Томский межвузовский центр
дистанционного образования

Ю.Б. Гриценко

СИСТЕМНОЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ

Учебное пособие

Томск–2006

Гриценко Ю.Б.
Системное программное обеспечение: Учебное пособие. — Томск: Томский межвузовский центр дистанционного образования, 2006. — 176 с.

Данное учебное пособие содержит курс «Системное программное обеспечение», изучаемый студентами специальности 230102 «Автоматизированные системы обработки информации и управление», обучающихся по дистанционной форме. В основу курса положено изучение архитектуры ЭВМ, режимов функционирования процессоров Intel и языка ассемблера IBM PC. Язык ассемблера — это символическое представление машинного языка. Все процессы в машине на самом низком, аппаратном уровне приводятся в действие только командами (инструкциями) машинного языка.

По-настоящему решить проблемы, связанные с аппаратурой (или даже, более того, зависящие от аппаратуры как, к примеру, повышение быстродействия программы), невозможно без знания ассемблера.

© Гриценко Ю.Б., 2006
© Томский межвузовский центр дистанционного образования, 2006

Введение

В 1980 году фирмой IBM был принят ряд практических шагов для создания нового персонального компьютера. Кульминация наступила 12 августа 1981 года, когда впервые была представлена IBM PC. После публичного представления первых PC даже в самой IBM не осознавали, что они сделали. Последовал головокружительный успех, приведший первоначально даже к дефициту и невероятной удаче торговцев от IBM, которые смогли превратить силикон в золото.

В 1980 году понятие «персональный компьютер» трактовалось широко. Рынок технического обеспечения отличался большим разнообразием. Тем не мене, все персональные компьютеры можно было разделить на три большие группы.

Компьютер Apple — самый главный и долговременный конкурент по популярности, который сумел выжить со своими оригинальными решениями. Разработка Apple-2 имела оригинальные и перспективные решения, которые в скором времени стали использоваться и при создании IBM PC.

Tandy/Radio Shack. Над вторым лагерем производителей настольных компьютеров выше всех реял флаг фирмы Radio Shack. Фирма выпускала очень широкий ассортимент компьютеров, но наибольшее распространение во время появления IBM PC имела модель TRS-80. Это был настольный компьютер, у которого в одном корпусе размещались монитор, клавиатуры и вся электроника.

CP/M. Третью группу производителей малых компьютеров объединяла фирма Control Program for Microcomputers. Фирма производила мощные и широко распространенные компьютеры с микропроцессорами 8080 и Z80 и гибкой операционной системой. Низкая стоимость и надежность привели к широкому распространению данной модели и ее использованию как стандарта.

Стратегия IBM.

Возможно, самым легким путем для достижения целей была покупка. Лучшим выбором была бы Apple. Однако Apple

не могла быть целью IBM. Продукция Apple, в первую очередь, ориентировалась не на деловых людей, а на любителей. Удовлетворить потребности любителей не являлось главной целью IBM. Radio Shack никогда не могла служить альтернативой. Производство компьютеров не являлось главным бизнесом фирмы. Другие производители были еще менее привлекательны. IBM не хотела покупать чей-либо гараж, оптимистично названный заводом по производству микрокомпьютеров.

Фирма имела свой собственный опыт по производству малых компьютеров. Она уже выпускала свой собственный переносной компьютер — модель 5100. Эта модель использовалась внутри самой фирмы и никогда не рассматривалась как коммерческая модель.

Сомнения не было — машина IBM должна была быть реализована на микропроцессоре. Встал вопрос — какой микропроцессор использовать, чип Apple 6502 был уже устаревшим даже в 1981 году. Этот чип мог оперировать только с 8-ю битами с частотой 1 МГц. Отсюда его производительность не могла сравниться с CP/M и микропроцессором Z80.

Intel 8088. Микропроцессор Intel 8088 имел много общих черт с Z80, и теоретически все программное обеспечение Z80 можно было бы легко приспособить для Intel 8088. Использовать Intel 8086 не было смысла, так как при этом значительно возрастала стоимость всей системы.

После микропроцессора следующий наиболее важный вопрос — это вопрос выбора памяти. Большинство программ того времени требовали оперативной памяти немногим больше 16 Кб. IBM пошла дальше. И обеспечила гораздо большие возможности. Она обеспечила 64 Кб памяти PC. Ориентируясь на будущее, IBM предусмотрела возможность установки дополнительной памяти вплоть до 512K. Для того времени это были немыслимые ресурсы, хотя программ, требующих их, тогда еще не было написано.

Каждому компьютеру требуется система отображения. IBM разработала свой собственный дисплей, который использовался как внутри фирмы, так и в выпускаемой продукции.

Клавиатура требуется для управления компьютером, а также для ввода информации. Вместо того чтобы следовать тех-

нологии Apple, которая поместила клавиатуру в один корпус с процессором, IBM повесила клавиатуру на провод, разместив ее в отдельном корпусе. При этом она руководствовалась опытом работы собственных сотрудников с такой клавиатурой.

Практически во всех компонентах современных PC на базе процессоров Intel имеется незримое присутствие корпорации IBM. На сегодняшний день корпорация IBM больше не занимает лидирующего положения на рынке PC, тем не мене, она продолжает производить компьютеры на базе процессоров Intel и продолжает оказывать значительное влияние на индустрию персональных компьютеров и серверных платформ.

История развития процессоров Intel

Каковы бы не были компоненты компьютера, в основе всего лежит процессор, именно он определяет важнейшие параметры компьютера.

Первый микропроцессор был изготовлен в 1971 году фирмой Intel Corporation. Это был 4-х битовый микропроцессор 4004, предназначенный для использования в калькуляторах.

В 1972 году появился восьми битовый микропроцессор 8008, широко использовавшийся при производстве персональных компьютеров.

В 1974 году был создан более интересный микропроцессор 8080. Множество его микрокоманд было расширено, и он стал первым процессором, умевшим делить числа. Несколько инженеров фирмы имели идеи по усовершенствованию этого процессора и они покинули Intel и создали корпорацию Zilog Corporation и подарили миру процессор Z80.

В 1976 году фирма Intel начала усиленно работать над микропроцессором 8086. Это был 16-ти битовый микропроцессор с 20-ти битовой адресной шиной, что дало ему возможность адресовать 1 Мб оперативной памяти.

Через год после презентации 8086 Intel объявил о разработке микропроцессора 8088. Он являлся близнецом 8086: 16-ти битные регистры, 20 адресных линий, тот же набор команд — все то же, за исключением одного, — шина данных была уменьшена до 8 бит. Это позволяло полностью использовать широко распространенные в то время 8-битные элементы технического обеспечения. Именно

этот процессор использовала IBM при реализации своего первого персонального компьютера.

Презентация IBM персонального компьютера AT в 1984 году сфокусировала все внимание на микропроцессор Intel 80286. Сам по себе микропроцессор был представлен еще в 1982 году. Большим преимуществом Intel 80286 была способность работать с дополнительной памятью до 16 Мб. Так же появилась возможность использования виртуальной памяти и мультизадачность. Для использования новых возможностей процессор необходимо было переключать в защищенный режим. Реальный режим был предназначен для совместимости с предыдущими моделями.

В 1985 году инженерам корпорации Intel удалось создать настоящий шедевр — микропроцессор 80386. Intel 80386 имел все положительные качества своих предшественников. Также он был снабжен дополнительными качествами. Особенно привлекала возможность работать без ограничений, связанных с сегментацией памяти. Intel 80386 был мощнее своих предшественников. Размеры его регистров шины данных были удвоены до 32 бит. Число адресных линий было так же увеличено до 32 бит, что дало возможность адресовать до 4 Гб. На сегодняшний день дальнейшим развитием архитектуры Intel являются процессоры Pentium, Pentium Pro (адресная шина увеличена до 36 бит), Pentium II, Pentium III, Pentium IV.

1. Основные принципы построения и архитектура ЭВМ

1.1 Язык машины

1.1.1 Двоичная и шестнадцатеричная система исчисления

Существует два способа представления сигналов для хранения и передачи и обработки информации: аналоговый и цифровой. Аналоговый сигнал по своей природе является непрерывным и находится в некотором диапазоне. Значение его непрерывно меняется во времени. Практически все сигналы, с которыми сталкивается человек, являются аналоговыми. Например, запись звука на магнитной кассете. Цифровой сигнал является дискретным сигналом и имеет только два значения — высокий и низкий уровень. Обычно низкий сигнал интерпретируется как «0», а высокий — как «1».

В повседневной деятельности мы привыкли к счету с применением десяти цифр «1», «2», «3» и т.д. Вследствие чисто технических особенностей компьютер применяет другой метод счета, в котором задействованы только две цифры: «0» и «1». Например, до пяти он считает следующим образом: «1», «10», «11», «100», «101». Данные числа являются двоичными. Таким образом, двоичное число «10» соответствует десятичному «2». Для обозначения двоичного числа к ним принято приписывать символ «b», например «1010b».

Увеличивая число элементов, можно увеличивать число значений до необходимой величины. Таким образом, мы получили возможность хранить числа в любом диапазоне. Элемент памяти, содержащий один двоичный разряд, называется битом. Общее число значений зависит от числа разрядов по следующему закону: **Число значений=2^число разрядов**.

Один бит является слишком маленькой единицей информации. Он содержит только два значения: 0 или 1, что слишком мало для использования в одной операции. Обычно биты группируют по восемь. Единица информации, состоящая из восьми

битов, называется байтом. Для большего значения используется следующие единицы информации:

- слово — 2 байта или 16 бит, максимальное значение 65535 (2^16-1);
- двойное слово — 4 байта или 32 бита, максимальное значение 2^32-1;
- четверное слово — 8 байт или 64 бита, максимальное значение 2^64-1.

Двоичные числа обладают существенным недостатком – выглядят они длинно и громоздко. Для этого были введены шестнадцатеричные числа. Они предназначены для более компактного записывания двоичных чисел. В таблице 1.1 приведено соотношение цифр различной арифметики.

Таблица 1.1–Соотношение цифр различной арифметики

Десятичные	Двоичные	Шестнадцатиричные
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

В таблице 1.2 приведем примеры работы шестнадцатеричной арифметики, в которой единица информации представлена двумя байтами — словом.

Таблица 1.2–Примеры работы шестнадцатиричной арифметики

№	Десятичная арифметика	Шестнадцатеричная арифметика
1	2 + 3 = 5	2h[1] + 3h = 5h
2	2 – 3 = -1	2h – 3h = FFFFh
3	9 + 5 = 14	9h + 5h = Eh
4	9 – 5 = 4	9h – 5h = 4h
5	-1 + 5 = 4 65536 + 5 = 4	FFFFh + 5h = 4h
6	15 + 3 = 18	Fh + 3 = 12h

С первой и четвертой строкой все ясно, шестнадцатеричная арифметика ведет себя так же как и десятичная. Если внимательно рассмотреть таблицу 1.2, то приведенные в третьей строке примеры тоже становятся понятны, десятичное число «14» заменяется шестнадцатеричным «Eh». А вот вторая и пятая строки на первый взгляд ведут себя странно. Шестнадцатеричное число FFFF соответствует единице со знаком минус: «-1». Пример в стоке 6 показывает, что при сложении чисел «F» и «3» происходит перенос «1» в следующий разряд. Теперь посмотрим, что происходит, когда мы складываем «5h» и «FFFFh»:

$$\begin{array}{r} 0005h \\ +\ \ FFFFh \\ \hline 10004h \end{array}$$

Происходит последовательный перенос единицы в крайнюю левую позицию. Если мы игнорируем эту единицу, то получаем правильный ответ, а именно «4». Данная ситуация называется переполнением, так как теперь число пятизначное, а в слове хранится четыре цифры.

Допустим FFFFh равным 65536. Это положительное число и оно максимальное из тех, которые возможно записать при помощи шестнадцатеричных цифр. Число FFFFh — это число без знака («unsigned»). Если принимаем, что FFFFh знаковое число то оно будет соответствовать «-1». Фактически, все числа от FFFFh до 8000h ведут прекрасно себя как отрицательные числа.

---

[1] К шестнадцатеричному числу принято приписывать символ «h»

Для чисел со знаком переполнение не является ошибкой, а для чисел со знаком переполнение — ошибка.

1.1.2 Перевод чисел из одной системы исчисления в другую

Рассмотрим перевод шестнадцатеричных чисел в десятичные числа.

$$\begin{array}{l} \text{E} \longrightarrow 14 \times 16 = 224 \\ + \\ 6 \longrightarrow \quad 6 \times 1 = \quad 6 \\ \hline \text{E6h} \qquad\qquad = 230 \end{array}$$

Возьмем число «E6h», «E» — это шестнадцатеричная цифра 14, и там 16 таких цифр (по аналогии с 10 для десятичного числа). Таким образом, «E6h» — это четырнадцать раз по шестнадцать и шесть единиц. Для отделения разрядов шестнадцатеричного числа можно применять следующие четыре числа.

$$16^3 = 4096$$
$$16^2 = 256$$
$$16^1 = 16$$
$$16^0 = 1$$

Теперь научимся переводить десятичные числа в шестнадцатеричную форму.

$$\begin{array}{l} 1069/16 = 66 \ \ (1069 - 66 * 16 = 13) \\ \quad 66/16 = 4 \ \ (66 - 4 \times 16 = 2) \\ \quad\ 4/16 = 0 \ \ (4 - 0 \times 16 = 4) \\ \hline 1069 \qquad = \qquad 42\text{Dh} \end{array}$$

Для этого 1069 разделим на 16, остаток от деления будет давать нам первую шестнадцатеричную цифру, далее целую часть от деления разделим еще раз на шестнадцать, на этот раз остаток даст нам вторую цифру и т.д.

1.2 Архитектура ЭВМ и ее свойства

На современном компьютерном рынке наблюдается большое разнообразие различных типов компьютеров. Поэтому возможно предположить возникновение у потребителя вопроса — как оценить возможности конкретного типа (или модели) компьютера и его отличительные особенности от компьютеров других типов (моделей). Рассмотрения для этого одной лишь только структурной схемы компьютера недостаточно, так как она принципиально мало чем различается у разных машин: у всех компьютеров есть оперативная память, процессор, внешние устройства.

Различными являются способы, средства и используемые ресурсы, с помощью которых компьютер функционирует как единый механизм. Чтобы собрать воедино все понятия, характеризующие компьютер с точки зрения его функциональных программно-управляемых свойств, существует специальный термин — архитектура ЭВМ.

Архитектура ЭВМ — это абстрактное представление ЭВМ, которое отражает ее архитектуру, схемотехническую и логическую организацию [1]. Понятие архитектуры ЭВМ является комплексным и включает в себя:

- структурную схему ЭВМ;
- средства и способы доступа к элементам структурной схемы ЭВМ;
- организацию и разрядность интерфейсов ЭВМ;
- набор и доступность регистров;
- организацию и способы адресации памяти;
- способы представления и форматы данных ЭВМ;
- набор машинных команд ЭВМ;
- форматы машинных команд;
- обработку нештатных ситуаций (прерываний).

Все современные ЭВМ обладают некоторыми общими и индивидуальными свойствами архитектуры. К числу *общих архитектурных свойств и принципов* можно отнести [1]:

- Принцип хранимой программы. Согласно ему, код программы и ее данные находятся в одном адресном пространстве в оперативной памяти.
- Принцип микропрограммирования. В состав процессора входит блок микропрограммного управления. Этот блок для каждой машинной команды имеет набор действий-сигналов, которые нужно сгенерировать для физического выполнения требуемой машинной команды.
- Линейное пространство памяти — совокупность ячеек памяти, которым последовательно присваиваются номера (адреса) 0, 1, 2, …
- Последовательное выполнение программ. Процессор выбирает из памяти команды строго последовательно. Для изменения прямолинейного ходя выполнения программ необходимо использовать специальные команды (команды условного и безусловного перехода).
- Процессор не различает команды и данные, поэтому важно в программе всегда четко разделять пространство данных и команд.
- Безразличие к целевому назначению данных. Машине все равно, какую логическую нагрузку несут обрабатываемые ею данные.

Перечень *индивидуальных свойств и принципов микропроцессоров* весьма велик, наиболее значимыми являются [1]:

- Суперскалярная архитектура. Важным элементом архитектуры, появившимся в i486, стал конвейер — специальное устройство, при котором выполнение команд в микропроцессоре разбивается на несколько этапов. В i486 они следующие:
    - выборка команд из кэш-памяти или оперативной;
    - декодирование команды;
    - генерация адреса;
    - выполнение операции с помощью АЛУ (арифметико-логическое устройство);
    - запись результата.
- Преимуществом такого подхода является то, что очередная команда после ее выборки попадает в блок декодирования.

Таким образом, блок выборки свободен и может выбрать следующую команду. В результате на конвейере могут находиться в различной стадии выполнения пять команд. Микропроцессоры, имеющие один конвейер, называются скалярными, а два и более — суперскалярными.

- Раздельное кэширование кода и данных. Кэширование — это способ увеличения быстродействия системы за счет хранения часто используемых данных и кодов в так называемой «кэш-памяти первого уровня», находящейся внутри микропроцессора.
- Предсказание правильного адреса перехода. Под переходом понимают запланированное алгоритмом изменение последовательного характера выполнения программы. Типичная программа на каждые 6-8 команд содержит одну команду перехода. Последствия этого предсказать нетрудно: при наличии конвейера через каждые 6-8 команд его нужно очищать и заполнять заново в соответствии с адресом перехода. Все преимущества конвейеризации теряются. Поэтому в архитектуру Pentium был введен блок предсказания переходов. Вероятность правильного предсказания перехода составляет около 80%.

Режимы работы процессора. Процессоры семейства Intel x86, начиная с 80286, имеют несколько режимов работы:

- Реальный режим работы. Предназначен для совместимости с младшими моделями процессоров.
- Защищенный режим. Основной режим работы процессоров. Именно в нем доступны все особенности новых моделей процессоров — такие, как многозадачность, защита программ пользователей, возможность заботы с большим объемом памяти, виртуальная память и т.д.
- Режим системного управления (SMM). В этом режиме доступны дополнительные возможности процессора.
- Режим Virtual-86. Этот режим схож с реальным режимом, однако может быть включен только в защищенном режиме. В этом режиме возможно выполнение нескольких приложений реального режима.

Адресное пространство в системах на базе процессоров x86. Адресное пространство зависит от разрядности шины адреса процессора. Каждый тип процессоров имеет разную разрядность шины адреса и шины данных. В процессоре 8088 и 8086 использовалась 20-ти разрядная шина адреса, и, следовательно, объем адресуемой памяти составлял 2^20 или 1 Мб. В процессоре 80286 использовалась 24-х разрядная шина адреса, и, следовательно, объем адресуемой памяти составлял 2^24 или 16 Мб. В процессорах 80386 и 80486, используется 32-х разрядная шина адреса, и, следовательно, объем адресуемой памяти составляет 2^32 или 4 Гб. Начиная с процессоров Pentium PRO, адресная шина была увеличена до 36 бит (64 Гб). Однако не все адресное пространство отводится под оперативную память, часть этого пространства отводится под ПЗУ и видеобуфер. Детальное описание адресного пространства приводится в таблице 1.3.

Таблица 1.3—Адресное пространство

Диапазон адресов	**Размер**	**Описание**
0h-3FFh	1024	Таблица векторов прерываний для реального режима
400h-4FFh	256	Переменные базовой системы ввода вывода (ROM BIOS).
500-9FFFFh	-	Нижняя память, доступная для программ пользователя
A0000h-AFFFFh	10000h	Видеобуфер для графических режимов
B0000h-B7FFFh	8000h	Видеобуфер для монохромных текстовых режимов
B8000h-BFFFFh	8000h	Видеобуфер для цветных текстовых режимов
C0000h-FFFEFh	3FFF0h	ПЗУ
FFFF0h-FFFF4h	5	Команда JMP на первый код запуска или перезапуска компьютера
FFFF5h-FFFFDh	9	Дата создания базовой системы ввода — вывода
FFFFEh-FFFFFh	2	Код идентификации IBM PC

Продолжение таблицы 1.3—Адресное пространство

Диапазон адресов	Размер	Описание
100000h-FFFFFFh	-	Оперативная память, доступная начиная с 80286 процессора
1000000h-FFFFFFFFh	-	Оперативная память, доступная начиная с 80386 процессора
100000000-FFFFFFFFF	-	Оперативная память, доступная начиная с Pentium Pro

Работа с внешними устройствами. Во всех ранних, а также и современных, моделях процессоров для работы с внешними устройствами используется отдельная шина адреса и шина данных. Разрядность этой шины составляет 16 бит, и, следовательно, возможно адресовать 65536 различных ячеек, называемых портами. Доступ к портам может осуществляться как 8-ми битовый, так и 16-ти и 32-х битовый. Однако в отличие от памяти доступ к портам как к словам или двойным словам должен быть выровнен и на слово двойное слово соответственно. Таким образом, число 8-ми битовых портов — 65536, 16 битовых — 32768, 32-х битовых — 16384.

1.3 Программная модель микропроцессора Intel Pentium

1.3.1 Состав программной модели

Любая выполняющаяся программа получает в свое распоряжение определенный набор ресурсов микропроцессора. Эти ресурсы необходимы для выполнения и хранения в памяти команд программы, данных и информации о текущем состоянии программы и микропроцессора. Набор этих ресурсов представляет собой *программную модель микропроцессора.*

Программные модели более ранних микропроцессоров (i486, Pentium) отличаются меньшим размером адресуемого пространства оперативной памяти (2^{32}-1, так как разрядность их шины адреса составляет 32 бита) и отсутствием некоторых групп регистров. Для каждой группы регистров в скобках обозначено, начиная с какой модели данная группа регистров поя-

вилась в программной модели микропроцессоров Intel. Если такого обозначения нет, то это означает, что данная группа регистров присутствовала в микропроцессорах 1386 и i486. Более ранние микропроцессоры архитектуры Intel мы не рассматриваем ввиду их архаичности, но это вовсе не означает того, что данная книга не может использоваться для их программирования.

Программную модель микропроцессора Intel составляют [1]:

- пространство адресуемой памяти (для Pentium III — до 2^{36}- 1 байт);
- набор регистров для хранения данных общего назначения (eax/ax/ah/al, ebx/bx/bh/bl, edx/dx/dh/dl, ecx/cx/ch/cl, ebp/bp, esi/si, edi/di, esp/sp). Регистры этой группы используются для хранения данных и адресов;
- набор сегментных регистров (cs, ds, ss, es, fs, gs). Регистры этой группы используются для хранения адресов сегментов в памяти;
- набор регистров состояния и управления — (это регистры, которые содержат информацию о состоянии микропроцессора, исполняемой программы и позволяют изменить это состояние:
    - регистр флагов eflags/flags;
    - регистр указатель команды eip/ip;
- системные регистры — это регистры для поддержания различных режимов работы, сервисных функций, а также регистры, специфичные для определенной модели микропроцессора;
- набор регистров устройства вычислений с плавающей точкой (сопроцессора) (st(0), st(l), st(2), st(3), st(4), st(5), st(6), st(7)). Регистры этой группы предназначены для написания программ, использующих тип данных с плавающей точкой;
- набор регистров целочисленного MMX-расширения (mmx0, mmx1, mmx2, mmx3, mmx4, mmx5, mmx6, mmx7), отображенных на регистры сопроцессора (впервые появились в архитектуре микропроцессора Pentium MMX);

- набор регистров MMX-расширения с плавающей точкой (xmm0, xmm1, xmm2, xmm3, xmm4, xmm5, xmm6, xmm7) (впервые появились в архитектуре микропроцессора Pentium Ш);
- программный стек. Это специальная информационная структура, работа с которой предусмотрена на уровне машинных команд.

Программные модели ранних микропроцессоров Intel составляют лишь небольшую часть приведенной программной модели. Так, в программную модель микропроцессора i8086 входят 8- и 16-битные регистры общего назначения, сегментные регистры, регистры flags, ip и адресное пространство памяти размером до 1 Мбайт.

Для обеспечения работоспособности программ, написанных для младших 16-разрядных моделей микропроцессоров фирмы Intel, начиная с i8086, микропроцессоры i486 и Pentium имеют, в основном, 32-разрядные регистры. Их количество, за исключением сегментных регистров, такое же, как и у i8086, но размерность больше, что и отражено в их обозначениях, — они имеют приставку е (Extended).

1.3.2 Регистры общего назначения

Регистры общего назначения используются в программах для хранения:

- операндов логических и арифметических операций;
- компонентов адреса;
- указателей на ячейки памяти.

Все эти регистры доступны для хранения операндов без особых ограничений, хотя при определенных условиях некоторые из них все же имеют жесткое функциональное назначение, закрепленное на уровне логики работы машинных команд. Среди всех этих регистров особо следует выделить регистр esp. Его не следует использовать явно для хранения каких-либо операндов программы, так как в нем хранится указатель на положение вершины стека программы. Все регистры этой группы позволяют обращаться к своим «младшим» частям (рис. 1.1). Рассмат-

ривая этот рисунок, заметьте, что использовать для самостоятельной адресации можно только младшие 16- и 8-битные части этих регистров. Старшие 16 битов этих регистров как самостоятельные объекты недоступны. Это сделано, как мы отметили выше, для совместимости с младшими 16-разрядными моделями микропроцессоров фирмы Intel. Перечислим регистры, относящиеся к группе регистров общего назначения. Так как эти регистры физически находятся в микропроцессоре внутри арифметико-логического устройства (АЛУ), то их еще называют *регистрами АЛУ:*

- eax/ax/ah/al (Accumulator register) — *аккумулятор.* Применяется для хранения промежуточных данных. В некоторых командах использование этого регистра обязательно;
- ebx/bx/bh/bl (Base register) — *базовый регистр.* Применяется для хранения базового адреса некоторого объекта в памяти;
- ecx/cx/ch/cl (Count register) — *регистр-счетчик.* Применяется в командах, производящих некоторые повторяющиеся действия. Его использование зачастую неявно и скрыто в алгоритме работы соответствующей команды. К примеру, команда организации цикла loop, кроме передачи управления команде, находящейся по некоторому адресу, анализирует и уменьшает на единицу значение регистра есх/сх;
- edx/dx/dh/dl (Data register) — *регистр данных.* Так же как и регистр eax/ax/ah/ al, он хранит промежуточные данные. В некоторых командах его использование обязательно; для некоторых команд это происходит неявно.

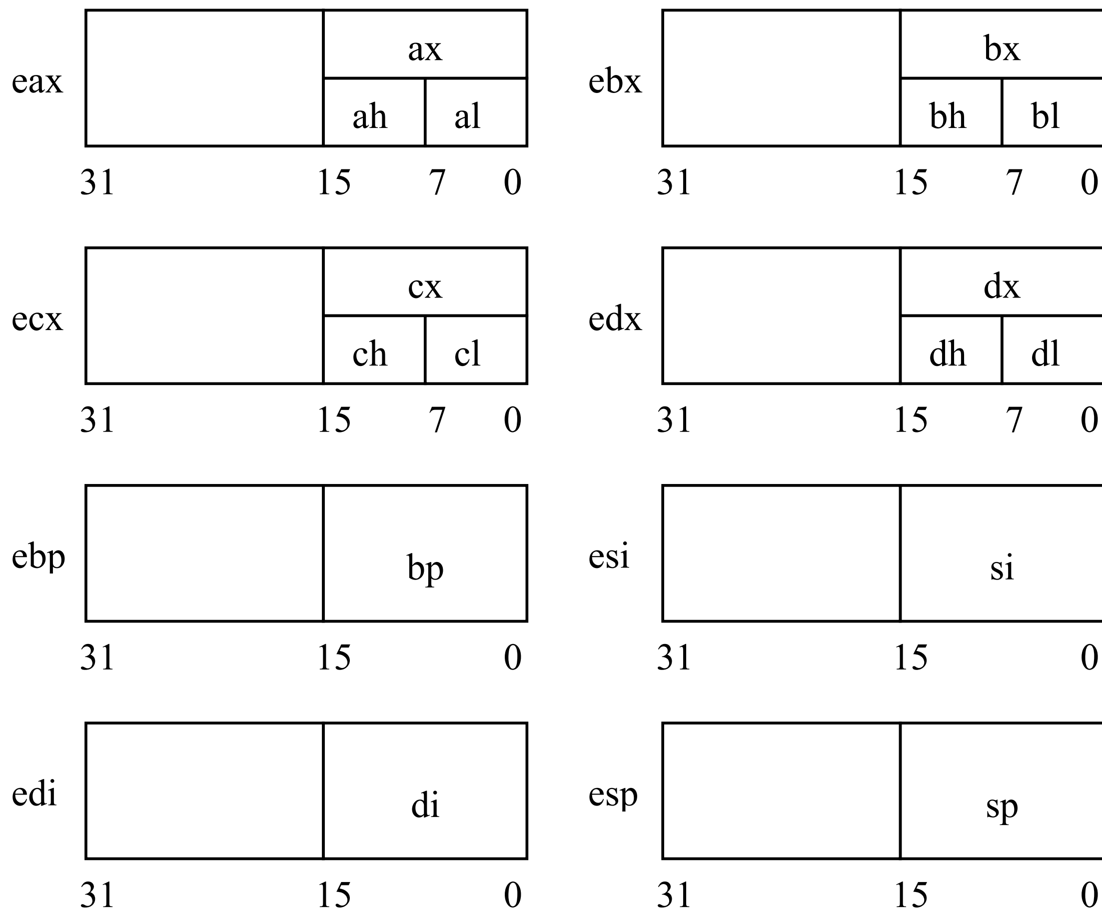


Рис. 1.1–Регистры общего назначения

Следующие два регистра используются для поддержки так называемых цепочечных операций, то есть операций, производящих последовательную обработку цепочек элементов, каждый из которых может иметь длину 32, 16 или 8 бит:

- esi/si (Source Index register) — *индекс источника.* Этот регистр в цепочечных операциях содержит текущий адрес элемента в цепочке-источнике;
- edi/di (Destination Index register) — *индекс приемника (получателя).* Этот регистр в цепочечных операциях содержит текущий адрес в цепочке-приемнике.

В архитектуре микропроцессора на программно-аппаратном уровне поддерживается такая структура данных, как стек. В свое время мы подробно познакомимся с тем, как его использовать. Для работы со стеком в системе команд микро-

процессора есть специальные команды, а в программной модели микропроцессора для этого существуют специальные регистры:

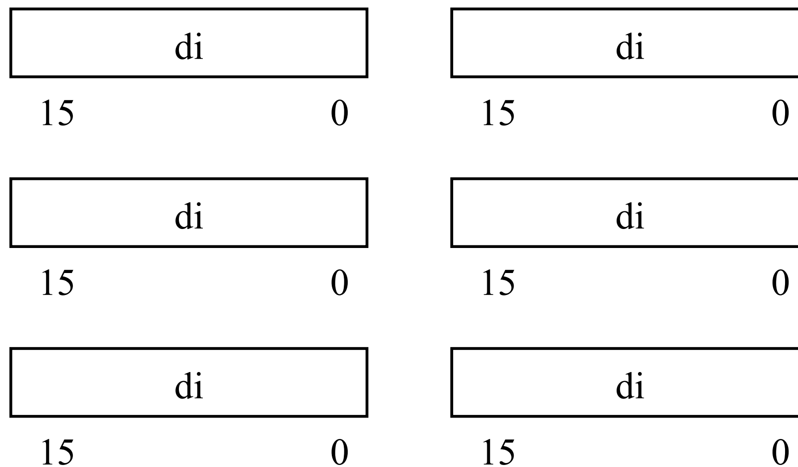
- esp/sp (Stack Pointer register) — *регистр указателя стека.* Содержит указатель вершины стека в текущем сегменте стека;
- ebp/bp (Base Pointer register) — *регистр указателя базы кадра стека.* Предназначен для организации произвольного доступа к данным внутри стека.

Не спешите пугаться столь жесткого функционального назначения регистров АЛУ. На самом деле большинство из них может использоваться при программировании для хранения операндов практически в любых сочетаниях. Возможные варианты использования этих регистров приведены в синтаксических диаграммах команд в Справочнике. Но, как мы отметили выше, некоторые команды используют фиксированные регистры для выполнения своих действий. Это нужно обязательно учитывать. Использование жесткого закрепления регистров для некоторых команд позволяет более компактно кодировать их машинное представление. Знание этих особенностей позволит вам при необходимости хотя бы на несколько байт сэкономить память, занимаемую кодом программы.

1.3.3 Сегментные регистры

В программной модели микропроцессора имеется шесть сегментных регистров: cs, ss, ds, es, gs, fs (рис. 1.2). Их существование обусловлено спецификой организации и использования оперативной памяти микропроцессорами Intel. Она заключается в том, что микропроцессор аппаратно поддерживает структурную организацию программы в виде трех частей, называемых сегментами. Соответственно, такая организация памяти называется сегментной. Для того чтобы указать на сегменты, к которым программа имеет доступ в конкретный момент времени, и предназначены сегментные регистры. Фактически, с небольшой поправкой, как мы увидим далее, в этих регистрах содержатся адреса памяти, с которых начинаются соответствующие сегменты.

Логика обработки машинной команды построена так, что при выборке команды, доступе к данным программы или к стеку неявно используются адреса во вполне определенных сегментных регистрах. Микропроцессор поддерживает следующие типы сегментов:



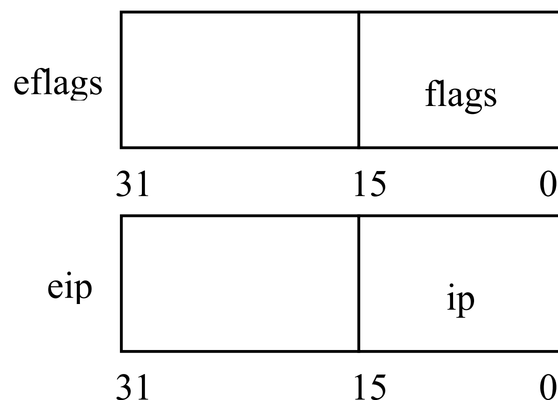
1.2–Сегментные регистры

- *Сегмент кода.* Содержит команды программы. Для доступа к этому сегменту служит регистр cs (code segment register) — *сегментный регистр кода.* Он содержит адрес сегмента с машинными командами, к которому имеет доступ микропроцессор (то есть эти команды загружаются в конвейер микропроцессора);
- *Сегмент данных.* Содержит обрабатываемые программой данные. Для доступа к этому сегменту служит регистр ds (data segment register) — *сегментный регистр данных,* который хранит адрес сегмента данных текущей программы;
- *Сегмент стека.* Этот сегмент представляет собой область памяти, называемую стеком. Работу со стеком микропроцессор организует по следующему принципу: последний записанный в эту область элемент выбирается первым. Для доступа к этому сегменту служит регистр ss (stack segment register) — *сегментный регистр стека,* содержащий адрес сегмента стека;

- *Дополнительный сегмент данных.* Неявно алгоритмы выполнения большинства машинных команд предполагают, что обрабатываемые ими данные расположены в сегменте данных, адрес которого находится в сегментном регистре ds. Если программе недостаточно одного сегмента данных, то она имеет возможность использовать еще три дополнительных сегмента данных. Но в отличие от основного сегмента данных, адрес которого содержится в сегментном регистре ds, при использовании дополнительных сегментов данных их адреса требуется указывать явно с помощью специальных префиксов переопределения сегментов в команде. Адреса дополнительных сегментов данных должны содержаться в регистрах es, gs, fs (extension data segment registers).

1.3.4 Регистры состояния и управления

В микропроцессор включены несколько регистров, которые постоянно содержат информацию о состоянии как самого микропроцессора, так и программы, команды которой в данный момент загружены на конвейер. Используя эти регистры, можно получать информацию о результатах выполнения команд и влиять на состояние самого микропроцессора. К этим регистрам относятся (рис. 1.3):



1.3–Регистры флагов и указателя команд

- eflags/flags (flag register) — регистр флагов. Разрядность eflags/flags — 32/16 бит. Отдельные биты данного регистра имеют определенное функциональное назначение и называются флагами. Младшая часть этого регистра полностью аналогична регистру flags для i8086 (рис. 1.4).

- eip/ip (Instruction Pointer register) — указатель команд. Регистр eip/ip имеет разрядность 32/16 бит и содержит смещение следующей подлежащей выполнению команды относительно содержимого сегментного регистра cs в текущем сегменте команд. Этот регистр непосредственно недоступен программисту, но загрузка и изменение его значения производятся различными командами управления, к которым относятся команды условных и безусловных переходов, вызова процедур и возврата из процедур. Возникновение прерываний также приводит к модификации регистра eip/ip.

Исходя из особенностей использования, флаги регистра eflags/flags можно разделить на три группы:

- 8 флагов состояния. Эти флаги могут изменяться после выполнения машинных команд. Флаги состояния регистра efl ags отражают особенности результата исполнения арифметических или логических операций. Это дает возможность анализировать состояние вычислительного процесса и реагировать на него с помощью команд условных переходов и вызовов подпрограмм. В таблице 1.4 приведены флаги состояния и указано их назначение;
- 1 флаг управления. Обозначается как df (Directory Flag). Он находится в десятом бите регистра eflags и используется цепочечными командами. Значение флага df определяет направление поэлементной обработки в этих операциях: от начала строки к концу (df = 0) либо, наоборот, от конца строки к ее началу (df = 1). Для работы с флагом df существуют специальные команды eld (снять флаг df) и std (установить флаг df). Применение этих команд позволяет привести флаг df в соответствие с алгоритмом и обеспечить автоматическое увеличение или уменьшение счетчиков при выполнении операций со строками;
- 5 системных флагов, управляющих вводом/выводом, маскируемыми прерываниями, отладкой, переключением между задачами и виртуальным режимом 8086 (табл. 1.5). Прикладным программам не рекомендуется модифицировать без необходимости эти флаги, так как в большинстве случаев это приведет к прерыванию работы программы.

Таблица 1.4–Флаги состояния и их назначение

Мнемоника флага	Флаг	Номер бита в eflags	Содержание и назначение
cf	Флаг переноса (Carry Flag)	0	1 — арифметическая операция произвела перенос из старшего бита результата. Старшим является 7, 15 или 31-й бит в зависимости от размерности операнда; 0 — переноса не было
pf	Флаг паритета (Parity Flag)	2	1 — 8 младших разрядов (этот флаг — только для 8 младших разрядов операнда любого размера) результата содержат четное число единиц; 0 — 8 младших разрядов результата содержат нечетное число единиц
af	Вспомогательный флаг переноса (Auxiliary carry Flag)	4	Только для команд, работающих с BCD-числами. Фиксирует факт заема из младшей тетрады результата: 1 — в результате операции сложения был произведен перенос из разряда 3 в старший разряд или при вычитании был заем в разряд 3 младшей тетрады из значения в старшей тетраде; 0 — переносов и заемов в(из) 3 разряд(а) младшей тетрады результата не было
zf	Флаг нуля (Zero Flag)	6	1 — результат нулевой; 0 — результат ненулевой
sf	Флаг знака (Sign Flag)	7	Отражает состояние старшего бита результата (биты 7, 15 или 31 для 8, 16 или 32-разрядных операндов соответственно):

Продолжение таблицы 1.4

Мнемо-ника флага	Флаг	Номер бита в eflags	Содержание и назначение
of	Флаг пере-полнения (Overflow Flag)	11	Флаг of используется для фиксирования факта потери значащего бита при арифме-тических операциях: 1 — в результате операции происходит перенос (заем) в(из) старшего, знакового бита результата (биты 7, 15 или 31 для 8, 16 или 32-разрядных операндов соот-ветственно); 0 — в результате операции не происходит переноса (заема) в(из) старшего, знакового бита результата
iopl	Уровень При-вилегий вво-да-вывода (Input/Output Privilege Level)	12, 13	Используется в защищенном режиме работы микропроцес-сора для контроля доступа к командам ввода-вывода в зависимости от привилегиро-ванности задачи
nt	Флажок вло-женности задачи (Nested Task)	14	Используется в защищенном режиме работы микропроцес-сора для фиксации того фак-та, что одна задача вложена в другую

Таблица 1.5–Системные флаги

Мнемоника флага	Флаг	Номер бита в *eflags*	Содержание и назначение
tf	Флаг трассировки (Trace Flag)	8	Предназначен для организации пошаговой работы микропроцессора. 1 — микропроцессор генерирует прерывание с номером 1 после выполнения каждой машинной команды. Может использоваться при отладке программ; 0 — обычная работа.
if	Флаг прерывания (Interrupt enable Flag)	9	Предназначен для разрешения или запрещения (маскирования) аппаратных прерываний. 1 — аппаратные прерывания разрешены; 0 — аппаратные прерывания запрещены.
rf	Флаг возобновления (Resume Flag)	16	Используется при обработке прерываний от регистров отладки.
vm	Флаг виртуального (Virtual 8086 Mode)	17	Признак работы микропроцессора в режиме виртуального 8086. 1 — процессор работает в режиме виртуального 8086; 0 — процессор работает в реальном или защищенном режиме
ac	Флаг контроля выравнивания (Alignment Check)	18	Предназначен для разрешения контроля выравнивания при обращениях к памяти. Используется совместно с битом am в системном регистре cr0.

