МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное
учреждение высшего образования
«СЕВАСТОПОЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

В.Г. Козырев

АРХИТЕКТУРА КОМПЬЮТЕРНЫХ СИСТЕМ

Учебное пособие

Севастополь
СевГУ
2017

ББК 32.973.23-02
УДК 681.3(075.8)

Р е ц е н з е н т ы:
Ю.И. Нечаев - профессор кафедры вычислительной техники и информационных технологий Санкт-Петербургского морского технического университета, заслуженный деятель науки РФ, д.т.н., профессор
С.А. Орлов - директор суперкомпьютерного центра Томского государственного университета, к.ф-м.н, доцент
Ю.К. Апраксин - профессор кафедры информационных технологий и компьютерных систем Севастопольского государственного университета, д.т.н., профессор

Козырев В.Г.
Архитектура компьютерных систем: учеб. пособие. – Севастополь: СевГУ, 2017. – 250 с.

В пособии изложены теоретические основы архитектурного построения компьютерных систем: порядок обработки потока машинных команд, набор программно-доступных регистров компьютера, форматы и режимы их адресации, система памяти и каналы ввода-вывода, организация обмена информацией между органами компьютера, система прерываний, способы защиты памяти, типовые машинные команды. Общие положения демонстрируются на примере одной из наиболее распространенных в настоящее время процессорных архитектур IA-32.

Рассмотренная проблематика актуальна для специалистов по системной инженерии и прикладному программированию, занимающихся разработкой и эксплуатацией компьютеризированных систем различного технического назначения.

Предназначена для студентов университетов, обучающихся по направлению подготовки 27.03.04 – Управление в технических системах.

ББК 32.973.23-02
УДК 681.3(075.8)

© ФГАОУВО «Севастопольский государственный университет», 2017

ОГЛАВЛЕНИЕ

Предисловие…….. 5

Введение…… 6

Глава 1. Компьютеры. Основные понятия……. 8

1.1. Понятие компьютера. Аппаратура и программное обеспечение…… 8

1.2. Процессор…… 16

1.3. Память…….. 32

1.4. Контроллеры и порты ввода-вывода……..... 36

1.5. Системная магистраль…… 48

1.6. Периферийные устройства…… 49

1.7. Интерфейс…… 49

Глава 2. Элементы архитектуры ЭВМ…… 53

2.1. Понятие об архитектуре и организации компьютера…….. 53

2.2. Особенности архитектуры и организации процессоров Intel…. 58

2.3. Регистры процессора…….. 68

2.4. Сегменты в памяти……. 76

2.5. Регистр флагов…… 88

2.6. Арифметические операции над двоичными числами……. 93

2.7. Стек.……. 108

Глава 3. Обмен информацией в ЭВМ…….. 116

3.1. Структура аппаратуры и потоки информации ЭВМ……..... 116

3.2. Базовые машинные операции…… 123

3.3. Операции внешних пересылок……. 126

3.4. Машинные циклы…….. 134

3.5. Важное заключение…… 138

3.6. Эволюция шинной архитектуры…….. 139

Глава 4. Порядок работы компьютера…….…….. 143

4.1. Программный процесс…….…….. 143

4.2. Информация в ЭВМ. Программа и данные. Структура команды…….…… 143

4.3. Размещение программы в основной памяти…….….. 153

4.4. Продвижение процессора по программе …… 154
4.5. Особенности продвижения по программе процессоров Intel… 156
4.6. Пуск ЭВМ и ее функционирование после пуска. Командные циклы…… 162
4.7. Три режима ввода-вывода…….. 168
4.8. Разложение команд процессоров Intel по машинным циклам.. 171
Глава 5. Прерывания программы…… 175
5.1. Понятие прерываний…… 175
5.2. Обработка внешних прерываний…… 180
5.3. Разрешение и запрещение прерываний……... 181
5.4. Обработка внутренних и программных прерываний…….. 182
5.5. Прерывания и исключения процессоров Intel……. 183
5.6. Обработка прерываний и исключений процессорами Intel…… 188
5.7. Приоритеты прерываний и исключений…….. 194
5.8. Прерывание – эффективный механизм функционирования ЭВМ…… 196
Глава 6. Защита памяти …… 197
6.1. Методы защиты…….. 197
6.2. Защита приложений (прикладная защита)…….. 202
6.3. Защита по привилегиям (системная защита). Кольца защиты.. 205
6.4. Передача управления между уровнями привилегий…….. 211
Список литературы……..…….. 216
Приложение А. Состав программного обеспечения ЭВМ…….. 218
А.1. Системное ПО…… 218
А.2. Прикладное ПО…… 225
Приложение Б. Сводка прикладных команд IA-32…….. 227
Б.1. Функциональная классификация команд…… 227
Б.2. Обозначения…… 229
Б.3. Команды обработки данных……. 231
Б.4. Команды передачи управления …… 243
Б.5. Команды изменения/проверки состояния процессора…….. 248

ПРЕДИСЛОВИЕ

Настоящее учебное пособие предназначается для студентов направления подготовки «Управление в технических системах». Оно написано по материалам лекций, читанных автором на протяжении многих лет в Севастопольском государственном университете по дисциплинам «Архитектура компьютерных систем», «Микропроцессорные системы», «Проектирование и программирование МП систем» и другим. При попытках рекомендовать по этим дисциплинам учебную литературу обнаруживалось, что она не в полной мере отвечает потребностям указанной специальности. Она либо чересчур детализирует описание аппаратных средств компьютера, процессора и других компонентов, рассматривая их вплоть до принципов работы отдельных вентилей и временных диаграмм сигналов (как, например [6, 7, 22]), либо излагает только состав машинных команд и язык ассемблера ([1-3]). Ряд учебников ([10, 11, 17, 24-26]) достаточно энциклопедичны, но имеют большой объем, превышающий возможности учебного плана по указанному направлению подготовки. Иные содержащиеся в них сведения мало актуальны для прикладных специалистов, занимающихся не разработкой, а применением промышленно выпускаемых процессоров и микросхем окружения. Упомянутые обстоятельства и побудили автора к написанию собственной книги.

При работе над ней автор исходил из соображения, что материал по архитектуре компьютерных систем, независимо от степени его теоретического или прикладного характера, должен опираться на общую систему основополагающих принципов, отражающих построение и функционирование любых вычислительных систем. Вместе с тем книга должна содержать описание конкретных типов микропроцессорных устройств, иначе общие положения, не подкрепленные конкретными формами их реализации, не могут быть достаточно хорошо усвоены студентами. Автор постарался воплотить указанные соображения в настоящем учебном пособии, стремясь добиться систематического, глубокого и всестороннего осуществления намеченных идей.

С целью наиболее аккуратного представления сложившихся к настоящему моменту взглядов на архитектуру компьютеров пособие составлено с опорой на материалы многих известных литературных источников. Их список приводится в конце учебника. Выбор этих книг связан с их весьма высоким научным и учебно-методическим потенциалом. Цитируемые места из них берутся в кавычки и/или даётся ссылка на источник.

Автор надеется, что предлагаемое учебное пособие окажется полезным для студентов университетов, обучающихся по программам подготовки специалистов в области информационных технологий и систем управления техническими объектами.

ВВЕДЕНИЕ

Архитектура компьютерных систем представляет собой тот раздел компьютерных наук, который позволяет, на его основе, вести разработку компьютерных систем, не вникая в детали внутреннего построения отдельных компонентов компьютера – микрочипов процессора, устройств памяти, контроллеров ввода-вывода и других. Для решения возникающих задач разработчику необходимо располагать лишь сведениями о внутренних, программно доступных элементах и внешних сигналах устройств. Внутренняя организация компонентов скрыта от пользователя, им не контролируется, а ее ресурсы вовлекаются в функционирование системы путем активизации их через внешний интерфейс и программно доступные элементы. Совокупность этих, «видимых» для пользователя, элементов и их описание мы традиционно и называем архитектурой компьютера.

Поскольку функции комплектующих изделий предопределены и предоставляются в распоряжение пользователя в неизменном виде, то разработка компьютерной системы сводится по существу к разработке ее интерфейса – средств сопряжения готовых изделий между собой, а также к оснащению новой системы целевыми машинными программами. Те же самые сведения нужны для обслуживания и эксплуатации готовых компьютерных систем. Указанное обстоятельство существенно упрощает работу специалистов. Собственно, придание удобных пользовательских свойств процессору и другим компонентам компьютера, обеспечивающих простоту в применении и функциональную полноту, составляет одну из целей компаний производителей. В силу сказанного учебная дисциплина «Архитектура компьютерных систем» занимает важное место в системе подготовки технических специалистов.

Настоящее учебное пособие ориентировано на студентов, обучающихся по направлению подготовки «Управление в технических системах», и может быть использовано также по смежным направлениям, таким как «Информатика и вычислительная техника», «Информационные системы и технологии» и др. В нем излагаются элементы архитектуры компьютеров пользовательского и управляющего типов. Особенности архитектуры суперкомпьютеров не затрагиваются, для их изучения служат отдельные дисциплины. Общие положения демонстрируются на примере одной из наиболее распространенных в настоящее время процессорных архитектур – архитектуры IA-32 семейства Intel 80x86 и программно-совместимых с ним процессоров, на которых построена значительная часть компьютеров указанных типов.

В первой главе пособия дается описание структуры компьютера и места, занимаемого архитектурой в системе его аппаратного и программного обеспечения. Вторая глава посвящена обсуждению архитектуры набора команд, или, как говорят, архитектуры в узком смысле. Третья и четвертая главы затрагивают вопросы построения архитектуры в широком смысле, акцентируясь на принципах аппаратной реализации и организации интерфейса систем. Пятая

содержит изложение механизмов обработки прерываний процессором, шестая – методов защиты памяти компьютера.

Рассмотренные вопросы актуальны для специалистов по системной инженерии, системному и прикладному программированию, давая им общую научно-методологическую направленность методов создания и анализа компьютерных систем различного назначения.

Глава 1. КОМПЬЮТЕРЫ. ОСНОВНЫЕ ПОНЯТИЯ

1.1 Понятие компьютера. Аппаратура и программное обеспечение

Понятие ЭВМ. *Компьютером* или *электронной вычислительной машиной* (*ЭВМ*) называют *электронную, программируемую* (т.е. управляемую хранимой в памяти ЭВМ программой) машину для *автоматической* обработки *цифровых* данных [12].

Под *данными* понимаются не только и не столько числа, а всякая информация в дискретном (цифровом) представлении: всевозможные тексты, сигналы, изображения и т.п.

Компьютер, в отличие от программируемого калькулятора, обладает, кроме того, *высокой производительностью*, *большим объемом памяти* и *развитой многоуровневой организацией* [25].

Термины «компьютер» и «ЭВМ» имеют равноправное хождение. При возможном появлении в будущем компьютеров, работающих на оптических, квантовых и других физических принципах, понятие «электронная вычислительная машина», отражающее электронную природу процессов в машине, станет более узким, чем понятие «компьютер». Однако общее выражение «вычислительная машина» вполне равнозначно термину «компьютер».

Компьютерной или *электронной вычислительной системой*, *вычислительной системой* обычно называют *компьютер* или *несколько компьютеров в комплексе с периферийным оборудованием* и *специальным программным обеспечением для определенного применения или класса применений*. Например: 1. Компьютерная система разработки программ для встраиваемых микроконтроллеров*; 2. Информационно-управляющая вычислительная система (ИУВС), например бортовой компьютер на самолете в составе автопилота; 3. Компьютерная система автоматизированного обучения; 4. Система автоматизированного проектирования (САПР) (называемая также АРМ – автоматизированное рабочее место или Workbench – буквально «рабочая скамья»).

* Микроконтро́ллер (МК; англ. Micro Controller Unit, MCU; controller – регулятор, управляющее устройство) – микросхема, предназначенная для управления различными устройствами – бытовыми приборами, периферийными аппаратами компьютеров, техническим оборудованием в промышленности, на транспорте и др. отраслях экономики. Типичный микроконтроллер содержит на одном кристалле процессор, ОЗУ и (или) ПЗУ, а также периферийные компоненты: устройства ввода-вывода, таймеры и др. По сути, микроконтроллер – это однокристальный компьютер, способный выполнять относительно простые задачи. Впрочем, ранее для обозначения МК как раз и употреблялся термин «однокристальная микро ЭВМ»

При оснащении компьютерной системы соответствующими устройствами ввода-вывода, например аналого-цифровыми и цифро-аналоговыми преобразователями, она может производить обработку не только цифровой, но и аналоговой информации, что характерно для ИУВС, большинства МК и других управляющих компьютеров. Иными словами, ЭВС – это проблемно ориентированная ЭВМ или комплекс ЭВМ.

Коротко: *Компьютер – это универсальная машина для обработки информации. Компьютерная система – специализированное применение компьютера.*

Обработка данных, обработка информации в информатике – это любое преобразование информации из одного вида в другой, производимое по строгим формальным правилам – правилам, не допускающим никаких отклонений, двойного толкования, выполняемых однозначно, точно и неукоснительно. Обработка производится в форме систематизированной последовательности операций, совершаемых с данными, прежде всего в компьютере, для получения новой информации путем вычислений, пересмотра и уточнения имеющейся информации, хранящейся в основной памяти компьютера, на магнитном или оптическом диске, магнитной ленте, флэш-карте и др.

Данные в компьютере имеют вид цифр (двоичных чисел), представляющих числовую, текстовую, графическую и звуковую информацию. К ней могут относиться результаты измерений и вычислений, научные или технические факты, перечни названий, имен, книжных заглавий, тексты книг, кинофильмы, музыкальные записи и т.п. информация.

Основные операции обработки данных, выполняемые компьютером на *нижнем, аппаратном уровне* – уровне реализации машинных команд, – это *арифметические* операции сложения, вычитания, умножения и деления, *логические* операции и *сдвиги*, *пересылки*, а также операции *принятия решений* на основе сопоставления данных. Последние достигаются командами типа: «Если выполняется условие *a*, выполнить команду программы *P*, если *a* не выполняется, следовать команде *Q*».

В структуре возможных *высокоуровневых операций* над данными, реализуемых уже не отдельными машинными командами, а машинными программами, можно выделить следующие основные:

- *сбор данных* – накопление информации с целью обеспечения достаточной полноты для принятия решений;
- *формализация данных* – приведение данных, поступающих из разных источников, к одинаковой форме, чтобы сделать их сопоставимыми между собой, то есть повысить их уровень доступности;
- *фильтрация данных* – отсеивание «лишних» данных, в которых нет необходимости для принятия решений; при этом должен уменьшаться уровень «шума», а достоверность и адекватность данных должны возрастать;
- *сортировка данных* – упорядочение данных по заданному признаку с целью удобства использования; повышает доступность информации;
- *архивация данных* – организация хранения данных в удобной и легкодоступной форме; служит для снижения экономических затрат по долгосрочному хранению данных или передаче их по сети в экономном, сжатом виде и повышает общую надежность информационного процесса в целом;
- *защита данных* – комплекс мер, направленных на предотвращение утраты, несанкционированного воспроизведения и модификации данных;
- *транспортировка данных* – прием и передача (доставка и поставка) данных между удаленными участниками информационного процесса; при этом источник данных в информатике принято называть *сервером,* а потребитель – *клиентом;*
- *преобразование данных* – видоизменение данных, направленное на выполнение условий того или иного метода обработки. Например, для транспортировки цифровых потоков данных по каналам телефонных сетей необходимо преобразование цифровых данных в некое подобие звуковых сигналов, чем и занимаются специальные устройства – телефонные модемы. В информационно-управляющих вычислительных системах преобразование данных включает математические расчеты управляющих воздействий, подаваемых на технические объекты в соответствии с целью управления. Математические

расчеты разного рода осуществляются также при решении научных, конструкторских и инженерных задач.

Аппаратура и программное обеспечение [5]. В ЭВМ или системе прежде всего различают *аппаратуру* (*аппаратные, аппаратурные средства*) и *программное оснащение*, или *обеспечение* (*программные средства*) (ПО). Под аппаратными средствами понимают разнообразные механические, электромеханические, магнитные и электронные элементы, устройства, приборы, блоки, из которых реализована машина. К программным средствам (ПО) относят все программы, связанные с машиной. Обычно в ПО включают также языки программирования и, иногда, документацию, описания и руководства по языкам и программам. К ПО можно относить, кроме того, всевозможные магнитные, электронные и другие вещественные носители информации с записанными на них программами.

По традиции аппаратуру, в противоположность программам, принято считать жесткой, не изменяемой частью системы (так называемый «жесткий товар» – hard ware). В то же время программы считаются мягкой, изменяемой частью системы («мягкий товар» – soft ware). Это верно в том отношении, что на фиксированной конфигурации аппаратуры можно выполнять самые разные и создавать новые программы. Что же касается изменчивости программ, то готовые, отлаженные программы менее подвержены изменениям, чем аппаратура, и можно указать множество примеров создания новых компьютеров под существующее программное обеспечение.

Дело в том, что разработка программ крайне трудоемка, причем в то время, как стоимость аппаратуры с усовершенствованием технологии и автоматизацией производства неуклонно и быстро снижается, удешевить создание программ не удается. Во многих компьютерных системах стоимость программного оснащения в десятки раз превышает стоимость аппаратуры. И вполне закономерной представляется тенденция перепоручить аппаратуре все больше функций, осуществлявшихся до того программным путем, а также фиксировать программы путем неизменяемой записи в постоянные запоминающие устройства.

Как видно, разделение функций между аппаратурой и программами не является ни однозначным, ни неизменным. Более того, такие «гибриды», как программы в ПЗУ или микропрограммы в управляющей памяти процессора невозможно безоговорочно отнести ни к аппаратуре, ни к программному оснащению. [5]

Основные компоненты ЭВМ перечислены в табл. 1.1.

Таблица 1.1 – Состав компьютера

<table>
<tr><td colspan="4">ЭВМ</td></tr>
<tr><td colspan="2">Аппаратура</td><td colspan="2">Программы</td></tr>
<tr><td>Центральная часть (ядро):
• ЦП – центральный процессор
• ОП – основная память
• КВВ – контролеры ввода/вывода
• Внутренняя магистраль</td><td>ПУ – периферийные устройства:
• УСП – устройства для связи ЭВМ с пользователем
• Внешняя память
• УСО – устройства для связи ЭВМ с техническими объектами
• СТ – средства телекоммуникации</td><td>Системное ПО:
• ОС – операционная система
• ИС – инструментальные системы (системы программирования: C++, Pascal и т.п.)
• Сервисные программы</td><td>Прикладное ПО:
• Расчетные (Mathcad, Matlab, Ansys) и конструкторские (Acad)
• системы управления базами данных (dBase, Visual FoxPro) и табличные процессоры (Excel)
• Управляющие программы для систем автоматического управления; др. программы</td></tr>
</table>

Структура аппаратуры. В аппаратуре компьютера или системы различают внутреннюю и внешнюю (центральную и периферийную) части (рис. 1.1).

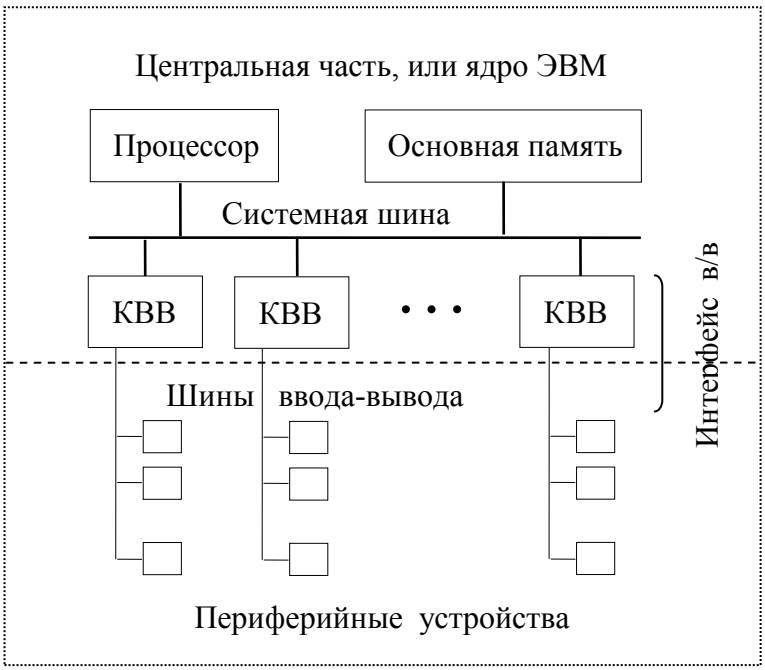


Рис. 1.1. Структура аппаратных средств ЭВМ.
КВВ – контроллер ввода-вывода

К **внутренней (центральной) части**, или **ядру** компьютера – **собственно компьютеру** относятся:

1) Центральный процессор (ЦП) или комплекс процессоров и непосредственно взаимодействующие с ЦП органы:
2) Основная (главная, первичная, оперативная) память (ОП);
3) Контроллеры ввода-вывода (КВВ);
   и сопрягающая все эти органы между собой
4) Системная (внутренняя, внутримашинная) магистраль («шина») или магистрали («шины»).

Периферийные устройства, или «периферию», принято разделять на:

1) Устройства ввода и вывода данных для связи компьютера с пользователем (УСП);
2) Устройства внешней (периферийной, вторичной, массовой) памяти;
3) Устройства ввода и вывода данных для связи компьютера с техническими объектами – датчиками и исполнительными механизмами (рулевыми машинками и др.) (УСО);
4) Средства телекоммуникации (телекоммуникационные средства), предназначенные для передачи данных на большие расстояния.

Типичный состав аппаратуры персонального компьютера показан на рис. 1.2. Отдельные органы компьютера рассматриваются в п. 1.2-1.6. (Сокращение «п.» обозначает параграф; в данном случае параграфы 1.2-1.6.)

Структура программного обеспечения. В отличие от аппаратуры программное оснащение, или обеспечение (ПО) не разделяют на центральное и периферийное, или внутреннее и внешнее. ПО делится на *системное* (*общее*) и *прикладное* (*специальное, функциональное*) [5]. Программы, относящиеся к системному ПО, называются *системными*, а к прикладному ПО – *прикладными*. Описание состава системных и прикладных программ компьютера можно найти в приложении А.

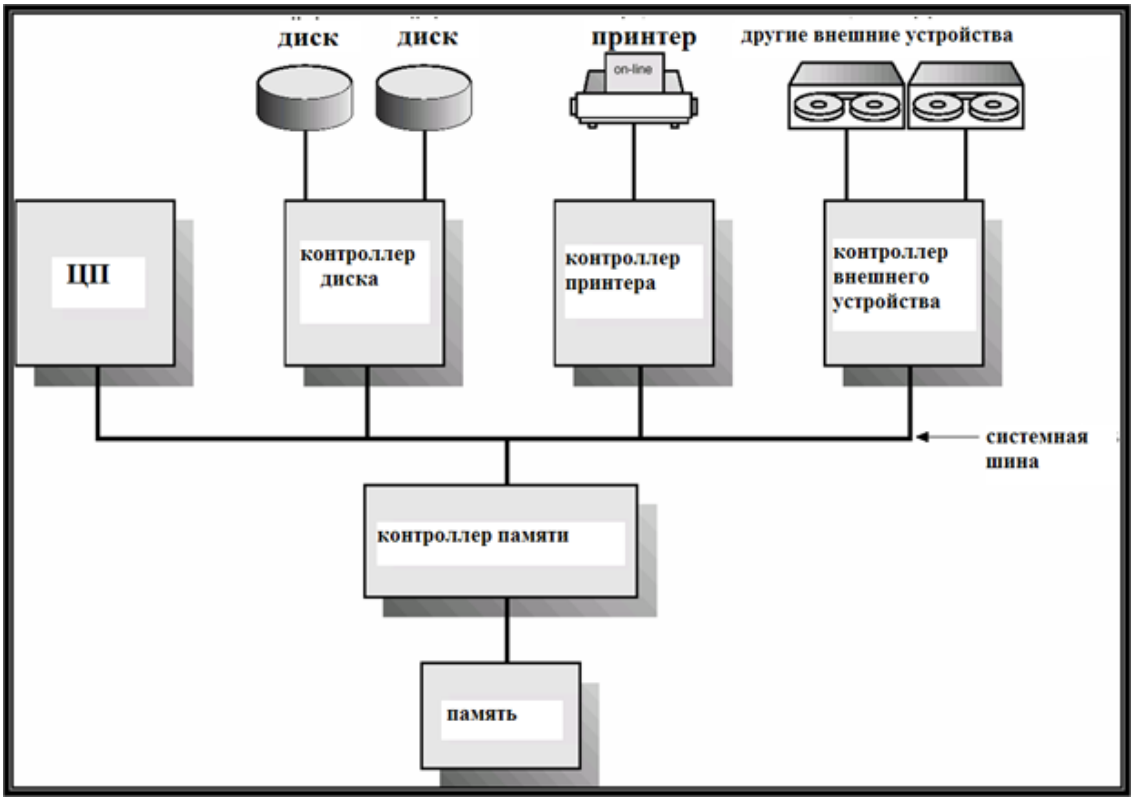


Рис. 1.2. Структура персонального компьютера

Классификация компьютерных систем [20]. Современные компьютеры представлены большим разнообразием устройств – от *суперкомпьютеров* до *мобильных приборов*.

Суперкомпьютеры (*super-computers*) – мощные многопроцессорные компьютеры, наиболее современные из которых имеют производительность до нескольких *petaflops* (10^{15} вещественных операций в секунду; аббревиатура *flops* расшифровывается как *floating-point operations per second*). Пример – суперкомпьютер «Ломоносов», установленный в МГУ. Суперкомпьютеры используются для вычислений, требующих больших вычислительных мощностей, сверхвысокой производительности и большого объема памяти. В реальной практике это, прежде всего, задачи моделирования – например, моделирования климата в регионе и прогнозирования на основе построенной модели погоды в данном регионе на ближайшие дни. Особенностью суперкомпьютеров является их параллельная архитектура – как правило, все они являются многопроцессорными. Соответственно, ОС для суперкомпьютеров должны поддерживать распараллеливание решения задач и синхронизацию параллельных процессов, одновременно решающих подзадачи некоторой программы.

Многоцелевые компьютеры, или *компьютеры общего назначения (mainframes)* – традиционное историческое название для компьютеров, распространенных в 1950-х – 1970-

х гг., еще до эпохи всеобщего распространения персональных компьютеров. Другое их название – *большие* и *малые*, или *мини, ЭВМ*. Именно для mainframe-компьютеров («больших и мини ЭВМ») создавались первые ОС. Типичные примеры таких компьютеров: IBM 360/370; из отечественных – М-220, БЭСМ-6. На таких компьютерах решались все необходимые задачи – от расчета зарплаты сотрудников в организации до расчета траекторий космических ракет. Подобный компьютер выглядел достаточно неуклюже и громоздко и мог занимать целый большой зал. Параметры ранних mainframe-компьютеров были весьма скромными: быстродействие – несколько тысяч операций в секунду, оперативная память – несколько тысяч ячеек (слов). Недостаточно удобным был пользовательский интерфейс (интерактивное взаимодействие с компьютерами было реализовано гораздо позже, в 1960-х гг.). Тем не менее, на таких компьютерах решались весьма серьезные задачи оборонного и космического назначения. С появлением персональных и портативных компьютеров классические mainframe-компьютеры ушли в прошлое. Однако следует подчеркнуть, что именно в операционных системах для mainframe-компьютеров были реализованы все основные методы и алгоритмы, которые впоследствии были использованы в ОС для персональных (настольных и карманных) компьютеров и мобильных устройств.

Кластеры компьютеров (computer clusters) – группы компьютеров, физически расположенные рядом и соединенные друг с другом высокоскоростными шинами и линиями связи. Кластеры компьютеров используются для высокопроизводительных параллельных вычислений. Наиболее известны в мире компьютерные кластеры, расположенные в исследовательском центре CERN (Швейцария) – том самом, где находится большой адронный коллайдер. Как правило, компьютерные кластеры располагаются в исследовательских институтах и в университетах, в том числе, например, в Петродворцовом учебно-научном комплексе СПбГУ они используются в Петродворцовом телекоммуникационном центре (ПТЦ), на математико-механическом и физическом факультетах. Операционная система для кластеров должна, помимо общих возможностей, предоставлять средства для конфигурирования кластера, управления компьютерами (процессорами), входящими в него, распараллеливания решения задач между компьютерами кластера и мониторинга кластерной компьютерной системы. Примерами таких ОС являются ОС фирмы Microsoft – Windows 2003 for clusters; Windows 2008 High-Performance Computing (HPC).

