

АССЕМБЛЕР



Основы программирования
на языке ассемблера

Интерфейс с языком C

Программирование
сопроцессора i80x87

Резидентные программы



Дискета содержит
архив пакета a86 v4.05,
исходные тексты программ
с рассматриваемыми примерами

Андрей Жуков
Андрей Авдюхин

Ассемблер

Санкт-Петербург
«БХВ-Петербург»

2002

УДК 681.3.068+800.92Ассемблер
ББК 32.973.26-018.1
Ж86

Жуков А. В., Авдюхин А. А.

Ж86 Ассемблер. — СПб.: БХВ-Петербург, 2002. — 448 с.: ил.
ISBN 5-94157-133-X

Книга является руководством по программированию на ассемблере для микропроцессорных систем на базе i80x86 и посвящена практическому применению этого языка на примере и с использованием ассемблера a86. Рассматриваются дополнительные возможности языков ассемблера: макрокоманды и связь с языками высокого уровня. Приводится обзор стилей языков ассемблера для разных вычислительных систем. В качестве иллюстрации применения ассемблеров рассмотрены различные вопросы, связанные с многозадачностью, — обработка прерываний и резидентные программы. Изложенный материал снабжен примерами, контрольными вопросами и заданиями к практическим работам.

Для начинающих программистов

УДК 681.3.068+800.92Ассемблер
ББК 32.973.26-018.1

Группа подготовки издания:

Главный редактор	<i>Екатерина Кондукова</i>
Зам. главного редактора	<i>Анатолий Адаменко</i>
Зав. редакцией	<i>Анна Кузьмина</i>
Редактор	<i>Петр Науменко</i>
Компьютерная верстка	<i>Ольги Сергиенко</i>
Корректор	<i>Зинаида Дмитриева</i>
Дизайн обложки	<i>Игоря Цырульников</i>
Зав. производством	<i>Николай Тверских</i>

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 05.04.02.

Формат 70×100^{1/16}. Печать офсетная. Усл. печ. л. 36,12.

Тираж 4000 экз. Заказ №

"БХВ-Петербург", 198005, Санкт-Петербург, Измайловский пр., 29.

Гигиеническое заключение на продукцию, товар № 77.99.02.953 Д.001537.03.02
от 13.03.2002 г. выдано Департаментом ГСЭН Минздрава России.

Отпечатано с готовых диапозитивов
в Академической типографии "Наука" РАН
199034, Санкт-Петербург, 9 линия, 12.

Содержание

Введение	13
Структура изложения	14
ЧАСТЬ I. ОСНОВЫ АССЕМБЛЕРА	17
Глава 1. Данные, имена, типы	19
1.1. Структура программы	19
1.2. Директивы определения данных	20
1.3. Обозначение чисел	22
1.4. Символические обозначения чисел, выражения	23
1.5. Переменные и метки	25
1.6. Типы имен	27
1.7. Типы и выражения	28
Глава 2. Определение имен	31
2.1. Алгоритм трансляции	31
2.2. Повторное определение имен	37
2.3. Локальные имена	37
2.4. Предопределенные имена	38
2.5. Имя <i>end</i>	38
Глава 3. Практикум по программированию данных	42
3.1. Запуск ассемблера <i>a86</i>	42
3.2. Программирование данных	44
Глава 4. Просмотр данных в отладчике	48
4.1. Запуск и завершение сеанса отладки	48
4.2. Экран отладчика	49
4.3. Окна отображения данных	50
4.4. Сохранение нажатий клавиш	51
4.5. Форматы вывода данных	51
4.5.1. Базовые форматы	52
4.5.2. Составные форматы	53
4.6. Задания на самостоятельную работу	54

Глава 5. Способы адресации	56
5.1. Данные процессора	56
5.2. Обозначения операндов машинных команд	58
5.3. Способы адресации операндов	59
5.3.1. Регистровая и непосредственная адресация	59
5.3.2. Адресация данных в памяти	60
Прямая адресация	60
Косвенная адресация	62
5.3.3. Ограничение на адресацию операндов в памяти	65
Глава 6. Система команд i80x86	66
6.1. Режим непосредственного выполнения в d86.....	66
6.2. Способы адресации операндов	67
6.2.1. Регистровая, непосредственная и прямая адресация.....	67
6.2.2. Косвенная адресация.....	68
Косвенная адресация по значению одного регистра.....	69
Косвенная адресация по сумме значений двух регистров	69
6.3. Обзор системы команд процессора i80x86	69
6.3.1. Команды пересылки.....	70
6.3.2. Арифметические команды	71
6.3.3. Логические команды	72
6.3.4. Команды сдвигов и вращений	72
6.3.5. Команды передачи управления	73
Адресация в командах передачи управления.....	74
Команды условных переходов	75
6.3.6. Воздействие команд на флаги	77
6.3.7. Строковые команды	80
Глава 7. Программирование циклов	83
7.1. Поиск в массиве байтов	83
7.2. Поиск в массиве слов	84
7.3. Поиск байта со значением больше заданного	86
7.4. Подсчет байтов в заданном диапазоне значений	87
7.4.1. Алгоритмическое решение.....	87
7.4.2. Табличное решение	88
Глава 8. Исследование программ в d86.....	90
8.1. Пример исследуемой программы	90
8.2. Названия регистров в отображении данных	92
8.3. Режимы выполнения.....	92
8.4. Постоянные точки останова	94
8.5. Редактирование команд.....	94
8.6. Принудительный останов.....	94
Глава 9. Примеры программ.....	96
9.1. Обработка данных на уровне бит	96
9.2. Вложенные циклы.....	97

9.3. Программирование ввода/вывода.....	98
9.4. Проблема опережающих ссылок	101
9.5. Решение проблемы опережающих ссылок	102
9.6. Упаковка четырехбитных кодов.....	103
9.7. Задания на составление программ	105
9.7.1. Задания первого уровня сложности.....	105
9.7.2. Задания второго уровня сложности.....	107

Глава 10. Сегменты и ехе-программы 112

10.1. Сегментная модель памяти.....	112
10.2. Сегменты в com-программе	116
10.3. Сегменты в ехе-программе.....	117
10.4. Особенности подготовки ехе-программы.....	121
10.5. Построение ехе-программ из нескольких модулей	123
10.6. Практикум	125
10.6.1. Компоновка.....	125
10.6.2. Организация отладки.....	126
10.6.3. Компоновка многомодульной программы.....	127
10.6.4. Задание на самостоятельную работу.....	128

ЧАСТЬ II. РАСШИРЕННЫЕ ВОЗМОЖНОСТИ АССЕМБЛЕРА 129

Глава 11. Макрокоманды и условная трансляция 131

11.1. Макрокоманды.....	132
11.1.1. Макрокоманды без параметров.....	132
11.1.2. Макрокоманды с параметрами.....	133
11.1.3. Правила подстановки параметров.....	135
11.1.4. Функции от параметров	136
11.1.5. Циклы по параметрам	138
R-циклы	139
Q-циклы	139
Задание шага в r- и q- циклах	140
11.1.6. Цикл по литерам параметра.....	142
11.1.7. Циклы, не зависящие от параметров.....	143
11.1.8. Вложенные циклы	145
Вложенные циклы с фиксированным числом повторений.....	145
Сложная обработка фактических параметров.....	146
11.2. Средства условной трансляции.....	146
11.2.1. Директива <code>#if..#endif</code>	147
11.2.2. Макроопределения и условная трансляция	147
11.2.3. Условия	149
Арифметические отношения	149
Объединение условий.....	149
Проверка существования.....	150
Определение имен в командной строке	150
Инвертирование логических значений.....	151
Пример условной трансляции в макроопределениях.....	152
11.2.4. Условная трансляция в циклах.....	154

11.3. Метки в макроопределениях	155
11.4. Средства отладки макрокоманд	158
11.5. Практикум	159
11.5.1. Макрокоманды без циклов	159
11.5.2. Макрокоманды с циклами	160
11.5.3. Функции от параметров	161
11.5.4. Вложенные циклы	161
11.5.5. Условная трансляция в макроопределениях	162
11.6. Вызов макрокоманд в i86.....	163

Глава 12. Структурный ассемблер..... 164

12.1. Логика условных переходов	164
12.2. Оператор <i>if</i> в i86.....	166
12.3. Способы реализации структурного ассемблера.....	167
12.3.1. Реализация при помощи макроассемблера.....	167
12.3.2. Реализация при помощи препроцессора.....	167
12.4. Структурный ассемблер bsp86.....	168
12.4.1. Синтаксис логических операторов.....	168
Оператор цикла с постусловием.....	169
Оператор цикла с предусловием.....	169
Оператор счетного цикла	170
Оператор бесконечного цикла.....	170
Выход из цикла	171
Оператор условного выполнения	171
Последовательность операторов	172
Оператор мультиветвления.....	173
12.5. Реализация мультиветвления в bsp86	174
12.6. Выполнение примеров.....	177
12.7. Диагностические сообщения bsp86	179
12.7.1. Сообщения первого типа	179
12.7.2. Особенности лексического анализатора.....	180
12.7.3. Сообщения второго типа	180
12.7.4. Ошибки в списках вариантов.....	181
12.8. Пример использования bsp86	182
12.9. Практикум	183

Глава 13. Интерфейс с языком С..... 185

13.1. Машинное представление данных языка С.....	185
13.2. Правила использования регистров	186
13.2.1. Сегментные регистры	186
13.2.2. Регистры общего назначения	187
13.2.3. Регистр флагов	187
13.3. Ассемблерные вставки	187
13.3.1. Ассемблерные вставки в Microsoft/Borland-С.....	187
13.3.2. Ассемблерные вставки в Watcom-С.....	188
13.4. Интерфейс С-ассемблер при раздельной трансляции.....	191
13.4.1. Соглашения о сегментах кода	191
13.4.2. Соглашения об именах С-объектов	192

13.4.3. Соглашения о результате функции	194
13.4.4. Передача параметров через регистры	195
13.4.5. Передача параметров через стек	197
Передача при ближних вызовах	197
Кадр стека при дальних вызовах	199
Инструкции <i>enter</i> и <i>leave</i>	200
Порядок передачи параметров.....	202
Соглашения об удалении параметров.....	203
Сокращенные обозначения параметров и локальных данных	204
Параметры дальние указатели	205
Обращение к глобальным данным.....	206
13.5. Выполнение примеров.....	209
Глава 14. Обработка BCD-данных.....	212
14.1. Форматы BCD.....	212
14.2. Операции над упакованными BCD	213
14.2.1. Сложение и вычитание	213
14.2.2. Умножение и деление	216
14.2.3. Дополнительные возможности <i>aad</i> и <i>aam</i>	218
14.3. Операции над упакованными BCD	219
14.3.1. Инструкции <i>daa</i> и <i>das</i>	219
14.3.2. Преобразования для деления и умножения.....	220
14.4. Операции над знаковыми BCD	221
14.5. Команды, воздействующие на флаг <i>a</i>	221
14.6. Практикум	222
Глава 15. Математический сопроцессор	224
15.1. Проверка наличия FPU	225
15.2. Загрузка и выгрузка целых чисел.....	225
15.3. Недопустимые операции и NaN	228
15.4. Организация массива данных в виде стека	228
15.5. Вычисления в стековой машине.....	230
15.5.1. Двуместные операции	230
15.5.2. Выражения в обратной польской записи.....	231
15.5.3. Расширения стековой машины в i80x87.....	232
15.6. Представление данных в FPU.....	233
15.7. Стандартный формат вещественных данных	236
15.8. Программная модель i80x87	237
15.8.1. Флаги особых ситуаций	239
15.8.2. Битовые поля управляющего слова	239
15.8.3. Битовые поля слова состояния.....	240
15.8.4. Доступ к указателям инструкции и операнда.....	242
15.9. Операции i80387	243
15.9.1. Пересылки	243
Загрузка данных	243
Команды выгрузки.....	245
Команда обмена	247

15.9.2. Арифметические операции	248
Основные арифметические операции.....	248
Операции над знаковым битом	249
Округление до целого.....	250
Получение остатка от деления.....	251
Извлечение корня	252
Масштабирование	252
Операции сравнения и тестирования	253
15.9.3. Трансцендентные операции.....	254
Тригонометрические операции	254
Возведение в степень.....	255
15.9.4. Команды управления.....	258
15.10. Практикум	259

ЧАСТЬ III. УПРАВЛЕНИЕ ВНЕШНИМИ УСТРОЙСТВАМИ, ПРЕРЫВАНИЯ, РЕЗИДЕНТНЫЕ ПРОГРАММЫ 263

Глава 16. Управление внешними устройствами 265

16.1. Внешние устройства в программной модели вычислительной системы.....	265
16.2. Инструкции для доступа к портам	266
16.3. Исследование внешних устройств вручную	268
16.3.1. Запуск редактора портов	268
16.3.2. Определение наличия устройства	268
16.3.3. Доступ к регистрам устройства	269
Коммутация по чтению-записи.....	269
Управление коммутацией через отдельный порт	270
16.3.4. Управление устройствами.....	274
Принцип последовательной связи	274
Основные регистры адаптера последовательной связи.....	275
Последовательная передача в диагностическом режиме	276
Получение ошибки переполнения приемника	277
16.4. Управление устройствами по программе	277
16.4.1. Наблюдение за состоянием в режиме периодического опроса	278
16.4.2. Реакция на особые состояния в режиме прерываний	280
Задание адреса перехода при прерывании	280
Выполнение прерывания процессором i80x86.....	281
Контроллер прерываний 8259A	282
Пример организации прерываний от внешнего источника	284
Каскадное включение контроллеров прерываний.....	287
16.5. Измерение временных характеристик устройства	287

Глава 17. Прерывания и исключения 290

17.1. Прерывания и векторные вызовы подпрограмм.....	290
17.2. Типы исключений	293
17.3. Итоговая классификация прерываний.....	294
17.4. Программирование исключений.....	295
17.4.1. Trap-исключения.....	295

17.4.2. Fault-исключения.....	297
17.4.3. Различие между fault- и trap-исключениями	299
17.4.4. Практикум по trap- и fault-исключениям.....	300
17.5. Обработка прерываний при наличии системной процедуры	302
17.5.1. Сохранение и восстановление векторов.....	302
17.5.2. Дополнение к установленной процедуре обработки прерывания	306
17.6. Внешние прерывания.....	308
17.6.1. Доступ к данным из процедуры обработки прерывания.....	308
17.6.2. Ограничения на использование функций операционной системы.....	311
17.6.3. Определение причины прерывания	312
17.6.4. Практикум по внешним прерываниям	315
Прерывание от клавиатуры	315
Прерывание от последовательного канала связи.....	317
Глава 18. Резидентные программы	319
18.1. Установка резидентной процедуры	320
18.2. Взаимодействие с TSR-программой по данным	323
18.3. Уменьшение размера занимаемой памяти.....	327
18.4. Выгрузка TSR-программы	327
18.5. Вызовы DOS в TSR-процедурах	335
18.5.1. Способы определения состояния DOS	335
18.5.2. Pop-up программы	337
ЧАСТЬ IV. ПРИЛОЖЕНИЯ	343
Приложение 1. Биты, байты, слова, знаковые и беззнаковые числа.....	345
Приложение 2. Коды литер в стандарте ASCII	351
Приложение 3. Позиционные коды клавиш	354
Приложение 4. Функции BIOS-DOS	357
Функции BIOS для работы с клавиатурой.....	357
Функции прерывания 021 DOS	358
Функции низкоуровневого ввода/вывода.....	358
Функции ввода	358
Функции вывода.....	358
Функции для работы с файлами и потоками.....	359
Стандартные потоки ввода/вывода	359
Функции чтения и записи	359
Функции для создания, открытия и закрытия файлов	360
Функции для завершения программы.....	362
Функции для TSR-программ.....	362
Приложение 5. Настройки запуска a86	364
Ключи, или опции запуска a86.....	365
Переменная окружения a86.....	368

Приложение 6. Операторы и инструкции a86	369
Операторы	369
Инструкции.....	371
Условные обозначения.....	371
Общие правила установки флагов	372
Воздействие команд на флаги.....	373
AAA — Ascii Adjust after Addition	374
AAD — ASCII Adjust before Division	375
AAM — ASCII Adjust after Multiply	376
AAS — ASCII Adjust after Subtraction.....	377
ADC — Add with Carry	378
ADD — Addition.....	379
AND — Logical AND.....	379
BOUND — Check Array Index Against Bounds.....	380
CALL — Call Procedure	380
CBW — Convert Byte to Word	383
CLC — Clear Carry Flag.....	383
CLD — Clear Direction Flag	383
CLI — Clear Interrupt Flag.....	384
CMC — Complement Carry Flag	384
CMP — Compare.....	384
CMPS/CMPSB/CMPSW — Compare String Operands.....	385
CWD — Convert Word to Doubleword	387
DAA — Decimal Adjust after Addition	387
DAS — Decimal Adjust after Subtraction	388
DEC — Decrement by 1	389
DIV — Unsigned Divide	389
ENTER — Make Stack Frame for Procedure Parameters	390
IDIV — Signed Divide	391
IMUL — Signed Multiply	393
IN — Input from Port	393
INC — Increment by 1	394
INS/INSB/INSW — Input from Port to String.....	394
INT/INTO — Call to Interrupt Procedure	396
IRET — Interrupt Return.....	397
J<x> — Jump if Condition is Met.....	397
JMP — Jump	399
LAHF — Load Flags into AH Register	401
LEA — Load Effective Address	401
LEAVE — High Level Procedure Exit	402
LDS/LES — Load Far Pointer	402
LODS/LODSB/LODSW — Load String Operand.....	403
LOOP/LOOP<x> — Loop Control with CX Counter.....	404
MOV — Move Data	405
MOVS/MOVSb/MOVSsw — Move Data from String to String.....	405
MUL — Unsigned Multiplication of AL or AX.....	407
NEG — Two's Complement Negation	408

NOP — No Operation.....	408
NOT — One's Complement Negation.....	409
OR — Logical OR.....	409
OUT — Output to Port.....	410
OUTS/OUTSB/OUTSW — Output String to Port.....	410
POP — Pop a Word from the Stack.....	412
POPA — Pop all General Registers.....	412
POPF — Pop Stack into FLAGS.....	413
PUSH — Push Operand onto the Stack.....	413
PUSHA — Push all General Registers.....	414
PUSHF — Push Flags Register onto the Stack.....	414
REP/REP<x> — Repeat Following String Operation.....	415
RET — Return from Procedure.....	416
RCL/RCR/ROL/ROR — Rotate.....	417
SAHF — Store AH into Flags.....	418
SAL/SAR/SHL/SHR — Shift Instructions.....	418
SBB — Subtraction with Borrow.....	419
SCAS/SCASB/SCASW — Scan String Data.....	419
STC — Set Carry Flag.....	421
STD — Set Direction Flag.....	421
STI — Set Interrupt Flag.....	421
STOS/STOSB/STOSW — Store String Data.....	422
SUB — Subtraction.....	423
TEST — Logical Compare.....	423
WAIT — Wait until BUSY# Pin is Inactive.....	424
XCHG — Exchange Register/Memory with Register.....	424
XLAT/XLATB — Table Look-up Translation.....	425
XOR — Logical Exclusive OR.....	425

Приложение 7. Совместимость a86 с традиционными ассемблерами..... 426

Приложение 8. Прерывания от i80x87..... 431

Синхронизация процессора и сопроцессора.....	431
Обработка прерываний.....	432
Подключение сигнала прерывания.....	434
Внешнее прерывание через irq13.....	434
Внутреннее прерывание 16.....	435

Приложение 9. Ответы на контрольные вопросы из части I..... 436

К разделу 1.2.....	436
К разделу 1.4.....	436
К разделу 1.7.....	437
К разделу 2.1.....	439
К разделу 2.5.....	440
К разделу 5.1.....	441
К разделу 5.3.2. Прямая адресация.....	441
К разделу 5.3.2. Косвенная адресация.....	442

К разделу 5.3.3	442
К разделу 7.4.2	442
К разделу 10.1	443
К разделу 10.5	444
Приложение 10. Ошибки в a86 v4.05	445
Ошибка при оптимизации инструкции <i>call far</i>	445
Ошибка в операторе <i>bit</i>	445
Приложение 11. Описание дискеты	446

Введение

Ассемблер — это *язык* символического кодирования машинных инструкций, адресов и данных, а также *транслятор* с этого языка.

Главная особенность языка ассемблера — то, что в нем непосредственно отражены инструкции процессора и организация памяти. Так, например, инструкция ассемблера

```
inc ax
```

обозначает машинную команду "увеличить содержимое регистра процессора *ax* на единицу". Каждая машинная команда в программе кодируется отдельной инструкцией ассемблера.

Языки ассемблера различны, поскольку системы команд разных вычислительных систем не совпадают, а для аналогичных команд могут быть приняты разные обозначения.

Например, в процессорах семейства i80x86 имеется инструкция умножения, а в микроконтроллерах серии i8048 — нет. Соответственно, инструкции *mul*, *imul* ассемблера для i80x86 не имеют аналогов в ассемблере для i8048. Напротив, команда записи числа 10 в младший байт регистра-аккумулятора имеется в обоих семействах процессоров, но в ассемблерах обозначается по-разному:

```
mov    al, 10      mov    a, #10
```

Аналогичные команды в ассемблерах для микропроцессоров i8080 и i8085:

```
mov    #10, r0     ldi     a, 10
```

"Стандарт" ассемблера ограничен одним семейством микропроцессоров; он устанавливается авторами процессора и первого ассемблера для него.

Вместе с тем, ассемблеры — даже для разных вычислительных систем — во многом похожи, поскольку во всех процессорах имеется набор типовых инструкций — пересылка, арифметико-логические операции, переходы. Кро-

ме того, в любой современной вычислительной системе память образуется наборами 8-битных байтов.

Эта книга знакомит читателя с языком ассемблера для одного из наиболее развитых микропроцессоров — i80x86. После изучения ассемблера для i80x86 вы сможете самостоятельно осваивать ассемблеры для произвольных вычислительных систем.

Ассемблер a86 разработан автором asm86 — первого "официального" ассемблера для i8086. Ассемблер a86 следует "стандарту" `tasm`, но гораздо проще в обращении и содержит меньше ошибок. Особенности ассемблера a86 и отладчика d86 позволили нам представить в этой книге уникальные практики по программированию данных, освоению системы команд процессора i80x86, изучению форматов данных и системы команд арифметического сопроцессора i80x87.

После освоения a86 переход на более распространенные ассемблеры `tasm` или `tasm` не представляет сложности.

Структура изложения

Книга состоит из четырех частей.

Часть I, с изложением базовых средств и принципов ассемблера, включает в себя 10 глав. В них рассматриваются: определение данных, имен и типов, методы адресации, система команд i80x86, программирование циклов, сегментная модель памяти и `exe`-программы.

Часть II, с изложением дополнительных возможностей языка ассемблера, состоит из пяти глав (11—15). В них рассмотрены: макрокоманды и средства условной трансляции, структурное программирование на ассемблере, интерфейс с языком C, обработка данных BCD-формата, форматы данных и система команд арифметического сопроцессора i80x87.

В части III (главы 16—18) изучаются: основы управления внешними устройствами, обработка прерываний и исключений, резидентные программы.

Часть IV содержит дополнительный материал и состоит из 11 приложений.

В приложении 1 приведены отдельные сведения из базового курса информатики, помогающие в изучении материала первой главы.

Приложения 2 и 3 содержат сведения, имеющие отношение к вводу/выводу символьной информации.

В приложении 4 дан краткий перечень функций операционной системы, используемых в примерах.

В приложении 5 рассмотрены настройки запуска ассемблера a86.

Приложение 6 представляет собой краткий справочник по операторам ассемблера и системе команд i80x86.

В приложение 7 рассмотрены вопросы совместимости с традиционными ассемблерами.

Приложение 8 посвящено организации аппаратных прерываний от арифметического сопроцессора i80x87.

В приложении 9 содержатся ответы на вопросы из первой части, выборочно.

В приложении 10 приведен перечень ошибок a86 v4.05, обнаруженных до апреля 2002 г.

К книге прилагается дискета с пакетом a86 v4.05 (2000 г.) и исходными текстами примеров. Описание дискеты дано в приложении 11.

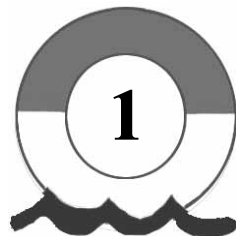


Часть I

Основы ассемблера

- Глава 1.** Данные, имена, типы
- Глава 2.** Определение имен
- Глава 3.** Практикум по программированию данных
- Глава 4.** Просмотр данных в отладчике
- Глава 5.** Способы адресации
- Глава 6.** Система команд i80x86
- Глава 7.** Программирование циклов
- Глава 8.** Исследование программ в d86
- Глава 9.** Примеры программ
- Глава 10.** Сегменты и ехе-программы

ГЛАВА 1



Данные, имена, типы

Я вас научу, с какого конца редьку есть!

Из к/ф "Сказ про то, как царь Петр арапа женил"

Основные вопросы первых двух глав — определение *данных*, определение и использование *имен*, понятие *типа* имени. Мы начинаем знакомство с ассемблером с данных, имен и типов потому, что эти объекты и понятия, с одной стороны, составляют существенную часть языка и, с другой стороны, в наименьшей степени зависят от системы команд микропроцессора.

1.1. Структура программы

Предварительно укажем местоположение данных в исходном тексте программы. Структура исходного текста программы на языке ассемблера а86, для получения исполняемой программы com-формата, следующая:

```
jmp start      ; первая исполняемая инструкция -
                ; команда обхода области данных
...
...           ; директивы определения данных
...
start:         ; начало программного кода
...
...           ; инструкции
...
ret           ; завершающая инструкция "возврат из
                ; подпрограммы", для выхода в DOS
```

Программа представляет собой последовательность операторов, по одному на строке.

В качестве оператора может быть задана:

- ☐ инструкция ассемблера, т. е. машинная инструкция в символическом виде;
- ☐ директива определения данных;
- ☐ директива ассемблера, т. е. команда для управления трансляцией.

Перед инструкцией — в той же строке — может быть поставлена метка (например, `start:`); аналогично, перед директивой определения данных можно поставить уникальное имя, которое в дальнейшем обозначает адрес данных в символическом виде. Литера ";" открывает комментарий — до конца строки.

В редких случаях — обычно, в учебниках — программа состоит только из инструкций или, еще реже (как в начале этой книги), из одних данных. (В таких случаях команда обхода данных не нужна.) В начале знакомства с ассемблером мы ограничиваемся данными и именами для их определения, составляя неисполняемые программы, без инструкций.

1.2. Директивы определения данных

Директивы `db`, `dw` и `dd` резервируют соответственно байты, 2-байтные слова и 4-байтные двойные слова. После оператора `db` (или `dw`, или `dd`) записывается значение элемента данных; знаком вопроса обозначается произвольная величина (неопределенное исходное значение элемента данных).

Примеры:

```
dw    10           ; 2-байтное слово со значением 10
dd    ?            ; двойное слово с произвольным значением
db    '?'          ; байт со значением ASCII-кода вопроса
db    '8'          ; байт со значением ASCII-кода '8'
db    1            ; байт со значением 1
db    1            ; еще один байт со значением 1
db    1            ; и еще один байт со значением 1
```

Следующие друг за другом операторы `db` (или `dw`, или `dd`) можно записать в одну строку, разделив значения запятыми:

```
db    '?', '8', 1, 1, 1
```

Обозначения ASCII-символов в директивах `db` можно сгруппировать, задав их одной строкой в кавычках:

```
db    '?8', 1, 1, 1
```

Повторяющиеся элементы внутри оператора `db` (или `dw`, или `dd`) группируются при помощи конструкции `dup`:

```
db    '?', '8', 3 dup 1
```

Конструкции `dup` допускают вложенность. Например, директива

```
db 2 dup (1, 3 dup 0)
```

задает двукратное повторение последовательности `(1, 3 dup 0)`:

```
db 1, 3 dup 0, 1, 3 dup 0
```

В итоге, эта директива означает:

```
db 1, 0, 0, 0, 1, 0, 0, 0
```

В табл. 1.1 приведены диапазоны чисел в машинном представлении в зависимости от размерности данных. Диапазоны *знаковых* и *беззнаковых* чисел в табл. 1.1 объединены, поэтому результирующие диапазоны несимметричны. Например, один байт представляет знаковое число в пределах от -128 до $+127$ и/или беззнаковое число от 0 до 255 ; результирующий диапазон — от -128 до 255 .

Таблица 1.1. Диапазоны данных

Размерность	Диапазон
Байт	$-128 - +255$
Слово	$-32\,768 - +65\,535$
Двойное слово	$-2\,147\,483\,648 - +4\,294\,967\,295$

Примечание

Один и тот же набор бит могут представлять разные значения в зависимости от того, считается ли оно знаковым или беззнаковым. Так, например, последовательность бит `11111111` задает беззнаковое число 255 и, в то же время, знаковое число -1 .

Слово можно задать по байтам: сначала младший байт, затем старший. Аналогично, двойное слово может быть определено парой смежных слов, или четверкой байт по последовательным адресам.

Примечание

В архитектуре Intel принято, что наиболее значимая часть числа из двух байтов/слов находится в байте/слове с *наибольшим* адресом.

В качестве примера приведем варианты определения слова со значением 256 и двойного слова со значением $65\,539$:

```
dw 256
dw 0100 ; 256
db 0, 1
```

```
dd    010003    ; 65539
dw    3, 1
db    3, 0, 1, 0
```

Контрольные вопросы

1. Запишите без использования `dup` директиву определения данных:
`dw 3 dup (2 dup 5, 7), 9`
2. Задайте одной директивой (без использования `dup`, сгруппировав ASCII-коды) следующие данные:
`db 'AB'`
`db '1', 2 dup (3 dup '?', 'x'), '0'`
3. Задайте с использованием вложенных конструкций `dup` следующие данные:
`dd 6, 1, 0, 1, 0, 5, 1, 0, 1, 0, 5`
4. Определите десять байт и десять слов с произвольными (неопределенными) значениями.
5. Запишите директиву определения последовательности байт со значениями 'A', 'B', 'C', 0, 'D', 'E' сгруппировав обозначения ASCII-кодов.
6. Задайте последовательность данных: 30 байт со значениями 100, два слова — со значениями -5 и 19, двойное слово со значением 1000000.
7. Запишите директиву определения 12 байт со значениями 256.
8. Задайте одной директивой `db` следующую последовательность данных: пять слов со значением 1, четыре байта со значением 1, пять слов со значением 1, четыре байта со значением 1, и т. д. всего 100 раз.

1.3. Обозначение чисел

Числа могут быть заданы в двоичной, восьмеричной, десятичной и шестнадцатеричной системе счисления, а также кодами ASCII (литера в кавычках).

По умолчанию действует десятичная система счисления. Если первая цифра числа ноль, число считается шестнадцатеричным. Суффиксы `b`, `q`, `d`, `h` позволяет изменить систему счисления для одного (текущего) числа — на двоичную, восьмеричную, десятичную, шестнадцатеричную соответственно.

Внимание!

Если первая цифра 0, суффикс `d` или `b` воспринимается как шестнадцатеричная цифра.

Примеры (в скобках справа показаны десятичные значения):

```
14      — десятичное число          (14)
012     — шестнадцатеричное число   (18)
```

12h	— шестнадцатеричное число	(18)
11b	— двоичное число	(3)
011q	— восьмеричное число	(3)
011b	— шестнадцатеричное число	(283)
11d	— десятичное число	(11)
011d	— шестнадцатеричное число	(285)
ah	— имя	
0ah, 0a	— шестнадцатеричные числа	(10)

Рекомендуется взамен суффиксов `b` и `d` применять аналогичные суффиксы `xB` и `xD` — ассемблер не перепутает их с шестнадцатеричной цифрой.

Примеры:

1011, 01011xD	— десятичное
01011, 1011h	— шестнадцатеричное
1011q, 1011Q	— восьмеричное
01011xB	— двоичное
041	— ASCII-код буквы A
'A'	— ASCII-код буквы A
13	— ASCII-код служебного символа 'возврат каретки'
10	— ASCII-код служебного символа 'перевод строки'

Числа, кратные 1024, можно обозначать с использованием суффикса `k`. Суффикс `k` — это коэффициент 1024; число перед `k` — десятичное. В качестве примера приведем варианты определения 4 Кбайт данных:

```
db 4096 dup ?
db 4k dup ?
dw 2k dup ?
dd 1k dup ?
```

1.4. Символические обозначения чисел, выражения

Смысл введения символов для обозначения чисел — такой же, как и в языках высокого уровня: упростить корректировку параметров программы, сделать исходный текст понятней.

Символ для обозначения числа вводится директивой:

```
<name> equ <const>
```

где `<name>` — имя, составленное по общим правилам (начинается с буквы или литеры подчеркивания, за которой может следовать любое количество букв, литер подчеркивания и цифр), `<const>` — это число или *выражение*, составленное из чисел и/или символов, обозначающих числа.

Примеры:

```
_dozen    equ 12
size2     equ 200
sign_bit  equ 1000000xb
max_byte  equ 255
```

Операции, допустимые в выражениях, приведены в табл. 1.2.

Таблица 1.2. Операции, допустимые в выражениях

Операция	Действие или результат	Очередность
()	Изменение очередности операций	1
bit n	Число, в котором n-й бит установлен в 1, а остальные сброшены (биты нумеруются от нуля)	
high n	Значение старшего байта числа n	2
low n	Значение младшего байта числа n	
k / n	Целая часть результата деления k на n	
k mod n	Остаток от деления k на n	
k * n	Произведение чисел k и n	3
k shl n	Сдвиг двоичного кода числа k на n бит влево	
k shr n	Сдвиг значений разрядов числа k вправо на n бит	
k + n	Сумма чисел k и n	4
k - n	Разность чисел k и n	
not k	Поразрядная инверсия числа k	5
k and n	Поразрядное логическое умножение чисел k и n	6
k or n	Поразрядное логическое сложение чисел k и n	7
k xor n	Поразрядное исключающее "или" чисел k и n	
*k by n	Число, старший байт которого равен k, а младший n	8

Примечание

После знака операции +, -, / или * число с минусом следует задавать в скобках, например: -4*(-2). К сожалению, в этой ситуации транслятор a86 не считает отсутствие скобок ошибкой. В результате, выражение -4*-2, оказывается, равно -2.

Примеры:

```

alfa      equ 10 / 4                ; 2
beta      equ low (alfa * 083)      ; 6
gamma     equ low alfa * 083        ; 0106
          db 1 shl 5                 ; 100000xb
          db bit 5                   ; 100000xb
          db not bit 0               ; 11111110xb
          db bit 1 + bit 3           ; 1010xb
          dw beta + 1 dup 'A' by 10

```

Последняя директива задает семь (`beta+1`) слов со значением ASCII-кода литеры 'A' в старшем байте и числом 10 в младшем байте.

Контрольные вопросы

1. Запишите директиву определения десяти кило-слов данных, каждый *байт* которых содержит ASCII-код литеры '0'.
2. Запишите число, в котором установлен пятый разряд, двумя способами — при помощи суффикса `xb` и при помощи конструкции `bit n`.
3. Запишите десятичные числа, соответствующие обозначениям:
`015`, `15`, `11b`, `17q`, `0a`, `bit 3`, `'A'`
4. Определите константу `hours`, равную произведению числа дней в году на число часов в сутках. Используя имя `hours`, определите слово со значением числа часов в году; затем, по-прежнему используя `hours`, задайте двойное слово со значением количества минут в году, и двойное слово со значением числа секунд в году.
5. Определите константу `len` со значением 119, затем определите слово со значением площади квадрата со стороной `len`.
6. Определите константу `hip` со значением 314, затем определите слово со значением площади прямоугольного равнобедренного треугольника со сторонами `hip`.
7. При помощи операций `bit` и операции `+` или `or` определите константу, содержащую единицы в третьем и седьмом разрядах; определите константу с другим именем, но с тем же значением, используя вместо `bit` операцию `shl`.
8. При помощи операций `bit`, операции `+` или `or`, а также операции `not` определите байт, в котором сброшены нулевой и третий разряды.

1.5. Переменные и метки

Пользовательские имена — это имена, определенные в тексте программы. Пользовательские имена применяются для обозначения адресов и констант взамен чисел. Рассмотренные в *разд. 1.4* символические обозначения чисел — это один из вариантов определения и применения пользовательских имен.

Адреса в символьной форме обозначаются именами *переменных* и *меток*.

Переменная — это символическое обозначение адреса *первого байта* (слова/двойного слова), определенного директивой `db` (`dw/dd`). Имя переменной задается перед директивой `db` (`dw/dd`); в дальнейшем это имя может быть использовано для обозначения адреса данных в *операндах инструкций*.

Метка — это символическое обозначение адреса *инструкции*. Метка определяется в начале строки как имя с двоеточием. Имя метки может использоваться для задания адреса перехода в командах *передачи управления*.

Примеры:

```

        jmp  start           ; использование метки start
                               ; в инструкции перехода

area1   db    2, -8, 7       ; определение переменной area1
        dw   -1
area2   dw    2, -8, 7       ; определение переменной area2

start:                               ; определение метки start
        inc  area1
        inc  area2+2
        inc  area2-2
        ...

```

Имя `area1` обозначает адрес *первого байта* из трех байт, определенных директивой `db`. Эти байты можно адресовать как `area1`, `area1+1`, `area1+2` (слагаемое задает смещение в байтах). Аналогично, слова, определенные директивой `dw`, можно обозначать как `area2`, `area2+2`, `area2+4` (смещение — вдвое, т. к. каждое слово занимает два байта).

Мнемоника `inc` обозначает инструкцию инкремента (увеличение на 1). Первая из таких инструкций задает инкремент байта по адресу `area1`; после ее выполнения значение байта равно трем. Вторая инструкция задает инкремент слова, исходное значение которого равно `-8`. Выполнение третьей инструкции увеличивает слово с начальным значением `-1`.

Внимание!

Обозначения `area1+3` и `area2-2` неэквивалентны (хотя это один и тот же адрес), поскольку `area1+/-<n>` адресует *байт*, а `area2+/-<n>` — *слово*.

Выводы из рассмотренного примера:

- ❑ с именем переменной связан *тип*: переменная обозначает либо адрес байта, либо адрес слова, либо адрес двойного слова — в зависимости от директивы — `db`, `dw` или `dd`, перед которой определено имя;
- ❑ одна и та же *мнемоника инструкции* может задавать операцию над байтом, словом или двойным словом — в зависимости от типа переменной.

1.6. Типы имен

Тип символьных констант и меток — `abs`. Типы переменных — `byte ptr` (сокращенно `byte`), `word ptr` (сокращенно `word`), `dword ptr` (сокращенно `dword`); `ptr` значит `pointer` — указатель.

Тип позволяет одинаково обозначать операцию над байтом, словом или двойным словом. Так, например, обозначение `inc` задает операцию увеличения на 1. Чего именно — байта, слова или двойного слова? Это следует из типа операнда.

Тип назначается автоматически при определении имени. Явное указание типа при помощи обозначений `abs`, `byte`, `word`, `dword` используется в `extrn`-описаниях (см. *разд. 9.5 и 10.5*); обозначения `byte`, `word`, `dword` можно применять для временного изменения типа переменной. Временное преобразование к типу `abs` выполняется оператором `offset`.

Пример с именами, определенными в *разд. 1.5*:

```
inc word ptr areal      ; (1)
inc word areal          ; (2)
inc byte area2+2        ; (3)
inc start               ; (4)   ?!
inc dword start         ; (5)
mov ax, area2           ; (6)
mov ax, offset area2    ; (7)
```

Инструкции (1) и (2) задают инкремент *слова* по адресу `areal`, несмотря на то, что `areal` обозначает адрес *байта*. Аналогично, инструкции (3) и (5) задают инкремент байта и двойного слова, независимо от типа имени; обратите внимание, что выполнение (5) приведет к искажению кода инструкции по адресу `start`. Инструкция (6) задает копирование *содержимого* слова по адресу `area2` в регистр `ax` процессора (значение `ax`, в результате, станет равно двум). Напротив, инструкция (7) задает запись в `ax` значения *адреса* `area2`.

Инструкция (4) воспринимается как ошибка, поскольку тип операнда — `abs`. Этот тип может обозначать либо константу, либо адрес инструкции. Константу нельзя изменить по определению, а модификация машинной команды — это либо ошибка, либо сомнительный программистский трюк. В (5) последнее ограничение обошли за счет временного преобразования типа.

Преобразования типов действуют временно — в пределах текущего обозначения имени. Так, например, имя `areal` сохраняет тип `byte`, несмотря на преобразование типа в инструкциях (1) и (2).

1.7. Типы и выражения

Рассмотрим правила *совместимости типов* в выражениях и правило определения типа *результата выражения*.

Правила совместимости:

1. Над именами типа `abs` выполнимы все операции, предусмотренные в выражениях (напоминаем, что типом `abs` характеризуются числа и метки).
2. Имя переменной (тип `byte`, `word`, `dword`) в выражении допускается только справа и/или слева от знаков `+` и `-`. (Имя переменной обозначает начальный адрес данных, определенных директивой `db/dw/dd`.)

Правило определения типа результата: результат операции `-` или `+` принимает тип *левого* операнда.

Исключение: результат операции `+` с одной переменной всегда получает тип этой *переменной*.

В следующем примере, где иллюстрируется правило определения результата, использован оператор `type <name>`. Этот оператор возвращает количество байт, которое занимает переменная `<name>`:

- ❑ 1, если переменная типа `byte`;
- ❑ 2, если `word`;
- ❑ 4, если `dword`;
- ❑ 0, если `<name>` — константа или метка (тип `abs`).

Пример:

```

byt1 db 4 dup ?
word1 dw ?

obj1 equ type (byt1 - 1)           ; 1 - "byte"
obj2 equ type (word1 - 1)          ; 2 - "word"
obj3 equ type (1 - byt1)           ; 0 - "abs"
obj4 equ type (1 - word1)          ; 0 - "abs"
obj5 equ type (word1 - byt1)        ; 2 - "word"
obj6 equ type (byt1 - word1)        ; 1 - "byte"
obj7 equ 2 - word1                  ; "abs"

_obj1 equ type (byt1 + 1)           ; 1 - "byte"
_obj2 equ type (word1 + 1)          ; 2 - "word"
_obj3 equ type (1 + byt1)           ; 1 - "byte"
_obj4 equ type (1 + word1)          ; 2 - "word"
_obj5 equ type (word1 + byt1)        ; 2 - "word"
_obj6 equ type (byt1 + word1)        ; 1 - "byte"
_obj7 equ 2 + word1                  ; "word"
```

```
var1    dw    byte1, 10 dup word1
var2    dw    offset byte1, 10 dup offset word1
var3    db    offset (word1 - byte1) dup ?
var4    db    offset word1 - offset byte1 dup ?
```

Обратите внимание: в списке значений, следующем за директивой `dw` (в частности, после `dup`), допускаются имена любого типа. Если указана переменная, то элемент данных принимает значение адреса; таким образом, `<var>` и `offset <var>` в качестве инициализирующих значений — это одно и то же.

Внимание!

Директива `equ` предназначена не только для задания символических констант. В общем случае, директива `<name> equ <exp>` вводит новое имя `<name>`, со значением и *типом* выражения `<exp>`. Так, имя `_obj7` в примере определено — при помощи `equ` — как переменная типа `word` с адресом на два больше, чем адрес `word1`.

Примеры ошибок и неточностей:

```
var5    db    (word1 - byte1) dup ?
var6    db    word1
var7    db    offset byte1
```

Ошибка — в первом операторе: счетчик повторов в конструкции `dup` должен быть типа `abs`. В следующих двух операторах допущена неточность: старший байт 16-разрядного адреса переменной будет отброшен (вероятно, с потерей значащих разрядов).

Контрольные вопросы

1. Перечислите имена и их типы, определенные в следующем фрагменте:

```
noodle   equ   '0' by 0
needle   db    40k / 7 dup 0
al       dw    offset needle - 0100
```

2. Найдите ошибки в нижеприведенном фрагменте:

```
src       db    "You're genius
dest      dw    bit 16 + offset src
size      equ   (dest - src) / 2
```

3. Данные определены таким образом:

```
byte1     db    1, 2
word1     dw    03040, 05060
```

Какие значения получают данные после выполнения последовательности инструкций:

```
inc    byte1
inc    byte1+1
inc    byte1+2
inc    word1
inc    word1+1
```

4. Данные определены таким образом:

```
word1    dw    03040, 3 dup 0, 0ffff
```

Запишите инструкцию, выполнение которой увеличит на единицу слово со значением -1 .

5. Данные определены следующим образом:

```
word1    db    3k dup 0, 0ff
byte1    dw    ?
```

Запишите инструкцию, выполнение которой увеличит на единицу последний байт, заданный директивой `db`, а также инструкцию, выполнение которой инкрементирует последнее *слово* из последовательности, заданной директивой `db`. (Попробуйте адресовать эти данные, используя имя `byte1`.)

6. Данные заданы следующим образом:

```
abcd     db    'ABCD'
digit    db    '0123456789'
```

Какими станут значения данных после выполнения инструкций:

```
inc      abcd
inc      abcd+6
inc      digit-3
```

7. Данные заданы следующими объявлениями:

```
byte1    db    -2, 2, 3
```

Запишите команду для инкремента слова со значением `02fe` и команду для инкремента слова со значением `0302`.

8. Данные заданы следующими объявлениями:

```
dword1    dd    0102ff04
```

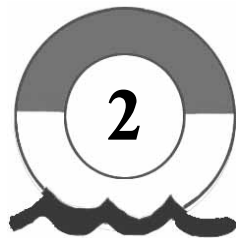
Запишите команду, которая увеличивает на 1 слово со значением `02ff`, входящее в заданное двойное слово. Запишите команду, которая увеличивает на 1 байт со значением 4 в заданном двойном слове.

9. Данные заданы следующими объявлениями:

```
areal     db    (080 shr 3) dup ?
area2     dw    (offset area2 - offset areal) dup ?
dummy     db    ?
size      equ    offset dummy - offset areal
```

Чему равна константа `size`?

ГЛАВА 2



Определение имен

В этой главе рассматриваются различные вопросы, имеющие отношение к определению и использованию имен, и в первую очередь, механизм определения имен при трансляции.

2.1. Алгоритм трансляции

Транслятор считывает исходный текст по строкам, от начала до конца дважды. При первом просмотре транслятор обнаруживает имена, введенные пользователем (т. е. константы, метки, переменные), для каждого имени определяет атрибуты (числовое значение и тип) и записывает эту информацию в *таблицу имен*. При втором просмотре транслятор определяет значения *данных* и *кодов инструкций* и записывает их в выходной файл.

На обоих просмотрах используется *счетчик адресов* — для резервирования памяти. В начале каждого просмотра этот счетчик устанавливается равным некоторому стартовому значению (например, 0100 при трансляции *com-программ*). При разборе инструкции или директивы определения данных счетчик адресов увеличивается на число байтов, занимаемое *машинным представлением* инструкции или данных. Так, при помощи счетчика адресов, транслятор отслеживает распределение памяти в результирующей программе.

Каждой *метке* или *переменной*, обнаруженной на первом просмотре, присваивается текущее значение счетчика адресов.

Текущее значение счетчика адресов на ассемблере обозначается значком \$ (или ключевым именем *this*). Обозначение \$ допускается в выражениях наряду с именами и числами; считается, что \$ имеет тип *abs*.

Рассмотрим алгоритм трансляции на примере программы, которая содержит исключительно данные:

```
start:                                     ; (1)
msg    db    'Very loooooong taaaall Saaally!' ; (2)
size   equ   $-msg                         ; (3)
       db    size dup ?                    ; (4)
       dw    offset zz - start              ; (5)
zz     db                                         ; (6)
xx     equ   msg + 1                         ; (7)
```

В начале программы определена метка `start`. (Используется в операторе (5) для вычисления объема машинного кода программы.) Имя `start` записывается в таблицу имен с числовым значением 0100 и типом `abs`. После просмотра строки (1) значение `$` не изменилось, поскольку в этой строке не задано ни данных, ни инструкции — только имя. Содержимое таблицы имен после трансляции строки (1) показано в табл. 2.1.

Таблица 2.1. Таблица имен после трансляции строки (1)

Имя	Значение	Тип	Значение счетчика адресов
start	0100 (256)	abs	\$ = 256

В строке (2) транслятор обнаруживает имя `msg` и записывает его в таблицу со значением 0100 и типом `byte`. В этой строке задана директива определения 30 байт, поэтому `$` увеличивается на 30 и становится равным 286. Содержимое таблицы имен после трансляции строки (2) показано в табл. 2.2.

Таблица 2.2. Таблица имен после трансляции строки (2)

Имя	Значение	Тип	Значение счетчика адресов
start	0100 (256)	abs	
msg	256	byte	\$ = 286

В строке (3) пользователь определил константу `size` — в виде разности между текущим значением счетчика адресов и значением имени `msg` (таким образом, значение `size` равно $286 - 256 = 30$). Обратите внимание, величина `size` равна числу байт, заданных предыдущей директивой `db`. Имя `size` записывается в таблицу со значением 30 и типом `abs`. Содержимое таблицы имен после трансляции строки (3) показано в табл. 2.3.

В строке (4) определена последовательность `size` байт; значение `size` берется из таблицы. В результате, оказывается, что объем данных, заданных строками (2) и (4), одинаков. Счетчик адресов увеличивается на 30 и становится равным 316. Таблица имен не изменяется.

Таблица 2.3. Таблица имен после трансляции строки (3)

Имя	Значение	Тип	Значение счетчика адресов
start	0100 (256)	abs	
msg	256	byte	
size	30	abs	\$ = 286

В строке (5) задано определение слова. Счетчик адресов увеличивается на 2 и становится равным 318. В таблицу имен ничего не записывается.

Примечание

Значение слова, заданного строкой (5), при первом просмотре вычислить невозможно, поскольку выражение `zz – start` содержит ссылку на еще не определенное имя `zz` (так называемая опережающая ссылка). Транслятор предполагает, что имя `zz` определено дальше; если это не так, при втором просмотре будет зафиксирована ошибка.

Предпоследняя строка программы содержит определение `zz` и не задает данных. Значение `$` не изменяется, в таблицу имен записывается информация о `zz`. Содержимое таблицы имен после трансляции строки (6) показано в табл. 2.4.

Таблица 2.4. Таблица имен после трансляции строки (6)

Имя	Значение	Тип	Значение счетчика адресов
start	0100 (256)	abs	
msg	256	byte	
size	30	abs	
zz	318	byte	\$ = 318

В последней строке программы задано определение `xx`. Напомним, что оператор `equ` создает новое имя и устанавливает его тип и значение равными типу и значению выражения. Выражение `msg + 1` вычисляется по информации из таблицы имен; в результате, создается имя `xx` со значением 257 и типом `byte`. Содержимое таблицы имен после трансляции строки (7) показано в табл. 2.5.

После завершения первого просмотра `a86` записывает в таблицу специальное имя `end`, присваивая ему конечное значение счетчика адресов. (Можно считать, что метка `end` всегда неявно задана в конце исходного текста). Содержимое таблицы имен к началу второго просмотра показано в табл. 2.6.

Таблица 2.5. Таблица имен после трансляции строки (7)

Имя	Значение	Тип	Значение счетчика адресов
start	0100 (256)	abs	
msg	256	byte	
size	30	abs	
zz	318	byte	
xx	257	byte	\$ = 318

Таблица 2.6. Таблица имен в конце первого просмотра

Имя	Значение	Тип	Значение счетчика адресов
start	0100 (256)	abs	
msg	256	byte	
size	30	abs	
zz	318	byte	
xx	257	byte	
end	318	abs	\$ = 318

После завершения первого просмотра счетчик адресов устанавливается в *исходное значение*, и начинается второй просмотр. На втором просмотре ассемблер создает выходной файл и, по мере считывания исходного текста, заполняет выходной файл кодами инструкций и данных. На втором просмотре величина `zz` известна (318), и слово, заданное строкой (5), инициализируется значением $318 - 256 = 62$. (Выражение `zz - start` в примере представляет суммарный объем данных.)

Примечание

Единственное назначение имени `zz` в рассмотренном примере — формирование в строке (5) значения размера программы. Поскольку для обозначения адреса первого байта за пределами кода программы предусмотрено специальное имя — `end`, оператор (5) может быть задан в виде `dw end - start`, и оператор (6) не нужен.

Как следует из рассмотренного примера, определение значений выражений, включающих в себя имена, может быть отложено до второго просмотра. Вместе с тем, *число байт*, занимаемых машинным представлением инструкции или данных, должно быть известно уже при первом просмотре. В част-

ности, из этого следует, что *счетчик повторов* в конструкции `dup` должен быть известен при первом же просмотре.

По умолчанию `as86` создает выходной файл в формате `com`. Файл в `com`-формате содержит только те коды, которые были заданы исходным текстом программы, без дополнительной служебной информации. В рассмотренном примере выходной файл генерируется в размере 62 байт, и содержимое его в шестнадцатеричном формате по байтам следующее:

```
56 65 72 79 20 6C 6F 6F 6F 6F 6F 6E 67 20 74 61
61 61 61 6C 6C 20 53 61 61 61 6C 6C 79 21 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 3E 00
```

Почему при трансляции `com`-программы начальное значение счетчика адресов равно 0100? При запуске `com`-файла загрузчик получает от DOS блок памяти размером 64 Кбайт (*программный сегмент*) и записывает в него по адресам 0—255 (в *префикс программного сегмента*, сокращенно *PSP*) информацию о среде выполнения. Затем со смещения 256 (0100) в программный сегмент копируется содержимое `com`-файла, после чего DOS передает управление в программный сегмент по смещению 0100. Поэтому отсчет адресов при трансляции `com`-программы начинается с 0100.

Контрольные вопросы

1. Постройте таблицу имен для следующего исходного текста:

```
hoh      dd    5 dup end — hoh, ?
yoy      dw    ($ — hoh) dup (end — yoy)
what     equ   offset yoy — hoh
let      db    what dup ?
zeit     equ   hoh + 1
zz       db
```

2. Постройте таблицу имен для следующего исходного текста:

```
szl      equ   (bit 2) * 7
neck     db    szl dup (1, 4 dup ?), ?
          dw    szl dup (1, 4 dup ?), ?
          dd    szl dup (1, 4 dup ?), ?
total    equ   end — szl
bottle   equ   neck + 8
```

3. Постройте таблицу имен для следующего исходного текста:

```
alfa     dw    1
          db    13 dup (offset beta — end)
beta:
gamma     dw    (1k + 1 — offset alfa) dup ?
cold      equ   gamma — 2
zz        equ   end — alfa
```

4. Постройте таблицу имен для следующего исходного текста:

```

        db    'Rancid'
        dw    end

dumb:
pump:
bumb    equ    $
        dd    11 dup 11
lamp    dw    ($ - dumb) * 2 dup ?
zunz    equ    lamp - 1
zz      equ    $ - end

```

5. Укажите ошибку, возникающую на первом просмотре, в следующем исходном тексте:

```

start:
msg     db    'Ha-ha...'
        db    $ - msg dup ?
        dw    1, 12, zz - start, (zz - start) dup ?

zz:

```

6. Укажите ошибку, возникающую на первом просмотре, в следующем исходном тексте:

```

        dw    (offset lamp - dumb) dup ?

dumb:
        dd    12 dup offset end
lamp    dw    (offset lamp - dumb) * 2 dup ?

```

7. Укажите ошибку в следующем исходном тексте:

```

alfa    db    (gamma - alfa) * 2
        dw    alfa, beta, 4 dup gamma
gamma    equ    $ - alfa

```

8. Постройте таблицу имен для следующего исходного текста:

```

alfa    db    end - gamma
        dw    1k dup alfa
        dd    4 dup ?

beta:
gamma    equ    $ - alfa
silly    equ    alfa - 1

```

9. Укажите ошибку в следующем исходном тексте:

```

start    dd    1
zz        equ    $
msg1     db    1k dup (1, 7 dup 'A', ?)
        db    $ - msg2 dup ?
        dw    zz - start dup ?

msg2     db    1k dup (1, 7 dup 'A', ?)

```

2.2. Повторное определение имен

Повторное определение имени является ошибкой в том случае, если изменяются атрибуты имени — его числовое значение и/или тип. Это правило справедливо только для а86. В других ассемблерах (и в языках высокого уровня) повторное определение имен недопустимо в принципе.

Пример:

```
alfa      equ    12                ; (1)
gamma:    ;                        ; (2)
gamma:    ;                        ; (3)
beta      db                ; (4)
beta      db    1, 10          ; (5)
alfa      equ    49 shr 2       ; (6)
beta:     ;                        ; (7)
```

Повторное определение имени `alfa` в строке (6) не противоречит первому определению в строке (1) — значение 12 и тип `abs` остаются прежними. Также, атрибуты не меняются и при повторном определении `gamma` в строке (3), т. к. значение `$` для строк (2) и (3) одинаково, а тип по-прежнему `abs`. Аналогично, второе определение `beta` — в строке (5) — не вносит ничего нового ни в значение, ни в тип имени. Напротив, третье определение `beta` — в строке (7) — конфликтует с предшествующими определениями и потому является ошибкой.

2.3. Локальные имена

В а86 введена особая категория имен — *локальные*. (Точнее было бы назвать их *переопределяемыми*.) К локальным именам относятся все имена, начинающиеся с буквы, за которой следует одна или более цифр. Локальные имена допускают многократные переопределения, с произвольным изменением значения и типа.

Пример:

```
11      db    1                ; (1)
        dw    offset 11        ; (2)
        dw    offset >11       ; (3)
12      equ    1                ; (4)
11      dw    12               ; (5)
```

Имя `11` переопределено в строке (5), с изменением атрибутов. Символ `>` в строке (3) предписывает транслятору использовать только последующее определение имени. В результате указания `>` перед `11`, атрибуты `11` становятся неопределенными. Пока это имя не определено заново (в строке (5)), допускаются ссылки только вперед на последующее определение. Во избе-

вание разночтений, в промежутке между (3) и (5) правильной считается лишь точная ссылка >11, а ставшее теперь двусмысленным обозначение 11 считается ошибкой.

```

11      db      1                      ; (1)
        dw      offset 11              ; (2)
        dw      offset >11            ; (3)
        dw      offset 11              ; (4)  ошибка!
11      dw      12                      ; (5)

```

2.4. Предопределенные имена

Как во всех языках программирования, в языке ассемблера имеются *встроенные*, или *предопределенные имена* (также их называют ключевыми). Определение таких имен в программе недопустимо, а86 в подобных ситуациях выдает ошибку "Misplaced Built-In Symbol" (встроенное имя неуместно).

Примеры:

```

ax      equ     12
mov     db      10, 20
b       dw      5
w       dw      ?
d       dw      ?
byte    db      ?
st      db      "Hello, world"

```

Все определения выше ошибочны, поскольку:

- ❑ ax — имя, обозначающее один из регистров процессора i80x86;
- ❑ mov — обозначение мнемоники инструкции;
- ❑ b, w, d — принятые в а86 сокращения ключевых слов byte, word, dword и ключевых фраз byte ptr, word ptr, dword ptr;
- ❑ st — обозначение стека сопроцессора i80x87.

2.5. Имя end

Имя end в а86 определяется самим транслятором, в конце первого просмотра. Значение end — адрес первого свободного байта за пределами программы.

Если пользователь сам определяет имя end, строка с таким определением полностью игнорируется. (Иначе, в конце первого просмотра транслятор ловил бы себя на повторном определении end.)

В языке а86 строка, начинающаяся с ключевого слова end, которое по традиции обозначает конец исходного текста на перфоленте (!), пропускается (но ошибкой не считается — ради совместимости с ассемблерами

masm/tasm/wasm). Следует запомнить, что определение имени `end` приведет к *искажению программы* при трансляции; найти причину будет очень трудно, т. к. исходный текст выглядит "несомненно" правильным.

Пример:

```
begin      db      "There's no point in asking,"      ; (1)
           db      "Cause I won't reply."             ; (2)
           db      "Oh, just remember,"               ; (3)
           db      'I don"t decide.'                  ; (4)
end         db                                           ; (5)
half_vacant equ    end - begin                         ; (6)
           db      1k dup ?                            ; (7)
```

Транслятор прекратит обработку строки (5) сразу при считывании `end`. В результате, значение константы `half_vacant` — это вовсе не число литер в процитированной половине куплета, а "расстояние" от начала куплета до *конца программы* — на 1024 больше предполагаемого.

Наиболее распространенная грубая ошибка, связанная с именем `end`:

```
start:
    ...
    jmp  end
    ...
end:   int    020      ; возврат в DOS
    ...

{end:}
```

Инструкция `int 020` даже не будет записана в `com`-файл. Задуманный переход к команде выхода в DOS не самом деле выведет за пределы кода, определенного в программе, — на неявно заданную метку `end`.

Контрольные вопросы

1. Найдите ошибку:

```
l1      dd      5 dup end - l1, ?
l2      dw      ($ - l1) dup (end - >l1)
what    equ     offset >l1 - l2
this    equ     l1 + 3
l1      db
```

2. Постройте таблицу имен для следующего исходного текста:

```
l1      db      12 dup 1
l1      db      ($ - l1) dup ?
end     db      ($ - l1) dup ?
l1      db      ($ - l1) dup ?
l2:
where    equ     l1 + 1
```

3. Постройте таблицу имен для следующего исходного текста:

```

alfa      dw    1
          db    13 dup (offset beta - end)

beta:
gamma     dw    (1k + 1 - offset alfa) dup ?
zz        equ   end - alfa
hole      equ   alfa + 1

```

4. Укажите ошибку, возникающую на первом просмотре, в следующем исходном тексте:

```

start:
msg       db    'Ha-ha...'
          db    this - msg dup ?
          dw    1, 12, zz - start, (zz - start) dup ?

zz:

```

5. Укажите ошибку, возникающую на первом просмотре, в следующем исходном тексте:

```

          dw    (offset lamp - dumb) dup ?

dumb:
          dd    12 dup offset end

lamp      dw    (offset lamp - dumb) * 2 dup ?

```

6. Укажите ошибку в следующем исходном тексте:

```

alfa      db    (gamma - alfa) * 2
          dw    alfa, beta, 4 dup gamma
gamma     equ   $ - alfa

```

7. Постройте таблицу имен для следующего исходного текста:

```

alfa      db    end - gamma
          dw    1k dup alfa
          dd    2 dup ?

beta:
tutto     equ   alfa - 2
gamma     equ   $ - alfa

```

8. Укажите ошибку в следующем исходном тексте:

```

start     dd    1
zz        equ   this
msg1      db    1k dup (1, 7 dup 'A', ?)
          db    $ - msg2 dup ?
          dw    zz - start dup ?
msg2      db    1k dup (1, 7 dup 'A', ?)

```

9. Постройте таблицу имен для следующего исходного текста:

```
        db    'Ramones '
        dw    end

dumb:
bumb    equ   $
        dd    2 dup ?
lamp    dw    ($ - dumb) * 2 dup ?
z1      equ   lamp + 10
zz      equ   $ - end
```

ГЛАВА 3



Практикум по программированию данных

Пробежав взглядом по списку, содержавшему свыше пятидесяти размеров и указаний, Слобольд понял, что у дамы, которой предназначалось платье, на животе три груди, причем каждая своей величины и формы. Помимо того, на спине у нее несколько больших горбов. На талию отводилось всего восемь дюймов, зато четыре руки, судя по проймам рукавов, по толщине не уступят стволу молодого дуба. О ягодицах не упоминалось вообще, однако величина клеши подразумевала чудовищные вещи.

Роберт Шекли. "Заказ"

Тема главы — знакомство с транслятором a86 и получение с его помощью com-программы с данными, но без инструкций.

3.1. Запуск ассемблера a86

Программа на языке ассемблера a86 транслируется запуском командной строки:

```
a86.com <prog>.8 [опции]
```

где <prog>.8 — имя файла с исходным текстом программы (вместо <prog> подставляется конкретное имя файла).

Если трансляция успешна, создаются одноименный файл с расширением com (com-программа) и одноименный файл с расширением sym, содержащий таблицу имен для отладчика d86.

Если обнаружены ошибки, исходный файл копируется в одноименный файл с расширением old, затем в исходный файл вставляются сообщения об ошибках. Каждое сообщение состоит из двух строк: в первой содержится указатель ошибки, во второй — пояснение.

Приведем пример. Программа в файле test1.8 содержит ошибку. При трансляции в файл test1.8 вставляются два сообщения, как показано в листинге 3.1. Первое из них говорит о том, что *все* сообщения об ошибках будут удалены, если при исправлениях оставить это первое сообщение.

Листинг 3.1. Вставка сообщений об ошибках в исходный текст программы

```
~^
#ERROR messages will be removed if you leave these first two lines in
start:
w_var    dw    256
dozen    equ    12
d_var    dd    dozen
z1       dw
tasm     db
masm     db    "We're MASM & TASM, old-style behemoth assemblers.",13,10
a86      db    "I'm A86, fast & tiny useful one!"
~         ^
#ERROR 32: Bad String
```

Значок ^ указывает на начало ошибочной конструкции. В этом примере строка после db "плоха" потому, что не закрыта. Повторите трансляцию, поставив двойную кавычку в конце строки:

```
a86      db    "I'm A86, fast & tiny useful one!"
```

При трансляции без ошибок сообщения из исходного текста исчезнут. Если удалять сообщения вручную, то по паре строк сразу (строку с указателем и строку с пояснением).

Вставка сообщений в исходный текст отменяется опцией +E (a86 <prog>.8 +E); при этом исходный текст с сообщениями об ошибках копируется в файл с расширением err.

Ссылки на неопределенные символы в исходном тексте не отмечаются. Сведения об этих символах записываются в файл с расширением und. Выполните трансляцию программы test1.8, определив значения данных по адресу tasm неизвестными именами, например:

```
tasm     db    awful, boring
```

Запуск a86 с ключом +L позволяет получить отчет о ходе выполнения трансляции; такой отчет называется листингом и сохраняется в файл с расширением lst. В листинге показаны изменения счетчика адресов, размещение и заполнение памяти инструкций и данных, в конце — содержимое таблицы имен.

Запустите на трансляцию исправленный вариант test1.8 с ключом +L (a86 test1.8 +L). Откройте test1.lst и обратите внимание на следующие особенности:

- ❑ при выводе таблицы имен типы abs, byte, word, dword обозначены как ":", mb (memory byte), mw (memory word), md (memory dword) соответственно;
- ❑ локальные имена из таблицы не выводятся (где z1?);
- ❑ содержимое памяти, заданное директивой db/dw/dd, отображается не полностью, если число байт велико.

Последнее ограничение снимается опцией +h<n>, где <n> — количество дополнительных строк для вывода. Например, для расширения вывода на десять дополнительных строк выполните:

```
a86 test1.8 +L +h10
```

или

```
a86 test1.8 +Lh10
```

3.2. Программирование данных

Практическое программирование начнем с простейших программ, которые содержат только данные. Такие программы можно использовать для создания файлов с данными для тестирования других программ. Рассмотрим пример.

Пусть требуется запрограммировать повторяющуюся последовательность байт со значениями 0, 1, -2 и -1; в начале файла должны быть включены два слова — со значением общей длины выходного файла и со значением количества повторов последовательности. Длина последовательности — 2,5 Кбайт. Решение показано в листинге 3.2.

Листинг 3.2. Решение задачи (см. var_0.8)

```
start:
    dw    offset end - start

size    equ  5k / 2          ; длина последовательности
r_cnt   equ  size / 4        ; число повторов

        dw    r_cnt
seq     db    r_cnt dup (0, 1, -2, -1)

size    equ  $ - seq         ; длина получившейся последовательности
```

Повторное определение имени `size` введено для контроля совпадения ожидаемой (расчетной) длины последовательности с ее фактическим размером. Проверьте, что произойдет при трансляции, если в `var_0.8` между `seq` и повторным определением `size` вставить еще один байт.

После трансляции выбранного варианта задания проверьте совпадение результата с готовым решением в файле `var_<n>.dat` (<n> — номер варианта) при помощи утилиты `cmp` или `fc` (с ключом `/b`). Для рассмотренного варианта номер ноль:

```
cmp var_0.com var_0.dat
```

или

```
fc /b var_0.com var_0.dat
```

Примечание

Полученный `com`-файл не следует пытаться вызвать на выполнение; с таким же успехом можно переименовать любой текстовый файл (размером не больше 64 Кбайт) в файл с расширением `com` и запустить его как программу.

Прежде чем переходить к выполнению задания, рекомендуем выполнить трансляцию заданий из предыдущих глав — для проверки ответов, которые были получены вами "вручную".

Варианты заданий

1. Запрограммируйте повторяющуюся последовательность байт длиной 3 Кбайт со значениями: `bit 0`, `bit 2`, `bit 4`, `bit 6`. В конце файла следует задать еще один байт, со значением 0.
2. Запрограммируйте чередование байта со значением 1 и слова со значением 1. Общий объем данных — 3 Кбайт.
3. Запрограммируйте чередование последовательностей:
 - 10 байт со значением -1;
 - 20 байт со значением 0;
 - 30 байт со значением 1.Общий объем данных — 3000 байт.
4. Запрограммируйте чередование байт со значениями 27 и ASCII-кода 'X'. Общий объем данных — 1 Кбайт.
5. Запрограммируйте три последовательности данных в одном файле:
 - байты со значением 10, объем последовательности 1 Кбайт;
 - слова со значением 10, объем последовательности 1 Кбайт;
 - двойные слова со значением 10, объем последовательности 1 Кбайт.

6. Запрограммируйте 100-кратное повторение следующей последовательности:

- слово со значением 3;
- пять повторов последовательности байт со значениями 0, 1, 1, 1;
- двойное слово со значением 256.

В этих заданиях, ввиду их простоты, эффект от использования имен незначительный. В файле `bmp_0.8`, где запрограммированы данные в `bitmap`-формате, имена используются по максимуму, для следующих целей:

- ☐ по трем параметрам (ширина, высота и число бит на точку) вычисляются все константы, используемые для определения заголовков `bmp`-файла;
- ☐ для определения общего объема файла используется имя `end`;
- ☐ объем отдельных фрагментов вычисляется как разность между именами, в частности, с использованием текущего значения счетчика адресов \$;
- ☐ расхождение между расчетным и фактическим размерами таких данных, как палитра и растр, исключается за счет повторного определения соответствующих имен.

Запустите на трансляцию пример `bmp_0.8` следующим образом:

```
a86 bmp_0.8 TO bmp_0.bmp
```

При таком вызове `a86` записывает результат не в одноименный `com`-файл, а в файл, указанный после `TO`. Картинка, полученная в файле с расширением `bmp`, просматривается либо в Windows (например, редактором Paint), либо в Norton Commander в режиме просмотра файлов (клавиша <F3>) — по расширению `bmp` должна подключаться утилита `bitmap.exe`. После удачной загрузки — если данные соответствуют `bmp`-формату — программа `bitmap.exe` позволяет по нажатию клавиши <F6> получить изображение в цвете, а по клавише <+> — увеличить изображение.

Задания на программирование данных растра

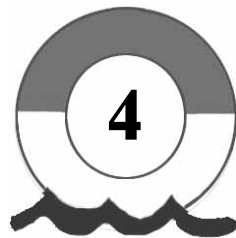
1. Запрограммируйте две горизонтальные линии, наверху и внизу. Для задания данных растра достаточно трех директив `db`: первая задает нижнюю линию, вторая — промежуток, третья — верхнюю линию.
2. Одной директивой `db` запрограммируйте "тельняшку" с толщиной линий 2 через промежуток 2.
3. Одной директивой `db` запрограммируйте "тельняшку" с толщиной линий 3 через промежуток 1.
4. Одной директивой `db` запрограммируйте "часток" с толщиной линий 4 через промежуток 4.
5. Одной директивой `db` запрограммируйте "шахматную доску" с размером клетки 1×1, а затем, тоже одной директивой, доску с размером клетки 4×4.
6. Запрограммируйте окантовку четырьмя линиями единичной толщины по периметру. Для задания данных растра достаточно трех директив `db`: одна — для нижней линии, вторая — для промежутка, третья — для верхней линии.

7. Запрограммируйте прямоугольник в центре картинки, отделенный от краев промежутками в 8 линий. Для задания данных раstra хватит трех директив `db`: для нижнего промежутка, для прямоугольника и для верхнего промежутка.
8. Одной директивой `db` запрограммируйте покраску левой половины картинки, затем — верхней половины картинки.

Примечание

Меняйте *только* содержимое данных с адреса `Pixels`. Также, не следует отключать повторное определение `byteCount` в конце программы, поскольку оно гарантирует правильность объема данных раstra.

ГЛАВА 4



Просмотр данных в отладчике

В этой главе рассмотрены средства отладчика d86, предназначенные для отображения данных, определенных в программе на a86.

Отладчик d86 представляет собой полноэкранную программу для исследования исполняемых com- и exe-программ для i80x86. При отображении программы отладчик пользуется таблицей имен, которая была сохранена при трансляции в sym-файле. В результате, d86 не только отображает программу в символическом виде, но также позволяет использовать при отладке символические имена, определенные в исходной программе.

4.1. Запуск и завершение сеанса отладки

Отладчик запускается командной строкой:

```
d86 [<file> [<cmd_tail>]]
```

где `<file>` — отлаживаемая программа с расширением `com` или `exe`, `<cmd_tail>` — параметры командной строки для отлаживаемой программы. Информация о символах, определенных пользователем в отлаживаемой программе, автоматически считывается из одноименного файла с расширением `sym`.

Выполните трансляцию программы `damn.8` и запустите отладку `damn.com`:

```
d86 damn.com
```

Обратите внимание на воспроизведение данных, определенных между инструкцией `jmp start` и меткой `"start:"`. Отладчик не пользуется исходным текстом программы и поэтому не знает, где были определены данные, а где инструкции. В результате, он пытается все содержимое `com`-программы расшифровать как набор инструкций и при этом не показывает имен пере-

менных (переменная — это символический адрес данных) — только метки (метка — это адрес инструкции в символическом виде).

Для выхода из отладчика введите команду **q** и нажмите клавишу <Enter>.

Удалите файл `damn.sym` и вновь запустите отладку `damn.com`. Таблицы имен теперь нет, и вместо имени `start` отладчик ставит адрес в числовом виде.

Продемонстрируем ошибочный запуск. Выйдите из отладчика и войдите вновь следующим образом:

```
d86 damn.8
```

Ошибка в том, что отладчику в качестве исполняемой программы передан исходный текст. Отладчик воспринял файл `damn.8` как загрузочный модуль в формате `com`, поэтому содержимое экрана отличается от того, что мы видели при задании `damn.com`.

Примечание

Независимо от расширения (`8`, `asm`, `com`, `exe` и т. д.), отладчик считает, что файл всегда содержит *исполняемый* код. Различие между форматами `com` и `exe` устанавливается им по характерной *структуре заголовка* `exe`-файла. Если признаки такой структуры не выявлены, предполагается формат `com`.

Выйдите из отладчика, выполните трансляцию программы `test1.8` и вновь войдите в отладчик:

```
d86 test1.com
```

4.2. Экран отладчика

В начале работы `d86` генерирует полноэкранное изображение и ожидает ввода команд.

В верхнем левом углу — результат расшифровки машинного кода отлаживаемой программы. В первой колонке выводится адрес первого байта машинной инструкции, во второй — сама инструкция. Красным курсором `#` отмечена текущая инструкция для выполнения, адрес которой определяется регистром `ip` (*указатель инструкции*).

В нижнем левом углу отображается содержимое *регистров процессора* в формате шестнадцатеричных слов. Над второй колонкой регистров в восьми позициях выводятся символы, обозначающие состояние восьми *флагов процессора*.

В нижнем правом углу находится область отображения данных — шесть строк, пронумерованных от единицы. В строке с номером `0` всегда выводится содержимое *системного стека* в формате шестнадцатеричных слов (изначально стек пуст).

В верхнем правом углу в начале работы выводятся сведения об авторе a86/d86. При нажатии клавиши <F10>, на этом месте последовательно отображаются:

- экран состояния;
- продолжение последней строки отображения данных;
- экран состояния арифметического сопроцессора, если таковой имеется.

Нажмите комбинацию клавиш <Alt>+<F10>. Вы перешли в режим подсказки. В этом режиме ввод команд сопровождается пояснениями. В исходном состоянии, пока не начат ввод очередной команды, нажатия клавиши <F10> позволяют пролистать перечень команд. Включить/выключить режим подсказки можно *в любой момент* нажатием комбинации клавиш <Alt>+<F10>.

4.3. Окна отображения данных

Нажмите клавишу <1>. Красный маркер отмечает начало первой строки отображения данных. Введите b,0100 и нажмите клавишу <Enter>. Спецификация b задает отображение по байтам в шестнадцатеричном формате, начиная с адреса 0100.

Продолжим вывод на следующую строку. Для этого нажмите клавишу <2>, затем введите двойную кавычку " (при помощи клавиши <Shift>). В начале строки отладчик показывает адрес, с которого начинается продолжение вывода. Аналогично продолжите вывод на третью строку.

Продолжим вывод на 14-строчное *дополнительное окно данных* в верхнем правом углу экрана. Для этого достаточно поместить дополнительное окно данных наверх. Первый способ — нажимайте клавишу <F10>, пока не появится окно с данными. Второй способ — нажмите комбинацию клавиш <Ctrl>+<N> или <Ctrl>+<P> (вывод следующих или предыдущих 14 строк данных).

Выведите следующие три страницы, пользуясь комбинацией клавиш <Ctrl>+<N>. Данные в дополнительном окне больше не являются продолжением данных основного окна. Чтобы подогнать адреса в дополнительном окне к адресам основного окна, нажмите клавиши <1> и <Enter>.

Задайте в строке 5 отображение данных с адреса `masm` по байтам в шестнадцатеричном формате; используйте символическое имя — `masm`.

Обратите внимание на изменения в дополнительном окне. Какая строка основного окна управляет им?

Отключите вывод в третьей строке. Для этого, выбрав третью строку, нажмите клавишу пробел. Аналогично, отключите вывод в первой строке.

Заново задайте вывод в строке 5, но вместо запятой используйте литеру `"/"`. Как изменилась установка дополнительного окна?

Перед выходом из отладчика сохраним нажатия клавиш, чтобы восстановить последовательность действий при последующем сеансе.

4.4. Сохранение нажатий клавиш

Переключитесь в режим `help`, после чего нажмите клавишу `<k>`. Подсказка сообщает, что вводится либо команда отладчика из группы "keystrokes", либо непосредственно исполняемая инструкция ассемблера (этот режим рассматривается в следующей работе). Нажмите клавишу пробел, после чего выберите вариант `s` (сохранение предыдущих нажатий клавиш). После ввода литеры "s" и пробела подсказка предлагает задать имя файла. Введите имя, совпадающее с именем файла отлаживаемой программы, завершите ввод нажатием клавиши `<Enter>`.

Выйдите из отладчика и вновь войдите с той же командной строкой. Нажатия клавиш были сохранены в файле с расширением `d8k`. При запуске отладчика нажатия из *одноименного* файла проигрываются автоматически.

Примечание

При автономном запуске отладчика, т. е. без указания отлаживаемой программы, проигрываются нажатия из файла `d86.d8k`.

Сохраните нажатия клавиш в файле с именем, составленным из `alt` и латинской буквы, например, `alta`. Затем выполните команду начальной установки сеанса отладки с очисткой памяти нажатых клавиш: `<s> <f> <Enter>`. Нажмите комбинацию клавиш `<Alt>+<A>`.

4.5. Форматы вывода данных

Выберите свободную строку, нажав клавишу `<0>`. Если режим `help`, в правом верхнем углу появится таблица *спецификаций форматов* (сменить режим можно в любой момент, нажав комбинацию клавиш `<Alt>+<F10>`).

Ниже приведен перечень форматов в четыре столбца в алфавитном порядке.

ASCII 2	H ?	Octal W 6	V ?
ByteHex 2	I ?	Port B	WordHex 4
Char ?	Join	Q Octal B 3	X Skip
Decimal W	K ?	Raw text 1	Y ?
EightBit B 9	Line, LF	String, null	Z addr is end
Float	Mark 	Text 1	= loc not val
Gap	N Decimal B	Unskip	@ memB=count

Спецификация формата в строке вывода задается первой буквой названия формата. Например, буква `a` задает формат ASCII, буква `w` — формат WordHex. Заметим, что всем буквам латинского алфавита соответствует спецификация: зарезервированные буквы `h`, `i`, `k` и пр. тоже задают вывод — отображают знак вопроса. Цифры в конце показывают число позиций для одного элемента данных. Форматы, выделенные курсивом, задают вывод самих элементов данных; это — *базовые форматы*. Прочие форматы — либо зарезервированные (отображают знак вопроса), либо *вспомогательные*. Вспомогательные форматы задают разделитель (`Gap`, `Join`, `Mark`) или управляют разверткой отображения (`Skip`, `Unskip`).

4.5.1. Базовые форматы

В табл. 4.1 приведены базовые форматы для отображения данных в числовом виде.

Таблица 4.1. Базовые форматы для отображения чисел

Обозначение	Размерность	Формат отображения
b	Байт	Шестнадцатеричный
d	Слово	Десятичный
e	Байт	Двоичный
n	Байт	Десятичный
o	Слово	Восьмеричный
q	Байт	Восьмеричный
w	Слово	Шестнадцатеричный

Примечание

Спецификации `f` и `p` в табл. 4.1 не включены, т. к. в *части I* книги они нам не понадобятся. Спецификация `f` предназначена для отображения *действительных чисел* (4-, 8- и 10-байтных), `p` задает отображение *портов ввода/вывода* в формате шестнадцатеричных байт.

Для отображения данных в формате ASCII-символов предназначены спецификации `a`, `c`, `l`, `r`, `s`, `t`. Спецификация `l` выключает вывод при достижении конца текстовой строки (коды 13, 10), спецификация `s` прекращает вывод по концу строки ASCIIZ (строка ASCIIZ ограничена кодом `null` — нулем).

Разница между форматами `a`, `c`, `r`, `t` — в количестве позиций для вывода одного символа, а также в способе отображения *нестандартных символов*. Нестандартные символы — это символы, значение которых либо меньше 020

("невидимые" управляющие символы), либо больше 07f (вторая половина кодовой таблицы, где обычно определены литеры национального алфавита и символы псевдографики).

При отображении нестандартных символов в d86 используется *символ-двойник* из диапазона кодов 020—07f. Символ приводится к своему двойнику за счет прибавления или вычитания числа, кратного 040. Например, двойник символа null (код 0) — это символ @ (код 040), а двойник символа cr ("возврат каретки", код 0d) — это символ M (код 04d).

Спецификация `t` задает вывод двойников без пометок, т. е., по этой спецификации "настоящий" символ M и символ возврата каретки выглядят одинаково. Напротив, форматы ASCII и Char предусматривают вывод двойника пометками ^, \$ или #, которые обозначают соответственно -040, +040 и +080.

В формате Char вывод стандартного символа занимает одну позицию на экране, а вывод нестандартного — две; формат ASCII задает вывод всегда в две позиции.

В формате RawText ("сырой" текст) двойники вообще не используются — литеры выводятся прямо в видеопамять, без преобразований. Символ null в формате RawText не отображается, зато можно увидеть литеры кириллицы, что не позволяет никакой другой формат.

4.5.2. Составные форматы

Составной формат дает возможность:

- ☐ отображать в одной строке данные в разных форматах;
- ☐ управлять числом элементов в строке;
- ☐ задавать пропуск элементов или их повторный показ.

Для задания составного формата в спецификации указывается подряд несколько букв, определяющих формат последовательных элементов отображения.

Например, формат `nbw` задает вывод десятичного байта, шестнадцатеричного байта и шестнадцатеричного слова из последовательных ячеек памяти. Остаток строки в окне отображения выводится по последней букве спецификации; в примере остаток будет заполнен шестнадцатеричными словами — по спецификации `w`.

Повторяющиеся буквы можно обозначить сокращенно. Например, формат `nnnnbw` записывается как `4nbw`.

Счетчик в конце спецификации задает число повторений спецификации *в целом*. Например, `2b3wa5` означает пятикратный вывод последовательности, состоящей из двух шестнадцатеричных байт, трех шестнадцатеричных слов и одного символа ASCII. Если строка не вмещает такого отображения, вы-

вод урезается по границе спецификации (в данном случае, после отображения очередной последовательности двух шестнадцатеричных байтов, трех шестнадцатеричных слов и одного символа ASCII).

В составных форматах находят применение вспомогательные форматы Mark, Gap и Join. Формат Mark используется для замены разделителя-пробела на разделитель в виде литеры "|". Формат Gap расширяет обычный разделитель дополнительным пробелом. Формат Join, напротив, ликвидирует разделитель, так что соседние элементы вывода объединяются.

Спецификации `x` и `u` используются для управления *указателем вывода*. При развертывании изображения этот указатель автоматически увеличивается после вывода очередного элемента данных. Спецификация `x` дополнительно сдвигает указатель, и в результате очередной элемент пропускается (не выводится). Напротив, `u` задает сдвиг указателя в обратную сторону, так что один и тот же участок памяти отображается повторно.

Спецификация `x` со счетчиком повторений дает возможность пропустить вывод последовательности данных, не представляющих интереса. Напротив, спецификация `u` позволяет отобразить один и тот же фрагмент данных в разных форматах. Например, спецификация `eunucul` задает отображение одного и того же байта в трех форматах — в двоичном (`e`), в десятичном (`n`) и в виде литеры (`c`).

Примечание

Каждый экземпляр спецификации для управления указателем вывода задает смещение этого указателя на *один* байт.

4.6. Задания на самостоятельную работу

Задайте спецификацию для отображения в требуемом виде данных из программы `array.8`. Во всех вариантах задания данные определены в виде массива структур. Все массивы заданы в одной программе. Данные для первого варианта расположены по адресу `var1`, для второго — по адресу `var2` и т. д. Варианты задания приведены в табл. 4.2 с описанием структуры элемента массива и способа отображения.

Таблица 4.2. Варианты заданий

Вариант	Элемент массива	Требуемое отображение
1	Два байта и слово	Первый байт — в формате ASCII, второй — в десятичном формате, затем слово — в шестнадцатеричном

Таблица 4.2 (окончание)

Вариант	Элемент массива	Требуемое отображение
2	Четыре байта и слово, причем четвертый байт всегда равен нулю	Три байта — в формате символов, затем слово в десятичном формате, с пропуском четвертого байта
3	Слово и строка ASCIIZ нефиксированной длины	Слово в десятичном формате, затем строка до литеры null
4	Байт и строка ASCIIZ нефиксированной длины	Байт в шестнадцатеричном формате, дополнительный разделитель, затем строка символов
5	Двойное слово, строка из шести символов, байт	Младшее слово двойного слова — в шестнадцатеричном формате, пропуск старшего слова, шесть символов, байт в десятичном формате
6	Байт и строка ASCIIZ нефиксированной длины	Байт в десятичном формате, разделитель вместо пробела, затем строка
7	Два байта, два слова и байт	Байт в десятичном формате, байт в восьмеричном формате, затем пара слов в десятичном формате; последний байт не отображается
8	Байт и слово	Первый байт — сначала в двоичном формате, затем он же в шестнадцатеричном формате, затем слово — в виде двух символов "сырого" текста

Запустите на трансляцию программу array.8, войдите в d86 с полученным файлом данных array.com. Задайте вывод первых восьми элементов массива, последовательно, по одной структуре в каждом окне. Начните отображение в первой строке вывода, для продолжения в последующих строках используйте клавишу "<", а последние элементы отобразите в дополнительном окне.



Способы адресации

В этой главе рассмотрены способы адресации операндов машинных инструкций, применительно к микропроцессорам семейства i80x86.

5.1. Данные процессора

Кроме данных, определенных директивами `db/dw/dd`, программа на ассемблере имеет доступ к данным самого процессора.

Данные процессора, доступные программе, в дальнейшем называются регистрами. Из них наиболее часто используются восемь 16-разрядных регистров *общего назначения*, которые в программе обозначаются как `ax`, `bx`, `cx`, `dx`, `sp`, `bp`, `si`, `di`.

Обозначения, приведенные в табл. 5.1, отражают специальное назначение регистров в ряде инструкций. Например, регистр `cx` часто используется как счетчик, регистр `ax` — как источник входных данных и результат команды. (Так, например, один из сомножителей команды умножения `mul` всегда находится в регистре `ax`, а младшая часть результата также записывается в этот регистр.)

Таблица 5.1. Обозначения регистров процессора i8086

Обозначение 16-разрядного регистра	Обозначение старшего байта	Обозначение младшего байта	Расшифровка обозначения
ax	ah	al	Accumulator
bx	bh	bl	Base
cx	ch	cl	Counter

Таблица 5.1 (окончание)

Обозначение 16-разрядного регистра	Обозначение старшего байта	Обозначение младшего байта	Расшифровка обозначения
dx	dh	dl	
sp			Stack Pointer
bp			Base Pointer
si			Source Index
di			Destination Index

Байты регистров `ax`, `bx`, `cx`, `dx` доступны по отдельности. На ассемблере они обозначаются `ah`, `al` (старший и младший байты регистра `ax`), `bh`, `bl` (старший и младший байты `bx`), `ch`, `cl` (старший и младший байты `cx`), `dh`, `dl` (старший и младший байты `dx`).

Пример:

```
mov    dx, 0           ; (1)
inc    dh              ; (2)
mov    al, dh          ; (3)
```

Команда (1) записывает в `dx` ноль, затем команда (2) инкрементирует старший байт `dx`. В результате, в `dx` сформировано число 256. (Почему не 1?) Команда (3) копирует содержимое `dh` в `al`, так что `al` теперь содержит 1. Целиком содержимое `ax` не известно, поскольку значение `ah` не определено — в `ah` ничего не записывали.

Контрольные вопросы

1. Какие значения примут регистры `si`, `di` в результате выполнения следующей последовательности инструкций?

```
mov    ah, bit 0
mov    dh, 0
mov    cl, ah
mov    ch, dh
mov    si, cx
mov    di, ax
```

2. Какое значение примет регистр `ax` в результате выполнения следующей последовательности инструкций?

```
mov    ax, 0ffh by (bit 1 + bit 4)
inc    ah
```

3. Какое значение примет регистр `ax` в результате выполнения следующей последовательности инструкций?

```
mov    ax, 01ff
inc    ax
inc    al
```

4. Какое значение примет регистр `ax` в результате выполнения следующей последовательности инструкций?

```
mov    ax, 01ff
inc    al
inc    ax
not    ax      ; инверсия разрядов в ax
```

5. Какое значение примет регистр `dx` в результате выполнения следующей последовательности инструкций?

```
mov    ah, -2
mov    dh, -1
mov    dl, ah
inc    dh
```

6. Какое значение примет регистр `ch` в результате выполнения следующей последовательности инструкций?

```
mov    cx, (bit 1) by -1
inc    cl
inc    cx
inc    ch
```

5.2. Обозначения операндов машинных команд

Операнды — это данные, которые обрабатываются машинной инструкцией. Например, команда `add` суммирует два операнда, а результат помещает на то место, откуда она прочитала первый операнд.

В обозначении машинной инструкции на ассемблере операнды следуют за мнемоникой операции. Если операндов два, то результат помещается на место первого операнда. В команде с двумя операндами их размерность (байт, слово или двойное слово) должна быть одинаковой.

Примеры задания операндов:

```
nop                ; (1)
inc    ax           ; (2)
mov    ax, 10       ; (3)
add    al, bl        ; (4)
shl    ax, 1         ; (5)
shl    ax, cl        ; (6)
```

Комментарии.

- ❑ Команда (1) — без операндов, а по смыслу это пустая операция (реализована как обмен содержимого регистра `ax` с содержимым того же регистра `ax`).
- ❑ Команда (2) — инкремент, операнд у нее один (в данном случае содержимое регистра `ax`). Операнд считывается из регистра `ax`, а результат операции (увеличение на 1) записывается в регистр `ax`.
- ❑ Команда (3) — копирование, считывает второй операнд (число 10) и записывает прочитанное в первый операнд — в регистр `ax`.
- ❑ Команда (4) — сложение, считывает операнды из регистров `al` и `bl`, а результат помещает на место первого операнда — в регистр `al`.
- ❑ Команда (5) считывает первый операнд, сдвигает его значение влево на число разрядов, заданных вторым операндом, записывает результат на место первого операнда.
- ❑ Команда сдвига (6) представляет исключение из правила равной размерности операндов: число сдвигов задано *байтовым* регистром `cl`, а сдвигаемая величина, в данном случае,— *слово* (содержимое регистра `ax`).

5.3. Способы адресации операндов

Адресация операндов — это задание *местонахождения* операндов (т. е., где находится число для операции или куда поступает результат).

В ассемблерах для `i80x86` местоположение операндов обозначается только в том случае, если оно не фиксировано.

```
mul    cx        ; cx * ax → dx, ax
inc    dx        ; dx + 1 → dx
```

Например, команда `mul` всегда считывает один сомножитель из регистра `ax`, а результат неизменно помещает в пару регистров `dx`, `ax`. Поэтому в обозначении инструкции `mul` указывается местонахождение только одного из сомножителей. Напротив, местонахождение операнда команды `inc` должно быть указано явно, т. к. операнд может быть задан в любом из регистров, а также в памяти данных по произвольному адресу.

5.3.1. Регистровая и непосредственная адресация

Операнд машинной команды находится или в регистре процессора, или непосредственно в инструкции, или в памяти данных.

Первый вариант адресации обозначается названием регистра; адресация называется *регистровой*.

Примеры:

```
mul    ax
inc    si
mov    dh, cl
```

Второй вариант: операнд задан в самой инструкции; адресация называется *непосредственной*.

Примеры:

```
add    al, 2           ; (1)
add    ax, 2           ; (2)
shl    cx, 4           ; (3)
inc    9               ; (4)  ?!
add    3, ax           ; (5)  ?!
```

Непосредственное значение при трансляции записывается в конец кода команды (в начале кода — признак "второй операнд непосредственный"). В инструкциях языка ассемблера непосредственный операнд обозначается числом или именем типа *abs*. Результат команды не может быть записан на место непосредственного операнда, поэтому команды (4) и (5) ошибочны, и при их трансляции выдается ошибка "Constant/Label Not Allowed".

