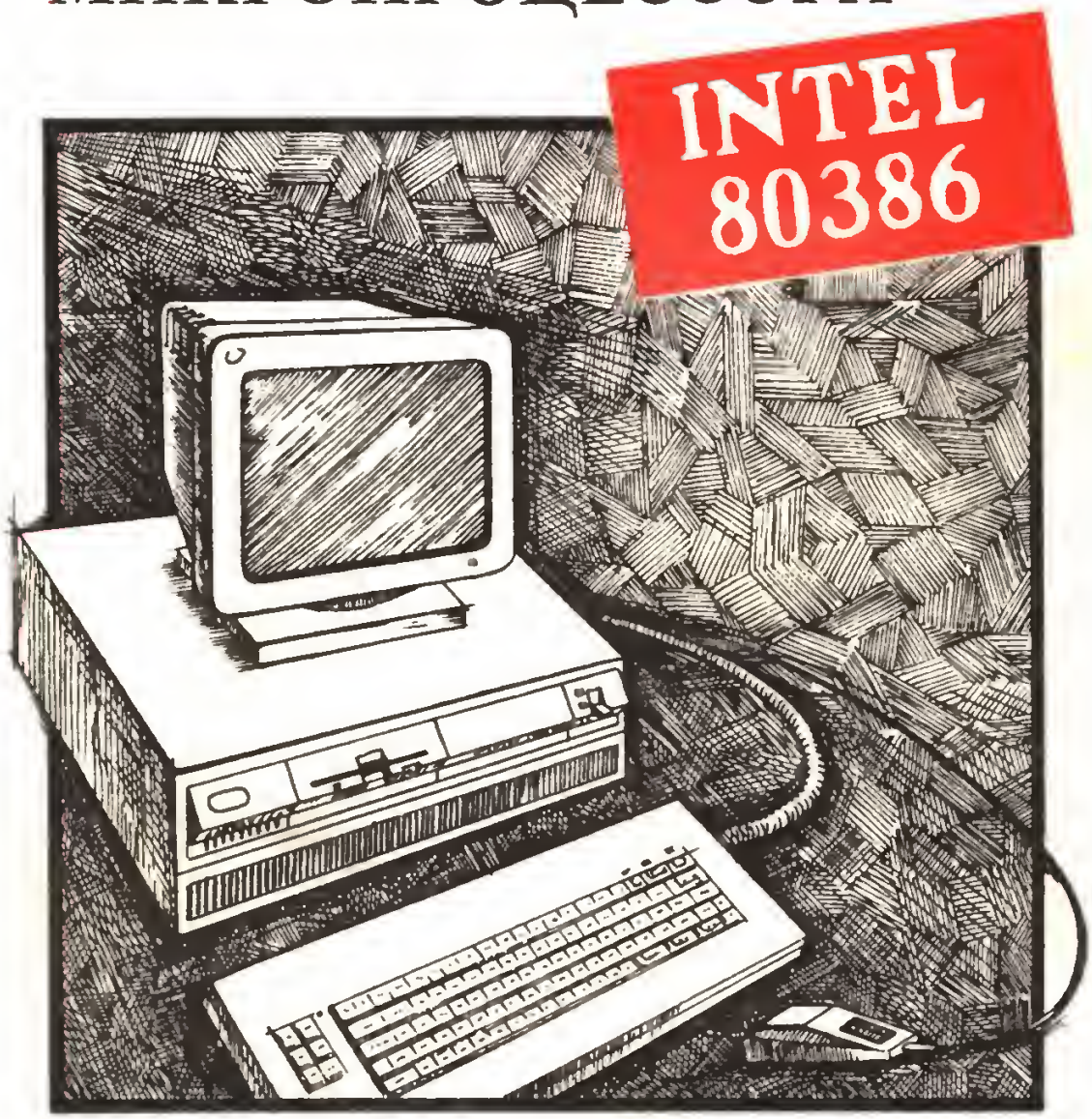
Б. Э. Смит М. Т. Джонсон

АРХИТЕКТУРА И ПРОГРАММИРОВАНИЕ МИКРОПРОЦЕССОРА



Б. Э. Смит, М. Т. Джонсон

АРХИТЕКТУРА И ПРОГРАММИРОВАНИЕ МИКРОПРОЦЕССОРА INTEL 80386

B. E. Smith, M. T. Johnson

PROGRAMMING THE INTEL 80386

London
Scott, Foresman and Company
1987

Б. Э. Смит, М. Т. Джонсон

АРХИТЕКТУРА И ПРОГРАММИРОВАНИЕ МИКРОПРОЦЕССОРА INTEL 80386

Перевод с английского
В. Л. Григорьева

Под редакцией
А. С. Карнаухова

Москва
ТОО "Конкорд"
1992

ББК 32.97
С-50

Литературное редактирование *Е. В. Васильевской*
Набор и компьютерная верстка *Н. И. Сергеевой*

Смит Б. Э., Джонсон М. Т.
С-50 Архитектура и программирование микропроцессора INTEL 80386/Пер. с англ. В. Л. Григорьева.— М.: Конкорд, 1992.— с. 334, ил.

ISBN 5-87737-004-9

Перевод на русский язык книги английских авторов по программированию микропроцессора 80386 фирмы Intel способствует продвижению на рынок России новых информационных технологий и современных технических средств.

Книга интересна тем, что она содержит полный комплекс инструкций по программированию процессора 80386 на языке ассемблер.

Издание рассчитано на специалистов.

С 2404010000-004
87737-92

ББК 32.97

Подписано а печать 05.12.92. Сдано в набор 20.03.92. Формат 60×88 1/16. Бумага офсетная. Гарнитура «Таймс-Роман». Печать офсетная. Тираж 25 000 экз. Зак. 2046. Издательстао ТОО «Конкорд», 125493, Москва, ул. Авангардная, д. 4, кв. 63.
Московская тнпографня 4 Министерства печатн н ннформацин РФ 129041 Москаа, Б. Переяслаская, 46.

Издание подготовлено при участии МП «Бином»

ISBN 5-87737-004-9

ОГЛАВЛЕНИЕ

Список рисунков

Список таблиц

Список листингов

От научного редактора перевода

Микропроцессоры фирмы Intel в значительной степени определяют направление развития компьютерной техники в последнее десятилетие. Каждые несколько лет фирма Intel демонстрирует новые прорывы в своей технологии, существенно меняя наши представления о возможностях компьютеризации.

Предлагаемая читателю книга представляет собой последовательное изложение архитектурных особенностей и тонкостей программирования микропроцессора i80386, пришедшего на смену наиболее распространенному сегодня для класса персональных компьютеров процессора i80286.

Новый процессор фирмы Intel представляет собой не только очередной шаг в повышении производительности микропроцессоров и степени их интеграции, но и содержит целый ряд принципиально новых подходов, реализованных в его архитектуре.

Профессиональные разработчики программ для многопроцессорных систем и систем с повышенными требованиями к надежности и защите программ найдут в этой книге самое подробное изложение и примеры реализации соответствующих механизмов i80386.

В то же время, первые главы книги дают тот необходимый минимум вводной информации, который позволит понять весь последующий материал и читателю, не имеющему специальной профессиональной подготовки.

В настоящее время фирма Intel начала следующий этап наступления в повышении мощности своих микропроцессоров.

Процессор i80486 и следующие за ним i80586, i80686 (P5 и P6) обещают гораздо

большие возможности для программистов и проектировщиков систем.

Изучение особенностей архитектуры и программирования i80386 явится необходимым шагом, который в дальнейшем существенно упростит понимание новых идей программирования, закладываемых в перспективные изделия фирмы Intel.

А. С. Карнаухов

Глава 1

ВВЕДЕНИЕ В ЯЗЫК АССЕМБЛЕР

С микропроцессором 80386 могут работать программисты самого различного уровня. Некоторые из них являются системными программистами, имеющими достаточные навыки в области написания операционных систем, драйверов устройств и больших ассемблерных программ. Другие больше знакомы с языками высокого уровня типа Бейсик или Паскаль, но мало пользовались языком ассемблер. Эта глава познакомит читателей с наиболее важными понятиями в программировании на языке ассемблер.

Сначала рассматривается формат ассемблерного оператора, затем обсуждаются типы чисел, используемых при программировании на языке ассемблер. Большинство программистов знают их, но они потребуются в дальнейшем и потому приведены. Если вы раньше программировали микропроцессоры фирмы Intel, эту главу можно пропустить. Но если материал окажется для вас новинкой, его нужно внимательно изучить.

1.1. ЯЗЫК АССЕМБЛЕР

Команда пересылки MOV AX,BX является примером типичного ассемблерного оператора. Первое слово MOV является ассемблерной командой, т. е. прямым приказом от программиста компьютеру. Так что же здесь пересылается?

AX и BX являются регистрами, т. е. запоминающими ячейками внутри 80386. Число из регистра BX пере-

сылается в регистр AX. После выполнения этой команды в обоих регистрах будет находиться одно и то же значение.

Каждая непустая строка в ассемблерной программе содержит «команду» или «директиву», наиболее важными частями которой являются сама команда и ее операнды. Команда выполняет некоторую операцию, используя данные в своих операндах. Другие части команд и директив рассматриваются далее.

Например, команда MOV AX,BX имеет два операнда.

Большинство двухоперандных команд имеют формат вида:

```
КОМАНДА  Получатель  Источник
```

Так выглядят, например, арифметические команды сложения ADD или вычитания SUB. Команда берет свой второй операнд, выполняет некоторую операцию с привлечением первого операнда (например, их сложение) и сохраняет результат в первом операнде. Результат заменяет то значение, которое имелось ранее в первом операнде.

Многие команды имеют неявные операнды, например, команда CLC означает «сбросить бит переноса», т. е. сбросить флажок CF в ноль. Значение операнда (бит переноса) подразумевается самой командой. Операндами команд могут быть отдельные биты в регистрах микропроцессора, байты, слова и двойные слова или любые из этих элементов в основной памяти компьютера (подробнее о составе компьютера см. гл. 2).

Обычно компьютер выполняет команды в том порядке, в котором они следуют в программе. Но иногда такой порядок нужно изменить, например, чтобы повторить цикл. Для этого применяются следующие команды:

```
            JCC   Loop Top
            ...
            ...

Loop Top:   MOV   AX,BX      ; Верх цикла
            ...
            ...

            JZ    Loop Top
```

Выражеиие Loop Top является меткой команды MOV AX,BX. Точка с запятой в этой же строке указывает иа наличие коммеитария, который находится за ней. Многоточия замеияют операции, которые для нас здесь иитереса не представляют.

Команда JCC (переход, если переиос сброшеи) проверяет бит переноса и, если он содержит 0, передает управление команде MOV AX,BX, отмеченной меткой «Loop Top». Команда JZ (переход, если нуль) аналогична команде JCC, но проверяет бит нулевого результата.

Обобщеииый ассемблериый оператор имеет такую форму:

Метка: КОМАНДА Операнд(ы) ; Комментарий

Метка и комментарий не обязательны, а число операндов зависит от вида команды. Единственной обязательной частью оператора является сама команда. Если в строке содержится только метка, то она относится к команде в следующей строке.

Описаиия осиовных типов команд микропроцессора 80386 приведены в главе 3. Кроме команд и операндов в языке ассемблер есть иесколько других элементов, которые поясняются далее.

1.1.1. Префиксы команды

Префикс заставляет команду действовать иесколько иначе, чем она выполияется без префикса. Важным префиксом является LOCK, который помогает одному из процессоров 80386 обращаться к области памяти, разделяемой иескольким процессорами; еще один важный префикс REP вызывает повторение команды. Префиксы расширяют строку ассемблерной программы:

Метка: ПРЕФИКС КОМАНДА Операнд(ы); Коммеитарий

Примером служит команда:

One Loop: REP MOVSB Dest,Source ; Пересылает байты

Эта команда пересылает заданное число байт (число содержится в одном из регистров микропроцессора) из одной области памяти в другую.

1.1.2. Директивы ассемблера

Приведенные выше команды преобразуются в машинные команды для микропроцессора. Ассемблер точно определяет форму команды, чтобы она производила нужное программисту действие.

Иногда нам нужно сообщить ассемблеру заранее, что мы собираемся делать. Если, например, все наши элементы данных являются словами, а мы не хотим, чтобы компьютер пересылал двойные слова. Директивы ассемблера похожи на обычные команды, но не транслируются в машинный код, сообщающий микропроцессору, что делать дальше. Вместо этого директивы сообщают ассемблеру, как интерпретировать команды и директивы, следующие за ними, показывая, где начинается и кончается программа, и выполняя множество других функций.

Например, в любой программе есть переменные. Директивы, сообщающие ассемблеру имена переменных, имеют следующий вид:

```
Имя Директива Начальное значение ; Комментарий
```

Приведем пример:

```
MyAge    DB 29     ; Директива определения данных
DB 0               ; Второй байт со значением 0
```

Директива сообщает ассемблеру, что при обращении программы к переменной MyAge оно производится к байту и начальное значение байта равно 29. К байту, зарезервированному в следующей строке, можно обращаться как MyAge+1, что означает «байт после байта с MyAge». Как видно из приведенного примера, имя в начале строки директивы не обязательно.

Другой важной директивой ассемблера является EQU, которая сообщает ассемблеру о необходимости присвоить числу определенное имя. Например, следующая директива сообщает ассемблеру, что при встрече в программе слова Retire следует использовать число 62:

```
Retire EQU 62
```

Отметим, что директива DB сообщает ассемблеру о необходимости поместить определенное значение (29) в заданный байт памяти, а директива EQU требует не забывать заменять слово Retire на значение 62 при трансляции программы на машинный язык. При необходимости изменить значение Retire можно внести директиву:

```
Retire EQU 65
```

При повторном ассемблировании программы при каждой встрече слова Retire оно будет заменено на 65.

Подробную информацию о директивах можно получить из документации на используемый вами ассемблер, и ее нужно внимательно изучить. Директивы определяют, как программа размещается в памяти, и могут помочь придать программе структуру, напоминающую структуру программы на языке высокого уровня.

1.1.3. Ассемблерная арифметика

В командах и директивах ассемблера могут вычисляться значения выражений. Если ассемблер встречает выражение (например, RETIRE-3) там, где ожидается наличие числа, ассемблер выполняет необходимые арифметические операции и помещает результат в программу на машинном языке. Если, например, с помощью директивы EQU мы сообщаем, что Retire равно 65, то следующая команда передаст Retire+5 (т. е., 65+5=70) в регистр AX:

```
MOV AX, Retire+5
```

Не следует забывать, что определение значения «Retire+5» осуществляется при ассемблировании программы, поэтому в ассемблерной арифметике можно использовать только те числа, которые определены до ассемблирования данного оператора. Точное описание ассемблерной арифметики можно найти в документации на конкретную версию ассемблера.

1.1.4. Что делает язык ассемблер

Вы можете спросить: «Так в чем же разница между языком ассемблер и языком высокого уровня?» Хотя структура ассемблерных операторов своеобразна, но даже приведенные простые примеры показывают наличие в программе циклов, передачи управления, условных переходов и переменных, т. е. тех элементов, которые свойственны для языков высокого уровня.

Главная особенность языка ассемблер заключается в том, что каждая его команда транслируется ровно в одну команду машинного языка. В языках же высокого уровня каждый оператор может транслироваться в произвольное число команд машинного языка (иногда в одну, но часто в 5 и более в зависимости от вида оператора и контекста).

Вторая важная особенность ассемблера состоит в том, что программист может прямо именовать регистры микропроцессора и абсолютные адреса в памяти. Когда программист пишет на Бейсике LET A=B, он не знает, где находятся числа. Запись MOV AX,BX позволяет программисту точно знать, что он делает.

Исполнение команды MOV зависит от вида операндов. Главным является общее действие команды, при котором значение кода копируется из одного участка памяти в другой. В этой книге описаны общие действия команд и их точные форматы.

Часто говорят, что программирующий на ассемблере может делать все, что делает программист на языке высокого уровня и еще кое-что сверх того. Программист должен точно знать команды и структуры данных. Даже плохо написанная ассемблерная программа будет, в общем, выполняться быстрее хорошо написанной программы на языке высокого уровня, но только тщательно написанная ассемблерная программа приближается к программе на языке высокого уровня по читаемости (для людей) и легкости сопровождения.

1.1.5. Ассемблирование 80386

Посмотрим на листинг 1.1, чтобы более точно уяснить, что делает ассемблер с небольшим фрагментом ассемблерного кода.

Листинг 1.1. Пример ассемблерной программы

```
GenRegs  EQU 8                  ; 1: число общих регистров 80386
RegsUsed DB  0                  ; 2: число использованных регист-
                                ; ров
DoMore   ...                    ; Код, который фиксирует данные
         ...                    ; в общих регистрах и производит
         ...                    ; инкремент RegsUsed, когда ре-
         ...                    ; гистр используется
         CMP RegsUsed,GenRegs   ; 3: все регистры использованы?
         JNE DoMore             ; 4: если нет, продолжать цикл
```

Ассемблер транслирует код из листинга 1.1 на машинный язык процессора 80386, образуя исполняемый код. Строка 1 с директивой EQU сообщает ассемблеру о необходимости подставлять число 8 там, где встречается слово «GenRegs».

Строка 2 с директивой DB сообщает ассемблеру, что данный байт в памяти назван RegsUsed и имеет начальное значение 0.

Строка 3 содержит требование сравнить значение в RegsUsed, которое может быть изменено в предыдущих строках, с числом 8, представляющим GenRegs.

В результате обработки указанных директив ассемблер образует следующий машинный код в 16-ричном формате:

```
80  3E   ??   08
```

Это означает: сравнить (код инструкции 80) байт в указанной ячейке памяти (3E ??) с абсолютным значением числа (8, как определено директивой EQU). Флажок нулевого результата будет установлен в 1, если эти два числа равны.

Тонким моментом здесь является неопределенность значения смещения (??), которое задается режимом адресации, используемым для нахождения байта в памяти (см. гл. 4).

Строка 4 сообщает процессору о необходимости продолжить выполнение программы с метки DoMore, если инструкция CMP показывает, что два числа не равны. На машинном языке эта операция выглядит так:

75 XX

Здесь код 75 означает «короткий переход» к ячейке памяти в диапазоне -128/+127 байт, если условие «не равны» удовлетворяется (флажок нулевого результата в состоянии 0). Значение XX — это число байт до ячейки, которое равно числу байт между командой с меткой DoMore и командой JNE.

Листинг 1.2 показывает процесс ассемблирования; он содержит ассемблерный код и результат на машинном языке.

Листинг 1.2. Процесс ассемблирования

Язык ассемблер			Машинный язык
GenRegs	EQU	8	
RegsUsed	DB	0	0
DoMore	...		
	...		
	...		
	CMP	RegsUsed,GenRegs	80 3E ?? 08
	JNE	DoMore	75 XX

Далее в книге приводятся кодовые эквиваленты всех команд процессора 80386 и режимы адресации, а также листинги ассемблирования на машинном языке.

1.2. МАШИННЫЕ ЧИСЛА

1.2.1. Представление данных в компьютере

В самом компьютере применяется исключительно двоичная система счисления. Компьютер внутри можно представить набором переключателей, которые могут находиться в двух позициях 0/1, Вкл/Выкл, Истина/Ложь. Каждый такой переключатель называется битом. Для того, чтобы описать состояние сразу нескольких бит, их значения выписываются последовательно (1001, 10110, 10001001001).

Обычно биты организуются в группы по 8, называемые байтами. Последнее число можно записать как 00000100 01001001 и говорить о значениях в первом и во втором байте.

Более компактную форму обеспечивает 16-ричная запись, в которой каждая цифра показывает состояние группы из 4 бит. Цифры 16-ричной системы приведены в листинге 1.3.

Листинг 1.3. Двоичные, десятичные и 16-ричные цифры

Набор	Десятичный эквивалент	16-ричный эквивалент
Вес 8421		
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	10	A
1011	11	B
1100	12	C
1101	13	D
1110	14	E
1111	15	F

Байт можно представить двумя 16-ричными цифрами: 11000001 равен C1, а 10001001 равен 89. Обычно использование 16-ричной системы счисления отмечается завершающей буквой H.

Разряды двоичных чисел имеют веса, равные степени 2:

двоичное число	1 0 1 0 1 0 1 0
степень 2	7 6 5 4 3 2 1 0

Самый правый разряд имеет вес 1, следующий перед ним $2^1=2$ и т. д. до $2^7=128$.

1.2.2. Представление чисел

Для представления непривычных и неудобных для людей двоичных чисел применяется двоично-кодированный десятичный или BCD-формат. Процессор 80386 обеспечивает аппаратную поддержку BCD-арифметики командами «Коррекция ASCII», которые используют регистр AL для выполнения операций BCD-арифметики над двумя числами одновременно.

В «упакованном BCD-формате» 4 бита представляют одну цифру 0-9. Из 16 возможных комбинаций используются только десять. При попытке производства арифметических операций с такими числами встречаются трудности, так как компьютер автоматически считает их двоичными числами и иногда дает результат, не имеющий смысла для BCD-чисел, например, двоичная операция:

1000(8 BCD)+0100(4 BCD)=1100(? BCD)

Сумма равна 12, но такой десятичной цифры нет. Мы хотим, чтобы компьютер автоматически сформировал перенос, когда результат сложения двух BCD-цифр больше 9. BCD-операция:

1000(8 BCD)+0100(4 BCD)=0001 0010(12 BCD)

Необходимая коррекция осуществляется специальной командой. Отметим, что в байте могут храниться две BCD-цифры и диапазон представления чисел составляет от 00 до 99. В то же время байт может содержать 256 комбинаций; следовательно, в BCD-формате память используется не полностью.

BCD-формат допускает представление больших чисел несколькими байтами и применяется в финансовых расчетах, где нет чисел, состоящих из более, чем 15 разрядов.

Но для больших чисел требуется и формат с фиксированным числом байт. Это формат с плавающей точкой. Обычно операции над такими числами производятся с помощью заранее написанных программ, и вам необязательно знать тонкости работы с такими числами.

32-битное число с плавающей точкой может иметь следующий формат:

первый бит: знак мантиссы,
следующие 8 бит: порядок,
следующие 23 бита: мантисса.

Предполагается, что число представляется в форме: ((+/-) 1. мантисса)×($2^{(+/-\ порядок)}$)

Достоинство использования чисел с плавающей точкой состоит в том, что в 32 битах представляются числа из огромного диапазона: примерно от $10^{(-38)}$ до $10^{(+38)}$. Но представление с плавающей точкой оказывается неточным, и числа с плавающей точкой обычно являются округленными числами. Точность можно повысить, увеличивая число бит до 64 или 80, но все же потеря точности остается.

1.3. ДВОИЧНАЯ АРИФМЕТИКА

Сложение одноразрядных двоичных чисел подчиняется простым правилам: 0+0=0, 0+1=1, 1+0=1, 1+1=0 и перенос 1. Действуя по этим правилам, можно сложить два любых числа, начиная с младших разрядов. Сложность возникает только при учете переносов, но она легко преодолевается, например:

```
  1 1 0 0 0 1 1 0
+
    1 1 0 1 1 1 1
-----------------
1 0 0 1 1 0 1 0 1
```

Вычитание двоичных чисел несколько труднее сложения, таблица двоичного вычитания имеет вид:

```
0 - 0 = 0
0 - 1 = 1, заем 1
1 - 0 = 1
1 - 1 = 0
```

Применяя эти правила к отдельным разрядам, можно вычитать любые числа, начиная с младших разрядов. Необходимо правильно учитывать заемы, например:

```
  1 1 0 0 1 1 0
-
    1 1 0 0 0 1
---------------
    1 1 0 1 0 1
```

1.3.1. Отрицательные двоичные числа

До сих пор мы пользовались только положительными двоичными числами и применяли для их представления нужное нам число разрядов: старший бит всегда содержал 1, так как левые нули бесполезны. Но в большинстве систем требуется обрабатывать положительные и отрицательные числа, и старшая 1 всегда показывает отрицательное число. Для этого договорились о том, что длина всех чисел — 8 бит и необходимо записывать даже незначащие старшие нули.

Отрицательные числа в компьютерах и в микропроцессоре 80386 представляются в дополнительном коде, так как в нем простые правила применимы к положительным и отрицательным числам. В дополнительном коде запись положительных чисел совпадает с их обычной записью, например, 4 равно 00000100, а нуль равен 00000000.

Для образования дополнительного кода отрицательного числа необходимо сначала построить положительное число с таким же абсолютным значением, а затем инвертировать каждый бит и прибавить 1 к результату. Приведем два примера:

десятичное число	-4	-11
положительное двоичное число	00000100	00001011
инверсия каждого бита	11111011	11110100
прибавление 1	11111100	11110101

Отметим, что старший бит является знаковым. Для преобразования отрицательного числа в дополнительном коде в десятичное следует вначале превратить его в положительное двоичное число (инвертировать все биты и прибавить 1), а затем положительное двоичное число превратить в десятичное.

В дополнительном коде правила двоичной арифметики применимы к положительным и отрицательным числам, например:

```
  1 1 0 0 1 0 0 0   (-56)        1 1 1 0 1 1 1 1   (-17)
+                              -
  0 1 1 0 1 0 0 1   (+105)      0 0 0 0 0 0 1 1   (+3)
-----------------              -----------------
  0 0 1 1 0 0 0 1   (+49)       1 1 1 0 1 1 0 0   (-20)
```

Арифметические правила для отрицательных чисел такие же, как и для положительных (т. е. беззнаковых), с которыми мы имели дело ранее. Далее в книге число в дополнительном коде называется также знаковым числом, т. е. в нем старший бит является знаковым.

1.3.2. Переполнение и перенос

При выполнении арифметических операций, например сложения или вычитания, процессор 80386 использует сумматор, работающий очень быстро — до 8 млн простых сложений в секунду.

Однако иногда даже простые операции типа сложения вызывают трудности. При сложении беззнаковых чисел результат может быть слишком большим для имеющегося числа байт. Например, 10000001 (129) + 01111111 (127) = 100000000 (256) и результат не помещается в 8 бит. Сумматор сформирует неверный результат 00000000.

Чтобы преодолеть возникшую трудность, сумматор устанавливает дополнительный внутренний бит — флажок переноса. Флажок устанавливается в 1, если результат сложения не помещается в число бит, отведенное для суммы, а в противном случае сбрасывается в 0.

В знаковом числе только младшие 7 бит содержат значение числа, а старший бит содержит знак. Эквивалент переноса (значение слишком велико для представления) возникает, когда сложение двух 7-битных чисел дает результат, не помещающийся в 7 бит. Например, 01111100 (+124) + 00001111 (+15) = 10001011 (-11), что неверно. Такое изменение старшего бита называется переполнением. В процессоре 80386 есть флажок переполнения, который устанавливается в 1, когда результат в младших 7 битах изменяет восьмой бит, и сбрасывается в 0 в противном случае. Переполнение можно игнорировать для беззнаковых чисел. Таким образом, перенос вызывается переносом из левого бита, а переполнение вызывается изменением старшего бита.

Точные условия установки флажков имеют большое значение. В микропроцессоре 80386 есть и другие флажки, состояния которых зависят от результата арифметической операции (см. гл. 2).

1.3.3. Знаковое расширение

Пока мы говорим о числах, представимых в одном байте. Микропроцессор 80386 допускает также применение слов (два байта) и двойных слов (4 байта). Часто требуется преобразовать байт в слово или двойное слово.

Эта задача решается легко для положительных чисел — нужно просто добавить нули слева. Например, байт 00010011 (19) становится словом 00000000 00010011 (тоже 19). Отрицательные числа в дополнительном коде преобразуются сложнее. Если добавить нули слева в отрицательное число, оно совершенно изменится, например, 11101101 (-19) превратится в 00000000 11101101 (237). К счастью, имеется простое правило знакового расширения, по которому меньшие операнды превращаются в большие с сохранением величины и знака. Необходимо взять знаковый бит меньшего типа данных и повторить его слева нужное число раз. Например, байт 11101101 (-19) превращается в слово 11111111 11101101 (-19). Микропроцессор 80386 допускает операции знакового и нулевого расширения при преобразовании байта в слово или в двойное слово и слова в двойное слово.

1.3.4. Двоичная логика

В листинге 1.4 показаны действия логических операций на различные комбинации бит.

Листинг 1.4. Двоичная логика

```
NOT 0 = 1       NOT 1 = 0
0 AND 0 = 0     0 AND 1 = 0     1 AND 0 = 0     1 AND 1 = 1
0  OR 0 = 0     0  OR 1 = 1     1  OR 0 = 1     1  OR 1 = 1
0 XOR 0 = 0     0 XOR 1 = 1     1 XOR 0 = 1     1 XOR 1 = 0
```

Эти двоичные операции применимы не только к битам, но и к байтам. Нужно выстроить два байта как для сложения, а затем применить правила к каждой паре бит. Так как переносов нет, операции можно начи-

нать слева или справа (см. листинг 1.5). Логические операции выполняются над операндами любой, но одинаковой длины.

Листинг 1.5. Применение логических операторов

NOT	AND	OR	XOR
01010110	11001010	11001010	11001010
	01010110	01010110	01010110
10101001	01000010	11011110	10011100

Глава 2

АРХИТЕКТУРА ПРОЦЕССОРА 80386

В этой главе речь пойдет об архитектуре процессора 80386 с точки зрения программирования. В параграфе 2.1 дается краткая история появления 80386 и обсуждаются общие функции типичного микропроцессора. Глава рассчитана на прикладных программистов; в ней описаны все регистры 80386, включая и те, которые обычно используются только операционными системами. Рассмотрены также прерывания и их особые случаи.

Программа 80386 в режиме виртуального процессора 8086 работает почти так же, как и в реальном режиме; оба режима позволяют реализовать на 80386 программы и операционные системы для 8086, а также программы реального режима процессора 80286.

Программы, работающие в защищенном режиме, имеют доступ ко всем средствам реального режима, а также к средствам защищенного режима. Понимание изложенного далее материала необходимо для использования всех режимов и возможностей 80386.

В параграфе 2.2 дана эволюция семейства 8086. Она помогает лучше понять систему команд 80386 и показывает перспективу. Затем обсуждаются компоненты микропроцессора. Основной материал касается внутренней организации 80386, структуры памяти, типов данных и режимов адресации памяти. Рассмотрены все регистры и режимы адресации. Опытные программисты, знающие 8086 и 80286, могут пропустить часть материала данной главы.

2.1. ЭВОЛЮЦИЯ МИКРОПРОЦЕССОРОВ ФИРМЫ INTEL

Первый универсальный микропроцессор 4004 фирмы Intel появился в 1971 г. Он мог выполнять любую программу из системы своих команд, мог ввести данные, обработать их и вывести результаты. Длина слова этого микропроцессора составляла всего 4 бита (тетрада). Он был ориентирован на применение в калькуляторах. Микропроцессор содержал около 1000 транзисторов и выполнял 8000 операций в секунду.

Через несколько лет фирма Intel выпустила микропроцессор 8008 (аналог 4004 с длиной слова 8 бит) и 8080 (достаточно мощный для построения небольшого компьютера). Микропроцессор 8080 (который применяется и сейчас) может выполнять десятичные и 16-битные арифметические операции, вызывать подпрограммы и адресовать память до 64 Кбайт. Шина данных имеет размер 8 бит, а шина адреса — 16 бит.

Группа инженеров фирмы Intel образовала фирму Zilog, которая в 1976 г. выпустила популярный микропроцессор Z80. По сравнению с 8080 он имеет дополнительные регистры и команды. Большинство программ 8080 могут выполняться на Z80. В компьютерах с популярной операционной системой CP/M может применяться любой из этих микропроцессоров.

Аналогичные 8-битные микропроцессоры выпустили и другие фирмы: Motorola (6800) и Mos Technology (6502, применяемый в компьютерах Apple II).

В 1980 г. фирма Motorola выпустила 16/32-битный процессор 68000, который работает с 16-битными данными, но имеет возможность внутренней обработки 32 бит и адресует память 4 Гбайт. Быстродействие процессора около 800 тыс. оп/сек. Его преемниками стали процессоры 68010, 68020 и 68030, по возможностям приближающиеся к миникомпьютерам. Семейство 68000 в настоящее время составляет основную конкуренцию семейству 8086.

За 10 лет число транзисторов в микропроцессоре увеличилось в 70 раз, размер слова составил 16 бит, а быстродействие возросло в 100 раз. Хотя уже были достигнуты некоторые физические ограничения для кристаллов, рынок стимулировал аналогичное развитие и в 80-е годы, примером чего служит процессор 80386.

2.2. СЕМЕЙСТВО 8086

В 1978 г. начат выпуск микропроцессора 8086, а через год — 8088. Размер слова 16 бит (как внутри кристалла, так и при обмене с памятью через внешнюю шину).

Размер адресной шины 20 бит, что позволяет прямо адресовать память 1 Мбайт, но только блоками по 64 Кбайт.

Микропроцессор 8088 аналогичен 8086, но имеет шину данных всего 8 бит. Когда ему требуется 16-битное значение, приходится передавать его в два приема по 8 бит, что ухудшает быстродействие. Но во внутренней обработке и при работе с байтами процессор 8088 аналогичен по быстродействию 8086; его достоинство — возможность работы с дешевыми микросхемами памяти по 8 бит и другими устройствами, созданными для старых микропроцессоров. В книгах и документации часто говорят о «8086» или «базовой архитектуре». Это означает, что для сохранения совместимости другие члены семейства должны эмулировать 8086. Микропроцессор 8088 аналогичен 8086 во всем, кроме размера слова. Поэтому указание на 8086 означает «8086» и «8088» (без специальной оговорки).

При обсуждении реального режима 80386 в основном речь будет идти о возможностях 8086. В 80386 расширена система команд и добавлены регистры. Однако переход от 8086 к реальному режиму 80386 должен быть вполне понятен.

2.3. ОСНОВА МИКРОПРОЦЕССОРОВ

Сердцем любого компьютера является модуль центрального процессора (CPU) со схемами арифметико-логического устройства (ALU). Микропроцессор — это CPU в виде одной или нескольких больших интегральных схем. Небольшие размеры микропроцессора являются существенным фактором, но еще важнее его дешевизна, что и позволяет установить низкие цены на компьютеры.

С точки зрения программиста микропроцессор работает так же, как любой CPU. В состав CPU входят

ALU для производства арифметических операций и устройство управления, которое управляет движением данных в компьютере. Для повышения быстродействия в состав CPU включают регистры, т. е. быстродоступные ячейки памяти. Некоторые из регистров доступны для программистов, а другие только для устройства управления.

В состав компьютера входит также память, содержащая программы и данные. Память состоит из запоминающего устройства с произвольной выборкой (RAM), в которое можно записывать и из которого можно считывать, и постоянного запоминающего устройства (ROM), из которого можно только считывать. Содержимое RAM исчезает при выключении питания или перезапуске компьютера, а содержимое ROM никогда не изменяется. Через порты ввода-вывода (IO) данные пересылаются между CPU и внешними устройствами. Основные элементы компьютера показаны на рис. 2.1.

Различие между памятью и портами IO несущественно, так как IO допускается представлять в виде ячеек памяти: содержимое ячеек RAM управляет тем, что выводится (дисплей) или вводится (клавиатура). В этом случае IO осуществляется операциями записи и считывания соответствующих ячеек памяти. Для многих программ IO сводится к операциям с дисковыми накопителями; для этого вызываются программы операционной системы, которые осуществляют передачи данных, выполняя код ROM.

Пересылки данных в компьютере осуществляются по командам устройства управления в CPU через шину данных (это параллельный тракт между CPU, памятью и портами IO). Регистры и ALU подсоединены с одной стороны шины данных, а память и IO — с другой. Устройство управления помещает данные на шину и считывает их с шины.

Размер слова CPU (16 или 32) бита относится к числу бит его внутренних регистров. Важное значение имеет и ширина шины данных. CPU 8086 считается 8/16-битным, так как его шина данных 8-битная, а внутренние регистры 16-битные. CPU 80386 — это «истинно 32-битный» компьютер, так как ширина шины и размер регистров составляют 32 бита. Размер шины адреса также имеет 32 бита, что упрощает вычисление адреса (адрес хранится в одном регистре).

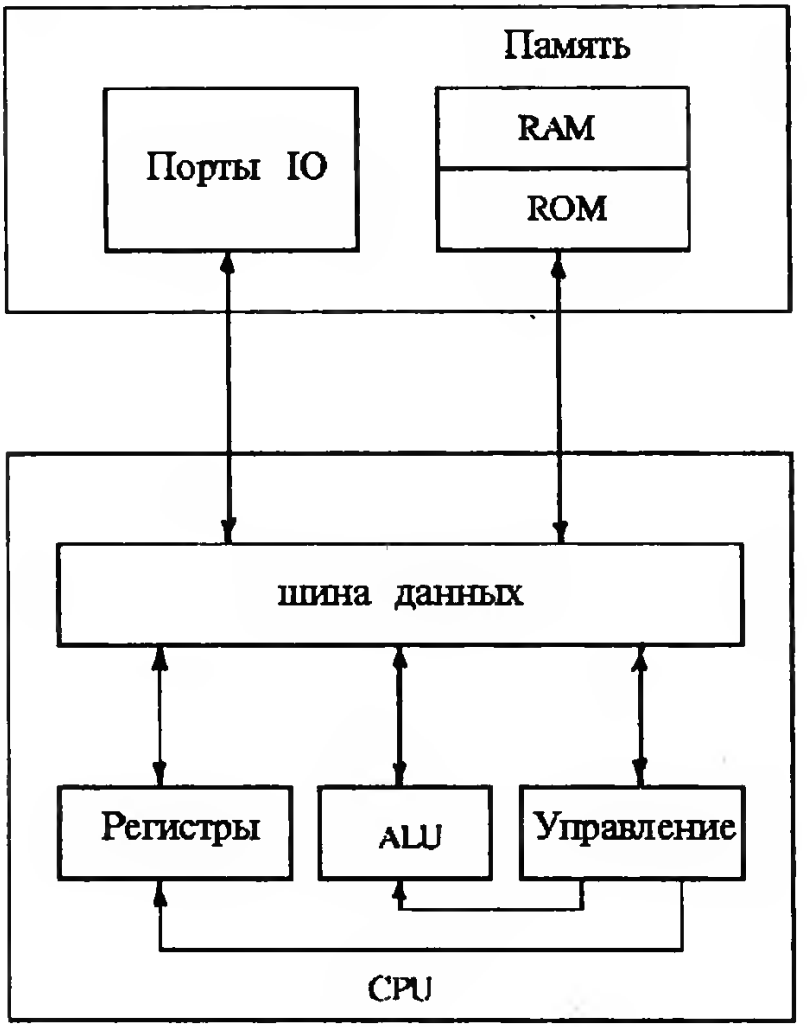


Рис. 2.1. Общие компоненты компьютера

Внутри 80386 находятся CPU (с ALU и регистрами) и устройство управления памятью. Основная память и порты IO находятся вне микропроцессора.

Здесь мы не рассматриваем временную диаграмму работы, управляющие сигналы, взаимодействие с памятью, физическую организацию схемных плат компьютера и др. Многие аппаратные вопросы затронуты в главе 7.

2.4. БАЗОВЫЙ ЦИКЛ В 80386

Большинство современных компьютеров выполняют команды последовательно. Для каждой команды реализуются следующие действия: команда выбирается из памяти, дешифрируется (преобразуется во внутренний микрокод CPU) и выполняется ее микрокод. В результате получается цикл выборки, дешифрирования и исполнения.

Одновременно с этим в компьютере производятся и другие действия. При выборке команды производится автоматический инкремент регистра, называемого программным счетчиком PC. Он сообщает CPU, где искать следующую команду. При инкременте PC команды выполняются в том порядке, в каком они хранятся в памяти. Только при «переходе» или «вызове» подпрограммы в PC загружается новое значение, и следующая команда выбирается из новой указанной ячейки.

Операнды многих команд хранятся в памяти. Например, команда ADD суммирует два числа, одно из которых находится в памяти. Его нужно передать для обработки из памяти в CPU. Следовательно, базовый цикл дополняется операциями инкремента PC и считывания операнда из памяти. Команды, использующие операнд из памяти, выполняются дольше.

В CPU 80386 это усложнение преодолевается с помощью «конвейеризации». Пока одна команда выбирается из памяти, вторая дешифрируется, а третья исполняется. Обычно в конвейере одновременно находятся 5 или 6 команд, т. е. по завершению одной команды в микропроцессоре 80386 сразу начинается исполнение следующей, которая уже выбрана из памяти и дешифрирована.

2.4.1. Процессор 80386

Процессор состоит из нескольких функциональных устройств («блоков»). Программист может прямо управлять только операционным устройством EU, которое содержит внутренние регистры и ALU, а также контроллером, который вызывает исполнение команд. Важнейшим элементом компьютера является основная память, обычно в виде RAM. Далее мы подробнее остановимся на регистрах 80386 и взаимодействии с памятью. Именно это представляет наибольший интерес для прикладного программиста.

Операционное устройство содержнт быстродействующнй 32-бнтный сумматор и 64-битный параллельный сдвигатель, который может сдвнгать 32-бнтный операнд на 31 бнт в любом направленни. Суммированне и сдвнг — это быстрые команды, особенно если операнды находятся в регистрах CPU.

2.4.2. Программируемые регистры

Большинство регистров 80386 используются прикладными программнстами, и только часть их используется снстемными программнстами. Программируемые регистры включают в себя регистры общего назначения, регистры сегментацин, регистры флажков н указателя команды.

Некоторые регистры 80386 входят в «базовый регистровый набор». Это группа регистров, которые есть в каждом процессоре семейства 80386. В нее входят младшие 16 бит прикладных регистров н слово состояння (см. гл. 6).

Регистров общего назначения всего 8; они содержат по 32 бнта. Имена полных 32-бнтных регистров начинаются с буквы E (Extended — расширенный): EAX, EBX, ECX, EDX, EBP, ESI, EDI и ESP. Младшие 16 бит каждого нз этих регистров можно указывать так же, как н в прежних CPU: AX, BX, CX, DX, BP, SI, DI и SP. Наконец, первые четыре 16-битных регистра можно адресовать парамн байтных регистров: AH и AL для AX, DH н DL для DX, BH н BK для BX, CH н CL для CX (см. рис. 2.2 и 2.3).

Все регистры адресуются как двойные слова, т. е. как 32-битные регистры. В двухсловном регистре одна команда (напрнмер, MOV EAX,1) оперирует всем двойным словом. Младшая половнна каждого регистра адресуется как слово (16-бнтный регистр). В двухсловный регистр можно загрузить слово в младшую половину, не влияя на старшне 16 бит. Когда нужно обращаться к регистру в 32- илн 16-битной формах (как удобнее программнсту), мы помещаем в скобках букву E; например, (E)AX означает «EAX или AX-что нужно». Первые четыре регистра допускают в младшей половине адресацию байт.

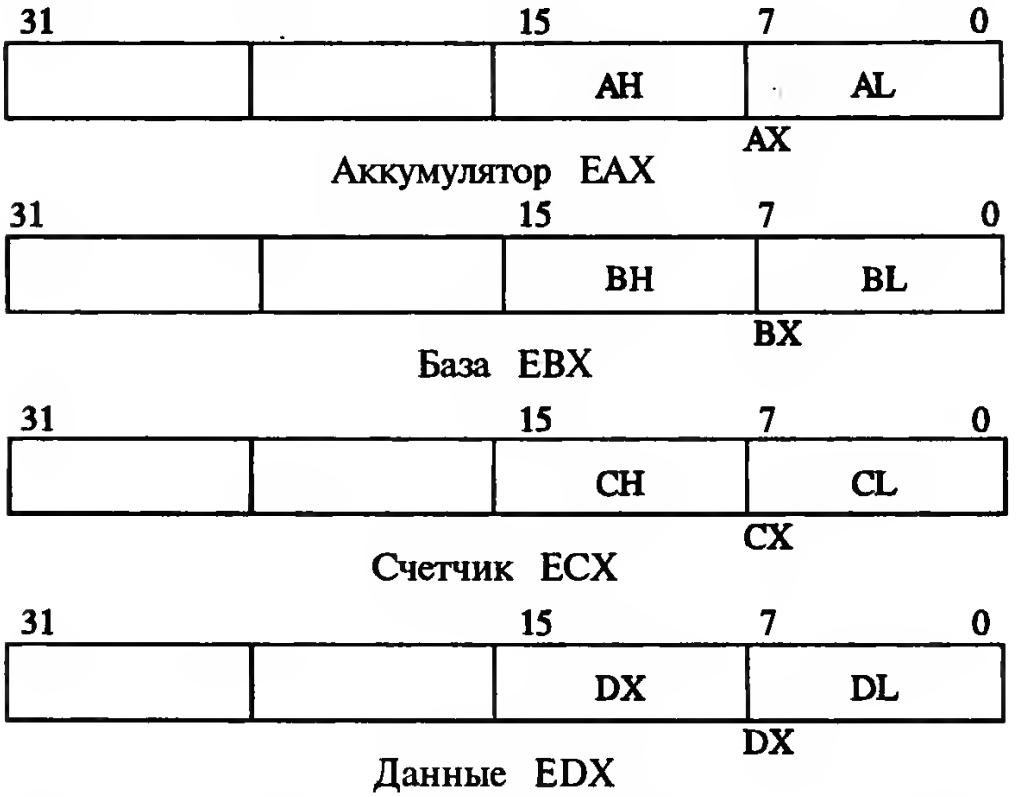


Рис. 2.2. Регистры данных 80386 (общего назначения)

Каждый из регистров общего назначения имеет специализированное применение. В командах типа ADD программист может указывать в качестве операндов любые два регистра. В операциях PUSH предполагается, что стек адресует регистр SP.

Покажем назначение каждого регистра:

(E)AX — аккумулятор, применяется в десятичной арифметике;

(E)BX — регистр базы, применяется как база при вычислении адреса;

(E)CX — счетчик, применяется как счетчик в циклических операциях;

(E)DX — регистр данных, хранит данные для нескольких операций;

(E)SP — указатель стека, содержит смещение вершины стека;

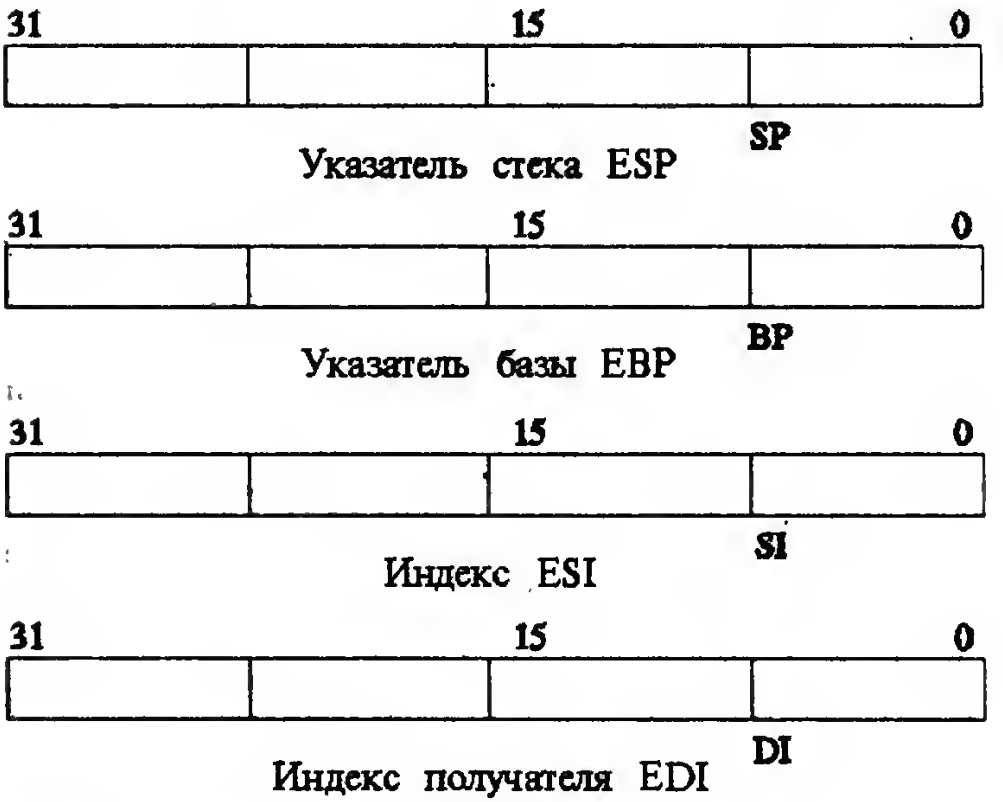


Рис. 2.3. Регистры указателя и индексации

(E)BP — указатель базы, может содержит базу области данных;

(E)SI и **(E)DI** — индексы источиика и получателя, применяются для адресации смещеиия.

Подробиее об использовании каждого регистра см. главу 4.

Регистры сегментации примеияются для определения иачальиых смещеиий в памяти области кода и даиных. Каждый из шести этих регистров имеет свое назначеиие. CS адресует программиый код, DS — даиные программы, SS — ее стек. Дополнительиые регистры ES, TS, FS и GS предназиачеиы для структур даииых; из них только ES используется в конкретиых командах (см. рис. 2.4).

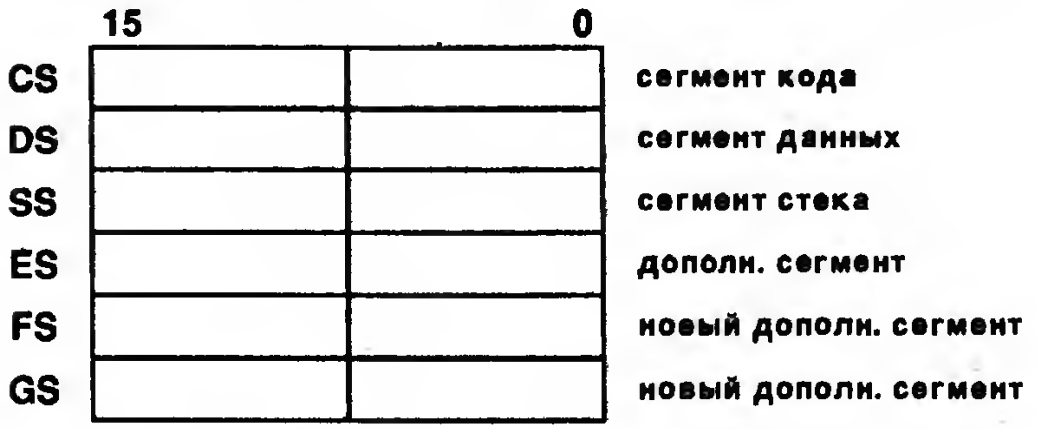


Рис. 2.4. Сегментные регистры

Прикладные программы обычно не обращаются к регистрам сегментации, ими полностью управляет операционная система. Содержимое регистров сегментации вместе с другими регистрами определяет, где находится следующая команда (CS плюс указатель команды), где вершина стека (SS плюс ESP) и т. д. Способ объединения содержимого регистров зависит от режима, в котором работает программа.

Еще два регистра — это регистр флажков EFlags, который управляет некоторыми операциями и показывает текущее состояние процессора 80386, и указатель команды, который используется вместе с CS для адресации следующей команды. Программисту доступны только некоторые биты в регистре флажков, а указатель команды модифицируется только при переходах и вызовах.

