меняем st(0)<->st(1)
    fcos                    ;вычисляем cos a
    fxch                    ;меняем st(0)<->st(1)
    fstp    a               ;выталкиваем a
    fstp    cos_a           ;выталкиваем cos_a
    fstp    sin_a           ;выталкиваем sin_a
;поворот изображения
;готовим xmm-регистр RXMM2 со значениями углов
    movlps  rxmm2,sin_a
    movhps  rxmm2,sin_a     ;RXMM2= cos_a sin_a cos_a sin_a
    movss   rxmm2,null
    subss   rxmm2,sin_a     ;RXMM2= cos_a sin_a cos_a -sin_a
cycl:   movlps  rxmm0,[esi] ;RXMM0= ? ? yi xi
    movhps  rxmm0,[esi]     ;RXMM0= yi xi yi xi
    shufps  rxmm0,rxmm0,0b0h;RXMM0= xi yi yi xi
    mulps   rxmm0,rxmm2     ;RXMM0=RXMM0*RXMM2=
                            ;= xi*cos_a yi*sin_a yi* cos_a xi*(-sin_a)
    shufps  rxmm1,rxmm0,31h ;RXMM1=? xi*cos_a ? yi* cos_a
    addps   rxmm0,rxmm1     ;RXMM0= ? (xi*cos_a+yi*sin_a) ?
                            ;(yi* cos_a+xi*(-sin_a))
    shufps  rxmm0,rxmm0,2   ;RXMM0= ? ? (yi* cos_a+xi*(-sin_a))
                            ;(xi*cos_a+yi*sin_a)
;сохраняем результат:
    movlps  [esi],rxmm0
;готовимся к вычислению нового положения для следующей координаты
    add     esi,8
    loopcycl
;...
```

На этом мы закончим рассмотрение примеров программирования XMM-расширения. При разработке приведенных ранее программ мы считали, что используемый транслятор ассемблера поддерживает любые команды процессора Intel, в том числе и XMM-команды. Реально ситуация далека от этой идеальной картины. Мы уже упоминали, что трансляторы ассемблера не всегда успевают за процессом развития системы команд. Особенно это касается транслятора TASM, который «брошен на произвол судьбы» фирмой Borland (Inprize), и на сегодняшний день его судьба выглядит довольно неопределенной. Другие фирмы-разработчики

трансляторов ассемблера мы не рассматриваем (не потому, что они хуже, — просто обсуждение достоинств и недостатков трансляторов ассемблера не является предметом данной книги). Поэтому на практике всегда можно столкнуться с ситуацией, когда любимый и хорошо знакомый транслятор не знает системы команд новейшего процессора. Как быть в этой ситуации? Как, не меняя транслятора, помочь ему понять неизвестные команды процессора?

Моделирование команд XMM-расширения

Задачу адаптации компилятора (TASM) к новым командам процессора и, в частности, к MMX-командам мы уже решали. Приступая к практическому использованию XMM-команд, вы снова столкнетесь с этой проблемой. Задачу адаптации TASM (или другого компилятора ассемблера) к командам XMM-расширения можно решить двумя способами:

- Во-первых, можно разработать включаемый файл, в котором для каждой XMM-команды реализовать макрокоманду, моделирующую на базе существующих команд нужную XMM-команду. Традиционно, фирма Intel, зная об инерционности процесса разработки новых версий трансляторов ассемблера, вместе с подмножеством новых команд разрабатывает соответствующий включаемый файл для их поддержки в ассемблерных программах. Для подмножества XMM-команд такой файл называется iaxmm.inc. Он ориентирован на транслятор MASM (фирмы Microsoft) и не пригоден (требует доработки) для TASM. Однако при доработке TASM необходимо иметь в виду вопрос об авторских правах. Некоторые проблемы использования файла iaxmm.inc совместно с TASM обсуждаются далее.