Настольные компьютеры (desktops) – это наиболее распространенные в настоящее время компьютеры, которыми пользуются дома или на работе все люди, от школьников и студентов до домашних хозяек. Такой компьютер размещается на рабочем столе и состоит из монитора, системного блока, клавиатуры и мыши. Параметры современного настольного компьютера, наиболее приемлемые для использования современных ОС: быстродействие (тактовая частота) процессора 1 – 4 ГГц, оперативная память – 1-16 гигабайт и более, объем жесткого диска (hard disk drive – HDD) – 200 Гб - 2 Тб и до 4 Тб (1 терабайт, Тб = 1024 Гб). Все разнообразие современных операционных систем (Windows, Linux и др.) – к услугам пользователей настольных компьютеров. При необходимости на настольном компьютере можно установить две или более операционных системы, разделив его дисковую память на несколько разделов (partitions) и установив на каждый из них свою операционную систему, так что при включении компьютера пользователю предоставляется стартовое меню, из которого он выбирает нужную операционную систему для загрузки.

Для обозначения компьютеров, имеющих небольшие физические размеры и выполненных на микропроцессорах и других микросхемах высокой степени интеграции, (к которым относятся настольные и другие подобные компьютеры) раньше использовался термин «*микро ЭВМ*», «*микрокомпьютер*». В 90-е годы XX века он вышел из употребления, когда размер таких компьютеров стал считаться обычным. Аналогично и *микропроцессоры*, т.е. процессоры, изготовленные по технологии производства больших и сверхбольших интегральных электронных микросхем, стали именоваться чаще всего просто *процессорами*, поскольку иных технологий их производства к тому моменту уже не существовало. В XXI ве-

ке термин «микро ЭВМ» снова вошёл в употребление в связи с появившейся популярностью компьютеров, размером с банковскую карту и сопоставимых по мощности с более старыми персональными компьютерами (см. ниже *карманные* и *носимые компьютеры*).

Портативные компьютеры (laptops, notebooks – дословно «компьютеры, помещающиеся на коленях»; «компьютеры-тетрадки») – это миниатюрные компьютеры, по своим параметрам не уступающие настольным, но по своим размерам свободно помещающиеся в небольшую сумку или рюкзак или, например, на коленях пользователя, летящего в самолете в командировку и не желающего терять времени даром. Ноутбуки стоят обычно в несколько раз дороже, чем настольные компьютеры с аналогичными характеристиками. На ноутбуках используются те же операционные системы, что и для настольных компьютеров (например, Windows или MacOS).

Карманные портативные компьютеры и органайзеры (КПК, handhelds, personal digital assistants – PDA) – это «игрушки для взрослых» в виде миниатюрного компьютера, помещающегося на ладони или в кармане, но по своему быстродействию иногда не уступающего ноутбуку. При всей привлекательности, серьезные недостатки КПК – это неудобство ввода информации (приходится пользоваться палочкой-*стилусом*, – ведь не носить же с собой еще и громоздкую клавиатуру, – либо микроскопической выдвижной клавиатурой, на которой фактически тем же стилусом только и можно работать), а также неудобство чтения информации на маленьком экране. Операционные системы для КПК аналогичны ОС для ноутбуков, но все же учитывают более жесткие ограничения КПК по объему оперативной памяти.

Мобильные устройства (mobile intelligent devices – мобильные телефоны, коммуникаторы) – это устройства, которыми каждый из нас пользуется постоянно для голосовой связи, реже – для записи или обработки какой-либо информации или для выхода в Интернет. Наиболее важные параметры мобильного устройства – это по-прежнему качество голосовой связи и время автономной работы батареи. Однако все большее значение приобретают встроенные в них цифровые фото- и видеокамеры. Операционные системы для мобильных устройств отличаются большей компактностью, ввиду более жестких ограничений по памяти.

Носимые компьютеры (wearable computers) –для повседневной жизни достаточно экзотические устройства, однако для специальных применений (например, встроенные в скафандр космонавта или в кардиостимулятор) они жизненно важны. Разумеется, их память и быстродействие значительно меньше, чем у настольных компьютеров, но критическим фактором является их сверхвысокая надежность, а для их операционных систем и прочего программного обеспечения – минимальное возможное время ответа (response time) – интервал, в течение которого система обрабатывает информацию от датчиков, от пользователя или из сети, превышение которого грозит катастрофическими последствиями. С этой точки зрения, ОС для носимых компьютеров можно отнести к *системам реального времени*.

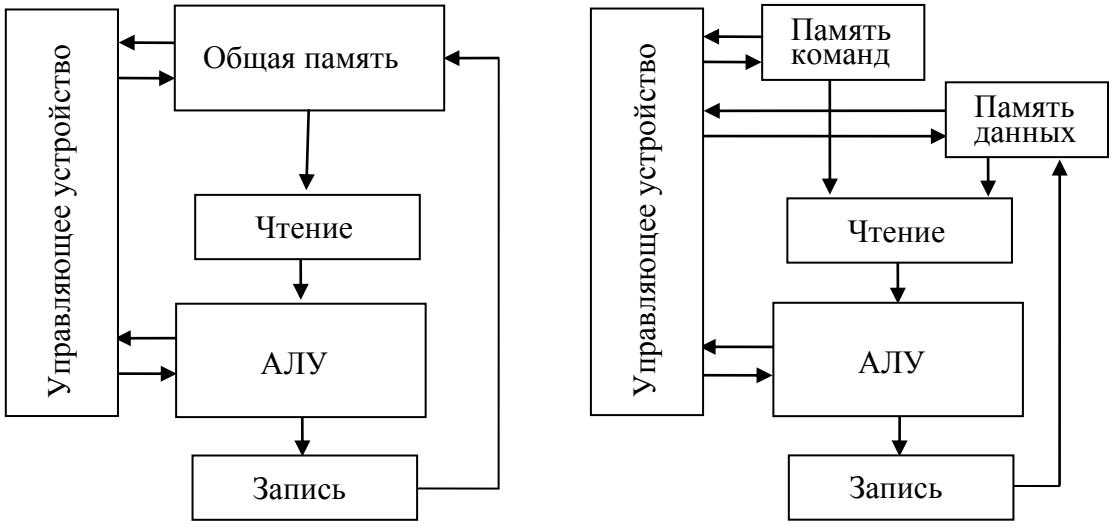
Распределенные системы (distributed systems) – это системы, состоящие из нескольких компьютеров, объединенных в проводную или беспроводную сеть. Фактически, таковы ныне все компьютерные системы (вспомните девиз «Сеть – это компьютер»). Все операционные системы должны, таким образом, поддерживать распределенный режим работы, средства сетевого взаимодействие, высокоскоростную надежную передачу информации через сеть.

Системы реального времени (real-time systems), или *управляющие компьютеры*, – вычислительные системы, предназначенные для управления различными техническими, военными и другими объектами в режиме реального времени, в темпе с процессом. Характеризуются основным требованием к аппаратуре и программному обеспечению, в том числе к операционной системе: недопустимость превышения времени ответа системы, т.е. ожидаемого времени выполнения типичной операции системы. Для ОС требования реального времени накладывают весьма жесткие ограничения – например, в основном цикле работы сис-

темы недопустимы прерывания (так как они приводят к неприемлемым временным затратам на их обработку).

Принстонская и гарвардская концепции построения вычислительных систем представляют собой наиболее общую совокупность идей, воплощенных в структуре ВС. Эти идеи были сформулированы в 40-х годах 20 века на этапе зарождения компьютерной техники и до сих пор остаются актуальными. Мы знаем их как гарвардскую и принстонскую (более известную как фон-неймановскую) архитектуры. [27]

Гарвардская архитектура была разработана Говардом Эйкеном, принстонская – Джоном (Яношем) фон Нейманом (математиком венгерского происхождения). Их основное отличие заключалось в том, что архитектура фон Неймана использовала единую память (и общую шину данных), а гарвардская предполагала наличие нескольких шин (в оригинале две: шина данных и шина команд), к одной из которых подключалась память данных, к другой – память команд (рис. 1.3).



Фон-неймановская архитектура Гарвардская архитектура

Рис. 1.3. Две концепции построения вычислительных систем

Согласно этой схеме в чистой архитектуре фон Неймана процессор в каждый момент времени может либо читать инструкцию, либо читать/записывать единицу данных из/в памяти. Оба действия одновременно происходить не могут, поскольку инструкции и данные используют один и тот же поток (шину).

В компьютере с использованием гарвардской архитектуры процессор может считывать очередную команду и оперировать памятью данных одновременно. Поэтому компьютер с гарвардской архитектурой при определенной сложности схемы быстрее, чем компьютер с архитектурой фон Неймана, поскольку потоки команд и данных расположены на раздельных физически не связанных между собой аппаратных каналах. Однако из-за более сложной схемы гарвардскую архитектуру используют обычно для построения простых спе-

циализированных машин, например, микроконтроллеров, а универсальные, например, настольные персональные компьютеры строятся по фон-неймановскому принципу.

Обобщая содержание рассмотренных подходов, можно назвать следующие общие принципы построения ВС, присущие принстонской концепции, называемые также *принципами фон Неймана*:

1) *Принцип двоичного кодирования*.
   Для представления данных и команд используется двоичная система счисления.
2) *Принцип однородности памяти*.
   Как программы (команды), так и данные хранятся в одной и той же памяти.
3) *Принцип адресуемости памяти*.
   Структурно основная память состоит из пронумерованных ячеек; процессору в произвольный момент времени доступна любая ячейка.
4) *Принцип последовательного программного управления*.
   Все команды располагаются в памяти и выполняются последовательно, одна после завершения другой, в порядке, определяемом программой.
5) *Принцип жесткости архитектуры.*
   Неизменяемость в процессе работы топологии, архитектуры, списка команд.

Компьютеры, построенные на этих принципах, относят к типу фон-неймановских.

Гарвардская концепция ВС отличается от принстонской во втором принципе, который для нее звучит так:

2) *Раздельное хранение команд и данных и их одновременная обработка*.
   Хранилище инструкций и хранилище данных представляют собой разные физические устройства; канал инструкций и канал данных также физически разделены.

В настоящей книге обсуждаются компьютеры с однородной памятью.

1.2. Процессор

Центральный процессор (ЦП), или просто «процессор» (Пр) представляет собой программно-управляемый (т.е. управляемый хранимой в памяти ЭВМ программой) функциональный орган ЭВМ, непосредственно производящий вычисления – арифметические и логические операции над данными и иные действия по их обработке. Помимо этого он руководит работой других устройств компьютера.

Процессор называют центральным потому, что наряду с ним в компьютере могут находиться другие – специализированные процессоры, например процессоры периферийных устройств (процессор клавиатуры и пр.).

Для обозначения процессора используют также аббревиатуры ЦПУ (центральное процессорное устройство), CPU (Central Processing Unit, дословно –

центральное обрабатывающее устройство), МП (микропроцессор), МП БИС (микропроцессорная большая интегральная схема) и др.

Современные ЦП, выполняемые в виде отдельных микросхем (чипов), реализующих все особенности, присущие данного рода устройствам, называют микропроцессорами. С середины 1980-х последние практически вытеснили прочие виды ЦП, вследствие чего термин «процессор» стал чаще всего восприниматься как обыкновенный синоним слова «микропроцессор». Тем не менее, это не всегда так: центральные процессорные устройства некоторых суперкомпьютеров даже сегодня представляют собой сложные комплексы больших (БИС) и сверхбольших (СБИС) интегральных микросхем.

Обработка данных в процессоре производится путем последовательного выполнения им некоторых операций над данными, проводимых по указке машинных команд. Команды поступают на процессор (считываются им) из *машинной программы*, хранимой в основной памяти ЭВМ. Программа представляет собой последовательность команд, записанных в памяти в том порядке, в каком они должны выполняться процессором. Команды и данные записываются в машине в виде двоичных чисел – комбинаций нулей и единиц. Каждая команда («инструкция») задает какую-нибудь одну, простую, элементарную операцию (например, сложить 2 числа). Процессор считывает команды из памяти и выполняет их в порядке считывания, одну после завершения другой. Программа является заданием процессору на решение определенной задачи. Поэтому программа по форме – это последовательность машинных команд, по содержанию – задание процессору на работу.* Сказанное означает, что все, что умеет делать компьютер, запрограммировано программистом и исполняется процессором.

* Согласно стандарту ISO 2382/1-84 машинная программа – это упорядоченный набор команд, подлежащий обработке.

Поскольку тактовые частоты современных процессоров приблизились к физическому пределу, для повышения их производительности применяется объединение нескольких копий процессора (так называемых «процессорных ядер», cores) в одном корпусе. Подобные приборы называются *многоядерными* процессорами. Существующие многоядерные процессоры имеют в своем составе от 2 до 32 ядер, параллельно работающих на общей памяти. В многоядерном процессоре поток команд, поступающий из памяти, разделяется (распараллеливается) внутри процессора между его ядрами, обрабатывающими каждое свою часть потока. Кроме многоядерных, в компьютерах применяются и *гибридные* процессоры, состоящие из центрального и графического (п.1.4) процессоров.

Машинная команда (*машинная инструкция*) служит приказом процессору на выполнение какой-нибудь простой операции. Электронные схемы каждого процессора могут распознавать и выполнять ограниченный набор простых команд, которые обычно не сложнее чем: сложить 2 числа; проверить, не является ли число нулем; сдвинуть код числа на один бит влево или вправо; скопировать кусок данных из одной части памяти в другую; и т.п. Более сложные действия производятся процессором посредством выполнения группы команд, т.е. программы. Все программы, на каком бы языке они не были бы написаны, перед выполнением должны быть превращены в последовательность команд [25].

Эти примитивные команды в совокупности составляют язык, на котором люди могут общаться с компьютером. Каждая команда на нем представлена в виде так называемого *опкода* (операционного кода) – двоичного кода отдельной операции. Такой язык называется *машинным языком*. Разработчик при создании нового компьютера должен решать, какие команды включать в машинный язык этого компьютера. Это зависит от назначения компьютера, от того, какие задачи он должен выполнять. Обычно стараются сделать машинные команды как можно проще, чтобы избежать сложностей при конструировании компьютера и снизить затраты на необходимую электронику [25].

Для удобства программирования вместо числовых опкодов, которые только и понимает процессор, обычно используют их условные буквенные обозначения – *мнемоники* команд. Набор таких мнемоник, вместе с некоторыми дополнительными возможностями, называется *языком ассемблера*.

Порядок функционирования процессора. Процессор ведет себя, в известном смысле, довольно просто. Он начинает функционировать сразу после пуска при условии, что к нему подводится электропитание и поступают сигналы от тактового генератора. Для пуска надо подать на процессор сигнал пуска (системного сброса, перезагрузки Сбр – Сброс, RESET). По этому сигналу процессор стартует – начинает отработку команд с некоторой пусковой ячейки памяти*, скажем, с нулевой (0). В этой ячейке должна храниться самая первая, стартовая команда компьютера. Номер пусковой ячейки заранее определен, он «известен» процессору, встроен в его электронную схему. Поэтому процессор всегда «знает», с какой ячейки памяти ему следует начать работу после пуска. Ему не надо сообщать об этом заранее.

* Память (основная память) состоит из большого числа одинаковых устройств – *ячеек*, каждая из которых хранит некоторую порцию информации. Устройство памяти и ее ячеек обсуждается в п.1.3 и 4.2.

После пуска процессор приступает к работе, он начинает выбирать и выполнять команды программы из памяти. Выборка и выполнение команд происходит последовательно, одна за другой, из очередных ячеек памяти, в которых они расположены, в том порядке, как они записаны в памяти. Отработка команд начинается, как мы сказали, с предопределенной – пусковой ячейки (например, с 0-й). Процессор выбирает (считывает) находящуюся в ней команду и расшифровывает ее. Благодаря этому он «узнаёт», какие операции над данными он должен выполнить по этой команде. «Узнав» состав операций, процессор принимается за их выполнение: читает число из ячейки памяти, или преобразует содержимое собственного регистра*, или совершает другие действия, предписанные командой. Выполнив эти действия, процессор переходит к следующей, второй команде программы. Для этого он обращается к следующей ячейке памяти, в которой хранится вторая команда, в нашем случае, к 1-й ячейке, и выбирает из нее и расшифровывает вторую команду. Расшифровав, он выполняет все операции над данными, заданные и этой командой. Затем, в таком же порядке процессор выбирает и выполняет третью команду из 2-й ячейки памяти,

четвертую – из 3-й ячейки и так далее, поочередно, одну за другой все другие команды программы.

* *Регистр* – устройство, которое (как и ячейка памяти) хранит некоторую порцию информации. Регистр состоит из набора триггеров (см. далее в этом параграфе). (Ячейки основной памяти тоже состоят из триггеров, т.е. являются своего рода микроскопическими регистрами, но могут также состоять из микроскопических емкостных элементов; п.4.2).

Подобным образом процессор функционирует при выполнении обычных команд – *команд обработки данных*. Эти команды предназначены для преобразования и пересылки *операндов* – данных, над которыми производятся операции. По указанию этих команд осуществляется сложение и вычитание чисел, передача кодов из регистра на регистр, запись результатов в память, чтение из памяти и т.д. После выполнения каждой такой команды процессор переходит к следующей, соседней ячейке памяти и выполняет команду из этой ячейки. Команды в этом случае выполняются в *линейном, естественном порядке*, так, как они размещены в памяти – от ячейки к ячейке (рис. 1.4). Такой порядок поддерживается также при выполнении команд изменения состояния процессора, упоминаемых ниже.

Если же по ходу программы процессору попадается *команда перехода* (*передачи управления*), то линейный порядок выполнения команд нарушается. Процессор ведет себя в этом случае иначе. Он прекращает линейное продвижение по программе и под действием команды перехода делает переход, «скачок» (go to, jump) вперед или назад по программе на новую ячейку, указанную командой перехода. Далее процессор продолжает счет уже с этой, новой, ячейки, номер который был задан командой перехода. Он начинает выбирать и выполнять команды из этой и следующих за ней ячеек. Например, если в 100-й ячейке находится команда «перейти на ячейку под номером 40», то после ячейки 100 процессор обратится не к 101-й ячейке, а вернется назад на 40-ю ячейку и начнет функционировать заново с этой, 40-й, ячейки памяти. В результате в программе образуется замкнутый цикл: процессор вновь будет выполнять команды из ячеек 40, 41, 42 и т.д. (см. рис. 1.4). Этот цикл будет бесконечным (с бесконечным числом повторений), если в ячейке 100 находится команда безусловного перехода на ячейку 40. Выход из него возможен только аппаратными средствами – перезапуском процессора сигналом «Пуск» или подачей сигнала прерывания. Если же используется команда перехода по условию (например, по условию повторения цикла, скажем, 7 раз), то после 7 повторений процессор уже не вернется назад, на начало цикла, а двинется дальше по программе (выйдет из цикла) и обратится за следующей командой к ячейке 101.

Образование циклов в программе является главным средством автоматизации вычислений и достижения производительного эффекта компьютера. С помощью команд передачи управления осуществляются циклы в программе, ветвления (переходы по условию), безусловные переходы, вызовы и возвраты из подпрограмм. Продвижение процессора по программе с помощью команд передачи управления называется *принудительным*. Поскольку номера ячеек на-

зывают также *адресами*, то говорят, что под действием команд передачи управления процессор переходит *по заданному адресу памяти*, а сам этот адрес называется *адресом перехода* (40 в примере на рис. 1.4).

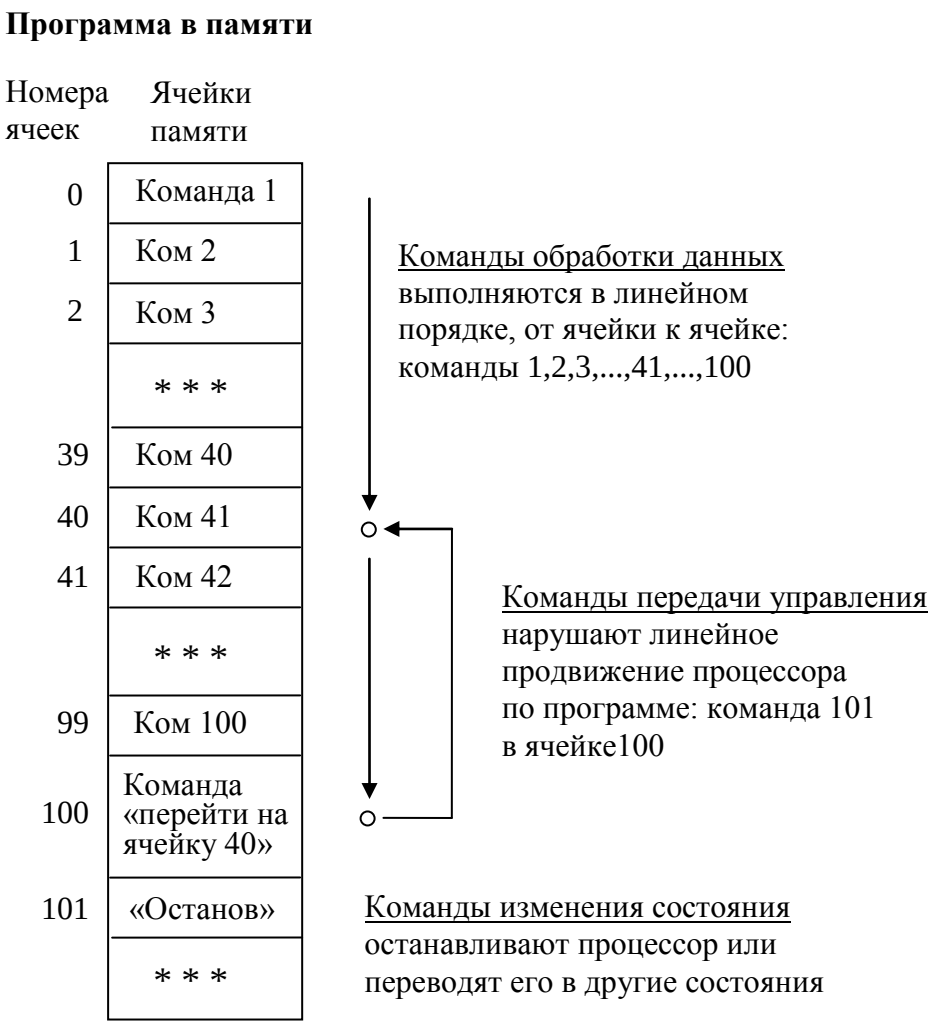


Рис. 1.4. Три категории команд процессора

Кроме команд обработки данных и передачи управления у любого процессора имеется третья категория команд, служащих для *изменения его состояния*. Одной из них является команда останова hlt (от англ. halt – остановка, прекращение) (см. рис. 1.5). По ее приказу процессор прекращает выполнение программы и переходит в состояние останова. В состоянии останова никакие команды (операции над данными) не выполняются. Процессор перестает продвигаться по программе, «замирает» в этом состоянии (в точке останова) и сохраняет содержимое всех своих регистров неизменным, каким оно было перед остановом. Это состояние длится до тех пор, пока процессор не будет выведен из останова. Вывести процессор из состояния «Останов» можно повторным запуском (путем подачи сигнала сброса/пуска RESET, Сбр), а также (для большинства процессоров) сигналом прерывания INTR, ЗПр (Interruption, Запрос прерывания). Под действием этих сигналов процессор снова возвращается в состояние «Счет программы» и либо заново стартует с пусковой ячейки памяти (по сигна-

лу Сбр), либо переходит на подпрограмму обработки прерывания (по сигналу ЗПр) (гл.4, 5). («гл.» – это «глава»; здесь: главы 4 и 5.) Перезапустить (сигналом Сбр) или прервать (сигналом ЗПр) процессор можно и в других точках программы, а не только в момент останова.

Говоря коротко, процессор является автоматическим устройством, которое сразу после пуска начинает выбирать из памяти и выполнять команды программы одну за другой, по очереди, в порядке возрастания их адресов; или по адресам переходов. Процессор беспрерывно повторяет однотипные действия: он *выбирает* и *выполняет* команды программы. Он не «умеет» делать, по существу, ничего иного, кроме *выборки* и *выполнения* команд. Это – автоматическое, циклически функционирующее устройство, предназначенное для *выборки* и *выполнения* команд.

Одновременно мы установили, что любой процессор располагает **тремя категориями команд**:

1. Командами *обработки данных*: пересылки (mov – от англ. «move») и ввода-вывода данных (in, out – от «input», «output»), арифметических (add, sub – «subtract» и т.д.), логических (and, or и т.д.), сдвиговых (sal – «Shift Arithmetic operand Left», sar – «…Right», rol – «Rotate operand Left», ror – «…Right»и т.д.) и др. операций;
2. Командами *передачи управления*: простого перехода (jmp – от «jump»), перехода по условию, или ветвления (j*c* – «jump on condition»), вызова (call) и возврата (ret – «return») из подпрограмм и др.;
3. Командами *изменения состояния* процессора, например командой hlt, переводящей процессор в состояние останова HALT. Сигналы прерывания INTR (Interruption) и сброса RESET выводят процессор из состояния HALT.

Здесь и далее во всех примерах используются машинные команды процессоров семейства Intel 80x86. Они приведены в Приложении Б. С системой этих команд, а также с языком ассемблера этого семейства, можно ознакомиться также в литературе [1, 2, 8, 9, 18].

Автоматическое программируемое процессирование. Мы выяснили, что процессор – это та часть компьютера, которая «знает», как складывать и вычитать числа, как проводить простые логические операции, но которая делает это только по заданию программы, хранимой в оперативной памяти ЭВМ. Отдельные команды и реализующие их операции выполняются процессором самостоятельно, с помощью собственной *аппаратуры,* без поясняющих указаний со стороны других устройств, а порядок следования этих операций устанавливается хранимой в памяти ЭВМ *программой*, процессор «умеет» выполнять отдельные команды (операции), а программа диктует ему порядок их выполнения. Благодаря подобному разделению функций между аппаратурой и программой процессор генерирует под управлением *программы* последовательность простых, *аппаратно* реализуемых операций – *процесс*, т.е. осуществляет *автоматическое программируемое процессирование*. Поэтому *процессор* – это *устройство, генерирующее процесс под управлением программы,* это – *ав-*

томатическое программно процессирующее устройство. «Процессор можно назвать «мозгом» компьютера, его двигателем или, если хотите, сердцем этой удивительной машины» [16].

Процессирование и программирование. Процессор выполняет действия, обратные алгоритмизации и программированию. При программировании решаемая задачи разбивается на простые задания, понятные машине, которые она может выполнить с помощью известных ей простых операций. При процессировании, наоборот, эти простые операции складываются процессором в целостную задачу. Если алгоритмизация и программирование – это *микрорасчленение* задачи на машинные операции, то процессирование – это, наоборот, *сочленение* элементарных машинных операций в законченное решение.

Таким образом, все многообразие функций, предоставляемых компьютером пользователю, достигается благодаря двум основополагающим факторам: способности процессора к *автоматическому программируемому процессированию* и искусству программиста описать этот процесс в терминах машинных операций путем *алгоритмизации и программирования* исходной задачи. В настоящей книге мы обстоятельно познакомимся с *организацией* автоматического программируемого процессирования и подробно изучим основу для написания программ – *архитектуру* процессоров и ЭВМ.

Состав процессора. Создано и выпускается множество модификаций микропроцессоров, выполненных по полупроводниковой технологии БИС и отличающихся совокупностью аппаратных средств и программного обеспечения. При рассмотрении общих принципов построения и работы микропроцессоров целесообразно не касаться особенностей конкретного изделия, а использовать понятие гипотетического (иллюстративного) прибора, имеющего минимальный состав оборудования и простой набор команд.

Любой микропроцессор (ЦП) содержит две главные функциональные части, или устройства (рис. 1.5):

1) Управляющее устройство (УУ) и
2) Арифметико-логическое, или операционное, устройство (АЛУ, ОУ).

К ним добавляется, кроме того,

3) Устройство сопряжения процессора с системной магистралью ЭВМ, или интерфейсное устройство (ИУ).

Все эти устройства соединены между собой посредством

4) Внутренней магистрали (или магистралей) процессора.

Для обмена информацией с внешними устройствами – памятью и периферией служит

5) Внешняя магистраль процессора, представленная набором выводов его микросхемы, подлежащих соединению с системной магистралью ЭВМ.