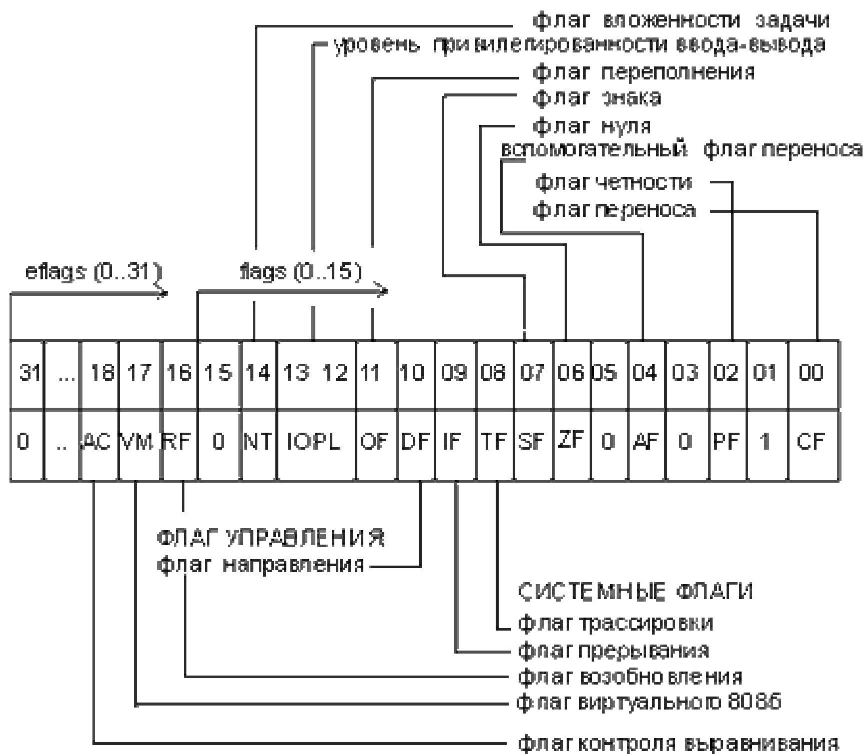


Рис. 1.4—Содержимое регистра eflags

1.3.5 Системные регистры микропроцессора

Само название этих регистров говорит о том, что они выполняют специфические функции в системе. Использование системных регистров жестко регламентировано. Именно они обеспечивают работу защищенного режима. Их также можно рассматривать как часть архитектуры микропроцессора, которая намеренно оставлена видимой для того, чтобы квалифицированный системный программист мог выполнить самые низкоуровневые операции.

Системные регистры можно разделить на три группы [1]:

- четыре регистра управления;
- четыре регистра системных адресов;
- восемь регистров отладки.

Регистры управления. В группу регистров управления входят 4 регистра: cr0, cr1, cr2, cr3.

Эти регистры предназначены для общего управления системой. Регистры управления доступны только программам с уровнем привилегий 0.

Хотя микропроцессор имеет четыре регистра управления, доступными являются только три из них — исключается cr1, функции которого пока не определены (он зарезервирован для будущего использования).

Регистр cr0 содержит системные флаги, управляющие режимами работы микропроцессора и отражающие его состояние глобально, независимо от конкретных выполняющихся задач. Назначение системных флагов:

- pe (Protect Enable), бит 0 — разрешение защищенного режима работы. Состояние этого флага показывает, в каком из двух режимов — реальном (pe=0) или защищенном (pe=1) — работает микропроцессор в данный момент времени.
- mp (Math Present), бит 1 — наличие сопроцессора. Всегда 1.
- ts (Task Switched), бит 3 — переключение задач. Процессор автоматически устанавливает этот бит при переключении на выполнение другой задачи.
- am (Aligment Mask), бит 18 — маска выравнивания. Этот бит разрешает (am=1) или запрещает (am=0) контроль выравнивания.
- cd (Cache Disable), бит 30 — запрещение кэш-памяти. С помощью этого бита можно запретить (cd=1) или разрешить (cd=0) использование внутренней кэш-памяти (кэш-памяти первого уровня).
- pg (PaGing), бит 31 — разрешение (pg=1) или запрещение (pg=0) страничного преобразования. Флаг используется при страничной модели организации памяти.

Регистр cr2 используется при страничной организации оперативной памяти для регистрации ситуации, когда текущая команда обратилась по адресу, содержащемуся в странице памяти, отсутствующей в данный момент времени в памяти. В такой ситуации в микропроцессоре возникает исключительная ситуация с номером 14, и линейный 32-битный адрес команды, вызвавшей это исключение, записывается в регистр cr2. Имея эту информацию, обработчик исключения 14 определяет нужную

страницу, осуществляет ее подкачку в память и возобновляет нормальную работу программы.

Регистр cr3 также используется при страничной организации памяти. Это так называемый регистр каталога страниц первого уровня. Он содержит 20-битный физический базовый адрес каталога страниц текущей задачи. Этот каталог содержит 1024 32-битных дескриптора, каждый из которых содержит адрес таблицы страниц второго уровня. В свою очередь каждая из таблиц страниц второго уровня содержит 1024 32-битных дескриптора, адресующих страничные кадры в памяти. Размер страничного кадра — 4 Кбайт.

Регистры системных адресов. Эти регистры еще называют регистрами управления памятью. Они предназначены для защиты программ и данных в мультизадачном режиме работы микропроцессора.

При работе в защищенном режиме микропроцессора адресное пространство делится:

- на глобальное — общее для всех задач;
- на локальное — отдельное для каждой задачи.

Этим разделением и объясняется присутствие в архитектуре микропроцессора следующих системных регистров (рис. 1.5):

- регистра таблицы глобальных дескрипторов gdtr (Global Descriptor Table Register), имеющего размер 48 бит и содержащего 32-битовый (биты 16–47) базовый адрес глобальной дескрипторной таблицы GDT и 16-битовое (биты 0—15) значение предела, представляющее собой размер в байтах таблицы GDT;
- регистра таблицы локальных дескрипторов ldtr (Local Descriptor Table Register) имеющего размер 16 бит и содержащего так называемый селектор дескриптора локальной дескрипторной таблицы LDT. Этот селектор является указателем в таблице GDT, который и описывает сегмент, содержащий локальную дескрипторную таблицу LDT;
- регистра таблицы дескрипторов прерываний idtr (Interrupt Descriptor Table Register), имеющего размер 48 бит и содержащего 32-битовый (биты 16–47) базовый адрес дескрип-

торной таблицы прерываний IDT и 16-битовое (биты 0—15) значение предела, представляющее собой размер в байтах таблицы IDT;

- 16-битового регистра задачи tr (Task Register), который, подобно регистру ldtr, содержит селектор, то есть указатель на дескриптор в таблице GDT. Этот дескриптор описывает текущий сегмент состояния задачи (TSS — Task Segment Status). Этот сегмент создается для каждой задачи в системе, имеет жестко регламентированную структуру и содержит контекст (текущее состояние) задачи. Основное назначение сегментов TSS — сохранять текущее состояние задачи в момент переключения на другую задачу.

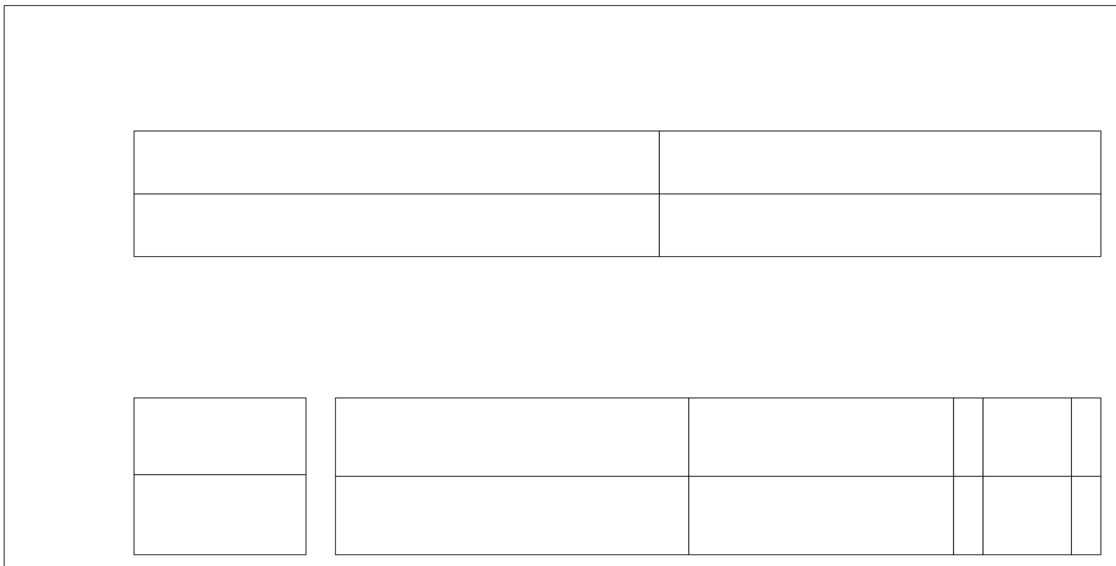


Рис. 1.5–Структура системных регистров адресов

Регистры отладки. Это группа регистров, предназначенных для аппаратной отладки. Средства аппаратной отладки впервые появились в микропроцессоре i486. Аппаратно микропроцессор содержит восемь регистров отладки, но реально из них используются только 6.

Регистры dr0, dr1, dr2, dr3 имеют разрядность 32 бит и предназначены для задания линейных адресов четырех точек прерывания. Используемый при этом механизм следующий: любой формируемый текущей программой адрес сравнивается с адресами в регистрах dr0...dr3, и при совпадении генерируется исключение отладки с номером 1.

Регистр dr6 называется регистром состояния отладки. Биты этого регистра устанавливаются в соответствии с причинами, которые вызвали возникновение последнего исключения с номером 1.

Перечислим эти биты и их назначение:

- b0 — если этот бит установлен в 1, то последнее исключение (прерывание) возникло в результате достижения контрольной точки, определенной в регистре dr0;
- b1 — аналогично b0, но для контрольной точки в регистре dr1;
- b2 — аналогично b0, но для контрольной точки в регистре dr2;
- b3 — аналогично b0, но для контрольной точки в регистре dr3;
- bd (бит 13) — служит для защиты регистров отладки;
- bs (бит 14) — устанавливается в 1, если исключение 1 было вызвано состоянием флага tf = 1 в регистре eflags;
- bt (бит 15) устанавливается в 1, если исключение 1 было вызвано переключением на задачу с установленным битом ловушки в TSS t = 1.

Все остальные биты в этом регистре заполняются нулями. Обработчик исключения 1 по содержимому dr6 должен определить причину, по которой произошло исключение, и выполнить необходимые действия.

Регистр dr7 называется регистром управления отладкой. В нем для каждого из четырех регистров контрольных точек отладки имеются поля, с помощью которых можно уточнить следующие условия, при которых следует сгенерировать прерывание:

- место регистрации контрольной точки — только в текущей задаче или в любой задаче. Эти биты занимают младшие восемь бит регистра dr7 (по два бита на каждую контрольную точку (фактически точку прерывания), задаваемую регистрами dr0, dr1, dr2, dr3 соответственно). Первый бит из каждой пары — это так называемое локальное разрешение; его установка говорит о том, что точка прерывания действует, если она находится в пределах адресного пространства те-

кущей задачи. Второй бит в каждой паре определяет глобальное разрешение, которое говорит о том, что данная контрольная точка действует в пределах адресных пространств всех задач, находящихся в системе;

- тип доступа, по которому инициируется прерывание: только при выборке команды, при записи или при записи/чтении данных. Биты, определяющие подобную природу возникновения прерывания, локализуются в старшей части данного регистра.

Вопросы для самопроверки

1. Какие способы представления сигналов существуют для хранения и передачи и обработки информации?
2. Назовите основную причину использования шестнадцатиричных чисел.
3. Дайте определение архитектуры ЭВМ.
4. Назовите общие архитектурные свойства и принципы ЭВМ.
5. В каких режимах может функционировать процессор Intel x86?
6. Расскажите о распределении адресного пространства в процессорах Intel x86.
7. Что использует процессор Intel x86 для доступа к внешним устройствам?
8. Дайте классификацию портов у процессора Intel x86.
9. Дайте определение программной модели микропроцессора.
10. Что входит в состав программной модели?
11. Для чего используются регистры общего назначения?
12. Почему не следует использовать регистр esp явно для хранения каких-либо операндов программы?
13. Какой регистр используют в качестве аккумулятора?
14. Какой регистр используют в качестве счетчика?
15. Какие регистры используют для поддержки циклических операций?
16. Какие регистры используются для работы со стеком?
17. Дайте классификацию сегментных регистров.
18. Какие регистры состояния и управления вы знаете?
19. Дайте классификацию флагов в регистре eflags.

20. Какой флаг отвечает за перенос в регистре eflags?
21. Какой флаг отвечает за переполнение в регистре eflags?
22. Какой флаг отвечает за знак в регистре eflags?
23. Какой флаг отвечает за определение нулевого значения в регистре eflags?
24. На какие группы делятся системные регистры?
25. Какой бит и в каком регистре отвечает за переключение микропроцессора в защищенный режим?
26. Для чего предназначены регистры gdtr, ldtr, idtr, tr?
27. Какую разрядность имеют регистры отладки?

2. Режимы функционирования процессора Intel x86

2.1 Реальный режим работы процессоров Intel x86

2.1.1 Управление памятью

После рестарта процессор находится в реальном режиме. Именно в реальном режиме функционирует базовая система ввода/вывода, загрузчики операционных систем, а также и некоторые операционные системы, например MS DOS.

В реальном режиме адрес имеет размер 20 бит, и, следовательно, максимальный объем адресуемой памяти составляет 1 Мб. Для формирования 20-битового адреса в памяти используются два 16-битовых регистра: сегментный регистр и регистр смещения (рис. 2.1).

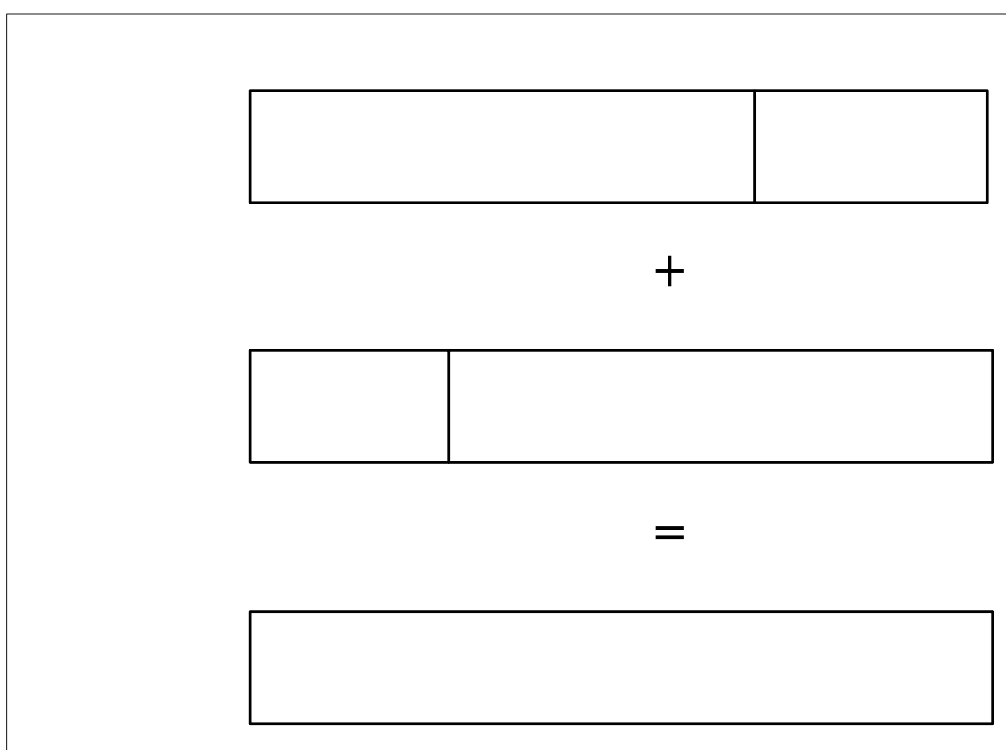


Рис. 2.1–Преобразование логического адреса в физический адрес в реальном режиме

Сегментный регистр определяет базовый адрес сегмента. В нем содержатся старшие 16 бит базового адреса сегмента, младшие четыре бита предполагаются равными 0. В регистре смещения находится смещение внутри сегмента (относительно базового адреса сегмента). Смещение может также задаваться непосредственной константой. Смещение добавляется к базовому адресу сегмента и таким образом получается 20 битовый адрес в памяти. Более просто можно описать вычисление следующим образом — 16-битовое значение сегментного регистра сдвигается на 4 бита влево и к нему добавляется значение смещения, в результате получается 20-битовое значение. Именно такая схема применялась в 8086 и 8088 процессорах.

Начиная с 80286 процессора для описания сегментов применяются теневые регистры (невидимая часть сегментных регистров). В них содержится адрес и предел (размер) сегмента. В реальном режиме предел всегда равен 64 Кб, а адрес сегмента вычисляется во время изменения видимой части сегментного регистра путем сдвига ее на 4-ре бита влево. Следует заменить, что, начиная с 80386 процессора, смещение может превышать 64 Кб за счет использования 32-х битовых методов адресации. Если смещение будет превышать предел, то произойдет исключение основной защиты.

2.2.2 Прерывания и исключения процессора

Прерывание — это ситуация, при которой выполнение основной программы приостанавливается, а управление передается на особую процедуру, называемую обработчиком прерывания.

Исключение — это прерывание, инициируемое самим процессором в ходе каких-либо внутренних ошибок (например, ошибка деления).

Для того чтобы изменить ход выполнения последовательности кодов в реальном режиме, необходимо изменить содержимое сразу двух регистров: регистра сегмента кода (CS) и указателя команды (IP). Значение, помещаемое в эти регистры, называется вектором прерывания.

В процессорах семейства Intel предусмотрено 256 прерываний и соответственно 256 векторов прерываний. Все вектора прерываний объединяются в таблицу, состоящую из 256 4-байтовых элементов и занимающую 1 Кб. В реальном режиме эта таблица находится в самом начале памяти по адресу 0:0 или (просто по 0-му физическому адресу) (рис. 2.2).

Каждый элемент таблицы состоит из двух полей. Первые два байта представляют собой значение, заносимое в регистр IP, последние два байта содержат значение, заносимое в регистр CS.

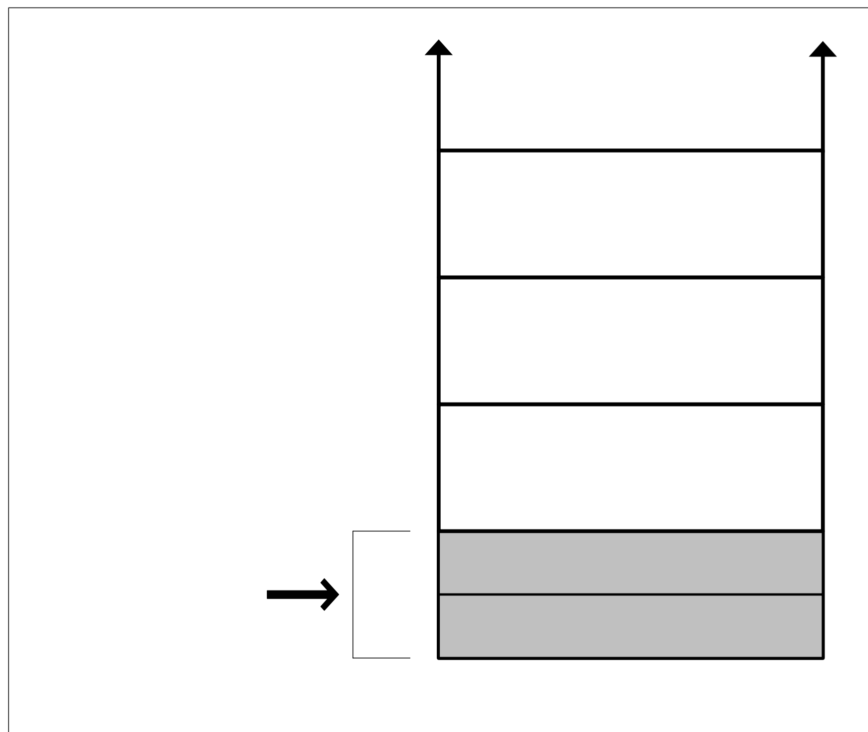


Рис. 2.2–Таблица векторов прерываний реального режима

Всего существует четыре вида прерываний: исключения, немаскируемое прерывание, маскируемые аппаратные прерывания и пользовательские прерывания. Исключения генерируются процессором. Аппаратные прерывания генерируются аппаратурой (клавиатуры, таймер). Пользовательские прерывания генерируются с помощью команды INT. Полный список прерываний и исключений реального режима приводится в табл. 2.1.

Таблица 2.1–Вектора прерывания реального режима

Номер прерывания	**Описание**
0h	Деление на 0 или переполнение операнда назначения
1h	Пошаговое прерывание отладчика
2h	Немаскируемое прерывание
3h	Точка останова
4h	Переполнение по команде BOUND
5h	Печать экрана на принтере
6h	Зарезервировано
7h	Зарезервировано
8h	Прерывание от таймера
9h	Прерывание от клавиатуры
0Ah-0Dh	Аппаратные прерывания
0Eh	Прерывание от контроллера гибких дисков
0Fh	Аппаратное прерывание
10h	Видео сервис
11h	Список оборудования
12h	Размер нижней памяти
13h	Работа с дисковой подсистемой
14h	Работа с последовательными портами
15h	Сервис AT
16h	Работа с клавиатурой
17h	Работа с принтером
18h	ROM BASIC
19h	Перезагрузка ОС
1Ah	Работа с таймером
1Bh	Реакция на Ctrl-Break
1Ch	Пользовательское прерывание от таймера
1Dh	Видео параметры
1Eh	Параметры дискеты
1Fh	Таблица графических символов
20h-2Fh	Прерывание DOS
33h	Работа с мышью
41h	Параметры жесткого диска 0
43h	Таблица символов EGA
46h	Параметры жесткого диска 1
50h	Прерывание от часов реального времени
67h	Функции расширенной памяти
	Неуказанные в этой таблице вектора могут использоваться дополнительно загружаемыми программами.

При возникновении прерывания процессор выполняет следующие действия:
1. Помещает в стек значение регистра IP.
2. Помещает в стек значение регистра CS.
3. Помещает в стек значение регистра FLAGS.
4. Загружает из таблицы векторов прерываний в соответствии с номером прерывания регистры IP и CS.
5. Запрещает аппаратные прерывания.

При выходе из прерывания процессор выполняет следующие действия:
1. Загружает из стека регистр FLAGS.
2. Загружает из стека регистр CS.
3. Загружает из стека регистр IP.

2.2.3 Взаимодействие с базовой системой ввода/вывода

В реальном режиме операционной системе и программам пользователя доступны функции базовой системы ввода/вывода, находящейся в ПЗУ. Вызов функций осуществляется посредством вызова прерываний. Для базовой системы ввода/вывода отводятся прерывания в диапазоне 10h-1Fh. Все необходимые данные помещаются в регистры общего назначения, затем происходит вызов функции по команде int n. После выполнения функции возвращаемые данные находятся в регистрах или памяти.

В данном пункте рассмотрим только три основных сервиса.

Видео сервис. Функции видео сервиса вызываются через 10h прерывание. Далее приводится описание некоторых функций [2].

Установка видеорежима

AH=0h; AL= видеорежим (табл. 2.2)

Установка позиции курсора

AH=2h; BH=видео страница; DH, DL = строка, колонка.

Выбор текущей видеостраницы

AH=5h; AL= номер страницы.

Чтение символа/атрибута из текущей позиции курсора

AH=8h; BH=видеостраница.

Возвращает: AL=код символа, AH=атрибут

Запись символа/атрибута в текущую позицию курсора

AH=9h; BH=видеостраницы; AL=код символа; CX=число символов; BL=атрибут.

Запись графической точки

AH=0Dh; BH=видеостраница; DX,CX=строка, колонка; AL=цвет

Чтение графической точки

AH=0Dh; BH=видеостраница; DX,CX=строка, колонка

Возвращает: AL=цвет

Запись символа в режиме телетайпа

AH=0Eh; AL=код символа; BL=цвет (для графических режимов)

Таблица. 2.2–Видео режимы и их характеристики

AL	**Тип**	**Формат**	**Ячей-ка**	**Цвета**	**Адаптер**	**Адрес**
0	Текст	40x25	8x8*	16/8	CGA, EGA	B800
1	Текст	40x25	8x8*	16/8	CGA, EGA	B800
2	Текст	80x25	8x8*	16/8	CGA, EGA	B800
3	Текст	80x25	8x8*	16/8	CGA, EGA	B800
4	Графика	320x200	8x8	4	CGA, EGA	B800
5	Графика	320x200	8x8	4	CGA, EGA	B800
6	Графика	640x200	8x8	2	CGA, EGA	B800
7	Текст	80x25	9x14*	3	MA, EGA	B000
8, 9, 0Ah	Режимы PCjr					
0Bh, 0Ch	Зарезервировано (внутренние для EGA BIOS)					
0Dh	Графика	320x200	8x8	16	EGA, VGA	A000
0Eh	Графика	640x200	8x8	16	EGA, VGA	A000
0Fh	Графика	640x350	8x14	3	EGA, VGA	A000
10h	Графика	640x350	8x14	1 или16	EGA, VGA	A000
11h	Графика	640x480	8x16	2	VGA	A000
12h	Графика	640x480	8x16	16	VGA	A000
13h	Графика	320x200	8x16	256	VGA	A000

Дисковый ввод/вывод. Функции дискового сервиса вызываются через 13h прерывание. Далее приводится описание некоторых функций [2].

Сброс устройства

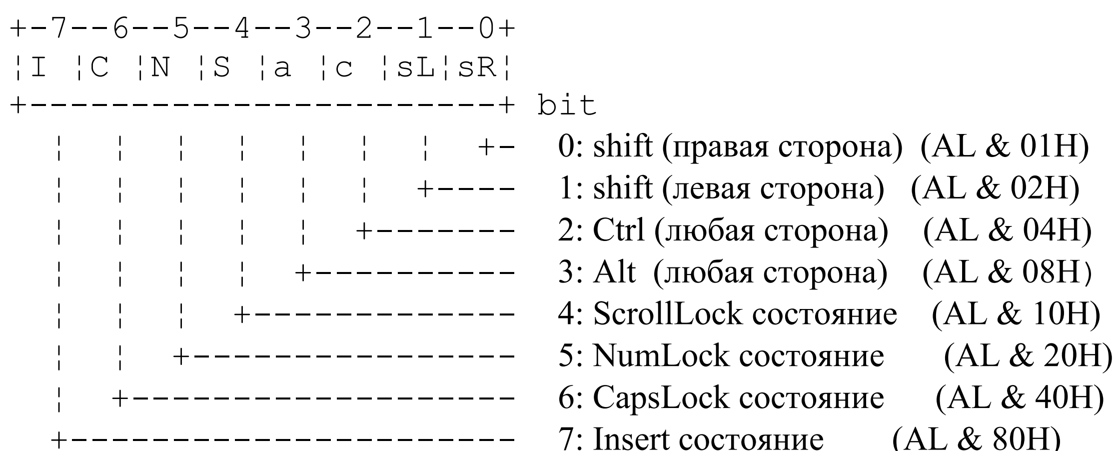
AH=0; DL=устройство (0,1,2 … -дисководы, 80h, 81h, 82h … жесткие диски).

Получение статуса ошибки после последней операции

AH=1; DL=устройство.
Возвращает: AL=код ошибки.
Чтение секторов
AH=2; DL=устройство; CH+два старших бита CL=номер цилиндра; CL=номер сектора; AL=число секторов; ES:BX=>адрес буфера.
Возвращает: cf=0 – нет ошибки; cf=1 – ошибка, AL=код ошибки
Запись секторов
AH=3; DL=устройство; CH+два старших бита CL=номер цилиндра; CL=номер сектора; AL=число секторов; ES:BX=>адрес буфера.
Возвращает: cf=0 – нет ошибки; cf=1 – ошибка, AL=код ошибки
Проверка секторов
AH=4; DL=устройство; CH+два старших бита CL=номер цилиндра; CL=номер сектора; AL=число секторов; ES:BX=>адрес буфера.
Возвращает: cf=0 – нет ошибки; cf=1 – ошибка, AL=код ошибки.

Ввод/вывод с клавиатуры. Функции сервиса клавиатуры вызываются через 16h прерывание. Далее приводится описание некоторых функций [2].

Чтение символа из буфера клавиатуры (ожидание, если символа нет)
AH=0.
Возвращает: AL=код символа; AH=код сканирования
Проверка наличия символа в буфере клавиатуры
AH=1.
Возвращает: zf=1 – символа нет; zf=0 – символ есть, AL=код символа; AH=код сканирования
Чтение состояния переключающих клавиш
AH=2.
Возвращает: AL=биты статуса



Поместить символ в буфер клавиатуры
AH=5; CL=код символа; CH=код сканирования.
Возвращает: AL=статус: 0=успешное завершение, 1=буфер полон.

2.2 Защищенный режим работы процессоров Intel x86

2.2.1 Управление памятью

32-битовые процессоры Intel обеспечивают значительную поддержку для операционных систем и программного обеспечения для системной разработки. Эта поддержка – часть системной архитектуры и включает возможности для помощи в организации следующих операций [3]:

- Управление памятью.
- Защита программных модулей.
- Многозадачность.
- Обработка прерываний и исключений.
- Управление кэшированием.
- Управление аппаратными ресурсами и питанием.
- Отладка и отслеживание производительности.

Системная архитектура Intel состоит из набора регистров, структур данных и инструкций, разработанных для обеспечения базовых операций системного уровня, таких как управление памятью, обработка исключений и прерываний, управление задачами, управление многопроцессорными системами. На рис. 2.3 изображены регистры и структуры данных [3].

Управление памятью в архитектуре Intel делится на две части: сегментация и трансляция страниц. Сегментация предоставляет механизм для изолирования индивидуального кода, данных и стека. Таким образом, несколько программ могут выполняться одновременно на одном процессоре, не перекрывая друг друга. Трансляция страниц предоставляет механизм для реализации виртуальной памяти с подкачкой страниц по запросу, где части программы отображаются на физическую память как необходимо. Трансляция страниц так же может использоваться для изоляции между несколькими задачами. В защищенном режиме обязательно должна использоваться какая-нибудь форма сегментации.

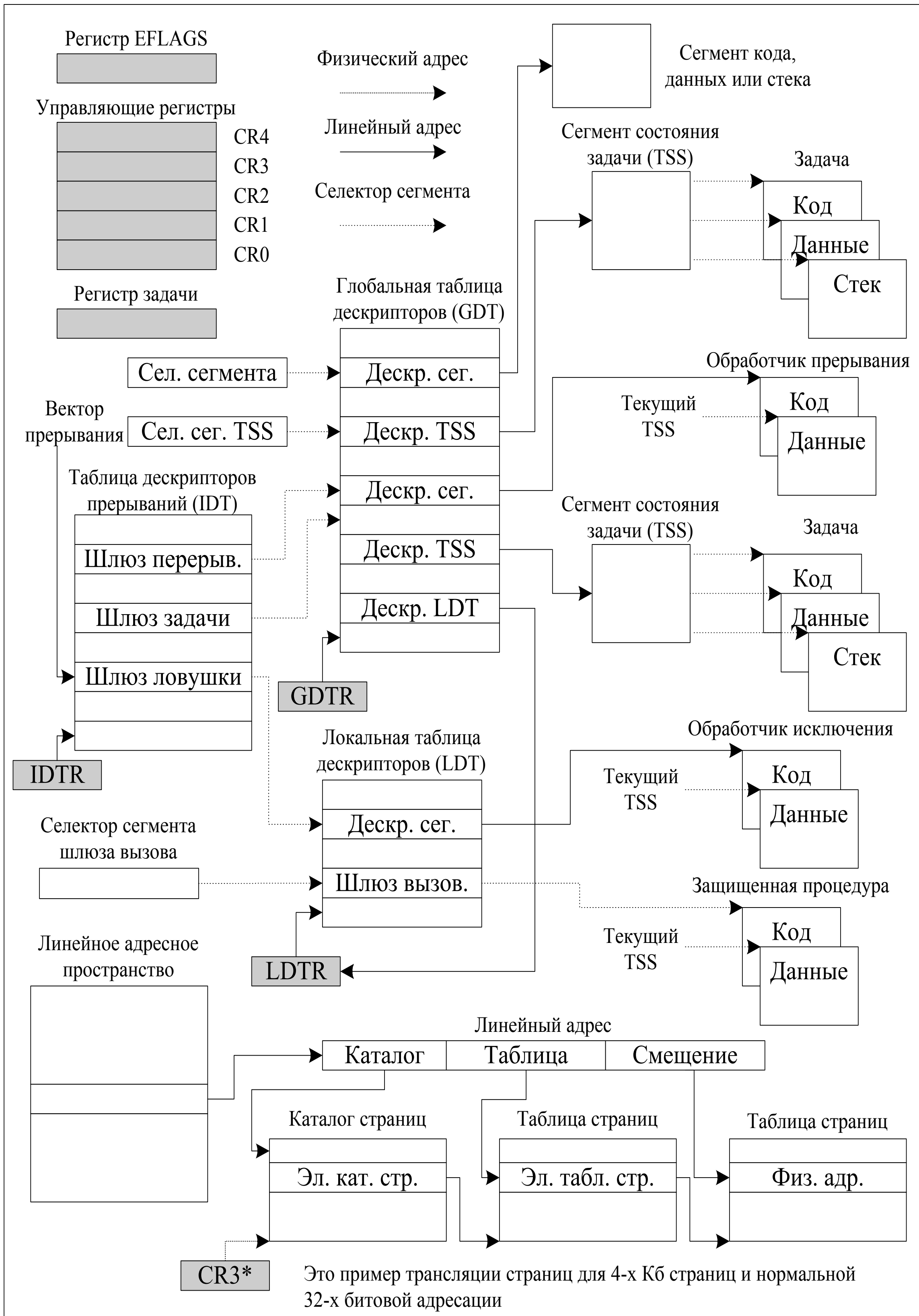


Рис. 2.3–Регистры и структуры данных, используемые в защищенном режиме

Как показано на рис. 2.4, сегментация обеспечивает механизм для разделения процессорного адресного пространства (называемого линейным пространством) на меньшие защищенные адресные пространства. Сегменты могут использоваться для содержания кода, данных, стека и системных структур данных.

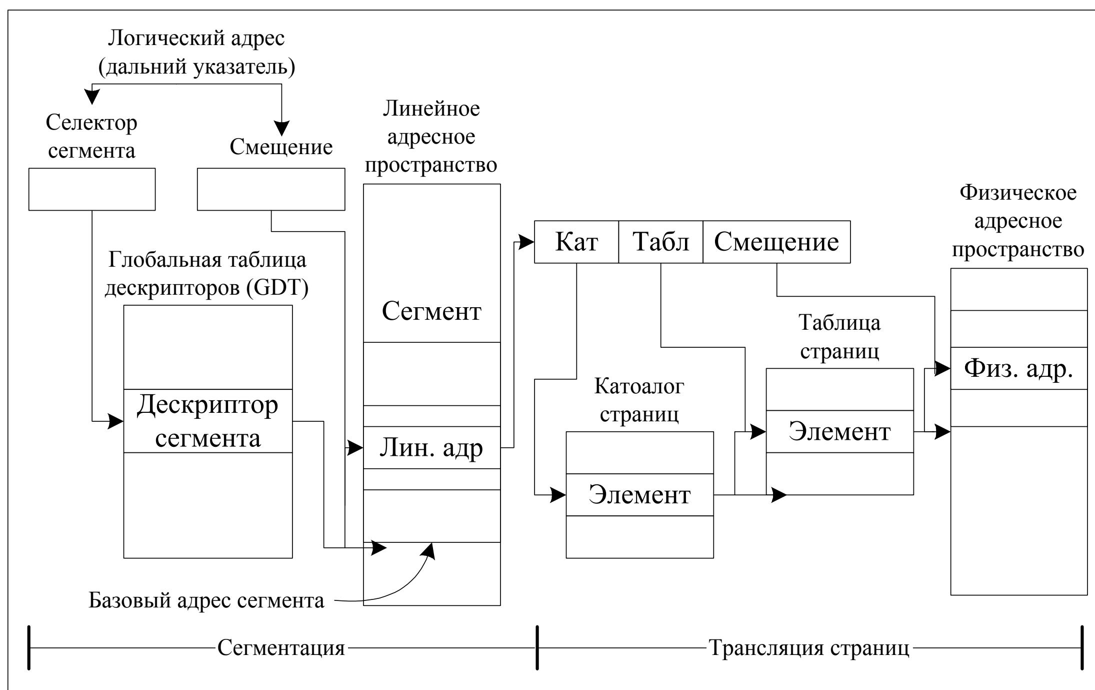


Рис. 2.4–Сегментация и трансляция страниц

Все сегменты содержатся внутри линейного адресного пространства процессора. Для доступа к любому байту информации в каком-либо сегменте необходим логический адрес (иногда его называют дальним указателем). Логический адрес состоит из селектора сегмента и смещения. Селектор сегмента — это уникальный идентификатор сегмента. Селектор сегмента также является смещением в таблице дескрипторов. Каждый сегмент имеет дескриптор сегмента, который определяет размер, права доступа, уровень привилегий, тип сегмента и расположение его первого байта внутри линейного пространства. Смещение добавляется к базовому адресу сегмента для доступа к конкретному байту внутри сегмента. Базовый адрес плюс смещение формируют линейный адрес внутри линейного адресного пространства процессора.

Если трансляция страниц не используется, то линейное адресное пространство непосредственно отображается на физическое адресное пространство процессора. Физическое адресное пространство определяется как диапазон адресов, которые процессор может генерировать на шине адреса.

В многозадачных операционных системах линейное пространство гораздо больше, чем пространство физической памяти, поэтому необходим метод виртуализации линейного адресного пространства. Эта виртуализация линейного адресного пространства производится через механизм трансляции страниц.

Трансляция страниц поддерживает среду с виртуальной памятью, где большое линейное пространство эмулируется за счет маленького количества физической памяти и некоторого количества дисковой памяти. При этом все сегменты делятся на 4-килобайтовые страницы, которые могут находиться либо в памяти или на диске. Когда задача пытается обратиться к странице, которой нет в памяти, происходит исключение и операционная система загружает недостающую страницу в память, после чего выполнение задачи возобновляется.

Для более экономного использования памяти используется двухуровневая схема трансляции страниц. На верхнем уровне находится каталог страниц, который содержит ссылки на таблицы страниц. Таблицы страниц содержат ссылки на сами страницы.

Трансляция страниц — это единственный способ, при котором возможно выполнение нескольких задач реального режима.

Механизм сегментации, поддерживаемый в архитектуре Intel, может быть использован для реализации различных системных разработок от минимальной Flat модели до строгой операционной среды с множеством задач, которые могут надежно выполняться [3].

Базовая Flat модель. Самая простая модель – базовая Flat модель, в которой операционная система и приложения имеют доступ к непрерывному несегментированному адресному пространству. Эта базовая модель прячет механизм сегментации от системного разработчика и прикладного программиста.

Для реализации этой модели необходимо, по крайней мере, два дескриптора сегмента. Один — для ссылки на сегмент кода, другой — для сегмента данных. Тем не менее, два этих сегмента отображаются на все адресное пространство (рис. 2.5).

Защищенная Flat-модель. Защищенная Flat-модель похожа на базовую модель, за исключением того, что пределы сегментов выставляются так, чтобы включить только пространство адресов физически существующей памяти (рис. 2.6). Эта модель предоставляет минимальный уровень аппаратной защиты против программных ошибок. Эта модель может использоваться совместно с трансляцией страниц для разделения различных задач.

Многосегментная модель. Многосегментная модель (рис. 2.7) использует все возможности механизма сегментации для обеспечения аппаратной защиты кода, структур данных, программ и задач. В данном случае каждая программа имеет свою таблицу дескрипторов сегментов и свои сегменты. Сегменты могут принадлежать одной программе, а могут и разделяться между несколькими программами. Доступ ко всем сегментам контролируется на аппаратном уровне. Проверка доступа может использоваться не только для защиты от ссылок, выходящих за границы сегмента, а   также для защиты от выполнения операций в некоторых сегментах. Информация прав доступа может использоваться для назначения колец защиты.