5.3.2. Адресация данных в памяти

Третий вариант адресации — операнд находится в памяти данных по некоторому адресу. В зависимости от способа задания адреса различают *прямую* и *косвенную* адресацию.

Прямая адресация

При прямой адресации значение адреса задано в конце кода команды (в коде команды также указан признак прямой адресации). При выполнении команда организует доступ к памяти по адресу, значение которого предварительно считывается процессором из машинной инструкции. На языке ассемблера прямая адресация обозначается *именем переменной* или числом (константой типа *abs*) в квадратных скобках. В последнем случае может потребоваться явное указание размерности операнда — байт, слово или двойное слово.

Пример:

```
inc    b_1             ; (1) "byte"
inc    w_1             ; (2) "word"
inc    [080]           ; (3) ?!
inc    byte ptr [080]   ; (4) "byte"
inc    byte [080]       ; (5) "byte"
```

```
inc      b [080]                ; (6) "byte"
inc      w [c_1]                ; (7) "word"
mov      word ptr [080], al      ; (8) ?!
mov      [080], al              ; (9) "byte"
mov      [080], ax              ; (10) "word"
mov      [080], 1               ; (11) ?!
```

Предполагается, что имя `c_1` определено ранее как константа (тип `abs`), имя `b_1` — как переменная типа `byte`, а имя `w_1` — как переменная типа `word`. При трансляции (1) и (2) размерность операции (инкремент байта или инкремент слова) определяется по *типу* имени. В команде (3) не хватает информации о размерности, и трансляция закончится ошибкой "Is it Byte or Word?". В командах (4) — (7) информация о размерности задана операторами преобразования типа; `byte ptr`, `word ptr` — полная форма записи этих операторов, `byte` или `b`, `word` или `w` — сокращенная.

В команде (8) — ошибка: размерности операндов разные (у первого — `word`, у второго — `byte`), что недопустимо в большинстве инструкций с двумя операндами (в таких случаях выдается сообщение "Byte/Word Combination Not Allowed"). Исходя из того, что размерности *должны быть* одинаковы, в командах (9) и (10) тип данных по адресу `[080]` выясняется из типа второго операнда. В команде (11) вновь не хватает информации о размерности, поскольку непосредственное значение `1` может быть представлено как байтом, так и словом.

Контрольные вопросы

1. Назовите способы адресации, которые использованы в следующих командах:

```
mov      ah, bh
inc      w [0]
add      byte [0fffe], type (w [0])
```

2. Укажите ошибки и объясните их причину:

```
mov      1 shl 3, ax
add      bx
mul      ax, cx
```

3. Укажите ошибки и объясните их причину:

```
add      [-2], 256
add      [10], 255
mov      [3], ax
inc      [0]
```

4. Какие способы адресации использует машинная команда, которая соответствует следующей команде на языке ассемблера:

```
mul      w [0]
```

5. Какова размерность каждой из следующих операций:

```
add    ax, 1
inc    byte ptr [0]
mov    ax, [0]
```

6. Какие способы адресации заданы в следующих командах и какие команды заданы неправильно?

```
add    5, 9
add    ax, 1
mov    al, cx
```

Косвенная адресация

Адрес данных в памяти может быть задан косвенно. Это означает, что в команде указывается местоположение элемента памяти, где находится *адрес* операнда. В процессорах семейства i80x86 таким элементом может быть только регистр, причем 16-разрядный (за исключением инструкций передачи управления, см. *разд. 6.3.5*).

Пример:

```
inc    b[si]
```

Адрес байта в памяти задан содержимым регистра *si*. Если, например, в *si* находится число 0103, то при выполнении этой команды операция увеличения на 1 будет выполнена над байтом по адресу 0103. Значение *si* при этом останется прежним.

Задавая разные значения в *si*, программа может обращаться к различным элементам памяти — при помощи одной и той же команды в цикле. Эта возможность широко используется при обработке массивов.

Предположим, в программе задан массив:

```
array  db      1, 3, 9, 56, -2, 8, 9 dup -4, 1k dup -1
size   equ     $ - array
```

Константа *size* содержит число байтов, зарезервированных директивой *db* с адреса *array*. Пусть в регистр *si* записан адрес первого байта массива *array*. Тогда для увеличения каждого элемента массива на 10 следует *size* раз выполнить в цикле пару команд:

```
add    b[si], 10
inc    si
```

Первая команда увеличивает на 10 значение байта, адрес которого находится в регистре *si*. Вторая команда увеличивает содержимое самого *si*, тем самым подготавливая доступ к следующему байту.

Процессор i80x86 способен использовать для косвенной адресации только четыре 16-разрядных регистра: `si`, `di`, `bx`, `bp`. То есть, если команда `inc b[bx]` возможна, то команды `inc b[ax]` не существует.

Примечание

Начиная с i80386, в качестве регистра для косвенной адресации годится также любой из 32-разрядных регистров, например `eax`.

При использовании косвенной адресации имеется возможность задать дополнительное постоянное *смещение*, которое будет прибавляться к значению регистра. Такая адресация называется *косвенной со смещением*.

Пример:

```
inc     b [si+0104]
```

Выражение `si+0104` не может быть вычислено при трансляции, поскольку содержит заранее неизвестную и, кроме того, нефиксированную величину — содержимое регистра `si`. Величина смещения (0104) записывается в конец кода команды (в начале кода записываются сведения о способе адресации и регистре, используемом для адресации). Сумма значения 0104 и содержимого `si` вычисляется процессором при каждом выполнении команды.

Косвенная адресация со смещением позволяет несколько иначе решить задачу с массивом `array`. В качестве постоянной составляющей используется адрес начала массива (определяется именем `array`), а в регистре `si` задается "расстояние" от начала массива. Перед началом цикла `si` обнуляют, и увеличивают его содержимое в каждом цикле на единицу:

```
add     [si+array], 10
inc     si
```

Заметим, что тип первого операнда инструкции сложения определяется в данном случае по типу имени, которым задано смещение — по типу имени `array`.

В языке ассемблера допускаются различные сочетания обозначений постоянной составляющей и регистра, но в любом случае обозначение *регистра* для косвенной адресации должно быть внутри квадратных скобок:

```
inc     array[si]
inc     [si]array
inc     [array+si]
```

Если постоянная составляющая равна нулю, транслятор генерирует команду с косвенной адресацией без смещения.

Пример:

```
inc     b -4[si](2 * bit 1)
```

Постоянная составляющая задана выражением $-4 + 2 * \text{bit } 1$, значение его — ноль. Поэтому транслятор применяет косвенную адресацию по содержимому регистра `si` без смещения.

В рассмотренных примерах обработки массива значение `si` увеличивается с шагом один; тем самым подготавливается доступ к следующему байту. Если требуется обрабатывать массив по словам, то значение регистра `si` в каждом цикле должно увеличиваться на два, командой `add`.

В рассмотренных вариантах косвенной адресации используется один регистр — из набора `si`, `di`, `bx`, `bp`. Наиболее развитый вариант косвенной адресации в `i80x86` — по сумме значений двух регистров; дополнительное постоянное смещение по-прежнему допускается.

Этот способ адресации ограничен четырьмя сочетаниями регистров из набора `si`, `di`, `bp`, `bx`. Перечисленные регистры образуют две группы: `si` и `di` — "индексные", `bx` и `bp` — "базовые"; для задания адреса допускается использование регистров только из разных групп (отсюда и название этого способа адресации — *базово-индексный*).

Примеры:

```
inc    b[si+bx]
inc    w[di][bx]
inc    w[bx+si]
inc    b[bp+di+2]    ; постоянное смещение 2
inc    b[si+di]      ; ?! (два индексных регистров)
```

Базово-индексная адресация обычно используется для доступа к элементам двумерных массивов или массивов структур.

Примечание

Косвенная адресация по сумме 32-разрядных регистров допускается с произвольным сочетанием регистров.

Контрольные вопросы

1. Напишите команды, составляющие тело цикла для инкремента слов, начиная с адреса `array`, с использованием косвенной адресации со смещением.
2. Напишите команды, составляющие тело цикла для инкремента массива слов, с использованием косвенной адресации без смещения.
3. Напишите команды, составляющие тело цикла для увеличения на 20 каждого второго слова последовательности данных с адреса `array`, с использованием косвенной адресации со смещением.
4. Данные заданы следующей директивой:

```
buf    dw        1k dup (1, 3, -1), -3
```

Напишите команды, образующие тело цикла для инкремента *слов* массива `buf`, с использованием косвенной адресации со смещением.

5. Данные заданы следующей директивой:

```
buf      dw      1k dup (1, 3, -1), -3
```

Напишите команды, образующие тело цикла для инкремента *байт*, начиная от адреса `buf`, с использованием косвенной адресации со смещением.

6. Напишите команды, образующие тело цикла для инкремента каждого четвертого байта последовательности данных, с использованием косвенной адресации со смещением и без смещения.

5.3.3. Ограничение на адресацию операндов в памяти

В тех командах, где требуется *явно* обозначать два операнда, в памяти данных может находиться только один из них.

Примеры:

```
mov      w [30], 200
add      bx, w [40]
add      b[0103], [si]          ; ?!
mov      w[si], [di]            ; ?!
```

Это ограничение связано с возможностями машинного представления команд в i80x86: в коде команд не предусмотрено место для задания второго операнда в памяти.

Примечание

Не следует путать — процессор, в принципе, способен адресовать пару операндов в памяти данных. Например, команда `movsw` считывает слово по адресу `[si]` и записывает его значение в слово по адресу `[di]`. Но в команде `movsw` операнды заданы *неявно*, т. е. в коде команды нет информации об адресах операндов.

Контрольные вопросы

Объясните, в чем заключается ошибка в каждой из следующих команд:

☐ `add [si+bx], w[0];`

☐ `inc [si+bx];`

☐ `inc w [bl];`

☐ `inc w[di+ax];`

☐ `add 1, b[si+di];`

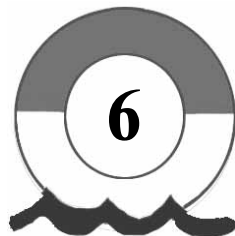
☐ `mov al, w [0];`

☐ `mov ax;`

☐ `add [si], b[di];`

☐ `inc b [bh].`

ГЛАВА 6



Система команд i80x86

...Агабек выхватил из пояса тыквенный кувшинчик с волшебным составом.

— Лимчезу! Пуцугу! Замнихоз — грозно возопил он, брызгая из кувшинчика на стражников. — Каламай, дочимоза, чимоза, суф, кабахас!

— Держи его! Держи! Хватай! Вяжи! Таши! Не пускай!.. — разногласно отвечали стражники своими заклинаниями.

Их заклинания, как и следовало ожидать, оказались неизмеримо могущественнее: через минуту Агабек был повергнут и связан по рукам и ногам.

Л. В. Соловьев. "Повесть о Ходже Насреддине"

Тема этой главы — практическое изучение способов адресации операндов и системы команд семейства микропроцессоров i80x86 в пределах i8086/286.

6.1. Режим непосредственного выполнения в d86

Для изучения машинных команд и способов адресации операндов используется режим *непосредственного выполнения команд* на языке ассемблера a86 в среде отладчика d86.

Обозначение команды на языке ассемблера вводится в командной строке (внизу экрана). Отладчик распознает команды ассемблера по числу букв в обозначении операции: операции отладчика обозначаются одной буквой (например, **k**, **q**), а мнемоники машинных команд — большим количеством (например, **mov**, **in**). Команда ассемблера передается транслятору a86 и, если она задана без ошибок, соответствующая машинная команда помещается в *буфер трансляции* и выполняется. Нажатие клавиши <F3> (**RepeatCmd**) позволяет повторить выполнение команды.

6.2. Способы адресации операндов

Для изучения способов адресации воспользуемся данными, которые определены в программе `exa1.8`.

Листинг 6.1. Данные для изучения способов адресации и системы команд (см. `exa1.8`)

```
(1)  d86_buffer:
(2)          db      20 dup 090
(3)  b_1      db      'AB ab'
(4)  c_1      equ     $ - b_1
(5)  b_2      db      '12 ab'
(6)  b_3      db      c_1 dup '&'
(7)  w_1      dw      1, -1
```

В строке (3) заданы пять байтов по адресам `b_1`, `b_1+1`,... `b_1+4`. В строке (7) определены три слова с адресами `w_1`, `w_1+2`, `w_1+4`. Метка `d86_buffer` в строке (1) предписывает отладчику местоположение буфера трансляции, а в строке (2) резервируется место для буфера. (Определение метки `d86_buffer` в программе не обязательно, но при изучении системы команд позволяет увидеть результат трансляции команды.)

Выполните трансляцию программы `exa1.8` с ключом `+L`. Из файла-листинга выпишите значения имен и запустите отладчик:

```
d86 exa1.com
```

Нажатия клавиш, записанные в `exa1.d8k`, задают отображение значения регистра `al` в двоичном и десятичном форматах, и — отдельно — вывод значения регистра `ax` в десятичном формате.

6.2.1. Регистровая, непосредственная и прямая адресация

Выполните команду:

```
mov ax, 7
```

Результат выполнения — число 7 в регистре `ax`. Местоположение результата (*приемника*) здесь задано при помощи регистровой адресации, а входное значение (*источник*) — непосредственным операндом 7. Обратите внимание на результат трансляции по метке `d86_buffer`. Сколько байтов занимает только что выполненная машинная команда?

Примечание

Команда `int 3` в буфере трансляции — это точка останова, которую отладчик записывает перед тем, как запустить программу с адреса `d86_buffer`.

Выполните команду `mov ah, c_1`. Поскольку имя `c_1` определено константой, значение `c_1` считается непосредственным.

Ассемблер `i86` позволяет задавать в команде `mov` более двух аргументов. Выполните:

```
mov al, dl, cl, 8
```

По результатам трансляции объясните смысл этой *макрокоманды*.

Выполните команды:

```
mov bx, d86_buffer
mov cx, [d86_buffer]
mov dx, offset d86_buffer
```

Значение `d86_buffer` считается непосредственным, т. к. имя `d86_buffer` определено в виде метки. Напротив, `[d86_buffer]` обозначает операнд в памяти, адресация прямая. Выражение `offset <имя>` имеет тип константы — `abs`, значение этого выражения равно значению имени. Какие значения записаны в регистры `bx`, `cx`, `dx`, и почему?

Выполните команды:

```
mov si, w_1
mov di, [w_1]
mov bp, offset w_1
```

Какие значения копируются в регистры `si`, `di`, `bp`, и почему?

Для записи в регистр значения *адреса* данных в памяти рекомендуется использовать команду `lea` (Load Effective Address). Выполните команду:

```
lea ax, w_1
```

Команда `lea` вычисляет адрес данных в памяти и записывает значение адреса в регистр, при этом обращение к данным не предпринимается. Какую адресацию задает обозначение `w_1`? Запишите в `bx` адрес `b_1` двумя способами: посредством `mov` и при помощи `lea`.

6.2.2. Косвенная адресация

Процессор `i80x86` поддерживает косвенную адресацию данных в памяти по значениям одного или двух регистров, в том числе с дополнительным смещением. Для косвенной адресации используются регистры `bx`, `si`, `di`, `bp`. Адрес образуется, в общем случае, суммой содержимого регистров и смещения.

Косвенная адресация по значению одного регистра

Сейчас в регистре `bx` записан адрес `b_1`. В свободном окне отображения памяти задайте спецификацию:

`b5, bx`

Выводятся данные, начиная с адреса, записанного в регистр `bx`.

Выполните команды:

```
inc b[bx]
inc b[bx+1]
```

Первая команда увеличивает на единицу значение байта по адресу `[bx]`. Во второй команде введено смещение, равное 1. В результате значения 'A' и 'B' изменились на 'B' и 'C'. Выполните аналогичные операции над байтами со значениями 'a' и 'b'.

Выполните следующие две команды и объясните результат:

```
mov bx, 2
mov [bx+b_1], '+'
```

Используя регистр `di` и смещение, заданное именем `w_1`, увеличьте на 1 слово со значением `-1` из строки (4). Какое значение должно быть в регистре `di`?

Косвенная адресация по сумме значений двух регистров

При таком способе адресации адрес данных в памяти образуется суммой содержимого двух регистров и, возможно, дополнительного смещения. Допустимые сочетания регистров: `[bx+si]`, `[bx+di]`, `[bp+si]`, `[bp+di]`.

Запишите в `bx` число 2. Выполните следующие команды, объясните результат:

```
lea si, b_1
inc b[bx+si]
```

6.3. Обзор системы команд процессора i80x86

В системе команд процессора i80x86 выделяются следующие основные группы:

- ☐ команды пересылки;
- ☐ арифметические команды;
- ☐ логические команды;
- ☐ команды сдвигов и вращений;

- ☐ команды передачи управления;
- ☐ строковые команды.

6.3.1. Команды пересылки

В группу команд пересылки, кроме рассмотренных команд `mov` и `lea`, входят:

- ☐ команда обмена `xchg`;
- ☐ команда чтения из таблицы (команда *перекодировки*) `xlat`;
- ☐ команда записи в стек `push`;
- ☐ команда чтения из стека `pop`.

Запишите в `ax` число `011ff` и выполните команды:

```
xchg al, ah
xchg ax, dx
```

Команда `xlat` считывает байт из таблицы, адрес которой задан в `bx`, по индексу, заданному в `al`. Результат записывается в `al`. Запишите в `bx` адрес `b_1`, в `al` — число 2, выполните `xlat` и объясните результат. С использованием `xlat` прочитайте значение третьего элемента таблицы `b_1`.

Команда `push` копирует заданный ей операнд размерностью слово в память по адресу `[sp-2]`, после чего уменьшает `sp` на два. Наблюдая за нижним окном отображения памяти (справа от `sp`), выполните команды:

```
push 012
push -1
push dx
```

В результате в нижнем окне отображаются три слова — от адреса, содержащегося в `sp`, до адреса, первоначально записанного в `sp`. Число перед двоеточием — это количество записанных слов. Регистр `sp` содержит адрес последней записи.

Команда `pop` выполняет обратное действие: считывает слово из памяти по адресу `[sp]` и увеличивает `sp` на два. Выполните команды:

```
pop bx
pop cx
pop dx
```

Первая команда `pop` считывает данные, записанные последней командой `push`; последняя команда `pop` считывает данные, записанные первой командой `push`. То есть, чтение данных происходит в очередности, обратной порядку записи, по принципу *стека*. Вершина стека — это данные по адресу `sp` (указатель стека).

Пользуясь стеком, обменяйте значения регистров `si` и `di`.

6.3.2. Арифметические команды

К арифметическим командам относятся:

- ☐ команда сложения `add`;
- ☐ команда вычитания `sub`;
- ☐ команда инкремента (увеличение на 1) `inc`;
- ☐ команда декремента (уменьшение на 1) `dec`;
- ☐ команда арифметической инверсии `neg`;
- ☐ команды умножения `mul/imul`;
- ☐ команды деления `div/ldiv`;
- ☐ команды расширения знака `cbw/cwd`.

Запишите в регистр `ax` число 7. Выполните команды:

```
sub  al, 6
neg  al
add  al, 3
inc  al
dec  al
```

Запишите в `bl` число 3 и выполните команду `mul bl`. Эта команда перемножает беззнаковые числа, заданные регистрами `bl` и `al`; результат помещается в `ax`. Команда `imul bl` выполняет аналогичные действия, рассматривая значения сомножителей как знаковые.

Проверим разницу между `mul` и `imul`. Байт со значением `11111111h` задает беззнаковое число 255 и знаковое число -1 . Запишите это значение в `al` и `bl`, и выполните `mul bl`. С теми же исходными значениями выполните команду `imul bl`.

Задайте в `bh` число 3, в `ax` — число 22. Выполните команду `div bh`. Эта команда делит беззнаковые числа, содержащиеся в `ax` (делимое) и `bh` (делитель). Целая часть частного помещается в `al`, остаток — в `ah`. Проверьте разницу между беззнаковым (`div`) и знаковым (`ldiv`) делением, задав делимое 22 и делитель -3 . Какое беззнаковое число соответствует знаковому -3 в диапазоне байта? Какой знак у остатка — такой, как у делимого, или такой, как у делителя?

Если операнд команд умножения и деления задан словом, то в операции кроме регистра `ax` участвует также `dx`. Запишите в `ax` и `bx` число 2. Выполните команду `mul bx`. Эта команда перемножает значения `bx` и `ax` и помещает результат в пару `dx, ax`: старшая часть результата — в `dx`, младшая — в `ax`.

Выполните команду `div bx`. Эта команда делит 32-разрядное число, заданное парой регистров `dx` (старшая часть) и `ax` (младшая часть), на значение регистра `bx`. Частное помещается в `ax`, остаток — в `dx`.

Команда `cbw` (Convert Byte to Word) предназначена для копирования значения `al` в `ax` с сохранением знака: знаковый бит числа в регистре `al` распространяется на все биты регистра `ah`. Запишите в `al` число `-2` и выполните команду `cbw`. Выполните `neg ax` и вновь — команду `cbw`. Объясните результаты и аналогично выясните действие команды `cwd` (Convert Word to Doubleword).

6.3.3. Логические команды

Команда логического отрицания `not` инвертирует все разряды операнда. Выполните команду `not al`.

Команды `and`, `or`, `xor` выполняют поразрядные логические операции "и", "или", "исключающее или". Операция производится параллельно над всеми парами одноименных разрядов источника и приемника.

Команда `and` используется как *выключатель* — для сброса в ноль тех разрядов приемника, которые сброшены в источнике (значение в источнике команды `and` называется *маской*). Запишите в регистр `al` число `11011xb`. Выполните команду `and al, 1100xb`. Запишите в регистр `al` число `-1` и выполните команду `and al, not (bit 1)`.

Команда `or` применяется как *включатель* — для установки в единицу тех разрядов приемника, которые установлены в источнике. Выполните команду `or al, bit 0`.

Команда `xor` используется как *переключатель* — для инвертирования тех разрядов приемника, которые установлены в источнике. Выполните несколько раз команду `xor al, bit 7` (для повтора используйте клавишу `<F3>`). Бит 7 меняет значение, остальные биты не меняются. Составьте и выполните команду для одновременного переключения разрядов 1, 3 и 4 регистра `al`. Составьте и выполните команду `xor`, которая инвертирует все разряды регистра `al`.

Если команде `xor` в качестве приемника и источника задан один и тот же регистр, то, очевидно, результат выполнения — ноль. Составьте и выполните команду `xor` для обнуления регистра `ax`.

6.3.4. Команды сдвигов и вращений

Запишите в `al` число `bit 0`. Выполните команду `shl al, 1` (логический сдвиг влево). Младший бит результата принимает значение 0, старший бит исходного значения теряется. Повторите эту команду до обнуления регистра `al`.

Задайте в `al` число 201. Выполните восемь раз команду `shr al, 1` (логический сдвиг вправо). Как изменяется младший бит результата? Что записывается в старший бит?

Команда арифметического сдвига на один разряд вправо/влево выполняет функцию деления/умножения знаковых (!) чисел на 2. Запишите в `al` число `-40` и выполните команду `sar al, 1`. Для выяснения абсолютного значения результата воспользуйтесь командой `neg al`. Установите `al` равным `20` и еще раз выполните `sar al, 1`. Что записывается в старший бит результата?

Команда арифметического сдвига влево `sal` не отличается от команды `shl` (разные обозначения одной и той же операции). Выполните команду `sal al, 1` и проверьте содержимое буфера трансляции.

Установите в `al` значение `080`. Выполните по 8 раз команды циклических сдвигов `ror al, 1` и `rol al, 1`. Как действуют эти команды?

Число сдвигов может быть задано либо константой, либо текущим значением регистра `cl`. Запишите в `cl` число `4` и выполните команды:

```
rol al, cl
sar al, 2
```

Первая команда меняет местами тетрады регистра `al`. Какое действие выполняет вторая команда?

6.3.5. Команды передачи управления

При выполнении программы регистр `ip` содержит адрес текущей машинной команды. При обработке текущей команды значение `ip` автоматически увеличивается на длину команды, и это новое значение становится адресом следующей исполняемой команды. В результате, команды отрабатываются по последовательно возрастающим адресам. Переход от одной последовательности команд к другой выполняется командами *передачи управления*, которые принудительно устанавливают значение `ip`.

Выполнение такой команды в непосредственном режиме `d86` означает передачу управления в обход точки останова `int 3` в буфере трансляции, и катастрофу. Чтобы программа остановилась по адресу перехода, необходимо по этому адресу поставить точку останова. Для исследования переходов воспользуемся программой `exa2.8`, в которой заранее запрограммированы семь точек останова по меткам `m1, m2, ..., m7`.

Листинг 6.2. Программа для исследования команд переходов (см. `exa2.8`)

```
_labels macro
#rx17
m#nx:  int      3
#em

        _labels

_w      dw      m4

d86_buffer:
```

В пределах этой программы проверка выполнения команд переходов — по адресам `m1`, `m2`, ..., `m7` — безопасна. (Слово `_w` со значением адреса `m4` понадобится нам для изучения *косвенных переходов*.)

Выполните трансляцию этой программы и запустите ее отладку. Нажатия клавиш, сохраненные в файле `exa2.d8k`, задают отображение адреса `_w` и данных по этому адресу.

Адресация в командах передачи управления

В группу команд передачи управления входят:

<code>jmp</code>	<code>r16/m16/imm</code>	— безусловный переход
<code>call</code>	<code>r16/m16/imm</code>	— вызов подпрограммы
<code>ret</code>		— возврат из подпрограммы
<code>ret</code>	<code>imm</code>	— возврат из подпрограммы с очисткой стека
<code>j<x></code>	<code>imm</code>	— переход по условию <code>x</code>
<code>loop<x></code>	<code>imm</code>	— переход по условию <code>(--cx != 0) && (x)</code>

Обозначения `r16`, `m16`, `imm` — это соответственно 16-разрядный регистр, слово в памяти и непосредственное значение (*immediate*). Заметим, что для адресации источника в командах `jmp` и `call` может быть использован *любой* из существующих методов адресации. Напротив, команды перехода по условию допускают только непосредственную адресацию.

Рассмотрим применение *регистровой* адресации в команде `jmp`. Запишите в `ax` значение `m3` и выполните команду `jmp ax`. В результате в `ip` записано значение из регистра `ax`.

Проверим выполнение `jmp` при адресации данных в *памяти*. В переменной `_w` задано значение метки `m4`. Выполните команду `jmp _w`. В результате в `ip` записано значение, заданное в переменной `_w`.

Наиболее часто в командах перехода используется *непосредственная* адресация. Выполните команду `jmp m5`. В `ip` записано значение `m5`.

Примечание

Чтобы выяснить, куда направлен переход, следует мысленно заменить команду `jmp <adr>` на несуществующую команду `mov ip, <adr>`. Тогда, например, команда `jmp ax`, которая задает нечто странное — "прыжок на регистр", — становится просто-напросто "командой" `mov ip, ax` — передачей содержимого `ax` в `ip`. Путаница возникает из-за того, что адресация здесь формально *регистровая*, а переход получается *косвенным*, поскольку его адрес задан содержимым регистра.

Команда `call` — это команда `jmp`, которая сохраняет в стеке адрес продолжения той последовательности команд, откуда было передано управление. Команда `ret` действует так, как действовала бы несуществующая инструк-

ция `pop ip` — считывает значение с вершины стека в `ip`. В результате (если между выполнением команд `call` и `ret` значение `sp` не меняли), по команде `ret` возобновится выполнение последовательности, прерванной при отработке `call`.

Выполните команду `call m2`. Значение `ip` стало равным `m2`, а в стек записано значение адреса команды `int 3`, которая в буфере трансляции следует за командой `call`. Чтобы убедиться в этом, переместитесь клавишей <Page Down> к метке `d86_buffer`.

Выполните макрокоманду `push m7, m6, m5`. В стеке сейчас записано всего четыре слова. Дважды выполните `ret`, а затем — команду `ret 2`. Необязательный непосредственный операнд команды `ret` задает приращение `sp` после чтения из стека. Как изменялись значения `ip` и `sp`?

Команды условных переходов

Команда условного перехода `j<x>`, или *ветвления*, выполняет запись непосредственного операнда в `ip` при выполнении условия `<x>`. Если условие не выполняется, то перехода нет, и команда равносильна пустой операции.

Запишите в `cx` ноль и выполните команду `jcxz m1`. Условие перехода `<cxz>` читается как `CX Zero`: `cx = 0`. Запишите в `cx` число 2, выполните `jcxz m3`.

Условия для большинства ветвлений основаны на значениях *флагов процессора*. Значения флагов устанавливаются в результате выполнения арифметических и логических команд, команд сдвига и вращения. Значения флагов характеризуют результат *в целом*, например, единичное значение флага `z` (`Zero`) свидетельствует о том, что результат равен нулю.

Значения флагов в `d86` отображаются над вторым столбцом регистров. Флаги, установленные в единицу, отображаются буквами:

- o — флаг переполнения (`Overflow`)
- s — флаг знака (`Sign`)
- z — флаг нулевого результата (`Zero`)
- a — флаг промежуточного переноса (`Aux carry`)
- e — флаг четности (`parity Even`)
- c — флаг переноса/заема (`Carry`)
- d — флаг направления (`Downwhere direction`)
- i — флаг разрешения внешних прерываний (`Interrupt enable`)

Флаги, сброшенные в ноль, не отображаются.

Примечание

Флаг `a`, используемый в операциях над двоично-десятичными числами, рассматривается в *части II*, управляющий флаг `i` — в *части III*, управляющий флаг `d` — в *разд. 6.3.7*.

Запишите в `al` число 1. Выполните команду `dec al`. Результат нулевой, и в строке флагов появилась буква `z`. Выполните команду `jz m1`. Условие перехода — Zero: $z = 1$. Еще раз выполните команду `dec al`. Результат равен 0ff, или -1 . Поэтому $z = 0$ (результат ненулевой), $s = 1$ (результат — отрицательное число). Вычитание единицы из нуля потребовало заема, поэтому $c = 1$.

С каждым из флагов `z`, `o`, `s`, `c` связаны одноименные условия, которые могут быть заданы в командах ветвления. Для каждого из таких условий есть обратное — `nz`, `no`, `ns`, `nc`. Например, команда `jc` задает переход при условии $c = 1$, а команда `jno` — при условии $o = 0$.

Отдельная группа условий отражает результат сравнения двух чисел. Для обозначения отношений `=`, `<>`, `>`, `<`, `>=`, `<=` используются буквы из табл. 6.1.

Таблица 6.1. Обозначения арифметических отношений

Обозначение	Расшифровка	Значение
<code>b</code>	Below — ниже	Беззнаковое "меньше"
<code>a</code>	Above — выше	Беззнаковое "больше"
<code>e</code>	Equal — равно	Любое "равно"
<code>g</code>	Greater — больше	Знаковое "меньше"
<code>l</code>	Less — меньше	Знаковое "больше"
<code>n</code>	Not — не	Префикс "не"

Выполните команду `cmp al, 1`. Эта команда сравнивает значения пары чисел, оценивая разность `al - 1`. В результате устанавливаются флаги, на которых основаны условия `=`, `<>`, `>`, `<`, `>=`, `<=`. В данном случае команда `cmp` сравнила числа 0ff (-1) и 1. Эти числа не равны, поэтому условие `e` (Equal) не выполняется, а обратное условие `ne` (Not Equal) выполняется. Условие `e` эквивалентно условию `z` (равенство чисел означает, что их разность равна нулю). Условие `a` (Above) выполняется, т. к. $0ff > 1$. Также выполняются условия `nb` (Not Below), `ae` (Above or Equal), `nbe` (Not Below-or-Equal). Заметим, что условия `a` и `nbe` эквивалентны. Условие `g` (Greater) не выполняется, т. к. $-1 < 1$. Выполняются условия `l` (Less), `le` (Less or Equal), `ng` (Not Greater), `nge` (Not Greater-or-Equal).

За командой `cmp` обычно следует команда, использующая полученную установку флагов. Проверьте выполнение команд:

```
ja m2
jl m3
jne m4
jna m5
```

Выполните команду `test al` (обратите внимание на буфер трансляции). Ассемблер `i86` допускает сокращенную форму записи команды `test al, al`, которая оценивает поразрядное логическое произведение операндов. Если `al` равно нулю, то и результат операции "и" равен нулю, и флаг `z = 1`. Если `al` не равно нулю, то и результат логического умножения `al` на `al` не равен нулю, и `z = 0`. Выполните команду `test w_1`. Прокомментируйте результат трансляции.

К командам условной передачи управления также относятся команды *циклов* (точнее, они используются для организации счетных циклов). Условие перехода для всех команд этой группы: `--cx != 0`. Содержимое `cx` уменьшается на 1, и переход производится, если результирующее значение `cx` не ноль. Запишите в `cx` число 1, выполните команду `loop m6`. В `cx` ноль, и перехода не было. Повторите эту команду. В `cx` теперь `0ffff`, переход произошел.

Условие перехода для команды `loopz` включает дополнительное условие `z = 1`. Переход произойдет, если `--cx` не ноль и при этом `z = 1`. Если `--cx` равно нулю или `z = 0`, переход отменяется. Команды `loopz` и `loope` эквивалентны. Аналогично составлено условие для команд `loopnz` и `loopne`. Сейчас флаг `z = 0` (в результате выполнения `test w_1`), а `cx = 0ffff`. Выполните команды `loopnz m2` и `loopz m3`. Прокомментируйте результаты.

Рассмотренные команды ветвлений в большинстве используют условия, основанные на флагах процессора. Хотя для установки значений флагов имеются специальные команды `test` и `cmp`, многие другие команды тоже устанавливают флаги — по результатам выполнения основной операции. Заметим также, что некоторые команды используют флаг `c` в качестве операнда. К ним относятся `adc` (сложение с прибавлением переноса), `sbb` (разность с вычитанием заема), `rcl/rcr` (циклический сдвиг, или вращение влево/вправо через флаг `c`).

6.3.6. Воздействие команд на флаги

Значения флагов состояния `c`, `o`, `z`, `s`, `e` устанавливаются большинством арифметических операций, отражая свойства результата:

- ☐ `c` — устанавливается, если *беззнаковый* результат вне диапазона;
- ☐ `o` — устанавливается, если *знаковый* результат вне диапазона (знаковое переполнение);
- ☐ `z` — устанавливается, если результат — ноль;
- ☐ `s` — устанавливается, если старший (знаковый) бит результата равен 1, т. е. результат — *отрицательное* число;
- ☐ `e` — устанавливается, если младший байт (!) результата содержит четное количество бит, установленных в 1 (четный паритет).

Воздействие команд на флаги состояния, по группам команд, показано в табл. 6.2. (Условные обозначения, принятые в табл. 6.3, раскрыты в табл. 6.3.)

Таблица 6.2. Воздействие команд на флаги состояния

Флаги состояния	o	c	s	z	e
Сложение и вычитание					
ADD ADC SUB SBB					
CMP NEG CMPS SCAS	+	+	+	+	+
Инкремент и декремент					
INC DEC	+	—	+	+	+
Умножение и деление					
MUL IMUL	+	+	?	?	?
DIV IDIV	?	?	?	?	?
Логические операции					
AND OR XOR TEST	0	0	+	+	+
Сдвиги и вращения					
SHL SHR (1)	+	+	+	+	+
SHL SHR (>1)	?	+	+	+	+
SAR	0	+	+	+	+
ROL ROR RCL RCR (1)	+	+	—	—	—
ROL ROR RCL RCR (>1)	?	+	—	—	—

Таблица 6.3. Обозначения, принятые в табл. 6.2

Обозначение	Значение флага
—	Не изменяется
+	Изменяется в соответствии с результатом
1	Устанавливается в единицу
0	Сбрасывается в ноль
?	Непредсказуемое или неопределенное

Инструкции сложения и вычитания воздействуют на все флаги состояния: флаги переполнения o и переноса c показывают, что результат — вне диапа-

зона знаковых или беззнаковых чисел соответственно; флаги знака *s*, нуля *z*, паритета *e* показывают, что результат отрицательный, нулевой, содержит четное количество бит, установленных в 1.

Запустите отладчик с программой `exa1.com`.

Запишите в *ax* и *bx* число 511. Хотя эти числа можно было бы сложить одной командой `add ax, bx`, воспользуемся байтовым сложением, чтобы на примере двойных байтов продемонстрировать сложение с *многократной точностью*.

Выполните команду `add al, bl`. Сложение младших байт *ax* и *bx* ($255 + 255$) вызвало перенос ($c = 1$), который необходимо учесть при сложении старших байт. Для этого используется вариант команды сложения `adc`. Выполните команду `adc ah, bh`. Оцените результат в *ax*. Результат правильный, т. к. при сложении старших байт перенос не зафиксирован. Выполните аналогичным образом сложение чисел -1 и -2 . Какой из результатов правильный — беззнаковый или знаковый?

Вычитание с многократной точностью выполняется аналогично, с использованием команды `sbb`; флаг *c* фиксирует заем.

Инструкции сравнения `cmp`, `cmps`, `scas` действуют подобно команде вычитания, но результат никуда не записывается (`cmps` и `scas` рассматриваются в *разд. 6.3.7*). Инструкции сравнения предназначены только для установки значений флагов.

Инструкции инкремента и декремента воздействуют на флаги состояния так же, как сложение и вычитание, но не изменяют флаг *c*. Запишите в *al* число -1 , сбросьте флаг *c* командой `clc` и выполните `add al, 1`. Восстановите значение *al* и флага *c*, затем выполните `inc al`.

Переполнение в результате арифметической инверсии `neg <arg>` возникает, если операнд — наименьшее из отрицательных чисел в диапазоне *<arg>*. Запишите в *al* число -128 , выполните `neg al`. Флаг *o* = 1, поскольку $+128$ "не помещается" в *al*. Флаг *c* устанавливается, если результат `neg` отличается от исходного значения. Исследуйте поведение флага *c* при выполнении `neg al` с исходными значениями *al* = 0, 1, -128 .

Установка флагов *c* и *o* при выполнении `mul/imul` означает, что получен результат *двойной размерности*. Запишите в *al* число 40 и выполните команду `mul al`. Флаги *c* и *o* установлены в 1, т. к. значащие цифры результата записаны не только в *al*, но и в *ah*. Повторите опыт с *al* = 10. Теперь результат помещается в *al* целиком, поэтому *c* = *o* = 0.

Поскольку результат логической команды — всегда в диапазоне, флаги *c* и *o* этими командами сбрасываются в 0. Флаги *z*, *s*, *e* отражают характеристики результата по общим правилам.

Команды логических сдвигов и команды вращений устанавливают флаг *c* равным значению разряда, выталкиваемого за пределы разрядной сетки. За-

дайте в `al` число 2, установите флаг `c` в единицу командой `stc` и выполните сдвиг `shr al, 1`. Флаг `c` фиксирует значение младшего бита исходного значения. Повторите команду `shr al, 1`.

Проверьте, какой бит определяет значение флага `c` при числе сдвигов больше одного: тот, который выталкивается первым, или тот, который выталкивается последним? Для этого задайте в `al` значение `bit 6`, обнулите флаг `c` командой `cnc` и выполните `shl al, 2`.

Команды арифметических сдвигов можно рассматривать как умножение или деление на степень двойки. Флаги состояния отражают результат по общим правилам, за одним исключением: значение флага `o` не определено, если число сдвигов больше единицы. Результат арифметического сдвига вправо (`sar`) — всегда в диапазоне, поэтому флаг `o` обнуляется.

Флаг `o` при выполнении вращений устанавливается по общим правилам, хотя не вполне понятно, что в этом случае означает арифметическое перемещение.

В завершение рассмотрим команды сохранения и восстановления флагов. Выполните команду `pushf`. Значения флагов в виде 16-разрядного слова записаны в стек. Инvertируйте флаг `c` командой `cnc`, затем выполните команду `popf`. Значения флагов на момент выполнения `pushf` восстановлены.

Команды прямой установки флагов состояния реализованы только для флага `c`. Это — `cnc` (обнуление `c`), `stc` (установка `c = 1`), `cmc` (инверсия `c`). Для флагов управления реализованы команды `cld` (обнуление `d`), `std` (установка `d = 1`), `cli` (обнуление `i`), `sti` (установка `i = 1`).

На флаги состояния не оказывают влияния команды:

- ☐ пересылки;
- ☐ передачи управления;
- ☐ логической инверсии (`not`).

6.3.7. Строковые команды

Строковые команды предоставляют специальные возможности для поэлементной (последовательной) обработки массивов байт и слов. Строковые команды реализуют операции чтения (`lods`), записи (`stos` и `movs`), сравнения (`scas` и `cmps`). Текущий элемент массива-источника адресуется регистром `si` (Source Index), текущий элемент массива-приемника — регистром `di` (Dest Index); если в операции участвует только один массив, то второй операнд — или `al`, или `ax`. После выполнения операции значение индексного регистра автоматически изменяется на 1 или 2 в зависимости от размерности операции. Изменение положительное, если флаг `d` (Downwhere direction) сброшен, и отрицательное, если `d = 1`. Размерность операции задает суф-

фикс b/w мнемоники команды (по умолчанию, т. е. без суффикса, операция байтовая).

Выполните команду `lea si, b_1`. В свободных окнах отображения памяти задайте спецификации:

```
a, b_1
a, b_2
a, si
a, di
```

Сбросьте флаг `d` командой `cld`. Выполните команду `lodsb`. В `al` записан байт из памяти по адресу `[si]`, после чего `si` увеличилось на 1. Повторите два раза команду `lodsb`, наблюдая за `al` и `si`, затем установите обратное направление командой `std` и еще пару раз выполните команду `lodsb`.

Запишите в `di` адрес `b_1`, в `al` — ASCII-код `'*`, установите прямое направление и выполните команду `stosb`. Значение из `al` записано в байт по адресу `[di]`, сам `di` увеличен на 1. Запишите в `ax` число `'1'` by `'2'` и выполните команду `stosw`. Как изменилось содержимое массива `b_1`, значение регистра `di`?

Запишите в `di` адрес `b_2`, в `al` — ASCII-код пробела. Выполните команду `scasb`. Эта команда сравнивает значения `[di]` и `al`, устанавливая флаги подобно команде `cmp`. Повторите `scasb`, пока значения `[di]` и `al` не совпадут (флаг `z` = 1).

Восстановите в `di` адрес `b_2`, в `si` запишите адрес `b_1`. Выполните один раз команду `movsb`. В чем заключается ее действие?

Выполните команду `cmps`. Эта команда сравнивает значения `[di]` и `[si]`, устанавливая флаги состояния. В строках `b_1` и `b_2` сейчас должны совпадать последние байты `'b'`. Наблюдая за значением флага `z`, повторите `cmps` до совпадения значений `[di]` и `[si]`.

Рассмотренные команды закливаются за счет использования *префиксов повторения* `rep<x>`. Префикс повторения `rep<x>` перед строковой командой блокирует автоматическое увеличение `ip`, пока выполняется условие, аналогичное условию перехода для команды `loop<x>`.

Запишите в `cx` число `c_1`. Это число байт в массивах `b_1`, `b_2`, `b_3`. В `si` задайте адрес `b_2`, в `di` — адрес `b_3`. Выполните команду `rep movsb`. Команда с префиксом `rep` может быть представлена в виде:

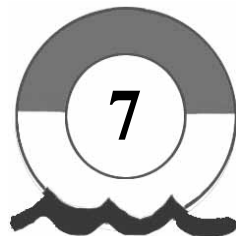
```
11:  jcxz      12
      movsb
      loop   11
12:
```

Префикс `rep` целесообразно использовать с командами `stos` и `movs`. С командами `scas` и `cmps` имеет смысл применять префиксы `repne` (`repnz`) и `repe`

(repz), которые задают дополнительное условие выполнения цикла по значению флага z. Запишите в al код 'b', в di — адрес b_3, в cx — число c_1. Выполните repne scasb. Цикл поиска 'b' завершился успешно, о чем свидетельствует значение флага z. Нулевое значение cx не свидетельствует ни за, ни против: выполнение условия ne могло произойти на *последней* итерации, как в данном случае. Обратите внимание, что di содержит адрес *следующего* байта после найденного. Такой эффект "выноса" указателей характерен для всех строковых команд.

Составьте команды для подготовки и выполнения поиска в строке b_1 байта со значением 0 и байта со значением, отличным от нуля.

ГЛАВА 7



Программирование циклов

В этой главе приведены примеры построения программ с *циклами* и *условным выполнением*. Все примеры на одну тему — поиск в массиве.

7.1. Поиск в массиве байтов

Задан массив байтов; требуется выяснить, есть ли среди них байт со значением 1. Решение приведено в листинге 7.1.

Листинг 7.1. Поиск в массиве байтов (см. find_1.8)

```
        jmp      start                      ; (1)
array   db      20, 3k dup (-1), 1        ; (2)
size    equ     $ - array                  ; (3)

start:                                     ; (4)
        lea     si, array - 1              ; (5)
        mov     cx, size                    ; (6)
l1:      ; (7)
        inc     si                          ; (8)
        cmp     b[si], 1                    ; (9)
        loopne  l1                          ; (10)
        jne     _not_found                  ; (11)
        ...
_not_found:
        ...
```

Выполнение начинается с команды (1). Переход нужен для обхода данных, определенных по адресу `array`. Константа `size` дает размер данных в байтах. Так как в каждом цикле этой программы обрабатывается один байт, `size`

используется в (6) для задания числа повторений цикла. Число повторений записывается в `cx`; этот регистр используется для счета итераций. При каждом выполнении (10) значение `cx` уменьшается на 1 и, если `cx` не стало нулем, выполнение возобновляется с начала цикла (7). Все команды из группы `loop<x>` завершают цикл при обнулении `cx`; команда `loopne` продолжает цикл, если выполняется еще одно дополнительное условие — `ne`. Установка флагов для последующей проверки дополнительного условия выполняется командой (9), которая сравнивает байт по адресу `[si]` с единицей.

В тело цикла (8), (9) включена команда инкремента регистра-указателя, для подготовки доступа к следующему байту массива. Эта команда не должна находиться между (9) и (10), потому что она воздействует на флаг `z`. (Если бы она стояла между (9) и (10), результаты сравнения были бы потеряны.) Поскольку `si` сначала увеличивается, а потом используется для доступа к данным, `si` перед началом итераций устанавливается (5) на 1 меньше.

Выход из цикла по команде (10) может произойти по условию `e` (когда искомое значение найдено) и по условию `cx = 0`. В примере из листинга 7.1 выход произойдет сразу по двум этим условиям, т. к. искомое значение находится в последнем байте массива и обнаруживается на последней итерации. По этой причине результат поиска после завершения цикла выясняется проверкой условия `ne` или `e` — но не `cxz`.

7.2. Поиск в массиве слов

Изменим условия задачи. Задан массив слов; требуется выяснить, есть ли среди них *слово* со значением 1. Решение приведено в листинге 7.2.

Листинг 7.2. Поиск в массиве слов (см. `find_2.8`)

```

    jmp      start                      ; (1)

array dw    20, 3k dup (-1), 1        ; (2) !
size  equ   $ - array                  ; (3)

start:                                     ; (4)
    lea     si, array - 2               ; (5) !
    mov     cx, size / 2                ; (6) !
11:                                     ; (7)
    add     si, 2                       ; (8) !
    cmp     w[si], 1                    ; (9) !
    loopne  11                          ; (10)
    jne     _not_found                  ; (11)
    ...
_not_found:
    ...

```

Изменения в (2), (5), (6), (8) и (9) связаны с удвоением размерности данных: значение адреса в `si` должно теперь увеличиваться с шагом два, что отразилось в (8) и (5); также, счетчик итераций теперь вдвое меньше (6) числа байт.

В листинге 7.3 приведен другой вариант решения задачи о поиске в массиве (на примере поиска слова) — с применением строковой команды `scas`.

Листинг 7.3. Поиск в массиве с применением строковой команды (см. find_3.8)

```
...
lea    di, array                ; (1) !
mov     cx, size / 2            ; (2)
mov     ax, 1                   ; (3) !
cld                                     ; (4) !
l1:                                     ; (5)
scasw                                     ; (6) !
loopne  l1                      ; (7)
jne     _not_found              ; (8)
...
```

Команда (5) сравнивает слово по адресу `di` с содержимым регистра `ax`, сохраняя результат сравнения во флагах, и, при сброшенном флаге направления `d`, увеличивает `di` на 2. Сброс флага `d` выполняется перед началом цикла в (4).

Использование команды `scas` с префиксом повторения `repne` позволяет обойтись без команды `loopne`. Префикс — это байт с уникальным значением, который ставится перед кодом команды. Префикс меняет способ выполнения команды. Префиксы *повторения* действуют лишь на *строковые* команды, запуская их выполнение подобно команде `loop<x>`. Решение задачи о поиске в массиве с использованием строковой команды и префикса повторения показано в листинге 7.4.

Листинг 7.4. Применение строковой команды с `rep`-префиксом (см. find_4.8)

```
...
lea     di, array                ; (1)
mov     cx, size / 2            ; (2)
mov     ax, 1                   ; (3)
cld                                     ; (4)
repne  scasw                    ; (5) !
jne     _not_found              ; (6)
...
```

Команда (5) содержит в себе и тело цикла, и управление циклом.

7.3. Поиск байта со значением больше заданного

Изменим условия исходной задачи. Требуется выяснить наличие байта со значением больше 1, считая данные знаковыми. Решение приведено в листинге 7.5.

Листинг 7.5. Поиск байта со значением больше заданного (см. find_5.8)

```

...
lea    si, array - 1                ; (5)
mov     cx, size                    ; (6)
l1:                                          ; (7)
inc     si                          ; (8)
cmp     b[si], 1                    ; (9)
jg      _found                      ; (10) !
loop    l1                          ; (11) !
_not_found:
...
_found:

```

По сравнению с первым примером изменилось условие поиска: вместо `e/ne` (равно/не равно) теперь используется `g` (Greater — знаковое "больше"). Поскольку в командах `loop<x>` условие `g` неприменимо, выход из цикла при успешном поиске выполняется отдельным ветвлением (10). Чтобы представить себе условие перехода при выполнении пары команд `cmp-j<x>`, мысленно поставьте между операндами команды `cmp` знак из набора `=, <>, >, <, >, <=, >=`, соответствующий условию `<x>`:

```
b[si] > 1 ; g
```

Тело цикла из листинга 7.5 упрощается заменой (8) и (9) на строковую команду, как показано в листинге 7.6.

Листинг 7.6. Краткий вариант программы из листинга 7.5 (см. find_6.8)

```

...
lea     di, array                    ; (1)
mov     cx, size                    ; (2)
mov     al, 1                       ; (3)
cld                                          ; (4)
l1:                                          ; (5)
scasb                                       ; (6) !
jg      _found                      ; (7)
loop    l1                          ; (8)

```

```
_not_found:
    ...
_found:
```

Дальнейшее упрощение цикла невозможно — как не существует команды `loopng`, так не существует и префикса `repng`.

7.4. Подсчет байтов в заданном диапазоне значений

Сформулируем другой вариант исходной задачи поиска: подсчитать число байтов со значением в диапазоне 10—99, результат сформировать в регистре `ax`.

7.4.1. Алгоритмическое решение

В листинге 7.7 приведен первый вариант решения — за счет использования команд сравнения и условных переходов. Проверка принадлежности очередного значения заданному диапазону закодирована в *инструкциях*.

Листинг 7.7. Подсчет байтов со значением в заданном диапазоне (см. `cnt_1.8`)

```
    ...
    lea     si, array                ; (1)
    mov     cx, size                 ; (2)
    cld                                ; (3)
    mov     dx, 0                    ; (4)
11:                                ; (5)
    lodsb                                ; (6)
    cmp     al, 10                    ; (7)
    jb      >l2                       ; (8)
    cmp     al, 99                    ; (9)
    ja      >l2                       ; (10)
    inc     dx                        ; (11)
12:    loop  11                       ; (12)
    mov     ax, dx                    ; (13)
    ...
```

Для уменьшения числа обращений к памяти данных (для ускорения программы) используем строковую команду (6). Инструкция `lodsb` копирует значение байта из `[si]` в `al` и увеличивает `si` (если флаг `d` сброшен). Так как, в результате, регистр `al` занят, для подсчета числа байт в заданном диапазоне временно используем свободный регистр `dx` — в командах (4, 11, 13).

Примечание

Применение префикса повторения с командой `lodsb` хоть и допустимо, но, очевидно, не имеет никакого смысла.

7.4.2. Табличное решение

Программирование *таблиц* позволяет в некоторых случаях полностью исключить команды ветвлений. Логика программы становится проще; также, в общем случае, сокращается время выполнения.

В листинге 7.8 приведен вариант решения задачи о подсчете байтов со значением в заданном диапазоне. Взамен команд сравнений и ветвлений используется таблица, определенная директивами `db`; каждому значению из диапазона 0—255 в этой таблице соответствует байт со значением ноль или один; это число прибавляется к счетчику `dx`. Доступ к таблице выполняется при помощи инструкции `xlat`.

Листинг 7.8. Табличное решение задачи о подсчете байтов в диапазоне (см. `cnt_2.8`)

```

...
p1      equ      10                ; (1)
p2      equ      99                ; (2)
map      db      p1 dup 0          ; (3)
         db      (p2 - p1 + 1) dup 1 ; (4)
         db      256 - ($ - map) dup 0 ; (5)
start:   ; (6)
         lea      si, array         ; (7)
         mov      cx, size          ; (8)
         lea      bx, map           ; (9)
         cld                     ; (10)
         mov      ax, dx, 0         ; (11)
11:      ; (12)
         lodsb                    ; (13)
         xlat                     ; (14)
         add      dx, ax            ; (15)
         loop     11               ; (16)
         mov      ax, dx            ; (17)
...

```

Таблица `map` (3, 4, 5) задает 256 байт, из них в единицу установлены элементы с номерами от 10 до 99. Чтение выполняется командой `xlat` (14), которая адресует данные в памяти по сумме содержимого `bx` и `al`; адрес таблицы записан (9) в `bx` заранее. Результат чтения помещается в регистр `al`. Пред-

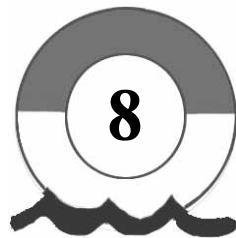
варительно старшая часть `ax` обнуляется (11), чтобы можно было целиком прибавлять `ax` к счетчику `dx` (15).

Запрограммируйте таблицу для решения задачи подсчета числа байт со значением в заданном диапазоне. Во всех вариантах задания, приведенных в табл. 7.1, диапазон составлен из нескольких поддиапазонов и/или отдельных значений.

Таблица 7.1. Варианты задания

Вариант	Значения
1	10—99, 101—122
2	'A'—'Z', 'a'—'z'
3	'A'—'Z', 'a'—'z', '0'—'9'
4	0, 5—255
5	0, ' ', 255
6	1—9, 11—19, 21—29 и т. д. до 241—255 включительно
7	1—3, 11—13, 21—23 и т. д. до 251—253 включительно

ГЛАВА 8



Исследование программ в d86

В этой главе рассмотрены средства отладчика d86 для исследования исполняемых программ.

Отладчик d86, как все отладчики, позволяет отрабатывать программу по частям между точками останова. За счет перерывов в отработке возникает возможность проследить порядок выполнения команд, и — в точках останова — оценить и изменить состояние вычислительного процесса (значения регистров, флагов, данных в памяти). Отладчик d86 позволяет выполнять фрагменты программы даже в произвольном порядке, за счет изменения значения регистра `ip` в точках останова. Также, имеется возможность на время текущего сеанса отладки вносить дополнения и исправления в машинные команды и данные в памяти — режим "заплаток" (Patch).

После прочтения данной главы рекомендуем выполнить в отладчике примеры программ из *главы 7* и проверить правильность выполнения задания из *разд. 7.4.2*.

8.1. Пример исследуемой программы

Программа `exa1.8` из листинга 8.1 выводит число на экран: по значению в регистре `ax` подпрограмма `wrInt` формирует массив байт, содержащий ASCII-коды цифр, которыми это число представлено в системе счисления по основанию, заданному в `bx`. После выполнения `wrInt` главная программа отображает полученный массив на экране.

Перед обращением к `wrInt` главная программа устанавливает входные параметры: `ax = 138`, `bx = 10` (т. е. задан вывод числа 138 как десятичного). В подпрограмме `wrInt` число повторно делят на основание системы счисления, собирая результат из остатков. Очередной ASCII-код цифры получают

как сумму остатка и ASCII-кода '0'. Процесс повторяется, пока результат деления не станет нулем.

Таблица 8.1. Получение десятичных цифр числа 138

Делимое	Делитель	Остаток	Код цифры
138	10	8	'8'
13	10	3	'3'
1	10	1	'1'
0			

Цифры результата, показанные в табл. 8.1, формируются, начиная с наименее значимых (сначала единицы, потом десятки, и т. д.). Если порядок получения цифр обратный, то и вывод цифр на экран следует выполнять в обратном порядке. В варианте решения, приведенном в листинге 8.1, цифры записываются в обратном порядке — от конца массива к началу; после завершения циклов преобразования полученная строка выводится в прямом порядке.

Вывод выполняется функцией 9 DOS; для этой функции адрес строки задается в dx, а конец строки ограничивается литерой '\$' (в рассматриваемой программе литера '\$' в конце массива задана изначально). Первый параметр #1 макрокоманды по имени dos (см. файл exal.inc) — это номер функции DOS. Макрокоманда clr используется для обнуления регистра, заданного первым параметром.

Листинг 8.1. Программа вывода значения числа на экран (см. exa1.8)

```
include exal.inc

        jmp      start

buf      db      16 dup ?
buf_end db      '$'

wrInt:
        push    di, dx, ax
        clr     di                ; индекс := 0

l1:
        clr     dx                ; старшая часть делимого
        div     bx                ; (dx,ax) / (bx) -> частное в ax
                                   ;                   -> остаток в dx
        add     dl, '0'          ; ASCII-код цифры
```

```

dec      di                ; di = -1, -2, ...
mov      buf_end[di], dl ; запись цифры в массив
test     ax
jnz      ll                ; если частное <> 0, продолжить

lea      dx, buf_end[di] ; адрес строки для вывода на экран
dos      9                 ; вывод

pop      ax, dx, di
ret

```

start:

```

mov      ax, 138           ; число для вывода
mov      bx, 10            ; основание системы счисления
call     wrInt
nop                                             ; эта команда нужна только для
                                             ; изучения остановов в d86
int      020              ; возврат в DOS

```

Выполните трансляцию программы exa1.8 и запустите отладчик:

d86.com exa1.com

8.2. Названия регистров в отображении данных

В начале сеанса отладки отработаны нажатия клавиш из файла exa1.d8k, для настройки отображения данных.

Обратите внимание на спецификации отображения данных. Адрес, с которого начинается отображение в окне 5 отладчика, образуется суммой постоянной величины `buf_end` и содержимого регистра `si`. При изменении значения `si` отображение будет сдвигаться. В общем случае, в d86 адрес данных в окне отображения можно задать числом, именем, обозначением регистра (допускается любой 16-разрядный регистр, а не только `si`, `di`, `bx`, `bp`), а также суммой имени (или числа) и регистра.

В первых строках заданы отображения содержимого регистров. Префикс = означает, что содержимое регистра задает не адрес, а само значение отображаемых данных. В этом случае задано отображение регистров `ax` и `dx` в десятичной системе счисления.

8.3. Режимы выполнения

Выполните один шаг (**SingStep**). Отработана команда `jmp start`. Красный курсор, отмечающий следующую выполняемую команду, перешел на метку `start`. Пройдите по шагам до первой команды подпрограммы (`push di`).

Переместите стрелками красный курсор вниз к команде `ret` и выполните команду отладчика **GoTillHere**. По этой команде на месте курсора ставится временная точка останова, и управление передается на команду, следующую за выполненной (в данном случае точка останова ставится на место инструкции `ret`, а управление передается на команду `push di`). Временная точка останова сбрасывается при остановке программы, независимо от причины останова.

В данном случае причина остановки — достижение временной точки останова, поставленной на команде `ret`. Выполните три следующие команды по шагам. Выполнение `int 020` блокируется отладчиком, поскольку эта инструкция распознается им как возврат в DOS.

Для возвращения в начало программы нажмите клавишу `<Home>`. Пройдя два шага, еще несколько раз нажмите клавишу `<Home>`. Клавишами-стрелками переместите курсор на какую-нибудь другую команду и повторите нажатия клавиши `<Home>`. По этой клавише указатель инструкции `ip` переключается между точкой входа в программу (0100) и командой, которая следует за последней выполненной. Заметим, что перемещение курсора клавишами-стрелками не выполняет команд; хотя значение `ip` при этом изменяется.

Переместите курсор к команде `dec di`. Выполните ее в пошаговом режиме. Теперь эта команда считается последней выполненной, и при нажатиях клавиши `<Home>` значение `ip` будет изменяться между адресом точки входа в программу и адресом команды `mov buf_end[di], dl`. Проверьте.

Вернитесь в начало программы. Найдите, перелистывая окна подсказки, команду **Go** (запуск, или продолжение). Нажмите клавишу `<g>`, затем пробел. Подсказка сообщает, что в команде **Go** можно указать временную точку останова. Пока не будем использовать эту возможность. Завершите ввод нажатием клавиши `<Enter>`. Так как во всей программе нет ни одной точки останова, выполнение прекратилось на инструкции выхода в DOS.

Еще раз перейдите к началу программы и выполните команду **Go**, задав временную точку останова `wrInt`. После остановки в начале подпрограммы в стеке записан адрес команды `nop`, следующей за `call`. Выполните команду отладчика **TrapRet**. По этой команде временная точка останова ставится по адресу, записанному на вершине стека.

Вновь вернувшись к началу, отработайте в пошаговом режиме все команды до `call`. Затем выполните команду отладчика **ProcStep**. Подпрограмма отработана целиком, за счет установки временной точки останова на месте `nop`.

8.4. Постоянные точки останова

Вернитесь к началу программы. Найдите команду **Set permanent breakpoint**, введите ее начальную букву **b** и пробел. Обратите внимание на окно подсказки. Эта команда позволяет изменить положение двух постоянных точек останова: если задать одну, то будет перезаписана та точка, которая была установлена прежде всех, а если не задать ничего, то обе точки будут сняты.

Задайте точку останова по адресу 010. Проверьте ее установку, вызвав отображение окна состояния комбинацией клавиш <Ctrl>+<S>. Сверяясь с окном состояния, задайте точку останова по адресу 020, затем по адресу 030. Временные точки не влияют на состояние постоянных точек. Для проверки выполните команду **g 040**, и отобразите окно состояния. Задайте команду **b** без аргументов, проверьте состояние.

8.5. Редактирование команд

Перейдите к инструкции `mov ax, 138` и выполните команду **Patch** (вход в режим "заплат"). Введите поверх команды `mov ax, 138` команду `mov ax, 125xq`. На следующей строке, куда переместился курсор после завершения ввода, задайте `mov bx, 8`. Для выхода из режима "заплат" нажмите клавишу <Esc>. Выйдите в начало программы и выполните пуск командой **Go**. Объясните результат.

8.6. Принудительный останов

В отладчике d86 специальных средств для остановки заиклившейся программы не предусмотрено. Тем не менее, характерная для всех отладчиков реакция на машинную инструкцию `int 3` позволяет "разбудить" d86 извне.

До запуска отладчика следует установить резидентную программу `brk.com` (рекомендуем записать ее вызов в `autoexec.bat`). Программа `brk.com` перехватывает аппаратное прерывание от клавиатуры, и при нажатии комбинации клавиш <Ctrl>+<Gray-> (<Gray-> — минус на дополнительной клавиатуре) выполняет команду `int 3`, которая предшествует команде `iret` возврата из прерывания. При отработке `int 3` управление передается отладчику.

Примечание

Резидентные программы, прерывания и, в частности, отладочные прерывания подробно рассмотрены в *части III*.

Для проверки принудительного останова перейдите в начало программы и введите на месте команды `jmp start` команду `jmp 0100`. Запустите програм-

му. Для прекращения бесконечного цикла нажмите комбинацию клавиш `<Ctrl>+<Gray->`. Поскольку отладчик блокирует выполнение `int 3`, после останова нужно обойти эту команду нажатием клавиши "стрелка-вниз", а затем выполнить `iret` в пошаговом режиме.

Восстановите `jmp start` по адресу 0100 и перейдите к команде `int 020`. Запишите на ее месте `jmp start`. Перейдите в начало, запустите программу, затем остановите ее. После выполнения `iret` управление передано в неизвестную программу. Очевидно, нажатие комбинации клавиш `<Ctrl>+<Gray->` прервало выполнение функции DOS (в данном случае — вывод на экран функцией 9). В подобных ситуациях после принудительного останова лучше завершить сеанс отладки командой **q**.

ГЛАВА 9



Примеры программ

Как? Разве я недостаточно упражняюсь? Прежде чем встать, я раз семь перевернусь с боку на бок. Неужели этого мало?

Ф. Рабле. "Гаргантюа и Пантагрюэль"

В этой главе рассматриваются, на примерах, следующие темы:

- ☐ обработка данных на уровне бит;
- ☐ составление программ с вложенными циклами;
- ☐ ввод исходных данных и вывод результатов;
- ☐ проблема опережающих ссылок.

9.1. Обработка данных на уровне бит

Задача: задан массив байт, для каждого байта требуется вычислить количество бит, установленных в 1, и записать полученное число на место исходного значения.

Сначала рассмотрим отладочный вариант решения — без ввода/вывода. Исходные данные определены в самой программе (с адреса `buf`), поэтому длина массива известна *заранее* (константа `size`).

Листинг 9.1. Отладочный вариант решения задачи о подсчете байтов (см. `var_1.8`)

```
        jmp     start

buf     db      bit 0, bit 1 + bit 7, -1, 0
size    equ     $ - buf

start:
        lea     si, buf
```

```

        mov     di, si
        cld
        mov     cx, size

11:     lodsb                    ; очередной байт — в al
        mov     ah, 0           ; счетчик бит := 0
        push    cx              ; сохранить счетчик внешнего цикла
        mov     cx, 8           ; установить счетчик вложенного цикла

12:     shr     al, 1
        adc     ah, 0           ; если флаг c = 1, то ah := ah + 1
        loop    12
        pop     cx              ; восстановить счетчик внешнего цикла

        mov     al, ah
        stosb                    ; счетчик бит — в массив
        loop    11
        int     020             ; выход в DOS

```

Последовательный доступ к битам `al` выполняется при помощи инструкции сдвига. При сдвиге вправо значение младшего бита копируется во флаг `c`, для увеличения счетчика бит командой `adc`.

Примечание

Процессоры от i80386 предоставляют дополнительные инструкции для доступа к биту слова или двойного слова. В этих инструкциях задается адрес данных и номер бита. Инструкции следующие: `bt` (Bit Test), `btc` (Bit Test & Complement), `bts` (Bit Test & Set), `btr` (Bit Test & Reset). Также имеются инструкции поиска ближайшего бита, установленного в единицу: `bsf` (Bit Scan Forward) и `bsr` (Bit Scan Reverse).

9.2. Вложенные циклы

Цикл от метки `11` до инструкции `loop 11` организует последовательную обработку байтов массива. Внутри этого цикла вложен еще один цикл (от метки `12` до инструкции `loop 12`), где очередной байт разбирают по битам. Этот вариант решения демонстрирует общую схему вложения `loop`-циклов с сохранением и восстановлением `cx`.

Подсчет бит будет выполняться быстрее, если завершать внутренний цикл сразу при обнулении `al`. В этом варианте решения инструкция `loop` не нужна (также — `push` и `pop`, введенные для сохранения и восстановления счетчика внешнего цикла). Внутренним циклом теперь управляют команды `test` и `jnz`.

```

...
12:
    shr    al, 1
    adc    ah, 0
    test   al
    jnz    12
    ...

```

9.3. Программирование ввода/вывода

Введем в программу операции *ввода/вывода*. Предусмотрим варианты ввода как с клавиатуры, так и из файла; аналогично, вывод — и на дисплей, и в файл. Эти варианты программируются одинаково, если воспользоваться *стандартными потоками*.

При запуске программы DOS позволяет связать так называемый стандартный *выходной* поток или с клавиатурой, или с любым файлом. Также, при запуске программы может быть, независимо, настроен стандартный *выходной* поток. Программа во всех вариантах одна и та же, она не зависит от *перенаправления* стандартных потоков, выполняемого операционной системой.

По умолчанию, DOS связывает стандартные потоки с клавиатурой и дисплеем. Связывание стандартных потоков с файлами задается при запуске программы следующим образом:

```
prog <data.in >data.out
```

Символом "<" задано связывание стандартного ввода программы prog с файлом data.in. Перенаправление выходного потока задается символом ">". Файл data.out создается операционной системой заново; если файл с таким именем существует, он урезается до нулевого размера.

Ниже представлен фрагмент программы с чтением из стандартного входного потока.

```

dos    macro
    mov    ah, #1
    int    021

#em

    mov     bx, 0           ; номер потока
    mov     cx, 1k
    lea     dx, buf
    dos     03f             ; чтение
    jc      read_error
    ...

```

Чтение выполняется запросом к DOS (int 021) с номером функции в ah, равным 03f. Для вызова DOS определена макрокоманда dos с одним пара-

метром — номером функции DOS. Перед обращением к `dos 03f` следует задать:

- ❑ в `cx` — максимальное число байт для чтения;
- ❑ в `dx` — адрес массива, куда будут записаны введенные данные;
- ❑ в `bx` — номер потока (ноль — номер стандартного входного потока).

Если при вводе произошла ошибка, флаг `c` устанавливается в 1. Если флаг `c` после выполнения функции `03f` сброшен, то в регистре `ax` записан счетчик прочитанных байтов.

Фрагмент программы для записи в стандартный выходной поток:

```
...
mov     bx, 1           ; номер потока
mov     cx, 1k
lea     dx, buf
dos     040             ; запись
jc      write_error
...
```

Параметры и выходные значения функции `040` (запись) назначены по аналогии с функцией `03f` (чтение):

- ❑ в `cx` — число байт для записи;
- ❑ в `dx` — адрес массива-источника данных;
- ❑ в `bx` — номер потока (единица — номер стандартного выходного потока);
- ❑ при завершении функции `c = 1` означает ошибку;
- ❑ если при завершении `c = 0`, то в `ax` — число записанных байтов.

Разница — лишь в номере функции DOS (`040` — запись, `03f` — чтение) и в номере потока (номер стандартного выходного потока равен 1).

Примечание

Программа имеет возможность получать доступ к файлам, не обращаясь к стандартным потокам, а создавая собственные потоки. Для этого ей потребуются открывать существующие файлы и создавать новые (см. приложение 4).

В следующем варианте программы, приведенном в листинге 9.2, массив исходных данных считывается из стандартного входного потока, результаты преобразований записываются в стандартный выходной поток.

Листинг 9.2. Решение задачи о подсчете байтов с вводом/выводом (см. `var_2.8`)

```
dos     macro
mov     ah, #1
```

```

int      021

#em

mov      bx, 0          ; номер потока
mov      cx, 08000      ; максимальное число байт для чтения
lea      dx, buf        ; адрес массива-приемника
dos      03f
jc       >19            ; ошибка?

mov      cx, ax          ; фактически прочитано
jcxz     >19            ; 0 байт ?

push     cx
call     make            ; вызов п/п обработки
pop      cx              ; сколько прочли, столько и запишем

mov      bx, 1          ; номер стандартного выходного потока
lea      dx, buf        ; адрес данных для записи
dos      040
jc       >19            ; ошибка?

mov      al, 0           ; errorlevel = 0
dos      04c             ; выход в DOS

19:
mov      al, 1           ; errorlevel = 1
dos      04c             ; выход в DOS

make:
lea      si, buf
mov      di, si
cld

11:
lodsb    ; al ← очередной байт массива
mov      ah, 0           ; счетчик единичных бит

12:
shr      al, 1           ; младший бит — во флаг "с"
adc      ah, 0
test     al              ; осталось еще что сдвигать?
jnz      12

mov      al, ah
stosb    ; записать счетчик бит
loop     11
ret

buf      db              ; место для массива

```

Обратите внимание — память для массива в программе не резервируется. Имя `buf` определяется в конце программы, обозначая начало массива. По-

сле загрузки `com`-программы между `buf` и стеком свободны по крайней мере 08000 байт, точнее, 0fffe—0200—`buf` (0fffe — начальное значение `sp`, 0200 — место под стек, с запасом).

Примечание

Использование памяти за пределами программы (но в границах выделенного при загрузке блока памяти) допустимо только для `com`-программ. При составлении `exe`-программ место для массивов следует резервировать в самой программе, т. к. взаимное расположение объектов исполняемой `exe`-программы заранее неизвестно.

9.4. Проблема опережающих ссылок

Имя `buf` определено в конце исходного текста, все ссылки на него — *опережающие*. Рассмотрим проблемы, которые могут возникнуть при наличии таких ссылок.

Если в инструкции встретилось неизвестное имя, транслятор на первом просмотре попытается определить тип имени по контексту.

```
mov    foo, ax           ; word
mov    oof, 1            ; byte/word (5/6)
mov    al, bee           ; abs/byte (2/3)
jmp    start             ; word/abs long/abs short (4/3/2)
```

Очевидно, что тип `foo` — `word`. Также понятно, что `oof` — переменная, неизвестно лишь какого типа — `word` или `byte`. Тип `bee` может быть `abs`, а может быть `byte`. Тип `start`, как ни странно, тоже неизвестен. Это либо `abs` (если `start` — метка), либо `word` (если `start` — слово, в котором задан адрес перехода).

В чем тут проблема? Зачем вообще знать тип имен на первом просмотре?

На первом просмотре транслятору необходимо зарезервировать место для каждой машинной инструкции, а длина ее как раз зависит от *типов* операндов. Если тип точно не известен, то и длина команды определена приблизительно, в каком-то диапазоне. (В примере возможное число байт для кода команды показано в скобках.)

Команды `jmp` вносят еще большую неопределенность. Если операнд `jmp` имеет тип `abs`, возможны два варианта кодировки дистанции (т. е. смещения адреса перехода относительно адреса самой команды `jmp`). Вариант `short` кодируется парой байт (смещение в пределах -128 — $+127$ задано одним байтом), а вариант `long` — тремя байтами (смещение кодируется словом).

Если транслятор не в состоянии точно определить длину команды по контексту, в котором появилось неизвестное имя, он выбирает один из вариантов по некоторой схеме.

Для операнда `jmp` все ассемблеры выбирают наиболее вероятный тип `abs`, а дальность перехода выбирается с запасом — `long`. В результате, переходы вперед зачастую кодируются с избытком, поскольку в программах управление чаще всего передается на небольшое расстояние.

В остальных случаях, не связанных с переходами, `a86` выбирает самый *короткий* вариант кодирования, окончательно определяя для себя тип неизвестного имени. Если последующее определение имени опровергает выбор, сделанный транслятором, ему приходится завершать работу с сообщением об ошибке: "Conflict with Previous Definition".

Трансляторы `masm/tasm`, в отличие от `a86`, резервируют место *по максимуму*. Если ко второму просмотру обнаруживается избыток, его приходится заполнять однобайтной инструкцией `nop`. Вместе с тем, при такой стратегии можно не спешить с окончательным выбором типа неизвестного имени. Поэтому при использовании `masm/tasm` последующее определение имени гораздо реже приводит к конфликтам.

```

mov      oof, 1
...
oof      db      ?

```

Приведенный выше фрагмент `a86` транслирует без ошибок и без избыточности, а `masm` и `tasm` после пяти байт кода инструкции `mov` вынуждены ставить `nop` — в лишнем шестом байте. Следующий фрагмент транслируется `masm/tasm` без избыточности (их наихудшие предположения подтвердились), но `a86` выдаст ошибку (не подтвердилось наилучшее предположение).

```

mov      oof, 1
...
oof      dw      ?

```

Какой вариант предпочтительней? Преобразования, выполняемые ассемблером, должны быть *предсказуемы*. Ассемблер не должен пытаться имитировать свойства языков высокого уровня, к чему склоняются разработчики `masm` и `tasm`. Что если `nop`-заплаты окажутся в теле цикла, критического по времени выполнения?

9.5. Решение проблемы опережающих ссылок

Проблемы, возникающие в связи с опережающими ссылками, решаются одним способом: такие ссылки надо ликвидировать, например, за счет *предварительного описания типа* имени при помощи директивы `extrn`:

```
extrn    buf:byte
```

или сокращенно

```
extrn  buf:b
```

Объявление `extrn` говорит о том, что имя `buf` имеет тип `byte`, а определение `buf` находится в произвольном модуле многомодульной программы (в частности, в этом же самом модуле).

Избежать избыточности при кодировании переходов вперед можно также за счет явного задания дальности перехода — ключевым словом `short` — после каждой (!) мнемоники `jmp` (например, `jmp short start`). Такой способ в `tasm/tasm` — единственный.

В языке `a86` проблема программирования коротких переходов вперед с легкостью решается при помощи локальных имен. Переход считается коротким, если операнд `jmp` задан локальным именем.

Примечание

До появления `i80386` избыточность могла возникнуть только при кодировании `jmp`, поскольку лишь эта команда перехода допускала варианты `short` и `long`. Начиная с `i80386` уже не только `jmp`, но все команды ветвлений могут быть представлены в двух формах — `short` и `long`. Следовательно, выигрыш от применения локальных меток в 32-разрядном варианте `a86` — в ассемблере `a386` — возрастает еще больше.

9.6. Упаковка четырехбитных кодов

Рассмотрим еще один пример манипуляций над битами. Пусть в дополнение к задаче вычисления количества бит, установленных в 1 (см. *разд. 9.1*), требуется упаковать полученные значения по два в байт.

Проиллюстрируем варианты упаковки. Предположим, что в двух соседних байтах содержатся следующие значения по битам:

```
x x x x 0 1 1 0      x x x x 1 0 0 0
```

По условиям задачи из *разд. 9.1* показанные здесь данные означают, что исходный массив содержит шесть и восемь единичных бит в соответствующих байтах. (Биты старших тетрад, помеченные `x`, интереса не представляют.) Результат упаковки должен выглядеть следующим образом:

```
0 1 1 0 1 0 0 0
```

Примечание

Во всех приведенных ниже вариантах упаковки результат формируется в регистре `al`.

Продемонстрируем общий способ упаковки. Предположим, что первый код (число 6) записан в регистр `dl`, а второй (число 8) — в `al`.

```

;          dl                      al
;
; x x x x 0 1 1 0          x x x x 1 0 0 0
;
;          shl          al, 4
;
; x x x x 0 1 1 0          1 0 0 0 x x x x
;
;          shr          dl, 1
;          rcr          al, 1
;
; 0 x x x x 0 1 1  ->c->    0 1 0 0 0 x x x
;
;          shr          dl, 1
;          rcr          al, 1
;
; 0 0 x x x x 0 1  ->c->    1 0 1 0 0 0 x x
;
;          shr          dl, 1
;          rcr          al, 1
;
; 0 0 0 x x x x 0  ->c->    1 1 0 1 0 0 0 x
;
;          shr          dl, 1
;          rcr          al, 1
;
; 0 0 0 0 x x x x  ->c->    0 1 1 0 1 0 0 0

```

В этом варианте решения биты из `dl` передаются в `al` по одному, через флаг `c`. (Конечно, четырехкратный повтор команд `shr` и `rcr` можно было бы оформить в виде цикла.) Если байты из одного слова, проще сдвинуть все слово сразу на 4 разряда. Как в первом варианте упаковки, сначала нужно "придвинуть" второй код к левой границе байта. Пусть второй байт находится в `al`, а первый — в `ah`.

```

;          ah                      al
;
; x x x x 0 1 1 0 x x x x 1 0 0 0
;
;          shl          al, 4
;
; x x x x 0 1 1 0 1 0 0 0 x x x x
;
;          shr          ax, 4
;
; 0 0 0 0 x x x x 0 1 1 0 1 0 0 0

```

Продemonстрируем еще один вариант упаковки с применением команд `and` и `or` (в этом варианте коды могут быть в разных словах):

```

;          ah                      al
;
; x x x x 0 1 1 0          x x x x 1 0 0 0
;
;          shl          ah, 4
;
; 0 1 1 0 0 0 0 0          x x x x 1 0 0 0
;
;          and          al, 0f
;
; 0 1 1 0 0 0 0 0          0 0 0 0 1 0 0 0
;
;          or           al, ah
;
; 0 1 1 0 0 0 0 0          0 1 1 0 1 0 0 0

```

9.7. Задания на составление программ

Ниже приведены две группы заданий. В первой группе — наименее сложные, во второй — средней сложности.

9.7.1. Задания первого уровня сложности

Выполните выбранное задание в двух вариантах.

1. Данные исходного массива или массивов определены в самой программе. Этот вариант — отладочный. Ввод/вывод программировать не следует, достаточно того, что программа работает в отладчике, и в окнах отображения памяти видны исходные данные и результаты выполнения. Формат отображения необходимо выбрать в соответствии со смыслом данных (массив кодов ASCII отображается в одном из форматов *c*, *a*, *t*, *r*; массив слов — в формате *w* или *d*; массив байтов — в формате *b* или *n*; если речь о битовых операциях, формат *e* предпочтительней).
2. Данные массивов считываются из файла через стандартный *входной* поток, а результаты записываются в другой файл через стандартный *выходной* поток. Если по условиям задачи на входе должна быть задана пара массивов одинаковой размерности, то все прочитанное разделите пополам; первая половина относится к первому массиву, вторая — ко второму (программа должна вычислить размер массива и начальный адрес второго массива, исходя из значения *ax* после выполнения *dos 03f*).

При отладке этого варианта вызывайте d86 следующим образом:

```
d86 var1.com <test_1.bin >out.dat
```

или

```
d86 var1.com
```

Во втором случае при вызове `dos 03f` (на команде `int 021`) программа остановится, ожидая ввода строки — до завершающего нажатия клавиши `<Enter>`. Обратите внимание, что по нажатии клавиши `<Enter>` к массиву введенных данных добавляются два служебных символа — "возврат каретки" (код 13) и "перевод строки" (код 10).

Варианты заданий

1. Найдите максимум в массиве слов, и замените элементы массива со значениями 0 на максимум. (В отладчике отобразите исходный массив, значение максимума и результирующий массив.)
2. Задан массив ASCII-кодов. Подсчитайте число двойных символов (например, `tutti buzz` содержит две таких пары). (В отладчике отобразите исходный массив и результат подсчета; при выполнении второго варианта в выходной файл запишите слово, содержащее результат подсчета.)
3. Вычислите сумму (в формате слова) значений массива байт. По сумме вычислите среднее значение в формате байта. (В отладчике отобразите исходный массив и значение среднего; при выполнении второго варианта в выходной файл запишите слово, содержащее результат подсчета среднего.)
4. Задан массив ASCII-кодов. Подсчитайте количество заглавных латинских букв. (В отладчике отобразите исходный массив и результат подсчета; при выполнении второго варианта в выходной файл запишите слово, содержащее результат подсчета.)
5. Задан массив, содержащий знаковые слова. Сформируйте на его месте массив с абсолютными значениями исходного массива. (В отладчике отобразите исходный и результирующий массивы; в выходной файл при выполнении второго варианта запишите результирующий массив.)
6. Задан массив ASCII-кодов. Подсчитайте количество латинских букв и, отдельно, количество литер десятичных цифр. (В отладчике отобразите исходный массив и результат подсчета; при выполнении второго варианта в выходной файл запишите полученные счетчики в формате слов.)
7. Заданы два массива байтов одинаковой длины. Сформируйте массив на месте первого, содержащий значения попарно совпадающих элементов исходных массивов. (В отладчике отобразите исходные массивы, результирующий массив и число совпадающих элементов; при выполнении второго варианта в выходной файл запишите результирующий массив.)
8. Задан массив ASCII-кодов. Сформируйте по его содержимому массив ASCII-кодов в обратной последовательности. (В отладчике отобразите исходный и результирующий массивы; при выполнении второго варианта в выходной файл запишите результирующий массив.)
9. Заданы два массива одинаковой длины. Сформируйте массив на месте первого массива, содержащий индексы попарно совпадающих элементов исходных массивов (отсчет индексов — от нуля). (В отладчике отобразите исходные массивы, результирующий массив и число совпадающих элементов; при выполнении второго варианта в выходной файл запишите результирующий массив.)

10. Задан массив знаковых байтов. Сформируйте массив, содержащий индексы тех элементов исходного массива, значения которых находятся в заданном диапазоне (отсчет индексов — от нуля). (В отладчике отобразите исходный массив, результирующий массив и число элементов в нем; при выполнении второго варианта в выходной файл запишите результирующий массив.)
11. Задан массив ASCII-кодов. Сформируйте на его месте массив ASCII-кодов, в котором все латинские буквы из исходного массива преобразованы к верхнему регистру. (В отладчике отобразите исходный и результирующий массивы; при выполнении второго варианта в выходной файл запишите результирующий массив.)
12. Задан массив беззнаковых слов. Сформируйте на его месте массив, содержащий значения тех элементов исходного массива, которые находятся вне заданного диапазона. (В отладчике отобразите исходный массив, результирующий массив и число элементов в нем; при выполнении второго варианта в выходной файл запишите результирующий массив.)
13. Заданы два массива знаковых слов, одинаковой длины. Запишите на месте первого массива результаты поэлементного умножения исходных массивов. (В отладчике отобразите исходные массивы, результирующий массив; при выполнении второго варианта в выходной файл запишите результирующий массив.)
14. Задан массив ASCII-кодов. Сформируйте на его месте массив ASCII-кодов, в котором все латинские буквы преобразованы к нижнему регистру. (В отладчике отобразите исходный и результирующий массивы; при выполнении второго варианта в выходной файл запишите результирующий массив.)
15. Заданы два массива беззнаковых слов одинаковой длины. Сформируйте на месте этих массивов результаты поэлементного деления исходных массивов (целую часть результатов сохраните на месте первого массива, а остатки — на месте второго). (В отладчике отобразите исходные и результирующие массивы; при выполнении второго варианта в выходной файл запишите массив остатков.)

9.7.2. Задания второго уровня сложности

1. Задан массив ASCII-кодов. Сформируйте гистограмму цифр от '0' до '9' — массив слов, число элементов в котором равно количеству цифр, значение нулевого элемента равно числу экземпляров цифры '0' в исходном массиве, значение первого элемента — числу экземпляров цифры '1', и т. д. до '9'.

Входные данные возьмите из файла `test1.bin`, получив его из `test1.8`. Результат должен быть следующим:

```
2 6 11 19 26 39 50 67 83 102
```

2. Подсчитайте число битов со значением 1 в битовом массиве; счет ведите в двойном слове с использованием команд `add` и `adc`. Входные данные возьмите из файла `test2.bin` (выполнив трансляцию `test2.8`). Результат счета должен быть 016000.

3. Получите из 2-битных кодов Грея 3-битные, из них — 4-битные и т. д. до восьмибитных. Результаты — промежуточные и окончательный — формируйте в области данных размером 256 байт (изначально первые четыре байта этой области содержат 2-битные коды).

Как получить коды Грея для $n+1$ бит, исходя из кодировки для n бит? Обозначим набор кодов для i бит — $G(i)$. Первая половина $G(n+1)$ — это набор $G(n)$, каждый код которого дополнен слева битом 0. Вторая половина $G(n+1)$ — это коды из $G(n)$ в обратном порядке, дополненные слева битом 1. Пример получения $G(3)$ из $G(2)$:

00 01 11 10	$G(2)$
000 001 011 010	первая половина $G(3)$
10 11 01 00	$G(2)$ в обратном порядке
110 111 101 100	вторая половина $G(3)$
000 001 011 010 110 111 101 100	$G(3)$ целиком

Результирующий массив запишите в файл, содержимое его должно совпадать с содержимым файла test3.bin.

4. Упаковка шестибитных кодов.

Задан массив байт, длина его кратна 4. В каждом байте имеют значение только младшие шесть бит. Требуется "сжать" последовательность бит так, чтобы ликвидировать незначимые старшие биты, с сохранением исходного порядка бит.

Ниже проиллюстрировано получение одной "упаковки", показаны:

- номера байтов, от 0 до 3;
- номера разрядов в байтах;
- четыре исходных байта (литерами +, −, *, # отмечены значащие разряды, вопросом — незначащие);
- три байта результата.

	0								1								2								3							
; 0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	
; +	+	+	+	+	+	+	?	−	−	−	−	−	−	?	?	*	*	*	*	*	*	?	?	#	#	#	#	#	#	?	?	
; +	+	+	+	+	+	+	−	−	−	−	−	−	−	*	*	*	*	*	*	#	#	#	#	#	#	#	#	#	#			

Примечание

Нумерация разрядов "слева направо" отражает смысл задачи, но противоречит условностям, принятым в обозначении команд сдвигов и вращений. Сдвиг в сторону *старших разрядов* именуют сдвигом "влево" и обозначают с использованием буквы l (например, shl, rcl). Аналогично, так называемый сдвиг "вправо" (например, ror, sar) — это сдвиг в сторону *младших разрядов*.

Входные данные возьмите из файла `test4.in` (6-битные коды Грея), результат запишите в файл и сравните его содержимое с `test4.out`.

5. Задана последовательность байтов. В каждом байте выделите биты с номерами от 0 до 5, а в шестом бите сформируйте дополнение до четности. Полученные семибитные коды требуется упаковать так, чтобы порядок бит не изменился (см. пояснения к заданию 4). Восемь кодов упаковываются в семь байт. Число байтов во входной последовательности должно быть кратно 8.

Входные данные возьмите из `test5.in` (6-битные коды Грея), результат запишите в файл и сравните его содержимое с `test5.out`.

6. Задана последовательность ASCII-кодов. Представьте эту последовательность в коде RADIX-50. Для этого каждая литера исходной последовательности предварительно перекодируется по табл. 9.1.

Таблица 9.1. Коды RADIX-50

Литера	Код RADIX-50
Пробел	0
A—Z	1—32xq
\$	33xq
.	34xq
(резерв)	35xq
0—9	36xq—47xq

Результаты перекодировки упаковываются по три в слово. Упаковка выполняется по формуле $(c_1 \times 50xq + c_2) \times 50xq + c_3$, где c_1 , c_2 , c_3 — коды, полученные в соответствии с табл. 9.1. Например, коды для литер 'ABC' — это 1, 2, 3. В результате упаковки получаем $(1 * 50xq + 2) * 50xq + 3 = 3223xq$.

Литеры, не принадлежащие множеству 'A'—'Z', ' ', '\$', '0'—'9', кодируются числом 35xq. Например, коды для последовательности литер 'AB', 0 — это 1, 2, 35xq, а результат упаковки равен 3255xq. Еще один пример: код RADIX-50 для последовательности литер 'X2B' составляет 115402xq.

Входные данные возьмите из `test6.in`, результат запишите в файл и сравните его содержимое с `test6.out`.

7. Задана последовательность беззнаковых байт **A**. Сформируйте сглаженную последовательность **B** по формуле:

$$B(i) = A(i-1)/4 + A(i)/2 + A(i+1)/4$$

(Обратите внимание, что ***B*** на два элемента короче, чем ***A***.) Для уменьшения погрешностей округления и сокращения числа делений используйте следующий вариант формулы:

$$B(i) = ((A(i-1) + A(i+1)) / 2 + A(i)) / 2$$

Во избежание переполнения при сложении, вычисления выполняйте над словами.

Входные данные возьмите из файла test7.in, запишите результат в файл и сравните его содержимое с test7.out.

8. Дана входная последовательность ASCII-кодов. Запишите ее спиралью в двумерную квадратную матрицу. Размерность матрицы подберите так, чтобы текст помещался в матрицу полностью или с избытком; кроме того, размерность должна быть нечетной. "Хвост" спирали дописывается null-кодами.

Для входных данных из файла test8.bin должна получиться матрица, показанная на рис. 9.1. В хвост спирали на рис. 9.1 дописаны два null-кода.

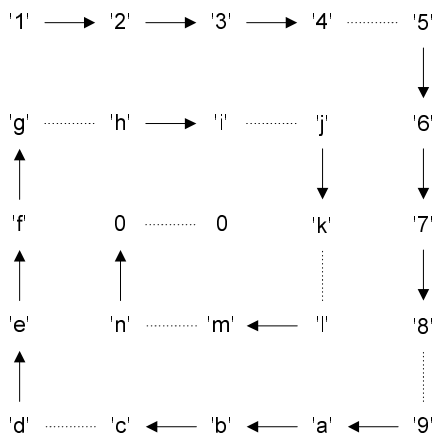


Рис. 9.1. Матрица, полученная из файла test8.bin

Соединительные линии по "граням" спирали подсказывают следующий вариант решения: входные литеры записываются в массив за несколько циклов ("витков"), на каждом из которых указатель записи перемещается на фиксированное число шагов влево, вниз, вправо и вверх; на каждом последующем витке число шагов уменьшается.

9. Входная последовательность содержит следующие данные: в первом байте — значение размерности квадратной матрицы (нечетное число в пределах от 3 до 127), в следующих байтах — саму матрицу по строкам. Сформируйте массив байт, считывая байты по спирали матрицы.

Образец входной последовательности — в файле test9.bin.

10. Подсчет количества последовательностей одинаковых соседних литер из множества 'A—Za—z'. Например, в последовательности из файла test10.bin насчитывается пять таких совпадений:

```
tutto   hooolla hooop 1lbb
  ^       ^ ^       ^       ^
```

Соседние пробелы игнорируются, пара 'll' — также не в счет. Три подряд буквы 'o' считаются одним совпадением, а не двумя.

11. Транспонируйте квадратную матрицу 8×8 , элементы матрицы — байты. Входные данные возьмите из test11.bin, получив его из test11.8. Для наблюдения в отладчике рекомендуем отображать по восемь ASCII-кодов в строке.
12. Дана последовательность ASCII-кодов. Преобразовать литеры в четырехбитные коды в соответствии с табл. 9.2. Литеры не из множества '0'—'9', '—', '.' считаются разделителем и кодируются значением 0с. Разделители, следующие подряд, объединяются — вся их последовательность шифруется одним кодом 0с. Конец последовательности кодируется значением 0f.