Структуру отдельных частей типичного микропроцессора иллюстрирует рис. 1.6. **Внутренние устройства** процессора включают:

1) УУ – *управляющее устройство*:

БУ и С – *блок управления и синхронизации*, вырабатывающий сигналы для синхронной работы отдельных устройств,

ДшК – *дешифратор команд*,
набор *управляющих регистров*:
СчК (PC, IP) – счетчик команд (программный счетчик, указатель инструкций) (например IP/EIP процессора Intel 80x86), указывающий на следующую команду для выборки из памяти,
РгК – регистр (или регистры) команд, размещающий в себе код текущей команды, и др.,
другие блоки;

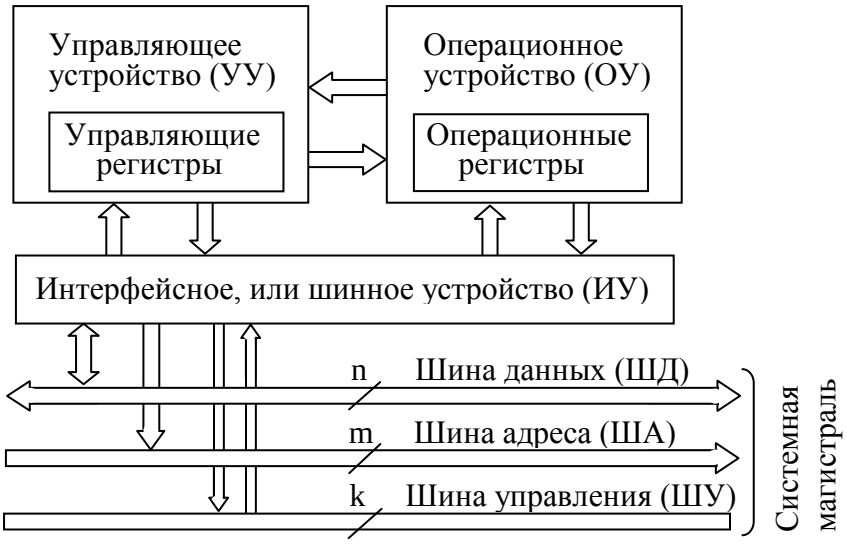


Рис. 1.5. Обобщенная структура процессора.
n, m, k – разрядности шин данных, адреса, управления

2) ОУ – *операционное устройство*:
АЛУ, или АЛБ – *арифметико-логическое устройство*, или *блок*, выполняющий арифметические и логические операции над данными, и
набор *операционных регистров*:
общих, таких как:
РгАк – регистр-аккумулятор (EAX для Intel 80x86),
РОН – регистры общего назначения (EBX, ECX и др.),
и с*пециальных*:
УС (SP) – указатель стека (ESP для Intel 80x86),
РгСП (Flags) – регистр состояния процессора, или регистр флагов,
другие, например сегментные регистры CS, SS, DS, … процессора Intel 80x86,
а также *вспомогательных, буферных регистров данных* (БРгД), предназначенных для временного хранения операндов и скрытых от программиста;
3) ИУ – *интерфейсное устройство* (устройство шинного интерфейса), образованное *буферными регистрами* (БРг), *усилителями-формирователями* шин данных, адресов и управления и др. компонентами;
4) *внутреннюю магистраль или магистрали*.

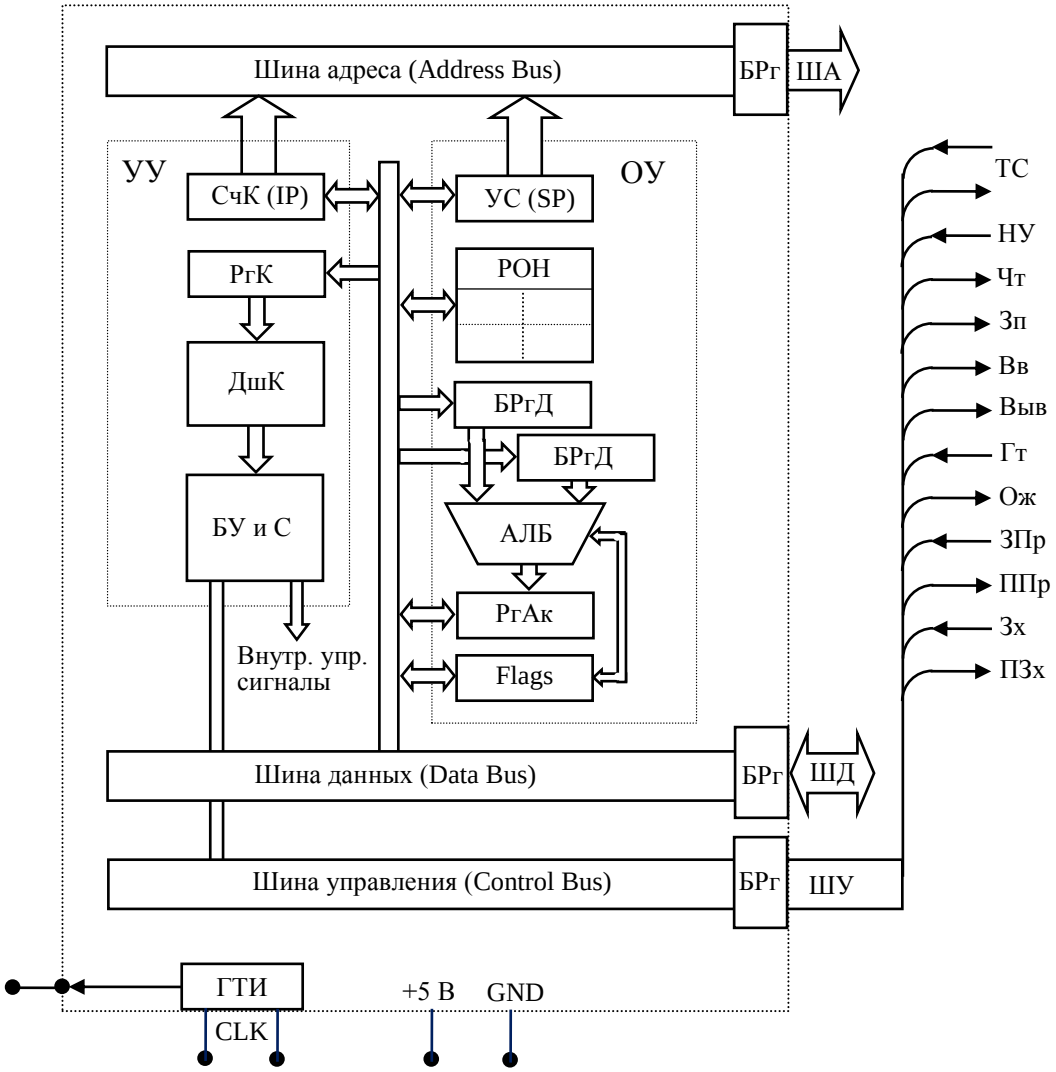


Рис. 1.6. Типовая структура микропроцессора

Внешние сигналы процессора. По внешней шине данных (ШД) процессор считывает коды команд из памяти, а также выдает и принимает коды данных из памяти и периферийных устройств, а по внешней шине адресов (ША) назначает ячейки памяти или регистры ПУ, в которых эти команды и данные хранятся (рис. 1.5, 1.6). Кроме того, процессор формирует и реагирует на различные управляющие сигналы:

Чт (чтение) и Зп (запись) – выходные сигналы процессора, служащие для обращения к памяти;

Вв (ввод) и Выв (вывод) – выходные сигналы для обращения к устройствам ввода-вывода (периферийным устройствам);

ЗПр (запрос прерывания) и ППр (подтверждение прерывания) – входной и выходной сигналы для организации прерываний;

Зх (захват) и ПЗх (подтверждение захвата) – входной и выходной сигналы для организации работы многопроцессорных систем,

Гт (готовность) и Ож (ожидание) – входной и выходной сигналы для синхронизации обмена данными процессора с медленно действующими внешними устройствами;

ТС – тактовые сигналы: входные от кварцевого генератора и выходные на внешние устройства, сообщающие устройствам, например, о начале цикла обмена данными по шине – шинного цикла;

НУ – сигнал начальной установки (сброса RESET).

Назначение внешних сигналов подробно разбирается в п. 3.1, 3.3, 3.4.

Функции устройств. *Управляющее устройство* выбирает команды из памяти, расшифровывает их, узнавая, что требует от него очередная команда, и посылает всем другим устройствам процессора и компьютера сигналы, включающие их на выполнение затребованных операций. Далее эти устройства выполняют заданные операции уже самостоятельно, без участия УУ.

УУ следит также за правильной очередностью выполнения команд и извлекает их из памяти в порядке этой очередности.

Для временного хранения команд и связанных с ними данных устройство управления может использовать набор регистров. Один из этих регистров называется *регистром команд* и служит для размещения кода выполняемой команды, выбираемой из основной памяти. Команда, находящаяся в данный момент в регистре команд, сообщает устройству управления следующую информацию [15]:

а) *адрес ячейки памяти или порта ввода-вывода*, в котором находятся данные;

б) *место*, в которое эти данные должны быть перемещены;

в) *действие*, которое должно быть выполнено.

Так работают команды обработки данных. Команды передачи управления (извлеченные из памяти и помещенные в регистр команд) сообщают устройству управления *адрес следующей команды*, которую он должен выбрать из памяти и выполнить вслед за текущей командой. Этот адрес мы назвали выше *адресом перехода*. Применительно к области памяти, представленной на рис. 1.4, команда передачи управления, хранящаяся в ячейке 100, сообщает процессору адрес перехода 40, и процессор выбирает и выполняет вслед за ней не команду из следующей ячейки 101, а команду из ячейки 40.

После выполнения команды другой регистр – *счетчик команд*, имеющийся в составе УУ, выдает адрес [15]. Этот адрес обеспечивает выборку из основной памяти (и исполнение) следующей команды. Другие названия счетчика команд: *программный счетчик, указатель инструкций* (Instruction Pointer, IP). Порядок функционирования счетчика команд описывается в п. 4.4, 4.5.

Таким образом, основными функциями УУ являются:

а) слежение за правильной очередностью выполнения команд и извлечение (выборка) их из памяти в порядке этой очередности;

б) дешифрация выбранных команд и посылка всем другим устройствам процессора и ЭВМ сигналов, включающих их на выполнение заданных командами операций.

Для осуществления указанных функций УУ содержит такие блоки:

- программный счетчик;
- регистр команд;
- дешифратор команд;
- блок управления выполнением команд; и др.

Аппаратное и микропрограммное управление выполнением команд. Выполнение машинных команд сводится к элементарным преобразованиям информации: передаче информации из одного регистра в другой, взятию обратного кода, сдвигу и т.д. Элементарные преобразования, неразложимые на более простые и выполняемые в течение одного такта сигналов синхронизации, называются *микрооперациями*. В *аппаратных, схемных блоках управления* для каждой операции, определяемой кодом операции машинной команды, конструируется свой набор логических схем, выполненных на диодах, транзисторах и т.д. и определяющих, какой управляющий импульс и в каком такте должен быть возбужден. Такие наборы схем и осуществляют управление микрооперациями, отвечающими разным командам. Реализация одной команды в этом случае достигается за счет однажды соединенных между собой логических схем, и логику управления выполнением команд можно изменить только путем переделок этих соединений. Поэтому ЭВМ с аппаратным устройством управления называют ЭВМ с жесткой логикой управления. Это понятие означает практическую невозможность каких-либо изменений в системе команд ЭВМ после ее изготовления.

Микропрограммное управление заключается в том, что команды выполняются путем запуска блоком управления так называемых *микропрограмм*, хранящихся в специальной, управляющей, памяти самого процессора (в ПЗУ микропрограмм управляющего устройства) и состоящих из еще более примитивных, чем команды, приказов – *микрокоманд*. Каждая микрокоманда реализует одну микрооперацию, а микропрограмма в целом – микрооперации, отвечающие одной машинной команде. В этом случае каждый потребитель или разработчик процессора может сконструировать из микрокоманд нужный набор операций (команд) в данной конкретной задаче. Замена набора команд достигается тогда заменой информации в управляющей памяти УУ без каких-либо переделок в аппаратуре. Однако в этом случае программисту необходимо знать все тонкости работы инженера – разработчика компьютера. Поэтому микропрограммные компьютеры считаются трудными для пользователя.

Операционное (*арифметико-логическое*) *устройство* можно рассматривать как устройство, помогающее устройству управления выполнять команды [15]. Оно может выполнять *арифметические* операции (например, сложение и вычитание). Кроме того, ОУ способно выполнять *логические* операции, т.е. обрабатывать коды чисел с помощью логических операций И, ИЛИ, сложение по модулю два и др. В распоряжении ОУ имеется также вспомогательная память в виде накапливающего регистра, принимающего результат операции, который принято называть *аккумулятором*, и других рабочих регистров.

Когда команда, находящаяся в данный момент в регистре команд устройства управления, требует выполнения некоторой арифметической или логической операции над данными, эти данные извлекаются из основной памяти и посылаются в ОУ. Затем ОУ по указанию устройства управления производит требуемую операцию. После завершения операции результат (новые данные) запоминаются в аккумуляторе. Таким образом, ОУ выполняет все арифметические и логические операции над данными, поступающими из памяти компьютера или из периферийных устройств.

Данные (числа, буквы и другие символы внешнего алфавита ЭВМ, коды пикселей на экране монитора, прочие данные в двоичном представлении), над

которыми производится операция, называются *операндами*. Для работы с ними ОУ имеет в своем составе несколько рабочих регистров и функциональных схем. *Регистры* ОУ предназначены для хранения операндов или их адресов в процессе выполнения операций, и в них заносятся результаты операций. Поэтому их принято называть *операционными* или *операндными* (или, как мы уже говорили, *рабочими*) регистрами. Те операционные регистры, которые предназначены для хранения собственно операндов, называют *регистрами данных*, а для хранения адресов операндов в памяти или в периферийных устройствах – *регистрами адреса*. У многих процессоров рабочие регистры универсальны: способны хранить и операнды, и их адреса. Тогда их называют *регистрами общего назначения* (РОН), *общими регистрами*.

Например, микропроцессор Intel Pentium имеет восемь рабочих 32-битных регистров общего назначения, способных хранить как операнды и результаты операций, так и их адреса. Наоборот, рабочие регистры микропроцессора MC68000 компании Motorola делятся на две группы: восемь 32-битных регистров данных и семь 32-битных регистров адреса.

Мы упомянули (в сноске на стр.19), что регистр – это устройство, хранящее некоторую порцию информации. Для лучшего понимания поясним более точно, что *регистром* (как, впрочем, и ячейкой памяти) называется электронное устройство, предназначенное для запоминания двоичного числа. Регистр представляет собой упорядоченный набор («линейку») триггеров (рис.1.7), обычно D.

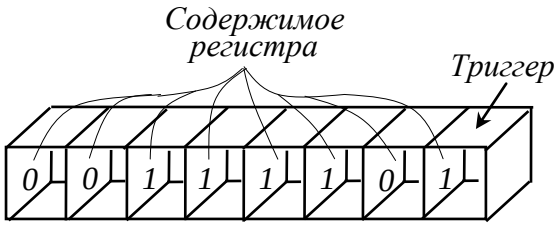


Рис. 1.7. Регистр – это линейка триггеров

Триггер (англ. trigger – курок) – электронное устройство, обладающее двумя устойчивыми состояниями и способное скачком переходить из одного состояния в другое под воздействием внешнего импульса. Каждому состоянию триггера соответствует определённый (высокий или низкий) уровень выходного электрического напряжения:

1) триггер установлен в единичное состояние – уровень «1» (обычно – напряжение от 2 до 5 В).

2) триггер сброшен в нуль – уровень «0» на выходе (напряжение от 0 до 1 В).

Установившееся состояние сохраняется сколь угодно долго и может быть изменено внешним импульсом или отключением напряжения питания. Таким образом, триггер является элементарным компонентом памяти, способным хранить наименьшую единицу информацию – один двоичный разряд: «0» или «1».

Триггер строится обычно на двух транзисторных логических элементах И-НЕ или ИЛИ-НЕ, охватывающих друг друга положительной обратной связью и за счет этого взаимно поддерживающих свое состояние.

Функциональные схемы («вентили», от нем. Ventil – клапан) вычисляют различные функции от двоичных операндов, представленных двузначными сигналами – логическими нулями («0») и единицами («1») [25]. Логический «0» представляется обычно сигналом от 0 до 1 В, а логическая «1» – сигналом от 2 до 5 В. Напряжение за пределами указанных величин недопустимо. С помощью вентилей производятся все необходимые преобразования операндов при пере-

даче их из одного операционного регистра на другой. Одним из вентилей процессора является двоичный сумматор, осуществляющий сложение двоичных чисел. Простейшие вентили (логические элементы) – это схемы И, ИЛИ, НЕ, ПОВТОРЕНИЕ.

К числу основных, аппаратно реализуемых операций по обработке данных, выполняемых ОУ по указке машинных команд, входят [1, 2, 8, 9, 12]:

а) пересылка (прием и выдача) операнда (слова или байта*) на регистр. Это могут быть внутренние, межрегистровые пересылки, происходящие между регистрами самого ОУ или процессора; и внешние пересылки, осуществляемые между регистрами ОУ, с одной стороны, и ячейками памяти или регистрами периферийных устройств, с другой;

б) сдвиг операнда на заданное число разрядов (битов) влево или вправо;

в) прибавление к слову или вычитание из него единицы (в более общем случае, некоторой const). Эта операция называется инкрементом и декрементом слова соответственно;

г) сравнение операндов по принципу «больше-меньше-равно»;

д) поразрядные логические операции «или» – or (дизъюнкция), «и» – and (конъюнкция) «исключающее или» – xor (exclusive or, сложение по модулю 2), «не» – not (побитное инвертирование);

е) суммирование и вычитание двух операндов, представляющих числа в одной и той же системе счисления; другие бинарные арифметические действия: умножение, деление и т.д.

ж) преобразование кодов операндов, включая дополнение, дешифрацию и др.

* *Байт* понимается традиционно как единица информации (операнд или команда) длиной в 8 двоичных разрядов (бит). *Слово* состоит из целого числа байт, в частном случае – из одного байта. Более подробное обсуждение этих понятий вы найдете в п.3.3, 4.2.

Часть регистров УУ (управляющих регистров) и ОУ (операционных регистров) доступны программисту, т.е. их содержимое можно изменять при помощи машинных команд. К программно-доступным регистрам относятся *программный счетчик* (управляющий регистр), *регистры общего назначения* и *аккумулятор* (операционные регистры) и др. Содержимое программного счетчика изменяется при помощи команд передачи управления (например, jmp), а РОН – при помощи команд обработки данных mov, add и др. (Однако, нельзя: mov IP,1000).

Количество программно-доступных (адресуемых, «видимых») регистров процессора сравнительно невелико. Оно варьируется от нескольких единиц (у простейших микропроцессоров) до нескольких десятков и более (у высокоорганизованных микропроцессоров и процессоров супер-ЭВМ).

Кроме ОУ и УУ в структуре процессора можно выделить третью, вспомогательную функциональную часть – устройство сопряжения (интерфейса) процессора с внешними по отношению к нему устройствами – памятью и периферийными устройствами – *интерфейсное устройство* (ИУ). ИУ осуществляет электрическое (по уровню и по мощности) и логическое (по составу и момен-

там выработки) согласование собственных сигналов процессора (микропроцессорной БИС) и сигналов, которые надо подавать на память и периферийные устройства, – сигналов системной магистрали (шины). Поэтому ИУ называют по-другому *шинным устройством, устройством шинного интерфейса* (ШИ) или *формирователем системной магистрали* (ФСМ) или *системным контроллером*, поскольку оно управляет системной шиной, формируя стандартные сигналы на системной шине ЭВМ, отвечающие нормативным требованиям. Сам же процессор (МП БИС) зачастую не вырабатывает всей номенклатуры нормативных системных сигналов ввиду ограничения на число выводов микросхемы. В связи с этим интерфейсное устройство выполняется обычно в виде отдельного, внешнего по отношению к МП БИС модуля (например, в виде так называемого «северного моста» – системного контроллера, п.1.4, 3.6), помогающего процессору формировать нужные сигналы.

В заключение отметим, что при всей сложности внутреннего построения процессор, с точки зрения пользователя, представляет собой функционально законченный прибор с раздельными шинами адреса и данных, набором внешних управляющих сигналов и группой собственных программно-доступных регистров (рис.1.8).

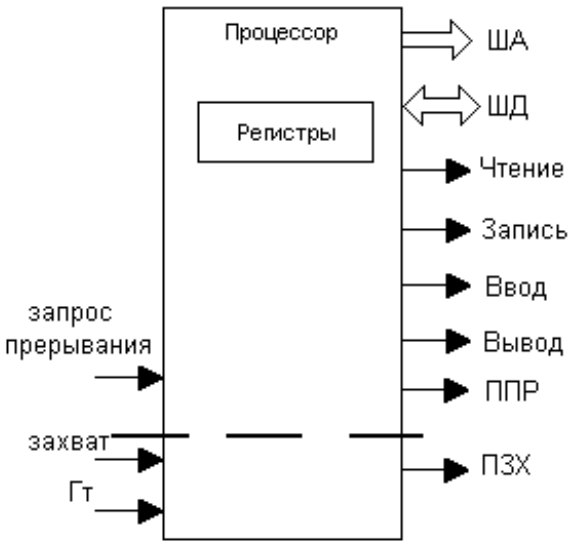


Рис. 1.8. Микропроцессор с точки зрения пользователя

Быстродействие и производительность процессора и компьютера. Термины «быстродействие» и «производительность» процессора или компьютера часто используют как синонимы, но им придают также и нормативно различающиеся значения.

Быстродействие процессора (оно же – *быстродействие ЭВМ*) – это число команд (или, как часто говорят, «операций»), выполняемых процессором за единицу времени (число операций в секунду, или *IPS* – от англ. instructions per second). Поскольку разные команды («операции») выполняются с разной скоростью, то говорят о *среднем быстродействии процессора* (*ЭВМ*) на разных классах задач, которое определяется с учетом вероятностей использования каждой операции при решении данного класса задач. В частности, для любой данной программы среднее быстродействие процессора есть просто отношение количества команд в программе к времени ее выполнения. Сейчас стало общепринятым характеризовать процессор по числу выполняемых им *MIPS* (Million of Instructions Per Second – «миллион команд в секунду».). В последние годы мощность вычислительных машин попадает в диапазон *BIPS* , или *GIPS* (Billion, Giga – миллиард команд в секунду).

Быстродействие процессора (не задерживаемого, в идеале, медленно функционирующими внешними устройствами и системной шиной) пропорционально его тактовой частоте. *Тактовая частота* (частота ядра, Internal Clock) – это количество электрических импульсов в секунду, подаваемых на процессор внешним генератором тактовых импульсов – одним из блоков, расположенных на материнской плате. Тактовая частота выдерживается с очень высокой точностью и лежит в мега или гигагерцовом диапазоне. Процессор, работающий на тактовой частоте 800 МГц, выполняет 800 миллионов рабочих тактов в секунду. В зависимости от сложности обрабатываемой команды процессору для ее выполнения иногда необходимы десятки и более тактов. Самые простые команды (например, команды межрегистровых пересылок данных внутри процессора) выполняются обычно за один такт. Быстродействие процессора в этом случае – *предельное, пиковое быстродействие* – совпадает с тактовой частотой. Чем выше тактовая частота ядра, тем выше скорость обработки данных. Современные микропроцессоры работают на частотах от 800 МГц до 4,7 ГГц. Соответственно их предельное быстродействие составляет от сотен до тысяч Mips. Микроконтроллеры (у них все или почти все команды обрабатываются, как правило, за 1 такт) обладают быстродействием порядка десятков Mips.

Развитые процессоры построены так, что даже сложные, многотактные, команды практически не снижают скорости их работы, и при обработке потока команд эта скорость зачастую доходит до пиковых значений – сотен миллионов операций в секунду и более (~ 10^3 Mips). Это обеспечивается за счет реализации *общего принципа достижения высокого быстродействия*, который звучит так*: компьютер* (*процессор*) *должен запускать как можно больше команд в секунду* [25]. С этой целью в структуру процессора вводятся средства распараллеливания (одновременного выполнения нескольких) операций, такие как буфер опережающей выборки команд, один или несколько конвейеров дешифрации и выполнения команд (п. 2.2). Также увеличивается ширина (разрядность) шины данных, что позволяет считывать из памяти по несколько команд сразу за один такт, помещая их в буфер опережающей выборки команд, и обмениваться данными с внешним окружением тоже большими порциями. Кроме того, используется кэш-память – быстродействующая память, расположенная между процессором и основной памятью (п. 2.2). Благодаря этому процессор способен запускать достаточно много команд в секунду. После считывания из памяти поток команд поступает на конвейер внутри процессора, и если конвейер справляется с этим потоком раньше, чем его надо обновлять, то собственно время обработки команд (в смысле времени на выполнение заданных ими операций) как бы выпадает из общего времени обработки потока. В результате быстродействие начинает определяться лишь скоростью считывания команд из памяти, например, из кэша, и ограничивается только частотой, а также шириной системной шины или шины кэша.

Частота системной шины (System Clock). Даже если процессор располагает развитыми механизмами распараллеливания операций и кэш-памятью, а также достаточно широкой шиной данных, его быстродействие может сущест-

венно ограничиваться частотой системной шины, которая всегда меньше частоты работы ядра процессора. Системная шина служит для связи процессора с остальными устройствами, и медленные пересылки информации по ней задерживают работу ядра. Частота системной шины современных микропроцессоров от 66 МГц до 1 - 1.5 ГГц. Если умножить ее на среднее число команд, приходящихся на одну выборку из памяти за один такт шины, то это, как правило, и будет некое суммарное *быстродействие* системы с развитым процессором. Таким образом, процессор имеет две частоты: тактовую частоту ядра и частоту системной шины, и системная шина является наиболее «узким» местом компьютера.

Умножение частоты. Соотношение частот ядра и шины не является произвольным. В современных процессорах используется умножение тактовой частоты: собственная частота процессора (частота ядра) определяется путем умножения частоты системной шины на определенный коэффициент – коэффициент умножения, который может быть равен 1.5, 2.0, 2.5, 3.0, 4.0 и доходит до 20 и более.

Учитывая зависимость работы процессора (и вообще любого устройства ЭВМ) от взаимодействия с другими компонентами, величину быстродействия уместно иногда относить к отдельным устройствам: процессору, памяти, устройствам ввода-вывода и т.д., и говорить о собственном, внутреннем быстродействии этих устройств. В этом случае понятие быстродействия не распространяется на всю ЭВМ, а вместо быстродействия используется понятие производительности ЭВМ [7].

Производительность ЭВМ, в отличие от быстродействия, определяется в общем случае как объем работ, осуществляемых ЭВМ в единицу времени. Производительность можно измерять, например, средним числом некоторых стандартных задач, решаемых за единицу времени, или средним числом отрабатываемых команд в секунду на этих классах задач (т.е. выше упомянутым средним быстродействием ЭВМ). Однако чаще производительность измеряется временем выполнения ЭВМ некоторых образцовых программ – типичных программ, которые могли бы выполняться на данной установке. Производительность зависит от быстродействия процессора и устройств ввода-вывода, частоты и ширины системной шины, порядка прохождения задач в ЭВМ и т.п. Она увеличивается при распараллеливании обработки потока команд: совмещении операций чтения-записи с обработкой, использовании мультипрограммного и мультипроцессорного режимов работы. Таким образом, единственной подходящей и надежной единицей измерения производительности является время выполнения реальных программ, и в первую очередь, стандартных тестовых программ.

Генератор тактовых импульсов (ГТИ) (также *синтезатор частот, тактовый генератор,* сленг. «клокер» – от англ. «clocker», clock generator, clock synthesizer) – устройство, формирующее основные тактовые частоты, используемые на материнской плате и в процессоре. Источником опорной, чрезвычайно стабильной частоты для него служит кварцевый резонатор («кварц»). Типичные частоты подобных устройств составляют от единиц до десятков МГц.

Тактовый генератор в современных персональных компьютерах состоит из опорного кварцевого генератора и микросхемы генерации. Опорная частота кварцевого генератора подается на вход микросхемы, которая на своих выходах формирует из опорного сигнала тактовые сигналы для различных компонентов системы: для процессора, для системной шины, для чипсета, для оперативной памяти и т. д. (рис. 1.9). Микросхема генерации умножает и делит частоту опорных импульсов, то есть выступает в качестве делителя и умножителя частоты. Современные микросхемы генерации являются программируемыми. Внутри отдельных компонентов (включая процессор) частоты от ГТИ могут еще дополнительно умножаться или делиться местными схемами синхронизации.