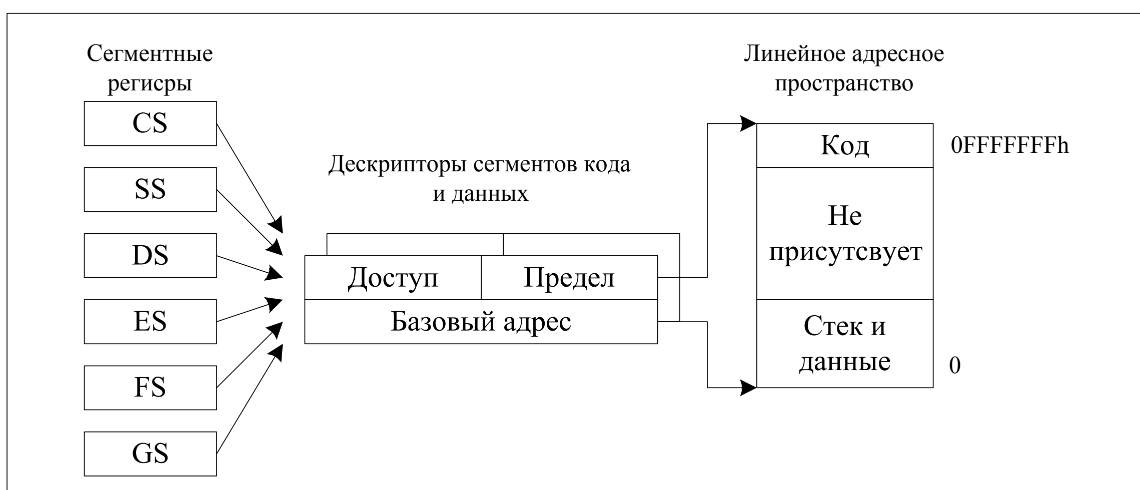


Рис. 2.5–Flat-модель

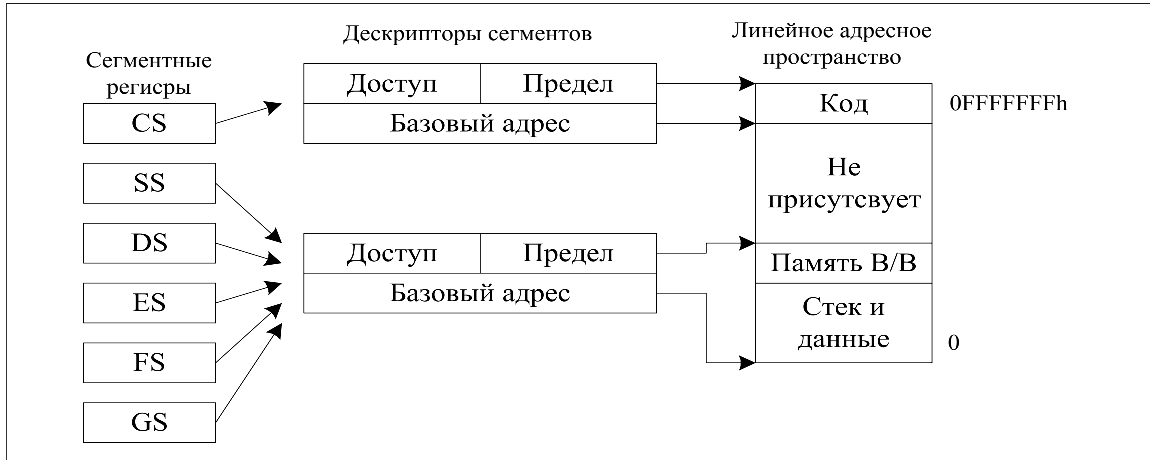


Рис. 2.6–Защищенная Flat-модель

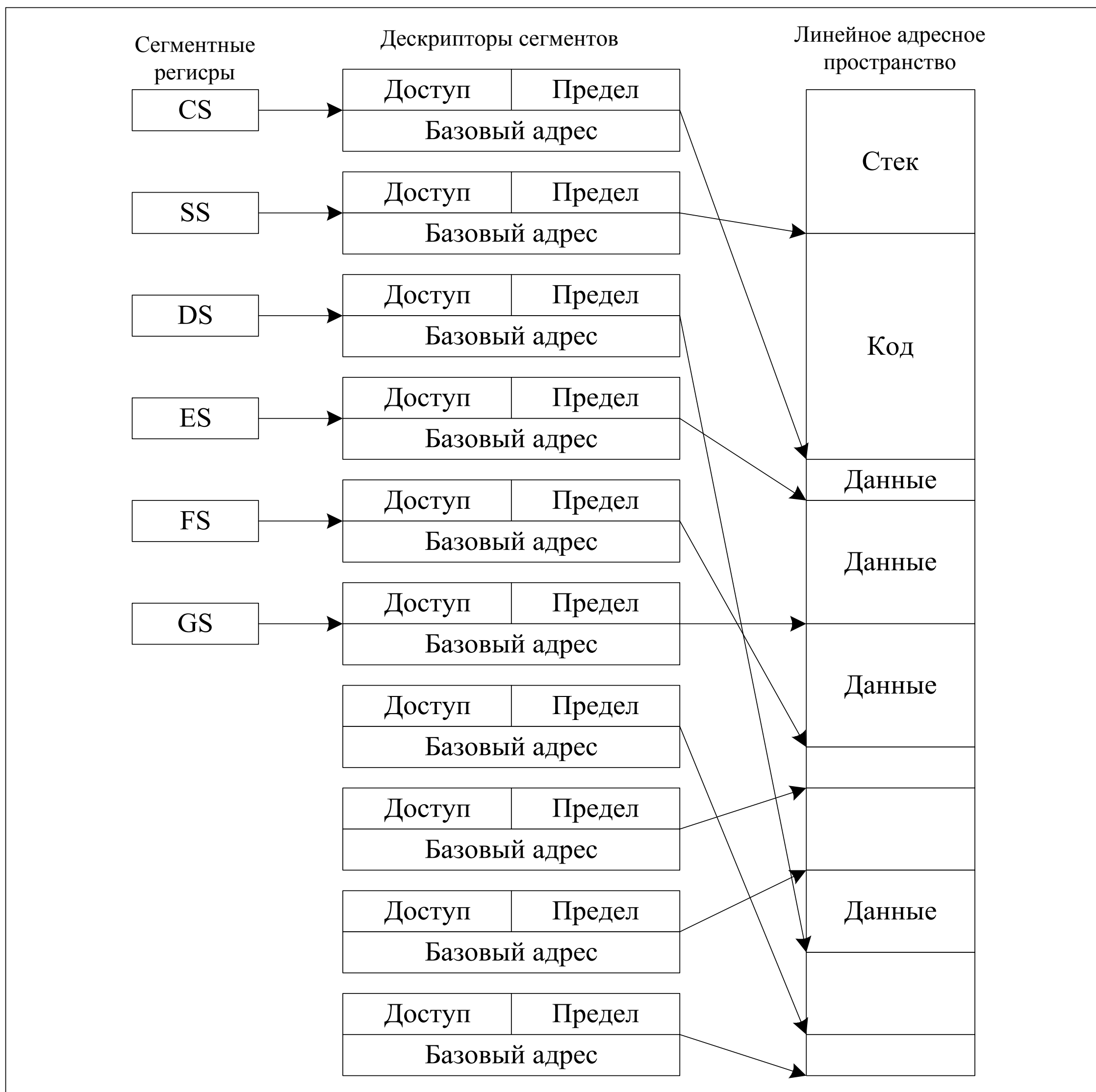


Рис. 2.7–Многосегментная модель

Трансляция страниц может использоваться с любой моделью сегментации. Механизм трансляции страниц делит линейное адресное пространство на 4 Кб страницы. Затем страницы линейного пространства отображаются на физическую память. Этот механизм так же предоставляет дополнительные возможности защиты.

В защищенном режиме архитектура Intel предоставляет нормальное ***физическое адресное пространство*** до 4 Гб. Это адресное пространство, которое процессор может адресовать на его адресной шине. Начиная с процессора Pentium Pro, архитектура Intel предоставляет адресное пространство 64 Гб.

Логические и линейные адреса. На уровне системной архитектуры в защищенном режиме процессор использует два этапа преобразования адресов для достижения физических адресов: преобразование логических адресов и страничную трансляцию линейного адресного пространства.

Даже с минимальным использованием сегментов каждый байт процессорного адресного пространства определяется логическим адресом. Логический адрес состоит из 16-битового селектора и 32-битового смещения (рис. 2.8). Селектор определяет сегмент, где расположен байт, а смещение определяет положение байта в сегменте, относительного базового адреса сегмента.

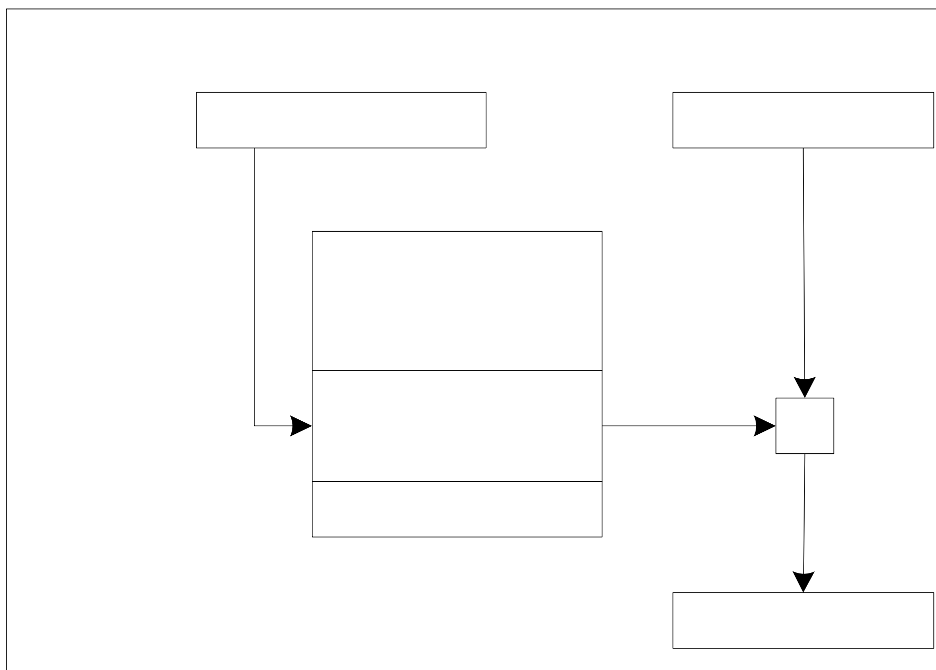


Рис. 2.8–Преобразование логического адреса в линейный

Процессор транслирует каждый логический адрес в линейный адрес. Линейный адрес — это 32-битовый адрес в процессорном линейном адресном пространстве. Как и физическое пространство линейное пространство является несегментированным, 2^32 байтовым адресным пространством с диапазоном от 0 до 0FFFFFFFFh. Линейное адресное пространство содержит все сегменты и системные таблицы для системы.

Для трансляции логического адреса в линейный процессор выполняет следующие действия [3]:

1. Использует селектор сегмента как смещение в таблицах GDT или LDT для определения дескриптора сегмента и считывает последний в процессор.
2. Делает проверки прав доступа к сегменту и попадание в диапазон адресов сегмента по полю предел.
3. Добавляет базовый адрес сегмента к смещению для формирования линейного адреса.

Селектор сегмента — это 16-битовый идентификатор сегмента. Он не указывает напрямую на сегмент, а вместо этого он определяет сегмент. Селектор сегмента содержит следующие поля:

- **Index** (биты с 3 по 15). Выбирает один из 8192 дескрипторов из GDT или LDT. Процессор умножает значение индекс на 8 и добавляет результат к базовому адресу GDT или LDT (из регистров GDTR или LDTR соответственно).
- **TI** (бит 2). Определяет таблицу дескрипторов для использования. Нулевое состояние определяет GDT, единичное состояние определяет LDT.
- **RPL** (биты 0 и 1). Указывает уровень привилегий селектора. Диапазон привилегий — от 0 до 3, где 0-й — самый привилегированный.

Первый элемент GDT не используется процессором. Селектор сегмента, который на него указывает, содержит нулевое значение во всех битах и используется как пустой селектор сегмента. Процессор не генерирует исключения при загрузке его в сегментный регистр кроме CS и DS, тем не менее, он генерирует исключение при попытке обращения к такому сегменту.

Чтобы уменьшить время трансляции и сложность кодирования, в процессоре используются регистры для шести селекторов. Каждый из этих регистров содержит селектор сегмента определенного типа: CS — код, SS — стек, DS — данные, ES — данные, FS — данные, GS — данные.

Для доступа к сегменту программа должна загрузить селектор этого сегмента в один из сегментных регистров. В программе может присутствовать неопределенное число сегментов, но непосредственно она может обращаться только к шести.

Каждый сегментный регистр имеет видимую и скрытую часть (невидимая часть еще называется кэшем дескриптора или теневым регистром). Когда селектор сегмента загружается в видимую часть регистра, процессор также загружает и скрытую часть регистра базовым адресом, пределом и информацией доступа из дескриптора, указанного селектором.

Дескриптор сегмента — это структура данных в GDT или LDT, которая предоставляет процессору размер, положение и информацию доступа и статуса. На рис. 2.9 изображен общий формат дескриптора для всех типов сегментов.

Рис. 2.9–Формат дескриптора сегмента

В дескрипторе сегмента определены следующие поля [3]:

- **Segment Limit** — Определяет размер сегмента. Процессор объединяет вместе два поля предела для получения 20-битового значения. Процессор интерпретирует это поле в зависимости от состояния флага гранулярности G.
    - Если бит G=0, то предел сегмента находится в диапазоне от 1 байта до 1 Мб.
    - Если бит G=1, то предел сегмента находится в диапазоне от 4 Кб до 4 Гб, но на границе 4 Кб.

  Процессор использует предел сегмента двумя разными путями в зависимости от того, является ли этот сегмент расширяемым вверх или вниз. Для сегментов расширяющихся вверх смещение в логическом адресе может находиться в диапазоне от 0 до предела сегмента. Для расширяющихся вниз сегментов смещение в логическом адресе может находиться в диапазоне от предела сегмента до 0FFFFh или 0FFFFFFFFh в зависимости от флага B. Стек в архитектуре Intel всегда растет вниз. Расширяющийся вниз сегмент – удобный механизм для создания расширяемых стеков.
- **Base address** — Определяет базовый адрес сегмента в линейном пространстве. Процессор объединяет вместе все три поля и получает 32-битовое значение.
- **Type** — Определяет тип сегмента. Интерпретация этого поля зависит от того, указывает ли флаг типа дескриптора на то, что это дескриптор приложения (код или данные) или системный дескриптор. Значение поля Type различно для кодового и системного дескрипторов и дескриптора данных.
- **S** — Определяет тип дескриптора – либо системный дескриптор (S=0), либо сегмент данных или кода (S=1).
- **DPL** — Определяет уровень привилегий сегмента. Значение поля может быть в диапазоне от 0 до 3. 0-й уровень самый привилегированный.

P — Указывает присутствие сегмента в памяти. P=1 — сегмент присутствует, P=0 — сегмент отсутствует. При попытке обра-

щения к отсутствующему сегменту происходит исключение процессора.

D/B — Выполняет различные функции в зависимости от типа дескриптора.

- o Сегмент кода. Этот флаг называется D флагом и определяет длину операнда и адреса по умолчанию. Если D=1, то 32-битовые операнды подразумеваются, если D=0, то 16-битовые операнды подразумеваются.
- o Сегмент стека. Этот флаг называется B флаг и определяет размер указателя стека. Если B=1, то используется ESP, если B=0, то используется SP.
- o Расширяющийся вниз сегмент. Этот флаг называется B флаг и определяет верхнюю границу сегмента. Если B=1, то верхняя граница — FFFFFFFFh, если B=0, то 0FFFFh.

- **G** — Определяет единицу измерения поля предела сегмента: в байтах или страницах (4 Кб).

Когда флаг S=1, дескриптор является системным дескриптором. Процессор распознает следующие типы системных дескрипторов [3]:

- Дескриптор сегмента локальной таблицы дескрипторов (LDT).
- Дескриптор сегмента состояния задачи (task state descriptor).
- Дескриптор шлюза вызова (call-gate descriptor).
- Дескриптор шлюза прерывания (interrupt-gate descriptor).
- Дескриптор шлюза ловушки (trap-gate descriptor).
- Дескриптор шлюза задачи (task-gate descriptor).

Эти типы дескрипторов делятся на две категории: дескрипторы системных сегментов и дескрипторы шлюзов. Дескрипторы системных сегментов указывают на системные сегменты (LDT и TSS сегменты). Дескрипторы шлюзов — сами по себе «шлюзы», которые содержат указатели на процедуры в кодовых сегментах или которые содержат селекторы для TSS.

Таблица дескрипторов сегментов — это массив дескрипторов сегментов (рис. 2.10). Таблица дескрипторов имеет переменную длину и может содержать 8192 дескриптора. Имеются две таблицы дескрипторов: глобальная таблица дескрипторов и локальная таблица дескрипторов.

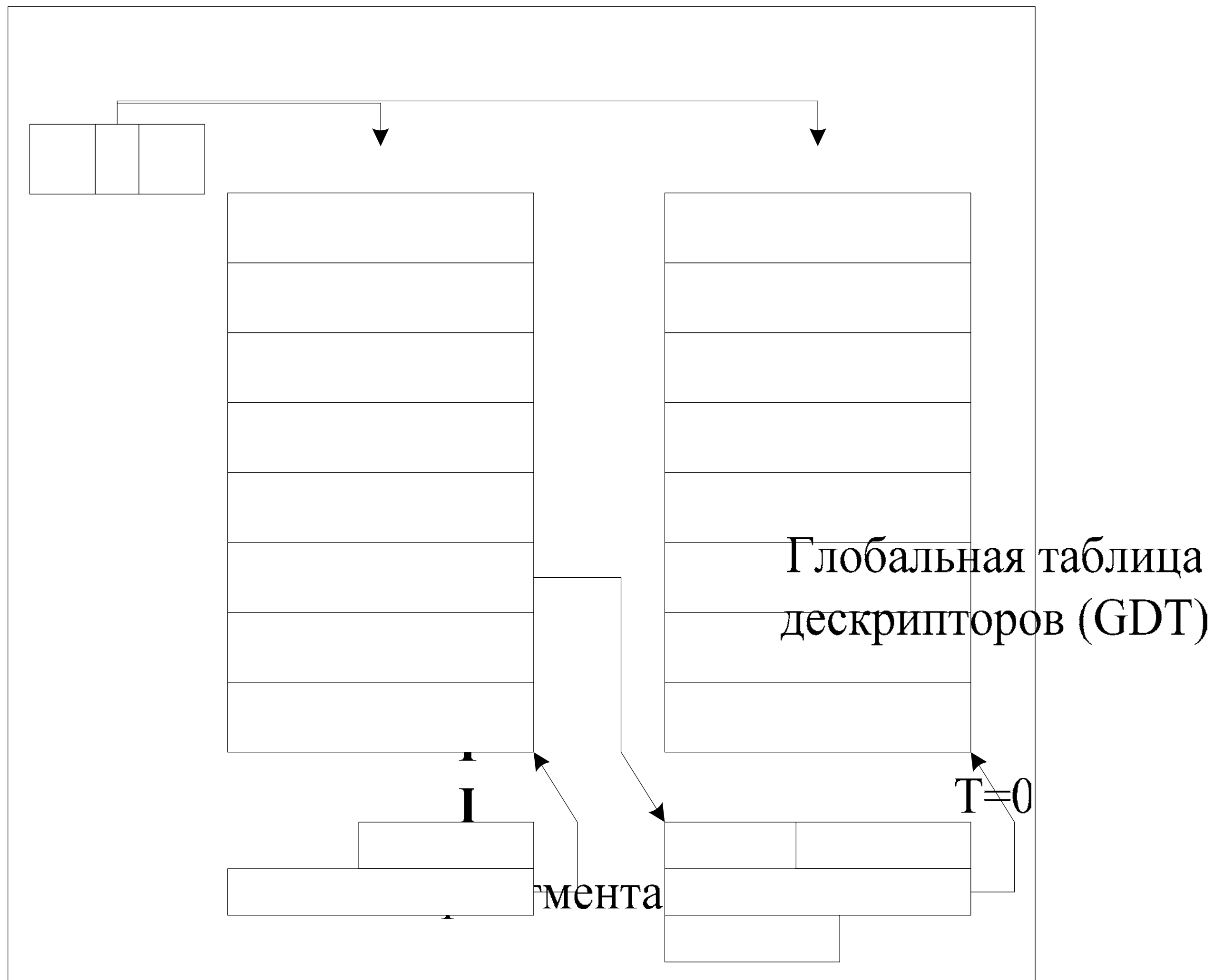


Рис. 2.10–Глобальная и локальная таблицы дискрипторов

Каждая система должна иметь одну GDT, которая может использоваться для всех программ и задач в системе. Наличие одной или более LDT опционально.

GDT — это не сегмент, а структура данных в линейном пространстве. Базовый линейный адрес и предел GDT должен быть загружен в регистр GDTR. Для максимальной производительности базовый адрес должен быть выровнен на границе 8 байт. Значение

поля предел обязательно должно быть на 1 меньше размера сегмента (8N-1).

Трансляция страниц виртуальной памяти. В защищенном режиме архитектура Intel позволяет отобразить линейное адресное пространство напрямую на физическое пространство или косвенно (используя механизм трансляции страниц) на небольшое количество физической памяти и часть дискового пространства [3].

Когда используется трансляция страниц, процессор делит линейное пространство на страницы физического размера (обычно 4 Кб). Когда программа обращается к памяти по логическому адресу, процессор преобразовывает линейный адрес в соответствующий физический. Если страница, содержащая линейный адрес, отсутствует в физической памяти, процессор генерирует исключение по отсутствию страницы. Обработчик исключения обычно указывает операционной системе о необходимости загрузить эту страницу в память. Когда страница загружена в физическую память, возврат из обработчика прерывания перезапускает инструкцию, которая вызвала исключение. Информация, которую процессор использует для отображения линейного адресного пространства в физическое, содержится в каталоге страниц и в таблицах страниц, хранимых в памяти.

Трансляция страниц управляется тремя флагами в регистрах процессора:

- PG флаг, бит 31 в CR0 (доступен во всех процессорах начиная с 80386). PG флаг включает механизм трансляции страниц.
- PSE флаг, бит 4 в CR4 (представлен в Pentium и Pentium Pro). Включает страницы большого размера: 4 Мб или 2 Мб (когда флаг PAE установлен).
- PAE флаг, бит 5 в CR4 (представлен в Pentium и Pentium Pro). Включает 36-битовую адресацию физической памяти.

Информация, которую процессор использует для трансляции линейного адресного пространства в физическое адресное пространство, содержится в четырех структурах данных:

- Каталог страниц — Массив 32 битовых элементов каталога страниц, содержащихся в 4-килобайтовой странице. До 1024 элементов может содержаться в каталоге страниц.

- Таблица страниц — Массив 32 битовых элементов таблицы страниц. До 1024 элементов может содержаться в таблице страниц.
- Страница — 4 Кб, 2Мб или 4Мб-непрерывное адресное пространство.
- Таблица указателей на каталоги страниц — Массив из 4-х 64-битовых элементов, каждый из которых указывает на каталог страниц.

Эти таблицы обеспечивают доступ либо к 4 Кб, либо 4Мб страницам, когда используется обычная 32 битовая адресация, либо 4Кб либо 2Мб страницы при использовании 36 битовой адресации.

На рис. 2.11 показана иерархия каталога страниц и таблиц страниц при трансляции 4 Кб страниц. Элементы каталога страниц указывают на таблицы страниц, а элементы в таблице страниц указывают на страницы в физической памяти. Этот механизм трансляции страниц может использоваться для адресации до 2^20 страниц.

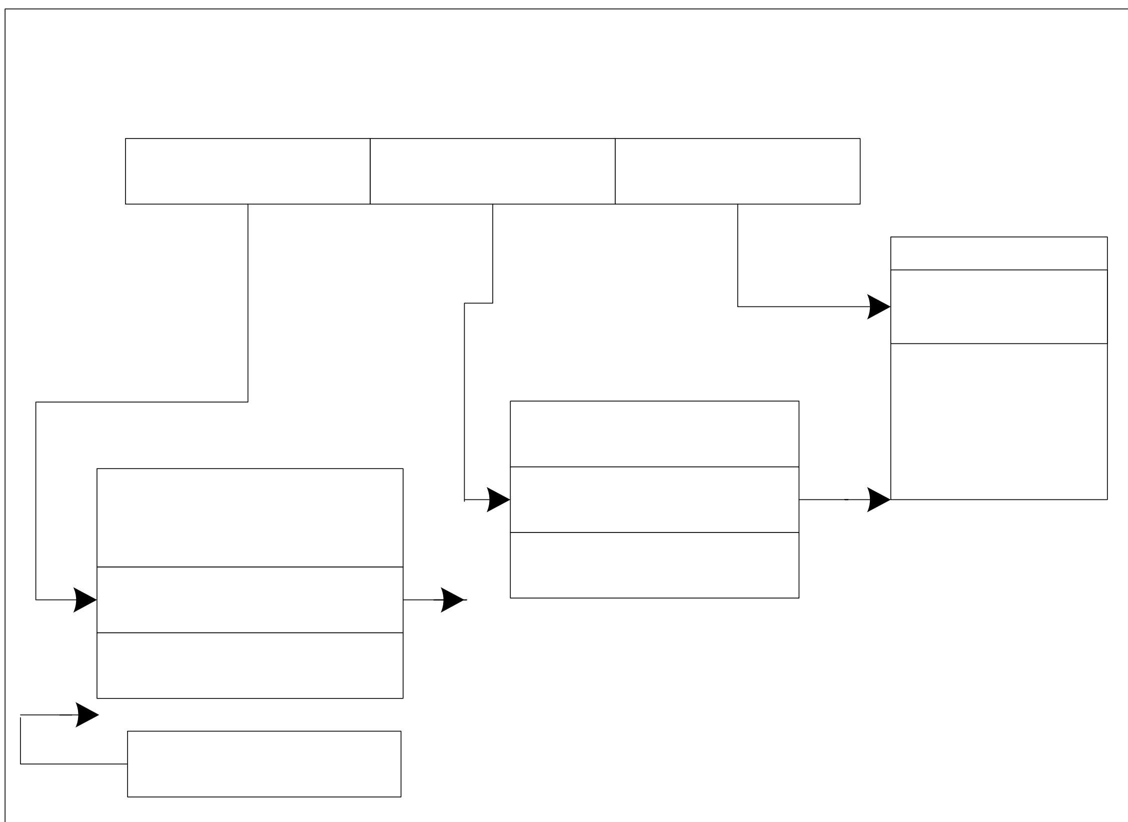


Рис. 2.11–Преобразование линейных адресов (4 Кб страницы)

Для того чтобы выбирать различные элементы таблицы, линейный адрес разделяется на три секции:

Линейный ад

31 22 21

- Элемент каталога страниц — биты с 22 по 31 определяют смещение элемента в каталоге страниц. Выбранный элемент содержит базовый физический адрес таблицы страниц.
- Элемент таблицы страниц — биты с 12 по 21 линейного адреса определяют смещение элемента в выбранной таблице страниц. Этот элемент содержит базовый физический адрес страницы в физической памяти.
- Смещение в странице — биты с 0 по 11 определяют смещение на физический адрес в странице.

На рис. 2.12 показано, как каталог страниц может использоваться для отображения линейных адресов в 4 Мб страницы. Элементы каталога страниц указывают на 4 Мб страницы в физической памяти. Этот метод трансляции страниц может использоваться для отображения до 1024 страниц в 4 Гб линейного адресного пространства.

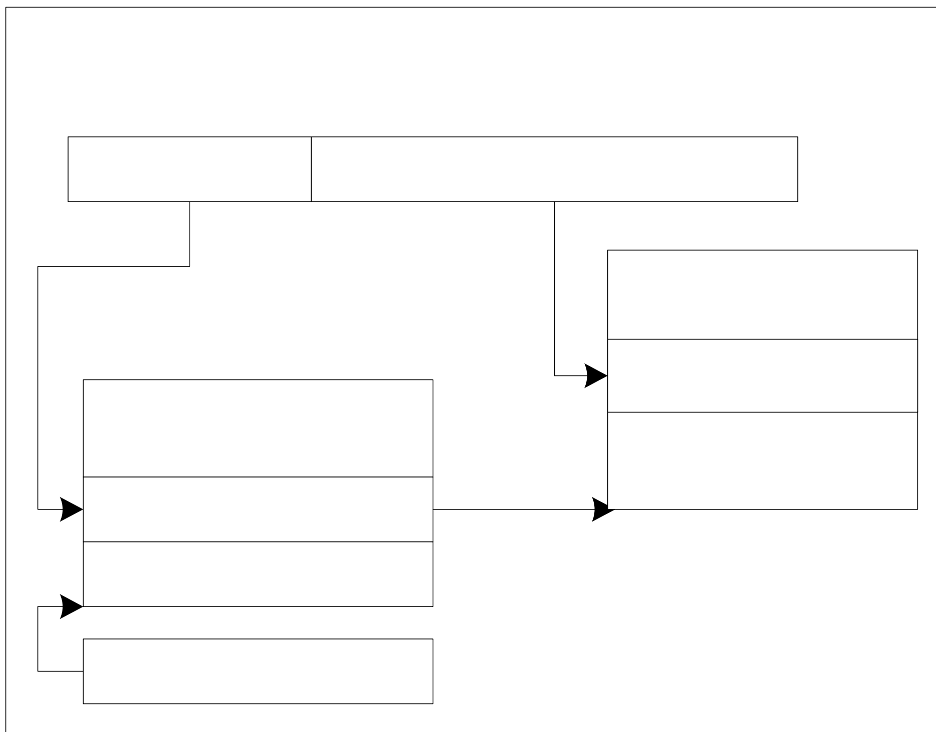


Рис. 2.12–Преобразование линейных адресов (4 Мб страницы)

Физический адрес текущего каталога страниц хранится в регистре CR3 (PDBR). Если предполагается использовать трансляцию страниц, то регистр PDBR должен быть загружен во время процесса инициализации. Затем регистр PDBR может быть

Линейный ад

3 2 2

1 2 1

изменен загрузкой нового значения в регистр CR3 или во время переключения задач.

2.2.2 Управление процессором

Программная инициализация для защищенного режима. После аппаратного сброса процессор находится в реальном режиме. В этом месте процесса инициализации несколько базовых структур данных и модулей кода должны быть загружены в физическую память для обеспечения дальнейшей инициализации процессора. Перед переключением процессора в защищенный режим программа инициализации должна загрузить минимальное число структур защищенного режима для обеспечения надежной работы процессора в защищенном режиме. Эти структуры данных включают следующие [3]:

- IDT защищенного режима.
- GDT.
- TSS.
- LDT (опционально).
- Каталог страниц и одна или несколько таблиц страниц (если используется трансляция страниц)
- Один или более модулей кода, содержащих необходимые обработчики прерываний и исключений.

Программа инициализации должна также загрузить следующие регистры:

- GDTR.
- (опционально) IDTR. Может быть загружен после переключения в защищенный режим.
- CR1 — CR4.
- Регистры типа диапазонов адресов (только для Pentium Pro).

После инициализации вышеперечисленных структур и регистров процессор может переключиться в защищенный режим посредством установки флага PE в регистре CR0.

Системные структуры данных защищенного режима. Содержимое системных структур защищенного режима зависит от типа управления памятью в операционной системе: flat, flat с трансляцией страниц, сегментированный, сегментированный с трансляцией страниц.

Для реализации модели памяти Flat без трансляции страниц программа инициализации должна, по крайней мере, загрузить GDT с одним кодовым сегментом и одним сегментом данных. Нулевой дескриптор, тем не менее, необходим. Модель памяти Flat с трансляцией страниц требует еще каталог страниц и хотя бы одну таблицу страниц.

В многосегментной модели может потребоваться дополнительные сегменты, как для операционной системы, так и сегменты и LDTs для каждого приложения.

Инициализация исключений и прерываний защищенного режима. Программа инициализации должна загрузить IDT защищенного режима, по крайней мере, для всех исключений процессора. Если шлюзы прерываний и ловушек используются, то дескрипторы шлюзов могут указывать на один и тот же сегмент кода. Если используются шлюзы задач, то для каждого обработчика требуются один TSS и сопутствующие сегменты кода, данных и задачи.

Если аппаратура позволяет генерировать прерывания, то дескрипторы шлюзов должны быть для одного или более обработчиков прерываний.

Инициализация трансляции страниц. Трансляция страниц управляется PG флагом в регистре CR0. Когда этот флаг сброшен, трансляция страниц отключена. Когда этот флаг установлен, трансляция страниц включена. Перед установкой флага PG следующие структуры и регистры должны быть проинициализированы:

- Должен быть загружен, по крайней мере, один каталог страниц и одна таблица страниц в физическую память.
- Управляющий регистр CR3 должен быть загружен базовым физическим адресом каталога страниц.

После завершения инициализации трансляция страниц может быть включена установкой бита PG в регистре CR0.

Инициализация многозадачности. Если не предполагается использовать механизм многозадачности, изменения между уровнями привилегий не допускаются. Для этого необходимо загрузить TSS в память и проинициализировать регистр TR.

Если предполагается использовать механизм многозадачности, изменения между уровнями привилегий допускаются. Программа инициализации должна загрузить хотя бы один TSS и соответствующий ему дескриптор TSS. TSS дескрипторы не должны быть помечены как занятые. Они должны быть помечены как занятые только процессором при запуске задачи.

После переключения процессора в защищенный режим инструкция LTR может быть использована, чтобы загрузить селектор сегмента для TSS дескриптора в регистр задачи. Эта инструкция помечает дескриптор занятым, но производит переключения задачи. Процессор, тем не менее, использует TSS для определения указателей стеков 0, 1 и 2 колец привилегий. Селектор сегмента TSS должен быть загружен до того, как произойдет первое переключение задачи, потому что переключение текущей задачи копирует текущее состояние в TSS.

После запуска инструкции LTR следующие операции с регистром TR будут вызывать переключение задач.

Переключение режимов. Для того чтобы использовать процессор в защищенном режиме, необходимо произвести переключение режимов. Однажды переключившись в защищенный режим, обычно нет необходимости переключаться в реальный режим. Программы, написанные для реального режима, можно запускать в режиме virtual-8086 [3].

Переключение в защищенный режим. Переключение в защищенный режим производится запуском инструкции MOV CR0, которая устанавливает PE флаг в регистре CR0. Выполнение в защищенном режиме начинается с CPL=0.

Процессоры 32-битовой архитектуры Intel имеют немного разные требования для переключения в защищенный режим. Для гарантии совместимости со всеми 32 битовыми процессорами рекомендуется выполнять следующие шаги:

- Отключить прерывания. Инструкция CLI маскирует все маскируемые прерывания. NMI прерывания могут быть отключены через внешние схемы.
- Выполнить LGDT инструкцию для загрузки GDTR регистра с базовым адресом GDT.

- Выполнить инструкцию MOV CR0, которая устанавливает PE флаг.
- Сразу же за инструкцией MOV CR0 выполнить FAR JMP или FAR CALL для очистки предвыборки процессора.
- Если предполагается использовать LDT, то загрузить LLDTR регистр.
- Выполнить инструкцию LTR для загрузки регистра задачи с сегментным селектором начальной задачи защищенного режима.
- После входа в защищенный режим сегментные регистры продолжают содержать значения, которые они содержали в реальном режиме, необходимо:
  - Перезагрузить регистры DS, SS, ES, FS и GS новыми значениями.
  - Выполнить CALL или JMP на новую задачу.
- Выполнить инструкцию LIDT для загрузки регистра IDTR адресом и пределом LDT.
- Запустить STI инструкцию для разрешения прерываний.

Переключение назад в реальный режим. Процессор переключается в реальный режим, если программа сбрасывает бит PE в регистре CR0. Процедура, которая входит в реальный режим, должна выполнить следующие шаги:

- Запретить прерывания. CLI инструкция маскирует маскируемые прерывания. Запретить NMI прерывание.
- Если трансляция страниц включена, то выполнить следующие операции:
  - Передать управление на линейный адрес, который идентично отображается в физический адрес.
  - Гарантировать, что IDT и GDT идентично отображаются.
  - Сбросить флаг PG в регистре CR0.
  - Загрузить CR3 нулевым значением.
- Передать управление в сегмент с пределом 64 Кб.
- Загрузить SS, DS, ES, FS и GS селекторами дескрипторов сегментов, приемлемых для реального режима.
- Загрузить LIDT инструкцию для загрузки таблицы векторов прерываний реального режима.

- Сбросить PE флаг в регистре CR0 для переключения в реальный режим.
- Выполнить FAR JMP инструкцию в программу реального режима.
- Перезагрузить SS, DS, ES, FS и GS для реального режима.
- Выполнить STI инструкцию для разрешения прерываний и разрешить NMI прерывание.

Механизмы защиты. В защищенном режиме архитектура Intel предоставляет механизм защиты на обоих уровнях — сегментном и трансляции страниц. Этот механизм обеспечивает возможность ограничения доступа к определенным сегментам или страницам в зависимости от уровней привилегий [3].

При использовании механизма защиты к каждой ссылке в память применяются различные проверки защиты. Все проверки выполняются до начала цикла обращения к памяти, любое нарушение приводит к возникновению исключения. Проверки защиты делятся на следующие категории:

- Проверка предела.
- Проверка типа.
- Проверка уровня привилегий.
- Ограничение адресного домена.
- Ограничение на точку входа в процедуру.
- Ограничение на множество команд.

Установка флага PE в регистре CR0 переключает процессор в защищенный режим, что в свою очередь включает сегментный механизм защиты. Частью механизма сегментной защиты является проверка, основанная на уровне привилегий, которая может быть отключена назначением всем сегментам уровня привилегий 0 (самый привилегированный).

Защита на уровне страниц автоматически включается после включения механизма трансляции страниц. Эта защита, также как и сегментная, не может быть отключена через управляющий бит. Тем не менее, ее можно отключить, выполнив следующие действия:

- Сбросить флага WP в регистре CR0.

- Установить флаги чтения / записи (R/W) и пользователь/супервизор (U/S) для каждого элемента каталога страниц и таблиц страниц.

Процессорный механизм защиты использует следующие поля и флаги в системных структурах данных для управления доступом к сегментам и страницам:

- Тип дескриптора (S). Определяет либо дескриптор системного сегмента, либо сегмента кода или данных.
- Тип. Определяет тип системного сегмента или сегмента кода/данных.
- Предел. Определяет размер сегмента совместно с флагами G и E.
- G. Определяет размер сегмента совместно с пределом и E флагом.
- E. Определяет размер сегмента совместно с пределом и G флагом.
- Уровень привилегий дескриптора (DPL). Определяет уровень привилегий сегмента.
- Запрашиваемый уровень привилегий (RPL). Указывает запрашиваемый уровень привилегий селектора сегмента.
- Текущий уровень привилегий (CPL). Показывает уровень привилегий текущей выполняемой программы или процедуры.
- Пользователь/супервизор (U/S). Определяет тип страницы.
- Чтение/запись (R/W). Определяет тип доступа к странице.

Проверка поля предела дескриптора сегмента предотвращает попытки доступа за пределы сегмента. Для всех типов сегментов, за исключением расширяющихся вниз, значение предела — это последний адрес, к которому возможен доступ.

Дескрипторы сегментов содержат информацию типа в двух местах: S флаге и поле типа. Далее приводится список примеров типичных операций, где проводится проверка типа:

- Когда селектор загружается в сегментный регистр.
  - Регистр CS может быть загружен только селектором кодового сегмента.
  - Селекторы сегментов для сегментов кода недоступных на чтение или системных сегментов не могут быть за-

груженены в сегментные регистры данных (DS, ES, FS и GS).
  - Только селектор сегмента данных доступного для записи может быть загружен в регистр SS.
- Когда селектор сегмента загружается в LDTR или TR.
  - LDTR может быть загружен только селектором LDT.
  - TR может быть загружен только селектором TSS.
- Когда инструкция обращается к сегменту, селектор которого уже загружен.
  - Инструкции не могут писать в кодовый сегмент.
  - Инструкции не могут писать в сегмент данных, недоступный для записи.
  - Инструкции не могут читать в кодовый сегмент, если флаг чтения не установлен.
- Когда операнд содержит селектор сегмента.
  - far CALL или JMP могут быть применимы только к кодовым сегментам, шлюзам вызова, шлюзам задач и TSS.
  - LLDT должна ссылаться на дескриптор для LDT.
  - LTR должна ссылаться на дескриптор для TSS.
  - LAR должна ссылаться на сегмент или дескриптор шлюза для LDT, TSS, шлюза вызова, шлюза задачи, кодового сегмента или сегмента данных.
  - LSL должна ссылаться на дескриптор сегмента для LDT, TSS, сегмента кода или сегмента данных.
  - Элементами IDT должны быть прерывания, ловушки или шлюзы задач.
- Во время внутренних операций.
  - На far CALL или JMP процессор определяет тип передачи управления – либо передача в другой кодовый сегмент или переключение задач.
  - На CALL или JMP через шлюз вызова процессор автоматически проверяет, что дескриптор сегмента, указанный в шлюзе вызова, определяет кодовый сегмент.
  - На CALL или JMP на новую задачу через шлюз задачи процессор автоматически проверяет, что дескриптор сегмента, указанный в шлюзе задачи, является TSS.