- Во-вторых, можно разработать программу-препроцессор, на вход которой подавать исходный файл с программой на ассемблере, содержащей новые команды процессора, а на выходе получать текст, адаптированный для компиляции старым транслятором ассемблера. Этот путь имеет то преимущество, что теперь при появлении новых команд можно, не внося больших корректив в технологию разработки программ, всего лишь определенным образом модифицировать файл-препроцессор, дополнив его возможностями обработки новых команд процессора. Более того, дополнив препроцессор средствами распознавания процессора (Intel или AMD), можно разрабатывать программы с использованием расширения 3DNow!. Рабочий материал для реализации этого способа вы можете найти среди файлов к книге в каталоге этой главы.

Поддержка XMM-команд в файле iaxmm.inc

Текст файла iaxmm.inc можно загрузить с официального сайта компании Intel (http://www.intel.com). С точки зрения структуры, включаемый файл iaxmm.inc представляет собой набор макрокоманд двух типов — основных и вспомогательных. Названия основных макрокоманд полностью совпадают с названиями XMM-команд, моделирование которых они обеспечивают. Вспомогательные макрокоманды расположены в начале файла и служат для обеспечения работы основных макрокоманд. В частности, эти макрокоманды устанавливают тип операндов, указанных при обращении к основной макрокоманде, причем делают это, исходя из режима

функционирования транслятора — 16- или 32-разрядного. Другое важное действие — установление соответствия между названиями XMM-регистров и регистров общего назначения. Дело в том, что для моделирования XMM-команд в 16- или 32-разрядном режиме работы ассемблера используются разные регистры общего назначения — 16-разрядные регистры в 16-разрядном режиме и 32-разрядные в 32-разрядном режиме.

Рассмотрим процесс моделирования XMM-команд. В качестве основы для моделирования выступает одна из команд основного процессора. Эта команда должна удовлетворять определенным требованиям. Каковы они? В поисках ответа посмотрим на машинные коды XMM-команд приложении А. Видно, что общими у них являются два момента:

- поле кода операции XMM-команд состоит из двух или трех байтов, один из которых равен 0fh;
- большинство XMM-команд используют форматы адресации с байтами modR/M и sib и, соответственно, допускают сочетание операндов как обычных двухоперандных команд целочисленного устройства — регистр — регистр или память-регистр.

Для моделирования XMM-команд нужно подобрать такую команду основного процессора, которая удовлетворяет этим двум условиям. Во включаемом файле iaxmm.inc в качестве таких команды присутствуют две — CMPXCHG и ADD. В процессе моделирования на место нужного байта кода операции этих команд помещаются байты со значениями кода операции соответствующей XMM-команды. Когда процессор «видит», что очередная команда является XMM-командой, то он начинает трактовать коды регистров в машинной команде как коды XMM-регистров и ссылки на память, размерностью соответствующей данной команде. В машинном формате команды нет символических названий регистров, которыми мы пользуемся при написании исходного текста программы, например AX или BX. В этом формате они определенным образом кодируются. Например, регистр AX кодируется в поле reg машинной команды как 000. Если заменить код операции команды, в которой одним из операндов является регистр AX, кодом операции некоторой XMM-команды, то это же значение в поле reg процессор будет трактовать как регистр rxmm0. Таким образом, в XMM-командах коды регистров воспринимаются соответственно коду операции. В табл. 18.9 приведены коды регистров общего назначения и соответствующих им XMM-регистров. В правом столбце этой таблицы содержится условное обозначение XMM-регистров, принятое в файле iaxmm.inc. Это же соответствие закреплено рядом определений в этом файле, которые иллюстрирует следующая программа:

```
DefineXMMxRegs Macro
IFDEF APP_16BIT
    rxmm0   TEXTEQU <AX>
    rxmm1   TEXTEQU <CX>
    rxmm2   TEXTEQU <DX>
    rxmm3   TEXTEQU <BX>
    rxmm4   TEXTEQU <SP>
    rxmm5   TEXTEQU <BP>
    rxmm6   TEXTEQU <SI>
    rxmm7   TEXTEQU <DI>
    RXMM0   TEXTEQU <AX>
```

```
    RXMM1   TEXTEQU <CX>
    RXMM2   TEXTEQU <DX>
    RXMM3   TEXTEQU <BX>
    RXMM4   TEXTEQU <SP>
    RXMM5   TEXTEQU <BP>
    RXMM6   TEXTEQU <SI>
    RXMM7   TEXTEQU <DI>
ELSE
    rxmm0   TEXTEQU <EAX>
    rxmm1   TEXTEQU <ECX>
    rxmm2   TEXTEQU <EDX>
    rxmm3   TEXTEQU <EBX>
    rxmm4   TEXTEQU <ESP>
    rxmm5   TEXTEQU <EBP>
    rxmm6   TEXTEQU <ESI>
    rxmm7   TEXTEQU <EDI>
    RXMM0   TEXTEQU <EAX>
    RXMM1   TEXTEQU <ECX>
    RXMM2   TEXTEQU <EDX>
    RXMM3   TEXTEQU <EBX>
    RXMM4   TEXTEQU <ESP>
    RXMM5   TEXTEQU <EBP>
    RXMM6   TEXTEQU <ESI>
    RXMM7   TEXTEQU <EDI>
ENDIF
endm
```

Таблица 18.9. Кодировка регистров в машинном коде команды

Код в поле reg	Регистр целочисленного устройства	XMM-регистр
000	AX/EAX	RXMM0
001	CX/ECX	RXMM1
010	DX/EDX	RXMM2
011	BX/EBX	RXMM3
100	SP/ESP	RXMM4
101	BP/EBP	RXMM5
110	SI/ESI	RXMM6
111	DI/EDI	RXMM7

Теперь в исходном тексте программы можно использовать символические имена XMM-регистров в качестве аргументов макрокоманд, моделирующих XMM-команды.

Рассмотрим, как в файле iammx.inc описано макроопределение для моделирования XMM-команды скалярной пересылки MOVSS.

```
;F3 0F 10 /r movss xmm1, xmm2/m32
;F3 0F 11 /r movss xmm2/m32, xmm1
movss   macro   dst:req, src:req
    XMMld_st_f3 opc_Movss, dst, src
endm
```

Понимание структуры приведенного макроопределения не должно вызвать у вас трудностей. Начать следует с того, что данная команда содержит вложенный вызов макрокоманды XMMld_st_f3, у которой две задачи — определить вариант сочетания операндов, после чего сформировать правильный код операции и подставить его на место соответствующих байтов в команде CMPXCHG. В результате этих действий команда CMPXCHG «превращается» в XMM-команду MOVSS:

```
<1> XMMld_st_f3 macro   op:req,dst:req, src:req
<2>     local   x, y
<3> DefineXMMxRegs
<4> IF (OPATTR(dst)) AND 00010000y  ;register
<5> x:  lock cmpxchg src, dst
<6> y:  org x
<7>     byte 0F3H,0Fh, op&_ld
<8> org y
<9> ELSE
<10>    x:  lock    cmpxchg dst, src
<11>    y:  org     x
<12>        byte    0F3H,0Fh, op&_st
<13>    org y
<14>    ENDIF
<15>    UnDefineXMMxRegs
<16>    endm
```

Центральное место в макроопределении XMMld_st_f3 занимают команда целочисленного устройства (в данном случае — CMPXCHG) и директива ORG. Первое действие данной макрокоманды — выяснить тип операнда приемника (dst) в макрокоманде MOVSS, так как он может быть и регистром, и ячейкой памяти. Это необходимо для правильного определения кода операции, которая будет управлять направлением потока данных. После того как определен приемник данных, с помощью условного перехода осуществляется переход на ветвь программы, где будет выполняться собственно формирование соответствующего XMM-команде MOVSS кода операции.