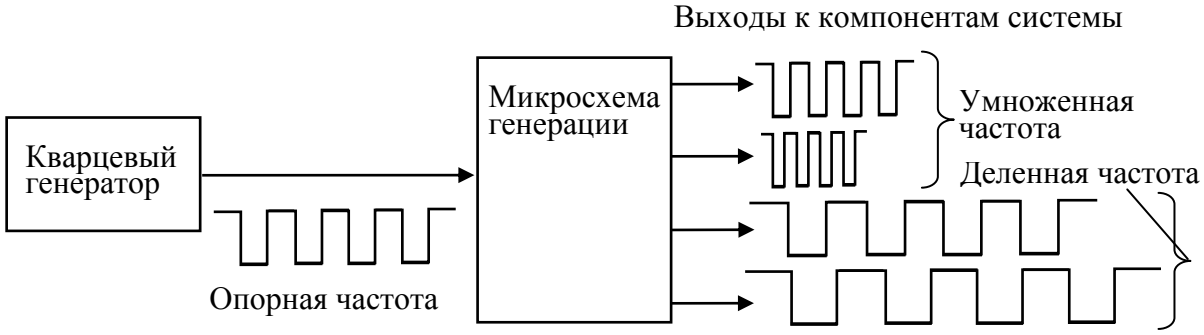


Рис. 1.9. Выработка тактовых сигналов в компьютере

1.3. Память

Назначение памяти и ее виды. Чтобы иметь возможность выполнять обработку данных по заданной программе, ЭВМ должна иметь память, в которой хранятся команды и данные [15]. Память вычислительной системы состоит из следующих частей: регистров процессора и периферийных устройств, основной памяти (часть которой может быть продублирована кэш-памятью) и внешней памяти (рис. 1.10).

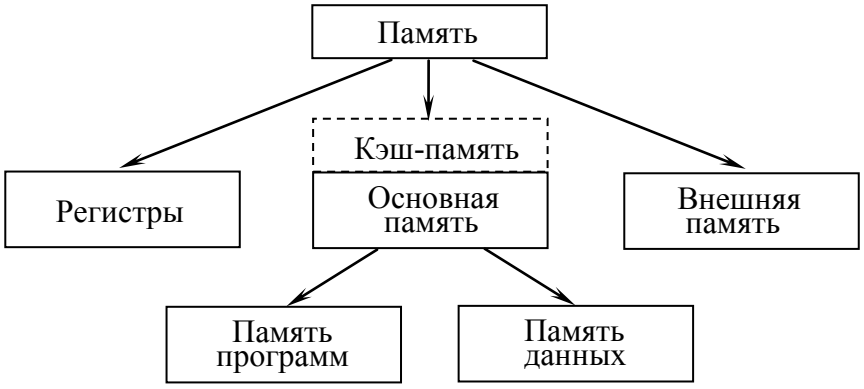


Рис. 1.10. Виды памяти ЭВМ

Многоуровневая организация памяти. Указанные части памяти образуют многоуровневую иерархическую структуру. По мере продвижения по уровням сверху вниз (на рис. 1.10 – слева направо) – от регистров процессора до внешней памяти меняются три параметра: время доступа, объем и стоимость

единицы хранимой информации. Время доступа и стоимость уменьшаются, а объем увеличивается [24, 25].

На самом верху иерархии находятся **регистры процессора**. Доступ к ним осуществляется быстрее всего, поскольку они работают на тактовых частотах процессора, время доступа составляет десятые доли наносекунд. Поэтому **регистровую память процессора** называют сверхоперативной памятью ЭВМ (СОП). Если в понятие СОП включать, как это обычно делается, только программно-доступные регистры, то объем таких регистров – десятки машинных слов. Регистры процессора используются для промежуточного, кратковременного хранения данных в процессе их обработки. Это энергозависимый вид памяти, т.е. при отключении питания от процессора информация в регистрах стирается.

Регистры имеются не только в процессоре, но и в периферийных устройствах. Это – регистры, или порты, ввода-вывода (п. 1.4). Свойства **периферийных регистров** сопоставимы со свойствами основной памяти**. Регистры**, процессорные и периферийные, представляют собой вспомогательную память, распределенную по всей системе.

Дальше в иерархической структуре идет **кэш-память**. Так называют память небольшого объема с высокой скоростью работы, «прозрачную» для программиста (невидимую программой). Кэш-память играет роль буфера между основной памятью и процессором. Основная идея кэш-памяти проста: в ней находятся слова, которые чаще всего используются. Если процессору нужно какое-нибудь слово: команда или операнд, сначала он обращается к кэш-памяти. Только в том случае, если слова там нет, он обращается к основной памяти. Если значительная часть слов находится в кэше, среднее время доступа значительно сокращается. Непосредственное время доступа составляет единицы наносекунд. Объем кэша составляет от 32 Кбайт до десятков мегабайт*. Отличается относительно высокой стоимостью и высоким энергопотреблением. Энергозависимый вид памяти. Более подробное описание кэш-памяти вы найдете в п. 2.2.

* Для измерения больших величин в информатике приняты следующие единицы:
1 К = 2^{10} = 1024 – двоичная тысяча (кило);
1 М = 2^{20} – двоичный миллион (мега);
1 Г = 2^{30} – двоичный миллиард (гига);
1 Т = 2^{40} – двоичный триллион (тера).

Основная память (ОП) предназначена для хранения текущих выполняющихся программ и необходимых им данных – операндов. Та часть ОП, в которой хранятся команды, называются *памятью программ*, а та часть, в которой хранятся данные, – *памятью данных*. ОП занимает среднее положение в иерархии памяти. Время доступа к ней составляет десятки наносекунд, а объем в настоящее время лежит в диапазоне от 1 Гбайт до сотен гигабайт. ОП более дешевая, чем кэш-память, так как реализуется с помощью динамических мик-

росхем (п. 4.2). Обладает высокой надежностью хранения данных и программ, расчетная вероятность ошибки – одна в десять лет.

Основная память реализуется на микросхемах ОЗУ и ПЗУ. ОЗУ – это оперативное запоминающее устройство, или RAM – Random Access Memory – память с произвольным доступом, как для чтения, так и для записи. ОЗУ – энергозависимый вид памяти: после выключения питания хранимая в нем информация искажается либо полностью разрушается. ПЗУ – постоянное запоминающие устройства, или ROM – Read Only Memory – память только для чтения. ПЗУ – энергонезависимый вид памяти, содержащаяся в нем информация сохраняется постоянно как при включенном, так и выключенном напряжении питания. Служит, например, для хранения BIOS. В последнее время широко используются другие виды энергонезависимой перезаписываемой памяти, например, флэш-память, или флэш-карты [24]. (Имя «флэш» было придумано разработчиком этого вида памяти потому, что процесс стирания содержимого памяти напомнило ему фотовспышку, от англ. flash – вспышка, мгновение).

Внутренняя память. Это понятие используют иногда для обозначения памяти внутренней части ЭВМ. К ней относят основную память, память процессора – набор его адресуемых рабочих регистров (сверхоперативная память ЭВМ – СОП), а также память контроллеров периферийных устройств – набор регистров (портов) ввода-вывода. Совокупность внутренней и внешней памяти ЭВМ образует всю память ЭВМ.

Внешняя память – магнитные диски (жесткие, или так называемые «винчестеры», и гибкие) и оптические (компакт-) диски, флэш-карты, накопители на магнитных лентах и проволоках и др. Внешняя память служит информационным складом, местом для долгосрочного хранения данных. Она самая дешевая и самая емкая – объем современных жестких дисков доходит до десятков терабайт. Также велик объем накопителей на лентах и проволоках. У флэш-карт, оптических и гибких магнитных дисков объем меньше. Однако внешняя память – это самый медленный уровень памяти. Доступ к дискам происходит, по крайней мере, в 10 раз медленнее для жестких, твердотельных дисков и в сотни раз медленнее для съемных накопителей (оптических дисков и флэш-карт), чем доступ к основной памяти. К тому же съемные накопители нужно еще взять и поместить в соответствующее устройство. Быстродействие накопителей на гибких дисках, магнитных лентах и проволоках еще меньше.

Состав основной памяти. Основная память состоит из большого числа идентичных технических устройств – блоков одинакового размера, служащих для запоминания двоичных чисел в некоторой физической форме (рис. 1.11) [15]. Эти запоминающие блоки называются *ячейками памяти*.

Ячейка памяти состоит из *элементов памяти*.

В большинстве ЭВМ ячейка состоит из 8 элементов. В каждом элементе может храниться наименьшая единица информации – одно двоичное одноразрядное число 0 или 1. Таким образом, в каждой ячейке содержится 8 двоичных разрядов. Поскольку эти 8 бит составляют 1 байт (п. 3.3, 4.2), говорят, что длина ячейки ЭВМ равна 1 байту.

Совокупность нулей и единиц, находящихся в восьми элементах памяти, представляет собой *содержимое* ячейки памяти.

Физически в качестве элемента памяти (простейшего запоминающего устройства) применяется триггер или микроскопический электроемкостной элемент («конденсатор») (п. 4.2). Высокий уровень напряжения на выходе триггера (обычно от 2 до 5 В) трактуется как логическая «1», низкий (от 0 до 1 В) – как логический «0». Аналогично заряженность «конденсатора» означает «1», разряженность – «0». Упорядоченная совокупность («линейка») подобных триггеров или «конденсаторов» и образует базовый запоминающий блок – ячейку памяти.

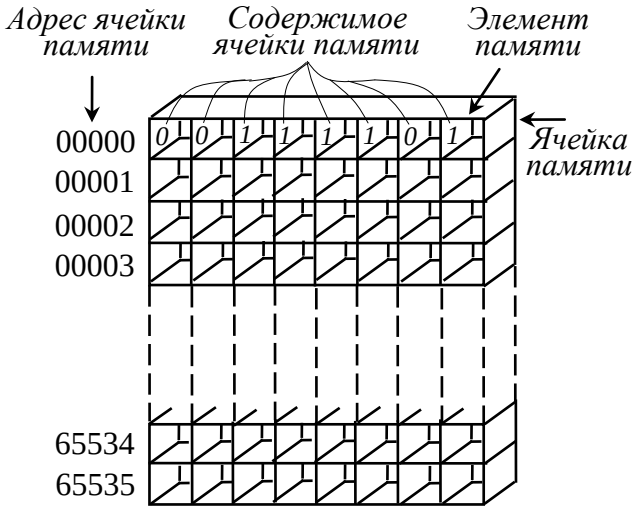


Рис. 1.11. Ячейки памяти

Каждая ячейка памяти имеет свой, уникальный *номер*, или *адрес*, известный программисту. Используя этот адрес, программист с помощью машинных команд может назначать ячейки и заносить или извлекать из них числа (операнды); или организовывать переходы по программе на новые команды, указывая номера их ячеек в командах перехода (адреса перехода). Адреса ячеек нужны и самому процессору, когда он автоматически выбирает следующую для исполнения команду после завершения предыдущей, выставляя ее адрес на память (п. 4.4, 4.5).

Максимальное количество адресов составляет 2^m, где m – разрядность адреса (его размер в битах), и адреса ячеек пробегают последовательный ряд значений 0, 1, 2, …, $2^m - 1$. Этим числом ограничивается размер основной памяти. Поскольку бóльшие значения адресов недопустимы, большее число ячеек памяти использовать нельзя. В Intel-совместимых компьютерах $m = 32$, и $2^m = 4\,294\,967\,296$, или 4 Гбайт информации (4 двоичных миллиарда байт).

Структура функционально законченной основной памяти показана на рис.1.12. В нее входят:

а) набор запоминающих блоков – *ячеек памяти*, каждая из которых имеет свой уникальный *номер*, или *адрес*, известный программисту;

б) схемы, управляющие выборкой (селекцией) ячеек – *селекторы*, или *дешифраторы, адреса*;

в) управляющие схемы, активизирующие выбранную ячейку на выполнение нужной операции: чтение или запись (согласно сигналам Чт или Зп от процессора) – *схемы управления чтением/записью*.

Процессор (а в режиме прямого доступа к памяти – периферийное устройство) выполняет с основной памятью две операции:

1) чтение и
2) запись.

При чтении процессор извлекает одну порцию данных (байт или слово) из выбранной им, адресованной ячейки. При записи – заносит данное в ячейку. В промежутках между этими операциями память хранит записанную в нее информацию. Операции чтения и записи в память называются *обращениями к памяти*.

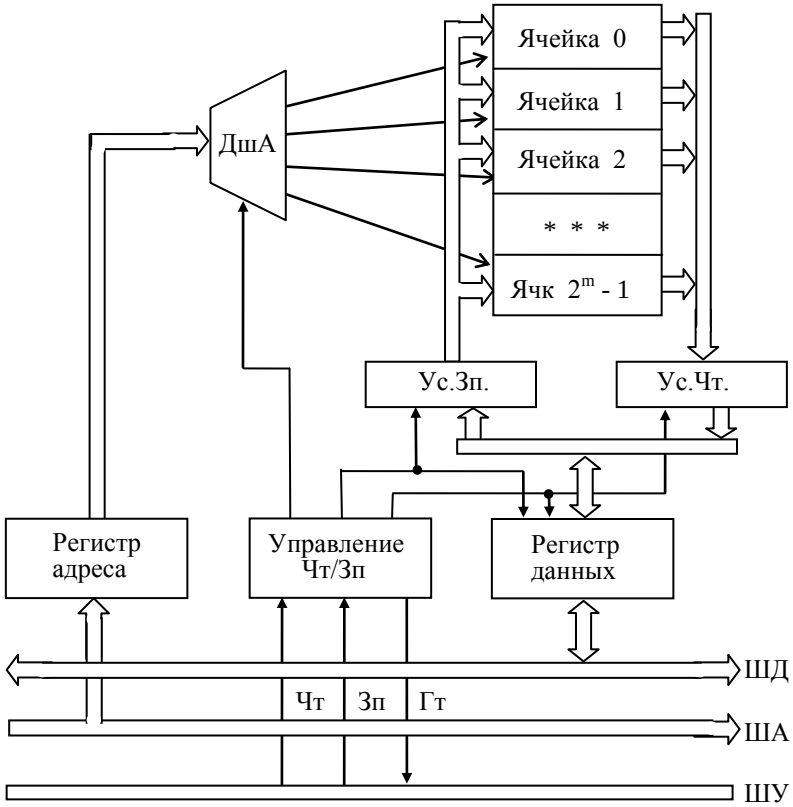


Рис. 1.12. Структура основной памяти.
Чт (MEMR – Memory Read) и Зп (MEMW – Memory Write) – сигналы управления чтением и записью; Гт (READY) – сигнал готовности запоминающего устройства; ДшА – дешифратор адреса; Ус.Чт и Ус.Зп – усилители чтения и записи

Процессоры Intel обращаются к памяти, например, по команде mov:

mov AL,CS:[15] ; AL←*m*(CS:15) – чтение 15-й однобайтной ячейки кодового сегмента памяти CS (Code Segment) в регистр процессора AL (описание регистров процессора содержится в п.2.3, а сегментов памяти – в п.2.4);

mov DS:[16], AX ; *m*(DS:16)←AX – запись в 16-ю двухбайтную ячейку сегмента данных памяти DS (Data Segment) из регистра AX. Здесь два байта из AX записываются в две однобайтные ячейки с адресами 16 и 17: байт из регистра AL – в ячейку 16, байт из регистра AH – в 17.

Память – пассивное устройство, она никогда не выступает инициатором проведения каких-либо операций в компьютере.

1.4. Контроллеры и порты ввода-вывода

Назначение контроллеров ввода-вывода. *Контроллеры ввода-вывода* (КВВ), или *контроллеры периферийных устройств*, – это устройства, предназначенные для передачи данных между центральной частью ЭВМ и периферийными устройствами и далее – внешней средой, они образуют каналы для такой передачи. Контроллеры также управляют периферийными устройствами во время этой передачи (отсюда их название «контроллеры», т.е. устройства управления).

Ввод и вывод. При указанной передаче данные могут вводиться или выводиться из ЭВМ. Подчеркнем, что операции ввода и вывода понимаются всегда по отношению к внутренней (центральной) части ЭВМ. Иначе, *вводом* называется передача данных из внешней части ЭВМ (включая передачу из внешней среды) во внутреннюю часть – в процессор или память, а *выводом* называется обратная передача – из внутренней части (из процессора или основной памяти) во внешнюю часть и далее, возможно, во внешнюю среду (рис. 1.13). Коротко: *ввод – это передача данных из периферии в ядро ЭВМ, а вывод – обратная передача: из ядра в периферию*.

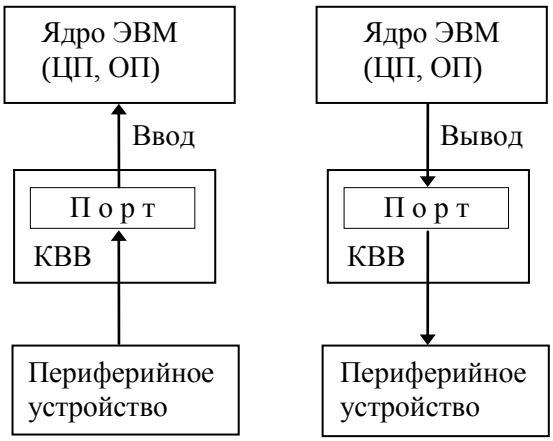


Рис. 1.13. Потоки данных
при вводе и выводе.
ЦП – центральный процессор;
ОП – основная память;
КВВ – контроллер в/в

Регистры (порты) ввода-вывода. В этих операциях данные передаются не напрямую между ядром и периферией по электрическим проводникам контроллеров, а через встроенные в контроллеры особые промежуточные регистры. Эти регистры называются *регистрами* или *портами ввода-вывода* (или *входными* и *выходными регистрами* или *портами*) (см. рис. 1.13). Каждый контроллер имеет в своем составе от нескольких единиц до нескольких десятков портов в/в. Порт представляет собой регистр (устройство памяти, линейку триггеров) и образует отдельный канал для передачи данных между ядром и периферией компьютера. Порты временно хранят вводимую и выводимую информацию, служат буфером для передаваемой информации, разделяют внутреннюю часть ЭВМ и ее периферию.

Необходимость в портах (регистрах) в/в связана с тем, что процессор и периферийное устройство одновременно не всегда оказываются готовы к обмену данными друг с другом. Так, процессор не способен в произвольный момент времени принять данное от ПУ (при вводе), так как в этот момент он может быть занят выполнением других операций. Например, записью числа в память

по системной магистрали. При этом и сама системная магистраль, по которой ПУ собирается ввести данное в процессор, тоже занята. Аналогично и ПУ тоже не может в любой момент времени принять данное от процессора (при выводе) из-за того, например, что оно еще не успело обработать предыдущее данное. Поэтому при осуществлении ввода-вывода требуется *синхронизация* работы процессора и периферийного устройства – выполнение операций обмена данными между ними должно быть обеспечено в строго определенные для каждого из них моменты времени. Эта задача и решается с помощью портов в/в.

Порты функционируют подобно почтовым ящикам для передачи корреспонденции. Они кратковременно хранят вводимые и выводимые данные. При *вводе* периферийное устройство помещает код данного (слово) в порт ввода независимо от того, готов процессор принять это слово в данный момент времени или нет и свободна ли системная шина от других передач данных. Одновременно ПУ сообщает процессору (тем или иным способом), что слово помещено в порт. Для этого ПУ может оставить сообщение в другом порте ввода, установив, например, определенный его бит в 1; или просигналить процессору напрямую специальным сигналом – запросом прерывания. В свою очередь, процессор принимает отправленное ему данное не сразу в тот самый момент, когда оно поступило в порт, а спустя какое-то время, после того как он и системная шина освободятся от выполнения текущей операции.

Аналогично при *выводе* процессор просто заносит порцию данных в порт вывода, не заботясь о том, чем в данный момент занято периферийное устройство и готово ли оно принять данное. (Предварительно процессор, конечно, удостоверяется, что порт свободен, т.е. предыдущее данное из него выбрано. Сообщение о выборке данного может быть оставлено периферийным устройством в другом порте путем перевода его специального бита в 0.) Далее процессор продолжает текущую работу. Со своей стороны ПУ принимает выведенное данное не одновременно с его поступлением в порт, а лишь при очередном просмотре порта. Просмотры могут повторяться с некоторой заранее обусловленной частотой или по мере поступления очередных данных.

Таким образом, порты ввода и вывода позволяют согласовать во времени работу процессора и периферийных устройств, несмотря на то, что эти органы компьютера обособлены друг от друга, функционируют каждый сам по себе, независимо друг от друга, с разными скоростями и возбуждаются разными событиями.

«Портами» в широком смысле называют также вообще «входы» в компьютер, через которые к системной шине подключаются периферийные аппараты. Такой «порт», «вход» (называемый также *интерфейсным, соединительным портом*) представлен разъемом на каком-либо элементе аппаратного обеспечения компьютера, в который включается кабель или вилка периферийного аппарата. Далее, внутри компьютера этот разъем сопрягается с контроллером в/в. Таким образом, интерфейсный порт – это функционально законченное устройство, представляющее собой контроллер в/в в совокупности с соединительным разъемом. Иногда разъем отдельно тоже именуется портом – входом в компьютер.

Различают несколько типов соединительных, интерфейсных портов: внутренний (таймерный), клавиатурный, коммуникационный, игровой (джойстик). Каждому порту выделяется группа адресов (это адреса находящихся в составе контроллера регистров в/в), по ко-

торым в порт (в регистры) записываются или из порта (регистров) считываются данные, служебная информация для программирования параметров порта и текущее состояние порта.

Через таймерный порт можно программировать частоту следования меток времени, используемых в электронных часах компьютера. Сигналы порта подсчитываются операционной системой, для хранения накопленного значения используется область памяти с определенным адресом.

Клавиатурный порт обеспечивает ввод кодов нажатых клавиш. Данные от порта накапливаются и обрабатываются в специальном кольцевом буфере клавиатуры в определенной области памяти.

Коммуникационные порты обеспечивают подключение таких внешних устройств как мышь, принтер, сканер, внешний модем и некоторых других. Эти порты подразделяются на последовательные (COM1 и COM2) и параллельные (LPT). Широкое распространение получил последовательный порт USB. Он обеспечивает высокоскоростное подключение к компьютеру сразу нескольких периферийных устройств: съемных флэш-карт, принтера, сканера, цифровых камер и т.п..

Коммуникационные порты используются также для межкомпьютерной связи в режиме Link.

Модуль портов ввода-вывода. Все порты (регистры) ввода-вывода доступны программисту, адресуемы, «видимы». Каждому порту (как и ячейке памяти) присвоен свой индивидуальный номер (адрес), известный программисту. Поэтому программист может вывести через порт любую информацию в периферийное устройство и извлечь из любого порта имеющееся в нем сообщение от периферийного устройства.

Для ввода и вывода в системе команд ряда процессоров служат специальные машинные команды. МП Intel имеет для этой цели команды ввода in (input) и вывода out (output). Например:

in AX, port ; AX←*p*(port) – ввести в регистр процессора AX 16-битное число из порта ввода с номером port;

out DX, AL ; *p*(DX)←AL – вывести в порт вывода, адрес которого находится в регистре DX процессора, содержимое однобайтного регистра процессора AL.

Поскольку адрес порта МП Intel 16-битный, максимальное количество портов ввода и вывода составляет $2^{16} = 65\,536 = 64$ К (по 65 536 тех и других). Особенностью архитектуры Intel является раздельная адресация памяти и периферийных устройств: порты и ячейки относятся к разным адресным пространствам и могут иметь одинаковые адреса (так называемая шина Multibus, «Мультишина»).

Иные процессоры (например, серии VAX, типа Alpha, МК PIC16F) не имеют специальных команд ввода-вывода, а используют для этой цели обычные команды пересылки данных с памятью. В этом случае порты в/в адресуются как ячейки памяти и под них отводится некоторая часть общего адресного пространства памяти (шина Unibus, «Общая шина»).

Порты образуют отдельный функциональный блок контроллеров ввода-вывода, отвечающий за буферирование потоков вводимых и выводимых данных. Его принято называть *модулем портов в/в* или просто «модулем ввода-вывода» (МВВ).

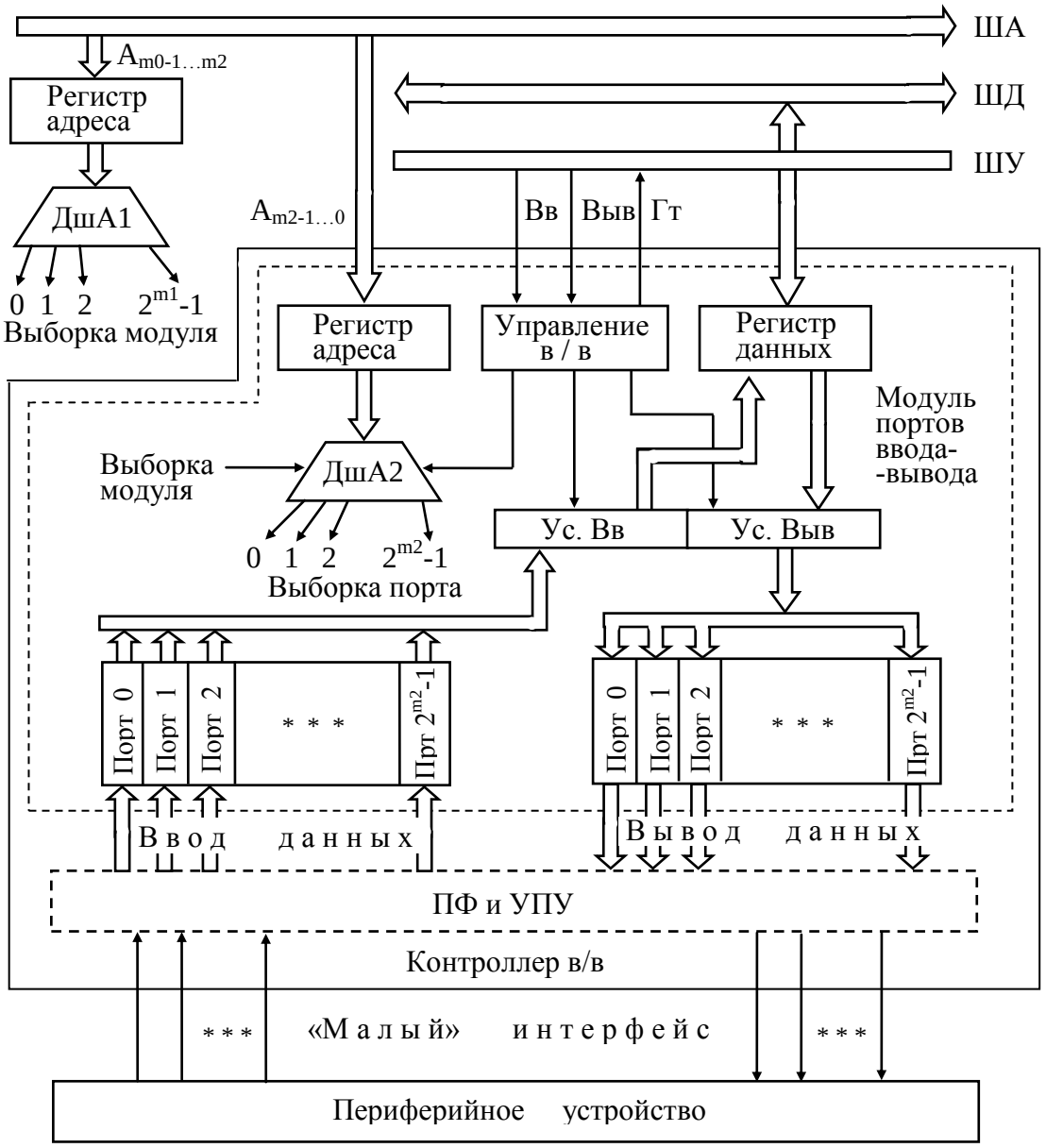


Рис. 1.14. Структура модуля портов ввода-вывода.
Вв (IOR) и Выв (IOW) – сигналы управления вводом-выводом; Гт – сигнал готовности периферийного устройства; Ус.Вв – усилитель-формирователь ввода; Ус.Выв – усилитель-формирователь вывода; ПФ – преобразователь форматов данных; УПУ – блок непосредственного управления периферийным устройством; $A_{m0-1…0}$ – адрес порта, состоящий из номера модуля в/в $A_{m0-1…m2}$ (старшие разряды адреса) и номера порта внутри модуля $A_{m2-1…0}$ (младшие разряды адреса); m0=m1+m2 – разрядность адреса порта

Полный состав МВВ (части контроллера в/в) представлен на рис.1.14. В него входят:

а) *порты ввода-вывода* – адресуемые регистры периферийного устройства;

б) *дешифратор* (*селектор*) их *адресов*;

в) *управляющие схемы*, открывающие порты по вводу и выводу,

и процессор выполняет с модулями (портами) ввода-вывода две операции:

1) ввод данных (из порта в процессор);

2) вывод данных (из процессора в порт).