- На CALL или JMP на новую задачу через прямую ссылку процессор автоматически проверяет, что дескриптор сегмента является TSS.
- При возврате из вложенной задачи процессор проверяет, что поле связи с предыдущей задачей указывает на TSS.

Сегментный механизм защиты распознает 4 уровня привилегий от 0 до 3. Больший номер означает меньший уровень привилегий. Рисунок 2.13 показывает, как эти уровни привилегий могут быть интерпретированы в качестве колец защиты.

Процессор использует уровни привилегий для предотвращения доступа менее привилегированных программ к более привилегированным программам или их данным.

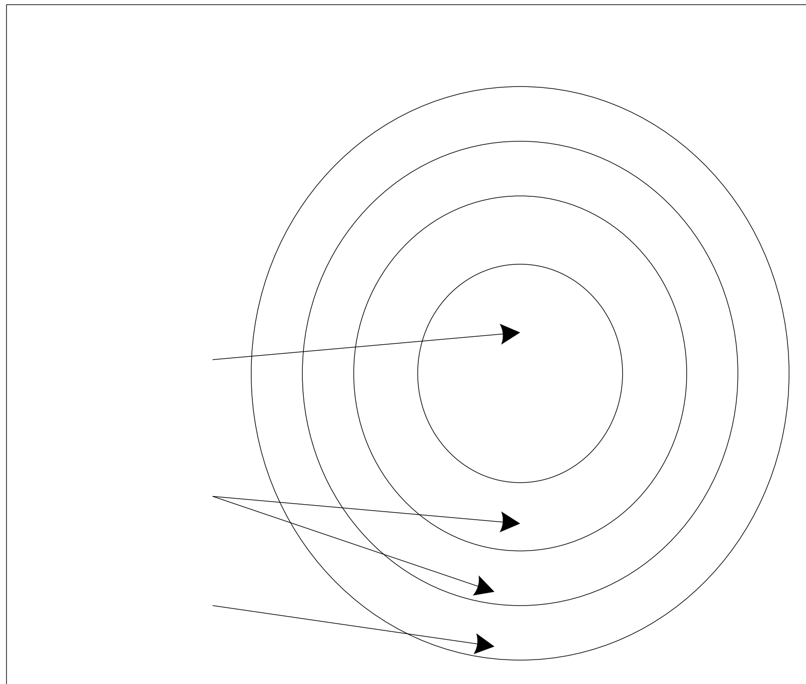


Рис. 2.13–Кольца защиты

Для выполнения проверок между сегментами кода и сегментами данных процессор распознает следующие три типа уровней привилегий [3]:

- Текущий уровень привилегий (CPL). CPL — это уровень привилегий текущей выполняемой программы или задачи. Обычно CPL равен уровню привилегий кодового сегмента, в котором происходит выполнение. Процессор изменяет CPL,

когда управление передается в кодовый сегмент с другим уровнем привилегий.

- Уровень привилегий дескриптора (DPL). DPL — это уровень привилегий сегмента или шлюза. Когда программа обращается к сегменту или шлюзу, DPL сегмента или шлюза сравнивается с CPL и RPL сегмента или шлюза. DPL интерпретируется по-разному в зависимости от типа сегмента или шлюза.
- Запрашиваемый уровень привилегий. RPL — это переопределяемый уровень привилегий, который назначается селектору сегмента. Процессор проверяет RPL вместе с CPL для определения доступности сегмента.

Уровни привилегий проверяются во время загрузки селекторов сегментов в сегментные регистры.

2.2.3 Обработка прерываний и исключений

Таблица дескрипторов прерываний (IDT) ассоциирует каждый вектор исключения или прерывания с дескриптором шлюза для процедуры или задачи, предназначенной для обслуживания ассоциированных векторов и прерываний [3]. Как и GDT и LDT, IDT — это массив 8 байтовых дескрипторов (в защищенном режиме). В отличие от GDT, первый элемент IDT может содержать дескриптор. Так как имеется только 256 векторов прерываний и исключений, нет необходимости более чем 256 дескрипторах. Она может содержать менее 256 дескрипторов, так как дескрипторы требуются только для векторов прерываний и исключений, которые могут произойти. Все пустые дескрипторные слоты в IDT должны иметь флаг присутствия P равный 0.

Базовый адрес IDT должен быть выровнен на границу 8 байт для максимизации производительности. Значение предела измеряется в байтах и добавляется к базовому адресу. Так как элементы IDT всегда 8 байтовые, то предел должен быть на 1 меньше, чем число дескрипторов, умноженное на 8 (то есть 8N-1).

IDT может находиться в любом месте линейного адресного пространства. Как показано на рис. 2.14, процессор определяет положение IDT, используя регистр IDTR. Этот регистр содержит 32-битовый базовый адрес и 16-битовый предел.

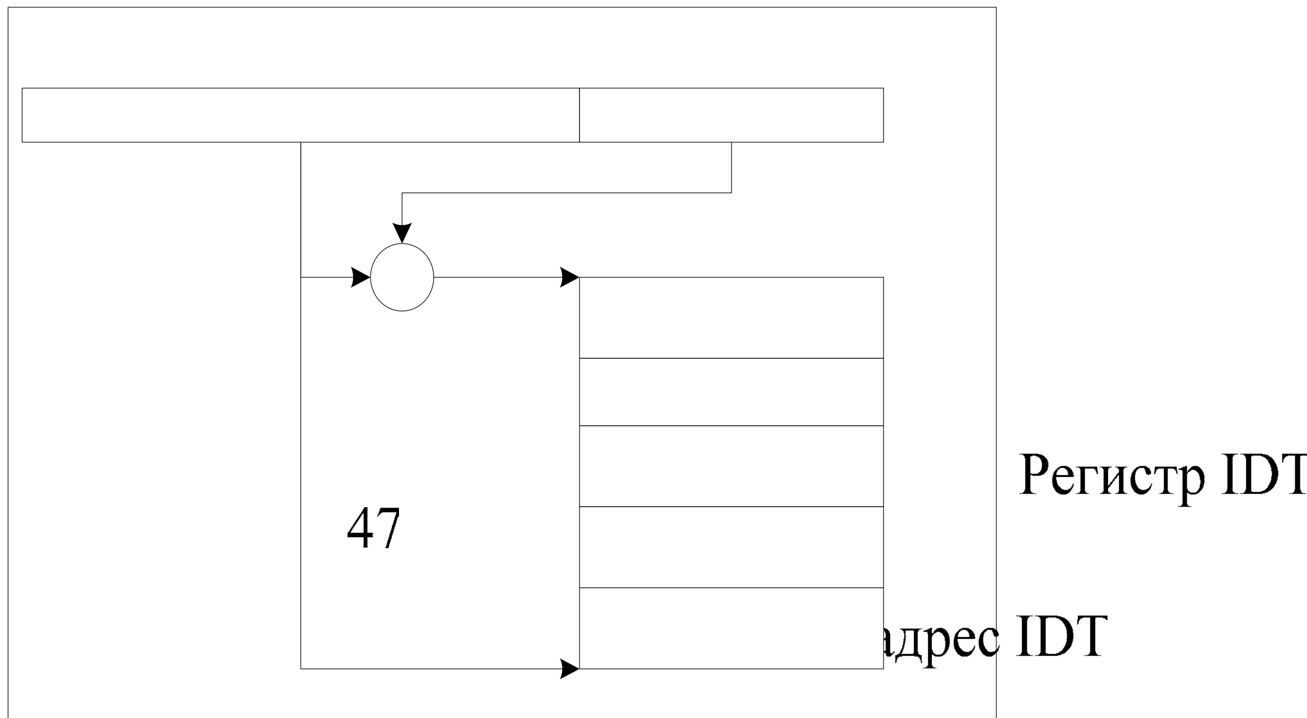


Рис. 2.14–Отношение между idtr и IDT

Инструкции LIDT и SIDT используются для загрузки и сохранения регистра IDTR. Инструкция LIDT может быть выполнена только на нулевом уровне привилегий.

Если вектор ссылается на дескриптор за границей таблицы IDT, генерируется исключение основной защиты.

IDT может содержать дескрипторы шлюзов трех видов:

- Дескриптор шлюза задачи.
- Дескриптор шлюза прерывания.
- Дескриптор шлюза ловушки.

Формат шлюза задачи, используемый в IDT, точно такой же, как и в GDT и LDT. Шлюз задачи содержит селектор сегмента TSS для обработчика исключения и/или прерывания.

Шлюзы прерываний и ловушек очень похожи на шлюзы вызова. Они содержат дальний указатель, который процессор использует для передачи управления в процедуру обработчика.

Эти шлюзы отличаются только тем, как процессор оперирует с IF флагом в регистре EFLAGS.

Процессор обрабатывает вызовы в обработчики исключений и прерываний точно так же, как он обрабатывает вызовы по команде CALL в процедуру или задачу. Для вызова исключения или прерывания процессор использует вектор прерывания или исключения как индекс в таблице IDT.

Шлюзы прерываний и ловушек ссылаются на процедуры обработчиков исключений и прерываний, которые запускаются в контексте текущей выполняемой задачи (рис. 2.15). Селектор сегмента для шлюза указывает на дескриптор сегмента кода либо в GDT, либо в текущей LDT. Поле смещения дескриптора шлюза указывает на точку входа в обработчик.

Когда процессор выполняет вызов процедуры обработчика, он сохраняет текущее состояние регистров EFLAGS, CS и EIP в стеке (см. рис. 2.16). Если предполагается наличие кода ошибки, то он заносится в стек за значением EIP.

Если обработчик запускается на текущем уровне привилегий, то используется текущий стек.

Если обработчик запускается на уровне привилегий численно меньшем, то происходит переключение стека. Когда происходит переключение стека, указатель старого стека тоже сохраняется в новом стеке. Селектор сегмента и указатель стека достаются из TSS текущей исполняемой задачи. Процессор копирует EFLAGS, SS, ESP, CS, EIP и код ошибки из стека прерванной процедуры в стек обработчика.

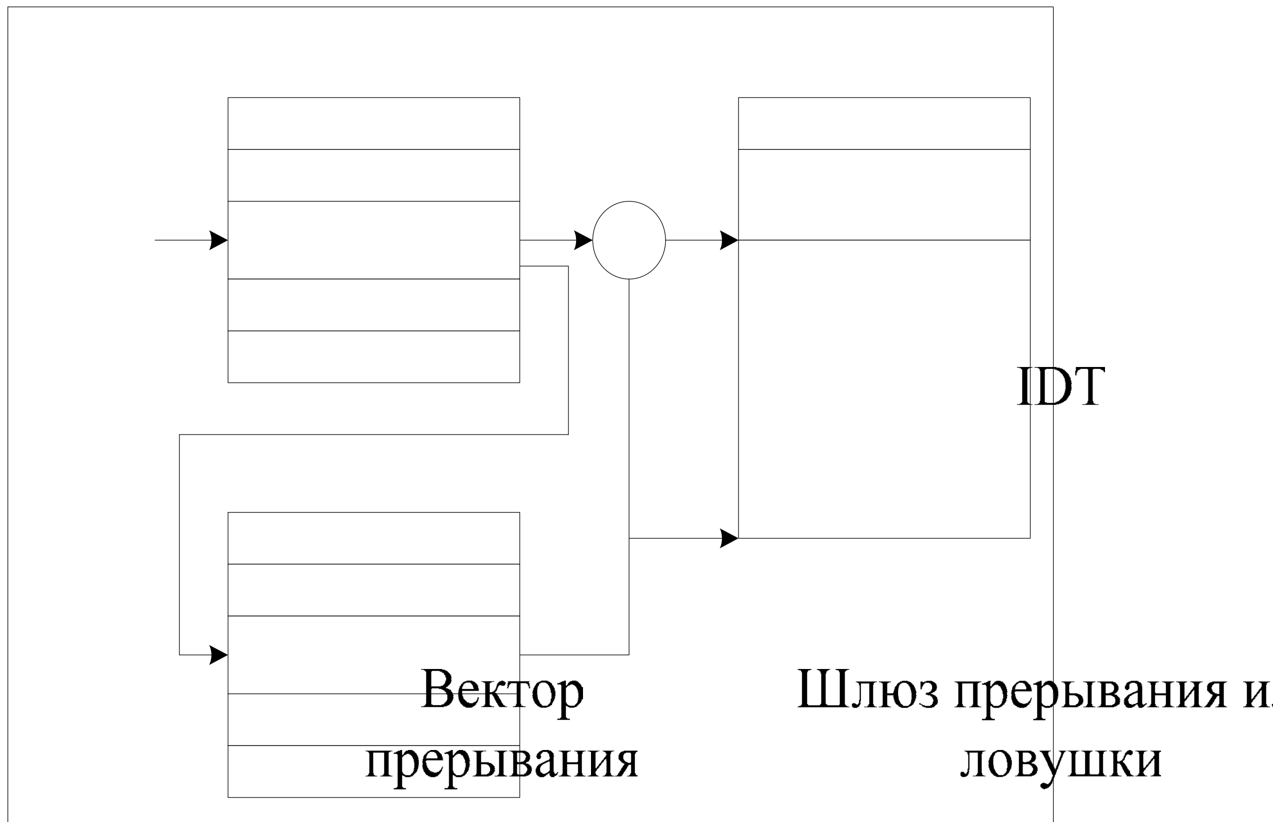


Рис. 2.15–Вызов процедуры прерывания

Для возврата из процедуры обработчика необходимо использовать инструкцию IRET. Эта инструкция отличается от RET тем, что она восстанавливает регистр EFLAGS. Поле IOPL восстанавливается, только если CPL=0. Флаг IF изменяется, только если CPL<=IOPL.

Если было переключение стека во время вызова обработчика, то инструкция IRET переключает стек обратно.

Защита уровня привилегий для процедур обработчиков исключений и прерываний похожа на ту, которая используется при обычном вызове процедуры. Процессор не позволяет передачу управления в процедуру менее привилегированного сегмента кода.

Так как прерывания и исключения обычно возникают не предсказуемо, то проверки привилегий будут проводиться относительно уровня привилегий, на котором возникло прерывание или исключение. Следующая техника позволяет избежать ошибок, связанных с уровнями привилегий:

- Обработчики исключений и прерываний могут быть помещены в подчиненный сегмент. Эта техника может быть ис-

пользована только в обработчиках, для которых достаточно данных, находящихся в стеке. Если обработчику нужны данные из сегмента данных, то он должен быть доступен с уровня привилегий 3, что делает его незащищенным.

- Обработчик может быть помещен в неподчиненный сегмент с уровнем привилегий 0. Этот обработчик будет всегда запускаться.

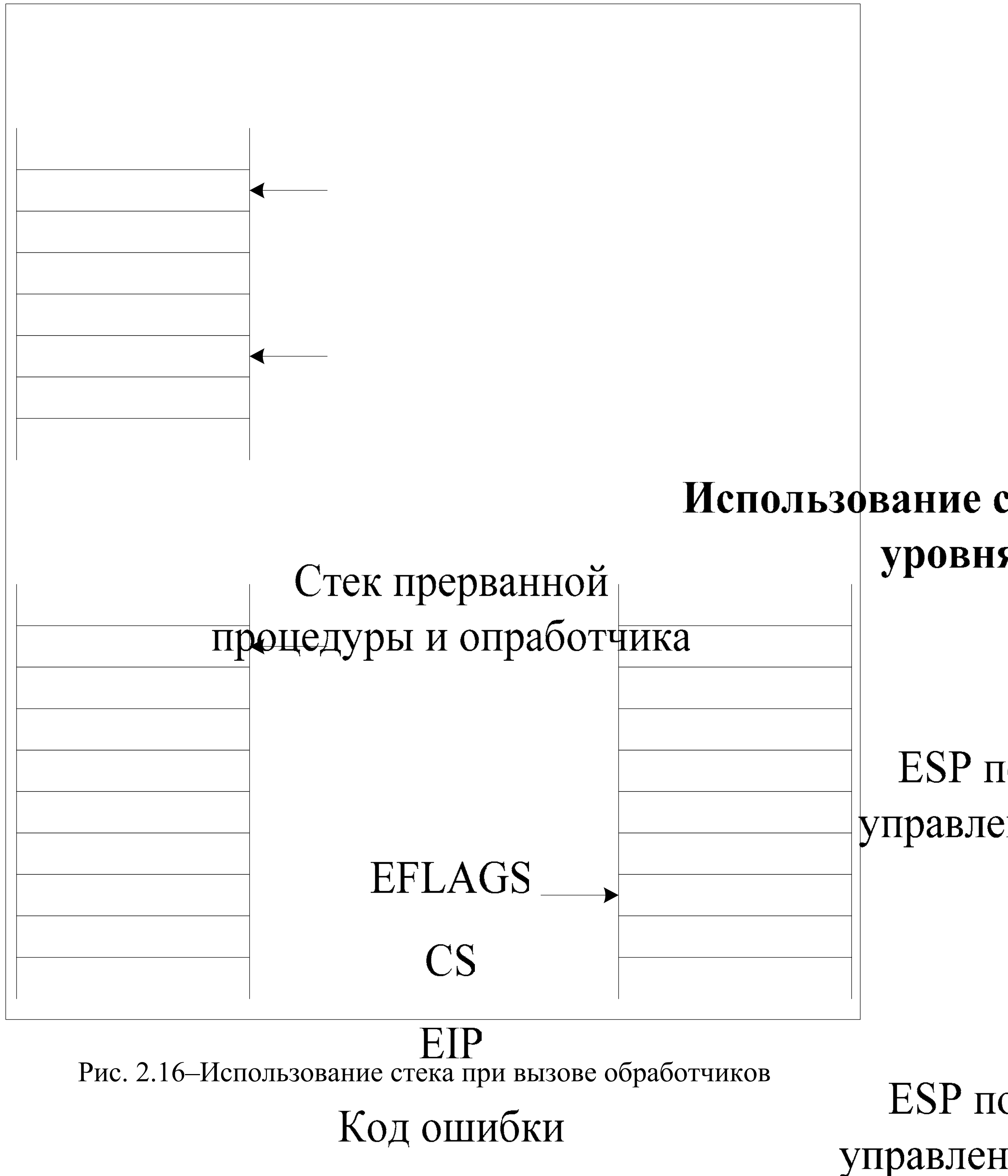


Рис. 2.16–Использование стека при вызове обработчиков

При доступе к обработчику исключения или прерывания через шлюз прерывания или ловушки процессор сбрасывает TF флаг, после сохранения содержимого EFLAGS в стеке. (На вызовы обработчиков исключений и прерываний процессор также сбрасывает флаги VM, RF и NT).

Единственное различие между шлюзами прерываний и шлюзами ловушек — это то, как процессор обрабатывает IF флаг. При использовании шлюза прерывания процессор сбрасывает IF флаг, а последующая инструкция IRET его восстанавливает. В случае шлюза ловушки процессор не изменяет содержимое этого флага.

При доступе к обработчику прерывания или исключения через шлюз задачи происходит переключение задач. Обработка исключения или прерывания в отдельной задаче дает несколько преимуществ:

- Весь контекст прерванной программы или задачи сохраняется автоматически.
- Новый TSS позволяет обработчику использовать новый стек уровня привилегий 0.
- Обработчик может быть изолирован от других задач использованием отдельного адресного пространства, например через LDT.

Недостаток обработки прерывания через отдельную задачу — это большее число тактов процессора на переключение контекста, что приводит к некоторым задержкам в обработке прерываний.

Шлюз задачи ссылается на TSS дескриптор в GDT. Переключение в задачу обработчика происходит точно так же, как обычное переключение задач. Обратная связь с прерванной задачей хранится в поле связи с предыдущей задачей текущего TSS. Если для исключения генерируется код ошибки, этот код ошибки в стек новой задачи.

Когда условие исключения относится к определенному сегменту, процессор помещает в стек обработчика прерывания код ошибки. На рис. 2.17 показан формат кода ошибки. Код ошибки состоит из селектора сегмента, но вместо TI флага и RPL поля используются следующие 3 флага:

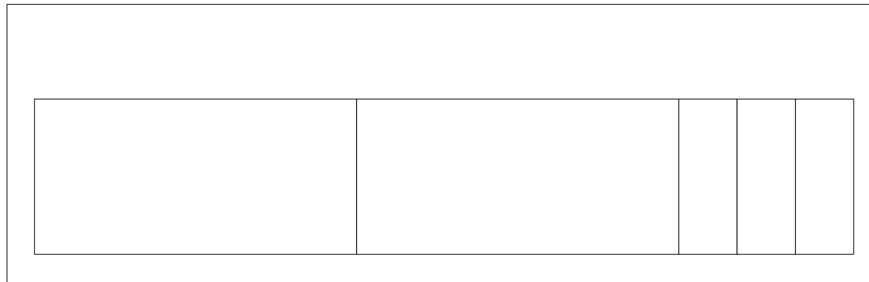


Рис. 2.17–Код ошибки

- EXT — Внешнее событие (бит 0). Когда установлен, указывает, что событие внешнее по отношению к программе вызвало исключение.
- IDT — Положение дескриптора (бит 1). Когда установлен, показывает, что индекс ссылается на дескриптор шлюза в IDT; когда сброшен, указывает, что индекс ссылается либо на GDT, либо на LDT.
- TI (бит 2). Используется, только когда IDT флаг сброшен. Когда установлен, индекс ссылается на LDT; когда сброшен — на GDT.
- Поле «индекс» предоставляет индекс в IDT, GDT или текущей LDT на селектор сегмента или шлюза ссылаемый кодом ошибки. В некоторых случаях сегмент кода может быть нулевым.

Формат кода ошибки для исключения ошибки страницы отличается от обычного кода ошибки.

Код ошибки помещается в стек как слово или двойное слово (в зависимости от размера шлюза прерывания, ловушки или задачи). Для поддержания выравнивания в стек помещается двойное слово, старшая часть кода ошибки зарезервирована. Код ошибки не удаляется из стека по инструкции IRET, обработчик должен удалить его перед выходом.

Код ошибки не помещается в стек для исключений, которые генерируются внешне или через инструкцию INT, даже если код ошибки нормально генерируется для этих исключений.

2.2.4 Управление задачами

Архитектура Intel обеспечивает механизм для сохранения состояния задачи, для выборки задачи для исполнения и для переключения из одной задачи в другую [3]. В защищенном режиме процессор все время работает в контексте какой-нибудь задачи. Даже простейшие системы должны определить, по крайней мере, одну задачу. Более сложные системы могут использовать возможности процессора по управлению задачами для поддержки многозадачных приложений.

Задача состоит из двух частей: пространства выполнения задачи и сегмента состояния задачи (TSS). Пространство выполнения задачи состоит из сегмента кода, сегмента стека, одного или более сегментов данных (рис. 2.18).

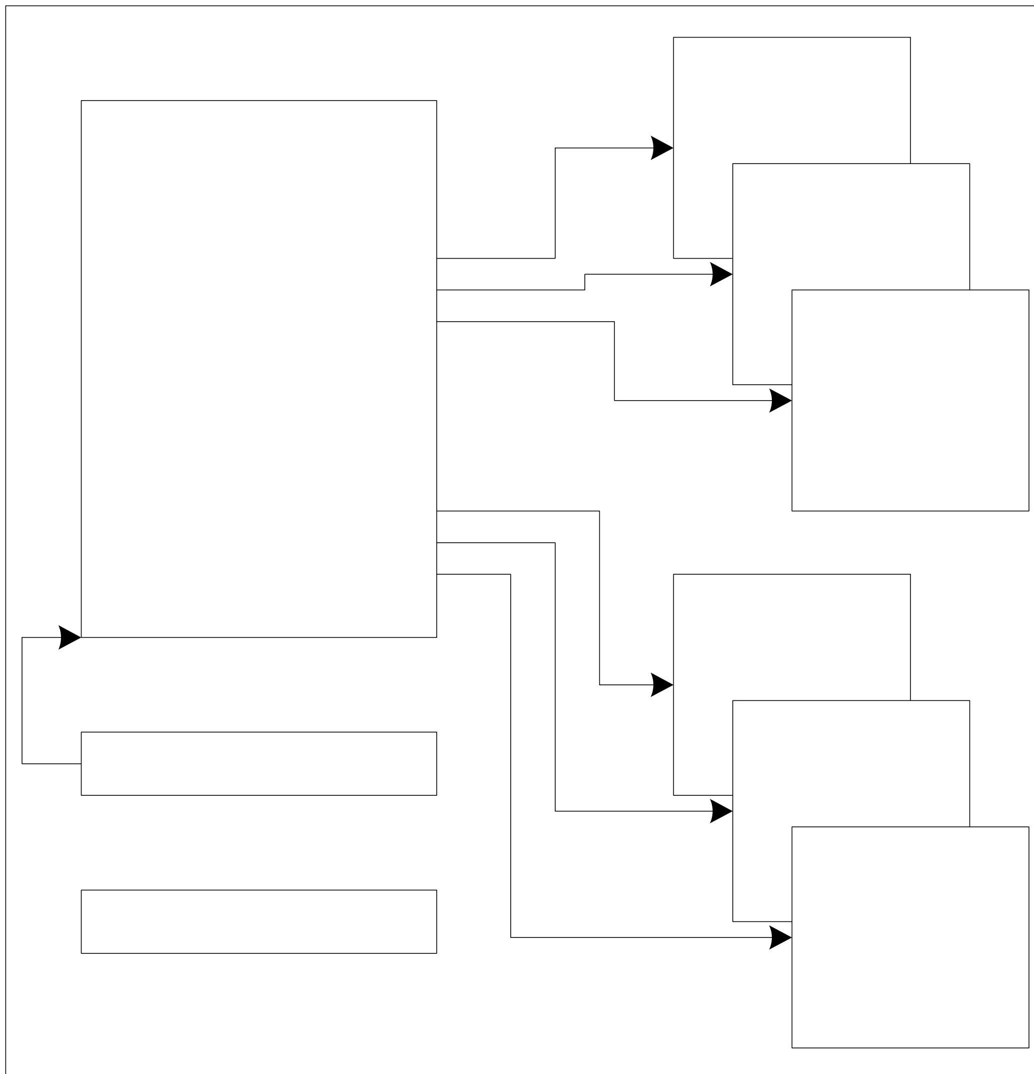


Рис. 2.18–Структура задачи

TSS определяет сегмент, который определяет пространство выполнения задачи, и предоставляет место для сохранения информации состояния. В многозадачных системах TSS также предоставляет механизм для связывания задач.

Следующие элементы определяют состояние текущей выполняемой задачи:

- Пространство выполнения текущей задачи, определяемое селекторами сегментов в сегментных регистрах (CS, DS, SS, ES, FS и GS).
- Состояние регистров общего назначения.
- Состояние регистра EFLAGS.
- Состояние регистра EIP.
- Состояние управляющего регистра CR3.
- Состояние регистра задачи.
- Состояние регистра LDTR.
- Базовый адрес карты В/В и карта В/В.
- Указатели стека для стеков привилегий 0, 1 и 2.
- Ссылка на предыдущую задачу.

Процессор определяет пять структур данных:

- Сегмент состояния задачи (TSS).
- Дескриптор шлюза задачи.
- Дескриптор TSS.
- Регистр задачи.
- NT флаг в регистре EFLAGS.

В защищенном режиме TSS и дескриптор TSS должны быть созданы, по крайней мере, для одной задачи, и селектор сегмента TSS должен быть загружен в регистр задачи.

Сегмент состояния задачи (TSS). Информация состояния процессора, необходимая для восстановления задачи, хранится в системном сегменте, называемом сегментом состояния задачи (TSS). На рис. 2.19 показан формат TSS задачи для 32-битового ЦПУ. Поля TSS делятся на две категории: динамические поля и статические поля [3].

Процессор обновляет динамические поля, когда задача приостанавливается, во время переключения задач. Далее приводятся динамические поля:

- Поля регистров общего назначения. Состояние регистров EAX, ECX, EDX, EBX, ESP, EBP, ESI и EDI на момент переключения.
- Поля селекторов сегментов. Селекторы сегментов из ES, CS, SS, DS, FS и GS на момент переключения задач.
- Поле регистра EFLAGS. Состояние регистра EFLAGS на момент переключения.
- Поле EIP. Состояние регистра EIP на момент переключения.
- Поле связи с предыдущей задачей. Содержит селектор сегмента TSS предыдущей задачи (модифицируется при переключении инициированным CALL, прерыванием, исключением).

Процессор читает статические поля, но не изменяет их. Эти поля устанавливаются при создании задачи. Далее приводятся статические поля:

- Поле селектора сегмента LDT. Содержит селектор сегмента LDT для задачи.
- Поле управляющего регистра CR3. Содержит базовый физический адрес каталога страниц, используемого задачей.
- Поля указателей стеков уровней привилегий уровня 0, 1 и 2. Эти указатели являются логическими адресами, состоящими из селекторов сегментов и смещений. Значения в этих полях статические для каждой задачи, в то время как значения SS и ESP изменяются во время переключения.
- T флаг. Когда установлен, процессор вызывает исключение при переключении задач.
- Поле базового адреса карты В/В. Содержит 16-битовое смещение от базового адреса до битовой карты разрешения ввода/вывода и битовую карту переназначения прерываний.

Если трансляция страниц используется, то следует избегать расположения границы страниц в TSS. Если же TSS находится в двух страницах, то необходимо, чтобы эти страницы присутствовали последовательно в физической памяти.

Дескриптор TSS. TSS, как и все другие сегменты, определяется дескриптором сегмента. На рис. 2.19 показан формат дескриптора TSS [3]. Дескриптор TSS может быть размещен только в GDT.

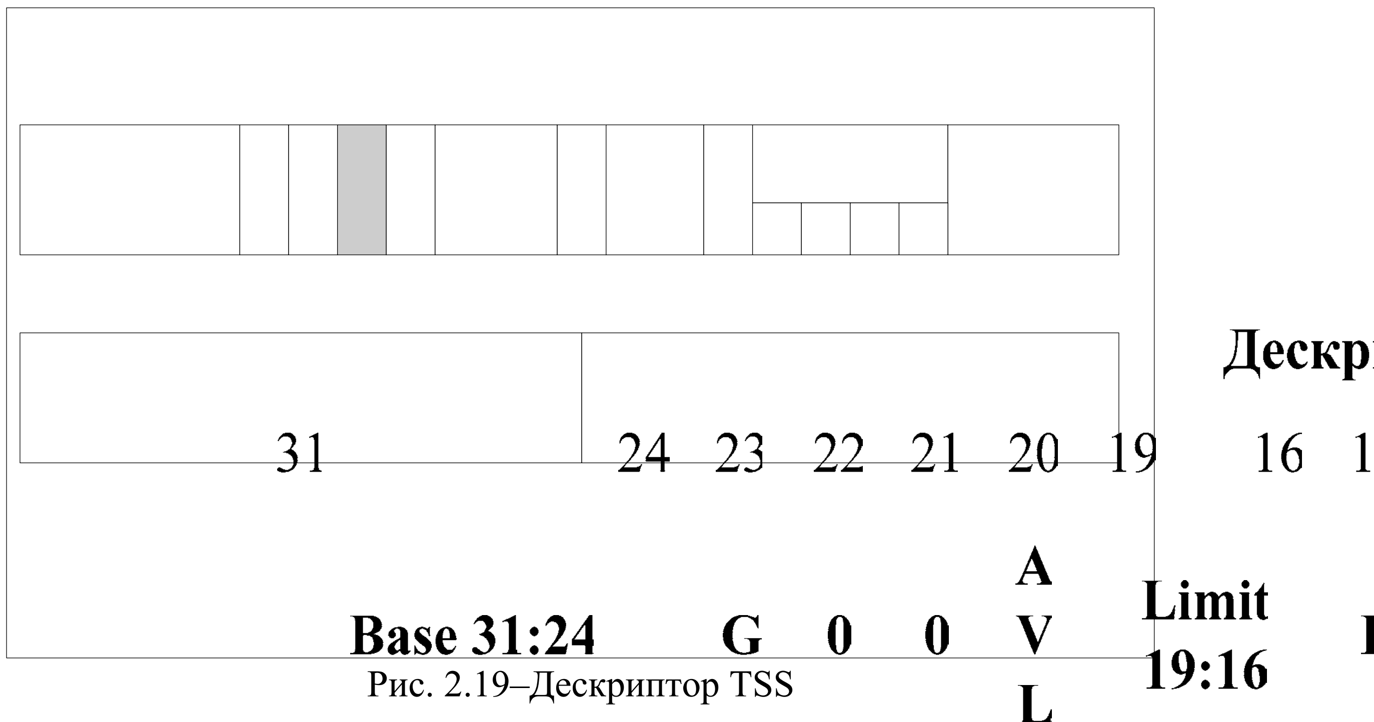


Рис. 2.19–Дескриптор TSS

Флаг занятости (В) в поле типа показывает занятость задачи. Занятая задача выполняется в текущий момент или приостановлена; значение 1001В определяет неактивную задачу; значение 1011В определяет занятую задачу. Задачи не рекурсивны.

База, предел и DPL поля и флаги гранулярности и присутствия выполняют те же самые функции, как в дескрипторах сегментов данных. Поле предела должно иметь значение, равное и большее 67Н. При попытке переключиться в задачу, чей TSS имеет меньший предел, произойдет исключение. Больший предел может потребоваться, если битовая карта ввода/вывода используется.

Любая программа или процедура, имеющая доступ к дескриптору TSS (т.е. чей CPL численно равен или меньше DPL дескриптора TSS), может запустить задачу через CALL или JMP.

Регистр задачи. Регистр задачи – 16-битовый селектор сегмента и весь дескриптор сегмента TSS текущей задачи. Эта информация копируется из GDT. На рис. 2.20 показано, как процессор осуществляет доступ к TSS, используя информацию из регистра задачи [3].

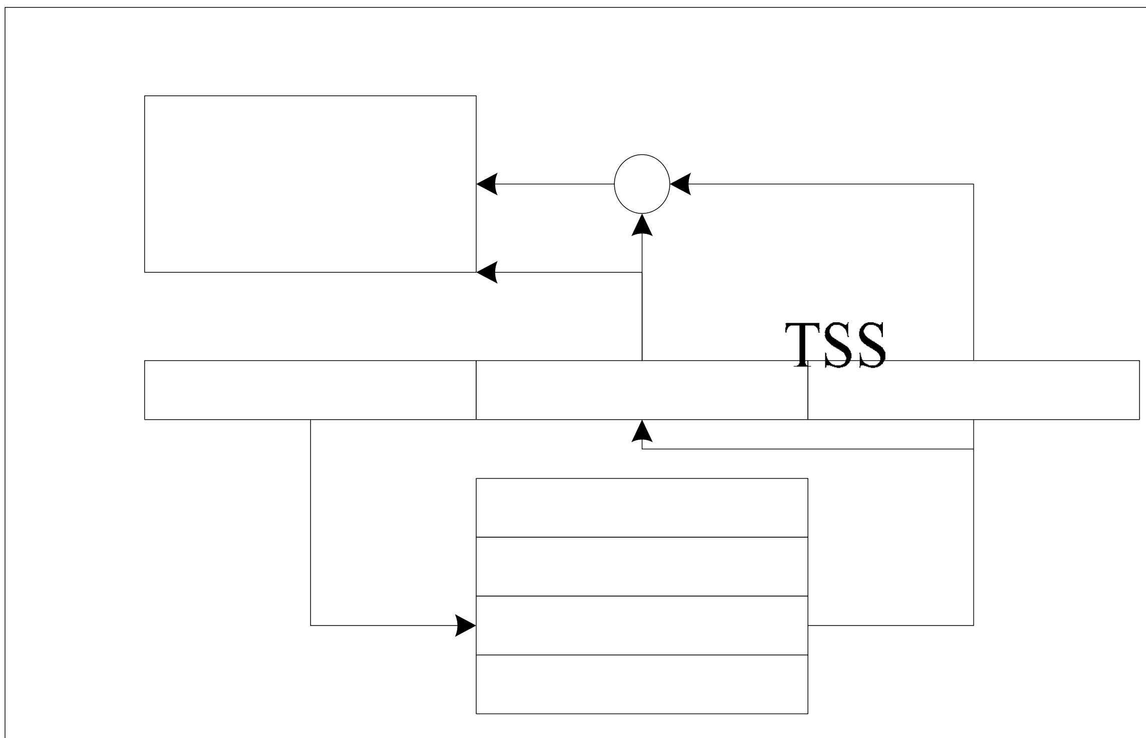


Рис. 2.20–Регистр задачи

Инструкции LTR и STR загружают и сохраняют соответственно видимую часть регистра TR. Инструкция LTR может быть запущена, только если CPL=0. При первой загрузке регистра TR через инструкцию LTR происходит загрузка невидимой части регистра и установка бита занятости. В дальнейшем любое изменение регистра TR приведет к переключению задач.

Дескриптор шлюза задачи. Дескриптор шлюза задачи предоставляет косвенную защищенную ссылку на задачу. На рис. 2.21 показан формат дескриптора шлюза задачи. Дескриптор шлюза задачи может быть помещен в GDT, LDT и IDT.

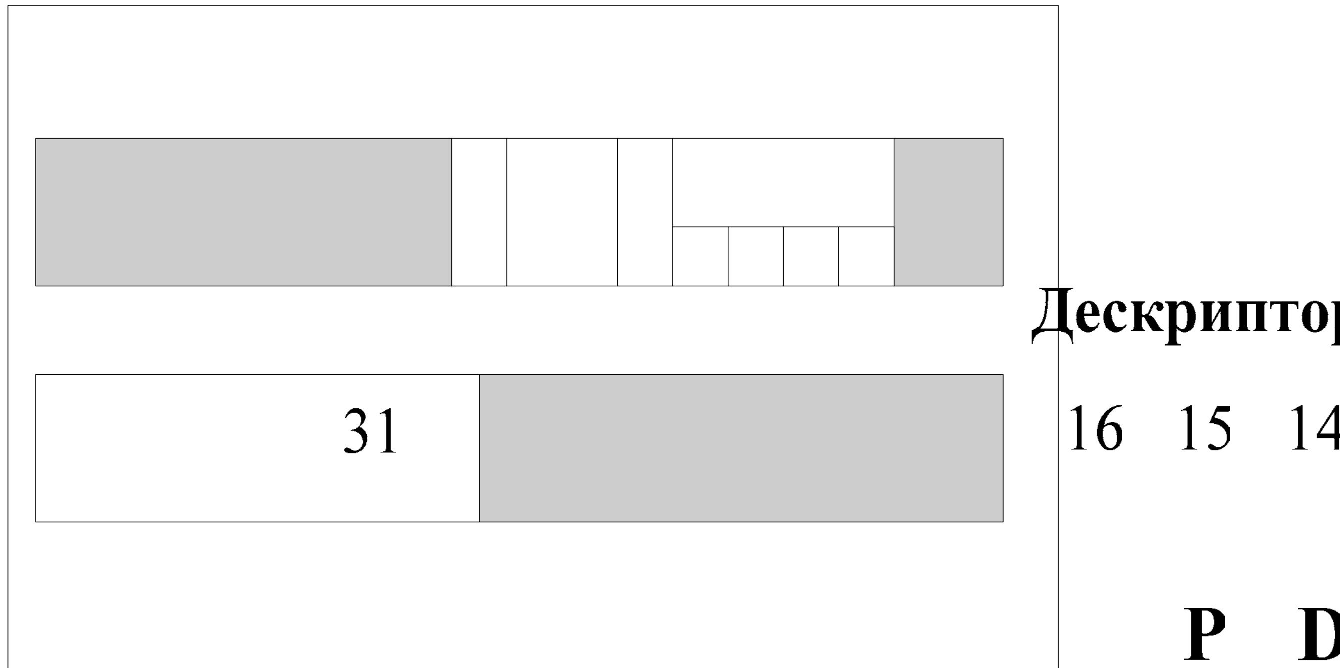


Рис. 2.21–Дескриптор шлюза задачи

Поле селектора сегмента TSS в дескрипторе шлюза задачи указывает на TSS дескриптор в GDT. RPL в этом селекторе сегмента не используется.

DPL дескриптора шлюза задачи управляет доступом к дескриптору TSS во время переключения задач. Когда программа или процедура выполняет CALL или JMP через шлюз задачи, CPL и RPL селектора шлюза должны быть меньше или равны DPL дескриптора шлюза задачи (когда используется шлюз задачи, DPL дескриптора TSS назначения не используется).

Доступ к задаче может быть получен либо через дескриптор шлюза задачи, либо TSS дескриптор. Обе эти структуры предоставляются для удовлетворения следующих нужд:

- Необходимость для задачи иметь только один флаг занятости. Так как флаг занятости хранится в дескрипторе TSS, каждая задача должна иметь только один дескриптор TSS. Однако может быть несколько шлюзов задач, ссылающихся на этот дескриптор.
- Необходимость предоставить выборочный доступ к задаче. Шлюзы задач удовлетворяют этой необходимости, потому что они могут располагаться в LDT и иметь DPL, отличный от DPL дескриптора TSS. Шлюзы задач дают операционной

системе большую свободу для ограничения доступа к специфическим задачам.

- Необходимость обслуживания прерываний в независимых задачах. Шлюзы задач также могут находиться в LDT. Когда вектор прерывания или исключения указывает на шлюз задачи, процессор переключается в соответствующую задачу.