Таблица 9.2. Таблица перекодировки к заданию 12

ASCII-код	4-битный код
'0'—'9'	0—9
'—'	0a
'.'	0b

Пример с данными из файла test12.bin:

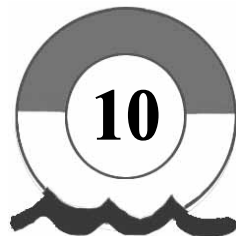
```
-3.14,! ?#,,16
```

Эта последовательность ASCII-кодов преобразуется в коды:

```
0a 3 0b 1 4 0с 1 6 0f
```

Полученная последовательность 4-битных кодов упаковывается по два в байт (в младшей тетраде байта — первый код из пары, в старшей тетраде — второй). Результат упаковки:

```
03a 01b 0с4 061 0f
```



Сегменты и ехе-программы

— Скажите, как вам помочь? Не выпить ли мне немного водки, чтобы вас подкрепил ее дух?

Джером К. Джером. "Пирושка с привидениями"

В этой главе рассматриваются:

- ❑ сегментная модель памяти в процессорах i80x86;
- ❑ организация программ с несколькими логическими сегментами.

10.1. Сегментная модель памяти

В инструкциях i8086/286 адрес данных (или адрес инструкции — в командах передачи управления) ограничен 16 разрядами. То есть значение адреса ограничено диапазоном 0—0ffff, и размер адресуемой памяти не превышает 64 Кбайт. Вместе с тем, адресная шина процессоров i8086/286 содержит 20 линий, что расширяет диапазон до 0ffffff; доступное адресное пространство составляет 1024 Кбайт.

Как при помощи 16-разрядных регистров адресовать 1024 Кбайт? Чтобы не вводить специальных 20-разрядных регистров для адресации данных и инструкций, при разработке процессора i8086 была принята *сегментная модель памяти*.

Внутри *сегмента* — блока памяти размером 64 Кбайт — смещение задается 16-разрядным значением. Положение сегмента в адресном пространстве 1024 Кбайт задается номером от 0 до 0ffff; начала сегментов отстоят друг от друга на 16 байт ("параграф"). На рис. 10.1 показаны: слева — номера сегментов, справа — 20-битные абсолютные адреса границ сегментов.

Например, в пределах сегмента номер 2 доступна память по абсолютным адресам 020—01001f. Абсолютный адрес 027 доступен в трех сегментах:

в сегменте 2 по смещению 7, в сегменте 1 по смещению 017, в сегменте 0 по смещению 027, что сокращенно (в формате *сегмент:смещение*) обозначается как 2:7, 1:017, 0:027.

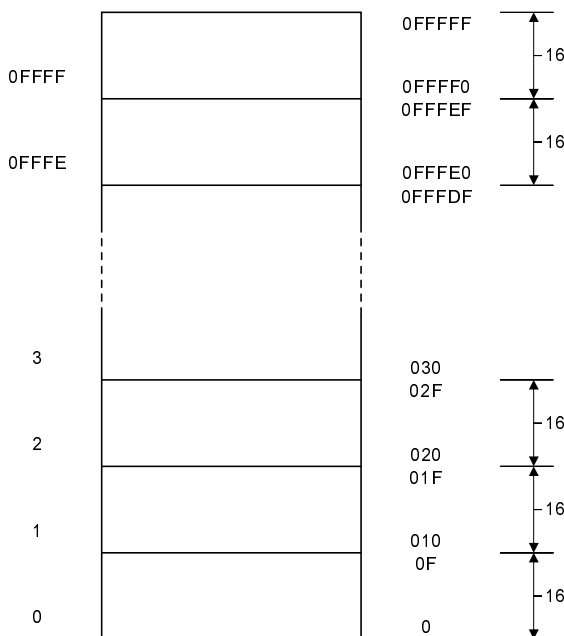


Рис. 10.1. Расположение сегментов

Сегменты, постоянно используемые процессором для адресации памяти при выполнении программы, следующие:

- ❑ текущий сегмент кода, из которого считывается очередная инструкция — по смещению, заданному `ip`;
- ❑ текущий сегмент данных, для доступа к операндам инструкций по смещению, заданному в инструкции;
- ❑ текущий сегмент стека, для доступа к стеку — по смещению, заданному регистром `sp` или `bp`.

Текущие значения номера сегмента кода, сегмента данных и сегмента стека хранятся в регистрах `cs` (Code Segment), `ds` (Data Segment), `ss` (Stack Segment). Еще один сегментный регистр — `es` (Extra data Segment, дополнительный сегмент данных) — используется для доступа к операнду-приемнику при выполнении строковых команд.

Итак, для считывания очередной инструкции используется пара 16-разрядных регистров `cs` и `ip`; для обращения к стеку — пара `ss` и `sp/bp`; для

обращения к данным — сегментный регистр `ds` и смещение, заданное в команде. Если смещение задано косвенно через `bp`, то вместо `ds` используется `ss`. Операнд-приемник строковой команды адресуется с использованием пары `es:di`.

Пример (предполагается, что тип имени `word1` — `word`, тип `m1` — `abs`):

```
inc     w [bx]           ; ds:bx
push    word1            ; ds:word1 -> ss:sp
call    m1               ;     -> ss:sp
mov     [bp+8], 1        ; 1 -> ss:bp+8
movsb                   ; ds:si -> es:di
```

В тех случаях, когда предполагается использование `ds`, можно взамен него временно (для одной команды) задать любой другой сегментный регистр, за счет *префикса переназначения* сегмента данных.

Примечание

Префикс переназначения позволяет заменить любой действующий по умолчанию сегментный регистр (не только `ds`), если полный адрес операнда не образуется парой `ss:sp` или `es:di`. С учетом этих ограничений, префикс переназначения действует на `ds` всегда, на `es` — никогда, а на `ss` — только при использовании косвенной адресации через `bp`.

В следующем примере показаны два способа задания префикса переназначения сегмента: общепринятый — с двоеточием при операнде, и предлагаемый а86 в качестве дополнения — перед командой. Указание префикса перед командой более наглядно, поскольку код префикса предшествует машинному коду операции.

```
es inc  w [bx]           ; es:bx
push    cs:[word1]       ; cs:word1 -> ss:sp
es call m1               ; ?
cs movsb                 ; cs:si -> es:di
mov     cs:[bp+8], 1     ; 1 -> cs:bp+8 (!)
```

В третьей команде из примера префикс не имеет смысла, т. к. использование `ds` в инструкции `call` не предполагается. В четвертой команде `cs` заменяет `ds`, в результате, операнд-источник при выполнении `movsb` находится в сегменте кода.

В последней команде — как раз тот случай, когда префикс имеет смысл, несмотря на то, что использование `ds` здесь не предполагается. По умолчанию, должен был бы использоваться `ss`, ввиду косвенной адресации через `bp`. В этом исключительном случае `ss` заменяется любым другим сегментным регистром, указанным в префиксе; в данном случае — `cs`.

Обратите внимание, что номер сегмента нигде не задан числом — только содержимым сегментного регистра. Непосредственное обозначение номера сегмента допустимо лишь в командах дальней (межсегментной) передачи управления:

```
call  0:56
jmp   0fff0:0
inc   w 0:[di]      ; ?!
dec   b 0200:1      ; ?!
```

Для записи значений в сегментные регистры допускается использование команды `mov` и еще нескольких команд из числа пересылок (`lds`, `les`, `push`, `pop`):

```
mov    es, 0b800
mov    cs, es        ; ?!
es mov b [0], '*'
```

Для записи в сегментный регистр по команде `mov` второй операнд может быть только регистром общего назначения, например `ax`. Первая команда из примера выше в машинном представлении не существует, это — встроенная макрокоманда `а86`. Вторая команда в принципе недопустима, поскольку непосредственная запись в регистр `cs` равносильна передаче управления (для этого есть команда `jmp`). Третья команда (при `es = 0b800`, в результате выполнения первой команды) записывает в видеопамять символ '*', который отображается в левом верхнем углу экрана.

Контрольные вопросы

1. Пусть значения сегментных регистров следующие: `es = 098`, `cs = 0fff0`, `ds = 0`, `ss = 05078`. По каким абсолютным адресам произойдет обращение к памяти при выполнении следующих команд (`sp = 0100`):

```
push   ax
es mov [2], bl
mov    bp, di, 2
inc    b [bp+di]
cs lodsb
```

2. Пусть значения в сегментных регистрах следующие: `es = 05`, `cs = 0ff0`, `ds = 0200`, `ss = 050`. По каким абсолютным адресам произойдет обращение к памяти при выполнении следующих команд? По какому адресу будет выполнен переход?

```
mov    bx, 8
mov    cs:[bx], 16
es mov [-2], bl
cs jmp [bx]
```

3. По абсолютному адресу 0417 находится байт с информацией о нажатии управляющих клавиш (<Ctrl>, <Ins>, <Alt> и т. п.). Используя из сегментных регистров только `ds`, скопируйте этот байт в видеопамять. Также решите эту задачу, используя только `es`. Можно ли воспользоваться `cs` вместо `ds` или `es`?
4. С использованием строковой команды `movsb` и префикса повторения составьте фрагмент, который копирует в видеопамять 2 Кбайта данных с *абсолютного* адреса 0400.
5. Найдите ошибки и бессмысленные префиксы в следующих командах (пусть тип имени `word1` — `word`, тип `m1` — `abs`):

```
cs push 1
mov     ss:[bp+di], 0ff
es jmp  word1
mov     cs, ds
ds inc  w [bx+4]
jmp     cs:m1
```
6. Запишите команду передачи управления по абсолютному адресу 0ffff8. Увеличьте значение байта по абсолютному адресу 0b8020 в 7 раз при помощи команды `mul`. Задайте команду инкремента десятого байта видеопамяти.

10.2. Сегменты в com-программе

В `com`-программе сегментные регистры установлены одинаково. После загрузки программа занимает один физический сегмент (64 Кбайт), `ip` = 0100, `sp` = 0ffffe (рис. 10.2).

Блок памяти, выделяемой загрузчиком для размещения программы (кода, данных и стека), называется *программным сегментом*. При загрузке исполняемого файла в `com`-формате программный сегмент равен одному физическому сегменту.

Префикс программного сегмента (Program Segment Prefix, PSP) длиной 256 байт создается в начале программного сегмента (при загрузке `com`-файла). В нем содержатся сведения об окружении программы. Так, например, с адреса 081 до 0ff в PSP записана часть командной строки, следующая за именем программы — параметры командной строки. (Строка параметров начинается пробелом по адресу 081 и заканчивается кодом возврата каретки 0d; длина строки параметров, не считая 0d, записана в байте по адресу 080.)

Если в ходе выполнения требуется доступ к данным вне сегмента `com`-программы, приходится программировать перенастройку `ds` и/или `es`. Исходные значения этих сегментных регистров легко восстановить из текущего содержимого `cs`, поскольку значение `cs` при выполнении инструкций `com`-программы фиксировано.

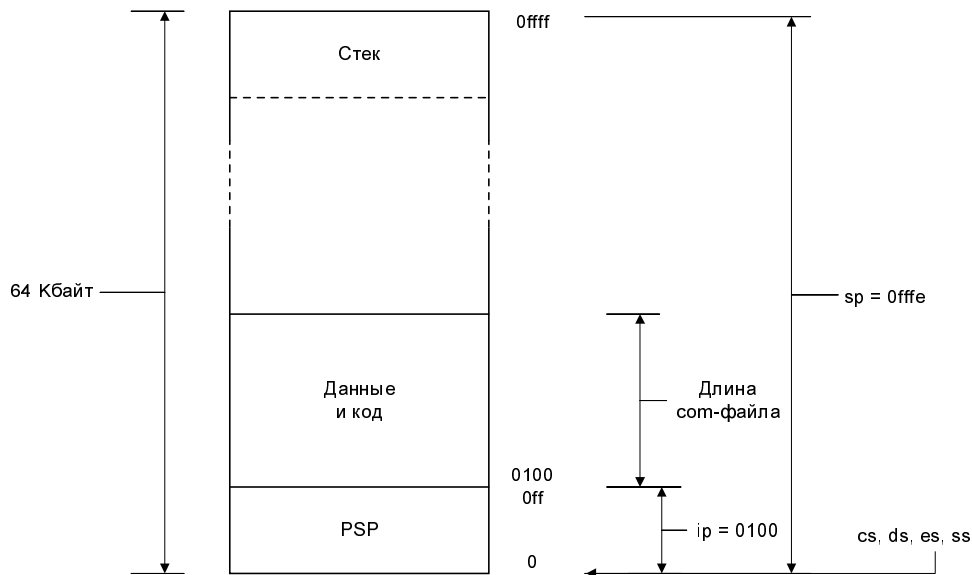


Рис. 10.2. Настройка сегментов после загрузки com-программы

Пример:

```
mov     ds, 0b800    ; вывод '*' в левом верхнем углу экрана
mov     [0], '*'      ; прямым отображением в видеопамять
mov     ds, cs        ; восстановление ds
```

10.3. Сегменты в ехе-программе

Программа в ехе-формате позволяет вводить неограниченное количество отдельно адресуемых областей кода и данных. (Стек один, и больше не требуется.) Размер каждой из таких областей может достигать размера одного физического сегмента (64 Кбайт). В дальнейшем будем называть эти области *логическими сегментами*.

По данным из заголовка ехе-файла загрузчик настраивает стек (*ss* и *sp*) и запускает программу (устанавливает *cs* и *ip*). Регистры *ds* и *es* содержат номер сегмента PSP. В программе необходимо предусмотреть запись в регистры *ds* и *es* для организации доступа к логическим сегментам данных. Что касается смены текущего сегмента кода, она выполняется за счет дальних переходов — только таким способом можно изменить *cs:ip*.

Пример исходного текста ехе-программы, в соответствии с рис. 10.3, приведен в листинге 10.1.

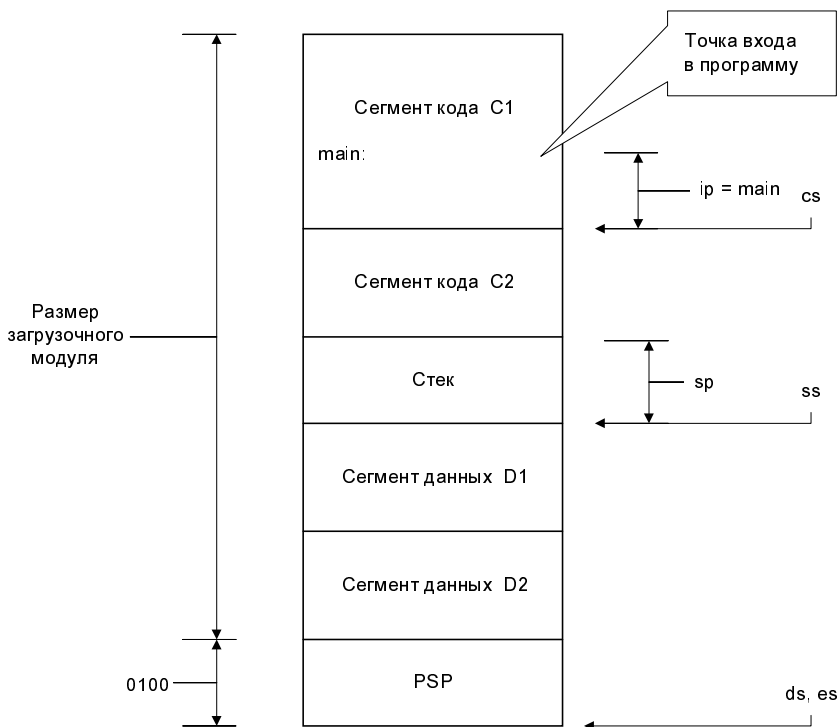


Рис. 10.3. Сегменты и сегментные регистры после загрузки exe-программы

Листинг 10.1. Программа из нескольких сегментов (см. prog_a.08)

```
seg_D_2 segment
src      db      'Very short logical segment', 13, 10, '$'
size     equ     $ - src

seg_D_1 segment
dst      db      size dup ?

seg_sp segment stack
         dw      0100 dup ?

seg_C_2 segment

main:
mov      ds, seg_D_2      ; (1)
mov      es, seg_D_1      ; (2)
```

```
mov     cx, size
lea     di, dst
lea     si, src
call    far copy           ; (3)

mov     ds, es             ; (4)
lea     dx, dst
dos     9
dos     04c
```

seg_C_1 segment

copy:

```
cld
rep movsb
retf
```

Границы и имена логических сегментов заданы директивой `segment`. При определении стека в директиве `segment` задан тип объединения `stack`, который предписывает настройку указателя стека `ss:sp` на старший адрес логического сегмента. Именем `main` отмечена точка *входа* в программу после загрузки.

Настройка `ds` и `es` выполняется сразу же в начале программы (1, 2). В результате, текущий сегмент данных — `seg_D_2`, текущий дополнительный сегмент данных — `seg_D_1`. Дальний вызов подпрограммы из логического сегмента `seg_C_1` приводит к переключению текущего сегмента кода (изменяется не только `ip`, но и `cs`). Подпрограмма `copy` копирует массив с адреса `ds:si` в массив по адресу `es:di`, а затем возвращает управление в точку вызова при помощи `retf` (при дальнем вызове в стеке сохранены значения `cs` и `ip` для команды (4), `retf` считывает из стека две записи и восстанавливает по ним `ip` и `cs`). После возврата выполняется вывод строки из `dst` на экран, затем выход в DOS.

Приведем еще один вариант этой программы, в котором данные и код объединены в один логический сегмент, а сегмент стека не задан. Если сегмент стека в программе не задан, то загрузчик устанавливает `ss` на начало программного сегмента, а `sp` — на максимум. (Если данные и код небольшого объема, то стек их не разрушит.)

На первый взгляд, полная аналогия с `com`-программой. Тем не менее, в `com`-программе начальное значение `ip` может быть только 0100, а здесь — произвольное, какое получится в результате подготовки исполняемой программы. Кроме того, здесь не настроен доступ к данным (`ds` и `es` указывают на PSP). В листинге 10.2 приведен исходный текст, соответствующий распределению сегментов на рис. 10.4.

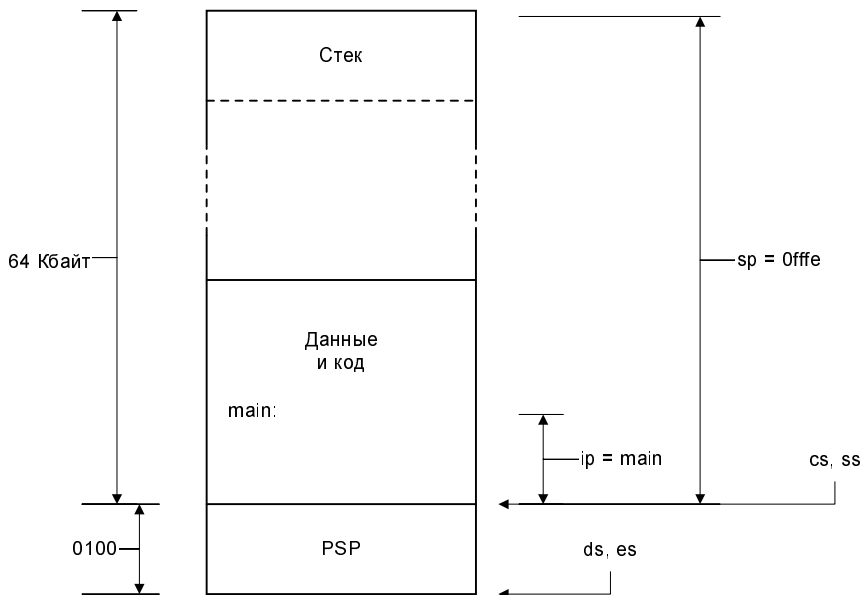


Рис. 10.4. Программа ехе-формата из одного логического сегмента

Листинг 10.2. Программа ехе-формата из одного сегмента (см. prog_b.08)

```

_prog    segment

src      db      'Single segment', 13, 10, '$'
size     equ     $ - src
dst      db      size dup ?

copy:
    cld
    rep movsb
    ret                                ; near!

main:
    mov     ds, _prog
    mov     es, ds
    mov     cx, size
    lea     di, dst
    lea     si, src
    call    copy                        ; near!
    lea     dx, dst
    dos     9
    dos     04c

```

Примечание

Подпрограмма `copy` вызывается как ближняя, возврат из нее — тоже ближний. Можно было бы оставить и "дальний" вариант, но теперь в нем нет необходимости, т. к. `copy` и `main` находятся в одном логическом сегменте.

10.4. Особенности подготовки ехе-программы

Особенности подготовки ехе-программы следующие:

- ☐ для каждого логического сегмента счет адресов при трансляции начинается с нуля;
- ☐ исходный текст на языке ассемблера транслируется в *объектный файл*;
- ☐ ехе-программа создается *компоновщиком* из объектных файлов;
- ☐ соответствие между именами логических сегментов и номерами физических сегментов устанавливается при загрузке.

Счетчик адресов `$` обнуляется, если имя, заданное в директиве `segment`, встречается впервые. Иначе счет адресов продолжается от того значения, на котором он закончился при трансляции предшествующей части логического сегмента. Иными словами, директива `<name> segment` либо открывает, либо продолжает логический сегмент с именем `<name>`. Одноименные фрагменты, заданные директивами `segment`, объединяются не только в пределах исходного модуля на языке ассемблера, но также и по всей совокупности объектных модулей (при *компоновке*).

Заметим, что фрагменты, заданные директивой `segment`, относятся к одному логическому сегменту только в том случае, если у них одновременно:

- ☐ одно и то же имя;
- ☐ одинаковый *класс*;
- ☐ тип объединения `public` (для фрагментов стека — `stack`).

Класс и тип объединения задаются после ключевого слова `segment`. Эти параметры необязательны; по умолчанию, имя класса — пустая строка, а тип объединения — `private`, что означает объединение только в пределах исходного модуля средствами ассемблера.

Пример:

```
_text segment public 'code'
```

Фрагмент под именем `_text` в этом примере относится к классу `'code'`. Поскольку его атрибут `public`, то этот фрагмент будет сливаться с аналогичными фрагментами (у которых имя — `_text`, класс — `'code'`, тип объединения — `public`) не только при трансляции, но и при *компоновке*.

Фрагменты с одним и тем же именем *класса* будут размещены в памяти рядом, даже если они относятся к разным логическим сегментам — в этом смысл классов.

Примечание

Классы в основном используются при трансляции с языков высокого уровня. Так, приведенное выше определение `_text ... 'code'` — стандартное определение логического сегмента кода языка C в модели памяти `small`.

Имеется еще одна директива для принудительного объединения сегментов, в обход правила "одинаковое имя, одинаковый класс, тип объединения — `public`". Это — директива `group`.

```
dgroup group _data, _const, _bss
```

Здесь объявлен логический сегмент с именем `dgroup`, объединяющий логические сегменты с именами `_data`, `_const`, `_bss`, в свою очередь объявленные директивами `segment`. Примерно в таком виде `group` используется при трансляции с языка C в `small`-модели; в результате, объединяются в один физический сегмент раздельно заданные логические сегменты инициализированных и неинициализированных переменных (`_data`, `_bss`) и констант размерностью больше одного слова (`_const`). Смещения объектов внутри `_data`, `_const`, `_bss` пересчитываются относительно сегмента `dgroup`. Теперь, чтобы настраивать сегментные регистры, вместо имен `_data`, `_const`, `_bss` следует использовать имя `dgroup`.

На этом закончим рассмотрение вопросов, связанных с объединением сегментов, и остановимся подробнее на том, каким образом имени логического сегмента ставится в соответствие номер физического сегмента.

Проблема заключается в том, что это соответствие невозможно установить при трансляции. Номера физических сегментов будут известны только после того, как загрузчик получит свободный блок памяти для программного сегмента. Какой адрес будет у этого блока, заранее неизвестно.

Вернемся к примеру `exe`-программы из листинга 10.1. Команды (1) и (2) содержат ссылки на имена сегментов, команда (3) подразумевает абсолютное значение сегмента `seg_C_1`. Эти команды должны быть скорректированы загрузчиком; с этой целью данные о местоположении команд (1—3) относительно начала программного сегмента записываются в *заголовке* `exe`-файла (в *таблице перераспределения*). В область загрузочного модуля `exe`-файла вместо абсолютных номеров сегментов записываются *относительные* номера — расстояния от начала программного сегмента в параграфах.

Примечание

Именно эти относительные номера показывает компоновщик в карте загрузки, т. е. в отчете о построении `exe`-файла (см. разд. 10.6.1).

Так, в программе на месте `seg_D_1` в (1) будет записан ноль (поскольку начало логического сегмента `seg_D_1` находится в начале программного сегмента). На месте `seg_D_1` в (2) будет записано значение 2 (начало логического сегмента `seg_D_1` отстоит от начала программного сегмента на два параграфа). На месте сегментной составляющей адреса перехода в (3) будет записано число 026 — т. к. логический сегмент `seg_C_1` находится в 026 параграфах от начала программного сегмента.

К этим предварительным значениям будет прибавлен номер *физического сегмента*, полученного загрузчиком. Где именно требуется такая коррекция, задано в таблице перераспределения.

10.5. Построение ехе-программ из нескольких модулей

Программа, в общем случае, может состоять из нескольких *модулей*. Исходные модули транслируются по отдельности, затем полученные объектные модули объединяются компоновщиком в исполняемую программу.

Модули программы, как правило, связаны друг с другом. Например, модуль *A* содержит набор подпрограмм, к которым обращается модуль *B*. Символические адреса подпрограмм (метки) в модуле *A* объявляются как *глобальные имена* (`public`), а в модуле *B* — как *внешние имена* (`extrn`).

Продемонстрируем `public`- и `extrn`-объявления на примере рассмотренной выше программы. Пусть подпрограмма `copy` задана в модуле `prog1.08`, а остальная часть программы — в главном модуле `prog2.08`.

Листинг 10.3. Модуль с объявлением `public`-имени (см. `prog1.08`)

```
_text    segment public 'code'

        public  copy

copy:
        cld
        rep movsb
        ret
```

Листинг 10.4. Главный модуль с объявлением `extrn`-имени (см. `prog2.08`)

```
_text    segment public 'code'

src      db      'Single segment', 13, 10, '$'
size     equ     $ - src
dst      db      size dup ?
```

```
extrn    copy:near

main:
    mov    ds, _text
    mov    es, ds
    mov    cx, size
    lea    di, dst
    lea    si, src
    call   copy
    lea    dx, dst
    dos    9
    dos    04c
```

В `extrn`-описаниях имя указывается вместе с типом — `b/w/d/abs`. Имена, используемые в `call/jmp`, уточняются атрибутом `near` или `far` (`near` — если переход в пределах текущего сегмента, т. е. без изменения `cs`; `far` — переход в произвольный сегмент, с переключением сегмента кода). Во всех модулях задана директива `segment` с одним и тем же именем и параметрами.

При использовании `a86` модули можно записать еще короче (заметим, что даже в приведенном варианте ассемблеры `masm` и `tasm` требуют больше директив: `end` в конце каждого модуля и `assume` в начале; см. приложение 7).

Листинг 10.5. Модуль с неявным объявлением `public`-имен (см. `prog1a.08`)

```
copy:
    cld
    rep movsb
    ret
```

Листинг 10.6. Главный модуль с неявным объявлением `extrn`-имени (см. `prog2a.08`)

```
_text    segment public 'code'

src       db    'Single segment', 13, 10, '$'
size      equ    $ - src
dst       db    size dup ?

main:
    mov     ds, _text
    mov     es, ds
    mov     cx, size
    lea     di, dst
    lea     si, src
    call    copy
```

```
lea      dx, dst
dos      9
dos      04c
```

Во всех модулях, кроме главного, директиву `segment` можно опустить — трансляция по умолчанию идет в сегмент `_text` класса `'code'` (тип объединения — `public`). Опущены `extrn`-описания — `a86` и так считает все неопределенные имена внешними. Объявление `extrn` в `a86` требуется тогда лишь, когда транслятор не в состоянии определить тип неизвестного имени при опережающих ссылках (см. *разд. 9.5*). Объявление `public` тоже требуется не часто — `a86` считает глобальными все имена, определенные в модуле, если только не задана директива `public`. Достаточно задать одно объявление `public`, чтобы сделать все остальные имена (не перечисленные в директивах `public`) невидимыми за пределами модуля.

Контрольные вопросы

1. Если настройки `ds` и `es` в начале `exe`-программы опустить, то в каких областях памяти будут находиться источник и приемник команды `movs`?
2. Что произойдет, если вызов `copy` запрограммирован как ближний, а выход из `copy` выполняется командой `retf`?
3. Каким будет результат выполнения подпрограммы `copy`, если предположить, что в начале `prog2` вместо настройки `ds` заданы следующие команды:

```
mov      ds, 0b800
mov      [0], '?'
```

4. Что произойдет при компоновке, если в `prog2`:
 - убрать атрибут `public` из директивы `segment`;
 - или поменять имя класса на `'data'`;
 - или поменять имя сегмента на `'prog2'`.

10.6. Практикум

По умолчанию `a86` создает исполняемую программу `com`-формата. Трансляция в `obj`-файл при использовании `a86` задается ключом `+o`. Чтобы отличать исходные файлы `exe`-программ, рекомендуем давать им расширение `08`.

10.6.1. Компоновка

Выполните трансляцию программы `prog_a.08`. Затем вызовите компоновщик `link`. Если не указывать параметров командной строки `link`, то имена входных и выходных файлов компоновщик спросит. Рекомендуется запускать `link` следующим образом:

```
link prog_a,,, /map
```

Здесь задана компоновка программы, состоящей из единственного объектного модуля `prog1.obj`, с записью результата в одноименный файл `prog1.exe`. Ключ `/map` заказывает отчет о результатах компоновки, или *карту загрузки*, которая будет записана в файл с расширением `map` (этот отчет получается более подробным, нежели при задании `map`-файла в диалоге с компоновщиком).

Примечание

Вместо `link` можно воспользоваться `tlink` (например, из Borland C). Вызов следующий: `tlink prog_a /m`, где `/m` задает генерирование карты загрузки.

Просмотрите содержимое полученного `map`-файла. В начале перечислены имена сегментов, в графе `Start` их расстояния (в байтах) от начала программного сегмента. Сегментные адреса *относительные* — именно в таком виде их значения и записаны в теле `exe`-файла. Класс не задан.

Примечание

В подробный отчет также включен перечень `public`-имен с указанием адресов в формате `сегмент:смещение`. Заметим, что сегментный адрес имени — относительный (отсчитан от начала загрузочного модуля). Абсолютный адрес определится только при загрузке, после получения блока памяти для загрузочного модуля.

Наиболее частые ошибки при компоновке:

- ❑ "Unresolved reference" или "Unresolved external": ссылка на имя не разрешилась, т. е. объекта с таким именем нет в компонуемых файлах.
- ❑ "Fixup overflow": ближний вызов для `call` невозможен, т. к. адресат находится в другом сегменте; "переполнение" означает, что переход должен быть дальним, как ближний он невозможен. Обычно причина в том, что компоновщик не стал объединять сегменты из разных модулей — скорее всего, не совпадают имена сегментов, имена классов, или тип объединения — не `public`.

10.6.2. Организация отладки

Вызовите отладчик с полученной `exe`-программой:

```
d86 prog_a.exe
```

Пройдите программу по шагам. Обратите внимание на странные имена меток, не соответствующие исходному тексту. Отладчик различает имена только по числовым значениям, не имея понятия о контексте (т. е., в каком логическом сегменте было определено имя). Поэтому, например, константа `size` вдруг отображается в виде метки (!) по адресу, совпавшему со значением `size`.

Из этого примера уже понятно, что от `sum`-файла лучше отказаться. При компоновке нескольких модулей — тем более, т. к. смещения при объединении сегментов корректируются; содержимое `sum`-файла, полученное при трансляции, совсем уже не соответствует исполняемой программе. Следует отключить генерирование `sum`-файла опцией `+s`; прежний `sum`-файл удалите вручную.

Корректная символьная информация содержится в *карте загрузки*. Например, для просмотра содержимого массивов `src` и `dst` в отладчике следует сначала выяснить значения этих имен из карты загрузки. Оба, оказывается, равны нулю. Запустите отладчик и задайте в двух окнах отображения памяти:

```
r,ds,0  
r,es,0
```

Первое из окон предназначено для отображения `dst`, но сейчас в нем выведено что-то другое. Это — содержимое *PSP* со смещения 0, поскольку по окончании загрузки `ds` настроен на *PSP*. Выполните команды настройки `ds` и `es` в начале программы, затем остальную программу.

Итак, при запуске `a86` для получения объектного модуля требуется задать ключи `+o` и `+s` (вместе — в виде `+os`). Для компоновки с модулями, разработанными на языках высокого уровня, рекомендуется также отключить преобразование имен глобальных объектов к верхнему регистру (ключ `+C`). В целом, вызов `a86` должен выглядеть как:

```
a86 +Cos prog_a.08
```

Внимание!

Ключ `+C` — обязательно с большой буквы!

Создайте программу `prog_b.exe`. Обратите внимание на изменения в `map`-файле.

10.6.3. Компоновка многомодульной программы

Выполните трансляцию модулей `prog1` и `prog2`. Для объединения модулей запустите компоновщик:

```
link prog1+prog2,,, /map
```

Результаты (еке- и `map`-файлы) будут сгенерированы с именем `prog1`. Если изменить порядок следования модулей, то результаты пойдут в файлы с именем `prog2`. Изменится и порядок расположения объектов в программе, о чем свидетельствуют `map`-файлы.

Проверьте возможность неявного задания `public`- и `extrn`-объявлений, построив программу из модулей `prog1a` и `prog2a`.

10.6.4. Задание на самостоятельную работу

На основе задания из *главы 9*, выполните следующее.

1. Получите решение задачи в ехе-формате, при условии: все данные определены в одном логическом сегменте, а весь код запрограммирован в другом сегменте.
2. Добавьте в ехе-программу отображение какой-либо числовой характеристики задачи. Для этого оформите подпрограмму `wrInt` вывода числа из *разд. 8.1* в виде отдельного модуля. Вызов `wrInt` запрограммируйте в главном модуле, где поместите решение задания из *главы 9*.

Примечание

Подпрограмма `wrInt` использует собственный буфер для записи цифр результата. Этот буфер следует поместить в сегмент данных, объединяемый с сегментом данных главного модуля (попросту, в один и тот же сегмент). Код `wrInt` и код главной программы также следует объединить в одном сегменте. При этом код и данные `wrInt` должны находиться в одном *исходном* модуле.



Часть II

Расширенные возможности ассемблера

Глава 11. Макрокоманды и условная трансляция

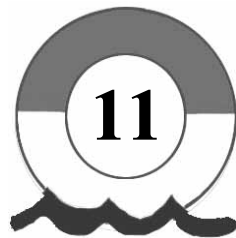
Глава 12. Структурный ассемблер

Глава 13. Интерфейс с языком C

Глава 14. Обработка BCD-данных

Глава 15. Математический сопроцессор

ГЛАВА 11



Макрокоманды и условная трансляция

— Я уже готов махнуть на все рукой и сдаться. Будем смотреть правде в глаза — мозгов у меня маловато. — Он опять разложил ряды спичек и начал выбирать их, играя сам с собой. — Все время работаю с ними и не вижу никакого улучшения.

— Конечно, ты работаешь! — сказал Элфи. — Все над чем-нибудь работают. Вставать с постели — это работа! Брать еду с тарелки и отправлять ее в рот — тоже работа! Но есть работа и упорная работа. Если ты хочешь выбиться, хочешь козырнуть чем-то, ты должен работать упорно... Конечно же, все работали во времена Джорджа Вашингтона, но Джордж Вашингтон работал упорно. Все работали во времена Шекспира, но Шекспир работал упорно. Я добился своего, потому что работаю упорно.

Курт Воннегут. "Утопия 14"

В этой главе рассмотрены средства для преобразования исходного текста программы в начале трансляции. Их применение позволяет сократить программирование похожих фрагментов исходного текста или исключить отдельные фрагменты в зависимости от условий.

Программирование сходных фрагментов сокращается при помощи *макрокоманд*. Макрокоманда представляет собой набор операторов, которому при определении назначено имя; это имя в дальнейшем задает подстановку операторов из набора. Параметры в определении макрокоманды позволяют задавать в сокращенной форме фрагменты одинаковой структуры, но разного содержания.

Директивы *условной трансляции* позволяют исключить из трансляции отдельные фрагменты исходного текста — в зависимости от атрибутов пользовательских имен. Применение директив условной трансляции избавляет от необходимости плодить копии исходного текста с разными версиями программы; все варианты хранятся в одном исходном файле.

11.1. Макрокоманды

Макрокоманда означает два объекта — макроопределение и макроподстановку — с общим именем. В *макроопределении* задана последовательность операторов, ей присвоено пользовательское имя. После того как макрокоманда определена, ее имя задает *подстановку* последовательности операторов из определения.

Фрагмент, заданный в макроопределении, в общем случае, содержит ссылки на формальные параметры; при подстановке эти ссылки заменяются фактическими параметрами, указанными после имени макрокоманды.

Определение состоит из имени, ключевого слова `macro`, и тела определения — до завершающей директивы `#em` (End of Macro).

11.1.1. Макрокоманды без параметров

Оператор *вызова макрокоманды* состоит из имени и списка фактических параметров. Если в теле макрокоманды нет ссылок на формальные параметры, то для вызова макрокоманды достаточно указать только ее имя, как в листинге 11.1.

Листинг 11.1. Макрокоманды без параметров (см. ex1.8)

```
print    macro
        mov     ah, 9
        int     021
#em

exit     macro
        mov     ah, 04c
        int     021
#em

...
done:    lea     dx, success
        print                   ; (1)
        mov     al, 0           ; errorlevel := 0
        exit
...
fail:    lea     dx, failure
        print                   ; (2)
        mov     al, 1           ; errorlevel := 1
        exit                   ; (3)
```

В листинге 11.1 определены следующие макрокоманды:

- `exit` — для выхода в DOS;
- `print` — для вывода сообщения.

Имена `success` и `failure` определены где-то в другом месте исходного текста, вместе со строками сообщений.

Определения макрокоманд `print` и `exit` при трансляции записываются в общую *таблицу имен* и считываются из таблицы в точках (1), (2) и (3); на вход транслятора поступает текст из макроопределений.

Листинг 11.2. Текст на входе транслятора в результате макроподстановок

```
...
done:  lea    dx, success
      mov    ah, 9
      int    021
      mov    al, 0          ; errorlevel := 0
      mov    ah, 04c
      int    021
      ...
fail:  lea    dx, failure
      mov    ah, 9
      int    021
      mov    al, 1          ; errorlevel := 1
      mov    ah, 04c
      int    021
```

Текст, поданный на вход транслятора, сохраняется в файле-листинге (если `a86` вызван с ключом `+L`): строки макроподстановки отмечаются в `lst`-файле буквой `m`, слева от подставляемых операторов. Текст в самом исходном файле не изменяется.

11.1.2. Макрокоманды с параметрами

Параметры в макроопределении позволяют настраивать подстановку, т. е. изменять формируемый фрагмент в зависимости от параметров вызова.

Формальные параметры, т. е. параметры в определении макрокоманды, обозначаются `<n>`, где `<n>` — порядковый номер параметра при вызове. Например, `#1` — первый параметр, `#2` — второй параметр, `#(10)` — десятый параметр. Номер задается или одной цифрой 1—9, или числом в скобках в пределах 1—127.

При вызове макрокоманды фактические параметры записываются после имени макрокоманды; если параметров несколько, они разделяются запятыми. Примеры определения и вызова макрокоманд с параметрами приведены в листинге 11.3.

Листинг 11.3. Макрокоманды с параметрами (см. ex2.8)

```
dos    macro
        mov     ah, #1
        int     021
#em

print  macro
        lea     dx, #1
        dos     9                ; (1)
#em

exit   macro
        print   #1                ; (2)
        mov     al, #2
        dos     04c              ; (3)
#em

...
start: print    hello_world
...
done:  exit     success, 0
...
fail:  exit     failure, 1
```

По сравнению с предыдущим примером здесь добавлен вывод сообщения (на метке `start`). В определениях макрокоманд `dos`, `print`, `exit` имеются ссылки на первый параметр `#1`. В вызовах также задано по одному параметру. Определения `print` и `dos` являются *вложенными*: в определении макрокоманды `print` задана подстановка ранее определенной макрокоманды `dos` (1), а в определении `exit` — аналогичное обращение к макрокомандам `print` (2) и `dos` (3).

В определении макрокоманды `dos` предполагается, что номер функции DOS задан первым параметром вызова. Первый параметр макрокоманды `print` — адрес строки для вывода на экран. В макрокоманде `exit` первый параметр задает адрес строки сообщения, а второй параметр — код возврата в DOS. (В batch-файлах код возврата доступен по имени `ErrorLevel`.)

Листинг 11.4. Результат макроподстановок для программы из листинга 11.3

```
...
start: lea     dx, hello_world
        mov     ah, 9
        int     021
...
```

```

done:   lea     dx, success
        mov     ah, 9
        int     021
        mov     al, 1
        mov     ah, 04c
        int     021
        ...
fail:   lea     dx, failure
        mov     ah, 9
        int     021
        mov     al, 1
        mov     ah, 04c
        int     021

```

11.1.3. Правила подстановки параметров

Подстановка фактических параметров вызова на место формальных параметров макроопределения выполняется по следующим правилам.

- ❑ Если фактический параметр не задан, то на место формального параметра подставляется пустой текст, т. е. ничего.
- ❑ При подстановке определены все 127 параметров; не заданные фактические параметры тоже определены, как пустой текст.
- ❑ Избыточные фактические параметры (на которые в теле макроопределения нет ссылок) игнорируются; задание лишних параметров не ошибка.
- ❑ Фактический параметр подставляется в том виде, в котором он задан — буквально, с любыми литерами кроме запятых, разделяющих параметры.
- ❑ Если фактический параметр задан в кавычках, его текст передается вместе с кавычками.
- ❑ Запятые входят в параметр, только если он задан в кавычках.
- ❑ Пробелы в начале и в конце текста фактического параметра при подстановке отбрасываются.

Листинг 11.5. Пример для иллюстрации правил подстановки параметров (см. ex3.8)

```

def_msg macro
#1      db      #2, 13, 10, '$'
#em

def_msg success, 'All is well, that ends well'
def_msg failure, 7, 'Hit ', 7, 'with cmd tail', 7
def_msg, 'Non-addressable message'
def_msg cr_lf

```

Макрокоманда `def_msg` определяет массив байт, заданный вторым параметром, добавляя к этому массиву литеры: возврат каретки, перевод строки, ограничитель '\$'. Имя, заданное первым параметром, ставится перед директивой `db`; таким образом, первый параметр задает символический адрес массива байт.

В первом вызове `def_msg` задано два параметра — столько, сколько предусмотрено в макроопределении. Запятая внутри строки `'All is well, that ends well'` не является разделителем параметров, поскольку находится внутри кавычек. Результат подстановки следующий:

```
success db      'All is well, that ends well', 13, 10, '$'
```

Во втором вызове `def_msg` задано шесть параметров. Избыточные параметры — `'Hit ', 7, 'with cmd tail', 7` — игнорируются; результат макроподстановки следующий:

```
failure db      7, 13, 10, '$'
```

В третьем вызове задано два параметра, первый — пустой. Результат подстановки синтаксически правильный, только у сообщения нет символического адреса:

```
db      'Non-addressable message', 13, 10, '$'
```

В последнем вызове задан только первый параметр; второй параметр, в результате, пустой. Итог макроподстановки — сообщение с адресом, но без содержания; в нем запрограммирован только перевод строки:

```
cr_lf db      13, 10, '$'
```

Рассмотренные примеры показывают, что количество фактических параметров в вызове может быть любым. Также, пустой параметр не является ошибкой сам по себе. Если результат подстановки не соответствует синтаксису ассемблера, то причина этого не в длине параметра, а в его содержании.

```
def_msg ax, 'Axiliary Ax'
dos
```

Выполните подстановки для этих вызовов, укажите синтаксические ошибки.

11.1.4. Функции от параметров

Функции `#n` (Number), `#v` (Value), `#s` (Size) задают подстановку не самого текста фактического параметра, а номера параметра, его числового значения или количества литер в тексте параметра. Эти *численные характеристики* параметра подставляются в текущей (по умолчанию, десятичной) системе счисления.

Пример использования функции `#s` — задание строки литер в стиле языка Turbo Pascal. Данные о строке содержатся в массиве из 256 байт; в началь-

Листинг 11.6. Макрокоманда для определения строки в стиле Turbo Pascal (см. ex4.8)

Результат макроподстановки, заданной в листинге 11.6, следующий:

```
hello db 16, 'Qqqqquuuuuuuuuuu!', (255 - 16) dup 0
```

В листинге 11.7 приведен фрагмент программы с применением функции `#v`. При выполнении этого фрагмента на экран будут выведены значения `1001xb`, `0100`, `'0'` в десятичном формате.

Листинг 11.7. Пример использования функции #v (см. ex5.8)

```

WrDec      macro
            jmp      >m1
            def_msg  m2, '#v1'
m1:         print    m2
#em

            WrDec    1001xb
            WrDec    0100
            WrDec    '0'

```

Функции, перечисленные в табл. 11.1, возвращают числовое значение, связанное с указанным параметром `<prm>`. Примеры использования этих функций мы только что рассмотрели.

Таблица 11.1. Функции, возвращающие числовую характеристику параметра

Функция	Название	Результат
n<prm>	Number	Номер параметра
v<prm>	Value	Значение выражения, заданного параметром
s<prm>	Size	Число литер в параметре

Функции из табл. 11.2 возвращают *параметр*. Функция #1 (Last) задает подстановку последнего фактического параметра. Функция #b<prm> (Before) подставляет параметр, предшествующий <prm>. Так, например, #b1 означает предпоследний параметр. Аналогично, #a<prm> (After) — это параметр, следующий за <prm>.

Таблица 11.2. Функции, возвращающие параметр

Функция	Название	Результат
1	Last	Последний параметр
b<prm>	Before	Параметр, предшествующий <prm>
a<prm>	After	Параметр, следующий за <prm>

Функции, приведенные в табл. 11.1 и 11.2, группируются под одним знаком #. Например, #n1 — это номер последнего параметра, #b1 — предпоследний параметр, #vbl — значение выражения, заданного предпоследним параметром в списке, #sbbbl — число литер в четвертом параметре от конца списка. Повторять #b в одной #-конструкции можно не больше четырех раз, #a — не больше трех.

Примечание

Функции #n, #a, #b применяются, как правило, к управляющей переменной *цикла*. Вне циклов, когда параметр фиксирован, пользы от функций #n, #a, #b никакой: #b2 — то же самое, что #1; #a2 — это #3; #n9 — число 9.

11.1.5. Циклы по параметрам

Цикл по параметрам позволяет выполнить для нескольких параметров подстановку по одному и тому же шаблону; на каждой итерации в подстановке участвует очередной фактический параметр.

Такие макроподстановки уже встречались нам при задании встроенных макрокоманд а86. Определения встроенных макрокоманд нам недоступны, поскольку они "защиты" в транслятор, а примеры вызовов следующие:

```
inc    ax, bx, cx
push   bp, bx
```

Машинные инструкции inc и push рассчитаны на один операнд. Встроенные в а86 макрокоманды inc и push принимают произвольное количество параметров, для каждого создается отдельная машинная инструкция.

R-циклы

Составим макрокоманду, аналогичную `push`. Назовем ее `_push`, поскольку имя `push` занято транслятором:

```
_push macro
#rxll
    push    #x
#er
#em
```

Цикл по параметрам задан `#`-конструкцией `#r<v><p1><p2>`, где `<v>` — обозначение управляющей переменной цикла (буква из набора `w, x, y, z`), а `<p1>` и `<p2>` — границы изменения переменной `#<v>`. Обозначение `#<v>` на каждой итерации задает подстановку очередного фактического параметра.

Верхний предел `<p2>` часто задают при помощи функции `#l`. В макрокоманде `_push` из примера цикл выполняется для всех параметров вызова — от `#1` до `#l`. Если в вызове `_push` вообще не задать параметров, то цикл не выполнится ни разу.

Тело `r`-цикла ограничивается директивой `#er`. Эту директиву задавать не обязательно, если конец цикла совпадает с концом макроопределения (т. е. когда за `#er` сразу следует `#em`). Пример `r`-цикла без `#er` показан в листинге 11.8.

Листинг 11.8. Пример цикла по параметрам (см. ex6.8)

```
_push macro
#rxll
    push    #x
#em

    _push   ax, bx, 2
```

В вызове `_push` задано три фактических параметра; цикл повторяется три раза со значениями `#x = #1`, `#x = #2`, `#x = #3`:

```
push    ax        ; #1
push    bx        ; #2
push    2          ; #3
```

Q-циклы

Подстановка параметров в `q`-цикле выполняется в обратном порядке — от конца списка параметров к началу; в этом — все отличие от `r`-цикла.

Цикл определяется директивой `#q<v><p1><p2>`. Номер `<p1>` должен быть больше номера `<p2>`, иначе цикл не выполнится ни разу. Конец `q`-цикла

обозначается директивой `#eq`, которую можно опустить, если за `#eq` следует `#em`.

В качестве примера `q`-цикла определим макрокоманду, аналогичную встроенной макрокоманде `mov`. Напомним возможности `mov` в `as6`:

```
mov    ax, bx, cx, 1
```

В результате макроподстановки генерируются три машинные инструкции пересылки; выполняющие копирование в последовательности $1 \rightarrow cx \rightarrow bx \rightarrow ax$:

```
mov    cx, 1
mov    bx, cx
mov    ax, bx
```

В каждом операторе присутствуют два смежных параметра из списка фактических параметров. Определение аналогичной макрокоманды приведено в листинге 11.9. Для доступа к параметру справа от текущего, в теле цикла применяется функция `#b` — `before`; на каждой итерации подставляются `#x` и `#bx` — текущий параметр и предшествующий ему.

Листинг 11.9. Цикл по параметрам в обратном порядке (см. ex7.8)

```
_mov    macro
#qxl2
        mov    #bx, #x
#em

        _mov    ax, bx, cx, 1
```

Рассмотрим обработку вызова `_mov`. Номер параметра изменяется от 4 (результат функции `#1`) до 2. То есть, переменная `#x` при обработке вызова из примера принимает значения `#4`, `#3`, `#2`. Подстановка выполняется три раза:

```
mov    cx, 1      ; mov    #3, #4
mov    bx, cx     ; mov    #2, #3
mov    ax, bx     ; mov    #1, #2
```

Определите `_mov` с использованием функции `#a` взамен `#b`. Как при этом изменится задание диапазона?

Задание шага в `r`- и `q`-циклах

По умолчанию, номер параметра в цикле изменяется с шагом 1. Чтобы увеличить значение шага, определение цикла следует завершать не `#er` или `#eq`, а `#e<n>`, где `<n>` — цифра от 1 до 9.

Увеличение шага позволяет обрабатывать параметры *группами*. Например, таблицу кодировки из *разд. 7.4.2* удобно было бы задавать следующим образом:

```
map_1    db
          map      'A', 'Z', 'a', 'z', 256
```

Напомним задачу. Требуется определить массив из 256 байт так, чтобы значения были ненулевыми у тех байтов, порядковые номера которых принадлежат одному из указанных диапазонов.

Первый диапазон в вызове макрокоманды `map` задан первой парой параметров: #1 и #2. Значения #1 и #2 для данного примера — 'A' и 'Z'. Второй диапазон задан парой #3 и #4 ('a' и 'z'). Последний параметр, без пары — это размер таблицы в байтах. В листинге 11.10 приведено решение задачи при условии, что значения параметров расположены в неубывающей последовательности.

Листинг 11.10. Макроопределение для задания таблицы принадлежности (см. `ex8_1.8`)

```
map      macro
          db      #1 dup 0
#rx21
          db      #x - #bx + 1 dup 1
          db      #ax - #x - 1 dup 0
#e2
#em
```

Переменная `#x` изменяется от #2 с шагом два. На каждой итерации `#x` представляет верхнюю границу очередного диапазона. Длина текущего диапазона вычисляется на основании `#bx` (это нижняя граница текущего диапазона). Расстояние от конца текущего диапазона до начала следующего (или до конца таблицы — на последней итерации) выясняется с использованием `#ax` (это нижняя граница следующего диапазона).

Для вызова из примера цикл разворачивается следующим образом:

```
db 'Z' - 'A' + 1 dup 1      ; #2 - #1 + 1 dup 1    (#x = #2)
db 'a' - 'Z' - 1 dup 0     ; #3 - #2 - 1 dup 0
db 'z' - 'a' + 1 dup 1     ; #4 - #3 + 1 dup 1    (#x = #3)
db 256 - 'z' - 1 dup 0     ; #5 - #4 - 1 dup 0
```

Последний параметр, без пары, задает суммарный размер таблицы. Значение его всегда одно и то же, поэтому следовало бы его сократить; вызов выглядел бы следующим образом:

```
map_1    db
         map      'A', 'Z', 'a', 'z'
```

Теперь при определении расстояния до конца таблицы полагаться на `#ax` нельзя — на последней итерации `#ax` подставляет пустой текст. Поэтому ориентируемся на верхнюю границу *предыдущего* диапазона. Использовать `#bx` при этом нельзя, т. к. на первой итерации `#bx` подставляет пустой текст. Поэтому верхняя граница "предыдущего диапазона" представлена именем `m1`, значение которого определено перед входом в цикл.

Листинг 11.11. Усовершенствованный вариант макроопределения `map` (см. ex8_2.8)

```
map      macro
m1       = -1
#rx11

         db      #x - m1 - 1 dup 0
         db      #ax - #x + 1 dup 1
         m1      = #ax

#e2
         db      255 - m1 dup 0

#em
```

Примечание

Операция `=` аналогична `equ`, но она позволяет повторно определить имя, даже если оно не локальное. (Локальное имя допускает переопределение в любом случае.) Оператор `=` традиционно используется при программировании макроопределений в разных языках ассемблера.

11.1.6. Цикл по литерам параметра

Цикл по литерам параметра задается конструкцией `#c<v><p>`. В цикле перебираются *литеры* параметра `<p>`, по одной на каждой итерации.

Рассмотрим вариант макрокоманды `map`, где диапазоны заданы парами литер. Пусть вызов — для диапазонов `'A'-'Z'` и `'a'-'z'` — выглядит так:

```
map_1    db
         map      AZaz
```

Поскольку в `c`-цикле кавычки игнорируются, следующий вызов аналогичный:

```
map_1    db
         map      'AZaz'
```

Примечание

Кавычки требуются только при задании хвостовых и ведущих пробелов. Без кавычек краевые пробелы отбрасываются.

Листинг 11.12. Определение `map_1` с использованием `c`-цикла (см. `ex8_3.8`)

```
map      macro
m1       = -1
#cx1
        db      '#x' - m1 - 1 dup 0
        db      '#ax' - '#x' + 1 dup 1
        m1      = '#ax'
#e2
        db      255 - m1 dup 0
#em
```

По сравнению с предыдущей реализацией `map` изменилось задание начала цикла — `#cx1` вместо `#rx11`. Значения параметров в теле цикла подставляются теперь внутрь кавычек. Переменной `#x` соответствует одна литера: в начале — `A`, затем `Z`, потом `a`, далее `z`. Задание диапазонов в вызове макрокоманды стало более компактным, но теперь нельзя задать границу в числовом виде — только литерой.

11.1.7. Циклы, не зависящие от параметров

Предположим, требуется задать последовательность байт со значениями 0, 2, 4, ..., 198. Для решения этой задачи нужно сотню раз повторить следующий фрагмент:

```
db      m1
m1      = m1 + 2
```

Предварительно `m1` должно быть определено нулем.

Макрокоманда для решения этой задачи программируется с использованием `r`- или `q`-цикла. Вызов — без параметров.

Листинг 11.13. Определение возрастающей последовательности байтов (см. `ex9_1.8`)

```
ladder  macro
m1      = 0
#rx1(100)
        db      m1
        m1      = m1 + 2
#em

ladder
```

В `r`-цикле из листинга 11.13 переменная `#x` принимает значения фактических параметров `#1`, `#2`, ..., `#100`. Параметры в вызове `ladder` не заданы вовсе, но по форме цикл правильный, поскольку все 127 фактических параметров определены, хотя бы и как пустые. Более компактное решение с применением функции `#n` приведено в листинге 11.14.

Листинг 11.14. Решение с применением функции `#n` (см. ex9_2.8)

```
ladder macro
#rx(1) (100)
    db      (#nx - 1) * 2
#em
```

Функция `#nx` (номер фактического параметра, соответствующего переменной `#x`) принимает значения 1, 2, 3, ..., 99, 100. В результате вызова `ladder` генерируется последовательность операторов:

```
db      (1 - 1) * 2
db      (2 - 1) * 2
..
db      (100 - 1) * 2
```

Поскольку номер параметра ограничен диапазоном 1—127, число повторений также ограничено 127. Для большинства практических приложений этого недостаточно. Имеется возможность увеличить верхнюю границу цикла до 254, но с оговоркой: результат функции `#n` замкнут в кольце 1—127. То есть, `#n(128)` равно единице, `#n(129)` — двойке и т. д.

Листинг 11.15. Пример увеличения верхней границы цикла до 254 (см. ex9_3.8)

```
ladders macro
#rx(1) (254)
    db      #nx
#em
```

При вызове этого варианта макрокоманды вместо последовательности, равномерно возрастающей от 1 до 254, получится две возрастающие последовательности: 1, 2, ..., 127, 1, 2, ..., 127. А если задать верхнюю границу равной 255, цикл будет *бесконечным*.

Листинг 11.16. Пример бесконечного цикла

```
sizyphe macro
#rx(1) (255)
    db      #nx
#em

sizyphe
```

Рекорд 254 итерации побивается при использовании вложенных `r`- или `q`-циклов. Есть и другое решение, не столь лаконичное, но универсальное — за счет директив условной трансляции в сочетании с оператором `#ex` принудительного завершения макроподстановки. Рассмотрим сначала первый вариант.

11.1.8. Вложенные циклы

Вложенные циклы применяются для сложной обработки фактических параметров, а также как один из вариантов задания циклов с фиксированным числом повторений.

Вложенные циклы с фиксированным числом повторений

Пусть требуется определить 1000 слов со значениями, убывающими от 1000 до 1. Решение с применением вложенных `q`-циклов приведено в листинге 11.17.

**Листинг 11.17. Вложенный цикл с фиксированным числом повторений
(см. ex10_1.8)**

```
down    macro
#qx(10)1
#qy(100)1
        dw      (#nx - 1) * 100 + #ny
#em

        down
```

Выражение `#nx - 1` принимает значения от 9 до 0 во внешнем цикле, а выражение `#ny` изменяется от 100 до 1 во внутреннем цикле.

Переменные, управляющие циклами разных уровней вложенности, в листинге 11.17 обозначены по-разному — для того, чтобы в теле вложенного цикла были доступны обе переменные. В другом варианте того же макроопределения, в листинге 11.18, обозначение `#z` в (2) соответствует переменной внутреннего цикла, и то же самое обозначение `#z` в (1) соответствует переменной внешнего цикла.

**Листинг 11.18. Одноименные переменные во вложенных циклах
(см. ex10_2.8)**

```
down    macro
#qz(10)1
        m1      = #nz - 1          ; (1)
```

```
#qz(100)1
    dw    ml * 100 + #nz    ; (2)
#em
```

Теоретически, число вложений не ограничено, но если оно больше четырех, придется дублировать имена переменных, как в примере из листинга 11.18. Напомним, что переменные цикла могут обозначаться всего четырьмя буквами — *x*, *y*, *z*, *w*.

Сложная обработка фактических параметров

Рассмотрим пример сложной обработки параметров, с применением вложенных циклов. Требуется задать матрицу следующего вида:

a(1)	a(2)	a(3)	..	a(n)
0	a(2)	a(3)	..	a(n)
0	0	a(3)	..	a(n)
..
0	0	0	..	a(n)

Значения *a(1)*–*a(n)* задаются в параметрах вызова макрокоманды; количество фактических параметров определяет размерность матрицы. Макроопределение и пример вызова для решения этой задачи приведены в листинге 11.19.

Листинг 11.19. Решение задачи по определению матрицы (см. ex11_1.8)

```
matrix macro
#rx11
    db    (#nx - 1) dup 0
#ryx1
    db    #y
#em

matrix 1, 3, 5, 7, 11, 13
```

Начальное значение переменной *#y* внутреннего цикла устанавливается равным текущему значению переменной внешнего цикла *#x*. После очередной итерации внешнего цикла диапазон изменения *#y* сокращается.

11.2. Средства условной трансляции

Условная трансляция дает возможность собирать текст, предназначенный для ассемблирования, из фрагментов исходного текста. Отдельные части исходного текста при этом отбрасываются. Дробление исходного текста директивами условной трансляции при развитии программы позволяет под-

держивать разные версии программы. Номер версии, заданный константой, входит в условия трансляции, как показано в примере из листинга 11.20.

Листинг 11.20. Задание условной трансляции разных версий программы (см. ex12_1.8)

```
version equ      1

#if version GT 3
    ...          ; (1)
#elseif version EQ 1
    ...          ; (2)
#else
    ...          ; (3)
#endif
```

В зависимости от значения константы `version` транслируется фрагмент (1), (2) или (3). В примере условия в директивах `#if`, `#elseif` заданы в виде отношений `GT` (Geater Than), `EQ` (Equal).

11.2.1. Директива `#if..#endif`

Область условной трансляции заключена между парой директив `#if..#endif`. Сначала проверяется условие в `#if`. Если оно выполнено, на вход транслятора передается фрагмент (1) — от `#if` до `#elseif`. Иначе, последовательно проверяются условия в последующих директивах `#elseif`. Если ни одно из условий в `#if` и `#elseif` не выполнено, выбирается фрагмент между `#else` и `#endif`.

В конструкции `#if..#endif` обязательны только ключевые слова `#if` и `#endif`. Допускается сколько угодно директив `#elseif` (или вообще ни одной) и только одна директива `#else` — тоже необязательная.

11.2.2. Макроопределения и условная трансляция

Сделаем так, чтобы макрокоманда `ladder` генерировала или байты, или слова — в зависимости от номера версии программы. Решение показано в листинге 11.21.

Листинг 11.21. Условная трансляция макроопределения (см. ex9_4.8)

```
version equ 1

ladder macro
#rx(1) (100)
```

```

#if version EQ 1
    db      (#nx - 1) * 2
#else
    dw      (#nx - 1) * 2
#endif
#em

```

В зависимости от величины имени `version`, в макроопределении сохраняется либо оператор `db`, либо оператор `dw`. В таком усеченном виде определение (!) `ladder` сохраняется в таблице имен. Например, если `version = 2`, макроопределение будет запомнено в виде:

```

ladder macro
#rx(1)(100)
    dw      (#nx - 1) * 2
#em

```

Можно ли запрограммировать оба варианта `ladder` в одной версии программы? Пусть вариант выбирается при каждом *вызове* — например, в зависимости от константы `size`, значение которой мы будем менять между вызовами. Чтобы макроопределение сохранялось в таблице имен в первозданном виде, вместе с директивами условной трансляции, сами эти директивы следует задать с двойным знаком `##` — как показано в листинге 11.22.

Листинг 11.22. Условная трансляция макровывоза (см. ex9_5.8)

```

ladder macro
#rx(1)(100)
##if size EQ 1
    db      (#nx - 1) * 2
##else
    dw      (#nx - 1) * 2
##endif
#em

...
size = 1
ld1   db
      ladder

size = 2
ld2   dw
      ladder
...

```

Двойной знак `##` предохраняет от немедленного применения условной трансляции при разборе макроопределения. Директивы сохраняются в таб-

лице имен вместе с макроопределением. Теперь, в зависимости от значения `size` в момент вызова, будет выбран один из вариантов — либо с `db`, либо с `dw`.

11.2.3. Условия

В качестве условий применимы любые выражения, возвращающие результат типа `abs`. Условие выполнено, если значение выражения не ноль.

Операторы, из которых составляются выражения, в основном уже рассмотрены в *разд. 1.4*. Ниже приведены операторы, возвращающие *логический* результат 0 (`false`) или `-1` (`true`) — из числа наиболее часто применяемых. (Полный перечень операций см. в *приложении 6*.)

Арифметические отношения

Условия на основе отношений `>`, `<`, `>=`, `<=`, `=`, `!=` задаются в виде:

`a <rel> b`

где `a` и `b` — выражения типа `abs`, `a <rel>` — отношение, заданное аббревиатурой из табл. 11.3.

Таблица 11.3. Обозначение арифметических отношений в логических выражениях

Аббревиатура	Название	Отношение
EQ	Equal — равно	=
NE	Not Equal — не равно	!=
GT	Greater Than — больше	>
GE	Greater or Equal — больше или равно	>=
LT	Less Than — меньше	<
LE	Less or Equal — меньше или равно	<=

Пример:

`big equ offset end GT 1k`

При таком определении имя `big` принимает значение `true` (`-1`), если общая длина исполняемой программы (вместе с `PSP`) больше 1 Кбайт.

Объединение условий

Для объединения условий предназначены операторы `or` и `and`. Слева и справа от `or` или `and` допускаются "неточные" логические значения, когда

только ноль означает false, а все, что не ноль — это true. Результат — уже "точное" логическое значение: false = 0, true = -1.

Пример:

```
size      equ offset end - 0100

#if size EQ 0 or size GT 64k - 0100
    ?-Bad prg size
#endif
```

Строка `?-Bad prg size` передается на вход транслятора, если программа пустая или, наоборот, настолько велика, что ее код может быть затерт стеком. Это оператор заведомо ошибочный, для аварийного завершения трансляции.

Проверка существования

Оператор `def <name>` проверяет, определено ли (существует ли) имя `<name>`.

Само имя, без `def`, — это тоже оператор проверки существования. Заданная в виде `<name>` проверка проходит даже в том случае, если имя `<name>` не определено: оператор `<name>` возвращает true, если имя определено и его значение не ноль. Напротив, оператор `def <name>` с неопределенным именем `<name>` воспринимается как ошибка и приводит к аварийной остановке трансляции.

Определение имен в командной строке

Имя, управляющее условной трансляцией, не обязательно определять в тексте программы — имеется возможность задать имя в командной строке при вызове `a86`:

```
a86 =ver1 my_prog +L
```

При таком вызове `a86` к началу трансляции `my_prog` определено имя `ver1` со значением 1 (см. приложение 5).

Номер версии теперь определяется уже не значением имени, а только фактом его существования. То есть, версия следует из того, какое из имен определено — `ver1`, `ver2`, `ver3` и т. д. Пример условной трансляции из листинга 11.22 выглядит теперь так, как показано в листинге 11.23.

Листинг 11.23. Условная трансляция макровывоза (см. ex12_2.8)

```
#if ver1
    ...                ; (2)
#elseif ver2 or ver3
    ...                ; (3)
```

```
#else
    ...                ; (1)
#endif
```

Последовательность проверок отличается от исходного варианта в листинге 11.22. Проверка, аналогичная `ver GT 3`, стала теперь невозможной — для ее реализации потребовалось бы оценить все имена `ver<n>` для $<n> = 4, 5, 6$ и т. д. до неизвестного предела. Поэтому фрагмент (1) помещен в `#else`.

Если не определено имя `ver3`, будет выдано сообщение об ошибке. Причина: если оператор `or` включен в проверки, то имя справа от него должно быть определено. Следует исключить оператор `or`, продублировав директиву `#elseif` и последовательность операторов (3):

```
    ...
#elseif ver2
    ...                ; (3)
#elseif ver3
    ...                ; (3)
```

Сделаем решение более компактным, за счет применения оператора логической инверсии и правила Моргана.

Инвертирование логических значений

Для инвертирования *логических* значений специально предназначен оператор `!`. Его применение в сочетании с оператором `<name>` допускается даже в тех случаях, когда `<name>` не определено (в этом случае выражение `!<name>` равно `true`).

С учетом этой особенности оператора логической инверсии, и пользуясь правилом Моргана $!(a \text{ or } b) = !a \text{ and } !b$, заменим условие из листинга 11.23 на его эквивалент, устойчивый к неопределенным именам. Решение — в листинге 11.24.

Листинг 11.24. Замена `or` на `and` в условиях трансляции (см. ex12_3.8)

```
#if ver1
    ...                ; (2)
#elseif !(!ver2 and !ver3)
    ...                ; (3)
#else
    ...                ; (1)
#endif
```

Выражение `!(!ver2 and !ver3)` транслируется, даже если среди имен `ver2` и `ver3` есть неопределенное имя.

Пример условной трансляции в макроопределениях

Продemonстрируем возможности условной трансляции в макроопределениях.

Пример — макрокоманда `movs?` для копирования заданного числа байт. Счетчик байт задан первым параметром. Адреса источника и приемника в параметрах не указываются; предполагается, что адреса установлены в парах регистров `ds:si` и `es:di` к моменту вызова `movs?`, а флаг `d` сброшен.

Задача состоит в том, чтобы выбрать наилучший по скорости вариант в зависимости от счетчика байт — для случая, когда счетчик задан константой. Если счетчик задан именем переменной или названием регистра, его значение при трансляции неизвестно, и оптимизация средствами условной трансляции невозможна.

Во всех вариантах макроподстановки используется команда `movsb` и/или `movsw`. В следующем первоначальном варианте макроопределения `movs?` тип и значение параметра не анализируются:

```
movs?    macro
        push    cx
        mov     cx, #1          ; (1)
        rep     movsb
        pop     cx
#em
```

Начнем оптимизацию с определения *типа* параметра. Если выражение `type #1` возвращает ноль, значит, параметр задан константой (тип `abs`), и его величину имеет смысл анализировать на этапе трансляции. Если `type #1` возвращает число 2, счетчик задан словом (регистром или переменной типа `word`). В этом случае остается только одна возможность оптимизации — исключение лишней команды (1), когда параметр задан регистром `cx`.

Листинг 11.25. Условная трансляция по типу параметра (см. ex13.8)

```
movs?    macro
##if (type (#1)) eq 2
        push    cx
        ##if !(#1 = cx)
            mov     cx, #1
        ##endif
        rep     movsb
        pop     cx
##elseif (type (#1)) eq 0
        _movs?  #1
##else
        ?-Illegal type
##endif
#em
```

Оператор `=` в листинге 11.25 проверяет тексты на совпадение, не делая различий между прописными и заглавными буквами (напротив, операторы `eq` и `ne` проверяют точное совпадение или несовпадение).

Если тип параметра не `word` и не `abs`, трансляция прекратится с ошибкой. Если параметр имеет тип `abs` (`type #1 eq 0`), вызывается макрокоманда `_movs?`

Определение вспомогательной макрокоманды `_movs?` показано в листинге 11.26. Если счетчик нечетный (что проверяется операцией `#1 mod 2`), то генерируется одна команда `movsb`, а затем — последовательность `movsw`. Если число повторений `movsw` (константа `m1`) больше или равно пяти, применяется циклический вариант — с префиксом повторения `rep`. Если `m1` меньше 5, то применяется вариант `movsw m1`. Число после `movsw` задает количество экземпляров машинной инструкции `movsw` — это еще одна встроенная макрокоманда `a86`. Ноль после `movsw` допускается, в этом случае не будет сгенерировано ни одного экземпляра машинной инструкции `movsw`.

Листинг 11.26. Условная трансляция по значению параметра (см. ex13.8)

```
_movs?    macro
##if (#1) mod 2                ; (1)
    movsb
##endif
    m1 = (#1)/2                ; (2)
##if m1 lt 5
    movsw    m1
##else
    push     cx
    mov      cx, m1
    rep      movsw
    pop      cx
##endif
#em
```

Обратите внимание: в выражениях (1) и (2) параметр поставлен в скобки — на тот случай, если он задан выражением, например, `offset dst - offset src`. Скобки гарантируют, что значение параметра будет вычислено до применения операций `/` и `mod`, приоритет которых выше, чем у операции вычитания. По этой же причине в скобках записано выражение `type #1` из листинга 11.25. Если в выражениях (1) и (2) из листинга 11.26 вместо `#1` задать `#v1`, то скобки станут необязательными.

Примечание

Если в качестве фактического параметра при вызове `_movs?` задать `#v1` вместо `#1`, то при трансляции возникнет множество ошибок — по непонятной причине.

Листинг 11.27. Тестирование макрокоманды `movs?` (см. ex13.8)

```

        jmp      start

src      db      'Put_me_in_a_wheelchair,_get_me_to_the_show_'
        db      'Harry,_harry,_harry_before_I_go,_I_go_'
        db      "I_can't_control_my_fingers_"
        db      "I_can't_control_my_toes,_no-o-o-o-ouuuuuuu_"

size     equ     $-src
dst      db      size dup ?

test_    macro
        mov      ax, cx, 2
cp_ax:   movs?   ax
cp_cx:   movs?   cx
cpy_0:   movs?   offset dst - offset dst
#rx1(14)
cpy_#nx:
        movs?   #nx
#em

start:
        lea      di, dst
        lea      si, src
        test_
        int      020

```

В макрокоманде `test_` проверки макрокоманды `movs?` заданы пятнадцать вызовов `movs?` с разными константными значениями (от 0 до 14) параметра, и еще два вызова — с параметрами `ax` и `cx`.

11.2.4. Условная трансляция в циклах

Возвращаясь к задаче из *разд. 11.1.7*, рассмотрим еще один вариант ее решения при числе итераций больше 254.

Сделаем `r`-цикл бесконечным, с выходом по условию. Назначим верхнюю границу 255, в тело цикла поместим блок условной трансляции, а внутрь него — директиву принудительного завершения макроподстановки `#ex`.

Листинг 11.28. Условная трансляция в циклах (см. ex9_6.8)

```

ladder   macro
ml       = 0

```

```

#rx(1) (255)
    dw      m1
    m1      = m1 + 2
    ##if m1 GE #v1
    #ex
    ##endif
#em

    ladder  2000

```

Управление будет передано на `#ex`, когда значение `m1` станет больше значения первого параметра (`#v1`).

11.3. Метки в макроопределениях

В рассмотренных примерах имена в макроопределениях использовались как рабочие переменные — переопределяемые константы со значением типа `abs`.

Что, если поставить в теле макроопределения метку? Например, метка в следующем варианте макрокоманды `print` позволяет задавать строку сообщения в качестве фактического параметра.

Листинг 11.29. Метки и имена переменных в макроопределении
(см. `ex14_1.8`)

```

print  macro
        jmp      skip
msg    db        "#1 $"
skip:  lea        dx, msg
        dos      9
#em

        print   enter value -->
        ...
        print   ?-bad input

```

Директива `db` находится в теле макроопределения. Чтобы данные не попали в поток выполнения, их следует обойти командой `jmp`. В результате появится метка — `skip`. (Также требуется символический адрес для записи в `dx`, поэтому определено еще одно имя — переменная `msg`.)

При втором вызове `print` определения `skip` и `msg` повторятся, и на каждое повторение транслятор выдаст ошибку "Conflicting Multiple Definition Not Allowed". Текст на входе транслятора показан в листинге 11.30.

Листинг 11.30. Результат макроподстановок в программе из листинга 11.29

```

...
jmp      skip
msg      db      "enter value --> $"
skip:    lea      dx, msg
mov      ah, 9
int      021
...
jmp      skip
msg      db      "?-bad input $"          ; (!)
skip:    lea      dx, msg                  ; (!)
mov      ah, 9
int      021
...

```

Повторных определений не избегать. Чтобы правильно выполнить их трансляцию, следует воспользоваться локальными именами, как показано в листинге 11.31.

Листинг 11.31. Локальные имена в макроопределении (см. ex14_2.8)

```

print    macro
        jmp      m1
m2       db      "#1 $"
m1:      lea      dx, m2
dos      9
#em

        print   enter value -->
        ...
        print   ?-bad input

```

Теперь в результате двух обращений к `print` будет сгенерирован текст, показанный в листинге 11.32.

Листинг 11.32. Результат макроподстановок в программе из листинга 11.31

```

...
jmp      m1
m2       db      "enter value --> $"
m1:      lea      dx, m2
mov      ah, 9
int      021
...
jmp      m1                      ; (!)

```

```

m2      db      "?-bad input $"
m1:     lea      dx, m2
        mov      ah, 9
        int      021
        ...

```

Трансляция пройдет успешно, но при передаче управления на команду, отмеченную в листинге 11.32, вместо "?-bad input " будет выведено "enter value --> " и, скорее всего, бесконечное число раз.

Причина в том, что второй экземпляр команды `jmp m1` ссылается на *первый* экземпляр метки `m1`. Ссылка на `m1` — опережающая, и это следует указать явно, как показано в листинге 11.33.

Листинг 11.33. Опережающие ссылки в макроопределении (см. ex14_3.8)

```

print  macro
        jmp      >m1          ; !!!
m2      db      "#1 $"
m1:     lea      dx, m2
        dos      9

#em

        print   enter value -->
        ...
        print   ?-bad input

```

Результат макроподстановок показан в листинге 11.34.

Листинг 11.34. Результат макроподстановок в программе из листинга 11.33

```

        ...
        jmp      >m1
m2      db      "enter value --> $"
m1:     lea      dx, m2
        mov      ah, 9
        int      021
        ...
        jmp      >m1
m2      db      "?-bad input $"
m1:     lea      dx, m2
        mov      ah, 9
        int      021
        ...

```

Из этого примера следует вывод: если ссылка на локальное имя опережающая, значок `>` перед именем обязателен — по крайней мере, в макроопреде-

лениях. По той же причине, локальные имена вне и внутри макроопределений должны различаться. Например, автор a86 предлагает вне макроопределений начинать локальные имена буквой `l` (например, `l1`, `l2` и т. д.), а в макроопределениях — буквой `m` (например, `m1`, `m2` и т. д.).

11.4. Средства отладки макрокоманд

Для отладки программ с макрокомандами рекомендуется заказывать при трансляции файл-листинг. Из `lst`-файла хорошо видно, как происходят макроподстановки.

Выполните трансляцию примера `ex1.8`:

```
a86 ex1.8 +L
```

В файле-листинге колонки означают, слева направо:

- ☐ порядковый номер оператора;
- ☐ значение счетчика адресов `$` (для операторов, резервирующих память);
- ☐ сгенерированные коды.

Ниже приведен фрагмент файла `ex1.lst`.

```
....
23                                     print                      ; (1)
24 011A B4 09                        m    mov ah, 9
25 011C CD 21                        m    int 021
....
27                                     exit
28 0120 B4 4C                        m    mov ah, 04c
29 0122 CD 21                        m    int 021
```

Операторы из макроопределений помечены буквой `m`.

Если в макрокоманде используются директивы условной трансляции, метка `m` в листинге еще не означает, что вариант выбран. Выполните трансляцию примера `ex9_5.8`. В полученном `lst`-файле все операторы `ladder` помечены буквой `m`. Что именно выбрано — `db` или `dw`, можно выяснить только из значения счетчика адресов (поле счетчика непустое, если оператор резервирует память).

```
10 = : 0001                                size = 1
11                                ld1    db
12                                ladder
13                                m
14                                m    #if size EQ 1
15 0100 00                                m    db (1 - 1) * 2
```

```

16                                m    #else
17                                m    dw (1 - 1) * 2
18                                m    #endif
..
608                                m    #if size EQ 1
609 0163    C6                    m    db (100 - 1) * 2
610                                m    #else
611                                m    dw (100 - 1) * 2
612                                m    #endif
613
614 = : 0002                                size = 2
615                                ld2    dw
616                                ladder
617                                m
618                                m    #if size EQ 1
619                                m    db (1 - 1) * 2
620                                m    #else
621 0164    00 00                    m    dw (1 - 1) * 2
622                                m    #endif
...
1212                               m    #if size EQ 1
1213                               m    db (100 - 1) * 2
1214                               m    #else
1215 022A    C6 00                    m    dw (100 - 1) * 2
1216                               m    #endif

```

11.5. Практикум

Проверьте правильность макроподстановок в вышеприведенных примерах — по листингам и в отладчике. Для проверки результатов `ex11`, `ex13`, `ex9_5` в `d86` для вас заранее подготовлены одноименные `keystroke`-файлы.

11.5.1. Макрокоманды без циклов

Разработайте программу копирования стандартного входного потока (`stdin`) в стандартный выходной поток (`stdout`). Чтение `stdin` и запись в `stdout` оформите в виде макрокоманд. Вызов макрокоманд следующий:

```
rd_stdin  buffer, 1k
wr_stdout buffer, 1k
```

Первый параметр — это адрес области чтения-записи, второй — счетчик байт. Если при чтении или записи возникает ошибка, следует вывести сообщение и завершить программу. Перед успешным завершением также отобразите приличествующее случаю сообщение. Используйте готовые макро-

определения из файла `macros.inc` (в частности, макрокоманду `dos` — для вызова функций чтения-записи).

11.5.2. Макрокоманды с циклами

Разработайте макрокоманду `_shr`, которая выполняет действия, аналогичные инструкции `shr`, но для данных многократной размерности. Пример вызова:

```
_shr    dx, ax, bx, 1
```

Последний параметр задает число сдвигов, он может быть либо константой, либо регистром `cl` — как в команде `shr`. Остальные параметры — это составляющие операнда многократной размерности. В данном случае суммарная размерность — 48 бит (три слова). Наиболее значимая составляющая — `dx`, наименее значимая — `bx`.

Перед тем как программировать цикл, представьте, как выглядит и как выполняется требуемая последовательность инструкций:

```
shr     dx, 1           ; c = ?
rcr     ax, 1
rcr     bx, 1
```

На рис. 11.1 показан фрагмент сдвига для первых двух слов в регистрах `dx` и `ax`.

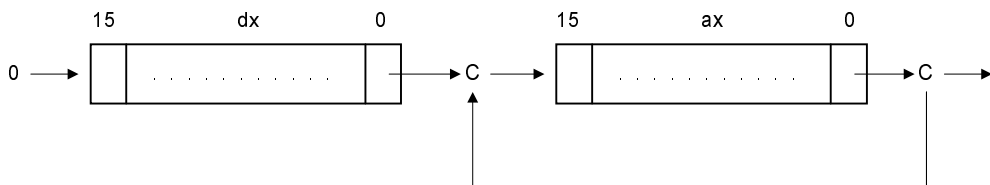


Рис. 11.1. Сдвиг двойного слова

Первая команда "выталкивает" младший разряд `dx` во флаг `c`, вторая — вставляет полученное значение `c` в старший разряд `ax` и, в свою очередь, выталкивает младший разряд `ax` в `c`. Третья команда — аналогично второй. С заменой `shr` на `rcr` фрагмент становится более однородным:

```
clc                               ; !
rcr     dx, 1
rcr     ax, 1
rcr     bx, 1
```

Запрограммируйте оба варианта реализации — с `shr` вначале, и без `shr`. Начните с более простого второго варианта.

Для проверки сравните результаты выполнения команды

```
shr      ax, 2
```

и макрокоманды

```
_shr     ah, al, 2
```

Разработайте аналогичную макрокоманду для сдвига влево — `_shl`. Порядок записи параметров — такой же, как в `_shr`. Последовательность обработки — обратная, поскольку сдвиг влево начинается с наименее значимой составляющей.

11.5.3. Функции от параметров

Разработайте макрокоманду `_page1`, которая определяет данные для оформления фрагмента оглавления. Например:

```
_page1   Preface, Acknowledgements, Chapter 1, Chapter 2, Index
```

Результат трансляции (com-файл) содержит следующие данные:

```
1. Preface .....
2. Acknowledgements .....
3. Chapter 1 .....
4. Chapter 2 .....
5. Index .....
```

Для каждого параметра следует задать генерирование одной директивы `db`. Для задания порядкового номера используйте функцию `#n`, для задания точек справа — оператор `dup`. Счетчик повторов для `dup` вычисляйте с использованием функции `#s`.

11.5.4. Вложенные циклы

Разработайте макрокоманду `_page2`, которая в результате вызова

```
_page2   Preface, Acknowledgements, Chapter 1, Chapter 2, Index
```

генерирует com-файл следующего содержания:

```
P r e f a c e .....
A c k n o w l e d g e m e n t s .....
C h a p t e r   1 .....
C h a p t e r   2 .....
I n d e x .....
```

Разрядка пробелами достигается за счет использования `c`-цикла для каждого из параметров. Внешний цикл запрограммируйте в виде `r`-цикла по всем параметрам — от `#1` до `#l`, внутрь него поместите `c`-цикл.

Разработайте макрокоманду `matrix`, которая задает двумерный массив следующей структуры:

```
a(1)      0      0      ..      0
0      a(2)    0      ..      0
0      0      a(3)    ..      0
..      ..      ..      ..      ..
0      0      0      0      a(n)
```

Значения `a(1)–a(n)` задаются в параметрах вызова макрокоманды; размерность матрицы определяется числом параметров.

11.5.5. Условная трансляция в макроопределениях

В листинге 11.8 определение макрокоманды `_push` выполнено так, что обращение к ней с пустым параметром в середине списка параметров приведет к синтаксической ошибке. Например, при вызове

```
_push ax,,1
```

создается последовательность операторов:

```
push ax
push          ; ?!
push 1
```

Второй оператор ошибочный. Введите в макрокоманду из листинга 11.8 следующее дополнение: при пустом параметре вместо команды `push` генерируется `sub sp, 2`. Включите в макроопределение директиву условной трансляции. Условие постройте на основе функции `#s` — параметр `#x` пустой, если число литер в нем (`#sx`) равно нулю.

Разработайте макрокоманду `even4`, которая генерирует инструкцию `nop` (байт со значением 090) столько раз, сколько необходимо для того, чтобы счетчик адресов `$` стал кратным 4. Используйте операцию `mod` и бесконечный `r`-цикл с оператором `#ex` внутри области условной трансляции.

Для проверки выполните трансляцию следующей программы:

```
sz_1:  cbw
        even4                      ; 3 nop's
sz_2:  mov     al, ah
        even4                      ; 2 nop's
sz_3:  mov     ax, 0
        even4                      ; 1 nop's
sz_0:
        even4                      ; 0 nop's
```

Первый вызов должен порождать три инструкции `por` (значение счетчика адресов в начале программы = 0100, команда `cbw` — однобайтная), второй — пару инструкций `por` (поскольку `mov al, ah` — двухбайтная), третий — одну (поскольку команда `mov ax, 0` — трехбайтная), последний вызов — ни одной.

11.6. Вызов макрокоманд в d86

Отладчик d86 при наличии `sym`-файла позволяет "вручную" вызывать макрокоманды, определенные в отлаживаемой программе.

Обращение к макрокоманде в режиме непосредственного выполнения приводит к немедленной передаче управления на первый из операторов подстановки. Поэтому макрокоманды с определением данных вызывать напрямую не следует — только в режиме "заплат" (Patch).

В любом случае, макрокоманда, вызываемая из d86, не должна включать в себя директив условной трансляции — в любых вариантах. Их выполнение в d86 обычно приводит к зависанию. Напротив, `#`-циклы в среде d86 работают устойчиво.

Вызовы макрокоманд с параметрами и с `#`-функциями в теле макроопределений допускаются, но синтаксические ошибки при неправильном задании параметров обычно приводят к зависанию отладчика.

Выполните в непосредственном режиме вызов следующей макрокоманды (см. `try.8`):

```
putch macro
    mov ah, 2

#cx1

    mov dl, '#x'
    int 021

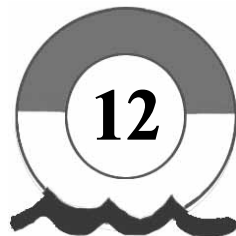
#em
```

Для этого выполните трансляцию `try.8` (содержит только макроопределение), затем запустите отладчик с `try.com` (`com`-программа создана, хотя размер ее нулевой) и выполните вызов:

```
putch abcd
```

Действие макроподстановки в режиме "заплат" (Patch) проверьте на примере макрокоманды `def_msg` из листинга 11.5. Выполните трансляцию файла `ex3.8`; зайдя в отладчик с `ex3.com`, нажмите клавишу <F7>. Красный курсор перешел из поля адреса в поле инструкции — режим "заплат" (Patch) включен. Введите вызов, например, `def_msg 7`, затем завершите режим "заплат" (Patch) повторным нажатием клавиши <F7>.

ГЛАВА 12



Структурный ассемблер

Любовь пройдет. Обманет страсть. Но лишена обмана
Волшебная структура таракана.
О, тараканьи растопыренные ножки, которых шесть!
Они о чем-то говорят, они по воздуху каракулями пишут,
Их очертания полны значенья тайного...
Да, в таракане что-то есть,
Когда он лапкой двигает и усиком колышет.

Н. М. Олейников. Из "Служения науке", 1932

В этой главе рассмотрено применение логических блоков из языков высокого уровня в программах, написанных на языке ассемблера.

12.1. Логика условных переходов

На машинном уровне логика выполнения программы определяется инструкциями, действие которых зависит от флагов состояния, — в первую очередь, инструкциями условных переходов `j<x>`. При использовании этих инструкций часто приходится мысленно преобразовывать условие выполнения в условие обхода (т. е. в условие "невыполнения").

Листинг 12.1. Иллюстрация инверсии условий (см. `count_1.8`)

```
...
mov      ah, 0          ; ah = 0;
11:      ; for (; --cx;) {
lods     ; al = *si++;
xlat     ; al = map[al];
test     al             ; if (al != 0)
jz       >l2
inc      ah             ; ah++;
12:     loop  l1          ; }
...
```

В фрагменте, приведенном в листинге 12.1, подсчитывается число байт массива со значениями из множества, заданного таблицей по адресу `bx`. В регистре `si` задан адрес первого байта (при выполнении цикла `si` указывает на очередной байт), длина массива — в `cx`. Результат счета формируется в `ah`.

Если байту, прочитанному из `[si]`, соответствует 1 в таблице `[bx]`, счетчик `ah` следует увеличить. Инструкция условного перехода `jz` задает *обход* инструкции `inc` при *невыполнении* условия `nz`. Таким образом, вместо того, чтобы задавать условие *выполнения* (как в операторе `if` языков C или Pascal), приходится кодировать инструкцию обхода с инвертированным условием.

Листинг 12.1 демонстрирует первый недостаток логики условных переходов — необходимость мысленно инвертировать условия при кодировании и чтении программы.

Следующий недостаток, связанный с применением команд переходов, более существенный. Для его иллюстрации рассмотрим структуру переходов в одной из процедур стандартной библиотеки транслятора с языка Pascal для ЭВМ PDP-11, на ассемблере MACRO-11.

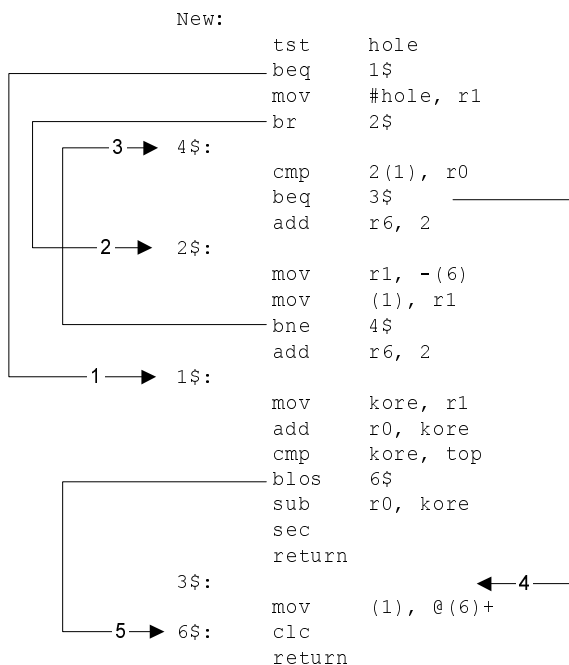


Рис. 12.1. Пример пересечения переходов

Количество всевозможных путей выполнения, или *кодекс*, определяет логическую сложность программы. Кодекс зависит от числа пересечений пере-

ходов. В примере на рис. 12.1 шесть пересечений: 1–3, 2–3, 1–4, 2–4, 3–4, 4–5 (в пределе — число сочетаний из 5 по 2). Все пути от `new` до `return` были протестированы в свое время, но развивать эту запутанную процедуру в ее первоначальном виде было бы трудно. Распутать ее тоже непросто — пришлось бы дублировать ряд фрагментов, инвертируя условия в ветвлениях и добавляя безусловные переходы.

Как ограничить логическую сложность программы при разработке? Ответ на этот вопрос дан в теории структурного программирования и подтвержден практикой применения языков высокого уровня: вместо переходов и ветвлений следует использовать вложенные последовательности логических блоков с одним входом и одним выходом.

Ассемблер, дополненный операторами условного выполнения (`if-else`) и циклами (`do-while`, `repeat-until` и пр.), называется *структурным ассемблером*.

В качестве иллюстрации применения логических блоков в ассемблере рассмотрим оператор `if` в `a86`. Несмотря на наличие оператора условного выполнения `if`, ассемблер `a86` не структурный, т. к. в `if` не предусмотрен `else`-вариант и отсутствуют конструкции для задания циклов `while` или `repeat`.

12.2. Оператор *if* в `a86`

В ассемблер `a86` включен оператор `if` с довольно ограниченными возможностями: после `if <x>` допускается задание только *одного* оператора — в той же строке.

Листинг 12.2. Программа из листинга 1 с использованием `if` (см. `count_2.8`)

```

...
mov     ah, 0

11:
    lodsb
    xlat
    test  al
    if nz inc     ah      ;      jz      <adr>
                          ;      inc     ah
    loop  11           ; <adr>:
...

```

В комментариях к `if` показан результат трансляции — инструкция перехода по инверсному условию, в обход оператора, заданного в теле `if`.

12.3. Способы реализации структурного ассемблера

Оператор `if` встроен в ассемблер `a86`, его обработка не требует дополнительного просмотра при трансляции. Реализация полноценного структурного ассемблера, как правило, предполагает отдельное *предварительное* преобразование операторов логики в метки и команды переходов. Преобразование может быть выполнено:

- макроассемблером;
- отдельной программой-препроцессором.