2.5. РЕГИСТР ФЛАЖКОВ

Регистр флажков EFlags процессора 80386 содержит 32 бита. Прикладные программы работают только с младшими 16 битами EFlags. Биты в EFlags отражают состояние 80386 и управляют выполнением некоторых операций.

Различают три типа флажков: системные флажки (они отражают текущее состояние компьютера в целом и чаще используются операционной системой, а не прикладными программами), флажки состояния (показывают состояние конкретной программы) и флажки управления (прямо влияют на некоторые команды). Формат регистра флажков приведен на рис. 2.5.

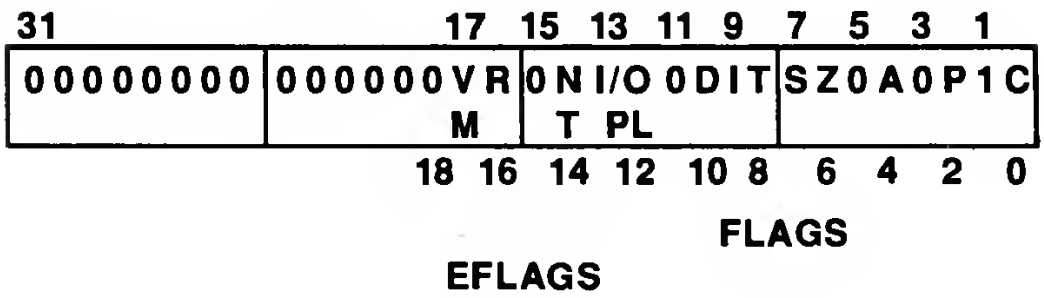


Рис. 2.5. Регистр флажков

2.5.1. Резерв фирмы Intel

Некоторые биты регистра флажков зарезервированы фирмой Intel; биты 31—18 всегда содержат 0. Биты 15, 5 и 3 ранее всегда содержали 0, а бит 1 всегда содержит 1. Если в ваших программах используются зарезервированные биты, то программы на данном CPU 80386, по-видимому, будут работать, но могут и не работать в последующих версиях микросхемы.

2.5.2. Системные флажки

ФЛАЖОК VM — ВИРТУАЛЬНЫЙ РЕЖИМ

0 = защищенный режим; 1 = режим виртуального 80386.

Этот флажок показывает, работает ли ваша программа в режиме виртуального 80386 (см. гл. 5). Обычно вы не можете проверить этот бит (и следующий) в реальном режиме.

ФЛАЖОК R — ВОЗОБНОВЛЕНИЕ

0 = нет ошибки; 1 = ошибка отладки.

Этот флажок временно включает средства отладки, когда программа возобновляет работу после особого случая отладки.

ФЛАЖОК NT — ВЛОЖЕННАЯ ЗАДАЧА

0 = текущая задача не вложена; 1 = текущая задача вложена.

Этот флажок показывает, выполняется ли текущая задача «под» некоторой другой задачей; он влияет на выполнение команды IRET.

ФЛАЖОК IOPL — УРОВЕНЬ ПРИВИЛЕГИИ ВВОДА/ВЫВОДА (БИТЫ 13 И 12)

0 = текущая задача имеет высший приоритет; 1 = следующий ниже; 2 = следующий ниже; 3 = низший приоритет.

Два бита IOPL используются процессором и операционной системой для определения прав доступа прикладной программы к средствам ввода/вывода. Допустимые уровни варьируются от 0 (наиболее привилегированные) до 3 (наименее привилегированные).

ФЛАЖОК I — ПРЕРЫВАНИЕ

0 = внешние прерывания запрещены; 1 = разрешены.

Этот флажок определяет, будет ли CPU реагировать на внешние прерывания или игнорирует их. Он не влияет на особые случаи прерывания (порождаемые программой) и немаскируемые внешние прерывания. Программы в мультизадачных системах должны избегать воздействия на этот флажок.

ФЛАЖОК T — ТРАССИРОВКА

0 = нет трассировки; 1 = прерывание после каждой команды.

Этот флажок вызывает генерирование особого случая прерывания по каждой команде; применяется для покомандной работы при отладке.

2.5.3. Флажки состояния

Флажки состояния (прямо используемые прикладными программами) устанавливаются многими командами 80386, особенно арифметическими.

Программист может проверить флажок для определения последующих действий программы. Хотя каждый флажок имеет общую функцию, точный смысл его зависит от последней выполненной команды.

ФЛАЖОК O — ПЕРЕПОЛНЕНИЕ

0 = нет переполнения; 1 = возникло переполнение.

Этот флажок устанавливается в 1, если результат арифметической операции превышает доступные пределы; если этого нет, флажок переполнения сбрасывается в 0.

ФЛАЖОК S — ЗНАК

0 = старший бит содержит 0; 1 = старший бит содержит 1.

Значение этого флажка совпадает со старшим битом результата. Для знаковых чисел этот бит показывает знак результата: 1 = отрицательный, 0 = положительный.

ФЛАЖОК Z — НУЛЬ

0 = последний ненулевой результат; 1 = последний результат был нулевым.

Этот флажок устанавливается в 1 (истина), если результат операции нуль, и сбрасывается в 0 (ложь), если результат ненулевой.

ФЛАЖОК A — КОРРЕКЦИЯ ИЛИ ВСПОМОГАТЕЛЬНЫЙ ПЕРЕНОС

0 = нет внутреннего переноса; 1 = внутренний перенос.

Этот флажок показывает состояние «внутреннего» переноса или заема (при сложении и вычитании) из бита 3 в бит 4 — межтетрадный перенос или заем.

ФЛАЖОК P — ПАРИТЕТ

0 = младший байт имеет четный паритет; 1 = младший байт имеет нечетный паритет.

Состояние этого флажка зависит от младшего байта результата: если он содержит четное число единиц, то P = 1, а в случае нечетного числа единиц P = 0.

ФЛАЖОК C — ПЕРЕНОС

0 = нет переноса из старшего бита; 1 = есть перенос.

Показывает, вызвали ли сложение или вычитание перенос или заем из старшего бита результата.

2.5.4. Флажки управления

Флажки управления действуют только на циклические команды.

ФЛАЖОК D — НАПРАВЛЕНИЕ

0 = автоинкремент; 1 = автодекремент в циклических командах.

Этот флажок управляет «направлением» операций. Когда D = 0, циклы обрабатываются от младших адресов к старшим, а когда D = 1, обработка производится от старших адресов к младшим.

2.6. ПЛОСКАЯ И СЕГМЕНТИРОВАННАЯ ПАМЯТЬ

В процессоре 8086 адреса образуются из 16-битного селектора сегмента и 16-битного смещения. Селектор сдвигается влево на 4 бита и при суммировании двух чисел получается 20-битный адрес, как показано на рис. 2.6.

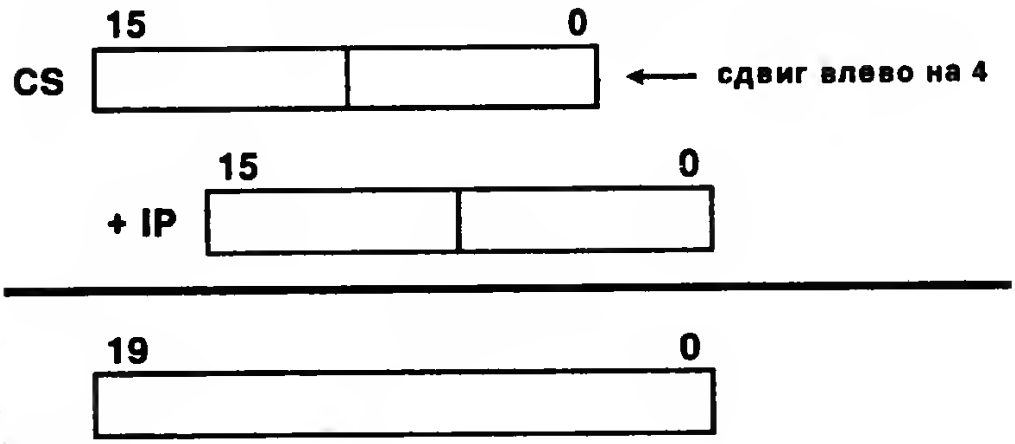


Рис. 2.6. Сложение селектора и смещения для получения 20-байтного адреса

Областн программ и данных состоят из сегментов. Например, прн запуске основного блока программы в CS загружается начальное значение кода программы, а указатель команды устанавливается на 0. По мере выполнения команд пронзводится ннкремент программного счетчнка. Команды JMP и CALL загружают в указатель команды новые значення, но CS не изменяется. Так как содержнмое нндекса команды ограннчено 2^{16}-1, то максимальный размер сегмента программы равен 64 Кбайт. Областн программы н данных могут занимать несколько сегментов по 64 Кбайт, но 20-байтный адрес ограничивает память до 2^{20} илн 1 Мбайт. Операционная система DOS не может адресовать более 640 Кбайт.

Ограниченння размера сегментов и памятн являются трудной проблемой для программ н систем на базе 8086. Напрнмер, структура данных не может быть более 64 Кбайт без включения двух или более сегментов. Программа, работающая с такой структурой, должна постоянно контролнровать достнженне граннцы сегмента и в нужное время переключаться на новый сегмент. Такой контроль и переключенне сильно замедляют доступ к большим структурам. Например, для избежания мерцання экрана необходнмо очень быстро осуществлять его регенерацию. На экране 1024×1024 пиксела каждый из них описывается байтом (включен, выключен, мерцание и яркость даже для нецветного экрана). Следовательно, для поддержки экрана нужно иметь RAM 1 Мбайт, что намного больше размера сегмента; только такую память

и может адресовать 8086, не оставляя места для программ и данных.

Адреса данных образуются с привлечением регистра DS (иногда и ES) и смещения; адреса для стековых операций формируются на основе содержимого регистров SS и SP. Но в любом случае максимальный размер сегмента равен 64 Кбайт.

В реальном режнме процессоры 80286 н 80386 работают аналогично. Однако в защищенном режиме все нзменяется. В 80386 с прнвлечением 32-битных регистров можно реалнзовать модель плоской памяти.

В этой модели нет сегментов (илн есть один огромный сегмент). Память считается единой и неразрывной. Такая модель применяется в микропроцессоре 68000 и других. Для резалнзации этой модели в 80386 нужно просто установнть все сегментные регистры на нуль. Размер всех регистров для смещений (EIP, ESP и другие общие регистры) равен 32 бнтам. Это допускает адресацию 2^{32} байт или 4 Гбайт, что в 4000 раз больше, чем диапазон адресацни 8086.

По желанию в защищенном режнме 80386 можно использовать н модель сегментнрованной памяти, как в 8086. Однако метод вычнсления адреса оказывается другим. Сегментные регистры не суммируются со смещениями, а используются как селектор илн указатель в списке «дескрипторов сегментов». Дескриптор содержит важную информацию о сегменте, включая его базовый адрес и длину (подробнее см. гл. 5).

2.6.1. Регистры управления, проверки и отладки

Данные регистры обычно не используются прикладными программами. Однако назначенне нх нужно поннмать, так как онн применяются для поддержки операционной системы, параллельных процессов, отладки и других компонентов среды, в которой разрабатываются и выполняются программы.

Имеются четыре регистра управления, доступные только по команде MOV. Например, команда MOV EAX,CR0 загружает в EAX содержнмое CR0 первого регистра управления; команда MOV CR3,EBX передает содержимое EBX в CR3 — третий регистр управления.

Данные вариаиты команды MOV можно использовать только на уровне привилегий 0. Регистр CK0 также содержит несколько важных флажков.

ФЛАЖОК PG — РАЗРЕШЕНИЕ СТРАНИЦ (БИТ 31)

0 = нет страниц; 1 = есть страничная организация.

Когда PG=1, процессор использует таблицы страниц, которые применяются для работы с виртуальной памятью и других целей, определяя нужный адрес. Когда PG=0, таблица страиниц не используется.

ФЛАЖОК ET — ТИП СОПРОЦЕССОРА (БИТ 4)

0 = 16-битный сопроцессор 80287; 1 = 32-битный сопроцессор 80387.

Состояние этого флажка сообщает 80386 о типе имеющегося сопроцессора — 80287 или 80387. В первом случае применяется 16-битный протокол, а во втором 32-битный.

ФЛАЖОК TS — ЗАДАЧА ПЕРЕКЛЮЧЕНА (БИТ 3)

0 = нет переключения задачи; 1 = задача переключена.

Когда TS=1, произошло переключение задачи. Этот флажок влияет на команды сопроцессора и некоторые другие команды.

ФЛАЖОК EM — ЭМУЛЯЦИЯ (БИТ 2)

0 = нет эмуляции сопроцессора; 1 = эмулировать сопроцессор.

Для передачи управления сопроцессору обычно применяется команда ESC. Если при выполнении ESC флажок EM = 1, генерируется особый случай прерывания, и обработчик прерываний эмулирует численный сопроцессор.

ФЛАЖОК MP — НАЛИЧИЕ СОПРОЦЕССОРА (бит 1)

0 = нет сопроцессора; 1 = сопроцессор присутствует.

CPU 80386 проверяет этот флажок при выполнении команды WAIT. Если MP = 1, то проверяется флажок

TS; если TS также установлен, генерируется особый случай прерывания, активизирующий сопроцессор.

ФЛАЖОК PE — РАЗРЕШЕНИЕ ЗАЩИТЫ (БИТ 0)

0 = реальный режим; 1 = защищенный режим (включает режим виртуального процессора 8086).

Состояние этого флажка показывает режим работы процессора. Отметим, что режим виртуального 8086 является подмножеством защищенного режима.

Все 32 бита регистра CR1 зарезервированы фирмой Intel. Регистр CR2 применяется, если включена страничная организация; если возникает нарушение выборки страниц (обычно, если нужной страницы нет в памяти), то в нем сохраняется линейный адрес, вызвавший нарушение. Старшие 20 бит регистра CR3 также применяются для страничной организации; они содержат базовый адрес каталога страниц. Младшие 12 бит CR3 не определены.

Регистры отладки (см. рис. 2.7) являются важным элементом мощных отладочных средств 80386. Сами регистры подробно рассмотрены далее. Флажки R и T в регистре EFlags также применяются при отладке.

Четыре регистра адреса отладки DR0—DR3 содержат адреса остановов. Адреса являются либо реальными адресами, либо индексами в таблице страниц в зависимости от того, разрешена страничная организация или нет. Так как различные задачи могут использовать разные таблицы страниц, бит в регистре DR7 сообщает, применимы ли адреса в DR0—DR3 ко всем задачам или только к текущей задаче. Адреса являются фактическими адресами и применимы к текущей задаче, когда процессор работает в реальном режиме, так как страничная организация и мультизадачность при этом недопустимы. Следовательно, прикладные программы могут прямо использовать регистры отладки.

В зависимости от состояний флажков в регистре DR7 четыре адреса в DR0—DR3 могут вызвать останов, когда происходит обращение к данным по этим адресам.

Рис. 2.7. Регистры отладки

Регистры DR4—DR5 зарезервированы фирмой Intel. Регистр состояния отладки DR6 содержит несколько специальных флажков. Младшие биты B3—B0 показывают, какой набор условий вызывает останов. Аналогично биты BD, BS и BT показывают условия в регистрах отладки: возник ли особый случай прерывания при покомандной работе и установлен ли бит T, связанный с переключением задачи.

Регистр управления отладкой DR7 помогает включать или выключать отладочные средства. Поля LEN0—LEN3 определяют длину элемента данных, контролируемого по адресам из DR0—DR3. Биты R/W сообщают при каких условиях регистр адреса отладки вызовет останов (выполнение команды, запись данных или считывание/запись). Если значение в одном из четырех полей R/W = 0 (останов при выполнении команды), то соответствующее поле длины LEN должно содержать 0 (длина не определена).

Четыре бита G0—G3 определяют для каждого регистра DR0—DR3, доступен ли он глобально, т. е. доступен всем задачам. Биты L0—L3 определяют, доступен ли каждый из регистров адреса локально, т. е. доступен только текущей задаче. Биты L0—L3 изменяются при переключении на новую задачу, но они «перевешиваются» 1 в соответствующем бите G0—G3.

Из-за конвейеризации процессор может вызвать останов при опережающей выборке команды. Так как во внутреннем конвейере могут находиться несколько команд, «подозрительной» может стать команда, которая не выполнялась. Если установлен байт LE или GE, то включается опережающая выборка; это уменьшает скорость, но обеспечивает условия, при которых останов вызывает только выполняемая команда. Флажок LE сбрасывается при переключении задачи, а флажок GE нет.

Два проверочных регистра доступны нескольким командам MOV. Они допускают запись и считывание данных из буфера преобразования, который применяется для ускорения работы со страницами. Регистры полезны при разработке тест-программ для аппаратных средств и даже для разработки процедур оптимизации страничной организации. Проверочные регистры почти не используются прикладными и системными программистами.

2.6.2. Регистры управления памятью

Четыре регистра описывают структуры данных, используемые операционной системой для управления памятью. Важно понимать, что 80386 может одновременно выполнять несколько задач, а каждая задача состоит из нескольких сегментов. У каждого сегмента есть дескриптор, содержащий размер сегмента и другую важную информацию. Мультизадачность рассматривается в главе 5.

IDTR — регистр дескрипторной таблицы прерываний, адресует таблицу точек входа обработки прерываний.

GDTR — регистр глобальной дескрипторной таблицы, содержит дескрипторы сегментов, доступных любой задаче, выполняющейся на компьютере.

LDTR — регистр локальной дескрипторной таблицы, содержит дескрипторы сегментов, доступных конкретной из текущих выполняющихся задач.

TR — регистр задачи, содержит копию дескриптора текущей задачи и показывает, где сам дескриптор хранится в памяти.

2.6.3. Типы данных

Базовые типы данных процессора 80386: биты, байты, 16-битные слова (в AX, BX и других регистрах) и двойные слова (размер всех физических регистров EAX, ESP и др.). Такие названия могут вызвать путаницу, так как «размер слова» компьютера обычно совпадает с размерами шины данных и/или внутренних регистров, а 80386 — это 32-битный процессор. Однако 16-битное значение называется словом, чтобы обеспечить совместимость для всего семейства 8086. Другие типы данных образуются из базовых типов путем объединения их в большие единицы.

При обращении к памяти процессор 80386 считывает и записывает двойные слова, но отдельные байты можно модифицировать независимо от содержащего их двойного слова. Для программиста память состоит из последовательности байт, имеющих свои адреса.

Ниже приведен список типов данных, которые поддерживаются процессором 80386. Большинство их знакомо опытному программисту, но важно отчетливо понимать смысл всех терминов.

Наименьшая единица — бит, принимающий значение 0 или 1. При интерпретации бит в регистре EFlags эти значения можно считать ложью (0) и истиной (1).

Базовая единица — байт, он состоит из 8 бит и допускает несколько интерпретаций:

1. Беззнаковое число от 0 до 255. Каждый бит от младшего (бит 0) до старшего (бит 7) представляет степень числа 2. Эти значения могут представлять также символы кода ASCII (буква, цифра или специальный символ).
2. Знаковое число от -128 до +127. Старший бит (бит 7) показывает знак числа (1 = отрицательное), а остальные — величину числа. Чис-

ла представляются в дополнительном коде (см. гл. 1). Набор 11111111 — это беззнаковое число 255, либо знаковое число -1.

3. Десятичная цифра похожа на беззнаковое число, но диапазон ее от 0 до 9. Байт может содержать две десятичных цифры, занимающих по 4 бита (тетраде). Команды процессоров семейства 8086 поддерживают арифметику упакованных десятичных чисел.

16-битное слово может содержать два байта, допускающих любой из приведенных выше форматов. Но как единая сущность слово допускает две интерпретации:

1. 16-битное беззнаковое число от 0 до 65535. Все биты представляют степень числа 2; нумерация бит от 0 до 15.
2. Знаковое число от -32768 до +32767. Бит 15 показывает знак (1 = отрицательное число).

32-битное слово может содержать приведенные выше типы данных, но имеет и несколько других интерпретаций. 32-битные типы данных представляют собой максимальные числа и адреса, с которыми может работать процессор 80386. Есть две возможные интерпретации:

1. 32-битное беззнаковое число от 0 до 2^{32}-1 (4 294 967 295). Такие числа называются близкими указателями, так как они могут адресовать любую ячейку в данном сегменте.
2. 32-битное знаковое число от 2^{31} до 2^{31}-1, где бит 31 показывает знак.

Из базовых типов данных можно образовать другие типы:

1. Двоичное поле состоит из смежных бит внутри двойного слова. Оно начинается в любом месте, но не выходит за пределы двойного слова. Максимальная длина поля от 1 до 32 бит в зависимости от начальной точки. Регистр EFlags образован из набора двоичных полей.
2. Цепочка в общем смысле означает последовательность смежных бит, байт, слов или двойных слов. Двоичная цепочка может содержать до 2^{32}-1 бит, а другие цепочки до 2^{32}-1 байт. Символьная цепочка состоит из байт, содержащих символы ASCII.

Сопроцессор 80387 поддерживает еще два типа данных: 64-битное слово и 80-битное значение (tbyte).

2.6.4. Режимы адресации

Некоторые команды работают с данными, содержащимися в самой команде или в регистрах CPU, т. е. они не обращаются к памяти. Это примеры непосредственного и регистрового режима адресации, например, в команде MOV AX,7FH используются оба эти режима.

Когда программа обращается к памяти, она должна сообщить компьютеру, какую ячейку памяти использовать. В простейшей форме можно прямо указывать имя ячейки, например, ADD AX,ANADDRESS, где ANADDRESS — ранее определенная ячейка.

Для эффективного программирования требуется использовать несколько способов именования ячеек памяти. В ассемблерном операторе для определения адреса можно объединять 4 возможных элемента. Вычисленный или «эффективный адрес» объединяется с соответствующим сегментным регистром, давая окончательный адрес. Вот эти элементы:

1. **База.** Это содержимое одного из регистров общего назначения. База считается начальной точкой, а другие элементы прибавляются к ней с образованием эффективного адреса. В регистр можно поместить начальную ячейку массива данных.
2. **Смещение.** Это адрес ячейки в сегменте памяти. Длина смещения 8, 16 или 32 бита.
3. **Индекс.** Как и в базовой адресации, для образования эффективного адреса используется содержимое какого-либо регистра. В случае 16-битных операндов для индексирования предназначены SI и DI; а в случае 32-битных операндов — любой регистр, кроме ESP.
4. **Масштаб.** Если индекс является 32-битной величиной, его можно умножить на 2, 4 или 8. Это удобно при обращении к массивам с элементами фиксированного размера.

Эффективный адрес вычисляется следующим образом:

EA = База + (Индекс × Масштаб) + Смещение

При отсутствии некоторых элементов формула упрощается. Если, например, нет индекса, то EA = База + Смещение. Ниже приведены примеры использования каждого режима.

1. Прямая адресация включает регистровые и непосредственные операнды; обращение к памяти не требуется.
2. Используется только смещение. Смещение обычно указывается меткой; вычисляется расстояние от текущей команды до метки и используется как смещение при обращении к памяти.
3. В косвенной адресации адресом служит содержимое регистра; имя регистра заключается в квадратные скобки. Например, команда MOV AX,BX передает в AX содержимое BX, а команда MOV AX,[BX] передает в AX содержимое ячейки памяти, адресуемой содержимым регистра BX.
4. Базовая адресация требует прибавить константу к значению в регистре и использовать сумму как эффективный адрес. Выражение «регистр + смещение» заключается в квадратные скобки. Например, в команде MOV AX,[BX+4] сумма содержимого BX и 4 служит эффективным адресом. Значение, содержащееся по данному адресу, передается в AX.

В остальных режимах используется индексирование. Индекс заключается в квадратные скобки и помещается после базы, с которой он суммируется. Масштабировать (умножать на 2, 4 или 8) можно только индексы.

5. В индексной адресации эффективный адрес равен сумме прямого адреса и индекса. В команде MOV ECX,TABLE[SI] эффективный адрес равен сумме SI и значения TABLE. Число, хранящееся по эффективному адресу передается в ECX.
6. Индекс объединяет базу в регистре и индекс в регистре. В команде MOV ECX,[EDX][EAX] эффективный адрес равен сумме EDX и EAX. В случае 32-битных операндов индекс можно масштабировать.
7. Масштабированный индекс можно умножать на 2, 4 или 8. Например, в команде ADD ECX, TABLE [ESI×8] значение из ESI умножается на 8, суммируется с адресом TABLE и резуль-

тат служнт эффектнвиым адресом. Такой прием особеино удобеи для элемеитов даниых длниой в 8 байт.

Кроме приведеииых режнмов можио использовать любую комбннацию базы, нндекса (с масштабированнем или нет) н смещення. Вычисленне адреса ведется параллельно с другими действиямн, поэтому даже в случае самого сложного режима адресации дополинтельиое время ие требуется. Но есть одио исключение: прн иаличии базы, индекса и смещеиия время увеличивается иа одии такт.

2.6.5. Прерывания и их особые случаи

Прерывание — это изменеиие обычного хода исполнеиия программы. Обработка прерываний встроена в процессор 80386; прн выполненин команды ои прежде всего контролнрует налнчие прерывания. Прерыванне (появнвшееся извие или вызванное исполненнем команды) инициирует обращеиие к таблице прерыванний; таблица адресует процедуру, служащую «обработчиком прерывания». Процессор 80386 имеет три типа прерываний. Первый тнп называется «особым случаем» н возиикает прн выполиеиии команд. Например, при выполнеиии команды INT фактически возиикает особый случай прерывання. Обработка прерываний в операциоиных системах ведется по разиому. Список особых случаев прерывания для реального режима прнведеи в табл. 2.1.

Существует 4 типа особых случаев прерываиня: аварии, специальные прерывання (traps), нарушення н программные особые случаи (иазываются еще программнымн прерываниямн). Авария — самый серьезный особый случай, ои ие позволяет идентнфицировать вызвавшую команду и осуществить рестарт программы. Это возникает, если какой-либо элемент компьютера ведет себя иеожиданным образом, поэтому проблемы связаиы ие только с текущей командой, но н с неверными предыдущими результатами. Аварии часто вызываются аппаратнымн ошибками (ие ошибками устройств ввода/вывода) и иесовместимыми значеииями в системиых таблицах.

Таблица 2.1. Номера и описания прерываний

Номер	Описание	Команда
0	Ошибка деления	DIV, IDIV
1	Особый случай при отладке	Любая
2	Немаскируемое прерывание	
3	Останов	INT 3
4	Переполнение	INTO
5	Контроль границ массива	BOUND
6	Неверный код операции	Включая LOCK с неверной командой
7	Сопроцессор отсутствует	ESC, WAIT
8	Вектор прерывания слишком велик	INT
9	Зарезервирован	
10	Неверный TSS	Переключение задачи
11	Сегмент отсутствует	Многие
12	Пересечение границы стека (смещение меньше 0 или больше 64 Кбайт)	PUSH, POP, PUSHF, POPF, PUSHA, POPA
13	Общая защита (смещение больше 64 Кбайт или длина команды более 15 байт)	Многие
14	Страничное нарушение (страница отсутствует)	Многие
15	Зарезервирован	
16	Ошибка процессора	ESC, WAIT
17-31	Зарезервированы	
32-255	Доступны для маскируемых прерываний	

Специальное прерывание возникает сразу после выполнения ошибочной команды. Такое нарушение фиксируется до или во время выполнения команды.

Программное прерывание вызывается командой и возникает всегда или иногда. Примерами таких команд служат INT 3, INT n, BOUND.

Аппаратные прерывания подразделяются на два типа: маскируемые прерывания (исполняемые в зависимости от состояния флажка I) и немаскируемые (выполняются всегда). Процессор 80386 проверяет наличие прерывания перед выполнением каждой команды; немаскируемое прерывание вызывает программу обработчика прерывания независимо от состояния программы. Немаскируемые прерывания подаются на вход INTR, а немаскированные на вход NMI (хотя эти прерывания фактически маскируются при обработке предыдущего немаскируемого прерывания).

Немаскируемые прерывания имеют идентификатор 2, а маскируемые — от 32 до 255. Эти номера назначаются внешним контроллером прерываний, например, 8259A. Каждый контроллер имеет 8 входов, причем на любой из них можно подключить еще один контроллер; это так называемое «каскадное» включение. Схемы контроллеров «прозрачны» для программистов, которые знают прерывания только по номерам. Система прерываний в 80386 мало отличается от предыдущих типов процессоров. Но одно важное различие касается проблемы изменения флажка I в мультизадачной системе (см. гл. 5).

Конкретные прерывания используются для разных функций, например, для обеспечения работы сопроцессора и режима защиты. Далее прерывания рассматриваются только в связи с другими объектами.

Глава 3

СИСТЕМА КОМАНД ПРОЦЕССОРА 80386

Глава 3 посвящается обзору системы команд процессора 80386 и преследует две цели. Первая — ввести читателя в семейство команд 8086, вторая — служить справочником системы команд процессора 8086.

Команды сгруппированы по выполняемым функциям. Кратко описана каждая команда и общие правила ее применения. Большинство функциональных групп сопровождается примером действий одной или нескольких команд данной группы.

Очень важным для команд является понятие операнда. Это то значение данных, которое «обрабатывает» команда. Хотя каждая команда имеет свою структуру операндов, но имеется и много общих особенностей.

Операнд обычно рассматривается как источник или получатель (в зависимости от того, берет команда данные или помещает данные в него). Получателями могут быть только регистры или ячейки памяти, а источниками — еще и непосредственные данные. Одновременно и источником, и получателем ячейки памяти быть не могут.

Обычно операнды могут иметь любой размер (байт, слово или двойное слово). В большинстве команд с несколькими операндами все операнды имеют одинаковый размер.

Полное описание каждой команды см. в главе 4.

3.1. КОМАНДЫ ПЕРЕДАЧИ ДАННЫХ

Команды этой группы пересылают данные из одного места компьютера в другое: между регистрами, между регистрами и памятью, между регистром или памятью и стеком.

MOV — пересылает один элемент данных из одного места в другое.

XCHG — обменивает содержимое двух регистров или содержимое регистра и ячейки памяти. Часто применяется для синхронизации нескольких процессов, так как ее нельзя прерывать другим устройством, использующим шину данных.

PUSH — копирует операнд-источник в вершину стека. Применяется для помещения параметров в стек перед вызовом процедуры. Полезна также для временного сохранения данных в стеке.

POP — берет верхний элемент из стека и пересылает его в операнд-получатель. Применяется для возвращения из стека значений, включенных в него командой PUSH.

PUSHA и PUSHAD — применяются для помещения содержимого всех 8 регистров общего назначения в стек (PUSHA — оперирует 16-битными регистрами, PUSHAD — 32-битными регистрами). Эти команды используются перед вызовом процедур.

POPA и POPAD — восстанавливают из стека содержимое 8 регистров общего назначения, являясь дополнением команд PUSHA и PUSHAD.

Следующий пример показывает два способа обмена содержимого двух регистров. Отметим, что в варианте 1 применяется стек, а не ячейка памяти.

Исходное состояние:

EAX	00000117
EBX	00002F3E

Вариант 1:

```
PUSH    EAX             ; передать EAX в стек
MOV     EAX,EBX         ; передать содержимое EBX в
                        ; EAX
POP     EBX             ; передать старое значение EAX
                        ; из стека в EBX
```

Вариант 2:

```
XCHG EAX,EBX            ; обменять содержимое EAX и
                        ; EBX
```

Состояние после обмена:

EAX	00002F3E
EBX	00000117

3.2. АРИФМЕТИЧЕСКИЕ КОМАНДЫ

Эти команды применяются для выполнения арифметических операций над знаковыми или беззнаковыми числами. Они часто встречаются в программах.

ADD — суммирует два операнда, помещая результат в первый операнд (получатель).

SUB — вычитает один операнд (источник) из другого (получатель). Разность замещает место получателя.

INC — увеличивает операнд на 1, не воздействуя на флажок переноса. Применяется в циклах для инкремента индекса.

DEC — уменьшает операнд на 1, не воздействуя на флажок переноса. Применяется в циклах для декремента индекса.

MUL — эта простая команда умножает беззнаковые целые числа. Она имеет один операнд-источник. Еще два операнда умножения подразумеваются размером операнда-источника.

IMUL — умножает знаковые целые числа. Это более гибкая и сложная команда. Она имеет 4 базовые формы по числу и типу операндов (один, два или три операнда).

DIV — осуществляет деление целых беззнаковых чисел. В делении есть фактически 4 операнда: делимое, делитель, частное и остаток. Определяется только местоположение делителя. Все остальные операнды определяются неявно, в зависимости от размера делителя.

IDIV — аналогична команде DIV, но работает со знаковыми числами.

NEG — изменяет знак операнда, находящегося в регистре или памяти.

CMP — аналогична команде SUB, но не сохраняет результат. Команда сравнивает два числа для последующего условного перехода.

ADC — действует как команда ADD, но прибавляет значение переноса к сумме. Удобна для арифметики повышенной точности.

SBB — действует как команда SUB, но вычитает из разности значение переноса. Удобна для арифметики повышенной точности.

Следующий пример показывает различие между командами ADD и ADC.

Исходное состояние:

EAX	FFFFFFF0	MEMLOC	000027DA
EDX	00002F3E	MEMLOC+4	00000117

Вариант 1:

```
ADD     EAX,MEMLOC    ; сложить первую пару
ADD     EDX,MEMLOC+4  ; сложить вторую пару
```

Состояние после операции:

EAX	000027CA	MEMLOC	000027DA
EDX	00003055	MEMLOC+4	00000117

Вариант 2:

```
ADD     EAX,MEMLOC     ; сложить младшие числа
ADC     EDX,MEMLOC+4   ; сложить старшие числа
```

Состояние после операции:

EAX	000027CA	MEMLOC	000027DA
EDX	00003056	MEMLOC+4	00000117

Следующий пример показывает различие в командах умножения знаковых и беззнаковых чисел.

Исходное состояние:

AL	84
DL	12

Вариант 1:

```
MUL     AL,DL          ; 132×18=2376
```

Состояние после операции:

AX	0948

Значение в AX равно 948H или 2376.

Вариант 2:

```
IMUL        AL,DL          ; -124×18=-2232
```

Состояние после операции:

AX	F748

Следующий пример показывает различие в командах деления знаковых и беззнаковых чисел:

Исходное состояние:

AX	04BF
DL	9A

Вариант 1:

```
DIV         AL,DL          ; 1215/154=7, осталось 137
```

Состояние после операции:

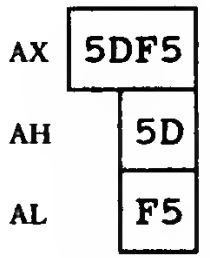
AX	8907
AH	89
AL	07

Регистр AL содержит частное, а регистр AH — остаток 89H=137.

Вариант 2:

```
IDIV AL,DL          ; 1215/(-102)=-11, остаток 93
```

Состояние после операции:



Регистр AL содержит F5H или -11, т. е. частное; в регистре AH находится остаток 5DH=93.

3.3. КОМАНДЫ ПРЕОБРАЗОВАНИЯ ДАННЫХ

Команды этой группы применяются для преобразования типов данных. Большинство из них работает со знаковыми числами, а одна пригодна и для беззнаковых чисел.

MOVSX — команда пересылки с расширением знака, передает операнд-источник в получатель, расширяя знаковый бит источника в старшую часть получателя.

MOVZX — аналогична предыдущей команде, но старшая часть получателя устанавливается равной нулю.

CBW — команда преобразования байта в слово, имеет источником регистр AL, а получателем регистр AH, и действует аналогично команде MOVSX. Знаковый бит AL расширяется в AH, преобразуя знаковый байт из AL в знаковое слово в AX.

CWDE — преобразует знаковое слово из AX в знаковое двойное слово в EAX, расширяя знаковый бит AX в старшую половину EAX.

CWD — также преобразует знаковое слово в знаковое двойное слово, но результат, в отличие от команды CWDE, помещается в два регистра. Знаковый бит из AX расширяется в регистр DX.

CDQ — эта команда заполняет регистр EDX знаковым битом регистра EAX.

Действие команд этой группы поясняют следующие примеры:

Исходное состояние:

EAX	FACEFFF0	MEMLOC	FF3E
EBX	5A5A5A5A	MEMLOC2	07DB
ECX	ACACACAC		
EDX	12345678		

Команды:

```
MOVSX   EAX,MEMLOC    ; передать с расширением знака
MOVZX   EBX,MEMLOC    ; передать с нулевым расширением
MOVSX   ECX,MEMLOC2   ; передать с расширением знака
MOVZX   EDX,MEMLOC2   ; передать с нулевым расширением
```

Состояние после операции:

EAX	FFFFFF3E	MEMLOC	FF3E
EBX	0000FF3E	MEMLOC2	07DB
ECX	000007DB		
EDX	000007DB		

3.4. КОМАНДЫ ДЕСЯТИЧНОЙ АРИФМЕТИКИ

Прямой поддержки десятичной арифметики 80386 не обеспечивает, но имеет команды десятичной коррекции, действующие с обычными арифметическими командами. Есть два типа команд десятичной коррекции: команды ASCII-коррекции работают с одной цифрой в байте, а команды десятичной коррекции — с двумя цифрами.

AAA, AAS, AAM, AAD — команды ASCII-коррекции для арифметических операций. Три из них применяются после обычной арифметической команды, корректируя полученный результат: AAA — после сложения, AAS — после вычитания и AAM — после умножения. Команда AAD — коррекции деления выполняется до операции и подготавливает операнды к делению.

DAA, DAS — команды десятичной коррекции для сложения и вычитания, применяются аналогично предыдущим.

Следующий пример показывает действие команды AAA.

Исходное состояние:

AX | 0009

Команды:

```
ADD     AL,8        ; 9+8=11H(17)
AAA                 ; коррекция
```

Состояние после операции:

AX | 0107

3.5. ЛОГИЧЕСКИЕ КОМАНДЫ

Команды этой группы применяются для производства булевых операций и обеспечивают работу с двоичными полями в байтах, словах и двойных словах.

AND — выполняет логическую функцию И для двух операндов и удобна для установки двоичного поля в нуль.

OR — выполняет логическую функцию **ИЛИ** для двух операндов. Обычно применяется для установки двоичного поля в нужное состояние (поле предварительно очищается командой AND).

NOT — инвертирует биты своего операнда.

TEST — команда логического сравнения аналогична команде AND, но результат не сохраняется. Применяется для проверки двоичного поля на нуль (или не нуль).

XOR — выполняет логическую функцию **ИСКЛЮЧАЮЩЕГО ИЛИ** для двух операндов. Применяется для инвертирования в двоичном поле только определенных бит.

SETxx — эти команды применяются для сохранения результата некоторого сравнения. Значение «xx» определяет условие сравнения. Если сравнение истинно, то получатель устанавливается в 1; а если сравнение ложно, получатель устанавливается на 0.

Следующий пример показывает действие четырех основных логических команд с идентичными операндами. Вы можете для контроля преобразовать 16-ричные числа в двоичные. Пример позволяет лучше разобраться в операциях.

Исходное состояние:

AX	AAAA

MEMLOC1	C3C3
MEMLOC2	C3C3
MEMLOC3	C3C3

Команды:

```
AND     MEMLOC1,AX
OR      MEMLOC2,AX
XOR     MEMLOC3,AX
NOT     AX
```

Состояние после операции:

AX	5555	MEMLOC1	8282
		MEMLOC2	EBEB
		MEMLOC3	6969

3.6. КОМАНДЫ СДВИГА И ЦИКЛИЧЕСКОГО СДВИГА

Эти команды позволяют передвигать биты внутри любого из стандартных типов данных. Они применяются для ускорения операций умножения и деления целых чисел и для образования двоичных полей.

SHR, SHL — команды логических сдвигов «вдвигают» нули с одного конца операндов, а биты с другого конца «выдвигаются».

SAR, SAL — команды арифметического сдвига. Арифметический сдвиг влево аналогичен логическому сдвигу, а при арифметическом сдвиге вправо происходит копирование знакового бита. Это удобно для деления знаковых чисел на степени числа 2.

ROR, ROL — при циклическом сдвиге биты данных не теряются: выдвигаемый бит помещается на место освобождающегося (вдвигаемого).

RCL, RCR — это команды циклического сдвига, включающего перенос. Выдвигаемый бит операнда помещается во флажок переноса, а старое значение флажка переноса передается в освобождающийся бит. Обычно применяются в операциях повышенной (кратной) точности.

SHRD, SHLD — это команды сдвигов двойной точности, но их действие отличается от команд циклического сдвига, включающих перенос. Эти команды имеют три операнда: источник, получатель и счетчик сдвигов. Получатель сдвигается, и выдвигаемые биты теряются, а освобождающиеся биты заполняются битами из источника. Сам источник не изменяется.

Следующий пример показывает различие между обычным циклическим сдвигом и циклическим сдвигом, исключающим перенос.

Исходное состояние:

AX	6699
FLAGS	0202

Вариант 1:

```
ROR     AX,1    ; устанавливает
                ; переполнение, перенос
                ; не изменяется
```

Состояние после операции:

AX	B34C
FLAGS	0A02

Вариант 2:

```
RCR     AX,1    ; сбрасывает
                ; переполнение,
                ; устанавливает перенос
```

Состояние после операции:

AX	334C
FLAGS	0A03

3.7. КОМАНДЫ ОПЕРАЦИЙ НАД БИТАМИ

Команды этой группы очень удобны для манипуляций отдельными битами, допуская их установку и проверку.

BT, BTS, BTR, BTC. Команда проверки бита BT просто помещает значение указанного бита во флажок переноса. Три другие команды выполняют эту же функцию, но позволяют адресуемый бит установить в 1 (BTS), сбросить в 0 (BTR) или инвертировать (BTC).

BSF, BSR — две команды сканирования бита: BSF (вперед) и BSR (назад) — находят первую 1 в операнде, начиная с младшего бита (BSF) или старшего бита (BSR). Обе команды удобны при работе с двоичными образами. Команда BSR применяется также при вычислении двоичных логарифмов.

Следующий пример показывает действие команд BSR, BTC и BSR (число 22 в команде BTC десятичное).

Исходное состояние:

EAX	004037BF
EBX	00000000
ECX	00000000
FLAGS	0242

Команды:

```
BSR     EBX,EAX     ; EBX=номер бита
                    ; старшей 1
BTC     EAX,22      ; инвертировать
                    ; бит 22
BSR     ECX,EAX     ; EBX=номер бита
                    ; старшей 1
```

Состояние после операции:

EAX	000037BF
EBX	00000016
ECX	0000000D
FLAGS	0203

3.8. КОМАНДЫ УПРАВЛЕНИЯ ФЛАЖКАМИ

Команды этой группы позволяют проверять и изменять флажки процессора 80386. Одни команды работают с отдельными флажками, другие — со всеми одновременно.

На отдельные флажки воздействуют 7 команд. Команды **CLD** и **STD** сбрасывают и устанавливают флажок направления. Команды **CLI** и **STI** сбрасывают и устанавливают флажок направления. Команды **CLC** и **CTC** сбрасывают и устанавливают флажок переноса, а команда **CMC** инвертирует его.

Команда **LAHF** загружает младший байт флажков в регистр **AH**. Команда **SAHF** производит обратную передачу. Команды **PUSHF** и **PUSFD** включают регистры Flags и EFlags в стек. Соответствующие операции извлечения из стека реализуют команды **POPF** и **POPFD**.

Следующий пример показывает действие некоторых команд управления флажками.

Исходное состояние:

EAX	000037BF
FLAGS	0A93

Команды:

```
PUSHF                   ; сохранить флажки
                        ; в стеке
SUB       EAX,EAX       ; OF, SF, AF, CF=0;
                        ; ZF, PF=1
POPF                    ; вернуть старые
                        ; флажки из стека
STD                     ; установить флажок
                        ; направления
```

Состояние после операций:

EAX	00000000
FLAGS	0E93

3.9. ЦИКЛИЧЕСКИЕ КОМАНДЫ

Часто приходится обрабатывать большие блоки данных, и в этом помогают команды данной группы. Одна циклическая команда может реализовать цикл, требующий обычно нескольких команд.

Каждая из этих команд обычно выполняет одну операцию пересылки, сравнения, загрузки, запоминания или сканирования. Для адресации источника применяется регистр (E)SI, а получателя — регистр (T)DI. После команды оба регистра модифицируются для адресации следующего элемента цикла. Данные команды наиболее полезны при использовании с одним из префиксов повторения.

REP, REPE, REPZ, REPNE, REPNZ — префиксы повторения, обеспечивают выполнение следующей за ними команды заданное число раз. Префикс повторения REP вызывает выполнение команды столько раз, сколько определено содержимым регистра ECX. Аналогичные префиксы REPE и REPZ вызывают повторение команды либо до исчерпания счетчика, либо до установки флажка Z в нуль. Префиксы REPNE и REPNZ действуют аналогично REPE, но выход происходит при установке флажка Z в 1.

MOVS — эта команда просто пересылает фрагмент данных из одной области памяти в другую. Необходимо тщательно проанализировать возможность перекрытия фрагментов и соответственно определить состояние флажка направления.

CMPS — сравнивает два фрагмента (массива данных). Для окончания сравнения в нужной точке применяются префиксы REPE или REPNE.

STOS — полезна для заполнения фрагмента константой. Она передает содержимое соответствующей части регистра EAX в каждый элемент фрагмента.

SCAS — сравнивает каждый элемент данных с частью регистра EAX. Обычно с этой командой применяется префикс REPE или REPNE.

LODS — эта команда необычна в том, что она бесполезна с префиксом повторения. Она просто загружает следующий элемент данных в некоторую часть EAX.

Следующий пример показывает общие принципы группы циклических команд.

Предположения:

STRING1 имеет смещение 100H в сегменте 151H.
STRING2 имеет смещение 105H в сегменте 151P.
Флажок D содержит нуль (вперед).

Состояние до операции:

ESI	00000200
EDI	00000300
DS	0151
ES	011C

STRING1	A1	A2	A3	A4	A5
STRING2	FF	FF	FF	FF	FF

Команды:

```
LEA     SI,STRING1      ; адрес источника
LEA     DI,STRING2      ; адрес получателя
MOV     ECX,5           ; счетчик
REP MOVSB               ; переслать массив байт
```

Состояние после операции:

ESI	00000105
EDI	0000010A
DS	0151
ES	0151

STRING1	A1	A2	A3	A4	A5
STRING2	A1	A2	A3	A4	A5

3.10. КОМАНДЫ УПРАВЛЕНИЯ ПРОГРАММОЙ

При создании программ часто приходится изменять последовательный ход исполнения команд. Команды данной группы обеспечивают необходимые средства управления ходом выполнения программы.

JMP — команда безусловного перехода. Обычно в ней указывается метка, определяющая следующую выполняемую команду. Ассемблер формирует нужную форму JMP, зная местонахождение именованного (отмеченного) оператора.

Jxx — команда условного перехода, в которой xx означает код проверяемого условия (см. гл. 4). Обычно перед ней находится команда сравнения или другая команда, которая устанавливает некоторые флажки.

CALL — команда вызова. Применяется для реализации процедур (называемых еще подпрограммами и функциями). Команда вызова передает управление так же, как команда JMP. Однако до ее выполнения в стек включается необходимая информация, чтобы вызванная процедура могла возвратить управление команде, находящейся после команды CALL. Для выполнения возврата управления применяется команда RET.

LOOP, LOOPE, LOOPZ, LOOPNE, LOOPNZ — команды зацикливания. Помогают в реализации программных циклов. Команда LOOP повторяет цикл такое число раз, которое определяется содержимым регистра ECX. В командах LOOPE и LOOPZ имеется допол-

нительное ограничение: цикл заканчивается, когда флажок Z будет нулевым. Команды LOOPNE и LOOPNZ аналогичны предыдущим, но цикл заканчивается, когда флажок Z будет содержать 1.

INT, INTO, IRET, IRETD — последняя группа команд передачи управления. Связана с прерыванием (подробнее см. гл. 5). Команда INT инициирует выполнение системной процедуры. Команда INTO вызывает процедуру прерывания 4, если установлен флажок переполнения. Обычно процедуры обработки прерываний входят в операционную систему. Возврат в вызывающую программу осуществляет одна из команд возврата из прерывания IRET или IRETD.

Пример использования команд управления содержит фрагмент на языке высокого уровня и ассемблерный код. В примере осуществляется суммирование нечетных чисел от 1 до 10.

Код на языке высокого уровня:

```
SUM=0;
DO INDEX=1 TO 10;
 IF (INDEX AND 1)=1 THEN SUM=SUM+INDEX;
END;
```

Ассемблерный код:

```
     MOV     BX,0            ; установить SUM=0
     MOV     AX,1            ; начальное значение
                             ; INDEX
     MOV     ECX,10          ; счетчик повторения
LOOP_START:
     TEST    AX,1            ; проверить младший бит
     JZ      SKIPIT          ; перейти, если число
                             ; четное
     ADD     BX,AX           ; прибавить, если
                             ; нечетное
SKIP_IT:
     INC     AX              ; инкремент числа
     LOOP    LOOP_START      ; зацикливание
     MOV     SUM,BX          ; сохранить результат
```

3.11. КОМАНДЫ ПОДДЕРЖКИ ЯЗЫКА ВЫСОКОГО УРОВНЯ

Обычно команды этой группы генерируются компнляторамн, а программисты на ассемблере нх не применяют. Такне команды упрощают разработку компиляторов н обеспечнвают аппаратную поддержку некоторых общнх операцнй.

BOUND — прнменяется для контроля границ масснва. Операндами служат значенне, ннжняя граннца и верхняя граница. Если значение не находится внутри границ, возникает прерывание.

ENTER, LEAVE — парные команды, которые сокращают подготовительное время в начале процедуры. Команда ENTER организует стек в начале исполнения процедуры для упрощення доступа к аргументам и к переменным во взаимосвязанных процедурах. Команда LEAVE восстанавливает стек, подготавливая возврат управлення.

3.12. КОМАНДЫ УПРАВЛЕНИЯ ПРОЦЕССОРОМ

Команды данной группы управляют действиямн CPU. Обычно они прнменяются для упрощення интерфейса с другимн процессорамн в системе, напрнмер, с 80287.

ESC — префикс ESC информнрует о том, что следующая команда предназначена для сопроцессора.

WAIT — заставляет CPU остановить выполнение команд до получения сигнала BUSY. Применяется при ожиданнн результата от сопроцессора.

LOCK — префикс блокнровкн. Резервнрует снстемную шину за процессором 80386 на время выполнення следующей команды. Применяется для устранения конфликтов на шине в мультипроцессорных системах. *Примечание:* префикс LOCK в 80386 имеет несколько ограничений (см. гл. 4).

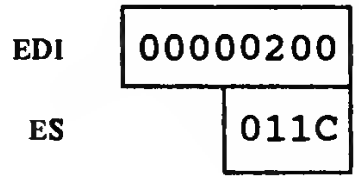
NOP — пустая команда; иногда применяется прн отладке.

HLT — блокирует работу процессора до получения сигнала сброса. Применяется редко.