На рис. 2.22 показано, как шлюзы задач в LDT, GDT и IDT могут указывать на одну задачу.

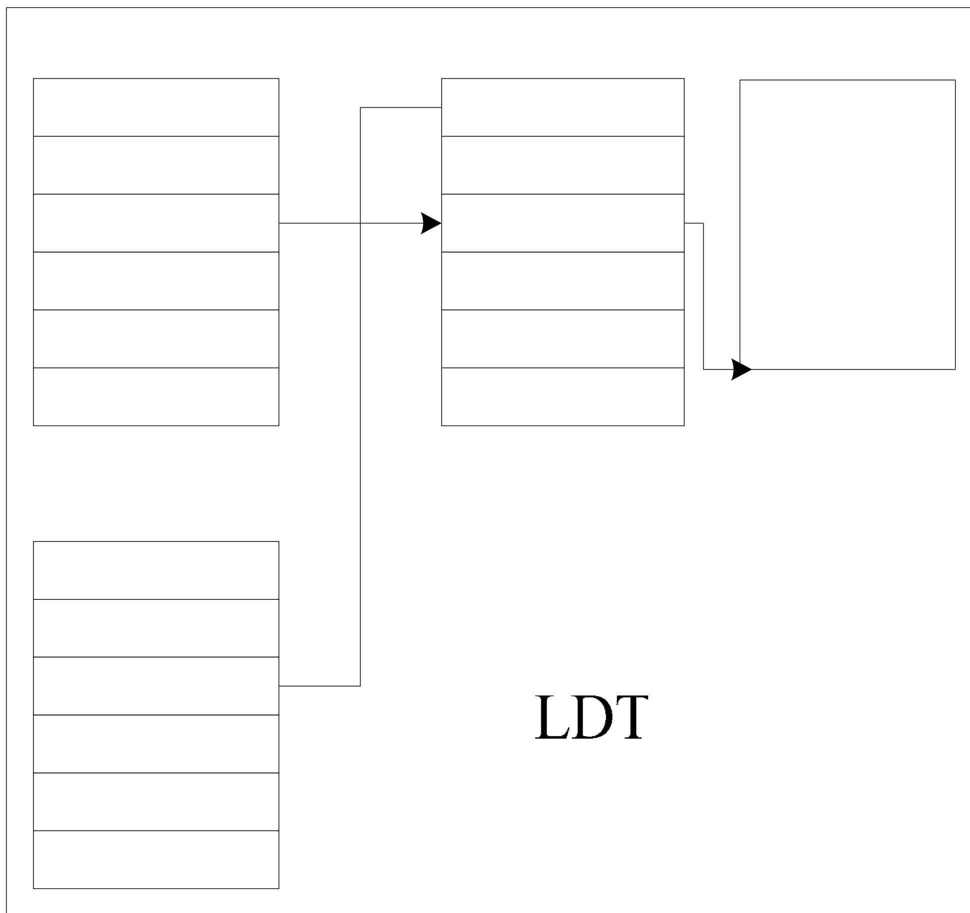


Рис. 2.22–Ссылки шлюзов задач на ту же задачу

Переключение задач. Процессор передает управление на другую задачу в любом из следующих четырех случаев:

- Текущая программа, задача или процедура выполняет JMP или CALL инструкции на дескриптор TSS в GDT.
- Текущая программа, задача или процедура выполняет JMP или CALL инструкции на дескриптор шлюза задачи в GDT или LDT.

- Вектор прерывания или исключения указывает на дескриптор шлюза задачи в GDT.
- Текущая задача запускает инструкцию IRET, когда NT флаг в регистре EFLAGS установлен.

Процессор выполняет следующие операции, когда переключается на новую задачу [3]:

1. Получает селектор сегмента TSS для новой задачи как операнд инструкций JMP или CALL из шлюза задачи или из поля связи с предыдущей задачи.
2. Проверяет, что переключение допустимо. Правила привилегий доступа к данным применяются к командам JMP или CALL. CPL текущей задачи и дескриптор TSS селектора сегмента новой задачи должны быть меньше или равны DPL дескриптора TSS или шлюза задачи, на которые происходит ссылка. Исключения, прерывания (за исключением прерываний, генерируемых командой int) и инструкция IRET допускаются для переключения задачи независимо от DPL шлюза задачи или TSS дескриптора назначения. Для прерываний, генерируемых по команде INT, DPL проверяется.
3. Проверяет, что дескриптор TSS новой задачи помечен присутствующим и имеет допустимый предел (больше или равно 67h).
4. Проверяет, что новая задача допустима (call, jmp, исключение или прерывание) или занята (возврат по iret).
5. Проверяет, что текущий TSS, новый TSS и все сегментные дескрипторы, используемые в переключении, отображены через трансляцию страниц в системную память.
6. Если переключение задачи инициируется командами JMP или IRET, процессор сбрасывает флаг занятости (B) в дескрипторе TSS текущей задачи. Если переключение инициируется командой CALL, исключением или прерыванием, флаг занятости (B) остается установленным.
7. Если переключение задачи инициируется командами JMP или IRET, процессор сбрасывает NT флаг во временно сохраненном имидже регистра EFLAGS. Если инициируется командами CALL или JMP, исключениями или прерываниями, NT флаг остается неизмененным.

8. Сохраняет состояние текущей задачи и TSS текущей задачи. Процессор находит базовый адрес текущего TSS в регистре задачи и копирует состояние следующих регистров в TSS: все регистры общего назначения, сегментные селекторы из сегментных регистров, временно сохраненный имидж регистра EFLAGS, и регистр указателя текущей команды (EIP).
9. Если переключение задач было инициировано инструкцией CALL, исключением или прерыванием, процессор устанавливает NT флаг в имидже регистра EFLAGS нового TSS. Если переключение было инициировано инструкцией IRET, процессор восстанавливает NT флаг из имиджа EFLAGS. Если инициируется инструкцией JMP, NT флаг остается неизмененным.
10. Если переключение задачи инициируется командой CALL, JMP, исключением или прерыванием, процессор устанавливает флаг занятости (B) в дескрипторе TSS новой задачи. Если инициируется инструкцией IRET, флаг занятости остается установленным.
11. Устанавливает TS флаг в управляющем регистре CR0.
12. Загружает регистр задачи селектором сегмента и дескриптором TSS новой задачи.
13. Загружает состояние задачи из TSS. Информация состояния задачи включает LDTR регистр, CR3 регистр, EFLAGS регистр, EIP регистр, регистры общего назначения и сегментные регистры.
14. Начинает выполнение новой задачи.

Использование флага занятости для избежания рекурсивного переключения задач. TSS позволяет сохранять только один контекст, тем не менее, если задача уже вызвана, рекурсивный вызов приведет к потере состояния для первого вызова. Флаг занятости в дескрипторе сегмента TSS предоставляет возможность предотвратить повторное переключение в одну и ту же задачу. Процессор управляет флагом занятости следующим образом:

1. При запуске задачи процессор устанавливает этот флаг.
2. Если во время переключения задача помещается в цепочку вложенных задач, флаг занятости для текущей задачи остается установленным.

3. При переключении задач процессор генерирует исключение основной защиты, если флаг занятости установлен (исключение составляет IRET инструкция, где предполагается, что флаг занятости должен быть установлен).
4. Когда задача завершается по инструкции JMP или IRET, процессор сбрасывает бит занятости.

Изменение цепочки задач. На однопроцессорной системе в ситуациях, когда необходимо удалить задачу из цепочки связанных задач, используется следующая процедура:

1. Запрещаются прерывания.
2. Изменяется поле связи с предыдущей задачей.
3. Сбрасывается флаг занятости в дескрипторе сегмента TSS удаляемой задачи.
4. Разрешаются прерывания.

В многопроцессорных системах необходимо добавить дополнительные операции синхронизации и сериализации в процедуру для гарантии, что TSS и его дескриптор сегмента заблокированы на время изменения бита занятости и поля связи с предыдущей задачей.

Адресное пространство задачи. Адресное пространство задачи состоит из сегментов, к которым имеет доступ задача. Эти сегменты включают в себя сегменты кода, данных, стека и системных сегментов, имеющих ссылки из TSS и любые другие сегменты, к которым имеет доступ код задачи. Эти сегменты отображаются в линейное адресное пространство процессора, которое в свою очередь отображается на физическое адресное пространство [3].

Поле LDT в TSS может использоваться для предоставления задаче своей локальной таблицы дескрипторов. Предоставление задаче своей LDT позволяет изолировать ее адресное пространство от других задач.

Возможно, чтобы несколько задач использовали одну LDT, что позволяет им разделять сегменты.

Так как все задачи имеют доступ к GDT, дескрипторы разделяемых сегментов можно помещать в таблицу.

Если включена трансляция страниц, каждая задача может иметь свое уникальное линейное пространство адресов, исполь-

зуя поле CR3 в TSS. Несколько задач также могут разделять некоторое множество таблиц страниц.

Задачи могут быть отображены в линейное адресное пространство и в физическое адресное пространство двумя способами:

- Одно преобразование линейного адреса в физический адрес преобразование разделяет между всеми задачами. Когда трансляция страниц отключена, это единственный способ. Когда трансляция страниц включена, эта форма преобразования адресов достигается использованием одного каталога страниц для всех задач. В этом случае может быть реализована виртуальная память с подкачкой по запросу.
- Каждая задача может иметь свое уникальное линейное пространство, отображаемое в физическое пространство. Это достигается использованием разных каталогов страниц для каждой задачи. Так как регистр CR3 загружается при каждом переключении задач, каждая задача может иметь свой каталог страниц.

Линейные адресные пространства разных задач могут отображаться на совершенно разные физические адреса. Если элементы разных каталогов страниц указывают на разные таблицы страниц, а элементы разных таблиц страниц указывают на разные страницы, то задачи не разделяют физических адресов.

При любом методе отображения линейного адресного пространства сегменты состояния задач (TSSs) для всех задач должны находиться в разделяемой области физического пространства, которое доступно всем задачам. При использовании трансляции страниц также требуется, чтобы отображение адресов TSS не изменялось при чтении и изменении ее процессором во время переключения задач. Физическое адресное пространство, в котором отображается GDT, также должно быть разделяемым. На рис. 2.23 показано как могут перекрываться линейные адресные пространства с целью разделения физических адресов.

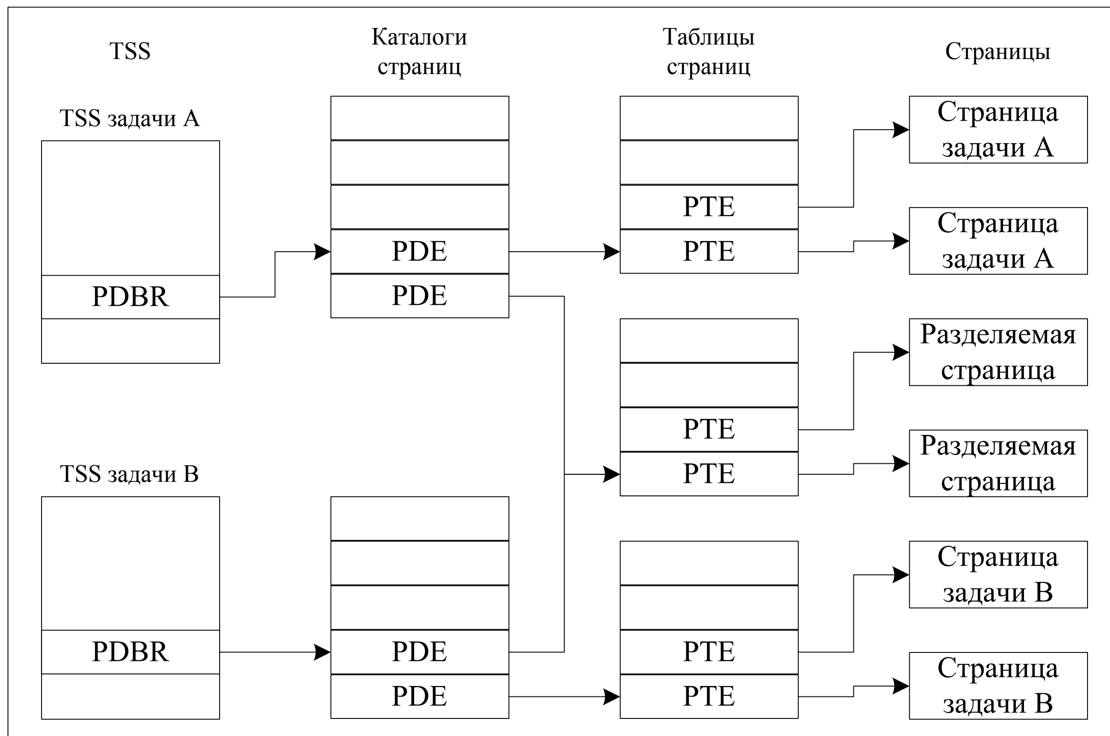


Рис. 2.23–Перекрытие линейных адресных пространств

Сегмент состояния 16-битовой задачи. 32-битовая архитектура Intel распознает формат 16-битовой TSS, так же как и в Intel 286 процессорах. Он поддерживается для совместимости с программным обеспечением, созданным для ранних процессоров.

Важно знать следующую информацию о 16-битовом TSS:

- Не нужно использовать 16-битовый TSS для реализации задачи virtual-8086.
- Допустимый предел сегмента — 2Ch.
- 16-битовый TSS не содержит поля CR3. При переключении в эту задачу будет использоваться трансляция страниц от старой задачи.
- I/O базовый адрес в включен в 16-битовый TSS.
- Когда сохраняется состояние задачи в 16-битовый TSS, старшие 16 бит регистров ESP и EIP теряются.

Когда регистры общего назначения загружаются или сохраняются, старшие 16 бит регистров изменяются и не сохраняются.

2.3 Эмуляция 8086

Процессоры архитектуры Intel (начиная с 80386) предоставляют два способа для запуска программ, созданных для 8086 процессора [3]:

- Реальный режим.
- Virtual-8086 режим.

Virtual-8086 режим является специальным типом задачи, которая запускается в защищенном режиме. Когда операционная система переключается в Virtual-8086 задачу, процессор эмулирует Intel 8086 процессор. Среда выполнения процессора во время состояния эмуляции такое же, как в реальном режиме. Основное различие между двумя режимами — это то, что в Virtual-8086 режиме эмулятор использует некоторые сервисы защищенного режима.

Включение Virtual-8086 режима. Процессор работает в Virtual-8086 режиме, когда VM флаг в регистре EFLAGS установлен. Этот флаг может быть установлен, когда процессор переключается в новую задачу или возвращается в Virtual-8086 режим после IRET инструкции.

Программное обеспечение не может изменить состояние VM флага непосредственно в регистре EFLAGS. Вместо этого оно изменяет этот флаг в имидже регистра EFLAGS, хранимом в TSS или в стеке перед выполнением инструкции IRET.

Процессор проверяет VM флаг в следующих условиях:

- При загрузке сегментных регистров.
- При декодировании инструкций, для определения недопустимых инструкций для Virtual-8086 режиме и какие инструкции чувствительны к IOPL.
- Во время проверки привилегированных инструкций, на доступ к страницам или когда выполняются другие проверки на разрешение.

Структура Virtual-8086 задачи. Virtual-8086 задача состоит из следующих элементов:

- 32 битовый TSS для задачи.
- 8086 программа.
- Virtual-8086 монитор.
- Сервисы 8086 операционной системы.

TSS новой задачи должен быть 32-битовым, так как 16-битовый TSS не содержит старших битов регистра EFLAGS. Процессор входит

в virtual-8086 режим для запуска 8086 программ и возвращается в защищенный режим для запуска virtual-8086 монитора.

Virtual-8086 монитор — это 32-битовый код защищенного режима, который выполняется на CPL=0. Монитор состоит из инициализации, обработчиков прерываний и исключений, процедур эмуляции ввода/вывода.

Сервисы 8086 операционной системы состоят из ядра и процедур операционной системы, которые вызывает 8086 программа. Эти сервисы могут быть реализованы одним из следующих образов:

- Они могут быть включены в 8086 программу. Этот подход желателен при следующих причинах:
  - Код 8086 программы модифицирует мод сервисов 8086 операционной системы.
  - Нет достаточно времени на перенос сервисов 8086 операционной системы в основную системы.
- Они могут быть реализованы или эмулированы в virtual-8086 мониторе. Этот подход желателен при следующих причинах:
  - Процедуры 8086 операционной системы могут быть значительно легче скоординированы между несколькими virtual-8086 задачами.
  - Может быть сохранена память из-за отсутствия необходимости дублирования кода процедур.
  - Процедуры 8086 операционной системы могут быть легко эмулированы вызовами в основную систему.

Транслирование страниц в Virtual-8086 задачах. Если программа, запущенная в virtual-8086 режиме, может использовать только 20 битовый линейный адрес, процессор преобразует эти адреса в 32 битовые перед отображением их в физические.

Трансляция страниц может не использоваться для одной virtual-8086 задачи, однако трансляция страниц полезна или необходима в следующих ситуациях:

- Когда запускаются несколько Virtual-8086 задач. Трансляция страниц позволяет отобразить нижние 1 Мб на разные участки физической памяти.
- При эмуляции отображения 64 Кб за 1 Мб на первые 64Кб.
- При разделении сервисов 8086 операционной системы и кода ПЗУ, общих для всех программ 8086.

- При перенаправлении или отлавливании ссылок на устройства ввода/вывода отображаемые в память.

Защита в Virtual-8086 задаче. В программах 8086 нет защиты между сегментами. Любой из следующих способов может использоваться для защиты системного программного обеспечения от 8086 программ:

- Резервирование первого 1 Мб плюс 64 Кб линейного пространства каждой Virtual-8086 задачи.
- Использование U/S флага в элементах таблицы страниц. В virtual-8086 режиме процессор работает на CPL=0.

Вход в Virtual-8086 режиме. На рис. 2.13 показаны методы включения и отключения Virtual-8086 режима. Процессор переключается в virtual-8086 режиме в любой из следующих ситуаций:

- Переключение задачи, когда VM флаг в имидже регистра EFLAGS в TSS установлен.
- Возврат из обработчика прерывания или исключения, когда VM флаг в имидже регистра EFLAGS в стеке установлен.

Рис. 2.13–Вход и выход в/из Virtual-8086 режим(а)

При переключении задач для входа в virtual-8086 режим TSS должен быть 32-битовым. Процессор изменяет VM флаг до загрузки сегментных регистров. В зависимости от значения VM флага процессор интерпретирует содержимое сегментных регистров.

Выход из Virtual-8086 режима. Процессор может выйти из virtual-8086 режима только через прерывание или исключение. Далее приводятся ситуации, в которых прерывание или исключение приведет к выходу из virtual-8086 режима:

- Процессор обслуживает аппаратные прерывания, приостанавливая выполнение 8086 приложения. При приеме аппаратного прерывания процессор переключается в защищенный режим и либо переключается на задачу, либо передает управление через шлюз прерывания или ловушки.
- Процессор обслуживает исключения, вызванные кодом Virtual-8086 задачи, или обслуживает прерывания, принадлежащие этой задаче.
- Процессор обслуживает программные прерывания, генерируемые кодом virtual-8086 задачи (такие как прерывания MS DOS).
- Аппаратный сброс, инициированный через ножки процессора RESET или INIT, является специальным типом прерывания. При сигналах RESER или INIT процессор переключается в реальный режим.
- Запуск команды HLT в virtual-8086 режиме вызовет прерывание основной защиты.

Ввод/вывод в Virtual-8086 режиме. Процессор предоставляет защиту I/O для проведения I/O, который совместим со средой и прозрачен программам 8086. Разработчики могут использовать любой из следующих подходов:

- Защитить пространство адресов I/O и генерировать исключения для всех попыток выполнить I/O напрямую.
- Позволить 8086 программам выполнять I/O напрямую.
- Генерировать исключение на попытку к особым портам.
- Генерировать исключение на попытку к особым отображенным в памяти портам.

Обработка прерываний и исключений в virtual-8086 режиме. При генерации прерывания или исключения процессор переключается в защищенный режим и выполняет переход на обработчик защищенного режима. При завершении обработки прерывания процессор возвращает управление назад в 8086 программу. Обработчик прерывания может сам выполнить весь сервис, а может отразить его назад в virtual-8086 задачу для выполнения.

Начиная с процессоров Pentium, предоставляются дополнительные возможности, такие как вызов обработчиков напрямую, как в реальном режиме, поддержка виртуальных прерываний и т.д.

Вопросы для самопроверки

1. В каком режиме функционирования находится процессор после рестарта?
2. Как происходит преобразование логического адреса в физический в реальном режиме?
3. Чему равен предел сегмента в реальном режиме функционирования процессора?
4. Сколько векторов прерывания предусмотрено в реальном режиме и чему равен объем таблицы векторов прерывания?
5. По каким адресам расположены таблицы векторов прерывания в реальном и защищенном режимах?
6. Какие виды прерываний существуют?
7. Какие действия выполняет процессор при возникновении прерывания в реальном режиме?
8. Какие функции доступны операционной системе и программам пользователя в реальном режиме?
9. Поддержку каких функций обеспечивают 32-битовые процессоры Intel для системной разработки?
10. На какие части делится управление памятью в архитектуре Intel?
11. Для чего используют сегменты?
12. Дайте определения селектора и дескриптора сегмента.
13. В чем отличие линейного и физического адреса в защищенном режиме?

14. Дайте характеристику моделям сегментации.
15. Какие модели сегментации могут использовать трансляцию страниц?
16. Какой максимальный размер физического адресного пространства предоставляет архитектура Intel в защищенном режиме?
17. Расскажите о преобразовании логического адреса в линейный.
18. Какие типы сегментных дескрипторов распознает процессор Intel?
19. Какой размер имеют страницы линейного пространства?
20. Расскажите о преобразовании линейных адресов в физический.
21. Расскажите об инициализации защищенного режима.
22. Отчего зависит содержимое системных структур защищенного режима?
23. Какой флаг и в каком регистре отвечает за инициализацию трансляции страниц?
24. Как происходит переключение в защищенный режим?
25. Как происходит работа механизма защиты на сегментном уровне и уровне трансляции страниц?
26. Какой уровень привилегий является самым привилегированным?
27. Какие типы уровней привилегий распознает процессор для выполнения проверок между сегментами кода и сегментами данных?
28. На какую границу должен быть выровнен базовый адрес IDT для максимизации производительности?
29. Какие виды дескрипторов шлюзов может содержать IDT?
30. Расскажите, как проходит вызов процедуры прерывания в защищенном режиме.
31. Опишите структуру стека при вызове обработчиков прерываний в защищенном режиме.
32. Из каких частей состоит задача? Какие элементы определяют состояние текущей выполняемой задачи?
33. Для каких целей используется сегмент состояния задачи?
34. Опишите структуру регистра задачи.
35. Что собой представляет дескриптор шлюза задачи?

36. Опишите процесс переключения задач.
37. Какими способами могут быть отображены задачи в линейное и физическое адресное пространство?
38. Для каких целей используется режим Virtual-8086?
39. Из каких элементов состоит Virtual-8086 задача?
40. Опишите процесс входа и выхода в/из Virtual-8086 режим(а).

3. Ассемблер Intel 80x86

3.1 Назначение языка Ассемблера и создание исполняемой программы на языке Ассемблер

Язык ассемблера является языком низкого уровня. В этом языке имеется однозначное соответствие между командами языка и машинными кодами. Эта особенность позволяет использовать его для программирования вместо утомительного программирования в машинных кодах. Очевидно, что самая оптимальная программа как по быстродействию, так и по памяти может быть написана только на языке ассемблера (естественно, опытным программистом).

Однако не следует питать особых иллюзий по поводу ассемблера. Создание программ на ассемблере требует много времени и усилий по сравнению с написанием программ на языке высокого уровня. Тем более что код, генерируемый современными компиляторами с языков высокого уровня, лишь немногим уступает в производительности ассемблерному коду. Особенно это справедливо для математических задач. И еще один очень важный момент — это переносимость программ. Ассемблер — это машинный язык, и, следовательно, программы, написанные на нем, абсолютно не переносимы между разными типами компьютеров.

Наилучший способ — это комбинирование языков высокого уровня и ассемблера при создании программ. Для написания основного тела программы следует использовать язык высокого уровня (например, С), а для написания узких мест (где требуется максимальная производительность, например, отрисовка графических примитивов) и аппаратно зависимых процедур следует использовать ассемблер. Тем не мене, знание языка ассемблера необходимо, так как далеко не все можно написать на языках высокого уровня, особенно это касается архитектурных особенностей системы и работы с аппаратурой.

Для написания программ на языке ассемблере вы можете воспользоваться любым текстовым редактором, поддерживающим ко-

дировку ASCII-символов, например «Блокнот»/«Notepad» из ОС Windows или встроенным текстовым редактором FAR, DN, NC и др.

Для создания исполняемых файлов из программ, написанных на языке ассемблере, вам необходимо использовать компилятор TASM.EXE[2] и линковщик TLINK.EXE.

TASM.EXE компилирует программные модули ассемблера в объектные модули OBJ. А TLINK.EXE из нескольких модулей делает один исполняемый файл EXE или COM. Более подробный синтаксис использования TASM.EXE и TLINK.EXE можно получить запустив эти программы без параметров.

3.2. Структура программы на ассемблере

3.2.1 Синтаксис ассемблера

Программа на ассемблере представляет собой совокупность блоков памяти, называемых сегментами памяти. Программа может состоять из одного или нескольких таких блоков-сегментов. Каждый сегмент содержит совокупность предложений языка, каждое из которых занимает отдельную строку кода программы.

Предложения ассемблера бывают четырех типов [1]:

- *команды или инструкции*, представляющие собой символические аналоги машинных команд. В процессе трансляции инструкции ассемблера преобразуются в соответствующие системы команд микропроцессора;
- *макрокоманды* — оформляемые определенным образом предложения текста программы, замещаемые во время трансляции другими предложениями;
- *директивы*, являющиеся указанием транслятору ассемблера на выполнение некоторых действий. У директив нет аналогов в машинном представлении;
- *строки комментариев*, содержащие любые символы, в том числе и буквы русского алфавита. Комментарии игнорируются транслятором.

---

[2] Можно также использовать MASM.EXE

Предложения, составляющие программу, могут представлять собой синтаксическую конструкцию, соответствующую команде, макрокоманде, директиве или комментарию. Для того чтобы транслятор ассемблера мог распознать их, они должны формироваться по определенным синтаксическим правилам.

Формат предложений ассемблера[3]:

Оператор директивы [;текст комментария]
Оператор команды [;текст комментария]
Оператор макрокоманды [; текст комментария]

Формат директив:

[Имя] директива [операнд1…операндN][;комментарий]

Формат команд и макрокоманд

[Имя метки:] КОП [операнд1…операндN][;комментарий]

Здесь:

- *имя метки* — идентификатор, значением которого является адрес первого байта того предложения исходного текста программы, которое он обозначает;
- *имя* — идентификатор, отличающий данную директиву от других одноименных директив;
- *код операции (КОП) и директива* — это мнемонические обозначения соответствующей машинной команды, макрокоманды или директивы транслятора;
- *операнды* — части команды, макрокоманды или директивы ассемблера, обозначающие объекты, над которыми производятся действия. Операнды ассемблера описываются выражениями с числовыми и текстовыми константами, метками и идентификаторами переменных с использованием знаков операций и некоторых зарезервированных слов.

Допустимыми символами при написании текста программ являются:

---

[3] Квадратные скобки означают не обязательные параметры

- все латинские буквы: **A–Z**, **a–z**. При этом заглавные и строчные буквы считаются эквивалентными;
- цифры от **0** до **9**;
- знаки **?**, **@**, **$**, **_**, **&**;
- разделители **, . [ ] ( )** < > { } + / * **% ! ' " ?** \ = # ^.

Предложения ассемблера формируются из *лексем*, представляющих собой синтаксически неразделимые последовательности допустимых символов языка, имеющие смысл для транслятора.

Лексемами являются:

- *идентификаторы* — последовательности допустимых символов, использующиеся для обозначения таких объектов программы, как коды операций, имена переменных и названия меток. Правило записи идентификаторов заключается в следующем: идентификатор может состоять из одного или нескольких символов. В качестве символов можно использовать буквы латинского алфавита, цифры и некоторые специальные знаки — _, ?, $, @. Идентификатор не может начинаться символом цифры. Длина идентификатора может быть до 255 символов, хотя транслятор воспринимает лишь первые 32, а остальные — игнорирует. Регулировать длину возможных идентификаторов можно с использованием опции командной строки **mv**. Кроме этого, существует возможность указать транслятору на то, чтобы он различал прописные и строчные буквы либо игнорировал их различие (что и делается по умолчанию). Для этого применяются опции командной строки **/mu, /ml, /mx**;
- *цепочки символов* — последовательности символов, заключенные в одинарные или двойные кавычки;
- *целые числа* в одной из следующих систем счисления: *двоичной, десятичной, шестнадцатеричной*. Отождествление чисел при записи их в программах на ассемблере производится по определенным правилам:
    - **Десятичные числа** не требуют для своего отождествления указания каких-либо дополнительных символов, например 25 или 139.

- Для отождествления в исходном тексте программы **двоичных чисел** необходимо после записи нулей и единиц, входящих в их состав, поставить латинское “**b**”, например 10010101**b**.
- **Шестнадцатеричные числа** имеют больше условностей при своей записи:
  - *Во-первых*, они состоят из цифр **0...9**, строчных и прописных букв латинского алфавита **a**, **b**, **c**, **d**, **e**, **f** или **A**, **B**, **C**, **D**, **E**, **F**.
  - *Во-вторых*, у транслятора могут возникнуть трудности с распознаванием шестнадцатеричных чисел из-за того, что они могут состоять как из одних цифр 0...9 (например, 190845), так и начинаться с буквы латинского алфавита (например, **ef15**). Для того чтобы "объяснить" транслятору, что данная лексема не является десятичным числом или идентификатором, программист должен специальным образом выделять шестнадцатеричное число. Для этого на конце последовательности шестнадцатеричных цифр, составляющих шестнадцатеричное число, записывают латинскую букву “**h**”. Это обязательное условие. Если шестнадцатеричное число начинается с буквы, то перед ним записывается ведущий ноль: **0**ef15**h**.

3.2.2 Директивы сегментации

В ходе предыдущего обсуждения мы выяснили все основные правила записи команд и операндов в программе на ассемблере. Открытым остался вопрос о том, как правильно оформить последовательность команд, чтобы транслятор мог их обработать, а микропроцессор — выполнить.

При рассмотрении архитектуры микропроцессора мы узнали, что он имеет шесть сегментных регистров, посредством которых может одновременно работать:

- с одним сегментом кода;
- с одним сегментом стека;
- с одним сегментом данных;
- с тремя дополнительными сегментами данных.

Еще раз вспомним, что физически сегмент представляет собой область памяти, занятую командами и (или) данными, адреса которых вычисляются относительно значения в соответствующем сегментном регистре.

Синтаксическое описание сегмента на ассемблере представляет собой следующую конструкцию:

Имя сегмента SEGMENT [тип выравнивания] [тип комбинирования] [класс сегмента] [тип размера сегмента]
…
содержимое сегмента
…
Имя сегмента ENDS

Важно отметить, что функциональное назначение сегмента несколько шире, чем простое разбиение программы на блоки кода, данных и стека. Сегментация является частью общего механизма, связанного с концепцией модульного программирования. Она предполагает унификацию оформления объектных модулей, создаваемых компилятором, в том числе с разных языков программирования. Это позволяет объединять программы, написанные на разных языках. Именно для реализации различных вариантов такого объединения и предназначены операнды в директиве SEGMENT. Рассмотрим их подробнее:

- *Атрибут выравнивания сегмента* (тип выравнивания) сообщает компоновщику о том, что нужно обеспечить размещение начала сегмента на заданной границе. Это важно, поскольку при правильном выравнивании доступ к данным в процессорах i80x86 выполняется быстрее. Допустимые значения этого атрибута следующие:
    - BYTE – выравнивание не выполняется. Сегмент может начинаться с любого адреса памяти;
    - WORD – сегмент начинается по адресу, кратному двум, то есть последний (младший) значащий бит физического адреса равен 0 (выравнивание на границу слова);
    - DWORD – сегмент начинается по адресу, кратному четырем, то есть два последних (младших) значащих бита равны 0 (выравнивание на границу двойного слова);

- PARA – сегмент начинается по адресу, кратному 16, то есть последняя шестнадцатеричная цифра адреса должна быть 0h (выравнивание на границу параграфа);
- PAGE – сегмент начинается по адресу, кратному 256, то есть две последние шестнадцатеричные цифры должны быть 00h (выравнивание на границу 256-байтной страницы);
- MEMPAGE – сегмент начинается по адресу, кратному 4 Кбайт, то есть три последние шестнадцатеричные цифры должны быть 000h (адрес следующей 4-Кбайтной страницы памяти).

По умолчанию тип выравнивания имеет значение PARA.

- *Атрибут комбинирования сегментов* (комбинаторный тип) сообщает компоновщику, как нужно комбинировать сегменты различных модулей, имеющие одно и то же имя. Значениями атрибута комбинирования сегмента могут быть:
  - PRIVATE – сегмент не будет объединяться с другими сегментами с тем же именем вне данного модуля;
  - PUBLIC – заставляет компоновщик соединить все сегменты с одинаковыми именами. Новый объединенный сегмент будет целым и непрерывным. Все адреса (смещения) объектов, а это могут быть, в зависимости от типа сегмента, команды и данные, будут вычисляться относительно начала этого нового сегмента;
  - COMMON – располагает все сегменты с одним и тем же именем по одному адресу. Все сегменты с данным именем будут перекрываться и совместно использовать память. Размер полученного в результате сегмента будет равен размеру самого большого сегмента;
  - AT xxxx – располагает сегмент по абсолютному адресу параграфа (параграф – объем памяти, кратный 16, поэтому последняя шестнадцатеричная цифра адреса параграфа равна 0);
  - STACK – определение сегмента стека. Заставляет компоновщик соединить все одноименные сегменты и вычислять адреса в этих сегментах относительно регистра ss.

По умолчанию атрибут комбинирования принимает значение PRIVATE.

- *Атрибут класса сегмента* (тип класса) – это заключенная в кавычки строка, помогающая компоновщику определить соответствующий порядок следования сегментов при собирании программы из сегментов нескольких модулей. Компоновщик объединяет в памяти все сегменты с одним и тем же именем класса (имя класса, в общем случае, может быть любым, но лучше, если оно будет отражать функциональное назначение сегмента);
- *Атрибут размера сегмента*. Для процессоров i80386 и выше сегменты могут быть 16 или 32-разрядными. Это влияет, прежде всего, на размер сегмента и порядок формирования физического адреса внутри него. Атрибут может принимать следующие значения:
    - USE16 – это означает, что сегмент допускает 16-разрядную адресацию. При формировании физического адреса может использоваться только 16-разрядное смещение. Соответственно, такой сегмент может содержать до 64 Кбайт кода или данных;
    - USE32 – сегмент будет 32-разрядным. При формировании физического адреса может использоваться 32-разрядное смещение. Поэтому такой сегмент может содержать до 4 Гбайт кода или данных.

Все сегменты сами по себе равноправны, так как директивы SEGMENT и ENDS не содержат информации о функциональном назначении сегментов. Для того чтобы использовать их как сегменты кода, данных или стека, необходимо предварительно сообщить транслятору об этом, для чего используют специальную директиву ASSUME. Эта директива сообщает транслятору о том, какой сегмент, к какому сегментному регистру привязан. В свою очередь, это позволит транслятору корректно связывать символические имена, определенные в сегментах.

Далее приведем пример программы с использованием стандартных директив сегментации:

```
data segment para public 'data' ;сегмент данных
      message db 'Hello World,$' ; описание строки
data ends

stk segment stack
      db 256 dup ('?') ; размер сегмента стека
stk ends

code segment para public 'code' ; начало сегмента
 ; кода
main proc ;начало процедуры main
      assume cs:code,ds:data,ss:stk
 mov  ax,data ; адрес сегмента данных
 ; в регистр ax
      mov   ds,ax ; ax в ds

      ...

      mov   ah,9
      mov   dx,offset message
      int   21h ; ah=9 функция 21h прерывания
 ; выводит строку на экран, адрес
 ; которой хранится в регистре dx,
 ; строка должна обязательно
 ; заканчиваться символом $
      ...

      mov   ax,4c00h ; пересылка 4c00h в регистр ax
      int   21h ; вызов прерывания с номером 21h
main  endp ; конец процедуры main
code  ends ; конец сегмента кода
end   main ; конец программы с точкой входа main
```

Для простых программ, содержащих по одному сегменту для кода, данных и стека, хотелось бы упростить ее описание. Для этого в трансляторы MASM и TASM ввели возможность использования *упрощенных директив сегментации*. Но здесь возникла проблема, связанная с тем, что необходимо было как-то компенсировать невозможность напрямую управлять разме-

щением и комбинированием сегментов. Для этого, совместно с упрощенными директивами сегментации, стали использовать директиву указания модели памяти **MODEL**, которая частично стала управлять размещением сегментов и выполнять функции директивы **ASSUME** (поэтому при использовании упрощенных директив сегментации директиву ASSUME можно не использовать). Эта директива связывает сегменты, которые в случае использования упрощенных директив сегментации имеют предопределенные имена с сегментными регистрами (хотя явно инициализировать *ds* все равно придется).

Теперь перепишем вышеприведенную программу с использованием упрощенных директив сегментации.

```
masm ;режим работы TASM: ideal или masm
model small ; модель памяти

.data ; сегмент данных
 message db 'Hello World,$' ; описание строки

.stack ;сегмент стека
 db 256 dup ('?') ; сегмент стека

.code ;сегмент кода
main proc ; начало процедуры main
 mov ax,@data ; заносим адрес сегмента данных в
 ; регистр ax
 mov ds,ax ;ax в ds

      ...

 mov  ah,9
 mov  dx,offset message
 int  21h ; ah=9 функция 21h прерывания
 ; выводит строку на экран, адрес
 ; которой хранится в регистре dx
      ...

 mov ax,4c00h ; пересылка 4c00h в регистр ax
 int 21h ; вызов прерывания с номером 21h
main endp ; конец процедуры main
end main ; конец программы с точкой входа main
```

Обязательным параметром директивы MODEL является *модель памяти*. Этот параметр определяет модель сегментации памяти для программного модуля. Предполагается, что программный модуль может иметь только определенные типы сегментов, которые определяются упомянутыми нами ранее *упрощенными директивами описания сегментов*. Эти директивы приведены в таблице 3.1.

Таблица 3.1–Упрощенные директивы определения сегмента

Формат директивы (режим MASM)	**Формат директивы (режим IDEAL)**	**Назначение**
.CODE [имя]	CODESEG [имя]	Начало или продолжение сегмента кода
.DATA	DATASEG	Начало или продолжение сегмента инициализированных данных. Также используется для определения данных типа near
.CONST	CONST	Начало или продолжение сегмента постоянных данных (констант) модуля
.FARDATA [имя]	FARDATA [имя]	Начало или продолжение сегмента инициализированных данных типа far

Наличие в некоторых директивах параметра **[имя]** говорит о том, что возможно определение нескольких сегментов этого типа. С другой стороны, наличие нескольких видов сегментов данных обусловлено требованием обеспечения совместимости с некоторыми компиляторами языков высокого уровня, которые создают разные сегменты данных для инициализированных и неинициализированных данных, а также констант.

При использовании директивы **MODEL** транслятор делает доступными несколько идентификаторов, к которым можно обращаться во время работы программы, с тем, чтобы получить информацию о тех или иных характеристиках данной модели памяти (таблица 3.3). Перечислим эти идентификаторы и их значения (табл. 3.2).

Таблица 3.2–Модели памяти

Модель	**Тип кода**	**Тип дан-ных**	**Назначение модели**
TINY	near	near	Код и данные объединены в одну группу с именем DGROUP. Используется для создания программ формата .com.
SMALL	near	near	Код занимает один сегмент, данные объединены в одну группу с именем DGROUP. Эту модель обычно используют для большинства программ на ассемблере
MEDIUM	far	near	Код занимает несколько сегментов, по одному на каждый объединяемый программный модуль. Все ссылки на передачу управления — типа far. Данные объединены в одной группе; все ссылки на них — типа near
COMPACT	near	far	Код в одном сегменте; ссылка на данные — типа far
LARGE	far	far	Код в нескольких сегментах, по одному на каждый объединяемый программный модуль

Параметр модификатор директивы MODEL позволяет уточнить некоторые особенности использования выбранной модели памяти (табл. 3.3).

Таблица 3.3–Модификаторы модели памяти

Значение модификатора	**Назначение**
use16	Сегменты выбранной модели используются как 16-битные (если соответствующей директивой указан процессор i80386 или i80486)
use32	Сегменты выбранной модели используются как 32-битные (если соответствующей директивой указан процессор i80386 или i80486)
dos	Программа будет работать в MS-DOS

Необязательные параметры — язык и модификатор языка, — определяют некоторые особенности вызова процедур. Необходимость в использовании этих параметров появляется при написании и связывании программ на различных языках программирования.