Эти операции называются *обращениями к портам* периферийных устройств.

По своему составу модули портов в/в напоминают память ЭВМ, но, в отличие от памяти, хранящей текущие программы и обрабатываемые в них данные, МВВ осуществляют буферирование (временное хранение) потоков передаваемой информации в операциях ввода и вывода.

Виды портов ввода-вывода. В процессе ввода-вывода передается информация двух видов: *управляющие данные* (управляющие слова) и *собственно данные* (данные-сообщения, данные-операнды). Соответственно порты, служащие для передачи управляющих слов, называются *портами управления* (или управления/состояния), а для передачи данных – *портами данных*.

Управляющие слова от процессора называются также *командными словами*. Они инициируют действия, не содержащие непосредственно передачу данных, например, запуск устройства, запрещение прерываний, перемещение каретки принтера и т.п.

Управляющие слова от периферийного устройства называются *статусными словами, словами состояния*. Они содержат информацию об определенных признаках, например, о готовности устройства к обмену данными, о наличии ошибок при обмене и т.п. Состояние обычно представляется в декодированной форме – один бит для каждого признака.

Наиболее общая программная модель модуля в/в содержит *четыре порта*: два для ввода данных (входной порт данных) и параметров состояния (порт состояния), и два для вывода данных (выходной порт данных) и параметров управления (порт управления) (рис. 1.15). В зависимости от особенностей устройства общая модель конкретизируется, например, отдельные порты состояния и управления объединяются в один порт, в устройстве ввода (вывода) имеется только порт вводимых (выводимых) данных, для ввода и вывода операндов используется только один двунаправленный порт.

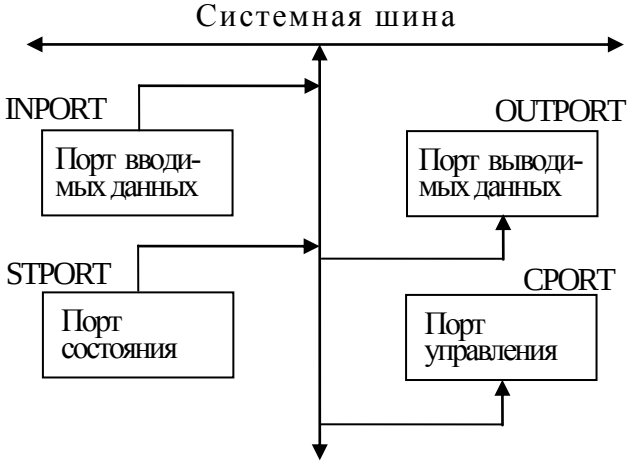
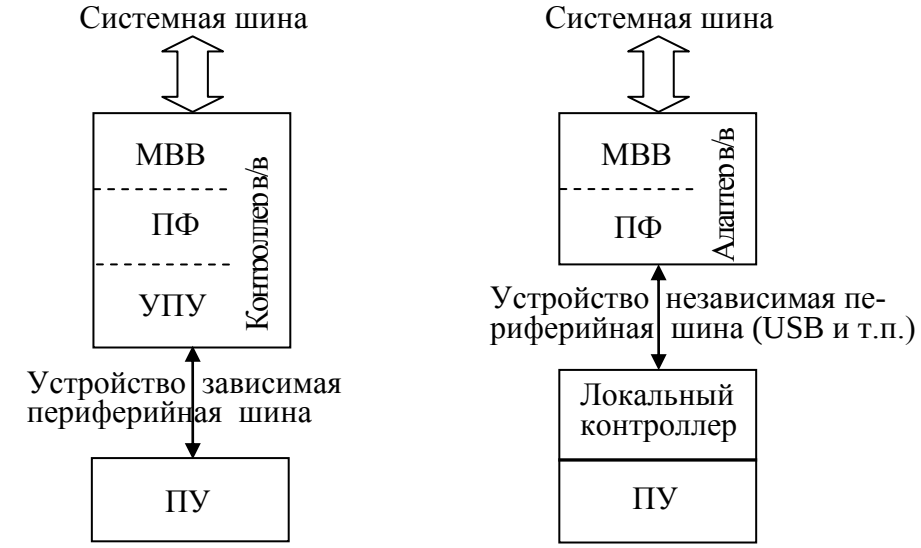


Рис. 1.15. Общая программная модель периферийного устройства

Функциональный состав контроллеров ввода-вывода. Контроллеры в/в не только образуют буферные каналы для передачи вводимых и выводимых данных. Они еще управляют работой внешних устройств во время этой передачи: накопителей на магнитных дисках, дисплеев, принтеров, основной памяти в режиме прямого доступа и др.

КВВ выполняет следующие функции:

1) передает вводимые и выводимые из компьютера данные. При этом он осуществляет временное хранение (буферирование) данных на своих внутренних регистрах (регистрах, портах в/в) с целью синхронизации передачи. Порты в/в совместно с адресными и управляющими схемами составляют первый функциональный блок контроллера, который мы назвали *модулем портов в/в* (МВВ; рис. 1.16);
2) в процессе передачи преобразует данные в нужные форматы – из форматов внешних сигналов периферийных устройств в форматы сигналов системной магистрали, понятные процессору, – при вводе и обратно – при выводе. Совокупность компонентов, осуществляющих подобную функцию, образует второй функциональный блок КВВ – *преобразователь форматов* (ПФ на рис. 1.16);
3) управляет функционированием периферийных устройств, посылая на них нужные сигналы, активизирующие устройства на выполнение целевых операций. Этим занимается третий функциональный блок КВВ, носящий имя *блока непосредственного управления периферийным устройством* (УПУ) или *локального контроллера* (см. рис. 1.16).



а) Совместное расположение блоков контроллера

б) Удаленное расположение блока УПУ: адаптер + локальный контроллер

Рис. 1.16. Структура контроллера в/в.
МВВ – модуль портов в/в; ПФ – преобразование форматов данных;
УПУ – блок управления ПУ (локальный контроллер);
ПУ – периферийное устройство

Адаптеры ввода-вывода и локальные контроллеры ПУ. Контроллер в/в может быть выполнен в виде единого устройства, содержащего все три упомянутых функциональных блока (рис. 1.16, *а*); или он бывает конструктивно разделен на два относительно обособленных функциональных устройства (рис. 1.16, *б*):

а) Устройство сопряжения стандартного интерфейса вычислительной системы с интерфейсом ПУ (функция буферирования и преобразования), которое называется *адаптером ввода-вывода* или *контроллером шины в/в*;
б) Локальное устройство управления конечным оборудованием ПУ (собственно контроллер ПУ – УПУ, *локальный контроллер*, функция управления).

Адаптер является средством сопряжения какого-либо устройства с какой-либо шиной или интерфейсом вычислительной системы. Задача адаптера ввода-вывода – преобразование специфического интерфейса устройства ввода-вывода в стандартный интерфейс шины персонального компьютера (ПК). Это означает, что адаптеры выполняют функции 1-2 и располагаются обычно на плате ЦП (материнской, или системной, плате). Таким образом, в ПК могут быть установлены адаптер дисплея, адаптер принтера (например, адаптер USB-шины – «USB- контроллер») и так далее для каждого устройства.

А любая электроника, расположенная на самом периферийном аппарате, по-прежнему называется контроллером. Контроллер служит тем же целям сопряжения, но при этом подразумевается его некоторая активность – способность к самостоятельным действиям после получения команд от обслуживающей его программы. Поэтому контроллеры выполняют функции 1-3; либо только функцию 3, и тогда располагаются на самом периферийном устройстве.

В современных персональных ЭВМ, построенных на базе Intel совместимых процессоров, контроллеры отдельных ПУ встроены в чипы так называемых северного (Northbridge) и южного (Southbridge) мостов.* Они именуются также системным и функциональным контроллерами соответственно. Северный мост обслуживает системную шину, оперативную память и некоторые другие компоненты, южный – работу с ПУ. В состав северного моста входят: контроллер системной шины FSB (формирователь системной магистрали; п.1.2), контроллеры оперативной памяти, графической шины AGP (видеопроцессора), внутренней шины DMI (для соединения северного моста с южным). В состав южного моста входят: контроллер шины PCI, USB-контроллер, контроллер дисковода (DMA-контроллер), контроллер Ethernet, часы реального времени и некоторые другие. (См. также п.3.6.)

* Северный и южный мосты получили свое название благодаря «географическому» расположению на материнской плате. Внешне мосты – это квадратной формы микрочипы, расположенные под процессором, но Северный – в верхней части материнской платы, а Южный – в нижней.

Процессоры ввода-вывода. В смысле данного выше определения к контроллерам можно отнести и периферийные процессоры. Однако мы все-таки будем различать термины контроллер и процессор ввода-вывода. Процессором ввода-вывода или периферийным процессором назовем только такое устройство, которое может самостоятельно выбирать команды из памяти, т.е. программируемое устройство, самостоятельно организующее вычислительный процесс. Примеры ПВВ: акселераторы 2D- и 3D-графики (графические процессоры); элементы внешней памяти; звуковая карта; сетевая карта. Процессоры ввода-вывода определяются также по критерию сложности, хотя эта граница и расплывчата. Таким образом, если устройство имеет программу, которая выполняется независимо от ЦП или является устройством с множеством сложных функций, то его можно отнести к ПВВ.

В отличие от этого контроллер ввода-вывода не способен самостоятельно избирать команды и для осуществления своих функций требует управления извне. Простым примером

служит контроллер последовательного, например USB, канала. Для того чтобы он начал передавать посылку по линии связи, обычно требуется отдать ему определенную команду (в большинстве случаев это запись передаваемых данных в порт контроллера). Далее, несмотря на то, что контроллер самостоятельно принимает посылки, поступающие по линии связи, выделяет из них данные и сохраняет в своем внутреннем буфере, дальнейшие действия по приему посылок он не способен выполнять, пока из его буфера не будут забраны принятые ранее данные. Режимы работы этого контроллера (скорость, количество стоп-бит, контроль четности и т.п.), опять-таки, задаются извне: путем записи в порты контроллера соответствующих значений. Именно в этом значении мы определили выше понятие КВВ.

Вывод данных на монитор. Видеоадаптер.

Как создается изображение на экране монитора. Изображение на экране монитора состоит из тысяч крошечных световых квадратиков («точек»), называемых *пикселями*, *видеопикселями*, *пэлами* или *элементами изображения* (англ. pixel – picture element; русск. пиксель или пиксел). Они образуются при поступлении электрического сигнала в заданную точку экрана. Сам экран подобен листу миллиметровки. Световые точки на нем выстроены в ровные ряды – *строки*. Совокупность точечных строк образует *графическую сетку*, или *растр* (двумерный массив точек), а само изображение, состоящее из точек, называется *растровым изображением* (рис. 1.17). Каждый элемент изображения (пиксель) может быть определен координатой в растре (на «листе миллиметровки»). Например, 19 позиция в строке (19 столбик), 3 строка: координата (19, 3). При построении изображения элемент либо высвечивается, либо находится в выключенном состоянии. Дополнительно каждому элементу могут задаваться определенный цвет и яркость свечения.

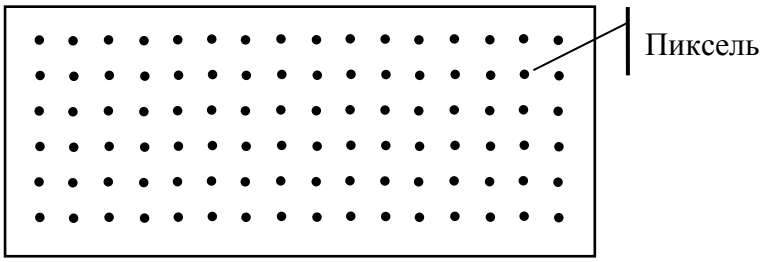


Рис. 1.17. Графическая сетка (растр) экрана

Качество изображения определяется разрешающей способностью. Разрешающая способность зависит от числа элементов изображения на экране монитора. Это число равняется *размеру графической сетки* – произведению числа точек в строке (числа столбцов) на число строк: M×N. На современных компьютерах используются такие размеры графической сетки: 800×600, 1024×768, 1152×864, 1280×1024, 1920×1200, 2560×1600.

Для создания и управления изображением на экране служит **видеоадаптер** (также **видеокарта, графический адаптер, графическая плата, графический ускоритель, 3D-карта**). Так называют компонент компьютера, к которому подключается монитор. Он представляет собой электронное устройство, преобразующее двоичный код изображения, находящийся в памяти компьютера (в видеопамяти), в видеосигнал для монитора.

Видеоадаптер обычно выполнен в виде отдельной платы, которая вставляется в один из слотов расширения* на материнской плате, а также может быть интегрирован прямо в материнскую плату – как в виде отдельного чипа, так и в качестве составляющей части северного моста.

* – щелевой (англ. *slot* означает «щель») разъем, обычно в компьютере, соединённый с системной шиной и предназначенный для установки дополнительных модулей (*карт расширения*), расширяющих конфигурацию устройства.

Состав видеоадаптера. Видеоадаптер состоит из двух главных частей: дисплейного процессора (или еще и графического процессора) и видеопамяти (рис.1.18). Кроме них имеются цифро-аналоговый преобразователь (ЦАП, RAMDAC – Random Access Memory Digital-to-Analog Converter), служащий для вывода информации, полученной от дисплейного или графического процессора, на аналоговый монитор; видео-ПЗУ – постоянное запоминающее устройство, в котором записаны наборы экранных шрифтов, отображаемых адаптером в текстовых и графических режимах, функции BIOS для работы с видеоадаптером, служебные таблицы; система охлаждения.

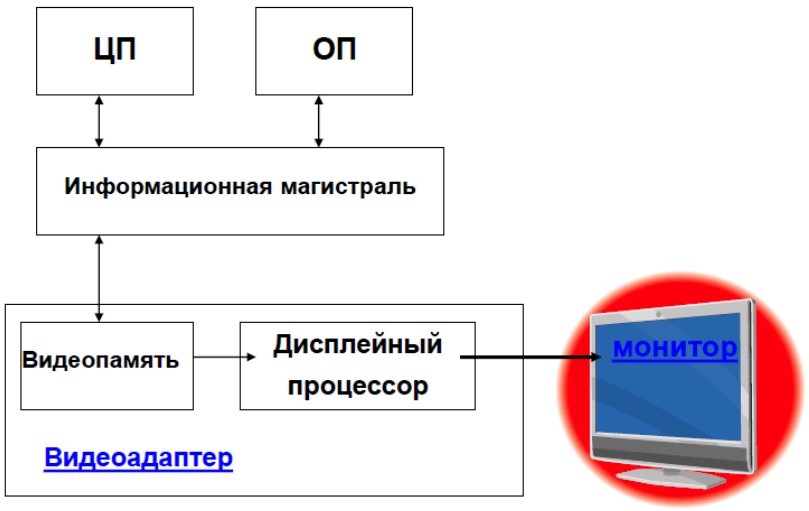


Рис. 1.18. Схема системы вывода изображения на экран

Ряд компонентов: дисплейный процессор, иногда – плата адаптера имеют собственные программно-доступные регистры, не входящие в состав видеопамяти. Они подразделяются на регистры управления/состояния и регистры данных (см. рис. 1.15). Регистры управления служат для задания различных аспектов отображения. Для этой цели используется система команд (командных слов). ЦП выводит команду в определенный регистр управления (например, в регистр, определяющий местоположение курсора), а адаптер исполняет ее, внося коррективы в изображение (перемещает курсор в нужную позицию на экране). Команды используются в следующих случаях: позиционирование и установка размера курсора, очистка и обновление экрана, изменение режима работы дисплея (текстовый - графический, 40 позиционный - 80 позиционный). С помощью регистров состояния ЦП получает информацию о состоянии адаптера

и результатах выполнения команд. Например, вводит координаты курсора. Через регистры данных передаются собственно параметры изображения.

Дисплейный процессор (также **видеоконтроллер**) – это специализированный процессор. Читает содержимое видеопамяти и в соответствии с ним и текущим режимом работы (текстовым или графическим) управляет функционированием дисплея, передавая на него (через ЦАП) электрические сигналы в последовательные точки экрана для высвечивания/гашения пикселей. Таким образом, к видеопамяти имеют доступ два процессора: центральный и дисплейный. Центральный процессор записывает информацию в видеопамять, а дисплейный – периодически читает ее и передает на монитор, на котором эта информация превращается в изображение. Передача информации на монитор производится с той частотой, с какой монитор выводит кадры изображения на экран (60 или 100 раз в секунду).

Графический процессор (Graphics Processing Unit – графическое процессорное устройство, GPU). Современные видеоадаптеры не ограничиваются простым выводом изображения, они имеют встроенный графический процессор, который может производить дополнительную обработку, снимая эту задачу с центрального процессора. Графический процессор занимается расчетом изображения, или, как говорят, выполняет графический рéндеринг (англ. rendering – «визуализация») – так называют процесс получения изображения по модели с помощью компьютерной программы, выбираемой из видеопамяти. На основе программы он рассчитывает местоположение всех точек, формирующих трехмерные (или двумерные) линии или формы, освобождая от этой обязанности центральный процессор, и заносит результаты расчета в видеопамять. Таким образом, изображение, хранящееся в видеопамяти, может генерироваться и постоянно изменяться не только центральным, но и графическим процессором. Графический процессор является основой графической платы, именно от него зависят быстродействие и возможности всего устройства. Современные графические процессоры по сложности мало чем уступают центральному процессору компьютера, и зачастую превосходят его как по числу транзисторов, так и по вычислительной мощности, благодаря большому числу универсальных вычислительных блоков.

Видеопамять – это собственная память монитора. Она расположена на плате видеоадаптера и предназначена для хранения информации об изображении, выводимой на экран. Видеопамять является электронным энергозависимым запоминающим устройством (ОЗУ). Равноценное ее название видео-ОЗУ, Video-RAM.

Видеопамять по отношению к GPU выполняет те же функции, что и оперативная память – по отношению к центральному процессору: она хранит весь «строительный материал» необходимый для создания изображения – текстуры (двумерные изображения поверхностей), геометрические данные, цвета, программы для определения параметров объекта и изображения и т.д. В текстовом режиме это – коды символов, выводимые в позиции строк, и др.; в графическом – двоичные данные о состоянии каждого пикселя экрана: высвечен/погашен,

яркость, цвет. В видеопамяти хранятся также промежуточные, невидимые на экране элементы изображения, представляющие «строительный материал» для создания изображения.

Важно отметить, что видеопамять разделяет с основной памятью компьютера общее пространство адресов, т.е. логически (для центрального процессора) она является частью адресного пространства компьютера и допускает считывание и запись информации путем обращения по адресу с помощью тех же команд, что используются для обращения к обычной памяти.

В базовом варианте распределения памяти для IBM/PC под память дисплея отводилось 128 Кбайт. В современных компьютерах объем видеопамяти составляет 1 Гбайт и более из общего адресного пространства компьютера. Последнее, например, для процессоров 32-битной архитектуры Intel (п. 2.2) имеет размер 4 Гбайт.

Доступ к видеопамяти организован через два порта (канала). Один из этих портов отводится для центрального процессора, другой – дисплейного и графического. Центральный процессор записывает информацию в видеопамять через свой порт, а дисплейный – периодически читает ее через свой порт и передает на монитор, на котором эта информация превращается в изображение. Через этот же порт к видеопамяти обращается (читает и записывает в нее информацию) графический процессор. Такой подход обеспечивает возможность быстрого доступа к общему полю памяти одновременно и со стороны процессора и со стороны дисплея и, следовательно, разрешает проблему конфликтов при доступе к ней. Обычная память такой способностью не обладает. Видеосхема обновляет экран 60 (или 100) раз в секунду, делая меняющееся изображение отчетливым и ясным.

Мониторы (видеопамять) персональных ЭВМ могут работать **в двух режимах: текстовом (алфавитно-цифровом) и графическом**, и передавать выводимую информацию в черно-белом или цветном изображении.

В текстовом режиме экран монитора разбивается на отдельные строки – обычно на 25 строк, имеющих по 80 или 40 позиций каждая в зависимости от варианта режима. В эти позиции выводятся только символы: буквы, цифры, знаки, псевдографические символы и т.п., хотя многие из этих символов можно использовать для построения простых линейных конструкций. Коды символов записываются в видеопамяти в заранее заданном формате. Коды имеют длину от двух байт и более. Например, младший байт может содержать ASCII код символа, старший – атрибуты: мигание символа или интенсивность фона (бит 7), цвет фона (6…4), интенсивность символа (3), цвет символа (6…4).

Графический режим используется главным образом для представления сложных изображений: схем, графиков, диаграмм, рисунков и т.п., хотя и допускает возможность представления символов различного размера и формы. В этом режиме в видеопамяти хранится информация о каждом пикселе изображения. Задавая программно параметры пикселей, можно управлять яркостью и цветом их свечения.

Можно сказать, что современная видеокарта – это, по сути, второй самостоятельный компьютер внутри персонального компьютера.

1.5. Системная магистраль

Системная (внутренняя, внутримашинная) магистраль (или магистрали) предназначена для электрического соединения функциональных органов ядра компьютера (процессора, памяти и контроллеров ввода-вывода) между собой и передачи информации между ними.

По системной магистрали процессор считывает команды из памяти, вводит данные из периферийных устройств, считывает операнды из памяти и выдает результаты счета на память и ПУ. По ней же он назначает устройство, с которым он будет производить обмен информацией в текущем машинном цикле, выставляя в магистраль адрес устройства – ячейки памяти или порта ввода-вывода. По системной магистрали пересылаются также управляющие сигналы и, кроме того, данные в режиме прямого доступа к памяти.

Физически системная магистраль представляет собой набор проводников, или линий, которые проложены на плате или в гибком шлейфном кабеле параллельно друг другу. По каждому проводнику передается один двоичный разряд – бит информации. Высокий уровень электрического потенциала в проводнике соответствует логической единице, а низкий – нулю.

Проводники магистрали разделяются на **три группы (подмагистрали, шины)**:

- шину данных (по которой передаются данные – команды и операнды);
- шину адресов (в нее процессор выставляет адреса ячеек и портов в/в, где данные хранятся);
- шину управления (для передачи сигналов, определяющих тип операции – чтение, запись, ввод, вывод, подтверждение прерывания и др.) (п.3.3).

Простые компьютеры имеют одну, общую для всех устройств системную магистраль, к которой подключаются все устройства центральной части компьютера. Развитые ЭВМ располагают в своей структуре несколькими (двумя-тремя) системными магистралями. Например, местной (локальной) – высокоскоростной, но «короткой» (для подключения небольшого числа быстродействующих устройств памяти и ввода-вывода) и общесистемной – низкоскоростной, но «длинной» (для большого числа устройств). Эти магистрали отделяются друг от друга буферными схемами – так называемыми мостами (bridge) (п.1.4, 3.6). Мосты регулируют прохождение данных из одной магистрали в другую. В суперЭВМ используется большое число различных внутримашинных магистралей, передача информации по которым происходит одновременно и независимо друг от друга.

Основные сигналы магистрали процессоров Intel рассмотрены в п.3.1, 3.2.

1.6. Периферийные устройства

Внешнюю (периферийную) часть (**периферийные устройства**) удобно подразделять на:

1. Устройства ввода-вывода данных для связи компьютера с пользователем (УСП): а) устройства ввода – клавиатура, переключатели, мышь, сканер, джойстик, микрофон и т.п.; б) устройства вывода – индикаторы, дисплеи, мониторы, принтеры, плоттеры, звуковые громкоговорители, или динамики, наушники и т.п.
2. Устройства периферийной (внешней, вторичной, массовой) памяти для хранения больших объемов информации: жесткие диски, флэш-карты, съемные диски, дискеты.
3. Устройства ввода-вывода для связи компьютера с различного рода внешними техническими объектами (УСО) – источниками сигналов (датчиками) и приемниками сигналов (исполнительными механизмами, линиями связи и т.д.). К УСО относятся: аналого-цифровые (АЦП) и цифро-аналоговые (ЦАП) преобразователи, сигнализаторы, реле, таймеры, счетчики событий, мультиплексоры (коммутаторы).
4. Средства телекоммуникации, реализующие передачу данных на большие расстояния и образующие, как правило, телекоммуникационную сеть (сеть связи, сеть передачи данных). Телекоммуникационными сетями являются телефонные, радио, телевизионные и компьютерные сети. Для построения компьютерных сетей применяются сетевые адаптеры, модемы (модуляторы-демодуляторы), кабели, телефонные линии, мультиплексоры, демультиплексоры. В качестве узлов связи в компьютерных сетях используются специализированные сетевые устройства: концентраторы, мосты, коммутаторы, маршрутизаторы (роутеры), шлюзы. [13, 24]

1.7. Интерфейс

Интерфейс ввода-вывода. С контроллерами в/в связано одно из важнейших понятий ЭВМ – понятие интерфейса ввода-вывода. Контроллеры в/в как раз и образуют физическую основу этого интерфейса (рис.1.19).

Интерфейсом ввода-вывода называется совокупность аппаратных и программных средств, служащих для сопряжения периферийных устройств (клавиатуры, мыши, монитора и т.п.) с центральной частью ЭВМ (процессором и памятью) и обеспечивающих обмен информацией между ними в операциях ввода-вывода. *Аппаратная часть* интерфейса включает в себя контроллеры ввода-вывода (п.1.4) и шины ввода-вывода, представленные набором электрических линий и передаваемых по ним сигналов. *Программная часть* интерфейса представляет собой совокупность программ управления вводом-выводом, осуществляющих ввод-вывод данных, управляющих слов и слов состояния ПУ, а также их анализ, дешифрацию и сохранение итоговой информации в памяти. Программы управления в/в подразделяются на программы идентификации типа

информации (данные, управляющие символы и т.д.), программы преобразования форматов, программы-драйверы («водители») ПУ для организации непосредственного доступа операционной системы к периферийному устройству с целью ввода-вывода информации, программы обработки запросов прерываний, супервизор ввода-вывода и др.

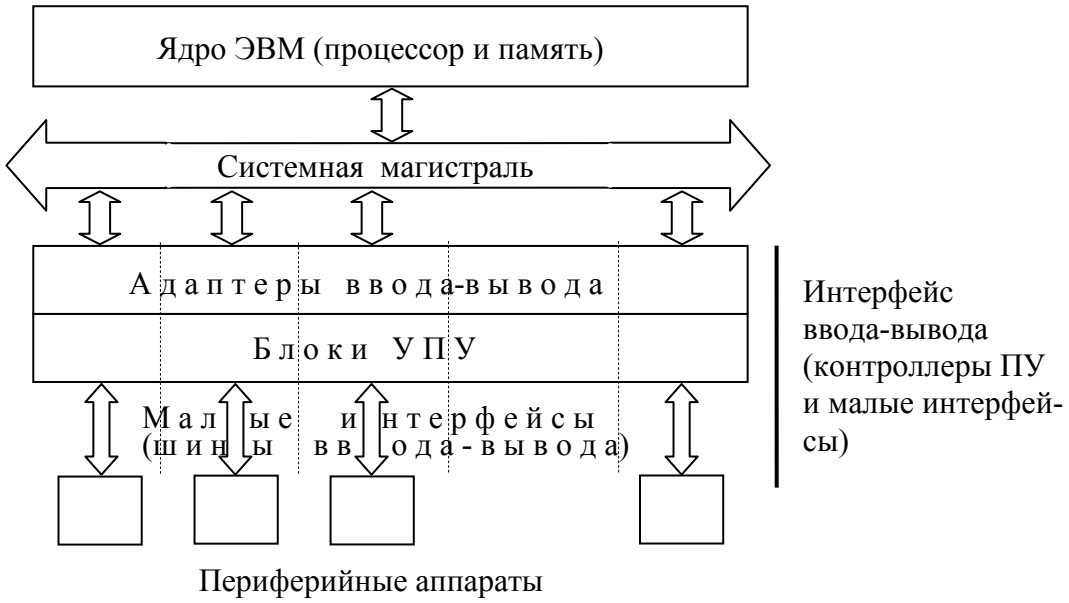


Рис. 1.19. Состав интерфейса ввода-вывода

Малые интерфейсы. Каждый тип периферийного устройства функционирует по своим особым физическим принципам, и нуждается в собственном интерфейсе ввода-вывода – контроллере (или адаптере) и шине в/в, соединяющей контроллер и ПУ. Подобный интерфейс (т.е. контроллер и шина в/в) предназначен для сопряжения ЭВМ с определенными типами периферийных аппаратов, и он называется *малым* или *внешним* или *периферийным*. Малые интерфейсы отличаются друг от друга принципами передачи данных. Они могут быть устройство зависимыми и устройство независимыми. К унифицированным устройство независимым малым интерфейсам относятся, например, COM-порт, LPT-порт, USB-порт и т.п. Полный интерфейс ввода-вывода состоит из множества малых интерфейсов.