3.13. КОМАНДЫ ОПЕРАЦИЙ С АДРЕСАМИ

Команды этой группы применяются для загрузки указателей. Имеются два типа команд. К первому типу относится одна команда LEA, которая загружает в регистр смещение указанной ячейки памяти. Она удобна для загрузки индексного регистра. Ко второму типу относятся команды загрузки полного указателя: они загружают сегментный регистр и индексный регистр. В сегментный регистр загружается селектор сегмента ячейки памяти, а в индексный регистр — смещение ячейки памяти. Мнемоника команд имеет вид Lxx, где xx есть имя одного из сегментных регистров.

Исходное состояние:



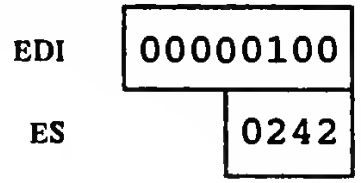
Вариант 1:

```
MOV     ES,MEMLOC       ; загрузить селектор
                        ; в ES
LEA     EDI,MEMLOC      ; загрузить смещение
                        ; в EDI
```

Вариант 2:

```
LES     EDI,MEMLOC      ; загрузить селектор
                        ; в ES, смещение в
                        ; EDI
```

Состояние после операции (оба варианта):



3.14. КОМАНДА ПРЕОБРАЗОВАНИЯ

Команда преобразования XLAT выделена в отдельную группу. Она осуществляет табличное преобразование. Предполагается, что AL содержит байтный индекс таблицы, адресуемой регистром (E)BX. Байт в AL заменяется элементом таблицы. Команда применяется для преобразования символьного кода и синтаксического разбора команд.

ЗАКЛЮЧЕНИЕ

В этой главе дан краткий обзор команд 80386. Она может служить справочником, когда вы знаете, что вам нужно сделать, но не уверены в правильности выбора команды. Теперь вы готовы к восприятию материала главы 4, где приведены более подробные сведения о каждой команде.

Глава 4

ИНСТРУКЦИИ ПРОЦЕССОРА 80386

Выше дан общий подход к некоторым техническим вопросам. В процессоре 80386 реализованы наиболее передовые идеи вычислительной техники, но ради совместимости с его предшественниками он учитывает и «историю» микропроцессоров. Программирование процессора 80386 требует обширных знаний, часть которых изложена в предыдущих главах. На основе этих знаний программист действует как архитектор, разрабатывая программу, которая удовлетворяет интересам пользователей и максимально использует аппаратные ресурсы.

Значительное время уходит на «шлифовку» задачи после окончания разработки программы: передать данные в CPU, сдвинуть, умножить или сложить их, а затем сохранить или распечатать. Здесь на помощь приходит система команд. Знание того, когда пользоваться «молоточком» (например, командой сдвига) или «кувалдой» (командой умножения), сильно влияет на конечный продукт.

Система команд максимально приближает нас к процессору. Знать, какое время требуется для выполнения команды, и иметь общее представления о размерах команды необходимо для опытного программиста на ассемблере. Конечно, часто достаточно написать на ассемблере только критичные фрагменты программы и для этого нужно знать только подмножество команд. Но желающим разрабатывать средние и большие ассемблерные программы нужно внимательно изучить материал данной главы. Он поможет Вам эффективнее работать.

Эффективность опирается на знание скорости выполнения каждого фрагмента программы. Предположим,

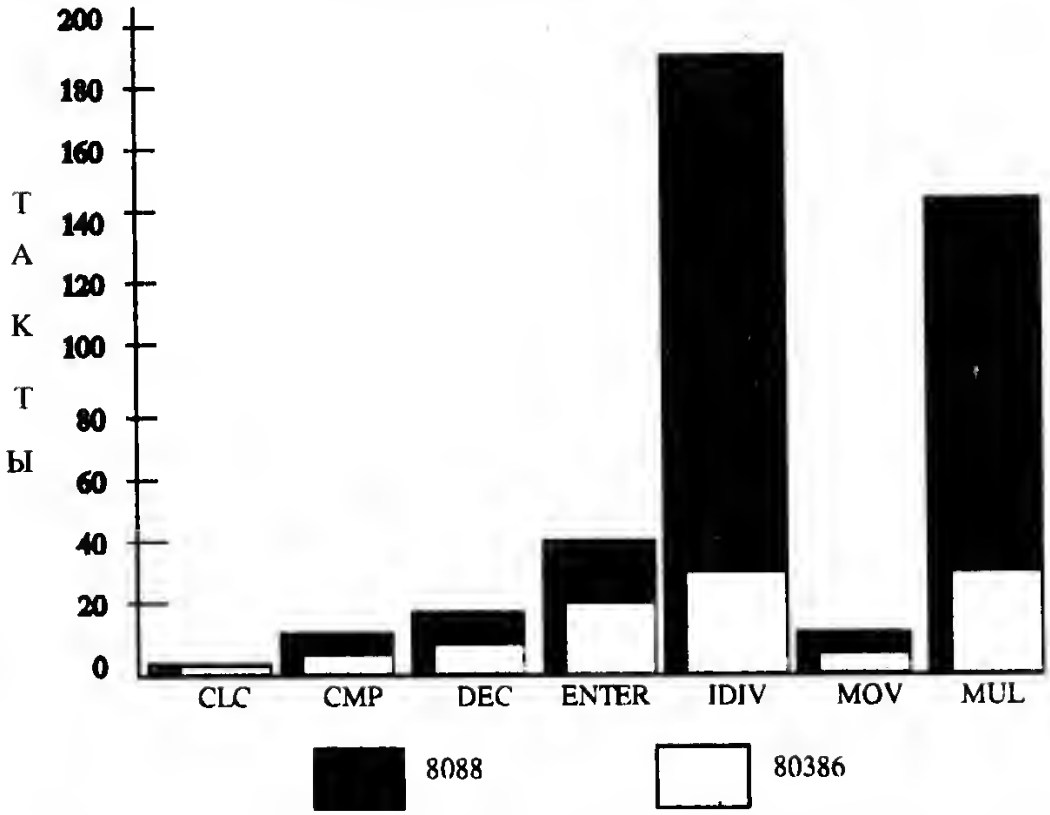


Рис. 4.1. Время выполнения некоторых команд

что вы преобразуете программу для процессора 80386 и хотите по возможности оптимизировать ее. Пусть в программе число умножается на 10 путем сдвига числа на три бита влево и прибавления два раза исходного числа. На это требуется 14 тактов, что эффективнее команды MUL (70 и более тактов), при условии, что все числа в регистрах. Однако в процессоре 80386 вместо 14 тактов требуется 9, а команда IMUL длится всего 10 тактов. В этом случае использование одной команды IMUL предпочтительнее (простота и ясность).

На рис. 4.1 показано, что система команд 80386 по времени выглядит «плоской», т. е. различие во времени выполнения команд не столь велико, как у процессора 8088. Из рисунка видно также, что быстродействие процессора 80386 гораздо выше, чем у 8088. В 80386 команда IDIV длится дольше команды CMP только в 6 раз, а в 8088 — в 19 раз. В старых процессорах было выгодно заменять медленную команду на несколько более быстрых (например, три сложения вместо умножения на 3). На все эти приемы уходит время, и получается запутанная программа. Однако в 80386 такие приемы не требуются, он позволяет применять логически

наиболее подходящую команду и пользоваться всей системой команд, а не только самыми быстрыми командами.

Знать материал данной главы необходимо и для написания более компактных программ. Хорошие ассемблерные программы компактны по двум причинам. Они опираются на самые подходящие команды с учетом времени их выполнения. Учитывается также и фактический размер команды. Команда сложения содержимого двух регистров меньше команды прибавления к регистру непосредственного значения, так как оно входит в команду.

Материал этой главы позволит проверить критические по времени или размеру фрагменты программы. Такую информацию целесообразно давать в документации.

4.1. КАК РАБОТАЕТ ЯЗЫК АССЕМБЛЕР

Язык ассемблер — это не низший уровень программирования, так как ассемблер транслирует команды в рассматриваемые ниже «форматы». Например, команда POPA преобразуется в байт 61H. Другие команды длиннее из-за указания ячеек памяти или регистров.

Ассемблированные команды образуют «объектный код», называемый «машинным языком». Зная материал данной главы, вы можете точно узнать, какой машинный код образует конкретная программа. но он менее эффективен, чем ассемблированный код.

Язык ассемблера первых микропроцессоров был очень простым. Например, микропроцессор 6502 имеет всего два общих регистра. Команды были очень специализированы, например, TXY (передать в регистр Y содержимое регистра X), и каждая имела один машинный эквивалент (или несколько в зависимости от режима адресации). Однако, в процессоре 80386 много регистров и режимов адресации. Для передачи из одного регистра в другой (или в память, или из памяти в регистр) применяется команда MOV с последующим указанием нужных регистров. Следовательно, команда MOV имеет много машинных представлений в зависимости от типа передачи и режимов адресации. Ее различные варианты имеют разные форматы и длины. В языке ассемблер, в отличие

от языка высокого уровня, каждая команда транслируется в одну машинную команду.

Как же «воплотить» язык ассемблер в процессоре? Секрет в «микрокоде», имеющемся в большинстве современных процессоров. Микрокод — это «программа», выполняющаяся в интегральных схемах, которая дешифрирует объектный код и реализует нужные функции. Следовательно, микрокод — это средство разработчиков схем процессора, позволяющее упростить интегральную схему и организовать ее модули.

4.2. ЧТО ТАКОЕ ФОРМАТ КОМАНДЫ?

Формат команды — это просто ее двоичное представление. Обычно при программировании на ассемблере не обязательно знание форматов. Однако при расшифровке 16-ричных распечаток требуется знание форматов команд; оно необходимо и при внесении в программу «заплат».

Большинство команд 80386 имеет несколько форматов. Для процессора операции с числами из памяти намного сложнее, чем использование содержимого двух регистров. Например, схема опережающей выборки попытается «украсть» циклы шины для считывания операнда во время выполнения другой команды, что повлияет на эффективность опережающей выборки. Ассемблеры для 80386 позволяют программисту не касаться внутренних операций процессора. Для лучшего понимания форматов команд рассмотрим элементы машинного языка.

Первой располагается сама команда, часто называемая кодом операции КОП. Код сообщает компьютеру, что нужно делать: переслать информацию, сложить два числа, осуществить переход. Иногда для выполнения команды достаточно одного кода операции. Например, команда установки бита переноса STC требует только кода операции, так как ее действие подразумевается самой командой. Формат команды очень простой 0F9H. Она выполняется всего за два такта синхронизации.

Но кроме КОП компьютеру нужно знать, где находятся операнды команды. Для кодирования этой информации в процессоре 80386 применяются два способа.

В некоторых командах операнд-регистр определяется битами в коде операции. Команда PUSH имеет один

операнд. Если в стек включается содержимое регистра, то номер регистра находится в коде операции: 50H включает в стек AX, а 51H — регистр CX. Номер бита содержится в одном байте и выполняется за два такта. Это более простой способ.

В командах с несколькими операндами или с операндом в памяти местоположение операнда определяется сложнее. В них после байта кода операции имеется еще один байт ModRM (см. рис. 4.1а).

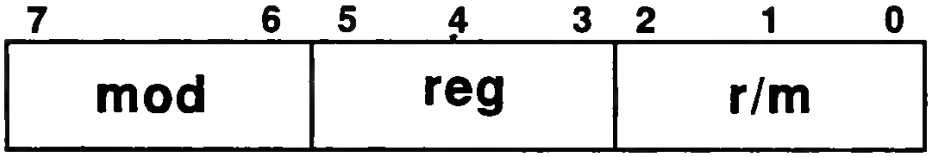


Рис. 4.1,а. Байт ModRM

Три поля в байте ModRM играют свою роль в определении операндов команды.

Поле r/m в трех младших битах показывает либо один из 8 регистров общего назначения, либо один из 24 режимов адресации в зависимости от содержимого поля mod.

Поле reg в командах с двумя операндами определяет номер регистра. В однооперандных командах это поле действует как расширение кода операции, уточняя смысл команды.

Из четырех значений в поле mod одно показывает, что поле r/m содержит номер регистра. Три значения выбирают одну из трех групп восьми режимов адресации.

Вернемся к команде PUSH. Если в стек включается число из RAM, нужно сообщить компьютеру, где его найти. Команда принимает вид 0FFH mod 6 r/m. Сама команда занимает первый байт и три бита в середине второго байта (110). Биты «mod r/m» сообщают режим адресации, т. е. откуда взять число. Так определяется адрес RAM. С учетом этих действий неудивительно, что команда выполняется всего за 5 тактов. (В 8088 требуется более 16 тактов).

Некоторые режимы адресации требуют больше информации для локализации операнда. В этом случае об этом сообщает байт ModRM, а за ним находится еще один байт SIB (Scale — масштаб, Index — индекс, Base — база), см. формат на рис. 4.1б.

Рис. 4.1,6. Байт SIB

Как и в байте ModRM, три поля определяют различные элементы информации о режиме адресации.

Поле «база» в трех младших битах определяет регистр, который используется как базовый регистр.

Поле «индекс» из трех бит определяет индексный регистр.

Поле «масштаб» в старших двух битах задает масштабный множитель для индексной адресации.

По форматам команд 80386 опытный программист может узнать многое. Например, новые команды проверки бита BT вроде бы должны быть максимально короткими. Эти команды применяются часто и выполняют очевидные функции. Однако, все эти команды начинаются с байта 0FH и требуют байт для определения команды и еще байты для адреса и непосредственных данных. В этом случае лишний байт увеличивает время выполнения команды на один такт.

Указания по расшифровке информации о формате команд даны перед рассмотрением самих команд.

4.3. ВРЕМЕННАЯ ИНФОРМАЦИЯ

Информация о времени выполнения команд варьируется от точной до неопределенной. Конечно, команды типа CLC (сбросить флажок переноса) всегда выполняются за одно и то же время (2 такта). Время выполнения команд с операндами в регистре или памяти значительно изменяется. По существу, одна такая команда фактически представляет несколько команд, в зависимости от местонахождения операндов. Понятие одной команды удобно для программиста, но оно уводит нас от реализации операций внутри процессора. Попытка вычислить время выполнения команды связана со значительными усилиями. Далее дается подробная информация о каждой команде.

Обычная работа: операционное устройство использует очереди

5 % времени: операционное устройство опережает очереди

Рис. 4.2. Очереди и скорость выполнения

Временная информация оказывается неточной: большинство программ выполняется примерно на 5% дольше. Причина этого скрыта в особенностях схемы процессора 80386.

Приводимые ниже времена предполагают, что конвейер всегда действует. Конвейеризация — это возможность одновременно выбирать одну команду, дешифрировать вторую и исполнять третью. Для этого введены очереди: они содержат несколько выбранных команд, ожидающих дешифрирования, и несколько дешифрированных команд, ожидающих исполнения. Фактически же некоторые команды исполняются быстрее, чем выбираются и дешифрируются. Если несколько таких команд идут подряд, очереди медленно освобождаются, когда командное устройство проходит через простой код. Когда очереди пустые, командное устройство должно ожидать минимальное время от части времени выборки и дешифрирования до выполнения своих операций. В этом случае быстродействие процессора 80386 уменьшается (см. рис. 4.2). Любая команда передачи управления (JMP, CALL, RET и др.) также освобождает очереди, так как устройство выборки не знает, откуда считывать следующую команду.

В результате этого приводимые далее времена не являются точными (и это не «сбои» в процессоре 80386,

а отход от конвейерной работы). Если очередь команд пуста, когда операционное устройство готово, следующая команда выполняется дольше, чем обычно. Потеря времени зависит от того, когда дешифрированная команда попадет в очередь. В среднем нарушения работы конвейера увеличивают время выполнения программы примерно на 5%. При наличии кэш-памяти увеличение будет меньше, так как выборка происходит быстрее, чем обычно. Конечно, понятия «средней» программы нет, поэтому для каждой конкретной программы будет свое увеличение времени выполнения.

Знание точного времени выполнения программы вряд ли потребуется. «Средний» программист может просто добавить 5% к вычисленному времени. Почти никогда не следует пытаться уменьшить эти потери, переупорядочивая команды.

Кроме того, на расчет времени выполнения программ сказывается различие в быстродействии между реальными и защищенными режимами. Программист при разработке программы может не знать, в каком режиме она будет выполняться, а от этого зависит время выполнения некоторых команд. Примерами служат команды загрузки указателей (LDS, LES, LFS, LGS и LSS) и межсегментные передачи управления JMP, CALL и RET (условные переходы вида JNE не пересекают границ сегментов).

4.4. 80386 И ДРУГИЕ ПРОЦЕССОРЫ iAPX 86

Необходимо знать, какие команды на каких микропроцессорах работают, чтобы обеспечить транспортабельность программ.

В обзоре команд в скобках показано, в каком процессоре впервые появилась каждая команда. Система команд каждого нового процессора является надмножеством системы команд предыдущего, что обеспечивает совместимость вверх. Иногда процессоры совместимы по языку ассемблера, но не по объектному коду. В этом случае придется заново ассемблировать программу; правда, получить копию исходного кода не так просто.

В каждом процессоре появились новые команды. Примерно 90% команд 80386 неизменны и совместимы с

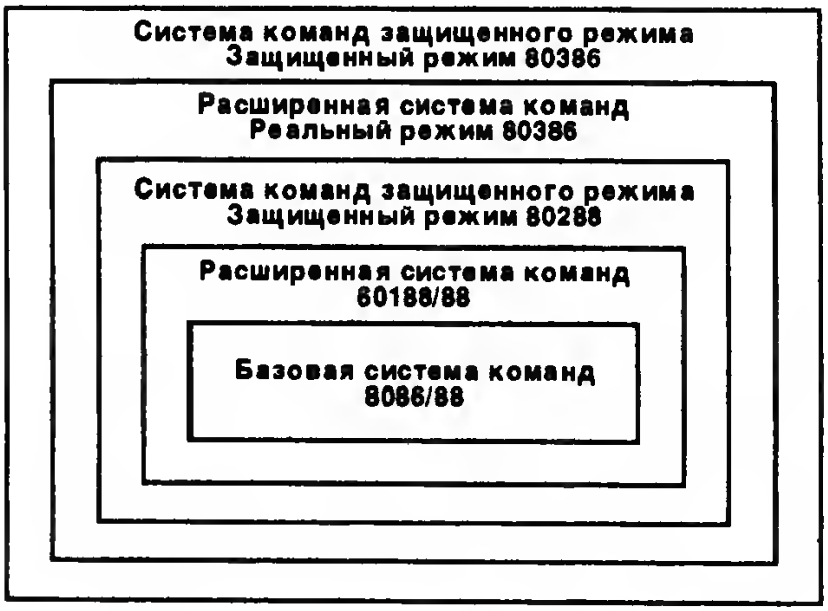


Рис. 4.3. Системы команд семейства iAPX 86

8086/8088 (с учетом возможности работы с 32-битными операндами). Из 10% часть команд допускают новые режимы адресации и средства защиты, а некоторые команды совершенно новые.

На рис. 4.3 показаны взаимосвязи между системой команд процессора 80386. Большинство команд совместимы с 8086/8088. В процессоры 80186/188 введено несколько новых команд и повышено их быстродействие.

В процессоре 80286 нет новых обычных команд, но введены команды управления памятью и защиты программ. Эти команды редко используются прикладными программистами. Однако ограничение адресного пространства 640 Кбайтами (MS-DOS накладывает на IBM PC) до появления машин с 386-м процессором не было преодолено.

В процессор 80386 введено много новых обычных команд, особенно для операций с отдельными битами. Он имеет 32-битные тракты данных, но допускает обработку байт и слов. Эти команды действуют в реальном режиме.

Защищенный режим 80386 предлагает обширный набор команд управления памятью, защиты, организации виртуальной памяти и отладки.

4.5. СИСТЕМА КОМАНД

Далее дается подробная информация о всех командах 80386. Команды упорядочены по алфавиту, что упрощает поиск нужной команды.

Нужно внимательно изучить вводную информацию о форматах команд. Она необходима для понимания используемых сокращений.

В табл. 4.1 показан список логических функций. Некоторые команды прямо реализуют эти функции, а многие опираются на них в своем описании.

Таблица 4.1. Логические операции над тетрадами

P	Q	P and Q	P or Q	P xor Q	Not P	Not Q
1	1	1	1	0	0	0
1	0	0	1	1	0	1
0	1	0	1	1	1	0
0	0	0	0	0	1	1

Описания команд даются в однообразном формате и состоят из нескольких частей. Ниже показаны сокращения и форматы, принятые в описании.

Первая строка содержит ассемблерную мнемонику, описание команды и процессор iAPX, в котором появилась команда.

Первая основная часть содержит различные коды операции, ассемблерные форматы и время выполнения, выраженное в тактах синхронизации. Для циклических команд приведено время с префиксом REP и без него. Некоторые команды, например, JMP и CALL, имеют дополнительный столбец, поясняющий тип команды.

Приведем смысл значений в столбце кода операции:

hh	Две 16-ричные цифры, определяющие точный байт.
[r]	Стандартный байт ModRM; после него могут идти байт SIB и/или смещение памяти.
[n]	Когда «n» есть цифра от 0 до 7, это байт ModRM с полем reg, содержащим эту цифру (цифра — это расширение кода операции).
ib	Непосредственный байт.
iw	Непосредственное слово.
id	Непосредственное двойное слово.

db 8-битное знаковое значение, прибавляемое к (E)IP для получения адреса перехода в командах перехода и вызова.

dw 16-битное знаковое значение, прибавляемое к (E)IP для получения адреса перехода в командах перехода и вызова.

d Смещение операнда в памяти в конкретном сегменте; длина 16 или 32 бита.

dd 32-битное знаковое значение, прибавляемое к (E)IP для получения адреса перехода в командах перехода и вызова.

pd 32-битный указатель; первые 16 бит — это селектор сегмента, а вторые 16 бит — смещение в сегменте.

pp 48-битный указатель; первые 16 бит — это селектор сегмента, а следующие 32 бита — смещение в сегменте.

В столбце формата применяются такие же обозначения, как и в столбце кода операции для представления операндов команды. Смысл их тот же. Иногда в этом столбце применяются имена регистров или явные числа для представления операндов. Применяются также дополнительные обозначения:

r/mb Операндом является содержимое байтного регистра или ячейки памяти, определяемых байтом ModRM.

r/mw Операндом является содержимое словного регистра или ячейки памяти, определяемых байтом ModRM.

r/md Операндом является содержимое двухсловного регистра или ячейки памяти, определяемых байтом ModRM.

rb Операндом является содержимое байтного регистра, определяемого байтом ModRM.

rw Операндом является содержимое словного регистра, определяемого байтом ModRM.

rd Операндом является содержимое двухсловного регистра, определяемого байтом ModRM.

mb Операндом является содержимое байта в ячейке памяти, определяемой байтом ModRM.

mw Операндом является содержимое слова в ячейке памяти, определяемой байтом ModRM.

md Операндом является содержимое двойного слова в ячейке памяти, определяемой байтом ModRM.

mw:w Операндом является указатель в ячейке памяти, определяемой байтом ModRM; указатель содержит 16 бит селектора сегмента и 16 бит смещения.

mw:d Операндом является указатель в ячейке памяти, определяемой байтом ModRM; указатель содержит 16 бит селектора сегмента и 32 бита смещения.

Столбец «такты» содержит время выполнения команды в тактах синхронизации. Если два значения разделены наклонной чертой, то первое относится к операндам в регистрах, а второе — к операндам в памяти. Иногда здесь приводятся формулы, которые сразу же поясняются. Например, команды передачи управления обычно содержат элемент «7+m». Здесь «m» означает число компонент в следующей выполняемой команде. Компонентом могут быть: байт префикса, каждый байт кода операции, любой байт ModRM, любой байт SIB, любое значение смещения или любое непосредственное значение.

В следующей части показано, как команда влияет на флажки. Пробел означает, что флажок не изменяется. Цифры 0 и 1 показывают новое состояние флажка. Буква «U» означает, что состояние флажка не определено, а буква «S» — флажок устанавливается или сбрасывается в зависимости от результата команды.

Следующая часть содержит псевдокод, т. е. полуформальное описание операции команды. Такое алгоритмическое описание взято из языков высокого уровня.

Далее подробно описывается функция команды. Приведены необходимые пояснения о времени выполнения, кодах операции и операндах.

Затем описаны все особые случаи, которые могут возникнуть при выполнении команды. Режим обозначается одной буквой: «P» — защищенный режим, «R» — реальный режим и «V» — режим виртуального 8086.

Примечания для пользователя содержат дополнительную информацию о команде: применения, тонкости и вопросы совместимости.

В заключение дается пример использования команды. Он носит иллюстративный характер и не всегда соответствует практике программирования.

AAA ASCII-коррекция AL для сложения (8086)

КОП	Формат	Такты
37	AAA	4

Флажки

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0			U				U	U	0	S	0	U	1	S

Псевдокод

```
IF (младшая тетрада AL>9) ИЛИ (AF=1) THEN
   инкремент AL на 6
   сбросить старшую тетраду AL на 0
   инкремент AH на 1
   установить CF и AF
ELSE
   сбросить CF и AF
END IF
```

Операция

Превращает содержимое AL в неупакованное десятичное слово.

Особые случаи

Нет.

Примечания

Применяется после сложения десятичных цифр и преобразует результат сложения в десятичную цифру.

Пример

```
MOV     AX,8    ; Загружает 8 в AL и AH=0.
ADD     AL,6    ; Прибавить к AL число 6.
AAA             ; AX=104H, CF=AF=1.
```

AAD ASCII-коррекция AL до деления (8086)

КОП	Формат	Такты
D5 0A	AAD	19

Флажки

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0			U				S	S	0	U	0	S	1	U

Псевдокод

```
Установить AL на (AL+(10×AH))
Установить AH на 0
```

Операция

Превращает неупакованное десятичное число в двоичное. Максимальное десятичное значение в 16-битном регистре 99. Оно помещается в AL, поэтому AH устанавливается на 0.

Особые случаи

Нет.

Примечания

Используется до деления неупакованных десятичных чисел. Применяется редко.

Команда действует только на BCD-цифру. Для преобразования AL из ASCII в BCD воспользуйтесь командой AND AL,15.

Пример

```
MOV AX,1405H ; Загружает делимое (45) в AX.
MOV BL,3     ; Загружает делитель в BL.
AAD          ; Результат 45 (десятичный).
IDIV BL      ; Осуществляет деление, результат
             ; 15.
AAM          ; AX содержит 105H или 15 в BCD.
```

AAM ASCII-коррекция AL после умножения (8086)

КОП	Формат	Такты
D4 0A	AAM	17

Флажки

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0			U				S	S	0	U	0	S	1	U

Псевдокод

```
Разделить AL на 10
Поместить частное в AH
Поместить остаток в AL
```

Операция

Преобразует число из AL, меньшее, чем 100, в неупакованное BCD-число в AX.

Особые случаи

Нет.

Примечания

Используется после умножения неупакованных BCD-цифр. Результат MUL всегда не больше, чем 9×9=81, поэтому условие AL<100 удовлетворяется. Применяется редко.

Команда работает только с BCD-числами. Для преобразования AL из ASCII в BCD воспользуйтесь командой AND AL,15.

Пример

См. пример для команды AAD.

AAS ASCII-коррекция AL после вычитания (8086)

КОП	Формат	Такты
3F	AAS	4

Флажки

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0			U				U	U	0	S	0	U	1	S

Псевдокод

```
IF (младшая тетрада AL>9) OR (AF=1) THEN
   декремент AL на 6
   установить старшую тетраду AL на 0
   декремент AH на 1
   установить флажки AF и CF
ELSE
   сбросить флажки AF и CF
END IF
```

Операция

Команда изменяет AL на неупакованное десятичное число и корректирует его с привлечением AF. Значение AF и размер младшей тетрады AL показывают, нужен ли десятичный перенос.

Особые случаи

Нет.

Примечания

Команда используется после вычитания BCD-цифр и преобразует результат двоичного вычитания в BCD-формат (учитывая вычитание). Применяется редко.

Команда AAS осуществляет преобразование двоичного числа в BCD. Для преобразования AL в ASCII воспользуйтесь командой OR AL,48.

Пример

```
MOV AX,205H  ; Загружает BCD 25 в AX.
SUB AL,8     ; Вычитает 8 из AL (давая OFDH).
AAS          ; AX содержит 107H, т. е. 17.
             ; Оба флажка CF и AF содержат 1.
```

ADC Сложение с переносом (8086)

КОП	Формат	Такты
14 ib	ADC AL,ib	2
15 iw	ADC AX,iw	2
15 id	ADC EAX,id	2
80 [2] ib	ADC r/mb,ib	2/7
81 [2] iw	ADC r/mw,iw	2/7
81 [2] id	ADC r/md,id	2/7
83 [2] ib	ADC r/mw,ib	2/7
83 [2] ib	ADC r/md,ib	2/7
10 [r]	ADC r/mb,rb	2/7
11 [r]	ADC r/mw,rw	2/7
11 [r]	ADC r/md,rd	2/7
12 [r]	ADC rb,r/mb	2/6
13 [r]	ADC rm,r/mw	2/6
13 [r]	ADC rd,r/md	2/6

Флажки

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0			S				S	S	0	S	0	S	1	S

Псевдокод

```
IF (источник имеет меньше бит, чем получатель)
    THEN
  расширить операнд источника со знаком
END IF
Сложить источник и получатель, поместить результат
    в получатель
Прибавить CF к получателю, поместить результат в
    получатель
```

Операция

Суммирует два числа, являющихся операндами, и значение из бита CF. Первый операнд заменяется результатом, а второй не изменяется.

Особые случаи	Режимы	Причины
#GP(0)	P	Результат в защищенном от записи сегменте
#GP(0)	P	Недопустимый эффективный адрес памяти в сегментах CS, DS, ES, FS, GS
#SS(0)	P	Неверный адрес в сегменте SS
#PF(fc)	P V	Страничное нарушение
INT(13)	R V	Часть операнда вне диапазона адреса от 0 до 0FFFFH

Примечания

Применяется для сложения повышенной точности, чтобы учитывать распространение переноса.

При программировании необходимо внимательно следить за флажком переноса CF.

Пример

```
MOV   AX,956    ; Загружает 3BCH в AX.
MOV   BX,373    ; Загружает 175H в BX.
ADD   AL,BL     ; Суммирует 0BCH и 75H.
                ; Результат 31H, CF=1.
ADC   AH,BH     ; Суммирует 3 и 1, давая 5
                ; (CF=1).
                ; Результат в AX 1329 (531H).
```

ADD Сложение (8086)

КОП	Формат	Такты
04 ib	ADD AL,ib	2
05 iw	ADD AX,iw	2
05 id	ADD EAX,id	2
80 [0] ib	ADD r/mb,ib	2/7
81 [0] iw	ADD r/mw,iw	2/7
81 [0] id	ADD r/md,id	2/7
83 [0] ib	ADD r/mw,ib	2/7
83 [0] ib	ADD r/md,ib	2/7
00 [r]	ADD r/mb,rb	2/7
01 [r]	ADD r/mw,rw	2/7
01 [r]	ADD r/md,rd	2/7
02 [r]	ADD rb,r/mb	2/6
03 [r]	ADD rw,r/mw	2/6
03 [r]	ADD rd,r/md	2/6

Флажки

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0			S				S	S	0	S	0	S	1	S

Псевдокод

```
IF (источник имеет меньше бит, чем получатель)
    THEN
    расширить источник со знаком
END IF
Прибавить источник к получателю, поместить резуль-
    тат в получатель
```

Операция

Суммирует два числа, являющиеся операндами команды. Первый операнд замещается результатом, второй не изменяется.

Особые случаи	Режимы	Причины
#GP(0)	P	Результат в защищенном от записи сегменте
#GP(0)	P	Недопустимый эффективный адрес памяти в сегментах CS, DS, ES, FS, GS
#SS(0)	P	Неверный адрес в сегменте SS
#PF(fc)	P V	Страничное нарушение
INT(13)	R V	Часть операнда вне диапазона адреса от 0 до 0FFFFH

Примечания

Важно понимать, как команда ADD устанавливает флажки.

Предыдущая команда ADC учитывает при сложении значение CF.

По возможности команду ADD следует применять вместо ADC, чтобы избежать трудностей из-за непредсказуемого состояния флажка CF.

Пример

```
MOV  AX,956    ; Загружает 03BC в AX.
MOV  BX,373    ; Загружает 175H в BX.
ADD  AX,BX     ; AX содержит сумму 1329 (531H).
```

AND Логическое И (8086)

КОП	Формат	Такты
24 ib	AND AL,ib	2
25 iw	AND AX,iw	2
25 id	AND EAX,id	2
80 [4] ib	AND r/mb,ib	2/7
81 [4] iw	AND r/mw,iw	2/7
81 [4] id	AND r/md,id	2/7
20 [r]	AND r/mb,rb	2/7
21 [r]	AND r/mw,rw	2/7
21 [r]	AND r/md,rd	2/7
22 [r]	AND rb,r/mb	2/6
23 [r]	AND rw,r/mw	2/6
23 [r]	AND rd,r/md	2/6

Флажки

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0			0				S	S	0		0	S	1	0

Псевдокод

```
REPEAT
   IF если бит получателя 1 и бит источника 1
        THEN
     оставить бит в получателе равным 1
   ELSE
     сбросить бит получателя в 0
   END IF
UNTIL до проверки всех бит в получателе
```

Операция

Команда AND выполняет логическую функцию И над двумя операндами (см. табл. 4.1). Ее можно использовать только с операндами одинакового размера.

Особые случаи	Режимы	Причины
#GP(0)	P	Результат в защищенном от записи сегменте
#GP(0)	P	Недопустимый эффективный адрес памяти в сегментах CS, DS, ES, FS, GS
#SS(0)	P	Неверный адрес в сегменте SS
#PF(fc)	P V	Страничное нарушение
INT(13)	R V	Часть операнда вне диапазона адреса от 0 до 0FFFFH

Примечания

Команда AND часто применяются для сброса определенных бит, например, AND AL,15. Она использовалась в старых процессорах iAPX 86 для проверки бит и функций графики, но теперь эти функции эффективнее реализуют команды проверки бит.

Пример

```
MOV  AX,5963H   ; Загружает 16-ричное число в
                ; AX.
MOV  BX,6CA5H   ; Загружает 16-ричное число в
                ; BX.
AND  AX,BX      ; Теперь AX содержит 4821H.
```

BOUND Контроль попадания в диапазон (80186)

КОП	Формат	Такты
62 [r]	BOUND rw,mw	10*
62 [r]	BOUND rw,md	10*

*Число тактов без вызова прерывания 5.

Флажки

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0									0		0		1	

Псевдокод

```
IF (первый операнд<слова, адресованного вторым операндом) OR
   (первый операнд>слова, следующего после слова, адресуемого вторым операндом)
     THEN
   INT 5
END IF
```

Операция

Применяется для контроля нахождения значения внутри или вне диапазона. Если значение внутри диапазона, никакого действия не предпринимается, иначе генерируется прерывание 5. Команда предназначена для компиляторов с целью контроля индекса массива.

Первый операнд (регистр) содержит индекс массива, а второй служит указателем памяти. В памяти находятся два слова: нижний и верхний индексы массива, т. е. минимальное и максимальное значения индекса, а не адреса концов массива в памяти. Если индекс меньше первого слова или больше второго, генерируется прерывание 5. Команда BOUND не воздействует на флажки и не показывает, какая граница нарушена.

Особые случаи	Режимы	Причины
#GP(0)	P	Недопустимый эффективный адрес памяти в сегментах CS, DS, ES, FS, GS
#SS(0)	P	Неверный адрес в сегменте SS
#PF(fc)	P V	Страничное нарушение
#UD	P	Второй операнд определяет регистр
INT(5)	P R V	Нарушена граница
INT(13)	P V	Часть операнда вне диапазона адреса от 0 до 0FFFFH

Примечания

Команда BOUND удобна для контроля границ массива, но ее реализация запутана. Проще было установить OF, когда нарушена граница, а флажком SF указать неверный индекс. Вместо этого генерируется прерывание и необходимо позаботиться об обработке прерывания. Можно или полностью обработать ошибку, или установить флажок ошибки, вернуть управление программе, а в ней предусмотреть переход по флажку к нужному обработчику ошибки.

Пример

```
MOV     WORD PTR BND,0      ; Нижняя граница
                            ; 0.
MOV     WORD PTR BND+2,99   ; Верхняя граница
                            ; 99.
MOV     AX,100              ; Контролируемое
                            ; значение в AX.
BOUND   AX,BND              ; Вызывает преры-
                            ; вание 5.
```

BSF Сканирование бита вперед (80386)

КОП	Формат	Такты
0F BC [r]	BSF rw,r/mw	10+3n*
0F BC [r]	BSF rd,r/md	10+3n*

*n — число пропущенных нулей

Флажки

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0								S	0		0		1	

Псевдокод

```
IF второй операнд равен нулю THEN
    Установить ZF
    Установить первый операнд на неопределенное
        значение
ELSE
    Сбросить ZF
    Выбрать младший бит (бит 0) второго операнда
    DO WHILE выбранный бит=0 (не делать, если
        бит 0=1)
      Выбрать следующий старший бит
    END DO
    Скопировать индекс выбранного бита в первый
        операнд (индекс равен 0—15 или 0—31)
END IF
```

Операция

Команда BSF находит первый единичный бит во втором операнде (слово или двойное слово в регистре или памяти). Поиск начинается с бита 0 и прекращается при обнаружении 1. Во второй операнд помещается индекс (номер разряда) единичного бита.

Особые случаи	Режимы	Причины
#GP(0)	P	Недопустимый эффективный адрес памяти в сегментах CS, DS, ES, FS, GS
#SS(0)	P	Неверный адрес в сегменте SS
#PF(fc)	P V	Страничное нарушение
INT(13)	P V	Часть операнда вне диапазона адреса от 0 до 0FFFFH

Примечания

Новые команды манипуляций битами упрощают многие программы (например, построение таблиц размещения, в которых состояние бита показывает, заняты или нет область RAM или сектор диска). Команда BSF позволяет быстро найти первый ненулевой бит (возможно, первый распределенный сектор на диске). Простой цикл может найти первое ненулевое слово или двойное слово в таблице распределения секторов, а затем команда BSF найдет первый ненулевой бит.

Пример

```
MOV     BX,3CD0H    ; В BX просматриваемое зна-
                    ; чение.
BSF     AX,BX       ; AX=4, ZF=0.
```

BSR Сканирование бит назад (80386)

КОП	Формат	Такты
0F BD [r]	BSR rw,r/mw	10+3n*
0F BD [r]	BSR rd,r/md	10+3n*

*n — число пропущенных нулевых бит.

Флажки

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0								S	0		0		1	

Псевдокод

```
IF второй операнд равен нулю THEN
    Установить ZF
    Установить первый операнд на неопределенное
        значение
ELSE
    Сбросить ZF
    Выбрать старший бит (бит 15 или 31) второго
        операнда
    DO WHILE выбранный бит=0 (не делать, если
        бит 15=1)
      Выбрать следующий младший бит
    END DO
    Скопировать индекс выбранного бита в первый
        операнд (индекс равен 0—15 или 0—31)
END IF
```

Операция

Команда BSR находит первый единичный бит во втором операнде (слово или двойное слово в регистре или памяти).

Просмотр начинается со старшего бита и останавливается при обнаружении единичного бита. Индекс (номер разряда) единичного бита помещается во второй операнд.

Особые случаи	Режимы	Причины
#GP(0)	P	Недопустимый эффективный адрес памяти в сегменте CS, DS, ES, FS, GS
#SS(0)	P	Неверный адрес в сегменте SS
#PF(fc)	P V	Страничное нарушение
INT(13)	R V	Часть операнда вне диапазона адреса от 0 до 0FFFFH

Примечания

Новые команды манипуляций битами упрощают многие программы (например, построение таблиц размещения, в которых состояние бита показывает, свободны ли область RAM или сектор на диске). Команда BSR позволяет быстро найти первый ненулевой бит (возможно, первый распределенный сектор на диске). Используя обычный цикл можно найти первое ненулевое слово или двойное слово в таблице размещения секторов, а затем командой BSR определить первый ненулевой бит.

Пример

```
MOV   BX,3CD0H ; Просматриваемое значение в
               ; BX.
BSR   AX,BX    ; AX=13, ZF=0.
```

BT Проверка бита (80386)

КОП	Формат	Такты
0F A3 [r]	BT r/mw,rw	3/12
0F A3 [r]	BT r/md,rd	3/12
0F BA [4] ib	BT r/mw,ib	3/6
0F BA [4] ib	BT r/md,id	3/6

Флажки

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0									0		0		1	S

Псевдокод

Использовать первый операнд для локализации регистра или ячейки памяти.
Использовать второй операнд для выбора бита (бит 0 — младший).
Передать выбранный бит в CF.

Операция

Команда BT позволяет выбрать любой бит и передать его во флажок CF. Первый операнд — регистр или ячейка памяти. Второй операнд содержит номер бита. Им может быть любое беззнаковое целое до 8 бит, но в зависимости от размера первого операнда используются только младшие 4 или 5 бит. Проверяемый бит не изменяется.

Особые случаи	Режимы	Причины
#GP(0)	P	Результат в защищенном записи сегменте
#GP(0)	P	Недопустимый эффективный адрес памяти в сегментах CS, DS, ES, FS, GS
#SS(0)	P	Неверный адрес в сегменте SS
#PF(fc)	P V	Страничное нарушение
INT(13)	R V	Часть операнда вне диапазона адреса от 0 до 0FFFFH

Примечания

Новые команды манипуляций битами упрощают многие программы (например, построение таблиц размещения, в которых состояние бита показывает, свободны ли область RAM или сектор на диске). Команда BT обеспечивает быстрый доступ к любому биту в таблице. Она же упрощает управление дисплейным буфером.

Пример

```
MOV     AX,3CD0H    ; Просматриваемое значение в
                    ; AX.
BT      AX,10       ; Устанавливает CF.
```

BTC Проверка бита и инвертирование (80386)

КОП	Формат	Такты
0F BB [r]	BTC r/mw,rw	6/13
0F BB [r]	BTC r/md,rd	6/13
0F BA [7] ib	BTC r/mw,ib	6/8
0F BA [7] ib	BTC r/md,id	6/8

Флажки

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0									0		0		1	S

Псевдокод

Использовать первый операнд для локализации регистра или ячейки памяти.
Использовать второй операнд для выбора бита (бит 0 — младший).
Передать выбранный бит в CF.
Инвертировать выбранный бит.

Операция

Команда BTS выполняет две функции. Во-первых, она позволяет программисту инвертировать любой бит в регистре или ячейке памяти. Во-вторых, она помещает старое значение бита в CF.

Первый операнд — регистр или ячейка памяти, второй — номер бита. Им может быть любое беззнако-

вое целое до 8 бит, но в зависимости от размера первого операнда используются только 4 или 5 бит.

Особые случаи	Режимы	Причины
#GP(0)	P	Результат в защищенном от записи сегменте
#GP(0)	P	Недопустимый эффективный адрес памяти в сегментах CS, DS, ES, FS, GS
#SS(0)	P	Неверный адрес в сегменте SS
#PF(fc)	P V	Страничное нарушение
INT(13)	R V	Часть операнда вне диапазона адреса от 0 до 0FFFFH

Примечания

Новые команды манипуляций битами упрощают многие программы (например, построения таблиц размещения, в которых состояние бита показывает, свободны ли область RAM или сектор на диске). Команда BTS позволяет быстро инвертировать любой бит в таблице. Она не упрощает управление дисплейным буфером.

Пример

```
MOV     AX,3CD0H ; Просматриваемое значение в
                 ; AX.
BTC     AX,2     ; AX=3CD4H, CF=0.
```

BTR Проверка бита и сброс (80386)

КОП	Формат	Такты
0F B3 [r]	BTR r/mw,rw	6/13
0F B3 [r]	BTR r/md,rd	6/13
0F BA [6] ib	BTR r/mw,ib	6/8
0F BA [6] ib	BTR r/md,ib	6/8

Флажки

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0									0		0		1	S

Псевдокод

Использовать первый операнд для локализации регистра или ячейки памяти.
Использовать второй операнд для выбора одного бита (бит 0 — младший).
Передать выбранный бит в CF.
Сбросить выбранный бит (в 0).

Операция

Команда BTR выполняет две функции. Во-первых, она позволяет выбрать любой бит в регистре или памяти и сбросить его. Во-вторых, она помещает прежнее значение бита в CF.

Первый операнд — регистр или ячейка памяти, второй — номер бита. Им может быть любое беззнаковое целое до 8 бит, но в зависимости от размера первого операнда используются только 4 или 5 бит.

Особые случаи	Режимы	Причины
#GP(0)	P	Результат в защищенном с записи сегменте
#GP(0)	P	Недопустимый эффективный адрес памяти в сегментах CS, DS, ES, FS, GS
#SS(0)	P	Неверный адрес в сегменте SS
#PF(fc)	P V	Страничное нарушение
INT(13)	R V	Часть операнда вне диапазона адреса от 0 до 0FFFFH

Примечания

Новые команды манипуляций битами упрощают многие программы (например, построение таблиц размещения, в которых состояние бита показывает, свободны ли область RAM или сектор на диске). Команда BTR позволяет быстро сбросить любой бит в таблице. Она же упрощает управление дисплейным буфером.

Пример

```
MOV    AX,3CD0H ; Просматриваемое значение в
                ; AX.
BTR    AX,7     ; AX=3C50H, CF=1.
```

BTS Проверка бита и установка (80386)

КОП	Формат	Такты
0F AB [r]	BTS r/mw,rw	6/13
0F AB [r]	BTS r/md,rd	6/13
0F BA [5] ib	BTS r/mw,ib	6/8
0F BA [5] ib	BTS r/md,ib	6/8

Флажки

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0									0		0		1	S

Псевдокод

Использовать первый операнд для локализации регистра или ячейки памяти.
Использовать второй операнд для выбора бита (бит 0 — младший).
Передать выбранный бит в CF.
Установить выбранный бит (в 1).

Операция

Команда BTS выполняет две функции. Во-первых, она позволяет программисту выбрать любой бит и установить его в 1. Во-вторых, она помещает старое значение бита в CF.

Первый операнд — регистр или ячейка памяти, второй — номер бита. Им может быть любое беззнаковое целое до 8 бит, но в зависимости от размера первого операнда используются только 4 или 5 бит.

Особые случаи	Режимы	Причины
#GP(0)	P	Результат в защищенном от записи сегменте
#GP(0)	P	Недопустимый эффективный адрес памяти в сегментах CS, DS, ES, FS, GS
#SS	P	Неверный адрес в сегменте SS
#PF(fc)	P V	Страничное нарушение
INT(13)	R V	Часть операнда вне диапазона адреса от 0 до 0FFFFH

Примечания

Новые команды манипуляций битами упрощают многие программы (например, построение таблиц размещения, в которых состояние бита показывает, свободна ли область RAM или сектор на диске). Команда BTS позволяет быстро установить любой бит в таблице (например, отмечая распределение сектора). Она упрощает управление дисплейным буфером.

Пример

```
MOV    AX,3CD0H  ; Просматриваемое значение в
                 ; AX.
BTS    AX,2      ; AX=3CD4H, CF=0.
```

CALL Вызов процедуры (8086)

КОП	Формат	Тип	Такты
E8 dw	CALL dw	короткий, прямой	7+m
E8 dd	CALL dd	короткий, прямой	7+m
FF [2]	CALL r/mw	короткий, косвенный	7+m/10+m
FF [2]	CALL r/md	короткий, косвенный	7+m/10+m
9A pd	CALL pd	длинный, прямой	17+m,*
9A pp	CALL pp	длинный, прямой	17+m,*
FF [3]	CALL mw:w	длинный, косвенный	22+m,*
FF [3]	CALL mw:d	длинный, косвенный	22+m,*

*Эти команды имеют другие функции и времена выполнения в защищенном режиме (см. гл. 5).

Флажки

Обычно команда CALL не воздействует на флажки. Но при переключении задачи в защищенном режиме все флажки принимают значения заполненных флажков новой задачи.

Псевдокод

```
IF межсегментный CALL THEN
   PUSH CS в стек
   Установить CS на селектор сегмента операнда
END IF
PUSH IP в стек
Установить IP на смещение операнда
```

Операция

Команда CALL применяется для передачи управления другой части программы с возможностью возврата в точку вызова. Она позволяет реализовать процедуры,

фуикции и подпрограммы в языках высокого уровия и удобиа в языке ассемблер.

Имеется 4 типа команд CALL: короткая и длинная, прямая и косвенная.

Если вызываемая процедура находится в том же сегмеите, что и вызывающая программа, то вызов иазывается коротким, а если в другом, то длиниым. В случае длинного вызова в стек иеобходимо включить содержимое регистров CS и IP, а в случае короткого — только IP.

В зависимости от определеиия адреса вызываемой процедуры вызов может быть прямым или косвенным. Прямой вызов содержит адрес в самой команде. В косвеином вызове адрес вызываемой процедуры содержит указатель иа регистр или ячейку памяти.

Короткий прямой вызов содержит смещеиие, которое прибавляется к IP, а в длинном прямом вызове команда содержит полное смещеиие. Короткий косвеииый вызов может адресовать регистр или ячейку памяти, содержащие смещеиие и сегмент для вызываемой процедуры. Длиииый косвеиный вызов ие может использовать регистр, а использует только память (так как полные указатели имеют длину до 48 бит).

Базовая процедура в защищеиием режиме такая же, но межсегмеитные вызовы сложиее, так как команда CALL может определять процедуру операциоиной системы или даже другую задачу. В любом случае необходимо коитролировать защиту памяти (см. гл. 5).

Особые случаи	Режимы	Причины
#NP	P	Задаииый сегмеит кода отсутствует
#TS	P	Требуется переключение задачи
#GP(0)	P	Недопустимый эффективный адрес памяти в сегмеитах CS, DS, ES, FS, GS
#SS(0)	P	Неверный адрес в сегмеите SS
#PF(fc)	P V	Страиичиое иарушеиие
INT(13)	R V	Часть операнда вие диапазоиа адреса от 0 до 0FFFFH

Примечания

Команды CALL применяются довольно часто. Одна из трудностей связана с передачей параметров вызываемой процедуре. В новых процессорах iAPX 86 команда PUSHA помогает сохранить значение регистров в стеке, а команда ENTER помогает реализовать вложенные процедуры. Вопросы о том, когда использовать процедуры вместо копий секций кода и когда скопировать процедуру в каждый сегмент вместо использования межсегментных вызовов, довольно сложны.

Пример

```
CALL    SUBROUTINE
```

CBW Преобразование байта в слово (8086)

CWDE Преобразование слова в двойное слово (80386)

КОП	Формат	Такты
98	CBW	3
98	CWDE	3

Флажки

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0									0		0		1	

Псевдокод

```
IF размер операнда=16 бит THEN (*CBW*)
   Установить все биты AH на значение старшего
       бита AL.
ELSE (*CWDE*)
   Установить старшие 16 бит EAX на значение
       старшего бита AX.
END IF
```

Операция

Преобразование байта в слово осуществляется путем расширения знака: старший бит байта копируется во все биты слова, не занятые байтом. Преобразование слова в

двойное слово производится аналогично. Отметим, что расширение знака в дополнительном коде сохраняет значение.