Описанные нами стандартные и упрощенные директивы сегментации не исключают друг друга. Стандартные директивы используются, когда программист желает получить полный контроль над размещением сегментов в памяти и их комбинированием с сегментами других модулей.

Упрощенные директивы целесообразно использовать для простых программ и программ, предназначенных для связывания с программными модулями, написанными на языках высокого уровня. Это позволяет компоновщику эффективно связывать модули разных языков за счет стандартизации связей и управления.

3.2.3 Создание COM-программ

Все вышеприведенные директивы сегментации и примеры программ предназначены для создания программ в EXE-формате[4]. Компоновщик LINK автоматически генерирует особый формат для EXE-файлов, в котором присутствует специальный начальный блок (заголовок) размером не менее 512 байт.

Для выполнения можно также создавать COM-файлы. Примером часто используемого COM-файла является COMMAND.COM.

Размер программы. EXE-программа может иметь любой размер, в то время как COM-файл ограничен размером одного сегмента и не превышает 64К. COM-файл всегда меньше, чем соответствующий EXE-файл; одна из причин этого — отсутствие в COM-файле 512-байтового начального блока EXE-файла.

Сегмент стека. В EXE-программе определяется сегмент стека, в то время как COM-программа генерирует стек автоматически. Таким образом, при создании ассемблерной програм-

---

[4] За исключением модели памяти TINY при использовании упрощенных директив сегментации.

мы, которая будет преобразована в COM-файл, стек должен быть опущен.

Сегмент данных. В EXE-программе обычно определяется сегмент данных, а регистр DS инициализируется адресом этого сегмента. В COM-программе все данные должны быть определены в сегменте кода. Ниже будет показан простой способ решения этого вопроса.

Инициализация. EXE-программа записывает нулевое слово в стек и инициализирует регистр DS. Так как COM-программа не имеет ни стека, ни сегмента данных, то эти шаги отсутствуют.

Когда COM-программа начинает работать, все сегментные регистры содержат адрес префикса программного сегмента (PSP), — 256-байтового (шест. 100) блока, который резервируется операционной системой DOS непосредственно перед COM или EXE программой в памяти. Так как адресация начинается с шестнадцатиричного смещения 100 от начала PSP, то в программе после оператора SEGMENT кодируется директива ORG 100H.

Обработка. Для программ в EXE и COM форматах выполняется ассемблирование для получения OBJ-файла, и компоновка для получения EXE-файла. Если программа создается для выполнения как EXE-файл, то ее уже можно выполнить. Если же программа создается для выполнения как COM-файл, то компоновщиком будет выдано сообщение:

Warning: No STACK Segment

(Предупреждение: Сегмент стека не определен)

Ниже приведем пример COM-программы:

```
CSEG  Segment 'Code'
assume CS:CSEG,DS:CSEG,ES:CSEG,SS:CSEG
org   100h
start:
      ...

      mov   ah,9
      mov   dx,offset message
      int   21h ; ah=9 функция 21h прерывания
 ; выводит строку на экран, адрес
```

```
 ; которой хранится в регистре dx
      ...

 int 20h ; выход из COM-программы
 message db 'Hello World,$' ; описание строки
ends
end start
```

3.2.4 Использование различных способов адресации

Имеющиеся способы адресации можно условно разделить на следующие группы: регистровая, непосредственная и с указанием адреса памяти. Последняя группа включает в себя несколько способов адресации: прямая, базовая, индексная, базово-индексная, базовая со смещением, индексная со смещением, базово-индексная со смещением.

Регистровая адресация.

Способ применим ко всем программно-адресуемым регистрам процессора. Операнд при этом находится в регистре.

push DS ;сохранить содержимое DS в стеке.

Непосредственная адресация.

Операнд может быть числом, кодом ASCII, адресом, иметь символьное обозначение.

mov AX, 0101h ;16-ричное число помещается в регистр AX;
mov DL, '1' ;код ASCII символа '1' загружается в DL.

Адресация с указанием адреса памяти.

Прямая адресация.

В команде указывается либо символическое обозначение ячейки памяти, над содержимым которой должна выполниться операция, либо адрес непосредственно можно указать в качестве операнда. Во втором случае нужно предварительно настроить какой-либо сегментный регистр на начало нужного участка памяти, дальше через символ «:» в квадратных скобках указывается смещение в регистре.

mov DL, memx ;байт памяти с названием memx
;засылается в регистр DL;
mov DL,ES:[2] ;в регистр DL засылается байт
; памяти с абсолютным

```
;адресом, начало сегмента памяти
;находится в ES,
;смещение в сегменте — в квадратных скобках.
```

Косвенная адресация.

Базовая адресация.

В команде указывается относительный адрес ячейки памяти, над содержимым которой должна выполниться операция, Этот адрес находится в регистре, обозначение которого заключается в квадратные скобки, или вычисляется сложением содержимого двух регистров.

Используются регистры:

BX (базовый регистр), в этом случае адрес сегмента данных, если не задан явно, предположительно хранится в DS (регистр сегмента данных);

BP (регистр — указатель базы), в этом случае в качестве сегментного регистра используется регистр SS (регистр сегмента стека).

```
mov DL, ES:[BX] ;данные из ячейки памяти,
                ;сегмент которой задан в ES,
                ;а смещение – в BX, помещаются
                ;в регистр DL;
mov DL, [BX]    ;команда аналогична предыдущей,
                ;за исключением
                ;того, что сегментный адрес ячейки
                ;памяти находится в DS.
```

Индексная адресация.

Аналогична базовой.

Используются регистры:

SI (регистр — индекс источника) и DI (регистр — индекс приемника). В этом случае адрес сегмента данных, если не задан явно, хранится в DS (регистр сегмента данных).

```
mov AL, [DI]    ;сегментный адрес ячейки памяти
                ;находится в DS.
```

Базовая адресация памяти со смещением.

Относительный адрес операнда определяется суммой содержимого регистра (BX, BP) и указанного в команде числа – смещения. При использовании регистра BP в качестве сегмент-

ного регистра по умолчанию подразумевается регистр SS. При использовании BX — регистр DS.

```
mas db 1,2,3,4,5        ;массив символов
mov BX, offset mas      ;в BX – относительный адрес
                        ;ячейки mas
mov DL, 2[BX]           ;Относительный адрес нужного
                        ;элемента массива
                        ;вычисляется как сумма содержимого
                        ;BX и смещения,
                        ;заданного константой 2
или
mov DL,[BX+2]
или
mov DL,[BX]+2
```

Индексная адресация памяти со смещением.

Аналогична базовой адресации со смещением.

Используются регистры:

SI (регистр — индекс источника) и DI (регистр — индекс приемника). В этом случае адрес сегмента данных, если не задан явно, хранится в DS (регистр сегмента данных).

```
mov DL, 2[SI]     ;Относительный адрес нужного элемента
                  ;памяти вычисляется как сумма содержимого
                  ;SI и смещения, заданного константой 2
или
mov DL,[SI+2]
или
mov DL,[SI]+2
```

Базово-индексная адресация памяти.

Относительный адрес операнда определяется суммой содержимого базового и индексного регистров. Допускается использование следующих пар:

[BX][SI], [BX][DI], [BP][SI], [BP][DI]. Если в качестве базового регистра используется BX , то в качестве сегментного подразумевается DS. Если в качестве базового регистра — BP, то в качестве сегментного — SS. При необходимости сегментный регистр можно указать явно.

```
mov BX, [BP][SI]  ;в BX засылается слово из стека
```

```
                        ;(сегментный адрес в SS), смещение
                        ;вычисляется как сумма содержимого BP и SI.
```

Базово-индексная адресация со смещением.

Относительный адрес операнда определяется суммой содержимого базового и индексного регистров и дополнительного смещения. При адресации используются те же пары регистров, что и при базово-индексной адресации. Также если в качестве базового регистра используется BX , то в качестве сегментного подразумевается DS. Если в качестве базового регистра — BP, то в качестве сегментного — SS. При необходимости сегментный регистр можно указать явно.

```
mov mas[BX][SI],10      ;Число 10 засылается в ячейку
                        ;памяти, сегментный адрес которой
                        ;хранится в DS, а смещение равно
                        ;сумме содержимого BX и SI
                        ;и смещения ячейки mas.
```

Описание методов адресации можно найти в [3].

3.3 Изучение функций ввода/вывода, арифметических и логических команд

3.3.1 Функции прерываний ввода/вывода

В операционной системе существует большая группа функций 21h прерывания (прерывания DOS). Небольшую часть этих функций составляют функции ввода вывода информации.

Для вызова какого-либо прерывания необходимо:

- в регистр AH занести номер функции прерывания;
- в зависимости от типа прерывания в какие-либо регистры занести дополнительные параметры;
- использовать команду int с указанием номера прерывания.

С функцией 9h прерывания 21h вы познакомились на предыдущей разделе.

Для вызова функции ввода символа используют 1h-функцию 21h прерывания. После нажатия символа на клавиатуре в регистре AL сохраняется код ASCII нажатого символа.

Для вывода символа на экран можно воспользоваться функцией 2h прерывания 21h. В регистр AL помещается ASCII-код символа и вызывается прерывание 21h.

Дополнительную информацию по различным функциям прерываний операционной системы можно взять в электронном справочнике thelp.

3.3.2 Арифметические команды

Сложение двоичных чисел без знака. Микропроцессор выполняет сложение операндов по правилам сложения двоичных чисел. Проблем не возникает до тех пор, пока значение результата не превышает размерности поля операнда. Например, при сложении операндов размером в байт результат не должен превышать число 255. Если это происходит, то результат оказывается неверным. К примеру, выполним сложение: 254 + 5 = 259 в двоичном виде. 11111110 + 0000101 = 1 00000011. Результат вышел за пределы восьми бит и правильное его значение укладывается в 9 бит, а в 8-битовом поле операнда осталось значение 3, что, конечно, неверно. В микропроцессоре этот исход сложения прогнозируется и предусмотрены специальные средства для фиксирования подобных ситуаций и их обработки. Так, для фиксирования ситуации выхода за разрядную сетку результата, как в данном случае, предназначен флаг переноса cf. Он располагается в бите 0 регистра флагов eflags/flags. Именно установкой этого флага фиксируется факт переноса единицы из старшего разряда операнда. Естественно, что программист должен предусматривать возможность такого исхода операции сложения и средства для корректировки. Это предполагает включение участков кода после операции сложения, в которых анализируется флаг cf. Анализ этого флага можно провести различными способами. Самый простой и доступный — использовать команду условного перехода jcc. Эта команда в качестве операнда имеет имя метки в текущем сегменте кода. Переход на эту метку осуществляется в случае, если в результате работы предыдущей команды флаг cf установился в 1. В системе команд микропроцессора имеются три команды двоичного сложения:

- inc операнд — операция инкремента, то есть увеличения значения операнда на 1;
- add операнд_1,операнд_2 — команда сложения с принципом действия: операнд_1 = операнд_1 + операнд_2;
- adc операнд_1,операнд_2 — команда сложения с учетом флага переноса cf: операнд_1 = операнд_1 + операнд_2 + значение_cf.

Обратите внимание на последнюю команду — это команда сложения, учитывающая перенос единицы из старшего разряда. Механизм появления такой единицы мы уже рассмотрели. Таким образом, команда adc является средством микропроцессора для сложения длинных двоичных чисел, размерность которых превосходит поддерживаемые микропроцессором длины стандартных полей.

Рассмотрим пример вычисления суммы чисел:

```
<3> ;prg1
<3> masm
<3> model small
<4> stack 256
<5> .data
<6> a db 254
<7> .code ;сегмент кода
<8> main:
<9> mov ax,@data
<10> mov ds,ax
<11> ...
<12> xor ax,ax
<13> add al,17
<14> add al,a
<15> jnc m1;если нет переноса, то перейти
 ;на m1
<16> adc ah,0;в ax сумма с учетом переноса
<17> m1: ...
<18> exit:
<19> mov ax,4c00h ;стандартный выход
<20> int 21h
<21> end main ;конец программы
```

В строках 13–14 создана ситуация, когда результат сложения выходит за границы операнда. Эта возможность учитывает-

ся строкой 15, где команда jnc (хотя можно было обойтись и без нее) проверяет состояние флага cf. Если он установлен в 1, то это признак того, что результат операции получился больше по размеру, чем размер операнда, и для его корректировки необходимо выполнить некоторые действия. В данном случае мы просто полагаем, что границы операнда расширяются до размера AX, для чего учитываем перенос в старший разряд командой ADC (строка 15).

Сложение двоичных чисел со знаком. Микропроцессор не подозревает о различии между числами со знаком и без знака. Вместо этого у него есть средства фиксирования возникновения характерных ситуаций, складывающихся в процессе вычислений. Некоторые из них мы рассмотрели при обсуждении сложения чисел без знака:

- флаг переноса cf, установка которого в 1 говорит о том, что произошел выход за пределы разрядности операндов;
- команду adc, которая учитывает возможность такого выхода (перенос из младшего разряда).

Другое средство — это регистрация состояния старшего (знакового) разряда операнда, которая осуществляется с помощью флага переполнения of в регистре eflags (бит 11).

Дополнительно к флагу of при переносе из старшего разряда устанавливается в 1 и флаг переноса cf. Так как микропроцессор не знает о существовании чисел со знаком и без знака, то вся ответственность за правильность действий с получившимися числами ложится на программиста. Проанализировать флаги cf и of можно командами условного перехода jc\jnc и jo\jno соответственно.

Что же касается команд сложения чисел со знаком, то они те же, что и для чисел без знака.

Вычитание двоичных чисел без знака. Как и при анализе операции сложения, порассуждаем над сутью процессов, происходящих при выполнении операции вычитания. Если уменьшаемое больше вычитаемого, то проблем нет, — разность положительна, результат верен. Если уменьшаемое меньше вычитаемого, возникает проблема: результат меньше 0, а это уже число со знаком. В этом случае результат необходимо завер-

нуть. Что это означает? При обычном вычитании (в столбик) делают заем 1 из старшего разряда. Микропроцессор поступает аналогично, то есть занимает 1 из разряда, следующего за старшим, в разрядной сетке операнда.

Таким образом, после команды вычитания чисел без знака нужно анализировать состояние флага cf. Если он установлен в 1, то это говорит о том, что произошел заем из старшего разряда и результат получился в дополнительном коде.

Аналогично командам сложения, группа команд вычитания состоит из минимально возможного набора. Эти команды выполняют вычитание по алгоритмам, которые мы сейчас рассматриваем, а учет особых ситуаций должен производиться самим программистом. К командам вычитания относятся:

- dec операнд — операция декремента, то есть уменьшения значения операнда на 1;
- sub операнд_1,операнд_2 — команда вычитания; ее принцип действия:
  операнд_1 = операнд_1 — операнд_2;
- sbb операнд_1,операнд_2 — команда вычитания с учетом заема (флага cf): операнд_1 = операнд_1 — операнд_2 — значение_cf.

Как видите, среди команд вычитания есть команда sbb, учитывающая флаг переноса cf. Эта команда подобна adc, но теперь уже флаг cf выполняет роль индикатора заема 1 из старшего разряда при вычитании чисел.

Рассмотрим пример программной обработки ситуации:

```
<1> ;prg2
<2> masm
<3> model small
<4> stack 256
<5> .data
<6> .code ;сегмент кода
<7> main: ;точка входа в программу
<8> ...
<9> xor ax,ax
<10> mov al,5
<11> sub al,10
<12> jnc m1 ;нет переноса?
<13> neg al ;в al модуль результата
```

```
<14> m1: ...
<15> exit:
<16> mov ax,4c00h ;стандартный выход
<17> int 21h
<18> end main ;конец программы
```

В этом примере в строке 11 выполняется вычитание. С указанными для этой команды вычитания исходными данными результат получается в дополнительном коде (отрицательный). Для того чтобы преобразовать результат к нормальному виду (получить его модуль), применяется команда neg, с помощью которой получается дополнение операнда. В нашем случае мы получили дополнение дополнения или модуль отрицательного результата. А тот факт, что это на самом деле число отрицательное, отражен в состоянии флага cf. Дальше все зависит от алгоритма обработки.

Вычитание двоичных чисел со знаком. Здесь все несколько сложнее. Последний пример показал то, что микропроцессору незачем иметь два устройства — сложения и вычитания. Достаточно наличия только одного — устройства сложения. Но для вычитания способом сложения чисел со знаком в дополнительном коде необходимо представлять оба операнда — и уменьшаемое, и вычитаемое. Результат тоже нужно рассматривать как значение в дополнительном коде. Но здесь возникают сложности. Прежде всего, они связаны с тем, что старший бит операнда рассматривается как знаковый.

Отследить ситуацию переполнения мантиссы можно по содержимому флага переполнения of. Его установка в 1 говорит о том, что результат вышел за диапазон представления знаковых чисел (то есть изменился старший бит) для операнда данного размера, и программист должен предусмотреть действия по корректировке результата.

Умножение чисел без знака. Для умножения чисел без знака предназначена команда

mul сомножитель_1.

Как видите, в команде указан всего лишь один операнд-сомножитель. Второй операнд — сомножитель_2 — задан неявно. Его местоположение фиксировано и зависит от размера сомножителей. Так как в общем случае результат умножения

больше, чем любой из его сомножителей, то его размер и местоположение должны быть тоже определены однозначно. Варианты размеров сомножителей и размещения второго операнда и результата приведены в таблице 3.4.

Таблица 3.4–Расположение операндов и результата при умножении

сомножитель_1	**сомножитель_2**	**Результат**
Байт	al	16 бит в ax: al – младшая часть результата; ah – старшая часть результата
Слово	ax	32 бит в паре dx:ax: ax – младшая часть результата; dx – старшая часть результата
Двойное слово	eax	64 бит в паре edx:eax: eax – младшая часть результата; edx – старшая часть результата

Из таблицы видно, что произведение состоит из двух частей и в зависимости от размера операндов размещается в двух местах – на месте сомножитель_2 (младшая часть) и в дополнительном регистре ah, dx, edx (старшая часть). Как же динамически (то есть во время выполнения программы) узнать, что результат достаточно мал и уместился в одном регистре или что он превысил размерность регистра, и старшая часть оказалась в другом регистре? Для этого привлекаются уже известные нам по предыдущему обсуждению флаги переноса cf и переполнения of:

- если старшая часть результата нулевая, то после операции произведения флаги cf = 0 и of = 0;
- если же эти флаги ненулевые, то это означает, что результат вышел за пределы младшей части произведения и состоит из двух частей, что и нужно учитывать при дальнейшей работе.

Умножение чисел со знаком. Для умножения чисел со знаком предназначена команда:

imul операнд_1[,операнд_2,операнд_3].

Эта команда выполняется так же, как и команда mul. Отличительной особенностью команды imul является только формирование знака.

Если результат мал и умещается в одном регистре (то есть если cf = of = 0), то содержимое другого регистра (старшей части) является расширением знака — все его биты равны старшему биту (знаковому разряду) младшей части результата.

В противном случае, (если cf = of = 1) знаком результата является знаковый бит старшей части результата, а знаковый бит младшей части является значащим битом двоичного кода результата.

Если вы посмотрите описание команды imul, то увидите, что она допускает более широкие возможности по заданию местоположения операндов. Это сделано для удобства использования.

Деление чисел без знака. Для деления чисел без знака предназначена команда:

div делитель.

Делитель может находиться в памяти или в регистре и иметь размер 8, 16 или 32 бит. Местонахождение делимого фиксировано и так же, как в команде умножения, зависит от размера операндов. Результатом команды деления являются значения частного и остатка.

Варианты местоположения и размеров операндов операции деления показаны в таблице 3.5.

Таблица 3.5–Расположение операндов и результата при делении

Делимое	Делитель	Частное	Остаток
16 бит в регистре ax	Байт регистр или ячейка памяти	Байт в регистре al	Байт в регистре ah
32 бит dx – старшая часть ax – младшая часть	Слово 16 бит регистр или ячейка памяти	Слово 16 бит в регистре ax	Слово 16 бит в регистре dx
64 бит edx – старшая часть eax – младшая часть	Двойное слово 32 бит регистр или ячейка памяти	Двойное слово 32 бит в регистре eax	Двойное слово 32 бит в регистре edx

После выполнения команды деления содержимое флагов неопределенно, но возможно возникновение прерывания с номером 0, называемого “деление на ноль”. Этот вид прерывания относится к так называемым исключениям. Эта разновидность прерываний возникает внутри микропроцессора из-за некоторых аномалий во время вычислительного процесса. Прерывание 0, “деление на ноль”, при выполнении команды div может возникнуть по одной из следующих причин:

- делитель равен нулю;
- частное не входит в отведенную под него разрядную сетку, что может случиться в следующих случаях:

– при делении делимого величиной в слово на делитель величиной в байт, причем значение делимого в более чем 256 раз больше значения делителя;

– при делении делимого величиной в двойное слово на делитель величиной в слово, причем значение делимого в более чем 65 536 раз больше значения делителя;

– при делении делимого величиной в учетверенное слово на делитель величиной в двойное слово, причем значение делимого в более чем 4 294 967 296 раз больше значения делителя.

Деление чисел со знаком. Для деления чисел со знаком предназначена команда

idiv делитель

Для этой команды справедливы все рассмотренные положения, касающиеся команд и чисел со знаком. Отметим лишь особенности возникновения исключения 0, “деление на ноль”, в случае чисел со знаком. Оно возникает при выполнении команды idiv по одной из следующих причин:

- делитель равен нулю;
- частное не входит в отведенную для него разрядную сетку.

Последнее, в свою очередь, может произойти:

– при делении делимого величиной в слово со знаком на делитель величиной в байт со знаком, причем значение делимого в более чем 128 раз больше значения делителя (таким образом, частное не должно находиться вне диапазона от –128 до +127);

– при делении делимого величиной в двойное слово со знаком на делитель величиной в слово со знаком, причем значение

делимого в более чем 32 768 раз больше значения делителя (таким образом, частное не должно находиться вне диапазона от –32 768 до +32 768);

– при делении делимого величиной в учетверенное слово со знаком на делитель величиной в двойное слово со знаком, причем значение делимого в более чем 2 147 483 648 раз больше значения делителя (таким образом, частное не должно находиться вне диапазона от –2 147 483 648 до +2 147 483 647).

3.3.3 Логические команды

В системе команд микропроцессора есть следующий набор команд, поддерживающих работу с логическими данными:

and операнд_1, операнд_2 — операция логического умножения. Команда выполняет поразрядно логическую операцию И (конъюнкцию) над битами операндов операнд_1 и операнд_2. Результат записывается на место операнд_1.

or операнд_1, операнд_2 — операция логического сложения. Команда выполняет поразрядно логическую операцию ИЛИ (дизъюнкцию) над битами операндов операнд_1 и операнд_2. Результат записывается на место операнд_1.

xor операнд_1,операнд_2 — операция логического исключающего сложения. Команда выполняет поразрядно логическую операцию исключающего ИЛИ над битами операндов операнд_1 и операнд_2. Результат записывается на место операнд_1.

test операнд_1,операнд_2 — операция “проверить” (способом логического умножения). Команда выполняет поразрядно логическую операцию И над битами операндов операнд_1 и операнд_2. Состояние операндов остается прежним, изменяются только флаги zf, sf, и pf, что дает возможность анализировать состояние отдельных битов операнда без изменения их состояния.

not операнд — операция логического отрицания. Команда выполняет поразрядное инвертирование (замену значения на обратное) каждого бита операнда. Результат записывается на место операнда.

3.3.4 Команды сдвига

Команды этой группы также обеспечивают манипуляции над отдельными битами операндов, но иным способом, чем логические команды, рассмотренные выше. Все команды сдвига перемещают биты в поле операнда влево или вправо в зависимости от кода операции.

Количество сдвигаемых разрядов — счетчик_сдвигов — располагается, как видите, на месте второго операнда и может задаваться двумя способами:

- статически, что предполагает задание фиксированного значения с помощью непосредственного операнда;
- динамически, что означает занесение значения счетчика сдвигов в регистр cl перед выполнением команды сдвига.

Команды линейного сдвига делятся на два подтипа:

- команды логического линейного сдвига;
- команды арифметического линейного сдвига.

К командам логического линейного сдвига относятся:

- shl операнд,счетчик_сдвигов (Shift Logical Left) — логический сдвиг влево. Содержимое операнда сдвигается влево на количество битов, определяемое значением счетчик_сдвигов. Справа (в позицию младшего бита) вписываются нули;
- shr операнд,счетчик_сдвигов (Shift Logical Right) — логический сдвиг вправо. Содержимое операнда сдвигается вправо на количество битов, определяемое значением счетчик_сдвигов. Слева (в позицию старшего, знакового бита) вписываются нули.

Команды арифметического линейного сдвига отличаются от команд логического сдвига тем, что они особым образом работают со знаковым разрядом операнда:

- sal операнд,счетчик_сдвигов (Shift Arithmetic Left) — арифметический сдвиг влево. Содержимое операнда сдвигается влево на количество битов, определяемое значением счетчик_сдвигов. Справа (в позицию младшего бита) вписываются нули. Команда sal не сохраняет знака, но устанавливает флаг cf в случае смены знака очередным выдвигаемым би-

том. В остальном команда sal полностью аналогична команде shl;

- sar операнд,счетчик_сдвигов (Shift Arithmetic Right) — арифметический сдвиг вправо. Содержимое операнда сдвигается вправо на количество битов, определяемое значением счетчик_сдвигов. Слева в операнд вписываются нули. Команда sar сохраняет знак, восстанавливая его после сдвига каждого очередного бита.

К командам циклического сдвига относятся команды, сохраняющие значения сдвигаемых бит. Есть два типа команд циклического сдвига:

- команды простого циклического сдвига;
- команды циклического сдвига через флаг переноса cf.

К командам простого циклического сдвига относятся:

- rol операнд,счетчик_сдвигов (Rotate Left) — циклический сдвиг влево. Содержимое операнда сдвигается влево на количество бит, определяемое операндом счетчик_сдвигов. Сдвигаемые влево биты записываются в тот же операнд справа.
- ror операнд,счетчик_сдвигов (Rotate Right) — циклический сдвиг вправо. Содержимое операнда сдвигается вправо на количество бит, определяемое операндом счетчик_сдвигов. Сдвигаемые вправо биты записываются в тот же операнд.

К командам циклического сдвига через флаг переноса cf относятся следующие:

- rcl операнд,счетчик_сдвигов (Rotate through Carry Left) — циклический сдвиг влево через перенос. Содержимое операнда сдвигается влево на количество бит, определяемое операндом счетчик_сдвигов. Сдвигаемые биты поочередно становятся значением флага переноса cf.
- rcr операнд,счетчик_сдвигов (Rotate through Carry Right) — циклический сдвиг вправо через перенос. Содержимое операнда сдвигается вправо на количество бит, определяемое операндом счетчик_сдвигов. Сдвигаемые биты поочередно становятся значением флага переноса cf.

3.4 Модульное программирование

3.4.1 Процедуры на языке ассемблера

Для оформления процедур существуют специальные директивы proc/endp и машинная команда ret — возврат управления из процедуры вызывающей программе.

Формат команды ret: ret число.

Работа команды зависит от типа процедуры:

- для процедур ближнего типа — восстановить из стека содержимое eip/ip;
- для процедур дальнего типа — последовательно восстановить из стека содержимое eip/ip и сегментного регистра cs.
- если команда ret имеет операнд, то увеличить содержимое esp/sp на величину операнда число; при этом учитывается атрибут режима адресации — use16 или use32:
    - если use16, то sp=(sp+число), то есть указатель стека сдвигается на число байт, равное значению число;
    - если use32, то sp=(sp+2*число), то есть указатель стека сдвигается на число слов, равное значению число.

Процедуры могут размещаться в программе:

- в начале программы (до первой исполняемой команды);
- в конце программы (после команды выхода в операционную систему);
- промежуточный вариант — тело процедуры располагается внутри другой процедуры или основной программы (с использованием команды безусловного перехода jmp);
- в другом модуле.

Вызов близкой или дальней процедуры с запоминанием в стеке адреса точки возврата осуществляется *командой call* (формат: call метка). Выполнение команды не влияет на флаги.

При ближней адресации в стек заносится содержимое указателя команд eip/ip и в этот же регистр загружается новое значение адреса, соответствующее метке;

При дальней адресации в стек заносится содержимое указателя команд eip/ip и cs. Затем в эти же регистры загружаются новые значения адресов, соответствующие дальней метке.

3.4.2 Передача аргументов через регистры

Существуют следующие варианты передачи аргументов в процедуру (модуль):

- *Через регистры*. Данные становятся доступными немедленно после передачи управления процедуре. Очень популярный способ при небольшом объеме передаваемых данных.
- *Через общую область памяти*. Необходимо использовать атрибут комбинирования сегментов — common. Сегменты будут перекрываться в памяти и, следовательно, совместно использовать выделенную память.
- *Через стек*. Наиболее часто используемый способ. Суть его в том, что вызывающая процедура самостоятельно заносит в стек передаваемые данные, после чего производит вызов вызываемой процедуры.
- *С помощью директив extrn и public*. Директива extrn предназначена для объявления некоторого имени внешним по отношению к данному модулю. Это имя в другом модуле должно быть объявлено в директиве public. Директива public предназначена для объявления некоторого имени, определенного в этом модуле и видимого в других модулях.

Синтаксис директив:

```
Extrn имя:тип, …, имя:тип
Public имя, …, имя
```

Здесь имя — идентификатор, определенный в другом модуле. В качестве идентификатора могут выступать:

- имена переменных (определенных операторами db, dw и т.д.);
- имена процедур;
- имена констант (определенных операторами = и equ).

Возможные значения типа определяются допустимыми типами объектов для этих директив:

- если имя — это переменная, то тип может принимать значения byte, word, dword, pword, fword, qword и tbyte;
- если имя — это процедура, то тип может принимать значения near или far;
- если имя — это константа, то тип должен быть abs.

Остановимся боле подробно при передаче параметров через стек. Приведем пример структуры вызываемой процедуры при ближней адресации.

```
asmpr proc  near
;пролог
 push bp
 mov  bp,sp
. . .
;доступ к элементам стека
 mov ax, [bp+4] ;доступ к N-элементу
 mov ax, [bp+6] ;доступ к N-1-элементу
 mov ax, [bp+8] ;доступ к N-2-элементу
. . .
; эпилог
 mov  sp,bp ;восстановление sp
 pop  bp ;восстановление bp

 ret   ;возврат в вызывающую программу
asmpr endp  ;конец процедуры
```

На рис. 3.1. показана структура стека при ближней (а) и дальней (б) адресации внутри вызываемой процедуры.

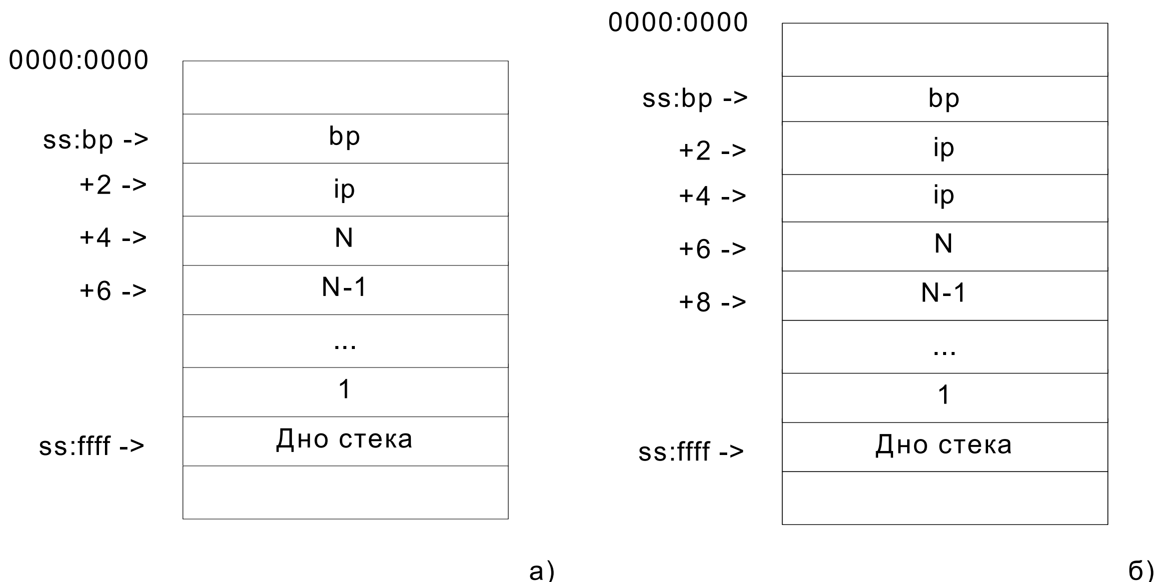


Рис. 3.1–Структура стека в вызываемой процедуре при использовании различных видов адресации: а — ближняя адресация; б — дальняя адресация.

При использовании дальней адресации для доступа к элементам стека требуется поправка на +2.

```
;доступ к элементам стека при дальней адреса-
ции
 mov ax, [bp+6] ;доступ к N-элементу
 mov ax, [bp+8] ;доступ к N-1-элементу
 mov ax, [bp+10] ;доступ к N-2-элементу
```

3.4.3 Возврат результата из процедуры

Существует три варианта возврата результата из процедуры:

- С использованием регистров.
- С использованием общей памяти.
- С использованием стека. Здесь возможны два варианта:
  - Использование для возвращаемых аргументов тех же ячеек в стеке, которые применялись для передачи аргументов в процедуру.
  - Предварительное помещение в стек наряду с передаваемыми аргументами фиктивных аргументов с целью резервирования места для возвращаемого значения.

3.4.4 Макросредства языка ассемблера

Псевдооператоры equ и =

К простейшим макросредствам языка ассемблера можно отнести псевдооператоры equ и "=" (равно). Их мы уже неоднократно использовали при написании программ. Эти псевдооператоры предназначены для присвоения некоторому выражению символического имени или идентификатора. Впоследствии, когда в ходе трансляции этот идентификатор встретится в теле программы, макроассемблер подставит вместо него соответствующее выражение. В качестве выражения могут быть использованы константы, имена меток, символические имена и строки в апострофах. После присвоения этим конструкциям символического имени его можно использовать везде, где требуется размещение данной конструкции.

Синтаксис псевдооператора equ:

имя_идентификатора equ строка или числовое_выражение

Синтаксис псевдооператора “=”:

имя_идентификатора = числовое_выражение

Несмотря на внешнее и функциональное сходство, псевдооператоры equ и “=” отличаются следующим:

- из синтаксического описания видно, что с помощью equ идентификатору можно ставить в соответствие как числовые выражения, так и текстовые строки, а псевдооператор “=” может использоваться только с числовыми выражениями;
- идентификаторы, определенные с помощью “=”, можно переопределять в исходном тексте программы, а определенные с использованием equ – нельзя.

Ассемблер всегда пытается вычислить значение строки, воспринимая ее как выражение. Для того, чтобы строка воспринималась именно как текстовая, необходимо заключить ее в угловые скобки: <строка>. Кстати сказать, угловые скобки являются оператором ассемблера, с помощью которого транслятору сообщается, что заключенная в них строка должна трактоваться как текст, даже если в нее входят служебные слова ассемблера или операторы. Хотя в режиме Ideal это не обязательно, так как строка для equ в нем всегда трактуется как текстовая.

Псевдооператор **equ** удобно использовать для настройки программы на конкретные условия выполнения, замены сложных в обозначении объектов, многократно используемых в программе более простыми именами и т. п.

Псевдооператор “=” удобно использовать для определения простых абсолютных (то есть независящих от места загрузки программы в память) математических выражений. Главное условие то, чтобы транслятор мог вычислить эти выражения во время трансляции.

Макрокоманды

Идейно макрокоманда представляет собой дальнейшее развитие механизма замены текста. С помощью макрокоманд в текст программы можно вставлять последовательности строк (которые логически могут быть данными или командами) и даже более того — привязывать их к контексту места вставки.

Макрокоманда представляет собой строку, содержащую некоторое символическое имя — имя макрокоманды, предназначенное для того, чтобы быть замещенной одной или несколькими другими строками. Имя макрокоманды может сопровождаться параметрами.

Обычно программист сам чувствует момент, когда ему нужно использовать макрокоманды в своей программе. Если такая необходимость возникает и нет готового ранее разработанного варианта нужной макрокоманды, то вначале необходимо задать ее шаблон-описание, который называют макроопределением.

Синтаксис макроопределения следующий:

```
имя_макрокоманды macro список_аргументов
 тело макроопределения
endm
```

Есть три варианта размещения макроопределения:

- В начале исходного текста программы до сегмента кода и данных с тем, чтобы не ухудшать читабельность программы. Этот вариант следует применять в случаях, если определяемые вами макрокоманды актуальны только в пределах одной этой программы.
- В отдельном файле. Этот вариант подходит при работе над несколькими программами одной проблемной области. Чтобы сделать доступными эти макроопределения в конкретной программе, необходимо в начале исходного текста этой программы записать директиву include имя_файла.
- В макробиблиотеке. Если у вас есть универсальные макрокоманды, которые используются практически во всех ваших программах, то их целесообразно записать в так называемую макробиблиотеку. Сделать актуальными макрокоманды из этой библиотеки можно с помощью все той же директивы include.

Если в программе некоторая макрокоманда вызывается несколько раз, то в процессе макрогенерации возникнет ситуация, когда в программе один идентификатор будет определен несколько раз, что, естественно, будет распознано транслятором как ошибка. Для выхода из подобной ситуации применяют ди-

рективу **local**, которая имеет следующий синтаксис: local список_идентификаторов.

3.5 Интерфейс с языками высокого уровня

3.5.1 Формы комбинирования программ на языках высокого уровня с ассемблером

Существуют следующие формы комбинирования программ:

- Использование ассемблерных вставок. Эта форма сильно зависит от синтаксиса языка высокого уровня и конкретного компилятора.
- Использование внешних процедур и функций (на уровне объектных модулей). Это более универсальная форма комбинирования. Она имеет ряд преимуществ:
    - Написание и отладку программ можно производить независимо.
    - Написанные программы можно использовать в других проектах.
    - Облегчается модификация и сопровождение программ в течение жизненного цикла проекта.

Разработка таких подпрограмм на языке ассемблера требует ясного представления о том, каким образом взаимодействуют подпрограммы в разных языках (таблица 3.6). Передача аргументов, как правило, осуществляется через стек.

Таблица 3.6—Сравнение механизмов взаимодействия подпрограмм

Механизмы	Си	Паскаль
Тип подпрограммы	Функция	Процедура или функция
Направление передачи аргументов	Справа налево	Слева направо
Аргументы передаются	По значению (адрес это указатель)	По значению и по адресу
Процедура чистящая стек	Вызывающая	Вызываемая

3.5.2 Соглашения о связях для языка Си

Если в программе на языке Си используется обращение к внешнему модулю, написанному на другом языке, необходимо включить в программу прототип внешней функции (описание точки входа), например:

```
    extern "C" void asmpoc(char ch, unsigned x,
unsigned y)
```

Функция asmpoc должна описываться на ассемблере следующим процедурным блоком:

```
    public _asmproc
     _asmproc proc ...
     ...
     _asmproc endp
```

Блок extrn "C" добавляет к имени точки входа функции префикс (символ подчеркивания). Некоторые компиляторы Си не требуют наличия "C", символ подчеркивания добавляется по умолчанию.

Тип ассемблерной подпрограммы зависит от используемой компилятором модели памяти. Модели памяти и типы указателей на подпрограммы сведены в таблице 3.7.

Таблица 3.7–Соотношение моделей памяти и типов указателей

Модель	**Ключ BCC-компилятора**	**Указатель на функцию**	**Указатель на данные**
Small	-ms	near, CS	near, DS
Compact	-mc	far	near, DS
Medium	-mm	near, CS	far
Large	-ml	far	far

Для малой модели памяти (Small) сегмент кода ассемблерной подпрограммы объединяется с кодовым сегментом главной программы на СИ, который для малой модели всегда имеет имя _TEXT. Это же имя должен иметь сегмент кода ассемблерной внешней подпрограммы. Можно также использовать упрощенные директивы сегментации, где нет необходимости указывать имя сегмента.

Для большой модели памяти (Large) сегмент кода в ассемблерной подпрограмме не объединяется и может иметь любое имя.

3.5.3 Соглашение о связях для языка Паскаль

В программу на языке Паскаль необходимо включить прототип (описание) точки входа во внешнюю подпрограмму или функцию по правилам описания заголовка процедуры или функции:

```
    Procedure asmproc(ch:char;x,y,kol:integer);
external;
```

Директива EXTERNAL указывает, что описание подпрограммы не содержится в главной программе. Для указания файла, содержащего объектный модуль внешней подпрограммы, в программе на Паскале используется директива {$L путь_до_объектного_модуля}.

Компоновщик объединяет этот сегмент с сегментом главной программы.

Занесение аргументов в стек при вызове подпрограммы производится в порядке следования аргументов в списке прототипа. При передаче по значению сами аргументы заносятся в стек, даже массивы.