Формирование кода операции XMM-команды MOVSS производится с помощью директивы ORG, которая предназначена для изменения значения счетчика адреса. В строках 6 или 11 директива ORG устанавливает значение счетчика адреса равным адресу метки x. Адрес метки x является адресом первого байта машинного кода команды CMPXCHG. Директива DB в следующих строках размещает по этому адресу байтовые значения 0F3H,0Fh, op&_ld или 0F3H,0Fh, op&_st в зависимости от того, какое действие производится — загрузка (_ld) или сохранение (_st). Значение opc_Movss, с помощью которого формируются значения op&_st и op&_ld, определены в начале файла iaxmm.inc:

```
;...
opc_Movss_ld = 010H
opc_Movss_st = 011H
;...
```

Интерес представляет еще один характерный момент. Он касается порядка следования операндов в команде и роли бита d второго байта кода операции машинной команды. Мы обсуждали этот момент для моделирования команд MMX-расширения. Для XMM-команд рассуждения аналогичны, за исключением команды, на базе которой проводится моделирование, CMPXCHG. Например, значение второго байта кода операции opc_Movss_st=011H (0001 0001b). Он имеет значение бита d = 1, то есть данные передаются из регистра в память (процессор — память). Это нам и позволило в команде CMPXCHG изменить порядок следования операндов, в противном случае транслятор ассемблера команду не пропустит. Но команда MOVSS позволяет также производить передачу и в обратном направлении — память — процессор. Для того чтобы правильно смоделировать машинное представление XMM-команды, необходимо определить, каким объектом является операнд dest (приемник) — ячейкой памяти или регистром. После этого будет ясен тип второго

операнда, так как реализованы могут быть лишь две схемы расположения операндов: регистр — регистр и память — регистр. Для выяснения типа операнда ассемблер MASM (не TASM) предоставляет оператор OPATTR, который имеет следующий синтаксис:

```
OPATTR (выражение)
```

Его эквивалент — другой оператор, работающий и в TASM:

```
.type выражение
```

Эти операторы в зависимости от типа операнда, указанного в качестве операнда выражение, возвращают байтовые значения, которые приведены в табл. 18.3. Анализ возвращаемых значений производится строкой

```
IF (OPATTR(dst)) AND 00010000y  ;register
```

Если операнд dest — регистр, то формируется один код операции, если нет, то — другой.

Адреса операндов в памяти формируются таким же образом, как и для целочисленных команд, а их трактовка зависит от конкретной XMM-команды.

Модельно-зависимые регистры

Физически модельно-зависимые средства процессора представляют собой набор модельно-зависимых регистров (Model Specific Registers, MSR). Наличие и назначение этих регистров, по определению, зависит от конкретной модели процессора, что, в свою очередь, не гарантирует их поддержку будущими процессорами архитектуры Intel. Некоторые наиболее удачные из этих средств становятся стандартными для будущих моделей, а некоторые из них могут и не прижиться либо включаться лишь в архитектуру отдельных моделей процессоров для поддержки специфических условий их эксплуатации. Один из модельно-зависимых регистров, ставшим, если можно так выразиться, стандартом де-факто, будет рассмотрен в практической части данного раздела.

MSR-регистры обеспечивают управление различными аппаратно- и программно-зависимыми средствами, включая:

- счетчики мониторинга производительности;
- расширения отладки;
- средства поддержки исключения машинной ошибки и архитектуры машинного контроля (MCA);
- регистры MTRR для поддержки расширенных средств кэширования.

Чтение и запись в MSR-регистры осуществляются командами RDMSR и WRMSR. Большинство MSR-регистров инициализируются при программной инициализации процессора, многие из них можно впоследствии установить в соответствии с конкретными потребностями программы. Приведем краткое описание команд RDMSR, WRMSR и CPUID.