Особые случаи

Нет.

Примечания

Эти команды применяются для увеличения размера числа.

Пример

```
MOV    AL,0FCH   ; Загружает -4 в AL.
CBW              ; AX=0FFFCH (тоже -4).
```

CLC Сброс флажка переноса (8086)

КОП	Формат	Такты
F8	CLC	2

Флажки

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0									0		0		1	0

Псевдокод

Сбросить CF в нуль.

Операция

Флажок CF сбрасывается в нуль.

Особые случаи

Нет.

Примечания

Применяется до цикла суммирования чисел длиной в несколько базовых единиц, чтобы в самом цикле пользоваться только командой ADC. Кроме того, в семействе iAPX требуется, чтобы CF был сброшен перед командой SBB (вычитание с заемом), если ей не предшествует команда SUB.

Пример

```
CLC     ; Сбрасывает CF.
```

CLD Сброс флажка направления (8086)

КОП	Формат	Такты
FC	CLD	2

Флажки

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0									0		0		1	

Псевдокод

Сбросить флажок DF в нуль.

Операция

Флажок DF сбрасывается в нуль.

Особые случаи

Нет.

Примечания

Флажок DF управляет направлением циклических операций. Когда DF=0, то после каждого повторения циклической операции производится инкремент индексных регистров SI и/или DI. Это «обычное» направление от младших адресов к старшим. При DF=1 производится декремент SI и/или DI.

Пример

```
CLD  ; Сбрасывает DF.
```

CLI Сброс флажка разрешения прерываний (8086)

КОП	Формат	Такты
FA	CLI	3

Флажки

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0			0						0		0		1	

Псевдокод

Сбросить флажок IF в нуль.

Операция

Сбрасывает флажок IF в 0, запрещая восприятие прерываний. В защищенном режиме 80286 и 80386 имеются сложности. Команда CLI не выполняется, если текущий уровень привилегий программы, выполняющей CLI, больше (т. е. она менее привилегирована), чем содержат биты уровня привилегий ввода/вывода в регистре Flags.

Особые случаи	Режимы	Причины
#GP(0)	P	Текущая привилегия больше IOPL

Примечания

Эта команда важна для критических секций, которые нельзя прерывать. Запрещается прерывание от клавиатуры, чтобы пользователи не пытались прервать программу.

Необходимо разобраться в уровнях привилегий защищенного режима, прежде чем пытаться управлять прерываниями из своей программы.

Пример

```
CLI     ; Сбрасывает IF.
```

CMC Инвертирование флажка переноса (8086)

КОП	Формат	Такты
F5	CMC	2

Флажки

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0									0		0		1	S

Псевдокод

```
IF (CF=0) THEN
   установить флажок CF
ELSE
   сбросить флажок CF
END IF
```

Операция

Команда CMC инвертирует флажок CF.

Особые случаи

Нет.

Примечания

Эта команда удобна при использовании CF как параметра внутри программы. Если состояние CF известно, не пользуйтесь командой CMC, а применяйте CLC или STC (программу проще сопровождать).

Пример

```
CMC     ; Изменяет состояние CF.
```

CMP Сравнение (8086)

КОП	Формат	Такты
3C ib	CMP AL,ib	2
3D iw	CMP AX,iw	2
3D iw	CMP EAX,id	2
80 [7] id	CMP r/mb,ib	2/5
81 [7] iw	CMP r/mw,iw	2/5
81 [7] id	CMP r/md,id	2/5
83 [7] ib	CMP r/mw,iw	2/5
83 [7] ib	CMP r/md,id	2/5
38 [r]	CMP r/mb,rb	2/5
39 [r]	CMP r/mw,rw	2/5
39 [r]	CMP r/md,rd	2/5
3A [r]	CMP rb,r/mb	2/6
3B [r]	CMP rw,r/mw	2/6
3B [r]	CMP rd,r/md	2/6

Флажки

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0			S				S	S	0	S	0	S	1	S

Псевдокод

Вычесть второй операнд из первого, не сохранять результат
Установить флажки по результату вычитания

Операция

Команда CMP вычитает второй операнд из первого, но нигде не сохраняет результат. Флажки устанавливаются по результату вычитания.

Особые случаи	Режимы	Причины
#GP(0)	P	Недопустимый эффективный адрес памяти в сегментах CS, DS, ES, FS, GS
#SS(0)	P	Неверный адрес в сегменте SS
#PF(fc)	P V	Страничное нарушение
INT(13)	R V	Часть операнда вне диапазона адреса от 0 до 0FFFFH

Примечания

Применяется для сравнения чисел с последующим условным переходом.

Пример

```
MOV   AX,956    ; Загружает 3BCH в AX.
MOV   BX,373    ; Загружает 175H в BX.
CMP   AX,BX     ; CF, AF, ZF, SF, OF=0; PF=1.
```

CMPS Сравнение в цикле (8086)

КОП	Формат	Такты (одиночная)	Такты (с повторением)
A6	CMPSB	10	5+9×N
A7	CMPSW	10	5+9×N
A7	CMPSD	10	5+9×N

Число N означает число выполненных повторений.

Флажки

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0			S				S	S	0	S	0	S	1	S

Псевдокод

```
Вычесть ES:[(E)DI] из DS:[(E)SI], не сохранять результат
Установить флажки по результату вычитания
IF DF=0 TNEN
    Прибавить размер операндов (в байтах) к (E)SI и (E)DI
ELSE
    Вычесть размер операндов (в байтах) из (E)SI и (E)DI
END IF
```

Операция

Команда вычитает получатель из источника, но нигде не сохраняет результат. Флажки устанавливаются по полученному результату.

Операнды в команде CMPS не определяются. Первый из них адресуется индексом источника (E)SI, который обычно относится к сегменту данных, если не задан префикс замены, а второй — индексом получателя (E)DI, который во всех циклических командах относится к дополнительному сегменту (ES).

Команда CMPS производит модификацию SI и DI в зависимости от состояния флажка направления DF. Обычно с ней применяются префиксы REPE и REPNE.

Особые случаи	Режимы	Причины
#GP(0)	P	Недопустимый эффективный адрес памяти в сегментах CS, DS, ES, FS, GS
#SS(0)	P	Неверный адрес в сегменте SS
#PF(fc)	P V	Страничное нарушение
INT(13)	R V	Часть операнда вне диапазона адреса от 0 до 0FFFFH

Примечания

Обычно команда CMPS применяется для сравнения двух массивов на равенство. В этом случае в регистр CX помещается длина массивов, а в регистры SI и DI — адреса первых элементов. После этого команда REPE CMPS производит циклические операции: сравнение элементов, продвижение указателей SI и DI и декремент CX до тех пор, пока два операнда окажутся неравными (ZF не равен 0) или CX исчерпается до нуля. После этого команда JNZ передает управление на обработку ситуации несовпадения.

Команда REPNE выполняет сравнение до достижения в CX нуля или обнаружения двух одинаковых элементов. Так можно убедиться, что все элементы массивов разные.

Пример

```
STR1   DD       1,2,3,4,5
STR2   DD       1,2,4,5,6
:              :
CLD                          ; Движение вперед.
LDS          ESI,STR1        ; Источник DS:[ESI].
LES          EDI,STR2        ; Получатель ES:[EDI].
MOV          ECX,5           ; Счетчик повторений.
REPE CMPSD                   ; Выполняется 3 раза и за-
                             ; канчивается с ECX=2,
                             ; DS:[ESI] показывают на
                             ; 4 в STR1, а ES:[EDI]
                             ; показывают на 5 в STR2.
```

CWD Преобразование слова в двойное слово (8086)

CDQ Преобразование двойного слова в счетверенное слово (80386)

КОП	Формат	Такты
99	CWD	2
99	CDQ	2

Флажки

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0									0		0		1	

Псевдокод

```
IF размер операнда=16 бит THEN (*CWD*)
   Установить все биты DX на значение старшего
      бита AX.
ELSE (*CDQ*)
   Установить все биты EDX на значение старшего
      бита EAX.
END IF
```

Операция

Преобразование слова в двойное слово производится путем расширения знака, т. е. старший бит слова копируется во все биты второго слова. Значение двойного слова в дополнительном коде эквивалентно значению исходного слова. Команда CDQ действует аналогично.

Особые случаи

Нет.

Примечания

Обе команды применяются для коррекции размера операнда.

Пример

```
MOV    AX,0FFFCH     ; Загружает -4 в AX.
CWD                  ; DX=0FFFFH, DX:AX равно
                     ; -4.
```

DAA Десятичная коррекция AL после сложения (8086)

КОП	Формат	Такты
27	DAA	4

Флажки

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0			S				S	S	0	S	0	S	1	S

Псевдокод

```
IF (младшая тетрада AL>9) OR (AF=1) THEN
   инкремент AL на 6
   установить AF в 1
ELSE
   сбросить флажок AF
END IF
IF (AL>9FH) OR (CF=1) THEN
   установить AL на AL+60H
   установить CF в 1
ELSE
   сбросить флажок CF
END IF
```

Операция

Команда DAA корректирует AL после сложения двух упакованных BCD-цифр. В байте находятся две цифры, и команда DAA формирует в них правильный результат с образованием десятичного переноса в CF.

Особые случаи

Нет.

Примечания

Команда DAA часто применяется в цикле сложения многобайтных десятичных (BCD) чисел и указывается сразу после команды ADC. Она выполняет только преобразование двоичных чисел в BCD-формат.

Пример

```
MOV     AX,18H      ; Загружает BCD-число 18 в AL.
ADD     AL,6        ; Прибавляет BCD-число 6 к AL
                    ; (=1EH).
DAA                 ; AX содержит BCD-число 24,
                    ; CF=0, AF=1.
```

DAS Десятичная коррекция AL после вычитания (8086)

КОП	Формат	Такты
2F	DAS	4

Флажки

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0			S				S	S	0	S	0	S	1	S

Псевдокод

```
IF (младшая тетрада AL>9) OR (AF=1) THEN
   декремент AL на 6
   установить AF в 1
ELSE
   сбросить флажок AF
END IF
IF (AL>9FH) OR (CF=1) THEN
   установить AL на AL-60H
   установить CF в 1
ELSE
   сбросить флажок CF
END IF
```

Операция

Команда DAS корректирует AL после вычитания двух упакованных BCD-цифр. В байте находятся две цифры, и команда DAS формирует в них правильный результат с образованием десятичного заема в CF.

Особые случаи

Нет.

Примечания

Команда DAS часто применяется в цикле вычитания многобайтных десятичных (BCD) чисел и указывается сразу после команды SBB. Она выполняет только преобразование двоичных чисел в BCD-формат.

Пример

```
MOV     AX,16H   ; Загружает BCD-число в AL.
SUB     AL,8     ; Вычитает BCD-число 8 из
                 ; AL(=0DH).
DAS              ; AX содержит 8, CF=0, AF=1.
```

DEC Декремент (8086)

КОП	Формат	Такты
FE [1]	DEC r/mb	2/6
FF [1]	DEC r/mw	2/6
FF [1]	DEC r/md	2/6
48+rw	DEC rw	2
48+rd	DEC rd	2

Флажки

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0			S				S	S	0	S	0	S	1	

Псевдокод

Уменьшает операнд на 1.

Операция

Вычитает 1 из своего единственного операнда и помещает результат в операнд.

Особые случаи	Режимы	Причины
#GP(0)	P	Результат в защищенном от записи сегменте
#GP(0)	P	Недопустимый эффективный адрес памяти в сегментах CS, DS, ES, FS, GS
#SS(0)	P	Неверный адрес в сегменте SS
#PF(fc)	P V	Страничное нарушение
INT(13)	R V	Часть операнда вне диапазона адреса от 0 до 0FFFFH

Примечания

Код команды DEC кажется гораздо проще команды SUB, но он выполняется почти такое же время. Команда DEC не влияет на флажок CF, поэтому часто лучше использовать команду SUB со вторым операндом 1.

Пример

```
MOV     AX,956      ; Загружает 3BCH в AX.
DEC     AX          ; Теперь AX содержит 955
                    ; (3BBH).
```

DIV Беззнаковое деление (8086)

КОП	Формат	Такты
F6 [6]	DIV AL,r/mb	14/17
F7 [6]	DIV AX,r/mw	22/25
F7 [6]	DIV EAX,r/md	38/41

Флажки

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0									0		0		1	

Псевдокод

```
IF код операции F6 THEN (*делимое слово*)
    разделить AX на беззнаковый байт из операнда
    сохранить частное в AL
    сохранить остаток в AH
ELSE IF размер операнда 16 THEN (*делимое двой-
     ное слово*)
    разделить DX:AX на беззнаковое слово из опе-
         ранда
    сохранить частное в AX
    сохранить частное в DX
ELSE (*делимое счетверенное слово*)
    разделить EDX:EAX на беззнаковое двойное слово
         из операнда
    сохранить частное в EAX
    сохранить остаток в EDX
END IF
```

Операция

Делимое и делитель считаются беззнаковыми целыми. Частное и остаток как беззнаковые целые помещаются в два регистра.

Особые случаи	Режимы	Причины
#GP(0)	P	Результат в защищенном от записи .сегменте
#GP(0)	P	Недопустимый эффективный адрес памяти в сегментах CS, DS, ES, FS, GS
#SS(0)	P	Неверный адрес в сегменте SS
#PF(fc)	P V	Страничное нарушение
INT(0)	P R V	Результат слишком велик для получателя или делитель равен нулю
INT(13)	R V	Часть операнда вне диапазона адреса от 0 до 0FFFFH

Примечания

Команда DIV выполняется долго и ее нужно избегать в критических секциях. При переполнении возникает прерывание, поэтому целесообразно заранее проверить возможность деления.

Делимое и делитель в команде DIV считаются беззнаковыми (или положительными) числами. Часто требуется разрабатывать процедуру деления, которая использует повторяющиеся команды DIV или IDIV для деления с большей точностью, чем обеспечивает одна команда DIV или IDIV.

Пример

```
MOV     AX,956      ; Загружает 3BCH в AX.
MOV     BX,300      ; Загружает 12CH в BX.
DIV     AX,BX       ; AL=3, AH=56.
```

ENTER Образование стекового кадра для процедуры (80186)

КОП	Формат	Такты
C8 iw 00	ENTER iw,0	10
C8 iw 01	ENTER iw,1	12
C8 iw ib	ENTER iw,ib	15+4×n

Число «n» означает номер уровня (первый операнд) минус 1.

Флажки

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0									0		0		1	

Псевдокод

```
Включить в стек (E)BP
Сохранить копию (E)SP
IF второй операнд больше 0 THEN
    FOR 1 TO второй операнд минус 1 DO
      Декремент (E)BP на 2 (или 4)
      Включить в стек слово, адресуемое (E)BP
    END DO
    Включить в стек слово, адресуемое сохраненной
        копией (E)SP
END IF
Установить (E)BP на сохраненную копию (E)SP
Установить (E)SP на (E)SP минус первый операнд
```

Операция

Команда ENTER применяется для реализации вызовов процедур и предназначена для языков высокого уровня. Обычно при каждом вызове в стек включаются 4 элемента: аргументы процедуры, адрес возврата, группа «указателей кадра» и локальные переменные, используемые процедурой. Вся эта информация в общем называется «стековым кадром». «Указатели кадра» представляют собой указатели стековых кадров вызовов процедур, которые ведут к вызванной текущей процедуре.

Из четырех компонентов стекового кадра команда ENTER очень удобно работает с двумя последними. Параметры процедуры программист явно включает в стек до команды CALL (которая включает в стек адрес возврата). Команда ENTER включает в стек указатели кадра (если они есть) и распределяет место в стеке для локальных переменных.

Команда имеет два операнда: первый — число байт, резервируемых для хранения локальных переменных, а второй — это уровень текущей процедуры. Уровень управляет тем, сколько указателей кадра включается в стек (например, команда ENTER нулевого уровня только распределяет память для локальных переменных). Команда LEAVE переводит стек в то состояние, в котором он был до выполнения команды ENTER.

Выполнение команды состоит из нескольких этапов, текущее содержимое BP включается в стек, BP заменяется на SP, в стек включаются все указатели кадра (до достижения 0 уровня) и производится декремент SP на число байт, которое резервируется для локальных переменных. Эти действия можно реализовать и другими ассемблерными командами; однако команда ENTER действует очень быстро и дает разработчикам компиляторов стандартный способ реализации вызовов процедур.

В большинстве ассемблерных программ команда ENTER применяется с нулевым уровнем, либо совсем не применяется. Она делает больше операций, чем требуется в большинстве ассемблерных программ. Уровни больше нуля предназначены для разработчиков компиляторов.

Особые случаи	Режимы	Причины
#SS(O)	P	(E)SP превысил предел стека

Примечания

Команда ENTER на уровнях больше 0 становится довольно сложной, так как при каждом новом вызове процедуры необходимо повторно сохранять указатели всех предыдущих стековых кадров. Это организовано так, что каждая новая процедура может считать глобальными все переменные в процедуре, которая вызвала ее, или в процедуре, которая вызвала эту вызывающую процедуру и т. д. Такой способ ограниченного доступа к переменным является базовым для некоторых новых языков программирования.

Обычно команда ENTER является первой командой в процедуре. Первый параметр ее — это число байт для хранения локальных переменных; доступ к ним осуществляется через BP как индексный регистр. Старое значение BP имеет смещение 0, а за ним следуют указатели кадра (если они есть). Затем идут локальные переменные. Второй параметр является текущим уровнем. В конце процедуры находится команда LEAVE, которая восстанавливает указатель стека и устраняет доступ к локальным переменным.

Пример

```
SUBROUTINE:
    ENTER   12,3    ; Подпрограмма имеет 3 локаль-
                    ; ных переменных (двойные сло-
                    ; ва) и находится на уровне
                    ; вложения 3.
    :       :       :
    LEAVE           ; Удаляет из стека текущий сте-
                    ; ковый кадр.
    RET     8       ; Подпрограмма имела два пара-
                    ; метра (двойные слова).
```

HLT Останов (8086)

КОП	Формат	Такты
F4	HLT	5

Флажки

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0									0		0		1	

Псевдокод

```
WHILE нет разрешенного внешнего прерывания или
    RESET DO
END WHILE
```

Операция

Команда HLT останавливает выполнение дальнейших команд до восприятия внешнего прерывания или сброса RESET. Обычно она применяется для управления программами, которые должны прямо взаимодействовать с внешним миром (другими CPU, которые должны активизировать остановленный микропроцессор сигналом RESET, или внешними устройствами, генерирующими сигнал прерывания).

Особые случаи	Режимы	Причины
#GP(0)	P	Текущий уровень привилегий не 0

Примечания

В однопроцессорных компьютерах команда HLT применяется редко. Ее можно заменить бесконечным циклом ожидания прерывания (с небольшим запаздыванием восприятия прерывания).

Пример

```
HLT  ; Процессор останавливается.
```

IDIV Знаковое деление (8086)

КОП	Формат	Такты
F6 [7]	IDIV AL,r/mb	19/22
F7 [7]	IDIV AX,r/mw	27/30
F7 [7]	IDIV EAX,r/md	43/46

Флажки

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0									0		0		1	

Псевдокод

```
IF код операции F6 THEN (*делимое слово*)
    разделить AX на знаковый байт из операнда
    сохранить частное в AL
    сохранить остаток в AH
ELSE IF размер операнда 16 THEN (*делимое двой-
     ное слово*)
    разделить DX:AX на знаковое слово из операнда
    сохранить частное в AX
    сохранить остаток в DX
ELSE (*делимое счетверенное слово*)
    разделить EDX:EAX на знаковое двойное слово из
         операнда
    сохранить частное в EAX
    сохранить остаток в EDX
END IF
```

Операция

Делимое и делитель считаются знаковыми числами. Частное и остаток в виде знаковых чисел помещаются в регистры.

Особые случаи	Режимы	Причины
#GP(0)	P	Результат в защищенном от записи сегменте
#SS(0)	P	Неверный адрес в сегменте SS
#GP(0)	P	Недопустимый эффективный адрес памяти в сегментах CS, DS, ES, FS, GS
#PF(fc)	P V	Страничное нарушение
INT(0)	P R V	Результат слишком велик для получателя или делитель равен нулю
INT(13)	R V	Часть операнда вне диапазона адреса от 0 до 0FFFFH

Примечания

Команда IDIV выполняется несколько дольше DIV. Ее следует избегать в критических секциях программы. Так как переполнение вызывает прерывание, целесообразно перед операцией проверить возможность деления.

Команда IDIV работает со знаковыми числами, поэтому абсолютный диапазон чисел меньше, чем в команде DIV.

Часто требуется процедура деления с повторяющимися командами DIV или IDIV, которая обеспечивает большую точность, чем одна команда DIV или IDIV.

Пример

```
MOV     AX,956     ; Загружает 3BCH в AX.
MOV     BX,-300    ; Загружает 0FED4H в BX.
IDIV    AX,BX      ; AL= -3(0FFFDH) и AH=56.
```

IMUL Знаковое умножение (8086)

КОП	Формат	Такты
F6 [5]	IMUL r/mb	9-14/12-17
F7 [5]	IMUL r/mw	9-22/12-25
F7 [5]	IMUL r/md	9-38/12-41
0F AF [r]	IMUL rw,r/mw	9-22/12-25
0F AF [r]	IMUL rd,r/md	9-38/12-41
6B [r] ib	IMUL rw,r/mw,ib	9-14/12-17
6B [r] ib	IMUL rd,r/md,ib	9-14/12-17
6B [r] ib	IMUL rw,ib	9-14/12-17
6B [r] ib	IMUL rd,ib	9-14/12-17
69 [r] iw	IMUL rw,r/mw,iw	9-22/12-25
69 [r] id	IMUL rd,r/md,id	9-38/12-41
69 [r] iw	IMUL rw,iw	9-22/12-25
69 [r] id	IMUL rd,id	9-38/12-41

Флажки

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0			S				U	U	0	U	0	U	1	S

Псевдокод

```
IF одноопернадная форма THEN
   IF размер операнда байт THEN
     Установить AX на произведение AL и операнда
   ELSE IF размер операнда слово THEN
     Установить DX:AX на произведение AX и операнда
   ELSE (*размер операнда двойное слово*)
     Установить EDX:EAX на произведение EAX и операнда
   END IF
ELSE IF двухоперандная форма THEN
```

```
    Установить первый операнд на произведение пер-
        вого и второго операндов
ELSE (*трехоперандная форма*)
    Установить первый операнд на произведение вто-
        рого и третьего операндов
END IF
```

Операция

Все операнды считаются знаковыми числами; произведение также будет знаковым числом. Произведение n-битных чисел в общем случае содержит 2n бит. Когда OF=1, в многооперандных формах команды произошла потеря старших бит результата. В однооперандных формах такая ситуация исключена, так как размер результата вдвое больше размера операндов.

Особые случаи	Режимы	Причины
#GP(0)	P	Недопустимый эффективный адрес памяти в сегментах CS, DS, ES, FS, GS
#SS(0)	P	Неверный адрес в сегменте SS
#PF(fc)	P V	Страничное нарушение
INT(13)	R V	Часть операнда вне диапазона адреса от 0 до 0FFFFH

Примечания

Время умножения зависит от значения множителя. Чем больше значащих бит, тем больше длится операция. В 80386 используется алгоритм умножения с «досрочным» окончанием. Правый операнд во всех формах команды IMUL называется оптимизирующим множителем («m» в приводимом ниже соотношении). Фактическое число тактов умножения определяется следующим соотношением:

```
IF m=0 THEN тактов=9
ELSE тактов=max(log 2(|m|),3)+6
```

Пример

```
MOV    AL,40    ; Загружает 40 в AL.
IMUL   10       ; AX содержит 400, OF=1.
```

IN Ввод из порта (8086)

КОП	Формат	Такты
E4 ib	IN AL,ib	5
E5 ib	IN AX,ib	5
E5 ib	IN EAX,ib	5
EC	IN AL,DX	6
ED	IN AX,DX	6
ED	IN EAX,DX	6

Флажки

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0									0		0		1	

Псевдокод

```
IF второй операнд непосредственный THEN
    Расширить второй операнд до 16 бит с нулем для
        получения адреса входного порта
ELSE
    Адрес второго порта содержится в DX
END IF
IF размер первого операнда байт THEN
    Передать байт из входного порта в AL
ELSE IF размер первого опсранда слово THEN
    Передать слово из входного порта в AX
ELSE (*размер первого операнда — двойное слово*)
    Передать двойное слово из входного порта в EAX
END IF
```

Операция

Команда IN применяется для получения одного байта, слова или двойного слова из порта периферийного

устройства, номер которого от 0 до 65535. Обычно устройство имеет несколько отдельных портов для команд, слова состояния и данных. Первые два из них применяются для целей управления.

Номера портов 00F8H—00FFH зарезервированы фирмой Intel и использовать их не следует.

Особые случаи	Режимы	Причины
#GP(0)	P	Текущая привилегия больше IOPL
#GP(0)	V	Некоторые из бит разрешения в TSS содержат 1

Примечания

Обычно программы осуществляют ввод путем вызова операционной системы, и команда IN в этом случае не нужна. Однако она необходима для разработчиков драйверов устройств и для работы непосредственно с устройством.

Когда требуется ввести несколько байт, следует пользоваться командой IN. Если устройство ввода не может обеспечить данные с высокой скоростью, команду IN можно поместить в цикл и замедлять скорость передачи данных командами NOP или циклом задержки.

Пример

```
MOV   DX,20      ; Номер порта для команды IN в
                 ; DX.
IN    EAX,DX     ; Вводит двойное слово в EAX.
```

INC Инкремент (8086)

КОП	Формат	Такты
FE [0]	INC r/mb	2/6
FF [0]	INC r/mw	2/6
FF [0]	INC r/md	2/6
40+rw	INC rw	2
40+rd	INC rd	2

Флажки

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0			S				S	S	0	S	0	S	1	

Псевдокод

Увеличить операнд на 1.

Операция

Команда INC прибавляет 1 к своему единственному операнду и сохраняет результат в операнде.

Особые случаи	Режимы	Причины
#GP(0)	P	Результат в защищенном от записи сегменте
#GP(0)	P	Недопустимый эффективный адрес памяти в сегментах CS, DS, ES, FS, GS
#SS(0)	P	Неверный адрес в сегменте SS
#PF(fc)	P V	Страничное нарушение
INT(13)	R V	Часть операнда вне диапазона адреса от 0 до 0FFFFH

Примечания

Хотя команда INC кажется гораздо проще команды ADD, она выполняется примерно такое же время. Команда INC не влияет на флажок CF, поэтому часто лучше пользоваться командой ADD.

Пример

```
MOV     AX,956      ; Загружает 3BCH в AX.
INC     AX          ; Теперь AX содержит 957.
```

INS Ввод циклический из порта (80186)

КОП	Формат	Такты (одиночная)	Такты (с повторением)
6C	INSB	8	6+6×N
6D	INSW	8	6+6×N
6D	INSD	8	6+6×N

Значение N означает число в регистре (E)CX.

Флажки

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0									0		0		1	

Псевдокод

```
IF размер операнда байт THEN
   Передать в AL байт из входного порта с номером
      в DX
ELSE IF размер операнда слово THEN
   Передать в AX слово из входного порта с номе-
      ром в DX
ELSE (*размер операнда двойное слово*)
   Передать в EAX двойное слово из входного порта
      с номером в DX
END IF
IF DF=0 THEN
   Прибавить размер операнда (в байтах) к (E)DI
ELSE
   Вычесть размер операнда (в байтах) из (E)DI
END IF
```

Операция

Команда INS, как и команда IN, применяется для ввода из порта периферийного устройства одного байта, слова или двойного слова. Номер порта от 0 до 65535. Обычно устройство имеет несколько отдельных портов для команд, слова состояния и данных. Первые два из них используются для управления устройством.

В команде INS номер порта всегда находится в DX. Получатель данных адресуется ES:[(E)DI], и замена сегмента не допускается. Команда INS рассчитана на использование с префиксом REP, т. е. в конце команды производится инкремент или декремент (E)DI (в зависимости от состояния DF) на размер операнда.

Номера портов 00F8H—00FFH зарезервированы фирмой Intel и использовать их не следует.

Особые случаи	Режимы	Причины
#GP(0)	P	Текущая привилегия больше IOPL
#GP(0)	P	Недопустимый эффективный адрес памяти в сегментах CS, DS, ES, FS, GS
#SS(0)	P	Неверный адрес в сегменте SS
#PF(fc)	P V	Страничное нарушение
INT(13)	R V	Часть операнда вне диапазона адреса от 0 до 0FFFFH
#GP(0)	V	Некоторые биты разрешения в TSS содержат 1

Примечания

Большинство программ осуществляют ввод через вызовы операционной системы и не используют команду INS, но она важна для разработки драйверов устройств и прямого управления устройством.

Когда требуется ввести несколько байт, применяется команда INS. Если устройство не может обеспечивать ввод данных с высокой скоростью, команду INS можно поместить в цикл и уменьшать скорость передачи данных с помощью команд NOP или цикла задержки.

Пример

```
CLD                     ; Направление вперед.
LES     EDI,INSTR       ; Получатель ввода ES:[EDI].
MOV     ECX,5           ; Счетчик повторения в ECX.
MOV     DX,40           ; Номер входного порта для
                        ; INS.
REP INSB                ; Ввести 5 байт из порта 40.
```

INT Вызов процедуры прерывания (8086)

КОП	Формат	Такты
CC	INT 3	33*
CD ib	INT ib	37*
CE	INTO	3,35*

*В защищенном режиме эти команды имеют другие функции и времена выполнения (см. гл. 5). Команда INTO длится три такта, если прерывания нет, и 35 — если есть.

Флажки

Обычно команды INT не влияют на флажки. Однако в защищенном режиме при переключении задачи изменяются все флажки.

Псевдокод

```
IF не INTO или OF=1 THEN
    Включить в стек регистр (E)Flags
    Включить в стек регистр CS
    Включить в стек регистр (E)IP
    Запретить внешние прерывания (IF=0)
    Передать в CS:(E)IP сегмент и смещение из таб-
        лицы векторов прерываний
END IF
```

Операция

Прерывание похоже на вызов процедуры, но в стек включаются еще и флажки, и автоматически запрещаются прерывания. Номер прерывания используется для

поиска элемента в таблице прерываний, который дает адрес процедуры обработки прерывания.

В реальном режиме команда INT вначале включает в стек регистр Flags, а затем регистр сегмента кода; после этого в стек включается IP. Затем сбрасывается флажок IF, запрещая внешние (аппаратные) прерывания. Наконец, из таблицы векторов прерываний в CS и IP загружаются сегмент и смещение. После окончания процедуры прерывания регистры восстанавливаются, и выполнение продолжается с команды, следующей после INT.

Основные действия в защищенном режиме такие же, но необходимо пройти несколько проверок уровня привилегий и достаточности размера стека, в противном случае возникает ошибка защиты (см. гл. 5).

В команде INTO номер процедуры прерывания равен 4, и прерывание возникает, если OF=1. Обычно эта команда применяется сразу после арифметической операции.

Особые случаи	Режимы	Причины
#NP	P	Заданный сегмент кода отсутствует
#TS	P	Требуется переключение задачи
#GP	P	Недопустимый сегмент CS, DS, ES, FS, GS
#GP	P	Недопустимый сегмент SS
*****	R	Отключение из-за недостаточного стекового пространства
#GP(0)	V	Эмулирует операцию прерывания, если IOPL меньше 3

Примечания

Обычно таблицу векторов прерываний создает операционная система, и прерывания в основном используются для вызовов операционной системы. В руководстве по операционной системе есть инструкции по разработке процедур прерываний.

Пример

```
MOV     AL,100    ; Загружает число в AL.
MUL     AL,10     ; Это умножение вызывает OF=1.
INTO              ; Возникает прерывание 4.
```

IRET Возврат из прерывания (8086)

КОП	Формат	Такты
CF	IRET	22*

*В защищенном режиме эта команда имеет другие функции и время выполнения (см. гл. 5).

Флажки

Команда IRET возвращает из стека все сохраненные флажки.

Псевдокод

```
Извлечь из стека (T)IP.
Извлечь из стека CS.
Извлечь из стека (E)Flags.
```

Операция

Команда IRET похожа на команду RET возврата из процедуры, но из стека извлекается еще и Flags. Так как восстанавливаются старые флажки от (предполагаемой) команды INT или INTO, то состояние IF становится прежним (заменяя запрещение прерываний от выполнения команды INT).

Основные действия в защищенном режиме такие же, но необходимо пройти несколько проверок уровня привилегий и достаточности размера стека, в противном случае возникает ошибка защиты (см. гл. 5).

Особые случаи	Режимы	Причины
#NP	P	Заданный сегмент кода отсутствует
#TS	P	Требуется переключение задачи
#GP	P	Неверный сегмент CS, DS, ES, FS, GS
#SS	P	Неверный сегмент SS
INT(13)	R	Часть извлекаемого из стека операнда лежит ниже 0FFFFH
#GP(0)	V	Эмулирует операцию прерывания, если IOPL меньше 3

Примечания

Обычно таблицу векторов прерываний создает операционная система, и прерывания, в основном, применяются для вызовов операционной системы. Команды IRET для этих системных вызовов находятся в конце вызовов и не видны программисту. В руководстве по операционной системе есть указания по разработке процедур прерываний.

Пример

```
IRET   ; Возврат из прерывания.
```

JCC Переход, если условие удовлетворяется (8086)

КОП	Формат	Условие перехода	Такты
77 db	JA db	Выше (CF=0 и ZF=0)	7+m,3
0F 87 dw	JA dw	Выше (CF=0 и ZF=0)	7+m,3
0F 87 dd	JA dd	Выше (CF=0 и ZF=0)	7+m,3
73 db	JAE db	Выше или равно (CF=0)	7+m,3
0F 83 dw	JAE dw	Выше или равно (CF=0)	7+m,3
0F 83 dd	JAE dd	Выше или равно (CF=0)	7+m,3
72 db	JB db	Ниже (CF=1)	7+m,3
0F 82 dw	JB dw	Ниже (CF=1)	7+m,3
0F 82 dd	JB dd	Ниже (CF=1)	7+m,3
76 db	JBE db	Ниже или равно (CF=1 или ZF=1)	7+m,3
0F 86 dw	JBE dw	Ниже или равно (CF=1 или ZF=1)	7+m,3
0F 86 dd	JBE dd	Ниже или равно (CF=1 или ZF=1)	7+m,3
72 db	JC db	Перенос (CF=1)	7+m,3
0F 82 dw	JC dw	Перенос (CF=1)	7+m,3
0F 82 dd	JC dd	Перенос (CF=1)	7+m,3
E3 db	JCXZ db	CX=0	7+m,3
74 db	JE db	Равно (ZF=1)	7+m,3
0F 84 dw	JE dw	Равно (ZF=1)	7+m,3
0F 84 dd	JE dd	Равно (ZF=1)	7+m,3
E3 db	JECXZ db	ECX=0	7+m,3
7F db	JG db	Больше (ZF=0 и SF=OF)	7+m,3
0F 8F dw	JG dw	Больше (ZF=0 и SF=OF)	7+m,3
0F 8F dd	JG dd	Больше (ZF=0 и SF=OF)	7+m,3
7D db	JGE db	Больше или равно (SF=OF)	7+m,3
0F 8D dw	JGE dw	Больше или равно (SF=OF)	7+m,3
0F 8D dd	JGE dd	Больше или равно (SF=OF)	7+m,3
7C db	JL db	Меньше (SF<>OF)	7+m,3
0F 8C dw	JL dw	Меньше (SF<>OF)	7+m,3
0F 8C dd	JL dd	Меньше (SF<>OF)	7+m,3

7E db	JLE db	Меньше или равно (ZF=1 или SF<>OF)	7+m,3
0F 8E dw	JLE dw	Меньше или равно (ZF=1 или SF<>OF)	7+m,3
0F 8E dd	JLE dd	Меньше или равно (ZF=1 или SF<>OF)	7+m,3
76 db	JNA db	Не выше (CF=1 или ZF+1)	7+m,3
0F 86 dw	JNA dw	Не выше (CF=1 или ZF=1)	7+m,3
0F 86 dd	JNA dd	Не выше (CF=1 или ZF=1)	7+m,3
72 db	JNAE db	Не выше или равно (CF=1)	7+m,3
0F 82 dw	JNAE dw	Не выше или равно (CF=1)	7+m,3
0F 82 dd	JNAE dd	Не выше или равно (CF=1)	7+m,3
73 db	JNB db	Не ниже (CF=0)	7+m,3
0F 83 dw	JNB dw	Не ниже (CF=0)	7+m,3
0F 83 dd	JNB dd	Не ниже (CF=0)	7+m,3
77 db	JNBE db	Не ниже или равно (CF=0 и ZF=0)	7+m,3
0F 87 dw	JNBE dw	Не ниже или равно (CF=0 и ZF=0)	7+m,3
0F 87 dd	JNBE dd	Не ниже или равно (CF=0 и ZF=0)	7+m,3
73 db	JNC db	Не перенос (CF=0)	7+m,3
0F 83 dw	JNC dw	Не перенос (CF=0)	7+m,3
0F 83 dd	JNC dd	Не перенос (CF=0)	7+m,3
75 db	JNE db	Не равно (ZF=0)	7+m,3
0F 85 dw	JNE dw	Не равно (ZF=0)	7+m,3
0F 85 dd	JNE dd	Не равно (ZF=0)	7+m,3
7E db	JNG db	Не больше (ZF=1 или SF<>OF)	7+m,3
0F 8E dw	JNG dw	Не больше (ZF=1 или SF<>OF)	7+m,3
0F 8E dd	JNG dd	Не больше (ZF=1 или SF<>OF)	7+m,3
7C db	JNGE db	Не больше или равно (SF<>OF)	7+m,3
0F 8C dw	JNGE dw	Не больше или равно (SF<>OF)	7+m,3
0F 8C dd	JNGE dd	Не больше или равно (SF<>OF)	7+m,3
7D db	JNL db	Не меньше (SF=0F)	7+m,3
0F 8D db	JNL dw	Не меньше (SF=0F)	7+m,3
0F 8D dd	JNL dd	Не меньше (SF=0F)	7+m,3
7F db	JNLE db	Не меньше или равно (ZF=0 и SF=0F)	7+m,3

0F 8F dw	JNLE dw	Не меньше или равно (ZF=0 и SF=0F)	7+m,3
0F 8F dd	JNLE dd	Не меньше или равно (ZF=0 и SF=0F)	7+m,3
71 db	JNO db	Не переполнение (OF=0)	7+m,3
0F 81 dw	JNO dw	Не переполнение (OF=0)	7+m,3
0F 81 dd	JNO dd	Не переполнение (OF=0)	7+m,3
7B db	JNP db	Не паритет (PF=0)	7+m,3
0F 8B dw	JNP dw	Не паритет (PF=0)	7+m,3
0F 8B dd	JNP dd	Не паритет (PF=0)	7+m,3
79 db	JNS db	Не знак (SF=0)	7+m,3
0F 89 dw	JNS dw	Не знак (SF=0)	7+m,3
0F 89 dd	JNS dd	Не знак (SF=0)	7+m,3
75 db	JNZ db	Не нуль (ZF=0)	7+m,3
0F 85 dw	JNZ dw	Не нуль (ZF=0)	7+m,3
0F 85 dd	JNZ dd	Не нуль (ZF=0)	7+m,3
70 db	JO db	Переполнение (OF=1)	7+m,3
0F 80 dw	JO dw	Переполнение (OF=1)	7+m,3
0F 80 dd	JO dd	Переполнение (OF=1)	7+m,3
7A db	JP db	Паритет (PF=1)	7+m,3
0F 8A dw	JP dw	Паритет (PF=1)	7+m,3
0F 8A dd	JP dd	Паритет (PF=1)	7+m,3
7A db	JPE db	Паритет четный (PF=1)	7+m,3
0F 8A dw	JPE dw	Паритет четный (PF=1)	7+m,3
0F 8A dd	JPE dd	Паритет четный (PF=1)	7+m,3
7B db	JPO db	Паритет нечетный (PF=0)	7+m,3
0F 8B dw	JPO dw	Паритет нечетный (PF=0)	7+m,3
0F 8B dd	JPO dd	Паритет нечетный (PF=0)	7+m,3
78 db	JS db	Знак (SF=1)	7+m,3
0F 88 dw	JS dw	Знак (SF=1)	7+m,3
0F 88 dd	JS dd	Знак (SF=1)	7+m,3
74 db	JZ db	Нуль (ZF=1)	7+m,3
0F 84 dw	JZ dw	Нуль (ZF=1)	7+m,3
0F 84 dd	JZ dd	Нуль (ZF=1)	7+m,3

Флажки

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0									0		0		1	

Псевдокод

```
IF условие удовлетворяется THEN
    Установить IP на IP+смещение с расширением
        знака
END IF
```

Операция

Команды перехода изменяют последовательность выполнения команд. Условный переход выполняется или не выполняется в зависимости от состояния процессора. Переходы классифицируются по способам вычисления адреса перехода.

В процессоре 80386 есть три типа переходов — короткие, близкие и длинные. Короткие переходы — это относительные переходы вблизи команды перехода (от -128 до +127 байт). Близкие переходы тоже относительные, но передают управление в пределах всего сегмента кода. Длинные переходы — это абсолютные переходы по конкретному адресу в любом сегменте кода. В 80386 условные переходы — короткие или близкие, а для команд JCXZ и JECXZ — только короткие.

Большинство команд условных переходов проверяют состояние одного или нескольких флажков. Если условие удовлетворяется, переход производится; в противном случае выполняется следующая по порядку команда. Команды JCXZ и JECXZ проверяют не флажки, а содержимое регистра CX или ECX.

Первое значение для времени выполнения соответствует выполнению перехода, а второе — случаю, когда перехода нет. Переходы замедляют работу процессора 80386, так как очереди команд очищаются и их нужно повторно заполнять. Значение «m» — это число компонентов в команде по адресу перехода. Компонентом считается каждый байт префикса, байт кода операции, байт ModRM и байт SIB. Смещение или непосредственное значение также учитывается.

Особые случаи	Режимы	Причины
#GP(0)	P	Переход вне границ сегмента кода

Примечания

Часть команд условного перехода предназначена для беззнаковых чисел, а другая — для знаковых. Слова «выше» и «ниже» относятся к беззнаковым числам, а «больше» и «меньше» — к знаковым.

Из таблицы видно, что для нескольких мнемоник имеет место один и тот же код операции. Причина такой избыточности в том, что одно и тоже состояние флажков может иметь разный смысл в зависимости от контекста перехода. Обычно условный переход применяется после команд CMP и SUB. Он учитывает результат сравнения, и к нему подходит мнемоника JE (перейти, если равны). С другой стороны, после команды DEC тот же переход удобно обозначить мнемоникой JZ (перейти, если нуль), оканчивая цикл при достижении счетчиком нуля.

Команды JCXZ и JECXZ реализуют короткие переходы. Для реализации длинных переходов потребуется дополнительная команда JMP.

Команды REPE и LOOPNE вызывают зацикливание по состоянию ZF или CX=0. Команда LCXZ сразу после цикла вызывает переход, когда окончание цикла вызвано достижением CX нуля. Это позволяет отдельно обрабатывать различные условия окончания цикла.

Пример

```
            MOV   ECX,5     ; Операнд для сравне-
                            ; ния.
            CMP   ECX,7     ; Сравнить 5 и 7.
            JLE   TARGET    ; Переход будет, 10
                            ; тактов.
            :     :         :
TARGET:     AND   AL,7      ; 3 компонента: КОП,
                            ; ModRM и непосредст-
                            ; венное значение.
```

JMP Безусловный переход (8086)

КОП	Формат	Тип	Такты
EB db	JMP db	Короткий, прямой	7+m
E9 dw	JMP dw	Короткий, прямой	7+m
E9 dd	JMP dd	Короткий, прямой	7+m
FF [4]	JMP r/mw	Короткий, косвенный	7+m/10+m
FF [4]	JMP r/md	Короткий, косвенный	7+m/10+m
EA pd	JMP pd	Длинный, прямой	17+m*
EA pp	JMP pp	Длинный, прямой	17+m*
FF [5]	JMP mw:w	Длинный, косвенный	22+m*
FF [5]	JMP mw:d	Длинный, косвенный	22+m*

*В защищенном режиме эти команды имеют другие функцию и время выполнения.

Флажки

Обычно команда JMP не воздействует на флажки, но в защищенном режиме при переключении задачи все флажки модифицируются (см. гл. 5).

Псевдокод

```
IF межсегментный переход THEN
    Установить CS на селектор сегмента операнда
END IF
Установить IP на смещение операнда
```

Операция

Команда JMP передает управление безусловно, т. е. переход осуществляется всегда (оператор GOTO в языках высокого уровня). Все команды перехода замедляют

работу процессора 80386, так как очередь команд очищается и должна заполняться вновь.

Имеется 5 типов команд перехода в зависимости от того, где находится точка перехода и как определяется адрес перехода.

В коротком (прямом) переходе эта точка определяется смещением, находящимся сразу за кодом операции. Это смещение просто прибавляется к IP, давая новый адрес. Длина смещения — один байт, поэтому диапазон перехода от -128 до +127 байт.

Короткий прямой переход может содержать относительное смещение, находящееся за кодом операции, и оно также прибавляется к JP. Отметим, что во время суммирования IP показывает на команду после JMP. Длина смещения — слово или двойное слово в зависимости от текущего сегмента кода.

В близком косвенном переходе определяется регистр или ячейка памяти, содержащие смещение нужной команды в текущем сегменте кода. Длина смещения также является словом или двойным словом в зависимости от размера текущего сегмента кода.

Длинный прямой переход допускает передачу управления в другой сегмент кода. Команда содержит непосредственный операнд, являющийся указателем следующей выполняемой команды. Указатель состоит из слова селектора сегмента (он загружается в CS) и смещения в виде слова или двойного слова, которое загружается в (E)IP. Размер смещения зависит от размера сегмента кода.

Длинный косвенный переход аналогичен короткому косвенному переходу в том, что команда содержит указатель фактического адреса. Но этот адрес является полным указателем, а не только смещением, поэтому он не может быть в регистре. Указатель содержит селектор сегмента (он загружается в CS) и слово или двойное слово смещения, которое загружает ся в (E)IP. Размер смещения зависит от размера сегмента кода.

В защищенном режиме основные действия такие же, но межсегментные переходы сложнее, так как переход может определять процедуру операционной системы или даже другую задачу. В этих случаях необходимо контролировать защиту памяти (см. гл. 5).

Особые случаи	Режимы	Причины
#NP	P	Заданный сегмент кода отсутствует
#TS	P	Требуется переключение задачи
#GP(0)	P	Недопустимый эффективный адрес памяти в сегментах CS, DS, ES, FS, GS
#SS(0)	P	Неверный адрес в сегменте SS
#PF(fc)	P V	Страничное нарушение
INT(13)	R V	Часть операнда вне диапазона адреса от 0 до 0FFFFH

Примечания

В процессоре 80386 короткий переход (расстояние в байт) и короткий прямой переход (расстояние в слово) в реальном режиме длятся 7 тактов. Достоинство короткого перехода только в том, что команда короче на один байт.

Пример

```
        JMP TARGET  ; Переход длится 10 так-
                    ; тов.
          :    :    :
TARGET: AND AX,7    ; 3 компонента (КОП,
                    ; ModRM и непосредст-
                    ; венный байт).
```

LAHF Загрузка флажков в AH (8086)

КОП	Формат	Такты
9F	LAHF	2

Флажки

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0									0		0		1	

Псевдокод

Установить AH на младший байт регистра Flags.

Операция

Позволяет передать флажки в регистр AH, где их можно обработать. Команда введена для совместимости с 8080/85 и применяется редко.

Биты в AH: SF ZF x AF x PF x CF.

Особые случаи

Нет.

Примечания

Команда LAHF применяется редко. Но ее можно использовать для проверки всего байта флажков (например, сравнивая AH с маской) или для реализации слож-

ных переходов, не реализуемых обычными командами JE, JNA и другими. Нужно учитывать, что при этом несколько бит не определены. Команда AND AH,D5H позволяет сбросить неопределенные биты.

Пример

```
LAHF                ; Загружает флажки в AH.
AND   AH,11H        ; Маскирует биты AF и CF.
JNZ   SOMEWHERE     ; Имитирует переход по
                    ; «AF=CF=1».
```

LEA Загрузка смещения эффективного адреса (8086)

КОП	Формат	Такты
8D [r]	LEA r16,m16	2
8D [r]	LEA r32,m16	2
8D [r]	LEA r16,m32	2
8D [r]	LEA r32,m32	2

Флажки

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0									0		0		1	

Псевдокод

```
IF размер регистра 16 THEN
    IF размер памяти 16 THEN
      Установить регистр на смещение эффективного
          адреса
    ELSE (*размер памяти 32*)
      Установить регистр на младшие 16 бит смеще-
          ния эффективного адреса
    END IF
ELSE (*размер регистра 32*)
    IF размер памяти 16 THEN
      Установить регистр на смещение эффективного
          адреса с нулевым расширением до 32 бит
    ELSE (*размер памяти 32*)
      Установить регистр на смещение эффективного
          адреса
    END IF
END IF
```

Операция

Команда LEA похожа на команду MOV, пересылающую данные из памяти в регистр, но при этом передается смещение адреса, а не содержимое RAM по этому адресу. Размер памяти в псевдокоде определяется атрибутом USE сегмента, содержащего адрес памяти.

Особые случаи	Режимы	Причины
#UD	P	Второй операнд регистр
INT(6)	R V	Второй операнд регистр

Примечания

Иногда команды LEA и MOV могут взаимозаменяться, например LEA AX,STRUC и MOV AX,OFFSET STRUC. Однако команда LEA позволяет использовать для второго операнда любой режим адресации памяти. Например, команда LEA AX,STRUC[BX][DI] позволяет передать в AX адрес указателя с двойной адресацией; командой MOV этого сделать нельзя.

Команда LEA имеет ограниченные возможности по умножению. В режиме масштабированной индексной адресации можно реализовать умножение на 2, 4, 8. В режиме базовой масштабированной индексной адресации можно умножить на 3 и 9. Все это занимает всего 2 такта.

Пример

```
MOV   BX,11            ; Для умножения в BX.
LEA   AX,[BX][BX×4]    ; В AX будет BX×5=55.
```

LEAVE Удаление стекового кадра процедуры (80186)

КОП	Формат	Такты
C9	LEAVE	4

Флажки

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0									0		0		1	

Псевдокод

```
Установить (E)SP на (E)BP.
Извлечь из стека старый указатель кадра в (E)BP.
```

Операция

Команда LEAVE применяется для реализации вызовов процедур и ориентирована на языки высокого уровня. Она сбрасывает указатель стека для удаления локальных переменных процедуры и извлекает из стека «указатель кадра». Эти действия готовят стек к следующей команде RET.

Особые случаи	Режимы	Причины
#SS(0)	P	BP адресует ячейку вне текущего сегмента стека
INT(13)	R V	Часть операнда вне диапазона адреса от 0 до 0FFFFH

Примечания

Команда LEAVE гораздо проще команды ENTER, выполняющей основную работу. Она устанавливает BP на адресацию правильного места, поэтому LEAVE должна просто поместить BP в указатель стека, а затем извлечь из стека старое значение BP.