По умолчанию компилятор Паскаля вставляет в программу команду CALL дальнего вызова. Если необходимо организовать ближний вызов, то перед прототипом внешней подпрограммы следует поставить директиву {$F-} — отмена дальнего вызова. Эта директива действует только на одну подпрограмму и формирует для нее CALL ближнего обращения. Директива не действует на адреса аргументов — в стек заносится полный адрес.

3.6 Примеры работы с дисковой памятью в реальном режиме

Для работы с файлами используются функции прерывания 21h.

Создание файла: функция 3Ch

```
MOV AH,3CH ; Запрос на создание
MOV CX,00 ; обычного файла
LEA DX,PATHNM1 ; ASCIIZ5 строка
INT 21H ; Вызов DOS
JC error ; Переход по ошибке
MOV HANDLE1,AX ; Сохранение файлового номера
```

Запись файла: функция 40h

```
HANDLE1 DW ?
OUTREC DB 256 DUP (' ')
…
MOV AH,40H      ; Запрос записи
MOV BX,HANDLE1  ; Файловый номер
MOV CX,256      ; Длина записи
LEA DX,OUTREC   ; Адрес области вывода
INT 21H         ; Вызов DOS
JC error2       ; Проверка на ошибку
CMP AX,256      ; Все байты записаны?
JNE error3
```

Закрытие файла: функция 3Eh

```
MOV   AH,3EH     ; Запрос на закрытие файла
MOV   BX,HANDLE1 ; Файловый номер
INT   21H        ; Вызов DOS
```

Открытие файла: функция 3Dh

```
MOV   AH,3DH     ; Запрос на открытие

MOV   AL,00      ; Только чтение6
LEA   DX,PATHNM1 ; Строка в формате ASCIIZ
INT   21H        ; Вызов DOS
JC    error4     ; Выход по ошибке
MOV   HANDLE2,AX ; Сохранение номера в DW
```

Чтение файла: функция 3Fh

```
HANDLE2      DW   ?
INPREC       DB   512 DUP (' ')
```

---

[5] `PATHNM1 DB 'C:\TEST.ASM',0`

[6] 0 Открыть файл только для ввода
1 Открыть файл только для вывода
2 Открыть файл для ввода и вывода

```
…
MOV  AH,3FH      ; Запрос на чтение
MOV  BX,HANDLE2  ; Файловый номер
MOV  CX,512      ; Длина записи
LEA  DX,INPREC   ; Адрес области ввода
INT  21H         ; Вызов DOS
JC   error5      ; Проверка на ошибку
CMP  AX,00       ; Прочитано 0 байтов?
JE   endfile
```

Получение размера свободного дискового пространства: функция 36h

```
MOV  AH,36H ; Запрос на
MOV  DL,0   ; текущий дисковод
INT  21H    ; Вызов DOS
```

Удаление файла: функция 41h

```
MOV  AH,41H      ; Запрос на удаление
LEA  DX,PATHNAM  ; ASCIIZ-строка
INT  21H         ; Вызов DOS
```

Управление файловым указателем: функция 42h

Система DOS имеет файловый указатель, который при открытии файла устанавливается в 0 и увеличивается на 1 при последовательных операциях записи или считывания. Для доступа к любым записям внутри файла можно менять файловый указатель с помощью функции 42h, получая в результате прямой доступ к записям файла.

Для установки файлового указателя необходимо поместить в регистр BX файловый номер, а в регистровую пару CX:DX — требуемое смещение в байтах. Для смещений до 65.535 в регистре CX устанавливается 0, а в DX — смещение. В регистре AL должен быть установлен один из кодов, который определяет точку отсчета смещения:

0 — смещение от начала файла.

1 — смещение текущего значения файлового указателя, которое может быть в любом месте, включая начало файла.

2 — смещение от конца файла. Размер файла (и следовательно смещение до конца файла) можно определить, устано-

вив регистровую пару CX:DX в 0 и используя код 2 в регистре AL.

В следующем примере устанавливается файловый указатель на смещение 1024 байта от начала файла:

```
MOV   AH,42H      ; Установка указателя
MOV   AL,00       ; от начала файла
LEA   BX,HANDLE1 ; Установка файлового номера
MOV   CX,00       ;
MOV   DX,1024     ; Смещение 1024 байта
INT   21H         ; Вызов DOS
JC    error
```

Правильно выполненная операция сбрасывает флаг CF и возвращает новый указатель в регистровой паре DX:AX. Неправильная операция устанавливает флаг CF в 1 и возвращает в регистре AX код 01 (ошибка кода отсчета) или 06 (ошибка файлового номера).

Переименование файла: функция 56h

```
MOV   AH,56H         ;Запрос на переименование
LEA   DX,oldstring ; DS:DX
LEA   DI,newstring ; ES:DI
INT   21H            ; Вызов DOS
```

Другие дополнительные функции по работе с файлами вы можете найти в справочнике thelp.

Вопросы для самопроверки

1. Какое основное назначение языка ассемблера?
2. Какую роль играют программы tasm.exe и tlink.exe в процессе создания программ на языке ассемблер?
3. Какие типы предложений языка ассемблера вы знаете?
4. Какие символы являются допустимыми для написания программ на языке ассемблер?
5. Расскажите о синтаксисе директив сегментации.
6. Каково назначение директив ASSUME и MODEL?
7. В чем отличие com-программ от exe-программ?

8. Какое прерывание можно использовать для реализации функций ввода/вывода?
9. Какие ограничения имеют арифметические команды?
10. Назовите логические команды языка ассемблера.
11. В чем разница между логическим и арифметическим сдвигом?
12. На какие два типа делятся циклические сдвиги?
13. Где могут располагаться в программе процедуры?
14. Какие способы передачи параметров в процедуру вы знаете?
15. Какие способы получения параметров из процедуры вы знаете?
16. Опишите структуру стека при ближнем и дальнем способе адресации.
17. В чем отличие операторов EQU и = ?
18. Расскажите о роли макросов в процессе программирования на языке ассемблер.
19. Для чего предназначена директива local?
20. В чем основное отличие процедур от макросов?
21. Сравните механизмы взаимодействия подпрограмм языка ассемблер с языками Си и Паскаль.
22. Расскажите про соглашения о связях для языков Си и Паскаль.

4. Управление внешними устройствами

4.1 Видеоподсистема

4.1.1 История развития видеоадаптеров

Сначала существовал только один тип персональных компьютеров IBM, который комплектовался тоже только однотипными видеодисплеями. Его экран был однотонно зеленым. Текст изображался голубым шрифтом, а из графических средств реализовывались только псевдографика.

Много времени прошло с тех пор, и все технологии компьютерных подсистем шагнули далеко вперед. Видеосистемы совершенствовались, как ни что другое, буквально с каждым днем.

Почти полностью все развитие видеостандартов происходило на основании видеоадаптеров, предлагаемых IBM в своих компьютерах. Прогресс шел постоянно, начиная от жутко зеленого экрана до сегодняшних полноцветных дисплеев с высокой разрешающей способностью. Параллельно увеличивалось вредное влияние видеосистем на глаза человека.

Адаптер монохромного дисплея (MDA). Этот адаптер часто называют просто MDA от Monochrome Display Adapter, хотя его официальное имя — Monochrome Display, или Parallel Printer Adapter.

Слово монохромный отражает самую важную характеристику MDA. Он был создан для работы с одноцветным дисплеем. Первоначально он работал с экранами зеленого цвета, которыми обеспечивались преимущественно все системы IBM того времени.

MDA является символьной системой, не обеспечивающей никакой другой графики, за исключением расширенного множества символов IBM. Это был первый адаптер IBM и до недавнего времени он был лучшим адаптером для обработки текстов, обеспечивающим самое четкое изображение символов, по сравнению с любыми дисплейными системами, выпущенными до PS/2.

Текстовый режим был целью разработки адаптера. Тогда IBM не могла вообразить, что кому-нибудь понадобится рисовать схемы на дисплее.

Цветной графический адаптер (CGA). Первым растровым дисплейным адаптером, разработанным IBM для PC, был цветной графический адаптер — CGA (Color Graphics Adapter). Представленная альтернатива MDA ослепила привыкший к зеленому компьютерный мир. Новый адаптер обеспечивал 16 ярких чистых цветов. Помимо этого, он обладал способностью работать в нескольких графических режимах с различной разрешающей способностью.

CGA — это многорежимный дисплейный адаптер. Он может использоваться и для символьных, и для побитных технологий. Для каждой из них он реализует несколько режимов. Он содержит 16 Кб памяти, прямо доступной центральному микропроцессору.

Hercules. Hercules Graphics Technology Inc., возглавляемая Кевином Дженкинсом, разработала прекрасное устройство, ставшее практически единственным стандартом, не разработанным IBM.

Hercules Graphics Card или HGC реализовал ставшие к тому времени очевидными решения. Этот адаптер в дополнение к посимвольному отображению, реализуемому MDA, обеспечивал растровую графику. Но, чтобы еще больше усилить разработку, фирма добилась поддержки этих возможностей программными разработками фирмы Lotus Development Corporation. Lotus 1-2-3 вскоре стал самым популярным пакетом PC.

Улучшенный графический адаптер (EGA). К 1984 году недостатки CGA стали очевидными. Это выявилось благодаря широкому его распространению. Тяжело читаемый текст и грубая графика портили зрение лучше всякого другого приспособления.

Как ответ на заслуженную критику появился улучшенный графический адаптер – EGA. Улучшение было многосторонним: возросшая разрешающая способность, возможность обеспечивать графический режим монохромных экранов, в том числе любимых IBM зеленых дисплеев; увеличение используемого множества подпрограмм BIOS, зашитых в ПЗУ PC и XT.

Video Graphics Array - VGA. Наименование VGA происходит от VLSI-чипа, используемого в компьютерах типа PS/2. Большинство цепей платы EGA (включая эмуляцию видеочипа 6845 фирмы Motorola) были реализованы одним чипом, который IBM первоначально назвала «Video Graphics Array». Имя этого чипа в скором времени стало использоваться для описания целой системы, возможно потому, что оно было созвучно с аббревиатурой CGA и EGA. Таким образом, это новое имя получало логическое обоснование.

В действительности VGA является дальнейшей разработкой IBM своих предыдущих стандартов. Он обеспечивает все предыдущие режимы и расширяет их до новых границ, формируя изображения с лучшей цветностью и большей разрешающей способностью.

Тем не менее, VGA не самый лучший и не самый передовой адаптер. Даже до представления VGA существовали видеосистемы, обеспечивающие лучшую точность изображения с более богатой цветовой гаммой.

4.1.2 Стандарт VGA

Видеоадаптер VGA был разработан корпорацией IBM и волей судьбы стал стандартом де-факто для всех производителей. Любая видеокарта может работать в режиме VGA. Использование видеорежимов VGA позволяет гарантировать, что программа будет работать на любом современном компьютере. Хотя возможности этого адаптера сильно ограничены, рассмотрение его очень важно, так как практически все операционные системы изначально работают, используя только функции VGA, и лишь только при наличии драйвера они могут использовать все дополнительные возможности видеоадаптеров.

Корпорация IBM сделала все возможное для программной совместимости VGA с младшими моделями видеоадаптеров и практически ничего для аппаратной совместимости. VGA полностью совместим со всеми моделями на уровне базовой системы ввода/вывода (BIOS), а также совместим с адаптером EGA на уровне портов ввода/вывода.

Формирование цвета. Адаптер VGA позволяет отображать на экране одновременно до 256 цветов при низком разрешении и до 16 цветов при высоком разрешении.

Ради совместимости с младшими моделями для формирования 16 цветовых изображений применяется двухуровневое формирование цвета (рис. 4.1). Дешифратор определяет по информации в видеопамяти номер цвета текущего пикселя. По этому номеру выбирается один из индексных регистров. В адаптере EGA в этих регистрах находились двухбитовые значения красной, зеленой и синей компонент цвета (число цветов в палитре – 64). В VGA в индексных регистрах содержится восьмибитовый номер регистра цифроаналогового преобразователя, в котором находятся шестибитовые значения красной, зеленой и синей компонент цвета (число цветов в палитре – 262144).

При формировании двухсот пятидесяти шести цветных изображений индексные регистры не используются. Данные из видеопамяти напрямую попадают в цифро-аналоговый преобразователь.

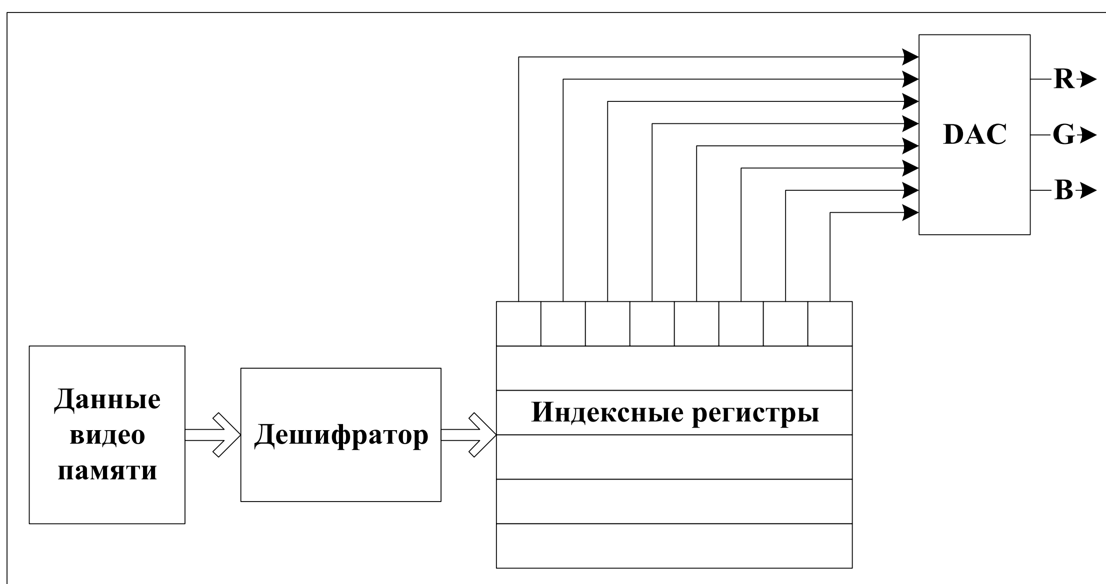


Рис. 4.1–Формирование цвета

Видеопамять и ее отображение в адресное пространство. Адаптер VGA может работать как в текстовых режимах, так и в графических. Для формирования изображения используется внутренняя память адаптера, называемая видеопамятью. Изображение, построенное в этой памяти, моментально отобража-

ется на экране дисплея. Назначение участков памяти и ее используемый размер при формировании изображения зависит от режима, в котором работает адаптер. Например, для формирования изображения в текстовом режиме без учета шрифтов достаточно иметь 4 Кб памяти, в то время как для построения изображения в графическом режиме 640x480 с 16 цветами необходимо иметь около 230 Кб видеопамяти. Практически все адаптеры VGA комплектовались 256 Кб видеопамяти для поддержания всех видеорежимов.

Для того чтобы программное обеспечение могло строить изображение в видеопамяти, последняя отображается на часть адресного пространства процессора, вследствие чего изображение строится обычными процессорными командами, ссылающимися на память. Адреса, в которые отображается видеопамять, также зависят от установленного видеорежима. В компьютерах IBM было выделено два диапазона, в которые отображается видеопамять: с B0000 по BFFFF и с A0000 по AFFFF, т.е. два окошка по 64 кб каждое. Разработчики сделали так, что в каждый конкретный момент может использоваться только одно окошко в зависимости от видеорежима. Для того чтобы адресовать в 256 Кб видеопамяти, она была поделена на 4 блока по 64 Кб каждый, называемые битовыми плоскостями. Функциональное назначение каждой битовой плоскости зависит от установленного видеорежима и будет рассмотрено далее.

Формирование текстового изображения. При формировании текстового изображения видеопамять функционально разделяется на две части: матрицу символов и атрибутов, и таблицы шрифтов для знакогенератора. Коды символов находятся в нулевой битовой плоскости, атрибуты — в первой битовой плоскости, а таблицы шрифтов для знакогенератора находятся во второй битовой плоскости. Третья битовая плоскость не используется.

Изображение на экране строится следующим образом. Из битовой плоскости 0, начиная с адреса сканирования, выбирается 80 символов, а из плоскости 1 выбирается 80 байт атрибутов, после чего формируется строка символов, используя текущий шрифт. Затем выбираются из плоскостей 0 и 1 следующие 80 байт для формирования второй строки и т.д.

Байт атрибута определяет цвет и способ отображения символа на экране. Младшие четыре бита определяют цвет символа (всего возможно 16 комбинаций и именно по этому значению происходит выбор индексного регистра). Следующие три бита определяют цвет фона (возможно 8 комбинаций и по значению происходит выбор индексного регистра). Самый старший бит может использоваться двумя способами: как бит мерцания символа или как дополнительный бит для формирования цвета фона.

В адресном пространстве процессора плоскости 0 и 1 отображаются в диапазоне адресов B8000-B8FFF, причем четный байт содержит код символа и попадает в плоскость 0, а нечетный байт содержит атрибут и попадает в плоскость 1. Плоскость 2 отображается в диапазоне адресов A0000-AFFFF. Доступ к этой плоскости по умолчанию запрещен.

Пример вывода ASCIIZ строки через видеобуфер:

```
…
Mov ax,3            ; установка текстового видеорежима
Int 10h
Mov ax,0b800h       ; установка сегментного регистра дос-
тупа
Mov es,ax           ; в видеобуфере
Mov si,offset HelloTxt; загрузка адреса строки
Mov di,160*y+x*2    ;формирование смещения в видео буфере
Mov ah,4eh          ; цвет отображаемой строки
Call PrintString    ; вызов процедуры печати
…
PrintString  proc
@@1:         Lodsb  ; загрузка байта строки
Or al,al            ; проверка на завершающий ноль
Je @@2
Stosw               ; вывод символа и атрибута на экран
Jmp @@1             ; повтор пока не встретится ноль
@@2:         ret
PrintString  endp

HelloTxt     db 'Hello World',0
```

Формирование графического изображения. В режиме графики каждый пиксель экрана кодируется одним разрядом в каждой битовой плоскости. Таким образом, цвет каждой точки кодируется в памяти четырьмя разрядами и может принимать

значение кода одного из 16 цветов. Именно этот код получается на выходе дешифратора, а далее идет стандартная процедура получения цвета.

Все четыре битовые плоскости отображаются одновременно на то же самое адресное пространство в диапазоне адресов A0000-AFFFF, т.е. работа происходит сразу со всеми плоскостями. При записи одного бита информация может попадать сразу в четыре плоскости, аналогично происходит и считывание. Предоставляется два режима считывания данных и три режима записи, причем при записи могут выполняться логические операции И, ИЛИ, ИСКЛЮЧАЮЩЕЕ ИЛИ, СДВИГ над исходной информацией и информацией, считанной из адресуемой ячейки памяти.

В графическом режиме с 256 цветами память не делится на плоскости, а представляется сплошным байтовым массивом. Каждый байт определяет цвет пикселя на экране. Значение этого байта подается на вход цифроаналогового преобразователя, минуя индексные регистры. Из-за ограничения отображаемой памяти в адресное пространство процессора в 64 Кб максимальное разрешение может быть 320x200 пикселей, что соответствует 64000 байтам. Изображение в данном режиме формируется путем последовательной выборки из видеопамяти 320 байт для формирования строк на экране. Низкое разрешение в этом режиме компенсируется большим числом одновременно отображаемых цветов, поэтому изображение в этом режиме очень часто выглядит лучше, чем в режиме 640x480 с 16 цветами.

Пример программы рисования точки:

```
…
mov ax,13h   ; установка режима 320x200 в 256 цветов
int 10h
mov ax,0a000h; установка сегм. регистра для доступа к
mov es,ax    ; видеобуферу
mov cx,100          ; координата X
mov dx,100          ; координата Y
mov al,15           ; цвет
call PutPixel       ; вызов процедуры отрисовки
…
PutPixel     proc
Movzx edi,dx        ; вычисление смещения в видеобуфере
Lea edi,[edi+edi*4]
```

```
Shl edi,6
Add di,cx             ; добавление компоненты X
Stosb                 ; рисование точки
ret
PutPixel     endp
```

Управление видеоадаптером. Управление видео адаптером может осуществляться двумя способами: через базовую систему ввода вывода или через порты ввода/вывода. Управление через базовую систему ввода вывода гораздо проще, однако, значительно медленнее, что может оказаться существенным для машинной графики. Управление через порты ввода/вывода требует детального изучения функционирования видео адаптера в целом.

Управление видеоадаптером обычно сводится к управлению следующими его компонентами:

- Взаимодействие с видеопамятью.
- Интерпретация содержимого памяти
- Программирование индексных регистров.
- Программирование регистров цифроаналогового преобразователя.

При программировании адаптера через порты ввода/вывода можно программировать каждую компоненту по отдельности.

При использовании базовой системы ввода вывода программирование адаптера обычно осуществляется при установке видеорежима. Однако, помимо этого, имеется ряд функций для управления индексными регистрами и регистрами цифроаналогового преобразователя. Помимо этого имеется еще ряд функций для отображения символов, точек и т.д. К сожалению, эти функции работают слишком медленно и обычно неприемлемы для использования.

Наилучший способ взаимодействия с видеоадаптером — это установить видеорежим через базовую систему ввода/вывода, а изображение строить, используя прямой доступ к памяти и портам ввода/вывода.

4.1.3 Современные видеоадаптеры. VESA

Современные видеоадаптеры уже давно перешагнули порог 320x200 в 256 цветов. Обычным разрешением является 1024x768 в 32Мб цветов и даже более. Естественно, что для этого был увеличен объем видеопамяти и изменены схемы преобразования цветов и отображения памяти в адресное пространство процессора. Единственное, неприятное, что появилось с появлением новых видеоадаптеров — это то, что адаптеры разных производителей и даже разные модели одного производителя не совместимы друг с другом. Это означает, что программа, написанная на одном компьютере, с очень большой вероятностью не будет работать на другом.

Дополнительные объемы видеопамяти и доступ к ней. Для построения изображения высокого разрешения с большим количеством цветов, например, для разрешения 640x480 в 16M цветов, необходимо иметь 1 Мб видеопамяти. Естественно, что в реальном режиме невозможно отобразить одновременно всю видео память в адресном пространстве процессора.

В современных видеоадаптерах объем видеопамяти достигает 4 Мб и даже в некоторых реализациях 8 Мб. Однако, видео память по-прежнему отражается в диапазонах A0000-AFFFF или B0000-BFFFF, т.е. блоками по 64 Кб. Эти диапазоны адресов являются своего рода окнами в видеопамять или банками видеопамяти. Какая именно часть видеопамяти отображается в окне, определяется через порты ввода/вывода видео адаптера, т.е. окно может перемещаться по видеопамяти. Через это окно осуществляется доступ к видеопамяти, как на запись, так и на чтение. Такой способ вполне приемлем в реальном режиме, однако в защищенном режиме он доставляет массу неудобств связанных с перемещением окна, тем более что диапазон адресов процессора вполне позволяет отобразить всю видеопамять целиком. Поэтому большинство видео карт предоставляют режим линейного отображения видео памяти в адресном пространстве процессора. В этом режиме вся видеопамять отображается в старших адресах адресного пространства, и нет необходимости в оконном отображении, за счет чего значительно повышается производительность.

За счет увеличения объема видеопамяти появилась возможность увеличить разрешение и количество цветов. Для увеличения числа цветов используются новые методы формирования цвета. Для совместимости поддерживаются 16 и 256 цветовые режимы, как это было в адаптере VGA, за исключением увеличения разрешения. Однако 256 цветов даже при очень большом разрешении маловато. Поэтому были введены еще три режима: режим с 32К цветами, режим с 64К цветами и режим с 16М цветами.

В режимах с 32К цветами для формирования цвета точки используются два соседних байтах, объединенных в 16 битовое слово. Это слово разделяется на три части по 5 бит, которые содержат красную, зеленую и синюю компоненты цвета. Значение старшего бита не используется. Значения, находящиеся в этих битах, преобразуются в аналоговый сигнал и поступают на монитор безо всяких регистров палитры.

В режимах с 64К цветами для формирования цвета точки также используются два соседних байта. В этих байтах по пять бит отводится под красную и синюю компоненту цвета и шесть бит используется под зеленую компоненту цвета. Палитра не используется.

В режимах с 16М цветами для формирования используются три соседних байта. В каждом байте содержится соответствующая компонента цвета. Значения, находящиеся в этих битах, преобразуются в аналоговый сигнал и поступают на монитор безо всяких регистров палитры.

Дополнительные графические возможности. Дополнительные графические возможности связаны со всевозможными ускорениями, такими как отрисовка линий и прямоугольников без участия процессора, аппаратное масштабирование, кэширование текстур, перенос изображений и т.д. К сожалению, все эти новые возможности различаются на разных видеоадаптерах, и для их использования необходимы специальные драйвера.

Также дополнительные графические возможности предоставляются для трехмерной графики, такие как аппаратное построение трехмерных поверхностей и даже аппаратное. Особенно в этом отношении хороши карты, работающие на шине AGP. Использование этой шины позволяет хранить текстуры в основ-

ной оперативной памяти. При текстурировании видеоадаптер сам обращается к оперативной памяти, не нагружая процессор.

VESA — универсальный интерфейс для программирования видео адаптеров. Цель разработки VESA — это стандартизация программного интерфейса для видео- и аудиоустройств. VBE интерфейс был введен для упрощения разработки приложений, в которых желательно использовать графические, видео- и аудиоустройства без специфических знаний внутренних операций соответствующих устройств.

VBE стандарт определяет множество расширений в сервисыVGA ROM BIOS. Эти функции могут быть доступны под DOS через прерывание 10h, или могут быть вызваны непосредственно высокопроизводительными 32-битовыми приложениями или операционными системами.

Основные принципы работы. Для начала необходимо определить наличие поддержки VESA. Для этого необходимо вызвать функцию получения информации о VESA и проверить возвращаемый результат. Далее можно установить желаемый видеорежим. После чего можно строить изображение через окна видеобуфера. Для перемещения окна по видеопамяти необходимо вызывать соответствующую функцию, указывая базовый адрес окна в единицах гранулярности.

В защищенном режиме необходимо сформировать дополнительные управляющие структуры для вызова функций. Более того, в этом режиме можно использовать плоский буфер для отображения всей видеопамяти в пространстве процессора и тем сам избежать дополнительных накладных расходов, связанных с перемещением окна видеобуфера.

Детальное описание функций и методов использования VESA можно найти в [4].

4.2 Клавиатура

4.2.1 История развития клавиатуры IBM

Главным устройством ввода большинства компьютерных систем является клавиатура. И до тех пор, пока системы распознавания голоса не смогут надежно воспринимать человеческую

речь, главенствующее положение клавиатуры едва ли изменится.

IBM разработала, по крайней мере, 8 разновидностей клавиатур для своих моделей персональных компьютеров.

Клавиатура PC/XT. Неприятности начались с первым типом клавиатуры, предложенным для PC/XT. Несмотря на критику прессы, эта разработка оставалась стандартом IBM до презентации AT. Она имела 83 клавиши. Два ряда функциональных клавиш располагались вертикально, слева от главной алфавитно-цифровой клавиатуры. Клавиши управления курсором были совмещены с отдельными цифровыми клавишами.

Главная критика пришлась на долю расположения периферийных клавиш. Функциональная клавиатура, расположенная под левую руку, не соответствовала ключам по просмотру экранных страниц, как это было тогда принято. Существенные трудности были вызваны отсутствием индикаторов. Кроме того, клавиша ввода была слишком мала.

Клавиатура AT. После нескольких лет критики IBM разработала и представила новую клавиатуру вместе с новой моделью. Это была AT. Ее клавиатура была снабжена специальной клавишей — Sys Req, предназначенной для многопользовательского использования. Клавиша ввода стала больше. Так же обеспечивалась необходимая индикация.

Но в действительности настоящие изменения лежали более глубоко. Не в пример клавиатуре PC, клавиатура AT стала программируемой. Ей было выделено свое собственное множество команд. Эти команды могут поступать с центрального блока. Один этот факт делает новую клавиатуру несовместимой с PC и XT. Хотя используются одни и те же разъемы, клавиатура PC/XT не будет работать при ее подключении к AT и наоборот.

Улучшенная клавиатура IBM. Вместе с производством модернизированных AT IBM начала выпускать новый тип клавиатуры, названной IBM улучшенной клавиатурой. Но все остальные называют ее расширенной клавиатурой. Хотя эта клавиатура электрически полностью совместима со своей предшественницей, расположение клавиш на ней вновь было изменено. Усовершенствование вылилось в увеличение числа клавиш. Их

общее количество достигло 101, что соответствует стандарту США.

В международных моделях добавляется еще одна клавиша. Дополнительных ключей было несколько. Клавиши по управлению курсором были продублированы, и их полное множество было выделено в отдельную группу. Появились две новые функциональные клавиши — F11 и F12. Вся дюжина функциональных клавиш переместилась на самый верхний ряд клавиатуры, слегка отделившись от алфавитно-цифровой зоны. Клавиши Ctrl и Alt были продублированы и размещены по обе стороны основной зоны.

Клавиатура PS/2. Модель PS/2 использует универсальную улучшенную клавиатуру IBM или клавиатуру уменьшенных размеров, специально разработанную для крошечной модели. Единственное отличие улучшенных клавиатур PS/2 и XT/AT — это разъем подключаемого кабеля. PS/2 использует простой миниатюрный DIN разъем, вместо стандартного DIN разъема клавиатуры PC/XT/AT.

Клавиатура совместимых компьютеров. Производители, совместимые PC, шли в ногу с IBM и адаптировали свою клавиатуру к расширяющимся стандартам. Некоторые производители, смутившись критики расположения клавиш на клавиатуре IBM, постарались внести свою собственную изысканность в это устройство. Одно существенное улучшение было реализовано рядом производителей совместимых компьютеров – они установили снизу клавиатуры переключатель совместимости. Два положения этого переключателя позволяют выбирать электрические параметры соединения при подключении к PC/XT или AT.

4.2.2 Взаимодействие с клавиатурой

Общие сведения. Клавиатура — это основное средство связи пользователя с системой. В этом разделе термин «система» относится к контроллеру клавиатуры (8042) на системной плате. Блок-схема клавиатуры приведена на рис. 4.2.

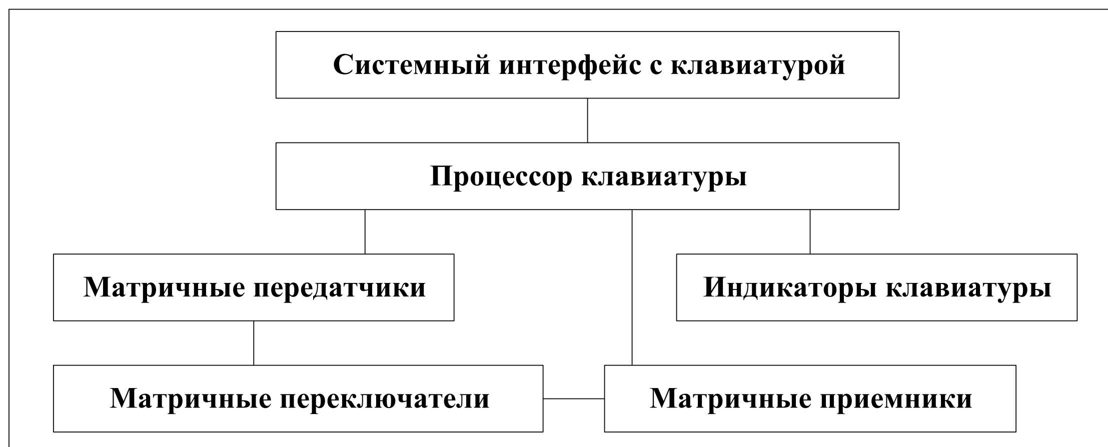


Рис. 4.2–Блок-схема клавиатуры

В клавиатуре имеется микропроцессор, который сканирует ее в поиске нажатых клавиш. Этот микропроцессор контролирует также свою линию связи с системой. По этой линии связи от системы поступают команды управления клавиатурой, а системе — коды сканирования и подтверждения. Данные сканирования клавиатуры вырабатываются при нажатии или опускании клавиши. Например, при нажатии клавиши генерируется код нажатия, а при отжатии — код отжатия. Эти коды нажатия и отжатия вместе называются «кодами сканирования» или «кодами клавиш».

101-клавишная клавиатура имеет буфер, организованный в виде очереди. Этот клавиатурный буфер может хранить 32 кода (нажатия и отжатия) в режиме 1 и 16 кодов сканирования в режимах 2 и 3. Сканкоды сохраняются в буфере до тех пор, пока система не будет готова их принять. Если буфер клавиатуры переполнен, то новый код не генерируется, а в буфер помещается код переполнения, для которого в буфере резервируется место.

За исключением клавиши Pause все клавиши 101-клавишной клавиатуры работают на нажатие/отжатие и являются повторяемыми. Скорость и задержка повторений могут быть изменены. Если удерживаются две и более клавиш, повторяется код только той клавиши, которая была нажата последней. После отжатия этой клавиши автоматическое повторение прекращается, даже если остальные клавиши остаются нажатыми.

Режимы работы клавиатуры. У 101-клавишной клавиатуры есть три режима работы – 1, 2 и 3. Каждый из этих режимов может быть выбран с помощью команд от системы. Эти режимы предоставляют широкий диапазон функциональных возможностей и совместимости для различных прикладных систем.

Контроллер клавиатуры генерирует фиксированный набор кодов нажатия и отжатия для каждой из клавиш на клавиатуре. При нажатии клавиши контроллер клавиатуры посылает в систему код нажатия этой клавиши. В режиме 1 клавиатура генерирует однобайтовый код отжатия, состоящий из кода нажатия плюс 80h. Когда клавиатура работает в режимах 2 и 3, код отжатия представляет собой двухбайтовый код – код F0h и код нажатия клавиши.

Режим 1. Коды сканирования, вырабатываемые расширенной клавиатурой в режиме 1, совместимы с кодами сканирования клавиатуры PC XT. Поскольку никакого преобразования кодов сканирования в этом режиме не происходит, коды сканирования идентичны системным кодам, требуемым для ввода в BIOS.

В наборе кодов сканирования режима 1 каждой клавише присваивается базовый код сканирования, а в некоторых случаях и дополнительные коды для генерирования в системе искусственных состояний переключения регистров. Коды сканирования автоматического повторения каждой клавиши идентичны одиночным кодам сканирования.

Режим 2 — это устанавливаемый по умолчанию режим расширенной клавиатуры, который система выбирает при инициализации после включения питания. В этом режиме код сканирования контроллера клавиатуры 8042 преобразует генерируемые клавиатурой коды нажатия в системные коды, требуемые BIOS. За исключением системных кодов новых клавиш расширенной клавиатуры системные коды соответствуют кодам сканирования, которые генерируются в режиме 1.

Режим 3. Набор кодов сканирования, генерируемых в режиме 3, отличается от генерируемых в режимах 1 и 2. В этом режиме должно быть запрещено преобразование кодов у контроллера клавиатуры 8042, поскольку контроллер не может преобразовывать этот набор кодов сканирования. Прикладные сис-

темы должны взять на себя обработку кодов сканирования, генерируемых клавиатурой, поскольку 8042 и BIOS не могут обрабатывать набор кодов сканирования, генерируемых в режиме 3.

В наборе кодов сканирования каждой клавише присваивается уникальный 8-разрядный код нажатия, который передается при нажатии клавиши. При отжатии каждая клавиша отправляет код отжатия. Код отжатия состоит из двух байтов, первый из которых является префиксом отжатия (F0h), а второй идентичен коду нажатия данной клавиши. В этом наборе кодов каждая клавиша посылает только один код сканирования и состояние любой клавиши не зависит от состояния других.

4.2.3 Программирование клавиатуры

Как после нажатия, так и после отжатия клавиши контроллер клавиатуры генерирует аппаратное прерывание IRQ1. Обработчик прерывания INT 9 читает из порта 60 байт данных и начинает обработку нажатия (отжатия) клавиши. Результатом работы INT 9 могут являться:

- установка флажков состояния клавиатуры (после отжатия клавиш CapsLock, NumLock и пр.);
- включение специальных режимов обработки (после нажатия Alt, Ctrl, Shift и пр.);
- вызов специальных программ — печать экрана, пауза и пр.;
- перевод сканкодов в двухбайтовые коды (ASCII код и последний байт сканкода) и занесение их в буфер клавиатуры.

Управление клавиатурой возможно через порты ввода/вывода 60, 61, 64; прерывание INT 9 и BIOS.

Управление клавиатурой через порты ввода/вывода. Доступ к клавиатуре через порты ввода/вывода является самым низкоуровневым методом доступа. Такой метод обычно используется только прерываниями INT9 и прерываниями BIOS.

С помощью порта 61 можно определить, доступен ли контроллер клавиатуры, и разрешить (запретить) доступ к нему. Причем, если значение, возвращаемое из порта 61, содержит в седьмом разряде нуль, то контроллер клавиатуры не доступен. Пример работы с портом 61 из прерывания INT 9:

```
In al,61h     ; прочитать состояние
Mov ah,al     ; сохранить состояние
Or al,80h     ; разрешить работу клавиатуры
Out 61h,al
Xchg ah,al    ; восстановить статус
Out 61h,al
```

Порт 60 предназначен для чтения данных с клавиатуры и сохранен для совместимости с клавиатурой PC XT. С помощью порта 64 можно читать данные с клавиатуры (коды клавиш и статус), программировать и осуществлять настройку клавиатуры. Рекомендуется использовать порт 60 для чтения сканкодов, а 64 — для определения статуса и программирования клавиатуры (список основных кодов команд приведен в таблице 4.1). Такое разделение обеспечивает максимальную гибкость программам.

```
In al,60h     ; прочитать сканкод
Mov bl,al     ; bl-сканкод
In al,61h     ; прочитать состояние
Mov ah,al     ; сохранить состояние
Or al,80h     ; разрешить работу клавиатуры
Out 61h,al
Xchg ah,al    ; восстановить статус
Out 61h,al
Mov al,20h    ; послать сигнал конца прерывания
Out 20h,al    ; в контроллер прерываний
```

При программировании контроллера клавиатуры необходимо первоначально в порт 64 записать код команды, сделать задержку и затем записать байт данных:

```
Mov al,код команды
Out 64,al              ; запись кода команды в порт
mov ecx,2500h          ; задержка
@@1:   in al,64h; проверка готовности контроллера клавиатуры
test al,2
loopnz @@1
mov al,байт данных
out 64h,al             ; запись байта данных в порт
```

Таблица 4.1–Коды команд клавиатуры

Код	**Функция**	**Назначение**
Edh	Установка индикаторов	Установка и сброс светодиодных индикаторов на клавиатуре
Eeh	Эхо	Средство диагностики
FDh	Альтернативный режим	Происходит сброс клавиатуры и установка режима
EFh-F1h	Резерв	Холостые команды
F2h	Индикатор расширенной клавиатуры	Это команда, передаваемая при включении питания. Клавиатура в ответ передает идентификатор клавиатуры
F3h	Частота повторения клавиш	Установка задержки и автоповтора нажатой клавиши
F4h	Разрешение клавиатуры	Разрешает сканирование клавиатуре
F5h	Запрещение клавиатуры	Останавливает сканирование и сбрасывает настроенные параметры
F6h	Условие по умолчанию	Сбрасывает настроенные параметры
F7h-Fah	Установка всех клавиш	Устанавливает параметры сразу для всех клавиш
FBh-FDh	Установка отдельных клавиш	Устанавливает параметры отдельных клавиш
Feh	Посылка повторно	Клавиатура переспосылает последний посланный байт
FFh	Сброс	После этой команды клавиатура запрещается либо до принятия системой кода ответа либо до выдачи на клавиатуру другой команды.

Управление клавиатурой через базовую систему ввода/вывода. Обработка клавиатуры с помощью INT 16 является основным методом чтения данных с клавиатуры и определения ее статуса в прикладных программах.

INT 16 представляет пользователю следующие подфункции:

- Чтение данных с клавиатуры с ожиданием нажатия.
- Проверка клавиатуры.
- Получение статуса клавиатуры.
- Установка скорости генерации символов повтора и паузы.
- Запись символа в буфер клавиатуры.

Детально функции приводятся в приложении.

4.3 Дисковая подсистема

4.3.1 Цилиндр, дорожка, головка, сектор

Дисковая подсистема в этом разделе рассматривается с двух позиций: общих для всех типов дисковых устройств архитектурных компонентов, определяющих основные принципы доступа к данным, и специфических особенностей основных устройств – гибких и жестких дисков.

Основой всех дисковых устройств является магнитный носитель, имеющий форму диска. Поверхность диска разделена на концентрические окружности (рис. 4.3), отсчет которых на жестких дисках начинается от центра, а на гибких — от края. Эти окружности называются дорожками.