Необходимость в нескольких малых интерфейсах объясняется широкой номенклатурой периферийных устройств, среди которых есть как сравнительно медленно действующие (например, принтеры), так и высокоскоростные (по сравнению с процессором) устройства (накопители на магнитных дисках и лентах, видеомониторы), и которые к тому же отличаются разнообразием типов, имеют различную физическую природу и архитектуру, могут располагаться на значительных расстояниях от ЭВМ и характеризоваться многими другими особенностями. Все это делает задачу построения всего интерфейса в/в или отдельных внешних интерфейсов достаточно сложной в схемо- и системотехническом отношении.

Обязательным компонентом малого интерфейса является, как мы указали, шина ввода-вывода – набор линий связи, соединяющих контроллер непосредственно с периферийным аппаратом – принтером, дисководом, мышью и т.д. Подобный набор линий, а также система передаваемых по ним сигналов, отвечающих заданным, обычно унифицированным протоколам обмена, тоже называют *внешним* (или *малым*) *интерфейсом* (или *интерфейсом ПУ*) (см. рис.1.19 и 1.16, *а*). Шины внешних интерфейсов выводятся на внешние разъемные гнезда компьютера (размещаемые преимущественно на задней панели системного блока). Через эти разъемы («порты») к компьютеру и подключаются периферийные аппараты: клавиатура, монитор, мышь, флэш-карты и т.п. при помощи соединительных шлангов или собственных разъемных штекеров.

Другие виды интерфейсов ЭВМ. Наряду с внешними интерфейсами ЭВМ говорят и о внутреннем, или внутримашинном, или машинном (системном) интерфейсе. Под ним понимают совокупность средств соединения всех устройств центральной части ЭВМ в единую систему. Это, прежде всего, сама системная магистраль компьютера, а также дополнительные согласующие и усилительные электронные схемы. Ввиду высокой степени совместимости серийных микросхем процессора, памяти и контроллеров в/в объединение их в целостную систему не представляет сложной задачи и производится в некоторых случаях при помощи одной только системной магистрали с минимальным привлечением микросхем окружения. Поэтому системную магистраль часто тоже именуют системным интерфейсом.

Нами упомянуты два вида интерфейса ЭВМ: *внутренний* и *внешние* (рис.1.20). Кроме них можно говорить и об *интерфейсах между отдельными устройствами* ЭВМ, например, между центральным процессором и арифметическим сопроцессором, между процессором (микропроцессорной интегральной схемой) и системной магистралью, между разными периферийными устройствами и т.д. И, вообще, *интерфейсом ЭВМ* называют любой набор аппаратных, программных и конструктивных средств, предназначенных для сопряжения устройств ЭВМ между собой. В интерфейс входят линии, шины и магистрали, сигналы и электронные схемы, алгоритмы, программы и протоколы обмена, конструктивные элементы (например, разъемные соединители) и т.п. средства сопряжения.

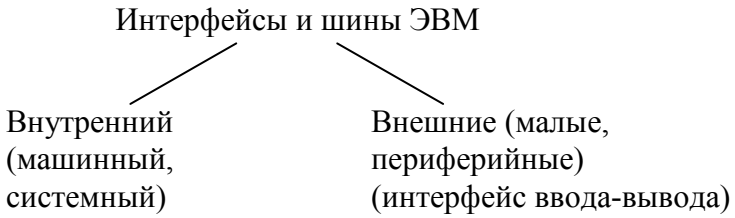


Рис. 1.20. Интерфейсы ЭВМ

Из сказанного можно также заключить, что смысловое содержание терминов «интерфейс», «интерфейсный модуль», «канал», «контроллер», «адаптер» довольно многообразно. Эти термины не являются ни однозначными, ни

общепринятыми, и значение каждого зависит от особенностей использования обозначаемых ими компонентов компьютера.

Унифицированный интерфейс. Большую роль для практического создания интерфейсов ЭВМ играет их стандартизация и унификация. *Унифицированный интерфейс* – это совокупность правил, устанавливающих единые принципы взаимодействия устройств ЭВМ, а также сами физические компоненты и машинные программы, обеспечивающие взаимодействие устройств по этим правилам. Стандартизации и унификации в интерфейсе обычно подлежат: форматы передаваемой информации; сигналы управления и состояния; номенклатура и характер линий связи; порядок (алгоритмы) функционирования; передающие и приемные электронные схемы, в т.ч. аппаратурные средства соединения устройств (разъемы, связи); электрофизические параметры сигналов и их временные диаграммы; конструктивные решения; наконец, программные средства, описывающие характер сигналов, их логические соотношения и режимы передачи. Унифицированный интерфейс может отвечать требованиям на все или только часть этих характеристик.

Выбор унифицированного интерфейса или, в общем случае, совокупности интерфейсов является одним из центральных, определяющих моментов в проектировании ЭВМ и других вычислительных систем. Говоря коротко, построить ЭВМ на серийной элементной базе – это и значит, прежде всего, спроектировать ее интерфейс. Как правило, ЭВМ строятся с использованием нескольких интерфейсов, например, интерфейса модулей центральной части системы, т.е. внутримашинного интерфейса (соответственно МП, ОЗУ, ПЗУ), интерфейсов подключения периферийных устройств (внешних интерфейсов). Выбор внутримашинного интерфейса определяется типом микропроцессора. Внешние интерфейсы (адаптеры и контроллеры ПУ и шины в/в) выбираются исходя из особенностей периферийных устройств и требований к характеристикам ввода/вывода. В случае МП ИУВС разрабатывается интерфейс подключения объекта управления (датчиков информации, исполнительных механизмов, аналого-цифровых и цифро-аналоговых преобразователей, устройств регистрации и т.п. устройств). В многопроцессорных системах необходим интерфейс между отдельными микропроцессорами.

Глава 2. ЭЛЕМЕНТЫ АРХИТЕКТУРЫ ЭВМ

2.1. Понятие об архитектуре и организации компьютера

«Компьютер – понятие многостороннее. Это и сложный схемотехнический агрегат, и универсальная логико-арифметическая модель, и большая электронная схема и еще множество конструкторских, технологических, гигиенических и др. проблем. Все эти стороны важны и просто необходимы. Но не схемы и электрические цепи составляют сущность компьютера, и не в количествах и скоростях переключения электронных вентилей состоит производимый компьютером полезный эффект. Сущность компьютера в автоматической реализации программно-управляемых последовательностей операций – *процессов*.» [5]

Именно с этой стороны компьютер характеризует его архитектура. Термин «архитектура компьютера», «архитектура системы» часто употребляется как в узком, так и в широком смысле этого слова. В узком смысле под архитектурой понимается архитектура набора команд, т.е. способность компьютера (процессора) выполнять определенный набор команд. Архитектура набора команд служит границей между аппаратурой и программным обеспечением и представляет ту часть системы, которая видна программисту или разработчику компиляторов. Следует отметить, что это наиболее частое употребление этого термина.

В широком смысле архитектура охватывает понятие организации системы, включающее в себя такие высокоуровневые аспекты разработки компьютера, как систему памяти, структуру системной шины, организацию ввода/вывода, идеологию обработки процессором потока команд и распараллеливания операций и т.п.

Итак, в узком смысле под **архитектурой компьютера, процессора (архитектурой набора команд)** будем понимать абстрактное представление процессора и машины в терминах основных – программно-доступных – функциональных модулей и узлов, языка ЭВМ, структуры данных. Архитектура процессора это воплощенная в аппаратуре и встроенных программах основа программируемого процессирования, включающая те аспекты структуры ЭВМ, которые являются видимыми для пользователя. К ним относятся:

- набор регистров компьютера, доступных пользователю. Это регистры самого процессора, ячейки основной памяти и порты (регистры) ввода-вывода периферийных устройств;
- типы, форматы и длина размещаемых в этих регистрах данных (форматом данного называют его двоичное представление с описанием назначения каждого бита);
- способы (режимы) адресации регистров, ячеек и портов;
- наконец, сама система команд;
- а также система прерываний.

Коротко, *архитектура* – это все то, что машина предоставляет программисту, работающему на уровне машинных команд, все, что нужно знать о машине системному программисту, что видит в машине программист, когда он работает на языке машинных команд.

«Поскольку сейчас ни один нормальный человек не пишет программ на машинном языке, можно переделать это определение. Программа уровня архитектуры процессора — это то, что выдает компилятор (в данный момент мы игнорируем вызовы операционной системы и символический язык ассемблера). Чтобы произвести программу уровня команд, составитель компилятора должен знать, какая модель памяти используется в машине, какие регистры, типы данных и команды имеются в наличии и т. д. Вся эта информация в совокупности и определяет архитектуру процессора.» [25]

Таким образом, архитектура – это, прежде всего, язык команд, вместе с тем, что за ним стоит, что в нем отражено. Это «видимая» часть внутренней конструкции процессора, обеспечивающая его совместимость с определенным набором команд. В отличие от организации, архитектура не определяет особенностей реализации аппаратной части ЭВМ, времени выполнения команд, степени параллелизма при выполнении программ. Программно не доступные, не отраженные в языке команд объекты: магистрали, буферные регистры, скрытые механизмы убыстрения доступа к памяти, схемы исправления ошибок, средства распараллеливания операций и т.п. к архитектуре не относятся – они относятся к организации.

Организация процессора – это то, как процессор устроен, как реализованы выполняемые им функции. Это система логико-арифметических, схемотехнических и системотехнических мероприятий, направленных на рациональное построение процессора и придание ему нужных пользовательских свойств, включая заданную архитектуру.

Говоря коротко, термин «архитектура» используется для описания возможностей, предоставляемых ЭВМ, а термин «организация» определяет, как эти возможности реализованы, архитектура отвечает на вопрос «что ЭВМ может?», а организация – «как она это делает?».

Показателем тождественности архитектуры ЭВМ является то, что всякая программа в машинном коде, выполняющаяся на одной из этих машин, выполнима на любой другой из них, причем получаемые результаты совпадают. ЭВМ, удовлетворяющие данному условию, называют обладающими одной и той же архитектурой или совместимыми на уровне архитектуры или совместимыми на уровне машинного языка (кода). Несмотря на то, что эти ЭВМ могут значительно различаться в организации или в других отношениях, например, в отношении скорости выполнения программ, с точки зрения программистов они совершенно одинаковы и их программное обеспечение является единым для всех ЭВМ. [5]

Архитектура компьютера, понимаемая в широком смысле, отражает главные аспекты его аппаратной организации. Она представляет построение компьютера с точки зрения системного инженера – разработчика компьютерно-

го «железа», а не только системного программиста (иногда, чтобы окончательно не запутаться, используется термин «микроархитектура»). В этом случае к архитектуре относят ключевые компоненты аппаратуры компьютера, отвечающие за принципиальные вопросы его функционирования. Именно: *архитектурой компьютера, микроархитектурой* называют концептуальную структуру компьютера, отражающую общий принцип его аппаратного устройства и организации работы, характеристики основных узлов и блоков, структуру управляющих и информационных связей между ними, способы реализации ими заданных целей и параметров. Архитектура в широком смысле включает такие высокоуровневые, общесистемные компоненты разработки компьютера и процессора, как

- структурную схему компьютера с описанием ее состава по функциональным органам и связей между ними;
- структуру, «внутреннюю конструкцию» процессора: состав, назначение и связи между блоками, идеологию обработки процессором потока команд, механизмы распараллеливания операций, кэширование памяти и др.;
- систему памяти, пространство адресуемых ячеек памяти, их размер, организацию чтения/записи в ячейки;
- систему портов (регистров) ввода/вывода, пространство адресов и размер портов, организацию ввода/вывода;
- организацию и разрядность интерфейсов ЭВМ: собственных, «первичных», внешних шин и сигналов процессора; системной шины или шин; внешних (периферийных) шин (шин ввода-вывода); контроллеров ввода-вывода;
- способы представления и форматы данных ЭВМ;
- наряду с указанными компонентами к архитектуре относят систему команд, способы адресации данных, систему прерываний и т.п., т.е. те аспекты построения компьютера, которые составляют архитектуру набора команд – построение компьютера с точки зрения системного программиста.

Все эти сведения нужны системному инженеру для обслуживания и разработки компьютерных систем на основе использования серийной элементной базы покупных, промышленно выпускаемых микрочипов процессора и его окружения.

Коротко: термин «архитектура компьютера» не имеет в настоящее время однозначного толкования, и его можно понимать как,

- во-первых, «абстрактное представление процессора, отражающее его совместимость с определенным набором команд» (архитектура в узком смысле, архитектура с точки зрения системного программиста),

- во-вторых, «комплекс высокоуровневых аппаратных компонентов компьютера, отвечающих за принципиальные вопросы его функционирования» (архитектура в широком смысле, архитектура с точки зрения системного инженера).

Во всех случаях *архитектура* даёт общее описание структуры и функций компьютера на уровне, достаточном для понимания его принципов работы и

системы команд, не включающее деталей технического и физического устройства компьютера.

Слово «архитектура» в понятии «архитектура компьютера» сходно по значению с его привычным употреблением в понятии «архитектура зданий». Архитектурой зданий, сооружений называют их облик, внешний вид. Подобно этому и архитектура компьютера означает функциональный облик компьютера с точки зрения системного программиста или системного инженера. Понятие архитектуры компьютера закономерно появилось ввиду сложности современных вычислительных машин как понятие, охватывающее комплекс общих вопросов построения компьютера, существенных в первую очередь для пользователя, интересующегося главным образом возможностями машины, а не частностями ее технического исполнения.

Во избежание ошибки напомним, что **структура компьютера** (как и любого объекта или системы) – это, согласно общепринятому определению, совокупность его функциональных элементов и связей между ними. Элементами могут быть самые различные устройства – от основных логических узлов компьютера до простейших схем. Структура компьютера графически представляется в виде структурных схем, с помощью которых можно дать описание компьютера на любом уровне детализации.

Основные виды компьютерных архитектур [20]. Двумя основными видами архитектур набора команд, используемыми компьютерной промышленностью на современном этапе развития вычислительной техники, являются архитектуры CISC и RISC.

CISC (Complicated или *Complex Instruction Set Computers – компьютеры с усложненной системой команд)*. Процессоры с CISC-архитектурой реализуют так называемую полную систему команд. Они характеризуются следующим набором свойств: сравнительно небольшое число регистров общего назначения, большое количество машинных команд, некоторые из которых семантически аналогичны операторам высокоуровневых языков программирования и выполняются за много тактов, большое количество способов адресации, большое количество форматов команд различной разрядности, преобладание двухадресного формата команд, наличие команд обработки регистр-память. Время выполнения команды может составлять от 1 до 12 тактов. К приборам с CISC-архитектурой относятся МК компании Intel с ядром MCS-51, которые поддерживаются в настоящее время целым рядом производителей, МК семейств HC05, HC08 и HC11 фирмы Motorola и ряд других.

Стремление авторов CISC-архитектур было сделать аппаратуру как можно более «умной». Однако жесткое «вшивание» сложных алгоритмов выполнения команд в «железо» приводило к тому, что аппаратура исполняла каждый раз некоторый общий алгоритм команды, требовавший десятков или даже сотен тактов процессора, но как-либо оптимизировать выполнение этих команд с использованием конкретной информации о длине строки, косвенной цепочки и т.д. возможности не было. Другой недостаток CISC-архитектур в том, что подобные групповые операции на время их выполнения фактически останавливали работу конвейера (pipeline) – реализованной в любой компьютерной архитектуре аппаратной оптимизации, параллельного выполнения нескольких соседних команд при условии их независимости друг от друга по данным.

RISC (Reduced Instruction Set Computers – компьютеры с сокращенным набором команд). Разработка этой архитектуры связана с осознанием важности упрощения набора команд для построения быстрых вычислительных машин. Процессоры с RISC-архитектурой имеют сокращенную до минимума систему команд (обычно до 30-40), их семантика проста, арифметические операции выполняются только в регистрах, а для считывания из памяти в регистр и записи из регистра в память используются специальные команды. Поэтому для выполнения сложных операций приходится использовать несколько команд. При этом все команды имеют формат фиксированной длины (например, 12, 14 или 16 бит), выборка команды из памяти и ее исполнение осуществляется за один такт синхронизации. В качестве аргумента разработчики RISC-процессоров утверждали, что даже если системе RISC приходится выполнять 4 или 5 команд вместо одной, которую выполняет CISC, RISC все равно

выигрывает в скорости, так как RISC-команды выполняются в 10 раз быстрее (поскольку они не интерпретируются) [25]. Другой особенностью RISC-архитектур является большое число регистров общего назначения, образующих так называемый регистровый файл (512, 1024, 2048 регистров и т.д.). Это позволяет большему объему данных храниться в регистрах на процессорном кристалле большее время и упрощает работу компилятору по распределению регистров под переменные.

К МК с RISC-процессором относятся МК AVR фирмы Atmel, МК PIC16 и PIC17 фирмы Microchip и другие. Классическим RISC-процессором является процессор Alpha 21064 компании DEC.

Учитывая преимущества производительности RISC, можно было бы предположить, что такие компьютеры, как Alpha, станут доминировать над компьютерами CISC (процессоры Intel и т. д.) на рынке. Однако ничего подобного не произошло [25]. Возникает вопрос: почему?

Во-первых, компьютеры RISC были несовместимы с другими моделями, а многие компании вложили миллиарды долларов в программное обеспечение для продукции Intel. Во-вторых, как ни странно, компания Intel сумела воплотить те же идеи в архитектуре CISC. Процессоры Intel, начиная с i486, содержат RISC-ядро, которое выполняет самые простые (и обычно самые распространенные) команды за один такт синхронизации, а по обычной технологии CISC интерпретируются более сложные команды. В результате обычные команды выполняются быстро, а более сложные и редкие – медленно. Хотя при таком «гибридном» подходе производительность ниже, чем в архитектуре RISC, новая архитектура CISC имеет ряд преимуществ, поскольку позволяет использовать старое программное обеспечение без изменений. [25]

VLIW (Very Long Instruction Word – компьютеры с широким командным словом) – подход к архитектуре компьютеров, сложившийся в 1980-х – 1990-х гг. Основная идея данного подхода – *статическое планирование параллельных вычислений компилятором* на уровне отдельных последовательностей команд и подкоманд. При данной архитектуре каждая команда является «широкой» (long) и содержит несколько *подкоманд*, выполняемых параллельно за один машинный такт на нескольких однотипных устройствах процессора – например, в таком компьютере может быть два устройства сложения, два логических устройства, два устройства для выполнения переходов и т.д. Задачей компилятора является оптимальное планирование загрузки всех этих устройств в каждом машинном такте и генерация таких (широких) команд, которые позволили бы оптимально загрузить на каждом такте каждое из устройств. Достоинством такой архитектуры является возможность распараллеливания вычислений, недостатком – сложность (по сравнению с RISC -архитектурой). Примеры компьютеров таких архитектур: из зарубежных – компьютеры Cray X/MP, Cray Y/MP и др.; из отечественных – многопроцессорный вычислительный комплекс «Эльбрус-3».

EPIC (Explicit Parallelism Instruction Computers – компьютеры с явным распараллеливанием) – по архитектуре аналогичны VLIW, но с добавлением ряда важных усовершенствований: например, *спекулятивных* вычислений – параллельного выполнения обеих веток условной конструкции с вычислением условия. Подход сложился и используется с 1990-х гг. Примеры процессоров данной архитектуры – Intel IA-64, AMD-64.

Multi-core computers (многоядерные компьютеры) – получившая наиболее широкую популярность в настоящее время архитектура компьютеров, при которой каждый процессор имеет несколько *ядер (cores)*,объединенных в одном кристалле и параллельно работающих на одной и той же общей памяти, что дает широкие возможности для параллельных вычислений. В настоящее время известные *многоядерные* процессоры фирмы Intel (Core 2 Duo, Dual Core и др.), а также мощные *многоядерные* процессоры фирмы Sun / Oracle: Ultra SPARC-T1 («Niagara») – 16-ядерный процессор**;** Ultra SPARC-T2 («Niagara2») – 32-ядерный процессор. Все ведущие фирмы мира заняты разработкой и выпуском все более мощных многоядерных процессоров. Соответственно, создатели операционных систем для таких

компьютеров разрабатывают базовые библиотеки программ, позволяющие в полной мере использовать возможности параллельного выполнения на многоядерных процессорах.

Hybrid processor computers (компьютеры с гибридными процессорами) – новый, все шире распространяющийся подход к архитектуре компьютеров, при котором процессор имеет *гибридную* структуру – состоит из (многоядерного) центрального процессора (CPU) и (также многоядерного) графического процессора (GPU – Graphical Processor Unit). Такая архитектура была разработана, в связи с необходимостью параллельной обработки графической и мультимедийной информации, что особенно актуально для компьютерных игр, просмотре на компьютере высококачественного цифрового видео и др. Гибридная архитектура является новым «интеллектуальным вызовом» для разработчиков компиляторов, которым необходимо разработать и реализовать адекватный набор оптимизаций как для центральных, так и для графических процессоров. Примерами таких архитектур являются новые процессоры фирмы AMD, а также графические процессоры серии Tesla фирмы NVidia.

2.2. Особенности архитектуры и организации процессоров Intel

Доминирующее положение на рынке универсальных микропроцессоров занимают микропроцессоры с системой команд x86. Основными производителями их являются компании Intel, AMD, VIA и, в меньших объемах, Cyrix, IDT, Transmeta, WinChip. Имя x86 закрепилось за данной системой команд благодаря тому, что впервые она была реализована в процессоре 8086 компании Intel (переводится с английского как «интегральная электроника») и продолжила свое развитие в процессорах Intel 80186, 80286 (i286), 80386 (i386), 80486 (i486). Более поздние модели семейства i80x86 стали называть именами собственными (например, Pentium), чтобы иметь возможность зарегистрировать их как торговую марку. Начиная с i386 эти процессоры 32-битные, а с Opteron и Athlon 64 фирмы AMD и Pentium 4F, Pentium D0 фирмы Intel – 64-битные. Ежегодный рост выпуска указанной категории процессоров составляет 10 – 15 %. Доля остальных процессоров (архитектура которых относится главным образом к RISC-типу) составляет около 20 % рынка.

Архитектура 32-битных процессоров с системой команд x86 обозначается как IA-32 (от Intel Architecture - 32 bit). Ее 64-битное расширение, разработанное компанией AMD, именуется AMD64 (также x86-64, AMD64/Intel64/EM64T). Оно имеет почти полную обратную совместимость с IA-32 и реализуется в линейках процессоров Opteron, Athlon 64, Pentium 4F, D0. К числу 64-битных относится также архитектура IA-64, разработанная совместно компаниями Intel и Hewlett Packard и несовместимая с архитектурой x86 (процессоры Intel Itanium).

В настоящей книге мы будем обращаться к архитектуре IA-32, являющейся также и ядром 64-битных процессоров AMD64. Главными чертами этой архитектуры являются 32-разрядные регистры, аппаратно-реализованный механизм виртуальной (сегментированной) адресации памяти и мультизадачность. Общее представление о процессорах IA-32 дает схема на рис. 2.1. На схеме ALU (Arithmetical Logical Unit), или DPU (Digital Processor Unit), – арифметико-логическое устройство, или целочисленное операционное устройство;

FPU (Floating Processor Unit) – операционное устройство (сопроцессор) для реализации арифметики над числами с плавающей точкой.

Разрядность. Процессор IA-32 (его целочисленное устройство DPU, Digital Processor Unit) обрабатывает данные длиной 8, 16 и 32 бита, имеет 36 программно-доступных регистров, из них 8 – общие, являющиеся расширением аналогичных регистров 8086. Внутренний блок арифметики с плавающей запятой FPU (Floating Processor Unit) содержит еще 16 программных регистров и оперирует с числами длиной 80 бит (64 бита – мантисса, 15 бит – порядок и 1 бит – знак числа). Несмотря на то, что архитектура системы осталась все еще 32-разрядной, внутри процессора стали использоваться 128- и 256 разрядные шины передачи данных. Внешняя шина данных увеличена до 64 бит. Область адресов оперативной (основной) памяти (ОП) составляет 4 Гбайт (2^{32}), так как шина адреса 32-битная, а начиная с процессора Pentium Pro (и далее у Pentium II/III) – 64 Гбайта (2^{36}, т.е. 36 бит у адреса). У процессоров Athlon 64, Phenom, Turion 64 и других архитектуры AMD64, а также Intel Itanium, Itanium 2 и т.д. архитектуры IA-64, появившейся на рубеже 1990-х – 2000-х годов, шина адреса и регистры уже 64-битные. Область адресов внешних устройств (портов ввода и вывода) у всего семейства Intel 80x86 – 64 Кбайта (по 64 К тех и других, адрес порта 16-битный), причем эта область изолирована от адресов памяти.

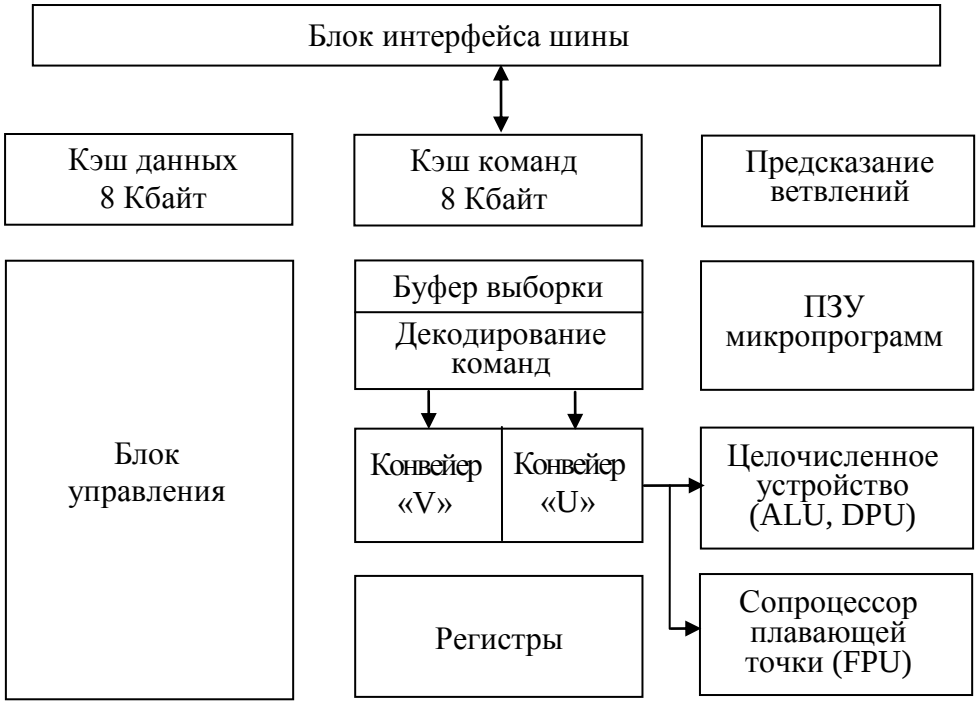


Рис. 2.1. Процессоры Intel и AMD

Процессор IA-32 способен работать в двух режимах – режиме реальных адресов без защиты памяти (так называемый «реальный» режим) и в режиме виртуальных адресов и защищенной памяти («виртуальный» или «защищен-

ный» режим). При работе в **реальном режиме** процессор ведет себя как быстрый 8086 с возможностью обрабатывать 32-разрядные операнды в памяти и собственных регистрах. Можно использовать 32-разрядные регистры и для адресации памяти, но 32-битный внутрисегментный адрес должен быть меньше, чем 0000FFFF. При превышении этого значения происходит прерывание 13. Адрес ячейки, который выставляется в шину адреса, (так называемый «физический адрес», именно его воспринимает память) 20-битный (адреса лежат в диапазоне 00000000 – 000FFFFF). Поэтому максимальный размер памяти в реальном режиме составляет 2^{20} байтов = 1 Мбайт, и эта доступная память занимает первый мегабайт общего адресного пространства памяти. Кроме того сразу после пуска процессора программа занимает последние 64 Кбайта памяти. Но первый же межсегментный переход переводит процессор в первый мегабайт памяти, после чего программы могут размещаться только в этом мегабайте.

Реальный режим устанавливается автоматически после **пуска процессора**, происходящего по сигналу системного сброса RESET (другое название – CLR – Clear, очистка). Затем процессор начинает работать в режиме реальных адресов с ячейки памяти, имеющей адрес FFFFFFF0, при запрещенных прерываниях по входу INTR. Реальный режим необходим для подготовки работы процессора в защищенном режиме.