Команды RDMSR и WRMSR

Команда RDMSR (ReaD from Model Specific Register) выполняет чтение из MSR-регистра. Действие команды заключается в проверке двух условий: во-первых,

проверяется наличие нулевого уровня привилегированности кода, во-вторых, проверяется наличие в регистре ECX значения, адресующего один из MSR-регистров. Если хотя бы одно из этих условий не удовлетворяется, то выполнение команды RDMSR заканчивается. Если удовлетворяются оба условия, то значение MSR-регистра, адресуемого содержимым регистра ECX, помещается в пару 32-разрядных регистров EDX:EAX.

Команда WRMSR (WRite to Model Specific Register) производит запись значения в один из 64-разрядных MSR-регистров. Действие команды заключается в проверке тех же двух условий: во-первых, проверяется наличие нулевого уровня привилегированности кода, во-вторых, проверяется наличие в регистре ECX значения, адресующего один из MSR-регистров. Если хотя бы одно из этих условий не удовлетворяется, то работа команды заканчивается. Если удовлетворяются оба условия, то значение пары 32-разрядных регистров EDX:EAX пересылается в 64-разрядный MSR-регистр, номер которого задан в регистре ECX.

Рассмотрим подробнее счетчик меток реального времени (Time Stamp Counter, TSC), который представляет собой одно из средств мониторинга производительности.

Счетчик меток реального времени TSC

В этом разделе мы рассмотрим один из подходов к решению проблемы измерения скорости работы программ. Интерес к данной теме у программистов всегда повышен и подобен интересу рыболовов к размерам выловленной рыбы. Когда программист разрабатывает алгоритм реализации некоторой задачи, то он обязательно пытается оценить скорость, с которой будет работать программа по этому алгоритму. В учебнике нередко для решения какой-либо задачи предлагается несколько способов ее программной реализации, но при этом оставляется открытым вопрос об оценке их временной эффективности. В практической части этого раздела попытаемся ликвидировать этот пробел. Основное внимание мы уделим некоторым средствам процессора Intel, предназначенным для оценки эффективности функционирующих на них программ. В значительной степени рассматриваемый далее материал также применим и к процессорам AMD, так как эти процессоры тоже имеют подобные средства.

Счетчик TSC — это регистр, содержимое которого инкрементируется с каждым тактом процессорного ядра. Каждый раз при аппаратном сбросе (сигналом RESET) отсчет в этом счетчике начинается с нуля. Разрядность регистра обеспечивает счет без переполнения в течение сотен лет. Счетчик продолжает счет как при исполнении инструкции HLT, так и при остановке процессора по сигналу STPCLK# (для энергосбережения). Чтение счетчика обеспечивает инструкция RDTSC, установкой бита CR4.TSD ее можно сделать привилегированной (доступной лишь при CPL = 0). Чтение и запись TSC возможны также по инструкциям обращения к MSR (при CPL = 0), причем запись может выполняться только в младшие 32 бита, а старшие биты при операции записи обнуляются. Присутствие счетчика TSC определяется по инструкции CPUID (EAX = 1). Если в результате ее вызова бит 4 регистра EDX равен 1, то процессор поддерживает счетчик меток реального времени.

Команда RDTSC (ReaD from Time Stamp Counter — чтение 64-разрядного счетчика меток реального времени) не имеет операндов. Машинный код этой коман-

ды — 0f 31. Команда проверяет состояние второго бита регистра CR4.TSD (Time Stamp Disable — отключить счетчик меток реального времени):

- если CR4.TSD = 0, то выполнение команды RDTSC разрешается на любом уровне привилегий;
- если CR4.TSD = 1, то выполнение команды RDTSC разрешается только на нулевом уровне привилегий.

Если после данной проверки выясняется, что на текущем уровне привилегий выполнение команды разрешено, значения 64-разрядного MSR-счетчика TSC сохраняется в паре 32-разрядных регистров EDX:EAX. В противном случае работа команды заканчивается.

Разработаем две макрокоманды, которые позволят нам получать количественную оценку эффективности кода, начиная от полной программы и заканчивая отдельной командой. Эти макрокоманды не будут отличаться компактностью, но этого от них и не требуется, так как они являются лишь средством оценки эффективности, используемым на этапе отладки, в конечном коде их наличие не требуется.