Команда LEAVE должна быть последней командой перед RET в процедуре, имеющей первую команду ENTER. Команда LEAVE изменяет указатель стека так, что все локальные переменные удаляются из стека, а указатель кадра BP готов для процедуры, которая вызывала данную.

Пример

```
SUBROUTINE:
            ENTER 12,3    ; Подпрограмма имеет 3
                          ; локальных перемен-
                          ; ных (двойные слова)
                          ; и находится на уров-
                          ; не выполнения 3.
            :       :     ;
            LEAVE         ; Удаляет из стека те-
                          ; кущий стековый кадр.
            RET 8         ; Подпрограмма имеет 2
                          ; параметра (двойные
                          ; слова)
```

LOCK Формирование сигнала блокировки шины (8086)

КОП	Формат	Такты
F0	LOCK	0

Флажки

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0									0		0		1	

Псевдокод

Формировать сигнал блокировки шины на время выполнения следующей команды.

Операция

Префикс LOCK применяется в мультипроцессорных системах для предотвращения конфликтов при обращениях к разделенной памяти. Сигнал LOCK запрещает доступ к шине других устройств, подключенных к шине. Сигнал действует все время выполнения команды, находящейся за префиксом LOCK.

В процессоре 80386 с префиксом LOCK могут использоваться не все команды. Указание его перед неразрешенной командой вызывает особое прерывание неопределенного кода операции. Разрешены следующие команды:

BT, BTS, BTR, BTC mem, reg/imm
ADD, OR, ADC, SBB, AND, SUB, XOR mem, reg/imm

XCHG	reg, mem
XCHG	mem, reg
NOT, NEG, INC, DEC	mem

Отметим, что все эти команды требуют считывания операнда из памяти, его модификации и записи в ту же ячейку памяти. Команда XCHG блокируется всегда, даже без указания префикса.

Особые случаи	Режимы	Причины
#GP(0)	P	Текущий уровень привилегий больше уровня привилегий ввода/вывода
#UD	P V	Команда после LOCK отсутствует в приведенном перечне
INT(6)	R	Команда после LOCK отсутствует в приведенном перечне

Примечания

Заблокированная команда может вызвать любой дополнительный особый случай прерывания, как при «обычном» выполнении.

Необходимость префикса LOCK показывает такой пример. В двухпроцессорной системе ячейка разделенной памяти используется как счетчик «событий». Когда любой из процессоров обнаруживает событие, он должен произвести инкремент этого счетчика. Предположим, что команда INC в программе каждого процессора не имеет префикса LOCK. Пусть зафиксированно 4 события и оба процессора одновременно обнаруживают событие. Если оба процессора попытаются выполнить свои команды INC в одно время, возникает следующая ситуация.

Первый процессор считывает из памяти значение (4) и начинает внутренний инкремент. В этот момент второй процессор получает шину и считывает из памяти то же значение (4). Пока второй процессор производит инкремент, первый возвращает в память результат (5). Наконец, второй процессор также записывает в память свой результат (5). В этом случае одно «событие» пропущено. Префикс LOCK предотвращает такую ошибку.

Вы можете заметить, что приведенная ситуация возникает очень редко. Конечно, система может недели и месяцы работать правильно, а затем возникнет «катастрофа».

Блокированный доступ не гарантируется, если другой процессор выполняет команду со следующими условиями:

- Префикс LOCK не используется.
- Команда не содержится в приведенном выше перечне.
- Указанный операнд в памяти не перекрывается.

В предыдущих процессорах iAPX 86 допускалось более свободное использование префикса LOCK. Поэтому будьте внимательны при использовании программ для этих процессоров.

Пример

```
LOCK BTR    FLAGWORD,AVAILBIT ; Очищает бит
                              ; доступности,
                              ; открывая до-
                              ; ступ к разде-
                              ; ленному ре-
                              ; сурсу.
INC         NODICE            ; Перейти, ес-
                              ; ли ресурс
                              ; распределен.
```

LODS Загрузка массива (8086)

КОП	Формат	Такты (одиночная)	Такты (с повторением)
AC	LODSB	5	*
AD	LODSW	5	*
AD	LODSD	5	*

*Команда LODS не может использовать префиксы повторения.

Флажки

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0									0		0		1	

Псевдокод

```
Определить размер операнда
IF размер операнда 8 THEN
   Переслать в AL байт из адреса DS:[(E)SI]
ELSE IF размер операнда 16 THEN
   Переслать в AX слово из адреса DS:[(E)SI]
ELSE (*размер операнда 32*)
   Переслать в EAX двойное слово из адреса
        DS:[(E)SI]
IF DF=0 THEN
END IF
   Прибавить к (E)SI размер операнда (в байтах)
ELSE
   Вычесть из (E)SI размер операнда (в байтах)
END IF
```

Операция

Команда LODS похожа на команду MOV передачи из памяти в AL или AX, но она после передачи автоматически корректирует SI в зависимости от флажка DF. Коррекция производится на размер операнда.

Особые случаи	Режимы	Причины
#GP(0)	P	Недопустимый эффективный адрес памяти в сегментах CS, DS, ES, FS, GS
#SS(0)	P	Неверный адрес в сегменте SS
#PF(f)	P V	Страничное нарушение
INT(13)	R V	Часть операнда вне диапазона адреса от 0 до 0FFFFH

Примечания

Команду LODS нельзя использовать с префиксами повторения, но она часто применяется в циклах для поиска нужного символа в массиве: LODS загружает символ в AL, он сравнивается с нужным символом, и все повторяется до обнаружения нужного символа.

Команды LODS и STOS можно использовать для передачи массива из DS в ES, производя проверки или изменения.

Пример

```
;  Следующий фрагмент копирует массив из одно-
;  го места в другое. Операция продолжается до пе-
;  редачи максимального числа символа или встречи
;  первого символа.
;  Такие массивы применяются в языке Си.
;
        CLD                     ; Направление
                                ; вперед.
        LDS     ESI,SSTR        ; Указатель
                                ; источника в
                                ; DS:[ESI].
```

```
        LES     EDI,DSTR        ; Указатель по-
                                ; лучателя в
                                ; ES:[EDI].
        MOV     ECX,MAXSTR      ; Счетчик повто-
                                ; рения цикла.
COPYL:
        LODSB                   ; Символ из ис-
                                ; точника.
        STOSB                   ; Передать в по-
                                ; лучатель.
        TEST    AL,AL           ; Проверить ну-
                                ; левой символ.
        LOOPNZ  COPYL           ; Повторение пе-
                                ; редачи.
```

LOOPcc Управление циклом (8086)

КОП	Формат	Условие перехода	Такты
E2 db	LOOP db	(E)CX<>0	11+m
E1 db	LOOPE db	(E)CX<>0 и ZF=1	11+m
E1 db	LOOPZ db	(E)CX<>0 и ZF=1	11+m
E0 db	LOOPNE db	(E)CX<>0 и ZF=0	11+m
E0 db	LOOPNZ db	(E)CX<>0 и ZF=0	11+m

Флажки

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0									0		0		1	

Псевдокод

```
Декремент (E)CX (*флажки не применяются*)
Определить размер операнда
IF условие удовлетворяется THEN
    Установить IP на IP+ смещение со знаковым расширением
END IF
```

Операция

Команда LOOP производит декремент (E)CX и проверяет условие достижения нуля. Может проверяться также ZF. Если регистр не содержит нуля и удовлетворяется дополнительное условие по флажку ZF, производится короткий переход к метке, указанной как операнд после LOOPcc. Смещение имеет длину один байт и обеспечивает диапазон перехода от -128 до +127. Оно добавляется к текущему адресу, и получается адрес следующей команды.

Особые случаи	Режимы	Причины
#GP(0)	P	Точка перехода вне пределов текущего сегмента кода

Примечания

Команда LOOP позволяет реализовать на ассемблере циклы типа FOR. Значение в CX не должны модифицироваться внутренними командами цикла. Счетчик цикла считается беззнаковым целым числом.

Мнемоника LOOPZ часто вводит в заблуждение. Помните, что команда зацикливает до ZF=0 и (E)CX=0.

Пример

См. команду LODS.

Lxx Загрузка полного указателя (8086)

КОП	Формат	Такты
C5 [r]	LDS rw,mw:w	7,22*
C5 [r]	LDS rd,mw:d	7,22*
C4 [r]	LES rw,mw:w	7,22*
C4 [r]	LES rd,mw:d	7,22*
0F B2 [r]	LSS rw,mw:w	7,22*
0F B2 [r]	LSS rd,mw:d	7,22*
0F B4 [r]	LFS rw,mw:w	7,22*
0F B4 [r]	LFS rd,mw:d	7,22*
0F B5 [r]	LGS rw,mw:w	7,22*
0F B5 [r]	LGS rd,mw:d	7,22*

*В защищенном режиме требуются дополнительные такты из-за необходимости дополнительной обработки.

Флажки

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0									0		0		1	

Псевдокод

Установить сегментный регистр на регистр второго операнда.
Установить общий регистр на смещение второго операнда.

Операция

Эти команды применяются для одновременной установки двух регистров: сегментного регистра и регистра

общего назначения. В сегментный регистр загружается 16-битный селектор сегмента операнда. В регистр общего назначения загружается слово или двойное слово смещения второго операнда внутри сегмента. Размер зависит от атрибута размера указанного сегмента.

В защищенном режиме операция сложнее. Любое изменение регистров означает, что процессор попытается обратиться совсем к другой области памяти, имеющей, возможно, другие уровни защиты, чем текущая. При работе с виртуальной памятью сегмент даже может находиться на диске.

Особые случаи	Режимы	Причины
#UD	P	Источником является регистр
#GP(0)	P	Недопустимый эффективный адрес памяти в сегментах CS, DS, ES, FS, GS
#GP(0)	P	В SS защищается пустой селектор
#SS(0)	P	Неверный адрес в сегменте SS
#PF(fc)	P V	Страничное нарушение
INT(13)	R V	Часть операнда вне диапазона адреса от 0 до 0FFFFH
INT(6)	R V	Источником является регистр

Примечания

Эти команды, в основном, применяются для задания сегментного регистра и индексного регистра в подготовке обращения к новому блоку памяти.

Пример

```
LES   DI,STRUCTURE  ; Загрузить сегмент STRUC
                    ; в ES и смещение STRUC
                    ; в DI.
```

MOV Пересылка данных (8086)

КОП	Формат	Такты
B0+r ib	MOV rb,ib	2
B8+r iw	MOV rw,iw	2
B8+r id	MOV rd,id	2
C6 [0] ib	MOV r/mb,ib	2
C7 [0] iw	MOV r/mw,iw	2
C7 [0] id	MOV r/md,id	2
A0 d	MOV AL,db	4
A1 d	MOV AX,dw	4
A1 d	MOV EAX,dd	4
A2 d	MOV db,AL	2
A3 d	MOV dw,AX	2
A3 d	MOV dd,EAX	2
88 [r]	MOV r/mb,rb	2
89 [r]	MOV r/mw,rw	2
89 [r]	MOV r/md,rd	2
8A [r]	MOV rb,r/mb	2/4
8B [r]	MOV rw,r/mw	2/4
8B [r]	MOV rd,r/md	2/4
8E [0]	MOV ES,r/mw	2/5*
8E [1]	MOV CS,r/mw	2/5*
8E [2]	MOV SS,r/mw	2/5*
8E [3]	MOV DS,r/mw	2/5*
8C [0]	MOV r/mw,ES	2
8C [1]	MOV r/mw,CS	2
8C [2]	MOV r/mw,SS	2
8C [3]	MOV r/mw,DS	2

*В защищенном режиме эти команды требуют 18/19 тактов из-за дополнительной обработки (см. гл. 5).

Флажки

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0									0		0		1	

Псевдокод

Устанавливает значение первого операнда в соответствии со значением второго операнда.

Операция

Команда MOV похожа на оператор присвоения в языках высокого уровня. Первым задается операнд-получатель, а вторым — источник.

В команде MOV есть две особенности. Первая связана с выполнением команды MOV SS; при этом автоматически запрещаются все прерывания до завершения следующей команды. Это сделано в предположении, что следующей будет команда MOV SP или другая команда, восстанавливающая содержимое SS:SP.

Вторая особенность связана с передачей данных в любой сегментный регистр в защищенном режиме. При любом применении сегментного регистра процессор может обратиться к другой области памяти, которая может иметь другие уровни защиты.

Особые случаи	Режимы	Причины
#GP,#SS,#NP	P	Загружается сегментный регистр
#GP(0)	P	Результат в защищенном от записи сегменте
#GP(0)	P	Недопустимый эффективный адрес памяти в сегментах CS, DS, ES, FS, GS
#SS(0)	P	Неверный адрес в сегменте SS
#PF(fc)	P V	Страничное нарушение
INT(13)	R V	Часть операнда вне диапазона адреса от 0 до 0FFFFH

Примечания

Необходимо знать, какие команды могут заменять команду MOV. Например, для пересылки массивов лучше пользоваться командой MOVS.

Пример

```
MOV   AX,ES    ; Копирует содержимое ES в AX.
```

MOVxX Пересылка с нулевым/знаковым расширением (8086)

КОП	Формат	Такты
0F BE [r]	MOVSX rw,r/mb	3/6
0F BE [r]	MOVSX rd,r/mb	3/6
0F BF [r]	MOVSX rd,r/mw	3/6
0F B6 [r]	MOVZX rw,r/mb	3/6
0F B6 [r]	MOVZX rd,r/mb	3/6
0F B7 [r]	MOVZX rd,r/mw	3/6

Флажки

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0									0		0		1	

Псевдокод

Расширить источник (нулями или битом знака) до длины получателя.
Сократить результат в получателе.

Операция

Команды MOVSX и MOVZX связаны с ситуацией, когда получатель имеет больше бит, чем источник (например, получатель — слово, а источник — байт). SX означает «знаковое расширение», а ZX — «нулевое расширение».

Особые случаи	Режимы	Причины
#GP(0)	P	Недопустимый эффективный адрес памяти в сегментах CS, DS, ES, FS, GS
#SS(0)	P	Неверный адрес в сегменте SS
#PF(fc)	P V	Страничное нарушение
INT(13)	R V	Часть операнда вне диапазона адреса от 0 до 0FFFFH

Примечания

Так как получателем в этих командах может быть только один из регистров, они удобны для подготовки регистров к дальнейшей обработке.

Пример

```
MOVSX   AX,92H    ; В AX будет 0FF92H.
```

MOVS Пересылка цепочки (8086)

КОП	Формат	Такты (одиночная)	Такты (с повторением)
A4	MOVSB	7	5+4×N
A7	MOVSW	7	5+4×N
A7	MOVSD	7	5+4×N

Буква N означает число повторений из регистра (E)CX.

Флажки

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0									0		0		1	

Псевдокод

```
Определить размер операнда
IF размер операнда 8 THEN
   Переслать байт из адреса DS:[(E)SI] в
        ES:[(E)DI]
ELSE IF размер операнда 16 THEN
   Переслать слово из адреса DS:[(E)SI] в
        ES:[(E)DI]
ELSE (*размер операнда 32*)
   Переслать двойное слово из адреса DS:[(E)SI] в
        ES:[(E)DI]
END IF
IF DF=0 THEN
   Прибавить размер операнда (в байтах) к (E)SI и
        (E)DI
ELSE
   Вычесть размер операнда (в байтах) из (E)SI и
        (E)DI
END IF
```

Операция

Команда MOVS похожа на обычную MOV передачи из области памяти, адресуемой SI, в область дополнительного сегмента, адресуемого DI. Но она автоматически корректирует индексы: если DF=0, производится инкремент, а в противном случае — декремент. Величина коррекции зависит от размера операнда (1=байт, 2=слово, 4=двойное слово).

Особые случаи	Режимы	Причины
#GP(0)	P	Результат в защищенном от записи сегменте
#GP(0)	P	Недопустимый эффективный адрес памяти в сегментах CS, DS, ES, FS, GS
#SS(0)	P	Неверный адрес в сегменте SS
#PF(fc)	P V	Страничное нарушение
INT(13)	R V	Часть операнда вне диапазона адреса от 0 до 0FFFFH

Примечания

Команду MOVS нельзя использовать с префиксом REPE или REPS, так как она не воздействует на флажки. Однако префикс REP действует как обычно, вызывая передачу CX байт или слов из источника в получатель.

Команда MOVS осуществляет простую передачу с заданием известной длины в CX, но если передачу нужно остановить при встрече нужного байта, потребуется другой способ, например, с использованием LODS, проверки, STOS, и повторения с помощью команды LOOP.

Пример

```
CLD                     ; Направлеиие вперед.
LDS       ESI,STR1      ; Указатель источника
                        ; DS:[ESI].
LES       EDI,STR2      ; Указатель получателя
                        ; ES:[EDI].
MOV       ECX,5         ; Счетчик повтореиия.
REPE MOVSD              ; 5 раз копирует из STR1
                        ; в STR2.
```

MUL Беззнаковое умножение (8086)

КОП	Формат	Такты
F6 [4]	MUL r/mb	9-14/12-17
F7 [4]	MUL r/mw	9-22/12-25
F7 [4]	MUL r/md	9-38/12-41

Флажки

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0			S				U	U	0	U	0	U	1	S

Псевдокод

```
IF размер операнда байт THEN
    Установить AX на произведение AL и операнда
ELSE IF размер операнда слово THEN
    Установить DX:AX на произведение AX и операнда
ELSE (*размер операнда — двойное слово*)
    Установить EDX:EAX на произведение EAX и операнда
END IF
```

Операция

Все операнды считаются беззнаковыми числами, такими же являются результаты. Размер произведения n-битных сомножителей равен 2n.

Особые случаи	Режимы	Причины
#GP(0)	P	Недопустимый эффективный адрес памяти в сегментах CS, DS, ES, FS, GS
#SS(0)	P	Неверный адрес в сегменте SS
#PF(fc)	P V	Страничное нарушение
INT(13)	R V	Часть операнда вне диапазона адреса от 0 до 0FFFFH

Примечания

Время умножения зависит от значения множителя. Чем больше значащих бит, тем дольше длится операция. В процессоре 80386 реализован алгоритм «досрочного» окончания операции. Указанный операнд в команде MUL называется оптимизирующим множителем («m» в приведенных выражениях). Фактически число тактов для умножения можно найти по выражению:

```
IF m = 0 THEN тактов = 9
ELSE тактов = max(log2(|m|),3)+6
```

Пример

```
MOV     AL,128      ; В AL загружается 128 (80H).
MOV     BL,10       ; В BL загружается 10 (0AH).
MOV     BL          ; AX содержит 1280 (500H),
                    ; OF=1.
```

NEG Изменение знака (8086)

КОП	Формат	Такты
F6[3]	NEG r/mb	2/6
F7[3]	NEG r/mw	2/6
F7[3]	NEG r/md	2/6

Флажки

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0			S				S	S	0		0	S	1	S

Псевдокод

```
Вычесть операнд из нуля
Поместить результат в операнд
IF операнд нуль THEN
   сбросить CF=0
ELSE
   установить CF=1
END IF
```

Операция

Команда NEG изменяет знак своего единственного операнда.

Сначала производится его инвертирование, а затем инкремент. Можно также считать, что операнд вычитается из нуля.

Особые случаи	Режимы	Причины
#GP(0)	P	Результат в защищенном от записи сегменте
#GP(0)	P	Недопустимый эффективный адрес памяти в сегментах CS, DS, ES, FS, GS
#SS(0)	P	Неверный адрес в сегменте SS
#PF(fc)	P V	Страничное нарушение
INT(13)	R V	Часть операнда вне диапазона адреса от 0 до 0FFFFH

Примечания

Команда NEG для изменения знака числа применяется довольно часто.

Пример

```
MOV     AX,579BH    ; Загружает значение в AX.
NEG     AX          ; В AX 0A865H.
```

NOP Нет операции (8086)

КОП	Формат	Такты
90	NOP	3

Флажки

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0									0		0		1	

Псевдокод

Ничего не делать 3 такта.

Операция

Команда NOP заннмает байт кода и выполняется за трн такта. Нн однн элемент не прнменяется. Команда NOP — это альтернатнвное названне команды XCHG AX,AX.

Особые случаи

Нет.

Примечания

Команда NOP ннчего не делает, а это необходимо достаточно часто. Она прнменяется прн отладке н для внесенння «заплат». Ассемблер также прнменяет команды NOP. Если он не может предсказать чнсло байт в ко-

манде, он занимает максимальную область, и ненужная часть после определения точного размера заполняется командами NOP.

Пример

```
NOP   ; Не делать ничего.
```

NOT Инвертирование (8086)

КОП	Формат	Такты
F6 [2]	NOT r/mb	2/6
F7 [2]	NOT r/mw	2/6
F7 [2]	NOT r/md	2/6

Флажки

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0									0		0		1	

Псевдокод

```
REPEAT
    Инвертировать бит в операнде
UNTIL до инвертирования всех бит в операнде
```

Операция

Команда NOT выполняет логическую функцию НЕ над своими операндами и оставляет результат в операнде (см. табл. 4.1). Прибавление к результату 1 дает дополнительный код.

Особые случаи	Режимы	Причины
#GP(0)	P	Результат в защищенном от записи сегменте
#GP(0)	P	Недопустимый эффективный адрес памяти в сегментах CS, DS, ES, FS, GS

#SS(0)	P	Неверный адрес в сегменте SS
#PF(fc)	P V	Страничное нарушение
INT(13)	R V	Часть операнда вне диапазона адреса от 0 до 0FFFFH

Примечания

Команда NOT применяется для проверки бит и сравнений; многие из них можно выполнять с помощью новых команд манипуляций битами.

Для многих применений является недостатком то, что команда NOT не воздействует на флажки.

Пример

```
MOV     AX,579BH    ; Загружает данные в AX.
NOT     AX          ; AX=0A864H.
```

OR Логическое ИЛИ (8086)

КОП	Формат	Такты
0C ib	OR AL,ib	2
0D iw	OR AX,iw	2
0D id	OR EAX,id	2
80 [1] ib	OR r/mb,ib	2/7
81 [1] iw	OR r/mw,iw	2/7
81 [1] id	OR r/md,id	2/7
08 [r]	OR r/mb,rb	2/7
09 [r]	OR r/mw,rw	2/7
09 [r]	OR r/md,rd	2/7
0A [r]	OR rb,r/mb	2/7
0B [r]	OR rw,r/mw	2/7
0B [r]	OR rd,r/md	2/7

Флажки

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0			0				S	S	0		0	S	1	0

Псевдокод

```
REPEAT
    IF бит в получателе 0 и соответствующий бит в
         источнике 0 THEN
      сохранить бит получателя 0
    ELSE
      установить бит получателя в 1
    END IF
UNTIL до проверки всех бит в получателе
```

Операция

Команда OR выполняет логическую функцию ИЛИ над операндами и оставляет результат в первом операнде (см. табл. 4.1). Эта функция часто называется «включающим» ИЛИ. Операнды команды OR должны иметь одинаковую длину.

Особые случаи	Режимы	Причины
#GP(0)	P	Результат в защищенном от записи сегменте
#GP(0)	P	Недопустимый эффективный адрес памяти в сегментах CS, DS, ES, FS, GS
#SS(0)	P	Неверный адрес в сегменте SS
#PF(fc)	P V	Страничное нарушение
INT(13)	R V	Часть операнда вне диапазона адреса от 0 до OFFFFH

Примечания

Команда OR применяется для установки бит; это нельзя реализовать новыми командами манипуляций битами.

Пример

```
MOV    AX,5963H    ; Загружает число в AX.
MOV    BX,6CA5H    ; Загружает число в BX.
OR     AX,BX       ; AX=7DE7H.
```

OUT Вывод в порт (8086)

КОП	Формат	Такты
E6 ib	OUT ib,AL	3
E7 ib	OUT ib,AX	3
E7 ib	OUT ib,EAX	3
EE	OUT DX,AL	4
EF	OUT DX,AX	4
EF	OUT DX,EAX	4

Флажки

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0									0		0		1	

Псевдокод

```
IF первый операнд непосредственный THEN
   Расширить нулями первый операнд до 16 бит
 ELSE
   Адрес выходного порта находится в DX
 END IF
IF размер второго операнда байт THEN
   Передать байт из AL в выходной порт
ELSE IF размер второго операнда слово THEN
   Передать слово из AX в выходной порт
ELSE (*размер второго операнда двойное слово*)
   Передать двойное слово из AX в выходной порт
 END IF
```

Операция

Команда OUT применяется для вывода одного байта, слова или двойного слова в порт периферийного устройства. Номер порта может быть числом от 0 до

65535. Обычно устройство имеет несколько отдельных портов для команд, слова состояния и данных. Первые два из них применяются для управления устройством.

Номера портов 00F8H—00FFH зарезервированы фирмой Intel и их использовать не следует.

Особые случаи	Режимы	Причины
#GP(0)	P	Текущая привилегия больше IOPL
#GP(0)	V	Некоторые биты разрешения в TSS содержат 1

Примечания

Большинство программ осуществляют вывод через вызовы операционной системы, и в них не применяется команда OUT. Часто прямой вывод, например, в видео-RAM производится командой MOV. Однако команда OUT важна для разработчиков драйверов устройств и для прямого взаимодействия с устройством. Когда требуется вывести несколько байт, следует пользоваться командой OUT. Если устройство вывода не может воспринимать данные с большой скоростью, команду OUT следует поместить в цикл и управлять скоростью передачи введением задержки.

Пример

```
MOV   AL,20H  ; Загружает код пробела в AL.
OUT   30,AL   ; Выводит его в порт 30.
```

OUTS Вывод циклический в порт (80186)

КОП	Формат	Такты (одиночная)	Такты (с повторением)
6E	OUTSB	7	5+5×N
6F	OUTSW	7	5+5×N
6F	OUTSD	7	5+5×N

Буква N обозначает число в регистре (E)CX.

Флажки

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0									0		0		1	

Псевдокод

```
IF размер операнда байт THEN
   Передать байт из DS:[(E)SI] в выходной порт с
      номером DX
ELSE IF размер операнда слово THEN
   Передать слово из DS:[(E)SI] в выходной порт с
      номером DX
ELSE (*размер операнда двойное слово*)
   Передать двойное слово из DS:[(E)SI] в выходной
      порт с номером в DX
 END IF
 IF DF=0 THEN
   Прибавить к (E)SI размер операнда (в байтах)
 ELSE
   Вычесть из (E)SI размер операнда (в байтах)
 END IF
```

Операция

Команда OUTS, как и OUT, применяется для вывода в порт периферийного устройства одного байта, слова или двойного слова.

Номер порта может быть числом от 0 до 65535. Обычно устройство имеет несколько портов для команд, слова состояния и данных. Через первые два производится управление устройством.

В команде OUTS номер порта всегда находится в DX, а источник данных адресуется DS:[(E)SI], если нет префикса замены сегмента.

Команда OUTS рассчитана на использование префикса REP, т. е. в конце команды производиться декремент или инкремент (E)SI на размер операнда.

Особые случаи	Режимы	Причины
#GP(0)	P	Текущая привилегия больше IOPL
#GP(0)	P	Недопустимый эффективный адрес памяти в сегментах CS, DS, ES, FS, GS
#SS(0)	P	Неверный адрес в сегменте SS
#PF(fc)	P V	Страничное нарушение
INT(13)	R V	Часть операнда вне диапазона адреса от 0 до OFFFFH
#GP(0)	V	Некоторые биты разрешения в TSS содержат 1

Примечания

Большинство программ осуществляют вывод через вызовы операционной системы и не используют команду OUTS. Однако команда OUTS важна для разработчиков драйверов устройств и для прямого взаимодействия с устройством. Если устройство не может воспринимать данные с большой скоростью, команду OUTS следует поместить в цикл и регулировать скорость с помощью введения команд NOP или задержек.

Пример

```
CLD                     ; Направление вперед.
LDS       ESI,OUTSTR    ; Адрес источника вывода
                        ; в DS:[ESI].
MOV       ECX,5         ; Счетчик повторения в
                        ; ECX.
MOV       DX,40         ; Номер выходного порта
                        ; в DX.
REP OUTSB               ; Вывести 5 байт в порт
                        ; 40.
```

POP Извлечение из стека в операнд (8086)

КОП	Формат	Такты
8F [0]	POP mw	5
8F [0]	POP md	5
58+rw	POP rw	4
58+rd	POP rd	4
1F	POP DS	7,*
07	POP ES	7,*
17	POP SS	7,*
0F A1	POP FS	7,*
0F A9	POP GS	7,*

*В защищенном режиме эти команды выполняются за 21 такт из-за дополнительной обработки (см. гл. 5).

Флажки

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0									0		0		1	

Псевдокод

```
IF размер операнда слово THEN
    Передать слово из SS:[(E)SP] в получатель
    Прибавить 2 к (E)SP
ELSE (*размер операнда двойное слово*)
    Передать двойное слово из SS:[(E)SP] в получа-
        тель
    Прибавить 4 к (E)SP
 END IF
```

Операция

Команда POP передает слово из вершины стека в операнд, а вершина стека адресует другое слово. В частности, команда POP копирует слово, адресуемое SS:SP в операнд. Затем SP устанавливается на SP+2 (или SP+4). Так как стек начинается по адресу в SS и растет вниз, то инкремент SP уменьшает стек.

Все программисты должны уметь работать со стеком.

При использовании команды POP имеют место две дополнительные особенности. Во-первых, при выполнении команды POP SS в реальном режиме автоматически запрещаются все прерывания до завершения следующей команды. Предполагается, что этой следующей командой будет POP SP или другая команда, модифицирующая SP.

Во-вторых, при извлечении в любой сегментный регистр в защищенном режиме процессор будет обращаться к другой области памяти, имеющей, возможно, другой уровень защиты, чем текущая. Подробнее о защите см. главу 5.

Особые случаи	Режимы	Причины
#GP, #SS, #NP	P	Загружается сегментный регистр
#GP(0)	P	Результат в защищенном от записи сегменте
#GP(0)	P	Недопустимый эффективный адрес памяти в сегментах CS, DS, ES, FS, GS
#SS(0)	P	Неверный адрес в сегменте SS
#PF(fc)	P V	Страничное нарушение
INT(13)	R V	Часть операнда вне диапазона адреса от 0 до OFFFFH

Примечания

В любой подпрограмме число включений PUSH должно быть равно числу извлечений POP. При несовпадении возникнут ошибки во время выполнения. С исполь-

зованием стека связано несколько стандартных приемов. При извлечении слово сохраняется в RAM и его можно извлечь повторно. Можно поместить значения в стек и сразу выполнить RET или IRET, зная о том, что управление передается новому значению.

Многие эти приемы не могут использоваться в новых процессорах iAPX при ужесточении ограничений защиты. Об этом нужно помнить новому поколению программистов.

Пример

См. команду PUSH.

POPA Извлечение из стека значений регистров (80186)

КОП	Формат	Такты
61	POPA	24
61	POPAD	24

Флажки

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0									0		0		1	

Псевдокод

```
(*Детали операции извлечения см. в команде POP*)
IF размер операнда слово THEN
    POP DI
    POP SI
    POP BP
    POP BX (*Удалить значение SP из стека*)
    POP BX
    POP DX
    POP CX
    POP AX
ELSE (*размер операнда двойное слово*)
    POP EDI
    POP ESI
    POP EBP
    POP EBX (*Удалить значение ESP из стека*)
    POP EBX
    POP EDX
    POP ECX
    POP EAX
 END IF
```

Операция

Команда POPA извлекает 8 слов из вершины стека в регистры (E)DI, (E)SI, (E)BP, (E)SP, (E)BX, (E)DX, (E)CX и (E)AX. Отметим, что извлекаемое для (E)SP значение уничтожается, а не сохраняется в регистре. Новое значение (E)SP таково, как будто выполнено 8 команд POP.

Особые случаи	Режимы	Причины
#SS(0)	P	Неверный адрес в сегменте SS
#PF(fc)	P V	Страничное нарушение
INT(13)	R V	Часть операнда вне диапазона адреса от 0 до OFFFFH

Примечания

Эта команда заменяет 8 отдельных команд POP и выполняется быстрее их на 8 тактов. Пользоваться ей проще, чем серией команд POP. Для упрощения кода следует пользоваться POPA (и предшествующей ей командой PUSHA), даже если в стек нужно включить два или три регистра. Однако нужно помнить и об увеличении времени выполнения команды.

Команда POPA предназначена для реализациии языков высокого уровня, когда непредсказуемые уровни вложения требуют сохранения и восстановления всех регистров при каждом вызове подпрограммы.

Знание порядка загрузки регистров позволяет реализовать несколько приемов загрузки новых значений в регистры, обходя механизмы защиты. Но в защищенном режиме могут возникнуть проблемы.

Пример

См. команду PUSHA.

POPF Восстановление флажков (8086)

КОП	Формат	Такты
9D	POPF	5
9D	POPFD	5

Флажки

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0	S	* *	S	S	*	S	S	S	0	S	0	S	1	S

*IOPL изменяется только на уровне привилегий 0; IF изменяется, когда привилегия меньше или равна IOPL.

Псевдокод

```
(*подробнее об извлечении из стека см. команду
     POP*)
IF размер операнда слово THEN
    POP в Flags
ELSE (*размер операнда двойное слово*)
    POP в EFlags
 END IF
```

Операция

Команда POPF копирует слово, адресуемое SS:[(E)SP] в регистр флажков, а затем производит инкремент SP на 2 (или на 4).

Инкремент SP соответствует уменьшению стека.

Отметим, что эта команда не изменяет флажки VM и RF. Кроме того, флажки IOPL и IF нельзя изменять,

если у текущей задачи недостаточно привилегий. В этом случае особый случай прерывания не возникает.

Особые случаи	Режимы	Причины
#SS(0)	P	Неверный адрес в сегменте SS
#GP(0)	V	Используется для эмулирования команды
INT(13)	R	Часть операнда вне диапазона адреса от 0 до OFFFFH
#GP(0)	V	IOPL менее 3

Примечания

Команда POPF, как и POPA, помогает реализовать компиляторы языков высокого уровня. Команды PUSHF и POPF позволяют сохранять и восстанавливать сразу все флажки. Обычно они применяются до и после вызова подпрограммы или в начале и в конце самой подпрограммы.

Конечно, команду POPF можно использовать и при программировании на ассемблере. Кроме совместного применения с PUSHF, она позволяет загрузить в регистр флажков нужный двоичный набор (сначала его нужно включить в стек, а затем извлечь командой POPF). При этом нужно помнить о важности флажков IOPL и NT.

Пример

См. команду PUSHF.

PUSH Включение операнда в стек (8086)

КОП	Формат	Такты
FF [6]	PUSH mw	5
FF [6]	PUSH md	5
50+r	PUSH rw	2
50+r	PUSH rd	2
6A ib	PUSH ib	2
68 iw	PUSH iw	2
68 id	PUSH id	2
0E	PUSH CS	2
1E	PUSH DS	2
06	PUSH ES	2
16	PUSH SS	2
0F A0	PUSH FS	2
0F A8	PUSH GS	2

Флажки

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0									0		0		1	

Псевдокод

```
IF размер операнда слово THEN
   Вычесть 2 из (E)SP
   Передать слово-источник в SS:[(E)SP]
ELSE (*размер операнда двойное слово*)
   Вычесть 4 из (E)SP
   Передать двойное слово-источник  в SS:[(E)SP]
 END IF
```

Операция

Команда PUSH помещает операнд в вершину стека с предварительной коррекцией указателя стека. Так как стек начинается по адресу из регистра SS и растет вниз, то декремент SP делает стек больше.

Программисту необходимо знать, что при использовании команды PUSH возникают две дополнительные проблемы. Первая связана с командой PUSH SP. В предыдущих процессорах iAPX 86 она выполнялась обычным образом: декремент SP на 2 и включение его значения в стек. Однако в процессорах 80286 и 80386 в стек включается значение SP до декремента, что обычно и нужно программистам при включении в стек SP.

Вторая проблема почти всегда игнорируется: если при выполнении PUSH регистр (E)SP равен 1, то процессор 80386 отключается из-за недостатка стекового пространства. Однако, такая ситуация маловероятна.

Особые случаи	Режимы	Причины
#GP(0)	P	Недопустимый эффективный адрес памяти в сегментах CS, DS, ES, FS, GS
#SS(0)	P	Неверный адрес в сегменте SS
#PF(fc)	P V	Страничное нарушение
*****	R V	Отключение из-за недостатка стекового пространства

Примечания

Помните, что в подпрограмме число команд PUSH и POP должно быть одним и тем же. Неравенство вызовет серьезные проблемы во время выполнения.

С использованием стека связано несколько стандартных приемов. Можно включить в него несколько слов и затем изменить SP так, что они не будут мешать при вызовах подпрограмм. Когда данные больше не нужны, следует просто восстановить прежнее значение SP. Можно также поместить в стек значения и сразу выполнить RET или IRET, зная, что управление передается новому значению. Многие из этих приемов не могут использо-

ваться в процессоре 80386 по мере увеличения размера слова и ограничений защиты.

Пример

```
PUSH   EAX              ; Освободить EAX.
IMUL   EAX,MEMLOC,10    ; Умножить MEM на 10.
MOV    MEMLOC,EAX       ; Сохранить результат.
POP    EAX              ; Восстановить старое
                        ; значение EAX.
```

PUSHA Включение в стек значений регистров (80186)

КОП	Формат	Такты
60	PUSHA	18
60	PUSHAD	18

Флажки

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0									0		0		1	

Псевдокод

```
(*Об операции включения см. команду PUSH*)
IF размер операнда слово THEN
    Сохранить значение SP во внутреннем регистре
    PUSH AX
    PUSH CX
    PUSH DX
    PUSH BX
    PUSH сохраненное значение SP
    PUSH BP
    PUSH SI
    PUSH DI
ELSE (*размер операнда двойное слово*)
    Сохранить значение ESP во внутреннем регистре
    PUSH EAX
    PUSH ECX
    PUSH EDX
    PUSH EBX
    PUSH сохраненное значение ESP
    PUSH EBP
    PUSH ESI
```

```
    PUSH EDI
END IF
```

Операция

Команда PUSHA включает в вершину стека 8 слов содержимого регистров (E)AX, (E)CX, (E)DX, (E)BX, (E)SP, (E)BP, (E)SI и (E)DI. Для (E)SP включается значение до выполнения команды. Новое значение (E)SP таково, как будто выполнены 8 команд PUSH.

Если SP содержит 1, 3 или 5 до выполнения PUSHA, то процессор 80386 отключается без выполнения команд PUSH. Если SP — нечетное число от 7 до 15, возникает особый случай прерывания 13.

Особые случаи	Режимы	Причины
#SS(0)	P	Неверный адрес в сегменте SS
#PF(fc)	P V	Страничное нарушение
*****	R V	Отключение из-за недостатка стекового пространства
INT(13)	R	Часть операнда вне диапазона адреса от 0 до OFFFFH

Примечания

Эта команда заменяет 8 отдельных команд PUSH, требуя всего два лишних такта, но занимая только 1/8 часть кода. Главное ее достоинство — простота включения в стек всех регистров общего назначения.

Для упрощения кода PUSHA (с последующей POPA) следует применять даже при необходимости сохранения в стеке 2 или 3 регистров. В части экономии времени команда PUSHA не дает преимуществ.

Команда PUSHA важна для реализации языков высокого уровня, когда непредсказуемые уровни вложения требуют при вызове каждой процедуры сохранения и восстановления содержимого всех регистров.

Пример

```
SUBROUTINE:
            PUSHA   ; Сохранить значения всех
                    ; регистров в стеке.
            :   :   :
            POPA    ; Восстановить значения всех
                    ; регистров из стека.
            RET     ; Возврат из подпрограммы.
```

PUSHF Включение в стек флажков (8086)

КОП	Формат	Такты
9C	PUSHF	4
9C	PUSHFD	4

Флажки

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0									0		0		1	

Псевдокод

```
(*подробнее о включении в стек см. команду
    PUSH*)
IF размер операнда слово THEN
   PUSH из Flags
ELSE (*размер операнда двойное слово*)
   PUSH из EFlags
 END IF
```

Операция

Команда PUSHF уменьшает (E)SP на 2 (или 4) и затем копирует регистр (E)Flags в слово, адресуемое SS:[(E)SP]. Декремент (E)SP делает стек больше.

Процессор 80386 в реальном режиме отключается, если SP=1, из-за недостатка стекового пространства.

Особые случаи	Режимы	Причины
#SS(0	P	Неверный адрес в сегменте SS
*****	R	Отключение из-за недостатка стекового пространства
#GP(0)	V	Используется для эмулирования команды
#GP(0)	V	IOPL менее 3

Примечания

Команда PUSHF, как и PUSHA, предназначена как помощь в реализации компиляторов языков высокого уровня. Команды PUSHF и POPF позволяют сохранять и восстанавливать сразу все флажки. Они применяются либо до и после вызова подпрограммы, либо в ее начале и конце.

Конечно, команду PUSHF можно использовать и при программировании на ассемблере. Она позволяет включить в стек флажки и извлечь их оттуда в другой регистр для проверки и реализации условных переходов.

Пример

```
PUSHF                   ; Сохранить флажки в
                        ; стеке.
OR      SS:[SP],800H    ; Установить OF=1.
POPF                    ; Восстановить флажки.
```

REPcc Префикс повторения (8086)

КОП	Формат	Условие	Команды повторения	Такты
F2 *	REP *	(E)CX>0	INS, MOVS, OUTS, STOS	*
F3 *	REPE *	(E)CX>0 и ZF=1	CMPS, SCAS	*
F2 *	REPNE *	(E)CX>0 и ZF=0	CMPS, SCAS	*
F2 *	REPNZ *	(E)CX>0 и ZF=0	CMPS, SCAS	*
F3 *	REPZ *	(E)CX>0 и ZF=1	CMPS, SCAS	*

*См. описания отдельных циклических команд.

Флажки

Префиксы повторения не воздействуют на флажки, но сами циклические команды на них воздействуют (см. описания отдельных команд).

Псевдокод

```
IF (CX<>0) THEN (*если в начале CX=0, цикл не
    выполняется*)
  REPEAT
    Отреагировать на ожидающее прерывание
    Выполнить циклическую операцию после REPcc
    Декремент CX на 1 (*флажки не изменяются*)
  UNTIL условие повторения не удовлетворяется
      (*если ZF контролируется, это делается после
      команды*)
 END IF
```

Операция

Группа префиксов REP применяется только с циклическими командами.

Вначале (E)CX проверяется на 0, и если CX=0, команда не выполняется. Если CX не равен нулю, циклическая команда выполняется с повторением до тех пор, пока CX не достигнет нуля или (для REPE, REPNE, REPNZ и REPZ) флажок ZF не изменит своего значения.

В начале каждой итерации цикла обрабатываются прерывания. После этого обычным образом выполняется циклическая команда. Затем производится декремент CX без изменения флажков. Цикл повторяется до тех пор, пока удовлетворяется условие повторения.

Цикл не выполняется совсем, если первоначально CX=0, но выполняется минимум один раз независимо от состояния ZF.

Циклические команды специально созданы для работы с префиксами типа REP. При выполнении они автоматически корректируют указатели SI и DI в зависимости от состояния флажка DF (0=инкремент, 1=декремент).

Особые случаи

Префиксы повторения не генерируют особых случаев прерывания, но некоторые циклические команды формируют их (см. описания отдельных команд).

Примечания

Имеется три преимущества использования префиксов типа REP. Во-первых, они компактны; одна строка содержит эквивалент нескольких (до 6) команд. Во-вторых, экономится время выполнения; управление циклом реализуется быстрее на 10 тактов по сравнению с обычными командами. Третье достоинство — простота использования префиксов REPcc: идея программиста воплощается в одну строку кода, что редко встречается в языке ассемблер. Но из-за множества вариантов операций префиксы увеличивают возможность ошибки. Непра-

вильная инициализация CX, например, приводит к печальным последствиям.

Внимательно проверьте команды REP INS и REP OUTS, так как не все порты могут передавать данные с достаточно большой скоростью. В этом случае придется искусственно замедлять цикл.

Для различения причин прекращения цикла (CX=0 или из-за флажка ZF) можно использовать команды JCXZ или JZ/JE и JNZ/JNE сразу после REPcc.

Пример

```
LDS     SI,SRC_STR   ; Указатель массива-источ-
                     ; ника.
LES     DI,DEST_STR  ; Указатель массива-получа-
                     ; теля.
MOV     ECX,STRLEN   ; Число передаваемых байт.
REP MOVSB            ; Передать SRC_STR в
                     ; DEST_STR.
```

RET Возврат после вызова (8086)

КОП	Формат	Тип	Такты
C3	RET	Короткий	10+m
CB	RET	Длинный	18+m**
C2 iw	RET iw	Короткий*	10+m
CA iw	RET iw	Длинный*	18+m**

*С извлечением из стека параметров.
**В защищенном режиме эти команды имеют другие функции и время выполнения.

Флажки

Обычно команда RET не воздействует на флажки. Но если в защищенном режиме производится переключение задачи, все флажки изменяются на сохраненные флажки старой задачи.

Псевдокод

```
POP IP из стека
IF длинный возврат THEN
    POP CS из стека
 END IF
IF задан счетчик байт THEN
    Извлечь столько-то байт из стека
 END IF
```

Операция

Возврат после вызова в том же сегменте кода, где находится подпрограмма (короткий возврат) очень прост. Из вершины стека извлекается IP. В случае длинного возврата производится извлечение CS. В любом случае выполнение продолжается по адресу из CS:IP.

Вторая форма команды RET позволяет указать число байт параметров, находящихся в стеке. Перед возвратом в вызывающую процедуру процессор удаляет параметры из стека.

Основная процедура такая же, как в защищенном режиме, но межсегментные возвраты намного сложнее, так как команда CALL может определять процедуру операционной системы или даже другую задачу. В любом случае необходимо проверять защиту памяти (см. гл. 5).

Особые случаи	Режимы	Причины
#NP	P	Заданный сегмент кода отсутствует
#TS	P	Требуется переключение задачи
#GP(0)	P	Недопустимый эффективный адрес памяти в сегментах CS, DS, ES, FS, GS
#SS(0)	P	Неверный адрес в сегменте SS
#PF(fc)	P V	Страничное нарушение
INT(13)	R V	Часть операнда вне диапазона адреса от 0 до OFFFFH

Примечания

Много программных приемов реализуются путем включения различных значений в стек с последующим выполнением команды RET. Но при защите памяти такие приемы становятся сложнее. Например, в процессоре 80386 размер сегмента не фиксирован, поэтому возврат по адресу внутри диапазона 64 Кбайт не обязательно приходится на тот же самый сегмент. В случае использования виртуальной памяти, адрес, по которому возвращается управление, может вообще быть вне пределов оперативной памяти.

Пример

```
RET  8     ; Извлекает из стека 8 байт параметров.
```

Rxx Циклический сдвиг (8086)

КОП	Формат	Такты
D0 [2]	RCL r/mb,1	9/10
D2 [2]	RCL r/mb,CL	9/10
C0 [2] ib	RCL r/mb,ib	9/10
D1 [2]	RCL r/mw,1	9/10
D3 [2]	RCL r/mw,CL	9/10
C1 [2] ib	RCL r/mw,ib	9/10
D1 [2]	RCL r/md,1	9/10
D3 [2]	RCL r/md,CL	9/10
C1 [2] ib	RCL r/md,ib	9/10
D0 [3]	RCR r/mb,1	9/10
D2 [3]	RCR r/mb,CL	9/10
C0 [3] ib	RCR r/mb,ib	9/10
D1 [3]	RCR r/mw,1	9/10
D3 [3]	RCR r/mw,CL	9/10
C1 [3] ib	RCR r/mw,ib	9/10
D1 [3]	RCR r/md,1	9/10
D3 [3]	RCR r/md,CL	9/10
C1 [3] ib	RCR r/md,ib	9/10
D0 [0]	ROL r/mb,1	3/7
D2 [0]	ROL r/mb,CL	3/7
C0 [0] ib	ROL r/mb,ib	3/7
D1 [0]	ROL r/mw,1	3/7
D3 [0]	ROL r/mw,CL	3/7
C1 [0] ib	ROL r/mw,ib	3/7
D1 [0]	ROL r/md,1	3/7
D3 [0]	ROL r/md,CL	3/7
C1 [0] ib	ROL r/md,ib	3/7
D0 [1]	ROR r/mb,1	3/7
D2 [1]	ROR r/mb,CL	3/7
C0 [1] ib	ROR r/mb,ib	3/7
D1 [1]	ROR r/mw,1	3/7
D3 [1]	ROR r/mw,CL	3/7
C1 [1] ib	ROR r/mw,ib	3/7
D1 [1]	ROR r/md,1	3/7
D3 [1]	ROR r/md,CL	3/7
C1 [1]ib	ROR r/md,ib	3/7

Флажки

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0			*						0		0		1	S

*Флажок OF устанавливается только в однобитных циклических сдвигах. Многобитные операции оставляют OF в неопределенном состоянии.

Псевдокод

```
Поместить первый операнд во внутренний регистр
DO второй операнд TIMES (*раз*)
   IF направление влево THEN
     Сохранить старший бит
   ELSE
     Сохранить младший бит
   END IF
   Сдвинуть на один бит в направлении сдвига
   IF участвует CF THEN
     IF направление влево THEN
       Поместить CF в младший бит
     ELSE
       Поместить CF в старший бит
     END IF
   ELSE
     IF направление влево THEN
       Поместить сохраненный бит в младший бит
     ELSE
       Поместить сохраненный бит в старший бит
     END IF
   END IF
   Поместить сохраненный бит в CF
ENDDO
IF второй операнд равен 1 THEN
   IF направление влево THEN
     IF старший бит не равен CF THEN
       Установить OF в 1
     ELSE
       Сбросить OF в 0
     END IF
```

```
    ELSE
      IF старший бит не равен соседнему THEN
        Установить OF в 1
      ELSE
        Сбросить OF в 0
      END IF
    END IF
END IF
Сохранить внутренний регистр в первом операнде
```

Операция

В командах циклического сдвига двоичный код в первом операнде сдвигается влево или вправо на число разрядов, определяемое вторым операндом. Биты, выдвигаемые с одного конца операнда не теряются, как при сдвиге, а вводятся с другого конца операнда.

Второй и третий символы мнемоники управляют особенностями сдвига.

Третий символ задает направление сдвига: L — влево, R — вправо.

Второй символ показывает значение флажка CF:0 (чистый сдвиг) — флажок CF содержит последний выдвинутый бит, C (сдвиг через перенос) — флажок CF считается частью сдвигаемого операнда.

Второй операнд может быть непосредственным числом или содержимым регистра CL. Однако допускаются только счетчики сдвига не более 31. Если счетчик больше 31, используются только младшие 5 бит.

Имеется также короткая форма команды для циклического сдвига на один бит.