12.3.1. Реализация при помощи макроассемблера

Смысл этого варианта в том, что операторы логических блоков задаются в виде *макрокоманд*, а преобразование выполняется макроассемблером. Необычно то, что все составляющие логических блоков — не только `if`, `while`, `repeat`, но и `else`, `end` — задаются в виде обособленных макроопределений. В результате, при подстановках, отмечающих середину или конец логического блока, т. е. при вызове `else` или `end`, в макроопределениях приходится учитывать предшествующие вызовы макрокоманд, задающих начало блока (`if`, `while`, `repeat`). Условие для оператора `if` или `while` задается в виде параметра.

Макрокоманды позволяют реализовать не только условное выполнение и циклы, но и оператор мультиветвления (то, что в языке `C` называется `switch`, в `Pascal` — `case`, в `Basic` — `on goto`), с возможностью задания вариантов в виде *выражений базового ассемблера*, в частности, с пользовательскими именами.

Возможности преобразования ограничены языком макрокоманд. Кроме того, в обозначения логических операторов приходится вводить дополнительные литеры (например, подчеркивание) — во избежание конфликтов с ключевыми словами базового ассемблера.

Пример макроопределений для операторов языка структурного ассемблера приведен в архиве `struca86.zip`.

12.3.2. Реализация при помощи препроцессора

Трансляция логических блоков при помощи отдельной программы дает больше свободы в выборе синтаксиса структурного ассемблера, т. к. обработка логических операторов выполняется *независимо* от ассемблера. Кроме того, сам транслятор может быть запрограммирован с использованием любых средств — от языков высокого уровня до специальных инструментов

для разработки лексических и синтаксических анализаторов (lex и yacc, например).

Вместе с тем, независимость от ассемблера не позволяет реализовать полноценный оператор мультиветвления — для вычисления препроцессором вариантов, заданных выражениями ассемблера, тем более при наличии в них пользовательских имен, потребовалось бы в точности воспроизвести функции ассемблера. В результате, имеющиеся реализации препроцессоров структурного ассемблера мультиветвление не поддерживают.

Далее рассматривается смешанный вариант реализации структурного ассемблера — bsp86, разработанный авторами этой книги. В языке bsp86 реализованы все логические блоки с удобным синтаксисом и подробной диагностикой. Большая часть блоков логики полностью обрабатывается препроцессором; мультиветвление транслируется препроцессором в *вызовы макрокоманд*, которые обрабатываются макроассемблером.

12.4. Структурный ассемблер bsp86

Прототипы bsp86 — структурный ассемблер BSP (Block Structure Processor) фирмы Connell Scientific Graphics и структурный ассемблер SALUT (Structure Assembly Language UTility) фирмы IBM; синтаксис заимствован из языка высокого уровня Modula-2.

Программа на языке bsp86 содержит операторы логики и операторы ассемблера. Препроцессор bsp86 выявляет логические операторы и записывает в выходной файл соответствующие метки и инструкции переходов на языке a86. Неизвестные ему операторы, в частности операторы a86, копируются в выходной файл без изменений.

12.4.1. Синтаксис логических операторов

В языке bsp86 предусмотрены следующие логические операторы:

- ☐ циклы: DO-END, WHILE-END, FOR-END, REPEAT-UNTIL;
- ☐ выход из цикла — EXIT;
- ☐ условное выполнение — IF-ELIF-ELSE-END;
- ☐ последовательность операторов;
- ☐ оператор мультиветвления — CASE.

По сравнению с прототипами, в bsp86:

- ☐ расширен набор циклов;
- ☐ в операторе условного выполнения введен вариант ELIF;
- ☐ реализован оператор мультиветвления.

Оператор цикла с постусловием

Цикл с постусловием определяется в форме:

```
REPEAT
    <seq>
UNTIL <x>
```

Условие выхода <x> проверяется в конце цикла, после каждого выполнения последовательности операторов <seq>. Если <x> выполнено, цикл завершается, а если не выполнено, то продолжается. Поскольку условие проверяется *после* выполнения тела цикла, то <seq> обрабатывается один раз, как минимум.

Ниже приведен пример оператора REPEAT-UNTIL (см. repeat.b8); справа показан результат преобразования:

```

                                REPEAT_1__:
REPEAT                          rcl      al, 1
    rcl      al, 1              jnc      REPEAT_1__
UNTIL c                          UNTIL_1__:
```

Условие завершения *c* транслируется в ветвление по обратному условию — *nc*. Цифры в названиях меток REPEAT_1__ и UNTIL_1__ означают порядковый номер логического блока в программе.

Оператор цикла с предусловием

Форма цикла с предусловием следующая:

```
WHILE <x>
    <seq>
END
```

Условие выполнения — <x> — проверяется в начале цикла. Если <x> истина, то обрабатывается последовательность <seq>, после чего возобновляется проверка <x>. Если условие <x> не выполнено, цикл завершается. Поскольку проверка условия поставлена *перед* телом цикла, может оказаться, что последовательность <seq> не будет выполнена ни разу.

Пример оператора WHILE-END (см. while.b8) и результат преобразования:

```

                                WHILE_1__:
WHILE nc DO                     jc      END_1__
    rcl      al, 1              rcl      al, 1
END                              jmp     WHILE_1__
                                END_1__:
```

Условие выхода *nc* транслируется в ветвление по обратному условию — *c*. Результат по сравнению с циклом REPEAT-UNTIL более громоздкий: к ветвлению добавился также безусловный переход.

Оператор счетного цикла

Счетный цикл задается в одной из следующих форм:

```
FOR cx          FOR cx
  <seq>         <seq>
END            END <x>
```

Счетный цикл введен для того, чтобы выразить в структурах логики *инструкции циклов* — `loop/loop<x>`.

При трансляции инструкция цикла ставится после `<seq>`. Если в операторе счетного цикла задано дополнительное условие выхода `<x>`, транслятор выбирает инструкцию `loop<nx>`, а иначе — просто `loop`. (Напомним, что допустимые значения `<x>` для инструкций цикла — `z`, `nz`, `e`, `ne`.) Регистр `cx` — управляющая переменная цикла, другие регистры не допускаются. Задание `cx` обязательно (для наглядности).

Примеры оператора `FOR-END` (см. `for_1.b8`) и результат преобразования:

```
FOR cx          FOR_1__:
  ...          ...
END            loop    FOR_1__
              END_1__:

FOR cx          FOR_2__:
  ...          ...
  cmp    dh, 13      cmp    dh, 13
END e          loopne  FOR_2__
              END_2__:
```

Условие `<x>`, записанное справа от `END`, — это дополнительное условие выхода из цикла. Цикл завершается, если `(--cx) == 0` или `<x>` — истина.

Оператор бесконечного цикла

Бесконечный цикл задается в форме:

```
DO
  <seq>
END
```

Результат преобразования:

```
DO_1__:
  <seq>
  jmp    DO_1__
END_1__:
```

Выход из цикла

Оператор выхода из цикла задается в одной из форм:

```
EXIT                EXIT <x>
```

Оператор выхода может быть задан в теле любого из рассмотренных выше циклов; вне цикла он недопустим. Оператор выхода преобразуется в инструкцию перехода на команду, следующую за циклом. Если условие выхода <x> не задано, то переход безусловный (jmp), а если задано, то условный (j<x>).

Пример выхода из бесконечного цикла (см. for_exit.b8) и результат преобразования:

```
FOR cx                FOR_1__:
...                   ...
    cmp     dl, 13      cmp     dl, 13
EXIT e                je      END_1__
...                   ...
END                   loop    FOR_1__
                        END_1__:
```

Оператор условного выполнения

Оператор условного выполнения задается в одной из следующих форм:

```
IF <x1>               IF <x1>
    <seq1>             <seq1>
END                   ELSE
                     <else_seq>
                     END

IF <x1>               IF <x1>
    <seq1>             <seq1>
ELSIF <x2>            ELSIF <x2>
    <seq2>             <seq2>
...                   ...
ELSIF <xk>            ELSIF <xk>
    <seqk>             <seqk>
END                   ELSE
                     <else_seq>
                     END
```

Во всех вариантах условие <xi> — это условие выполнения последовательности <seqi>. В начале выполнения проверяется условие <x1>. Если <x1> истина, то управление передается на <seq1> и после отработки этой последовательности — на выход. Если <x1> не выполнено, то <seq1> пропускается (обход инструкцией условного перехода j<nx1>) и проверяется условие

<x2>. Если ни одно из условий не выполнено, управление передается на последовательность <else_seq> или на выход (если вариант ELSE не задан).

Пример оператора IF-ELSIF-ELSE-END (см. if_1.b8) и результат преобразования:

	IF_1__:
IF a	jna ELSE_1_1__
sub al, 3	sub al, 3
ELSIF b	jmp END_1__
add al, 7	ELSE_1_1__:
ELSE	jnb ELSE_1_2__
xor al, al	add al, 7
END	jmp END_1__
	ELSE_1_2__:
	xor al, al
	END_1__:

Условие <x1> (a) транслируется в ветвление по обратному условию (na). Переход jna задает обход последовательности операторов <seq1> между IF и ELSIF. Если условие выполнено, то управление передается на <seq1>, после чего — командой jmp — на выход. Если <x1> не выполнено, управление передается на проверку, заданную ELSIF.

Обработка ELSIF выполняется по аналогии с IF. Управление передается на ELSE, если оба условия <x1> и <x2> не выполнены. После <else_seq> инструкция jmp не нужна, т. к. <else_seq> находится в конце логического блока.

Вторая цифра в метках ELSE_1_1__ и ELSE_1_2__ означает порядковый номер варианта IF-ELSIF внутри текущего оператора условного выполнения.

Последовательность операторов

Последовательность операторов в bsp86 состоит из операторов bsp86 и операторов базового ассемблера.

Вся программа представляет собой последовательность операторов. Эта общая последовательность, в свою очередь, включает в себя вложенные последовательности в телах логических блоков, т. е. в <seq>. В этих вложенных последовательностях также могут содержаться логические блоки bsp86. (Единственное исключение, связанное с особенностями реализации bsp86: внутри мультиветвления задание других мультиветвлений не допускается.)

Пример вложенных логических блоков (см. do_exit.b8):

DO	DO_1__:
rcl al, 1	rcl al, 1
IF c THEN	IF_2__:
rcr al, 1	jnc ELSE_2_1__
EXIT	rcr al, 1
END	jmp END_1__

```

END                                ELSE_2_1__:
                                END_2__:
                                jmp      DO_1__
                                END_1__:

```

Оператор условного выполнения задан *внутри последовательности* операторов, образующих тело бесконечного цикла. В свою очередь, оператор выхода из цикла задан внутри блока условного выполнения.

Оператор мультиветвления

Формы задания оператора мультиветвления следующие:

CASE <reg>	CASE <reg>
<var_list_1>	<var_list_1>
<seq_1>	<seq_1>
...	...
<var_list_k>	<var_list_k>
<seq_k>	<seq_k>
END	ELSE
	<else_seq>
	END

Выбор из множества <seq_i>/<else_seq> определяется значением *селектора* — регистра <reg>. Регистр-селектор должен допускать косвенную адресацию; это регистр из набора bx, bp, si, di.

В каждом списке вариантов <var_list_i> задано, в общем случае, несколько значений селектора, для которых выполняется <seq_i>. Значения ограничены диапазоном беззнакового байта 0—255 и должны быть уникальными в пределах текущего мультиветвления (т. к. выбирается только одна последовательность).

Если входной величине селектора <reg> соответствует значение из списка <var_list_j>, выполняется <seq_j>. Если входное значение селектора в списках отсутствует, выполняется <else_seq>, если ELSE-вариант задан (если не задан, то не выполняется ничего — управление сразу передается на выход). После выполнения выбранной последовательности операторов управление автоматически передается на выход, как в операторе case языка Pascal (напротив, в операторе switch языка C выход из мультиветвления следует задавать явно — оператором break, иначе управление будет передано на следующую последовательность).

Примечание

Между началом оператора и передачей управления на выбранную последовательность значения флагов процессора и регистра <reg> изменяются!

Пример мультиветвления на языке bsp86 (см. case.b8):

```
CASE bx OF
| 3, 1 shl 1
        mov     ax, 1
| 5
        mov     ax, 2
ELSE
        xor     ax, ax
END
```

Первый список вариантов содержит значения 3 и 2 (1 shl 1), заданные в виде выражений а86. Если селектор `bx` равен трем или двум, управление будет передано на инструкцию `mov ax, 1`, а после ее выполнения — на выход. Если входное значение `bx` равно пяти, управление передается на инструкцию `mov ax, 2`, а после нее — на выход. Если же входное значение `bx` не 5, не 3 и не 2, то выполняется переход на инструкцию `xor ax, ax`.

Примечание

При входе в выбранную последовательность значение `bx` отличается от исходного. То есть, `bx` при передаче управления на `mov ax, 2` уже не равно пяти, и т. д. Флаги тоже изменяются!

Чтобы ориентироваться в машинных инструкциях мультиветвления (что может понадобиться при отладке), рассмотрим принцип реализации этого логического блока в bsp86.

12.5. Реализация мультиветвления в bsp86

Мультиветвление основано на косвенном переходе по адресу из таблицы. Значение селектора `<reg>` используется в качестве индекса для доступа к таблице адресов.

В первом приближении, таблица адресов перехода содержит 256 записей — для всевозможных значений селектора в диапазоне 0—255. Для мультиветвления, продемонстрированного в *разд. 12.4.1*, элементы таблицы с индексами 2 и 3 содержат адрес `<seq_1>`, элемент с индексом 5 — адрес `<seq_2>`, а элементы с индексами 0, 1, 4, 6—255 — адрес `<else_seq>`. Реализация представлена в листинге 12.3.

Листинг 12.3. Реализация мультиветвления из *разд. 12.4.1* в первом приближении

```
shl     bx, 1                ; (1)
mov     bx, w[map+bx]        ; (2)
jmp     bx                   ; (3)
```

```

map      dw      2 dup else_seq
          dw      2 dup seq_1
          dw      else_seq
          dw      seq_2
          dw      255-6 dup else_seq

seq_1:
    mov     ax, 1
    jmp     end__

seq_2:
    mov     ax, 2
    jmp     end__

else_seq:
    xor     ax, ax

end__:

```

Значение индекса в `bx` преобразуется (1) в смещение от начала таблицы `map`. Из таблицы в `bx` считывается (2) значение адреса, по нему выполняется косвенный переход (3).

Если предварительно анализировать значение `<reg>`, таблица адресов значительно сокращается. В примере из *разд. 12.4.1* значения селектора для выбора одной из последовательностей `<seq_i>` находятся в пределах 2—5; вне этого диапазона выбирается `<else_seq>`. Выход значения `<reg>` за пределы диапазона 2—5 легко проверить и без таблицы, что позволяет сократить ее до четырех элементов, как показано в листинге 12.4.

Листинг 12.4. Компактная реализация мультиветвления из *разд. 12.4.1*

```

    sub     bx, 2                ; (1)
    cmp     bx, 3                ; (2)
    ja      else_seq            ; (3)
    ...

map      dw      2 dup seq_1
          dw      else_seq
          dw      seq_2

```

В начале (1) из `<reg>` вычитается 2 — нижняя граница диапазона 2—5. В результате, диапазоны 0—1, 2—5, 6—255 отображаются в диапазоны, соответственно, -2— -1 (0ffe—0fff), 0—3, 4—253. Если после вычитания (1) беззнаковая величина в `<reg>` больше трех (4—253 и 0ffe—0fff), это означает выбор `else_seq`, что выполняется командами (2) и (3).

В дополнение к сокращению таблицы, имеется возможность более компактно запрограммировать инструкции для организации косвенного перехода:

```
shl    bx, 1
jmp    w[map+bx]    ; !
```

Чтение адреса из таблицы и переход объединены в одной инструкции `jmp`.

Реализация мультиветвления в `bsp86` несколько отличается от рассмотренного идеального варианта. В листинге 12.5 приведен результат трансляции примера из *разд. 12.4.1* при помощи `bsp86`.

Листинг 12.5. Реализация мультиветвления из *разд. 12.4.1* средствами `bsp86`

```
jmp    CASE_1__                ; (1)

include bsp86.skl              ; (2)

CASE_1_1__:
    mov    ax, 1                ; (3)
jmp    END_1__
CASE_1_2__:
    mov    ax, 2                ; (4)
jmp    END_1__
ELSE_1__:
    xor     ax, ax               ; (5)
jmp    END_1__                 ; (6)

CASE_1__:
    ; (7)
__max_min__    1 shl 2, bit 1,1  ; (8)
__switch__     bx,ELSE_1__       ; (9)
__def_adr__    CASE_1_1__,1 shl 2, bit 1 ; (10)
__def_adr__    CASE_1_2__,1      ; (11)
__adr_map__    ; (12)

END_1__:
```

Обратите внимание, что транслятор не генерирует ни инструкций разветвления, ни таблицы адресов! Вместо этого он формирует *параметры* для вызова *макрокоманд*, определенных в файле `bsp86.skl` (подключается (2) директивой `include`).

Макрокоманда `__max_min__` (8) предназначена для вычисления максимума и минимума из значений фактических параметров. Результаты сохраняются в константах и используются в дальнейшем макрокомандой `__switch__` (9), которая генерирует следующие инструкции:

- ☐ `sub` для заданного регистра-селектора;
- ☐ условный переход `ja` по заданному адресу;
- ☐ прочие команды разветвления, не требующие настройки.

В фактических параметрах `__max_min__` перечислены все списки вариантов через запятую. Параметр `__def_adr__` — отдельный список: в первом вызове (10) — это `var_list_1`, во втором (11) — `var_list_2`. По результатам перечисленных макрокоманд, последняя макрокоманда `__adr_map__` (12) создает таблицу адресов.

Для чего понадобились макрокоманды? Для того, чтобы в списках вариантов можно было использовать выражения а86. Транслятор `bsp86` не пытается дублировать функции а86; вместо этого он собирает списки, заданные в исходном тексте после разделителей "|", и передает их на обработку а86 через макровыводы.

Накапливая в памяти списки вариантов, транслятор параллельно обрабатывает последовательности, заданные в вариантах. Для каждой последовательности он создает метку входа, транслирует последовательность (в которой, в общем случае, содержатся вложенные логические блоки), генерирует инструкцию для выхода. В результате, информация для оформления макровыводов полностью собрана лишь после завершения трансляции всех последовательностей в теле мультиветвления.

Поэтому команды для организации разветвления и таблица адресов записываются в конце. В итоге, порядок расположения частей мультиветвления не совпадает с идеальным вариантом из листинга 12.4. Но каким бы ни был порядок расположения частей, выполнение следует начинать с команд разветвления. Поэтому добавлен безусловный переход в начале (1), и еще один (6) — после `<else_seq>`.

12.6. Выполнение примеров

Вызовите `bsp86.exe`. В ответ будет выведена информация о способе вызова (следует задать входной и выходной файлы) и возможных опциях. Опция включена, если ей предшествует "+", и выключена, если "-". Справа от каждой опции показано значение по умолчанию (плюс или минус в скобках).

Опция `u`, если включена, настраивает `bsp86` на литеры верхнего регистра. То есть, при задании `+u` в командной строке, идентификатор `IF` распознается как ключевое слово (начало блока условного выполнения), а `if` копируется в выходной файл без анализа, как неизвестное слово.

Если опция `u` выключена (`-u`), `bsp86` настроен на поиск ключевых слов в нижнем регистре.

В любом случае, `bsp86` не преобразует имена к верхнему регистру, в отличие от большинства ассемблеров (но по аналогии с языками высокого уровня).

Примечание

Опции `a`, `b` управляют добавлением двойного подчеркивания к именам меток.

Выполните трансляцию примера из файла `while.b8`:

```
bsp86 while.b8 while.8
```

Получите исполняемый файл:

```
a86 while.8
```

Выполните примеры из *разд. 12.4.1*, кроме `case.b8`. В примерах `while.b8`, `repeat.b8`, `do_exit.b8` действие одно и то же — сдвиг `al` до тех пор, пока в старшем бите не окажется 1. В `for_1.b8` и `for_exit.b8` задан посимвольный вывод зашифрованного сообщения. В `for_1.b8` первый цикл выводит сообщение целиком, а второй цикл прекращает вывод, когда выведен символ возврата каретки `cr`. В примере `for_exit.b8` единственный цикл завершается перед тем, как выведен `cr`, за счет использования оператора `EXIT` в теле цикла.

Шифрация для `for_1.b8`, `for_exit.b8` выполнена по схеме:

$$a(i) \leftarrow b(i) \text{ xor } b(i-1),$$

где $b(i = 1-n)$ — литеры исходного сообщения, $a(i = 1-n)$ — литеры зашифрованного сообщения; $b(0) = 0$. Расшифровка выполняется по этой же схеме.

Выполните трансляцию `case.b8`; при вызове `a86` закажите листинг:

```
a86 case.8 +L
```

Из листинга исключены условные операторы, а также операции, не представляющие интереса при отладке (например, обнуление рабочих имен `o1-o256` в цикле) — за счет директив `.NOLIST` (блокировка листинга) и `.LIST` (возобновление листинга). В итоге, в листинг попадает лишь то, что вошло в исполняемый файл:

1 0100 E9 11 00	jmp CASE_1__
100	CASE_1_1__:
101 0103 B8 01 00	mov ax, 1
102 0106 E9 22 00	jmp END_1__
103	CASE_1_2__:
104 0109 B8 02 00	mov ax, 2
105 010C E9 1C 00	jmp END_1__
106	ELSE_1__:
107 010F 33 C0	xor ax, ax
108 0111 E9 17 00	jmp END_1__
109	CASE_1__:
147 0114 83 EB 02	m sub bx, __min__
150 0117 83 FB 03	m cmp bx, __max__ - __min__
151 011A 77 F3	m ja __no_case__
152 011C D1 E3	m shl bx, 1
153 011E 2E FF A7 23 01	m cs jmp w >o256 [bx]

```

154                                m      o256 dw
969 0123   03 01                  m      dw o2
983 0125   03 01                  m      dw o3
1001 0127   0F 01                  m      dw __no_case__
1011 0129   09 01                  m      dw o5
1020                                END_1__:
```

Определения имен `__no_case__`, `o2`, `o3` и `o5` в `lst`-файле не показаны. Имени `__no_case__` соответствует метка `CASE_1__` или метка `END_1__`. Наличие имени `o<j>` в таблице адресов следует понимать так, что для значения селектора `j` будет выбрана какая-то из последовательностей `<seq>`.

Проверьте в отладчике выполнение мультиветвления из файла `case.b8` с разными значениями `bx` на входе.

12.7. Диагностические сообщения bsp86

Препроцессор `bsp86` выдает сообщения об ошибках двух типов. Сообщения первого типа выводятся при нарушении структуры логических блоков, второго типа — при лексических ошибках, исчерпании ресурсов и прочих нарушениях, не связанных с логикой. Поскольку препроцессор не контролирует ни условия, ни списки вариантов, ни тем более операторы ассемблера, то ошибки, допущенные в этих объектах, обнаруживает `a86`.

12.7.1. Сообщения первого типа

Удалите `END` в примере `while.b8`. При трансляции `bsp86` выдаст сообщение:

```

Line 8 (level 1) : syntax error 'int'
Expected : END
```

В сообщении указан номер строки `Line` и текущий уровень вложенности логических блоков `level` (отсчитывается от нуля). В этих сообщениях имеется предложение `Expected`, где перечислены ожидаемые в данном контексте ключевые слова.

После `syntax error` выведено слово, с которым связана ошибка. В примере идентификатор `'int'` — косвенная причина ошибки. Препроцессор обычно пропускает незнакомые слова, в частности мнемоники инструкций `a86`. Настоящая причина ошибки состоит в том, что файл закончен до завершения конструкции `WHILE-END`. Идентификатор `'int'` "виноват" лишь в том, что на его месте должно было быть ключевое слово `END`!

В ситуациях, не связанных с неожиданным завершением ввода, `syntax error` всегда показывает ключевое слово `bsp86`. Для проверки составьте программу из единственного ключевого слова `ELSE`. При трансляции будет выдано сообщение:

```
Line 1 (level 0) : syntax error 'ELSE'  
Expected : REPEAT EXIT IF WHILE FOR CASE DO
```

Перед тем, как рассмотреть вторую группу ошибок, остановимся на особенностях лексического анализатора bsp86.

12.7.2. Особенности лексического анализатора

Лексический анализатор выбирает первое слово в начале строки исходного текста, считая разделителем пробел, табуляцию и перевод строки. Если первое слово ключевое, то — без проверок — выбираются первые два слова и, отдельно, сохраняется остаток строки после второго слова. Второе слово используется при разборе операторов IF, ELSIF, UNTIL, WHILE, END (FOR-END), FOR. Остаток строки при разборе вариантов мультиветвления содержит список вариантов после '|'.

Комментарии ассемблера — ';' — не выявляются. Поэтому в тех строках, где заданы ключевые слова bsp86, комментарии ставить не следует. Они приведут к ошибкам при ассемблировании, если заданы в строке с вариантами мультиветвления.

Правильность условий в блоках логики при трансляции не проверяется. Второе слово строки исходного текста дополняется слева буквой 'j' — получается инструкция условного перехода. При этом данное условие может оказаться пустым или неправильным с точки зрения a86; ошибка будет обнаружена при ассемблировании. Проверьте — в примере if_1.b8 замените условие a на x и попробуйте получить исполняемый файл.

Условия инвертируются bps86 за счет добавления или удаления начальной литеры 'n'. Поэтому вместо допустимой в a86 пары условий pe/po (parity even/odd) следует пользоваться синонимом p/np (parity/not parity).

Условие ncxz, для которого нет соответствующего ветвления jncxz, при инверсии дает корректную инструкцию jcxz. Напротив, из условия cxz может получиться недопустимая инструкция jncxz. Для проверки выполните трансляцию if_1.b8, поменяв условие в IF на cxz, а условие в ELSIF — на ncxz.

12.7.3. Сообщения второго типа

Диагностические сообщения второго типа возникают при нарушении лексики или при исчерпании ресурсов; так или иначе, они не имеют отношения к структуре логических блоков. Формат первой строки сообщения — такой же, как для первого типа ошибок; вторая строка содержит текст с вопросом в начале:

- ❑ Bad chars after COMMENT — слово после COMMENT, представляющее ограничитель для последующего закрытия комментария, состоит более чем из одной буквы. Лексический анализатор bsp86 распознает директиву COMMENT в стиле a86 (см. файл if_1.b8).

- ❑ 'FOR' requires cx — после FOR не задан cx.
- ❑ EXIT without loop — оператор выхода из цикла задан вне тела цикла.
- ❑ Nested CASE not allowed — вложенное мультиветвление недопустимо. Макроопределения для генерирования мультиветвлений не предусматривают рекурсию, поэтому CASE внутри другого CASE реализовать невозможно.
- ❑ More than 64 '|' in CASE — больше 64 списков вариантов в конструкции мультиветвления.
- ❑ Nested COMMENT not allowed — вложенные комментарии недопустимы.

12.7.4. Ошибки в списках вариантов

Препроцессор не анализирует содержание списков вариантов мультиветвления. Его задача заключается лишь в том, чтобы запомнить тексты списков вариантов и — после завершения разбора конструкции мультиветвления — составить из этих текстов списки фактических параметров макрокоманд из bsp86.skl.

Диагностика в составе макрокоманд выявляет следующие семантические ошибки:

- ❑ дублирование значений вариантов;
- ❑ вариант со значением больше 255;
- ❑ пустой список вариантов после '|'.

В программе case.b8 ко второму списку вариантов добавьте вариант 2+1, через запятую. При трансляции case.8 ассемблер a86 выдаст сообщение:

```
__def_adr__      CASE_1_2__,5, 2 + 1
~v
?-Value <3> duplicated (see variant <2 + 1>)
#ERROR 37: Misplaced Built-In Symbol
```

Восстановите case.b8 и добавьте к первому списку вариант bit 8. Сообщение a86 об ошибке следующее:

```
__max_min__      3, 1 shl 1, bit 8,5
~v
?-Varaints not in range 0..255
#ERROR 37: Misplaced Built-In Symbol
```

Удалите первый список, оставьте только '|'. Сообщение a86 следующее:

```
__def_adr__      CASE_1_1__,
~v
?-Empty var list
#ERROR 37: Misplaced Built-In Symbol
```

Ситуация, когда между `CASE` и `ELSE-END` не задано ни одной литеры `|` (т. е. варианты отсутствуют), выявляется самым `bsp86` как нарушение структуры блока.

В вариантах не следует использовать *опережающие* ссылки. Добавьте в первый список вариантов из файла `case.b8` имя `Smith`. Ошибка обнаруживается, но проявляется довольно причудливо:

```
__max_min__      3, 1 shl 1, Smith,5
~                               ^
#ERROR 61: Overlapping Local Not Allowed
__switch__       bx,ELSE_1__
__def_adr__      CASE_1_1__,3, 1 shl 1, Smith
~                               ^
#ERROR 61: Overlapping Local Not Allowed
```

Примечание

Такое впечатление, что эти сообщения возникают в результате сбоя в выполнении функции `#v` в макрокомандах. Мы не знаем в точности, что происходит с `#v`, когда параметр этой функции содержит ссылку на неопределенное имя.

12.8. Пример использования `bsp86`

В качестве иллюстрации применения `bsp86` приведем решение известной учебной задачи — сортировки методом "пузырька".

Листинг 12.6. "Пузырьковая" сортировка на `bsp86` (см. `bsort.b8`)

```
jmp      start

area     db      094, 044, 055, 012, 042, 018, 067, 06
size     equ     $ - area

start:
mov      cx, size-1
FOR cx
  mov     dx, cx
  lea     bx, area          ; (2)
  REPEAT
    mov    ax, [bx]
    cmp    al, ah
    IF a
      xchg  al, ah          ; (1)
      mov   [bx], ax
  END
```

```
        inc     bx
        dec     dx
UNTIL z
END
int     020
```

При выполнении этой программы данные из области `area` будут упорядочены по возрастанию. Сортировка выполняется за счет обмена местами (1) соседних элементов массива.

12.9. Практикум

Выполните трансляцию программы `bsort.b8` и исследуйте ее в отладчике (в файле `bsort.d8k` подготовлены нажатия клавиш для отображения массива `area`). Если выполнять за один шаг цикл `REPEAT-UNTIL` от (2) до (2), видно, как постепенно перемещаются влево ("поднимаются") меньшие значения — "пузыри".

С использованием `bsp86` выполните задание, выбранное в *части I* книги.

Переведите программу двоичного поиска с языка `Modula-2` (см. файл `bsearch.mod`) на `bsp86`. Чтобы разобраться с программой, не обязательно знать `Modula-2`; достаточно знакомства с языками `C` и `bsp86`.

Двоичный поиск, или поиск делением пополам, применяется для обнаружения заданного значения в *отсортированном* массиве. Искомое значение сравнивается с медианой. В результате, определяется, в какой половине массива имеет смысл продолжать поиск. Затем действия повторяются, но уже не для всего массива, а для выбранной половины. Так продолжается, пока область поиска не сожмется в точку.

Сохранив логическую структуру программы, замените операторы и условия языка высокого уровня на операторы и условия `bsp86`. Переменным `Left`, `Right`, `Center` поставьте в соответствие регистры `si`, `bx`, `di`; искомое значение задайте в `al`.

Примечание

Процедура `Bin_Search` возвращает значение `TRUE` или `FALSE` типа `BOOLEAN`, которое выводится на экран библиотечной процедурой `WrBool`; функция `HIGH` возвращает верхний индекс массива, переданного в процедуру.

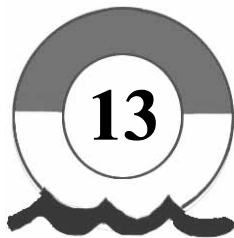
Проверьте подпрограмму в отладчике с разными значениями `al`. Обратите внимание на конечные значения индексов `Left` и `Right`.

Подпрограмма двоичного поиска позволяет не только проверить наличие заданного значения в массиве, но также определить, в какой позиции мас-

сива следует вставить элемент с заданным значением, чтобы не нарушить порядок. Взяв за основу разработанную процедуру, решите с использованием `bsp86` следующую задачу на эту тему. Имеется упорядоченный массив, требуется добавить к нему элементы из неупорядоченного массива — по одному последовательно.

Примечание

При вставке в крайних позициях массива возможны ошибки; от краевых эффектов просто избавиться, если поставить по краям массива барьеры: слева — минимально возможное значение, справа — максимально возможное.



Интерфейс с языком С

В настоящей главе подробно рассматривается совместное применение ассемблера и языка С в двух вариантах.

1. Фрагменты на языке ассемблера вставляются в исходную программу на С и транслируются совместно.
2. Модули на языке ассемблера и модули на языке С транслируются раздельно; полученные объектные модули объединяются компоновщиком в исполняемую программу.

Мы рассмотрим оба варианта в их крайних проявлениях: Microsoft/Borland-С и, с другой стороны, Watcom-С. Начнем с общих вопросов.

13.1. Машинное представление данных языка С

Рассмотрим машинное представление элементарных типов данных С: `char`, `short int`, `int`, `long int`, `float`, `double`.

Тип `char` занимает один байт. По умолчанию значение типа `char` трактуется как знаковое в диапазоне -128 — $+127$. Тип `unsigned char` — беззнаковый байт в диапазоне 0 — 255 .

Объект типа `short int`, или `short`, занимает одно слово. По умолчанию его значение знаковое, в диапазоне $-32\,768$ — $+32\,767$. Тип `unsigned short` — беззнаковый, со значением в диапазоне 0 — $65\,535$.

Типы `int` и `short int` в 16-разрядных приложениях совпадают.

Тип `long int`, или `long`, занимает двойное слово. По умолчанию значение знаковое, в диапазоне $-2\,147\,483\,648$ — $+2\,147\,483\,647$. Тип `unsigned long` представляет беззнаковые значения в диапазоне 0 — $4\,294\,967\,295$.

Типы `float` и `double` представляют действительные значения, заданные четырьмя или восемью байтами.

Примечание

Действительные значения и операции над ними подробно рассмотрены в *главе 15*.

Машинное представление `C`-указателей определяется их "дальностью". "Ближний" указатель задан словом, его значение представляет смещение в текущем сегменте данных. "Дальний" указатель — это двойное слово со значением "полного" адреса в формате сегмент:смещение.

По умолчанию дальность указателей определяется моделью памяти. В моделях с одним сегментом данных (`tiny`, `small`, `medium`) указатели короткие.

```
int * iptr;  
char _far * chptr;
```

Если модель памяти `compact`/`large`/`huge`, то `iptr` и `chptr` занимают по 4 байта каждый. Если модель памяти `tiny`/`small`/`medium`, то `iptr` занимает 2 байта, а `chptr` — все равно 4, поскольку его дальность установлена принудительно — `_far`.

13.2. Правила использования регистров

Рассмотрим правила использования регистров (сегментных регистров, регистров общего назначения и регистра флагов), принятые при трансляции с языка `C`.

13.2.1. Сегментные регистры

Регистры `ds` и `ss` устанавливаются при выполнении входного кода (`Entry`-кода) из стандартной библиотеки `C`. К моменту вызова функции `main` регистры `ds` и `ss` установлены так, как требует выбранная модель памяти, их значения должны оставаться такими же.

Правило: после выполнения ассемблерной подпрограммы или вставки значения регистров `ds` и `ss` должны остаться прежними.

Значение регистра `es` в `Microsoft/Borland-C` не определено и может быть изменено как в `C`-коде, так и в ассемблерных фрагментах. Если вы используете строковые команды (кроме `lods`), следует устанавливать `es`, причем заново каждый раз.

В `Watcom-C` изменение `es` не допускается.

13.2.2. Регистры общего назначения

В Microsoft/Borland-C подпрограмма может произвольно изменять значения `ax`, `bx`, `cx`, `dx`. Напротив, значения `si`, `di`, `bp`, `sp` в результате выполнения ассемблерного фрагмента или подпрограммы не должны измениться.

В Watcom-C никакие регистры не могут быть изменены.

13.2.3. Регистр флагов

Флаги состояния можно изменять как угодно.

По всей программе флаг направления `d` должен быть сброшен. (Нулевое значение `d` задает положительное приращение указателей `di` и `si` при выполнении строковых команд.) Если в фрагменте на ассемблере выполнена инструкция `std`, перед выходом следует задать `cld`.

Перейдем к рассмотрению наиболее простого варианта соединения C с ассемблером — ассемблерных вставок.

13.3. Ассемблерные вставки

При трансляции вставок ассемблер получает доступ к таблице имен C-транслятора. Это значит, что из вставки можно ссылаться на имена объектов, определенных в C-модуле. К ним относятся переменные, константы, функции, C-метки, формальные параметры функции и ее локальные переменные.

13.3.1. Ассемблерные вставки в Microsoft/Borland-C

В Microsoft/Borland-C вставка открывается ключевым словом `_asm`, после которого записывается или один оператор ассемблера, или несколько операторов в фигурных скобках. Несколько операторов могут быть также записаны в одной строке — через `;`. Как следствие, комментарии в стиле ассемблера недопустимы; зато имеется возможность воспользоваться комментариями в стиле C — `/* .. */`. Числовая константа задается в манере ассемблера или C, допускается любая форма.

Листинг 13.1. Ассемблерные вставки в среде Microsoft/Borland-C (см. `main1.c`)

```
void main() {
    char x = '*';

    _asm    ror     x, 2
```

```

_asm {
    test    x, 0xff
    jpe     skip
    xor     x, 1 shl 8
skip:
}

```

Во второй вставке запрограммировано дополнение до четности числа установленных в 1 бит байта *x*. Обратите внимание: числовая константа 0xff задана в стиле C.

Ограничения на допустимые конструкции ассемблера во вставках зависят от способа взаимодействия трансляторов (C и ассемблера). В ранних реализациях C, например, в Quick-C/Turbo-C, таблица C-имен доступна ассемблеру только на чтение. Поэтому никакие определения во вставках не допускаются. Запрещено все, что могло бы привести к расширению таблицы имен:

- ☐ ставить метку;
- ☐ определять массив директивой `db` или `dw` с именем;
- ☐ определять макрокоманды;
- ☐ пользоваться директивой `segment`.

Во всех реализациях Microsoft/Borland-C ассемблерная вставка препятствует оптимизации с использованием регистров, поскольку применение большинства регистров — `ax`, `bx`, `cx`, `dx`, `es` — транслятором уже не контролируется. Оптимизация C-функции, содержащей ассемблерную вставку, отключается.

13.3.2. Ассемблерные вставки в Watcom-C

Ассемблерные вставки в среде Watcom-C не отменяют оптимизацию, что достигается за счет явного указания изменяемых регистров.

Вставки оформляются в виде `inline`-функций с параметрами и результатом. Указывается, в каких регистрах должны быть записаны параметры, в каких — результат. Также в заголовке `inline`-функции перечисляются регистры, значение которых будет изменено.

Листинг 13.2. Ассемблерные вставки в среде Watcom-C (см. `main2.c`)

```

char even( char);
#pragma aux even =
    test    al, 0xff
    jpe     skip
    xor     al, 1 shl 8

```

```

skip:                                     \
      parm    [al]                       \
      modify  [al]                       \
      value   [al];

void main() {
    char x = even ('*');
}

```

Вызов функции `even` транслируется в последовательность инструкций:

```

      mov     al, '*'
      test    al, 0ffh
      jpe     skip
      xor     al, 080h
skip:
      mov     x, al

```

Обратите внимание: хотя ассемблерный фрагмент описан как функция, эта функция подставляемая, обращение к ней выполняется без машинной инструкции вызова, а в самой функции нет парной команды возврата. При каждом вызове тело функции вставляется заново целиком, как при вызове макрокоманды.

Объявление `parm` задает список регистров для передачи параметров. В примере параметр один — типа `char`; задана передача его в одном регистре — `al`.

Объявление `value` задает набор регистров для представления результата. В данном случае результат — типа `char`, возвращаем его в одном регистре — `al`.

В объявлении `modify` перечисляются регистры, значение которых в результате выполнения функции изменяется. С-транслятор сохраняет эти регистры в стеке, или отказывается от их использования при оптимизации — как ему покажется выгодней.

В примере с `even` объявление `modify [al]` избыточное, поскольку `al` используется для возвращения результата. Возможность пересечения множеств `modify`, `value` и `parm` учтена, так что ошибки здесь нет.

Рассмотрим более сложный пример. Определим вставку, которая заполняет ASCIIZ-строку заданным значением; длина строки возвращается в качестве результата.

Листинг 13.3. Заполнение ASCIIZ-строки заданным значением

```

/* main3.c */

int fill( char far *, char);

```

```
#pragma aux fill =
    mov     cx, -1
    mov     al, 0
    repne   scasb
    not     cx
    dec     cx

    mov     dx, cx
    mov     al, ah
    sub     di, 2
    std
    rep     stosb
    cld
    parm    [es di] [ah]
    modify  [al cx]
    value   [dx];
```

```
char * s = "I got a lot to say! I can't remember now!";
```

```
void main() {
    int x = fill (s, '!');
}
```

Первый параметр — адрес строки. Он передается как дальний указатель, в паре регистров `es:di`. Почему дальний, а не ближний? Затем, чтобы транслятор сам сформировал адрес данных `es:di` для команд `scasb` и `stosb`. Второй параметр — типа `char` — передается в `ah`.

Справа от `parm` в каждой паре квадратных скобок задан регистр (или их набор) для одного параметра.

```
parm    [es di] [ah]
parm    [es di ah]
parm    [es] [di] [ah]
```

Первая директива задает передачу первого параметра в `es:di`, второго — в `ah`. Вторая директива задает передачу лишь одного параметра, зато в трех регистрах. Третья директива определяет передачу трех параметров. Будьте внимательны!

Примечание

В директиве `modify` регистры перечисляются в квадратных скобках по отдельности или вместе — не имеет значения.

В программе из листинга 13.3 сначала требуется определить длину строки — число литер до байта со значением 0 (null-литера). После того как завершен поиск, `di` адресует байт *после* null-литеры. В `cx` вычисляется длина строки и

сохраняется в `dx`. Значение `di` уменьшаем так, чтобы `di` указывал на последний значащий байт строки. Затем, после команды `std` выполняется запись в обратном направлении. В конце восстанавливается исходное значение флага направления `d`.

Мы познакомились с двумя вариантами встраивания ассемблера в C. Перейдем к рассмотрению другого способа объединения ассемблера с C — за счет компоновки после раздельной трансляции.

13.4. Интерфейс C-ассемблер при раздельной трансляции

В этом варианте интерфейса следует согласовывать вызов *внешней* функции из C-программы с подпрограммой на ассемблере, заданной в отдельном модуле.

Реализации Microsoft/Borland-C и Watcom-C отличаются друг от друга способом передачи параметров. Microsoft/Borland-C передает параметры через стек, а Watcom предпочитает регистры. В последнем случае доступ к параметрам из подпрограммы оказывается проще. Поэтому в первых примерах мы ориентируемся на регистровую передачу.

Предварительно рассмотрим общие вопросы, не зависящие от способа передачи параметров: соглашения о сегментах кода, об именах функций, о значении функции.

13.4.1. Соглашения о сегментах кода

C-модуль вызывает ассемблерную подпрограмму из другого модуля. Ситуация нам знакома по материалу *главы 10*. Напомним — чтобы компоновка прошла успешно, следует объединить вызов и подпрограмму в одном сегменте (если вызов ближний), при этом имя подпрограммы указывается в `extrn`-объявлении вызывающего модуля и в `public`-объявлении вызываемого модуля.

Дальность вызова в C-программе по умолчанию определяется моделью памяти. В моделях `tiny`, `small` и `compact` код объединяется в один общий сегмент, команды вызова подпрограмм — короткие, внутрисегментные. Название сегмента кода для перечисленных моделей памяти стандартизировано:

```
_TEXT    SEGMENT PUBLIC 'CODE'
```

Подпрограмма, рассчитанная на ближний вызов, должна быть помещена в этот сегмент. В `a86` объявлять стандартный сегмент кода не обязательно — если директива `segment` не задана и все возвраты ближние (`ret`), назначается именно этот сегмент.

Примечание

Если в отсутствие `segment` обнаружена хотя бы одна инструкция `retf`, назначается `private`-сегмент с именем `<name>_ТЕХТ`, где `<name>` — название модуля (по умолчанию совпадает с именем исходного файла).

Если подпрограмма рассчитана на дальний вызов, она может быть помещена в `private`-сегмент с любым именем.

13.4.2. Соглашения об именах С-объектов

С-транслятор преобразует имена *глобальных* функций при записи в `obj`-файл, добавляя к ним литеру подчеркивания. Реализация С Microsoft/Borland добавляет подчеркивание в начале имени функции, а Watcom — в конце. Преобразование имен следует учитывать, определяя метку начала подпрограммы.

Примечание

Функция не глобальная, если она объявлена с атрибутом `static`.

Приведем пример интерфейса Watcom-С с `a86`. Проект состоит из главного модуля `main4.c` и модуля на ассемблере `subr4_n.08`.

Листинг 13.4. Интерфейс С-ассемблер при раздельной трансляции

```
/* main4.c */
char even (char);
char ch;

void main() { ch = even( 10); }

; subr4_n.08 (near version)
even_:                                ; !
    test    al
    jpe     ret
    xor     al, bit 7
    ret
```

Обратите внимание, директива `public even_` в `subr4_n.08` опущена. Напомним, если в `a86` не задано ни одной директивы `public`, то все имена (кроме локальных) считаются `public`-именами.

Параметр типа `char` передается подпрограмме в регистре `al`. Подпрограмма дополняет до четности число бит, установленных в 1, и возвращает результат типа `char` в регистре `al`. Код, сгенерированный С-транслятором в окрестности вызова, следующий:

```
mov     al, 10
call    near even_
mov     _ch, al
```

Рассмотренный вариант `even_` рассчитан на короткий вызов, поскольку возврат выполняется командой `ret`. Так как в `main4.c` дальность вызова функции `even` не указана, программа будет работать только в моделях памяти `tiny`, `small`, `compact`.

В листинге 13.5 приведен вариант модуля на ассемблере для моделей памяти `medium`, `large`, `huge`. Модуль `main4.c` прежний.

Листинг 13.5. Модуль на ассемблере для моделей памяти `medium`, `large`, `huge`

```
; subr4_f.08 (far version)

even_:
    test    al
    jpe     ret
    xor     al, bit 7
    retf                                ; !
```

В листинге 13.6 показано автоматическое преобразование ближнего возврата `ret` в дальний возврат `retf`. Преобразование выполняется автоматически в промежутке между директивами `<name> proc far` и `endp`.

Листинг 13.6. Автоматическое преобразование `ret` в `retf`

```
; subr4_fp.08

even_  proc    far                ; !
    test    al
    jpe     ret
    xor     al, bit 7
    ret                                           ; -> retf
endp                                         ; !
```

Примечание

В моделях памяти с отдельными сегментами (`compact`, `large`, `huge`) код каждого модуля поступает в отдельный `private`-сегмент с именем `<name>_TEXT`.

Сделаем так, чтобы подпрограмма не зависела от модели памяти. Для этого оформим ее в расчете на `far`-вызов, а в `C`-модуле явно зададим дальность вызова.

Листинг 13.7. Задание far-вызова в произвольной модели памяти

```
/* main4_f.c + (subr4_fp.08 | subr4_f.08) */

char _far even (char);          /* ! */
char ch;

void main() { ch = even( 10); }
```

В этих примерах результат типа `char` возвращается из функции в `al`. Рассмотрим соглашения о передаче результата для остальных элементарных типов данных `C`.

13.4.3. Соглашения о результате функции

Результат функции возвращается в регистре или в регистрах, в зависимости от размерности результата. В табл. 13.1 для каждого типа указаны: число байт в машинном представлении и регистр/регистры, в которых записывается результат.

Таблица 13.1. Машинное представление результата C-функции для элементарных типов

С-тип	Размер в байтах	Представление результата
char	1	al
int, short	2	ax
near ptr	2	ax
long	4	dx:ax
far ptr	4	dx:ax
float	4	dx:ax, st()
double	8	ax:bx:cx:dx, st()

Как видите, результат размерностью до четырех байт всегда возвращается в аккумуляторе — `al/ax/dx:ax`. Результат типа `float/double` возвращается либо в регистрах, либо на вершине стека сопроцессора (если транслятор настроен на подстановку инструкций `i80x87`).

Листинг 13.8. Пример формирования результата типа long

```
/* main5.c */
long even (long);
```

```

void main() { long x = even( 0x800001); }

; subr5.08

check    macro
#rx11

        test    #x
        jpe     >11
        cmc

11:
#em

even_:
        clc
        check   al, ah, dl, dh
        jc      ret
        xor     dh, bit 7
        ret

```

Входной параметр типа `long` передается Watcom-C в регистрах `dx:ax`, результат типа `long` возвращается в тех же регистрах. Поскольку флаг `e` отражает четность числа единиц только в младшем байте проверяемого значения, проверок должно быть четыре — по числу байт.

Перейдем к рассмотрению способов передачи параметров. Начнем с регистрового, применительно к Watcom-C.

13.4.4. Передача параметров через регистры

Для передачи параметров используются регистры `ax`, `dx`, `bx`, `cx` — в таком порядке. Число и назначение регистров определяется размерностью параметра.

- ☐ Параметр типа `int`, `short` и `near ptr` передается в одном регистре.
- ☐ Параметр типа `char` передается не в байтовом регистре, а в двухбайтовом, с предварительным преобразованием в `int` (способ преобразования определяется атрибутом `signed/unsigned`).
- ☐ Параметры типа `long`, `far ptr` передаются в паре регистров `dx:ax` или `cx:bx`.
- ☐ Если транслятору не указано использовать инструкции сопроцессора, параметры типа `float/double` передаются в регистрах (соответственно, в паре `dx:ax` или `cx:bx` — для `float`, или в четверке `ax:bx:cx:dx` — для `double`).

Параметры, оставшиеся после распределения регистров, передаются через стек.

Правила сохранения регистров:

- ❑ регистры, изменяемые при выполнении подпрограммы, должны быть сохранены на входе и восстановлены на выходе;
- ❑ исключение составляют регистры, в которых передаются значения параметров и возвращаются результаты.

Проиллюстрируем на примере функции `fill`, рассмотренной ранее в листинге 13.3.

Листинг 13.9. Передача параметров подпрограмме `fill`

```
/* main6.c */

int fill( char far *, char);
char * s = "I got a lot to say! I can't remember now!";

void main() {
    int x = fill (s, '!');
}
```

Первому параметру типа `far ptr` назначаются регистры `dx:ax`. Остаются для передачи параметров регистры `bx, cx`. Второму параметру (типа `char`) назначается первый из свободных регистров — `bx`. Результат функции (типа `int`) возвращается в `ax`.

Листинг 13.10. Доступ к параметрам из подпрограммы `fill_`

```
/* subr6.08 */
fill_:
    push    es, di, cx        ; (1)
    mov     es, dx            ; (2)
    mov     di, ax            ; (3)

    mov     cx, -1
    mov     al, 0
    repne   scasb
    not     cx
    dec     cx

    mov     al, bl            ; (4)
    mov     bx, cx            ; (5)
    sub     di, 2
    std
    rep     stosb
    cld
```

```

mov      ax, bx          ; (6)
pop      cx, di, es      ; (7)
ret

```

Пара `es:di` формируется командами (2) и (3). Команда (4) устанавливает значение `al` для `stosb` — по значению из младшего байта второго параметра. Длина строки, вычисленная в `cx`, сохраняется (5) в `bx`, а перед выходом копируется (6) из `bx` в выходное значение `ax`. Регистры, изменяемые при выполнении подпрограммы `fill_`, сохраняются в начале (1) и восстанавливаются (7) перед выходом. Регистр `bx` сохранять не обязательно, поскольку в нем был передан параметр.

Параметр передается через стек, если:

- ☐ в списке свободных регистров нет подходящих;
- ☐ параметр задан структурой, размер которой превышает 8 байт;
- ☐ параметр представляет собой действительное значение, при этом транслятору задано генерирование инструкций `i80x87`.

Примечание

При организации интерфейса с ассемблером обязательно задавать прототип функции. Иначе транслятору придется угадывать тип параметров — с характерным в таких ситуациях пессимизмом: если фактический параметр — типа `char` или `int`, то для формального параметра принимается наиболее "емкий" из `int`-типов — `long int`. Аналогично, `near ptr` на всякий случай "растягивают" до `far ptr`, а `float` преобразуют в `double`. В результате, выбранная транслятором схема распределения регистров для передачи параметров отличается от рассчитанной по общим правилам.

13.4.5. Передача параметров через стек

Этот способ передачи параметров пригоден для любого числа параметров и поддерживается всеми C-трансляторами. По сравнению с ранее рассмотренным регистровым способом, он порождает больше кода и работает медленней. Доступ к параметрам из подпрограммы усложняется и, кроме того, зависит от дальности вызова подпрограммы.

Передача при ближних вызовах

Вернемся к функции `even`, пример которой приведен в *разд.13.4.2*.

Обращение к `even` транслируется Microsoft/Borland-C следующим образом:

```

push     10                ; (1)
call     _even             ; (2)
add      sp, 2              ; (3)
mov      _ch, al

```

Значение параметра типа `char` записывается в стек (1) — как слово (байт записать в стек нельзя). После обращения (3) к `_even` команда (3) освобождает место в стеке, занятое параметрами.

Листинг 13.11. Подпрограмма `_even`, при передаче параметров через стек

```
; subr7.08 (+ main4.c)

_even:
    push    bp                ; (1)
    mov     bp, sp            ; (2)
    mov     al, [bp+4]        ; (3)
    test    al
    jpe     >11
    xor     al, bit 7
11:    pop     bp                ; (4)
    ret
```

На рис. 13.1 показан фрагмент стека на входе в подпрограмму `_even` — сразу после выполнения инструкции `call _even`.

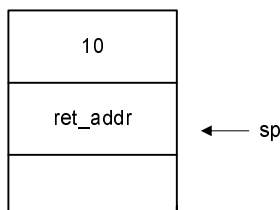


Рис. 13.1. Вершина стека при входе в подпрограмму `_even`

Под значением параметра в стеке записан адрес возврата `<ret_addr>`, в результате выполнения инструкции `call`. Вызов ближний, поэтому `<ret_addr>` — слово. Параметр находится по адресу `ss:sp+2`, но адресовать его через `sp` нельзя — косвенная адресация через `sp` не поддерживается.

Для доступа к произвольной записи стека специально предназначен регистр `bp`. При косвенной адресации с участием `bp` сегментная составляющая адреса берется из `ss` — как раз то, что требуется.

Раз мы используем `bp`, его следует сохранить — сразу при входе (1) в подпрограмму. Затем значение `bp` устанавливается (2) равным текущего значению `sp`; значение `bp` фиксировано на время выполнения подпрограммы.

На рис. 13.2 показан полученный *кадр стека* — фрагмент стека от первого параметра до записи, адресуемой `sp` после фиксации `bp`.

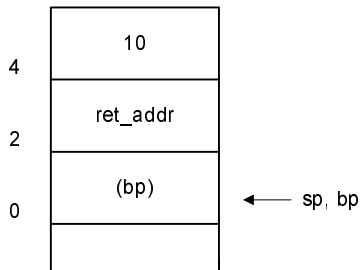


Рис. 13.2. Кадр стека после установки базы стека (слева показаны смещения от базы)

Цифры слева на рис. 13.2 — это смещения записей стека относительно *базы* стека (база стека — это запись, на которую указывает bp). Теперь параметр доступен по адресу `ss:bp+4`, и подпрограмма `_even` копирует (3) его значение в `al`.

На выходе (4) из подпрограммы исходное значение `bp` восстанавливается из стека, а сам стек теперь выглядит так же, как при входе в подпрограмму (см. рис. 13.1). После возврата командой `ret` в вызывающую программу в стеке остался один параметр, больше не нужный.

Кадр стека при дальних вызовах

Если вызов дальний, то размер `<ret_addr>` изменится в большую сторону, что приведет к смещению параметров относительно базы стека.

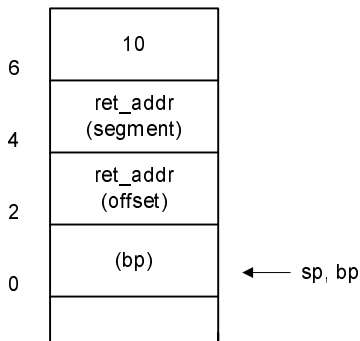


Рис. 13.3. Кадр стека при far-вызове подпрограммы `_even`

Смещение первого параметра при far-вызове, после фиксации `bp`, становится равным шести.

Листинг 13.12. Подпрограмма `_even` в расчете на дальний вызов

```
; subr8.08 (+ main4_f.c)

_even:
    push    bp
    mov     bp, sp
    mov     al, [bp+6]        ; (!)
    test    al
    jpe     >11
    xor     al, bit 7
11:    pop     bp
    retf                    ; (!)
```

Рассмотрим инструкции, упрощающие создание кадра стека и его ликвидацию, с дополнительной возможностью резервирования памяти для локальных данных.

Инструкции *enter* и *leave*

Инструкция `enter` заменяет пару команд, требуемых для сохранения `bp` и его установки. В дополнение, `enter <n>` резервирует `<n>` байт в стеке за счет уменьшения `sp`.

```
enter    <n>      ;      push    bp
                ;      mov     bp, sp
                ;      sub     sp, <n>
```

На рис. 13.4 показан кадр стека при `<n>`, равном четырем. Вызовы полагаем близкими — сейчас и в дальнейшем.

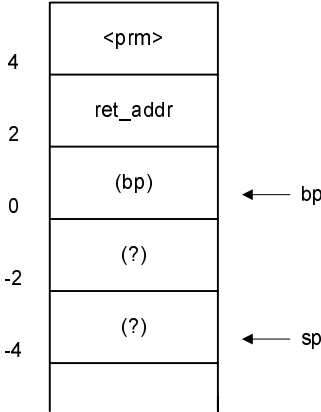


Рис. 13.4. Кадр стека при ближнем вызове после выполнения `enter 4`

За счет смещения вершины стека резервируются 4 байта вниз от базы стека. Эти данные адресуются отрицательными смещениями, начиная с `[bp-2]`, используются для хранения промежуточных результатов.

Примечание

Машинная инструкция `enter` содержит два параметра — объем локальных данных и лексический уровень подпрограммы. Последнее значение в контексте языка C всегда ноль, поскольку вложенные функции в C не поддерживаются.

На выходе, перед выполнением `pop bp` необходимо, чтобы `sp` указывал на базу стека. Для этого достаточно выполнить присвоение:

```
mov     sp, bp
```

Инструкция `leave` заменяет пару инструкций по восстановлению `sp` и `bp`:

```
leave           ; mov     sp, bp
                ; pop     bp
```

Листинг 13.13. Пример из листинга 13.11 с применением команд `enter` и `leave`

```
; subr8_1.08 (+ main4_f.c)

_even:
    enter     0
    mov      al, [bp+4]
    test     al
    jpe      >11
    xor      al, bit 7
11:    leave
    ret
```

Использование локальных данных проиллюстрируем примером, показанным в листинге 13.14. В нем приведена функция `trunc` для преобразования действительного значения типа `double` в целое число типа `int`, с отбрасыванием дробной части.

Листинг 13.14. Создание локальных данных и доступ к ним

```
/* main9.c */

int trunc( double);

void main() {
    int x = trunc (10.9 * 16.1 - 7.7);
}
```

```
; subr9.08
```

```
_trunc: enter    2
          fstcw   [bp-2]                      ; (1)
          mov     ax, [bp-2]
          and     ax, not (bit 10 + bit 11)
          or      ax, 3 shl 10
          xchg    ax, [bp-2]
          fldcw   [bp-2]                      ; (2)
          fld     q [bp+4]                    ; (*)
          frndint
          xchg    ax, [bp-2]
          fldcw   [bp-2]                      ; (3)
          fstp    w [bp-2]                    ; (**)
          mov     ax, [bp-2]                  ; (**)
          leave
          ret
```

Параметр — действительное число типа `double` (8-байтное). Временные данные (слово) предназначены для чтения-записи (1—3) регистра управления `i80x87`. Новое значение для установки требуемого режима округления, записывается (2) в регистр управления. После инструкции округления `frndint` восстанавливается (3) исходное значение, которое было сохранено в `ax`.

Примечание

В этом примере острая необходимость в локальных данных отсутствует — можно было воспользоваться записью, занятой параметром. Достаточно перенести команду (*) в начало, и — после загрузки параметра в сопроцессор — значение двойного слова по адресу `[bp+4]` больше не нужно, место свободно.

До сих пор в примерах был только один параметр. Рассмотрим порядок передачи через стек нескольких параметров.

Порядок передачи параметров

Если параметров несколько, они записываются в стек в обратном порядке — от конца списка параметров к началу. Заметим, что при передаче параметров через регистры порядок прямой.

В качестве примера рассмотрим вызов функции `dif`, которая возвращает результат вычитания параметров типа `int`.

```
/* main10.c */
int dif (int, int);

void main() { int seven = dif( 12, 5); }
```

Вызов `dif` транслируется следующим образом:

```
push    5           ; <prm2>
push    12          ; <prm1>
call    _dif
add     sp, 4
```

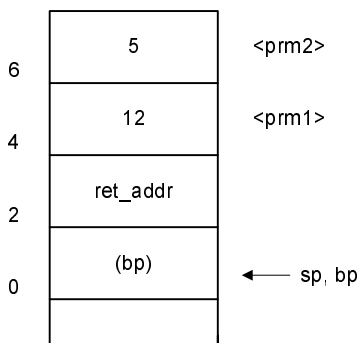


Рис. 13.5. Кадр стека после выполнения инструкции `enter` в подпрограмме `_dif`

Поскольку стек растет вниз, параметры расположены по возрастанию адреса — `<prm1>` примыкает к `<ret_addr>`, `<prm2>` находится над `<prm1>`.

Примечание

В трансляторах с языка Pascal (и его производных) параметры записываются в прямом порядке. Имеется возможность такой же порядок установить и для C-функций — для этого при объявлении прототипа функции следует задать атрибут `pascal`.

Подпрограмма `_dif`, предназначенная для вычисления разности `<prm1> - <prm2>`, выглядит следующим образом:

```
; subr10.08
```

```
_dif:  enter    0
        mov     ax, [bp+4]      ; a
        sub     ax, [bp+6]      ; b
        leave
        ret
```

Соглашения об удалении параметров

В трансляторах с языка C принято, что после завершения функции место, занятое параметрами, должна освобождать вызвавшая программа. Так же,

как и обратный порядок записи параметров, это решение вызвано наличием функций с переменным числом параметров.

В языках семейства Pascal число параметров для процедуры неизменно. Поэтому там применяется альтернативный вариант, и более экономный — параметры удаляет сама подпрограмма, перед завершением. В процессоре i80x86 для реализации этого варианта предусмотрена команда `ret/retf <n>`, где `<n>` — число, прибавляемое к `sp` после выполнения возврата.

В языке C такой способ удаления параметров применяется только к тем функциям, которые объявлены с атрибутом `pascal`.

```
/* main11.c */
int pascal dif (int, int);

void main() { int seven = dif( 12, 5); }
```

Подпрограмма `_dif` выглядит теперь следующим образом (напомним, что в результате применения атрибута `pascal` также изменился порядок передачи параметров):

```
; subr11.08

dif_:   enter    0
        mov     ax, [bp+6]      ; !
        sub     ax, [bp+4]      ; !
        leave
        ret     4              ; ret
                                   ; add    sp, 4
```

Сокращенные обозначения параметров и локальных данных

Обозначения `[bp+4]`, `[bp+6]`, `[bp-2]` и пр. сокращаются за счет введения синонимов, что имеет смысл при частых ссылках на данные в кадре стека. Ниже продемонстрирован первый способ введения синонимов — директивой `equ`.

```
; subr10_1.08 (+ main10.c)
```

```
prm1    equ     [bp+4]
prm2    equ     [bp+6]

_dif:   enter    0
        mov     ax, prm1
        sub     ax, prm2
        leave
        ret
```

В а86 кодирование доступа к параметрам упрощается также за счет определения и использования *базированных* структур:

```
; subr10_2.08 (+ main10.c)
```

```
struc [bp+4]
prm1      dw      ?
prm2      dw      ?
ends

_dif:     enter    0
          mov      ax, prm1
          sub      ax, prm2
          leave
          ret
```

По директиве `struc` ассемблер обнуляет счетчик адресов, как при трансляции сегмента. Имени `prm1` присваивается значение ноль, `prm2` получает значение два; тип этих имен — `word`. После директивы `ends` счетчик адресов восстанавливается, а имена `prm1` и `prm2` означают теперь `w[bp+4+0]` и `w[bp+4+2]`.

На этом изложение основного материала этой темы завершено. Далее рассмотрены специальные случаи, возникающие при передаче параметров через стек.

Параметры дальние указатели

С параметром типа `far ptr` мы уже встречались в *разд. 13.4.4* (см. листинг 13.9). Напомним:

```
/* main6.c */

int fill( char far *, char);
char * s = "I got a lot to say! I can't remember now!";

void main() {
    int x = fill (s, '!');
}
```

В примере с передачей параметров через регистры в *разд. 13.4.4* (см. листинг 13.10), пара `es:di` в подпрограмме `fill_` инициализируется значениями параметра `dx:ax`:

```
mov     es, dx      ; (2)
mov     di, ax      ; (3)
```

При передаче параметров *через стек* значение дальнего указателя находится в двойном слове по адресу `ss:[bp+4]`. Вполне допустимо устанавливать значения регистров из пары `es:di` по одному:

```
mov    di, [bp+4]
mov    es, [bp+6]
```

Быстрее устанавливать `es:di` одной командой — `les` (см. `subr6_1.08`):

```
les    di, [bp+4]
```

Команда `les <reg>, <mem32>` записывает в пару регистров `es:<reg>` значения из двойного слова `<mem32>`. В `<reg>` копируется младшее слово `<mem32>`, в `es` — старшее. Для установки пары `ds:<reg>` имеется аналогичная команда `lds`.

Обращение к глобальным данным

В рассмотренных примерах подпрограмма на ассемблере обращается к параметрам, переданным в регистрах или в стеке. Подпрограмма изменяет данные, определенные в модуле на языке C, исключительно косвенно — по адресу, переданному ей в параметре типа `near ptr/far ptr`, как в примерах с функцией `fill` из *разд. 13.3.2* (см. листинг 13.3) и *разд. 13.4.4* (см. листинг 13.10).

В некоторых случаях может потребоваться прямой доступ к данным, определенным в модуле на языке C, т. е. доступ по имени C-объекта. В других случаях подпрограмме требуются собственные статические данные, значения которых должны сохраняться между вызовами подпрограммы. (Например, в статических данных библиотеки, разработанной на ассемблере, хранятся настройки библиотеки — скажем, тип дисплея для графической библиотеки, рассчитанной на различные устройства отображения.)

Обращение к данным языка C и к собственным данным модуля предполагает соответствующую настройку регистра `ds`.

В моделях памяти с одним сегментом данных (`tiny`, `small`, `medium`) текущий сегмент данных `ds` установлен на группу `DGROUP`, в которой C-транслятор объединяет:

- ❑ инициализированные данные (сегмент `_DATA` класса `"DATA"`);
- ❑ константы размерностью больше слова (сегмент `CONST` класса `"CONST"`);
- ❑ неинициализированные данные (сегмент `_BSS` класса `"BSS"`);
- ❑ стек (сегмент `STACK` класса `"STACK"`).

Доступ к глобальным данным C

В моделях памяти `tiny`, `small`, `medium` значение `ds` фиксировано, и подпрограмма имеет возможность напрямую ссылаться на C-имена, не меняя установки `ds`.

Листинг 13.15. Доступ к С-объектам при фиксированном значении ds

```

/* main12.c */
void even_ch ();
char ch;

void main() { ch = 10; even_ch (); }

; subr12.08

        extrn    _ch:b                ; !
even_:
        test     _ch                    ; !
        jpe      ret                    ; !
        xor      _ch, bit 7             ; !
        ret

```

Примечание

Объявление `extrn` в подпрограмме `subr2` понадобилось лишь потому, что тип имени `_ch` в инструкциях `test` и `xor` требует уточнения.

Если модель памяти допускает несколько сегментов данных, то уже нет гарантии, что данные С-модулей непременно попадут в `DGROUP`.

Примечание

Казалось бы, наоборот, точно должны не попасть! Тем не менее, С-транслятор не торопится отправлять данные в `private`-сегменты. В Watcom-C, например, чтобы оказаться в отдельном сегменте, объект должен быть или не меньше 32 Кбайт, или должен быть задан с атрибутом `_far` (например, `int far x;`), причем вне функций.

Так или иначе, полной уверенности в местоположении данных нет. Доступ будет надежным только в том случае, если каждый раз заново устанавливать сегментный регистр.

Листинг 13.16. Доступ к С-объектам при нефиксированном значении ds

```

; subr12_d.08

        extrn    _ch:b

even_:
        push     ds
        mov      ds, seg _ch           ; !
        test     _ch
        jpe      >11

```

```

xor     _ch, bit 7
pop     ds
ret

```

Оператор `seg <name>` из листинга 13.6 дает значение сегментной составляющей адреса объекта `<name>`.

Доступ к статическим данным подпрограммы

При определении статических данных в модуле на языке ассемблера желательно поместить эти данные в `DGROUP`. В этом случае доступ к данным не потребует перенастройки `ds`.

Данные объявляются в одном из сегментов в составе `DGROUP` — в любом, кроме сегмента стека. Не следует помещать в `_BSS` данные с определенными начальными значениями (инициализированные данные) — при запуске содержимое `_BSS` может быть обнулено в результате выполнения входного кода из библиотеки `C`.

После объявления сегмента данных следует задать принадлежность этого сегмента группе `DGROUP`.

Листинг 13.17. Задание переменных в группе `DGROUP`

```

; subr4_d.08 (+ main4.c)

_DATA    segment public '_DATA'
limit    db      10

DGROUP   GROUP    _DATA                ; !

_TEXT    segment public 'CODE'
even _:

        dec      limit
        jnz      >11
        mov      ax, 04c01
        int      021                ; exit to DOS
11:      test     al
        jpe      ret
        xor      al, bit 7
        ret

```

В сегменте `_DATA` инициализированных данных определен байт с адресом `limit`. Подпрограмма при каждом ее вызове уменьшает счетчик `limit` и при его обнулении прекращает работу всей программы.

Если вы объявите статические данные в `private`-сегменте, данные в `DGROUP` не попадут, и для доступа к ним придется каждый раз настраивать `ds` или `es`.

13.5. Выполнение примеров

При трансляции ассемблерных вставок возможны ошибки, если вы используете расширения языка а86. Совместимость а86 с традиционными ассемблерами фирм Watcom, Microsoft, Borland рассмотрена в *приложении 7*.

Соединение С с ассемблером а86 при раздельной трансляции выполняется следующим образом:

1. Модуль на а86 транслируется с опциями +Cos (именно так, а не +cos).
2. При применении пакетного С-компилятора в командной строке указываются названия исходного файла С-модуля и obj-файла, полученного в результате трансляции модуля на а86. Например, при использовании утилиты Compile & Link (Watcom):

```
wcl /fpi87 /m main4.c subr4.obj
```

3. При использовании интегрированной среды разработки Borland/Microsoft-С следует создать проект (prj-файл или mak-файл) и включить в него с- и obj-файлы.
4. Для предварительного контроля результата компоновки рекомендуется заказывать карту загрузки.

Для отладки программы, полученной в результате раздельной трансляции, требуется остановка перед вызовом подпрограммы. Возможны два варианта:

1. Воспользоваться отладчиком в составе систем автоматизации программирования на языке С: Turbo Debugger, CodeView, Watcom Debugger. В этих отладчиках имеется возможность отображать С-программу как в виде исходного текста (для этого при трансляции следует заказать генерирование отладочной информации), так и в виде машинных команд на ассемблере. Для этого вы ставите точку останова на вызове подпрограммы, отображая программу в виде исходного текста, а остановившись, переключаете отображение на машинные команды. В результате, вы получаете возможность обозревать и обрабатывать по шагам машинный код в окрестности вызова, где заданы: передача параметров, вызов подпрограммы и очистка стека от параметров.
2. Воспользоваться d86, задав точку останова в коде программы. Точка останова задается инструкцией `int 3`. Рекомендуется перед вызовом отлаживаемой подпрограммы поставить вызов следующей подпрограммы (см. trap.08):

```
_trap:
trap_:  int     3
        retf
```

В модуле на языке С прототип `trap` объявляется следующим образом:

```
void _far trap ();
```

Подпрограмма `trap` годится для любой модели памяти и для любого С-транслятора. Не забудьте включить `trap.obj` в проект.

После загрузки `exe`-программы в `d86` запускайте программу командой **g**. Выполнение остановится на инструкции `int 3` в подпрограмме `trap`. Нужно сойти с команды `int 3` при помощи клавиши "стрелка вниз", а затем выполнить `retf` (нажатием клавиши `<F1>`), после чего вы вернетесь в окрестность вызова отлаживаемой подпрограммы. Поскольку указатель `sp` сдвинут относительно его исходного положения после загрузки `exe`-программы, то отображение стека прочитать невозможно. Следует сразу выполнить команду **Stack here** — нажатием комбинации клавиш `<Ctrl>+<T>`. Затем — двигаться по шагам.

Выполните примеры из *главы 13*. Если вы используете Watcom-C, в исходный текст примеров, иллюстрирующих передачу параметров через стек (рассчитанных на Microsoft/Borland-C), вставьте следующее определение из файла `ms_c.inc`:

```
#pragma aux MS_C " _*" \
    parm caller [] \
    value struct float struct routine [ax] \
    modify [ax bx cx dx es];
```

Здесь определен псевдоним `MS_C`, описывающий соглашения о вызовах Microsoft-C. В этом определении:

- ❑ `" _*"` — задает преобразование имен функций по схеме, принятой в Microsoft-C (литера подчеркивания добавляется в начале имени);
- ❑ `parm caller []` — определяет передачу параметров через регистры; заданный здесь пустой список регистров `[]` говорит о том, что регистры для этой цели не используются; опция `caller` означает, что место, занятое параметрами, освобождается вызывающей процедурой;
- ❑ `value struct...` — устанавливает один из вариантов соглашения о возврате результата в виде структуры (этот вопрос не рассматриваем, а применять эту возможность — возврат значения в виде структуры — не рекомендуем в принципе, особенно в паре С-ассемблер).

Между определением `MS_C` и объявлением прототипа функции поставьте оператор:

```
#pragma aux (MS_C) <name>;
```

В результате, Watcom-C ведет себя с функцией `<name>` так же, как Microsoft-C.

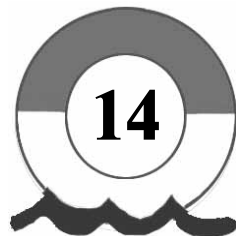
После выполнения примеров переделайте задание, выбранное вами в первой части книги.

1. Оформите вашу программу в виде С-программы с ассемблерными вставками; для удобства отладки задав во вставках остановки — `int 3`.

2. Оформите программу в виде подпрограммы на ассемблере с добавлением отдельного модуля на C. Останов задайте в виде вызова подпрограммы `trap`.

Внимание!

Оператор `trap` без скобок может быть пропущен транслятором без сообщения об ошибке. При этом `call trap` не генерируется! Будьте внимательны, при вызове любой C-функции скобки обязательны — `trap()` ;!



Обработка BCD-данных

Вот, например: раз, два, три! Ничего не произошло! Вот я запечатлел момент, в котором ничего не произошло.

Я сказал об этом Заболоцкому. Тому это очень понравилось, и он целый день сидел и считал: раз, два, три! И отмечал, что ничего не произошло.

За таким занятием застал Заболоцкого Шварц. И Шварц тоже заинтересовался этим оригинальным способом запечатлеть то, что происходит в нашу эпоху, потому что ведь из моментов складывается эпоха.

Даниил Хармс. "Однажды я пришел в Госиздат..."

В этой главе мы рассмотрим обработку числовых значений, заданных в двоично-десятичном формате, или BCD (Binary Coded Decimals).

14.1. Форматы BCD

Число в BCD-формате задано последовательностью десятичных цифр, каждая из которых занимает фиксированное число бит. Число бит для представления одной десятичной цифры равно либо четырем (тетрада), либо восьми (байт). Если под десятичную цифру отводится четыре бита, BCD называется "упакованным", а если восемь бит, то BCD — "неупакованное".

Пример BCD-числа 49 в упакованном и неупакованном форматах:

0 1 0 0 1 0 0 1 ; 049

0 0 0 0 0 1 0 0 0 0 0 0 1 0 0 1 ; 4 by 9
0 0 1 1 0 1 0 0 0 0 1 1 1 0 0 1 ; '4' by '9'

В программах на ассемблере упакованные BCD удобно задавать в шестнадцатеричном формате, например, число 49 задается в виде 049.

Чем заполнять старшие тетрады байтов неупакованного BCD-числа, точно не определено. При сложении и вычитании значение старшей тетрады без-

различно, но при умножении и делении старшие тетрады операндов должны быть заполнены нулями.

В последней строке из примера, BCD-число 49 определено парой литер '4' и '9'. В младших тетрадах заданы двоичные значения 4 и 9, а в старших тетрадах — двоичный код 0011 — избыток от задания цифр ASCII-кодами.

Возможность задания неупакованных BCD в виде ASCII-кодов отражена в названиях инструкций коррекции — например, `aaa` означает *Ascii Adjust after Addition*.

Знак BCD-числа, если значение считается знаковым, определяется — только для упакованных BCD — старшим битом старшей (знаковой) тетрады.

1xxx 0000 0000 ... 0100 1001	; -49
0xxx 0000 0000 ... 0100 1001	; +49

Отрицательные значения отличаются единицей в старшем бите старшей (знаковой) тетрады. Остальные биты этой тетрады значения не имеют. Разница между -49 и +49 заключается только в знаковой тетраде, остальные тетрады задают абсолютную величину числа.

Примечание

В системах с арифметическим сопроцессором, или *FPU* (Floating Point processor Unit), имеется возможность обрабатывать упакованные 10-байтные BCD (18 десятичных цифр + знаковый *байт*) при помощи *FPU*.

14.2. Операции над неупакованными BCD

Система команд `i80x86` предлагает средства для обработки неупакованных BCD по одной цифре за операцию. Обработка BCD из нескольких цифр требует нескольких итераций в зависимости от количества цифр.

14.2.1. Сложение и вычитание

Сложение и вычитание двоичных цифр выполняется теми же командами, которые применяются для обработки двоичных величин — `inc`, `dec`, `add`, `sub`. Складываемые цифры — двоичные значения в диапазоне 0—9 (0000xb—1001xb). Результат сложения или вычитания при использовании этих команд получается двоичный, что предполагает дальнейшее преобразование в формат BCD. Например, если прибавить единицу к тетраде 1001xb, получится 1010xb. Это десять, но в двоичном формате, а нам надо в BCD-формате; после коррекции результат должен стать равным 10000xb.

Инструкция `aaa` (*Ascii Adjust after Addition*) корректирует результат сложения двух десятичных цифр в младшей тетраде регистра `al`.

Примечание

В дальнейшем изложении тетрады байтового регистра обозначаются: <reg>-l (младшая) и <reg>-h (старшая).

Рассмотрим выполнение `aaa`. Если `al-l > 9`, то `aaa` корректирует величину в `al`. Значение `al-l` увеличивается на 6, после чего `al-h` обнуляется, а значение `ah` увеличивается на 1. Добавка 6 — это разность между представлением числа десять в неупакованном BCD-формате (10000xb) и в двоичном формате (1010xb). Можно считать, что `al-l` уменьшается на 10 — это то же самое, что добавление 6.

Если коррекция имела место, `aaa` устанавливает флаги `c` и `a`. Иначе, флаги `c` и `a` сбрасываются. Флаг *полупереноса* — `a` — отражает перенос из младшей тетрады в старшую.

При `ax = 1010xb` выполнение `aaa` приведет к следующим результатам. Значение в `al-l` равно `1010xb + 0110xb = 0000xb`, значение `al-h` равно `0000xb`, а значение `ah` равно `ah + 1 = 0 + 1 = 1`. Результат в `ax` — `10000xb`, т. е. десять в BCD-формате, что и требовалось получить.

Допустим, к `ax = 9` прибавили 8. Значение `al-l` стало равным `0001xb`, и произошел перенос единицы из бита 3 (флаг `a = 1`).

```

      1001
+     1000
-----
a <- 1 0001

```

При выполнении `aaa` значение `a = 1` задает коррекцию независимо от числа в `al-l`. Результат: значение `al_l` равно `0001 + 0110 = 0111 (7)`, `al_h` равно нулю, значение `ah` равно `ah + 1 = 0 + 1 = 1`. В итоге, `ax` равно `10111xb`, т. е. семнадцати в BCD-формате.

В целом, алгоритм выполнения инструкции `aaa` следующий:

```

IF (al-l > 9) OR (a = 1)
    al-l = al-l + 6
    ah = ah + 1
    c = a = 1
ELSE
    c = a = 0
END
al-h = 0

```

Обратите внимание, что `al-h` обнуляется в любой ситуации.

Коррекция после вычитания, по инструкции `aas` (Ascii Adjust after Subtraction), выполняется аналогично.

Рассмотрим пример, иллюстрирующий применение инструкции `aaa` при обработке неупакованных BCD-чисел из нескольких цифр. Требуется сло-

жить пару чисел из четырех десятичных цифр. Числа заданы в виде ASCII-строк по адресам *x* и *y*. Результат формируется в массиве по адресу *z*.

Листинг 14.1. Сложение неупакованных BCD многократной точности (см. ex1.8)

```
...
mov     bx, 3           ; i = 3..0 !
clc
11:     mov     al, x[bx]   ; (4)
        adc     al, y[bx]
        aaa
        mov     z[bx], al   ; (1)
        dec     bx         ; (2)
        jnl     11         ; (3)
...
```

Регистр *bx* — это индекс для доступа к массивам *x*, *y*, *z*. Чтение *x* и *y*, а также запись в *z* должны выполняться в обратном порядке — от младших цифр к старшим, поэтому значение *bx* уменьшается (2) на каждой итерации. Сложение выполняется с учетом переноса командой *adc*. До входа в цикл *c* = 0 (*clc*), а в цикле флаг *c* устанавливается по результату выполнения *aaa*.

Обратите внимание, что инструкция *adc* использует установку *c*, полученную при выполнении *aaa* на предыдущей (!) итерации. В ходе выполнения нескольких команд — (1), (2), (3), (4) — значение *c* не должно было измениться! Здесь обошлось без применения инструкций *pushf/popf*: инструкции *mov* и *j<x>* не действуют на флаги, а инструкция *dec* не затрагивает флаг *c* (!) — в отличие от *sub*, например.

Поскольку при выполнении команды *aaa* старшая тетрада *al* обнуляется, результат формируется в BCD-формате, но не в кодах ASCII. Если требуется в ASCII, то перед записью в массив *z* достаточно добавить код '0' к регистру *al*.

В варианте решения, приведенном в листинге 14.1, перенос между десятичными цифрами определяется по флагу *c*, установку которого выполняет *aaa*. Другой вариант приведен в листинге 14.2: вместо флага *c* используется автоматическое увеличение *ah* при коррекции.

Листинг 14.2. Вариант сложения неупакованных BCD многократной точности (см. ex2.8)

```
...
mov     bx, 3           ; i = 3..0 !
clc
mov     al, x[bx]       ; (1)
```

11:

```

mov     ah, x[bx-1]    ; (2)
add     al, y[bx]      ; (3)
aaa
mov     z[bx], al
mov     al, ah         ; (4)
dec     bx
jnl     11
...
```

К началу каждой итерации в регистре `al` подготовлено значение младшей цифры из очередной пары смежных цифр `x`. В начале цикла (1) в `ah` записывается старшая цифра пары. Таким образом, в `ax` находится пара смежных цифр из массива `x`. К регистру `al` прибавляется (3) соответствующая цифра `y` и, после выполнения `aaa`, в `ah` оказывается старшая цифра из пары цифр `x`, увеличенная в случае коррекции на единицу. Значение `ah` копируется (4) в `al` перед следующей итерацией.

В приведенных вариантах решения не учтена возможность переполнения. Так, результат сложения 7893 и 2639 должен быть 10532, а получается 0532. Старшая дополнительная цифра результата должна формироваться после цикла, по флагу `c` (см. ex2.8):

```

mov     al, 0
adc     al, 0           ; !
mov     z[bx], al      ; (bx = -1)
```

Для этой цифры следует зарезервировать байт перед массивом `z`.

14.2.2. Умножение и деление

Операции умножения и деления над операндами в формате BCD недопустимы. Под "коррекцией" в данном случае следует понимать преобразование формата — между BCD и двоичными (binary) числами. Преобразование из BCD в binary выполняется командой `aad`, а обратное — из binary в BCD — командой `aam`:

- ❑ `aad` (Ascii Adjust before Division) — преобразование пары десятичных цифр из `ax` в двоичное число в `al`, с обнулением `ah`.

Действие:

```

al = ah * 10 + al
ah = 0
```

флаги `s`, `p`, `z` устанавливаются по значению, полученному в `al`.

- ❑ `aam` (Ascii Adjust after Multiplication) — преобразование двоичного числа из `al` в пару десятичных цифр в `ax`.

Действие:

```
ah <- al / 10
al <- al mod 10
```

флаги *s*, *p*, *z* устанавливаются по значению, полученному в *al* (флаг *s* всегда сбрасывается, т. к. *al* ≥ 0 при любом исходном *ax*).

Примеры:

```
mov     ax, 0807
aad                     ; al <- 87, ah <- 0
aam                     ; ax <- 0807
```

"Ascii" в названиях этих инструкций — явная ошибка. Чтобы преобразование BCD→binary было правильным, в старших тетрадах BCD не должно быть ничего лишнего, так что ASCII-коды не годятся. Аналогично, при обратном преобразовании старшие тетрады обнуляются, и результат — вовсе не ASCII.

Предписания "after Multiplication" и "before Division" отражают лишь наиболее очевидный из способов применения инструкций *aam/aad*. Принято объяснять эти предписания следующим образом:

- ❑ перед делением следует преобразовать BCD в формат binary так, чтобы в *ax* подготовить делимое для команды *div <byte>*;
- ❑ после умножения, когда результат сформирован в *ax*, имеется возможность преобразовать результат в BCD при помощи *aam*.

На самом деле преобразование BCD→binary командой *aad* необходимо не только перед умножением, но и перед делением. Что касается обратного преобразования (*aam*), не обязательно выполнять его сразу после умножения, можно повременить до окончания вычислений.

Предположим, задана пара BCD-чисел *x* и *y*, каждое из двух цифр. Требуется вычислить $x = x \times 3 + y / 5$.

Листинг 14.3. Применение команд *aad/aam* (см. ex3.8)

```
...
mov     ax, x
aad                     ; BCD -> binary
mov     bl, 3
mul     bl              ; ax <- al * bl
mov     dx, ax

mov     ax, y
aad                     ; BCD -> binary
mov     bl, 5
```

```
div    bl      ; al <- ax / bl
cbw

add    ax, dx

aam                ; binary -> BCD
mov    x, ax
...
```

Как видите, применение `aad` связано не только с предстоящим делением, но и с умножением. Что касается `aam`, эта инструкция поставлена в конце последовательности вычислений, но не сразу после умножения.

14.2.3. Дополнительные возможности *aad* и *aam*

Во втором байте машинного кода `aad` и `aam` задано число десять — делитель для `aam` и множитель для `aad`. Оказывается, это значение может быть любым в диапазоне байта. В `i86` имеется возможность указать его после мнемоники команды, например:

```
aad 16      ; al = ah * 16 + al, ah = 0
aam 16      ; ah = al / 16, al = al mod 16
```

Инструкция `aad 16` позволяет одной командой, без сдвигов, преобразовать пару упакованных десятичных цифр в пару упакованных:

```
mov    ax, 2 by 9      ; ax = 0209
aad    16              ; al = 029
```

Инструкция `aam 16` выполняет обратное преобразование:

```
mov    al, 083
aam    16              ; al = 3, ah = 8
```

Рассмотрим пример преобразования числа, заданного ASCII-строкой, в упакованное BCD. Исходное значение расположено по адресу `ubcd`. Результат записывается в массив по адресу `pbcd`.

Листинг 14.4. Преобразование ASCII->BCD (см. ex4.8)

```
...
ubcd db    '1234567890'
sz    equ  $ - ubcd
pbcd db    sz/2 dup ?
...
```

```
    lea    si, ubcd + sz - 2
    lea    di, pbcd
    mov    cx, sz/2

11:
    std
    lodsw
    and    ax, 0f0f          ; Ascii -> binary
    xchg   al, ah
    aad    16                ; pack
    cld
    stosb
    loop   11
    ...
```

Чтение из `ubcd` выполняется в обратном порядке, поскольку цифры справа являются наименее значимыми; по этой же причине в цикле добавлена инструкция `xchg`. В каждом байте ASCII-кодов обнуляется старшая тетрада. После упаковки результат, полученный в `al`, сохраняется в очередном элементе массива `pbcd`.

Примечание

Подобные процедуры находят применение при обработке баз данных в `dbf`-формате. В этом формате значения всех типов данных хранятся в виде литер, как `ubcd` в примере. Если в системе обнаружен сопроцессор, то ASCII-данные преобразуются в формат упакованных 10-байтных BCD (18 цифр + знаковый байт) и передаются FPU. Результаты вычислений, прочитанные из FPU, преобразуются обратно в ASCII и в таком виде сохраняются в `dbf`-файле. Несмотря на сложность организации вычислений, хранение в формате ASCII в подобных приложениях выгодно, т. к. большую часть операций составляют не вычисления, а ввод и вывод.

14.3. Операции над упакованными BCD

Система команд `i80x86` предлагает средства для обработки упакованных BCD по две цифры за операцию. Предусмотрены команды `daa/das` для коррекции после сложения/вычитания.

14.3.1. Инструкции `daa` и `das`

Инструкция `daa` (Decimal Adjust after Addition) корректирует результат в `al` после сложения упакованных BCD из двух десятичных цифр. Если `al-1 > 9`, либо произошел перенос в старшую тетраду (установлен флаг `a`), то `al-1 = al-1 + 6`, `al-h = al-h + 1`. Затем, если `al-h > 9`, то `al-h = al-h + 6` и устанавливается флаг `c`.

Действие инструкции `das` (Decimal Adjust after Subtraction) аналогично. Если $al-l = 10-15$ или возник заем из старшей тетрады, то $al-l = al-l + 10$, $al-h = al-h - 1$. Затем, если $al-h = 10-15$, то $al-h = al-h + 10$ и устанавливается флаг `c`.

Пример использования `daa` после `add`:

```

59      0101 1001
+ 12    0001 0010
-----
        0110 1011 > 1001
      + 0001 0110      ; al-l + 6, al-h + 1
--      ----
71      0111 0001

```

После сложения BCD-чисел 59 и 12, значение $al-l > 9$, поэтому `daa` прибавляет 6 к $al-l$ и 1 к $al-h$ ($al-l$ и $al-h$ при выполнении операций коррективки изолированы друг от друга). Полученная в $al-h$ цифра не требует коррекции, т. к. $al-h \leq 9$. Еще один пример:

```

29      0010      1001
+ 79    0111      1001
-----
        1010 <-a- 0010
              + 0001      0110
              ----
        1001 < 1011      1000
              0110
--      ----
1 08    c <-- 0001      1000

```

После сложения 29 с 79 установлен флаг `a`, поэтому `daa` прибавляет шесть к $al-l$ и единицу к $al-h$. Значение $al-h > 9$, поэтому к $al-h$ прибавляется 6 и устанавливается флаг `c`. Результат с учетом `c = 1` равен 108 в BCD-формате.

14.3.2. Преобразования для деления и умножения

Форматные преобразования между упакованными BCD и двоичными числами выполняются за счет совместного применения инструкций `aam/aad`:

```

aam      16      ; packed -> unpacked
aad      ; unpacked -> binary

aam      ; binary -> unpacked
aad      16      ; unpacked -> packed

```

14.4. Операции над знаковыми BCD

Мы рассмотрели основные операции над беззнаковыми BCD. Операции со знаковыми BCD без сопроцессора предполагают замену сложения на вычитание (и наоборот), если в знаковой тетраде старший бит установлен в 1. Чтобы установить знак результата деления или умножения, следует учесть знаки операндов.

При наличии арифметического сопроцессора i80x87 организация вычислений упрощается. Упакованное 10-байтное BCD копируется в регистры FPU; при загрузке значение автоматически преобразуется в формат с плавающей точкой. Результат, после вычислений в FPU, считывается в оперативную память с обратным преобразованием формата.

Листинг 14.5. Обработка 10-байтных упакованных BCD в i80x87 (см. ex5.8)

```

...
x      dt      00023_4567_8903_2345_6789
y      dt      08022_4565_8900_2341_6784
z      dt      ?
...

finit                ; clear fpu
fbld  x              ; push bcd to fpu
fbld  y
fadd                ; add
fbstp  z             ; pop bcd from fpu
...

```

Десятибайтные переменные *x*, *y*, *z* определены при помощи директивы *dt* — Define Ten bytes. Значения заданы в шестнадцатеричном формате (0 — первая цифра); *asb* позволяет записывать длинные шестнадцатеричные константы по словам, через *'_'*. Значение *y* отрицательное — в старшей тетраде восьмерка (1000). Напротив, *x* положительное, так что в результате сложения *x* и *y* в FPU будет получена разница абсолютных значений; результат считывается из FPU в память по адресу *z*, с преобразованием в формат BCD.

14.5. Команды, воздействующие на флаг *a*

В завершение рассмотрим, какие команды и каким образом воздействуют на состояние флага промежуточного переноса *a*. Эти сведения нужны в тех ситуациях, когда команды *aaa*, *aas*, *daa*, *das* выполняются не сразу после *add*, *adc*, *sub*, *sbb*, *inc*, *dec*, а после нескольких следующих команд (или вообще безотносительно к операциям сложения/вычитания).