На взгляд автора, применение этих макрокоманд должно подчиняться следующему алгоритму. Вызов первой макрокоманды, назовем ее profiler_in, должен зафиксировать момент, относительно которого будет производиться отсчет тактов процессора, то есть в начале профилируемого участка программы. Вызов второй макрокоманды profiler_out должен зафиксировать момент окончания работы на этом участке программы. Необходимо иметь в виду, что это «грязное» время работы программы, по которому можно производить только приблизительную оценку ее скорости работы. Для этого есть внутренняя и внешние причины. Внутренняя причина заключается в том, что полученное значение включает время, затраченное на работу некоторых команд, составляющих тело самой макрокоманды. Этот недостаток исправить легко. Что касается внешних причин, то они объективны по отношению к программе пользователя и мешают получению истинного времени профилирования. В качестве таких внешних причин могут быть программы операционной системы, которые могут приостанавливать на время программу пользователя. Внешней причиной является и отладчик, так как при работе в нем вообще нет смысла в оценке скорости.

Далее приведены макрокоманды profiler_in и profiler_out (листинг 18.14) с тестовым примером для проверки их работы. Данные макрокоманды производят корректировку результата своей работы с тем, чтобы исключить рассмотренные ранее внутренние причины «грязного» времени работы программы. Заметим, что не всякий транслятор ассемблера «знает» о новых командах процессора, в том числе и о команде RDTSC. По этой причине мы ее моделируем, инициализируя в сегменте кода 2 байта значениями машинного кода этой команды db 0fh,31h.

Листинг 18.14. Оценка производительности кода с помощью макрокоманд profiler_in и profiler_out

```
;prg18_7.asm
;...
bin_dec_fpu macro   string_bin_qword:REQ, string_pack:REQ,
adr_string_pack:REQ, len_string_pack:REQ, adr_string:REQ, len_string:REQ
local   cycl
```

```
;-----------------------------------------------------------
;макрокоманда вывода на консоль десятичного числа:
;на входе:
;string_bin_qword — адрес 64-разрядной ячейки (описанной dt) с преобразуемым
;двоичным целым числом
;string_pack - адрес 80-разрядной ячейки (описанной dt),
;в которой сохраняется упакованное десятичное значение
;adr_string_pack - ячейка с адресом string_pack (описанной dd)
;len_string_pack - длина string_pack
;adr_string - ячейка с адресом string (описанной dd). В string помещаются
;символы десятичных цифр для вывода.
;len_string - размер string (18 байт)
;---------преобразуем bin->dec ----------------------------
    finit
    fild string_bin_qword   ;заносим в сопроцессор двоичное целое число
    fbstp   string_pack     ;извлекаем упакованное десятичное
;---------распакуем---------------------------------------
    lds si,adr_string_pack
    add si,len_string_pack-2;на конец string_pack (18 упакован. десят. цифр)
    les di,adr_string
    mov cx,9                ;9 пар упакованных десятичных цифр
    xor ax,ax
cycl:   xor ax,ax
    std                     ;string_pack обрабатываем с конца
    lodsb                   ;в al очередные 2 упакованные десятичные цифры
;распаковываем - ah=младшая, al=старшая
    shl     ax,4
    rol     al,4
    or      ax,3030h;преобразуем в символьное представление
    xchg ah,al          ;ah=младшая, al=старшая
    cld                 ;в string записываем с начала
    stosw
    loop cycl
;---------выводим на консоль------------------------------
    mov     bx,1        ;стандартный дескриптор — экран
    mov     cx,len_string
    lds     dx,adr_string   ;формируем указатель на строку string
    mov     ah,40h      ;номер функции DOS
    int     21h         ;выводим
    jc      exit        ;переход в случае ошибки
    endm
profiler_in macro   val_1:REQ
;val_1 — ячейка памяти 64 бита (2*32) для сохранения момента