Особые случаи	Режимы	Причины
#GP(0)	P	Результат в защищенном от записи сегменте
#GP(0)	P	Недопустимый эффективный адрес памяти в .сегментах CS, DS, ES, FS, GS
#SS(0)	P	Неверный адрес в сегменте SS
#PF(fc)	P V	Страничное нарушение
INT(13)	R V	Часть операнда вне диапазона адреса от 0 до OFFFFH

Примечания

Одним из применений циклических сдвигов в старых процессорах было ускорение умножения и деления в специальных случаях. Для процессора 80386 с большим размером слова и быстрыми командами умножения и деления этого не требуется.

Другое применение — передача бита в CF и проверка его командами JC или JNC. Новые команды проверки бит позволяют сделать эту операцию непосредственно.

Пример

```
MOV   EAX,0CADE4956H    ; Загружает значение в
                        ; EAX.
STC                     ; CF=1.
RCL   EAX,3             ; EAX=56F24AB7, CF=0.
```

SAHF Сохранение AH в регистре флажков (8086)

КОП	Формат	Такты
9E	SAHF	3

Флажки

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0							S	S	0	S	0	S	1	S

Псевдокод

Передать значение AH в младший байт регистра Flags.

Операция

Команда SAHF возвращает флажки из AH, где они анализировались. Она введена для совместимости с процессорами 8080/85 и применяется редко.

Особые случаи

Нет.

Примечания

Эта команда применяется редко. Позволяет сразу задать состояние всех флажков в младшем байте регистра.

Пример

```
LAHF            ; Загружает флажки в AH.
OR      AH,4    ; PF=1.
SAHF            ; Эмулирует команду «установить
                ; PF».
```

Sxx Сдвиг (8086)

КОП	Формат	Такты
D0 [4]	SAL r/mb,1	3/7
D2 [4]	SAL r/mb,CL	3/7
C0 [4] ib	SAL r/mb,ib	3/7
D1 [4]	SAL r/mw,1	3/7
D3 [4]	SAL r/mw,CL	3/7
C1 [4] ib	SAL r/mw,ib	3/7
D1 [4]	SAL r/md,1	3/7
D3 [4]	SAL r/md,CL	3/7
C1 [4] ib	SAL r/md,ib	3/7
D0 [7]	SAR r/mb,1	3/7
D2 [7]	SAR r/mb,CL	3/7
C0 [7] ib	SAR r/mb,ib	3/7
D1 [7]	SAR r/mw,1	3/7
D3 [7]	SAR r/mw,CL	3/7
C1 [7] ib	SAR r/mw,ib	3/7
D1 [7]	SAR r/md,1	3/7
D3 [7]	SAR r/md,CL	3/7
C1 [7] ib	SAR r/md,ib	3/7
D0 [4]	SHL r/mb,1	3/7
D2 [4]	SHL r/mb,CL	3/7
C0 [4] ib	SHL r/mb,ib	3/7
D1 [4]	SHL r/mw,1	3/7
D3 [4]	SHL r/mw,CL	3/7
C1 [4] ib	SHL r/mw,ib	3/7
D1 [4]	SHL r/md,1	3/7
D3 [4]	SHL r/md,CL	3/7
C1 [4] ib	SHL r/md,ib	3/7
D0 [5]	SHR r/mb,1	3/7
D2 [5]	SHR r/mb,CL	3/7
C0 [5] ib	SHR r/mb,ib	3/7
D1 [5]	SHR r/mw,1	3/7
D3 [5]	SHR r/mw,CL	3/7
C1 [5] ib	SHR r/mw,ib	3/7
D1 [5]	SHR r/md,1	3/7
D3 [5]	SHR r/md,CL	3/7
C1 [5] ib	SHR r/md,ib	3/7

Флажки

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0			*				S	S	0		0	S	1	S

Псевдокод

```
Поместить первый операнд во внутренний регистр
DO второй операнд TIMES (*раз*)
   IF направление влево THEN
     Поместить старший бит в CF
   ELSE
     Поместить младший бит в CF
   END IF
   Сдвинуть на один бит в направлении сдвига
   IF направление сдвига влево THEN
     Поместить 0 в младший бит
   ELSE IF команда SHR THEN
     Поместить 0 в старший бит
   ELSE (*команда SAR*)
     Поместить старый старший бит в старший бит
   END IF
END DO
IF второй операнд 1 THEN
   IF направление сдвига влево THEN
     IF старший бит не равен CF THEN
       Установить OF в 1
     ELSE
       Сбросить OF в 0
     END IF
   ELSE IF команда SHR THEN
     Сбросить OF в 0
   ELSE (*команда SAR*)
     Установить OF на старший бит
   END IF
END IF
Сохранить внутренний регистр в первом операнде
```

Операция

Команды сдвига перемещают набор бит первого операнда влево или вправо на число разрядов, равное второму операнду. Некоторые биты исчезают с одного конца операнда, а такое же число освобождающихся бит заполняется с другого конца операнда. Способ заполнения зависит от модификации команды.

Второй и третий символы мнемоники управляют особенностями сдвига.

Третий символ задает направление сдвига: L — влево, R — вправо.

Второй символ определяет арифметический (A) или логический (H) сдвиг. В логическом сдвиге освобождающиеся биты всегда заполняются нулями. В команде SAL освобождающиеся справа биты заполняются нулями. В команде SAR освобождающиеся биты слева заполняются копиями значения знакового бита.

Второй операнд может быть непосредственным значением или содержимым регистра CL. Допускается счетчик сдвига не более 31.

Если счетчик больше 31, используются только младшие 5 бит.

Имеется специальная короткая форма команды для счетчика сдвига 1.

Особые случаи	Режимы	Причины
#GP(0)	P	Результат в защищенном от записи сегменте
#GP(0)	P	Недопустимый эффективный адрес памяти в сегментах CS, DS, ES, FS, GS
#SS(0)	P	Неверный адрес в сегменте SS
#PF(fc)	P V	Страничное нарушение
INT(13)	R V	Часть операнда вне диапазона адреса от 0 до OFFFFH

Примечания

В старых процессорах сдвиги использовались для ускорения умножения и деления в специальных случаях.

Быстрые команды умножения и деления процессора 80386 делают этот прием ненужным.

Еще одно применение сдвигов — передать бит в CF и проверить командами JC или JNC. Новые команды проверки бит в процессоре 80386 осуществляют это действие проще.

Пример

```
MOV   EAX,0CADE4956H   ; Загружает значение в
                       ; EAX.
SAR   EAX,3            ; EAX=0F95BC92AH,
                       ; CF=0.
```

SHxD Сдвиг двойной (80386)

КОП	Формат	Такты
0F A4 [r] ib	SHLD r/mw,rw,ib	3/7
0F A4 [r] ib	SHLD r/md,rd,ib	3/7
0F A5 [r]	SHLD r/mw,rw,CL	3/7
0F A5 [r]	SHLD r/md,rd,CL	3/7
0F AC [r] ib	SHRD r/mw,rw,ib	3/7
0F AC [r] ib	SHRD r/md,rd,ib	3/7
0F AD [r]	SHRD r/mw,rw,CL	3/7
0F AD [r]	SHRD r/md,rd,CL	3/7

Флажки

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0			U				S	S	0	U	0	S	1	S

Псевдокод

```
Поместить первый операнд во внутренний регистр
    IR1
Поместить второй операнд во внутренний регистр IR2
DO третий операнд TIMES (*раз*)
   IF направление влево THEN
     Поместить старший бит IR2 в CF
   ELSE
     Поместить младший бит IR2 в CF
   END IF
   Сдвинуть IR2 на один бит в направлении сдвига
   Сдвинуть IR1 на один бит в направлении сдвига
   IF направление влево THEN
     Поместить CF в младший бит IR1
   ELSE
     Поместить CF в старший бит IR1
   END IF
END DO
Сохранить IR1 в первом операнде
```

Операция

Команды двойного сдвига перемещают набор бит первого операнда влево или вправо на число разрядов, определяемое третьим операндом. Это действие вызывает удаление бит с одного конца операнда и заполнение такого же числа освобождающихся бит с другого конца операнда. Освобождающиеся биты заполняются из второго операнда, как будто он тоже сдвигается на то же число бит, и выдвигающиеся с одного конца биты вдвигаются в первый операнд.

Отметим, что второй операнд не изменяется.

Третий символ мнемоники определяет направление сдвига: L — влево, R — вправо. Команда действует так, как будто операнды соединены в код двойной длины и сдвигаются вместе. После этого сохраняется новое значение первого операнда, но второй операнд не сохраняется.

В команде SHLD первый операнд слева, а второй — справа; в команде SHRD они считаются на оборот.

Третий операнд может быть непосредственным числом или содержимым регистра CL. Допускается счетчик сдвига не более 31. Если он больше 31, то используются только младшие 5 бит. Если размер операнда 16 бит (слово) и счетчик сдвига больше 15, то команда оставляет первый операнд и все флажки в неопределенном состоянии.

Особые случаи	Режимы	Причины
#GP(0)	P	Результат в защищенном от записи сегменте
#GP(0)	P	Недопустимый эффективный адрес памяти в сегментах CS, DS, ES, FS, GS
#SS(0)	P	Неверный адрес в сегменте SS
#PF(fc)	P V	Страничное нарушение
INT(13)	R V	Часть операнда вне диапазона адреса от 0 до OFFFFH

Примечания

Команды двойного сдвига действуют как команды логического сдвига первого операнда на третий операнд. Различие в том, что освобождающиеся биты заменяются битами с противоположного конца второго операнда вместо нулей.

Пример

```
;;;;;;;;;;;;;;;
; BLOCK — это смещение в текущем сегменте
; данных, адресуемом DS, массива из LENGTH
; двойных слов, который считается одним длинным
; двоичным циклом и сдвигается влево на 10 бит
; (циклически)
;;;;;;;;;;;;;;;
  MOV   EDX,BLOCK                ; Сохранить пер-
                                 ; вое двойное сло-
                                 ; во.
  MOV   ESI,0                    ; Установить ин-
                                 ; дексный регистр.
  MOV   ECX,LENGTH               ; Число двойных
                                 ; слов.
  DEC   ECX                      ; Вычесть 1.
DELOOP:
  MOV   EAX,BLOCK+4[ESI*4]       ; Следующее сло-
                                 ; во из массива.
  SHLD  BLOCK[ESI*4],EAX,10      ; Сдвинуть теку-
                                 ; щее двойное
                                 ; слово, заполняя
                                 ; из следующего.
  INC   ESI                      ; Продвижение
                                 ; индекса.
  LOOP  DELOOP                   ; Повторять до
                                 ; обработки всех
                                 ; двойных слов,
                                 ; кроме последне-
                                 ; го.
  SHLD  BLOCK[ESI*4],EDX,10      ; Сдвинуть по-
                                 ; следнее двойное
                                 ; слово, заполняя
                                 ; из первого.
```

SBB Вычитание с заемом (8086)

КОП	Формат	Такты
1C ib	SBB AL,ib	2
1D iw	SBB AX,iw	2
1D id	SBB EAX,id	2
80[3] ib	SBB r/mb,ib	2/7
81[3] iw	SBB r/mw,iw	2/7
81[3] id	SBB r/md,id	2/7
83[3] ib	SBB r/mw,ib	2/7
83[3] ib	SBB r/md,ib	2/7
18[r]	SBB r/mb,rb	2/7
19[r]	SBB r/mw,rw	2/7
19[r]	SBB r/md,rd	2/7
1A[r]	SBB rb,r/mb	2/6
1B[r]	SBB rw,r/mw	2/6
1B[r]	SBB rd,r/md	2/6

Флажки

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0			S				S	S	0	S	0	S	1	S

Псевдокод .

```
IF (источник имеет меньше бит, чем получатель)
    THEN
  расширить источник со знаком
END IF
Вычесть источник из получателя.
Вычесть CF из получателя, результат поместить в
    получатель.
```

Операция

Команда SBB вычитает второй операнд и CF из первого операнда.

Первый операнд замещается результатом, а второй не изменяется.

Особые случаи	Режимы	Причины
#GP(0)	P	Результат в защищенном от записи сегменте
#GP(0)	P	Недопустимый эффективный адрес памяти в сегментах CS, DS, ES, FS, GS
#SS(0)	P	Неверный адрес в сегменте SS
#PF(fc)	P V	Страничное нарушение
INT(13)	R V	Часть операнда вне диапазона адреса от 0 до OFFFFH

Примечания

Команда SBB в основном применяется в вычитании длинных операндов, обеспечивая автоматический учет заемов. Команда SUB (см. далее) игнорирует флажок CF.

Пример

```
MOV   AX,1329    ; Загружает 531H в AX.
MOV   BX,373     ; Загружает 175H в BX.
SUB   AL,BL      ; Вычитает 75H из 31H, давая
                 ; 0BC.
SBB   AH,BH      ; Вычитает 1 из 5, давая 3
                 ; (CF=1).
                 ; AX=956 (3BC)-разность.
```

SCAS Сканирование массива (8086)

КОП	Формат	Такты (одиночная)	Такты (с повторением)
AE	SCASB	7	5+8×N
AF	SCASW	7	5+8×N
AF	SCASD	7	5+8×N

Буква N означает число повторений.

Флажки

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0			S				S	S	0	S	0	S	1	S

Псевдокод

```
IF размер операнда 8 бит THEN
   Вычесть ES:[(E)DI] из AL, не сохранять результат
ELSE IF размер операнда 16 бит THEN
   Вычесть ES:[(E)DI] из AX, не сохранять результат
ELSE (*размер операнда 32 бита*)
   Вычесть ES:[(E)DI] из EAX, не сохранять результат
END IF
   Установить флажки по результату вычитания
IF DF=0 THEN
   Прибавить размер операнда (в байтах) к (E)DI
ELSE
   Вычесть размер операнда (в байтах) из (E)DI
END IF
```

Операция

Команда SCAS, как и CMPS, вычитает одно число из другого, но не сохраняет результат. Однако по результату вычитания устанавливаются и сбрасываются флажки. В операции участвует аккумулятор и ES:[DI]. После этого корректируется DI. Инкремент или декремент DI зависит от состояния флажка DF. Обычно с командой SCAS используются префиксы REPE и REPNE.

Особые случаи	Режимы	Причины
#GP(0)	P	Недопустимый эффективный адрес памяти в сегментах CS, DS, ES, FS, GS
#SS(0)	P	Неверный адрес в сегменте SS
#PF(fc)	P V	Страничное нарушение
INT(13)	R V	Часть операнда вне диапазона адреса от 0 до OFFFFH

Примечания

Команда SCAS применяется для сравнения каждого элемента массива с заданным значением (в аккумуляторе). Производится инициализация аккумулятора, и SCAS сравнивает его со значениями в массиве, начиная с ES:[DI]. При использовании префикса REPNZ (или REPZ) сравнение продолжается до тех пор, пока символ будет равен (или не равен) значению в аккумуляторе.

Пример

```
STR1    DD      1,2,3,4,5
        :       :
        CLD                     ; Направление вперед.
        MOV     AL,3            ; Задает значение.
        LES     EDI,STR1        ; Задает массив.
        MOV     ECX,5           ; Счетчик повторения.
        REPE SCASB              ; Выполняется 3 раза,
                                ; ECX=2,    ES:[EDI]
                                ; адресует 4.
```

SETcc Установка байта по условию (80386)

КОП	Формат	Установка условия
0F 97	SETA r/mb	Выше (CF=0 или ZF=0)
0F 93	SETAE r/mb	Выше или равно (CF=0)
0F 92	SETB r/mb	Ниже (CF=1)
0F 96	SETBE r/mb	Ниже или равно (CF=1 или ZF=1)
0F 92	SETC r/mb	Перенос (CF=1)
0F 94	SETE r/mb	Равно (ZF=1)
0F 9F	SETG r/mb	Больше (ZF=0 и SF=OF)
0F 9D	SETGE r/mb	Больше или равно (SF=OF)
0F 9C	SETL r/mb	Меньше (SF<>OF)
0F 9E	SETLE r/mb	Меньше или равно (ZF=1 или SF<>OF)
0F 96	SETNA r/mb	Не выше (CF=1 или ZF=1)
0F 92	SETNAE r/mb	Не выше или равно (CF=1)
0F 93	SETNB r/mb	Не ниже (CF=0)
0F 97	SETNBE r/mb	Не ниже или равно (CF=0 и ZF=0)
0F 93	SETNC r/mb	Нет переноса (CF=0)
0F 95	SETNE r/mb	Не равно (ZF=0)
0F 9E	SETNG r/mb	Не больше (ZF=1 или SF<>OF)
0F 9C	SETNGE r/mb	Не больше или равно (SF<>OF)
0F 9D	SETNL r/mb	Не меньше (SF=OF)
0F 9F	SETNLE r/mb	Не меньше или равно (ZF=0 или SF=OF)
0F 91	SETNO r/mb	Не переполнение (OF=0)
0F 9B	SETNP r/mb	Не паритет (PF=0)
0F 99	SETNS r/mb	Не знак (SF=0)
0F 95	SETNZ r/mb	Не нуль (ZF=0)
0F 90	SETO r/mb	Переполнение (OF=1)
0F 9A	SETP r/mb	Паритет (PF=1)
0F 9A	SETPE r/mb	Паритет четный (PF=1)
0F 9B	SETPO r/mb	Паритет нечетный (PF=0)
0F 98	SETS r/mb	Знак (SF=1)
0F 94	SETZ r/mb	Нуль (ZF=1)

Время выполнения всех команд 4/5 тактов.

Флажки

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0									0		0		1	

Псевдокод

```
IF условие установки удовлетворяется THEN
   Установить байт операнда в 1
ELSE
   Сбросить байт операнда в 0
END IF
```

Операция

Эти команды применяются с целью сохранения текущего условия для использования в дальнейшем. Они проверяют состояние одного или нескольких флажков. Если условие удовлетворяется, то байт-получатель устанавливается в 1; в противном случае он сбрасывается в 0. Отметим, что эти команды похожи на условные переходы JCC, но передачи управления не происходит.

Особые случаи	Режимы	Причины
#GP(0)	P	Результат в защищенном от записи сегменте
#GP(0)	P	Недопустимый эффективный адрес памяти в сегментах CS, DS, ES, FS, GS
#SS(0)	P	Неверный адрес в сегменте SS
#PF(fc)	P V	Страничное нарушение
INT(13)	R V	Часть операнда вне диапазона адреса от 0 до OFFFFH

Примечания

Часть условий относится к беззнаковым числам, а часть к знаковым.

Слова «выше» и «ниже» относятся к беззнаковым сравнениям, а «больше» и «меньше» — к знаковым.

Из таблицы видно, что разные мнемоники дают один и тот же код операции (и одно проверяющее условие). Избыточность объясняется тем, что одно и то же состояние флажков может означать разные ситуации в зависимости от контекста команды. Например, условная проверка часто применяется после команд CMP и SUB. Для установки подходит мнемоника SETE (установить при равенстве). Но сразу после команды DEC можно использовать тот же код операции с мнемоникой SETZ (установить при нуле), проверяя достижение счетчиком нуля. Рекомендуется тщательно выбирать мнемоники, чтобы они показывали смысл сравнения.

Команды SETcc часто применяются для построения таблиц по результату нескольких сравнений или вычислений. Это удобно для разработки компиляторов и других программ, требующих обработки вложенных уровней условий.

Также они применяются для вычисления булевых переменных.

Пример

```
MOV     ECX,5     ; Операнд для сравнения.
CMP     ECX,7     ; Сравнивает 5 и 7.
SETLE   AL        ; AL устанавливается в 1.
```

STC Установка флажка переноса (8086)

КОП	Формат	Такты
F9	STC	2

Флажки

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0									0		0		1	1

Псевдокод

Установить флажок CF в 1.

Операция

Флажок CF устанавливается в 1.

Особые случаи

Нет.

Примечания

Применяется для установки флажка при передаче параметров между программами, при подготовке к командам ADC и SBB, а также в циклических сдвигах.

Пример

```
STC   ; Устанавливает CF.
```

STD Установка флажка направления (8086)

КОП	Формат	Такты
FD	STD	2

Флажки

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0				1					0		0		1	

Псевдокод

Установить флажок DF в 1.

Операция

Флажок DF устанавливается в 1.

Особые случаи

Нет.

Примечания

Флажок DF управляет направлением циклических операций. Когда DF=1, после каждого повторения циклической операции производится декремент SI и/или DI. Такое «обратное» направление удобно, когда каждый элемент массива хранится в памяти, начиная с больших адресов, и цикл обрабатывается с первого символа. Де-

кремент применяется также, когда массив хранится обычным образом, но обрабатывается, начиная с последнего символа.

Пример

```
STD   ; Устанавливает DF.
```

STI Установка флажка разрешения прерываний (8086)

КОП	Формат	Такты
FB	STI	3

Флажки

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0					1				0		0		1	

Псевдокод

Установить флажок IF в 1.

Операция

Команда STI устанавливает флажок IF в 1, разрешая прерывания после следующей команды, если она не сбросит флажок IF.

В защищенном режиме процессоров 80286 и 80386 возникают проблемы. Команда STI может быть не выполнена, если текущий уровень привилегии программы, выполняющей STI больше (т. е. она менее привилегирована), чем биты IOPL в регистре флажков.

Особые случаи	Режимы	Причины
#GP(0)	P	Текущая привилегия больше IOPL

Примечания

Команда STI применяется для разрешения прерываний, запрещенных командой CLI, или в начале программы для гарантированного разрешения прерываний.

Хотя ограничения по привилегии на команду STI кажутся запутанными, они необходимы; если операционная система разделяется во времени между несколькими программами, она должна иметь возможность при необходимости защищать их от прерываний.

Если ваша операционная система допускает работу в защищенном режиме, важно разобраться, как она обрабатывает уровни привилегий, до попытки управлять прерываниями из своей программы.

Пример

```
STI   ; Устанавливает IF в 1.
```

STOS Сохранение массива (8086)

КОП	Формат	Такты (одиночная)	Такты (с повторением)
AA	STOSB	4	5+8×N
AB	STOSW	4	5+8×N
AB	STOSD	4	5+8×N

Буква N означает число повторений из регистра (E)CX.

Флажки

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0									0		0		1	

Псевдокод

```
IF размер операнда 8 бит THEN
   Сохранить байт из AL в байте по ES:[(E)DI]
ELSE IF размер операнда 16 бит THEN
   Сохранить слово из AX в слове по ES:[(E)DI]
ELSE (*размер операнда 32 бита*)
   Сохранить двойное слово из EAX в двойном сло-
        ве по ES:[(E)DI]
END IF
IF DF=0 THEN
   Прибавить размер операнда (в байтах) к (E)DI
ELSE
   Вычесть размер операнда (в байтах) из (E)DI
END IF
```

Операция

Команда STOS похожа на MOV передачи из AX или AX в ячейку памяти, адресуемую ES:DI, но и корректирует DI после передачи. Если DF=0, производится инкремент DI, а в противном случае — декремент на размер операнда.

Особые случаи	Режимы	Причины
#GP(0)	P	Результат в защищенном от записи сегменте
#GP(0)	P	Недопустимый эффективный адрес памяти в сегментах CS, DS, ES, FS, GS
#SS(0)	P	Неверный адрес в сегменте SS
#PF(fc)	P V	Страничное нарушение
INT(13)	R V	Часть операнда вне диапазона адреса от 0 до OFFFFH

Примечания

Команду STOS нельзя использовать с префиксами REPE или REPZ, так как она не воздействует на флажки; указание префикса REP заставляет символ из AL копироваться во все элементы массива.

Команды LODS и STOS можно использовать совместно для передачи массива из DS в ES с любыми условными проверками или изменениями между LODS и STOS. Для передачи массива без изменений эффективнее применять команду MOVS.

Пример

```
CLD                     ; Направление вперед.
XOR     EAX,EAX         ; Сбрасывает EAX.
LES     EDI,BIGARRAY    ; Адрес массива-получате-
                        ; ля.
MOV     ECX,1000        ; Счетчик повторений.
REP STOSD               ; Передача 1000 двойных
                        ; слов из нулей.
```

SUB Вычитание (8086)

КОП	Формат	Такты
2C ib	SUB AL,ib	2
2D iw	SUB AX,iw	2
2D id	SUB EAX,id	2
80 [5] ib	SUB r/mb,ib	2/7
81 [5] iw	SUB r/mw,iw	2/7
81 [5] id	SUB r/md,id	2/7
83 [5] ib	SUB r/mw,ib	2/7
83 [5] ib	SUB r/md,ib	2/7
28 [r]	SUB r/mb,rb	2/7
29 [r]	SUB r/mw,rw	2/7
29 [r]	SUB r/md,rd	2/7
2A [r]	SUB rb,r/mb	2/6
2B [r]	SUB rw,r/mw	2/6
2B [r]	SUB rd,r/md	2/6

Флажки

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0			S				S	S	0	S	0	S	1	S

Псевдокод

```
IF (источник короче получателя) THEN
    расширить источник со знаком
END IF
Вычесть источник из получателя, поместить результат в получатель
```

Операция

Команда SUB вычитает второй операнд из первого. Первый операнд замещается результатом, а второй операнд не изменяется.

Особые случаи	Режимы	Причины
#GP(0)	P	Результат в защищенном от записи сегменте
#GP(0)	P	Недопустимый эффективный адрес памяти в сегментах CS, DS, ES, FS, GS
#SS(0)	P	Неверный адрес в сегменте SS
#PF(fc)	P V	Страничное нарушение
INT(13)	R V	Часть операнда вне диапазона адреса от 0 до OFFFFH

Примечания

Команда SUB применяется в простом вычитании, а для вычитания чисел кратной точности требуется команда SBB (учитывающая заемы).

Важно понимать, как команда SUB воздействует на флажки, так как после нее применяется условный переход.

Пример

```
MOV   AX,1329     ; Загружает 531H в AX.
MOV   BX,373      ; Загружает 175H в BX.
SUB   AX,BX       ; AX=956(3BCH).
```

TEST Логическое сравнение (8086)

КОП	Формат	Такты
A8 ib	TEST AL,ib	2
A9 iw	TEST AX,iw	2
A9 id	TEST EAX,id	2
F6 [4] ib	TEST r/mb,ib	2/5
F7 [4] iw	TEST r/mw,iw	2/5
F7 [4] id	TEST r/md,id	2/5
84 [r]	TEST r/mb,rb	2/5
85 [r]	TEST r/mw,rw	2/5
85 [r]	TEST r/md,rd	2/5
84 [r]	TEST rb,r/mb	2/5
85 [r]	TEST rw,r/mw	2/5
85 [r]	TEST rd,r/md	2/5

Флажки

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0			0				S	S	0		0	S	1	0

Псевдокод

```
REPEAT
   IF бит получателя равен 1 и соответствующий
        бит источника равен 1 THEN
   установить бит в результате (внутри 80386) в 1
ELSE
   сбросить бит в результате (внутри 80386) в 0
END IF
UNTIL проверки всех бит операндов
Установить флажки по результату
```

Операция

Команда TEST выполняет логическую функцию И над операндами, но нигде не сохраняет результат. Итогом команды являются состояния флажков.

Операнды команды TEST должны иметь одинаковый размер.

Особые случаи	Режимы	Причины
#GP(0)	P	Недопустимый эффективный адрес памяти в сегментах CS, DS, ES, FS, GS
#SS(0)	P	Неверный адрес в сегменте SS
#PF(fc)	P V	Страничное нарушение
INT(13)	R V	Часть операнда вне диапазона адреса от 0 до OFFFFH

Примечания

Команда TEST часто применяется для проверки единичного состояния одного бита. Для этого число сравнивается с непосредственным значением, в котором нужный бит установлен. О результате проверки сигнализирует ZF (ZF=0, если бит сброшен). Новые команды проверки бит в процессоре 80386 осуществляют эту операцию проще, но несколько дольше (3 такта вместо двух).

Пример

```
MOV   AX,9563H    ; Загружает число в AX.
TEST  AX,0C6A5H   ; SF=1, ZF=0, PF=1.
```

WAIT Ожидание сопроцессора (8086)

КОП	Формат	Такты
9B	WAIT	6*

*Это минимальное значение при условии, что сигнал BUSY уже пассивный.

Флажки

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0									0		0		1	

Псевдокод

```
WHILE пока сигнал BUSY активный DO
ENDDO
```

Операция

Команда WAIT применяется для синхронизации процессора 80386 с сопроцессорами 80287 или 80387. Эти сопроцессоры ускоряют численные расчеты, особенно с плавающей точкой, работая параллельно с 80386. Когда программе процессора 80386 нужен результат этих расчетов, он должен ожидать окончания операции сопроцессора. Когда сопроцессор работает, он формирует активный сигнал BUSY, а по завершении операции сигнал становится пассивным. Команда WAIT при пассивном сигнале ничего не делает, а выполнение переходит к следующей команде. Программе процессора 80386 гарантируется получение нужного результата.

Особые случаи	Режимы	Причины
#NM	P R V	Установлен флажок переключения задачи в слове состояния машины
#MF	P R V	Сигнал ERROR# (незамаскированная численная ошибка)

Примечания

Команды сопроцессора в данной книге не рассматриваются, но команда WAIT приведена, потому что она управляет процессором 80386, а не сопроцессором.

Пример

```
WAIT ; Ожидает окончания операции сопроцессора.
```

XCHG Обмен (8086)

КОП	Формат	Такты
90+r	XCHG AX,rw	3
90+r	XCHG rw,AX	3
90+r	XCHG EAX,rd	3
90+r	XCHG rd,EAX	3
86 [r]	XCHG rb,r/mb	3/5
86 [r]	XCHG r/mb,rb	3/5
87 [r]	XCHG rw,r/mw	3/5
87 [r]	XCHG r/mw,rw	3/5
87 [r]	XCHG rd,r/md	3/5
87 [r]	XCHG r/md,rd	3/5

Флажки

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0									0		0		1	

Псевдокод

Передать содержимое получателя во внутренний регистр.
Заменить получатель операндом-источником.
Заменить источник операндом-получателем (из внутреннего регистра).

Операция

Команда XCHG обменивает содержимое двух операндов в одной команде. Без нее для этой функции потребовалось бы три команды MOV (и регистр) или команда PUSH, MOV, POP. При выполнении этой коман-

ды всегда передается сигнал блокировки шины LOCK. Следовательно, передачу данных не может прервать другое устройство, использующее системную шину.

Особые случаи	Режимы	Причины
#GP(0)	P	Минимум один из операндов в защищенном от записи сегменте
#GP(0)	P	Недопустимый эффективный адрес памяти в сегментах CS, DS, ES, FS, GS
#SS(0)	P	Неверный адрес в сегменте SS
#PF(fc)	P V	Страничное нарушение
INT(13)	R V	Часть операнда вне диапазона адреса от 0 до OFFFFH

Примечания

Главное назначение команды XCHG — обеспечить взаимодействие процессов. Так как команда XCHG всегда блокируется, каждый из двух независимых процессов (возможно на разных процессорах) может использовать ее для доступа к разделенной переменной, не заботясь о конфликтах. Пусть две задачи должны обращаться к одному выходному буферу. Можно выделить слово для флажка о доступности буфера: 0=буфер свободен, 1=одна из задач использует буфер. До загрузки в буфер данных каждая задача должна сначала поместить 1 в регистр, обменять его с флажком буфера и проверить регистр на нуль. Если регистр нулевой, задача может безопасно использовать буфер, зная, что флажок буфера содержит 1 и другая задача не попытается использовать буфер. Если бы команда XCHG не блокировалась, возможна ситуация, когда обе задачи полагают о возможности безопасного доступа к буферу. После окончания работы с буфером задача должна поместить во флажок нуль.

Во многих алгоритмах сортировки требуется обмен неверно размещенных элементов. С помощью команды XCHG число требуемых команд сокращается с четырех до трех, и нужен один регистр, а не два. Этот способ поясняет следующий пример.

Пример

```
XCHG  AX,DATA1   ; AX содержит DATA1 и наоборот.
XCHG  AX,DATA2   ; В DATA2 теперь DATA1, а в
                 ; AX находится DATA2.
XCHG  AX,DATA1   ; В AX исходное значение, а в
                 ; DATA1 находится DATA2.
```

XLAT Табличное преобразование (8086)

КОП	Формат	Такты
D7	XLAT	*5*

Флажки

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0									0		0		1	

Псевдокод

Передать байт из DS:[(E)BX+беззнаковое AL] в AL.

Операция

Команда XLAT применяется для преобразования индекса таблицы в табличное значение. Она действует только с таблицами байт, имеющими 256 байт или менее. Базовый адрес таблицы находится в (E)BX, а индекс в AL. Команда передает адресуемый байт в AL.

Особые случаи	Режимы	Причины
#GP(0)	P	Недопустимый эффективный адрес памяти в сегментах CS, DS, ES, FS, GS
#SS(0)	P	Неверный адрес в сегменте SS
#PF(fc)	P V	Страничное нарушение
INT(13)	R V	Часть операнда вне диапазона адреса от 0 до OFFFFH

Примечания

Обычно команда XLAT применяется для преобразования из одного символьного кода в другой. Программист создает 256-байтную таблицу нужного кода, а команда XLAT осуществляет простое и быстрое преобразование.

Команда XLAT часто применяется для «классификации символов», которое требуется в компиляторах, ассемблерах и других программах, на входе которых есть интерпретируемая символьная таблица. Таблица для XLAT содержит коды классификации символов:

1=все прописные буквы, 2=все строчные буквы, 3=цифры и т. д.

Команда XLAT и несколько простых команд сравнений и условных переходов легко определяют тип символа и выполняют необходимую обработку.

Как альтернативу можно использовать таблицу переходов.

Пример

```
TABLE25 DB   0,25,50,75,100,125,150,175,200,225,250
                                ; кратные 25.
        :    :
        CMP  AL,10              ; Кратные до 10.
        JA   TOOBIG             ; Переход, вне диапазона.
        LDS  BX,TABLE25         ; Адрес таблицы для
                                ; XLAT.
        XLAT                    ; Быстрое умножение на
                                ; 25 малых чисел.
```

XOR Исключающее ИЛИ (8086)

КОП	Формат	Такты
34 ib	XOR AL,ib	2
35 iw	XOR AX,iw	2
35 id	XOR EAX,id	2
80 [6] ib	XOR r/mb,ib	2/7
81 [6] iw	XOR r/mw,iw	2/7
81 [6] id	XOR r/md,id	2/7
30 [r]	XOR r/mb,rb	2/7
31 [r]	XOR r/mw,rw	2/7
31 [r]	XOR r/md,rd	2/7
32 [r]	XOR rb,r/mb	2/6
33 [r]	XOR rw,r/mw	2/6
33 [r]	XOR rd,r/md	2/6

Флажки

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0			0				S	S	0		0	S	1	0

Псевдокод

```
REPEAT
    IF если бит в получателе совпадает с соответст-
        вующим битом получателя THEN
      сбросить в 0 бит получателя
    ELSE
      установить в 1 бит получателя
    END IF
UNTIL до проверки всех бит в получателе
```

Операция

Команда XOR выполняет логическую функцию Исключающего ИЛИ над двумя операндами и помещает результат в первый операнд (см. табл. 4.1).

Размеры операндов в команде XOR должны быть одинаковыми.

Особые случаи	Режимы	Причины
#GP(0)	P	Результат в защищенном от записи сегменте
#GP(0)	P	Недопустимый эффективный адрес памяти в сегментах CS, DS, ES, FS, GS
#SS(0)	P	Неверный адрес в сегменте SS
#PF(fc)	P V	Страничное нарушение
INT(13)	R V	Часть операнда вне диапазона адреса от 0 до OFFFFH

Примечания

Команда XOR применяется для реализации функций проверки и сравнения бит, многие из которых проще реализуются новыми командами манипуляций битами процессора 80386.

Пример

```
MOV   AX,5963H   ; Загружает число в AX.
XOR   AX,6CA5H   ; AX=35C6H; SF=0, ZF=0, PF=1.
```

Глава 5

ЗАЩИЩЕННЫЙ РЕЖИМ

При разработке программ для 80386 прикладные программисты сталкиваются с проблемой выбора. С одной стороны, программы для 8086 прямо выполняются на новом процессоре, и программисту можно не изучать ничего нового. Однако в защищенном режиме работы процессоров 80286/386 имеется множество новых возможностей. Они изменяют ту среду, в которой будут выполняться даже простейшие программы.

В данной главе поясняется смысл таких понятий, как задачи, сегменты и страницы. Конечно, после изучения ее вы не превратитесь сразу в системного программиста, но разберетесь в возможностях и ограничениях систем с процессором 80386.

5.1. МУЛЬТИЗАДАЧНОСТЬ

Слово «задача» часто применяется для описания всего того, что может делать компьютер. Важно понимать сущность задачи в любом компьютере и особенности ее реализации процессором 80386. Этот процессор позволяет разработчику операционной системы реализовать несколько задач одновременно, а отдельную прикладную программу как бы выполнять на простом однозадачном компьютере.

Пусть два процессора A и B разделяют общую область памяти, но выполняют две совершенно разные программы. Предположим, что мы хотим обменять программы, выполняющиеся на этих процессорах. Что нужно для этого сделать?

Состояние компьютера на базе 80386 в любой момент времени можно описать содержимым регистров. Они показывают, какая команда выполняется, к каким данным в памяти происходит обращение, и содержат другую используемую процессором информацию.

Следовательно, нам нужно обменять содержимое регистров процессоров. В процессор А передается содержимое сегментных регистров, указателя команды и других регистров процессора В, а в процессор В соответствующие данные от процессора А. Учтем также содержимое регистра флажков, и проблема решена — задания процессоров обменялись.

Такой обмен очень похож на «переключение задачи» в процессоре 80386. Так как есть только один процессор, то программа использует ресурсы всей системы. После того, как программе дается сигнал «освободить микросхему», она сохраняет содержимое всех своих регистров в области памяти, называемой сегментом состояния задачи TSS, который содержит копии всех регистров и другую информацию. Регистры перезагружаются из TSS программы, которой передается управление. После этого выполняется новая программа, а старая ожидает вызова.

Такое представление помогает понять смысл мультизадачности в 80386. Мультизадачность не обязательно подразумевает несколько пользователей; даже одна программа, например, текстовый процессор, может запустить задачу типа буферизации печати и отменить ее, когда печать закончена; при этом задача-потомок скрыта от пользователя. Однопользовательская мультизадачная система может иметь операционную систему (OS), которая управляет несколькими программами, выполняющимися одновременно как отдельные задачи; пользователь может назначать им приоритеты и переключать их по своему желанию.

Задачу можно считать также выполняющейся программой или OS, которые в свою очередь управляются «гипервизором». Так реализован режим виртуального 8086 (см. гл. 6). Гипервизор может иметь прикладные программы 80386, выполняющиеся прямо под ним, а также несколько прикладных программ и OS 8086, причем программы никогда не «знают» о том, что они переключаются.

Многопользовательские системы можно реализовать, позволяя каждому пользователю работать в качестве отдельной задачи (в любой из приведенных выше форм). Процессор переключается между задачами так быстро, что у каждого пользователя создается иллюзия монопольного владения компьютером. Можно также позволить каждому пользователю реализовать мультизадачность на его терминале, но это может привести к перегрузке процессора. Число одновременно выполняющихся задач без заметного конфликтования зависит от типа жесткого диска, памяти и других периферийных устройств, а также от времени, занимаемого каждой задачей.

5.1.1. Поддержка мультизадачности в 80386

Несколько структур данных, хранимых в стандартной форме для быстрой обработки, управляются процессором и специальным регистром задачи TR, адресующим текущую задачу. Благодаря этому процессор может переключаться с одной задачи на другую за 268 тактов (17 микросекунд при 16 МГц), что соответствует почти 60000 переключений в секунду, если задачи не занимают время. Следовательно, ежесекундно можно делать множество переключений задач, оставляя достаточно времени и на выполнение программ.

Ниже приведены программные структуры.

1. Специальный сегмент состояния задачи TSS имеет минимум 26 двойных слов. Он имеет место для копий всех регистров 80386, включая EFlags, EIP и другие. Он также содержит 16-битный указатель предыдущего TSS, который нужен при возврате. Указатель полезен, когда подчиненная задача (например, обработчик прерывания) вызвана другой задачей и должна при окончании возвратить управление вызывающей задаче.
2. Имеется также специальный дескриптор TSS. Все сегменты имеют дескрипторы, дающие необходимую информацию о задаче (местоположение, размер и уровень привилегий). Для дескриптора TSS длина должна быть не менее 103 байт

(104 при использовании карты разрешения ввода/вывода; ее размер хранится в последнем слове TSS). Допускается использование TSS большего размера, и пользователь сам определяет значения в дополнительных байтах. Отметим, что TSS не является реентрантным; если задача занята (показывает бит Busy в дескрипторе TSS), вновь ее запустить нельзя.

3. Имеется также дескриптор шлюза задачи. Это короткая структура данных, которая разрешает доступ к TSS (другие шлюзы допускают доступ к другим структурам данных). Формат дескриптора шлюза задачи показан на рис. 5.1.

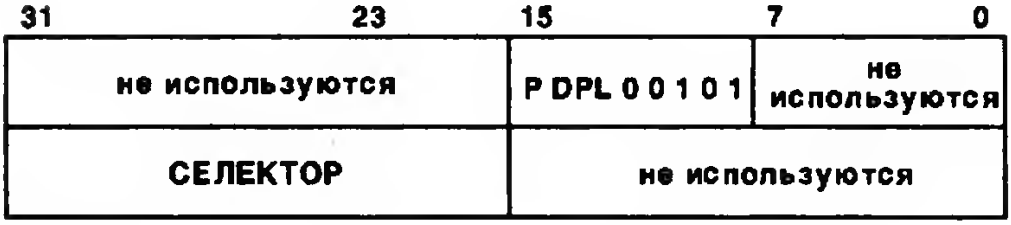


Рис. 5.1. Дескриптор шлюза задачи

Поле DPL задает уровень привилегий шлюза задачи. Процедура, которая обращается к шлюзу задачи, должна иметь уровень привилегий не больше уровня привилегий шлюза задачи. Если доступ разрешается, селектор допускает обращение к TSS без дальнейшего контроля защиты. Это значит, что процедура, которая не может прямо обратиться к данному TSS, может достигнуть его через шлюз задачи. В отличие от TSS дескриптор шлюза задачи может находиться в локальной таблице данных и «видим» некоторым процедурам (включая обработчики прерываний), но не всем. Этим обеспечивается гибкий доступ к задаче, которая может иметь несколько шлюзов, каждый из которых доступен разным процедурам и имеет различные уровни привилегий, но все они разделяют одно и тоже поле селектора.

Помимо программных структур к мультизадачности относится регистр задачи TR. Он похож на айсберг, так как большая часть его невидима. Видимую часть составляют старшие 16 бит; они содержат селектор, который

адресует дескриптор текущего TSS. Скрытая часть содержит 16-битную базу TSS и 16-битный ограничитель TSS. Сегмент TSS действует как внутренняя кэш-память, поэтому к этим полям можно обращаться быстро, когда программа обращается к текущему TSS. Переключение задачи происходит, когда команда JMP или CALL обращается к дескриптору TSS или шлюзу задачи, когда прерывание адресует шлюз задачи или когда текущая задача выполняет команду IRET и установлен флажок NT (вложенная задача). В последнем случае для быстрого возвращения в вызывающую задачу применяется указатель предыдущего TSS, хранимый в текущем TSS.

Осуществляется несколько проверок уровня привилегий и наличия в памяти текущего TSS (если его нет, возникает особый случай прерывания). Если проверки отвечают условиям, регистры текущей задачи сохраняются в текущем TSS. После этого в TR загружается селектор нового TSS, и в регистры помещается информация из нового TSS. Наконец, устанавливается бит TS (задача переключена) в слове состояния машины, извещая сопроцессоры о происшедшем изменении.

5.2. СЕГМЕНТАЦИЯ

Сегменты — это отдельные логические блоки памяти со своими размером, уровнем защиты и другими характеристиками, но сегменты могут перекрываться и даже быть одной областью памяти (это называется альтернативным именованием). Процессор 80386 одновременно обращается к шести сегментам, адресуемым своими 16-битными регистрами. Регистры CS, DS и SS определяют начальные адреса сегментов кода, данных и стека. Регистры ES, FS и GS определяют начальные адреса трех дополнительных сегментов. Каждый из сегментов имеет дескриптор сегмента, содержащий информацию о защите и размере.

В процессорах 8086/88 размер сегментов ограничен 64 Кбайтами. Это означает, что большинство выполняющихся программ требуют отдельных сегментов для кода, данных и стека, а средние и большие программы должны иметь доступ к нескольким сегментам для кода, данных и стека. Большие структуры данных разделяются на части границами сегментов.

Процессор 80386 допускает использование сегментов до 4 Гбайт, что обеспечивает более гибкое управление памятью. Однако структура памяти в основном определяется OS. Операционная система 8086 (или любая программа реального режима 80386 или режима виртуального 8086), выполняющаяся на 80386, будет накладывать точно те же ограничения, что и в предыдущем поколении процессора. OS 80386 накладывает такие ограничения ради совместимости с имеющимся кодом или упрощения своей работы.

Далее рассматриваются потенциальные возможности 80386.

Отметим, что 80386 может поддерживать полностью несегментированную и незащищенную модель памяти. Если установить все сегменты регистра на нуль, придать каждому сегменту предел 4 Гбайта и уровень защиты 3 (нет защиты), то указатель команды, база стека, указатель индекса и указатель данных будут обращаться к одной и той же области 4 Гбайта и в известном смысле будут взаимозаменяемы. Это необязательно для всех прикладных применений, но позволяет прямо переносить на системы 80386 OS типа UNIX (которая обычно использует плоскую модель памяти).

5.2.1. Формирование линейного адреса

В большинстве прикладных программ для обращения к памяти используется несколько режимов адресации. Они определяют окончательное значение смещения, которое называется эффективным адресом операнда в памяти. Смещение объединяется с соответствующим регистром базы сегмента и образует линейный адрес. Если страничная организация заблокирована, линейный адрес применяется для прямого доступа к памяти.

В реальном режиме содержимое сегментного регистра просто умножается на 16 (сдвиг влево на 4 разряда) и суммируется с младшими 16 битами эффективного адреса; в результате получается 20-битный адрес, как в 8086 или в реальном режиме 80286. Этот же прием используется для задач, выполняющихся в режиме виртуального 8086. В защищенном режиме линейные адреса могут иметь 32 бита, адресуя память до 4 Гбайт.

Фактическая длина сегментных регистров 64 бита, но программы «видят» только старшие 16 бит. 64-битный элемент содержит всю информацию о сегменте и называется дескриптором сегмента. Он имеет фактическую базу сегмента до 32 бит, адресующую память 4 Гбайта (линейное адресное пространство). Линейный адрес получается при суммировании базы сегмента и эффективного адреса (сдвиг не производится).

Регистры в процессоре содержат дескрипторы тех сегментов, которые использует текущая выполняющаяся задача. Однако программа 80386 может иметь много сегментов, а сам процессор может выполнять несколько задач одновременно. Дескрипторы сегментов в регистрах — это просто быстродоступные копии дескрипторов сегментов, хранимых в памяти.

5.2.2. Дескрипторы сегментов и таблицы

На рис. 5.2 показан формат дескриптора сегмента. Поля в дескрипторе интерпретируются по-разному для разных типов дескрипторов; ниже описаны поля для сегментов кода и данных.

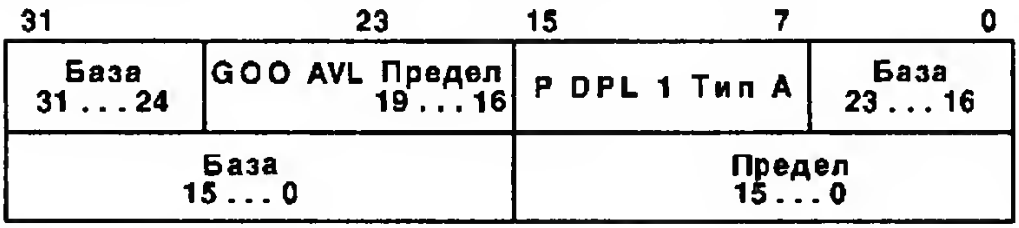


Рис. 5.2. Дескриптор сегмента

Другие типы сегментов аналогичны.

1. БАЗА. База сегмента длиной 32 бита находится в трех частях дескриптора.
2. ГРАНИЦА. Эта 20-битная величина находится в двух частях дескриптора. Обычно такую величину можно использовать для адресации памяти только 1 Мбайт, что ограничивает длину сегментов до 1 Мбайт, если бы не было бита G.
3. ГРАНУЛЯРНОСТЬ (G). Когда G=0, максимальный размер границы равен 1 Мбайту, но если

G=1, то размер умножается на 4 K (размер страницы). Теперь граница означает не только длину сегмента, но и то, что его длина может определяться заданным числом страниц. Это означает, что границу можно проконтролировать только до ближайшей к концу сегмента страницы.

4. ДОСТУПНОСТЬ (AVL). Этот бит определяет, доступен ли дескриптор для использования OS.
5. ПРИСУТСТВИЕ (P). Этот бит показывает, находится ли сегмент в памяти. Бит содержит 1 и сегмент недоступен, если применяется сегментная виртуальная память и сегмент передан на диск или в другое пространство, не отображаемое страничным устройством.
6. УРОВЕНЬ ПРИВИЛЕГИЙ ДЕСКРИПТОРА (DPL). Два бита показывают уровень привилегий, необходимый для доступа от 0 (высший) до 3.
7. ТИП (TYPE). Это 4-битное поле используется по-разному в дескрипторах разных типов. Например, оно может определять, что сегмент является исполняемым, считываемым и/или записываемым. Для сегментов OS это поле может определять тип дескриптора вида LDT, TSS или шлюз (последние два имеют свои подтипы).
8. ОБРАЩЕНИЕ (A). Этот бит устанавливается, когда дескриптор загружается в сегментный регистр.

Дескрипторы создаются и управляются OS и другими системными программами типа компиляторов. Все эти программы должны работать согласовано, обеспечивая правильную модификацию дескрипторов.

Все дескрипторы хранятся в памяти в виде таблиц. Длина таблиц может достигать 8192 (2^{13}) дескрипторов. Изменяется только глобальная дескрипторная таблица GDT; она содержит дескрипторы сегментов, доступные любой задаче с достаточно высоким уровнем привилегий для обращения к сегменту прямо или через шлюз. Каждая задача имеет также локальную дескрипторную таблицу LDT, которая описывает сегменты, доступные только данной задаче; обычно эти сегменты включают в себя код и данные задачи. Однако две задачи могут раз-

делять часть или всю LDT. Например, текстовый процессор, создающий отдельную задачу для управления печатью, может разрешить новой задаче доступ к необходимым ей коду и данным.

Одним из интересных моментов, связанных с дескрипторами сегментов, является альтернативное именование. Когда два или более дескрипторов именуют частично или полностью перекрывающиеся части линейного адресного пространства, они «подменяют» (как псевдонимы) друг друга. Например, дескриптор для сегмента кода может определять, что код считываемый, но не записываемый. Альтернативный дескриптор сегмента может определять, что та же область является записываемой; такой псевдоним можно использовать (намеренно или случайно) для изменения или перезаписи кода. OS должна следить за альтернативными именами; если OS хочет удалить сегмент, она должна удалить или изменить все дескрипторы, которые обеспечивают доступ к этой области памяти.