- ❑ Команды, устанавливающие флаг `a` по результату выполнения, — все команды из группы команд сложения/вычитания: `aaa`, `aas`, `daa`, `das`, `add`, `adc`, `sub`, `sbb`, `cmp`, `scas`, `cmps`, `dec`, `inc`.
- ❑ Команды, устанавливающие флаг `a` в неопределенное состояние, — это все команды типа умножения/деления: `div`, `idiv`, `imul`, `mul`, `aad`, `aam`.
- ❑ Команды пересылки, команды, связанные с передачей управления, а также команда `not` никак не воздействуют на флаги.
- ❑ Сдвиги и вращения на флаг `a` не влияют (как ни странно).
- ❑ Инструкция `neg` устанавливает флаги `a` и `c` одинаково: если исходное значение ноль, то `c = a = 0`, иначе `c = a = 1`.
- ❑ Логические инструкции `xor`, `or`, `and`, `test` сбрасывают все три флага переноса: `c = o = a = 0`.

Примечание

В документации Intel утверждается, что значение `a` в результате выполнения команды `neg` и логических инструкций не определено. Ничего подобного! Воздействие на флаг `a` проверено на i8086/286/386/486, Pentium-I и Pentium-II.

Имеется возможность сохранить значение флага `a`:

- ❑ в регистре `ah` при помощи инструкции `lahf` (вместе с флагами `s`, `z`, `p`, `c`);
- ❑ в стеке при помощи инструкции `pushf` (вместе со всеми флагами).

Затем сохраненное значение может быть восстановлено:

- ❑ из регистра `ah` инструкцией `sahf`;
- ❑ из стека инструкцией `popf`.

14.6. Практикум

Вызовите отладчик и в режиме непосредственного выполнения запишите в `al` число 9. Затем увеличьте его на 1 инструкцией `inc` и выполните `aaa`, наблюдая за изменением флагов `a`, `c`. Вновь запишите в `al` число 9, увеличьте `al` на восемь инструкцией `add` и выполните `aaa`.

Выполните трансляцию примера из файла `ex1.8` и пройдите его по шагам. Обратите внимание на изменения флага `c`. Дополните `ex1.8` так, чтобы результат был записан в ASCII-кодах.

Исследуйте в отладчике пример `ex2.8`, обратите внимание на изменения значений в регистрах `al`, `ah`.

Проверьте работу инструкций `aad` и `aam`. Запишите в `ax` число 8 `by` 7 и выполните `aad`. По этой инструкции $al = ah * 10 + al$, `ah = 0`. Выполните обратную команду — `aam`.

Проверьте в отладчике работу программы `ex3.8` — сначала с теми значениями, которые заданы в `ex3.8`, а затем со значениями $x = 35$, $y = 67$. Исправьте программу так, чтобы в `z` получился правильный результат 118.

Самостоятельно исследуйте инструкции `aam/aad` со следующими значениями параметра — 16, 1 и 0ff.

Прежде чем перейти к исследованию операций над упакованными BCD, проверьте пример упаковки BCD (исходный текст в `ex4.8`). Напишите программу для выполнения обратной процедуры, т. е. для преобразования `packed`→`unpacked` BCD.

Перейдем к исследованию операций с упакованными BCD.

Запишите в `al` число 059, и прибавьте 012. Выполните `daa`, наблюдая за изменением флагов `a`, `c` и регистра `ah`. Повторите опыт со значениями двадцать девять и семьдесят девять.

Измените программу из `ex4.8` так, чтобы можно было выполнить преобразование `packed` BCD→`binary`, за счет совместного применения инструкций:

```
aam    16                ; packed -> unpacked
aad                    ; unpacked -> binary
```

Исследуйте в отладчике пример `ex5.8`. Обратите внимание, что данные отображаются от младших цифр к старшим, а в исходном тексте они заданы наоборот, начиная со старших цифр.

В программе `p_bcd` удвоение 10-байтного BCD выполняется двумя способами — за счет команд `add`, `aam` в цикле, либо средствами FPU. Исследуйте `p_bcd` самостоятельно, напишите по аналогии программу утроения 10-байтного BCD.

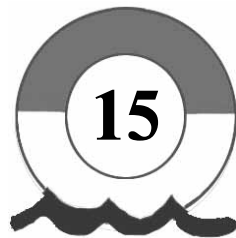
Напишите программу, которая вычисляет показания часов на момент, следующий через 9 мин 59 сек после запуска программы, а также на момент 2689 секунд тому назад. Текущее время прочитайте из часов реального времени *RTC* (Real Time Clock) при помощи макрокоманды `rd_tm` из файла `rtc.inc`. Пример вызова `rd_tm`:

```
rd_tm ch, cl, dh
```

Первый параметр вызова — часы, второй — минуты, третий — секунды. В примере значение секунд будет записано в `dh`, минуты сохранятся в `cl`, а часы — в `ch`.

Формат данных часов, минут и секунд в *RTC* — `packed` BCD. Заданный интервал представьте в любом формате — `packed` BCD или `binary`; важно, чтобы результат получился в формате упакованных BCD, и с учетом шкалы времени (например, результат 23:59:60 или 23:99:99 не очень хорош).

Результат вычислений проконтролируйте макрокомандой `show_tm`. Пример вызовов `rd_tm` и `show_tm` — в файле `rtc.8`.



Математический сопроцессор

- Это совсем не то, что маленькие 10-амперные аппараты для невротиков. Это стационарная 25-амперная машина с большим запасом прочности, предназначенная для действительно тяжелых, застарелых случаев.
- Как раз то, что у меня,— сказал Кэвел с простительной гордостью.

Роберт Шекли. "Терапия"

В этой главе последовательно рассмотрены различные вопросы, связанные с программированием математического сопроцессора. Изложение начинается с исследования в `d86` команд ввода/вывода данных и форматов данных; попутно рассматриваются принципы обработки данных с использованием "классических" стековых команд. Затем подробно рассматривается система команд.

В последующем изложении мы ориентируемся на сопроцессор `i80387`. Выбор `i80387` объясняется следующими соображениями:

- ❑ `i80387` совместим с последующими модификациями, отличается от них незначительно;
- ❑ программная модель `i80387` по сравнению с первым образцом сопроцессора (`i8087`) немного проще: отдельные команды управления сокращены без потери функциональности;
- ❑ в `i8087` допущена ошибка, следствием которой является необходимость в программной синхронизации процессора `i80x86` и сопроцессора; в `i80387` синхронизация полностью аппаратная.

Для обозначения сопроцессора используется распространенная аббревиатура `FPU` — `Floating Point Unit` (Intel наряду с этим названием применяет также "`Numeric Processor eXtension`", или `NPX`).

15.1. Проверка наличия FPU

Для выполнения практических работ в вашей системе должен быть установлен сопроцессор или его программный эмулятор.

Вызовите `d86` и несколько раз нажмите клавишу `<F10>`. После нескольких нажатий в правом верхнем углу экрана должно появиться окно следующего вида:

```
0: ----
1: ----
2: ----
3: ----
4: ----
5: ----
6: ----
7: ----
```

```
Proj infinity      Tags -----      c
80 precision      ST = 0              + NaN
round near
IP: E6D9:1E42     masked:      puozdi
OP: 0000:0000     occurred:
```

Если такое окно не появилось, FPU отсутствует. В этом случае остается только имитировать FPU (хотя бы `i8087`) при помощи программного эмулятора, например:

```
em87.com /L
```

Примечание

При использовании `em87` пошаговая отладка работает со сбоями: за один шаг выполняется несколько команд FPU.

Вызовите `d86` и откройте окно FPU. Если регистры 0—7 не пустые, выполните инструкцию инициализации — `finit`.

15.2. Загрузка и выгрузка целых чисел

Обмен данными между процессорами `i80x86` и `i80x87` выполняется через системную память (внутренние регистры этих процессоров взаимно недоступны).

Запишите в слово по адресу `0100` число 7. Отобразите значения четырех слов с адреса `0100` (в одной строке), затем выполните команды:

```
fild    w[0100]
fsqrt
fist    w[0102]
```

Первая команда — `fild` (Fpu Integer Load) — загружает целое число в регистр FPU номер 0. При загрузке FPU преобразует входное значение из формата знакового целого (16-разрядного) в формат с плавающей точкой.

Следующая команда — `fsqrt` (Fpu Square Root) — извлекает квадратный корень из числа в регистре 0. Результат помещается на место исходного значения — в регистр 0.

Команда `fist` (Fpu Integer STore) копирует содержимое регистра 0 в ячейку памяти по адресу 0100; при этом выполняется обратное преобразование — из внутреннего формата FPU в формат знакового двоичного целого с округлением. Текущий режим округления — к ближайшему значению (`round near`) — установлен в результате выполнения `finit`.

Потеря точности при выполнении операций фиксируется установкой флага `p` сопроцессора: справа от надписи "occurred:" появилась буква `p`. Признак `p` останется установленным, независимо от точности дальнейших вычислений.

Примечание

Регистр состояния с признаками *особых ситуаций* доступен для чтения, работа с ним обсуждается позже. Признаки особых ситуаций обнуляются при полном сбросе по команде `finit` или специальной инструкцией `fclex` (Fpu Clear Exceptions).

Рассмотренный пример демонстрирует обмен данными с FPU в формате целых знаковых 16-разрядных слов. Выполним аналогичный опыт с 32- и 64-разрядными целыми знаковыми значениями.

Запишите в двойное слово по адресу 0100 значение -255 , двумя командами: -255 — в младшее слово, -1 — в старшее. Затем выполните последовательность:

```
finit
fild    d[0100]
fchs
fist    d[0100]
```

Команда `fchs` (Fpu CHange Sign) инвертирует значение в регистре 0.

При считывании данных из регистра 0 в память, данные в регистре 0 сохраняются. Имеется возможность дополнительно задать очистку регистра 0 — буквой `p` (Pop) в конце мнемоники команды:

```
fistp   q[0100]
```

В рассмотренных примерах размерность операций загрузки/выгрузки `fild/fist(p)` задана явно — при помощи уточнителей типа `w` (word), `d`

(dword) и q (qword). В следующем примере смысл операций остается тем же, но операнды заданы именами; размерность машинной команды выбирается ассемблером по типу имени операнда.

**Листинг 15.1. Задание размерности операций загрузки/выгрузки
(см. ex1.8)**

```

        jmp      start

int16   dw
int32   dd
int64   dq      0

start:
        finit
        mov     int16, 7          ; (1)
        fild    int16
        fsqrt                    ; (2)
        fist     int16+2          ; (3)
        fclex                     ; (4)
        mov     ax, -255
        cwd
        mov     w int32, ax
        mov     w int32+2, dx
        fild    int32
        fchs
        fist     int32
        fistp    int64

        int     020

```

Выполните начало примера по шагам — до команды (2). Выполните инструкцию (2) и очистите признаки особых ситуаций командой `fclex`. Признак `p` сброшен. Выполните команду (3). Признак `p` вновь установлен, поскольку при записи в `int16` потеряна дробная часть результата.

Мы рассмотрели загрузку и выгрузку данных целочисленного формата. Загрузка данных BCD-формата выполняется аналогично, с указанием в инструкциях буквы `b` вместо `i` (см. листинг 14.5):

```

fbld    t[0100]
...
fistp    t[0100]

```

15.3. Недопустимые операции и NaN

В предыдущих опытах нам встретились особая ситуация под названием "потеря точности". Эта ошибка при работе с вещественными числами наименее значительная.

В файле `ex1.8` в команде, следующей за `finit`, замените операнд `7` на `sp`. Пройдите по шагам от начала программы до команды `fclex`. В результате извлечения корня из отрицательного числа регистр `0` содержит специальное значение "не число" — NaN (Not a Number).

Выполните команду `fchs`. Результат операции над NaN — по-прежнему NaN, но с другим знаком. В признаках особых ситуаций установлен флаг `i` (Invalid operation).

Выполните `fclex`, а затем — команду `fistp int16+2`. Операция с NaN вновь привела к установке флага `i`.

Проведем еще один опыт с NaN. Выполните последовательность:

```
finit
fild    int16
```

Повторяйте (нажатиями клавиши `<F3>`) последнюю команду, пока все регистры FPU не будут заполнены. При следующем повторении в регистре `0` окажется NaN. В данной ситуации произошла ошибка при загрузке: сохранность данных нарушена.

15.4. Организация массива данных в виде стека

При загрузке нескольких значений отладчик отображает последнее значение в первой строке; ранее загруженные значения сдвигаются "вниз" — в регистры с большими номерами. Выполните команды:

```
fldz
fldl
fldpi
```

Это — команды загрузки констант, определенных в самом FPU,— нуля, единицы и числа π . В результате сдвига данных при загрузке последнее значение — π — оказалось в регистре `0`, а число `0.0`, загруженное первым, оказалось ниже всех — в регистре `2`. Выполните следующие команды, предварительно задав отображение трех слов с адреса `0100`:

```
fistp   w[0100]
fistp   w[0102]
fistp   w[0104]
```

Число π , загруженное последним, выгружается первым (по адресу 0100 записано число 3), а ноль, загруженный первым, оказался при выгрузке последним (записан по адресу 0104).

Как видите, ввод/вывод в сопроцессоре организован по принципу стека. Выполните команду `finit` и повторите команды загрузки, наблюдая за отображением регистра `ST` (`Stack Top`).

После `finit` в `st` — число 7. Это абсолютное значение индекса в массиве данных `FPU`, состоящем из восьми элементов. На практике точное значение `st` особого интереса не представляет. Важно то, что адресация элементов массива данных выполняется относительно `st`, и при этом номера регистров не абсолютные, а относительные. При загрузке данные остаются на своих местах, изменяется только значение `st`: уменьшается на единицу по кольцу (т. е. за `st = 7` следует `st = 0`). В результате номера регистров, поскольку они относительные, увеличиваются также по кольцу: то, что было на вершине стека (в регистре 0), оказывается в регистре 1 и т. д. При выгрузке командой `fistp` значение `st` увеличивается, а номера регистров, соответственно, уменьшаются.

В `FPU` предусмотрены команды для изменения `st` без загрузки/выгрузки данных. Выполните:

```
fdecstp
```

Значение `st` уменьшилось, номера регистров в результате увеличились, причем на вершине стека значение не определено, т. к. регистр 0 теперь свободен. Выполните два раза команду:

```
fincstp
```

Значение в `st` увеличивается, а номера регистров уменьшаются; данные как бы сдвигаются вверх по кольцу.

Выполните команду:

```
ffree 7
```

Команда `ffree` помечает заданный регистр как незанятый. При этом данные в соответствующем элементе массива не стираются; признак "свободен" фиксируется в отдельном слове (по два бита на каждый из восьми регистров данных). Содержимое этого массива признаков, или *тегов*, отображается в `d86` над `ST` в символическом виде:

- ☐ прочерк означает "свободен";
- ☐ `f` (Float) — занят значением в формате с плавающей точкой;
- ☐ `z` (Zero) — занят нулевым значением;
- ☐ `i` (Invalid) — занят специальным значением, например, NaN.

Информация о тегах, так же как сведения о текущем значении `st` в большинстве практических приложений не нужны. Вместе с тем, команды `fincstp/fdecstp` могут представлять интерес при обработке массивов действительных чисел. Любая инструкция FPU обращается к вершине стека — к регистру 0, и за счет изменения `st` можно пропустить через регистр 0 все элементы стека.

15.5. Вычисления в стековой машине

Математический сопроцессор представляет собой стековую вычислительную машину с некоторыми расширениями, на которых мы остановимся позднее.

В стековой архитектуре операции загрузки и выгрузки, а также одноместные вычислительные операции (инвертирование, извлечение корня и т. д.) выполняются над значением на вершине стека. При загрузке и выгрузке содержимое стека сдвигается; при выполнении одноместной операции результат записывается на место операнда — на вершину стека, без сдвига содержимого стека. Двуместные операции (сложение, вычитание, умножение и деление) выполняются над парой элементов на вершине стека; при этом операнды удаляются из стека, а результат загружается в стек.

Операнды и результат в этих операциях адресуются неявно. Операции, выполняемые по описанной схеме, в дальнейшем называются *классическими* стековыми операциями.

15.5.1. Двуместные операции

Выполните сброс, загрузите в FPU пару значений и вызовите команду сложения `fadd`. На месте двух значений в регистрах 0 и 1 теперь находится результат сложения — в регистре 0. Выполните умножение `fmul`. Результат — NaN, поскольку второй операнд двуместной операции не определен.

Выясним, какой из регистров 0 или 1 является источником, а какой приемником.

```
fldpi      ; π
fldl       ; 1      π
fdiv       ; π/1
```

Результат деления — π ; следовательно, делитель находится в регистре 0, а делимое — в регистре 1. Таким образом, приемник двуместной операции — это вершина стека, а источник — следующий элемент стека. В общем случае, операция `<op>` выполняется по схеме: $1 \text{ <op> } 0 \rightarrow 0$.

Рассмотрим пример вычисления сложного выражения в стековой машине. Требуется найти разность площадей двух окружностей радиусом a и b . То есть вычислить выражение: $(a \times a - b \times b) \times \pi$.

Листинг 15.2. Вычисление выражения классическими стековыми операциями (см. ex2.8)

```

        jmp      start

_a      dw      8
_b      dw      12
diff    dw      ?

start:
        finit
        fild     _a      ; _a
        fild     _a      ; _a      _a
        fmul     ; a2
        fild     _b      ; _b      a2
        fild     _b      ; _b      _b      a2
        fmul     ; b2      a2
        fsub     ; a2-b2
        fldpi
        fmul
        fistp    diff
        int      020

```

В комментариях показаны значения регистров 0—2 в результате выполнения операций. Конечный результат при заданных значениях *a* и *b* должен получиться отрицательным.

15.5.2. Выражения в обратной польской записи

Программа из листинга 15.2 правильная, но способ ее составления может показаться непонятным. Последовательность и чередование операций загрузки и двуместных вычислительных операций становятся очевидны, если записать исходное выражение в *обратной польской*, или *постфиксной* нотации, при которой знак операции следует за операндами.

Пример преобразования в обратную польскую запись (фрагменты в постфиксной нотации ограничены квадратными скобками):

$((a * a) - (b * b)) * \pi$

$([aa *] - [bb *]) * \pi$

$[aa * bb * -] * \pi$

$[aa * bb * - \pi *]$

Обратная польская запись в точности описывает последовательность операций загрузки и двуместных арифметических действий в стековой машине.

Буква или цифра означает загрузку, а знакам арифметических операций соответствуют команды `fmul`, `fdiv`, `fsub`, `fadd`.

Для автоматического преобразования выражения в польскую запись запустите программу `postfix.exe`. После приглашения `'>'` введите исходную формулу без пробелов, с однобуквенными именами или числами, состоящими из одной цифры, например:

```
(a*a-b*b)*p
```

Завершение ввода по нажатию клавиши `<Enter>` подтверждается звуковым сигналом, после чего целиком выводится выражение в обратной польской записи. Программа ожидает нажатия любой клавиши, после чего очищает экран. Результат должен получиться следующим:

```
bb*aa*-p*
```

При помощи `postfix` напишите программу для вычисления сопротивления цепи при параллельном соединении сопротивлений `r1`, `r2`, `r3`. При значениях 19, 23 и 28 должно получиться 8, после округления до целого. Для загрузки единицы пользуйтесь командой `fld1`.

После проверки исходного варианта задайте `r1 = 0` и выполните программу по шагам. После деления на ноль флаг сопроцессора `z = 1` (by Zero division), а в регистре 0 — значение `+Infinity` (плюс бесконечность). При последующем сложении конечных значений с `+Infinity` результат по-прежнему `+Infinity`, но после итогового деления `1.0/+Infinity` получается правильное значение — 0.0.

Выясните, каков результат чтения `+Infinity` в двухбайтовое слово.

Составьте программы для вычисления выражений:

1. $(x - (12 + y))/100 + 2$, при $x = -125897$, $y = 084512$
2. $(1/2 + (1/3 + 1/4) \times 7)/17$
3. $1/(1+s/(1+s/(1+s)))$, где $s = \sqrt{2}$

Результаты для контроля:

1. $-2.102210e3$
2. 0.26960784314
3. 0.52859547919

15.5.3. Расширения стековой машины в i80x87

Расширения классической стековой машины в `i80x87`, в основном, следующие:

- некоторые одноместные операции дают два результата — основной и побочный, в регистрах 0 и 1;

- ❑ при выполнении некоторых двуместных операций один из операндов остается на прежнем месте;
- ❑ во многих двуместных операциях предусмотрена возможность задания одного операнда в системной памяти;
- ❑ многие двуместные операции, в частности операции загрузки и выгрузки, а также операции базовой арифметики позволяют адресовать данные в глубине стека;
- ❑ введены операции с обратным порядком операндов по схеме:
 $0 <op> 1 \rightarrow 0$.

Теперь перейдем к рассмотрению формата вещественных чисел с плавающей точкой; начнем с внутреннего представления данных в FPU.

15.6. Представление данных в FPU

Число в формате с плавающей точкой занимает регистр FPU размером 80 бит. Данные такого *расширенного* формата передаются между FPU и системной памятью без преобразований, т. е. без потери точности.

Пример:

```

...
ext    dt      1.00001e-387
...
fld     ext
...
fstp    ext
...
```

Вызовите в отладчике программу `inner`. Эта программа предоставляет определения макрокоманд для наблюдения за данными FPU по разрядам. Выполните в непосредственном режиме `d86` макрокоманду `what`:

```
what 1.625
```

По этой макрокоманде в FPU загружается указанное значение, которое затем копируется в память в расширенном формате — в переменную `extend`. По значению разрядов этой переменной формируется ASCII-строка с литерами '0' и '1'. Битовые поля, образующие число в формате FPU, отображаются `d86` в виде фрагментов полученной ASCII-строки.

```

r1,sign,, 0
r15,exp,, 0111111111111111
r32,sgnd,, 11010000000000000000000000000000
"013E    00000000000000000000000000000000
```

Это — изображение расширенного вещественного числа в отладчике. Формат расширенного вещественного числа показан на рис. 15.1 (треугольником отмечено положение двоичной точки).

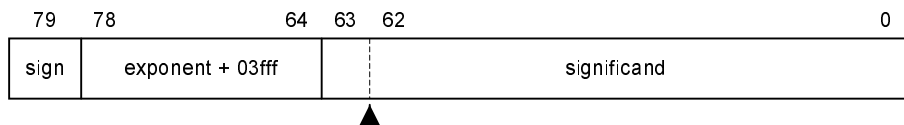


Рис. 15.1. Формат расширенного вещественного числа

Прочитаем отображение, полученное в отладчике. Разряд `sign` — это знак мантиссы. Ноль в поле `sign` означает, что число неотрицательное. За `sign` следуют 15 битов поля `exp`. После них — 64 бита (восемь байтов) абсолютного значения мантиссы — `sgnd` (Significand).

Принято, что двоичная точка находится всегда справа от старшего бита мантиссы. С учетом этого, мантисса в примере равна 1.101.

Соответствующее десятичное значение:

$$1 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3},$$

что означает $1 + 1/2 + 1/8$, или $1 + 0.5 + 0.125 = 1.625$

В поле `exp` задано значение порядка, увеличенное на 03fff. В примере `exp` содержит как раз значение 011 1111 1111 1111 (03fff), так что порядок равен нулю. В итоге, показанное в отладчике число равно $+1.625 \times 2^0$.

Выясним, как закодировано вдвое меньшее число:

```
what      0.8125
```

В поле `exp` — 011 1111 1111 1110, т. е. порядок стал -1 . При этом значения `sign` и `sgnd` не изменились. Таким образом, заданное значение равно $+1.625 \times 2^{-1} = 1.625/2$.

Выясните представление чисел 3.25 и -1.625 .

Выполним обратное действие — закодируем по битам число в расширенном формате с плавающей точкой, при поддержке макрокоманды `bits`. Первым параметром `bits` следует — обязательно (!) — задать `exp`, а в последующих параметрах — произвольной длины список с номерами битов, значение которых следует установить в единицу (остальные биты обнулятся). Пример кодирования числа 1.0:

```
bits      03fff, 63
```

Закодируйте числа 2.0 и 0.5.

Исследуем границы диапазона вещественных чисел. Рассмотрим, какие значения получаются при минимальном порядке (`exp` = 0).

Выполните:

```
bits    0, 62
bits    0, 61
bits    0, 0
```

Результаты следующие:

```
Den\1  +1.68105157155604675 E-4932
Den\2  +8.40525785778023374 E-4933
Den\63  +3.64519953188247459 E-4951
```

Den означает, что число денормализовано, т. е. его мантисса хранится не в виде 1.mm..mm, а в виде 0.mm..mm. Число после Den\ показывает, насколько далеко вправо сместился первый значащий разряд.

Пока значение не слишком мало, FPU хранит его в нормализованном виде: бит 63 остается равным 1. При уменьшении нормализованного числа вдвое, значение `exr` уменьшается на 1, а `sgnd` остается прежнем. Когда `exr` = 0, дальнейшее его уменьшение невозможно, и число может быть уменьшено только за счет сдвига `sgnd` вправо; при этом старший бит `sgnd` сбрасывается. Произошла *денормализация*. Денормализация позволяет расширить диапазон чисел при приближении к нулю за счет снижения точности (число значащих разрядов `sgnd` уменьшается).

Ноль в бите 63 допустим только в том случае, если `exr` = 0. При загрузке числа извне (из системной памяти) больше возможностей для задания недопустимых значений: "ненормальных" и псевдоденормализованных.

Например, при попытке денормализовать нормализованное число `bits 1, 63`, задав его в виде `bits 2, 62`, получится правильное значение; тем не менее, `d86` отмечает его как `Unn\` (`Unnormal`). Также, при попытке представить нормализованное значение `bits 1, 62` в виде `bits 0, 63`, результат вовсе неправильный (обратите внимание на счетчик после Den\). Такое число с `exr` = 0 и единицей в бите 63 называется псевдоденормализованным.

Вычисления с ненормальными и псевдо-денормализованными значениями дают NaN-результат. В массиве `тегов` все виды ненормализованных значений отмечены буквой `i` (`Illegal`). Ноль отмечается в `тегах` как `z` (`Zero`), хотя, по сути, — это тоже денормализованное число (`exr` = 0) с `sgnd` = 0.

Рассмотрим теперь специальные значения на противоположной, верхней границе диапазона.

```
bits 07fff, 63
bits 07fff, 62
```

Первое значение `+Infinity`, второе — `NaN`. Очевидно, `exr` = 07fff зарезервировано для представления специальных величин. Вот пример числа, близкого к максимуму — с `exr` = 07ffe:

```
bits 07ffe, 63
```

Выясняется, что величина `exr` для нормализованных значений находится в диапазоне `1—07ffe`. Соответственно, значение порядка ограничено диапазоном `−03ffe—+03ffe`. Нулевое значение `exr` зарезервировано для задания близких к нулю денормализованных чисел, а `exr = 07fff` представляет бесконечность.

Проверьте выполнение арифметических действий над денормализованными числами. После вычислений рекомендуем обновлять ASCII-отображение, вызывая `what` без параметров (показывает значение регистра 0).

Выясните, что представляет собой по разрядам значение NaN.

15.7. Стандартный формат вещественных данных

Внутреннее представление данных в `i80x87` отличается от стандарта IEEE 754. В стандарте IEEE 754 допускаются только нормализованные числа (ноль — особый случай, и отмечается в тегах специальным образом). Единица перед двоичной точкой становится избыточной, она теперь не задается, а подразумевается. Поле `sgnd` теперь содержит только биты после двоичной точки.

По стандарту IEEE 754 данные в формате с плавающей точкой могут быть представлены как 32-битные (однократная точность) или как 64-битные (удвоенная точность).

Примечание

В языке C этим размерностям соответствуют типы `float` и `double` (в Turbo Pascal дополнительно введен тип `extended`, соответствующий внутреннему 10-байтовому представлению данных в `i80x87`).

На рис. 15.2 показано двоичное представление 32- и 64-битных данных в формате с плавающей точкой (треугольником показано положение двоичной точки).

Данные этих типов в программе определяются при помощи директив `dd` и `dq`. Для загрузки в FPU и для копирования из FPU используются инструкции `fld`, `fst(p)`. Чтение из FPU в память сопровождается форматными преобразованиями: данные приводятся к диапазону 32- или 64-разрядных, с округлением.

Проведем опыты с 32-разрядным представлением чисел с плавающей точкой. Запустите в отладчике программу `float.com` и выполните:

```
what 1.625
```

Теперь в `sgnd` вместо 1101 записано значение 101, т. к. старшая единица (та, что в расширенном формате стоит перед "точкой") задана неявно. Выполните:

```
what 1.0e-40
```

Результат выглядит, как денормализованное число в расширенном формате. Но здесь все числа нормализованные, поэтому нулевое значение `exp` задает порядок $-07f$.

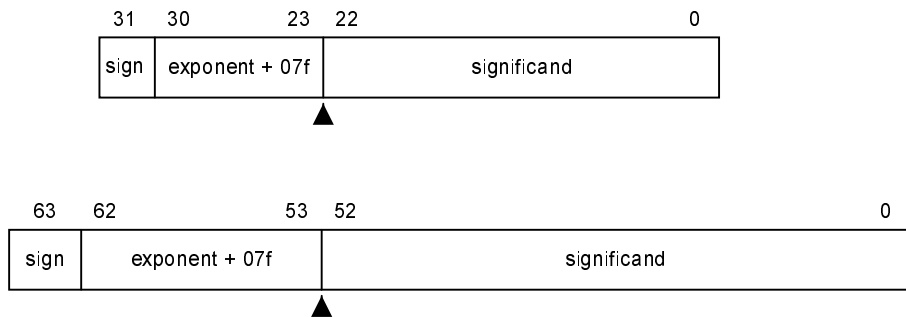


Рис. 15.2. Формат 32- и 64-битных вещественных данных

Одна из особенностей стандартного формата — в том, что ноль может быть задан только как специальное значение. Поскольку значение мантиссы равно $1.\langle \text{sgnd} \rangle$, то даже при `sgnd` = 0 значение мантиссы остается ненулевым (1.0). Признаком нуля является сочетание `exp` = `sgnd` = 0. При выполнении `fld` число с `exp` = `sgnd` = 0 записывается в FPU как 0.0.

Максимальное значение `exp` по-прежнему зарезервировано для представления бесконечности. Выполните:

```
what    1.0e40
fld     float
```

В поле `exp` все биты установлены в 1 (бесконечность). Результат запроса `what` сохранился в переменной `float` (тип `dword`), и при загрузке `float` бесконечность преобразуется из 32-битного формата в 80-битный.

Перепишите задание, выбранное при изучении первой части книги, используя вместо целых чисел данные 32- и/или 64-битного формата с плавающей точкой. Исходные данные и результат отображайте в `d86` при помощи спецификаций `fd/fq/ft`.

15.8. Программная модель i80x87

Представим результаты проведенного нами исследования FPU в виде программной модели.

Программно доступная часть сопроцессора i80x87 включает в себя:

- ☐ массив данных из восьми десятибайтных регистров;
- ☐ 16-разрядный регистр тегов;

- ❑ 16-разрядное управляющее слово;
- ❑ 16-разрядное слово состояния;
- ❑ 48-разрядные указатель инструкции и указатель операнда.

На рис. 15.3 слово тегов (Tag Word) показано целиком внизу слева, а также — в виде массива двухбитных полей Tag Fields — справа от Data Array.

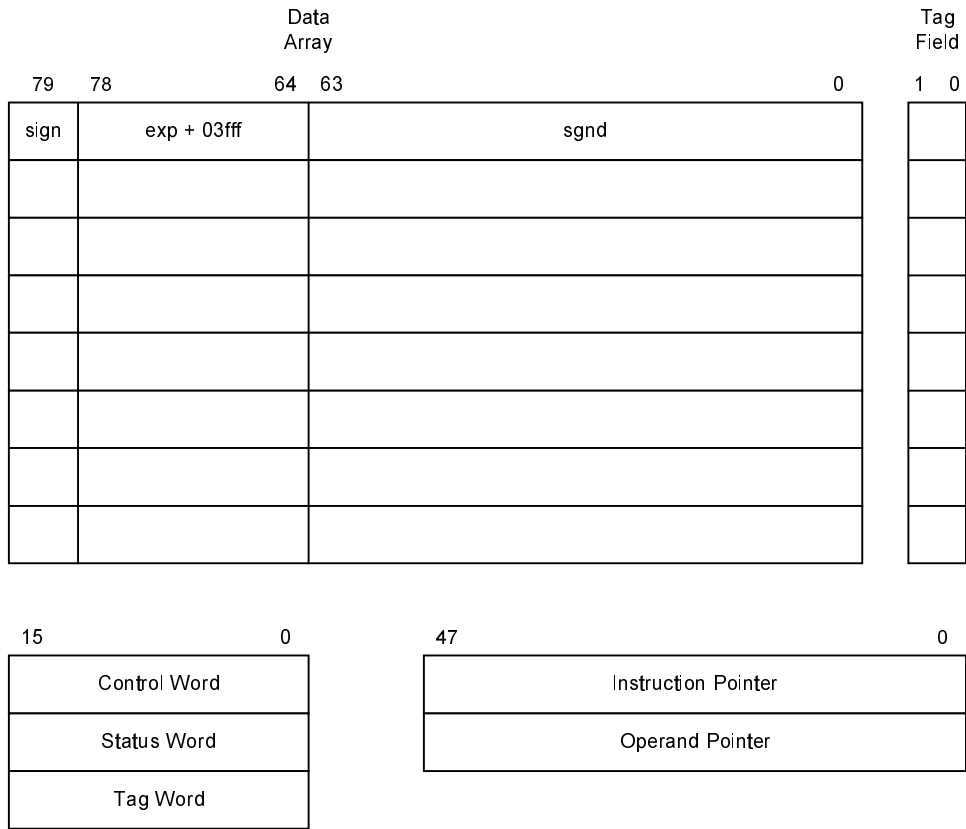


Рис. 15.3. Регистры FPU i80x87

Массив данных Data Array содержит восемь элементов по 80 бит, в расширенном формате с плавающей точкой: в битах 78—64 задан смещенный (увеличенный на 03fff) порядок, в младших 64 битах — мантисса, бит 79 — знак мантиссы.

На рис. 15.4 показаны слово состояния sw (Status Word) и управляющее слово cw (Control Word).

Обозначения шести младших разрядов (p,u,o,z,d,i) в sw и cw совпадают. В sw — это обозначения флагов особых ситуаций: p — потеря точности,

i — неправильная операция, и т. д. В sw — это обозначения флагов запрета аппаратных прерываний при возникновении одноименных особых ситуациях.

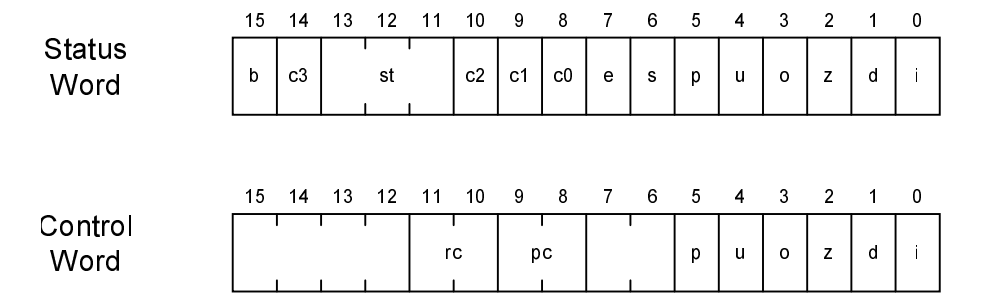


Рис. 15.4. Слово состояния и управляющее слово

15.8.1. Флаги особых ситуаций

Единица во флаге p,u,o,z,d,i слова состояния sw устанавливается аппаратно при возникновении особой ситуации. Флаги — сразу все — сбрасываются командами finit, fclex. Единица во флаге p,u,o,z,d,i управляющего слова sw запрещает прерывание по одноименной особой ситуации; для разрешения прерывания соответствующий флаг в sw должен быть программно сброшен.

Одноименные флаги sw и sw обозначают следующие особые ситуации:

- ☐ i — Invalid operation (недопустимая операция);
- ☐ d — Denormalized operand (денормализованный операнд);
- ☐ z — Zero divide (деление на ноль);
- ☐ o — Overflow (переполнение, результат представлен в виде бесконечности);
- ☐ u — Underflow (результат настолько мал, что может быть представлен только в виде денормализованного числа);
- ☐ p — Precision lost (потеря точности).

После выполнения finit в sw установлены все биты запрета прерываний. После finit или fclex флаги особых ситуаций в sw сброшены.

15.8.2. Битовые поля управляющего слова

Значение в битовом поле rc (Rounding Control) задает режим округления, как показано в табл. 15.1. Значение rc устанавливается программно; изначально (и после finit) равно 00. Константы для установки rc определены в файле fpu.inc.

Таблица 15.1. Режимы округления в зависимости от *gc*

Код <i>gc</i>	Отображение в <i>d86</i>	Режим округления
00	round near	К ближайшему, а если значение посередине, то к четному
01	round $-\text{inf}$	Вниз, в сторону $-\text{Infinity}$
10	round $+\text{inf}$	Вверх, в сторону $+\text{Infinity}$
11	round zero	Отбрасывание лишних разрядов, в результате, уменьшение абсолютного значения

Битовое поле *pc* (Precision Control) управляющего слова *sw* не рассматривается — изменение точности не влияет на скорость вычислений. Поле *pc* было введено для эмуляции сопроцессоров, предшествующих *i8087*.

15.8.3. Битовые поля слова состояния

Рассмотрим битовые поля слова *sw*, помимо уже известных нам флагов особых ситуаций *p, u, o, z, d, i*.

- ❑ Флаг *e* (Error summary) в *sw* устанавливается в 1, если установлен хотя бы один из флагов особых ситуаций *p, u, o, z, d, i*.
- ❑ Флаг *s* (Stack Fault) устанавливается в 1, если ошибка выполнения операции вызвана переполнением или антипереполнением стека данных; флаг *s* позволяет уточнить причину ошибки при установке флага *i*.
- ❑ Флаг *b* (Busy) оставлен для совместимости с *i8087*; в *i80387* флаг *b* дублирует флаг *e* (Error summary).
- ❑ Поле *st* (Stack Top) — трехбитный "регистр" указателя стека. Начальное значение 7, после первой операции загрузки *st* становится равным 0.
- ❑ Флаги *c0...c3* — флаги состояния, отражают результат выполнения операции в сопроцессоре (подобно *flags* центрального процессора).

Рассмотрим флаги *c0...c3* подробнее. Значение сочетаний этих флагов показано в табл. 15.2.

Таблица 15.2. Установка флагов *c3, c2, c0* в результате сравнений

<i>c3</i>	<i>c2</i>	<i>c0</i>	Значение
1	1	1	Числа несопоставимы
0	0	0	>
0	0	1	<
1	0	0	=

В отличие от флагов центрального процессора, флаги `c0..c3` отражают, в основном, только результат операций сравнения. Вычислительные операции изменяют `c0..c3` непредсказуемо и бессмысленно — даже вычитание.

Анализ результатов сравнения выполняется за счет чтения слова состояния `sw` в системную память или в регистр `ax`. Затем состояние FPU из `ax` передается в `flags` командой `sahf`:

```
rd_flags macro
    fstsw    ax        ; Fpu STore Status Word
    sahf     ; ah -> flags
#em
```

Флаги сопроцессора `c3`, `c2`, `c0` отображаются, в результате, во флаги `z`, `e`, `c` центрального процессора; соответствие между флагами процессоров отражено в табл. 15.3. Флаг `c` центрального процессора определяет условие `b` (Below) и его производные, флаг `z` — условие Zero, флаг `e` (parity Even) означает ошибку, когда числа несопоставимы. В d86 флаги `c3`, `c2`, `c0` отображаются в правом нижнем углу окна FPU — названиями соответствующих флагов CPU (т. е. какие значения примут флаги CPU, если мы выполним сейчас `rd_flags`)

Таблица 15.3. Соответствие между `sw` (FPU) и `flags` (CPU)

sw	flags	Отображение в окне FPU
c0	c	c
c1	-	.
c2	e	u (Unordered)
c3	z	z

Флаг `c1` в сравнениях не участвует; при ошибке стека (`s = i = 1`) он позволяет отличить переполнение (`c1 = 1`) от антипереполнения (`c1 = 0`). (Речь идет о переполнении стека, а не об арифметическом переполнении результата.)

Команда `fxam` проверяет значение на вершине стека и устанавливает `c3—c0` так, как показано в табл. 15.4. Флаг `c1` после выполнения `fxam` содержит копию бита `sign` регистра 0, даже если регистр свободен.

Таблица 15.4. Значения флагов `c3`, `c2`, `c0` после `fxam`

c3	c2	c0	Значение
0	0	0	Формат не поддерживается
0	0	1	NaN

Таблица 15.4 (окончание)

c3	c2	c0	Значение
0	1	0	Нормализованное число
0	1	1	Бесконечность
1	0	0	Ноль
1	0	1	Регистр свободен
1	1	0	Денормализованное число

В окне FPU отладчика под отображением флагов c3—c0 (в виде "c", ".", "u", "z") также показан вариант их интерпретации на случай выполнения команды `fexam`. Предполагаемый результат `fexam` выводится в виде одного из сообщений: `+Norm`, `−Norm`, `Empty`, `−Nan`, `+Nan`, `Unn (Unnormal)`, `−Inf`, `+Inf`.

15.8.4. Доступ к указателям инструкции и операнда

Значения указателей инструкции и операнда представляют интерес при обработке прерываний по особым ситуациям.

В начале выполнения новой операции в указателях `Instruction Pointer` и `Operand Pointer` сохраняются: адрес текущей инструкции FPU и адрес ее

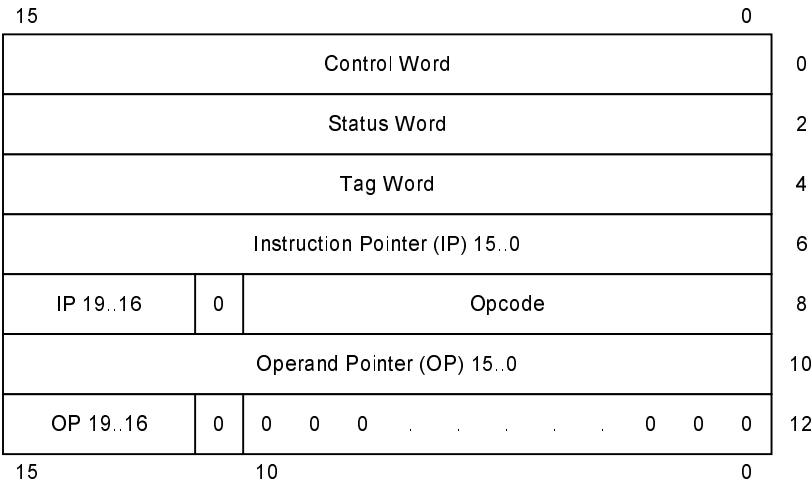


Рис. 15.5. Образ FPU в памяти при выполнении `fstenv/fsave` в реальном режиме i80x86

операнда в системной памяти (если задан). Запомненные значения могут быть прочитаны в память командами сохранения состояния FPU — `fstenv` (Fpu STore ENVironment) и `fsave`. Заполнение первых 28 байт образа FPU в памяти представлено на рис. 15.5.

Для хранения IP и OP предусмотрены 48-разрядные регистры — на случай работы в защищенном режиме. В реальном режиме IP и OP заданы в виде 20-разрядного линейного адреса. Указатели сохраняются в памяти так, что младшее слово содержит смещение, а старшее слово представляет значение сегмента (за вычетом поля Opcode — для IP).

15.9. Операции i80387

Операции FPU можно разбить на следующие группы:

- ☐ пересылки, в том числе загрузка констант;
- ☐ арифметические операции;
- ☐ сравнения;
- ☐ трансцендентные операции;
- ☐ управляющие операции.

15.9.1. Пересылки

Инструкции пересылок включают в себя:

- ☐ загрузку данных `f(i/b)ld`
- ☐ выгрузку данных `f(i/b)st(p)`
- ☐ обмен данных в стеке `fxch`.

Загрузка данных

Загрузка данных выполняется командой:

```
f(i/b)ld <src>
```

Источник `<src>` — это либо адрес переменной в памяти типа `w/d/q/t`, либо номер регистра FPU (число от 0 до 7). Команда с опцией `i` (`fild`), предназначенная для загрузки целочисленного значения, допускает только данные в памяти типа `w/d/q`. Команда с опцией `b` (`fbld`) для загрузки BCD-числа допускает только данные в памяти типа `t`. Команда без опции (`fld`) для загрузки действительного числа допускает данные в памяти типа `d/q/t`, а также номера регистров FPU.

Команда `fld` в `a86` позволяет в качестве `<src>` задавать также непосредственное значение с плавающей точкой, например, `fld 5.1`. Такое задание числа при загрузке транслируется следующим образом (имена `m1`, `m2` не создаются):

```

        cs fld  t >m1
        jmp     >m2
m1      dt      5.1
m2:

```

При отладке это очень удобно, но в работающей программе такие макрокоманды дают избыток кода и замедляют выполнение. Значение вещественной константы в работающей программе лучше определять отдельной переменной в памяти.

Примечание

Смысл `fld 2` и `fld 2.0` совершенно разный; `fld 2` задает загрузку значения из регистра 2, а `fld 2.0` — загрузку значения 2.0. В ассемблерах `tasm/masm/wasm` команды наподобие `fld 2` записывают в форме `fld st(2)`.

Продemonстрируем возможности загрузки значения из регистра данных, когда источник задан числом 0—7.

Листинг 15.3. Загрузка значений из регистров данных (см. fld.8)

```

fld      8.0      ; 8
fld      7.0      ; 7      8
           ;
fld      1        ; 8      7      8
           ;
fld      1        ; 7      8      7      8
           ; |
fld      0        ; 7      7      8      7      8
fld      5

```

При выполнении последней команды регистр 5 не содержит числа (свободен), что приводит к загрузке NaN.

Приемник во всех операциях загрузки — это бывший регистр 7, который после уменьшения `st` становится регистром 0. Источник всегда считывается перед уменьшением `st`, так что команда `fld 0` размножает значение регистра 0, а две команды `fld 1` дублируют пару значений на вершине стека.

Для доступа к константам, аппаратно встроенным в FPU, предусмотрены следующие инструкции, загружающие:

- ☐ ноль — `fldz`;
- ☐ единицу — `fld1`;
- ☐ число π — `fldpi`;
- ☐ $\log_2(10)$ — `fldl2t`;
- ☐ $\log_2(e)$ — `fldl2e`;
- ☐ $\log_{10}(2)$ — `fldlg2`;
- ☐ $\ln(2)$ — `fldln2`.

Примечание

В а86 имена `pi`, `l2t`, `l2e`, `lg2`, `ln2` определены отдельно — константами, зашитыми в `i80x87`; а86 позволяет задавать эти имена в качестве операндов `fld`, т. е. команда `fldpi` может быть записана в виде встроенной макрокоманды `fld pi`. Первый вариант экономичнее, второй вариант несколько выигрывает в наглядности.

Команды выгрузки

Выгрузка выполняется командой:

```
f(i/b)st(p) <dst>
```

Источник — всегда регистр 0. Приемник `<dst>` — это либо адрес переменной в памяти (типа `w/d/q/t`), либо номер регистра 0—7.

Опция `p` (`Pop`) задает освобождение регистра 0 с последующим увеличением `st`; в результате, значение "выталкивается" из стека. Выгрузка BCD-числа, а также выгрузка в память целого значения типа `q` выполняются только с опцией `p` (причина заключается в экономии кодов команд). Таким образом команд `fist q[0]` и `fbst t[0]` не существует; допустимые команды — `fistp q[0]` и `fbstp t[0]`.

Особенность выгрузки следующая: адрес `<dst>` вычисляется до увеличения `st`. По этой причине команды с `<dst> = 0—7` дают следующие интересные эффекты:

- ❑ Инструкция `fstp 0` задает выгрузку регистра 0 (фиксированный источник) в регистр 0 (приемник, заданный в команде), после чего вершина стека освобождается. В результате, команда `fstp 0` просто-напросто удаляет верхний элемент стека.
- ❑ Команда `fst i` копирует регистр 0 в регистр `i`, даже если он занят; остальные данные остаются на своих местах, т. к. `st` не изменяется.

Листинг 15.4. Копирование значений из регистров данных (см. фрагмент 1 из `fstp.8`)

```
finit
fldl          ; 1
fldz          ; 0      1
fst    3      ; 0      1      —      0
```

Содержимое регистра 0 по команде `fst 3` копируется в регистр 3.

Команда `fstp i` копирует регистр 0 в регистр `i`, затем освобождает регистр 0 и увеличивает `st`; возникает эффект "передачи" значений между регистрами 0—(`i`−1) по кольцу. Для иллюстрации приведем второй фрагмент из `fstp.8`.

**Листинг 15.5. Передача значений по кольцу командой `fstp`
(см. фрагмент 2 из `fstp.8`)**

```
finit
fld      5.0
fld      4.0
fld      3.0      ; 3      4      5
                ;
fstp     3        ;
                ; 3      4      5      3
                ; 4      5      3

fstp     3        ; 5      3      4
fstp     3        ; 3      4      5

fstp     0        ; 4      5
fucomp
```

Команда `fstp 3` сначала копирует число 3.0 из регистра 0 в регистр 3, затем освобождает регистр 0. Вращение выполняется влево по кольцу: число 5.0 перешло из регистра 2 в регистр 1, число 4.0 — из регистра 1 в регистр 0, а число 3.0 из регистра 0 попало в регистр 2. За три таких команды числа вернулись на исходные позиции.

Инструкция `fstp 0` выталкивает число из регистра 0; для очистки сразу двух регистров применен один из вариантов инструкции сравнения — с двойным `Pop`. Результат сравнения в данном случае интереса не представляет.

При выполнении `fstp i` сдвигаются, на самом деле, все регистры — не только 0—(i−1). При первом выполнении `fstp i` в регистр *i* записывается значение из регистра 0; при следующих повторях один за другим уничтожаются остальные значения, находящиеся первоначально в области i—8. Поэтому кольцевая передача данных между регистрами 0—(i−1) по команде `fstp i` практически применима лишь тогда, когда остальные регистры свободны.

В завершение остановимся на различиях между командами `fld i` и `fst(p) i`.

По команде `fld` число из регистра *i* вводится в стек — через регистр 0, со сдвигом остальных регистров вниз. Напротив, при выполнении команды `fst(p) i` значения либо сдвигаются вверх, либо остаются на прежних местах; при этом регистр 0 является источником, а регистр *i* — приемником.

Команда `fld` никогда не перезаписывает существующее значение (запись в занятый регистр — это ошибка, с результатом NaN и с установкой флага *i*). Напротив, команде `fst(p) i` позволено копировать регистр 0 в произвольный регистр *i*, независимо от состояния последнего.

**Листинг 15.6. Разница между командами fld 1 и fst 1
(см. фрагмент 3 из fstp.8)**

```
finit
fld      5.0
fld      3.0      ; (3      5)
fld      1      ; 5      3      5
fstp     0      ; (3      5)
fst      1      ; 3      3
```

Результаты fld 1 и fst 1 совершенно различные при одинаковых исходных данных.

Примечание

Команды загрузки/выгрузки записывают NaN, если не определено значение источника (т. е., когда свободен регистр 0 в момент выполнения fst (p), или свободен регистр i при выполнении fld i).

Команда обмена

Команда fxch i меняет местами значения в регистрах 0 и i. Команду fxch 1 на языке ассемблера допускается задавать без операндов — fxch; в машинных кодах это одно и то же. Команда fxch — одна из самых быстрых.

Примечание

В Pentium предусмотрено выполнение fxch в параллель с другими инструкциями FPU, причем *только* для этой команды.

Специальной команды для обмена местами регистров с произвольными номерами i и j, не равными нулю, не предусмотрено. Хотя бы один операнд должен быть на вершине стека. Такой обмен несложно выполнить в три команды.

Листинг 15.7. Обмен местами значений в регистрах i и j (см. fxch.8)

```
_fxch macro
    fxch    #1
    fxch    #2
    fxch    #1

#em

fld      4.0
fld      3.0
fld      2.0
fldl1    ; 1      2      3      4
_fxch    1, 3 ; 1      4      3      2
```

15.9.2. Арифметические операции

Арифметические операции в i80x87 — это четыре действия арифметики и несколько дополнительных команд на их основе. К группе арифметических операций также относят извлечение квадратного корня.

Основные арифметические операции

Обозначения операций FPU похожи на названия операций центрального процессора (add, sub, mul, div); только добавлена буква f в начале (Fpu).

Среди арифметических операций имеются команды с *реверсированием* — это subr и divr. Эти команды считывают операнды в обратном порядке: приемник меняется с источником; результат по-прежнему записывается в *приемник*. Буква p (Pop) в конце задает удаление значения из регистра 0, она может быть добавлена, только если приемник — не регистр 0.

Варианты задания арифметических команд показаны в табл. 15.5, где <op> — операция из множества add, sub, mul, div.

Таблица 15.5. Форматы арифметических инструкций

Формат команды	Действие	Примеры
f<op>	0 ← 1 <op> 0	
f<op>p	Вершина стека формируется по классической схеме	fadd
f<op> i	0 ← 1 <op> 0	fadd 0
f<op> 0, i		fadd 1
		fadd 0, 7
f<op> i, 0	i ← i <op> 0	fdiv 2, 0
f<op>p i, 0	i ← i <op> 0	
	Затем регистр 0 выталкивается из стека	fsubp 1, 0
f<op> mem	0 ← 0 <op> mem	fadd d[0200]
fi<op> mem	0 ← 0 <op> int(mem)	fiadd d[0180]

Листинг 15.8. Пример задания арифметических команд (см. фрагмент 1 из take4.8)

```
fld      8.0
fld      2.0      ; 2      8
```

```

fdiv          ; 4
fldl1         ; 1          4
fdivr         ; 0.25
fadd 0        ; 0.5
fldl1         ; 1          0.5
fadd 1, 0     ; 1          1.5
fld 3.0       ; 3          1          1.5
fsub 2, 0     ; 3          1          -1.5
fdivp 2, 0    ; 1          -0.5

```

Команда `fdiv` выполняет операцию $1/0 \rightarrow 0$; операнды удаляются из стека, затем загружается результат.

Команда `fdivr` аналогична, но меняет местами входные значения ($0/1 \rightarrow 0$).

Команда `fadd 0` выполняется по схеме $0/0 \rightarrow 0$; число в регистре 0 удваивается.

Команда `fadd 1, 0` выполняет операцию $1+0 \rightarrow 1$.

Команда `fdivp 2, 0` выполняет операцию по схеме $2/0 \rightarrow 2$, а затем удаляет значение из регистра 0.

Операции `fdivr/fsubr` и `fdiv/fsub 0, 1` похожи, но не совсем одинаковы. Схема выполнения операции в обоих случаях одна и та же ($0 <_{op} 1 \rightarrow 0$), но в первом случае стек сокращается ("классический" исход), а во втором случае в регистре 1 остается значение источника.

Листинг 15.9. Операции с обратным порядком операндов (см. фрагмент 2 из `take4.8`)

```

finit
fld 2.0
fld 8.0 ; 8          2
fdivr   ; 4

finit
fld 2.0
fld 8.0 ; 8          2
fdiv 0, 1 ; 4          2

```

Операции над знаковым битом

Корректировка бита `sign` выполняется командами:

- ☐ инвертирования, или изменения знака — `fchs`;
- ☐ обнуления знака, или получения абсолютного значения — `fabs`.

Эти команды не нуждаются в указании операндов; источник и приемник — всегда содержимое регистра 0.

Округление до целого

Команда `frndint` округляет значение в регистре 0 до целого, в соответствии с установкой режима округления в битовом поле `rc` (Rounding Control) управляющего слова `sw`.

Примечание

В файле `fpu.inc` определены константы для программирования FPU, в том числе для установки `rc`.

Пример с установкой `rc` и проверкой выполнения `frndint` приведен в `round.8`. Для смены режима округления используется макрокоманда `set_rc`, с параметром в символическом виде — `near/down/up/chop/zero` (`chop` и `zero` — синонимы).

Листинг 15.10. Управление округлением (см. `round.8`)

```
set_rc macro
    fstcw    cw                ; read cw
    mov     ax, cw
    and     ax, not round_ctrl
    or      ax, round_#1
    mov     cw, ax
    fldcw   cw                ; write cw
#em

    ...
    fldpi           ; i.f
    fld     0       ; i.f      i.f
    set_rc  chop
    frndint           ; i      i.f
    fsub    1, 0     ; i      0.f
    ...
```

Для установки `rc` используется команда `fstcw` (Fpu SToRe Control Word), которая выполняет чтение `sw` в память по заданному адресу, и команда `fldcw` (Fpu LoAD Control Word), которая записывает в `sw` значение из памяти. Имя `sw` в примере обозначает переменную в памяти, определенную где-то при помощи директивы `dw`.

В кодовой части примера использован режим округления с отбрасыванием дробной части (задан константой `chop`), для разделения действительного числа `i.f` на целую `i` и дробную части `0.f`. Дробная часть получена в результате вычитания `i` из исходного значения `i.f`.

Получение остатка от деления

Команды `fprem` и `fprem1` — Fpu Partial Remainder — формируют в регистре 0 частичный остаток от деления регистров 0 и 1 (т. е., схема операции такая же, как для команды `fdivr`, только регистр 1 не удаляется).

При выполнении `fprem(1)` значение `exp` в делимом уменьшается не более чем на 63; таким образом, если разница между порядком делимого и порядком делителя превышает 63, то получается действительно *частичный* остаток, и в этом случае требуется повторить выполнение команды `fprem(1)`. Признак частичного остатка — единица во флаге `c2`.

Команда `fprem(1)` применяется для уменьшения аргумента периодической функции до такого предела, когда значение функции может быть получено с наименьшей погрешностью. Например, при вычислении тригонометрических функций аргумент должен быть не больше 2^{63} , что достигается применением `fprem(1)` со значением делителя 2π .

Команда `fprem` реализована не в соответствии со стандартом IEEE 754 — он еще не появился при выпуске i8087. В i80387 введена дополнительная команда `fprem1`, несколько отличающаяся в реализации от `fprem`, в соответствии со стандартом. Адресация операндов не изменилась. При программировании на i80387 и выше следует пользоваться командой `fprem1` — она должна быть точнее.

Контроль частичного остатка (`c2 = 1`) выполняется при помощи макрокоманды `rd_flags` из `fpu.inc`:

```
rd_flags macro
    fstsw    ax        ; Fpu SToRe Status Word
    sahf     ; ah -> flags
#em
```

В макрокоманде `rd_flags` команда `fstsw` (Fpu SToRe Status Word) запоминает значение `sw` в `ax`. Команда `sahf` затем копирует `ah` в `flags`. В результате, флаг `c2` отображается во флаг `p` (Parity) центрального процессора (в окне отображения FPU в d86 состояние флага `c2` выводится буквой `u` — Unordered).

Листинг 15.11. Вычисление остатка от деления (см. `fprem1.8`)

```
fld      0.1
fld      1.0e20 + 0.14
11:      fprem1
         rd_flags
         jp      11
```

Полученный результат 0.0447 ощутимо отличается от ожидаемого значения 0.04. Причина состоит в том, что число 0.1 представлено приближенно. (При помощи макрокоманды `what 0.1` в составе `inner.com` легко убедиться в том, что 0.1 в двоичном представлении — бесконечная периодическая дробь.) Делимое из примера, скорее всего, тоже не является точным. Команда `fprem(1)` реализована так, что делитель многократно вычитается из делимого, и если делитель представлен приближенно, ошибка быстро нарастает.

Примечание

Команды `fprem` и `fprem1` различаются способом округления при вычитании: независимо от установки `fs` в `sw`, `fprem` использует режим округления `zero`, а `fprem1` — режим `near`, что должно давать более точный результат.

Извлечение корня

Команда `fsqrt` извлекает квадратный корень из значения в регистре 0, помещая результат в регистр 0. Корень из отрицательного числа — NaN. Корень из отрицательного нуля дает -0 (-0 также может получиться из $+0$ в результате выполнения `fchs`).

Масштабирование

Команда `fextract` раскладывает число в регистре 0 на мантиссу и экспоненту, сохраняет их в регистрах 0 и 1, перезаписывая исходное значение в регистре 0.

Результат `fextract` проверяется в отладчике при вызове `d86` с программой `inner.com`. После запроса `what 0.625` следует выполнить команду `fextract`. Значение в регистре 0 станет равным 1.25 (мантисса), а в регистре 1 появится число -1 (двоичный порядок без смещения).

Если исходное значение равно 0, то выполнение `fextract` приводит к ошибке `z` (Zero divide); в регистре 0 остается значение 0, а в регистр 1 записывается $-\text{Infinity}$.

Команда `fextract` применяется:

- ☐ в отладчиках;
- ☐ для преобразования действительного числа в строку ASCII для вывода;
- ☐ при возведении в степень.

Команда "масштабирования" `fscale` выполняет обратное действие, т. е. умножает значение в регистре 0 на двойку в степени, заданной регистром 1. Результат помещается в регистр 0. При возведении в степень используется только целая часть значения в регистре 1, дробная часть в расчет не принимается. В любом случае, значение в регистре 1 остается без изменений.

Если в регистре 1 задано $-\text{Infinity}$, то в результате выполнения `fscale` в регистре 0 формируется ноль. Если в регистре 1 задано $+\text{Infinity}$, то при ненулевом значении в регистре 0 результат `fscale` — тоже $+\text{Infinity}$.

Операции сравнения и тестирования

Команда `fxam` анализирует содержимое регистра 0 с учетом соответствующего тега и устанавливает, в результате, флаги `c3`, `c2`, `c0`, как показано в табл. 15.4. Во флаге `c1` сохраняется значение бита `sign`, независимо от состояния регистра 0.

Команда `ftst` и команды семейства `fcom` (`Fpu Compare`) устанавливают флаги `c3`, `c2`, `c0` по результатам сравнения — так, как показано в табл. 15.2. Значения флагов `c3`, `c2`, `c0` затем копируются в флаги центрального процессора; соответствие между флагами сопроцессора и центрального процессора показано в табл. 15.3.

Команда `ftst` сравнивает число в регистре 0 с нулем.

Команды семейства `fcom` сравнивают регистр 0 с заданным операндом или с регистром 1, если операнд не задан.

`f(u/i)com(p/pp) <src>`

Опция `u` (`Unordered`) имеет значение только в особых ситуациях — при сравнении так называемых `Q NaN` — значений (`Quiet NaN`). (Этот тип `NaN`-величин мы не рассматриваем.) Если числа несопоставимы, то — независимо от опции `u` — флаги `c3`, `c2`, `c0` устанавливаются в 1, и взводится флаг особой ситуации `i` (`Invalid operation`); если задана опция `u`, то при сравнении `QNaN` флаг `i` не устанавливается.

Опция `p` (`Pop`) задает удаление содержимого регистра 0 из стека после сравнения; опция `pp` удаляет содержимое двух регистров — 0 и 1.

При включении опции `i` источник `<src>` содержит адрес целого значения в памяти. Сравнение с длинными целыми (типа `q`) не поддерживается; допускается сравнение только с целыми значениями типа `w`, `d`.

При выключенной опции `i` в `<src>` допускается задание адреса действительного значения в системной памяти. Сравнение с данными расширенной точности (тип `t`) не реализовано, допускаются только типы `d` и `q`.

Примеры задания инструкций семейства `fcom`:

```
fcom
fcomp
fcompp
fcom    2
fcomp   2
fcomp   q[0120]
ficomp  d[0200]
```

Инструкция `fcompp`, помимо основного назначения, используется для удаления из стека значений регистров 0 и 1.

15.9.3. Трансцендентные операции

К трансцендентным операциям относятся:

- ☐ тригонометрические операции;
- ☐ возведение в степень и вычисление логарифмов.

Тригонометрические операции

Команды для вычисления тригонометрических функций — `fsin`, `fcos`, `fsincos`; от сопроцессора i8087 унаследованы команды `fptan` и `fpatan` — `Fpu Partial TANGent/ArcTANGent`.

Команды `fsin`, `fcos`, `fsincos` вычисляют функции \sin или/и \cos от аргумента, заданного в регистре 0. Результат `fsin`, `fcos`, а также первый результат выполнения `fsincos` (\sin), помещается на место исходного значения. Второй результат `fsincos` (\cos) загружается в стек, сдвигая вниз значение \sin ; в итоге, по команде `fsincos` в регистр 0 записывается \cos , в регистр 1 — \sin .

Команда `fptan` помещает результат вычисления тангенса на место регистра 0, а затем выполняет команду `fld1`. В итоге, регистр 0 содержит единицу, а в регистре 1 сохранен результат вычисления тангенса.

Единица в регистре 0 нужна была в i8087 для дальнейших вычислений (в отсутствие инструкций `fsin/fcos` значения \sin/\cos определяются из \tan). Например, косинус в i8087 приходилось вычислять по формуле $\cos(a) = 1 / \sqrt{1 + \tan^2(a)}$, как показано в листинге 15.12.

Листинг 15.12. Вычисление косинуса по значению тангенса (см. `tg_cos.8`)

```
fld      0.5235987756      ; a=30°
fptan                    ; 1      b=tg(a)
fxch                          ; b      1
fmul      0                ; c=b^2   1
fadd      0, 1              ; c+1   1
fsqrt                     ; d      1
fdiv                     ; 1/d
                        ; 0.866
```

Перечисленные выше тригонометрические команды выполняются без значительных погрешностей, если абсолютное значение аргумента меньше 2^{63} . Если операнд больше 2^{63} , значение в регистре 0 остается без изменений, а $c2 = 1$. Если $c2 = 1$, необходимо — командой `fprem1` — уменьшить абсо-

лютное значение операнда на величину, кратную 2π , а затем повторить тригонометрическую команду.

Примечание

В i80387 диапазон допустимых значений расширился неизмеримо — в i8087 тангенс вычисляли лишь в пределах первого октанта, а после выполнения `fprem` требовались еще дополнительные действия в зависимости от номера октанта.

Команда `fpatan` — двуместная: в ней предварительно выполняется операция `fdiv` с удалением значений регистров 0 и 1 из стека и загрузкой результата. Затем вычисляется арктангенс частного. Окончательный результат сохраняется в регистре 0. Диапазон операндов не ограничен.

Возведение в степень

Возведение в степень выполняется командой `f2xm1`, при поддержке инструкций `fyl2x`, `fyl2xp1`.

Команда `f2xm1` возводит число 2 в степень, заданную регистром 0. Результат, уменьшенный на единицу, сохраняется в регистре 0. Команда `f2xm1` выполняется только для дробных значений степени — значение регистра 0 должно быть в открытом диапазоне $-1 — +1$, иначе результат не определен.

Вычисление x^y для произвольных значений x и y выполняется по формуле:

$$x^y = 2^{(y \times \log_2(x))}$$

Предварительное вычисление значения степени ($z = y \times \log_2(x)$) выполняется командой `fyl2x`. При $x = 10$ или $x = e$ вычислять логарифм не требуется (в i80387) — результаты вычислений готовы в виде встроенных констант, загружаемых командами `fldl2t`, `fldl2e`.

Команда `fyl2x` вычисляет значение $y \times \log_2(x)$, где y задано в регистре 1, а x — в регистре 0. Величина x должна быть больше нуля. Результат (после удаления значений из регистров 0 и 1) загружается в регистр 0.

При значениях x , близких к нулю, точность выполнения `fyl2x` падает. Следует в такой ситуации использовать команду `fyl2xp1`. В отличие от `fyl2x`, для команды `fyl2xp1` регистр 0 должен содержать не x , а разность между x и 1 ($\varepsilon = x - 1$). Абсолютная величина ε должна быть меньше $1 - (\sqrt{2})/2$ (~ 0.2929).

Решение задачи в общем виде выполняется в три этапа:

1. Вычисление $z = y \times \log_2(x)$.
2. Выделение в числе z целой и дробной частей — $i.f$.
3. Вычисление $2^{(i.f)} = 2^i \times 2^{0.f}$.

При $x = 2$ вычисление z отпадает за ненадобностью. Начнем обсуждение для $x = 2$, в обратном порядке — с последнего шага. Примеры подпрограмм взяты из row_1.8.

Возведение числа 2 в целую степень

Для получения целой степени числа два используется команда масштабирования `fscale`. Подпрограмма `pow2_i` вычисляет 2^i ; значение степени задано на входе в регистре 0, туда же помещается результат.

Листинг 15.13. Подпрограмма для возведения 2 в целую степень (см. row_1.8)

```
pow2_i:          ; i
                ; 1
    fldl         ; 1
    fscale       ; 1*(2^i)
    fxch        ; i
    fstp 0       ; 2^i
    ret
```

Команды `fxch` и `fstp 0` удаляют значение i на выходе.

Возведение числа 2 в дробную степень

Подпрограмма `pow2_f` вычисляет 2^f — для значений f в пределах открытого интервала -1 — $+1$. Входное значение и результат — в регистре 0.

Листинг 15.14. Подпрограмма для возведения 2 в дробную степень (см. row_1.8)

```
pow2_f:          ; f
    f2xm1       ; 2^f-1
    fldl         ; 1
    fadd        ; 2^f
    ret
```

Вычисление целой и дробной частей значения степени

Чтобы возвести число два в произвольную степень z , необходимо выделить целую и дробную части z . С этой целью временно устанавливается режим округления зоро, и при помощи `frndint` извлекается целая часть — i . Дробная часть f получается вычитанием $z - i$.

Для получения целой части предназначена подпрограмма `trunc`. Передача параметра и возвращение результата — через регистр 0.

Листинг 15.15. Подпрограмма для получения целой части числа (см. pow_1.8)

```
trunc:                                ; z
    enter    4
    push     ax
    fstcw    [bp-2]
    mov      ax, [bp-2]
    and      ax, not round_ctrl
    or       ax, round_zero
    mov      [bp-4], ax
    fldcw    [bp-4]
    frndint                    ; int(x)
    fldcw    [bp-2]
    pop      ax
    leave
    ret
```

В локальных данных подпрограммы, адресуемых [bp-2] и [bp-4], формируется новое значение cw с установкой zero и сохраняется исходное значение cw. Команда fldcw (Fpu Load Control Word) выполняется дважды — для загрузки нового значения cw и для восстановления исходного.

Подпрограмма split разделяет число на целую и дробную части, вызывая trunc. Входное значение z задано в регистре 0; на выходе из подпрограммы в регистре 0 содержится дробная часть $z(f)$, а в регистре 1 — целая часть $z(i)$.

Листинг 15.16. Подпрограмма для выделения целой и дробной частей (см. pow_1.8)

```
split:                                ; z
    fld      0                        ; z          z
    call     trunc                    ; i          z

    fxch                                ; z          i
    fsub     1                        ; f=z-i       i
    ret
```

Возведение числа 2 в произвольную степень

Подпрограмма pow2_z вычисляет 2^z для произвольного значения z — отдельно для целой части (i) и отдельно для дробной части (f). Окончательный результат получается как произведение результатов для целой и дробной частей. Входное и выходное значения — в регистре 0.

Листинг 15.17. Подпрограмма для возведения 2 в произвольную степень (см. pow_1.8)

```
pow2_z:          ; z
    call    split ; f          i
    call    pow2_f ; 2^f       i
    fxch    ; i                2^f
    call    pow2_i ; 2^i       2^f
    fmul    ; 2^z
    ret
```

Вычисление логарифмов

Для решения задачи в общем виде требуется предварительно вычислить выражение $z = y \times \log_2(x)$. При $x = 10$ и $x = e$ подсчитывать логарифм не требуется — на этот случай предусмотрены константы $\log_2(10)$ и $\log_2(e)$, загружаемые по командам `fldl2t` и `fldl2e`.

Вычисление степеней для оснований, отличных от 2, 10 или e , выполняется командой `fyl2x`, а при малых значениях x — командой `fyl2xpl`.

15.9.4. Команды управления

Большинство команд управления рассмотрено в предшествующем изложении, поэтому здесь ограничимся кратким обзором.

- ❑ Команда `finit` выполняет общий сброс FPU; в результате `tc = near`, флаги особых ситуаций (`sw`) сброшены, флаги запрета прерываний (`sw`) по особым ситуациям установлены, все регистры данных помечены как незанятые, `st = 7`.
- ❑ Команды `fdecstp/fincstp` увеличивают/уменьшают `st` на единицу по кольцу, тем самым "передвигая" значения между регистрами вниз/вверх.
- ❑ Команда `ffree i` помечает регистр `i` как незанятый.
- ❑ Команда `fclex` сбрасывает признаки особых ситуаций в `sw`.
- ❑ Команда `fstcw/fstsw` считывает значение из `sw/sw` в 16-битную ячейку памяти, адрес которой указан в команде. В команде `fstsw` допускается операнд `ax`.
- ❑ Команда `fldcw` записывает в `sw` значение 16-битной ячейки памяти, адрес которой указан в команде. Подобная команда для записи в `sw` не предусмотрена.
- ❑ Команда `fnop` равносильна пустой операции. Некоторые команды управления, унаследованные от i8087, i80287, недействительны в i80387 и воспринимаются как `fnop`.

- ❑ Команда `fstenv <m14byte>` считывает в системную память регистры FPU, кроме регистров данных (см. рис. 15.5). Побочный эффект — установка в `sw` флагов запрета прерываний по особым ситуациям.
- ❑ Команда `fldenv <m14byte>` записывает в FPU образ его регистров из памяти. Формат образа тот же, что и при чтении командой `fstenv`.
- ❑ Команда `fsave <m94byte>` считывает в системную память данные FPU — целиком, включая регистры данных,— и выполняет `finit`. Структура образа данных в начале такая же, как при считывании "окружения" командой `fstenv`. За первыми 14 байтами окружения следуют копии восьми регистров данных, начиная с регистра 0 (т. е. от вершины стека), по 10 байт.
- ❑ Команда `frstor <m94byte>` устанавливает состояние FPU полностью, в соответствии с образом из системной памяти.
- ❑ Команда `wait/fwait` останавливает центральный процессор до завершения текущей операции в FPU. Тем самым, процессор синхронизирует свою работу с сопроцессором. (Строго говоря, эта команда относится не к FPU, а скорее к центральному процессору.) В i80387 такая синхронизация имеет смысл *только* при обработке аппаратных прерываний, возникающих в связи с ошибками FPU. В i8087 иначе — без `wait/fwait` после каждой команды FPU система ломается.

Ассемблеры для i80x86/87 вставляют `wait` автоматически, когда они настроены на трансляцию для i8087. В традиционных ассемблерах — это настройка по умолчанию. Напротив, а86 при трансляции определяет тип сопроцессора и вставляет команды `wait`, только если он обнаружил i8087.

15.10. Практикум

Выполните трансляцию и исследуйте в отладчике примеры, начиная с *разд. 15.8*.

Загрузите несколько значений и выполните команду `fstenv` (для исследования образа вручную рекомендуется задать адрес приемника равным 0 — начало PSP). В полученном образе измените значение, соответствующее `st`, и загрузите образ командой `fldenv`.

Для доступа к данным образа используйте переменные, определенные в структуре `env` или `fpu` из файла `fpu.inc`. Для работы в отладчике "вручную" достаточно выполнить трансляцию `fpu.inc` (полученный `fpu.com` окажется пустым, но имена из `fpu.inc` сохраняются в `fpu.sym`).

```

struc
ctrl    dw    ?
state   dw    ?
tags    dw    ?
ip_ptr  dd    ?      ; ip + opcode
opr_ptr dd    ?      ; linear addr of operand (20 bits)
env      ends

```

Например, слово тегов после считывания образа по адресу 0 доступно по имени `tags`.

Повторите опыт, воспользовавшись на этот раз командой `fsave` (предварительно загрузите несколько значений). При выполнении `fsave` произошел общий сброс; восстановите состояние командой `frstor`.

Исследуем возникновение особых ситуаций. Разделите 1 на 3. Какой флаг установлен в результате и что он означает? Разделите 1 на 0. Вопрос аналогичный. Выполните действия, в результате которых будут установлены флаги `d`, `o`, `u`.

Исследуйте команду `ftst`, записывая в регистр 0 положительное число, отрицательное число, NaN. Установка флагов `c0..c3` отображается в `d86` в виде литер "c", ".", "u", "z".

Выполните действия, в результате которых будет установлен флаг `s`, оцените при этом значение `c1` (1 при переполнении стека, 0 — при антипереполнении).

Задавая на вершине стека значения 0, NaN и +Infinity, проверьте установку флагов `c3..c0` в результате вызовов `fxam`. Воспользуйтесь интерпретацией `c3..c0` в `d86`: +Norm, -Norm, Empty, -Nan, +Nan, Unn, -Inf, +Inf. Освободите регистр 0 и проверьте результат `fxam` для этой ситуации.

Выполните полный сброс FPU, затем загрузите в FPU три разных числа. Выполните вычитания между следующими регистрами, без удаления данных из стека:

0 — 1 → 1

2 — 0 → 2

1 — 0 → 0

Поменяйте местами значения регистров 1 и 2 при помощи `fxch`. Скопируйте командами `fld`, `fst(p)` значение из регистра 2 в регистр 5. При помощи `fstp` прокрутите по кольцу регистры 0, 1, 2. Удалите значения регистров 0 и 1 из стека при помощи команды сравнения.

Напишите программу, которая загружает в стек числа 1.0, 2.0, 3.0, затем формирует в регистре 5 число -5.0 так, чтобы остальные регистры (кроме 0, 1, 2, 5) в итоге были свободны. Затем программа в цикле вызывает подпро-

грамму, которая передает по кольцу значения между регистрами 0, 1, 2, в каждом цикле восстанавливая регистр 5 — при помощи `fxch`.

Сформируйте на вершине стека число -0.0 .

Вычислите значение $199975 \bmod 3$. Выполните пример `fprem1.8` для нескольких значений дробной части делимого, используя в одной версии команду `fprem1`, а в другой — `fprem`. Версию задавайте в командной строке при вызове `a86`.

Напишите программу для вычисления выражения $x * (x - 2) / \sqrt{x + 1}$. Значение x задано в переменной типа `d` в формате действительного числа. Загрузка x из памяти должна быть только одна — в начале; загруженное значение дублируется в стеке командами `fld/fst(p)`.

Составьте программу для поиска максимума из пяти значений, загруженных в регистры 0—4. Решение — в двух вариантах:

1. Поиск выполняется итеративно, в цикле — за счет вращений или обменов, при помощи команд из набора `fdecstp`, `fincstp`, `fstp 5`, `fxch`.
2. Поиск выполняется в образе, прочитанном командой `fsave`; смещение данных от начала образа задавайте именем `st07`, которое определено в файле `fru.inc` в составе следующей структуры:

```
struc
ctrl    dw    ?
state   dw    ?
tags    dw    ?
ip_ptr  dd    ?      ; ip + opcode
opr_ptr dd    ?      ; linear addr of operand (20 bits)
st07    dt    8 dup ? ; st(0)..st(7)
fpu      ends
```

Сравнивать числа в формате с плавающей точкой несложно: смещение порядка (добавление `03fff`) было введено для того, чтобы при сравнении не нужно было отдельно анализировать поле `exp`. В результате, с учетом расположения `sign` в старшем бите, числа в формате с плавающей точкой сравниваются как обычные целые знаковые значения многократной точности.

Сравнение начинается со старших слов и, пока слова из разных значений совпадают, сравнение продолжается со смещением в сторону младших слов. При несовпадении слов цикл завершается. Если несовпадение возникло на первом слове, то сравнение завершается проверкой `ja/jb`. Если не совпадают последующие пары слов, то сначала нужно выполнить беззнаковую проверку `ja/jb` — для выяснения того, какая из пары мантисс оказалась больше по абсолютной величине; а затем проверить отдельно `sign` — если он установлен, то результат сравнения обратный.

Составьте программу генерирования гармонических колебаний по следующей рекуррентной схеме:

$$y(n+1) = A \times y(n) + B \times x(n)$$

$$x(n+1) = C \times y(n) + D \times x(n)$$

$A = D = \cos(b)$, $B = -C = \sin(b)$, $b = 2\pi / k$, где k — число точек на период. Начальные значения: $y(0) = 0$, $x(0) = 1$ (на "выходе" x — косинус, на "выходе" y — синус). Для вычисления коэффициентов A — D воспользуйтесь калькулятором. Рекомендуемое значение k — 20—40.



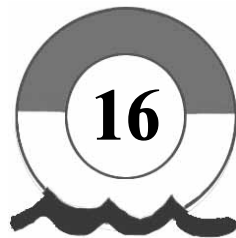
Часть III

Управление внешними устройствами, прерывания, резидентные программы

Глава 16. Управление внешними устройствами

Глава 17. Прерывания и исключения

Глава 18. Резидентные программы



Управление внешними устройствами

Все пуговицы, все блохи, все предметы что-то значат.
И неспроста одни ползут, другие скачут.
Я различаю в очертаниях неслышимый разговор:
О чем-то сообщает хвост,
на что-то намекает бритвенный прибор.

Тебе селедку подали. Ты рад.
Но не спеши ее отправить в рот.
Гляди, гляди! Она тебе сигналы подает.

Н. М. Олейников. "Озарение". 1932

В этой главе рассмотрен доступ к регистрам адаптеров внешних устройств и управление внешними устройствами — сначала вручную при помощи резидентного редактора портов `grogrt`, а затем по программе. Во многих опытах используется последовательный канал связи. Если в вашей системе мышь подключена к COM-порту, следует отключить драйвер мыши — закомментировать в `autoexec.bat` вызов драйвера.

16.1. Внешние устройства в программной модели вычислительной системы

Устройства, образующие вычислительную систему, можно разделить на *системные* и *внешние*, или *периферийные*. Системные устройства обеспечивают манипулирование данными в памяти — по программе, которая тоже хранится в памяти. К системным устройствам относятся процессор и непосредственно доступная ему (системная) память. Внешние устройства — это дополнительные компоненты вычислительной системы, предназначенные для взаимодействия с "внешним миром" (в первую очередь, с пользователем).

Подключение внешних устройств к системной шине, для связи с процессором, выполняется через дополнительные переходные устройства — *адаптеры*.

В программной модели вычислительной системы адаптеры внешних устройств представлены *регистрами ввода/вывода*, которые подключены к системной шине и доступны процессору для чтения-записи. Регистры ввода/вывода различаются по адресам.

Примечание

В дальнейшем изложении термин "адаптер" почти не используется, а термином "устройство" обозначается как внешнее оборудование, так и адаптер, представляющий это оборудование в программной модели вычислительной системы.

В архитектуре IBM PC регистры ввода/вывода соединяются с системной шиной через *порты*. Порт рассматривается как байтовая ячейка в пространстве *адресов ввода/вывода*; диапазон адресов — от 0 до 0ffff. К порту либо подключен регистр внешнего устройства, либо не подключено ничего.

Поскольку общая размерность регистров устройства, как правило, превышает один байт, для соединения с устройством используется массив портов. Начальный адрес массива портов при соединении с некоторым устройством называют *базовым адресом* этого устройства. Так, например, адаптер последовательного канала связи COM1 в IBM PC подключен к системной шине через порты с адресами 03f8—03fd; следовательно, базовый адрес COM1 равен 03f8.

16.2. Инструкции для доступа к портам

Доступ к портам обеспечивают две группы команд процессора: *in* — для чтения, *out* — для записи. Возможен доступ либо к одному порту за одну операцию, либо (для процессора i80286 и выше) сразу к двум смежным портам, которые рассматриваются как 16-битное слово в пространстве адресов ввода/вывода, либо (начиная с процессора i80386) сразу к четырем смежным портам. При доступе к слову или двойному слову в пространстве адресов ввода/вывода адрес должен быть кратен 2 или 4 соответственно.

Примечание

При доступе к слову или двойному слову в пространстве адресов *системной памяти* допускается задание нечетного адреса; чтение-запись слова по нечетному адресу выполняется медленней, но не является ошибкой. С портами — иначе.

На практике чаще требуется доступ к отдельным портам. Обмен данными между портом/портами и процессором по командам *in* и *out* выполняется

всегда через регистр-аккумулятор `al/ax/eax` — в зависимости от числа портов в операции. Адрес порта или группы соседних портов задается либо содержимым `dx`, либо константой в пределах `0—0fff`. Таким образом, адрес в диапазоне `0100—0ffff` всегда задается регистром `dx`.

Примеры:

```
mov    dx, 03f8      ; адрес > 0ff, поэтому записываем его в dx
in     al, dx        ; считываем порт по адресу, заданному dx,
                    ; в регистр al

in     al, 040       ; адрес <= 0ff задан непосредственно

out    070, al       ; записываем содержимое регистра al
                    ; в порт по адресу 070

mov    dx, 03f8      ; записываем содержимое ax (al, ah) в два
out    dx, ax        ; смежных порта по адресам dx и dx+1
```

В систему команд процессора `i80286` включены также строковые варианты команды `in/out: ins/outs`. С их помощью можно считывать/записывать последовательность байт (`insb/outsb`) или слов (`insw/outsw`). Адрес порта задается в `dx` и при выполнении команды `ins/outs` не изменяется. Адрес приемника/источника в системной памяти задается в регистрах `es:di/ds:si` и автоматически изменяется после каждого цикла чтения-записи, в соответствии с установкой флага направления `d` и размерностью операндов (т. е., при байтовом обмене значения в регистрах `di/si` увеличиваются или уменьшаются на 1, а при обмене парами байт — на 2). Команды `ins/outs` допускают префикс повторения `rep`; строковая команда после `rep` повторяется до обнуления счетчика повторений в `cx`.

Пример:

```
mov    dx, 040       ; адрес порта
lea    di, dest      ; адрес массива-приемника в системной памяти
cld                    ; направление — в сторону увеличения адресов
mov    cx, 1000      ; счетчик повторений для rep
rep insb
int    020

dest    db
```

Команда `rep insb` в этом примере заменяет следующий цикл:

```
l1:    in     al, dx
        stosb
        loop  l1
```

16.3. Исследование внешних устройств вручную

Знакомство с внешними устройствами начнем с элементарных операций на уровне команд `in/out`. Эти операции можно выполнять в непосредственном режиме `d86`, но удобнее пользоваться резидентным редактором портов `gport.exe`, разработанным авторами этой книги.

16.3.1. Запуск редактора портов

Загрузите `gport` (запуск `gport` рекомендуем включить в `autoexec.bat`). Для вызова редактора нажмите комбинацию клавиш `<Alt>+<Z>`. Для доступа к порту наберите адрес порта в шестнадцатеричном формате (ведущий ноль задавать не обязательно) и нажмите клавишу `/` на дополнительной клавиатуре:

```
3f8/
```

Результат чтения порта `03f8` выводится справа от литеры `/`. Для повторного чтения порта с тем же адресом, вновь нажмите клавишу `</>`; результат вместе с адресом будет выведен на следующей строке. Для чтения порта по следующему адресу нажмите клавишу `<+>`, для чтения по предыдущему адресу — клавишу `<->`. Чтобы полностью задать адрес, следует сначала перейти к следующей строке — нажатием клавиши `<Enter>`.

16.3.2. Определение наличия устройства

Проведем простой опыт по определению количества адаптеров последовательной связи, установленных на вашем компьютере. В IBM PC обычно предусмотрено два таких адаптера (в DOS им присвоены имена `COM1`, `COM2`); каждый занимает от шести до восьми смежных портов (зависит от модели адаптера), с базовыми адресами `03f8` и `02f8`.

Чтобы определить наличие устройства, следует прочитать порты, предназначенные для подключения устройства. Результат чтения `0ff`, как правило, свидетельствует о том, что к порту либо ничего не подключено, либо подключен регистр, доступный только для записи. Если при чтении всех портов "устройства" получено значение `0ff`, то, скорее всего, никакого устройства нет.

Прочитайте шесть портов, начиная с адреса `03f8`, затем повторите чтение с адреса `02f8`. Сколько адаптеров последовательной связи установлено на вашем компьютере? Проверьте наличие адаптеров параллельного обмена, или LPT-адаптеров. Базовые адреса — `0378` и `0278`; каждый такой адаптер занимает три смежных порта.

Примечание

Если в вашей системе используется дисплей Hercules, то по адресу 03bc находится дополнительный LPT-адаптер, установленный на плате адаптера Hercules.

Рассмотренный способ дает достоверную информацию об отсутствии устройства. Вместе с тем, для идентификации устройства, обнаруженного по некоторому адресу, требуется иметь понятие о составе и возможных значениях его регистров.

Примечание

Идентификация устройства на основании только значения базового адреса не вполне достоверна. Например, в адресах, обычно занятых COM- и LPT-адаптерами, может оказаться адаптер Ethernet WD8003, у которого адрес настраивается переключателями на плате.

16.3.3. Доступ к регистрам устройства

Способ доступа к регистрам устройства определяется схемой их подключения к портам ввода/вывода.

Наиболее просто доступ организуется при непосредственном подключении регистров к портам. В этом случае чтение или запись порта означают чтение или запись соответствующего регистра устройства.

Обычно разработчики периферийного оборудования выбирают такой способ подключения, если регистров немного. Характерный пример — адаптер параллельной связи с принтером: каждый из трех байтовых регистров этого адаптера подключен к отдельному порту.

Если количество программно доступных регистров устройства превышает 6—7, то к одному порту обычно подсоединяют несколько регистров через схему коммутации.

Коммутация по чтению-записи

В этом варианте коммутации к одному порту подключается пара регистров. Выбор определяется уровнем сигнала чтения-записи при обращении к порту: при *чтении* порта к нему подключается один регистр, а при *записи* — другой.

Такой способ коммутации применяется, например, в адаптере последовательного канала связи, который мы рассмотрим подробнее в *разд. 16.3.4*. Регистр данных *передатчика* и регистр данных *приемника* последовательного канала связи подключены к общему порту, расположенному по смещению 0 от базового адреса устройства. При чтении этого порта к нему подключается регистр данных приемника последовательного канала связи, а при записи — регистр данных передатчика.

Прочитайте порт 03f8 (или 02f8, если адаптер по адресу 03f8 отсутствует). В результате чтения получено значение регистра данных приемника. Поскольку при чтении автоматически коммутируется регистр данных приемника, прочитать содержимое регистра данных передатчика невозможно. Изменится ли содержимое регистра данных приемника, если записать что-либо в порт 03f8?

Для записи в порт из редактора `gport` следует сначала прочитать порт, а затем — на той же строке — ввести значение для записи, закончив ввод нажатием одной из клавиш:

- `<Enter>` — запись;
- `<+>` — запись в текущий порт, затем чтение из следующего порта;
- `<->` — запись в текущий порт, затем чтение из предыдущего порта.

Запишите в порт 03f8 произвольное значение. При записи в порт 03f8 к нему был подключен регистр данных передатчика последовательного адаптера. Прочитайте порт 03f8, сравните результат чтения с только что записанным значением. Какой регистр вы читали?

Несовпадение значения, прочитанного из порта, с только что записанным значением — это обычное явление при работе с портами, т. к. отдельные разряды регистра устройства могут быть недоступны для записи. Продемонстрируем защиту от записи на примере регистра *идентификации прерываний* в составе последовательного адаптера. В этом регистре отражается *состояние* устройства, поэтому проектировщики адаптера сделали регистр недоступным для записи.

Регистр идентификации прерываний подключен к порту со смещением два. Прочитайте этот регистр. Единица в нулевом бите означает отсутствие заявки на аппаратное прерывание; единицы в битах 1—3 соответствуют трем различным событиям, приводящим к установке заявки. Запишите в этот регистр число 0e, что должно означать заявку на прерывание сразу по трем причинам. Тем самым мы попытались навязать устройству состояние, не соответствующее действительности. Вновь прочитайте регистр идентификации прерываний. Изменилось ли состояние устройства, отраженное в этом регистре?

Управление коммутацией через отдельный порт

При таком способе коммутации линии управления коммутатором подключены к одному из регистров, доступных для записи. Значение в этом регистре определяет выбор одного из нескольких регистров для подключения к какому-то порту.

Простейший вариант этого способа — для пары регистров — использован в последовательном адаптере. В этом адаптере имеется 16-разрядный регистр

делителя частоты, значение в котором определяет скорость приема-передачи. Этот регистр подключается к портам со смещениями ноль и один через коммутатор, к входам которого — в исходном состоянии устройства — подсоединены:

□ байтовый регистр *данных* (к порту по смещению ноль);

□ байтовый регистр *разрешения прерываний* (к порту по смещению один).

Прочитайте порты последовательного адаптера по смещениям ноль и один. Сейчас эти порты подключены к регистру данных и к регистру разрешения прерываний соответственно. Запомните прочитанные значения.

Прочитайте порт по смещению три. К нему подключен *управляющий* регистр адаптера, который определяет параметры настройки канала связи кроме скорости передачи. Старший бит этого регистра управляет коммутатором, который соединяет первые два порта либо с 16-разрядным регистром делителя частоты, либо с байтовыми регистрами: данных и разрешения прерываний. Для переключения на регистр делителя частоты установите старший бит управляющего регистра в единицу. Перед тем как записать требуемое значение (080), запомните прочитанное число.

Прочитайте первые два порта устройства. Через порт со смещением ноль сейчас доступен младший байт 16-разрядного делителя частоты, а через порт со смещением один — старший байт. Полученное значение переведите в десятичный формат и вычислите настройку скорости передачи по формуле $r = 115200 / dvd$, где r — скорость (бит/с), dvd — значение делителя частоты. Восстановите исходное значение в управляющем регистре и прочитайте первые два порта. Какие *регистры* устройства вы прочитали?

При увеличении числа регистров применяется более развитая схема коммутации — *адресная*. Адресная схема коммутации принята при доступе к CMOS-памяти в составе микросхемы Motorola MC146818. CMOS-память хранит сведения о конфигурации вычислительной системы, а также данные часов реального времени; вся эта информация находится в шестидесяти четырех байтовых регистрах. Для доступа к этим регистрам используются всего два порта: порт по адресу 070, соединенный с адресным коммутатором, и порт по адресу 071, к которому подключается один из регистров CMOS. Запись числа в порт 070 определяет номер регистра CMOS, подключенного к порту 071.

Запишите в порт 070 число 4. В результате, к порту 071 подключен четвертый регистр CMOS, в котором хранится значение текущего часа. Прочитайте порт 071. Данные часов реального времени (часы, минуты, секунды) представлены в BCD-формате (в каждой позиции шестнадцатеричного числа — десятичная цифра). Прочитайте текущие значения минут и секунд — в регистрах номер 2 и 0.