Из реального режима МП программно переводится в **защищенный режим**, который является естественным состоянием процессора. В защищенном режиме появляется возможность использования всего адресного пространства памяти объемом в 4 Гбайт, многозадачной обработки информации, защиты памяти с помощью четырехуровневого механизма привилегий и страничной организации памяти. Перевод процессора в защищенный режим производится путем загрузки 1 в бит PE (Protection Enable) управляющего регистра CR0 (п.2.3). В защищенном режиме длина всех видов адресов – логических и физических составляет 32 бита (адреса лежат в диапазоне 00000000 - FFFFFFFF).

Внутри защищенного режима возможно переключение (путем установки в единицу флажка VM – Virtual Mode) в виртуальный режим 8086 (**режим V86**), позволяющий выполнять прикладные программы 8086 с использованием защиты, страничной организации памяти и мультизадачного окружения. В виртуальном режиме процессор IA-32 эквивалентен нескольким параллельно работающим процессорам 8086. (В реальном режиме он ведет себя как один 8086.) Таким образом, одновременно могут быть запущены несколько программ, которые выполняются как бы различными процессорами 8086.

Наличие диспетчера памяти на кристалле обеспечивает поддержку сегментации (виртуальной адресации) и страничной организации памяти средствами МП, четыре уровня защиты.

Сегментация является средством управления пространством логических (виртуальных) адресов, она обеспечивает мобильность и повышает защищенность программ. *Сегменты* – это непрерывные участки памяти, выделяемые в ней самим процессором по назначению программы. Программа назначает для

сегментов начальные (базовые) адреса, размер (в байтах или страницах) и другие характеристики (атрибуты).

Каждой задаче доступны до 16384 (2^{14}) сегментов размером до 4 Гбайт каждый (2^{32} байт). Общий объем логического (виртуального, кажущегося) адресного пространства памяти составляет $2^{14}x2^{32} = 2^{46}$ = 64 Тбайт (терабайт).

Программы обращаются к сегментированной памяти путем указания начального адреса (базы) сегмента и адреса ячейки внутри сегмента (смещения). Блок сегментации автоматически суммирует их, получая несегментированный, линейный адрес, отсчитываемый от начала всей памяти (от ячейки 00000000). Линейный адрес и выставляется процессором в шину адреса (ША) в качестве физического адреса, если отсутствует страничное деление памяти. Если такое деление есть, то линейный адрес предварительно преобразуется в физический с учетом открытой страницы.

Сегментация памяти полезна для организации легко переносимых модулей. Для переноса достаточно изменить в программе номер (селектор) сегмента, а внутрисегментные адреса остаются без изменения. Кроме того, сегментация используется для организации защиты памяти (путем наделения сегментов соответствующими уровнями привилегий и другими характеристиками защищенности от проникновения в них других программ).

Повышается надежность программ и систем. Например, размещение программного стека и самой программы в различных сегментах позволяет предотвратить наложение данных и команд при переполнении стека.

Блок **страничной адресации** памяти действует на более низком уровне, разбиение памяти на страницы возможно только в защищенном режиме (при условии программной установки режима страниц в бите PG – Paging). Каждый сегмент делится на страницы размером по 4 Кбайта, которые могут размещаться в любом месте памяти. Если сегментация полезна для организации в памяти локальных модулей, то страницы нужны системному программисту для эффективного использования физической памяти системы – для заполнения всех, даже небольших и разрозненных ее участков, если потребуется, нужной информацией, которые в этом случае не будут бесполезно пустовать. Страничная организация памяти в нашем пособии не рассматривается. С ней можно ознакомиться в [4, 9, 19].

Так как адресуемая физическая память содержит не более 4 Гбайт, то полностью использовать все сегменты микропроцессор может только при использовании **виртуальной памяти**. Виртуальная память отличается от обычной, оперативной памяти тем, что какие-то ее редко используемые фрагменты могут находиться на диске и подгружаться в реальную оперативную память по мере необходимости. Такая организация памяти позволяет снять ограничения, накладываемые объемом физической памяти, установленной на компьютере, и адресовать все 64 Тбайт (2^{46}) виртуальной памяти.

Виртуальная память противоположна по своей организации и форме обращения «электронному диску». «Электронный диск» – это часть ОЗУ, отведенная для хранения дисковой информации. Обращение к «электронному дис-

ку» производится пользователем подобно обращению к обычному, магнитному или оптическому диску. Виртуальная память, наоборот, физически располагается на диске, но адресуется программистом как оперативная память и подгружается в ОП по мере обращений к ней. Виртуальная память позволяет значительно расширить логическое адресное пространство оперативной памяти, но понижает ее быстродействие. Наоборот, электронный диск резко повышает скорость операций, но уменьшает объем доступной ОП.

Мультизадачность означает, что на одном процессоре одновременно могут выполняться несколько программ. Операционные системы (ОС), поддерживающие мультизадачность, выполняют периодическое переключение между задачами, учитывая их приоритет. При этом каждой задаче может отводиться вся доступная данной ОС виртуальная память. Важным достоинством мультизадачной ОС является то, что она выполняет диспетчеризацию задач только в защищенном режиме работы процессора. Это означает, что аварийное завершение одной задачи не приводит к остановке других задач и всей операционной системы в целом.

Понятие мультизадачности связано с понятием *многопользовательности* и появилось в 60-е годы 20 века. В эти годы были разработаны программные и аппаратные механизмы для разделения компьютера между группами пользователей. Их цель – создать одновременно у каждого пользователя иллюзию монопольного владения компьютером. Моделируемая копия компьютера каждого пользователя называется *задачей*, а создание и использование на компьютере нескольких задач называется *мультизадачностью*. Один из способов реализации мультизадачности, называемый *разделением времени*, заключается в разделении каждой секунды на кванты, в течение которых выполняются разные задачи. Именно благодаря разделению времени все пользователи одновременно уверены в том, что у каждого из них есть своя, хотя и более медленная копия компьютера. Операционные системы, которые используют разделение времени, называются *системами с разделением времени* (ОС РВ).

Конвейеризация команд. Важными элементами организации Pentium и последующих моделей являются конвейеризация команд и кэширование памяти. Они имелись и у i486, а конвейеризация (но не внутреннее кэширование) – у i386. Эти элементы наиболее полно воплощают в себе принцип параллельных систем и служат мощным средством ускорения обращения к памяти.

Конвейеризация команд означает, что процессор одновременно выполняет несколько операций по обработке потока команд. Пока одна команда выбирается из памяти, вторая дешифрируется, а третья исполняется.

Чтобы делать все это одновременно сам процессор должен быть разделен на полунезависимые устройства, каждое из которых одновременно с другими реализует свою функцию. У высокоорганизованных процессоров такими устройствами являются три основные части процессора: шинная, управляющая и операционная, а также, возможно, отдельные их функциональные блоки (рис. 2.2).

Шинное устройство считывает команды в процессор. Непосредственно этой операцией занимается относящийся к ШУ блок опережающей выборки команд. Он использует шину, когда она не занята операционным устройством, считывает команды из памяти и помещает их в очередь. Очередь располагается в специальном наборе регистров процессора. Эти регистры (*регистры очереди команд, буфер выборки команд*; см. рис.2.1) являются аналогом регистра команд простых процессоров. Если регистр команд простого процессора хранит одну выполняемую команду, выбранную из памяти, то регистры очереди хранят сразу несколько выбранных команд, и эти команды обслуживаются далее процессором в порядке их очередности – так, как они хранились в программе в памяти.

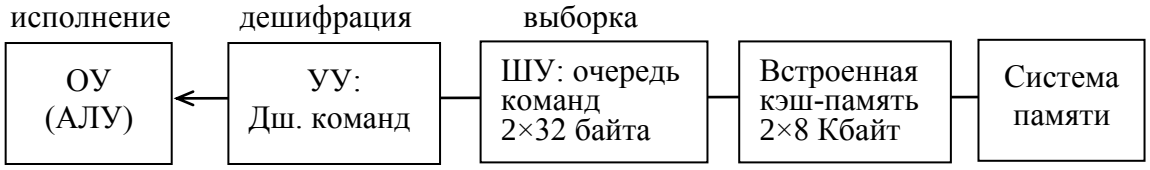


Рис. 2.2. Конвейер команд

Размер очереди команд зависит от типа процессора. Для МП IA-32, имеющего два конвейера команд, это 2×32 байта (что соответствует 2×(6…14) командам).

Наличие двух конвейеров команд у процессора принято называть «суперскалярной архитектурой». Подобная организация позволяет процессору выполнить одновременно более одной команды за один такт. Две очереди команд МП IA-32 работают параллельно, обеспечивая реализацию команд над данными в одном такте в каждом конвейере. Конвейеры МП IA-32 называются U и V-конвейерами (см. рис.2.1), процесс параллельного выполнения двух команд – «спариванием» («pairing»). U-конвейер может выполнять любую команду процессора, тогда как V-конвейер – «простые» («simple») команды, определенные в соответствии с правилами «спаривания» команд (Instruction Pairing Rules). Спариваемые команды не должны иметь регистровых зависимостей. При формировании «спаренных» команд команда, выполняемая в V-конвейере, является всегда следующей после команды, реализуемой в U-конвейере.

Блок дешифрирования (декодирования) (входящий в состав управляющего устройства) преобразует команду в формат, стандартный для управляющего блока (блока выработки управляющих сигналов) и операционного устройства. Управляющий блок вырабатывает серии командных сигналов на все другие устройства, включая их на выполнение нужных операций. Операционное устройство подвергает операнды фактической обработке (преобразованию).

В очереди дешифратора процессора IA-32 могут храниться 2×6 команд. Частью дешифрованной команды является указатель управляющего ПЗУ, которое хранит микрокод, управляющий устройствами процессора.

Наконец, операционное устройство действует по командным сигналам от устройства управления, фактически выполняя команду. Оно содержит арифметико-логический блок и набор регистров. Сегментный блок вычисляет адреса и контролирует защиту сегментов. Адрес от этого блока подается в страничный

блок, который (если станицы включены) преобразует его в физический адрес. В противном случае используется линейный адрес от сегментного блока, выдаваемый в шину адреса в качестве физического. Сегментный блок использует также блок опережающей выборки для своих запросов и страничный блок для доступа к таблицам страниц в памяти.

Обычно на конвейере одновременно обрабатывается несколько команд (5-6 команд в каждом из двух конвейеров МП IA-32), то есть по завершению одной команды сразу начинается исполнение следующей, которая уже выбрана из памяти и дешифрирована.

После появления конвейера стало неизбежным появление блока предсказания ветвлений (условных переходов) (см. рис. 2.1). При переходах происходит полное обнуление конвейера, что снижает общую производительность. Блок переходов, превосходя события (расчеты ошибочной ветви данных), определяет, будет ли выполнен условный переход или нет. Для этого он использует динамический метод предсказания переходов, при котором не только анализируются данные и инструкции, которые готовятся к обработке процессором, но еще анализируется история аналогичных переходов, которую этот блок сам же и накапливает. Благодаря тому, что блок постоянно отслеживает итоговый результат (угадал или нет), и сравнивает его со своим предсказанием, дополняя собственную статистику, эффективность предсказаний в аналогичных ситуациях в будущем значительно увеличивается. Из-за такой тактики правильных предсказаний у данного блока гораздо больше, чем неправильных – современные процессоры от Intel и AMD в 95 - 97 % случаев верно определяют направление условного перехода. Естественно, благодаря этому, обнуление конвейера происходит относительно редко.

Внутренняя кэш-память. Дополнительно производительность повышается благодаря использованию кэш-памяти. *Кэш-память* (от англ. cache – тайник, тайный склад) представляет собой быстродействующую буферную память ограниченного объема, которая располагается между процессором и относительно медленной основной памятью и является прозрачной для программиста: программа не видит, проходит информация через нее или нет. Эффективнее увеличивать частоту кэша, а для этого он должен находиться как можно ближе к ядру процессора. Кэш-память может работать на частоте 1/4, 1/3, 1/2, 1/1 от частоты ядра (п. 1.2). Современные микропроцессоры имеют кэш объемом от 8 Кб до 5Мб.

В процессе работы отдельные блоки информации копируются из основной памяти в кэш-память, и когда процессор обращается за командами или данными, то сначала проверяется их наличие в кэш-памяти. Если они находится там, то они быстро извлекаются. Такой случай обращения называют *кэш-попаданием*. Если необходимая информация в кэш-памяти отсутствует, то она выбирается из основной памяти и одновременно заносится в кэш-память. В этом случае в кэш копируется целая строка (в случае МП IA-32 – 32-байтная), включающая затребованный байт. Заполнение 32-байтной строки (называемой также линией, кадром) кэш-памяти осуществляется быстрыми пакетными циклами в промежутках между магистральными циклами команд (чтобы не задерживать выполнение команд). Такой случай называют *кэш-промахом* (рис. 2.3).

Повышение быстродействия вычислительной системы достигается в том случае, когда кэш-попадания реализуются намного чаще, чем кэш-промахи.

Высокий процент кэш-попаданий обеспечивается благодаря тому, что в большинстве случаев программы обращаются к ячейкам памяти, расположенным вблизи от ранее использованных. Это свойство, которое называют локальностью программ, обеспечивает эффективность использования кэш-памяти.

Рис. 2.3. Операции с кэш-памятью
и с основной памятью процессора IA-32

Если при чтении произошел кэш-промах, то производится, как уже сказано, обновление целой строки кэш-памяти, то есть информация вводится в нее из основной памяти. Новая информация замещает ранее имевшуюся. При этом удаляется в первую очередь информация (строка), отмеченная как недостоверная. Например, сразу после пуска компьютера или после очистки кэша вся информация в нем считается недостоверной. Строка объявляется недостоверной также при аннулировании ее другим процессором по двунаправленной шине адреса. Для объявления строки недостоверной ее бит недостоверности V (Valid) сбрасывается в ноль. Если недостоверной информации в кэшэ нет, то чаще всего используется правило удаления кэш-информации, которая дольше всего оставалась невостребованной. Такой алгоритм обновления называется «замещение наименее используемой информации» (LRU – least recently used). В этом

случае блок управления кэш-памятью осуществляет контроль использования информации, обеспечивая реализацию этого алгоритма (что обеспечивается при помощи специальных битов, обозначаемых буквой B).

Отдельные области (страницы) основной памяти могут быть закрыты для кэширования аппаратно или программно. Внешняя аппаратура сообщает микропроцессору, что ячейка с данным адресом не может быть помещена в кэш переводом сигнала на входе МП KEN# во время цикла чтения памяти в пассивное состояние. Программное обеспечение выполняет аналогичное действие установкой бита PCD (Page Cache Disable) в таблице страниц. В этом случае процессор не производит заполнение строки кэш-памяти, а читает лишь затребованный байт из основной памяти (не копируя его в кэш).

Нормальная работа МП системы с кэш-памятью требует поддержки соответствия содержимого основной памяти и кэш-памяти. Такая поддержка называется *кэш-согласованием*. При чтении кэш-согласование обеспечивается копированием в кэш строки, содержащей затребованный байт. Запись в память также производится с реализацией процедуры кэш-согласования. При записи в ОП делается одновременная попытка записи в кэш, и при этом тоже возможны попадания и промахи в кэш. Если ячейка с целевым адресом есть в кэше (т.е. ранее была скопирована в него), то такая запись является кэш-попаданием. В этом случае в цикле записи производится запись по заданному адресу, как в соответствующую строку кэш-памяти, так и в адресуемую ячейку основной памяти. Эта операция называется *сквозной записью* (Writethrough).

Pentium использует и другой способ записи в кэш в случае кэш-попадания – так называемую *обратную запись* (WriteBack). При этом способе модифицируется сначала только строка в кэш-памяти. Данный способ приводит к сокращению занятости внешней магистрали за счет исключения ненужной записи в основную память. Перезапись модифицированной строки (такая строка помечается специальным признаком – битом «мусора») из кэш-памяти в основную память осуществляется позднее, посредством обратной записи, производимой при удалении данной строки; или для поддержания механизма кэш-согласования в многопроцессорных системах. Перезапись может быть инициирована аппаратно или программно. После перезаписи бит «мусора» сбрасывается.

Сквозная запись применяется только к тем областям памяти, которые нуждаются в немедленном кэш-согласовании, т.е. в данный момент разделяются несколькими микропроцессорами в многопроцессорных системах. Для пометки этих областей каждой разделяемой строке кэш-памяти (копии соответствующей строки основной памяти) присваивается соответствующее состояние (фиксируемое в специально признаке кэша). Во всех остальных случаях Pentium использует обратную запись. Т.е. если строка не помечена упомянутым выше признаком немедленного кэш-согласования, то используется обратная запись – сначала только в кэш, и лишь потом (по требованию внешних устройств или программы) – в ОП.

С помощью механизмов сквозной и обратной записи осуществляется поддержка упомянутого выше протокола кэш-согласования (Cache Consistency

Protocol) в многопроцессорных системах. Это согласование означает, что при изменении некоторых данных в кэш-памяти данных одним из процессоров системы остальные также будут получать при обращении к ним скорректированные данные.

Мы указали способы записи в память, реализуемые при условии попадания в кэш. При кэш-промахе, т.е. когда целевая ячейка в кэше не обнаружена (ранее не была скопирована), производится запись только в основную память и обновление строки кэш-памяти не производится. Т.о., при записи в память обращение к внутреннему кэшу производится только при условии кэш-попаданий.

Аннулирование строки кэш-памяти (удаление строки другим процессором). Процессор Pentium имеет программные и аппаратные средства для аннулирования строк внутреннего кэша. Под этим понимается удаление кэш-строки другим процессором, который произвел запись в основную память, дублируемую этой строкой. Это необходимо для того, чтобы при многопроцессорном режиме работы всегда имелось соответствие между содержимым внешней памяти и содержимым внутреннего кэша.

Аннулирование строки сводится к тому, что при обнаружении записи в ячейку основной памяти, которая дублируется внутренним кэшем, содержимое соответствующей строки во внутреннем кэше объявляется недействительным (недостоверным): бит недостоверности V=0, и внутреннее чтение по идентифицированному адресу запрещается. Для реализации аннулирования МП имеет механизм слежения за операциями записи других процессоров во внешнюю память. В этот механизм входят соответствующие управляющие сигналы и двунаправленная шина адреса.

Очистка кэша. Внутренняя кэш-память (команд и данных одновременно) может быть очищена внешними аппаратными средствами Pentium посредством подачи сигнала на вход FLUSH#, или программно с помощью команд INVLD (очистка) и WBINVLD (обратная запись и очистка). Очистка кэш-памяти сводится к очистке битов достоверности всех строк.

В МП IA-32 используются две раздельные внутренние кэш-памяти: команд и данных емкостью 8 Кбайт каждая, всего 16 Кбайт внутреннего кэша. Чтение команд производится из внутренней кэш-памяти команд, чтение данных – из внутренней кэш-памяти данных. Запись данных (операндов) производится только в кэш-память данных (во внутреннюю кэш-память команд нельзя производить запись).

Обращение к внутренней кэш-памяти ведется по тем же физическим адресам, что и к обычной памяти. Но эти адреса во внешнюю шину адресов не попадают, они передаются по внутренней шине процессора. Использование внутреннего кэша резко сокращает время доступа к информации.

В завершение кратко заключим, что *кэш-память – это быстро доступная копия части основной памяти, прозрачная для программиста*.

Управление внутренней кэш-памятью осуществляется битами CD (Cache Disable) и NW (No Write) регистра CR0, которые определяют четыре режима работы в соответствии с табл. 2.2.

Таблица 2.2 – Режимы работы внутренней кэш-памяти

CD	NW	Режим
1	1	Запрещены заполнение строки, сквозная запись и закрытие строки
1	0	Запрещены заполнение строки; сквозная запись и закрытие строки разрешены
0	1	Недопустимая комбинация. Загрузка этого кода в регистр CR0 вызывает исключение 13 с кодом ошибки 0
0	0	Разрешены заполнение строки, сквозная запись и закрытие строки

Внешняя кэш-память. В то время как *внутренняя* кэш-память реализуется непосредственно на кристалле процессора, *внешняя* выполняется в виде отдельного набора микросхем памяти на материнской плате. В микропроцессорных системах на основе МП IA-32 помимо внутренней кэш-памяти (1-й уровень кэширования, 8 Кбайт команд и 8 Кбайт данных) допускается использование дополнительной внешней кэш-памяти (2-й уровень кэширования) до 256 Кбайт. Для управления внешней кэш-памятью МП IA-32 вырабатываются специальные сигналы (в настоящем пособии они не рассматриваются). У процессоров P6+ кэш 2-го уровня располагается в одной упаковке с ядром и подключается к специальной внутренней шине процессора.

2.3. Регистры процессора

Для программы компьютер выглядит как система *программно-доступных регистров* – регистров, к которым возможен доступ. К ним относятся собственные регистры процессора, ячейки основной памяти и порты (регистры) периферийных устройств.

Описание программно-доступных, адресуемых регистров компьютера принято называть *регистровой* или *программной моделью компьютера* (или *процессора*). На рис.2.4 и 2.5 показана структура программной модели 32-битных Intel-совместимых процессоров. Программы процессоров Intel имеют доступ ко всем трем упомянутым и независимым категориям регистров компьютера [1, 2, 4, 8, 9, 19]:

- собственным регистрам процессора;
- ячейкам основной памяти и
- портам ввода и вывода периферийных устройств.

1. *Собственные регистры процессора*. Процессор IA-32 (старшие модели Intel, начиная с i386) (более точно – его целочисленное устройство DPU, Digital Processing Unit) содержит 36 программных регистров, из них 16 используются прикладными программами, а остальные – операционной системой.

   Прикладные регистры подразделяются на :

- общие (EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP);

- флаги (EFlags);
- указатель команд (EIP);
- сегментные (CS, SS, DS, ES, GS, FS) и связанные с ними программно-недоступные дескрипторные регистры.

Системные регистры включают:

- управляющие (CR0, CR1, CR2);
- системных адресов (GDTR, IDTR, LDTR, TSR);
- отладочные (DR0 - DR7);
- тестовые (TR3 - TR7).

Еще 16 регистров находятся в составе встроенного арифметического сопроцессора – блока FPU (Floating Point Unit), реализующего операции над числами с плавающей запятой.

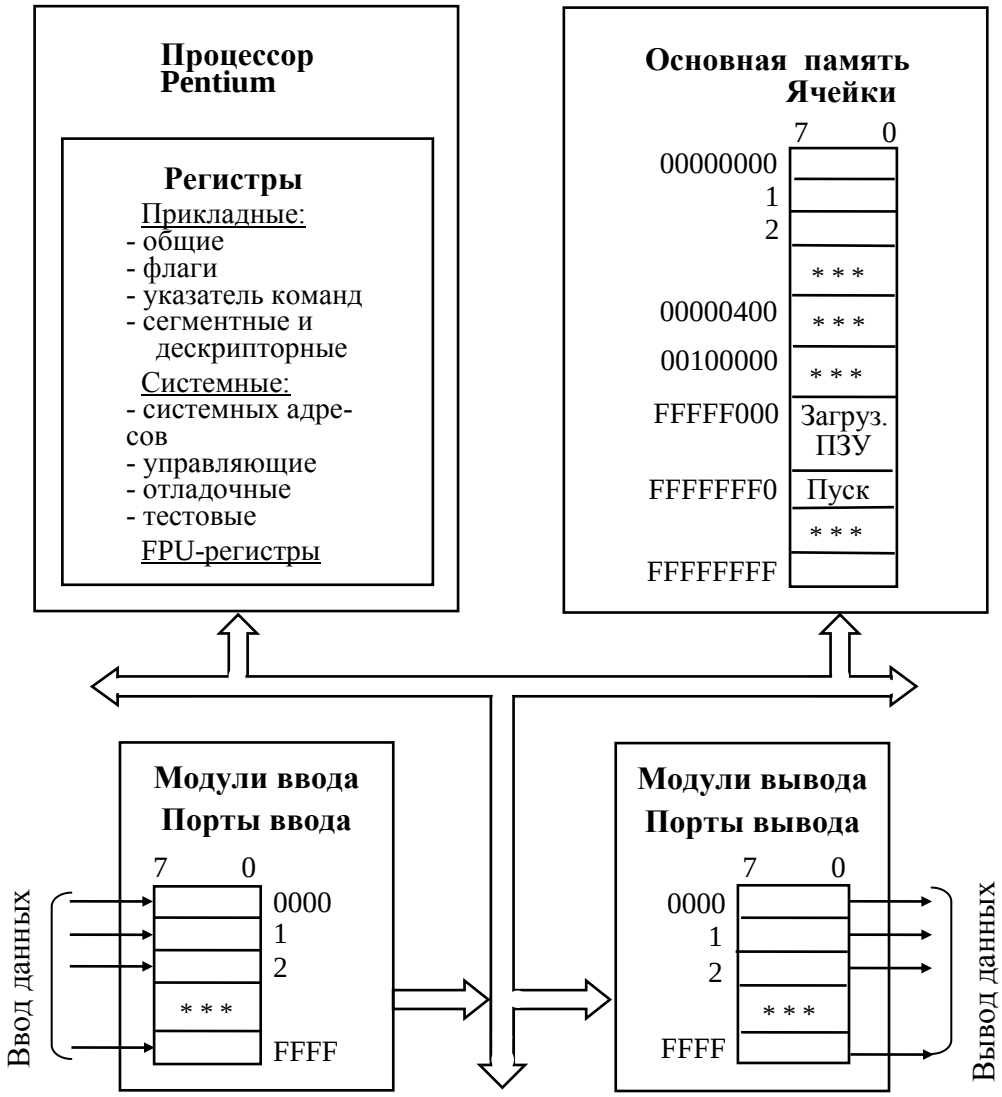


Рис. 2.4. Структура регистровой модели компьютера архитектуры IA-32

Рис. 2.5. Регистры целочисленного устройства процессора IA-32. Обведены регистры процессоров 8086 и Intel старших моделей в реальном и V86 режимах

2. *Ячейки основной памяти*. Ячейки памяти организованы побайтно – по одному байту (8 бит) каждая. Каждый байт имеет свой уникальный номер, или адрес. У процессора IA-32 адрес состоит из 32 битов и изменяется от 0 до FFFFFFFF, всего – 4 Гбайт информации. Любые два смежных байта мо гут быть использованы как слово, четыре байта – как удвоенное слово, восемь байт – как учетверенное слово. Байты в слове хранятся по правилу: младший байт – по младшему адресу, а адресом слова служит адрес его младшего байта. Адрес, который воспринимает память, называется физическим, адрес, фигурирующий в программе, – логическим и может отличаться от физического. Логические адреса процессоров Intel сегментируются (п.2.4).
3. Адресуемые *регистры* и другие *видимые элементы* периферийных аппаратов (например, индикаторы, входы запуска аналого-цифровых преобразователей и т.п.) – *порты ввода-вывода*. У процессоров IA-32 адреса портов не сегментируются. Адрес порта 16 битный, что позволяет различать 64 К однобайтных портов ввода и столько же вывода. Как и байты памяти, и по тому же правилу, смежные байты в портах объединяются в одинарные, двойные и (в некоторых последних моделях процессоров) учетверенные слова.

Внутренние (собственные) регистры целочисленного устройства процессоров архитектуры Intel делятся на прикладные и системные.

Прикладные регистры.

Среди прикладных регистров IA-32 есть 8 **общих регистров** (данных и адресов), содержащих по 32 бита и являющихся расширением 16-битных регистров МП 8086 (Extended registers):

- EAX – аккумулятор, или накопитель (Accumulator)
- EBX –база, указатель базы (Base)
- ECX – счетчик, счетчик циклов (Counter)
- EDX – регистр данных (Data)
- ESI – индекс источника (Source Index)
- EDI – индекс приемника (Destination Index)
- EBP – базовый указатель в стеке (Base Pointer)
- ESP – указатель стека (Stack Pointer)

В этих регистрах можно адресовать отдельно младшие 16-битные половины по именам AX, BX, …, SP, а в регистрах AX, BX, CX, DX – еще и младшие и старшие байты: AH и AL, BH и BL, CH и CL, DH и DL (H – старший байт, High, L – младший байт, Low).

Общие регистры предназначены для хранения операндов или их адресов в памяти, а DX – еще и адресов портов.

Примеры использования общих регистров в командах процессора:

```
add AX, 3
```

; AX←AX+3 – прибавить число 3 к содержимому регистра-аккумулятора AX и запомнить результат в аккумуляторе.

add DS:[AX], 3

; *m*(DS:AX)←*m*(DS:AX)+3 – прибавить число 3 к содержимому ячейки памяти, адрес которой внутри сегмента данных DS задается регистром AX, и занести в ту же ячейку.