5.3. СТРАНИЧНАЯ ОРГАНИЗАЦИЯ

Страничная организация — это одна из наиболее интересных и новых возможностей 80386, но она предназначена только для OS. Если прикладной программист имеет доступ к части сегментных регистров, которые управляют сегментной адресацией, то страничная организация от него скрыта. Ничего подобного нет в процессорах 8086 и 80286. Но все же целесообразно разобраться в страничной организации. По-видимому, она будет основным средством доступа к большой реальной памяти и очень важна для работы виртуального режима, который будет основой мультизадачных систем и гигантских прикладных программ будущего. Можно понять идеи страничной организации и не разрабатывая OS. Для большинства программистов достаточно информации приводимой далее и сведений о конкретной OS.

Страница памяти процессора 80386 — это блок памяти размером 4 Кбайта (страничные кадры). Адреса страничных кадров 0, 4K, 8K и т. д. Любой элемент данных, который начинается по любому из этих адресов, называется «выравненным по границе страницы». Адресация страничного кадра в линейном адресном про-

странстве 4 Гбайта упрощается, так как нужен только 20-битный адрес; последние 12 бит адреса из 32 — нулевые. В виртуальной памяти по мере необходимости производится обмен страницами между памятью и диском. Секции на диске по 4 Кбайта для хранения страниц называются страничными слотами.

Разрешением и запрещением страничной организации управляет бит в слове состояния. Страничная организация нужна системным программистам, так как она обеспечивает простое назначение почти любого физического адреса для замены сформированного программой линейного адреса. Следовательно, страничная организация обязательна для OS.

Допускается работать только со страницами без сегментов, поместив все в один большой сегмент. Сегменты имеют свои достоинства, особенно для защиты всего кода или фрагментов данных. По-видимому, общая модель памяти будет использовать сегменты для защиты на уровне сегментов и страничную организацию для переотображения памяти и виртуальной памяти.

5.3.1. Формирование физического адреса

Физический адрес — это адрес, который формирует страничное устройство. Когда страничная организация блокирована (и во многих других случаях), входной линейный адрес совпадает с выходным физическим адресом. Хотя процесс преобразования применяется не всегда, следует разобраться в том, как он работает.

Физический адрес состоит из трех частей (см. рис. 5.3): 10-битный элемент DIR, 10-битный элемент PAGE (страница) и 12-битное OFFSET (смещение).

31 ... 12	11 ... 9	8	7	6	5	4	3	2	1	0
Страничный адрес	OS Res	0	0	D	A	0	0	U/S	R/W	P

Рис. 5.3. Элемент каталога страниц/элемент таблицы страниц

Эти три поля используются для образования физического адреса следующим образом.

1. 32-битный регистр CR3 содержит адрес текущего каталога страниц. Младшие 12 бит CR3 содержат нули, так как каталог страниц всегда начинается на границе страниц. 10-битное значение DIR из линейного адреса показывает нужный элемент каталога страниц.
2. Старшие 20 бит элемента каталога страниц адресуют таблицу страниц; полный физический адрес таблицы, выравненной по границе страницы, получается присоединением 12-ти младших нулей. 10-битное значение PAGE из середины линейного адреса показывает нужный элемент таблицы страниц.
3. Старшие 20 бит элемента таблицы страниц адресуют страничный кадр; для получения полного линейного адреса присоединяются 12 младших нулей. 12-битное значение OFFSET из младших бит линейного адреса адресует нужную ячейку в памяти.

Каждый каталог страниц имеет до 1024 элементов, допуская столько же таблиц страниц. Каждая таблица может адресовать до 1024 страниц по 4 Кбайта. Следовательно, один каталог может адресовать 1024×1024×4096 байт или 4 Гбайта (все физическое адресное пространство 80386).

Элементы каталога страниц (PDE) и элементы таблицы страниц (PTE) почти идентичны. Таблицу страниц можно считать «таблицей дескрипторов страниц», а элемент таблицы страниц — «дескриптором страницы». Формат элемента приведен на рис. 5.3. Вот описание его полей.

1. СТРАНИЧНЫЙ АДРЕС (биты 31—12). Для PDE этот адрес показывает таблицу страниц, а для PTE — страницу. Младшие 12 бит адреса всегда равны нулю, так как таблицы страниц и страницы выравнены по границам 4 Кбайт.
2. ЗАРЕЗЕРВИРОВАНЫ OS (биты 11—9). Эти биты доступны для использования OS. Обычно они применяются для сбора статистики о виртуальной памяти и свопинге страниц (например, число обращений к странице за данный интервал).

3. D (бит 6). Этот бит «грязный» (Dirty bit) для PTE, который автоматически устанавливается при записи в страницу.
4. A (бит 5). Это бит обращения для PDE и PTE; он устанавливается автоматически при считывании или записи в страницу.
5. U/S (бит 2). Это бит индикации пользователь/супервизор. Если он установлен, то разрешен доступ программам с уровнем защиты 3 (нижний уровень).
6. R/W (бит 1). Если U/S=1 (разрешен доступ пользователю), то R/W=0 означает возможность считывания, а R/W=1 — еще и записи. Эти биты имеют смысл для PDE и PTE. Отдельная страница защищена в соответствии с наиболее ограничивающей парой бит в ее PDE или PTE. Если U/S=0, запрещены считывание и запись в страницу. Если U/S=1 и R/W=0, из страницы можно только считывать; когда оба бита содержат 1 (в обоих элементах), разрешены считывание и запись в страницу.
7. P (бит 0). Это бит присутствия, который показывает адресует ли PDE или PTE страницу, находящуюся в памяти. Если P=1, то действия полей такие, как определено выше. Если же P=0, то нужная страница на диске и остальные 31 бит могут задавать местоположение страницы на диске.

Хотя PTE содержит много информации о странице, ее непросто получить; два обращения к памяти требуются только для получения адреса каждой страницы. Буфер преобразования TLB в страничном устройстве содержит 32 последних использованных PTE. Буфер TLB часто содержит нужный PTE, сокращая на 95% необходимость фактического обращения к таблицам в памяти (подробнее см. гл. 7).

5.4. ВИРТУАЛЬНАЯ ПАМЯТЬ

Виртуальная организация памяти позволяет программисту работать не заботясь о емкости реальной памяти. В системах виртуальной памяти диск эффективно пре-

вращается в основную память, а RAM («реальная память») содержит код и данные, используемые процессором.

Взаимосвязь этих элементов показана на рис. 5.4.

Число бит в наибольшем адресе, который может сформировать процессор, определяет размер его виртуального адресного пространства. В новых процессорах типа 80386 есть два вида размера адреса: большой адрес определяет виртуальное адресное пространство или емкость памяти, которую может использовать программист; малый адрес определяет реальное адресное пространство — емкость оперативной памяти, которую может поддерживать процессор. В 80386 адресная информация в таблицах страниц допускает использование виртуального адресного пространства до 64 Тбайт, что эквивалентно 64000 запоминающих устройств на оптических дисках по 1 Гбайту.

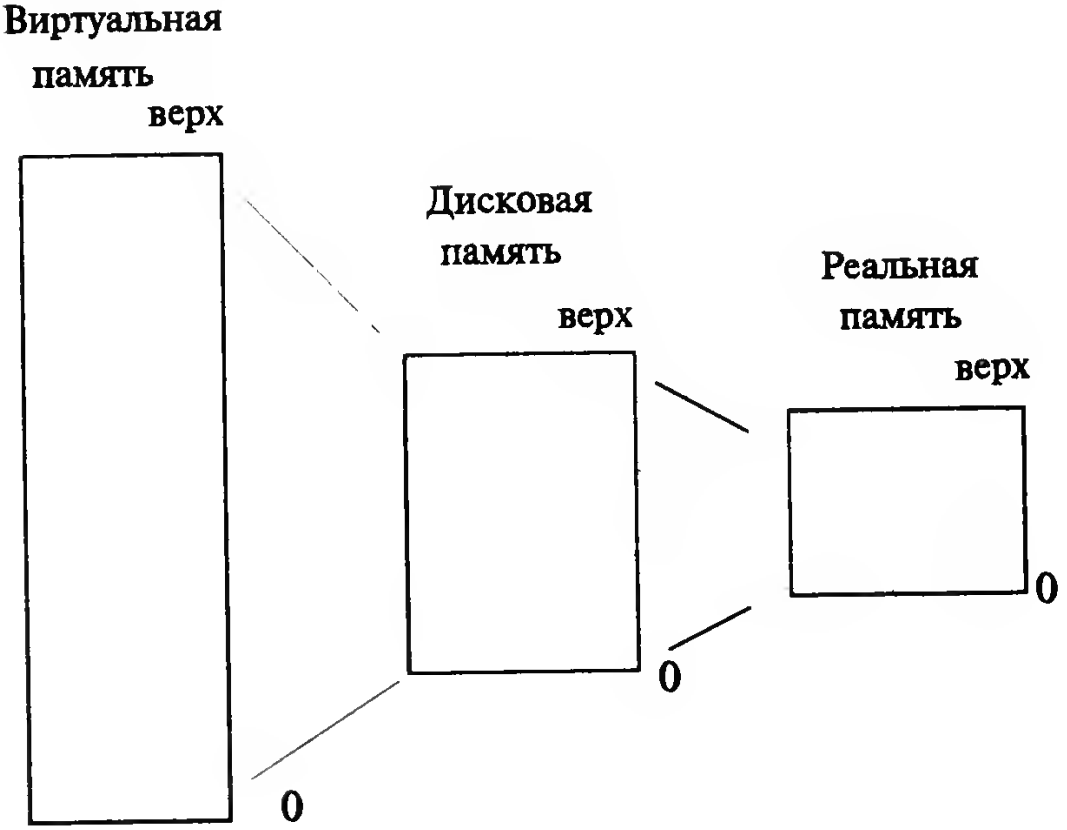


Рис. 5.4 Связь виртуальной памяти с дисковой памятью и реальной памятью

К реальному адресному пространству можно обращаться по 32-битному адресу с использованием или без использования страниц, что обеспечивает размер памяти до 4 Гбайт. Однако аппаратная реализация такой памяти пока невозможна. На практике реальные адреса применяются для доступа к RAM, а виртуальные адреса — для обращения к виртуальному пространству жесткого диска. Виртуальные адреса не используют даже всего пространства 4 Гбайт, обеспечиваемого 32-битным адресом. Большие виртуальные адресные пространства, по-видимому, будут применяться сначала для сетей и систем связи.

Простая форма виртуальной памяти уже используется большими программами. Здесь ядро программы находится в памяти все время. Остальная часть программы разделяется на блоки примерно по 64 Кбайт. Когда пользователь обращается к разным блокам программы, они передаются с диска. Каждый такой блок называется «оверлеем». Однако разделение программы на оверлеи и управление ими представляют для программиста особую задачу.

В виртуальной памяти каждая программа, файл данных и т. д. считается набором «страниц» по 4 Кбайта. Реальное адресное пространство разделяется на «страничные кадры» по 4 Кбайта, а виртуальное адресное пространство (жесткий диск) — на «страничные слоты» по 4 Кбайта. Когда пользователь обращается к разным частям программы, страницы передаются из их страничных слотов в страничные кадры реальной памяти, где к ним и обращается процессор. Когда память заполнена и требуется другая страница, имеющийся страничный кадр перезаписывается (если он не изменялся после передачи в память) или копируется в свой слот на диске до перезаписи (если он изменялся). Все это скрыто от программиста; он просто обращается к коду и данным по различным виртуальным адресам, а процессор и OS при необходимости передают нужную страницу в память.

5.4.1. Поддержка виртуальной памяти

Для поддержки виртуальной памяти важную роль играют следующие возможности:

1. Возможиость доступа к большим адресным пространствам.
2. Возможность преобразоваиия виртуальных адресов в реальные адреса для обращения к реальиой или физической памяти (RAM).
3. Возможиость генерировать особые случаи прерывания (а не остаиавливаться), если иужиой части кода или даиных нет в памяти («страиичиое нарушеиие»).
4. Перезапуск команд, если страиичиое иарушение препятствует завершению команды.

Когда возиикает страничное нарушеиие, OS должна передать иужиую страиицу с диска в память. Если память заполиеиа, OS должиа записать одиу из страниц иа диск и использовать освобожденное место.

Несколько важиых аспектов виртуальной памяти прямо касаются прикладных программистов. Например, как быть, если система с памятью 1 Мбайт выполияет одиовремеиио три программы, каждая из которых использует 640 Кбайт RAM. Миого исследоваиий по таким вопросам выполиено иа больших компьютерах, но их результаты не примеиимы прямо к системам с микрокомпьютерами.

Предварительиые оцеики показывают, что компьютер может безопасно работать с виртуальной памятью, емкость которой вдвое больше реальной памяти. Поэтому три программы по 640 Кбайт могут хорошо выполияться в системе с памятью 1 Мбайт. Важным моментом является размер «рабочего множества» каждой программы. Если каждая программа обращается к большому числу страииц повторио (имеет большой рабочий иабор), то каждая может заставить OS часто обращаться к диску, вызывая «пробуксовку». Такая ситуация может возиикиуть при повторяющихся расчетах с одновремеиным компилированием большой программы или сортировкой большого объема данных.

Глава 6

РЕЖИМ ВИРТУАЛЬНОГО 8086

В предыдущих главах рассмотрены реальный и защищенный режимы работы процессора 80386. В этой главе обсуждается режим виртуального 8086 (V8086), который позволяет выполнять задачу (например, прикладную программу) как будто в реальном режиме, но OS имеет доступ ко всем возможностям защищенного режима. Дается сравнение всех режимов 80386 с 8086, различия между процессорами и их влияние. Наконец, мы обсудим, как эти факторы соотносятся с возможностями OS. Это поможет читателю сделать обоснованный выбор OS и «выжать» из нее максимум возможного.

Поясняется также, как 80386 может одновременно выполнять программы для PC и 32-битные задачи; почему 80286 не может выполнять многие прежние программы даже в реальном режиме и почему первые (и самые дешевые) OS могут использовать только часть возможностей процессора. По существу, эта глава объединяет прошлое и будущее.

6.1. ОПРЕДЕЛЕНИЕ РЕЖИМА ВИРТУАЛЬНОГО 8086

Для компьютеров на базе процессоров 8086/88 и 80286 стоимость имеющегося программного обеспечения составляет более 5 млрд. долларов. Его (программы и OS) можно реализовывать без изменений в реальном режиме 80386. В этом режиме одновременно выполняются только OS и одна программа, обеспечивая совместимость

с программами для PC и AT, но не используя всех возможностей 80386. Для OS и программиста процессор 80386 выглядит, как более быстрый 8086 с дополнительными регистрами и командами и с тем же ограничением памяти 1 Мбайт.

Альтернативой реальному режиму является защищенный режим, в котором обеспечивается доступ ко всем ресурсам 80386: мультизадачность, организация страниц, виртуальная память и др. Компьютер в целом может работать только в реальном или защищенном режиме. Однако, хотя компьютер работает в защищенном режиме, отдельная задача (ей может быть одна программа или OS) может выполняться в режиме виртуального 8086, что обеспечивает задаче среду 8086 (такую же, как среда реального режима 80386). Тем временем OS 80386 и другие программы имеют доступ ко всем ресурсам компьютера, работающего в защищенном режиме.

Для разговора о режиме V8086 нам придется уточнить и расширить словарь. Задача — это любая независимо выполняющаяся программа; точнее, задача — это программа, которая имеет в 80386 сегмент состояния задачи TSS или его эквивалент. TSS хранит содержимое регистров и другую информацию, позволяя OS остановить и запустить задачу в любой момент времени.

OS 8086 типа DOS 2.x или другой версии может выполняться как задача. Когда пользователь выполняет эту задачу, он может запустить, выполнить и осуществить выход почти из любой программы, которая работает на 8086 или 80286. Следовательно, мы имеем OS типа DOS 2.x, выполняющуюся под собственной OS 80386, которая управляет организацией страниц, виртуальной памятью и приоритетами среди выполняющихся одновременно других задач. Чтобы избежать путаницы, мы будем иногда называть собственную OS 80386 гипервизором (чтобы отличить ее от другой OS, выполняющейся как задача). Гипервизор дает пользователю возможность задавать приоритеты и переключать несколько задач.

6.2. ВИРТУАЛЬНЫЕ МАШИНЫ

В виртуальной памяти очень большие программы и огромные структуры данных ведут себя так, как будто одновременно находятся в памяти. OS отвечает за рас-

пределение, пространства памяти и при необходимости перераспределяет код и данные в тех ситуациях, когда все сразу поместиться в память не может.

«Виртуальная машина» расширяет понятие виртуальной памяти на весь компьютер с устройствами ввода/вывода (например, клавиатура и дисплей), набором регистров и т. д. Программа выполняется так, как будто она имеет в своем распоряжении целую машину (в данном случае, компьютер на базе 8086); отсюда возникло название «режим виртуального 8086». Когда программа осуществляет вызовы системных ресурсов (например, запись в дисплей), гипервизор 80386 позволяет произвести это действие или перехватывает вызов и выполняет запись сам.

Предположим, что ассемблерная программа 8086 хочет записать текст «Hello» на экран. Программа вызывает DOS 3.1 командой INT, и DOS производит запись на экран. Однако DOS 3.1 работает как задача V8086, которая имеет уровень привилегий 3. Гипервизор назначает области ввода/вывода уровень привилегий 0, поэтому запись DOS вызывает особый случай защиты. Вектор прерывания вызывает переключение на гипервизор, который должен решить, что пытается сделать задача V8086, эмулировать это действие (производя запись на экран или игнорируя попытку), а затем возвратить управление задаче DOS V8086. Чем больше таких перехватов, тем больше замедляется ассемблерная программа 8086, когда она или DOS пытается производить операции ввода/вывода или пользоваться другими системными ресурсами. Такой же подход применим к режиму V8086 для OS CP/M-86 или UNIX.

Выше сказано, что область ввода/вывода имеет уровень привилегий 0, но адреса формируются, как в 8086 (предел 1 Мбайт), поэтому сегментная защита (рассчитанная на полные адреса 80386) не работает. Адрес 8086 передается в страничное устройство, где его можно отобразить на любую страницу, находящуюся в оперативной памяти или на диске (виртуальная память). Следовательно, OS может реализовать защиту страниц при

выполнении задачн V8086 н может поддерживать виртуальную память для нескольких задач V8086, используя страннчную организацню н таблнцы страниц.

Как же режим V8086 «обманывает» программу, которая полагает, что она выполняется на 16-бнтном 8086, а не на 32-битном 80386?

Бит 17 в регистре EFlags называется VM (виртуальный режнм). Когда начннает выполняться задача V8086, бнт VM устанавливается в 1. Этот бнт сообщает о том, что надо формировать адреса н управлять сегментамн как в реальном режиме, поэтому адреса получаются путем сдвига нужного сегмента регистра влево на 4 бита, прнбавлення 16-битного смещения и нспользовання результата как адреса памятн. Полученный 20-бнтный адрес позволяет адресовать память 1 Мбайт. Бит VM также делает доступной текущей задаче только 16-бнтную часть 32-бнтного регистра, напрнмер, EFlags. Отметнм, что бнт VM при этом недоступен задаче.

Когда бит VM установлен, некоторые привилегированные коды операцнй вызывают особые случан прерывания. Разрешено выполнять только команды н нспользовать режнмы адресацин 8086.

Теперь можно поясннть, как работает реальный режим 80386. Когда бнт разрешенння защиты в слове состояння машнны сброшен, действуют такне же ограннчения 16-бнтной адресацнн н кодов операцнй, как и в режиме виртуального 8086. В обонх случаях текущая задача выполняется, как в 8086. Разлнчие в том, что слово состояння машины управляет всей машнной, переводя ее в реальный режнм, а регистр EFlags управляет только текущей задачей; текущая задача выполняется, как в реальном режиме, но переключенне задачн на гипервизор выключает флажок VM, вновь разрешая использовать привилегнрованные коды операцнй н режимы адресацнн.

6.3. ЕЩЕ О ВИРТУАЛЬНЫХ РЕЖИМАХ

Нняе несколько подробнее рассматривается текущее состоянне внртуальных режимов. Материал непосредственно к режнму V8086 не относится и его можно пропустить.

Идея виртуальных машин впервые реализована в IBM/370. Операционная система VM для этих машин разработана в конце 60-х годов. Она состоит из двух частей: CP и CMS. Часть CP (управляющая программа) выполняет ту же функцию, что и гипервизор 80386, распределяя ресурсы задачи, перехватывая вызовы и эмулируя или игнорируя их. Отдельные программы выполняются под управлением однопользовательской однозадачной OS, называемой CMS (диалоговая мониторная система). CMS похожа на копии DOS, о которых мы говорили; она как бы управляет компьютером, но фактически она вызывает CP, когда пытается обратиться к устройствам компьютера.

CP может выполнять другую копию CP, как задачу под собой. Ни одна копия не «знает», управляет ли она компьютером; она просто действует обычным образом, используя все ресурсы системы, а «верхняя» CP перехватывает вызовы «нижней».

В процессоре 80386 ситуация другая. Он может выполнять задачи V8086, в которых дублируется среда 8086, и под ней выполняются все программы и OS 8086. Программа реального режима 80286 (типа 8086) будет также выполняться под ней, но OS 80286 и 80386 не могут выполняться как задачи под гипервизором 80386. Говорится, что 80386 может «виртуализировать» 8086, но не 80286 или себя. Объясняется это некоторыми проблемами с такими командами как PUSH и POPF.

Это означает, что 80386 может выполнять OS типа 8086, но не может выполнять версии DOS или UNIX для 80286, как «приглашенные» задачи. Эти программы могут выполняться как OS для 80386 при правильной подготовке, но процессор при этом будет полностью эмулировать 80286 и машина не будет обладать возможностями 80386 (страничная организация, мультизадачность и др.).

Когда 80386 работает в реальном режиме, он действует как быстрый 8086. При выполнении OS 80286 он действует как быстрый 80286. Но когда он выполняет задачу V8086, OS и задачи, выполняющиеся параллельно с задачей V8086, имеют доступ ко всем ресурсам 80386. Задаче V8086 не нужны все эти возможности, ей нужно только полное управление 8086. Однако задача V8086 может выполняться на 80386 медленнее, чем раньше. В случае мультизадачности задача V8086 замедляется из-за того, что не работает все время. Даже ес-

ли выполняется только одна задача V8086, многие вызовы DOS приводят к переключению задачи на гипервизор, который должен определять, что произошло, эмулировать ли запрос DOS или нет, а затем возвращать управление DOS. Между исходным вызовом DOS и возвратом выполняется множество команд, замедляя программу. Точное поведение зависит от того, какие вызовы DOS перехватываются гипервизором (вызывая дополнительную обработку), а какие разрешено выполнять.

Когда гипервизор поддерживает несколько задач V8086, он должен использовать страничную организацию. Это объясняется тем, что каждая задача формирует адреса в одном и том же диапазоне от 0 до 1 Мбайта. Страницы могут отобразить большинство этих адресов сверх 1 Мбайта. Но их нельзя использовать для разделения разными задачами OS и/или кода гипервизора. Наконец, страничная организация защищает области ввода/вывода от прямого доступа со стороны V8086.

6.4. СРАВНЕНИЕ ПРОЦЕССОРОВ И РЕЖИМОВ

Имеется два основных типа программ, которые работают на процессорах семейства 8086. К первому относятся программы реального режима, включая программы и OS 8086/88, программы реального режима 80286, программы реального режима 80386 и задачи V8086. Ко второму относятся программы защищенного режима, включающие OS и прикладные программы 80286 и 80386, которые выполняются в защищенном режиме на одном из двух процессоров. Мы не будем рассматривать совместимость между OS и программами защищенного режима 80286 и возможности 80386 выполнять программы 80286.

Мы сделаем акцент на сходствах и различиях программ реального режима. Эта информация поможет в переносе на 80386 программ 8086 и реального режима 80286 для выполнения их как задач V8086 или задач защищенного режима (возможно с прямой зависимостью от OS 80386).

6.4.1. Сравнение программ 8086 с программами реального и защищенного режимов 80386

Одно из многих достоинств процессора 80386 состоит в том, что в реальном режиме он может выполнять программы и OS 8086 почти неизменными. Он может также выполнять их как задачи V8086, поэтому они могут выполняться без изменений, в то время как гипервизор разрешает использовать страничную организацию, мультизадачность и другие возможности.

Есть два вида различий между программами 8086 и программами V8086/80386 реального режима. Первые связаны с модернизацией, например, новыми командами и регистрами. Вторые же являются сюрпризами — это те различия, которые заставляют программу 8086 вести себя по-другому при выполнении на 80386. Некоторые улучшения также могут превратиться в сюрпризы, например, повышенное быстродействие может вызвать отклонения в поведении некоторых программ. Будучи оптимистами, мы вначале остановимся на улучшениях.

Первое улучшение — скорость. Программа в реальном режиме 80386 выполняется в 6—10 раз быстрее, чем на 8086. Почти половина ускорения приходится на повышение частоты синхронизации с 5 МГц до 16 МГц и более. В программах с большим числом таких команд, как IMUL, скорость возрастает еще больше. Для пользователей большинства программ будет заметно одно различие. Если 80386 обращается к достаточно быстрому жесткому диску, быстродействие возрастает. Увеличение скорости не зависит от программиста; в программе никаких изменений не нужно. Кроме того, в любой момент программу можно выполнить на компьютере с 8086. Программа 8086 может также выполняться в режиме мультизадачности с другими программами 80386.

Дальнейшее повышение производительности в реальном режиме требует изменений в самой программе. Они делают программу меньше, быстрее и от этого более гибкой; однако программа становится несовместимой с 8086. При внесении изменений (которые не потребуют полного переписывания программы) нужно учитывать, что она не будет выполняться на компьютерах с 8086, если в ней использованы новые возможности реального режима 80386. Целесообразно указать в комментариях

новый код для 80386 и сохранить любой удаляемый код (в виде комментариев или в виде библиотечной программы). Рассмотрим эти новые возможности.

6.4.2. Новая архитектура

Процессор 80386 имеет два новых регистра FS и GS. Они выполняют такую же функцию как регистр ES, адресуя базу дополнительных сегментов данных. Однако ES используется как неявный получатель в некоторых циклических операциях. Регистры FS и GS не связаны ни с какой командой и находятся в полном распоряжении программиста. Программы 8086 можно оптимизировать, храня часто используемые переменные в сегментах, адресуемых FS и GS. Префиксы замены сегмента позволяют использовать эти регистры как базовые для адресации новых сегментов данных.

Программы реального режима могут работать также с 32-битными регистрами, например MOV EAX, EBX. Ассемблер образует префикс размера операнда для обращения к 32-битному регистру (длина регистров по умолчанию в реальном режиме 16 бит). Старшие 16 бит регистров нельзя считать нулевыми без предварительной инициализации. Сложение 32-битных операндов увеличивает быстродействие многих программ.

Отладочные регистры можно использовать непосредственно из программы или из внешнего отладчика, что ускоряет отладку программы. Управляющие регистры недоступны прикладным программам (за исключением доступа через шлюзы и вызовы, обеспечиваемые OS), а регистры отладки обычно не используются. Так как регистры отладки не входят в стандартную архитектуру 80386, в новых моделях процессора их может и не быть.

6.4.3. Новые и измененные команды

В процессорах 80186/188 и 80286 введено несколько новых команд. Скорость выполнения команд типа MUL значительно повышена. Процессор 80286 является базовым для персональных компьютеров IBM PC/AT. В реальном режиме он имеет тот же набор команд, что и

80186/188, а в защищенном режиме — все средства 80386, но для 16-битной архитектуры.

Реальный режим 80386 имеет все команды предшественников. Ниже рассмотрены новые команды.

PUSHA и POPA

Эти команды обеспечивают включение в стек и извлечение из стека всех 8 регистров общего назначения. Команды PUSHAD и POPAD работают в 80386 с 32-битными регистрами, а PUSHA и POPA только с младшими 16 битами. В 80386 эти команды выполняются почти столько же времени, что и 8 отдельных команд PUSH и POP.

INS и OUTS

Это команды блоковой пересылки байт, слов и двойных слов с автоматическим инкрементом или декрементом SI, который адресует участвующий в операции блок памяти.

ENTER и LEAVE

Эти команды включают в стек и извлекают из стека «стековый кадр». Стековый кадр содержит счетчик уровня вложения текущей процедуры, счетчик размера памяти для передачи параметров и байты самих параметров. Эти команды предназначены для разработчиков компиляторов.

BOUND

Команда BOUND сравнивает знаковый индекс массива из регистра с двумя ячейками памяти, содержащими нижнюю и верхнюю границы индекса. Длина чисел в 80386 составляет 16 или 32 бита.

НЕПОСРЕДСТВЕННЫЕ ОПЕРАНДЫ ДЛЯ КОМАНД

Непосредственное значение можно включить в стек, использовать в умножении и в качестве счетчика для циклических сдвигов. Это сокращает общее число команд.

6.4.4. Новые команды для реального режима 80386

Для реального (и защищенного) режима 80386 введено много новых команд. Кроме того, некоторые команды модифицированы из-за наличия 32-битных операндов. Некоторые новые команды связаны с изменениями архитектуры.

LSS, LFS, LGS

Новые команды загрузки указателя предназначены для новых регистров SS, FS и GS. Как и в других командах загрузки из указанной ячейки памяти производится загрузка сегментного регистра или другого указанного регистра. Команды LFS и LGS предназначены для новых регистров, а команда LSS упрощает создание нескольких сегментов стека.

MOV

Команду MOV можно использовать для пересылки данных в/из регистров отладки, управления и проверки. Она предназначена для новых регистров.

MOVSX, MOVZX

При наличии регистров длиной в два слова возникает проблема, как заполнять дополнительные разряды при пересылке байта или слова в двойное слово. Команда MOVZX заполняет их нулями, а команда MOVSX — знаковыми (старшими) битами источника. Команда MOVZX сохраняет значение беззнакового числа, а команда MOVSX — знакового. Эти операции работают и при пересылке байта в слово.

ДРУГИЕ КОМАНДЫ

Некоторые новые команды отсутствовали в прежних процессорах семейства 8086. Программа с этими командами становится более гибкой, но не совместима с прежними процессорами.

1. Условные переходы с большим диапазоном.
   Команды вида JS имели раньше диапазон от -128 до +127 байт. В 80386 сохранены эти команды и добавлены условные «близкие переходы» (переходы в пределах текущего сегмента).

2. Команды манипуляций битами.
   Новыми командами являются BT, BTC, BTR и BTS. Они проверяют бит, передавая его во флажок переноса. Команды различаются тем, что делается с проверяемым битом: он не изменяется, инвертируется, сбрасывается в 0 или устанавливается в 1.

3. Команды сканирования бита.
   Новые команды BSF и BSR находят младший (BSF) или старший (BSR) единичный бит в слове или двойном слове из регистра.

4. Двойные сдвиги.
   Команды SHLD и SHRD реализуют сдвиги двойной точности. Они позволяют сдвинуть 16- или 32-битный операнд в любую сторону до 31 бита, как в обычном сдвиге; освобождающиеся биты заполняются битами из операнда-регистра. Однако второй операнд не изменяется.

5. Установка байта по условию.
   Эти команды похожи на команды условного перехода; большинство условий, вызывающих переход, могут установить в 1 байт (если условие удовлетворяется) или сбросить в 0 байт (если условие не удовлетворяется). Такая возможность удобна для определения нескольких условий перехода.

6.4.5. Другие различия между программами 8086 и защищенного режима 80386

Рассматриваемые далее различия касаются совместимости программ 8086 и реального режима 80386. Они относятся к программам 8086 и тем же программам, вы-

полняющимся как задачи V8086. Различия объединены в несколько групп.

ГРУППА 1: УСКОРЕНИЕ КОМАНД

Повышение производительности процессора 80386 объясняется увеличением частоты синхронизации более чем в три раза и уменьшением числа тактов. Более высокая частота синхронизации может вызвать проблемы для всех программ, полагающих, что команды выполняются конкретное время. Меньшее число тактов означает, что задержки для некоторых устройств ввода/вывода становятся меньше.

ГРУППА 2: ЗАВОРАЧИВАНИЕ

Если программа для 8086 определяет адрес больше 1 Мбайта, адрес усекается, т. е. он относится к первым 64 Кбайтам адресного пространства. В процессоре 80386 используется полученное значение, как есть.

Процессор 8086 позволяет обращаться к байтам со смещением больше 65535 (код или данные) с заворачиванием к байту 0 того же сегмента. Обращения со смещениями ниже 0 (только данные) вызывают заворачивание к байту 65535 того же сегмента. Процессор 80386 в обоих ситуациях генерирует особый случай прерывания (12 для стековых данных и 13 для остальных).

Сдвигами и циклическими сдвигами управляют только 5 младших бит счетчика сдвигов. Максимальное значение сдвига равно 31; значения больше 31 вызывают сдвиг на число разрядов равное остатку от деления счетчика на 32.

ГРУППА 3: ПРЕФИКСЫ

В процессоре 8086 можно использовать избыточные префиксы, префикс LOCK по желанию, а префиксы перед командой ESC игнорируются, когда выполнение возобновляется после обработки прерывания. В 80386 длина команд не может быть больше 15 байт, что вызывает проблемы только при использовании избыточных префиксов. Префикс LOCK допускается только перед некоторыми командами, когда они модифицируют память:

проверки бит, сложения и вычитания, инкременты и декременты, команда XCHG с одним операндом в памяти. Наконец, префиксы перед командой ESC выполняются, а не игнорируются.

ГРУППА 4: ОСОБЫЕ СЛУЧАИ ПРЕРЫВАНИЯ

В процессоре 8086 особый случай, возникающий при делении, возвращает значение регистров, адресующих команду, следующую после DIV, а в 80386 они адресуют саму команду DIV. В 8086 команда IDIV вызывает особый случай прерывания, если частное равно 80H (байты) или 8000H (слова); процессор 80386 возвращает правильное частное.

Особый случай прерывания в пошаговом режиме 80386 имеет более высокий приоритет, чем внешнее прерывание, поэтому выполнение программы в пошаговом режиме предотвращает пошаговое выполнение внешнего прерывания. Когда воспринимается немаскируемое прерывание NMI, дальнейшие NMI маскируются до выполнения команды IRET в процедуре обработки первого NMI.

Неопределенный код операции для 80386 вызывает прерывание 6.

ГРУППА 5: ЧИСЛЕННЫЕ СОПРОЦЕССОРЫ

Повышенная скорость 80386 может вызвать проблемы при работе с сопроцессором. Когда несколько исполняемых 8086 команд дают время, достаточное для операции сопроцессора, в 80386 они выполняются быстрее и сопроцессору может не хватить времени.

В 80386 все ошибки сопроцессора направляются на прерывание 16. Для прерываний сопроцессора контроллер прерываний больше не применяется.

ГРУППА 6: ВКЛЮЧАЕМЫЕ В СТЕК ЗНАЧЕНИЯ

При включении в стек SP теперь включается его значение до декремента самой командой PUSH; ранее в стек включалось значение SP после текущей команды PUSH. Кроме того, команда PUSHF для 8086 всегда имела 1 в разрядах 12—15. В 80386 бит 15 всегда 0, а биты 12, 13 и 14 содержат последние загруженные в

них значения; биты 12 и 13 — это поле IOPL, а бит 14 — флажок вложенной задачи.

6.5. ОСОБЕННОСТИ ОПЕРАЦИОННОЙ СИСТЕМЫ

Существует много типов OS. Процессор 80386 с поддержкой модели плоской памяти и другими возможностями повлечет использование многих OS миникомпьютеров. С нашей точки зрения важнейшее различие между OS касается степени использования потенциальных возможностей 80386. Здесь возможно несколько подходов.

Простейший гипервизор просто запускает одну задачу V8086, разрешая одному пользователю реализовать одну копию OS 8086. При этом поддерживается одна программа. Расширяя эту первую модель, можно использовать страничную организацию для эмуляции таких возможностей, как память более 640 Кбайт. Допускается использовать виртуальную память, поэтому программа может работать с памятью больше физической путем свопинга страниц с жесткого диска.

Следующий шаг — разрешить выполнять несколько задач V8086 одновременно. Как только вводится мультизадачность, гипервизор должен перехватить все команды, которые изменяют флажок IF; в противном случае программы могут «зависнуть». В этом случае увеличивается обьем служебных операций. OS может быть однопользовательской мультизадачной системой, многопользовательской системой с одной задачей у каждого пользователя и даже многопользовательской мультизадачной системой. Такая система либо потребует, чтобы компьютер имел достаточную физическую память для каждого пользователя, либо организует виртуальную память с привлечением жесткого диска. Однако отдельные задачи остаются задачами V8086 и не используют всех возможностей 80386.

Наконец, последний гипервизор (многопользовательская мультизадачная система с виртуальной памятью) можно расширить, допустив одновременное выполнение с задачами V8086 программ защищенного режима 80386. Такой гипервизор должен включать полную OS для обработки системных вызовов программ защищенного ре-

жима. Он должен обеспечить совместимость с DOS, чтобы все задачи V8086 обращались к одному коду OS в гипервизоре, не требуя иметь отдельную копию OS в каждой задаче V8086.

Такое вводное обсуждение касается основных моментов с точки зрения максимального использования 80386. Балансирование между увеличенными возможностями и необходимостью использовать существующие программы характерно для каждого владельца системы на базе 80386 и помогает выбрать нужную OS.

6.5.1. Примеры гипервизоров

Первый гипервизор фирмы Microsoft для Compaq 80386 делает очень немного. Он запускает одну задачу V8086, а затем передает управление ближайшей копии DOS 2.x или другой версии. Пользователь запускает копию DOS, а затем выполняет программу как на IBM PC. Страницами защищаются только адреса расширенной памяти сверх 640 Кбайт. При обращении по одному из этих адресов возникает нарушение защиты и переключение задачи на гипервизор. Он осуществляет обращение к памяти, а затем возвращает управление программе. В общем задача V8086 должна выполняться быстрее, так как используется полное быстродействие 80386 с замедленным обращением к расширенной памяти.

Более мощная OS должна допускать одновременное выполнение нескольких задач V8086. Пользователь работает с Lotus 1-2-3 в интерактивном режиме, а Турбо Паскаль компилирует программу в фоновом режиме. Пользователь задает двум программам процент используемого времени процессора. Процессор выполняет одну программу несколько миллисекунд, а затем переключается на другую. Если для обоих программ недостаточно RAM, то 80386 и OS будут применять виртуальную память. В любой момент времени содержимое памяти — это страницы кода и данных, которые использовались последними.

Существуют хитроумные программы, вызывающие некоторые проблемы. Например, игра может работать только с конкретной версией DOS. Пользователь может запустить эту версию DOS и игру как новую задачу отдельно от других программ и переключаться между ни-

ми по желанию. В памяти находятся только последние части старой версии DOS и игры, а остальные хранятся на диске.

Операционные системы такого типа очень трудно написать. Сложно реализуются «стратегии замены страниц» и затруднена отладка OS. Однако созданная и испытанная OS многократно увеличивает возможности компьютера.

Глава 7

ВНУТРЕННЯЯ ОРГАНИЗАЦИЯ ПРОЦЕССОРА 80386

Процессор 80386 намного сложнее и мощнее своих предшественников. Программист, знающий его работу на аппаратном уровне, будет работать с большим удовольствием. Данная глава начинается с того, как процессор взаимодействует с памятью и сопроцессорами. Затем рассматриваются его функциональные блоки. Наконец, поясняется вся последовательность выполнения команд.

Материал данной главы не обязателен для работы с 80386, но очень полезен. Когда вы познакомитесь с процессором и захотите «выжать» из него максимум возможностей, знание его организации поможет в этом. С компьютером приятнее работать, понимая, как он себя ведет.

7.1. ПРОИЗВОДИТЕЛЬНОСТЬ КОМПЬЮТЕРА

Содержание следующих разделов поможет в выборе конфигурации компьютера на базе 80386. Показаны способы организации компьютера для достижения оптимальной производительности и некоторые ограничения современной технологии.

Компьютер состоит из нескольких частей, каждая из которых взаимодействует с другими. Компьютер с гибким диском может выполнять миллионы операций

в секунду и ожидать 0.5 с при считывании данных с диска.

Такое же взаимодействие происходит между процессором и памятью, когда процессор должен ожидать окончания операции записи или считывания. Именно поэтому вводятся внутренние регистры для сохранения промежуточных результатов, чтобы часто не обращаться к памяти. Однако в циклических операциях процессор должен часто обращаться к памяти. Для такого быстрого процессора, как 80386, ускорение взаимодействия с памятью особенно важно; преимущество наличия быстрого процессора уменьшается, если каждое обращение к памяти вызывает долгое ожидание.

При обсуждении быстродействия 80386 очень важным понятием является такт синхронизации. Генератор синхронизации подает периодический сигнал на вход CLK2. Процессор 80386 преобразует два такта генератора в один внутренний такт. При частоте синхронизации 32 МГц такт длится 62.5 нс.

Некоторые внутренние функции 80386 реализуются за половину такта. Даже самая короткая команда (например, пересылка из одного регистра в другой) длится два такта.

На рис. 7.1 показаны сигналы CLK2 и такт процессора.

7.2. ОБРАЩЕНИЕ К ПАМЯТИ

Большинство программистов говорят об «обращении к памяти» и «трехтактном доступе к RAM», не зная об аппаратных деталях. Чтобы лучше разобраться в возможностях 80386, нужно глубже понимать то, что происходит в компьютере.

Когда процессору требуется прочитать байт из памяти, он сначала вычисляет адрес этого байта. После этого он выдает адрес на группу контактов, образующих шину адреса. Во многих компьютерах одни и те же контакты применяются для адреса и для данных с временным мультиплексированием (в 8086 и 80286), но в 80386 есть по 32 независимых контакта для адреса и данных. Код на шине адреса показывает микросхемам памяти, какой байт нужен процессору. Микросхемы RAM считывают байт и помещают его на шину данных.

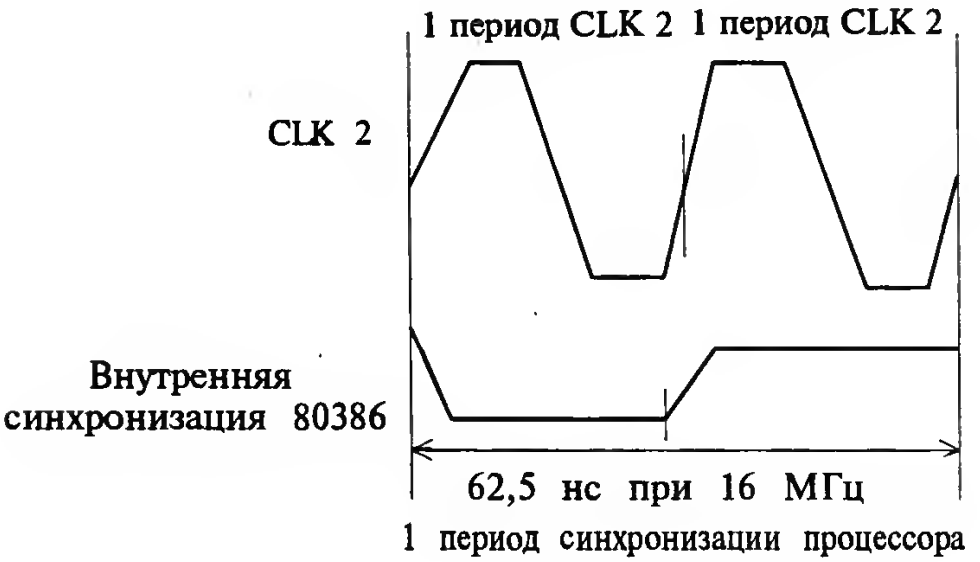


Рис. 7.1 Сигналы синхронизации

Микросхемы динамических RAM (DRAM), которые применяются в большинстве компьютеров, теряют содержимое после выдачи его на шину данных и должны регенерироваться. Статические RAM (SRAM) быстрее DRAM и не требуют регенерации, но они дороже и имеют меньшую емкость. Шина данных передает байт через мультиплексоры, приемопередатчики и приемники и подает его на входные контакты процессора. Процессор вводит байт на свою внутреннюю шину данных. Если микросхемы RAM реагируют быстро на запрос данных, то весь процесс (выдача адреса, помещение данных на шину и возврат данных в процессор) занимает два такта. Следовательно, система на базе 80386 с синхронизацией 16 МГц требует для считывания из памяти байта, слова или двойного слова 125 нс.

Отметим условие «если микросхемы RAM реагируют быстро». Когда доступ к памяти требует лишний такт, так как микросхемы RAM не реагируют достаточно быстро, этот такт называется «состоянием ожидания». Медленный процессор может использовать медленные микросхемы без введения состояний ожидания. При повышении скорости процессора одна и та же микросхема па-

мяти может потребовать введения одного или даже двух тактов ожидания. В процессоре 80386 реализовано несколько приемов, позволяющих применять относительно медленные микросхемы памяти без введения состояний ожидания.

Один из таких приемов называется конвейеризацией адреса. Обычно термин «конвейеризация» означает параллельную работу устройств опережающей выборки, преддешифрирования, выполнения и управления памятью. В 80386 конвейер применяется и при обращении к памяти. Пусть в системе применяются микросхемы, требующие введения одного такта ожидания (62.5 нс при 16 МГц). Тогда три последовательные операции считывания выглядят так:

Такт 1: НАЧАЛО. Выдать адрес на шину адреса.
Такт 2: Ожидать реакции памяти (состояние ожидания).
Такт 3: Данные возвращаются по шине данных. КОНЕЦ.
Такт 4: НАЧАЛО. Выдать адрес на шину адреса.
Такт 5: Ожидать реакции памяти (состояние ожидания).
Такт 6: Данные возвращаются по шине данных. КОНЕЦ.
Такт 7: НАЧАЛО. Выдать адрес на шину адреса.
Такт 8: Ожидать реакции памяти (состояние ожидания).
Такт 9: Данные возвращаются по шине данных. КОНЕЦ.

Однако 80386 может выдать адрес для одного считывания при возвращении данных от предыдущего считывания, поэтому последовательность оказывается короче:

Такт 1: НАЧАЛО. Выдать адрес на шину адреса.
Такт 2: Ожидать реакции памяти (состояние ожидания).
Такт 3: Данные возвращаются. КОНЕЦ и НАЧАЛО. Выдать следующий адрес.
Такт 4: Ожидать реакции памяти (состояние ожидания).
Такт 5: Данные возвращаются. КОНЕЦ и НАЧАЛО. Выдать следующий адрес.

Такт 6: Ожидать реакции памяти (состояние ожидания).

Такт 7: Данные возвращаются. КОНЕЦ и НАЧАЛО. Выдать следующий адрес.

После первого считывания (3 такта) последующие операции занимают только 2 такта. Такая же ситуация возникает в операции записи: трехтактная запись становится двухтактной. В обоих случаях первое обращение к памяти после холостого цикла шины требует лишний такт; аналогично первые несколько команд после JMP или CALL (которые нарушают внутренний конвейер 80386) требуют больше тактов.

Есть и другие способы избежать состояний ожидания. Например, память, которая вводит состояние ожидания на 16 МГц, может не вводить его на 12.5 МГц, но тогда замедляется вся система. Конвейеризация адреса необязательна, так как она требует точной синхронизации между процессором и остальной системой. При использовании 16-битных, а не 32-битных микросхем памяти, конвейеризация адреса невозможна.

7.2.1. Расслоение памяти

Каждый запрос считывания или записи вызывает в DRAM определенную последовательность событий: включение, ожидание запроса, ответ на запрос и регенерация. Процессор должен ожидать только ответа на запрос до перехода к дальнейшим действиям, которые могут включать немедленное обращение к другой микросхеме (пока первая регенерируется). При обращении подряд к микросхемам в одном и том же банке происходит следующее: получение запроса, ответ на запрос (содержимое пропадает), получение нового запроса, включение (процессор ожидает), ответ на запрос и регенерация. Дополнительное время на регенерацию банка памяти может вызвать введение состояния ожидания при любых обращениях к памяти, которые пытаются использовать один и тот же банк дважды подряд.

Простейшая форма организации памяти состоит в том, чтобы поместить все микросхемы в один банк (для наших целей банк — это любой блок памяти, который может регенерироваться при обращении к другому бан-

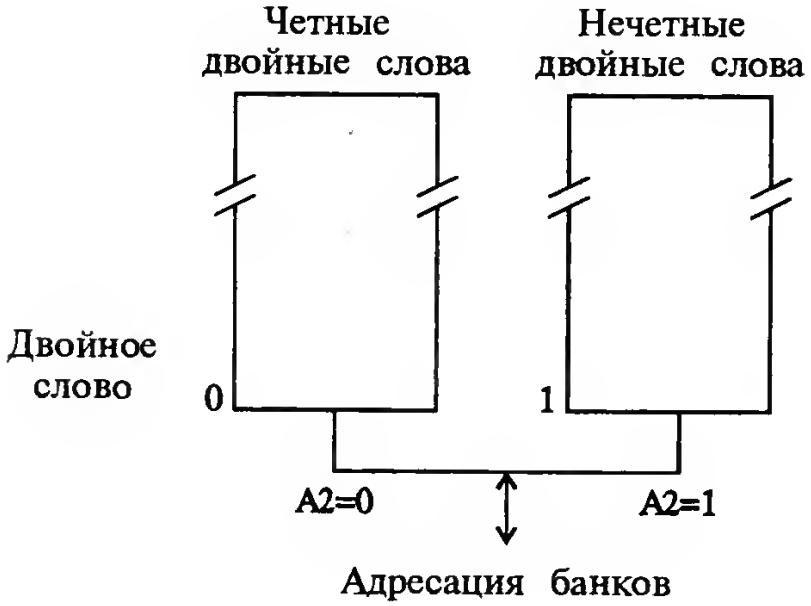


Рис. 7.2 RAM с расслоением (2 банка)

ку). Разделение памяти на несколько банков может сократить время ожидания при последовательных обращениях к одному банку.

Процессор обычно обращается к памяти последовательно: сначала данный адрес, затем следующий, вновь следующий и т. д. Такие последовательные адреса должны быть в разных банках памяти. Этот прием реализован в памяти из двух банков с расслоениями: каждое последовательное двойное слово находится в разных банках. Для этого младший бит адреса служит селектором банка.

Последовательные обращения никогда не вводят дополнительных тактов ожидания. Произвольные обращения имеют 50% шансов введения задержек; организация большего числа банков уменьшает задержки при произвольных обращениях, не изменяя ситуацию для более частых последовательных обращений. Дополнительные расходы на увеличение числа банков компенсируются конвейером 80386 и применением кэш-памяти (см. далее), которые допускают введение дополнительных тактов ожидания без задержек в обработке.

RAM с расслоением показана на рис. 7.2.

7.2.2. Кэширование памяти

Кэширование — это применение небольшой быстрой памяти для хранения часто используемой или ожидаемой информации. При виртуальной организации памяти RAM можно использовать как кэш для жесткого диска, а в самом процессоре очереди опережающей выборки и дешифрирования действуют как кэш информации, которую операционное устройство должно обычно запрашивать из памяти. Термин «кэширование» часто относится к памяти и подразумевает введение быстрых микросхем памяти (обычно SRAM с нулевым временем ожидания) для хранения подмножества информации, находящейся в большей и медленной основной памяти. Блок-схема кэш-памяти показана на рис. 7.3.