Примечание

В первых реализациях CMOS-памяти, после доступа к регистру данных адрес обнуляется или становится неопределенным. Поэтому рекомендуется перед каждым обращением к порту 071 записывать в порт 070 номер регистра, даже если этот номер прежний. Попробуйте выяснить, как изменяется адрес после доступа к регистру данных CMOS-памяти в вашем компьютере.

При *последовательной* коммутации регистры устройства подключаются к порту по очереди, после каждого обращения к порту. При такой организации отпадает необходимость в управлении коммутатором через отдельный порт.

Простейший пример последовательной коммутации — для пары байтовых регистров — представляет трехканальный таймер на базе микросхемы 8253/8254. Это устройство содержит три независимо работающих таймера (каналы с номерами 0, 1, 2). Централизованный регистр команд подключен к порту 043. Регистры данных у каждого канала свои: данные канала 0 доступны через порт 040, данные первого канала — через порт 041, второго — через порт 042. У каждого канала имеется отдельный набор 16-разрядных регистров данных; для нас интерес представляют: регистр счета и регистр начального значения счетчика, или регистр уставки. Доступ к байтам регистра уставки или регистра счета выполняется последовательно, через один порт. При чтении доступен очередной байт регистра *счета*, а при записи — очередной байт регистра *уставки*; как видите, здесь дополнительно используется коммутация по чтению-записи.

Когда счет разрешен, импульсы с частотой 1193 кГц поступают на вход вычитания регистра счета; с каждым импульсом значение счетчика уменьшается на единицу. При обнулении счетчика выполняются следующие действия:

- ❑ на выходе канала генерируется импульс, который используется другими устройствами: например, выход второго канала подключен через промежуточный клапан к динамике.
- ❑ если установлен режим циклического счета, в регистр счетчика записывается стартовое значение из регистра уставки, и счет возобновляется

Последовательная коммутация в этом устройстве используется для доступа к байтам 16-разрядного регистра. После каждого обращения к порту к нему подключается другой байт регистра. Для чтения или записи регистра целиком следует обратиться к порту два раза.

Для двукратного чтения порта в редакторе `gport` воспользуйтесь клавишей `<*>`. Наберите адрес 040 и нажмите клавишу `<*>` на дополнительной клавиатуре. При первом чтении через порт 040 доступен старший байт регистра счетчика, а при втором — младший байт. Несколько раз нажмите клавишу `<*>`. Значения меняются, следовательно, счет для этого канала разрешен.

Примечание

В первом канале доступен только младший байт регистра счета, что было задано BIOS при настройке канала.

Проверьте, аналогично, разрешен ли счет в канале 2. Для разрешения счета в этом канале следует установить в единицу нулевой разряд порта 061, подключенного к микросхеме интерфейса с периферией. Прочитайте порт 061 и запишите в него прочитанное значение с установленным в единицу младшим разрядом.

Примечание

Для преобразования шестнадцатеричных чисел, с которыми работает редактор `gport`, в двоичные воспользуйтесь резидентной программой `digit.exe` (после загрузки активизируется по комбинации клавиш `<Ctrl>+<Z>`). В строке "Hex" вводится число в шестнадцатеричном формате и — после нажатия клавиши `<Enter>` или в результате перехода на другую строку клавишей-стрелкой — выводится во всех форматах.

Старшая тетрада регистра, подключенного к порту 061, недоступна для записи. Поэтому если значение, прочитанное из порта 061, равно, например, 020 или 030, то при записи не обязательно задавать те же единицы в разрядах 4—7. В данном случае не будет ошибкой записать вместо 021 или 031 число 1. Установите нулевой бит в порте 061 и проверьте, идет ли счет во втором канале.

Продолжим работу с каналом 2 до получения звука заданной частоты. Чтобы импульсы с выхода этого канала поступали на динамик, следует открыть промежуточный вентиль — установить в единицу первый разряд порта 061. Выполните эту операцию; для выключения звука восстановите исходное значение в порте 061.

Чтобы изменить частоту звука, следует изменить начальное значение счетчика в регистре уставки. Запишите в порт 043 число 0b6 — код команды "запись двух байтов уставки для канала 2 и переход в режим циклического счета". По этой команде включается последовательная коммутация двух байтов регистра уставки; первая запись направляется в младший байт регистра, а вторая — в старший.

Запишем в регистр уставки число 1193 (шестнадцатеричное 04a9) следующим образом. После выдачи команды 0b6, первая запись в порт 042 равносильна установке значения в младшем байте регистра уставки. Поэтому записываем в порт 042 сначала число 0a9. Затем — число 4, которое попадает в старший байт регистра уставки, после чего автоматически запускается счет, если он разрешен — единицей в бите 0 порта 061.

Мы задали частоту 1000 Гц. Проверьте: после включения и выключения звука сравните его частоту на слух с результатом программы `sound`, вызвав ее

с параметром 1000 (т. е. следует вызвать `sound.exe 1000`). Получите звук с выбранной вами частотой в пределах 100—5000 Гц.

16.3.4. Управление устройствами

Управление устройством подразумевает:

- ☐ наблюдение за состоянием устройства;
- ☐ обмен данными между процессором и устройством;
- ☐ выдачу команд устройству.

Соответственно, регистры устройства разбиваются на следующие группы:

- ☐ регистры состояния;
- ☐ регистры данных;
- ☐ регистры управления.

В регистрах *состояния* отражается текущее состояние устройства, или события, связанные с его работой: завершение операций ввода/вывода (приема-передачи), ошибки, временные события (завершение цикла счета таймера, срабатывание будильника часов реального времени) и т. д.

Регистры *управления* предназначены для ввода команд, определяющих режим и/или параметры работы устройства.

Регистры *данных* обеспечивают запись данных в устройство для операций вывода (передачи) или чтение данных из устройства, полученных им в результате ввода (приема).

Работа с регистром управления проиллюстрирована на примере трехканального таймера. Работу с регистрами данных и состояния продемонстрируем на примере адаптера последовательной связи. Предварительно, в общих чертах, рассмотрим принцип последовательной связи, применительно к СОМ-портам IBM PC.

Принцип последовательной связи

При последовательной связи данные передаются по однопроводной линии — бит за битом, по одному в каждый момент времени. Уровень сигнала, соответствующий значению очередного бита, выдерживается передатчиком в течение фиксированного интервала времени (заданного при настройке); затем передатчик устанавливает и выдерживает уровень, соответствующий значению следующего бита, — и т. д., пока не будут таким образом "переданы" все биты. Одновременно и в том же темпе принимающая сторона периодически проверяет уровень сигнала в линии, накапливая данные, бит за битом. В каждом таком цикле обмена передается и принимается восемь *информационных* бит (настройка по умолчанию).

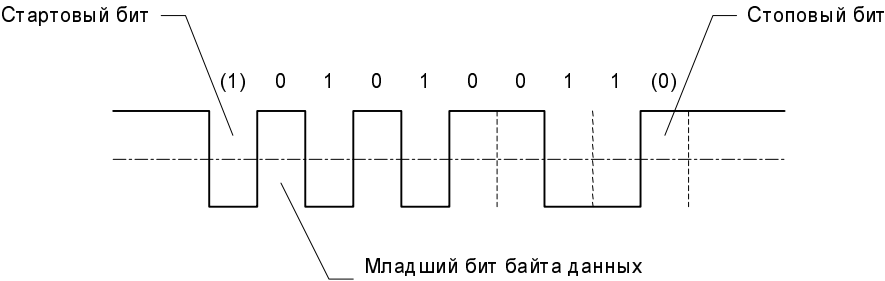


Рис. 16.1. Диаграмма последовательной передачи байта со значением 11001010_{хв}

В последующих опытах мы используем *асинхронную* приемопередачу, диаграмму которой иллюстрирует рис. 16.1. Передача начинается со служебного *стартового* бита. Назначение стартового бита — запуск приемника на другом конце линии, за счет изменения уровня сигнала: стартовый бит передается низким уровнем, в исходном же состоянии уровень сигнала высокий. Стартовый сигнал выдерживается в течение одного такта, затем передаются *информационные* биты. В конце цикла обмена передатчик устанавливает в линии исходный, высокий уровень сигнала и выдерживает его как минимум один такт — тем самым, "передается" служебный *стоповый* бит. Промежуток между циклами — в виде стоповой посылки — необходим для формирования перепада уровней при выдаче стартового бита в начале следующей посылки.

Основные регистры адаптера последовательной связи

Регистры данных приемника и передатчика подключены к порту со смещением ноль (базовый адрес COM1 равен 03f8) и коммутируются по чтению-записи. Регистр состояния находится по смещению пять. В табл. 16.1 показано назначение разрядов регистра состояния.

Таблица 16.1. Назначение разрядов регистра состояния последовательного адаптера

Бит	Состояние приемника
0	Готовность приемника (данные приняты)
1	Ошибка переполнения при приеме (до начала приема не был прочитан регистр данных приемника, где хранится результат предыдущего приема)
2	Ошибка четности при приеме
3	Ошибка синхронизации при приеме (искажение уровней в стоповой посылке)

Таблица 16.1 (окончание)

Бит	Состояние передатчика
5	Буферный регистр передатчика пуст, можно записывать байт для передачи
6	Сдвиговый регистр передатчика пуст, передачи закончены

Разряды 0—3 характеризуют состояние приемника, разряды 5 и 6 — состояние передатчика. Прочитайте порт 03fd, опишите состояния приемника и передатчика.

В табл. 16.1 упоминаются два регистра передатчика — буферный и сдвиговый. То, что называется регистром данных, — это *буферный* регистр, в который записываются данные через порт со смещением ноль. Данные, записанные в буферный регистр, поступают в *сдвиговый* регистр, как только закончена передача очередного байта. Байт из сдвигового регистра передается в линию по битам. Пока идет передача, сдвиговый регистр занят; а в освободившийся буферный регистр тем временем можно записать следующий байт для передачи.

Примечание

Поскольку запуск передачи и запуск приема по последовательному каналу выполняются автоматически, то отдельный регистр *управления* не нужен.

Последовательная передача в диагностическом режиме

Для наблюдения за результатами и процессом приемопередачи воспользуемся *диагностическим* режимом адаптера. В этом режиме выход передатчика замкнут внутри адаптера на вход приемника. Для включения диагностического режима установите в единицу бит 4 регистра *управления модемом*, который подключен к порту со смещением четыре.

Запишите в регистр данных передатчика произвольное значение. Запись в регистр данных автоматически запускает вывод (передачу) записанного значения. При работе вручную передача кажется мгновенной; при начальной загрузке BIOS настраивает последовательные адаптеры на скорость 2400 бит/с, поэтому передача байта вместе со служебными битами занимает не более 5 мсек.

Прочитайте регистр состояния. Бит готовности приемника установлен — передача завершена. Готовность приемника сохраняется до тех пор, пока принятые данные не будут прочитаны. Прочитайте регистр данных приемника, затем еще раз — регистр состояния. Состояние приемника изменилось: признак его готовности сброшен.

Для наблюдения за *процессом* приемопередачи уменьшите скорость до минимума. Скорость определяется значением делителя частоты (см. "Управление

коммутацией через отдельный порт" в разд. 16.3.3). Установите в делителе частоты максимальное значение 0ffff. В результате скорость снизится до $115200/65535 = 1,76$ бит/с, и длительность одного цикла приемопередачи увеличится до нескольких секунд.

Запишите в регистр данных передатчика произвольное значение. Сразу же прочтите регистр состояния и продолжайте читать, повторяя нажатия клавиши "/", пока состояние не станет равным 0b1, как в конце предыдущего опыта. Какие этапы можно выделить в наблюдаемом процессе?

Получение ошибки переполнения приемника

Проведем еще один опыт с целью вызвать ошибку *переполнения приемника*. Эта ошибка возникает, если очередной байт принят тогда, когда регистр данных приемника занят — не успели прочитать результат предыдущего приема.

Запишите в регистр данных передатчика произвольное число. Сразу же запишите в него еще одно число, и приступайте к чтению регистра состояния. Продолжайте чтение, пока состояние не станет равным 0b1. Какие этапы выделяются в наблюдаемом процессе?

Чтобы лучше понять результаты этого опыта, следует учесть буферизацию при приеме и при передаче. Напомним, буферный и сдвиговый регистры передатчика в исходном состоянии свободны, поэтому в буферный регистр допускается запись двух байтов подряд; первый байт немедленно поступает в пустой сдвиговый регистр, а второй дожидается своей очереди в буферном регистре.

Приемник включает в себя аналогичные регистры. В сдвиговом регистре приемника накапливается результат текущего цикла обмена; байт данных по окончании приема копируется из сдвигового регистра в буферный. В результате возникает "готовность" приемника; буферный регистр приемника с этого момента "занят" — до тех пор, пока данные не будут прочитаны извне. При этом, незанятый сдвиговый регистр в состоянии принимать следующие данные. Если к концу следующего цикла приемопередачи буферный регистр все еще занят (данные не прочитаны), фиксируется ошибка переполнения.

16.4. Управление устройствами по программе

Управление устройством включает в себя:

- ☐ операции настройки;
- ☐ запуск операций вывода;

- реакцию на завершение операций ввода;
- восстановление после ошибок.

В большинстве случаев, для управления устройством необходима информация о *состоянии* устройства, отраженная в соответствующем *регистре*. Так, например, вывод через последовательный канал возможен лишь тогда, когда регистр передатчика свободен. Запись данных в порт последовательного адаптера по смещению 0 бесполезна, если буферный регистр передатчика занят.

При управлении устройством по программе состояния, требующие вмешательства со стороны программы, обнаруживаются одним из следующих способов:

- *программа* периодически *опрашивает* регистр состояния устройства;
- *устройство* в определенных состояниях *прерывает* выполнение текущей программы, и процессор переключается на выполнение другой программы.

При первом способе, наблюдение за состоянием и реакция на события чередуются в программе; наблюдение выполняется *программно*. При втором способе, реакция по-прежнему программная, зато наблюдение — *аппаратное*; в результате, программа не тратит времени на выявление особых состояний устройства.

16.4.1. Наблюдение за состоянием в режиме периодического опроса

Наблюдение за состоянием устройства в режиме периодического опроса проиллюстрируем на примере передачи по каналу последовательной связи.

Листинг 16.1. Вывод в последовательный канал с опросом готовности (см. send.8)

```

base    equ    03f8        ; базовый адрес COM1
buf     equ    base        ; буферный регистр приемника/передатчика
state   equ    base+5      ; регистр состояния

mov     cx, 1000           ; счетчик циклов обмена
mov     bl, 'A'            ; данные для передачи
m2:     mov     dx, state
m1:     in      al, dx       ; опрос регистра состояния
test    al, bit 5          ; буферный регистр передатчика пуст?
jz      m1                ; если нет, продолжать опрос
```

```
mov    dx, buf           ; запись
mov    al, bl             ; в буферный регистр
out    dx, al            ; передатчика

loop   m2
ret
```

В этом примере запрограммирована передача тысячи байтов со значением ASCII-кода 'А'. Перед каждой передачей выполняется цикл опроса регистра состояния, пока не будет зафиксирован признак "буферный регистр передатчика пуст". После этого очередной байт записывается в буферный регистр данных передатчика.

Выполните программу из файла `send.8`. Хотя передача никак себя не проявляет, факт передачи косвенно подтверждается временем выполнения программы. Скорость передачи составляет 2400 бит/с (по умолчанию), и тысяча циклов (по 10 бит каждый, включая служебные биты) длится около 4 сек. При этом устройство выполняет "полезную" работу в максимальном темпе, а программа, напротив, работает вхолостую, расходуя большую часть времени на отслеживание состояния устройства. Чтобы оценить затраты времени на ожидание готовности, достаточно убрать команду `jb` из цикла ожидания и заново выполнить программу. Несмотря на то, что в этой программе запись в порт по смещению 0 выполняется 1000 раз, в линию передается максимум два байта — если программа еще не завершилась до того момента, как данные из буферного регистра передатчика были скопированы в сдвиговый.

Запрограммируйте опыт из *разд. 16.3.4* по получению ошибки переполнения приемника. В программе предусмотрите следующие действия:

- ☐ включение диагностического режима;
- ☐ двукратную запись в регистр данных передатчика, с ожиданием готовности буферного регистра передатчика между первой и второй записью;
- ☐ опрос регистра состояния, до появления единицы в бите 1;
- ☐ выключение диагностического режима.

Примечание

Когда этот опыт выполнялся "вручную", интервал между первой и второй записью в регистр передатчика был настолько большим, что буферный регистр успевал освободиться от первого байта (передача из буферного регистра в сдвиговый занимает не более одного такта работы устройства). При работе по программе перед второй записью цикл ожидания события "буферный регистр передатчика свободен" необходим, иначе второй байт будет потерян.

Чтобы освободить программу от постоянного опроса состояния устройства, следует обеспечить автоматическое (по "инициативе" устройства) *переключение* на другую программу, которая, выполнив необходимые действия (например, запись в регистр данных), вернет управление прерванной программе.

16.4.2. Реакция на особые состояния в режиме прерываний

Прерывание — это переход, вызванный событием в устройстве. Адрес перехода определяется источником прерывания. Переход выполняется по аналогии с вызовом подпрограммы: адрес возврата в прерванную программу автоматически сохраняется в стеке. Последовательность команд, расположенная по адресу перехода, должна завершаться инструкцией возврата из прерывания — `iret`.

При прерывании, в отличие от вызова подпрограммы, инструкция перехода не задана в программе. Вызов выполняется *устройством*, поэтому трудно предсказать, в каком месте программы он произойдет — момент прерывания определяется ритмом работы исполнительного устройства. Программа лишь *подготавливает* условия для прерывания, само же прерывание порождается устройством.

Задание адреса перехода при прерывании

В принципе (безотносительно к системам на базе i80x86) существует два способа задания адреса перехода при прерывании:

1. Адрес перехода для каждого источника прерывания установлен аппаратно.
2. Адрес перехода для каждого источника хранится в системной *таблице*, которая находится в фиксированном месте в памяти; каждому источнику аппаратно назначен уникальный *номер* в таблице.

Первый способ применяется в системах с небольшим количеством источников прерываний. Например, в микроконтроллере i8048 имеются всего два источника прерываний (таймер и дополнительное внешнее устройство); первый источник вызывает переход на инструкцию по абсолютному адресу 3, второй — по адресу 7.

Второй способ — *векторный* — применяется в системах с большим числом источников прерываний; этот способ рассматривается в дальнейшем, применительно к системам на базе микропроцессора i80x86.

Таблица адресов переходов в i80x86 находится по адресу 0:0 и содержит 256 значений (двойных слов) в формате сегмент:смещение. Для каждого источника прерываний определен номер в этой таблице.

Например, при обнулении регистра счета нулевого канала трехканального таймера вырабатывается заявка на прерывание с номером 8. (Источник прерывания — нулевой канал таймера, а причина прерывания — завершение цикла счета.) При выполнении процессором заявки на прерывание с номером 8 переход произойдет по адресу, значение которого хранится в восьмой записи таблицы. Записи нумеруются от нуля, каждая запись содержит по

четыре байта, поэтому адрес перехода находится в двойном слове по адресу 0:32. Младшее слово содержит значение смещения, старшее слово — сегментную составляющую адреса перехода. Таким образом, при выполнении прерывания от нулевого канала таймера новое значение *cs* находится в слове по адресу 0:34, а новое значение *ip* — в слове по адресу 0:32.

Выполнение прерывания процессором i80x86

Процессор отслеживает заявки на прерывание по уровню сигнала на входе INTR. Через выход INTA процессор оповещает источник прерывания о том, что заявка принята к выполнению (рис. 16.2).

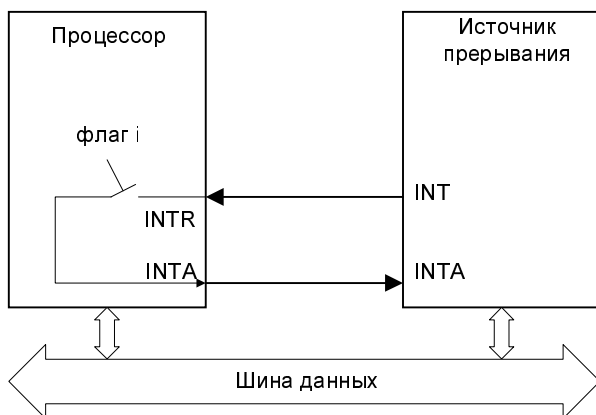


Рис. 16.2. Связь между внешним источником прерываний и процессором

Внешнее устройство выдает запрос на прерывание с выхода INT. Сигнал запроса поступает на вход INTR процессора. Процессор опрашивает этот вход после выполнения текущей инструкции, если не сброшен флаг разрешения внешних прерываний *i*.

Примечание

Для разрешения/запрета внешних прерываний в процессоре предусмотрены команды установки/сброса флага *i* — команды *sti/cli*.

Обнаружив сигнал на входе INTR, процессор посылает подтверждение через выход INTA. Внешний источник прерывания, приняв сигнал с входа INTA, выставляет на шину данных значение *номера прерывания*. Процессор считывает номер прерывания и выполняет "вход в прерывание" следующим образом.

1. Записывает в стек значение регистра флагов, затем полный адрес следующей инструкции (содержимое регистров *cs* и *ip*).

2. Обнуляет флаг *i*, запрещая тем самым прерывания от внешних источников.
3. Считывает двойное слово из таблицы адресов по номеру, полученному от источника прерывания.
4. Выполняет переход по адресу, считанному из таблицы.

На этом процесс прерывания заканчивается. Результат прерывания — переход по адресу из таблицы адресов. По этому адресу должна находиться программа, которая *обслуживает* прерывания: выполняет действия с регистрами устройства в соответствии с причиной прерывания (например, считывает принятые данные).

Затем, по завершении обслуживания, следует выполнить обратный переход — по адресу, сохраненному в стеке в момент входа в прерывание; также должны быть восстановлены флаги процессора. Эти действия выполняются инструкцией *iret*, которая считывает данные из стека в регистры *ip*, *cs*, *flags*.

В рассмотренной схеме аппаратной реализации прерывания подразумевается, что непосредственно к процессору подключен только один источник прерываний. В архитектуре IBM PC различные источники прерываний представлены процессору как *единственный* источник, за счет введения промежуточного элемента — *контроллера прерываний*.

Контроллер прерываний 8259A

Запросы от внешних источников подключены к входам *IRQ0*—*IRQ7* контроллера; с выхода *INT* контроллера на вход *INTR* процессора подается сигнал запроса от наиболее приоритетного источника. При получении ответного сигнала *INTA* контроллер выдает номер прерывания на шину данных; номер образуется суммой *IRQ*-номера с некоторым базовым значением, установленным при инициализации контроллера (рис. 16.3).

При поступлении заявок от разных источников выполняется заявка с наименьшим номером *IRQ*. Имеется возможность выборочно заблокировать заявки от отдельных *IRQ*-входов. Блокировку, или маскирование заявок, а также выбор заявки с наибольшим приоритетом обеспечивают три байтовых регистра (*iMr* — interrupt Mask register, *iRr* — interrupt Request register, *iSr* — interrupt Service register) и арбитр приоритетов (*PR* — Priority Resolver) (рис. 16.4).

Каждая из линий *IRQx* в контроллере подключена к разряду *x* регистра *iRr* через входной вентиль. Вентиль открыт, если в регистре маски *iMr* разряд *x сброшен*. Регистр маски подключен к порту 021; таким образом для разрешения запросов по линии *IRQx* следует обнулить разряд *x* порта 021.

Рассмотрим прохождение сигнала прерывания от внешнего источника, подключенного к входу *IRQx*, через контроллер прерываний. Пусть поступле-

ние запросов по линии IRQ_x разрешено — разряд x в регистре маски iMr сброшен. Если в регистре запросов iRr разряд x сброшен, то по фронту сигнала IRQ_x разряд x регистра iRr будет установлен в единицу. Запрос, в результате, запомнен, и арбитр приоритетов по значению регистра iSr принимает решение о возможности обслуживания запроса.

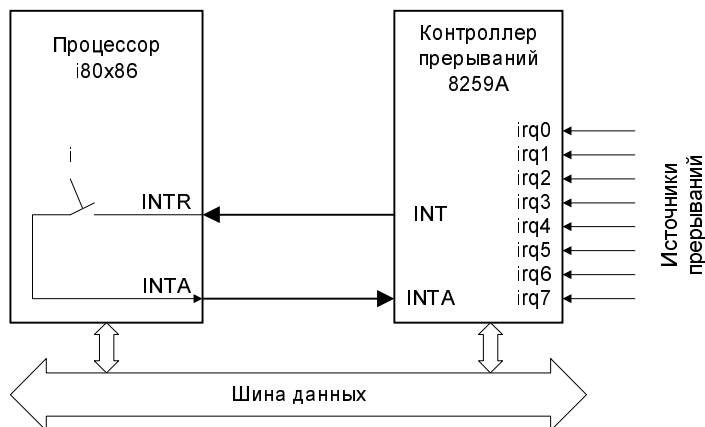


Рис. 16.3. Включение контроллера прерываний между процессором и внешними источниками прерываний (выходы **INT** внешних источников подключены к отдельным входам **IRQ**)

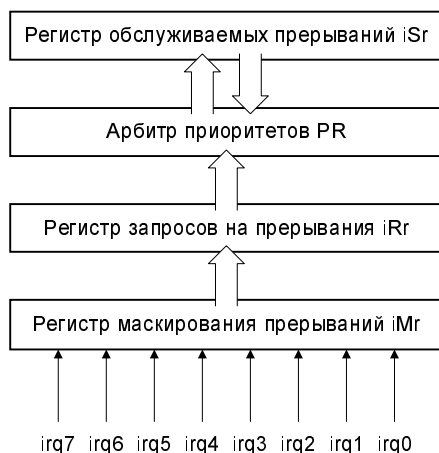


Рис. 16.4. Фрагмент структурной схемы контроллера прерываний

Заявка x из регистра iRr может быть принята к обслуживанию, если в регистре iSr не зафиксировано заявок с равным или большим приоритетом

(от IRQ-входов с номерами, значение которых больше или равно x). Если это условие выполнено, контроллер выставляет сигнал INT. Заявка принимается к обслуживанию при получении ответного сигнала INTA; при этом контроллер сбрасывает бит x в iRr , устанавливает бит x в iSr и выставляет на шину данных номер прерывания, соответствующий заявке x .

Установленный в iSr бит x теперь блокирует заявки с номерами $\geq x$ (в частности, последующие заявки от *того же* входа x). Блокировку необходимо снять — в программе обслуживания прерывания. Сброс бита x в iSr выполняется записью в порт 020 байта со значением 03 x , где x — номер IRQ-входа. На практике предпочитают пользоваться командой *неопределенного* сброса, которая подается записью в порт 020 значения 020. По этой команде в iSr сбрасывается заявка с наименьшим номером — как раз та, что обслуживается в настоящий момент как наиболее приоритетная.

Пример организации прерываний от внешнего источника

В качестве примера настройки и обслуживания прерываний приведем программу, которая подготавливает взаимодействие с адаптером последовательного канала связи в режиме прерываний. Прерывания в этом примере возникают каждый раз при освобождении буферного регистра передатчика. В этой же программе выполняется и обработка прерываний.

Листинг 16.2. Вывод в последовательный канал в режиме прерываний (см. i_send.8)

```
com = 1

#if com EQ 2
    base    equ    02f8      ; базовый адрес COM2
    IRQ     equ    3        ; номер IRQ-входа для COM2
#endif
#if com EQ 1
    base    equ    03f8      ; базовый адрес COM1
    IRQ     equ    4        ; номер IRQ-входа для COM1
#endif

    buf     equ    base      ; адрес порта регистра данных
    ier     equ    base+1    ; адрес регистра разрешения прерываний
    modem   equ    base+4    ; адрес регистра управления модемом

    ni      equ    8+IRQ     ; номер прерывания — это сумма базового
                                ; значения (8) и номера IRQ-входа

    jmp     start

count     dw      1000      ; счетчик передач
```

```

start:
    mov     ds, 0                ; сегментный адрес таблицы
    mov     w [ni*4], offset prog ; запись полного адреса prog
    mov     [ni*4+2], cs         ; в таблицу
    mov     ds, cs              ; восстановление ds

    mov     al, 0b              ; запись 0b в регистр управления
    mov     dx, modem           ; модемом (иначе прерывания
    out     dx, al              ; по готовности передатчика
                                ; не генерируются)

    mov     al, bit 1           ; разрешение прерываний
    mov     dx, ier             ; по готовности
    out     dx, al              ; передатчика

    in      al, 021             ; доступ к iRr для
    and     al, not (bit IRQ)    ; размаскирования заявок от
    out     021, al             ; IRQ-входа с номером IRQ

11:
                                ; основная программа
    mov     ah, 1               ; проверка нажатия клавиши
    int     016
    jnz     >l2                 ; выход из цикла, если нажата
    test    count               ; проверка счетчика передач
    jnz     l1                  ; выход, если обнулен

12:
    mov     dx, ier             ; в завершение – сброс
    mov     al, 0               ; разрешения прерываний
    out     dx, al              ; в регистре ier устройства
    ret

prog:
                                ; п/п обслуживания прерывания
    push    ds, es, ax, dx      ; сохранение регистров
    mov     ds, cs              ; принудительная установка ds = cs

    mov     ax, count           ; отображение младшего байта
    mov     es, 0b800           ; счетчика в верхнем левом углу
    es mov  [0], al             ; экрана – чтобы убедиться, что
                                ; прерывания происходят

    dec     count
    jnz     >l1                  ; последняя передача?
    mov     dx, ier             ; да – запрещаем дальнейшие
    mov     al, 0               ; прерывания
    out     dx, al

11:
                                ; независимо от того, последняя
                                ; передача или не последняя:
    mov     dx, buf             ; запись в регистр данных
    out     dx, al              ; передатчика – запуск передачи

```

mov	al, 020	; сброс заявки в регистре iSr
out	020, al	; контроллера прерываний
pop	dx, ax, es, ds	; восстановление регистров
iret		; возврат к прерванной программе

Текст программы приведен в файле `i_send.8`. Выполните трансляцию и запустите программу. Обратите внимание на символ в левом верхнем углу экрана — каждый раз при входе в прерывание этот символ изменяется.

Примечание

Важный момент в обработке аппаратных прерываний — это создание условий для обслуживания следующей заявки. Мало сбросить заявку в контроллере прерываний, необходимо также *снять причину* прерывания в самом устройстве. В рассмотренном примере причина — это готовность передатчика. Чтобы снять ее, нужно задать работу передатчику, записав что-нибудь в его регистр данных.

По аналогии с примером из листинга 16.2 запрограммируйте *прием* данных в режиме прерываний. Поскольку в диагностическом режиме последовательного адаптера прерывания по готовности *приемника* не генерируются, требуется внешний передатчик — например, мышь, подключенная к СОМ-адаптеру. При перемещении, при нажатии и отпускании кнопок мышь выполняет передачу трех или пяти байт (зависит от типа мыши). Предварительно на мышь подается питание — записью числа 0b в регистр управления модемом (что, к примеру, сделано в программе `i_send.8`, правда, с другой целью). Перед тем, как составлять программу, при помощи редактора `gr0t` проверьте работоспособность мыши: чтение порта по смещению 0 при перемещении мыши дает разные результаты.

Для разрешения прерываний по готовности *приемника* в регистре разрешения прерываний последовательного адаптера (в листинге 16.2 адрес этого регистра назван именем `ier`) установите бит 0, остальные обнулите. Причина прерывания в данном случае — готовность приемника; для *снятия причины* следует прочитать регистр данных — в процедуре обработки прерывания.

Запрограммируйте обработку прерываний по *ошибке приема*. Ошибка *неполнения* получается при перемещении мыши, когда принимаемые от нее данные не считываются из регистра данных приемника. Чтобы блокировать чтение данных, запретите прерывания по готовности *приемника* (нулевой бит в `ier` должен быть сброшен). Для разрешения прерывания по ошибке приема установите в `ier` бит 2. При обработке прерывания по ошибке причина прерывания снимается, в общем случае, чтением регистра *состояния* и регистра *данных*. В результате чтения регистра состояния в этом регистре сбрасываются биты ошибок, а при чтении регистра данных обнуляется бит готовности приемника.

Каскадное включение контроллеров прерываний

В IBM PC/AT количество внешних источников прерываний превышает восемь, и для их подключения одного контроллера 8259A недостаточно. Поэтому введен второй, *ведомый* контроллер, выход INT которого подключен к входу IRQ3 *ведущего* контроллера, как показано на рис. 16.5. Регистр `IMR` ведомого контроллера доступен через порт 0a1, а регистр команд — через порт 0a0.

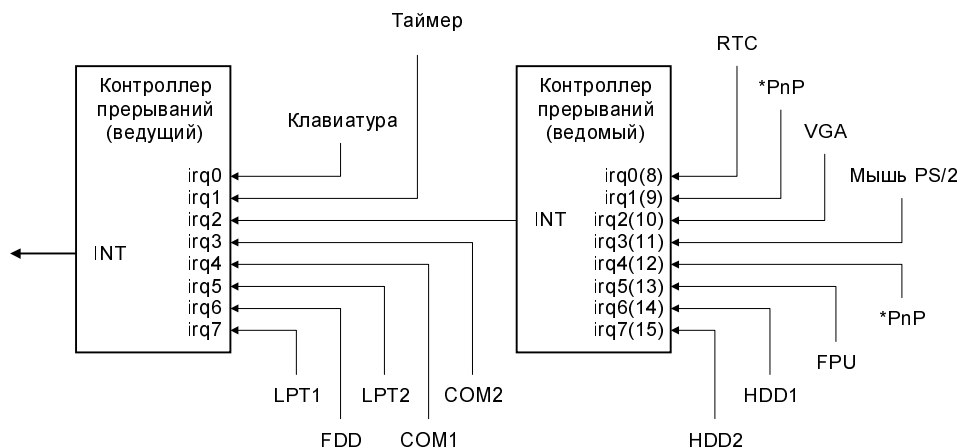


Рис. 16.5. Каскадное включение контроллеров прерываний в IBM PC/AT (PnP — устройство Plug & Play)

Для обслуживания устройства, подключенного к ведомому контроллеру, следует дополнительно открыть вход `IRQ2` *ведущего* контроллера; а в процедуре обработки прерывания — предусмотреть сброс заявок в обоих контроллерах.

В завершение, рассмотрим аппаратно-программный способ определения временных характеристик работы устройств, с использованием подпрограмм модуля измерения временных интервалов `pztimer` из книги М. Abrash "Zen of Assembly Language".

16.5. Измерение временных характеристик устройства

Модуль `pztimer` (Precision Zen Timer) предназначен для измерения интервалов времени в пределах 55 мсек с погрешностью в пределах 2 мкс. В качестве счетчика времени используется нулевой канал трехканального таймера 8253/8254.

Подпрограмма для включения отсчета времени `zTimerOn` переводит нулевой канал таймера в режим однократного счета, загружает в регистр счетчика максимальное значение и запускает счет; дополнительно, сбрасывается флаг `i` — для того, чтобы процедуры обслуживания прерываний не вносили погрешности в измерение.

После вызова `zTimerOn` программа переходит к отработке исследуемого фрагмента кода. Пока этот фрагмент выполняется, значение в счетчике нулевого канала уменьшается независимо от программы с периодом приблизительно равным 820 нсек (частота счета приблизительно равна 1193 МГц). Вызов `zTimerOff` в конце исследуемого фрагмента прекращает счет, сохраняя текущее значение регистра счета во внутренней переменной модуля `pztimer`. При последующем вызове `zTimerReport` запомненное значение выводится на экран, с преобразованием в микросекунды. Если между вызовами `zTimerOn` и `zTimerOff` прошло более 55 мсек (за это время канал 0 досчитал до нуля и остановился), подпрограмма `zTimerReport` сообщит о переполнении таймера.

Набор подпрограмм для измерения временных интервалов содержится в файле `pztimer.8`. Этот файл транслируется вместе с главной программой; для его подключения запишите в конце текста главной программы директиву:

```
include pztimer.8
```

В качестве примера определим настройку скорости последовательного канала связи. Для этого измерим интервал времени между запуском передачи и ее завершением, отслеживая состояние передатчика в режиме периодического опроса.

Листинг 16.3. Измерение времени передачи по последовательному каналу (см. `rate.8`)

```
base    equ    03f8        ; базовый адрес COM1
buf      equ    base        ; буферный регистр приемника/передатчика
state    equ    base+5      ; регистр состояния

call     zTimerOn
mov      dx, buf            ; запуск передачи
out      dx, al             ; что передаем — не имеет значения
mov      dx, state

m1:      in      al, dx
test     al, bit 6          ; передача закончена?
jz       m1

call     zTimerOff
call     zTimerReport
ret

include  pztimer.8
```

Запись в порт со смещением ноль автоматически запускает передачу записанного байта. По окончании передачи устанавливается единица в бите 6 регистра состояния, доступного через порт со смещением 5.

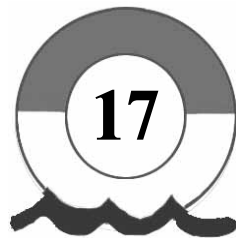
Исходный текст содержится в файле `rate.8`. Выполните несколько раз эту программу и оцените скорость передачи, исходя из того, что в одном цикле передается (по умолчанию) десять бит: стартовый, восемь информационных и один стоповый. Скорость передачи настроена на значение из стандартного ряда: 150, 300, 600, 1200, 2400, 4800, 9600, 19 200 и т. д. до 115 200 бод (бит/с).

Запрограммируйте опыт по определению задержки между записью в буферный регистр передатчика и его освобождением. Передача данных из буферного регистра в сдвиговый выполняется не мгновенно, а на очередном *такте* устройства (интервал между тактами равен длительности передачи одного бита). Так как момент запуска программы никак не связан с тактами устройства, при повторных измерениях следует ожидать значительного разброса результатов.

Определите темп передачи данных мышью — число посылок в единицу времени при передаче пакета из трех или пяти байт. В предположении, что данные в пределах пакета передаются равномерно, достаточно измерить интервал между первой и второй посылкой: включить счет по готовности приемника и выключить по ошибке переполнения приемника.

Примечание

Исследование любого устройства следует начинать из некоторого исходного состояния. В частности, опыты с приемником последовательного канала связи начинаются с чтения регистров состояния (сброс ошибок) и данных (сброс готовности приемника).



Прерывания и исключения

В *главе 16* прерывания рассмотрены в общих чертах, на примере управления внешним устройством. В этой главе изучаются прерывания всех типов; примеры разработки процедур обслуживания прерываний, в частности, с учетом системных процедур, разобраны в деталях.

17.1. Прерывания и векторные вызовы подпрограмм

Применительно к прерываниям от внешних устройств, в *главе 16* были рассмотрены следующие действия, связанные с обработкой прерываний:

1. Настройка процедуры обработки прерывания: запись адреса процедуры в системную таблицу адресов по индексу, равному номеру прерывания.
2. Создание условий для выполнения заявки на прерывание:
 - разрешение прерываний в регистре устройства;
 - открытие линии `IRQ` контроллера прерываний, к которой подключен запрос от устройства;
 - установка флага разрешения внешних прерываний — `i`.
3. Выполнение прерывания процессором:
 - сохранение регистра флагов и дальнего адреса возврата;
 - сброс флагов разрешения прерываний (`i`) и трассировки (`t`);
 - передача управления по таблице адресов.
4. Обслуживание прерывания в процедуре его обработки:
 - чтение/запись данных устройства или сброс ошибок — в соответствии с событием, вызвавшим прерывание;
 - сброс заявки в регистре `iSr` контроллера прерываний.

Прерывание подобно вызову подпрограммы: управление на процедуру обработки прерывания передается с сохранением в стеке адреса возврата. Вместе с тем, прерывание отличается от подпрограммного вызова: передача управления не запрограммирована, она вызывается исполнительным устройством.

Остальные различия не принципиальны, они отражают особенности организации процессора i80x86:

- ❑ при входе в прерывание процессор i80x86 сохраняет в стеке не только адрес возврата, но и значение *регистра флагов* (напротив, в микроконтроллерах Intel MCS48/51 сохраняется только адрес возврата, а сохранение и восстановление флагов программируется в процедуре обработки прерывания);
- ❑ адрес процедуры обработки прерывания в i80x86 задан в специальной таблице, расположенной в памяти по фиксированному адресу (напротив, в MCS48/51 фиксировано не местоположение данных, содержащих адрес перехода, а сам адрес).

Примечание

В микроконтроллерах MCS48/51 прерывание и вызов подпрограммы выполняются абсолютно одинаковым образом. Разница лишь в источнике: для подпрограммы — это явно заданная команда `call`, а для прерывания — незапрограммированное событие в устройстве.

На рис. 17.1 показано, как выполняется вход в прерывание в MCS51 (рис. 17.1, *а*) и в i80x86 (рис. 17.1, *б*). Система прерываний, принятая в i80x86, называется *векторной*. Вектор прерывания — это стартовая информация для процедуры обработки прерывания, включающая в себя адрес перехода и расположенная в таблице векторов, которая находится в памяти по известному адресу.

Примечание

В предыдущем разделе мы избегали общепринятого понятия "вектор", заменив его более узким понятием "адрес перехода". Хотя для i80x86 вектор и адрес перехода — одно и то же, в общем случае, вектор может включать в себя дополнительную информацию. Например, в системах архитектурной линии DEC вектор содержит также начальное значение регистра флагов, помимо адреса перехода. В i80x86 при входе в прерывание значение `flags` не устанавливается полностью: только обнуляются `i` и `t`.

Вызов подпрограммы по номеру вектора используется не только для обслуживания прерываний от устройств. В системах, использующих векторные прерывания, как правило, предусмотрена возможность непосредственного обращения к подпрограмме по номеру вектора. Вспомните, как программируется обращение к DOS. Инструкция `int 021` — это прямое обращение к процедуре, адрес которой задан в таблице векторов по индексу 021.

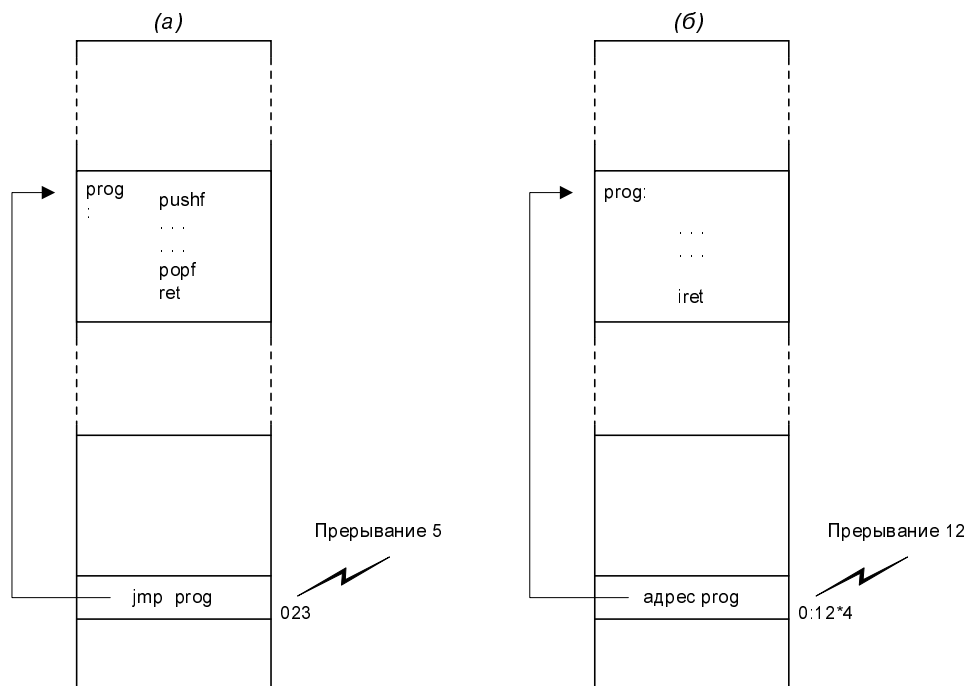


Рис. 17.1. Вызов процедуры обработки прерывания в MCS51 (а) и в i80x86 (б)

Еще один пример таких вызовов — аварийное завершение программы при выполнении инструкций `div/idi`, когда значение частного превышает диапазон приемника (так называемое переполнение при делении). Команда `div/idi` при возникновении такой ошибки неявно вызывает процедуру, адрес которой задан вектором 0.

Процедуры, связанные с векторами 021 и 0, составляют часть операционной системы. В целом, весь системный сервис BIOS-DOS образован такими *резидентными* процедурами обработки прерываний; одна часть (в основном, в составе BIOS) обслуживает внешние прерывания от устройств, входящих в минимальную конфигурацию IBM PC (например, от клавиатуры), а другая занимается обслуживанием *внутренних* прерываний, примеры которых мы привели.

Таким образом, в системах с векторными прерываниями механизм передачи управления по номеру вектора используется двояко:

- для асинхронной передачи управления по сигналу прерывания (`interrupt`) от устройства;
- для синхронной передачи управления — либо по команде `int <n>`, выполняющей безусловный вызов с явным указанием номера вектора `<n>`

в диапазоне 0—255, либо по результатам отдельных команд, генерирующих условный вызов с аппаратно фиксированным номером вектора.

Примечание

Синхронные вызовы подпрограмм через таблицу векторов называются *внутренними* прерываниями, или, согласно документации Intel, исключениями (exception). Исключение — не очень удачный термин; в DEC-системах синхронный векторный вызов обозначается термином "trap" — ловушка (есть даже одноименная инструкция), в противоположность внешнему прерыванию (обозначается термином "interrupt").

В отличие от внешних прерываний, внутренние прерывания не могут быть запрещены. Инструкция `int 021` и переполнение частного при отработке `div/idi` всегда приводят к вызову соответствующих процедур. Флаг `i` процессора блокирует только внешние прерывания.

Независимо от того, является ли прерывание внешним или внутренним, процессор входит в прерывание одинаковым образом: сохраняет регистры `flags`, `cs` и `ip` в стеке, обнуляет флаги `i` и `t` и передает управление по таблице векторов.

17.2. Типы исключений

Внутренние прерывания, или исключения, согласно документации Intel, подразделяются на три типа:

- ☐ Fault (сбой);
- ☐ Trap (ловушка);
- ☐ Abort (фатальная ошибка).

Тип исключения определяет различия в информации об адресе возврата, сохраняемой при входе в исключение.

- ☐ Trap-исключения, возникающие при выполнении команд `int <n>`, сохраняют в стеке адрес следующей инструкции — по аналогии с вызовом подпрограммы.
- ☐ Fault-исключения, возникающие, например, при переполнении результата команды `div/idi`, сохраняют в стеке адрес текущей инструкции, т. е. той самой инструкции, которая вызвала исключение. Выполнение `iret`, в итоге, приводит к повторению команды и к новому вызову исключения — до тех пор, пока в процедуре обработки исключения ситуация не будет исправлена.
- ☐ Abort-исключения возникают при невосстановимых системных сбоях, когда продолжение или повторная попытка невозможны; значение адреса возврата в стеке не определено.

Примечание

Большинство исключений типа `fault` и все исключения типа `abort` генерируются только в защищенном режиме процессора `i80x86`.

17.3. Итоговая классификация прерываний

На рис. 17.2 показан один из вариантов классификации прерываний с учетом терминологии, принятой в документации Intel.

Типы внутренних прерываний — `fault`, `trap`, `abort` — рассмотрены в предыдущем разделе. Трар-исключения, в свою очередь, можно разделить на программные — по инструкции `int <n>` — и на условные, например, исключение 4, возникающее при выполнении инструкции `into` в том случае, если установлен флаг знакового переполнения `o`.

Примечание

К условным прерываниям также можно отнести отладочное исключение номер 1, которое вызывается после каждой команды, если установлен флаг трассировки. В `i80486` исключение 1 вызывается также при совпадении линейного адреса операнда или инструкции с содержимым одного из отладочных регистров `dr0—dr3`, доступных для программной записи.

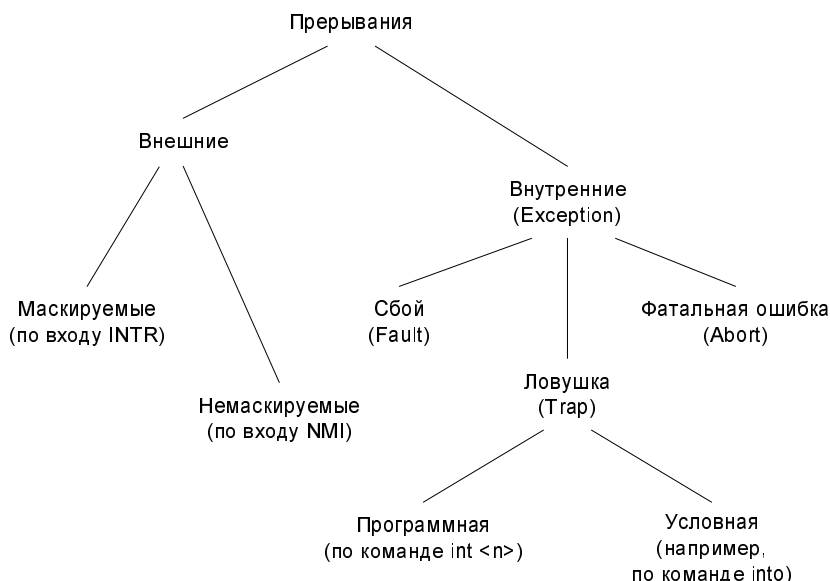


Рис. 17.2. Типы прерываний в `i80x87`

В схему на рис. 17.2 включены немаскируемые внешние прерывания. Напомним, прерывание может быть замаскировано процессором (за счет сброса флага *i*), если сигнал запроса на прерывание подключен к входу INTR процессора i80x86. В архитектуре IBM PC к входу INTR подключен выход INT ведущего контроллера прерываний. Напротив, запрос через дополнительный вход NMI (Non-Maskable Interrupt) процессора не маскируется и обслуживается сразу, независимо от установки флага *i*. К входу NMI обычно подключаются сигналы фатальных системных ошибок, например сигнал ошибки четности при обращении к памяти.

Примечание

В IBM PCXT к входу NMI были подключены (через схему логики NMI) сигналы сбоев при доступе к памяти и портам, и сигнал запроса на прерывание по особым ситуациям i8087 (по-видимому, из-за недостатка в IRQ-входах при наличии только одного контроллера прерываний). С развитием семейства i80x86 сигналы системных сбоев были перенесены внутрь процессора (исключение 18); варианты подключения сигнала прерываний по особым ситуациям i80x87 рассмотрены в *приложении 8*.

17.4. Программирование исключений

Подробное рассмотрение техники программирования прерываний и исключений начнем с последних — с ними проще, т. к. не требуется создавать условий для генерирования и прохождения сигнала запроса на прерывание.

17.4.1. Trap-исключения

Продemonстрируем обработку внутренних прерываний на примере условного trap-исключения 4. Это исключение генерируется при выполнении инструкции `into`, если установлен флаг знакового переполнения (`o = 1`).

Листинг 17.1. Пример обработки trap-исключений (см. trap.8)

```
print    macro
        push    ax, dx
        lea     dx, #1
        mov     ah, 9
        int     021
        pop     dx, ax
#em

        jmp     start

msg      db      '?-Overflow', 13, 10, '$'
```

```

n_vect equ    4

start:
    mov     ds, 0
    mov     w [n_vect * 4], offset new
    mov     w [n_vect * 4 + 2], cs
    mov     ds, cs

    mov     al, 07f
    add     al, 1                ; o = 1
    into                                ; (1)

    int     4                    ; (2)

    int     020

new:
    print   msg
    iret

```

В таблицу векторов записывается, со смещения $n_vect * 4$, дальний указатель на процедуру `new`, которая находится в той же программе. В этой процедуре заданы вывод сообщения и возврат — `iret`. Поскольку в стеке при входе в прерывание сохранен адрес *следующей* инструкции, то после выполнения `iret` прерванная программа продолжается дальше.

Процедура `new` в листинге 17.1 вызывается как по условию (1), так и напрямую (2) — как программное прерывание. Реакция `new` в обоих случаях одинаковая, поскольку причина вызова в ней не определяется.

Выясним причину исключения, находясь в процедуре обработки. В данном случае достаточно определить, какая инструкция находится по адресу, предшествующему адресу возврата.

Листинг 17.2. Определение причины исключения при его обработке (см. know_1.8)

```

...

_into db      '?-Overflow', 13, 10, '$'
_int4 db      'Joke (immediate int 4)', 13, 10, '$'

...

sample db
into          ; 1 byte

new:
    enter    0
    push     ax, es, di

```

```

les      di, [bp+2]      ; (1)
es mov   al, [di-1]      ; previous byte
cmp      al, sample
je       >l1             ; into?
print    _int4
jmp      >l2
11:      print    _into
12:      pop      di, es, ax
         leave
         iret

```

Инструкция, вызвавшая trap-исключение, находится над инструкцией, адрес которой сохранен в стеке. Чтение стека выполняется аналогично доступу к параметрам подпрограммы: регистр `bp`, после сохранения в стеке, устанавливается равным `sp`, после чего элементы стека доступны через указатель `bp` по смещениям, показанным на рис. 17.3.

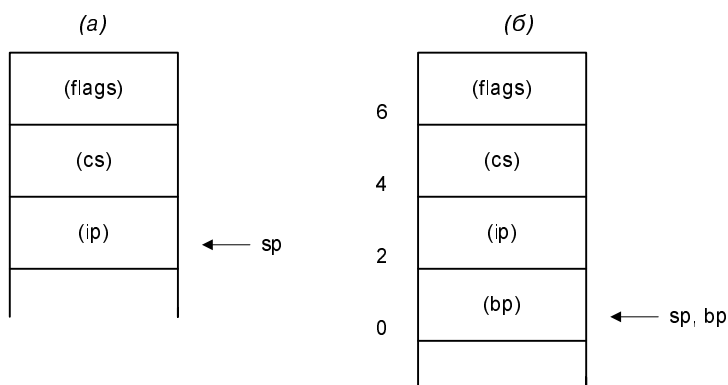


Рис. 17.3. Кадр стека при входе в прерывание — до (а) и после (б) выполнения `enter 0`

Команда (1) считывает из стека полный адрес возврата в пару регистров `es:di`. Решение принимается по результату сравнения содержимого байта по адресу `es:[di-1]` с образцом (`sample`) однобайтной инструкции `into`.

17.4.2. Fault-исключения

При обработке `fault`-исключения следует либо завершить программу, либо исправить ситуацию, изменив условия выполнения той команды, которая вызвала ошибку. Иначе программа заикнется на ошибочной команде, поскольку при входе в `fault`-исключение стек содержит адрес этой текущей команды.

Обработку fault-исключений в обоих вариантах рассмотрим на примере команды `bound`. Команда `bound <reg16>, <mem32>` вызывает fault-исключение по вектору пять, если знаковое значение 16-разрядного регистра `<reg16>` находится вне диапазона, определенного парой слов по адресу `<mem32>`. Команда предназначена для проверки индексов при доступе к массивам в языках высокого уровня.

Листинг 17.3. Пример обработки fault-исключения (см. fault.8)

```
...

_error db      '?-Bounds error', 13, 10, '$'
_abort db      'Task aborted', 13, 10, '$'
_exit  db      'Normal shutdown', 13, 10, '$'
range  dw      -1, 3

n_vect equ     5

start:
    ...

    mov     cx, -5
    bound   cx, range
    print   _exit
    int     020

new:
    print   _error
#if stop
    print   _abort
    mov     ax, 04c01      ; exit
    int     021
#else
    inc     cx             ; -4, -3, -2, -1
    iret
#endif
```

Процедура `new` в варианте `stop` завершает выполнение задачи. В варианте `!stop` (по умолчанию) процедура `new` пытается исправить значение в проверяемом регистре, увеличивая его на единицу. При начальном значении `-5` исключение будет вызвано четыре раза, до тех пор, пока значение не попадет в диапазон `-1—+3`.

Примечание

Приведенный выше вариант с коррекцией ошибочного значения практически бессмысленный — это лишь демонстрация. Процедура в составе отладчика

для языка высокого уровня должна была бы, как минимум, выяснить, какой именно регистр задан в коде `bound`, и предложить пользователю исправить значение этого регистра.

17.4.3. Различие между `fault`- и `trap`-исключениями

Разница между `fault`- и `trap`-исключениями проявляется при попытке определить тип исключения в процедуре обработки. Следующий пример для определения причины исключения 5 составлен по аналогии с примером из листинга 17.2; образец для сравнения — двухбайтная команда `int 5`.

Листинг 17.4. Попытка идентификации исключений разных типов (см. `know_2.8`)

```
...

_bound db      '?-Bounds error', 13, 10, '$'
_int5   db      'Joke (immediate int 5)', 13, 10, '$'

n_vect equ      5

limit   dd      1, 8

start:
    ...

#if mess
    mov     ax, 0
    int     5                ; (1)
    bound   ax, limit        ; (2)
#else
    int     5                ; (1)
    mov     ax, 0
    bound   ax, limit        ; (2)
#endif
    int     020

sample  dw
    int     5                ; 1 word

new:
    enter   0
    push    ax, es, di
    les     di, [bp+2]
    es mov  ax, [di-2]        ; previous word
    cmp     ax, sample
    je      >11              ; int 5?
```

```

print    _bound
mov      ax, 04c01      ; abort
int      021

```

```

11:      print    _int5
        pop      di, es, ax
        leave
        iret

```

Процедура `new` анализирует код команды по адресу, предшествующему адресу возврата. В варианте `mess` адрес возврата при вызовах (1) и (2) одинаков — это адрес команды `bound`, поскольку (1) вызывает `trap`-исключение, а (2) — `fault`-исключение. В результате, в варианте `mess` программа заикливается на выводе сообщения "Joke (immediate int 5)".

Как бы мы ни меняли логику `new`, найдется такое сочетание `int 5` и `bound`, что идентификация будет ошибочной.

Примечание

В IBM PC исключение 5 вызывается командой `int 5` из BIOS-процедуры обработки прерывания от клавиатуры — по нажатии комбинации клавиш `<Shift>+<PrtSc>`. Это оказалось неприятным сюрпризом для Intel при выпуске `i80186`, снабженного новой командой `bound`. Как видно из рассмотренных примеров, совместное использование вектора `trap`- и `fault`-исключением порождает проблемы с определением источника прерывания. В итоге, вызов исключения 5 по команде `bound` в BIOS-DOS не предусмотрен вовсе.

17.4.4. Практикум по `trap`- и `fault`-исключениям

По аналогии с рассмотренными примерами запрограммируйте следующие исключения:

- ☐ `fault`-исключение 0 при выполнении `div/ldiv`, когда частное не помещается в приемник, например, при делении на ноль;
- ☐ `trap`-исключение при выполнении `int 3`;
- ☐ `trap`-исключение 1 при установленном флаге трассировки `t`.

Примечание

Отладка прерываний и исключений обычно приводит к конфликтам, в первую очередь, с теми исключениями, которые устанавливает сам отладчик. По этой причине отладка *отладочных* исключений 1 и 3 невозможна в принципе. В общем случае, пошаговый режим работает устойчиво только в начале программы — до включения прерываний. Наименее опасный вариант отладки — выполнение фрагментов между постоянными точками останова (напомним, постоянные точки устанавливаются командой `b`, а выполнение программы между ними задается командой `g`).

Дадим пояснения к третьему заданию. Исключение 1 генерируется процессором после выполнения текущей инструкции, если в начале ее выполнения флаг $t = 1$. Специальных команд для установки и сброса флага t не предусмотрено, поэтому эти операции выполняются следующим образом:

- ❑ `flags` копируется в `ax`;
- ❑ в `ax` устанавливается или сбрасывается бит 8, соответствующий биту t в `flags`;
- ❑ `ax` копируется в `flags`.

Содержимое `flags` считывается в `ax` парой команд:

```
pushf
pop    ax
```

Обратная запись `ax` в `flags` выполняется командами:

```
push    ax
popf
```

При выполнении задания следует, установив вектор 1 на процедуру обработки исключения, взвести флаг t , а затем, выполнив небольшой цикл, сбросить t . В процедуре обработки исключения предусмотрите вывод на экран символа '*'. Сброс флага t выполните в двух вариантах.

1. В основной программе (в результате, сами инструкции для сброса t включаются в трассировку).
2. При обработке исключения (например, при первом же входе в исключение).

Частая ошибка при выполнении второго варианта следующая: в процедуру обработки исключения переносят последовательность инструкций для сброса t из основной программы первого варианта. Программировать сброс t в процедуре обработки — это лишнее, т. к. при входе в исключение t сброшен процессором автоматически. Исходное значение $t = 1$ сохранено в стеке вместе с содержимым регистра флагов, и будет восстановлено на выходе командой `iret`. Чтобы t был сброшен при возврате в главную программу, следует скорректировать значение флагов, сохраненное в стеке.

Примечание

Исключения 1 и 3 относятся к отладочным исключениям. Применение первого очевидно — организация пошагового режима отладки. С помощью `int 3` отладчик устанавливает точки останова, заменяя инструкцию по адресу останова на `int 3`. Это однобайтная инструкция, в отличие от остальных `int`-инструкций; поэтому `int 3` возможно записать на место команды любой длины, не затронув следующую команду.

17.5. Обработка прерываний при наличии системной процедуры

Для большинства исключений операционная система предусматривает резидентную процедуру обработки, хотя бы в виде "заглушки" из единственной команды `iret`. В ранее рассмотренных примерах мы произвольно изменяли векторы, разрывая связь между исключением и системной процедурой его обработки. В последующих примерах вектора восстанавливаются в программе перед ее завершением.

Примечание

В приведенных примерах мы "испортили" системные значения векторов 4 и 5, сделав невозможным вызов `into` и вывод копии экрана на принтер при нажатии комбинации клавиш `<Shift>+<PrtScrn>`. Поскольку память, занимаемая программами из файлов `except/known`, была уже несколько раз освобождена и вновь занята (если, конечно, вы их выполняли в текущем сеансе DOS), то по адресам, которые мы записали в таблицу векторов, находится случайный код (точнее, просто неадекватный), обращение к которому, вероятней всего, приведет к катастрофе.

17.5.1. Сохранение и восстановление векторов

В качестве примера рассмотрим замену и восстановление вектора прерывания 021, посредством которого прикладные программы обращаются к DOS.

Листинг 17.5. Сохранение и восстановление системного вектора (см. `save_1.8`)

...

```
msg_1  db      7, 'First print', 13, 10, '$'
msg_2  db      7, 'Second print', 13, 10, '$'

n_vect equ     021
old_v   dd      ?

start:

    lea     ax, new                ; (1)
    mov     dx, cs                 ; (2)

    mov     ds, 0
    xchg    w [n_vect * 4], ax     ; (3)
    xchg    w [n_vect * 4 + 2], dx ; (4)
    mov     ds, cs

    mov     w old_v, ax            ; (5)
    mov     w old_v + 2, dx        ; (6)
```

```

print    msg_1                                ; (7)

mov      ax, w old_v                          ; (8)
mov      dx, w old_v + 2                      ; (9)

mov      es, 0                                ; (10)
es mov   w [n_vect * 4], ax                   ; (11)
es mov   w [n_vect * 4 + 2], dx               ; (12)

print    msg_2
mov      ax, 04c00
int      021

```

new:

```

push     ds
mov      ds, 0b800
mov      w [0], 0f by '*'
pop      ds
iret

```

Запись нового значения вектора и сохранение прежнего значения выполняются одновременно за счет использования команд `xchg`. Команды (1) и (2) записывают в пару регистров `dx:ax` новое значение; после обнуления регистра `ds`, регистры `dx:ax` обмениваются (3, 4) значениями с парой слов вектора `n_vect`. Полученное в `dx:ax` системное значение сохраняется (5, 6) в памяти программы (предварительно восстанавливается `ds = cs`). В результате, системное значение вектора сохранено в двойном слове `old_v`, а в таблицу векторов по номеру 021 записан адрес процедуры `new`. Теперь запрос (7) на выполнение функции DOS передается в процедуру `new`, которая выводит '*' и возвращает управление.

После проверки исключения 021 значение системного вектора восстанавливается. В пару `dx:ax` записывается (8, 9) сохраненное значение из `old_v`, затем `dx:ax` копируется в вектор 021; для разнообразия, доступ к таблице векторов выполняется не через `ds`, а через `es` (10—12). Последующие запросы на выполнение функций DOS отправляются по назначению — в системную процедуру.

Примечание

Вызов функций DOS из процедуры обработки исключений, практикуемый в ранее рассмотренных примерах, здесь невозможен — он привел бы к рекурсии: процедура обработки исключения 021 вызывала бы исключение 021.

Во всех рассмотренных примерах доступ к таблице векторов выполняется напрямую, за счет изменения текущего основного или дополнительного сегмента данных (`ds` или `es`). При работе в DOS чтение и запись векторов может быть выполнено также функциями 035 и 025.

Номер вектора при вызове функций 025 и 035 задается в `al`. Результат функции 035 — значение вектора — возвращается в паре `es:bx`. При записи вектора функцией 025 новое значение должно быть задано в паре `ds:dx`.

В качестве иллюстрации рассмотрим установку пользовательской процедуры обработки `fault`-исключения 5.

Листинг 17.6. Применение функций чтения/записи вектора (см. `save_2.8`)

```
...

_error db    '?-Bounds error', 13, 10, '$'
_exit  db    'Task completed', 13, 10, '$'

range  dw    -1, 3

n_vect equ    5
old_v  dd    ?

start:
    mov     ax, 035 by n_vect
    int     021                      ; es:bx
    mov     w old_v, bx
    mov     w old_v + 2, es          ; save

    mov     ax, 025 by n_vect
    lea     dx, new                  ; (1)
    int     021

    bound   ax, range

exit:
    mov     ax, 025 by n_vect

#if silly
    mov     dx, w old_v              ; (2)
    mov     ds, w old_v + 2          ; (3)
#else
    lds     dx, old_v                ; (4)
#endif

    int     021
    mov     ds, cs                  ; (5)

    print   _exit
    int     020

new:
    print   _error
    jmp     exit                    ; (6)
```

В начале программы функцией 035 с `al = 5` считывается системное значение вектора 5, и результат, полученный в `es:bx`, сохраняется в `old_v`. Перед вызовом функции 025 в `ds:dx` должен быть записан полный адрес `new`. Команда (1) записывает смещение в `dx`. Сегментный адрес `new`, который требуется задать в `ds`, — это содержимое `cs`; сейчас, после запуска `com-программы`, `ds = cs`, так что `ds` для вызова функции 025 уже, оказывается, установлен.

Процедура обработки `fault`-исключения рассчитана на аварийное завершение программы. В подобных случаях можно позволить себе прямую передачу управления (6) в основную программу. При любом исходе команды `bound`, программа попадает на метку `exit`, где восстанавливается вектор.

Восстановление вектора при помощи функции 025 требует осторожности, ввиду необходимого изменения `ds`. В варианте `silly` после выполнения (3) доступ к данным программы через `ds` расстроен. Стоит лишь поменять местами команды (2) и (3), чтобы последствия изменения `ds` проявили себя: по смещению `old_v` будет считано значение не из программного сегмента текущей задачи, а из сегмента кода системной процедуры обработки прерывания — в соответствии с текущей настройкой `ds`! Во избежание этой ситуации пару (2, 3) рекомендуется заменять одной командой (4), а после вызова функции 025 как можно скорей восстанавливать (5) исходную установку `ds = cs`.

На время записи вектора функция 025 сбрасывает флаг `i`, блокируя возможный вызов внешнего прерывания в момент, когда вектор сформирован лишь наполовину (т. е. в промежутке между парой команд записи вектора). При записи вектора напрямую также рекомендуется на время записи обнулять `i` — по крайней мере, если вектор связан с внешним прерыванием:

```
pushf
cli
...           ; write vect
popf
```

Примечание

Пары команд `pushf-cli/popf` позволяют организовать прохождение критических участков программы, когда переключение выполнения в результате внешнего прерывания недопустимо. Запись вектора — один из примеров критического участка.

Почему рассмотренный способ восстановления векторов неприменим в примере из листинга 17.5?

17.5.2. Дополнение к установленной процедуре обработки прерывания

В рассмотренных примерах пользовательская процедура обработки прерывания отключает ранее установленную (системную) процедуру; в результате, обработка полностью выполняется новой процедурой. При разработке программного обеспечения в отсутствие операционной системы — это единственный вариант. Напротив, расширение функций работающей операционной системы (BIOS-DOS) предполагает возможность вызова прежней (системной) процедуры из вновь установленной процедуры.

Для этого новая процедура передает управление по адресу, сохраненному в памяти при чтении вектора, одним из следующих способов:

- ❑ на выходе, командой `jmp` взамен `iret`;
- ❑ из произвольной точки, парой команд `pushf-call`.

В первом варианте вновь созданная процедура передает управление *по цепи*, после того, как она полностью завершила свою работу. Во втором варианте управление после подпрограммного вызова системной процедуры возвращается в пользовательскую процедуру, т. е., новая процедура *пропускает* системную процедуру вперед себя.

Продемонстрируем встраивание процедуры обработки прерывания на примере исключения 4, взяв за основу пример из листинга 17.1. Сначала выполните тест, заданный в программе `into_1.8`, чтобы выяснить поведение системной процедуры.

Так как системная процедура никак себя не проявляет, а работа DOS при этом не нарушается, то, скорее всего, системная процедура представляет собой всего лишь заглушку из одной команды `iret`. В качестве системной процедуры установим резидентную процедуру из файла `into.8`.

Программа `into.8` будет подробно рассмотрена в *разд. 18.1*. Пока что, не вдаваясь в особенности этой программы, выполните трансляцию и запустите `into.com`, а затем проверьте реакцию системы при помощи теста `into_1`. Установленная теперь системная процедура проявляет себя выводом сообщения об ошибке.

Листинг 17.7. Дополнение к системной процедуре обработки прерывания (см. `chain.8`)

...

```
msg      db      '?-Overflow', 13, 10, '$'
```

```
n_vect  equ      4
```

```
old_v   dd        ?
```

```

start:
    lea    ax, new
    mov    dx, cs
    mov    es, 0
    es xchg w [n_vect * 4], ax           ; set & save
    es xchg w [n_vect * 4 + 2], dx
    mov    w old_v, ax
    mov    w old_v + 2, dx

    mov    al, 07f
    add    al, 1
    into

    mov    ax, w old_v                 ; restore
    mov    dx, w old_v + 2
    es mov w [n_vect * 4], ax
    es mov w [n_vect * 4 + 2], dx

    int    020

new:
#ifdef subr
    pushf                ; (1)
    call    old_v         ; (2)
#endif
    print   msg
#ifdef chain
    jmp     old_v         ; (3)
#else
    iret
#endif

```

Процедура `new` реализована в нескольких вариантах. В варианте по умолчанию она выполняется в монопольном режиме — выводит сообщение и возвращает управление командой `iret`. В варианте `chain` процедура `new` выполняет не `iret`, а дальний косвенный переход по адресу, заданному содержимым двойного слова `old_v`. При выполнении (3), `ip` устанавливается равным значению младшего слова из двойного слова `old_v`, а `cs` — равным значению старшего слова.

Примечание

Такой сложный переход генерируется транслятором на основании типа имени `old_v` — `dword`. Если бы `old_v` было определено как массив из двух слов (директивой `dw`), транслятор создал бы косвенный ближний переход (т. е. тоже по данным в памяти, но в пределах текущего сегмента кода). Наконец, определение `old_v` в виде метки создало бы переход непосредственный — по адресу, равному значению имени `old_v`, иными словами, прямо на `old_v`.

При передаче управления по цепи содержимое стека должно быть в точности таким же, как и при входе в `new`, поскольку, в конечном итоге, системная процедура должна (командой `iret`) вернуть управление в прерванную программу.

В варианте `subr` новая процедура обращается к системной, дописывая в стек текущие значения `flags`, `cs`, `ip`; тем самым, имитируется нормальный вход в системную процедуру, как если бы произошло прерывание. После выполнения системной процедурой команды `iret`, управление возвращается в `new` — на инструкцию, следующую за `call`.

Запись в стек значений `cs` и `ip` выполняется автоматически, в результате дальнего вызова подпрограммы, а значение `flags` предварительно записано командой `pushf`. В результате, в стеке подготовлена информация для возврата по команде `iret`.

Проверьте все варианты: по умолчанию, `chain`, `subr`, а также сочетание `chain` и `subr` (`a86 = chain = subr chain.8`).

17.6. Внешние прерывания

Приемы программирования прерываний, рассмотренные нами на примере исключений, в применении к внешним прерываниям требуют уточнений и дополнений.

17.6.1. Доступ к данным из процедуры обработки прерывания

Главная отличительная особенность внешних прерываний — асинхронность по отношению к выполняемой программе. Внешний запрос может прервать выполнение функции BIOS-DOS в тот момент, когда регистр `ds` настроен на данные операционной системы. С учетом этого обстоятельства, доступ из процедуры обработки прерывания к ее статическим данным требует перенастройки `ds`.

В качестве иллюстрации рассмотрим пример обработки прерывания от нулевого канала трехканального таймера 8253/8254.

Линия запроса на прерывание от устройства подключена к входу `IRQ0` (ведущего контроллера), которому соответствует номер вектора 8. По умолчанию канал 0 работает в режиме циклического счета; при каждом обнулении счетчика генерируется запрос на прерывание.

Примечание

В DOS регистр установки нулевого канала таймера содержит ноль, так что между прерываниями счет выполняется 010000 раз — начиная с 0, через 0ffff, 0ffe и т. д. до 0. Таким образом, при частоте счета ~1193 кГц частота запросов на прерывание составляет ~18,2 Гц (период — около 55 мсек).

Листинг 17.8. Обработка прерывания от таймера (см. timer_1.8)

```

set_seg macro
    mov     ax, #1

#qxbll
    mov     #x, ax
#em

rd_vect macro
    mov     ax, 035 by #1
    int     021                ; get vect
    mov     w old_#1, bx
    mov     w old_#1 + 2, es    ; save it
#em

wr_vect macro
    mov     ax, 025 by #1
    int     021
#em

    jmp     start

old_8     dd     ?
char      db     ?

new8:
    push    ax, ds, es        ; (1)

    set_seg es, 0b800
    set_seg ds, cs            ; (2)
    mov     al, char          ; (3)
    es mov  [0], al
    inc     char              ; (4)
#if alone
    mov     al, 020            ; (5)
    out     020, al
#endif
    pop     es, ds, ax        ; (6)
#if alone
    iret
#else
    cs jmp  old_8              ; (7)
#endif

start:
    rd_vect 8

```

```
lea      dx, new8
wr_vect 8

mov      ah, 0
int      016          ; (8)

lds      dx, old_8     ; (9)
wr_vect 8
set_seg  ds, cs

int      020
```

Процедура обработки прерывания выводит в угол экрана код из байта `char` и подготавливает значение `char` для следующего вызова. Главная программа ожидает (8) нажатия клавиши за счет вызова функции BIOS (`int 016`); в результате, процедура обработки прерывания от таймера периодически прерывает выполнение исключения 016.

Примечание

Исключение 016, вызываемое из прикладной программы, анализирует результаты внешнего (аппаратного) прерывания от клавиатуры, накапливаемые в специальном буфере ввода. Если задан ввод символа с ожиданием и при этом буфер ввода пуст, системная процедура обработки исключения 016 вынуждена разрешить внешние прерывания. Поэтому выполнение `int 016` прерывается процедурой `new8`.

В момент вызова `new8` регистр `ds`, скорее всего, настроен на область данных BIOS (сегмент 040). (Не так уж важно, как именно он настроен — главное, что не на программный сегмент процедуры `new8`.) Если выполнить команду (4) при `ds = 040`, изменится значение байта по адресу 040:0107 (значение имени `char`, из файла-листинга, составляет 0107). В результате, что-то в области BIOS будет разрушено. При этом байт `char` самой процедуры `new8` останется без изменений.

В общем случае, при входе в процедуру обработки внешнего прерывания правильно настроен только регистр `cs` — в результате передачи управления по таблице векторов. Поэтому после сохранения (1) всех изменяемых регистров следует сразу же настроить (2) доступ к данным процедуры: `ds := cs`. Тогда при выполнении (3, 4) доступ по смещению `char` направлен в сегмент данных самой процедуры.

В монопольном варианте (`alone`) в конце процедуры `new8` выполняется сброс заявки в контроллере прерываний и, после восстановления (6) регистров, управление возвращается в прерванную программу командой `iret`.

В альтернативном варианте сброс контроллера не программируется — его выполняет системная процедура. Чтобы передать ей управление, следует

выполнить косвенный переход по значению двойного слова `old_8`. Здесь возникает противоречие:

- ❑ команда `jmp` для передачи управления — последняя;
- ❑ до выполнения `jmp` регистр `ds` должен быть восстановлен (6);
- ❑ изменение `ds` нарушает доступ к `old_8` при отработке `jmp`.

Противоречие разрешается за счет переназначения сегмента данных при выполнении (7): вместо `ds` используется `cs`.

Примечание

Если бы не команда (2), то возможное прерывание в промежутке между (9) и `int 021` (в составе `wr_vect`) привело бы к искажению байта по смещению 0107 в кодовом сегменте системной процедуры обработки прерывания 8 (в этот момент `ds` содержит сегментный адрес системной процедуры).

Команда (2) в процедурах обработки внешних прерываний обязательна (или придется задавать доступ к данным через `cs` в каждой команде, где есть обращение к данным). Заметим, что в резидентных процедурах, которые подробно будут рассмотрены в *главе 18*, установка `ds := cs` необходима при обработке прерываний всех типов, не только внешних. Также, при обработке внешних прерываний и в резидентных процедурах обязательно переназначение сегмента данных в команде (7), при передаче управления передается по цепи.

Примечание

Рассмотренный пример работы с внешним прерыванием — один из наиболее простых на эту тему. Здесь, во-первых, не понадобилось открывать линию `IRQ` в регистре `iMr` контроллера прерываний, т. к. она уже открыта — для подсчета системного времени DOS и выдержки паузы на выключение приводов дисководов. Во-вторых, причина прерывания снимается самим устройством, без участия программы.

17.6.2. Ограничения на использование функций операционной системы

Как выяснилось, внешнее прерывание может произойти при выполнении функции операционной системы. Если при этом процедура обработки внешнего прерывания сама обратится к DOS, то DOS будет разрушена, поскольку при выполнении второго вызова промежуточные результаты первого (прерванного) вызова будут утеряны. Система DOS — однозадачная, и ее функции не допускают повторного вхождения.

По этой причине во всех примерах внешних прерываний вывод на экран выполняется записью непосредственно в видеопамять, без использования функций DOS.

Примечание

Выполнение исключения также может оказаться асинхронным, если это исключение вызывается из процедуры обработки *внешнего* прерывания. Характерный пример — программное прерывание `int 5`, которое вызывается из системной процедуры обработки прерывания от клавиатуры по комбинации клавиш `<Shift>+<PrtScrn>`. Еще один пример — программное прерывание `int 01c`, которое вызывается из системной процедуры обработки прерывания от таймера после того, как сброшена заявка в контроллере прерываний. В процедурах обработки исключений 5 и 01c вызов функций DOS недопустим, т. к. эти процедуры являются *продолжением* процедур обработки внешних прерываний.

17.6.3. Определение причины прерывания

Применительно к внешним прерываниям вопрос об определении причины возникает в тех случаях, когда устройству разрешено генерировать запрос по нескольким событиям.

Примечание

Программный вызов процедуры, рассчитанной на обслуживание внешнего прерывания, категорически запрещен — процедура выполнит обработку события, которого на самом деле не произошло. Так можно потерять часть данных для вывода (очередной байт для передачи записывается в *занятый* регистр данных передатчика) или "принять" избыток данных (очередной байт принимаемого блока данных считывается из буферного регистра приемника, когда прием не закончен).

В качестве примера рассмотрим обработку в режиме прерываний двух типов событий, происходящих в часах реального времени RTC (Real Time Clock). Доступ к регистрам RTC в составе CMOS-памяти рассмотрен в *разд. 16.3.3*.

События в примере следующие — ежесекундное обновление показаний часов и импульс внутреннего генератора RTC с частотой 1024 Гц по умолчанию. В комментариях к программе из листинга 17.9 соответствующие события обозначаются буквами U (Update-ended) и P (Periodic event). Наступление событий U и P отражается в битах 4 и 6 регистра C (т. е. регистра по адресу 0с). Прерывания по событиям U и P разрешаются установкой в единицу тех же разрядов в регистре B.

Чтобы определить причину прерывания от RTC, следует прочитать и проанализировать значение регистра состояния C. При чтении C биты 4 и 6 этого регистра автоматически сбрасываются; тем самым, снимается причина прерывания, что является необходимым для генерирования следующего запроса.

Листинг 17.9. Определение причины прерывания от RTC (см. rtc.8)

```
set_seg macro
    mov     ax, #1
```

```

#qxbll
        mov     #x, ax
#em

set_adr macro
        mov     al, #1
        out     070, al
#em

rd_rtc  macro
        set_adr #1
        in      al, 071
#em

wr_rtc  macro
##if #nl eq 2
        mov     al, #2
##endif
        xchg    al, ah
        set_adr #1
        xchg    al, ah
        out     071, al
#em

WaitEsc macro
m1:     mov     ah, 1
        int     021
        cmp     al, 01b          ; esc
        jne     m1
#em

        jmp     start

u_char  db      ?                ; Update-ended counter
p_char  db      ?                ; Periodic event counter

iMr2    db      ?                ; slave 8259 initial iMr
b_reg   db      ?

new70:                                     ; IRQ8
        push    ax, ds, es

        set_seg ds, cs
        set_seg es, 0b800

        rd_rtc  0c                ; al <- RTC state

        test    al, bit 4          ; Update-ended?
        jz      >11

```

```

        inc     u_char
        mov     ah, u_char
        es mov  [0], ah

11:
        test    al, bit 6      ; Periodic event?
        jz      >11
        inc     p_char
        mov     ah, p_char
        es mov  [2], ah

11:
        mov     al, 020        ; send EOI to:
        out     0a0, al        ; - slave 8259
        out     020, al        ; - master 8259

        pop     es, ds, ax
        iret

start:
        mov     ax, 02570      ; set vect
        lea     dx, new70
        int     021

        cli

        rd_rtc  0b             ; read b-reg
        mov     b_reg, al      ; save
        or      al, bit 4 + bit 6 ; enable UI and PI
        wr_rtc  0b

        in      al, 0a1         ; get iMr from slave
        mov     iMr2, al       ; save
        and     al, not bit 0   ; open IRQ0(8)
        out     0a1, al

        sti

WaitEsc

        mov     al, iMr2
        out     0a1, al        ; restore iMr for slave

        wr_rtc  0b, b_reg      ; restore b_reg in RTC

        int     020

```

Программа обработки прерывания идентифицирует событие в RTC, вызвавшее прерывание, и увеличивает соответствующий счетчик `u_char` или `p_char`. Счетчики выводятся прямым отображением в видеопамять. В основной программе ожидается нажатие клавиши <Esc>.

Примечание

В одном запросе может быть объединено несколько событий (в данном случае, U и P). В таких случаях обычно говорят об *одновременном* наступлении прерываний (или событий), что категорически неверно. Одно из событий наступает, конечно же, раньше, но система может *задержать* его обработку (например, в этот момент обрабатывается более приоритетное событие), и пока задержанное событие "длится", к нему может добавиться еще одно.

Запрос от RTC подключен к IRQ8, как принято обозначать вход IRQ0 ведомого контроллера прерываний. Базовое значение номера прерывания для ведомого контроллера равно 070. (Напомним, базовое значение для ведущего контроллера равно 8.) Номер прерывания образуется сложением номера входа 0—7 с базовым значением, таким образом номер прерывания для RTC равен 070.

При обслуживании прерывания команда EOI (End Of Interrupt) выдается в оба контроллера.

В приведенном примере предполагается, что вход IRQ2 ведущего контроллера, к которому подключен общий запрос на прерывание от ведомого контроллера, уже размаскирован. Поэтому в программе из листинга 17.9 размаскирован только вход ведомого контроллера.

17.6.4. Практикум по внешним прерываниям

Запрограммируйте обработку прерываний:

- ☐ от клавиатуры — в трех вариантах;
- ☐ от последовательного канала связи — по готовности приемника и по ошибке приема, с определением причины прерывания.

Примечание

Отлаживать внешние прерывания в i86 не рекомендуется — разве что запустить программу целиком командой `g`, для анализа данных на выходе. Остановка в процедуре обработки внешнего прерывания нежелательна в любом месте. Остановка до снятия причины прерывания и сброса заявки в контроллере может привести к блокировке прерываний от клавиатуры; остановка после — к повторному вызову этой же процедуры до того, как закончено выполнение первого вызова.

Прерывание от клавиатуры

При каждом нажатии и отпускании клавиши подается сигнал запроса на вход IRQ1 ведущего контроллера. Таким образом, номер прерывания от клавиатуры равен 9.

Регистры устройства подключены к микросхеме интерфейса с периферией, в частности код клавиши доступен через порт 060. В отличие от регистра

данных приемника последовательного канала, чтение принятых данных из порта 060 не влияет на состояние устройства. Причина прерывания при этом не сбрасывается, а данные при повторном чтении остаются теми же.

Как и в примере с системным таймером, программировать размаскирование входа IRQ1 контроллера не требуется, т. к. прерывания от клавиатуры в IBM PC открыты изначально.

В первом варианте программы процедура обработки прерывания должна выполнять следующие действия:

- ☐ отображать на экране — без использования функций BIOS-DOS — шестнадцатеричное значение байта, прочитанного из порта 060;
- ☐ передавать управление системной процедуре, по цепи.

В главной программе после установки вектора 9 следует задать ожидание нажатия клавиши <Esc> — макрокомандой `WaitEsc`.

Примечание

Вывод на экран без использования функций BIOS-DOS выполняется за счет записи кодов ASCII непосредственно в видеопамять — по четным адресам, например, в байты со смещением 0 и 2. (Байты по нечетным адресам задают цвет символа.)

Данные, прочитанные из порта 060, представляют собой 7-битный позиционный код клавиши; ноль в знаковом бите означает, что прерывание произошло при нажатии клавиши, единица — при отпускании.

Определите значения позиционных кодов для различных групп клавиш. Для проверки, значение кода клавиши <F1> равно 03b. Обратите внимание, позиционные коды букв и цифр отличаются от кодов ASCII (см. приложения 2 и 3).

Во втором варианте процедура обработки прерывания не передает управление системной процедуре. Применение функций BIOS-DOS для ожидания нажатия клавиши <Esc> теперь бесполезно, поскольку эти функции анализируют содержимое *буфера ввода*, который заполняется как раз системной процедурой обработки прерывания 9. При обнаружении нажатия клавиши <Esc> (при получении 1) в процедуре обработки прерывания следует установить признак в ячейке памяти; этот признак проверяется главной программой в цикле.

Поскольку системная процедура обработки отключена, в новую процедуру следует добавить:

- ☐ сброс текущей заявки в контроллере прерываний;
- ☐ снятие причины прерывания — посылкой импульса подтверждения в бите 7 порта 061 (установить и сразу же сбросить этот бит).

В третьем варианте программа завершается при нажатии комбинации клавиш <Ctrl>+<Esc>. Комбинация клавиш <Ctrl>+<Esc> обнаруживается —

при поддержке системной процедуры — следующим образом. Пользовательская процедура обработки прерывания, получив код 1 (код нажатия клавиши <Esc>), анализирует данные о состоянии специальных клавиш, сформированные системной процедурой: единица в разряде 2 байта по абсолютному адресу 0417 (040:017) означает, что нажата клавиша <Ctrl>. В третьем варианте объединяются первые два варианта следующим образом:

- ❑ пользовательская процедура обработки прерывания 9 передает управление системной процедуре, при помощи команды `cs jmp <mem32>`;
- ❑ главная программа закидывается на проверку признака завершения в ячейке памяти; признак устанавливается пользовательской процедурой при выявлении комбинации клавиш <Ctrl>+<Esc>.

Прерывание от последовательного канала связи

На основе примера из листинга 16.2 составьте программу, в которой разрешены прерывания по готовности приемника и по ошибке приема. В регистре разрешения прерываний (регистр по смещению 1) установите, соответственно, биты 0 и 2. В процедуре обработки прерываний определите причину и выполните следующие действия:

- ❑ при готовности приемника — прочитайте регистр данных, увеличьте счетчик приема и выведите его в нулевой байт сегмента видеопамати;
- ❑ при ошибке приема — прочтите регистр состояния и регистр данных; увеличьте счетчик ошибок и выведите его в байт 2 видеопамати.

При использовании мыши в качестве передатчика ошибки будут возникать довольно часто — из-за различия в скоростях передачи (1200 бод) и приема (2400 бод).

Примечание

Относительно медленная передача приводит к тому, что прием стоповой послышки начинается тогда, когда еще не закончена передача информационных бит. В результате, с вероятностью 50% приемник фиксирует "искажение стоповой послышки".

Причина прерывания может быть определена двумя способами:

- ❑ проверкой признаков в регистре состояния (см. табл. 16.1);
- ❑ чтением заявок из регистра идентификации прерываний, по очереди.

Содержимое регистра идентификации прерываний показано в табл. 17.1. Этот регистр в составе последовательного адаптера подключен к порту со смещением 2.

После чтения этого регистра следует сначала проверить бит 0: если он сброшен, то в битах 1 и 2 содержится код наиболее приоритетной заявки. Если бит 0 установлен, значит, заявок больше нет. Рекомендуется поместить

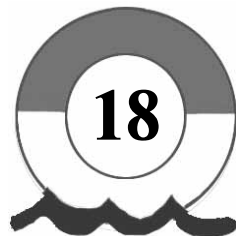
чение регистра идентификации прерываний и обслуживание заявок в тело цикла, и выполнять его до установки бита 0 в регистре идентификации прерываний.

Примечание

В ранней реализации последовательного адаптера — 8250 — допущена ошибка: при наличии заявки от приемника регистр идентификации прерываний не показывает заявку от передатчика.

Таблица 17.1. Результат чтения регистра идентификации прерываний COM-порта

Бит	Значение	Снятие причины
0	1 — нет прерываний, ожидающих обслуживания	
1-2	00 — ошибка приема	Чтение регистра состояния, затем регистра данных
	01 — данные приняты	Чтение регистра данных
	10 — буфер передатчика пуст	Запись в регистр данных
	11 — изменение состояния входных сигналов модема	Чтение регистра состояния модема (порт по смещению 6)



Резидентные программы

Как вдруг вчера, в половине пятого пополудни, в магазине иностранца-собственника является некто необычайной толщины и в нетрезвом виде, платит за вход и тут же, безо всякого предупреждения, лезет в пасть крокодила, который, разумеется, принужден был проглотить его, хотя бы из чувства самосохранения, чтобы не подавиться. Ввалившись во внутренность крокодила, незнакомец тут же засыпает. Ни крики иностранца-собственника, ни вопли его испуганного семейства, ни угрозы обратиться в полицию не оказывают никакого впечатления. Из внутри крокодила слышен лишь хохот и обещание расправиться розгами (sic), а бедное млекопитающее, принужденное проглотить такую массу, тщетно проливает слезы.

Ф. М. Достоевский. "Крокодил. Необыкновенное событие, или Пассаж в Пассаж"

Резидентная программа — это программа, которая завершает выполнение, не освобождая занятую память полностью. Обычно в памяти оставляют фрагмент программы, содержащий процедуры обработки прерываний; в транзитной (нерезидентной) части программы выполняется настройка векторов на процедуры обработки из резидентной части программы.

Примечание

В относительно редких случаях в памяти оставляют не процедуры, а данные — системную таблицу, адрес которой записывается в таблицу векторов. Вектор в данном случае используется не по назначению — это уже указатель на данные, а не на программу обработки прерывания. Смысл — в том, что любая пользовательская программа имеет возможность получить доступ к резидентной таблице.

После установки векторов выполнение программы как самостоятельного процесса заканчивается — но таким образом, что код и данные остаются в памяти. Отсюда — принятое в англоязычной технической литературе название *Terminate but Stay Resident*, или, сокращенно, *TSR*. Оставленная в памяти процедура обработки прерывания в момент ее вызова считается частью

текущего, т. е. прерванного процесса. В результате, обращение к функции DOS 04с из резидентной процедуры приводит к завершению *прерванной* программы, что применяется при обработке fault-исключений.

18.1. Установка резидентной процедуры

Завершение программы с сохранением ее фрагмента в памяти выполняется вызовом `int 027`, вместо обычного `int 020`. В регистре `dx` при вызове `int 027` должен быть задан адрес нижней границы области освобождаемой памяти. Фрагмент от смещения 0 до смещения, указанного в `dx`, сохраняется.

В качестве примера рассмотрим программу `into.8`, при помощи которой мы заменяли системную процедуру обработки исключения 4 в *разд. 15.5.2*. Программа `into.8` приведена в листинге 18.1.

Листинг 18.1. TSR-программа для обработки исключения 4 (см. `into.8`)

```

        jmp      start          ; (1)

msg     db      13, 10, '?-Trap (into)$'

new4:
        push    ax, dx, ds      ; (2)
        mov     ax, cs          ; (3)
        mov     ds, ax          ; (4)
        lea     dx, msg
        mov     ah, 9
        int     021

#if stop
        mov     ax, 04c01        ; exit to DOS
        int     021
#else
        pop     ds, dx, ax
        iret
#endif

start:
                                ; (5)
        mov     ax, 02504        ; write vector 4
        lea     dx, new4
        int     021
        mov     dx, start        ; (6)
        int     027              ; (7)

```

Программа записывает в четвертый вектор адрес процедуры `new4` и завершается (7) с сохранением в памяти своего программного сегмента — от начала

до адреса, заданного (6) в `dx` (в `dx` — адрес `start`). В памяти остается PSP (в адресах 0—0ff) и примыкающий к PSP фрагмент от (1) до (5). Код инициализации — от (5) до (7) — расположен после `new4`. При таком расположении кода инициализации, его можно удалить после выполнения, при этом сохранив процедуру `new4`.

В результате выполнения программы `into.com` в таблице векторов прерываний в четвертой записи установлен адрес процедуры `new4`, а сама процедура с небольшим количеством балласта осталась в памяти. Балласт — это PSP и команда (1).