;начала профилирования ("грязного")
pushad                  ;сохранение всех регистров общего назначения в стеке
db  0fh,31h ;RDTSC
    mov val_1+4,edx
    mov val_1,eax
popad           ;восстановление всех регистров общего назначения из стека
    endm
profiler_out    macro   val_1:REQ, val_2:REQ
;val_1 - ячейка памяти 64 бита (2*32), в которой при входе в макрос сохранен
;момент начала профилирования командой profiler_in. Далее в макросе эта
;ячейка содержит результат профилирования - число тактов процессора
;val_2 - ячейка памяти 64 бита (2*32), в которой сохраняется момент
;окончания профилирования ("грязный")
pushad          ;сохранение всех регистров общего назначения в стеке
db  0fh,31h     ;RDTSC - окончание профилирования
    mov val_2+4,edx
    mov val_2,eax
;профилируем pushad и popad с учетом двух shrd
pushad
popad
db  0fh,31h ;RDTSC
```

продолжение ⇨

Листинг 18.14 *(продолжение)*

```
;теперь необходимо получить чистое время профилирования, для чего результат
;необходимо скорректировать (уменьшить) на количество тактов процессора,
;потребное для выполнения пар команд PUSHAD\POPAD и MOV
    sub eax, val_2
    sbb edx, val_2+4    ;в edx:eax кол-ва тактов для выполнения lde[ команд
                        ;PUSHAD\POPAD и lde[ SHRD
    sub val_2, eax
    sbb val_2+4, edx    ;собственно корректировка
    mov eax, val_1
    sub val_2, eax
    mov eax,val_1+4
sbb val_2+4, eax        ;в val_2:val_2+4 - чистое количество тактов
                        ;процессора для выполнения
                        ;профилируемого участка
popad   ;восстановление всех регистров общего назначения из стека
;выводим
bin_dec_fpu val_2_q, string_pack, adr_string_pack, len_string_pack,
    adr_string, len_string
    endm
.data
val_2       label   dword
val_2_q     dq      0
val_1       label   dword
            dq      0
;в string_pack исходное значение из val_2_q в упакованном десятичном формате
string_pack     dt  0
len_string_pack=$-string_pack
adr_string_pack dd  string_pack
string              db  18 dup (0)  ;максимальный результат состоит
                                    ;из 18 десятичных цифр
len_string=$-string
adr_string          dd  string
.code
;...
;профилируем выполнение команд работы со стеком
    profiler_in val_1
push    eax
pop eax
    profiler_outval_1, val_2
exit:       ;выход из программы
;...
```

Составьте характерные примеры и протестируйте работу данной программы. Обратите внимание на то, что при задании пустой последовательности команд между парой макросов profiler_in и profiler_out все равно получается некоторая величина профилирования. Она постоянна, ее источник — сами команды RDTSC, которые требуют тактов процессора для своего исполнения. Это значение можно скорректировать разными способами, но можно и не трогать, а учитывать при подведении окончательных результатов тестирования нужного вам фрагмента кода. На компьютере автора это значение равно 2.

Заметьте, что данная программа имеет еще одно средство, которое может пригодиться вам в практической работе, — это макрокоманда bin_dec_fpu. Она разработана для визуализации результатов профилирования и выполняет перевод значения из двоичной системы счисления в десятичную. Ее можно включить в персональную библиотеку макрокоманд.

Итоги

- Процессоры постоянно совершенствуются, поэтому, если каждое нововведение отмечать увеличением номера модели, то, наверное, сейчас на рынке были процессоры Pentium C (сто) или Pentium D (пятьсот). По этой причине номера процессоров, так же как и номера выпусков программ, меняются при достаточно кардинальных изменениях архитектуры. Поэтому на данном этапе есть смысл рассматривать в архитектуре процессоров некоторый базовый слой и различные расширения этого базового слоя.
- Свойства конкретного процессора можно узнать с помощью команды `CPUID`.