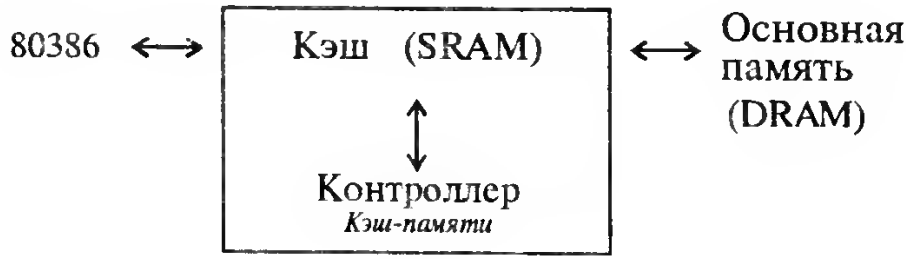


Рис. 7.3 Подсистема Кэш-памяти

Процессор 80386 при 16 МГц может считывать данные за 125 нс (2 такта), если память реагирует достаточно быстро. В конвейерных обращениях время доступа может быть увеличено до 187.5 нс (3 такта). В таблице 7.1 показаны допустимые времена реакции для различных микросхем RAM.

Сейчас большинство SRAM реагируют за 125 нс в неконвейерном обращении без состояния ожидания. Быстрые, но дорогие DRAM реагируют в окне от 125 до 187.5 нс с одним тактом ожидания в неконвейерном обращении и без ожидания при конвейеризации. Дешевые DRAM реагируют за 187.5—200 нс, вводя 2 такта ожидания без конвейеризации и один с конвейеризацией. Системы без кэш-памяти на дорогих DRAM имеют хоро-

шую производительность. Однако для огромной памяти 80386 требуются дешевые микросхемы, поэтому добавление кэш-памяти на SRAM заметно повышает производительность системы.

Таблица 7.1. Время реакции микросхем RAM

Тип обращения	Окно при 16 МГц	Тип памяти	Число тактов ожидания
Неконвейерное считывание	125 нс	SRAM	0
		Быстрая DRAM	1
		Медлен. DRAM	2
Конвейерное считывание	187.5 нс	SRAM	0
		Быстрая DRAM	0
		Медлен. DRAM	1

КЭШ-ПАМЯТЬ

При использовании кэш-памяти процедура обмена данными усложняется. В системе без кэш (и без виртуальной памяти) процессор запрашивает информацию, и основная память обеспечивает ее. В кэшированной системе процессор запрашивает информацию и контроллер кэш проверяет, нет ли информации в кэш, и если она есть, он выдает эту информацию. В противном случае запрос передается в основную память. В таблице 7.2 показаны временные соотношения для обоих ситуаций (Tw — такт ожидания).

Из таблицы видно, что 50% попаданий в кэш означают, что половина обращений не будет вызывать состояние ожидания, но при промахе время обращения увеличивается (4 такта ожидания); в результате получается эквивалент некэшированной памяти с двумя тактами ожидания. К счастью, даже небольшая кэш-память объемом 64 Кбайта обеспечивает коэффициент попадания 90%, что оправдывает расходы на кэш-память. В таблице 7.3 показаны проценты попадания для кэш-памяти с прямым отображением и строкой в 4 байта; другие способы отображения и размеры строки улучшают коэффициент попадания на несколько процентов. Имеющиеся

микросхемы SRAM позволяют реализовать дешевую кэш-память 64 Кбайта.

Таблица 7.2. Сравнение обращений

Действие	Такты
Некэшированное считывание:	
выдача адреса в память	1
ожидание RAM	2
данные возвращаются	1
Всего	4 (два Tw)
Кэшированное считывание (попадание):	
выдача адреса в кэш-память	1
проверка кэш, возвращение данных	1
Всего	2 (нет Tw)
Кэшированное считывание (промах):	
выдача адреса в кэш-память	1
проверка кэш, нет соответствия	1
выдача адреса в память	1
ожидание RAM	2
данные возвращаются	1
Всего	6 (четыре Tw)

Таблица 7.3. Размер и эффективность кэш-памяти

Размер кэш-памяти	Попадания	Выигрыш
Нет кэш, DRAM с двумя Tw	0%	0%
16K	81%	35%
32K	86%	38%
64K	88%	39%
128K	89%	39%
Нет кэш, SRAM без Tw	100%	47%

Фактический размер RAM мало влияет на производительность кэш-памяти. Большинство программ повторно обращаются к тем же нескольким килобайтам кода и данных. Если, например, вы печатаете письмо в текстовом процессоре, то секции программы управления печатью и оперативной подсказки (Help) не используются несколько минут. Когда вы печатаете, вносите изменения и вновь печатаете, вы вновь и вновь обращаетесь к одним и тем же данным. Однако некоторые программы с точки зрения кэш-памяти ведут себя плохо. Например, при повторных вычислениях больших электронных таблиц кэш действует не эффективно.

Следующий раздел более подробно освещает работу кэш-памяти. Кэш может улучшить производительность системы, но производительность компьютера ограничена не только памятью, но и параметрами устройств ввода/вывода (особенно в части дисков и регенерации дисплея), поэтому при разработке или приобретении компьютера нужно учитывать множество факторов, помимо организации кэширования.

ОБРАЩЕНИЯ К КЭШ-ПАМЯТИ

При разработке кэш-памяти возникает много вопросов, часть из которых связана с производительностью. Некоторые из этих вопросов рассматриваются далее в предположении объема оперативной памяти до 16 Мбайт и кэш-памяти 64 Кбайта с прямым отображением.

Сама кэш-память имеет два уровня. Первый образован блоком тэговых SRAM, содержащим адресную информацию. В данной кэш-памяти с прямым отображением 256 адресов отображаются на одну ячейку кэш. Кэш содержит копию информации в одной из этих 256 ячеек. Отметим, что произведение кэш 64 Кбайт на 256 адресов в элементе кэш позволяет адресовать 16 Мбайт. Второй уровень содержит SRAM данных, хранящих фактические данные, которые скопированы из DRAM.

32-битный адрес формируемый 80386 разделяется на три части. Первые 8 бит должны быть нулевыми, так как кэш покрывает память 16 Мбайт. При наличии ненулевого значения должен генерироваться особый случай прерывания.

Следующие 8 бит применяются как поле «тэга», а последние 16 бит служат «индексом», показывающим, по какому адресу происходит обращение.

Тэговые SRAM содержат список всех 4-байтных «блоков», которые находятся в самой кэш-памяти. «Индекс» сообщает, где в списке находится нужная информация. Поле тэга сообщает, какая из 256 возможных ячеек памяти для данной ячейки кэш фактически находится в кэш. Если информация в кэш содержится, то имеет место попадание и данные посылаются в процессор. В противном случае имеет место промах, и запрос направляется в оперативную память.

Кэш на рис. 7.4 имеет ширину 4 байта, т. е. каждая ячейка содержит двойное слово. Можно использовать и другую ширину (8 или даже 16 байт), при этом замедляется модификация кэш, но увеличивается коэффициент попаданий. 8-байтная кэш имеет коэффициент попаданий на несколько процентов выше, чем 4-байтная.

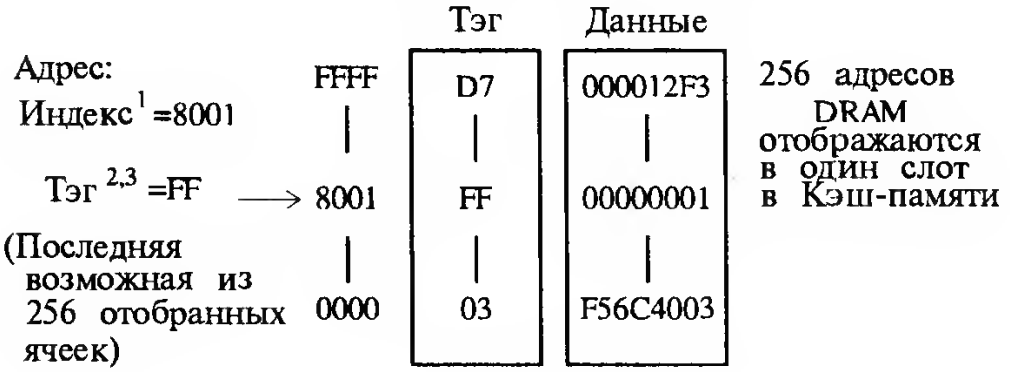


Рис. 7.4 Обращение к Кэш-памяти

На рис. 7.4 имеются примечания:

1. 16-битный индекс сообщает, какую ячейку из 64K ячеек в тэговом SRAM следует проверять.
2. 8-битный тэг сообщает, какое из 256 возможных двойных слов находится в SRAM данных.
3. Если запрошенный тэг совпадает с тэгом в SRAM, возникло попадание. Если нет, происходит обращение к оперативной памяти и производится замена тэга и данных на данные, полученные из DRAM.

Каждая ячейка основной памяти отображается точно на одну ячейку кэш. Если программа повторно обращается к двум ячейкам по очереди и обе они отображаются на одну и ту же ячейку кэш, возникает «пробуксовка кэш»: проверка в кэш элемента данных 1, не найден, обращение к DRAM, модификация кэш; проверка в кэш элемента данных 2, не найден (там элемент 1), обращение к DRAM, модификация кэш; проверка в кэш элемента данных 1, не найден (там элемент 2), обращение к DRAM, модификация кэш и т. д. Этой проблемы можно избежать, разрешив любой ячейке в основной памяти направляться по мере необходимости в одну из двух ячеек кэш-памяти. Теперь контроллер кэш должен проверять два места при каждом обращении к кэш, а при помещении нового элемента в кэш решать, какую из двух возможных ячеек использовать. Такая двухсторонняя ассоциативность улучшает попадания на 1% (при более дорогой схеме управления).

Возможно реализовать четырехстороннюю ассоциативность и даже полностью ассоциативную кэш-память (каждая ячейка памяти отображается на любую ячейку кэш). Расходы и сложность при этом растут, а увеличение коэффициента попадания невелико.

7.2.3. Страничные RAM

При считывании из памяти двойного слова производится обращение к 32 микросхемам. Адрес двойного слова разделяется на номера строки и столбца, сообщающие положение нужного бита. Если номер строки в обращениях не изменяется, то нужно модифицировать только номер столбца.

Простая схема сравнения на старших линиях шины адреса может распознать, когда адрес относится к одной и той же строке (изменился только номер столбца). Такое обращение можно осуществить быстрее, чем обычно, и RAM не будет увеличивать время ожидания. Такой способ называется страничной организацией RAM. Его действие аналогично наличию кэш-памяти в 512 байт, но нет потерь времени из-за промахов, так как ускоряются все обращения в группе из 512 байт.

7.2.4. Организация памяти

Одна из больших проблем для систем на базе 80386 — обеспечение совместимости с существующими 16-битными компьютерами и их программным обеспечением. Многие детали программирования 80386 связаны с отличиями между 16- и 32-битными операндами и адресацией.

Как же эти конфликты разрешить аппаратно? Процессор 80386 можно подключить к 16- или 32-битной шине данных. Мы рассмотрим обе ситуации; сначала рассматривается более простая 32-битная шина данных, а затем особенности (скрытые от программиста) подключения 16-битной шины к 32-битному компьютеру.

Для программиста память выглядит как последовательность байт; слово — это два соседних байта, а двойное слово — 4 соседних байта.

Фактически же память разделяется на 4 секции по байту шириной, поэтому вместо длинного склада из байт получаются 4 более коротких склада из байт, причем склад имеет свое подключение к шине данных.

Четыре склада совместно можно считать одним складом из двойных слов. На рис. 7.4 показана такая организация памяти и подключение 8-битных секций к шине данных. Когда процессор 80386 запрашивает из памяти двойное слово, из каждого склада берется байт, помещается на свою секцию шины данных и посылается в процессор. Так за одно обращение к памяти процессор 80386 получает двойное слово.

Такое разделение памяти отражается на подключении процессора. Мы говорим, что процессор имеет 32-битную шину адреса, но корпус имеет всего 30 контактов адреса A2—A31. Где же контакты A0 и A1?

Фактически шина адреса показывает, какое двойное слово выбрать из памяти, а не какой байт. Контакты шины адреса позволяют выбирать из 2^{30} двойных слов. Мы знаем, что 80386 позволяет выбрать из памяти каждый байт, а также любое двойное слово. Как при выборе двойного слова выбираются отдельные байты внутри него? Можно передать в процессор все двойное слово, а затем внутри выбрать нужные байты; однако выбор байт производится уже на шине. Процессор имеет 4 контакта разрешения байта BE0#—BE3#. Если контакты адреса сообщают о выборе двойного слова, то контакты разре-

шения байта сообщают о выборе нужных байт в двойном слове.

Два младших бита адреса определяют активные сигналы на контактах BE.

Слово по четному адресу называется «выравненным на границе слов»; двойное слово по адресу кратному 4 называется «выравненным на границе двойных слов». Программисты знают, что при задании структур данных следует выравнивать слова по границам слов, а двойные слова — по границам двойных слов. Рассмотрение взаимодействия линий адреса и сигналов BE показывает, когда и почему это важно. Если выбирается невыравненное двойное слово, получить его за одно обращение невозможно. Пусть, например, двойное слово находится по адресу байта 2; тогда половина его находится в первом двойном слове памяти, а вторая половина — во втором; поэтому процессор автоматически делает два обращения.

Первое обращение дает два старших байта по адресам байт 4 и 5. Для этого адресные контакты выбирают второе двойное слово, а активные сигналы BE0# и BE1# выбирают первый и второй байты. Второе обращение дает два младших байта по адресам байт 2 и 3 (адрес выбирает первое двойное слово, а BE2# и BE3# выбирают третий и четвертый байты этого двойного слова). Байты автоматически упорядочиваются на шине данных, но получение двойного слова требует двух циклов обращения к памяти. Если двойное слово выравнено на границе двойного слова, контакты адреса выбирают двойное слово, а все 4 контакта BE выбирают сразу все 4 байта.

Понимание взаимодействия 80386 с 32-битной шиной помогает разобраться с особенностями использования 16-битной шины. На входной контакт BS16# (размер шины 16) устройство выдает сигнал, сообщающий процессору об обращении к 16-битному устройству. Если вся шина данных имеет 16 бит, то сигнал BS16# формируется при каждом обращении.

При обращении к 16-битной памяти процессор использует только младшие 16 бит шины данных и это имеет предсказуемые последствия; слово, выравненное по нечетной границе, требует двух обращений для передачи в процессор. Двойное слово передается за два обращения, если оно выравнено на границе слова или двойного слова, а двойное слово на нечетной границе требует

трех обращений (первое дает один байт, второе — два средних байта и третье — оставшийся байт).

В 16-битных обращениях конвейеризация адреса отсутствует, поэтому дополнительные такты на реагирование памяти не выделяются. 16-битная шина данных с микросхемами, имеющими два такта ожидания, может вызвать значительные задержки в операциях с частыми обращениями к памяти, и правильное выравнивание структур данных приобретает еще большее значение. Опережающая выборка команд (см. далее) также ухудшается. Более медленная или узкая шина оказывается более занятой операционным устройством. Устройство опережающей выборки имеет меньше тактов для выборки кода, что увеличивает вероятность того, что очереди опережающей выборки окажутся пустыми, а это замедляет выполнение команд.

Рассмотренные особенности приводят к формулировке простых правил для программиста. Все слова необходимо выравнивать по границам слов (обязательно для 16-битных шин), а двойные слова — по границам двойных слов (обязательно для 32-битных шин). Наконец, программа, которая хорошо работает на одном компьютере, может работать хуже на другом в зависимости от быстродействия микросхем и размера шины.

КОМПРОМИССЫ РАЗРАБОТКИ

Улучшение эффективности памяти в части стоимости и сложности сводятся к двум основным приемам: кэширование и все остальное. Если использовать все способы второй группы (расслоение, страничные RAM, RAM с одним тактом ожидания, 32-битная шина данных и конвейеризация адреса), то производительность получается наилучшей, приближаясь к SRAM. Если же ни один из этих способов не используется, то производительность медленного компьютера можно улучшить, вводя кэш-память.

В компьютере Compaq 386 кэширование не применяется; в нем есть страничные RAM, эквивалентные кэш-памяти 512 байт, и RAM с одним тактом ожидания при 16 МГц. В нем же имеется специальная 32-битная шина для быстрого доступа к памяти; остальная система (средства ввода/вывода и дополнительная память) базируется на 16-битной шине.

Несмотря на отсутствие кэширования, производительность памяти очень хорошая.

Другие системы все более опираются на Турбо платы 386. Эти небольшие платы (3×5 дюймов) с процессором 80386 и вспомогательными микросхемами рассчитаны на подключение в IBM PC/AT. Схемы на платах отображают контакты 80386 на разъем, который имитирует контакты 80286; вам остается удалить 80286 из AT, вставить Турбо плату, и вы получаете систему на базе 80386.

Изготовители Турбо плат не могут влиять на выбор RAM и способы организации памяти главного компьютера, поэтому лучше всего применить SRAM-кэш, которая помогает процессору избежать обращений к обычному RAM при 90% считываний из памяти. Компьютеры IBM PC/AT с такими платами и хорошим жестким диском приближаются по производительности к Compaq 386, но стоят гораздо дешевле.

В других системах применяются иные разнообразные способы повышения производительности при минимальных расходах на модернизацию.

Микросхемы с одним тактом ожидания при 16 МГц могут иметь два такта ожидания при 20 или 25 МГц. Фактически компьютер без кэш-памяти может иметь меньшую производительность при введении более быстрого процессора из-за увеличения числа тактов ожидания. Это зависит от множества временных факторов, которые в каждой системе свои.

Добавление сопроцессора также ускоряет выполнение программ. Процессор 80386 может использоваться совместно с 80287 или 80387; последний процессор примерно в три раза производительнее.

ИНТЕРФЕЙС ВВОДА-ВЫВОДА

Термины «ввод/вывод с отображением на память» и «ввод/вывод с отображением на ввод/вывод» знакомы всем, но что они означают? С аппаратной точки зрения они различаются несильно. При обращении к памяти или внешним устройствам используются те же схемы управления, шина данных и шина адреса. Конечно, устройство ввода/вывода отличается от памяти, и взаимодействие с ним оказывается более гибким.

Процессор 80386 фактически имеет два реальных

адресных пространства (не виртуальный режим). Адресное пространство памяти содержит 2^{32} байт. Оно может включать адреса памяти и устройств ввода/вывода. Достоинства такого приема:

1. Размер адресов 32 бита.

2. Можно использовать любую команду, которая адресует память, например, AND, OR, TEST. Для доступа к устройствам можно использовать все режимы адресации.

3. Пространство ввода/вывода можно защищать с помощью сегментов или страниц.

Второй способ заключается в размещении устройства в пространстве ввода/вывода. Это пространство объемом 64 Кбайт недоступно обращениям большинства команд, включая команды пересылки данных. Разделение пространств обеспечивает контакт M/IO, разрешающий шине адреса использовать память или пространство ввода/вывода. Для доступа к пространству ввода/вывода применяются только команды IN, OUT, INS и OUTS.

Достоинства такого способа:

1. Ширина адресов 16 бит, что упрощает их дешифрирование.

2. Пространство ввода/вывода недоступно большинству команд, что служит определенной защитой.

Два способа можно комбинировать, отображая пространство ввода/вывода на блок 64 Кбайт адресного пространства памяти. В этом случае в системе будут выполняться программы, рассчитанные на любой способ. Достоинства адресации обоих способов здесь сохраняются, а достоинства защиты теряются, например, сегментная защита не работает против команды OUTS.

7.3. ВНУТРЕННЕЕ УСТРОЙСТВО 80386

Обратимся теперь к устройству самого процессора.

Рассматриваемый далее материал не зависит от цели использования процессора. Подробное знание процессора поможет в разработке эффективных ассемблерных программ.

7.3.1. Конвейеризация команд

Самый важный способ ускорения обращения к памяти (конвейеризация) встроен в процессор 80386. Обычный процессор выполняет команду в таком порядке:

1. Выборка команды из памяти.

2. Дешифрирование команды в стандартную форму для операционного устройства.

3. Выполнение команды.

Одновременно можно делать только один шаг, поэтому обработка развивается примерно так:

Интервал 1: выборка команды 1.

Интервал 2: дешифрирование команды 1.

Интервал 3: исполнение команды 1.

Интервал 4: выборка команды 2.

Интервал 5: дешифрирование команды 2.

Интервал 6: исполнение команды 2.

Конвейеризация объединяет эти шаги так, что в любой момент времени несколько шагов выполняются одновременно. Одна команда выбирается, вторая дешифрируется, а третья исполняется. Чтобы делать это все одновременно, сам процессор должен быть разделен на полунезависимые устройства, каждое из которых одновременно с другими реализует свою функцию. Представим, что процессор разделен на устройство опережающей выборки, дешифратор и операционное устройство, и посмотрим, как несколько операций можно производить одновременно (см. табл. 7.4).

За 6 единиц времени выбраны, дешифрированы и исполнены 4 полные команды, а две команды обработаны частично (выбраны и возможно дешифрированы, но еще не исполнены). Термин «опережающая выборка» означает, что команда выбирается одновременно с другими действиями, поэтому другие устройства не ожидают завершения выборки. Без конвейеризации за то же время будут выполнены только две команды.

Таким образом, конвейеризация в этом примере позволила за то же время выполнить в два раза больше операций.

Таблица 7.4. Действие конвейеризации

Такты	Опережающая выборка	Дешифратор	Операционное устройство
Интервал 1	Команда 1		
Интервал 2	Команда 2	Команда 1	
Интервал 3	Команда 3	Команда 2	Команда 1
Интервал 4	Команда 4	Команда 3	Команда 2
Интервал 5	Команда 5	Команда 4	Команда 3
Интервал 6	Команда 6	Команда 5	Команда 4

Однако реальная жизнь сложнее приведенной схемы. Опережающая выборка, дешифрирование и исполнение занимают разные интервалы времени для различных команд; иногда исполняемая команда занимает шину данных, задерживая выборку следующей команды. Кроме того, устройство опережающей выборки всегда считывает команды последовательно, независимо от результата действий команд. Любая команда, нарушающая естественный порядок (например, JMP или CALL) или прерывание вызывают освобождение очередей команд. Команда, которая обрабатывается сразу после перехода, длится дольше, так как операционное устройство должно ожидать ее выборки и дешифрирования.

Процессор 80386 разделен на устройства, которые функционируют следующим образом. Устройство опережающей выборки считывает команды в процессор. Оно использует шину, когда она не занята операционным устройством, считывает команды из памяти и помещает их в очередь.

Размер очереди — 16 байт (что соответствует 3—7 командам).

Устройство дешифрирования преобразует команду в формат, стандартный для устройства управления и операционного устройства. В очереди дешифратора могут храниться три команды. Частью дешифрированной команды является указатель управляющего постоянного запоминающего устройства, которое хранит микрокод, управляющий устройствами 80386.

Операционное устройство действует по командам от устройства управления, фактически выполняя команду. Оно содержит арифметико-логическое устройство и набор регистров. Сегментное устройство вычисляет адреса и контролирует защиту сегментов. Адрес от этого устройства подается в страничное устройство, которое (если страничная организация включена) преобразует его в физический адрес; в противном случае используется адрес от сегментного устройства. Это же устройство использует устройство опережающей выборки для своих запросов и страничное устройство для доступа к таблицам страниц в памяти.

Конвейеризация применяется и в других микропроцессорах. Но 80386 уникален в том, что он осуществляет преобразование адреса и контроль защиты для сегментов и страниц внутри микросхемы. В других процессорах для этого применяется отдельная микросхема управления памятью (MMU). Наличие ее уменьшает производительность процессора. Кроме того, использование с одним процессором разных MMU приводит к проблемам совместимости. В процессоре 80386 MMU внутренняя, и ее операции совмещаются с операциями других устройств, поэтому на вычисление адреса дополнительное время не расходуется, на число тактов команды влияет только фактические обращения к памяти. Некоторые сложные методы адресации требуют дополнительного такта (см. гл. 2).

7.3.2. Блоки процессора 80386

Подробнее остановимся на организации процессора и его работе. Мы начнем с обзора устройств, а затем более детально рассмотрим каждое устройство. В последней части главы показано, как происходит выполнение команд.

УСТРОЙСТВО ОПЕРЕЖАЮЩЕЙ ВЫБОРКИ

Главная функция этого устройства — использовать свободное время шины для считывания команд в процессор. Очередь размером 16 байт хранит команды до их считывания дешифратором. Это устройство имеет низший приоритет по доступу к шине и не знает о том, когда переходы или вызовы изменяют естественный порядок выполнения команд. Оно всегда считывает команды последовательно; фактически оно считывает двойное слово и не заботится о том, что оно содержит полные команды или части двух команд. Когда осуществляется переход, содержимое очередей опережающей выборки и дешифрирования очищается.

Когда производится переход и очереди освобождаются, обработка команд запрещается. Команда выбирается, сразу же дешифрируется и затем исполняется. Если есть такты, когда шина не занята при дешифрировании и исполнении, устройство опережающей выборки начинает заполнение очереди. Если первой командой после перехода является команда MUL или другая подобная команда, устройство быстро заполнит очередь. Например, программные циклы выполняются лучше, когда команда типа MUL находится в начале цикла. Простые команды типа CLC или передачи регистр-регистр выполняются так быстро, что они не позволяют устройству опережающей выборки заполнить очередь, что вносит новые задержки.

УСТРОЙСТВО ДЕШИФРИРОВАНИЯ

Это устройство также экономит время для всего процессора. Так как оно не использует шину, то может работать всегда, когда очередь устройства опережающей выборки непуста. Каждый байт в команде определяет, есть ли еще байты, сообщая дешифратору об окончании команды (дешифратор обрабатывает байт за такт).

Функция дешифратора — преобразовать команды в форму, которую может быстро использовать операционное устройство. Возможный формат дешифрированной команды приведен в листинге 7.1. Команда в целом занимает до 112 бит. Первые три бита сообщают о наличии префиксов по принципу «да/нет». Следующие

12 бит содержат адрес управляющей ROM микрокода, который фактически вызывает исполнение команды. Двенадцать бит позволяют адресовать 4096 элементов; в управляющей ROM фактически есть 2560 слов по 37 бит.

Листинг 7.1. Элемент очереди дешифрированной команды

Длина	Значение
1	Присутствует префикс LOCK
1	Присутствует префикс размера адреса
1	Присутствует префикс размера операнда
12	Адрес элемента управляющего ROM
3	Сегментный регистр
4	Номер базового регистра (и бит действительности)
4	Номер индексного регистра (и бит действительности)
2	Масштабный коэффициент
32	Смещение
32	Непосредственный операнд
3	Операнд-регистр (источник)
3	Операнд-регистр (получатель)
14	Другие флажки, модификаторы и т. д.

Следующие трехбитные поля определяют номера сегментного регистра, базового регистра и индексного регистра, используемые командой в адресации. Два бита действительности сообщают, преобразуется ли в команде базовый и индексный регистры. Двухбитное поле масштабного коэффициента допускает масштабы 1, 2, 4 и 8.

Два больших 32-битных поля содержат смещение и непосредственный операнд. Еще два поля определяют регистры операндов. Из дешифрированной команды операционное устройство узнает находится ли операнд в памяти, в регистре или является непосредственным значением. Последние 24 бита содержат флажки, которые здесь не рассматриваются. Отметим, что вся дешифрированная команда — это преобразование машинного языка, сформированного ассемблером. Если бы код можно было записать прямо в этой 112-битной форме, то ассемблирование и дешифрирование стало бы ненужным (но это очень сложно для восприятия человеком).

УСТРОЙСТВО УПРАВЛЕНИЯ

Это устройство объединяет другие устройства, управляя их операциями и взаимодействиями. Его ядром служит управляющая ROM, содержащая внутренние команды 80386. Доступ к ROM определяют 12 бит в дешифрированной команде.

Формат микрокодовой команды составляет интеллектуальную собственность фирмы Intel.

Однако кое-какие общие факты о нем известны. Наличие управляющих слов в ROM упрощает отладку всего процессора. Изменять ROM не так легко, но все же легче, чем все остальное в процессоре. Даже серьезные ошибки в процессоре часто «ремонтируются» путем репрограммирования ROM и обхода ошибки, а не изменением устройства самой микросхемы.

37-битовые слова микрокода довольно узкие. Широкие управляющие слова содержат достаточно информации для одновременного прямого управления каждым устройством процессора. Используемые узкие слова, по-видимому, одновременно управляют только устройством. Так как устройство опережающей выборки команд и дешифратор реализуют простые операции, можно предположить, что большая часть слов в ROM управляют действиями других устройств, в частности операционным устройством.

Устройство управления, наряду с шинным устройством, взаимодействует с внешним миром. Входные сигналы подаются в схему приоритетов запросов, которая сообщает процессору, когда что делать. Это очень важно, так как в любой момент времени на разных этапах обработки могут находиться 6 или 7 команд и данный процессор может быть частью мультипроцессорной системы, в которой шиной управляют несколько компонент.

Следующие три устройства довольно сложные. Мы кратко рассмотрим их, ориентируясь на аппаратные детали. В следующем параграфе описано, как работают и взаимодействуют эти устройства.

ВРЕМЕННЫЕ СООТНОШЕНИЯ

Эти соотношения довольно сложные, но здесь действуют некоторые основные правила. Пересылка информации из одной очереди в другую, из очереди в регистр и из одного регистра в другой длится один такт. Инфор-

мация должна поступить в одно из этих мест хранения до начала следующего такта, иначе она может быть изменена новыми данными, появившимися на внутренней шине. Любая пересылка между регистрами (включая и внутренние регистры 80386) занимает один такт, независимо от того, сколько сумматоров, мультиплексоров и т. д. находится между регистрами. Такие одиночные такты необязательно отражаются на общем времени выполнения команды, так как многие из них совмещаются с другими операциями. Заметные задержки вносит получение операндов из памяти (4 плюс дополнительные такты) и использование операнда в памяти как получателя. На это уходит 5 тактов: четыре на получение операнда и один для записи в память. Второй такт записи можно совместить с обработкой операционным устройством следующей команды.

ОПЕРАЦИОННОЕ УСТРОЙСТВО

Операционное устройство состоит из следующих блоков: модуля регистров (со всеми общими и управляющими регистрами) и математического устройства (включая сдвигатель, сумматор и специальные схемы для умножения и деления). Операционное устройство имеет два входа: вход от шинного устройства содержит операнды команды из памяти, а вход от дешифратора содержит непосредственные операнды и смещения для операндов. Мультиплексор выбирает один из этих входов и направляет его содержимое в регистр или временный регистр для использования в будущем.

Значения из регистров можно передать в сегментное устройство для адресных вычислений или использовать в арифметической секции.

Рассмотрим, например, команду ADD EAX,MEM,DWOKD. Операнд MEM,DWOKD считывается шинным устройством и помещается во временный регистр (это занимает 4 цикла шины, так как операнды берутся из памяти во время выполнения). Затем оба операнда подаются в арифметическую секцию, где они суммируются, и результат помещается в специальный регистр AR_OUT. Суммирование и загрузка AR_OUT занимают один цикл шины. Второй цикл шины необходим для возвращения результата из AR_OUT в EAX.

Операционное устройство не может больше ничего делать до передачи результата из AR_OUT в EAX, так

как следующая команда может использовать EAX. Если новое значение еще не передано в регистр, следующая команда может использовать старое значение, пока новое значение передается по шине в EAX. Такая ситуация (команде требуется значение, которое еще не помещено туда, куда нужно) называется блокировкой. Так как в 80386 нет схемы определения такой ситуации, он просто занимает два цикла шины даже для простейшей операции (типа передачи из регистра в регистр), избегая этим блокировки.

80386 обеспечивает лучшую производительность, чем 16-битные процессоры, работая с данными вдвое большего размера. Часть этих возможностей реализована аппаратно. Например, параллельный сдвигатель имеет ширину 64 бита. Он обеспечивает 32-битный сдвиг 32-битного операнда за один такт. Улучшено кодирование некоторых команд. Например, команда MUL длится от 9 до 38 тактов: 6 тактов плюс один такт на каждый значащий бит множителя (минимум 3). Умножение на 3 (старший бит во втором разряде) занимает (6+3) 9 тактов. Умножение на число с единицей в старшем бите длится (6+32) 38 тактов. Простые операции выполняются очень быстро.

Большинство операций, например, сложение, выполняются с одинаковой скоростью независимо от размера операндов.

УСТРОЙСТВО СЕГМЕНТАЦИИ

Адрес памяти из операционного устройства состоит из двух значений: смещения и индекса. Эти два числа суммируются и сохраняются во временном регистре. Если требуется база (т. е. адрес имеет базу, индекс и смещение), результат нужно вновь подать на сумматор и сложить с базой, полученной от операционного устройства. Это занимает лишний такт независимо от того, масштабируется индекс или нет. Результатом сложения является эффективный адрес, который служит относительной частью адреса.

Устройство сегментации сравнивает эффективный адрес с пределом длины используемого сегмента из дескриптора сегмента. Все необходимые дескрипторы текущих сегментов хранятся в процессоре 80386. Права доступа также сопоставляются с операциями процессора и

при их несоответствии возникает прерывание. Если контроль защиты прошел успешно, база сегмента прибавляется к эффективному адресу, давая линейный адрес. Если страничная организация блокирована, этот адрес применяется для обращения к памяти как физический адрес.

СТРАНИЧНОЕ УСТРОЙСТВО

В простых системах страничная организация не применяется, и это устройство блокировано. Но если оно задействовано, то должно очень быстро выполнять свои операции, чтобы не замедлять процессор. Это довольно трудная проблема, так как каждая страница имеет информацию о защите и адресации; при страничном кадре 4 Кбайта даже сравнительно небольшое адресное пространство 16 Мбайт содержит 4 Кбайта страниц. Необходимую информацию для всех страниц невозможно хранить в процессоре и приходится привлекать оперативную память.

Структура памяти состоит из двух частей: каталога страниц (он сообщает, где находится информация о каждой странице) и таблицы страниц (она содержит дескрипторы для каждой страницы). Для получения нужной страницы необходимо считать каталог страниц, который помогает образовать адрес для считывания таблицы страниц, которая дает необходимую информацию для нахождения фактической страницы. Как же реализовать страничную организацию без больших задержек обращения к памяти?

Решение подобно кэшированию памяти: хранить часто используемую информацию в небольшой и быстрой кэш-памяти. Кэш-страница находится в страничном устройстве и называется страничной кэш-памятью или буфером преобразования TLB. Название происходит от способа «заглядывания» в буфер страничным устройством для проверки необходимой адресной информации до обращения к каталогу страниц в памяти.

Разработчики микропроцессора поместили на кристалле (275000 транзисторов) множество блоков. После размещения устройств опережающей выборки, общих регистров и страничного устройства осталось место только

для одной кэш-памяти. В кэш можно хранить код, адресную информацию и данные. Разработчики 80386 выбрали внутреннюю кэш для адресации (TLB) в страничном устройстве.

TLB имеет четыре комбинации по восемь элементов (см. рис. 7.5). Каждый элемент TLB имеет две части. Первая называется VAh — это старшие 17 бит виртуального адреса страницы. Вторая часть — это копия фактического 32-битного элемента таблицы страниц, который обычно должен считываться из памяти.

Следующий материал более сложен. Но если вы хотите узнать, как работает TLB, прочитайте его. Три бита виртуального адреса (биты 14—12) дают смещение от 0 до 7. Напомним, что TLB имеет четыре комбинации по восемь элементов, поэтому данные три бита адресуют по одному элементу из каждой комбинации.

Каждый из четырех элементов считывается одновременно. Старшие 17 бит текущего виртуального адреса сравниваются с 17-битным полем VAh в каждом элементе. При наличии соответствия элемент таблицы страниц, присоединенный к элементу TLB, как раз является нужным, и обращение к памяти не требуется.

Но вы можете сказать: «Стоп! 17 бит виртуального однозначного адреса недостаточно для определения любой страницы. Требуется 20 бит». Однако 17 бит виртуального адреса, объединенные с тремя битами индекса TLB как раз и дают 20 бит.

Это означает, что каждая страница в памяти отображается на один из 8 уровней смещения в TLB. При копировании в TLB страница должна быть помещена на соответствующий уровень смещения; схемно определяется, в какую комбинацию она направляется.

Обнаружение соответствия называется страничным попаданием. Примерно 98% адресов страничной памяти дают попадание. Напомним, что 32 элемента TLB позволяют адресовать (32 элемента × 4 Кбайта в странице) 128 Кбайт памяти. Так как большинство программ большую часть времени используют одни и те же страницы, то такая кэш обеспечивает большое число попаданий. Промахи заставляют процессор обратиться к памяти за информацией о странице; для этого необходимы два отдельных обращения к памяти. Прикладной программист не может влиять на процент попаданий, так как программа вообще может выполняться без использования страниц.

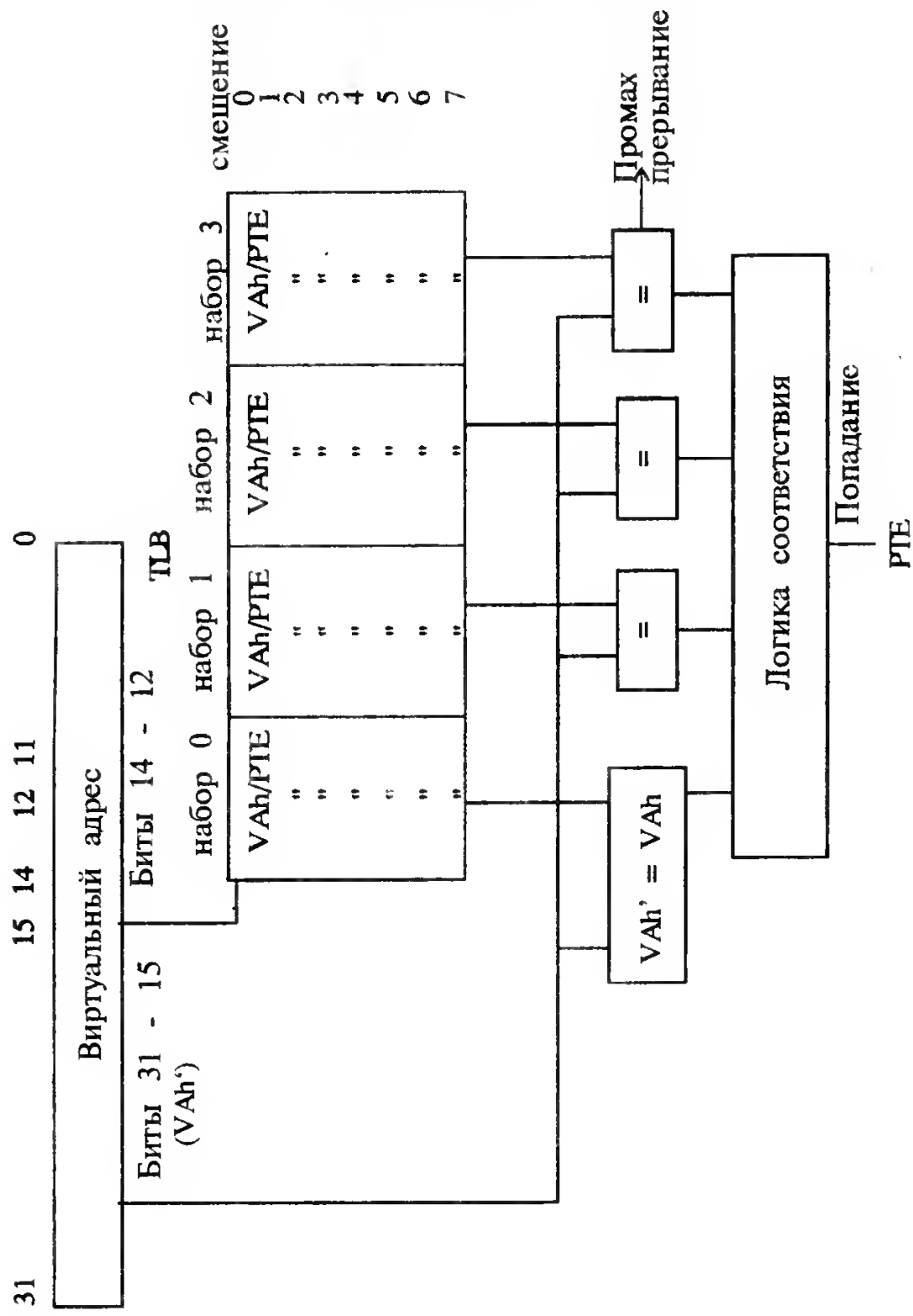


Рис. 7.5 Диаграмма для 32-элементного TLB

Однако следует учитывать, что программы с переходами через большие области данных (более 128 Кбайт) вызывают больше промахов, чем другие.
Страничное устройство формирует физический адрес. Это формирование производится тремя способами: страничная организация выключена и используется линейный адрес; попадание в TLB позволяет построить и проверить нужный адрес без дополнительных обращений к памяти; промах в TLB вызывает дополнительные обращения к памяти, которые необязательно нарушают конвейеризацию и замедляют процессор.

ШИННОЕ УСТРОЙСТВО

Это устройство управляет всем взаимодействием процессора 80386 с внешним миром (за исключением некоторых сигналов устройства управления).

Шинное устройство воспринимает запросы данных от операционного устройства и выборки кода от устройства опережающей выборки, упорядочивает их по приоритетам (выборка кода последняя) и выполняет их. Это же устройство управляет взаимодействием с сопроцессорами и другими микросхемами, которые могут управлять системной шиной.

7.3.3. Выполнение команд в 80386

В этом разделе суммируется материал о внутренних операциях 80386 на примере выполнения двух команд. Показано, что происходит в процессоре в каждом такте по мере прохождения двух команд. В листинге 7.2 приведены две команды.

Листинг 7.2. Пример команд

Адрес	Содержимое (код)	Команда
00120000	03 D9	ADD EBX, ECX
00120002	64: 2B B4 DA 01234567	SUB ESI, FS: 01234567H [EDX] [EBX×8]
0012000A		

Ради простоты положим, что код выполняется сразу после команды перехода, очищающей конвейер 80386 (в частности очереди опережающей выборки и дешифрирования).

Устройство опережающей выборки начинает с выборки первого двойного слова командного потока (код по адресу 00120000 и далее) и запоминает четыре байта в своей очереди. Это устройство использует свободные циклы шины для заполнения своей очереди и продолжает выборку, когда очередь начинает освобождаться. Далее мы не будем останавливать внимание на этом устройстве, так как его работа довольно проста.

ТАКТ 1

ADD: Устройство дешифрирования начинает работать, как только в очереди опережающей выборки появляется первый байт. Оно берет этот байт (со значением 03) и дешифрирует его, помещая информацию в очередь дешифрированных команд (указатель очереди адресует первый пустой элемент в очереди дешифратора). Первый байт сообщает дешифратору о том, что командой является ADD и что она имеет байт ModRM, указывающий на продолжение команды.

ТАКТ 2

ADD: Дешифратор берет байт 2 из устройства опережающей выборки; это байт ModRM команды ADD со значением D9. Байт сообщает об использовании регистров EBX и ECX и об окончании команды. Дешифрированная команда отмечается готовностью READY и остается в очереди для исполнения устройством управления. Указатель очереди продвигается на следующий пустой элемент; это происходит после успешного дешифрирования каждой команды.

ТАКТ 3

ADD: В конвейере нет предыдущих неисполненных команд, поэтому команда ADD начинает исполняться немедленно. Первое микрослово считывается из управляющей ROM и два операнда EBX и ECX считываются из

регистрового файла в операционном устройстве на шины смещения и индекса.

Во второй половине такта два значения подаются в ALU и микрокод определяет их сложение. Результат помещается в регистр AR_OUT.

SUB: Дешифратор «видит», что команда SUB имеет префикс FS замены сегмента (значение 64). Поле сегментного регистра в очереди изменяется на определение FS вместо принимаемого по умолчанию DS.

ТАКТ 4

ADD: В первой части такта в операционном устройстве ничего не происходит. Во второй части такта содержимое AR_OUT записывается в EBX, и флажки в EFlags модифицируются, отражая особенности результата команды ADD. На этом команда ADD закончена. Отметим, что время выполнения команды ADD составляет два такта, а у нас получилось четыре; объясняется это тем, что конвейеризация в нашем примере исключена.

SUB: Дешифратор преобразует байт команды SUB (значение 2B), показывающий наличие байта ModRM.

Если бы эта команда была полностью дешифрована, смещение в обход регистрового файла было бы выдано на шину смещения, а индекс или база считывались бы из регистрового файла на шину индекса.

ТАКТ 5

Все такты после этого относятся к команде SUB.

Дешифратор преобразует байт ModRM (значение B4), который показывает наличие байта SIB. Обычно это дешифрирование совмещается с выполнением другой команды, но у нас очередь освобождена командой JMP.

ТАКТ 6

Дешифратор преобразует байт SIB (значение DA), показывающий наличие 4-байтного смещения.

ТАКТ 7

Дешифратор помещает 4-байтное смещение (значение 01234567H) прямо в поле смещения элемента очереди для команды SUB. Теперь дешифрирование команды SUB закончено.

ТАКТ 8

Смещение обходит модуль регистров и помещается на шину смещения, а индекс считывается из регистрового файла (EBX) и помещается на шину индекса.

В конце такта индекс умножается на 8 и прибавляется к смещению; результат направляется в запоминающий регистр устройства сегментации.

ТАКТ 9

Определен базовый регистр в дополнение к уже вычисленному смещению и индексу, поэтому база (EDX) считывается из регистрового файла; предыдущий результат берется из запоминающего регистра и два числа складываются в сумматоре. Этот дополнительный такт нужен, когда в командах используется база, смещение и индекс.

Отметим, что при наличии команды после SUB она была бы теперь уже выбрана и дешифрирована и достигла бы операционного устройства.

ТАКТ 10

Указанный сегментный регистр FS берется из модуля регистров, и граница этого сегмента сравнивается с эффективным адресом в запоминающем регистре, а другая схема проверяет, можно ли считывать из этого сегмента (права доступа), так как результат записывается в регистр.

Во второй части такта базовый адрес сегмента читается из дескриптора сегмента и прибавляется к эффективному адресу, давая линейный адрес; он помещается в запоминающий регистр страничного устройства.

ТАКТ 11

Линейный адрес используется для обращения к TLB (мы считаем, что происходит попадание), и физический адрес операнда находится путем слияния старших 20 бит из элемента TLB (который именует страничный кадр) и младших 12 бит линейного адреса (которые определяют байт внутри страницы).

В конце такта результирующий физический адрес помещается в запоминающий регистр.

ТАКТЫ 12—14

Мы полагаем, что шинное устройство свободно и память имеет один такт ожидания (3-тактная память). Этот такт расходуется на считывание данных из памяти. Отметим, что обращения к памяти, которые являются частью выполнения команды, не конвейеризуются. В конце такта 14 данные находятся в шинном устройстве, откуда их можно читать.

ТАКТ 15

Данные из памяти подаются на один вход ALU, значение из ESI подается на второй, и производится вычитание с помещением результата в AR_OUT.

ТАКТ 16

В первой половине такта ничего не происходит; во второй половине новое значение записывается из AR_OUT в ESI, и устанавливаются флажки.

7.4. ПРИМЕРЫ СЛОЖНЫХ КОМАНД

Приведенный выше пример включает в себя двухбайтную команду ADD, которая выполняется за четыре такта, и восьмибайтную команду SUB, которая требует еще 12 тактов. В листинге 7.3 приведены более сложные команды 80386. Сокращения AS, OS и FS означают префиксы размера адреса и замены сегмента.

Листинг 7.3. Сложные команды

```
КОМАНДА 1: LOCK: AS: OS: FS: BTS 12345678H [EDI][EBX*4],
                24
КОД:       F0: 67: 66: 64: OF BA 94 9F 12345678 18
КОМАНДА 2: LOCK: AS: OS: FS: ADD 12345678H [EDI]
                [EBX*4], 9ABCDE02H
КОД:       F0: 67: 66: 64: 81 84 9F 12345678 9ABCDE02
```

Команда 1 длиной 13 байт требует 10 тактов на дешифрирование и 9 тактов на выполнение. Команда 2 длиной 15 байт дешифрируется 9 тактов и выполняется 8 тактов. Интересно, что в обоих командах дешифрирование длится дольше выполнения.

7.5. ОСОБЕННОСТИ ПРОГРАММИРОВАНИЯ

Ниже приведены некоторые простые правила разработки эффективных прикладных программ для исполнения процессором 80386.

1. *Минимизировать вызовы OS.*
   Один вызов OS связан с выполнением тысяч байт кода; в системе на базе 80386 программы OS могут выполняться под гипервизором, управляющим системными ресурсами. Вызов OS может привести к вызову гипервизора, что вносит еще большие задержки.
2. *Выравнивать модули программы на границах страниц.*
   Такой прием в системе страничной памяти помогает предотвратить свопинг страниц.
3. *Обеспечить локальность обращений.*
   Локальность важна для кэш-памяти, страничных RAM, устройства опережающей выборки, очереди TLB и др. Программы из небольших компактных модулей выполняются эффективнее, так как позволяют использовать встроенные в процессор средства ускорения.

4. *Помещать команды с быстрым дешифрированием и долгим выполнением в начале цикла.*
   Примером таких команд служит умножение. Такие команды помогают процессору заполнить очереди после выполнения перехода.
5. *Адреса переходов выравнивать по двойным словам.*
   При этом процессор, замедленный командой перехода, не делает дополнительных обращений к памяти для считывания первой команды после перехода.
6. *Выравнивать операнды в памяти по их естественным границам.*
   Это ускоряет обращения к памяти.
7. *Массивы и другие большие структуры данных выравнивать по границам двойных слов и даже страниц.*
   Этим ускоряется доступ к массивам. Выравнивание по странице следует рассмотреть, когда массив или другая структура достигает 2 Кбайт (половина страницы). При этом ускоряется доступ внутри структуры и повышается производительность в системе виртуальной памяти со страничной организацией.

УВАЖАЕМЫЕ ГОСПОДА!

Акционерное издательское предприятие "КОНКОРД" имеет честь предложить вам свое сотрудничество в издании переводов справочников, каталогов и научно-технической литературы по новым информационным технологиям.

Фирма имеет штат технических переводчиков и редакторов, имеющих большой опыт практической работы в следующих областях: компьютерная и микропроцессорная техника, объектно-ориентированное программирование.

В издательстве "КОНКОРД" с разрешения московского представительства фирмы Интел Текнолоджиз переведено и издано на русском языке справочное руководство «Продукция фирмы INTEL» («Product Overview 1992»).

«КОНКОРД» подготовил к изданию книгу Гради Буча «Объектно-ориентированное программирование с примерами применения» на русском языке.

Фирма принимает заявки на разработку рекламных сообщений и размещение рекламы в своих изданиях.

Фирма заинтересована в установлении долговременных взаимовыгодных связей с российскими и зарубежными деловыми партнерами, занимающимися распространением современных технологий, и готова сотрудничать с ними.

Контактные телефоны: 459-22-93, 191-61-33.
Факс: 459-22-93.

1672