Обратите внимание на настройку регистра `ds` (3, 4) в процедуре `new4`. При вызове прерывания регистр `ds`, скорее всего, настроен на данные текущей программы; во всяком случае, не на сегмент резидентной программы. В этой ситуации, напоминая, возможны два варианта настройки доступа к данным:

- ❑ индивидуально, для каждой команды, использующей данные в памяти, за счет префикса `cs` переназначения сегмента данных;
- ❑ для всей процедуры, за счет присвоения `ds = cs` на входе, с восстановлением `ds` на выходе.

Второй вариант предпочтительней в процедурах нетривиальной сложности; в данном случае он оказывается единственным возможным только потому, что при вызове функции 9 DOS адрес строки для вывода следует задать парой `ds:dx`.

Примечание

Команды `push` (2) в процедуре `new4` пользуются стеком *текущей* задачи. Полагая, что область памяти под стек прерванной задачи зарезервирована с запасом, мы позволяем TSR-программе "паразитировать" на стеке прерванной задачи. Если резидентной процедуре требуется значительный объем стека, следует переключать стек (т. е. устанавливать указатель `ss:sp`) на область памяти, зарезервированную в TSR-программе. Так, например, сделано в системной процедуре обработки прерывания 021.

Процедура `new4` выполнена в двух вариантах. (Сообщение об ошибке выводится в любом варианте.) В варианте по умолчанию управление возвращается в текущую программу. В варианте `stop` выполнение текущей программы завершается. (Так, например, устроена системная процедура обработки fault-исключения 0 — по ошибке деления);

Процедура `new4` обрабатывает прерывание монопольно, без передачи управления системной процедуре-заглушке. Передачу управления по цепи проиллюстрируем на примере обработки прерывания от таймера.

Преобразуем программу `timer_1.8` из листинга 17.8 в резидентную. Вариант `chain` в данном случае единственно возможный, поскольку системная про-

цедура обработки прерывания 8 выполняет важные для DOS функции: счет системного времени и контроль тайм-аутов при операциях с дисками.

**Листинг 18.2. TSR-программа обработки прерываний от таймера
(см. timer_2.8)**

```
set_seg macro
    mov     ax, #1
#qxb11
    mov     #x, ax
#em

rd_vect macro
    mov     ax, 035 by #1
    int     021                ; get vect
    mov     w old_#1, bx
    mov     w old_#1 + 2, es    ; save it
#em

wr_vect macro
    mov     ax, 025 by #1
    int     021
#em

    jmp     start

old_8    dd     ?
count    dw     ?

new8:
    push    ax, ds, es
    set_seg es, 0b800          ; video seg
    set_seg ds, cs
    mov     al, b count
    es mov  [0], al            ; show low byte
    inc     count
    pop     es, ds, ax
    cs jmp  old_8              ; chain int

start:
    rd_vect 8
    lea     dx, new8
    wr_vect 8

    lea     dx, start
    int     027
```

18.2. Взаимодействие с TSR-программой по данным

В рассмотренных примерах нет обмена информацией между TSR-процедурой и пользовательской программой.

Пример взаимодействия по данным — с указанием входных параметров и возвращением результатов — представляют вызовы TSR-функций DOS, через программное прерывание 021. Обмен данными в пределах байт и слов выполняется через регистры. В частности, `ah` на входе задает номер функции, а при передаче массива его адрес указывается в паре регистров `ds:dx`.

Программный интерфейс с базовой системой ввода/вывода (BIOS) демонстрирует возможность взаимодействия по данным с TSR-процедурой обработки *внешних* прерываний. BIOS-процедура для обслуживания прерываний от клавиатуры записывает полученные позиционные коды в 32-байтный кольцевой буфер с адреса `0:041e`. Прикладная программа запрашивает данные из этого буфера посредством вызова `int 016`. Процедуры обслуживания прерывания 9 и исключения 16 находятся в одной TSR-программе; первая накапливает асинхронно поступающие данные, а вторая синхронно считывает их — по запросам пользователя.

В качестве примера организации взаимодействия по данным дополним программу `timer_2.8` из листинга 18.2 так, чтобы прикладная программа могла получить значение счетчика прерываний.

Листинг 18.3. Интерфейс TSR-программы (см. `timer_3.8`)

```
...

new60:
#if by_ref
    mov     es, cs           ; (1)
    lea     bx, count       ; (2)
#else
    cs mov  ax, count       ; (3)
#endif
    iret
...

start:
...
    lea     dx, new60
    wr_vect 060
    ...
```

TSR-программа из листинга 18.3 включает в себя две процедуры обработки прерываний: `new8` — для обслуживания таймера, `new60` — для приема запро-

сов, посылаемых посредством программного прерывания `int 060`. Вектор `060` выбран потому, что в DOS он не используется.

В варианте `by_ref` счетчик передается *по ссылке*: отправителю возвращается (1, 2) *адрес* `count` в паре `es:bx`. Программа, получив адрес счетчика, имеет возможность не только читать, но также изменять текущее значение `count`. Например, счетчик может быть обнулен следующим образом:

```
int      060
es mov   w [bx], 0      ; timer_3.count
```

В варианте по умолчанию, счетчик передается *значением*: программа в результате вызова `int 060` получает в регистре `ax` *копию* текущего значения `count`.

Недостаток интерфейса в рассмотренном примере связан с применением вектора `060`. Хотя этот вектор "свободен", дисциплина его использования независимыми разработчиками программного обеспечения не определена. Для взаимодействия по данным с TSR-программами специально предусмотрено исключение `02f`, или "мультиплексное" прерывание (`mux`).

При вызове `int 02f` в регистре `ah` указывается идентификатор, или код TSR-программы. Часть кодов назначена системным TSR-процедурам (`print`, `assign`, `share` и т. д.), а коды в диапазоне `080—0ff` зарезервированы для вновь создаваемых TSR-программ. Процедура обработки исключения `02f` в составе TSR-программы должна, при совпадении значения `ah` с присвоенным ей кодом, взять на себя обслуживание вызова; иначе, передать управление по цепи. В конце цепи находится системная процедура-заглушка.

Номер функции при вызове `int 02f` задается в `al`. Назначение номеров функций произвольное, кроме номера ноль. Функция `0` должна возвращать в `al` информацию о наличии TSR-процедуры с кодом, заданным в `ah`. Если возвращаемое значение `0`, значит, TSR-программа не установлена, а если `-1`, то установлена.

Продemonстрируем организацию программного интерфейса через прерывание `02f`, переделав пример из листинга 18.3. Возьмем в качестве идентификатора TSR-программы значение `0ff`. Пусть функция `1` задает обнуление счетчика, а функция `2` — считывание его значения в регистр `ax`. Функция `0` — стандартная.

Листинг 18.4. Интерфейс через мультиплексное прерывание (см. `mux_1.b8`)

```
...
tsr_id equ      0ff
old_02f dd      ?
...
```

```

new_02f:
    cmp     ah, tsr_id
    IF ne
        cs jmp  old_02f          ; chain to other TSRs
    END
    push    bx
    cbw     ; ah := 0
    mov     bx, ax
    CASE bx
    | 0
        dec     al          ; al := -1
    | 1
        cs mov  count, 0
    | 2
        cs mov  ax, count
    END
    pop     bx
    iret

...
mux
#rx11
    mov     ax, tsr_id by #x
    int     02f

#em

start:
    mux     0          ; func = 0
    test    al
    IF nz
        print  reject
        int    020
    END
    ...
    rd_vect 02f
    lea     dx, new_02f
    wr_vect 02f
    ...

reject db      7, '?-Already installed', 13, 10, '$'

```

Идентификатор TSR-программы задан константой `tsr_id`. Обработка исключения 02f начинается с проверки идентификатора в `ah`. Если в запросе задан код, не равный `tsr_id`, значит, запрос адресован другой TSR-программе, и управление передается по цепи. Если `ah = tsr_id`, выполняется мультитветвление по значению `al`. Для этого содержимое `ax` — после очистки

его старшего байта — копируется в регистр-селектор `bx`, по значению которого выполняется мультиветвление (см. "Оператор мультиветвления" в разд. 12.4.1).

При `al = 0` (функция 0), в `al` формируется ответ `-1`, что означает присутствие TSR-программы с заданным кодом. При `al = 1` в счетчик `count` записывается ноль, а при `al = 2` в регистр `ax` считывается текущее значение `count`.

Макрокоманда `mux` упрощает программирование запросов. При каждом обращении к `mux` в регистр `ah` записывается код `tsr_id`, а в `al` — номер функции, заданный в качестве параметра.

Выполните трансляцию `mux_1.b8` (перед ассемблированием файл `bsp86.sk1` следует поместить в текущий каталог — вместе с исходным файлом):

```
bsp86.exe mux_1.b8 mux_1.8
```

```
a86 mux_1.8
```

Не запуская `mux_1.com`, вызовите с ним отладчик и выполните первую пару команд, реализующих вызов `mux 0`. Значение `ax` не изменилось; это означает, что ни одна из TSR-программ в цепи обработки прерывания `02f` не признала запрос с кодом `0ff`. В итоге, управление было передано на системную процедуру-заглушку.

Выйдите из отладчика и запустите `mux_1.com`. Повторите в отладчике предыдущий опыт. Значение в `ax` должно измениться, `al = 0ff (-1)`; т. е. запрос воспринят процедурой `new_02f`, которая в результате вызова `mux_1.com` поставлена в цепь обработки прерывания `02f`.

Если вызов `mux 0`, заданный в начале инициализирующей части программы `mux_1`, обнаруживает, что процесс с кодом `tsr_id` отвечает, то загрузка отменяется. Для проверки, выйдите из отладчика и выполните `mux_1.com` еще раз.

Для проверки функций 1 (обнуление счетчика прерываний от таймера) и 2 (чтение счетчика) войдите в отладчик с `mux_1.com` и выполните в непосредственном режиме макрокоманду `mux 2`.

Повторяя выполнение `mux 2` нажатиями клавиши `<F3>`, наблюдайте за изменением регистра `ax`. Проверьте обнуление счетчика, выполнив `mux 1, 2`.

Вызовы `mux` могут быть заданы в любой пользовательской программе, в частности, в инициализирующей части TSR-программы, что мы продемонстрировали на примере предотвращения повторной загрузки — по результату запроса `mux 0`.

Интерфейс TSR-программы через мультиплексное прерывание `02f` рассчитан на подключение до 128 TSR-программ. Стандартная функция 0 для проверки наличия TSR-программы позволяет избежать как повторной загрузки, так и выдачи команд и запросов не по адресу.

18.3. Уменьшение размера занимаемой памяти

В рассмотренных примерах TSR-программ код и данные, связанные с инициализацией, расположены внизу программы. Это позволяет при вызове `int 027` почти полностью удалить отработавшую часть программы, оставив лишь процедуры обработки прерываний и их статические данные. Так, в программе из листинга 18.4 будут удалены не только инструкции от адреса `start`, но и сообщение о повторной загрузке `reject`, которое не имеет никакого отношения к выполнению резидентных процедур `new8` и `new_02f`.

Объем памяти, занимаемой TSR-программой, также может быть сокращен за счет удаления копии переменных окружения (`PROMPT`, `COMSPEC`, `PATH` и т. п.). При запуске программы DOS создает копию этих системных имен, сохраняя сегментный адрес копии в `PSP` — по смещению `02c`.

Копия окружения находится в отдельном блоке памяти, не связанном с программным сегментом. Поэтому ее можно ликвидировать с использованием функции `049 DOS`, не удалив при этом всей программы. Копия окружения удаляется при выполнении инициализирующей части TSR-программы следующим вызовом (в `es` указывается сегментный адрес освобождаемого блока памяти):

```
mov     es, [02c]
dos     049
```

Загрузите TSR-программу `into.com` и выясните размер занимаемой памяти при помощи DOS-утилиты `mem` с ключом `/c`. Добавьте в `into.8` удаление копии окружения, выполните трансляцию и загрузите новый вариант `into.com`. Затем вновь выполните команду `mem /c`. Размер `into.com`, хотя и стал меньше, все же превышает длину `com`-файла — за счет памяти, занимаемой `PSP`.

В отличие от копии окружения, которая находится в отдельном блоке памяти, `PSP` является частью единого блока, выделенного программе при загрузке. По этой причине освобождение `PSP` — уже не столь простая операция. А поскольку размер `PSP` не такой уж большой, лучше эту память оставить.

Примечание

`PSP` может быть использован в качестве массива неинициализированных данных; если же таких данных требуется более 256 байт, может пригодиться копия среды DOS или код/данные инициализации.

18.4. Выгрузка TSR-программы

Само по себе удаление программного сегмента несложно: достаточно выполнить вызов `dos 049` с предварительной записью в `es` значения сегмента TSR-программы из регистра `cs`.

Предварительно, TSR-процедура должна быть отключена от обработки прерываний. Для этого, по-видимому, следует восстановить в таблице векторов значение, сохраненное при установке TSR-программы. Проблема заключается в том, что в результате такого восстановления будут отключены также процедуры в составе TSR-программ, загруженных после удаляемой. Удаление возможно лишь тогда, когда процедура обработки — первая в списке, т. е. вектор из таблицы указывает на эту процедуру.

Примечание

Речь идет о всех прерываниях, которые обслуживала удаляемая TSR-программа. Удаление возможно, если все соответствующие процедуры обработки находятся в начале цепей обработки прерываний.

Выясним причину этого ограничения. На рис. 18.1 показано включение процедуры **В** в обработку прерывания $\langle n \rangle$. В начале (рис. 18.1, *а*), вектор $\langle n \rangle$ указывает на системную процедуру **А**. При загрузке новой TSR-программы вектор $\langle n \rangle$ считывается и запоминается в данных этой программы. Так образуется указатель из **В** в **А**. (В рассмотренных примерах указатель **В**—**А** находится в двойном слове по имени `old_<n>`.) Затем адрес **В** записывается в таблицу векторов; тем самым, указатель начала списка устанавливается на **В** (рис. 18.1, *б*).

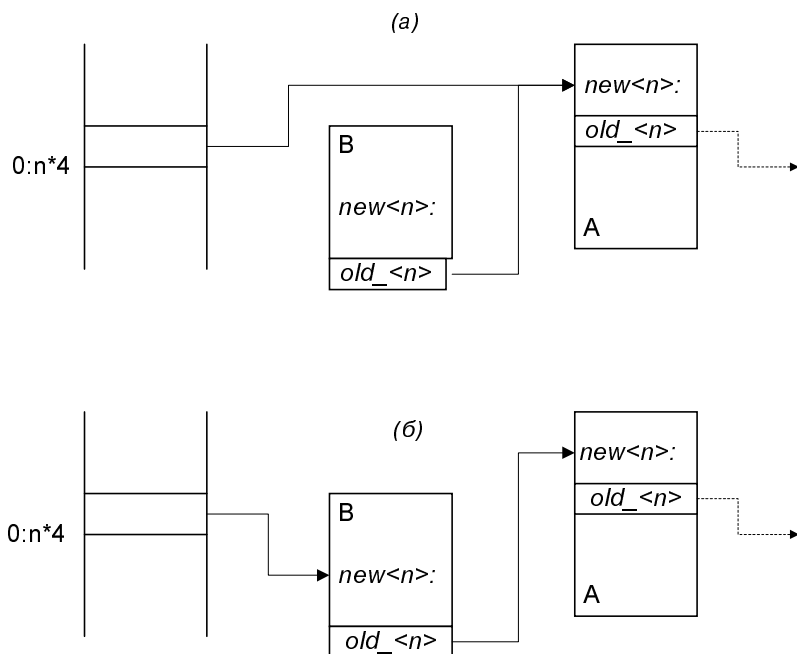


Рис. 18.1. Установка TSR-процедуры обработки прерывания $\langle n \rangle$

В ситуации, показанной на рис. 18.1, *а*, запись значения `old_<n>` в вектор `<n>` приведет к исключению процедуры **В** из списка; при этом исходная цепь будет восстановлена (рис. 18.1, *а*).

Предположим, после загрузки TSR-программы с процедурой **В** таким же образом установлена еще одна процедура обработки прерывания `<n>`, в составе другой TSR-программы (рис. 18.2, *а*). Процедура **В** уже не в начале списка. Если в этой ситуации записать в таблицу векторов указатель $\mathbf{B} \rightarrow \mathbf{A}$, то из списка будут исключены не только **В**, но и **С**, что иллюстрирует рис. 18.2, *б*.

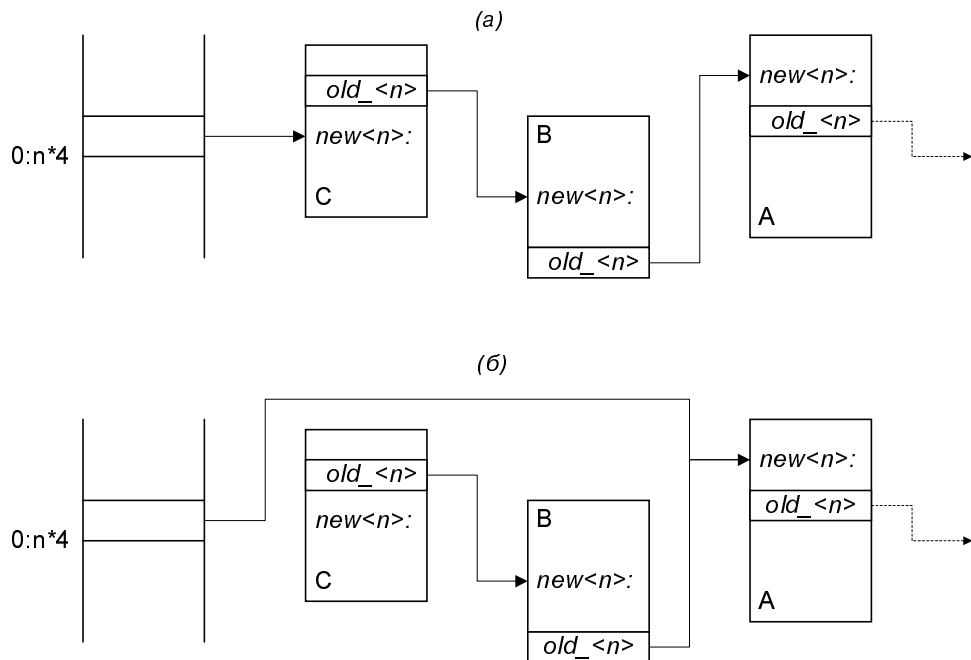


Рис. 18.2. Удаление TSR-процедуры (**В**) из середины списка

Примечание

Для исключения **В** из списка необходимо изменить указатель **С** так, чтобы он показывал на **А**. Это невозможно потому, что местоположение указателя в составе **С** неизвестно **В** (где находится `old_<n>` в нашей TSR-программе, мы знаем; но где расположен аналогичный указатель другой TSR-программы, знает лишь создатель). Другое дело — указатель из таблицы векторов, его адрес всем известен: `0:<n>*4`.

Итак, запись запомненного значения вектора возможна и имеет смысл, только если текущий вектор указывает на удаляемую процедуру; т. е. когда процедура находится в начале списка.

Для проверки достаточно сопоставить сегментные составляющие указателей. С учетом того, что сегментный адрес процедуры представлен значением регистра `cs` при выполнении TSR-программы, проверка для вектора `<n>` реализуется, например, следующими командами:

```
mov     ax, 035 by n    ; read vector n
int     021             ; es:bx
mov     ax, es
mov     bx, cs
cmp     ax, bx          ; es <> cs
jne     not_first
...
```

Такая проверка должна быть выполнена для всех прерываний, обслуживаемых TSR-программой. В следующем примере, представляющем развитие программы `muh_1.b8` из листинга 18.4, удаление TSR-программы выполняется по команде, посылаемой через мультиплексное прерывание 02f, функция -1.

**Листинг 18.5. Функция выгрузки в составе TSR-программы
(см. `muh_2.b8`)**

```
...

new_02f:
...

CASE bx
| 0
    dec     al
| 1
    cs mov  count, 0
| 2
    cs mov  ax, count
ELSE
    call    remove
END
...

remove:
push     ax, ds
set_seg ds, 0                ; vector table segment
mov      ax, cs              ; current progSeg

cmp      w [8 * 4 + 2], ax    ; check vect 8
jne      >11
cmp      w [02f * 4 + 2], ax  ; check vect 02f
jne      >11
```

```

cs mov  ax, w old_8           ; restore vect 8
mov     w [8 * 4], ax        ; (ofs)
cs mov  ax, w old_8 + 2
mov     w [8 * 4 + 2], ax    ; (seg)

cs mov  ax, w old_02f        ; restore vect 02f
mov     w [02f * 4], ax      ; (ofs)
cs mov  ax, w old_02f + 2
mov     w [02f * 4 + 2], ax  ; (seg)

push    es
set_seg es, cs              ; TSR progSeg -
dos     049                 ; - deallocate
pop     es                  ; (1)

11:
pop     ds, ax
ret

...
```

Подпрограмма `remove` вызывается из процедуры `new_02f` при значении `al` вне диапазона 0—2. В подпрограмме `remove` сегментные составляющие векторов 8 и 02f проверяются на равенство значению `cs`. Если проверка успешна, значения из `old_8` и `old_02f` копируются в вектора 8 и 02f, освобождается память, занятая TSR-программой.

Примечание

Программа освобождает память, в которой ей предстоит выполнить последние команды — от (1) до выхода из исключения 02f (`iret`). Этот участок программы — критический: при его выполнении распределение памяти по запросу из другой программы недопустимо. В рассмотренном примере критический участок защищен от внешних прерываний, т. к. флаг `i` при входе в `new_02f` был сброшен автоматически, и в дальнейшем не устанавливался (команды `sti` не было).

Так как `new_02f` не возвращает информации о выполнении функции -1, программа, пославшая запрос `muh -1`, должна проверить выгрузку TSR-программы — запросом `muh 0`.

Загрузите `muh_2.com`. Убедитесь, что `muh_2.com` присутствует в перечне TSR-программ, который выводится командой `mem /c`; также запомните размер резидентной части `muh_2`, для следующего опыта. В отладчике проверьте функции 0 и 2, пошлите запрос на выгрузку (`muh - 1`) и проконтролируйте его выполнение — командой `muh 0`. Выйдя из отладчика, убедитесь (командой `mem /c`) в том, что программа `muh_2.com` выгружена.

В листинге 18.6 приведен более компактный вариант функции `remove` — в форме макрокоманды с параметрами; в списке параметров при вызове `remove` указаны номера векторов.

Листинг 18.6. Реализация функции выгрузки в виде макрокоманды (см. mux_3.b8)

```

...

remove macro
    push    ax, ds
    set_seg ds, 0                ; vector table segment
    mov     ax, cs                ; current progSeg
#rx11
    cmp     w [#x * 4 + 2], ax
    jne     >m1
#er
#rx11
    cs mov  ax, w old_#x
    mov     w [#x * 4], ax        ; (ofs)
    cs mov  ax, w old_#x + 2
    mov     w [#x * 4 + 2], ax    ; (seg)
#er
    push    es
    set_seg es, cs
    dos     049                  ; deallocate
    pop     es
m1:
    pop     ds, ax
#em

new_02f:
    ...
    CASE bx
    ...
    ELSE
        remove 8, 02f          ; (1)
    END
    ...

start:
    ...
    mov     es, [02c]
    dos     049                ; deallocate environment
    ...

```

Также, в программу из листинга 18.6 добавлено освобождение памяти, занятой копией переменных окружения. Загрузите mux_3.com, сравните размер его TSR-части с значением из предыдущего опыта.

Проверим защиту от выгрузки. Установим TSR-программу, которая использует один из векторов `mux_3`, например, вектор `02f`. Загрузите одну из DOS-утилит `share`, `print` или `nlsfunc`.

Интерфейс с TSR-программой `share/print/nlsfunc` организован через исключение `02f`. При загрузке одной из этих утилит процедура обработки исключения `02f` в составе утилиты поставлена в начало списка (см. процедуру **С** на рис. 18.2), а процедура `new_02f` в составе `mux_3` оказалась второй (см. процедуру **В** на рис. 18.2). Удаление процедуры `mux_3` в этой ситуации привело бы, как минимум, к блокировке командного интерфейса утилиты, организованного через прерывание `02f`.

Выполните из отладчика запрос на удаление `mux_3` (`mux - 1`); убедитесь в том, что запрос не выполнен.

Как правило, запрос `mux 0` для проверки повторной загрузки помещают в инициализирующую часть TSR-программы — до записи векторов. Команду выгрузки также имеет смысл поместить в эту же программу, и вызывать по какой-нибудь опции командной строки. Например, выгрузку установленного экземпляра `mux_3.com` можно было бы задать следующим образом: `mux_3.com /r`.

Примечание

Выгрузка pop-up программ, которые мы рассмотрим далее, обычно иницируется иначе: по нажатии комбинации клавиш после того, как pop-up программа примет управление.

В следующем примере, в листинге 18.7, транзитная часть TSR-программы посылает запрос на выгрузку, предварительно убедившись в наличии ранее установленной TSR-части. Выгрузка задается в командной строке — опцией `/r`.

Листинг 18.7. Инициализирующая часть с командой выгрузки (см. `mux_4.b8`)

```

...
start:
    mux      0                ; al: 0 - not exist, -1 - exist
    cbw
    neg      ax               ; ax: 0 - ^exist, 1 - exist
    mov      bx, ax
    shl      bx, 1           ; bx: 0 - ^exist, 2 - exist

    mov      di, 081         ; ptr to cmd_tail
    mov      al, '/'
    cld
    mov      cx, 10

```

```

repne    scasb          ; seek '/'
IF e
    cmp    b[di], 'r'
    IF e
        inc    bx          ; bit 0 of bx — remove
    END
END

CASE bx
| bit 0
    ;                                {^exist, remove}

    print   7, '?-Nothing to remove'
    int     020

| bit 1 + bit 0
    ;                                {exist, remove} — deinstall

    mux     -1, 0          ; remove, than check if TSR exist
    test    al
    IF z
        print 'TSR removed'
    ELSE
        print 7, "?-Can't be removed"
    END
    int     020

| bit 1
    ;                                {exist, install}

    print   7, '?-Already installed'
    int     020
END

;                                {^exist, install}
...

lea    dx, start
int     027

```

Рассмотрим, что представляют собой варианты мультиветвления в этом примере. В начале, по результату `mux 0`, в `bx` формируется число, разряд 1 которого отмечает наличие в памяти экземпляра TSR-программы; все остальные биты `bx` сброшены. Затем, при обнаружении опции `/r` в регистре `bx` устанавливается бит 0. В результате, в младших двух битах регистра `bx` отражены: тип команды (удаление/установка) и наличие установленного экземпляра TSR-программы. В вариантах мультиветвления заданы три сочета-

ния из четырех возможных — такие, которые приводят к обычному завершению программы с выгрузкой из памяти.

Примечание

Командная строка сохраняется в PSP: с адреса 081 до конца PSP. Строка заканчивается байтом 0d, за которым следуют null-символы до конца PSP. Байт по адресу 080 содержит значение длины строки.

Загрузите `mux_4.com`. Проверьте защиту от повторной загрузки; затем выгрузите TSR-часть командой `mux_4 /r`. Вновь загрузите `mux_4.com`, установите одну из программ: `share`, `print` или `nlsfunc`, и проверьте выгрузку `mux_4`.

18.5. Вызовы DOS в TSR-процедурах

В рассмотренных примерах вызовы DOS были допущены только в процедурах обработки *синхронных* прерываний (т. е. исключений, закодированных в главной программе). В примерах процедур обработки *внешних* прерываний даже вывод на экран выполняется только прямой записью в видеопамять, без использования DOS.

Примечание

Выгрузка экземпляра установленной TSR-программы включает в себя вызов функции 049 DOS. Таким образом, команда `mux -1` безопасна только в том случае, если она задана в теле текущей задачи. Посылка команды `mux -1` из процедуры обработки внешнего прерывания, например прерывания от клавиатуры, связана с большим риском; велика вероятность, что как раз в этот момент текущая задача ожидает завершения функции DOS.

18.5.1. Способы определения состояния DOS

Предусмотрено два способа определения состояния DOS:

- ❑ опрос бита "Dos busy", или "InDos", адрес которого определяется функцией 034 DOS;
- ❑ обработка исключения 028 "I'm not doing much", которое вызывается DOS при выполнении функций 1—0с (низкоуровневый ввод/вывод).

Ноль в бите InDos означает, что DOS свободна, и разрешен вызов всех ее функций. В момент вызова программного прерывания 028, DOS ожидает ввода с клавиатуры, выполняя функцию из диапазона 1—0с. В этот момент разрешен вызов всех функций, кроме 1—0с.

Таким образом, InDos = 0 разрешает все функции без исключения, а программное прерывание 028 разрешает все функции, кроме 1—0с. Зачем вообще, в таком случае, пользоваться прерыванием 028? Для ответа на этот

вопрос проверим, в каких ситуациях InDos не равен нулю, и как часто вызывается исключение 028.

Листинг 18.8. TSR-программа для наблюдения за состоянием DOS (см. dos_stat.b8)

```

...
p_indos dd      ?
cnt_028 db      0

new_028:
    cs inc     cnt_028          ; count events
    call      view
    cs jmp     old_028         ; chain 028

new_8:
    call      view
    cs jmp     old_8           ; chain 8

view:
    push      ax, bx, ds
    cs lds     bx, p_indos
    mov       bl, [bx]         ; bh := Dos Busy
    add       bl, '0'          ; '0'/'1'
    set_seg    ds, 0b800
    mov       [0], bl          ; show 'Dos Busy'
    cs mov     al, cnt_028
    mov       [2], al          ; show trap 028 counter
    pop       ds, bx, ax
    ret

...

start:
    ...
    dos       034              ; get pointer to InDos
    mov       w p_indos, bx
    mov       w p_indos + 2, es
    ...

```

По исключению 028 и по прерыванию 8 выводится состояние байта InDos, адрес которого был получен при установке TSR-программы; также отображается счетчик исключений 028. (Прерывание от таймера используется только для отображения состояния.)

Примечание

Вызов dos 034 для получения адреса InDos выполняется в инициализирующей (синхронной) части TSR-программы. Было бы ошибкой пытаться определять адрес InDos в самой подпрограмме show, т. к. она вызывается из асинхронно выполняемой процедуры new_8.

Загрузите программу dos_stat.com и наблюдайте за состояниями DOS — в оболочке, например, в Norton Commander, а затем, выйдя из оболочки, в DOS-prompt. Как выясняется, последний вариант как раз и вынуждает применение исключения 028 наряду с проверкой InDos.

18.5.2. Pop-up программы

Применение функций DOS в процедурах обработки асинхронных прерываний рассмотрим на примере класса *всплывающих* (pop-up) TSR-программ, т. е. программ, вызываемых по нажатии клавиши или комбинации клавиш. Утилиты gprot и digit, которыми мы воспользовались в *главе 16*, — типичные примеры pop-up программ.

Примечание

Pop-up программа в узком смысле — это резидентная программа для организации *диалога*. В контексте использования функций DOS мы называем TSR-программу pop-up программой, если она включается от клавиатуры.

В состав pop-up программы, по определению, входит процедура обработки внешнего прерывания от клавиатуры, или, для краткости, kbii (KeyBoard Input Interrupt). Предположим, что при вызове kbii обнаружена комбинация клавиш, по которой должна быть выполнена подпрограмма user_act, включающая в себя вызовы DOS. Проверка возможности вызова функций DOS в самой kbii, вероятно, даст отрицательный результат. Нужно дождаться этой возможности, периодически проверяя условия, например, опрашивая InDos, пока он не станет равным нулю.

Ожидание в kbii недопустимо; поэтому следует, установив заявку, завершить обработку аппаратного прерывания. В итоге, проверка возможности вызова DOS выполняется вовсе не в kbii, а в процедурах обработки прерываний 8 и 028 — при наличии заявки. Прерывание 8 обеспечивает периодичность проверки InDos с частотой 18,2 Гц.

В примере из листинга 18.9 содержательная часть pop-up программы (процедура user_act) вызывается из прерывания 8 при наличии заявки и при условии InDos = 0. Заявка устанавливается в процедуре обработки прерывания 9, при нажатии комбинации клавиш <Ctrl>+<Esc>.

Листинг 18.9. Вызов pop-up процедуры по InDos = 0 (см. pup_1.b8)

```
...
sleep equ 0 ; pop_up states
wake equ 1
active equ 2

state db sleep

;----- timer interrupt
```

```

new_8:
    pushf                                ; call sys ISR
    cs call old_8

    push    ax, ds
    set_seg ds, cs                      ; ds ← ax ← cs

    cmp     state, wake                 ; should wake up?
    IF e
        push    si
        lds     si, p_indos
        test    b[si]                   ; dos free?
        pop     si
        IF z
            mov     ds, ax
            xchg    es, ax
            pusha
            mov     state, active        ; active
            call    user_act
            mov     state, sleep         ; sleep again
            popa
            xchg    es, ax
        END
    END
    pop     ds, ax
    iret

```

;----- kbi interrupt

```

new_9:
    push    ax
    in      al, 060                     ; if scan-code
    dec     al                          ;     = 1 (Esc) ?
    IF z
        push    ds
        set_seg ds, 040
        test    b [017], bit 2         ; "Ctrl" pressed?
        pop     ds
        IF nz
            cs cmp state, sleep         ; now sleeping?
            IF e
                cs mov state, wake      ; wake!
                in      al, 061
                or       al, bit 7       ; ack to keyboard
                out      061, al
            END
        END
    END

```

```

        xor     al, bit 7
        out     061, al
        mov     al, 020             ; EOI to master PIC
        out     020, al
        pop     ax
        iret                     ; no chain!
    END
END
END
pop     ax
cs jmp  old_9                     ; chain

```

;----- Pop-up routine

```

user_act:
    print     7, 'Hit a key => '
    dos       8
    print     7, 'Bravo!', 13, 10
    ret
    ...

```

Логика процедур `new_8` и `new_9` основана на введении переменной состояния `state`. В примере переменная `state` принимает значения из ряда {sleep, wake, active}; исходное состояние — `sleep`. При обнаружении заданной комбинации клавиш состояние изменяется на `wake`; это значение `state` является заявкой на выполнение `user_act`.

Состояние `state` проверяется с частотой приблизительно равной 18 Гц в процедуре `new_8`. Если `state = wake` и при этом `InDos = 0`, то состояние меняется на `active` и остается таким на все время выполнения `user_act`. После выхода из `user_act` устанавливается исходное состояние `sleep`.

Рассмотрим особенности выполнения `new_8` и `user_act`. Процедура `user_act` вызывается из `new_8` при смене состояния с `wake` на `active`. Эта процедура может выполняться сколь угодно долго, т. к. в ней ожидается ввод символа. При ожидании ввода символа DOS разрешает внешние прерывания, устанавливая флаг `i`. Таким образом, на фоне выполнения `user_act` возникают прерывания и, в частности, заново выполняется та же самая процедура `new_8`. Поскольку `state` не равно `wake`, процесс обработки очередного прерывания от таймера, выполняемый процедурой `new_8`, обходит вызов процедуры `user_act`. При завершении `user_act` флаг `i` сброшен — его первоначальное значение восстанавливается при завершении функции DOS. Что весьма кстати — участок между записью `sleep` в `state` и командой `iret` — критический.

Системная процедура обработки прерывания 8 вызывается в начале `new8` как подпрограмма. Таким образом, системная процедура выполняется с

опережением — для того, чтобы текущая заявка на прерывание была полностью обслужена до входа в долго выполняемую подпрограмму `user_act`.

Особенность процедуры обработки прерывания от клавиатуры для рор-уп программы заключается в том, что обработка комбинации клавиш, которыми активизируется `user_act`, не передается системной процедуре обработки. Процедура `new_9` в этой ситуации сама записывает команду EOI (End Of Interrupt) в контроллер прерываний и сама посылает импульс подтверждения в контроллер клавиатуры.

Активизировать `user_act` из DOS-prompt непросто, т. к. `InDos = 0` только в моменты выполнения программ. При ожидании ввода команд `InDos = 1`. Зато исключение 028 вызывается постоянно.

Примечание

Такое впечатление, что в DOS-prompt выполняется функция 0a — буферизованный ввод строки; напротив, в оболочках типа Norton Commander символы вводятся по одному без ожидания, так что `InDos` почти всегда равен нулю.

Дополним пример из листинга 18.9 процедурой обработки исключения 028. В `new_028` использовать `print` и функцию 8 DOS нельзя (почему?); кроме того, следует сохранить значение стека, которым DOS пользуется при выполнении функций 1—0c. В программе из листинга 18.10 пользовательская процедура с вызовами DOS выполнена в двух вариантах — `user_act1` и `user_act2`. Один вариант применяется при условии `InDos = 0`, а другой — из процедуры обработки прерывания 028.

Перед вызовом `user_act2` из `new_028` содержимое стека сохраняется в памяти TSR-программы (в примере сохраняется 0200 слов, в памяти от PSP). После завершения `user_act2` содержимое стека восстанавливается. Этот способ активизации TSR-процедуры, использующей вызовы DOS, успешно применяется, например, в знакомых нам рор-уп программах `rport` и `digit`.

Примечание

Чтобы процедура `user_act` была универсальной, в ней не должно быть функций низкоуровневого ввода (функции DOS с номерами 1—0c).

Листинг 18.10. Вызов рор-уп процедуры из обработки прерывания 028 (см. `rip_2.b8`)

```
...
jmp      start

org      0400    ; 0..0400 — stack copy

...

buf      db      7, 'Int 028 entry. Enter string => '
buf_sz   equ     $ - buf
```

```

io      macro
        mov     bx, #1
        lea     dx, buf
        mov     cx, buf_sz
        dos     #2

#em

user_act2:
        io      1, 040          ; write to stdout
        io      0, 03f          ; read from stdin
        ret

;----- "I'm not doing much" exception

new_028:
        pushf
        cs call old_028
        cs cmp   state, wake          ; should wake up?
        IF ne
            ired
        END

                                           ; copy stack to mem

        enter   0
        push    es, ds
        pusha
        set_seg es, cs
        mov     di, 0                  ; dest - mem
        set_seg ds, ss
        mov     si, bp                  ; source - stack
        mov     cx, 0200
        cld
        rep movsw
        popa
        pop     ds, es
        leave

        pusha
        push     es, ds
        set_seg es, ds, cs
        mov     state, active
        call    user_act2
        mov     state, sleep
        pop     ds, es
        popa

        enter   0
        push    es, ds

```

```
pusha
set_seg ds, cs
mov     si, 0                      ; source — mem
set_seg es, ss
mov     di, bp                    ; dest — stack
mov     cx, 0200
cld
rep movsw
popa
pop     ds, es
leave
iret
...
```

В процедуре `user_act2` для ввода и вывода используются функции чтения-записи потоков. Содержимое стека DOS при входе в `new_028` копируется (от кадра стека, сформированного командой `enter 0`) в память программы, начиная со смещения 0. После выполнения `user_act2` содержимое стека восстанавливается из памяти программы.

Дополните рор-ур программу из листинга 18.10 функцией выгрузки по нажатию комбинации клавиш `<Alt>+<X>` при активной рор-ур программе. То есть, пока рор-ур программа не активна (не нажата комбинация клавиш `<Ctrl>+<Esc>` или не разрешено использовать функции DOS), нажатие комбинации клавиш `<Alt>+<X>` на эту программу не действует — вообще не имеет к ней отношения. Только после того, как рор-ур программа получила возможность активизации, нажатие комбинации клавиш `<Alt>+<X>` воспринимается как *заявка* на выгрузку.



Часть IV

Приложения

- Приложение 1.** Биты, байты, слова, знаковые и беззнаковые числа
- Приложение 2.** Коды литер в стандарте ASCII
- Приложение 3.** Позиционные коды клавиш
- Приложение 4.** Функции BIOS-DOS
- Приложение 5.** Настройки запуска a86
- Приложение 6.** Операторы и инструкции a86
- Приложение 7.** Совместимость a86 с традиционными ассемблерами
- Приложение 8.** Прерывания от i80x87
- Приложение 9.** Ответы на контрольные вопросы из части I
- Приложение 10.** Ошибки в a86 v4.05
- Приложение 11.** Описание дискеты

ПРИЛОЖЕНИЕ 1

Биты, байты, слова, знаковые и беззнаковые числа

Арифметика — сиречь наука цифирная.

М. Ломоносов

Информация в вычислительной системе представлена набором элементов с двумя состояниями. Принято обозначать эти состояния знаками 0 и 1, а элемент называть двоичной цифрой, или *битом* (bit —от binary digit).

Смысл информации, зашифрованной наборами битов, определяется источником или назначением информации. Один и тот же набор единиц и нулей может означать совершенно разные данные: число в двоичном коде, число в двоично-десятичном коде, число в коде Грея или в позиционном коде; это может быть код литеры в стандарте ASCII, и т. д.

Например, набор бит 00110001 может означать следующее:

- десятичное число 49;
- число тридцать один в упакованном двоично-десятичном формате (см. разд. 14.1);
- фрагмент 32-битного числа в формате с плавающей точкой (см. рис. 15.2); если это старшие восемь бит числа, то в них содержится информация о том, что значение положительное, а двоичный порядок равен -15 ;
- код литеры 1 в формате ASCII (см. приложение 2);
- код, передаваемый в IBM PC процессором клавиатуры при нажатии клавиши <N> (см. приложение 3 и разд. 17.6.2);
- результат чтения регистра маски iMx контроллера прерываний, когда его входы с номерами 0, 4 и 5 закрыты (см. Контроллер прерываний 8259A в разд. 16.4.2);

и т. д.

Таким образом, смысл информации, заданной некоторым набором бит, определяется происхождением или назначением данных, а также системой *кодировки*, принятой для этого типа информации. Вычислительная машина — устройство для обработки *наборов бит*, которые могут означать различные данные, не обязательно только числа. Число — это частный случай данных, лишь одна из возможных трактовок.

Целые числа в ЭВМ *кодируются* в позиционной системе счисления по основанию 2. Каждый бит из последовательности, образующей число, представляет двоичную цифру с весом 2 в степени i , где i — номер бита. Биты нумеруются от нуля; таким образом, вес бита в начале последовательности (т. е. бита с номером 0) равен 1, вес следующего бита — 2, следующего за этим — 4, и т. д. Например, набор бит 110001 представляет следующее число:

$$1 \times 2^0 + 1 \times 2^4 + 1 \times 2^5 = 1 + 16 + 32 = 49$$

В современных ЭВМ вся последовательность бит, образующих память, разделена на группы по 8. Группа из восьми бит называется *байтом*. То есть, память разбита на последовательность байтовых *ячеек* — по 8 бит в каждой ячейке.

Порядковый номер ячейки, или ее *адрес*, используется процессором для доступа к байту; доступ к отдельным битам предполагает задание адреса ячейки и номера бита в ней — от нуля до семи.

Процессор i80x86 позволяет обращаться к нескольким соседним ячейкам как к целому: к паре ячеек (i8086/286) или к четырем ячейкам сразу (начиная с i80386). Пара смежных ячеек образует 16-битное *слово*, а четыре ячейки — *двойное слово* (32-битное). Адресом слова или двойного слова считается адрес байта *в начале* пары или четверки ячеек. Например, адрес слова из пары ячеек, расположенных по адресам 36 и 37, равен 36; а адрес старшего слова в двойном слове из четырех ячеек по адресам 37, 38, 39, 40 равен 39.

При задании чисел биты старшего байта слова имеют вес в 256 раз больше, чем одноименные биты младшего байта, т. е. байта с меньшим адресом. Аналогично, в двойном слове числовое значение каждого последующего байта в 256 раз больше, чем у предыдущего (таким образом, старшее слово в $256 \times 256 = 65536$ раз более "значимо" по сравнению с младшим словом).

Рассмотрим пример. Пусть байты по смежным адресам содержат последовательности бит, показанные в табл. П1.1.

Слово по адресу 36, составленное из пары байт с адресами 36 и 37, содержит набор бит 00000001 00000110, что соответствует числу $256+6=262$. Слово по адресу 37, составленное из пары байт с адресами 37 и 38, содержит последовательность бит 00000000 00000001 — это число 1. Слово по адресу 38 (из байтов с адресами 38 и 39) содержит набор 00000001 00000000, что соответствует двоичному коду числа 256.

Таблица П1.1. Данные для иллюстрации составления слов из ячеек

Адрес	Набор битов	Число
39	00000001	1
38	00000000	0
37	00000001	1
36	00000110	6

Примечание

В системах архитектурной линии Intel принято, что байт с наименьшим адресом в составе слова имеет и наименьший вес — является наименее значимым, что обозначают аббревиатурой *lsb* (*least significand byte*). Аналогично, числовое значение бита с наименьшим номером — в составе байта, слова или двойного слова — тоже наименьшее (равно 1).

Если записать биты из табл. П1.1 подряд в порядке убывания номеров, то наиболее значимые биты в составе байтов, слов и двойных слов окажутся слева. Также слева будут находиться наиболее значимые байты в словах или в двойных словах. В результате такой записи биты внутри байтов и байты внутри слов выстроены в порядке убывания значимости, как показано в табл. П1.2.

Таблица П1.2. Запись битов и байтов в порядке убывания значимости

Адрес ячейки	39	38	37	36
Значения бит	00000001	00000000	00000001	00000110
Нумерация бит в байтах	76543210	76543210	76543210	76543210
Нумерация бит в слове по адресу 36			15 8	7 0
Нумерация бит в слове по адресу 37		15 8	7 0	
Нумерация бит в слове по адресу 38	15 8	7 0		

Такой порядок задания чисел, когда сначала записываются наиболее весомые цифры, кажется привычным и естественным. Номера битов и адреса байтов при такой записи убывают.

В противоположность этому, адреса байтов при записи операторов программы слева направо сверху вниз... возрастают! Порядок задания в исходном тексте программы значений байт из табл. П1.1 и табл. П1.2 показан в табл. П1.3.

Таблица П1.3. Нумерация битов и байтов в исходном тексте

Задание байтов в программе	00000110	00000001	00000000	00000001
Адрес ячейки	36	37	38	39
Нумерация бит в байтах	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
Нумерация бит в слове по адресу 36	7 0	15 8		
Нумерация бит в слове по адресу 37		7 0	15 8	
Нумерация бит в слове по адресу 38			7 0	15 8

В пределах каждого отдельного байта число записано в привычной для нас форме — от старших цифр к младшим. Но в пределах слов порядок битов нарушен: старший бит номер 15 находится во втором байте, а младший — с номером 0 — в первом байте.

Наблюдаемый эффект "перестановки байт" проявляется только при задании слов (двойных слов) набором *байт* (или при задании двойных слов двумя отдельными словами). Если же задавать слово целиком, то в его пределах биты записаны последовательно. Например, слово по адресу 36 из табл. П1.2 задается в двоичной системе счисления как 00000001 00000110.

Во многих задачах из первых двух глав требуется задать слово парой байт; значение слова при этом определяется директивой *db* (define bytes) вместо *dw* (define words). С учетом эффекта "перестановки" байт, слово со значением 1 следует определить как:

```
db        1, 0
```

Следующее определение задает по байтам слово со значением 256:

```
db        0, 1
```

Второй байт находится в ячейке со старшим адресом, поэтому его вес в 256 раз больше, чем у первого байта.

Чтобы задать аналогичным образом слово со значением -2 , следует представить число -2 в дополнительном коде — последовательностью из шестнадцати бит; а затем записать полученный код при помощи директивы `db`, поменяв местами первые и последние 8 битов:

```
dw      1111111111111101xb
db      11111101xb, 11111111xb
```

В конце числа `xb` означает двоичную систему счисления. Аналогичные директивы при использовании десятичной системы счисления:

```
dw      -2
db      -2, -1
```

В этом примере подразумевается кодировка отрицательных целых чисел по схеме, известной под названием *дополнительного кода*. В настоящее время этот код для представления отрицательных чисел — наиболее распространенный.

Дополнительный код задает числа со *знаком*. Знак определяется старшим битом, т. е. седьмым битом байта или пятнадцатым битом слова. Если знаковый бит установлен, то число отрицательное, а если сброшен, то число либо положительное, либо ноль.

Наряду со знаковыми числами, в `i80x86` предусмотрено использование чисел без знака. Старший бит байта или слова при задании беззнакового числа является наиболее значимым, а отрицательные значения не предусмотрены.

Особенность дополнительного кода — в том, что сложение/вычитание *знаковых* чисел выполняется по тем же правилам, что и для *беззнаковых*.

Каким же образом следует кодировать отрицательные числа, чтобы к ним были применимы операции беззнаковой арифметики? Начнем ответ с простого вопроса: каким должно быть число -1 , чтобы при прибавлении к нему числа $+1$ (по обычным правилам двоичной арифметики) получился бы ноль? Если речь идет о байте, оказывается, что число -1 по битам представлено как `11111111`. Проверим:

```
11111111      ; -1
+ 00000001    ; +1
-----
(1)00000000   ; 0
```

Получен ноль, что и требовалось. (Перенос из старшего бита не в счет, в данных он не сохраняется.) Кодировка первого из отрицательных чисел выяснена; получение кодов следующих отрицательных значений не представляет сложности. На практике применяется такое правило: отрицательное значение в дополнительном коде получается из кода беззнакового числа с тем же абсолютным значением, если все биты *проинвертировать* и *прибавить 1* к результату инверсии.

Так, например, число -1 (11111111) получается из числа $+1$ (00000001), и наоборот, следующим образом (стрелке \rightarrow соответствует операция поразрядной инверсии):

$$\begin{array}{rcl}
 11111111 & \rightarrow & 00000000 \\
 + & & 1 \\
 \hline
 & & 00000001 \rightarrow 11111110 \\
 & & + & & 1 \\
 & & \hline
 & & 11111111
 \end{array}$$

Наименьшее (отрицательное) знаковое значение в диапазоне байта — это -128 (код = 10000000); а наибольшее (положительное) — $+127$ (код = 01111111). Если старший бит ноль, то знаковые и беззнаковые значения совпадают (в диапазоне 0—127). При установленном в 1 старшем разряде знаковые и беззнаковые значения получаются различными; например, байт с кодом 11111111 задает знаковое число -1 и беззнаковое число 255.

Инструкции сложения/вычитания в процессоре i80x86 выполняются одинаково для знаковых и беззнаковых данных. В ряде инструкций разница между знаковыми и беззнаковыми данными учитывается.

Примечание

Хотя инструкции сложения/вычитания знаковых и беззнаковых чисел одни и те же, но инструкции деления/умножения — разные (см. разд. 6.3.2).

ПРИЛОЖЕНИЕ 2

Коды литер в стандарте ASCII

В этом приложении приведены данные о кодировке литер в стандарте *American Standard Code for Interface Interchange* (ASCII). В табл. П2.1 содержатся коды печатных, или видимых литер, в диапазоне 32—127.

Таблица П2.1. ASCII-коды видимых литер

Dec	Hex	Литера	Dec	Hex	Литера	Dec	Hex	Литера
32	020	<пробел>	47	02F	/	62	03E	>
33	021	!	48	030	0	63	03F	?
34	022	"	49	031	1	64	040	@
35	023	#	50	032	2	65	041	A
36	024	\$	51	033	3	66	042	B
37	025	%	52	034	4	67	043	C
38	026	&	53	035	5	68	044	D
39	027	'	54	036	6	69	045	E
40	028	(55	037	7	70	046	F
41	029)	56	038	8	71	047	G
42	02A	*	57	039	9	72	048	H
43	02B	+	58	03A	:	73	049	I
44	02C	,	59	03B	;	74	04A	J
45	02D	-	60	03C	<	75	04B	K
46	02E	.	61	03D	=	76	04C	L

Таблица П2.1 (окончание)

Dec	Hex	Литера	Dec	Hex	Литера	Dec	Hex	Литера
77	04D	M	94	05E		111	06F	o
78	04E	N	95	05F		112	070	p
79	04F	O	96	060	'	113	071	q
80	050	P	97	061	a	114	072	r
81	051	Q	98	062	b	115	073	s
82	052	R	99	063	c	116	074	t
83	053	S	100	064	d	117	075	u
84	054	T	101	065	e	118	076	v
85	055	U	102	066	f	119	077	w
86	056	V	103	067	g	120	078	x
87	057	W	104	068	h	121	079	y
88	058	X	105	069	i	122	07A	z
89	059	Y	106	06A	j	123	07B	{
90	05A	Z	107	06B	k	124	07C	:
91	05B	[108	06C	l	125	07D	}
92	05C	\	109	06D	m	126	07E	~
93	05D]	110	06E	n	127	07F	□

Примечание

Вторая половина кодовой таблицы (в диапазоне 128—255) с литерами национального алфавита здесь не показана. В отладчике d86 литеры из второй половины кодовой таблицы отображаются только в режиме Raw Text (r); а в форматах Ascii (a) и Char (c) они выводятся двойными литерами, составленными из знака # (или \$) и литеры из приведенной здесь первой половины кодовой таблицы.

Табл. П2.2 содержит стандартные названия невидимых управляющих "литер" и их коды, в диапазоне 0—31.

Таблица П2.2. Управляющие коды

Dec	Hex	Аббревиатура	Название
0	0	NUL	Null (end of string)
1	1	SOH	Start Of Heading

Таблица П2.2 (окончание)

Dec	Hex	Аббревиатура	Название
2	2	STX	Start Of teXt
3	3	ETX	End Of teXt
4	4	EOT	End Of Transmission
5	5	ENQ	ENQuery
6	6	ACK	ACKnowledge
7	7	BEL	Bell
8	8	BS	BackSpace
9	9	HT	Horizontal Tab
10	0A	LF	Line Feed
11	0B	VT	Vertical Tab
12	0C	FF	Form Feed
13	0D	CR	Carriage Return
14	0E	SO	Shift Out
15	0F	SL	Shift In
16	010	DLE	Data Line Escape
17	011	DC1	Device Ctrl 1 (X-ON)
18	012	DC2	Device Ctrl 2
19	013	DC3	Device Ctrl 3 (X-OFF)
20	014	DC4	Device Ctrl 4
21	015	NAK	Negative AcKnowledge
22	016	SYN	Synchronous idle
23	017	ETB	End of Transmit Block
24	018	CAN	CANsel
25	019	EM	End of Medium
26	01A	SUB	SUBstitute
27	01B	ESC	ESCape
28	01C	FS	File Separator
29	01D	GS	Group Separator
30	01E	RS	Record Separator
31	01F	US	Unit Separator

ПРИЛОЖЕНИЕ 3

Позиционные коды клавиш

Процессор клавиатуры передает в IBM PC байт данных при нажатии и при отпускании клавиши. В старшем бите данных содержится признак нажатия/отпускания (при нажатии сброшен в ноль), а в младших семи битах — *позиционный код*, или *скан-код* для идентификации клавиши. Позиционные коды стандартной клавиатуры приведены в табл. ПЗ.1.

Таблица ПЗ.1. Скан-коды стандартной клавиатуры

Код	Клавиша	Код	Клавиша
1	<Esc>	16	<Q>
2	<1>	17	<W>
3	<2>	18	<E>
4	<3>	19	<R>
5	<4>	20	<T>
6	<5>	21	<Y>
7	<6>	22	<U>
8	<7>	23	<I>
9	<8>	24	<O>
10	<9>	25	<P>
11	<0>	26	<[> (<{>)
12	<-> (<_>)	27	<]> (<}>)
13	<=> (<+>)	28	<Enter>
14	<Backspace>	29	Правая клавиша <Ctrl>
15	<Tab>	30	<A>

Таблица ПЗ.1 (окончание)

Код	Клавиша	Код	Клавиша
31	<S>	59	<F1>
32	<D>	60	<F2>
33	<F>	61	<F3>
34	<G>	62	<F4>
35	<H>	63	<F5>
36	<J>	64	<F6>
37	<K>	65	<F7>
38	<L>	66	<F8>
39	<.;> (<.:>)	67	<F9>
40	<'> (<">)	68	<F10>
41	<'> (<~>)	69	<NumLock>
42	Правая клавиша <Shift>	70	<ScrollLock>
44	<Z>	71	<7> (<Home> на цифровой клавиатуре)
45	<X>	72	<8> (<↑> на цифровой клавиатуре)
46	<C>	73	<9> (<PgUp> на цифровой клавиатуре)
47	<V>	74	<—> на цифровой клавиатуре
48		75	<4> (<←> на цифровой клавиатуре)
49	<N>	76	<5> на цифровой клавиатуре
50	<M>	77	<6> (<→> на цифровой клавиатуре)
51	<.,> (<<>>)	78	<+> на цифровой клавиатуре
52	<.> (<>>)	79	<1> (<End> на цифровой клавиатуре)
53	</> (<?>)	80	<2> (<↓> на цифровой клавиатуре)
54	Правая клавиша <Shift>	81	<3> (<PgDn> на цифровой клавиатуре)
55	<*> на цифровой клавиатуре	82	<0> (<Insert> на цифровой клавиатуре)
56	Правая клавиша <Alt>	83	<.> (на цифровой клавиатуре)
57	<Пробел>	87	<F11>
58	<Caps Lock>	88	<F12>

В табл. ПЗ.2 приведены скан-коды для расширенной клавиатуры. Клавиши расширенной клавиатуры генерируют два байта данных: сначала байт со значением 224, а затем уже скан-код.

Примечание

Расширенная клавиатура — это не то, что дополнительная клавиатура. Дополнительная клавиатура (Keypad) есть набор клавиш с цифрами и стрелками, расположенных как в калькуляторе. Расширенная клавиатура — это клавиши, добавленные к некогда бывшей "стандартной" клавиатуре. В этой клавиатуре нет отдельных клавиш-стрелок; стрелки имеются только на Keypad, вместе с цифрами.

Таблица ПЗ.2. Скан-коды расширенной клавиатуры

Код	Клавиша	Код	Клавиша
28	<Enter> на цифровой клавиатуре	75	<←→>
29	Правый <Ctrl>	77	<→→>
53	</> на цифровой клавиатуре	79	<End>
56	Правая клавиша <Alt>	80	<↓>
71	<Home>	81	<PageDown>
72	<↑>	82	<Insert>
73	<PageUP>	83	<Delete>

Примечание

Информация по клавишам <PrintScreen> и <Pause/Break> здесь не приведена. Коды этих клавиш расширены многократно, причем число байт и их кодировка сложным образом зависят от одновременного нажатия управляющих клавиш.

ПРИЛОЖЕНИЕ 4

Функции BIOS-DOS

В этом приложении приведено описание функций BIOS и DOS — в основном, тех, что используются в примерах программ или хотя бы упоминаются.

Функции BIOS для работы с клавиатурой

Базовая система ввода/вывода (BIOS) предоставляет три функции для работы с клавиатурой. Функции выполняются системным драйвером клавиатуры по запросу `int 016`. Номер функции на входе задается в регистре `ah`, результат возвращается в `ah` и `al`.

Функция 0 — чтение символа с ожиданием. Символ считывается из буфера ввода, который заполняется системной процедурой обработки прерываний от клавиатуры. Если буфер ввода пуст, функция ожидает появления в нем символа.

Результат: `ah` — скан-код, `al` — код ASCII; `al = 0` означает, что скан-код расширенный (см. приложение 3).

Функция 1 — проверка наличия символов в буфере ввода.

Результат: если флаг `z` установлен, буфер пуст; иначе, при `z = 0`, в `ah` возвращается скан-код, а в `al` — код ASCII (т. е. результат — как при выполнении функции 0, только символ *не удаляется* из буфера ввода).

Функция 2 — чтение байта состояния специальных клавиш.

Результат: регистр `al` содержит копию байта из области данных BIOS по адресу 0:0417. Для всех битов этого байта 1 означает, что *в данный момент* нажата специальная клавиша; соответствие между битами и клавишами показано в табл. П4.1.

Таблица П4.1. Биты байта состояния специальных клавиш

Бит	Нажатая клавиша	Бит	Нажатая клавиша
7	<Ins>	3	<Alt>
6	<CapsLock>	2	<Ctrl>
5	<NumLock>	1	<Shift> (левый)
4	<ScrollLock>	0	<Shift> (правый)

Функции прерывания 021 DOS

При обращении к DOS через программное прерывание 021 (командой `int 021`) предварительно следует задать номер функции в регистре `ah`. Адрес массива данных в функциях DOS, как правило, определяется парой `ds:dx`.

Функции низкоуровневого ввода/вывода

Ввод выполняется из стандартного входного потока, связанного, по умолчанию, с клавиатурой. Вывод идет в стандартный выходной поток, который направлен, по умолчанию, на дисплей.

Функции ввода

Функция 1 — ввод символа с ожиданием и с эхо-печатью.

Функция 8 — ввод символа с ожиданием, без эха.

Результат: в `al` — ASCII-код введенного символа. Если ноль, значит, нажата клавиша с расширенным кодом и требуется повторное чтение для получения второго байта со скан-кодом (см. приложение 3).

Примечание

Функция 1 отличается наличием эхо-печати: при нажатии клавиши соответствующая ей литера автоматически выводится на экран. При выполнении этих функций DOS проверяет признак нажатия комбинации клавиш `<Ctrl>+<Break>`, что позволяет прекратить выполнение программы.

Функции вывода

Функция 2 — вывод на экран символа, заданного в регистре `dl`.

Функция 9 — вывод строки с адреса `ds:dx` до литеры `$`.

Примечание

Функции 2 и 9 выявляют и специальным образом обрабатывают *управляющие символы*: коды 0d (CR) или 0a (LF) вызывают возврат курсора начало строки и переход на следующую строку, код 7 (Bell) вызывает звуковой сигнал, а код 8 (BS) — перевод курсора влево (см. приложение 2).

Функции для работы с файлами и потоками

Начиная с версии 2.0, в DOS реализованы файловые функции в стиле UNIX. При открытии или создании файла DOS возвращает числовой код открытого файла — файловый манипулятор. При последующих операциях чтения, записи, позиционирования или закрытия, на файл ссылаются при помощи полученного в начале кода.

Стандартные потоки ввода/вывода

Файловые манипуляторы с номерами 0—4 по умолчанию связаны с всегда открытыми стандартными потоками ввода/вывода.

Таблица П4.2. Стандартные потоки ввода/вывода

Номер файлового манипулятора	Поток
0	Стандартный ввод (по умолчанию, клавиатура)
1	Стандартный вывод (по умолчанию, экран)
2	Стандартный вывод для ошибок (всегда экран)
3	Стандартное дополнительное устройство (AUX или COM1)
4	Стандартный вывод на печать (PRN или LPT1)

Стандартные потоки автоматически инициализируются при запуске программы. Для работы с ними используются функции чтения-записи 03f/040; открывать и закрывать стандартные потоки не требуется.

Примечание

Имеется возможность при запуске программы "перенаправить" потоки 0 и 1, связав их с файлами. Например, при вызове `prog <test.in >test.out` стандартный вход подключен не к клавиатуре, а к файлу `test.in`; стандартный вывод направлен не на экран, а в файл `test.out`.

Функции чтения и записи

Функции чтения и записи используются со стандартными потоками, а также с файлами — открытыми или заново созданными.

Функция 03f — чтение из потока или из файла

Входные параметры:

- ☐ `bx` — файловый манипулятор;
- ☐ `cx` — число байт для чтения;
- ☐ `ds:dx` — адрес буфера для результатов чтения.

Результат: если флаг `c` = 0, то операция успешна, в `ax` записано число прочитанных байт; если `c` = 1, чтение прошло с ошибкой, в `ax` записан код ошибки.

Примечание

Данные считываются с текущей позиции, указатель файла при этом перемещается вперед. Если `ax` < `cx`, это означает, что при чтении *из файла* достигли конца файла. При чтении *из стандартного потока* выходное значение `ax`, как правило, меньше значения `cx` — в `cx` задана максимальная длина строки, а `ax` показывает длину введенной строки, включая завершающий код CR.

Функция 040 — запись в поток или в файл

Входные параметры:

- ☐ `bx` — файловый манипулятор;
- ☐ `cx` — число байт для записи;
- ☐ `ds:dx` — адрес буфера с данным для записи.

Результат: если флаг `c` = 0, то операция успешна, в `ax` записано число записанных байт; если `c` = 1, чтение прошло с ошибкой, в `ax` записан код ошибки.

Примечание

Запись начинается в текущей позиции файла, после записи указатель перемещается вперед. Если после выполнения функции `ax` <> `cx`, то, вероятно, нет свободного места на диске.

Функции для создания, открытия и закрытия файлов

Для доступа к файлу его следует открыть или создать заново. При этом DOS возвращает файловый манипулятор для идентификации файла при чтении, записи и закрытии.

Функция 03с — создание файла

Функция 03с создает пустой файл — заново, уничтожая содержимое существующего файла. Входные параметры:

- ☐ `ds:dx` — адрес ASCIIZ-строки с именем файла;
- ☐ `cx` — атрибут файла (кроме метки тома).

Результат: если $c = 1$, в ax — код ошибки; иначе, в ax — манипулятор созданного файла.

Имя файла на входе задается строкой с null-кодом в конце (так называемая строка ASCIIZ). Второй параметр — атрибут файла — это байт с признаками в битах 0—5, значение которых показано в табл. П4.3. Биты 6 и 7 не используются.

Таблица П4.3. Атрибуты файла

Бит	Название	Значение
0	Read only	Из файла можно читать, но записывать в него нельзя
1	Hidden	Файл невидим для команд DOS: <code>dir</code> , <code>copy</code> , <code>del</code> и т. д.
2	System	Аналогично биту Hidden (унаследовано от системы CP/M)
3	Volume	Файл представляет собой метку тома в корневом каталоге логического диска
4	Subdirectory	Файл представляет собой каталог
5	Archive	Обнуляется после архивирования утилитой <code>backup</code> ; при внесении исправлений вновь устанавливается в 1

Некоторые наиболее часто встречающиеся значения атрибута:

- ☐ 0 — обычный файл;
- ☐ 8 — метка тома (только в корневом каталоге);
- ☐ 010 — каталог;
- ☐ 020 — "архивный" файл, не восстановленный посредством утилиты `backup` или `xcopy`.

Функция 03d — открытие файла

Для доступа к существующему файлу, его следует сначала "открыть". Результат открытия — получение файлового манипулятора; указатель текущей позиции файла устанавливается в начало.

Параметры на входе:

- ☐ `al` — режим доступа (0 — только чтение, 1 — только запись, 2 — чтение и запись);
- ☐ `ds:dx` — адрес ASCIIZ-строки с именем файла.

Результат: если $c = 1$, то операция выполнена неудачно, в ax — код ошибки; иначе, в регистре ax записан код манипулятора, назначенного файлу при открытии.

Функция 03e — закрытие файла

После завершения операций чтения-записи файл следует закрыть. При этом на диске сохраняется содержимое буферов вывода, связанных с закрываемым файлом, корректируется время и дата модификации (если была хоть одна запись) и освобождается файловый манипулятор.

При вызове функции 03e файловый манипулятор задается в регистре `bx`. Нулевое значение флага `c` на выходе свидетельствует об успешном выполнении функции.

Примечание

При завершении программы функцией 04c (вместо вызова прерывания 020) закрытие файлов выполняется автоматически.

Функции для завершения программы

Функция 04c — завершение выполнения программы с возвращением кода завершения, заданного в регистре `al`. Код завершения под именем `ErrorLevel` доступен в `batch`-файлах.

Примечание

При использовании этой функции автоматически закрываются файлы, открытые программой. В альтернативном варианте, при завершении программы посредством вызова прерывания 020, файлы не закрываются автоматически; кроме того, в регистре `cs` должен содержаться сегментный адрес PSP, что в `exe`-программах с несколькими сегментами кода может оказаться проблемой.

Функция 031 — завершение резидентной программы.

Параметры вызова: `dx` — размер резидентной части в параграфах по 16 байт (что позволяет устанавливать TSR-программу размером больше 64 Кбайт), `al` — код завершения, аналогично функции 04c.

Примечание

Рекомендуется предварительно освободить блок памяти с копией переменных окружения (вызовом функции 049 со значением `es` равным значению слова из PSP по смещению 02c).

Функции для TSR-программ

С резидентными программами связаны функции чтения-записи вектора прерывания, функция 031, функция для получения адреса байта с признаком "DOS занята", и функция для удаления блока памяти.

Примечание

Работа с файлами в TSR-программе требует применения дополнительных функций, которые здесь не приведены.

Функция 025 — запись в вектор содержимого пары регистров `ds:dx`; номер вектора задается в регистре `al`.

Функция 035 — чтение вектора в пару регистров `es:bx`; номер вектора задается в регистре `al`.

Функция 034 — чтение адреса байта "InDos" в пару регистров `es:bx`.

Примечание

Когда байт по этому адресу равен нулю, вызовы DOS из TSR-программы разрешены.

Функция 049 — освобождение ранее распределенного блока памяти; адрес блока на входе задается в регистре `es`; функция выполнена успешно, если на выходе из функции `c = 0`.

Примечание

Функция 049 используется для работы с динамической памятью, которая резервируется и освобождается в процессе выполнения программы. В контексте TSR-программ эта функция используется при загрузке — для удаления копии переменных окружения, и при выгрузке — для удаления резидентной части программы.

ПРИЛОЖЕНИЕ 5

Настройки запуска a86

Командная строка для вызова a86 начинается с имени программы (a86), за которым следуют опции (ключи), имена исходных файлов для трансляции и необязательные имена выходных файлов. По умолчанию выходным файлам назначается имя первого входного файла.

Выходные файлы со стандартными расширениями (com, sym, lst, xrf) могут быть указаны в любом месте командной строки. Для получения выходных файлов с нестандартными расширениями следует сначала задать исходные файлы, а затем слово TO. После TO перечисляются имена выходных файлов в следующем порядке:

- ☐ имя исполняемого или объектного модуля;
- ☐ имя файла с таблицей символов;
- ☐ имя файла-листинга;
- ☐ имя файла с таблицей перекрестных ссылок.

Исходные файлы, образующие текст программы, могут быть заданы групповым именем с метасимволами * и ?. Так, например, команда `a86 *.8` заказывает трансляцию исходного текста, собранного из всех файлов текущего каталога с расширением 8. Исходные файлы передаются на вход транслятора в алфавитном порядке.

При помощи знака "=" в командной строке имеется возможность определить при запуске a86 произвольное имя, доступное в программе. Имя указывается после знака "=" и воспринимается ассемблером так, как если бы оно было определено в исходной программе. Порядок задания имен имеет значение. Так, например, если вызов a86 задан в виде `a86 =ver1 prog1.8 =debug prog2.8`, то к началу трансляции prog1.8 определено имя ver1, а к началу трансляции prog2.8 — имя debug.

Имя, заданное после "=", определено значением 1. Если между "=" и именем поставлен знак логической инверсии "!", то имя определено значением 0. Например, при вызове `a86 =ver1 prog1.8 !=ver1 prog2.8` имя `ver1` при трансляции `prog1.8` определено значением 1 (true), а при трансляции `prog2.8` — значением 0 (false).

В отличие от пользовательских имен, порядок задания ключей значения не имеет: заданный ключ применяется ко всем исходным файлам и не может быть изменен.

Ключи с одинаковым знаком допускают объединение; например: ряд ключей `+L49 +C +O +S` может быть записан сокращенно как `+L49COS`.

Все опции `a86` по умолчанию выключены (ключи заданы со знаком "-").

Ключи, или опции запуска a86

- ❑ `+C` — запись имен в `obj-` и `sum-`файлы *без преобразования* строчных букв (из так называемого нижнего регистра) в заглавные (т. е. в буквы верхнего регистра). Тем не менее, при трансляции регистр не имеет значения (например, имена `area` и `Area` обозначают один и тот же объект). В `obj-` и `sum-`файлы имя записывается так, как оно было задано в последней из директив `public` или `extrn`; если имя не входит в списки `public/extrn`, оно сохраняется в том виде, в котором впервые появилось в исходном тексте.

Примечание

Опцию `+C` рекомендуется применять, когда предполагается компоновка полученного `obj-`модуля с независимо разработанными `obj-` и `lib-`модулями, например, при объединении ассемблера с языком C.

- ❑ `+c` — отключение преобразования букв в пользовательских именах, как при трансляции, так и при выводе ее результатов; имена `area` и `Area` в этом случае обозначают разные объекты.
- ❑ `+D` — для констант, начинающихся с нуля, устанавливает десятичную систему счисления; в результате, константа `010` обозначает число десять, а не шестнадцать.
- ❑ `+E` — запрещает запись сообщений об ошибках в исходный файл; копия исходного текста с включенными сообщениями записывается в файле с расширением `err`. По умолчанию, сообщения записываются в исходный файл, только если он находится в текущем каталоге (иначе все-таки создается `err-файл`).
- ❑ `+f` — в `obj-`режиме включает программную эмуляцию сопроцессора: команды `i80x87` транслируются в вызовы подпрограмм из стандартной библиотеки для вычислений с плавающей точкой, предоставляемой Microsoft, Borland и пр.

- ❑ **+F** — включает режим автоматической вставки инструкций `fwait` перед каждой инструкцией сопроцессора `i8087` в системах на базе `i8086/186`. Задавать эту опцию при трансляции для процессора `i80286` не обязательно. (Тип процессора определяется `a86` или автоматически, или задается опцией `P`.)
- ❑ **+G n** — набор опций для улучшения совместимости с традиционными ассемблерами `masm/tasm/wasm`. Число `n` представляет сумму `G`-опций: например, `+G10` задает пару `G`-опций с кодами 2 и 8.

Коды `G`-опций:

- 1 — включает режим генерирования длинных (трехбайтных) инструкций для безусловных переходов вперед на *локальную* метку (например, для `jmp >11`). По умолчанию, генерируется короткий (двухбайтный) переход, в предположении, что локальная метка должна находиться поблизости.
- 2 — отключает оптимизацию инструкций `lea`. По умолчанию `a86` заменяет `lea` на `mov`, когда это возможно. Например, инструкция `lea si, area` транслируется в команду `mov si, offset area`.
- 4 — включает режим генерации ссылок в `obj`-файлах, используемый утилитой `MIX` компилятора `Power C`.
- 8 — при необходимости принимать решение о типе имени (при опережающей ссылке) задает выбор типа "memory" (`byte`, `word`, `dword`). По умолчанию, `a86` выбирает для неопределенных имен тип `abs`. (Неопределенные имена в командах `jmp` и `call` считаются типа `abs`, независимо от задания этой опции).
- 16 — требует в `obj`-режиме явного объявления неопределенных (внешних) имен директивой `extrn`. По умолчанию, внешними считаются все необъявленные имена.

- ❑ **+N n** — при выводе листинга задает `<n>` дополнительных строк для отображения дампа (шестнадцатеричных значений байтов, сгенерированных оператором ассемблера).
- ❑ **+I n** — задает начальную позицию в строке листинга для вывода исходного текста; тем самым, определяет число позиций для вывода дампа. Максимальное значение `<n>` — 127. По умолчанию, исходный текст выводится с позиции 34, так что в начале строки остается место не более чем для шести байт дампа (вывод каждого байта занимает по три символа). Для отключения вывода номеров строк значение `<n>` следует задать на 128 больше.
- ❑ **+L [n]** — задает создание файла-листинга. Необязательный параметр `<n>` — это сумма кодов `L`-опций. Коды 1—3 — взаимоисключающие.

Коды L-опций:

- 1 — блокирует вывод директив условной трансляции и текста, исключенного в результате действия этих директив.
- 2 — блокирует вывод текста, исключенного директивами условной трансляции; сами директивы отображаются.
- 3 — разрешает вывод директив условной трансляции и текста, исключенного в результате их действия (опция по умолчанию).
- 4 — включает отображение операторов, полученных в результате макроподстановок; иначе, в листинг помещаются только вызовы макрокоманд.
- 16 — включает вывод счетчика адресов на всех строках листинга, независимо от того, была ли распределена память при трансляции оператора ассемблера. Иначе, значение счетчика адресов выводится только на тех строках, трансляция которых привела к увеличению счетчика.
- 32 — включает вывод таблицы символов в конце листинга.

По умолчанию, если после ключа +L ничего не указано, код <n> равен 39, что означает:

- включен вывод таблицы символов (задан L-код 32);
- включен вывод значения счетчик адресов только на тех строках, трансляция которых привела к резервированию памяти (L-код 16 не задан);
- включен вывод результатов макрорасширений (L-код 4 задан);
- разрешено отображение директив условной трансляции и текста, исключенного в результате их действия (L-код 3 задан).

Примечание

Листинг генерируется и без задания ключа +L, если в командной строке задано имя выходного файла с расширением `lst`.

❑ +O — генерирование `obj`-файла для последующей компоновки; по умолчанию генерируется исполняемый `com`-файл.

❑ +P n — задает тип процессора, который будет выполнять полученную программу. По умолчанию, `a86` ориентируется на тип процессора, на котором он запущен. Значения <n>:

- 0 — 8088/8086;
- 1 — 8088/86/186;
- 2 — 8088/86/186/286.

Примечание

Если тип процессора < 2, включается режим автоматической вставки команд `fwait` перед каждой инструкцией сопроцессора; для отмены этого режима предусмотрена опция `+F`.

- ❑ `+S` — блокирует создание файла с таблицей имен (`sym`-файла).
- ❑ `+X` — включает генерирование файла перекрестных ссылок (`xrf`).

Переменная окружения `a86`

При запуске `a86` в конец командной строки добавляется содержимое одноименной переменной окружения. Например, если в `autoexec.bat` задана команда `SET a86=+O`, то опция `+O` будет включена в каждый вызов `a86` по умолчанию.

Опции, явно указанные в командной строке, имеют высший приоритет. Если при вызове `a86` задан ключ `-O`, то ключ `+O` из переменной окружения `a86` теряет силу.

Если включить в переменную `a86` имена исходных файлов, то эти файлы будут транслироваться всегда и в первую очередь. Это позволяет автоматически подключать файлы макроопределений. Например, команда `SET a86=c:\work\mas.inc` обеспечивает трансляцию файла `mas.inc` из каталога `c:\work` вместе с любой программой.

ПРИЛОЖЕНИЕ 6

Операторы и инструкции a86

В этом приложении представлены в алфавитном порядке операторы и инструкции. *Операторы* образуют выражения, которые вычисляются при трансляции. *Инструкции* обозначают в символическом виде коды машинных команд. Разница между операторами и инструкциями в том, что первые действуют только *при трансляции*, а вторые — исключительно *при выполнении* программы. Некоторые из инструкций a86 задают набор из нескольких машинных команд, т. е. являются *встроенными макрокомандами*.

Операторы

Операторы приведены в табл. П6.1 в порядке убывания приоритета. Символы *k* и *n* означают числа, имена или выражения, а также, в ряде случаев, регистры или строки в кавычках, что следует из описания действия или результата операции. Звездочкой отмечается оператор (или вариант его действия), характерный только для a86.

Таблица П6.1. Операторы в порядке убывания приоритета

Операция	Действие или результат	Приоритет
()	Изменение очередности операций	12
[k]	Преобразование значения <i>k</i> типа <i>abs k</i> к типу <i>mem</i> (данные в памяти)	
k . n	Сумма <i>k</i> и <i>n</i> (используется обычно для обозначения полей структур, где <i>k</i> — адрес структуры в памяти, <i>n</i> — имя поля)	11
k n	Сумма <i>n</i> и <i>k</i> (пробел — только разделитель). Операнды при вычислении выражения меняются местами, поэтому если это переменные разной размерности, то тип выражения = типу <i>второго</i> операнда (<i>n</i>)	

Таблица П6.1 (продолжение)

Операция	Действие или результат	Приоритет
$k \text{ ptr } n$	Сумма k и n (используется в выражениях типа <code>byte ptr 0</code>)	10
$*b/byte \ k$	Преобразование значения k к типу <code>byte</code> (1 байт)	
$*w/word \ k$	Преобразование значения k к типу <code>word</code> (2 байта)	
$*d/dword \ k$	Преобразование значения k к типу <code>dword</code> (4 байта)	
$*q/qword \ k$	Преобразование значения k к типу <code>qword</code> (8 байт)	
$*t/tbyte \ k$	Преобразование значения k к типу <code>tbyte</code> (10 байт)	
$offset \ k$	Преобразование значения k к типу <code>abs</code>	
$seg \ k$	Сегментный адрес k (в <code>com</code> -режиме возвращает <code>cs</code>)	
$type \ k$	Число байтов, занятых k (если тип = <code>abs</code> , то 0)	
$ref \ k$	True (–1), если на имя k есть ссылка, а иначе false (0)	
$def \ k$	True (–1), если имя k определено, а иначе false (0)	
$*bit \ n$	Число, в котором n -й бит установлен в 1, а остальные сброшены (биты нумеруются от нуля)	9
$high \ n$	Значение старшего байта числа n	8
$low \ n$	Значение младшего байта числа n	
k / n	Целая часть результата деления k на n	
$k \bmod n$	Остаток от деления k на n	
$k * n$	Произведение чисел k и n	
$k \text{ shl } n$	Сдвиг двоичного кода числа k на n бит влево	
$k \text{ shr } n$	Сдвиг значений разрядов числа k вправо на n бит	
$k + n$	Сумма k и n	7
$k - n$	Разность между k и n	
$k \text{ lt } n$	True (–1), если $k < n$, иначе false (0)	7
$k \text{ le } n$	True (–1), если $k \leq n$, иначе false (0)	
$k \text{ gt } n$	True (–1), если $k > n$, иначе false (0)	
$k \text{ ge } n$	True (–1), если $k \geq n$, иначе false (0)	
$k \text{ ne } n$	True, если k не равно n ($*n$ и k могут быть строками в " ")	
$k \text{ eq } n$	True, если k равно n ($*n$ и k могут быть строками в " ")	
$*k = n$	True, если строки k и n совпадают с точностью до регистра	

Таблица П6.1 (окончание)

Операция	Действие или результат	Приоритет
!k	True, если k не определен (*или определен как 0); false, если k определен ненулевым значением	6
not k	Поразрядная инверсия числа k	
k and n	Поразрядное логическое умножение чисел k и n	5
k or n	Поразрядное логическое сложение чисел k и n	4
k xor n	Поразрядное исключающее "или" чисел k и n	
k:n	Дальний указатель в сегмент k по смещению n (например, dd 0:0417 или jmp 0ffff:0)	3
k:[n]	Ячейка по смещению n в сегменте, заданном сегментным регистром k (например, mov ax, es:[bx])	
*k by n	Число, старший байт которого равен k, а младший n	
short k	Метка k находится в пределах 127 байт от команды jmp	2
long k	Метка k находится в пределах 64 Кбайт от jmp	
st(n)	Номер n регистра FPU; оператор st в a86 сохранен для совместимости с masm при кодировании инструкций fpu; например, fadd st(0) равносильно fadd 0	
\$, this	Текущее значение счетчика адресов	
*#k else n	Параметр #k, если он задан в макровывозе, а иначе — n	
k dup n	Повтор списка n через запятую k раз; для формирования списка значений после db/dw/dd/dq/dt	1

Инструкции

Ниже приведены инструкции в алфавитном порядке. При их описании приняты следующие обозначения.

Условные обозначения

В разделах "Псевдокод" алгоритм выполнения инструкции записан с использованием логических блоков структурного ассемблера bsp86 (см. разд. 12.4.1).

Также применяются некоторые очевидные конструкции языков высокого уровня:

- комментарии — в парных скобках (*** и ***);
- символы =, <>, >=, <=, >, <, означающие сравнение двух значений;
- оператор *x := y*, означающий присвоение у переменной *x*.

Типы операндов обозначаются метасимволами: "r" — регистр, "m" или "mem" — данные в памяти, "imm" — непосредственный операнд. Для уточнения размерности после "r", "m" или "imm" записывается число бит — 8, 16 или 32. Например, "r/m16" означает, что в качестве операнда допускается 16-битный регистр или слово в памяти.

Содержимое регистра обозначается его именем, например, имя "ax" означает и регистр ax, и его содержимое. Имя регистра в квадратных скобках [] означает содержимое памяти по адресу, равному величине регистра. Например, "es:[di]" обозначает данные в сегменте памяти, номер которого задан регистром es, по смещению, равному значению регистра di. Если сегментный регистр не указан, значение сегмента данных берется из регистра ds. (Круглые скобки ничего не означают.)

Обозначение "src" — это операнд-источник. Если операндов два, то на языке ассемблера источник принято записывать после приемника. Приемник обозначается как "dst" и записывается на языке ассемблера первым операндом после мнемоники инструкции. Если в команде оба операнда источники, они обозначаются как "LeftSrc" и "RightSrc".

Звездочкой отмечается форма задания инструкции, характерная только для a86.

Общие правила установки флагов

В разделах "Флаги после выполнения" встречаются ссылки на общие правила установки флагов. Вот эти правила (см. также *разд. 6.3.6*):

- c — устанавливается, если *беззнаковый* результат вне диапазона;
- o — устанавливается, если *знаковый* результат вне диапазона (знаковое переполнение);
- z — устанавливается, если результат равен нулю;
- s — устанавливается, если старший (знаковый) бит результата равен 1, т. е. результат — *отрицательное* число;
- e — устанавливается, если младший байт (!) результата содержит четное количество бит, установленных в 1 (четный паритет).

Воздействие команд на флаги

Воздействие команд на флаги состояния, по группам команд, показано в табл. П6.2.

Таблица П6.2. Воздействие команд на флаги состояния

Флаги состояния	o	c	s	z	e
Сложение и вычитание					
ADD ADC SUB SBB					
CMP NEG CMPS SCAS	+	+	+	+	+
Инкремент и декремент					
INC DEC	+	—	+	+	+
Умножение и деление					
MUL IMUL	+	+	?	?	?
DIV IDIV	?	?	?	?	?
Логические операции					
AND OR XOR TEST	0	0	+	+	+
Сдвиги и вращения					
SHL SHR (1)	+	+	+	+	+
SHL SHR (>1)	?	+	+	+	+
SAR	0	+	+	+	+
ROL ROR RCL RCR (1)	+	+	—	—	—
ROL ROR RCL RCR (>1)	?	+	—	—	—

Обозначения к таблице П6.2

Обозначение	Значение флага
—	Не изменяется
+	Изменяется в соответствии с результатом
1	Устанавливается в единицу
0	Сбрасывается в ноль
?	Меняется непредсказуемо (может быть, и предсказуемо, но не для всех типов процессоров)

AAA — Ascii Adjust after Addition

ASCII-коррекция после сложения

Псевдокод

```
IF (младшая тетрада al > 9) или (a = 1)
    al := al + 6
    ah := ah + 1
    a := 1
    c := 1
ELSE
    c := 0
    a := 0
END
старшая тетрада al := 0
```

Описание

Выполняется после команды `add` с результатом в регистре `al`. Младшая тетрада в операндах `add` должна находиться в диапазоне BCD чисел 0—9. В этом случае `aaa` корректирует `al` так, что в нем содержится первая десятичная цифра суммы.

Если при сложении имел место перенос в следующую десятичную цифру, команда `aaa` увеличивает содержимое `ah` и устанавливает в 1 флаги переноса и промежуточного переноса. Если переноса не было, флаги `c` и `a` обнуляются, а значение `ah` не изменяется. Старшая тетрада `al` обнуляется в любом случае.

Примечание

Если входное значение `al > 0f9`, к `ah` прибавляется дополнительная единица. Смысл этой операции непонятен, т. к. результат сложения не может быть настолько большим даже при суммировании BCD-чисел в виде ASCII-кодов.

Для преобразования цифры, полученной в `al`, в код ASCII '0'—'9', после `aaa` следует выполнить инструкцию `or al, '0'`.

Флаги после выполнения

Флаги `a` и `c` устанавливаются, как описано выше. Значения флагов `o`, `s`, `z` и `e` не определены.

Пример

```
mov    ax, 0108    ; в al — цифра 8, в ah — цифра 1
add     al, 6       ; ax = 010e, a = 0
aaa     ; ax = 0204, c = a = 1
```

AAD — ASCII Adjust before Division

ASCII-коррекция перед делением

aad	Преобразование неупакованного BCD-числа в двоичное число
*aad imm8	Упаковка байтов ax, вес старшего байта = imm8

Псевдокод

```
IF (aad) THEN
    k = 10
ELSE (* aad imm8 *)
    k = imm8
END
al := ah * k + al
ah := 0
```

Описание

Традиционная форма команды `aad` (без непосредственного операнда) применяется для преобразования двух цифр числа в неупакованном BCD-формате (младшая цифра — в `al`, старшая — в `ah`) в двоичное число. Эта операция выполняется перед делением и умножением, т. к. результаты выполнения этих двоичных операций над BCD-числами скорректировать невозможно.

Ассемблер `a86` допускает задание в `aad` непосредственного операнда со значением множителя. Значение множителя хранится в машинном коде `aad` и, как выясняется, может быть любым в диапазоне `0—0ff`. Эта недокументированная возможность процессора `i80x86` позволяет, например, преобразовать командой `aad 16` неупакованное BCD-число из `ax` в упакованное BCD-число в `al`.

Примечание

Несмотря на "ASCII" в названии команды, старшие тетрады байтов `ax` должны быть *обнулены*. То есть, например, число сорок пять перед выполнением `aad` может быть задано как `045`, но никак не литерами '4' и '5'.

Флаги после выполнения

Флаги `s`, `z`, `e` устанавливаются по общим правилам; значения флагов `o`, `a` и `c` не определены.

Примеры

```
mov ax, '45' ; в ax — ASCII-число
and ax, 0f0f ; обнуление старших тетрад в ax
```

```
aad                ; в ax — двоичное число 45
. . .
mov    ax, 0405    ; в ax — неупакованное BCD
aad    16           ; в al — упакованное BCD (045)
```

AAM — ASCII Adjust after Multiply

ASCII-коррекция после умножения

aam	Преобразование двоичного числа в неупакованное BCD
*aam imm8	Получение двух цифр al в системе счисления по основанию imm8

Псевдокод

```
IF (aam) THEN
    k = 10
ELSE (* aam imm8 *)
    k = imm8
END
ah := al / k
al := al MOD k
```

Описание

Команда `aam` в традиционной форме (без непосредственного операнда) преобразует двоичное число из `al` в неупакованное BCD-число в `ax`. Чтобы преобразование было корректным, входное значение должно быть меньше 100. Преобразование выполняется за счет деления `al` на 10; целая часть результата помещается в `ah`, остаток — в `al`.

Примечание

Название команды выбрано неудачно. Во-первых, цифры получаются не в коде ASCII ('0'—'9'), а в неупакованном BCD-формате с нулями в старших тетрадах (0—9). Во-вторых, эта команда не обязательно следует за умножением; она выполняется после *серии вычислений* над двоичными значениями — для преобразования конечного результата в неупакованный BCD-формат.

Ассемблер `a86` допускает задание в `aam` непосредственного операнда со значением делителя (по умолчанию, десять). Значение делителя непосредственно отражено в машинном коде `aam` и, как выясняется, может быть задано произвольным в пределах 0—0ff (допускается даже ноль, хотя это не имеет практического смысла; причем деление на ноль при выполнении `aam` не

приводит к генерированию исключения по вектору 0). Эта недокументированная возможность i80x86 позволяет, в частности, командой `aam 16` преобразовать *упакованное* BCD-значение из `al` в *неупакованное* BCD-число в `ax`.

Флаги после выполнения

Флаги `s`, `z`, `e` устанавливаются по общим правилам; значения флагов `o`, `a` и `c` не определены.

Примеры

```
mov    al, 45      ; в al — двоичное число 45
aam                    ; в ax — неупакованное BCD (0405)
. . .
mov    al, 067     ; в ax — упакованное BCD (067)
aam    16          ; в ax — неупакованное BCD (0607)
```

AAS — ASCII Adjust after Subtraction

ASCII-коррекция после вычитания

Псевдокод

```
IF (младшая тетрада al > 9) или (a = 1)
    al := al - 6
    ah := ah - 1
    a := 1
    c := 1
ELSE
    c := 0
    a := 0
END
старшая тетрада al := 0
```

Описание

Выполняется после команды `sub` с результатом в `al`. Младшая тетрада в операндах `sub` должна находиться в диапазоне 0—9. В этом случае `aas` корректирует `al` так, что в нем содержится первая десятичная цифра разности.

Если при вычитании единиц произошел заем из цифры десятков, `aaa` уменьшает содержимое `ah` и устанавливает в `1` флаги переноса и промежуточного (межтетрадного) переноса. Если заема нет, флаги `c` и `a` обнуляются, `ah` не изменяется. Старшая тетрада `al` обнуляется в любом случае.

Для преобразования цифры, полученной в `al`, в код ASCII '0'—'9', после `aas` следует выполнить инструкцию `or al, '0'`.

Флаги после выполнения

Флаги `a` и `s` устанавливаются, как описано выше. Значения флагов `o`, `s`, `z` и `e` не определены.

Пример

```
mov    ax, 0205    ; в al — цифра 5, в ah — цифра 2
sub     al, 8        ; ax = 02fd, a = 1
aas                      ; ax = 0107, c = a = 1
```

ADC — Add with Carry

Сложение с переносом

Псевдокод

```
dst := dst + src + c
```

Описание

Команда `adc` вычисляет сумму операндов и значения флага переноса. Результат записывается на место первого операнда (`dst`). Эта команда обычно применяется в операциях сложения повышенной размерности, когда слагаемые образуются последовательностями байт или слов, и необходимо учитывать распространение переноса.

Флаги после выполнения

Флаги `o`, `s`, `z`, `a`, `c`, `e` устанавливаются по общим правилам.

Пример

```
mov     ax, -1
mov     dx, 1        ; dx:ax = 01ffff (131071)
add     ax, 3        ; dx:ax := dx:ax +
adc     dx, 2        ; + 020003 (131075)
                      ; dx:ax = 040002 (262146)
```

ADD — Addition

Сложение

Псевдокод

```
dst := dst + src
```

Описание

Команда `add` вычисляет сумму операндов и записывает результат на место первого операнда (`dst`).

Флаги после выполнения

Флаги `o`, `s`, `z`, `a`, `c`, `e` устанавливаются по общим правилам.

Пример

```
mov    al, -4      ; al = -4 = 252
add    al, 3        ; al = -1 = 255, s = e = 1, o = z = a = c = 0
```

AND — Logical AND

Логическое "И"

Псевдокод

```
dst := dst AND src
c := 0
o := 0
```

Описание

Выполняет логическое умножение пар одноименных разрядов `dst` и `src`, записывая результат в `dst`.