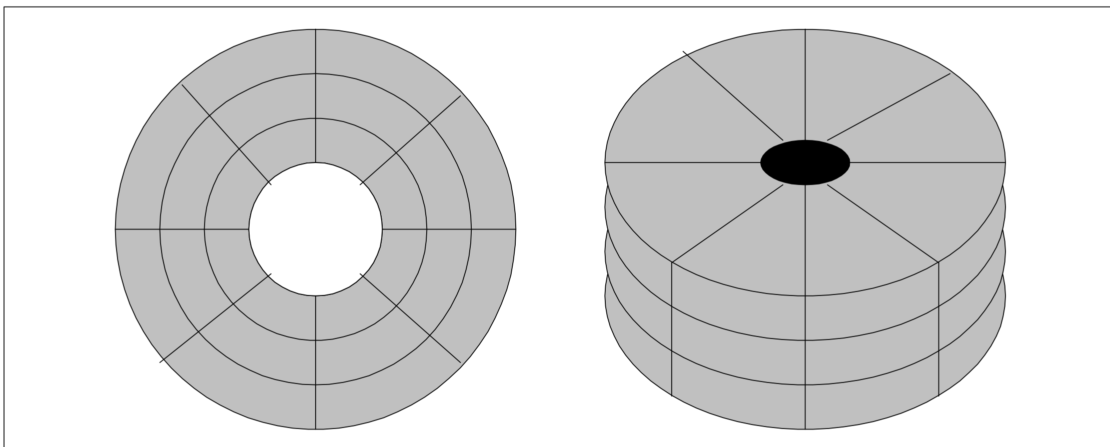


Рис. 4.3–Структура диска

Группа дорожек на разных поверхностях, имеющих одинаковый номер, называются цилиндром и определяются одним параметром – номером цилиндра.

Для каждой поверхности имеется головка считывания/записи. Таким образом, номер головки определяет номер поверхности.

Сектор представляет собой зону дорожки, в которой собственно и хранятся данные. Количество секторов на дорожке определяется многими параметрами, но в основном длиной полей данных и служебных полей.

Цилиндр, головка и сектор являются основными архитектурными и системными компонентами. Именно эти параметры указываются при операциях чтения/записи.

4.3.2 Жесткие диски. История развития

Наименование устройства подчеркивает его отличие от гибкого диска. Гибкий диск имеет гибкую основу, в то время как в жестком — магнитное покрытие наносится на жесткую подложку.

Появление винчестера. Для наименования жестких дисков используется несколько имен. Часто жесткие диски называют винчестерами из-за используемой ими технологии. Основополагающим принципом этой технологии является плавающая головка чтения-записи. По этой технологии головка плывет по воздуху наподобие крыла самолета. Вращающийся жесткий диск создает воздушный поток, обладающий достаточной подъемной силой, чтобы удержать головку над поверхностью диска на расстоянии нескольких микрон.

Имя технологии дала первая разработка устройства с ее использованием. Устройство было создано IBM и имело кодовое название 3030. Согласно легенде, винтовка этого типа завоевала Запад. Такие же намерения были и у разработчиков устройства. Наименование получило широкое распространение не только при описании типа устройства, но и для описания технологии.

Еще одна история повествует о том, что данное устройство получило название «винчестер», потому что технология плавающей головки была разработана лабораторией IBM в Англии, городе Винчестере. Что касается IBM, она официально придерживается первой версии.

Технология Уитни. Однако теперь не все жесткие диски изготавливаются по технологии винчестера. Новая технология, названная Уитни, позволяет получить меньшие, более быстрые и более надежные конструкции, которыми теперь заменяют головки винчестера.

Технология Бернулли. Iomega Corporation модифицировала технологию винчестера. Вместо использования воздушного потока вращающегося диска для поддержания головок над его поверхностью в этих дисководах используется гибкий диск, который сгибается вокруг головки под силой воздушного потока. Достоинством такой системы является то, что вращающийся диск имеет существенно меньший вес и более устойчив к разрушению головок. Недостатком – то, что он постоянно гнется, что приводит к его износу.

4.3.3 Работа с дисками через BIOS

Адресация информации на дисковых накопителях. Работа с дисками через базовую систему ввода/вывода осуществляется через 13 прерывание. Через это прерывание вызываются функции сервиса дисковой системы, такие как чтение/запись секторов, проверка секторов, получение параметров устройства, форматирование и т.д.

Для адресации секторов на диске используются три восьмибитовых регистра: DH, CH и CL. В регистре DH находится номер поверхности или головки, в младших 6 битах CL находится номер сектора, а старшие биты присоединяются к регистру CH, и в них находится номер цилиндра. Максимальное число дорожек может быть 1024, максимальное число головок — 64 и максимальное число секторов — 63. Таким образом, максимальный адресуемый через BIOS объем — 8 Гб. Для использования дисков большей емкости необходимо работать с ними через порты ввода/вывода или использовать новую версию базовой системы ввода/вывода.

Режимы перекодирования параметров. Несмотря на то, что для указания номера поверхности отводится 8 бит, это вовсе не означает, что физически имеется 128 дисков на одной оси, так как такое устройство очень сложно создать. Обычно на од-

ной оси располагается до 8 дисков, соответственно номер поверхности не превышает 16. В то же время число дорожек на поверхности диска достигает нескольких десятков тысяч, в то время как через BIOS можно адресовать только 1024. Исходя из ограничений, накладываемых базовой системой ввода/вывода, применяется несколько методов перекодирования параметров дисковых запросов (несколько методов отображения логических секторов в физические). Обычно используется два режима: CHS и LBA.

Режим CHS — это режим, который устанавливается по умолчанию. В режиме CHS для адресации используются три поля: адрес сектора, номер головки и номер цилиндра. Сектора нумеруются от 1 до максимального значения текущего CHS метода преобразования, которое не может превышать 255. Головки нумеруются от 0 до максимального значения, которое не может превышать 15. Цилиндры нумеруются от 0 до максимального значения, которое не может превышать 65535. Во время команды инициализации CHS режима происходит запрос максимальных значений номера сектора и номера головок. При этом максимальное значение дорожки вычисляется автоматически.

Режим LBA предоставляет возможность адресовать максимальное дисковое пространство через базовую систему ввода/вывода путем дополнительного преобразования параметров, размеры которых начинают соответствовать размерам, принятым в BIOS. Следующее выражение всегда является истиной:

LBA=((Цилиндр*число_головок_на_цилиндре+Головка)*число_секторов_на_дорожке)+Сектор-1,

где число_головок_на_цилиндре и число_секторов_на_дорожке являются значениями текущего режима преобразования.

4.3.4 Работа с дисками через порты ввода/вывода

Работа с дисковыми устройствами через порты ввода/вывода значительно сложнее, чем через BIOS.

Особенно все усложняет то, что разные дисковые устройства программируются по-разному. Например, если BIOS унифицирует взаимодействие с дисками, то схемы взаимодействия

через порты ввода/вывода с контроллером гибких дисков, с контроллером IDE дисков и с контроллером SCSI дисков абсолютно различны.

Тем не менее, несмотря на этот значительный недостаток, практически все операционные системы не используют BIOS для взаимодействия с дисковой подсистемой. Это объясняется тем, что BIOS разработан для работы в реальном режиме, и более того, функции сервиса дисковой подсистемы. Невозможно одновременно работать сразу с несколькими дисковыми устройствами. И даже если имеется только одно дисковое устройство, функции, предоставляемые базовой системой ввода вывода, не эффективны в многозадачной операционной системе.

Взаимодействие с контролерами дисковых накопителей сводится к подаче команд с соответствующими параметрами (при необходимости производится программирование контроллера прямого доступа к памяти) и получению ответов. Данные обычно передаются независимо от процессора. На сегодняшний день стандартизованы интерфейсы взаимодействия с гибкими дисками и IDE жесткими дисками. Команды и детальное их описание может быть найдено в соответствующей технической документации.

4.3.5 Примеры работы с дисковыми накопителями через BIOS

Определение числа жестких дисков в системе. Программа заносит в переменную HardDisksNumber число жестких дисков, установленных на компьютере.

```
mov HardDisksNumber,80h    ; номер первого жесткого диска
@@0:   mov ah,10h          ; проверка готовности диска
mov dl,HardDisksNumber     ; номер диска
int 13h                    ; вызов прерывания
or ah,ah                   ; проверка наличия ошибки
jne @@01                   ; устройства нет
inc HardDisksNumber ; переход к следующему устройству
cmp HardDisksNumber,88h; ограничение цикла
jb @@0              ; проверка следующего диска
@@01:  sub HardDisksNumber,80h; сброс старшего бита
```

Чтение/запись секторов. Программа считывает содержимое указанного сектора в буфер.

```
Mov ah,2      ; команда чтения секторов
```

```
Mov dl,номер диска
Mov dh,номер головки
Mov ch,номер цилиндра and 0ffh
Mov cl,номер сктора + (номер цилиндра and 300h) shr 2
Mov al,число секторов
Mov bx,offset Buffer
Int 13h
Jc Error
```

4.4 Подсистема таймера

Архитектура PC AT использует подсистему трехканального 16-разрядного таймера 8254 в качестве системного таймера. Программируемый таймер предназначен для получения программно управляемых временных задержек и генерации времязадающих функций. Таймер позволяет повысить эффективность программирования процессов управления и синхронизации внешних устройств, особенно в реальном масштабе времени.

Таймер содержит три независимых канала, каждый из которых может быть запрограммирован на работу в одном из шести режимов для двоичного или двоично-десятичного счета. Выход канала 0 связан с аппаратным прерыванием IRQ 0 и обеспечивает прерывание системного таймера для часов реального времени, используя режим 3. Выход канала 1 генерирует сигнал запроса регенерации динамической RAM, используется режим 2. Выход канала 2 генерирует тональный сигнал для динамика, используется режим 3.

После включения питания состояние таймера неопределенное. Режим работы каждого счетчика определяется при его программировании. Каждый счетчик должен быть запрограммирован, прежде чем он будет использоваться.

При программировании счетчика сначала записывается управляющее слово, а затем константа счета. Формат управляющего слова показан на рисунке 4.4. Запись управляющего слова происходит при записи в порт 43, запись констант счета в каналы 1,2,3 – при записи в порты 40,41,42.

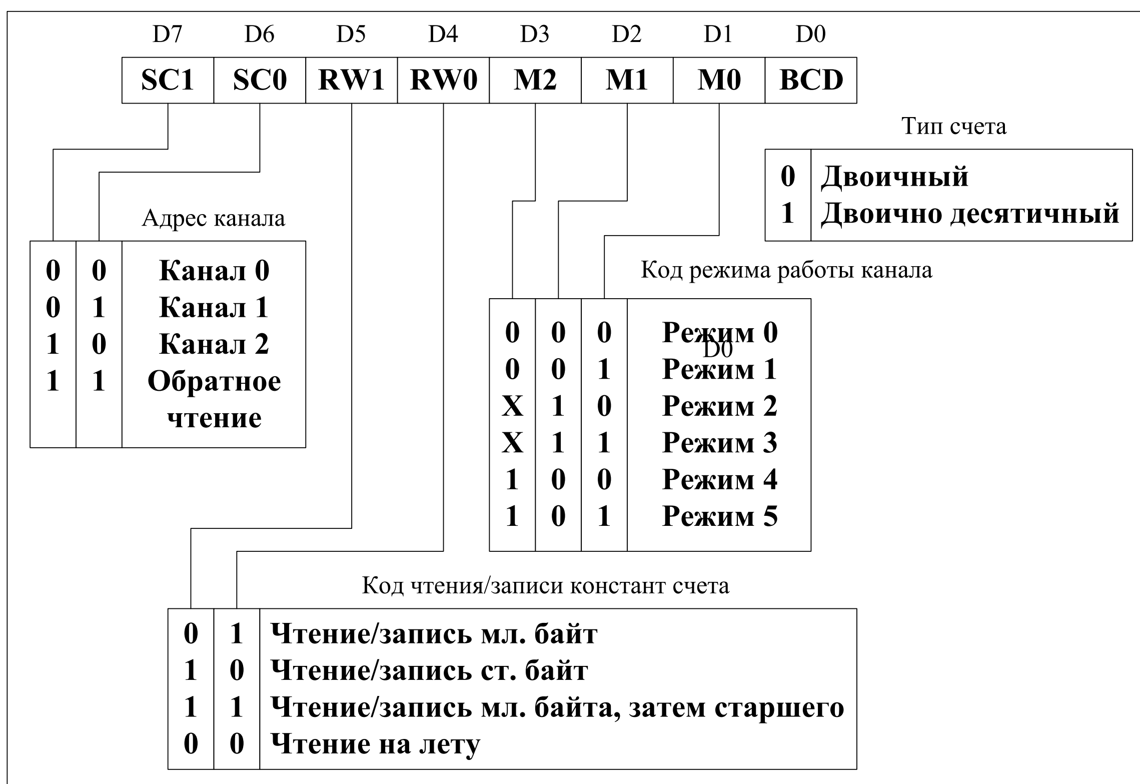


Рис. 4.4–Формат управляющего слова таймера

Процедура программирования предполагает использование следующих двух правил:

- В каждый счетчик управляющее слово должно быть записано перед загрузкой счета.
- Счетчик должен загружаться тем количеством байтов значений счета, которое указано в управляющем слове.

Например, чтобы установить интервал регенерации, необходимо выполнить следующую последовательность команд:

```
Mov al,54h
Out 43h,al
Mov al,0fh
Out 41h,al
```

Часто желательно читать содержимое счетчика без нарушения процесса счета. Имеется три метода чтения счетчиков: простое чтение, чтение на лету и обратное чтение.

Первый метод реализуется простой операцией чтения. При чтении счетчика его работа должна быть приостановлена. В противном случае счет может быть изменен в процессе чтения и его результат будет не определен.

Второй метод — чтение «на лету». Этот способ позволяет прочитать текущее значение счетчика в любой момент времени без останова счета. При выполнении команды «чтение на лету» текущее значение счета защелкивается в выходной защелке и находится там до начала выполнения первой команды считывания этого счетчика.

Третий метод предполагает использование команды «обратное чтение». По команде могут выполняться две операции: защелкивание текущего значения счетчиков и/или защелкивание текущего значения каналов. Эти операции выполняются либо независимо, либо совмещаются. Эта команда эквивалентна нескольким командам «чтение на лету».

Более детальное описание программирования подсистемы таймера приводится в [2].

4.5 Контроллер прерываний

4.5.1 Подсистема контроллера прерываний (8259A)

Все запросы на аппаратные прерывания из системной шины направляются через контроллеры прерываний 8259A. Эти контроллеры генерируют запросы прерываний на вход INTR микропроцессора, которые могут маскироваться в самом микропроцессоре программным путем.

Контроллеры прерываний могут принимать сигналы прерываний от нескольких устройств, назначать им приоритеты и прерывать работу процессора.

В архитектуре PC AT подсистема аппаратных прерываний состоит из двух контроллеров 8259A (главного — MASTER и подчиненного — SLAVE). Они объединены таким образом, что могут обслужить 15 запросов на прерывания. Выход INTR SLAVE соединяется с входом запроса на прерывание (IRQ) MASTER. Таким образом, SLAVE будет прерывать MASTER. Выход INTR MASTER соединен с входом INTR микропроцессора, что обеспечивает общий запрос на прерывание INTR. Структурная схема контроллеров приведена на рис. 4.5.

После включения питания MASTER и SLAVE инициализируются BIOS, в результате чего каждому запросу на прерывание устанавливается приоритет и присваивается код вектора.

Устройство ввода-вывода может запросить обслуживание путем подачи активного сигнала на один из входов запроса контроллера 8259A. Если контроллер удовлетворит запрос, его выход INTR активизируется и соответствующий сигнал поступает на вход INTR микропроцессора. После реакции процессора на этот сигнал контроллер помещает на шину данных байт вектора прерывания. Этот байт вектора прерывания не используется в качестве прямого адреса, а относится к одному из 256 типов прерываний. Старшие 5 битов байта вектора формируются при инициализации контроллера, а младшие три бита зависят от номера прерывания.

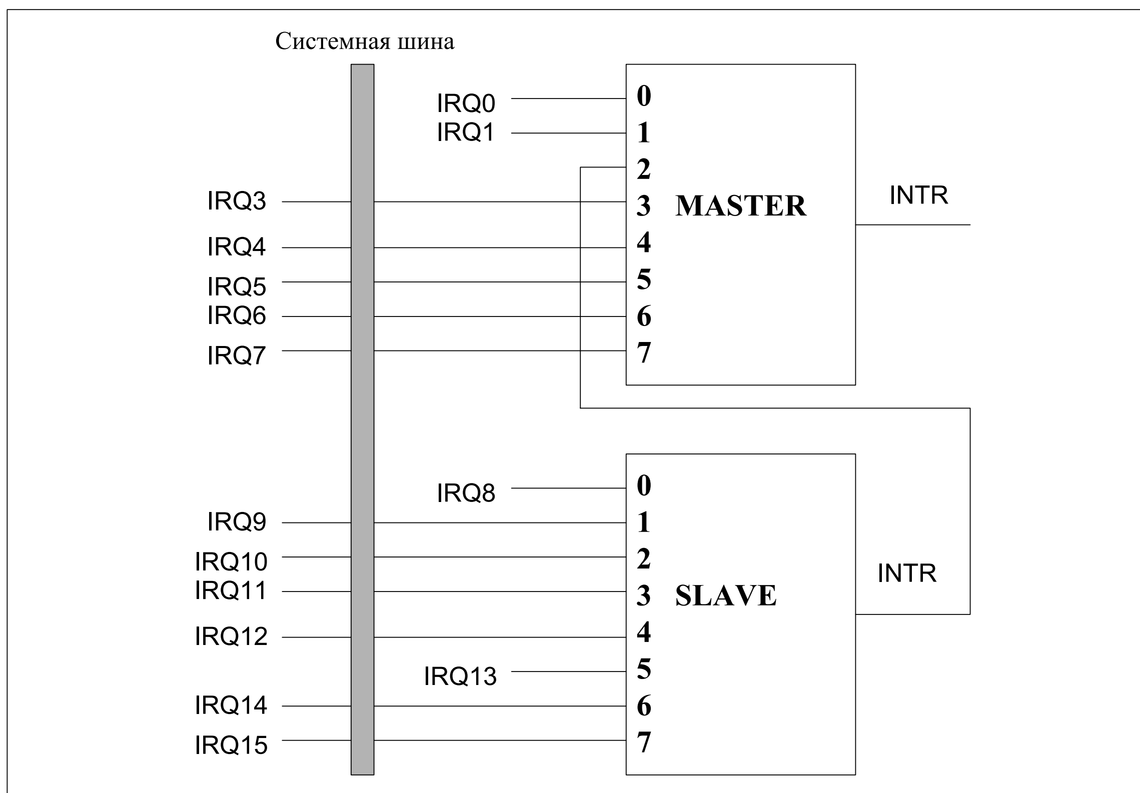


Рис. 4.5–Структурная схема подсистемы контроллеров 8259A

Для отслеживания поступаемых и обрабатываемых прерываний используются три регистра контроллера:

- IRR — 8-разрядный регистр запросов прерываний. Запоминает все запросы, поступающие на входы IRQ. Когда преры-

вание подтверждается, соответствующий разряд IRR сбрасывается.

- ISR — 8-разрядный регистр обслуживаемых запросов. Устанавливается после подтверждения запроса процессором. Сбрасывается обычно по окончании подпрограммы обслуживания прерывания. Установленный разряд ISR блокирует обслуживание всех запросов, имеющих тот же или более низкий приоритет.
- IMR — 8-разрядный регистр маски. Содержит маски, позволяющие заблокировать прохождение поступившего запроса в процессор. 0 разрешает соответствующий запрос, 1 — запрещает.

4.5.2 Программирование контроллера прерываний

Инициализация и установка режимов работы контроллера выполняется путем его программирования как устройства ввода-вывода с помощью команд ввода/вывода IN и OUT. Установка контроллера в исходное состояние и определение алгоритма обслуживания прерываний, а также его изменение в процессе работы осуществляется с помощью команд контроллера двух типов: команд инициализации (ICW) и рабочих команд (OCW).

Контроллер может выполнять следующий набор операций:

- Маскирование — индивидуальное маскирование запросов (OCW1), специальное маскирование обслуживаемых запросов (OCW3).
- Установку статуса уровней приоритета по установке исходного состояния (ICW1), по обслуженному запросу (OCW2), специальное (OCW2).
- Окончание прерываний — обычное (OCW2), специальное (OCW2), автоматическое (ICW4).
- Чтение регистра запросов (OCW3), регистра обслуженных запросов (OCW3), регистра маски (OCW1), результатов опроса (OCW3).

Команды контроллера представляют собой байты констант с соответствующей выбранному режиму конфигурацией, которые записываются в регистр AL процессора.

Для начала нормальной работы каждый контроллер в системе должен быть инициализирован с помощью командных слов инициализации (ICW), содержащих от двух до четырех байтов. Инициализация контроллера (рис. 4.6) производится только в указанном порядке. ICW1 заносится в порт 20H/A0h, а все последующие слова инициализации заносятся по порядку в порт 21h/A1h.

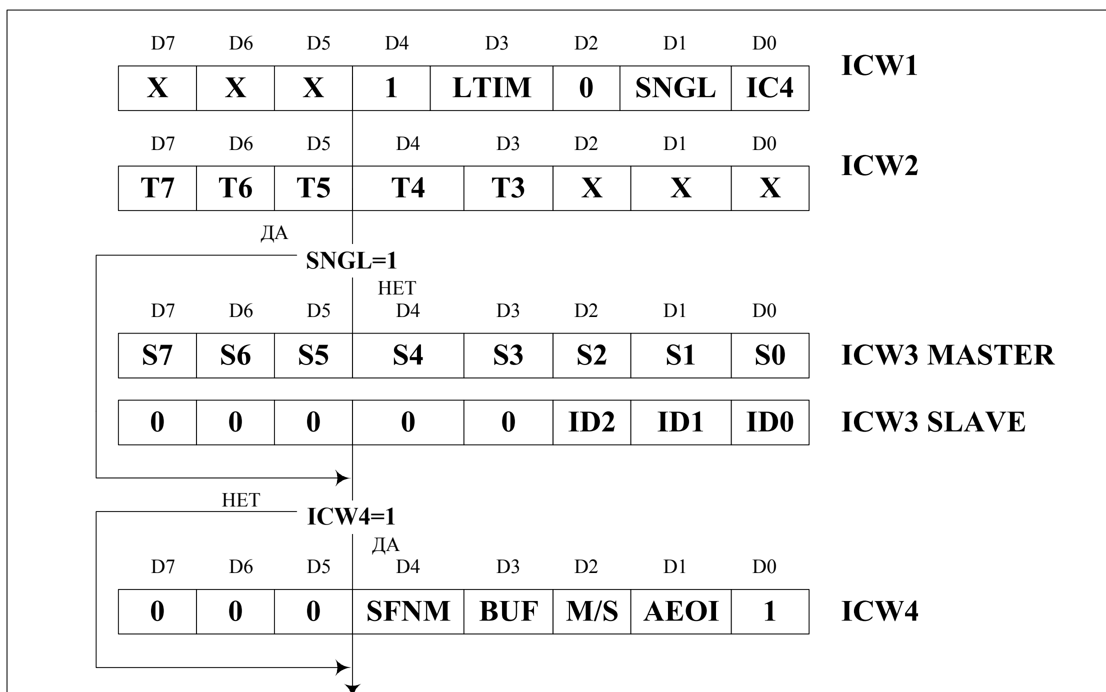


Рис. 4.6–Ход инициализации контроллера прерываний

Типовая программа инициализации контроллеров имеет следующий вид:

```
; MASTER
mov al,11h; ICW1, режим зап. по фронту, каскадный режим
out 20h,al; наличие в цепочке команды инициализации ICW4
mov al,8h; ICW2 Базовый вектор прерываний
out 21h,al
mov al,4; ICW3, указывает, что SLAVE поключен к IRQ2
out 21h,al
mov al,1; ICW4, установлен режим работы системы
out 21h,al
; SLAVE
mov al,11h; ICW1, режим зап. по фронту, каскадный режим
out 0A0h,al; наличие в цепочке команды инициал. ICW4
mov al,70h; ICW2 Базовый вектор прерываний
out 0A1h,al
```

```
mov al,2; ICW3, задает код SLAVE в системе
out 0A1h,al
mov al,1; ICW4, установлен режим работы системы
out 0A1h,al
```

После инициализации дальнейшее управление работой контроллера может быть осуществлено с помощью команд OCW. Для программирования различных режимов предусмотрены три команды OCW. В отличие от ICW для OCW не требуется никакой определенной последовательности.

OCW1 используется только для операций, связанных с маскированием. Это слово обеспечивает прямой канал связи с регистром маски прерываний IMR. Процессор может осуществлять чтение или запись IMR через OCW1.

OCW2 используется для окончания прерывания и изменения приоритетов. Далее приводится пример завершения прерывания.

```
Mov al,20h   ;команда обычного окончания прерывания
Out 20h,al
```

OCW3 используется для выдачи различных режимов и команд в контроллер. Слово OCW3 имеет дело с двумя основными классами операций, связанными с состоянием прерываний и маскированием прерываний.

4.6 Подсистема часов реального времени RTC/CMOS

4.6.1 Память подсистемы RTC

В архитектуре PC AT используется контроллер типа MC146818 в качестве часов реального времени RTC и памяти конфигурации системы.

Контроллер содержит 64 байта памяти: первые 14 байтов памяти используются непосредственно для внутренней схемы RTC, остальные 50 байтов используются для размещения информации о конфигурации системы. Карта адресов памяти приведена в табл. 4.2.

Таблица 4.2–Карта адресов памяти

Адрес регистра	**Функциональное назначение**
00h	Секунды
01h	Секунды будильника
02h	Минуты
03h	Минуты будильника
04h	Часы
05h	Часы будильника
06h	День недели
07h	Число месяца
08h	Месяц
09h	Год
0Ah	Регистр состояния A
0Bh	Регистр состояния B
0Ch	Регистр состояния C
0Dh	Регистр состояния D
0Eh	Байт диагностирования
0Fh	Байт кода сброса
10h	Тип НГМД
11h	Резерв
12h	Тип НМД
13h	Резерв
14h	Установленное оборудование
15h, 16h	Объем базовой памяти
17h, 18h	Объем расширенной памяти
19h	Тип НМД 1
20h	Тип НМД 2
1Bh-2Ch	Резерв
2Dh	Дополнительный флажок
2Eh-2Fh	Контрольная сумма содержимого адресов 10h-2Dh
32h	Текущий век в коде BCD
33h	Системная информация
34h-3Eh	Резерв
3Fh	Не используется

Все 64 регистра памяти RTC доступны для чтения и записи, кроме следующих, которые доступны только на чтение:

- Регистры состояния C и D.
- Разряд 7 регистра состояния A.
- Старший разряд байта секунд.
- Неиспользуемый байт 3Fh.

4.6.2 Функционирование подсистемы часов реального времени

Подсистема часов реального времени продолжает счет времени даже при отключенном питании компьютера. Помимо подсчета времени и даты она позволяет устанавливать будильник и генерировать периодические прерывания.

Цикл корректировки. Цикл корректировки проводится каждую секунду. Основная функция цикла корректировки — это изменения значения во всех байтах часов и даты, если это необходимо. Так же в это время происходит сравнение байтов данных часов с байтами будильника, и при совпадении включается будильник.

Обычно цикл корректировки занимает 248 мкс. Во время этого цикла программа процессора не имеет доступа к байтам времени, календаря и будильника. Если процессор будет читать эти байты до завершения корректировки, выход будет неопределенным. В это время устанавливается разряд состояния цикла корректировки UIP.

Далее предлагается два способа получения данных часов и даты.

В первом способе используется прерывание окончания корректировки. Если оно разрешено, прерывание вырабатывается после каждого цикла корректировки. При этом имеется свыше 999 мс для чтения действительных данных времени и даты. За это время данные могут быть помещены в память, где к ним возможен непрерывный доступ.

Во втором способе разряд текущей корректировки UIP в регистре A используется для того, чтобы определить, осуществляется ли в данный момент цикл корректировки. После того как установится UIP, через 244 мкс начнется цикл корректировки. Следовательно, если прочитан низкий уровень UIP, у пользователя есть, по меньшей мере, 244 мкс до того, как будут изменены

данные часов/календаря. Если разряд UIP установлен, данные часов/календаря могут оказаться неверными.

Прерывания. RTC обеспечивает три отдельных автоматических источника прерываний процессора. Прерывание будильника может быть запрограммировано на периодичность от одного раза в секунду до одного раза в день. Периодические прерывания могут быть выбраны с периодом от 0,5 с до 30,517 мкс. Для извещения программы о завершении цикла корректировки может быть использовано прерывание конца корректировки.

Нужный режим прерывания выбирает программа процессора. Три разряда в регистре B разрешают три прерывания. Каждый из трех источников прерываний имеет свой флажковый разряд в регистре C, который устанавливается независимо от состояния соответствующего разряда в регистре B.

В случае программного сканирования программа не разрешает прерывание. Флажковый разряд прерывания становится разрядом состояния, который может запрашиваться программой. Когда программа обнаружит, что этот флажок установлен, это указывает ей возникновение «прерывающего события» во времени последнего чтения этого разряда.

Второй метод использования флажковых разрядов реализуется при полностью разрешенных прерываниях. Программа может определить, что от RTC возбуждено прерывание при чтении регистра C. Единица в разряде 7 (IRQF) указывает, что было возбуждено одно или более прерываний. В процессе чтения регистра C сбрасываются все активные флажковые разряды и IRQF. Если программа обнаруживает установленный IRQF, она должна просмотреть каждый отдельный флажок в этом же байте. Может быть установлено и несколько флажковых разрядов прерываний.

4.6.3 Программирование доступа к памяти RTC

Для чтения и записи во внутреннюю память RTC используются команды процессора IN и OUT. Порт 70 используется совместно регистром маски NMI и регистром адресов памяти конфигурации. Чтобы оставить разрешенным прерывание NMI,

нужно чтобы разряд 7 адресов RTC был установлен в единицу при записи в порт 70.

Чтобы записать значение в память RTC, необходимо:

- Использовать OUT 70h,AL для указания адреса памяти конфигурации, которая будет изменена.
- Использовать OUT 71h,AL для записи данных в память RTC по адресу указанному в предыдущей команде.

Команды вывода слов (OUT DX,AX) для одновременной загрузки ячейки памяти и данных не допускаются.

4.7 Последовательный и параллельный порты

4.7.1 Последовательный порт

Единственный двухпроводный доступ к внешнему миру, обеспечиваемый персональными компьютерами, официально признанный IBM, до появления PS/2 был асинхронный информационный порт связи, называемый также асинхронным, последовательным или «обычным» портом. Так как этот порт работает по стандарту EIA (Electronics Industry Association) — RS232C, его также называют RS-232 портом.

Синхронная и асинхронная связь. В идеальном мире нужную простейшую цепь можно составить из двух проводов: один — для заземления, другой — для сигналов. Но наш мир, увы, не так совершенен, а компьютерный мир и подавно. Например, компьютер не имеет гарантии, что всегда получит первый бит последовательности, когда начнет прослушивать линию последовательных сигналов.

Для преодоления подобных проблем используются два способа последовательной передачи информации. Один из них предполагает использование посылающей и принимающей сторонами какого-либо дополнительного сигнала, позволяющего обеим системам всегда шагать в ногу. Эта технология синхронной связи используется, главным образом, в больших ЭВМ.

В большинстве асинхронных систем вся передаваемая информация делится на маленькие части, приблизительно рав-

ные одному байту. Каждый такой кусочек называется словом и может содержать от 5 до 8 информационных бит.

Так как это последовательная передача данных, биты слова передаются по одному в канал связи. К этим информационным битам добавляется специфический импульс двойной длины, названный стартовым битом. Он указывает на начало информационного слова. Еще один дополнительный бит — стоп-бит — указывает на конец слова. Между последним битом слова и первым стоп-битом часто вставляется бит четности для проверки целостности информации.

Программирование последовательного порта. На одном компьютере может быть установлено несколько последовательных портов. Для них назначаются символьные имена COM1, COM2 и могут быть установлены дополнительные порты. Для программирования последовательного порта отводится 10 регистров, адресуемых через порты ввода/вывода.

Для инициализации последовательного порта применяется следующая последовательность действий:

- Запись в регистр 3FBh управляющего байта с единицей в седьмом разряде.
- Запись константы деления в регистры 3F8h и 3F9h.
- Запись управляющего байта с нулем в седьмом разряде.
- Запись управляющего байта в регистр 3F9h.
- Запись управляющего байта в регистр 3FCh.

Буферный регистр передачи обеспечивает промежуточное хранение передаваемых знаков данных до их записи в регистр сдвига передачи.

Буферный регистр приема обеспечивает промежуточное хранение принимаемых знаков данных до их чтения в системную шину.

Делители частоты выбора скорости 1 и 2 служат для хранения 16-разрядной константы, которая изменяет коэффициент деления тактовой частоты, обеспечивая тем самым требуемую скорость обмена данными.

Регистр идентификатора прерываний хранит код условия прерывания и признак запроса прерывания.

Регистр разрешения прерываний управляет формированием сигнала запроса прерывания в соответствии с состоянием регистра.

Регистр управления линией обеспечивает выбор формата символа данных: количество информационных разрядов, количество стоповых разрядов, метод контроля и выбор адресуемых регистров.

Регистр управления модемом обеспечивает формирование выходных сигналов интерфейса RS232C.

Регистр состояния линии обеспечивает хранение информации о состоянии процесса приема или передачи.

Регистр состояния модема обеспечивает хранение информации о входных сигналах интерфейса RS232C и имитацию этих сигналов в диагностическом режиме.

4.7.2 Параллельный порт

Термин «параллельный порт» или параллельный интерфейс, еще известный под названием Centronics, является почти синонимом «принтерному порту». Параллельный порт — это самое простое средство для подключения принтера к вашей PC. Подсоедините его к этому порту и будет удивительно, если принтер будет работать с изъянами. Параллельный порт является одним из немногих разъемов PC, которые не требуют особой щепетильности.

Параллельная передача данных. Параллельные порты названы подходящим именем. Они передают информацию по 1 байту за раз, используя для этих целей 8 раздельных проводников — по одному на каждый бит. Эти биты информации передаются одновременно с одной и той же скоростью по индивидуальным проводникам.

Так как используются 8 линий передачи информации, информация может потенциально передаваться в 8 раз быстрее, чем по одной линии. С другой стороны, параллельные порты предполагают использование дорогих кабелей. Так же использование большого числа параллельных линий накладывает ограничения на длину кабеля.

Программирование параллельного порта. Формирование и прием сигналов интерфейса Centronics обеспечивается записью и чтением портов ввода/вывода.

Программные и аппаратные средства PC AT поддерживают до трех портов Centronics, которые принято обозначать LPT1, LPT2 и LPT3.

Поскольку PC обеспечивает вывод данных со скоростью до 150 Кб в секунду, принтер должен иметь возможность приостанавливать эту процедуру. В интерфейсе Centronics предусмотрено два варианта приостановки процесса вывода. Обычно PC работает с принтеров в режиме опроса. В этом случае для приостановки вывода данных в принтер используется сигнал BUSY.

Цикл опроса состояния выглядит так:

```
Mov dx,status; адрес соответствующего регистра состояния
@@1:   in al,dx            ; считывание данных их порта
test al,80h  ; проверка на установку разряда
jz @@1
```

Если программа предусматривает работу Centronics с использованием сигнала на запрос прерывания, то перед началом вывода данных необходимо размаскировать прерывание путем записи нуля в пятый разряд регистра управления. После передачи каждого знака из PC в принтер вырабатывается сигнал на прерывание.

Вопросы для самопроверки

1. Расскажите историю видеоадаптеров.
2. Приведите основные параметры стандарта VGA.
3. Приведите основные параметры интерфейса VESA.
4. Опишите процесс функционирования клавиатуры.
5. Какие режимы поддерживает клавиатура с 101 клавишей?
6. Какие способы клавиатуры вы знаете?
7. Что собой представляет структура жесткого диска?
8. Расскажите о способах работы с жесткими дисками.
9. Какие методы чтения счетчиков таймера существуют?
10. Как устроен контролер прерываний 8259A?

11. Какой порт необходимо использовать для работы с подсистемой реального времени?
12. В чем отличие программирования последовательного и параллельного портов?

Литература

1. Assembler / В. Юров. — СПб, 2001. – 624 с.: ил.
2. Руководство по архитектуре IBM PC AT/ Ж.К. Голенкова, А.В. Заблоцкий, М.Л. Мархасин и др.; Под общ. Ред. М.Л. Мархасина. — Мн.: ООО «Консул», 1992. — 949 с.: ил.
3. Intel Architecture Software Developer’s Manual. T1-3.
4. VESA BIOS Extension (VBE) Core Functions Standard Version 2.0/ Video Electronics Standard Association, 1994. 93 с.

Контрольные работы

Контрольная работа №1

Выполняется в виде электронного теста.

Контрольная работа №2

Написать программу с использованием языка Turbo Assembler, в которой пользователь вводит числа, производит с ними какие-либо арифметические или логические операции (в зависимости от варианта) и выводит результат на экран. Программа высылается в составе исполняемого файла и исходных текстов. Каждая строка программы должна сопровождаться подробными комментариями.

Вариант 1

Пользователь вводит два числа A и B в десятичном виде через процедуру, осуществляя передачу результата ввода через регистры. Программа должна:

1. Посчитать C=A+B. Установить все четные биты C. Расчет должен быть выполнен с использованием макроопределения.
2. Вывести на экран число C и все промежуточные числа в двоичном виде, с использованием процедуры, в которую осуществляется передача параметров через регистры.

Вариант 2

Пользователь вводит два числа A и B в десятичном виде через процедуру, осуществляя передачу результата ввода через стек. Программа должна:

1. Посчитать C=A-B. Установить все четные биты C. Расчет должен быть выполнен с использованием макроопределения.
2. Вывести на экран число C и все промежуточные числа в двоичном виде, с использованием процедуры, в которую осуществляется передача параметров через стек.

Вариант 3

Пользователь вводит два числа A и B в десятичном виде через процедуру, осуществляя передачу результата ввода через общую область памяти. Программа должна:

1. Посчитать C=A+B*2. Если третий бит числа C установлен, то вывести на экран C и все промежуточные числа в двоичном виде, в противном случае, вывести на экран C/2 и все промежуточные числа в двоичном виде. Расчет должен быть выполнен с использованием макроопределения.
2. Вывод на экран чисел, с использованием макроопределения.

Вариант 4

Пользователь вводит два числа A и B в десятичном виде через макрос. Программа должна:

1. Посчитать C=A/2+B. Установить все нечетные биты C. Расчет должен быть выполнен с использованием процедуры с передачей параметров через директивы.
2. Вывести на экран число C и все промежуточные числа в двоичном виде, с использованием макроса.

Вариант 5

Пользователь вводит два числа A и B в десятичном виде через процедуру, осуществляя передачу результата ввода через общую область памяти. Программа должна:

1. Посчитать C=(A+B)/4. Сбросить пятый бит числа C, если он установлен. Расчет должен быть выполнен с использованием макроопределения.
2. Вывести на экран число C и все промежуточные числа в двоичном виде, с использованием процедуры, в которую осуществляется передача параметров через общую область памяти.

Вариант 6

Пользователь вводит два числа A и B в десятичном виде через макрос. Программа должна:

1. Посчитать C=(A-B)*3. Выполнить циклический сдвиг полученного числа C на 3 бита вправо. Расчет должен быть выполнен с использованием макроопределения.

2. Вывести на экран число C и все промежуточные числа в двоичном виде, с использованием процедуры, в которую осуществляется передача параметров через регистры.

Вариант 7

Пользователь вводит два числа A и B в десятичном виде через процедуру, осуществляя передачу результата ввода через стек. Программа должна:

1. Посчитать C=A/2+B. Выполнить арифметический сдвиг C на 3 бит влево. Расчет должен быть выполнен с использованием макроопределения.
2. Вывести на экран число C и все промежуточные числа в двоичном виде, с использованием макроопределения.

Вариант 8

Пользователь вводит два числа A и B в десятичном виде через процедуру, осуществляя передачу результата ввода через стек. Программа должна:

1. Посчитать C=A+B*2. Обнулить все четные биты C. Расчет должен быть выполнен с использованием процедуры и передачи параметров также через стек.
2. Вывести на экран число C и все промежуточные числа в двоичном виде, с использованием процедуры, в которую осуществляется передача параметров через стек.

Вариант 9

Пользователь вводит два числа A и B в десятичном виде через макроопределение. Программа должна:

1. Посчитать C=A+(B-5h)*2. Расчет должен быть выполнен с использованием макроопределения.
2. Если установлен четвертый бит числа C то вывести на экран A и все промежуточные числа в десятичном виде, в противном случае вывести на экран число B и все промежуточные числа в десятичном виде. Вывод оформить в виде процедуры с передачей параметров через директивы extrn и public.

Вариант 10

Пользователь вводит два числа A и B в десятичном виде через макроопределение. Программа должна:

1. Посчитать C=(A+12h)/2+B. Обнулить все четные биты C. Расчет должен быть выполнен с использованием макроопределения.
2. Вывести на экран число C и все промежуточные числа в двоичном виде, с использованием процедуры, в которую осуществляется передача параметров через общую память.