- К расширениям базовой модели на данном этапе развития IA-32 можно отнести MMX- и XMM-расширения, а также пространство модельно-зависимых регистров MSR.
- В основе MMX-технологии лежит принцип SIMD — Single Instruction Multiple Data. Согласно этому принципу, обработка нескольких одинаковых по размеру элементов данных производится параллельно одной командой.
- MMX-расширение поддерживает несколько новых типов данных, представляющих собой совокупность упакованных байтов, слов, двойных слов и одно учетверенное слово. Эти типы данных имеют общий размер 64 бита, что согласуется с размером шины данных процессора Pentium.
- Аппаратно MMX-расширение реализовано в виде восьми регистров и дополнительного набора из 57 машинных команд. MMX-регистры имеют размер 64 бита и физически соответствуют регистрам сопроцессора.
- Важное значение имеет особенность формирования результатов работы MMX-команд при возникновении ситуации переполнения. Команды целочисленного устройства реагируют на нее в соответствии с принципом циклического переполнения. Среди MMX-команд также есть команды, работающие в соответствии с этим принципом, но большинство MMX-команд используют принцип знакового и беззнакового насыщения.
- Далеко не все трансляторы ассемблера поддерживают мнемоническое обозначение MMX-команд, поэтому их необходимо уметь моделировать. При моделировании важно учитывать, в какой среде (16- или 32-разрядной) будет работать разрабатываемое приложение.
- Для отладки программ, использующих MMX-расширение, можно применять обычные отладчики, главное требование к которым заключается в том, чтобы они поддерживали работу с сопроцессором. Если соблюдать (и, что немаловажно, понимать) определенные ограничения при работе с отладчиком, то особых проблем, как правило, не возникает. Что касается отладчика Turbo Debuger, то необходимо, чтобы во время работы с MMX-командами активным было окно `CPU` отладчика.
- Если вы разрабатываете приложение с использованием MMX-расширения для коммерческого (или некоммерческого) распространения, вам необходимо предусмотреть ситуацию, когда компьютер, на котором ваше приложение будет работать, не поддерживает MMX-команд. Для этого вы должны предваритель-

но, используя команду CPUID, установить этот факт, после чего передать управление на соответствующую ветвь программы, содержащую MMX-команды или не содержащую этих команд. Главное, чтобы эти две ветви реализовывали одинаковые функции.

- В дополнение к целочисленному MMX-расширению фирма Intel разработала MMX-расширение с плавающей точкой (XMM-расширение). Соответствующий аппаратный модуль SSE и система команд к нему были включены в архитектуру процессора Pentium III.
- Аппаратная часть SSE-модуля представляет собой восемь 128-разрядных регистров и один регистр управления.
- Новое расширение оперирует с данными в формате XMM, представляющими собой четыре упакованных вещественных числа в коротком формате с плавающей точкой.
- Система команд XMM-расширения обладает большей гибкостью по сравнению с MMX-расширением. В нее введены, кроме традиционных команд работы с данными, средства для различных перемещений упакованных элементов.
- В системе команд XMM-расширения есть команда деления, которой не было в MMX-расширении. Также XMM-расширение содержит команды взаимного конвертирования данных для трех форматов — формата XMM, формата MMX и обычного целочисленного двоичного формата.
- В систему команд Pentium III введены команды, которые позволяют управлять кэшированием данных. Как правило, массивы мультимедийных данных большие, и для работы с ними просто нет необходимости использовать механизм кэширования. Команды группы управления кэшированием позволяют производить работу с данными в памяти в обход механизма кэширования.
- В систему команд Pentium III также включены дополнительные команды для целочисленного MMX-расширения.
- MSR-регистры позволяют использовать в интересах программы специфические для конкретного процессора свойства. Некоторые из этих свойств становятся стандартными для всех последующих процессоров, и тогда есть смысл говорить об очередном расширении базовой архитектуры процессора. Более подробную информацию об MSR-регистрах можно получить в [13].