Флаги после выполнения

Флаги `c` и `o` сброшены; `e`, `s` и `z` устанавливаются по общим правилам. Флаг `a` сбрасывается — это проверено на i8086/286/386/486/586, хотя, согласно документации Intel, значение `a` не определено.

Пример

```
mov    dl, 'a'      ; dl = 1100001xb
and    dl, not bit 5 ; dl = 1000001xb = 'A'
```

BOUND — Check Array Index Against Bounds

Проверка индекса массива на выход за границы

Псевдокод

```
IF (LeftSrc < [RightSrc]) или (LeftSrc > [RightSrc + 2])
    вызов исключения 5
END
```

Описание

Проверяет, находится ли знаковая величина LeftSrc в границах, заданных парой слов по адресу RightSrc. В качестве LeftSrc может использоваться только 16-разрядный регистр. Если значение в LeftSrc меньше значения слова [RightSrc] или больше значения слова [RightSrc+2], генерируется исключение 5. Команда предназначена для проверки индексов при доступе к массивам, в языках высокого уровня.

Примечание

При входе в прерывание 5 по этой инструкции в стеке сохраняется адрес возврата на эту инструкцию, а не на следующую.

Пример

```
range    dw    -3, 2
        . . .
mov      ax, -3      ; dl = 1100001xb
bound    ax, range   ; -3 <= ax <= +2,
                    ; и прерывание 5 не генерируется
```

CALL — Call Procedure

Вызов процедуры

call imm16	Ближний вызов по непосредственному адресу
call r/m16	Ближний вызов по адресу из регистра или слова в памяти
call imm16:imm16	Дальний вызов по непосредственному указателю
call m32	Дальний вызов по указателю, заданному двойным словом в памяти

Псевдокод

```
CASE операнд
| imm16
    Push (IP)
    IP := imm16
| r/m16
    Push (IP)
    IP := [r/m16]
| imm16:imm16
    Push (CS)
    Push (IP)
    CS:IP := imm16:imm16
| m32
    Push (CS)
    Push (IP)
    IP := [m32]
    CS := [m32 + 2]
END
```

Описание

Вызывает процедуру по адресу, заданному операндом. Сохраняет при этом информацию об адресе следующей инструкции (адрес возврата), помещая ее в стек. Эта информация используется командой возврата из процедуры.

Ближний (*near*) вызов, создаваемый ассемблером при задании операнда типа *r/m16* или *abs*, не изменяет значение *cs*. Операнд типа *abs* задает переход по непосредственному смещению. Операнды типа *r/m16* обозначают 16-рядный регистр или слово в памяти, где задан адрес перехода. При ближнем вызове в стеке сохраняется значение *ip* для следующей (нижележащей) команды. Это значение будет записано в *ip* командой возврата *ret*, которая выполняется в вызванной процедуре.

Дальний (*far*) вызов, генерируемый ассемблером при задании операнда в виде *abs:abs* или *m32*, изменяет значение не только *ip*, но и *cs*. Адрес перехода определяется 4-байтным указателем, в младшем слове которого задано смещение, а в старшем слове — значение сегмента. Этот указатель в команде *call* задан либо непосредственно (*abs:abs*), либо значением двойного слова в памяти (форма *m32*).

При дальнем вызове в стеке сохраняются текущее значение регистра *cs* и смещение следующей инструкции. Сохраненные значения *cs* и *ip* следующей инструкции восстанавливаются из стека командой *retf* в вызванной процедуре.

Непосредственный дальний вызов процедуры, находящейся в том же сегменте, что и команда вызова, в а86 оптимизируется (когда вызов и процедура расположены в одном исходном модуле). Вместо дальнего вызова в форме `call imm:imm` ассемблер а86 генерирует пару команд:

```
push    cs
call    imm    ; непосредственный ближний вызов
```

Запись `cs` перед вызовом необходима для правильного возврата (по команде `retf`) из процедуры, рассчитанной на *дальний* вызов.

Внимание!

В com-программе задавать дальний вызов процедуры по имени не следует. Такой вызов, если он не оптимизирован, как показано выше, будет представлен в форме `call 0:imm` — с правильным смещением, но с *неопределенным* значением сегмента, поскольку оно неизвестно до загрузки программы. Поскольку com-программа не предусматривает коррекции непосредственных значений сегментов при загрузке, ноль в сегментной составляющей нулем и останется. В итоге, вызов будет неправильным. В версии 4.05 на оптимизацию в com-режиме рассчитывать не приходится (см. приложение 10). Поэтому, как в традиционных ассемблерах, так и в а86 v4.05 при написании com-программ дальние вызовы процедур по имени следует оптимизировать *вручную*.

Примеры

```
farptr dd    0ffff:0        ; reboot "procedure"
ptr     dw    offset dream
dream:
    ret

hell:
    retf

    call    dream        ; imm — ближний вызов по адресу dream
    lea     ax, dream
    call    ax            ; r16 — ближний вызов по адресу dream в ax
    call    ptr           ; m16 — ближний вызов по адресу dream в ptr
    call    0ffff:0       ; imm:imm — дальний вызов
                           ; по непосредственному указателю
    call    farptr        ; m32 — дальний вызов
                           ; по указателю в двойном слове
    push    cs            ; внутрисегментный вызов
    call    hell          ; дальней процедуры
```

CBW — Convert Byte to Word

Преобразование байта в слово

Псевдокод

`ax := знаковое расширение al`

Описание

Преобразует знаковый байт из `al` в знаковое слово в `ax`. Выполняется за счет копирования значения знакового бита регистра `al` во все биты регистра `ah`.

Пример

```
mov    al, -2      ; al = 11111110xb
cbw                    ; ax = 1111111111111110xb = -2
mov    al, 127     ; al = 01111111xb
cbw                    ; ax = 0000000001111111xb = +127
```

CLC — Clear Carry Flag

Сброс флага переноса

Псевдокод

`c := 0`

Описание

Обнуляет флаг переноса, или беззнакового переполнения в регистре флагов.

CLD — Clear Direction Flag

Сброс флага направления

Псевдокод

`d := 0`

Описание

Обнуляет флаг направления, в результате чего строковые операции будут увеличивать значения используемых ими индексных регистров.

CLI — Clear Interrupt Flag

Сброс флага разрешения прерываний

Псевдокод

```
i := 0
```

Описание

Запрещает внешние прерывания по входу INTR процессора.

CMC — Complement Carry Flag

Инвертирование флага переноса

Псевдокод

```
c := NOT c
```

Описание

Изменяет значение флага переноса на обратное.

CMR — Compare

Сравнение

Псевдокод

```
LeftSrc — RightSrc
```

Описание

Выполняет сравнение операндов, вычитая второй операнд из первого. По результату вычитания устанавливает значения флагов, но, в отличие от инструкции вычитания `sub`, не сохраняет разность. Эта инструкция обычно применяется в паре с инструкцией условного перехода.

Примечание

Несмотря на то, что первый операнд не является приемником, он не может быть задан в виде *непосредственного* значения — допустимые сочетания операндов у инструкции `cmp` такие же, как у `sub`.

Флаги после выполнения

Флаги o, s, z, a, e, c устанавливаются по общим правилам.

Пример

```
mov    al, -2      ; al = -2 = 254
cmp     al, 3
jnb     m2          ; 254 > 3, поэтому перехода на m2 не будет
j1l     m1          ; -2 < +3, поэтому переход на m1 произойдет
```

CMPS/CMPSB/CMPSW — Compare String Operands

Сравнение элементов строк

cmps LeftSrc, RightSrc	Сравнение элементов двух массивов, исходя из типа операндов
cmpsb/cmpsw	Сравнение байтов/слов из двух массивов
*cmpsb/cmpsw imm8	Генерирование imm8 экземпляров инструкции cmpsb/cmpsw

Псевдокод

```
IF байтовая операция
  ds:[si] — es:[di] (* сравнение байтов *)
  IF d = 0
    IncDec := 1
  ELSE
    IncDec := -1
  END
ELSE
  ds:[si] — es:[di] (* сравнение слов *)
  IF d = 0
    IncDec := 2
  ELSE
    IncDec := -2
  END
END
si := si + IncDec
di := di + IncDec
```

Описание

Сравнивает пару байт или слов, адреса которых заданы содержимым регистров ds:si и es:di. Сравнение выполняется за счет вычисления разности между операндами по адресам ds:si и es:di.

Примечание

Порядок операндов в операции вычитания ($[si] - [di]$) не соответствует названиям регистров — Source Index и Destination Index. Сравнение производится как бы в обратном порядке.

Результат сравнения отражается в установке флагов, сама разность не сохраняется. После сравнения корректируются значения указателей *si* и *di*. Если флаг направления сброшен (была выполнена инструкция *cld*), то указатели увеличиваются; если флаг *d* установлен, указатели уменьшаются. Приращение указателей равно 1 при сравнении байт, а при сравнении слов — 2.

Форма *cmps* не имеет машинного аналога. Встроенная макрокоманда *cmps* задает машинную инструкцию *cmpsb* или *cmpsw*, исходя из типа "операндов" (оба должны быть одинакового типа — или *word*, или *byte*). Имена массивов, заданные в параметрах макрокоманды *cmps*, на самом деле никак не связаны с текущей установкой указателей *es:di* и *ds:si* — эти имена используются *только* для определения размерности операции. Порядок "операндов" и вовсе не имеет значения.

Примечание

Вариант *cmps* был придуман еще в первом ассемблере для i80x86 ради возможности настройки размерности операции. В итоге, лишив смысла значения "операндов" и порядок их задания, получили иллюзию улучшения и реальный источник ошибок.

Использование инструкций *cmpsb/cmpsw* с префиксом повторения *repe* или *repne* позволяет организовать последовательность сравнений до несовпадения или совпадения. Счетчик-ограничитель предварительно записывается в регистр *cx* (см. REP/REPE/REPZ/REPNE/REPNZ).

Форма *cmpsb/cmpsw imm8* введена в а86 для задания *imm8* экземпляров инструкции. (Применительно к *cmps*-командам эта форма не имеет практического использования; аналогичные формы для других строковых команд имеют смысл.)

Флаги после выполнения

Флаги состояния устанавливаются по общим правилам.

Пример

```
lea    si, src
lea    di, dst
cmpsb
jne     ml           ; 1 = 1, поэтому перехода не будет
cmpsb   ; 2 - (-3 = 253)
```

```
ja      m1      ; 2 < 253, поэтому перехода не будет
jg      m2      ; 2 > -3, поэтому переход произойдет
. . .
src  db      1, 2
dst  db      1, -3
```

CWD — Convert Word to Doubleword

Преобразование слова в двойное слово

Псевдокод

```
IF ax < 0
    dx := 0ffff
ELSE
    dx := 0
END
```

Описание

Преобразует знаковое значение из `ax` в знаковое двойное слово в `dx:ax`. Выполняется за счет копирования знакового бита `ax` во все биты `dx`.

DAA — Decimal Adjust after Addition

Десятичная коррекция после сложения

Псевдокод

```
IF (младшая тетрада al > 9) или (a = 1)
    al := al + 6
    a := 1
ELSE
    a := 0
END
IF (al > 09f) или (c = 1)
    al := al + 060
    c := 1
ELSE
    c := 0
END
```

Описание

Корректирует значение `al` после сложения двух упакованных BCD-цифр. В байте `al` находятся две десятичных цифры, команда `daa` корректирует результат и записывает в `c` перенос в третью десятичную цифру.

Флаги после выполнения

Флаги `a` и `c` устанавливаются, как описано выше; флаги `o`, `s`, `z`, `e` — по общим правилам.

Пример

```
mov    ax, 098      ; в al — две десятичных цифры
add    al, 6         ; al = 09e
daa                    ; ax = 4, a = 1, c = 1 (перенос в сотни)
adc    ah, 0         ; ax = 0104 — три значащих десятичных цифры
```

DAS — Decimal Adjust after Subtraction

Десятичная коррекция после вычитания

Псевдокод

```
IF (младшая тетрада al > 9) или (a = 1)
    al := al - 6
    a := 1
ELSE
    a := 0
END
IF (al > 09f) или (c = 1)
    al := al - 060
    c := 1
ELSE
    c := 0
END
```

Описание

Корректирует значение `al` после вычитания пары упакованных BCD-цифр. В `al` находятся две десятичных цифры, команда `das` формирует в них правильный результат с образованием заема из третьей цифры во флаге `c`.

Флаги после выполнения

Флаги `a` и `c` устанавливаются, как описано выше; флаги `o`, `s`, `z`, `e` — по общим правилам.

Пример

```
mov    ax, 016      ; в al — две десятичных цифры
sub    al, 8         ; al = 0d
das                    ; ax = 8, a = 1, c = 0 (нет заема из сотен)
```

DEC — Decrement by 1

Уменьшение на 1

Псевдокод

```
dst := dst — 1
```

Описание

Вычитает из операнда число 1. В отличие от аналогичной команды `sub` с непосредственным операндом 1, инструкция `dec` не воздействует на флаг переноса.

Флаги после выполнения

Флаг `c` не изменяется; флаги `o`, `s`, `z`, `a`, `e` устанавливаются по общим правилам.

DIV — Unsigned Divide

Беззнаковое деление

div r/m8	Беззнаковое деление ax на байт r/m8 (al := частное, ah := остаток)
div r/m16	Беззнаковое деление dx:ax на слово r/m16 (ax := частное, dx := остаток)

Псевдокод

```
IF r/m8 (* делитель — байт *)
    частное := ax / (r/m8)
    IF частное помещается в байт
        al := частное
        ah := ax MOD (r/m8)
    ELSE
        вызов исключения 0
END
```

```
ELSE      (* делитель — слово *)
    частное := (dx:ax) / (r/m16)
    IF частное помещается в слово
        ax := частное
        dx := (dx:ax) MOD (r/m16)
    ELSE
        вызов исключения 0
    END
END
```

Описание

Выполняет деление беззнаковых значений с получением беззнаковых результатов. Делимое адресуется неявно; единственный операнд задает *дели- тель*. Размерность операции определяется ассемблером исходя из типа опе- ранда. Если операнд задан байтом, то делимое — ax, а целая часть частного и остаток сохраняются на месте старшего и младшего байта делимого. Если операнд задан словом, то делимое — dx:ax, а целая часть частного и остаток сохраняются на месте старшего и младшего слова делимого.

При переполнении, когда значение частного не помещается в байт (при де- лении слова на байт) или в слово (при делении двойного слова на слово), i80x86 вызывает исключение 0, сохраняя в стеке адрес команды div (i8086 сохраняет адрес *следующей* инструкции после div).

Флаги после выполнения

Значения флагов o, s, z, e, c не определены.

Пример

```
mov     ax, 5*11+7
mov     bl, 11
div     bl      ; al = 5, ah = 7
mov     ax, 11*256
div     bl      ; переполнение (256 > 255)
```

ENTER — Make Stack Frame for Procedure Parameters

Установка кадра стека для параметров процедуры

*enter imm16 enter imm16, 0	Установка кадра стека с уровнем вложенности ноль и с резерви- рованием imm16 байтов локальных данных
enter imm16, imm8	Установка кадра стека с уровнем вложенности imm8 и с резер- вированием imm16 байтов локальных данных

Псевдокод

```
(* для уровня 0 *)  
Push (bp)  
bp := sp  
sp := sp - imm16
```

Описание

Команда `enter` позволяет организовать произвольный доступ к данным в стеке, в частности, доступ к тем параметрам процедуры, которые были переданы ей в стеке.

Примечание

Вариант со значением второго операнда (`imm8`) > 0 , предназначенный для организации доступа к параметрам вложенных процедур в языках типа Pascal, не рассматривается.

Команда сохраняет `bp` в стеке и устанавливает `bp` равным текущему значению `sp`. Первый операнд задает количество байт для локальных данных процедуры. Память для этих данных резервируется в стеке за счет смещения `sp` вниз.

В результате выполнения `enter imm16` регистр `bp` указывает на копию своего первоначального значения в стеке. Ниже (по отрицательным смещениям) находятся локальные данные, а выше — адрес возврата, а затем параметры — в порядке, обратном порядку их записи в стек.

Примечание

Перед выходом из процедуры кадр стека следует удалить при помощи команды `leave`.

IDIV — Signed Divide

Знаковое деление

<code>idiv r/m8</code>	Знаковое деление <code>ax</code> на байт <code>r/m8</code> (<code>al</code> := частное, <code>ah</code> := остаток)
<code>idiv r/m16</code>	Знаковое деление <code>dx:ax</code> на слово <code>r/m16</code> (<code>ax</code> := частное, <code>dx</code> := остаток)

Псевдокод

```
IF r/m8 (* делитель — байт *)  
частное := ax / (r/m8)
```

```

IF частное помещается в байт
    al := частное
    ah := ax MOD (r/m8)
ELSE
    вызов исключения 0
END
ELSE (* делитель — слово *)
    частное := (dx:ax) / (r/m16)
    IF частное помещается в слово
        ax := частное
        dx := (dx:ax) MOD (r/m16)
    ELSE
        вызов исключения 0
    END
END
END

```

Описание

Выполняет деление знаковых значений с получением знаковых результатов. Делимое адресуется неявно; единственный операнд задает *делитель*. Размерность операции определяется ассемблером исходя из типа операнда. Если операнд задан байтом, то делимое — *ax*, а целая часть частного и остаток сохраняются на месте старшего и младшего байта делимого. Если операнд задан словом, то делимое — *dx:ax*, а целая часть частного и остаток сохраняются на месте старшего и младшего слова делимого.

При переполнении, когда значение частного не помещается в байт (при делении слова на байт) или в слово (при делении двойного слова на слово), i80x86 вызывает исключение 0, сохраняя в стеке адрес команды *div*, вызвавшей исключение.

Примечание

При знаковом делении, в принципе, возможны два варианта получения остатка. Например, $-5 / 3$ дает частное -1 и остаток -2 , или, иначе, частное может быть равным -2 , а остаток — $+1$. В i80x86 применяется первый вариант, когда знак *остатка* равен знаку *делимого*.

Флаги после выполнения

Значения флагов *o*, *s*, *z*, *e*, *c* не определены.

Пример

```

mov    ax, -5
cbw                ; dx:ax = -5
mov    bx, 3
div    bx          ; ax = -2, dx = +1

```

IMUL — Signed Multiply

Знаковое умножение

imul r/m8	ax := al * (байт r/m)
imul r/m16	dx:ax := ax * (слово r/m)

Описание

Выполняет знаковое умножение. Если в качестве операнда указан байтовый регистр или байт в памяти, то второй сомножитель всегда находится в `al`, а результат удвоенной размерности помещается в регистр `ax`. Аналогично, если операнд задан словом, то второй операнд берется из `ax`, а результат умножения помещается в `dx:ax`.

Флаги после выполнения

Значение флагов `s`, `z`, `a`, `e` не определено. Флаги знакового (`o`) и беззнакового (`c`) переполнения обнуляются, если результат умножения байтов/слов помещается в байт/слово. Иначе, флаги `o = c = 1`.

Пример

```
mov    al, -15
imul   al    ; ax := al * al = +225, c = o = 1
          ; в al не осталось места для знакового бита,
          ; знак результата представлен ah
```

IN — Input from Port

Ввод из порта

in al, imm8	Ввод из порта по адресу imm8 в регистр al
in ax, imm8	Ввод из пары портов с адреса imm8 в регистр ax
in al, dx	Ввод из порта по адресу, заданному dx, в регистр al
in ax, dx	Ввод из пары портов с адреса, заданного dx, в регистр ax

Описание

Передает байт или слово данных из порта или пары портов (ячеек в пространстве ввода/вывода), что определяется размерностью первого операнда (`al` или `ax`). Для доступа в пределах всего адресного пространства ввода/вывода `0—0ffff`, адрес задается `dx` и используется вариант команды с `dx` в качестве второго операнда. Для доступа в диапазоне адресов `0—0ff` имеется

альтернативный вариант инструкции с заданием непосредственного значения адреса во втором операнде.

Примечание

При доступе к паре портов адрес должен быть четным.

Пример

```
mov    dx, 03fb
mov    al, bit 7
out    dx, al          ; вывод числа 080 в регистр управления COM1
mov    dx, 03f8
in     ax, dx          ; ввод в ax 16-разрядного регистра
                        ; делителя частоты COM1
. . .
in     al, 041         ; чтение в al регистра счетчика первого
                        ; канала таймера
```

INC — Increment by 1

Увеличение на 1

Псевдокод

```
dst := dst + 1
```

Описание

Прибавляет к операнду число 1. В отличие от аналогичной команды `add` с непосредственным операндом 1, команда `inc` не затрагивает установку флага `c`.

Флаги после выполнения

Флаг `c` не изменяется; флаги `o`, `s`, `z`, `a`, `e` устанавливаются по общим правилам.

INS/INSB/INSW — Input from Port to String

Ввод из порта в строку

ins r/m8, dx insb	Ввод в es:[di] байта из порта по адресу dx
ins r/m16, dx insw	Ввод в es:[di] слова из пары портов по адресам dx и dx+1

Псевдокод

```
IF байтовая операция
  es:[di] := порт по адресу DX
  IF d = 0
    IncDec := 1
  ELSE
    IncDec := -1
  END
ELSE
  es:[di] := пара портов с адреса DX
  IF d = 0
    IncDec := 2
  ELSE
    IncDec := -2
  END
END
di := di + IncDec
```

Описание

Передаёт данные из порта (ячейки в пространстве адресов ввода/вывода), адрес которого задан содержимым регистра `dx`, в системную память по адресу `es:di`. В отличие от команды `in`, инструкция `ins` не допускает задания номера порта непосредственным значением — только через регистр `dx`.

Примечание

При доступе к паре портов адрес в `dx` должен быть четным.

После передачи данных указатель `di` автоматически корректируется. Его значение увеличивается, если флаг направления сброшен (была выполнена команда `cld`), иначе уменьшается. Шаг изменения `di` при вводе байта равен 1, а при вводе слова — 2.

Инструкции `ins r/m8`, `dx` соответствует машинная инструкция `insb`, а инструкции `ins r/m16`, `dx` — машинная инструкция `insw`. Соответствие устанавливается ассемблером по типу первого операнда.

Примечание

Первый операнд определяет только *размерность* операции. Он может быть задан любым байтом или словом, например, `al`, или `ch`, или `w[0103]`. Указанный регистр или ячейка памяти не имеет отношения к адресации строки в машинной команде — запись в любом случае выполняется по адресу `es:di`.

Инструкция `ins` используется также с префиксом повторения `rep`, для повторного ввода из одного и того же порта. Предварительно в `cx` записывается счетчик повторений.

INT/INTO — Call to Interrupt Procedure

Вызов процедуры обработки прерывания

int imm8	Прерывание по номеру, заданному непосредственным значением
into	Прерывание номер 4, если установлен флаг переполнения

Псевдокод

```
Push (flags)
i = 0 (* запрет внешних прерываний *)
t = 0 (* запрет трассировки *)
Push (cs)
Push (ip)
IF into
    num := 4
ELSE
    num := imm8
END
cs := 0:[num * 4 + 2]
ip := 0:[num * 4]
```

Примечание

Последующее описание операции применимо не только к перечисленным инструкциям программных прерываний, но и ко всем типам прерываний.

Описание

Вызывает программным способом процедуру обработки прерывания. Непосредственный операнд в диапазоне 0—255 задает индекс элемента в таблице векторов прерываний. Инструкция `into` вызывает прерывание по неявно заданному вектору 4, если установлен флаг `o`.

Таблица векторов представляет собой массив дальних указателей. Первые 32 вектора зарезервированы Intel для системных прерываний; часть их используется для внутренних прерываний (исключений).

Команда `int imm8` выполняется как дальний вызов подпрограммы, с той разницей, что до записи в стек адреса возврата в него включается копия регистра флагов. Процедура обработки прерывания возвращает управление инструкции `iret`, которая восстанавливает из стека регистр флагов и местоположение исполняемой инструкции.

Команда `int imm8` сохраняет в стеке флаги, значения `cs` и `ip` (в таком порядке), а затем передает управление по адресу, который хранится в таблице векторов по индексу `imm8`.

IRET — Interrupt Return

Возврат из прерывания

Псевдокод

```
ip := Pop()  
cs := Pop()  
flags := Pop()
```

Описание

Считывает три верхние записи стека, сформированные при входе в прерывание, в регистры `ip`, `cs` и в регистр флагов. В результате, эти записи удалены, а выполнение прерванной программы возобновлено.

J<x> — Jump if Condition is Met

Условный переход

Псевдокод

```
IF <x> = true  
    IP := dst  
END
```

Описание

Команда условного перехода, или ветвления, задается в виде `j<x> dst`, где `<x>` — условие, `dst` — адрес перехода, типа `abs`. Управление передается на `dst` при выполнении условия `<x>`; а иначе программа продолжается со следующей команды.

Все условия, кроме `sxz`, основаны на флагах, как показано в табл. П6.3. Значения флагов отражает результат выполнения предыдущих команд (например, команды сравнения `cmp`). Отношения "less" и "greater" отражают результат сравнения знаковых чисел, а "above" и "below" применяются для анализа беззнаковых результатов.

Поскольку одно и то же сочетание состояний флагов может быть интерпретировано несколькими способами, ассемблер предлагает для одного условия несколько обозначений. Например, команды `je` и `jz` обозначают одну машинную команду — переход при установленном флаге `z`. Мнемоникой `jz` (Jump if Zero) удобнее пользоваться при анализе результата вычитания, а мнемоника `je` (Jump if Equal) более уместна после сравнений.

Для всех условий $\langle x \rangle$, основанных на флагах, ассемблер предлагает обратное условие в виде $\langle nx \rangle$. Обратные условия удобны при задании пропуска, или обхода фрагмента программы; условие выполнения $\langle x \rangle$ при этом преобразуется в условие невыполнения — $\langle nx \rangle$.

Условие в команде `jcxz` основано не на флагах, а на текущем значении регистра `cx`. Переход выполняется, если `cx = 0`. Обычно команду `jcxz` ставят перед началом цикла, управляемого командой `loop`.

Таблица П6.3. Условия переходов $\langle x \rangle$ в командах $j\langle x \rangle$

$\langle x \rangle$	Название условия	Установка флагов
a	Above ($>$)	$z = c = 0$
ae	Above or Equal (\geq)	$c = 0$
b	Below ($<$)	$c = 1$
be	Below or Equal (\leq)	$c = 1$ или $z = 1$
c	Carry	$c = 1$
cxz	CX register is Zero	
e	Equal ($=$)	$z = 1$
z	Zero	$z = 1$
g	Greater ($>$)	$z = 0$ и $s = 0$
ge	Greater or Equal (\geq)	$s = 0$
l	Less ($<$)	$s \langle \rangle 0$
le	Less or Equal (\leq)	$z = 1$ и $s \langle \rangle 0$
na	Not Above ($!>$)	$c = 1$ или $z = 1$
nae	Not Above or Equal ($! \geq$)	$c = 1$
nb	Not Below ($!<$)	$c = 0$
nbe	Not Below or Equal ($! \leq$)	$z = c = 0$
nc	Not Carry	$c = 0$
ne	Not Equal ($!=$)	$z = 0$
ng	Not Greater ($!>$)	$z = 1$ или $s \langle \rangle 0$
nge	Not Greater or Equal ($! \geq$)	$s \langle \rangle 0$
nl	Not Less ($!<$)	$s = 0$
nle	Not Less or Equal ($! \leq$)	$z = 0$ и $s = 0$
no	Not Overflow	$o = 0$

Таблица П6.3 (окончание)

<x>	Название условия	Установка флагов
np	Not Parity	e = 0
ns	Not Sign	s = 0
nz	Not Zero	z = 0
o	Overflow	o = 1
p	Parity	e = 1
pe	Parity Even	e = 1
po	Parity Odd	e = 0
s	Sign	s = 1
z	Zero	z = 1

JMP — Jump

Безусловный переход

jmp imm	Ближний переход по непосредственному адресу
jmp r/m16	Ближний переход по адресу из регистра или слова в памяти
jmp imm:imm	Дальний переход по непосредственному указателю
jmp m32	Дальний переход по указателю, заданному двойным словом в памяти

Псевдокод

```
CASE операнд
| imm
    IP := imm
| r/m16
    IP := [r/m16]
| imm:imm
    CS:IP := imm:imm
| m32
    IP := [m32]
    CS := [m32 + 2]
END
```


LAHF — Load Flags into AH Register

Загрузка флагов в регистр ah

Псевдокод

ah := s:z?:a?:e?:c

Описание

Копирует младший байт регистра флагов в регистр ah. Соответствие между битами ah и флагами отражено в табл. П6.4.

Таблица П6.4. Расположение флагов в младшем байте регистра флагов

Номер бита	7	6	5	4	3	2	1	0
Флаг	s	z	?	a	?	e	?	c

LEA — Load Effective Address

Загрузка эффективного адреса

Описание

Команда задается в виде lea r16, mem. Вычисляет эффективный адрес (смещение) данных в памяти, результат записывает в регистр. Если адресация второго операнда прямая, a86 и tasm преобразуют lea в mov. Например, lea ax, b array преобразуется в mov ax, offset array.

Если адресация косвенная (пример — lea si, [di+bx]), то смещение заранее неизвестно (зависит от значения регистров) и вычисляется при каждом выполнении команды.

Команда lea может использоваться для получения в произвольном 16-разрядном регистре суммы константы и регистров (только пары базовый-индексный, или одного из них):

```
lea    ax, [di+bx-010]    ; mov    ax, di
                                ; add    ax, bx
                                ; add    ax, -010
```

Примечание

Второй операнд может быть задан регистром, но в этом случае при выполнении команды генерируется fault-исключение 6. Такая ошибка при использовании a86 может возникнуть, если вызвать a86 с ключом +g2 — тогда, например, lea ax, bx не будет преобразована в mov ax, bx.

LEAVE — High Level Procedure Exit

Выход из процедуры языка высокого уровня

Псевдокод

```
sp := bp
bp := Pop()
```

Описание

Выполняет действия, обратные действию инструкции `enter`. Запись в указатель вершины стека (`sp`) значения регистра `bp` (указателя кадра стека) освобождает память для локальных данных, распределенную в стеке при выполнении `enter`. В результате, `sp` указывает на копию `bp` в стеке, поэтому при последующем считывании вершины стека в регистр `bp` восстанавливается кадр стека вызывающей процедуры.

Примечание

После выполнения команды `leave` управление в вызвавшую процедуру возвращается командой `ret/retf`. Аргументы процедуры, переданные через стек, удаляются либо за счет выполнения `ret/retf` в форме с непосредственным значением, либо в вызывающей процедуре за счет увеличения `sp`.

LDS/LES — Load Far Pointer

Загрузка дальнего указателя

lds r16,m32	Загрузка пары регистров ds и r16 значениями из памяти
les r16,m32	Загрузка пары регистров es и r16 значениями из памяти

Псевдокод

```
CASE инструкция
| lds
    ds := word [m32 + 2]
| les
    es := word [m32 + 2]
END
r16 := word [m32]
```

Описание

Эти команды копируют значение двойного слова из памяти в заданный регистр общего назначения и в один из регистров сегмента *данных*. Значение

двойного слова в памяти представляет дальний указатель в формате сегмент:смещение, со смещением в младшем слове и номером сегмента в старшем.

Пример

```
farptr dd    0b800:0
        . . .
        les    di, farptr    ; es := 0b800, di := 0
```

LODS/LODSB/LODSW — Load String Operand

Загрузка элемента строки

lods m8 lodsb	Загрузка байта из ds:[si] в al
lods m16 lodsw	Загрузка слова из ds:[si] в ax
*lods/lodsw imm8	Задание imm8 экземпляров машинной команды lodsb/lodsw

Псевдокод

```
IF байтовая операция
    al := ds:[si]
    IF d = 0
        IncDec := 1
    ELSE
        IncDec := -1
    END
ELSE
    ax := ds:[si]
    IF d = 0
        IncDec := 2
    ELSE
        IncDec := -2
    END
END
si := si + IncDec
```

Описание

Загружает байт или слово из памяти по указателю ds:si, после чего корректирует значение в si. Если флаг направления d сброшен (была выполнена

инструкция `cld`), то `si` увеличивается; иначе уменьшается. Шаг изменения `si` равен 1 при загрузке байта, или 2 — при загрузке слова.

Форма `lods` не имеет машинного аналога. Встроенная макрокоманда `lods` задает машинную инструкцию `lodsb` или `lodsw`, исходя из типа "операнда". Значение "операнда" не определяет установку указателя `ds:si` — "операнд" используется *только* для определения размерности операции.

Использование этой инструкции с префиксом повторения допустимо, но не имеет смысла. Аналогично, не имеет практического применения форма `lodsb/lodsw imm8`, введенная в ассемблере `i86` для создания `imm8` экземпляров машинной команды для обработки строк.

LOOP/LOOP<x> — Loop Control with CX Counter

Управление циклом со счетчиком `cx`

loop dst	DEC cx; переход, если cx <> 0
loope/z dst	DEC cx; переход, если cx <> 0 и z = 1
loopne/nz dst	DEC cx; переход, если cx <> 0 и z = 0

Псевдокод

```
cx := cx — 1
BranchCond := (cx <> 0)
CASE инструкция
| loope, loopz
    BranchCond := (z = 1) AND BranchCond
| loopne, loopnz
    BranchCond := (z = 0) AND BranchCond
END
IF BranchCond
    IP := dst
END
```

Описание

Эти инструкции уменьшают на 1 значение регистра-счетчика `cx`, не изменяя флагов. Затем проверяются условия перехода (т. е. условие продолжения при организации цикла). Условие `cx <> 0` — общее для всех инструкций; команды `loope/z`, `loopne/nz` задают дополнительное условие по значению флага `z`. Дистанция перехода в инструкциях цикла ограничена 127 (короткий переход), как и для всех условных переходов в `i8086/286`.

MOV — Move Data

Копирование данных

Псевдокод

`dst := src`

Описание

- Копирует второй операнд в первый. (Не изменяет значений флагов.)
- В *a86* эта инструкция реализована в виде встроенной макрокоманды, которая позволяет запрограммировать:
- ☐ размножение данных источника в нескольких приемниках;
 - ☐ передачу данных между сегментными регистрами, а также запись в сегментный регистр непосредственного значения;
 - ☐ передачу слов между ячейками в памяти.

Примеры макроподстановок для перечисленных вариантов команды `mov` показаны в табл. П6.5.

Таблица П6.5. Примеры действия макрокоманды `mov` в *a86*

Макровывоз	Макроподстановка
<code>mov w [area], bx, 1</code>	<code>mov bx, 1</code> <code>mov w[area], bx</code>
<code>mov ds, cs</code>	<code>push ax</code> <code>mov ax, cs</code> <code>mov ds, ax</code> <code>pop ax</code>
<code>mov w[0], w[2]</code>	<code>push w[2]</code> <code>pop w[0]</code>

MOVS/MOVSБ/MOVSW — Move Data from String to String

Копирование данных между строками

<code>movs m16,m16</code> <code>movsb</code>	Копировать байт из ячейки по адресу <code>ds:[si]</code> в ячейку по адресу <code>es:[di]</code>
---	--

<code>movs m16,m16</code> <code>movsw</code>	Копировать слово по адресу <code>ds:[si]</code> в память по адресу <code>es:[di]</code>
<code>*movsb/movsw imm8</code>	Генерирование <code>imm8</code> экземпляров машинной инструкции <code>movsb/movsw</code>

Псевдокод

```
IF байтовая операция
  ds:[si] := es:[di] (* передача байтов *)
  IF d = 0
    IncDec := 1
  ELSE
    IncDec := -1
  END
ELSE
  ds:[si] := es:[di] (* передача слов *)
  IF d = 0
    IncDec := 2
  ELSE
    IncDec := -2
  END
END
si := si + IncDec
di := di + IncDec
```

Описание

Копирует в памяти байт или слово; адрес источника задан парой `es:di`, адрес приемника — парой `ds:si`. После чего значения регистров `di` и `si` изменяются — на 1, если передан байт, или на 2, при передаче слова. Знак изменения определяется установкой флага `d`: если `d = 0`, то приращение положительное, а иначе — отрицательное.

Безразмерный вариант `movs` обозначает машинную инструкцию `movsb` или `movsw`, в соответствии с типом "операндов" (у обоих тип должен быть одинаковым — или `word`, или `byte`). Значения операндов не имеют никакого отношения к значениям регистров-указателей `di` и `si`. Тем более не имеет значения порядок задания "операндов".

Примечание

Вариант `movs` был введен в первом ассемблере Intel/IBM для гибкой настройки размерности операции. В результате получена форма инструкции, в которой значения "операндов" и их порядок не отражают операндов машинной команды, что является дополнительным источником ошибок.

Использование инструкций `movsb/movsw` с префиксом `rep` позволяет копировать одной командой массивы байтов/слов. Счетчик повторений предварительно записывается в регистр `cx`.

Форма `movsb/movsw imm8` введена в a86 для задания `imm8` экземпляров инструкции. Ее следует применять вместо `rep movsb/movsw`, когда число повторов невелико.

Флаги после выполнения

Не изменяются, как и для всех команд копирования.

Пример

```
lea    si, src
lea    di, dst
mov     cx, 12
rep movsw          ; передача 12 слов из src в dst
lea     di, src
movsw    2          ; копирование слов 4 и 5 в начало src
. . .
src dw    1, 2, 10 dup 3, 4, 5
dst dw    12 dup ?
```

MUL — Unsigned Multiplication of AL or AX

Беззнаковое умножение

mul r/m8	ax := al * (байт r/m)
mul r/m16	dx:ax := ax * (слово r/m)

Описание

Выполняет беззнаковое умножение. Если в качестве операнда указан байтовый регистр или байт в памяти, то второй сомножитель всегда находится в `al`, а результат удвоенной размерности помещается в регистр `ax`. Аналогично, если операнд задан словом, то второй операнд берется из `ax`, а результат умножения помещается в `dx:ax`.

Флаги после выполнения

Значение флагов `s`, `z`, `a`, `e` не определено. Флаги знакового (`o`) и беззнакового (`c`) переполнения обнуляются, если результат умножения байтов/слов

помещается в байт/слово. Если же старший байт/слово результата содержит значащие биты произведения байтов/слов, то флаги `o` и `c` устанавливаются в 1.

Пример

```
mov    al, 5
mov    bl, 7
mul    al          ; ax := al * bl = 35, c = o = 0
```

NEG — Two's Complement Negation

Инвертирование числа в дополнительном коде

Псевдокод

```
IF dst = 0
    c := 0
ELSE
    c := 1
END
dst := -dst
```

Описание

Инвертирует число в дополнительном коде из регистра или ячейки памяти. Команда выполняется за счет вычитания значения операнда из нуля; результат записывается на место операнда.

Флаги после выполнения

Флаг `c` устанавливается в 1, если операнд не равен 0; иначе флаг `c` := 0. Флаги `o`, `s`, `z`, `e` устанавливаются по общим правилам.

NOP — No Operation

Пустая операция

Описание

Не выполняет никаких действий. Мнемоника `nop` — синоним команды `xchg ax, ax`.

NOT — One's Complement Negation

Инверсия разрядов операнда

Псевдокод

$r/m := \text{NOT } (r/m)$

Описание

Выполняет логическое инвертирование операнда; биты со значением 1 обнуляются, и наоборот.

Флаги после выполнения

Не изменяются! Эта команда — исключение; остальные арифметико-логические операции на флаги воздействуют.

OR — Logical OR

Логическое "ИЛИ"

Псевдокод

$\text{dst} := \text{dst OR src}$

$\text{c} := 0$

$\text{o} := 0$

Описание

Выполняет операцию логического сложения над парами одноименных разрядов *dst* и *src*, записывая результат в *dst*.

Флаги после выполнения

Флаги *c* и *o* сброшены; *e*, *s* и *z* устанавливаются по общим правилам. Флаг *a* сбрасывается — это проверено на i8086/286/386/486/586, хотя, согласно документации Intel, значение *a* не определено.

Пример

```
mov    dl, 8                ; dl = 001000xb
or     dl, 110000xb         ; dl = 111000xb = '8'
```

OUT — Output to Port

Вывод в порт

out imm8, al	Вывод значения регистра al в порт по адресу imm8
out imm8, ax	Вывод значения регистра ax в пару портов с адреса imm8
out dx, al	Вывод значения регистра al в порт по адресу, заданному dx
out dx, ax	Вывод значения регистра ax в пару портов с адреса, заданного dx

Описание

Передает байт или слово данных в порт или в пару портов, что зависит от размерности источника (al или ax). Для доступа в пределах всего доступного адресного пространства ввода/вывода (64k) адрес задается в dx, команда программируется в варианте out dx, al/ax. Для доступа в диапазоне адресов 0—0ff имеется дополнительная форма команды с заданием непосредственного значения адреса.

Примечание

При доступе к паре портов адрес должен быть четным.

Пример

```
mov    dx, 03fb
mov    al, bit 7
out    dx, al      ; вывод числа 080 в регистр управления COM1
mov    dx, 03f8
in     ax, dx      ; ввод в ax 16-разрядного регистра
                        ; делителя частоты COM1
. . .
in     al, 041     ; чтение в al регистра счетчика первого
                        ; канала таймера
```

OUTS/OUTSB/OUTSW — Output String to Port

Вывод из строки в порт

outs dx, r/m8 outsb	Вывод значения байта из ds:[si] в порт по адресу dx
outs dx, r/m16 outsw	Вывод значения слова из ds:[si] в пару портов по адресам dx и dx+1

Псевдокод

```
IF байтовая операция
  (порт по адресу dx) := ds:[si]
  IF d = 0
    IncDec := 1
  ELSE
    IncDec := -1
  END
ELSE
  (пара портов с адреса dx) := ds:[si]
  IF d = 0
    IncDec := 2
  ELSE
    IncDec := -2
  END
END
si := si + IncDec
```

Описание

Передаёт данные в порт, адрес которого задан содержимым регистра `dx`, из системной памяти с адреса `es:di`. В отличие от команды `out`, в `outs` непосредственное задание номера порта недопустимо.

Примечание

При доступе к паре портов адрес в `dx` должен быть четным.

После передачи данных указатель `si` автоматически корректируется. Его значение увеличивается, если флаг направления сброшен (была выполнена команда `cld`), а иначе уменьшается. Шаг изменения `si` при вводе байта равен 1, а при вводе слова — 2.

Инструкции `outs dx, r/m8` соответствует машинная инструкция `outsb`, а `outs dx, r/m16` — инструкция `outsw`. Соответствие устанавливается ассемблером по типу первого операнда.

Примечание

Первый операнд определяет только *размерность* операции. Он может быть задан любым байтом или словом, например, `al`, или `ch`, или `w[0103]`. Указанный регистр или ячейка памяти не имеет отношения к адресации строки в машинной команде — чтение в любом случае выполняется по адресу `ds:si`.

Инструкция `outsb/outsw` используется также с префиксом повторения `rep`, для повторного вывода в один и тот же порт. Предварительно в `cx` записывается счетчик повторений.

POP — Pop a Word from the Stack

Извлечение слова из стека

Псевдокод

```
dst := ss:[sp]  (* копирование слова *)
sp  := sp + 2
```

Описание

Замещает прежнее значение слова в памяти или регистра значением с вершины стека, которая адресуется парой регистров `ss:sp`. После чего `sp` увеличивается на 2 (вершина стека сдвигается вверх, тем самым, освобождая одну запись стека).

Инструкция `pop cs` в ассемблере недопустима. Восстановление `cs` из стека возможно только вместе с `ip`, что программируется инструкцией `retf`.

Инструкция `pop ss` запрещает внешние прерывания (по входам `INTR` и `NMI`) до завершения следующей инструкции. Это дает возможность переместить стек парой команд `pop ss` и `pop sp`, без риска пропустить прерывание в момент, когда значение `ss` изменено, а `sp` — еще нет.

В `а86` предусмотрена форма задания `pop` со списком приемников, как показано в примере.

Пример

```
pop    ds, dx, ax    ; pop    ds
                        ; pop    dx
                        ; pop    ax
```

POPA — Pop all General Registers

Извлечение из стека содержимого всех регистров общего назначения

Псевдокод

```
di := Pop()
si := Pop()
bp := Pop()
Pop ()      (* пропуск значения, соответствующего SP *)
bx := Pop()
dx := Pop()
cx := Pop()
ax := Pop()
```

Описание

Инструкция `popa` обычно выполняется после предшествующей парной команды `pusha`. Восстанавливает из стека сохраненные значения 16-разрядных регистров общего назначения; значение, соответствующее `sp`, пропускается.

POPF — Pop Stack into FLAGS

Восстановление регистра флагов из стека

Псевдокод

```
flags := ss:[sp] (* копирование слова *)  
sp := sp + 2
```

Описание

Считывает слово с вершины стека и записывает его значение в регистр флагов. Вершина стека (`sp`) перемещается вверх на два байта.

PUSH — Push Operand onto the Stack

Включение слова в стек

Псевдокод

```
sp := sp - 2  
ss:[sp] := src (* копируется слово *)
```

Описание

Уменьшает указатель стека `sp` на 2, затем записывает операнд в новую вершину стека.

Начиная с i80286, инструкция `push sp` записывает в стек *исходное* значение `sp` — до его уменьшения. Процессор i8086 по этой команде записывает в стек значение `sp` после уменьшения.

Начиная с процессора i80286, инструкция `push` допускает задание операнда *непосредственным* значением.

В a86 предусмотрена форма задания `push` со *списком источников*, как показано в примере.

Пример

```
push ax, 1, ds ; push ax
                ; push 1
                ; push ds
```

PUSHA — Push all General Registers

Включение в стек содержимого всех регистров общего назначения

Псевдокод

```
temp := sp
Push (ax)
Push (cx)
Push (dx)
Push (bx)
Push (Temp)      (* запись исходного значения sp *)
Push (bp)
Push (si)
Push (di)
```

Описание

Записывает в стек восемь регистров общего назначения, уменьшая `sp` на 16. Значение самого `sp` сохраняется таким, каким оно было в начале выполнения инструкции.

Примечание

Команда `pusha` не включает в стек значений сегментных регистров; для этого следует использовать отдельные команды `push`.

PUSHF — Push Flags Register onto the Stack

Включение регистра флагов в стек

Псевдокод

```
sp := sp - 2
ss:[sp] := flags (* копируется слово *)
```

Описание

Уменьшает указатель стека на 2, а затем записывает в новую вершину стека значение регистра флагов.

В дальнейшем копия флагов в стеке используется для восстановления исходного значения флагов. Между сохранением и восстановлением флагов их копия в стеке доступна для корректировки. Позиции флагов в регистре флагов отражены в табл. П6.6. Знаком "?" отмечены неопределенные значения.

Таблица П6.6. Содержимое регистра флагов

Номер бита	11	10	9	8	7	6	5	4	3	2	1	0
Флаг	o	d	i	t	s	z	?	a	?	e	?	c

Примечание

Флаги состояния, за исключением флага знакового переполнения *o*, находятся в младшем байте. Флаги управления — в старшем байте. Биты с 12 по 15 используются в защищенном режиме, изменять их значения не следует.

Пример

```
pushf
enter 0
or [bp+2], bit 8 ; установка бита, соответствующего
                  ; флагу разрешения трассировки (t)
leave
popf
```

REP/REP<x> — Repeat Following String Operation

Повтор последующей строковой операции

Псевдокод

```
WHILE cx <> 0
  обслужить прерывание, если есть заявка
  выполнить строковую команду
  cx := cx — 1
  IF строковая команда = (cmpsб/w или scasб/w)
    IF (repe/z и (z = 1)) или (repne/nz и (z = 0))
      EXIT
    END
  END
END
END
```

END

Описание

Команды `rep`, `repe` (`repeat while equal`), `repne` (`repeat while not equal`) — префиксы для заикливания строковых операций. Они задают повтор выполнения строковой команды, пока значение счетчика в регистре `сх` не равно нулю и выполняется дополнительное условие продолжения (`z = 1` при задании `repe`; `z = 0` при задании `repne`).

Префиксы `repz` и `repnz` — синонимы `repe` и `repne`.

Примечание

В отличие от инструкций `loop/loop<x>` префикс повторения проверяет `сх` перед выполнением, а не после. Поэтому при входном значении `сх = 0` строковая команда с префиксом повторения не будет выполнена ни разу. Цикл на основе `loop/loop<x>` был бы выполнен 64k раз.

RET — Return from Procedure

Возврат из процедуры

<code>ret</code>	Ближний возврат
<code>retf</code>	Дальний возврат
<code>ret imm16</code>	Ближний возврат с удалением <code>imm16</code> байт параметров
<code>retf imm16</code>	Дальний возврат с удалением <code>imm16</code> байт параметров

Псевдокод

```
ip := Pop()  
IF дальний возврат  
    cs := Pop()  
END  
IF задан imm16  
    sp := sp + imm16  
END
```

Описание

Передаёт управление по адресу возврата, сохранённому в стеке при выполнении инструкции вызова. Необязательный числовой параметр задаёт число байтов стека, которые следует удалить после передачи управления по адресу возврата. Обычно, это значения параметров, записанные в стек перед вызовом процедуры.

При ближайшем возврате из стека восстанавливается только значение смещения (записывается в `ip`). При межсегментном (дальнем) возврате из стека извлекается также номер сегмента, который записывается в `cs`.

RCL/RCR/ROL/ROR — Rotate

Циклические сдвиги, или вращения

<code>rcl r/m, imm8/cl</code>	Вращение через перенос значения <code>c+(r/m)</code> влево
<code>rcr r/m, imm8/cl</code>	Вращение через перенос значения <code>c+(r/m)</code> вправо
<code>rol r/m, imm8/cl</code>	Вращение <code>r/m</code> влево
<code>rор r/m, imm8/cl</code>	Вращение <code>r/m</code> вправо

Описание

Все инструкции вращения сдвигают биты первого операнда. Вращение влево (в мнемонике задано буквой **L**) сдвигает биты в направлении от младших к старшим, по кольцу: старший бит (бит 7 при вращении байт, или бит 15 при вращении слов) передается в младший бит. Значение бита, передаваемого по кольцу, копируется во флаг `c`. Вращение вправо (в мнемонике задано буквой **R**) — аналогично, но в противоположном направлении (от старших битов к младшим).

Инструкции вращения через перенос `rcl` и `rcr` включают флаг `c` в последовательность бит для вращения. При вращении через перенос влево (`rcl`) в бит 0 помещается не значение старшего бита, а значение флага `c`. Флаг `c` затем принимает исходное значение старшего бита. Инструкция `rcr` выполняется аналогично, но вращение направлено от старших бит к младшим.

Вращения повторяются столько раз, сколько задано битами 0—4 второго операнда (непосредственным значением или содержимым регистра `cl`). Поскольку используются только 5 бит, число вращений ограничено 31.

Флаги после выполнения

Флаг `c` содержит последний бит, выдвинутый за пределы разрядной сетки операнда. Флаг `o` определен по общим правилам, но только для числа вращений 1.

SAHF — Store AH into Flags

Запись ah в регистр флагов

Псевдокод

```
s:z:o:a:o:e:o:c := ah
```

Описание

Загружает младший байт регистра флагов значением из регистра ah. Расположение флагов в младшем байте регистра флагов показано в табл. П6.4 (см. LAHF).

SAL/SAR/SHL/SHR — Shift Instructions

Сдвиги, арифметические и логические

sal r/m, imm8/cl shl r/m, imm8/cl	Арифметический и логический сдвиг влево (умножение r/m на 2, imm8/cl раз)
sar r/m, imm8/cl	Арифметический сдвиг вправо (знаковое деление на 2, imm8/cl раз)
shr r/m, imm8/cl	Логический сдвиг вправо (беззнаковое деление r/m на 2, imm8/cl раз)

Описание

Команда арифметического сдвига влево sal (и ее синоним shl — логический сдвиг влево) смещает значения битов операнда от младших к старшим. Старший бит выталкивается за пределы разрядной сетки, но его значение сохраняется во флаге c; в младший бит записывается ноль. Каждый сдвиг на один разряд приводит к удвоению значения операнда, если при этом не потеряны значащие разряды. Потеря значащих разрядов отражается установкой в 1 флага o (при знаковом переполнении) или флага c (при беззнаковом переполнении).

Команды арифметического и логического сдвига вправо (sar и shr, соответственно) сдвигают биты операнда от старших к младшим. Бит 0 сдвигается во флаг c. При каждом сдвиге на один разряд значение операнда уменьшается в два раза. Команда shr выполняет беззнаковое деление, при каждом сдвиге обнуляя старший разряд. Команда sar выполняет знаковое деление, за счет сохранения первоначального значения в старшем (знаковом) бите.

Число сдвигов на один разряд задано битами 0—4 второго операнда (непосредственным значением или содержимым регистра cl). Число сдвигов, в результате, ограничено 31.

Флаги после выполнения

Флаги *c*, *z*, *e*, *s* устанавливаются по общим правилам. Значение флага *o* определено, если число сдвигов равно 1. При сдвигах влево *o* = 0, если старший бит результата равен значению *c* после выполнения команды (т. е. если значения двух старших битов были одинаковыми). После выполнения *sar* флаг *o* = 0, а после *shr* он равен старшему биту исходного значения.

SBB — Subtraction with Borrow

Вычитание *c* заемом

Псевдокод

`dst := dst - src - c`

Описание

Вычисляет разность между первым и вторым операндами, вычитая дополнительно значение флага *c*. Результат помещает на место первого операнда. Команда *sbb* применяется в операциях вычитания с данными повышенной размерности, образованных массивами байтов или слов. Вычитание выполняется поэлементно, при этом необходимо учитывать распространение заема.

Флаги после выполнения

Флаги *o*, *s*, *z*, *a*, *c*, *e* устанавливаются по общим правилам.

Пример

```
mov    ax, 0fffe
mov    dx, 1      ; dx:ax = 01fffe (131070)
sub    ax, 0ffff  ; dx:ax := dx:ax -
sbb    dx, 1      ; - 01ffff (131071)
                    ; dx:ax = 0fffffffff (-1)
```

SCAS/SCASB/SCASW — Scan String Data

Сканирование строки

scas m8 scasb	Сравнить байты <i>al</i> и <i>es:[di]</i>
------------------	---

scas m16 scasw	Сравнить слова ax и es:[di]
*scasw/scasb imm8	Сгенерировать imm8 экземпляров команды scasw/scasb

Псевдокод

IF байтовая инструкция

al ← es:[di]

IF d = 0

IndDec := 1

ELSE

IncDec := -1

END

ELSE

ax ← es:[di]

IF d = 0

IndDec := 2

ELSE

IncDec := -2

END

END

di := di + IncDec

Описание

Сравнивает значение регистра al/ax с байтом/словом по адресу es:di. Сравнение выполняется за счет вычисления разности, с отбрасыванием результата; отражается во флагах.

Если флаг направления сброшен (была выполнена инструкция cld), то значение di после выполнения команды увеличивается; а если флаг d установлен, то di уменьшается. Приращение равно 1 при байтовом сравнении, а при сравнении слов равно 2.

Форма scas не имеет машинного аналога; ассемблер преобразует ее в машинную команду scasb или scasw, исходя из типа "операнда". Значение "операнда" при этом никак не используется.

Инструкция scasb/scasw с префиксом повторения repe или repne дает возможность организовать цикл сравнений, пока значение регистра-аккумулятора не разойдется (repe) или не совпадет (repne) с данными массива. Длина массива предварительно записывается в регистр-счетчик cx (см. REP/REPE/REPZ/REPNE/REPZ).

Флаги после выполнения

Флаги состояния устанавливаются по общим правилам.

STC — Set Carry Flag

Установка флага переноса

Псевдокод

```
c := 1
```

Описание

Устанавливает флаг переноса, или беззнакового переполнения в 1.

STD — Set Direction Flag

Установка флага направления

Псевдокод

```
d := 1
```

Описание

Устанавливает флаг направления в 1, в результате чего строковые инструкции уменьшают значения используемых индексных регистров (*si* и/или *di*).

STI — Set Interrupt Flag

Установка флага разрешения прерывания

Псевдокод

```
i := 1
```

Описание

Разрешает внешние прерывания со входа *INTR* процессора.

Примечание

Процессор воспринимает сигнал прерывания не раньше чем в конце выполнения следующей команды. Так, например, если прерывания были запрещены, то пара команд *sti* и *ret* пройдет "без запинки". При выполнении в подобной ситуации пары команд *sti* и *cli* прерывания не успеют открыться. Для кратковременного "прослушивания" задержанных прерываний следует между *sti* и *cli* поставить, например, *nop*.

STOS/STOSB/STOSW — Store String Data

Запись данных в строку

stos m8 stosb	Записать байт из al в память по адресу es:[di]
stos m16 stosw	Записать слово из ax в память по адресу es:[di]
*stosb/stows imm8	Генерирование imm8 экземпляров команды stosb/stows

Псевдокод

```
IF байтовая операция
    es:[di] := al
    IF d = 0
        IncDec := 1
    ELSE
        IncDec := -1
    END
ELSE
    es:[di] := ax
    IF d = 0
        IncDec := 2
    ELSE
        IncDec := -2
    END
END
di := di + IncDec
```

Описание

Записывает содержимое регистра al/ax в памяти по указателю es:di и кор-
ректирует значение di. Если флаг направления d сброшен (была выполнена
инструкция cld), то di увеличивается; а иначе уменьшается. Шаг изменения
di равен 1 (при записи байта) или 2 (при записи слова).

Форма stос не имеет машинного аналога. Встроенная макрокоманда stос
задает машинную инструкцию stосb или stосw, исходя из типа "операнда".
Значение "операнда" не определяет установку указателя es:di — "операнд"
используется *только* для определения размерности операции.

Использование этих команд с префиксом повторения rep позволяет запол-
нить массив слов или байт одинаковыми значениями. В al/ax предвари-
тельно записывается инициализирующее значение, а в cx — счетчик.

Форма `stosb/stosw imm8`, допустимая в ассемблере *a86*, генерирует заданное количество *экземпляров* команды. Применяется для задания небольшого числа повторов операции.

Пример

```
mov    di, ax, 0
cld
mov     es, 0b800
mov     cx, 80*25    ; число знакомест на экране
rep     stows        ; запись ax = 0, начиная с адреса b8000:0
```

SUB — Subtraction

Вычитание

Псевдокод

```
dst := dst - src
```

Описание

Команда вычисляет разность операндов и записывает результат на место первого операнда (dst).

Флаги после выполнения

Флаги *o*, *s*, *z*, *a*, *c*, *e* устанавливаются по общим правилам.

TEST — Logical Compare

Логическое сравнение

Псевдокод

```
dst := LeftSrc AND RightSrc
c := 0
o := 0
```

Описание

Выполняет поразрядную операцию логическое "И" над заданными операндами. Модифицирует флаги по результатам операции, а само логическое произведение отбрасывается.

Эта команда обычно используется для проверки:

- значений отдельных битов;
- значения слова или байта на равенство нулю.

Программирование последней операции показано в примере (вместе с сокращенным вариантом задания команды, допустимом в a86).

Флаги после выполнения

Флаги `с` и `о` сброшены; `е`, `с` и `z` устанавливаются по общим правилам. Флаг `а` сбрасывается — это проверено на i8086/286/386/486/586, хотя, согласно документации Intel, значение `а` не определено.

Пример

```
test    al          ; test    al, al
jz      al_empty
test    w[0]        ; test    w[0], 0ffff
jz      mem_empty
```

WAIT — Wait until BUSY# Pin is Inactive

Ожидание снятия сигнала BUSY#

Описание

Останавливает выполнение инструкций центрального процессора до тех пор, пока сигнал на входе BUSY# не будет снят. К входу BUSY# подключен соответствующий сигнал от сопроцессора i80x87, таким образом команда `wait` (синоним `fwait`) останавливает *центральный процессор* до завершения текущей операции в *сопроцессоре*.

XCHG — Exchange Register/Memory with Register

Обмен между регистрами или между памятью и регистром

Псевдокод

```
temp := dst
dst := src
src := temp
```

Описание

Выполняет обмен содержимым двух операндов, заменяя три команды пересылки.

XLAT/XLATB — Table Look-up Translation

Табличное перекодирование

Псевдокод

```
al := ds:[bx + al]
```

Описание

Меняет значение индекса таблицы в регистре `al` на значение элемента таблицы по этому индексу. Входное значение `al` — это беззнаковое смещение в байтах от начала таблицы; сама таблица содержит не более 256-байтных элементов.

Инструкции `xlat m8` и `xlatb` ничем не отличаются; это одна и та же машинная команда. Необязательный операнд `xlat` в *а86* игнорируется.

XOR — Logical Exclusive OR

Логическое "исключающее ИЛИ"

Псевдокод

```
dst := dst XOR src  
c := 0  
o := 0
```

Описание

Выполняет операцию "исключающее ИЛИ" над парами одноименных разрядов `dst` и `src`, записывая результат в `dst`.

Флаги после выполнения

Флаги `c` и `o` сброшены; `e`, `s` и `z` устанавливаются по общим правилам. Флаг `a` сбрасывается, что проверено на i8086/286/386/486/586, хотя, согласно документации Intel, значение `a` не определено.

Пример

```
xor    bx, bx           ; bx = 0000xb  
xor    bx, 0f           ; bx = 1111xb  
xor    bx, bit 1        ; bx = 1101xb
```

ПРИЛОЖЕНИЕ 7

Совместимость a86 с традиционными ассемблерами

Что касается замужества с безмозглым и беспомощным созданием, то таковая жизнь не является отталкивающей для девушки с возвышенными мыслями.

Роберт Шекли. "Хожение Джозниса"

Согласно документации, ассемблер a86 на 99% совместим с `masm`. Это, очевидно, вовсе не относится к уникальному нетрадиционному языку макрокоманд a86 — конвертирование программ с макрокомандами и директивами условной трансляции между a86 и `tasm/masm/wasm` требует корректировки. (Соответствие между директивами условной трансляции `masm` и a86 показано на стр. 85 документации — в файле `a86_manu.txt`; пример макроопределения на языке `masm` см. там же, на стр. 84, п. 6.)

Примечание

Поскольку язык макрокоманд a86 гораздо более простой и гибкий, переписать на нем макрокоманды `tasm/masm/wasm` несложно. Напротив, перевод макрокоманд a86 на `tasm/masm/wasm` становится проблемой, если использованы циклы, в особенности q- и c-циклы. Почему-то в учебниках по традиционным ассемблерам не приводят примеров макрокоманд с циклами (циклы вне макрокоманд — пожалуйста; но вы попытайтесь внести их *внутри* макроопределения).

За исключением макрокоманд и средств условной трансляции, a86 распознает большую часть конструкций традиционных ассемблеров. Директивы для управления компоновкой (`segment`, `group`, `end` и т. п.) воспринимаются только в `obj`-режиме, а в `com`-режиме игнорируются.

При трансляции программ `tasm/masm/wasm` на a86 следует задать ключ `+D`, чтобы разбор *чисел* выполнялся в стиле `masm`. Ведущий ноль в обозначении числа в этом случае не является признаком шестнадцатеричного формата

(см. также опции ключа **G** для задания уровня совместимости с `masm` — в *приложении 5*).

Примечание

Если не задать ключ `+D`, то значения в двоичном формате с ведущими нулями будут вычислены неправильно (например, `00000001b` в а86 означает не 1, а `01bh`, т. е. 27).

В а86 допускаются *объявления* структур, но применять имя структуры в качестве *типа* при задании данных в памяти нельзя. В листинге П7.1 приведен пример объявления данных типа структуры в традиционном ассемблере.

Листинг П7.1. Объявление и инициализация структур в традиционном ассемблере

```
payrec  struc
        pname  db  'no name given'          ; (1)
        pkey   dw  ?
ends
```

```
payrec  3 dup (?)                ; (2)
payrec  <'Eric', 1811>           ; (3)
```

При трансляции операторов (2) и (3) а86 выдаст сообщение об ошибке "Unknown mnemonic". Задание в шаблоне структуры значения поля по умолчанию (1) не считается ошибкой, но применить это значение в операторе (2) не удастся, поскольку а86 поддерживает (по-видимому, принципиально) данные только *машинного* формата, объявляемые директивами `db/dw/dd/dq/dt`.

Операторы (2) и (3) без инициализации могут быть представлены в а86 следующим образом:

```
db  ((type payrec) * 3) dup ?
```

Пример определения с инициализацией показан в листинге П7.2.

Листинг П7.2. Инициализация структур макрокомандами (см. `struc.8`)

```
payrec  struc
        pname  db  'no name given '        ; 14 bytes
        pkey   dw  ?                        ; 2 bytes
ends
```

```
init_payrec  macro
m1:
##if #s1 eq 0
        db  'no name given '
```

```

##else
    db    #1
##endif
    org   m1 + offset pkey    ; (1)
##if #s2 eq 0
    dw    ?
##else
    dw    #2
##endif
#em
    init_payrec
    init_payrec
    init_payrec
    init_payrec    'Eric', 1811                ; (2)
    init_payrec    'Eric Isaacson, America', 1811 ; (3)

```

Число байтов, резервируемых для элемента `pname`, должно быть одинаковым (14 байт), независимо от длины параметра макрокоманды. Поэтому счетчик адресов при переходе к полю `pkey` устанавливается принудительно (1). При трансляции вызова (2) значение счетчика адресов директивой (1) *увеличивается*, поскольку задано всего четыре байта. При трансляции (3) директива (1) *уменьшает* значение счетчика адресов, в результате чего избыток данных (' America') отбрасывается.

В а86 не предусмотрена редко применяемая и сложная в реализации директива `record`, а также связанные с ней операторы `width` и `mask`. Помнить об этом не обязательно — а86 при трансляции `record` выдаст сообщение об ошибке "Unknown mnemonic".

При трансляции на а86 программ, разработанных для `tasm/masm/wasm`, следует обратить внимание на директиву `assume`, в особенности, в TSR-программах. Ассемблер а86 игнорирует `assume`, в результате чего доступ к данным может оказаться настроенным иначе, чем было задумано при разработке программы на `tasm/masm/wasm`.

Директива `assume` в традиционных ассемблерах задает автоматическую подстановку *префикса переназначения сегмента* при прямом доступе к данным *по имени*. (Также отслеживаются межсегментные передачи управления при задании адреса перехода именем метки.) Эта директива обязательная в традиционных ассемблерах, за исключением `wasm`. Пример — программа для `tasm` — приведен в листинге П7.3.

Листинг П7.3. Пример программы для `tasm` с директивой `assume` (см. `assume.asm`)

```

_data    segment byte public 'data'
d_1      dw        40 dup (?)
_data    ends

```

```

_extra segment byte public 'data'
e_1      db      'Need you illusion, though too much is so real?'
_extra   ends

_text    segment byte public 'code'
        assume   ds:_data, es:_extra, cs:_text

start:

        mov      ax, _data
        mov      ds, ax
        mov      ax, _extra
        mov      es, ax

        mov      al, e_1           ; mov    al, es:[e_1]
        mov      ah, byte ptr start ; mov    ah, cs:[start]
        mov      d_1, ax           ; mov    d_1, ax

        lea      si, e_1 + 2
        mov      ax, [si]          ; (?!!) mov  ax, [si]
        mov      d_1 + 2, ax

        mov      ax, 04c00h
        int      021h

ends

        end      start

```

После трансляции (tasm assume.asm) и компоновки (tlink assume.obj /m) запустите d86 с полученным файлом assume.exe. В окнах отображения памяти после выполнения первых четырех команд (настройка ds и es) будет показано содержимое массивов d_1 и e_1. В третьем окне — один байт по адресу start.

Следующая за настройками команда пересылки байта из e_1 в al при трансляции дополнена префиксом es, т. к. переменная e_1 определена в сегменте _extra, который, согласно assume, доступен через es. Команда записи в ah тоже дополнена префиксом — cs, поскольку метка start определена в сегменте _text, который доступен (предположительно) через cs.

При трансляции этой программы ассемблером а86 префиксы не будут подставлены, т. к. assume игнорируется. Эта директива принципиально не поддерживается, поскольку по опыту автора а86 она порождает больше проблем, чем их решает. Поясним, к чему приводит использование этой директивы.

Во-первых, текущая настройка сегментных регистров обязательно должна *соответствовать* предположениям, объявленным в assume. Соответствие устанавливается вручную, поэтому в программах с большим числом переключений сегментов вероятно расхождение между текущей и предполагае-

мой настройками сегментов. Во-вторых, действие `assume` распространяется только на операнды, указанные *именем*. При косвенном доступе через *указатели* автоматическая подстановка префикса, естественно, не выполняется.

Например, в программе из листинга П7.3 (в расчете на автоматическую подстановку префиксов) запрограммирован доступ ко второму слову из массива `e_1` через регистр-указатель `si`. При трансляции команды `mov ax, [si]` префикс `es` не подставляется, и чтение выполняется из главного сегмента данных — из массива `d_1`. Задуманная передача пары байт из массива `e_1` в `d_1` не выполнена.

Примечание

WASM не требует задания `ASSUME`. В TASM задание `ASSUME` обязательно, но от его навязчивых "услуг" можно попытаться отказаться: `ASSUME NOTHING` (см. файл `know.asm`). К слову, атрибут `NOTHING` допускается задавать также на месте "`seg_name`", например, `ASSUME CS:_DATA, ds:NOTHING, es:NOTHING` (объявление, встречающееся в TSR-программах).

При переносе программ `tasm/masm/wasm` на язык `a86` следует вручную поставить префиксы в тех инструкциях, где традиционный ассемблер подставляет их автоматически. Не забудьте также строковые команды с псевдооперандами типа `byte` или `word` (см. приложение 6), они были введены отчасти для расширения сферы влияния `assume`.

Если разное отношение к `assume` в `tasm/masm/wasm` и в `a86` может повлечь за собой ошибки, обнаруживаемые только при отладке, то прочие различия в "характерах" проявляются уже при трансляции. Подробнее особенности языка `a86` и совместимость его элементов с языком `masm` рассмотрены в главах 5, 9 и 10 авторской документации.

ПРИЛОЖЕНИЕ 8

Прерывания от i80x87

Сопроцессор i80x87 при выполнении float-инструкций устанавливает признаки ошибок, или *особых ситуаций* в регистре состояния (sw). Для разрешения прерывания от сопроцессора при возникновении особых ситуаций в управляющем слове (cw) следует обнулить одноименный бит маскирования прерываний.

Синхронизация процессора и сопроцессора

Незамаскированная особая ситуация в FPU приводит к генерированию запроса сразу после завершения команды, которая вызвала установку признака особой ситуации. Команда WAIT центрального процессора дает возможность дождаться *завершения* текущей операции i80x87. Команда WAIT гарантирует, что прерывание по незамаскированной особой ситуации, если оно должно произойти, будет обработано до продолжения программы в центральном процессоре. Вставка WAIT может понадобиться в том случае, когда команды FPU перемежаются командами центрального процессора. Если команды FPU следуют одна за другой, вставка WAIT не требуется.

Примечание

Поток FPU-инструкций синхронизируется автоматически — процессор, прежде чем переслать следующую инструкцию в сопроцессор, без дополнительных указаний сам ожидает завершения предшествующей FPU-инструкции. Вставка команд WAIT между инструкциями FPU нужна была только в паре 8086/8087 — на 8087 было возложено управление доступом к системной памяти, что приводило к фатальным конфликтам. В дальнейших модификациях центрального процессора и сопроцессора передача данных между FPU и системной памятью была передана центральному процессору. Получив и декодировав инструкцию, сопроцессор выставляет сигнал PEREQ (Processor Extention Operand REQuest), и процессор передает данные в шинных циклах ввода/вывода.

При обмене данными через системную память синхронизация FPU с центральным процессором выполняется аппаратно, без команд WAIT. Например, согласованное выполнение следующей пары команд не нуждается во вставке между ними WAIT:

```
fist    mem
mov     ax, mem
```

Процессор откладывает чтение ячейки `mem` до тех пор, пока в нее не будет записан результат инструкции FPU. После выполнения этой пары команд в `ax` находится результат записи из FPU в `mem`. Если бы более быстрый центральный процессор не стал дожидаться завершения `fist`, то в `ax` было бы записано исходное значение `mem`.

Обработка прерываний

При обработке прерывания следует:

- ❑ прочитать состояние FPU на момент ошибки;
- ❑ сбросить признаки особых ситуаций в слове состояния `sw`;
- ❑ определить в прочитанном массиве состояния причину ошибки по значениям полей, соответствующих `sw` и `sw`;
- ❑ ликвидировать последствия ошибки в программе.

Простейший пример обработки прерываний по особым ситуациям i80x87 приведен в листинге П8.1.

Листинг П8.1. Обработка прерывания от i80x87 (см. `fpu_int.8`)

```
include fpu.inc

        jmp     start

old_v2  dd      ?
cw      dw      ?
cw0     dw      ?
cw1     dw      ?
err_cnt dw      0                ; interrupt counter

start:
        mov     ax, 035h by 2    ; save vect 2
        int     021h
        mov     w old_v2, bx
        mov     w old_v2 + 2, es
```

```

lea    dx, trap2                ; set new vect
mov     ax, 025 by 2
int     021

finit
fstcw   cw0                     ; get ctrl word
mov     cw, ax, cw0
and     cw, not 111111xb       ; unmask all exceptions
fldcw   cw                      ; set new ctrl word

; -
fldpi   ; pi
fmul    0                      ; pi*pi
fadd    0                      ; (pi^2)*2
fldl    ; 1                    (pi^2)*2
fadd    ; 1+(pi^2)*2
fsqrt   ; sqrt(1+(pi^2)*2)
fstp    0                      ; -
fstp    0                      ; -
fstp    0                      ; -
fist    d end                  ; -
fwait

mov     ax, 025 by 2           ; restore old int 2
mov     dx, w old_v2
mov     ds, w old_v2 + 2
int     021
mov     ds, cs
int     020                    ; see err_cnt here

trap2:
cs inc  err_cnt                ; increment counter
fclex                                     ; clear errors
cs fldcw cw                    ; reinit cw
iret

```

При каждом прерывании увеличивается счетчик ошибок `err_cnt`.

В прерывании для сброса причины прерывания годятся только инструкции Non-Wait (без ожидания, т. е. без неявной команды WAIT): `fnstenv`, `fninit`, `fclex`. В `а86` команды `fstenv` и `fclex` транслируются в `fnstenv` и `fclex`, если не установлен режим автоматической вставки WAIT (см. опцию F в приложении 5). Если команда с ожиданием была выполнена до сброса причины прерывания, система виснет.

Подключение сигнала прерывания

В примере из листинга П8.1 обрабатывается прерывание 2, как если бы запрос на прерывание был подключен к входу NMI (Non-Maskable Interrupt). В IBM PC на базе 8086/87 и 80286/287 сигнал прерывания от FPU действительно был подключен к входу NMI (через схему логики NMI, вместе с сигналами сбоя при обращении к памяти и внешним устройствам).

Внешнее прерывание через irq13

В IBM PC на базе 80386/387 сигнал от сопроцессора подключен не к NMI, а к *контроллерам прерываний* через вход irq13 (вход номер 5 ведомого контроллера). В системах следующего поколения сигнал прерывания от i80x87 тоже подключен к irq13, но с возможностью перенастройки на *внутреннее* прерывание 16.

Для того чтобы программы, написанные для 8086/286, выполнялись на 80386, в последовавшей версии BIOS была предусмотрена системная процедура обработки прерывания 75 (от irq13), текст которой показан в листинге П8.2.

Листинг П8.2. Системная процедура обработка прерывания от irq13

```
push    ax, dx
xor     al, al
out     0f0, al          ; Busy off
mov     al, 020
out     0a0, al          ; EOI to master PIC
out     020, al          ; EOI to slave PIC
pop     dx, ax
int     2                ; !
iret
```

Через порты 0f0—0ff выполняется обмен данными между процессором и сопроцессором. (Большая часть этих портов недоступна для команд in/out, порт 0f0 доступен.) После генерирования заявки на прерывание сопроцессор переходит в режим ожидания, до записи нуля в порт 0f0. После записи 0 в порт 0f0 системная процедура обработки прерывания 75 выполняет обычные операции сброса заявок в контроллерах прерываний. Затем она передает управление процедуре обработки прерывания по вектору 2. Тем самым, BIOS *программно переводит* прерывание от входа irq13 контроллера прерываний в прерывание от входа NMI. Если вам понадобится запрограммировать собственную процедуру обработки прерываний по вектору 75, не забудьте включить в нее запись нуля в порт 0f0 (и, само собой, сброс контроллеров прерываний).

Внутреннее прерывание 16

Начиная с 486DX, сопроцессор практически уже не является внешним устройством. Появляется возможность замкнуть запрос на прерывание от сопроцессора внутри процессора. Для сохранения совместимости с программами, разработанными для 80386, в начальном состоянии процессора сигнал запроса выводится на irq13. При установке флага NE (Numeric Exception) в управляющем регистре CR0 запрос переключается с irq13 на линию внутреннего прерывания 16.

Флагу NE соответствует бит номер 5 регистра CR0. Его установка выполняется командами:

```
mov    eax, cr0
or     al, bit 5
mov    cr0, eax
```

Для сброса NE выполняется аналогичная последовательность команд, только вместо or следует использовать and:

```
and    al, not bit 5
```

Команды передачи между eax и cr0 в а86 могут быть заданы при помощи следующих макрокоманд:

```
rd_cr0 macro                ; eax := cr0
    db    0f, 020, 0c0
#em
wr_cr0 macro                ; cr0 := eax
    db    0f, 022, 0c0
#em
```

В IBM PC обработка внутреннего прерывания от FPU осложнена тем, что номер прерывания совпадает с номером программного прерывания для обращения к видеофункциям BIOS.

Вновь разработанная процедура обработки прерывания по вектору 010 должна определять источник прерывания и, в случае вызова функции BIOS, передавать управление системной процедуре. Иначе, в основной программе исключается системный ввод/вывод (в частности, функцией 9 DOS), который прямо или косвенно вызывает видеофункции BIOS.

ПРИЛОЖЕНИЕ 9

Ответы на контрольные вопросы из части I

В этом приложении приведены ответы на контрольные вопросы из *части I*, выборочно. Некоторые ответы приведены с решениями или с комментариями. В подзаголовках ниже указаны разделы с вопросами.

К разделу 1.2

1. `dw 5, 5, 7, 5, 5, 7, 5, 5, 7, 9`
2. `db 'AB1???x???x0'`
3. `dd 6, 2 dup (2 dup (1, 0), 5)`
7. Задача не имеет решения, поскольку значение 256 не может быть представлено байтом.
8. `db 100 dup (5 dup (1, 0), 4 dup 1)`

Слова заданы парами байт; число в первом байте слова имеет вес 1, а число в старшем байте имеет вес 256. Поэтому слово со значением 1 составляется из пары байт со значениями 1 и 0, а не наоборот (байты 0 и 1 задают слово со значением 256).

Заданная последовательность может быть также определена одной директивой `dw`:

```
dw 100 dup (5 dup 1, 2 dup (1 by 1))
```

К разделу 1.4

1. `dw 10k dup '0' by '0'`
или
`db 20k dup '0'`

2. 100000xb
bit 5
3. 21, 15, 3, 15, 10, 8, 041
4. hours equ 365 * 24
dw hours
dd hours * 60, hours * 60 * 60

Типичная ошибка:

```
hours equ 365 * 24
dw hours
mins dd hours * 60
dd mins * 60
```

Ошибка — в последнем операторе: имя `mins` представляет вовсе не значение двойного слова, а его *адрес*. Поэтому `mins * 60` — не число секунд. Если уж вводить дополнительное имя со значением числа секунд, то это следует сделать директивой `equ`:

```
hours equ 365 * 24
dw hours
mins equ hours * 60
dd mins, mins * 60
```

7. mask1 equ bit 3 + bit 7
mask2 equ (1 shl 3) or (1 shl 7)

Скобки не обязательны, поскольку приоритет у операторов `shl` и `bit` выше, чем у `or` или `+`.

8. db not (bit 0 + bit 3)

Ошибка:

```
db (not bit 0) + (not bit 3)
```

что означает

```
db 11111110xb + 11110111xb
```

В результате, был бы определен байт со значением `11110101xb`. Если бы в ошибочном операторе был `or`, то все разряды оказались бы в 1.

К разделу 1.7

1. Типы имен:

- `noodle` — `abs` (поскольку у выражения `'0'` `by 0` тип — `abs`);
- `needle` — `byte` (т. к. имя определено в директиве `db`);
- `a1` — `word`.

2. Ошибки.

- Не поставлена кавычка в конце строки "You're genius.
- Выражение `dest – src` имеет тип `word`, а операция деления к типу `word` неприменима.

Значение `bit 16 + offset src` превышает диапазон слова. Ошибкой это не считается, ассемблер просто отбрасывает старшее слово результата. (Также см. *разд. "Ошибка в операторе bit"* в приложении 10).

3. Данные расположены в памяти, как показано в табл. П9.1.

Таблица П9.1. Расположение данных и их адресация

Адресация относительно <code>byte1</code>	Значения байтов	Адресация относительно <code>word1</code>
<code>byte1</code>	1	<code>word1–2</code>
<code>byte1+1</code>	2	<code>word1–1</code>
<code>byte1+2</code>	040	<code>word1</code>
<code>byte1+2</code>	030	<code>word1+1</code>
<code>byte1+3</code>	060	<code>word1+2</code>
<code>byte1+4</code>	050	<code>word1+3</code>

В табл. П9.2 показаны значения данных в результате выполнения команд (курсивом отмечены байт или байты операнда команды). В крайнем левом столбце — исходные значения.

Таблица П9.2. Изменение данных при выполнении команд

	inc byte1	inc byte1+1	inc byte1+2	inc word1	inc word1+1
1	2	2	2	2	2
2	2	3	3	3	3
040	040	040	<i>041</i>	<i>042</i>	042
030	030	030	030	<i>030</i>	<i>031</i>
060	060	060	060	060	<i>060</i>
050	050	050	050	050	050

4. Слово со значением `–1` задано в виде `0ffff` (дополнительный 16-битный код числа `–1`). Это слово в массиве `word1` — четвертое, если считать от нуля. Смещение составляет 8 байт; поэтому правильный ответ — `inc word1+8`.

5. Последний байт, заданный директивой `db`, находится по адресу `word1+3k` (он же — `byte1-1`). Выражение `word1+3k` имеет тип `byte`, а выражение `byte1-1` — тип `word`. Поэтому варианты увеличения байта следующие:

```
inc    word1+3k
inc    b byte1-1
```

Последнее слово перед `byte1` находится по смещению `byte1-2`. Тип этого выражения — `word`, поэтому при адресации относительно `byte1` команда выглядит следующим образом:

```
inc    byte1-2
```

7. Слово со значением `02fe` соответствует байтам `-2` и `2`, и находится по адресу `byte1`. Инструкция, следовательно, должна быть записана в виде `inc w byte1`. Слово со значением `0302` соответствует байтам `2` и `3`; команда для его увеличения — `inc w byte1+1`.

К разделу 2.1

2. Таблица имен в конце первого просмотра показана в табл. П9.3.

Таблица П9.3. Таблица имен для варианта 2

Имя	Значение	Тип
sz1	28	abs
neck	0100 (256)	byte
total	1215	abs
bottle	264	byte
end	1243	abs

Характерная ошибка: имени `bottle`, определенному через посредством `equ`, приписывают тип `abs`.

Имя слева от `equ` принимает значение и тип выражения справа от `equ`. Выражение `neck+8` имеет тип `byte`, поэтому тип `bottle` — тоже `byte`.

4. Таблица имен в конце первого просмотра показана в табл. П9.4.

Таблица П9.4. Таблица имен для варианта 4

Имя	Значение	Тип
dumb	0108 (264)	abs
pump	264	abs

Таблица П9.4 (окончание)

Имя	Значение	Тип
bumb	264	abs
lamp	308	word
zunz	307	word
zz	0	abs
end	484	abs

- Ошибка в том, что не определен счетчик повторов для `dup` в третьем операторе. Значение `(zz - start)` перед `dup` должно быть известно уже на первом просмотре, но его вычислить невозможно, поскольку `zz` еще не определено (выражение содержит опережающую ссылку). Вместе с тем, выражение `zz - start` в качестве *инициализирующего* значения допускается, т. к. на первом просмотре заполнять память не требуется (только зарезервировать).
- Не определено имя `beta`. Ошибка возникает на втором просмотре, при инициализации памяти.

К разделу 2.5

- Попытка переопределить ключевое слово `this`, которое обозначает счетчик адресов (синоним `$`). Если заменить `this` на другое имя (на `_this`, например), обнаруживается еще одна ошибка — "Overlapping Local Not Allowed".

Причина ошибки — в обращении к недействительному экземпляру локального имени. Начиная со второго оператора, все ссылки на локальное имя `l1` должны быть опережающими (`>l1`), пока `l1` не будет определено вновь. В предпоследнем операторе это правило нарушено.

- Таблица имен показана в табл. П9.5.

Таблица П9.5. Таблица имен для варианта 2

Имя	Значение	Тип
l1	268	byte
l2	268	abs
where	269	byte
end	268	abs

Примечание

Из всех операторов `db` распределяет память только первый. Во втором и четвертом операторах счетчик повторов равен нулю, поскольку ассемблер при вычислении выражения `$ - 11` использует *новое значение* имени `11`. Третий оператор вовсе игнорируется, поскольку он начинается с имени `end` (даже если бы имя было другим, память все равно не была бы распределена, т. к. счетчик повторов при `dup` равен нулю).

К разделу 5.1

1. Регистр `si` принимает значение 1. Регистр `di` будет установлен лишь частично: его старший байт равен 1, а младший байт содержит копию неизвестного нам значения регистра `al`.
4. Первая команда `inc` увеличивает только младший байт `ax`. Старший байт не затрагивается, так что перенос при сложении `0ff` с единицей в `ah` не передается. После выполнения первой инструкции `inc` значение в `ax` = 0100. После `inc ax` значение `ax` = 0101. Результат последующей инверсии разрядов — 0fef.

К разделу 5.3.2. Прямая адресация

1. Способы адресации в командах:

```
mov    ah, bh      ; регистровая
inc    w [0]       ; прямая
add    byte [0fffe], type (w [0]) ; прямая и непосредственная
```

2. Ошибки:

```
mov    1 shl 3, ax  ; приемник задан константой
add    bx           ; должно быть два операнда
mul    ax, cx       ; должен быть один операнд
```

3. Ошибки:

```
add    [10], 255    ; требуется уточнение размерности операции,
                    ; поскольку 255 может быть представлено
                    ; как байтом, так и словом
inc     [0]         ; требуется уточнение размерности
```

4. Адресация прямая (для множителя) и регистровая (для множимого и результата).
6. Способы адресации и ошибки:

```
add    5, 9         ; непосредственная, что недопустимо
                    ; для первого операнда add
```

```
add    ax, 1      ; регистровая и непосредственная
mov    al, cx     ; регистровая; операнды разной размерности,
                  ; что для mov недопустимо
```

К разделу 5.3.2. Косвенная адресация

2. Тело цикла:

```
add    w [si], 10
add    si, 2
Число повторов = size/2
```

3. Тело цикла:

```
add    w [si+array], 20
add    si, 4
Число повторов = size/4
```

К разделу 5.3.3

Ошибки:

- ❑ `add [si+bx], w[0]` ; сразу два операнда в памяти
- ❑ `inc [si+bx]` ; нет информации о размерности операции
- ❑ `inc w [bl]` ; `bl` не используется в качестве указателя
- ❑ `inc w[di+ax]` ; `ax` для косвенной адресации непригоден
- ❑ `add 1, b[si+di]` ; приемник — константа, а в источнике —
; два индексных регистра сразу
- ❑ `mov al, w [0]` ; размерность приемника и источника
; разная, а для `mov` это не годится

и т. д.

К разделу 7.4.2

2. Определение таблицы:

```
map db 'A' dup 0
db ('Z' - 'A' + 1) dup 1
db ('a' - 'Z' - 1) dup 0
db ('z' - 'a' + 1) dup 1
db 256 - ($ - map) dup 0
```

5. Определение таблицы:

```
map db 1, ('.' - 1) dup 0
    db 1, (255 - '.' - 1) dup 0
    db 1
```

7. Определение таблицы:

```
map db 25 dup (0, 3 dup 1, 6 dup 0)
    db 0, 3 dup 1
    db 256 - ($ - map) dup 0
```

К разделу 10.1

1. Абсолютные адреса операндов в памяти:

```
push ax ; ss:sp-2 = 05078:0fe = 050780 + 0fe = 05087e
es mov [2], bl ; es:2 = 098:2 = 0982
mov bp, di, 2 ; (нет обращения к памяти, bp = di = 2)
inc b [bp+di] ; ss:bp+di = 05078:2+2 = 050784
cs lodsb ; cs:si = 0fff0:si
```

2. Абсолютные адреса операндов в памяти:

```
mov bx, 8 ; (нет обращения к памяти, bx = 8)
mov cs:[bx], 16 ; cs:bx = 0fff0:8 = 0ff8
es mov [-2], bl ; es:-2 = 050 - 2 = 04e
cs jmp [bx] ; cs:bx = 0ff8
```

Переход по команде `jmp` выполняется в текущем сегменте кода — по смещению, заданному в слове по адресу `0ff8`. Это слово после выполнения второй инструкции содержит число 16. Следовательно, адрес перехода равен `cs:16 = 0fff0 + 010 = 01000`.

3. Примеры решений с использованием либо `ds`, либо `es` приведены в табл. П9.6. Применить `cs` вместо `ds` или `es` нельзя. Изменение значения `cs` равносильно передаче управления в другой сегмент.

Таблица П9.6. Решения варианта 3 с использованием `ds` или `es`

Решение с использованием <code>ds</code>	Решение с использованием <code>es</code>
<code>mov ds, 0</code>	<code>mov es, 040</code>
<code>mov al, [0417] ; 0:0417</code>	<code>es mov al, [017] ; 040:017</code>
<code>mov ds, 0b800</code>	<code>mov es, 0b800</code>
<code>mov [0], al ; 0b800:0</code>	<code>mov es:[0], al</code>

4. Вариант решения:

```
mov     cx, 2k
mov     ds, 040
mov     es, 0b800
mov     di, si, 0
cld
rep movsb    ; es:[di++] = ds:[si++]
```

5. Ошибки и лишние префиксы:

```
cs push 1           ; cs не действует
mov     ss:[bp+di], 0ff ; ss лишний
mov     cs, ds       ; запись в cs командой mov запрещена
ds inc  w [bx+4]      ; ds — лишнее
```

Команда `jmp cs:m1` требует пояснений. Казалось бы, префикс `cs` не должен подействовать, т. к. тип имени `m1` — `abs` — не предполагает доступа к памяти. Тем не менее, обозначение `seg_reg:abs` трактуется (по крайней мере, в ассемблерах `a86` и `tasm`) как `seg_reg:[abs]`. То есть, это обозначение задает адрес данных в памяти. Поэтому инструкция `jmp cs:m1` — это переход по адресу, заданному содержимым слова по адресу `cs:[m1]`. Если бы префикс был задан в стиле `a86` (`cs jmp m1`), смысл команды изменился бы — это уже переход на метку `m1`, и префикс `cs` здесь не действует.

6. Передача управления по адресу 0ffff8:

```
jmp     0ffff:8
```

Умножение байта по адресу 0b8020 на 7:

```
mov     ds, 0b800
mov     al, 7
mul     b [020]    ; ax := al * ds:[020]
mov     [020], al
```

К разделу 10.5

5. Адресация будет выполняться относительно начала PSP. Поскольку программа, показанная в листингах 10.5 и 10.6, занимает не более 256 байтов, то команда копирования будет адресовать данные внутри области PSP.

6. В результате ближнего вызова подпрограммы в стек будет записан ближний адрес возврата. При выполнении дальнего возврата `retf` в `ip` будет восстановлено правильное значение смещения, но в `cs` попадет случайное значение, не связанное с предшествующим вызовом. То есть возврат будет ошибочным.

ПРИЛОЖЕНИЕ 10

Ошибки в a86 v4.05

В версии a86 v4.05 обнаружены две незначительные ошибки.

Ошибка при оптимизации инструкции *call far*

Если процедура описана как дальняя, то при ее вызове из того же сегмента a86 заменяет (в пределах исходного модуля) инструкцию `call far <target>` на пару команд — `push cs` и `call near <target>`.

В `obj`-режиме эта функция выполняется без ошибки, но в `com`-режиме в `call near` подставляется неправильное значение адреса перехода.

По предложению автора a86, до выпуска следующей версии эту ошибку можно обойти за счет отключения оптимизации в `com`-режиме. Автор также приводит последовательность действий для исправления с этой целью одной команды a86 (средствами d86).

К сожалению, отключение оптимизации дальних вызовов в `com`-режиме приводит к отказу от дальних вызовов в `com`-программе. При отключении оптимизации непосредственный `far`-вызов будет представлен в виде `call 0:imm` — с правильным смещением, но с неопределенным значением сегмента. Значение сегмента не будет скорректировано при загрузке, поскольку `com`-программа не содержит таблицы перераспределения для коррекции непосредственно заданных значений сегментов. В итоге, вызов будет неправильным.

Поэтому в версии v4.05 не следует вводить дальние процедуры в `com`-программах (все равно их не стоит вводить без необходимости, тем более в программах из одного сегмента). Иначе придется вручную оптимизировать `far`-вызовы.

Ошибка в операторе *bit*

Оператор `bit <n>` при $n = 0 \dots 15$ работает исправно, но при $n > 16$ возвращает ноль.

ПРИЛОЖЕНИЕ 11

Описание дискеты

На прилагающейся дискете записаны:

- ❑ в каталоге a86 — архив пакета a86 v4.05, скопированный с авторского сайта;
- ❑ в каталогах chap_<i> — исходные тексты программ из примеров, по главам (<n> — номер главы);
- ❑ аналогично, в каталогах _арх_<i> — примеры к приложениям.

Примечание

Программы из пакета a86 относятся к категории shareware. То есть, коммерческое применение a86/d86 без его приобретения (~90\$) квалифицируется как нарушение авторских прав. Условия покупки a86/a386 приведены в авторской документации; вместе с 16-разрядной версией (a86/d86) вы приобретаете 32-разрядную (a386/d386) с поддержкой новейших расширений процессоров Pentium. Остальные программы из каталогов chap_<i> — свободно распространяемые.