Примеры обращения к портам ввода-вывода через регистр DX.

in AL, DX

; AL ← *p*(DX) – ввести в регистр-аккумулятор AL данные из порта, адрес которого находится в регистре DX.

out DX, AX

; *p*(DX) ← AX – вывести в порт, адресуемый регистром DX, число из AX.

Регистр указатель стека.

Указатель стека (Stack Pointer) ESP/SP. Выражение ESP/SP обозначает одновременно регистр ESP и его младшую часть SP. Аналогичное обозначение мы будем использовать и для других регистров.

Регистр ESP/SP условно причисляют к общим регистрам, хотя он выполняет специальную функцию – хранит адрес вершины стека – последней загруженной ячейки в стеке (п.2.7).

16-битный формат SP используется в реальном и V86 режимах процессоров IA-32, т.е. работает только младшая часть регистра ESP. В защищенном режиме работает полный, расширенный, 32-битный регистр ESP.

Последнюю группу расширенных регистров процессоров составляют EIP/IP – указатель команд, или программный счетчик, и EFlags/Flags – флаговый регистр.

Оба они 32/16-битные. Их младшие 16 бит соответствуют регистрам IP и Flags процессора 8086. В реальном и V86 режимах МП IA-32 работают младшие половины этих регистров, в защищенном – полные, 32-битные регистры.

Указатель команд.

Регистр EIP/IP – указатель команд (Instruction Pointer), содержит всегда адрес следующей по порядку, еще не исполненной команды программы, которую процессор будет выбирать из памяти вслед за текущей, только что выбранной, командой. EIP/IP – это управляющий регистр, так как (в паре с CS – тоже управляющим регистром) он управляет порядком выборки команд из памяти (п.4.5).

Флаговый регистр.

Расширенный регистр флагов EFlags имеет длину32 бита, исходный Flags, являющийся подмножеством расширенного, – 16 бит (п.2.5). В регистре EFlags (хотя он и 32-битный) задействовано 18 бит. EFlags (как и Flags) состоит из отдельных триггеров, называемых флагами. Флаги Flags делятся на арифметические и управляющие. К арифметическим относятся флаги CF, OF, SF, ZF, PF, AC, к управляющим – IF, DF, TF. Дополнительные флаги расширенного регистра EFlags нами не рассматриваются.

Сегментные и дескрипторные регистры.

Процессор IA-32 имеет шесть 16-битных сегментных, или селекторных, регистров, которые содержат селекторы сегментов (Segment Selector), указывающие на текущие адресуемые сегменты памяти. К ним относятся:

- CS – регистр-указатель кодового, программного сегмента (Code Segment). В кодовом сегменте находится выполняемая в данный момент программа;
- SS – регистр-указатель стекового сегмента (Stack Segment). В стековом сегменте располагается стек;
- DS – указатель сегмента данных (Data Segment), в котором размещаются данные (операнды);
- ES, FS, GS – указатели дополнительных сегментов данных для хранения операндов.

С каждым из сегментных регистров связан программно-недоступный, «теневой» 64-битный дескрипторный регистр, являющийся аппаратным расширением сегментного регистра. Можно сказать, что фактическая длина сегментных регистров 80 бит, но программы «видят» только старшие 16 бит. Собственно они и называются указателем или селектором сегмента. Остальные 64 бита явно программе недоступны. Они являются аппаратным расширением селекторного регистра и называются дескрипторным регистром.

Наличие сегментных и дескрипторных регистров объясняется сегментной адресацией памяти. Каждый сегментный регистр с помощью своего «теневого» расширения – дескрипторного регистра указывает на свой сегмент – непрерывный участок памяти, задавая его начальный (базовый) адрес (base), длину и другие параметры, а ячейки внутри этого участка (сегмента) адресуются при помощи общих регистров EAX/AX, EBX/BX, … и/или указанием числового адреса; или при помощи регистра EIP/IP для кодового сегмента.

В силу сегментации адреса́, находящиеся в общих регистрах EAX/AX, EBX/BX, … (с возможными числовыми добавками), а также в указателе команд EIP/IP являются не абсолютными, отсчитываемыми от начала всей памяти, а относительными, отсчитываемыми от базовых (и абсолютных) адресов сегментов (base). Эти относительные адреса принято называть смещениями (offset) (относительно базы).

Сегментация памяти разбирается в п.2.4, 4.5.

Системные регистры.

Регистры системных адресов:

GDTR (Global Descriptor Table Register) – регистр глобальной дескрипторной таблицы GDT, 48 бит. Содержит 32-битный базовый адрес и 16-битный предел таблицы GDT, содержащей дескрипторы общесистемных сегментов.

IDTR (Interruption Descriptor Table Register) – регистр дескрипторной таблицы прерываний, 48 бит. Содержит 32-битный базовый адрес и 16-битный предел дескрипторной таблицы прерываний IDT, через которую произво-

дится обработка прерываний и особых случаев, возникающих в ходе выполнения программы.

LDTR (Local Descriptor Table Register) – регистр локальной дескрипторной таблицы и

TSR (Task State Register) – регистр задачи, оба по 16 бит. По существу, они выполняют функции сегментных регистров, т.е. опосредованно (через глобальную дескрипторную таблицу) определяют два важнейших сегмента, относящихся к текущей задаче.

Регистр LDTR определяет локальную дескрипторную таблицу LDT (Local Descriptor Table), которая содержит дескрипторы сегментов, являющихся «частной собственностью» текущей задачи. С помощью локальных дескрипторных таблиц в мультизадачной среде осуществляется изолирование задач друг от друга.

Регистр задачи, или состояния задачи, TSR определяет текущий сегмент состояния задачи TSS (Task State Segment). Такие сегменты ассоциируются с каждой задачей в системе и предназначены для сохранения контекста (состояния) задачи при необходимости переключения процессора с одной задачи на другую. Определение таблицы LDT и сегмента TSS с помощью селекторов означает, что они превращаются в «обычные» сегменты памяти, которыми могут манипулировать средства виртуальной памяти.

Регистры GDTR и LDTR рассматриваются в п.2.4, IDTR в п.5.6, а TSS в настоящем пособии не обсуждается.

Управляющие регистры:

CR0, CR1, CR2, CR3, CR4. Служат для фиксации общего состояния процессора [4, 9, 10].

В регистре CR0 имеются флажки PE (Protection Enable) и PG (PaGing), управляющие переключением в защищенный режим (PE) и в режим страничного разбиения памяти (PG). Другие функции этих регистров нами не рассматриваются.

Отладочные и тестовые регистры:

DR0 - DR7(Debug Registers) и TR3 - TR7 (Test Registers) также в настоящем пособии не изучаются.

Пуск процессора производится по сигналу системного сброса RESET с ячейки памяти, имеющей адрес FFFFFFF0. Затем процессор начинает работать в режиме реальных адресов при запрещенных прерываниях (по входу INTR).

Пусковая ячейка FFFFFFF0 должна находиться в загрузочном ПЗУ (ROM BIOS), которое занимает некоторую область адресного пространства, включающую его конец – последние 64 Кб с адресами FFFF0000 - FFFFFFFF. Из пусковой ячейки управление передается (при помощи команды перехода jmp) на загрузочную программу компьютера, находящуюся в ROM BIOS. Эта программа производит инициализацию (начальную установку) компьютера и загрузку пусковой части операционной системы с жесткого диска. После подготовки системных данных в памяти (GDT и IDT) ОС переключает процессор в

защищенный режим, который используется для выполнения задач компьютера. (См. также п.4.6.)

Сразу после пуска регистры процессора устанавливаются на значения, показанные в табл.2.3. В частности, сегменты данных DS, SS, ES, FS, GS установлены на базовые адреса, равные нулю (см. табл.2.3), и далее в реальном режиме могут располагаться (при последующих переключениях сегментов) только в первом Мбайте памяти. Кодовый сегмент CS после пуска находится в последних 64 Кбайт памяти, но после первого же межсегментного перехода его начальный адрес (база) тоже обнуляется, т.е. сегмент переводится в первый Мбайт памяти.

После переключения компьютера в защищенный режим сегменты могут располагаться в любых местах всего адресного пространства памяти в 4 Гбайт.

Таблица 2.3 – Значения регистров МП IA-32 после инициализации

Регистры	Содержимое
EIP	0000FFF0
CS	F000
CS base	FFFF0000
DS, SS, ES, FS, GS	0000
DS base, SS base, ES base, FS base, GS base	00000000
EFlags	xxx00002
EAX	00000000
EBX, ECX	Не определено
EDX	0400+(номер версии)
ESP, EBP, ESI, EDI	Не определено
EFlags	00000002h
GDTR, IDTR, LDTR, TR	База=0, граница=0
Регистры очереди команд пусты	
CS base – базовый (начальный) адрес кодового сегмента; DS base, SS base, ES base, FS base, GS base – базовые адреса сегментов данных	

Система прерываний Intel 80x86 допускает обработку 256-ти прерываний с номерами *n*=0,1,2,3….255, вызываемых сигналом INTR или программно, командой int. Переход на подпрограммы прерываний (ПП, ISR – Interruption Subroutine) осуществляется через *вектора прерываний*. Вектора прерываний (ВП) – это пусковые адреса подпрограмм прерываний. В реальном режиме каждый вектор занимает 2 слова, т.е. 4 байта, предназначенные для занесения в регистры CS и IP, и хранится в памяти. Под хранение всех векторов отводится первый Кб памяти по адресам 0…..3FF, называемый *таблицей векторов прерываний* (ТВП). В защищенном режиме вектор прерывания имеет 8 байт и называется *дескриптором прерывания*. Он содержит новые значения для CS и EIP, а остальные 16 бит дескриптора используются для других целей. Дескрипторы прерываний размещаются в специальной таблице дескрипторов прерываний IDT (Interruption Descriptor Table). Местонахождение таблицы в памяти задается системным адресным регистром IDTR, который содержит линейный начальный адрес таблицы, размер и другие характеристики.

NMI – другой сигнал прерывания (для экстренных ситуаций), вызывающий подпрограмму прерывания с номером 2.

Прерывания изучаются в гл.5.

2.4. Сегменты в памяти

Понятие сегмента памяти. Мы упоминали (в п.2.2, 2.3), что память условно делят на непрерывные участки, которые принято называть *сегментами* [3, 8, 9, 11, 18]. («Непрерывность» участка означает, разумеется, что он состоит из последовательных, смежных ячеек – ячеек со смежными адресами.) Местоположение каждого сегмента в памяти задается его начальным адресом и длиной. *Начальный адрес сегмента* – это адрес его самой первой однобайтной ячейки (т.е. меньший из адресов его ячеек) в общем адресном пространстве памяти. Его называют также *базовым адресом* или просто «базой» сегмента. Под *длиной, размером сегмента* понимают количество составляющих его однобайтных ячеек. Базовые адреса могут быть любыми, но на длины сегментов накладывается ограничение: для младших моделей процессоров 80x86 (8086, 8088, 80186) и в реальном режиме старших процессоров (80386, 80486, Pentium и др.) размер любого сегмента фиксирован и равен 2^{16} = 64 Кбайт = 65536 байт, а в защищенном режиме старших процессоров он устанавливается программно и может быть от 0 до 4 Гбайт.

В этих условиях абсолютный адрес A любой ячейки памяти можно представить в виде суммы A = base + offset, где base – *база* сегмента, к которому относится ячейка A, а offset – *смещение* или относительный адрес ячейки внутри сегмента, т.е. ее адрес (номер), отсчитанный от начала сегмента – от базового адреса base. Отсчет производится по ячейкам размером в один байт, и смещение пробегает значения 0, 1, 2, …, $\text{offset}_{\text{max}}$, где $\text{offset}_{\text{max}}$ – максимально допустимое значение смещения (называемое «пределом» или «границей» сегмента), равное размеру сегмента (в байтах) минус 1. Итак, *смещение* (offset) – это относительный адрес, адрес ячейки внутри сегмента, *внутрисегментный адрес*.

Соответственно, абсолютный адрес ячейки A, относящейся к данному сегменту, может принимать значения base, base +1, base + 2, …, base + $\text{offset}_{\text{max}}$.

Коротко: *база определяет начало сегмента, а смещение – расстояние от базы* (рис.2.6).

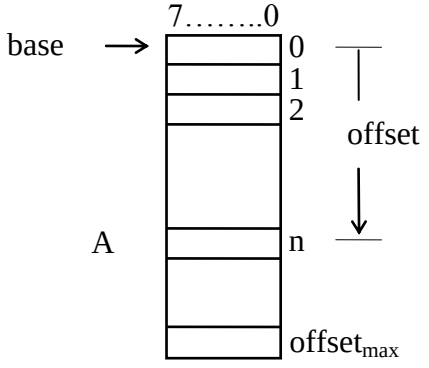


Рис. 2.6. Сегмент памяти.
base – база, offset=n – смещение,
A=base+offset – линейный адрес ячейки

Операция base + offset кратко записывается с помощью выражения base:offset (база:смещение, seg:offset, сегмент:смещение). Указанный способ определения адреса памяти называется *сегментной адресацией*, *сегментацией адреса*, *сегментацией памяти*. При нем полный, абсолютный, *линейный адрес* памяти A (Line address) представляется в виде двухкомпонентного, *сегментированного адреса* base:offset. Адрес A называется линейным, так как он отсчитывается от начала всей памяти и пробегает линейный ряд значений: от 00000 до FFFFF – для младших процессоров и в реальном режиме старших; и от 00000000 до FFFFFFFF – в защищенном режиме старших процессоров. Это обеспечивает доступ процессора к первому мегабайту памяти в реальном режиме и ко всему пространству памяти объемом 4 Гбайт – в защищенном режиме.

Организация сегментов. Сегменты памяти – это непрерывные участки памяти, выделяемые в ней самой программой. Программа назначает для сегментов начальные (базовые) адреса, размер, тип (программа, данные, стек), уровень защиты и другие характеристики (атрибуты). Поэтому сегменты являются программными, логическими единицами памяти, а не аппаратными, физическими. Они могут располагаться в любых местах памяти, отстоять друг от друга, но могут и перекрываться и даже быть одной и той же областью памяти. Обычно размещением сегментов в памяти занимается операционная система, а прикладная программа лишь сообщает, какие сегменты в памяти она хотела бы иметь. После этого она может работать с сегментами – записывать и читать из них данные. Напомним, что в реальном режиме процессоров Intel программно задаются только базовые адреса сегментов, а в защищенном – все атрибуты. Пример расположения сегментов в памяти показан на рис.2.7.

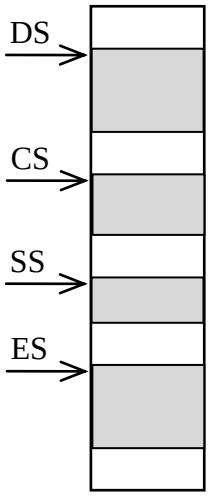


Рис. 2.7. Сегменты в памяти

Сегментированные, линейные и физические адреса. Программы обращаются к сегментированной памяти путем задания начального (базового) адреса сегмента («базы», base) и адреса ячейки внутри сегмента (внутрисегментного адреса, смещения относительно базы, offset). Базовый адрес должен быть занесен в процессор (вместе с другими атрибутами сегмента) в дескрипторный регистр, являющийся «теневым» расширением сегментного регистра (п.2.3). Это происходит автоматически в момент открытия сегмента – при загрузке 16-битного числа (называемого *селектором*) в сегментный регистр (о чем пойдет речь ниже).

Внутрисегментный адрес (смещение) назначается с помощью любого общего регистра процессора (EAX/AX, EBX/BX, ECX/CX, …) или напрямую – числовой константой, или их сочетанием; или же посредством указателя команд EIP/IP (для кодового сегмента, идентифицируемого регистром CS).

Пара чисел, записываемая как selector:offset (селектор:смещение, сегмент:смещение, база:смещение, seg:offset, base:offset), получила имя *сегментированного* или *логического адреса*. Блок сегментации автоматически транслирует сегментированный, логический адрес в несегментированный, *линейный адрес* (Line address), отсчитываемый от начала всей памяти (от ячейки 00000000). Это делается путем суммирования базы и смещения:

Line address = base+offset.

На основе линейного образуется *физический адрес* ячейки, т.е. адрес, который выставляется процессором в шину адреса (ША). Именно его воспринимает память. Если страничное деление памяти отсутствует, то в качестве физического используется линейный адрес, который и выставляется в ША. Если же страницы активированы, то линейный адрес предварительно преобразуется в физический с учетом открытой страницы. Страничное деление памяти включается установкой бита PG (page) регистра CR0 в «1» (единицу). Сам же порядок и инструменты разбиения памяти на страницы мы рассматривать в этой книге не будем.

Открытие сегментов. Для *назначения, открытия нового сегмента надо загрузить его селектор (16-битное число) в сегментный регистр*. Это число называют селектором (Selector) потому, что он выбирает («селектирует») один конкретный сегмент из множества возможных. Загрузка селектора в регистры DS, ES, FS, GS, SS осуществляется с помощью команды mov, а в регистр CS – с помощью команд передачи управления (jmp и др.). После загрузки селектора процессор автоматически, в цикле тех же команд mov, jmp, и др., определяет (установленным образом; см. далее) атрибуты сегмента – базу и другие характеристики и запоминает их в дескрипторном регистре, связанным с данным сегментным регистром. С этого момента он «видит» сегмент и может с ним работать: записывать и читать из него данные; или читать команды из кодового сегмента.

Поскольку у процессоров Intel нет команд занесения числа сразу в сегментный регистр (ввиду экономии количества кодов команд; за исключением регистра CS, куда селектор заносится одной командой jmp и др.), то используется две команды mov. Например, если мы хотим открыть новый сегмент данных Data Segment, идентифицируемый селектором 73h, надо сначала занести число 73h в общий регистр, а потом перенести его оттуда в сегментный:

```
mov AX, 73h     ; занести новый селектор 73h
mov DS, AX      ; в сегментный регистр DS через регистр AX (Здесь h, от англ. hexadecimal, hex – шестнадцатеричный, – это суффикс, обозначающий 16-ричную систему счисления.)
```

Первая команда заносит селектор 73h в регистр AX, а вторая – переносит его из AX в DS. После этого, в цикле второй команды, в дескрипторном регистре, связанном с DS, автоматически формируется базовый адрес и другие характеристики сегмента. Тем самым, сегмент данных устанавливается, становится доступным для процессора, и процессор может к нему обращаться.

Отметим, что в защищенном режиме работы процессора селектор способен открыть целевой сегмент лишь при выполнении условий доступа к нему. В противном случае вырабатывается прерывание защиты памяти (см. п. 6.2, 6.3).

Формирование базового адреса сегмента в реальном режиме. Базовый адрес сегмента памяти формируется в дескрипторном регистре по-разному в разных режимах. В реальном режиме он образуется простым сдвигом селектора на 4 бита влево («умножение на 16») и заносится в дескрипторный регистр (рис. 2.8). Получившееся 20-битное значение и принимается в качестве базового адреса сегмента. Его старшие 16 бит, как мы видим, совпадают с селектором, а младшие 4 бита заполняются нулями. Таким образом, *селектор в реальном режиме* – это и есть базовый адрес, но без четырех младших битов, которые равняются нулю, т.е. *селектор служит по существу прямым адресом сегмента*.

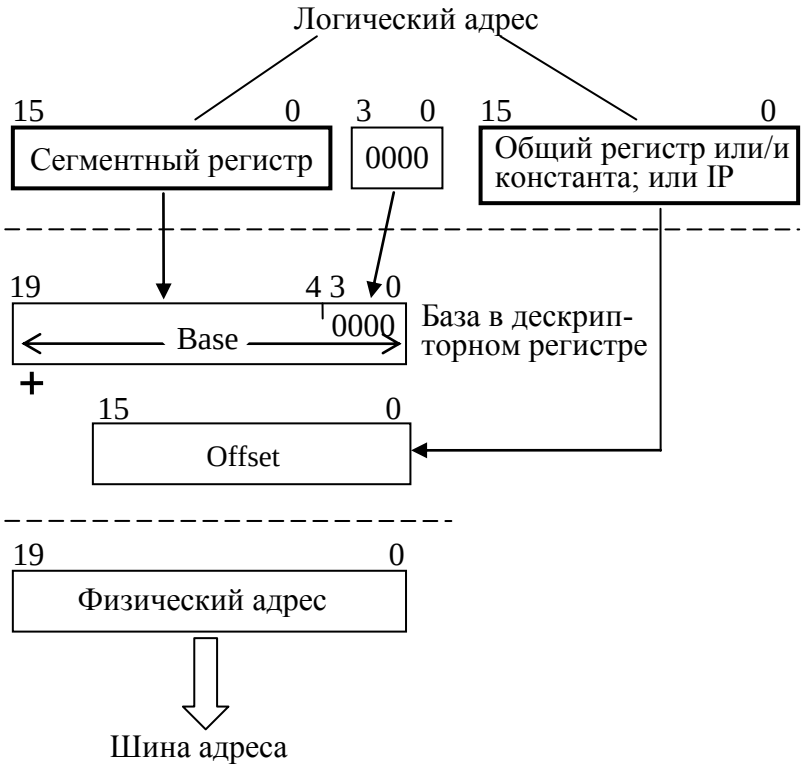


Рис. 2.8. Преобразование логического адреса
в физический в реальном режиме

Определение линейного адреса памяти в реальном режиме Смещение (offset) в реальном режиме 16-битное. Оно задается 16-битными общими регистрами или/и числовыми константами, а для CS – регистром IP. Линейный адрес вычисляется процессором по формуле:

Line address = base + offset = sr×16 + offset,

где sr – содержимое сегментного регистра, т.е. селектор; sr×16 = sr_0 – конкатенация (сочленение) sr и 4-х битного нуля, т.е. приписывание к sr четырех двоичных нулей справа («умножение на 16»), что дает базу сегмента base (см. рис. 2.8). Размер сегмента в этом режиме фиксирован и равен 64 Кбайта, а другие характеристики (атрибуты) не задаются.

Линейный адрес в реальном режиме имеет длину 20 бит, как и базовый адрес. Если при вычислении представленной суммы возникает перенос (из старшего разряда 19 в несуществующий разряд 20), то он игнорируется, нигде не запоминается, и за ячейкой с адресом 000FFFFF следует ячейка с нулевым адресом 00000000 (кольцевая организация памяти). Линейный адрес выставляется процессором в шину адреса, т.е. используется в реальном режиме как физический адрес памяти.

Особенности сегментации памяти в защищенном режиме. Дескрипторы. В защищенном режиме сегменты памяти определяются иначе, чем в реальном режиме. В данном режиме невозможно представить всю информацию о параметрах сегмента в одном 16-битном сегментном регистре. Базовый адрес сегмента в защищенном режиме 32-битный, а не 20-битный, и кроме того, назначаются другие характеристики. Реализация нескольких уровней привилегий и механизмов защиты задач требует больше информации о каждом сегменте. Основное отличие сегментации памяти в защищенном режиме от реального связано с защитой памяти и заключается в том, что теперь программе не разрешается обращаться по любому адресу памяти, который она может сформировать.

Таким образом, вместо простого определения сегмента путем загрузки его базового адреса в сегментный регистр системный программист определяет довольно много информации о каждом сегменте: его базовый адрес, длину (размер), целевое использование (для хранения кода, стека или данных), уровень привилегий и другие атрибуты. В задании атрибутов сегментов предоставляется довольно большая свобода. Например, базовый адрес может быть любым (не требуется его кратность 16-ти), а размер сегмента варьируется от одного байта до всего адресного пространства в 4 Гбайт. Последний размер показывает, что с помощью 32-битного смещения можно «пройти» все адресное пространство, не модифицируя содержимого сегментного регистра. Такая свобода определения сегментов предоставляет программисту полное управление использованием памяти и позволяет удовлетворить практически любые запросы памяти [9].

Для назначения сегментов используется специальная 64-битная структура данных, называемая *дескриптором сегмента* (segment descriptor). Дескриптор – это описатель сегмента. Он содержит базовый адрес и размер сегмента, тип (определяющий целевое использование сегмента – программа, данные, стек), уровень привилегий и дополнительную информацию о состоянии, т.е. все то, что необходимо знать о сегменте.

Число дескрипторов в работающей системе практически не ограничивается. Если не описать область адресного пространства памяти дескриптором, соответствующий диапазон адресов оказывается недоступным и процессор отказывается обращаться к нему. Когда программа ошибочно или намеренно формирует такой недействительный адрес, процессор «перехватывает» виноватую команду до попытки считывания/записи по неопределенному адресу.

Дескрипторы сегментов хранятся в памяти в составе массивов данных, сформированных в виде таблиц. Используется три типа дескрипторных таблиц:

- GDT – глобальная дескрипторная таблица (Global Descriptor Table);
- LDT – локальная дескрипторная таблица (Local Descriptor Table);
- IDT – дескрипторная таблица прерываний (Interruption Descriptor Table).

Таблица GDT содержит любые типы дескрипторов, кроме используемых при обработке прерываний. Все программы (задачи), выполняющиеся в системе, могут использовать эту таблицу для обращения к сегментам памяти; говорят еще, что таблицу GDT разделяют («коллективно используют») все задачи.

Таблицы LDT содержат дескрипторы сегментов, используемых при решении данной задачи. В принципе каждая задача может иметь отдельную таблицу LDT, которая включает дескрипторы сегментов, используемых при ее решении. Кроме того, LDT могут содержать дескрипторы *шлюзов* (gates) – точек входа в программы или задачи, которые используются при вызовах программ (подпрограмм) и переключениях задач (гл.6). В случае совместного использования сегментов таблицы LDT могут полностью или частично перекрывать друг друга.

Таблица IDT используется для реализации прерываний (гл.5).

Все таблицы могут иметь размеры от 8 до 64 Кбайт, т.е. содержать до 8192 дескрипторов.

Формат дескриптора приведен на рис. 2.9. Дескриптор содержит 32-битный линейный адрес сегмента (базовый адрес), 20-битный размер сегмента и 12 бит других атрибутов сегмента.

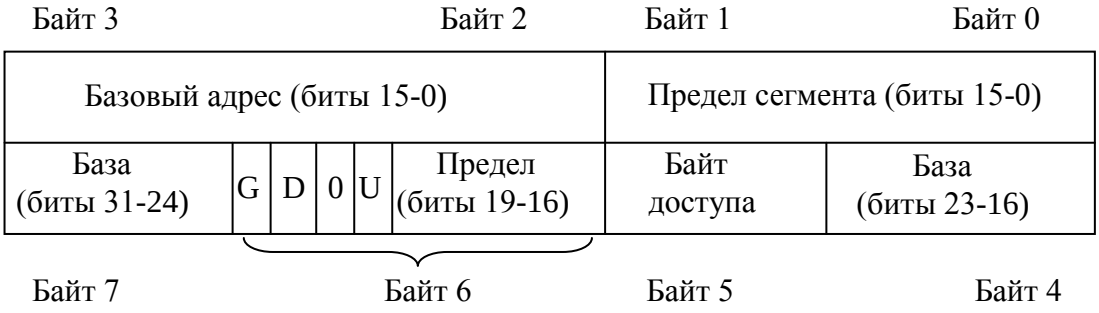


Рис. 2.9. Общий формат дескриптора

Базовый адрес. 32-битный базовый адрес занимает байты 2, 3, 4 и 7 дескриптора. Он определяет начальный адрес сегмента в линейном адресном пространстве 4 Гбайта и может быть любым. Именно этот адрес процессор сформирует при задании нулевого смещения (offset=0).

Предел, или **граница сегмента** задается в 20-битном поле байтов 0, 1 и младшей половины байта 6. Предел равен размеру сегмента в байтах или страницах минус 1 в зависимости от значения бита гранулярности G (7 бит байта 6): при G=0 – в байтах и при G=1 – в страницах объемом по 4 Кбайта. Можно считать, что предел задает максимально допустимое смещение или последнюю адресуемую страницу в конкретном сегменте. В результате сегмент может иметь

размер от 0 до 2^{20} = 1 Мбайт при G=0 и от 0 до 2^{32} = 4 Гбайт при G=1. Смещения или номера страниц, которые больше предела, вызывают особый случай защиты (прерывание 13; гл.5, 6).

Отдельные биты **6 байта** определяют следующие атрибуты сегмента.

Бит дробности (гранулярности) G (бит 7 в байте 6 или бит 55 в дескрипторе) уже рассмотрен нами.

Бит разрядности по умолчанию D (Default Size) (бит 6). Для сегментов данных бит D работает в паре с битом W (Word) в коде команды. Бит W определяет разрядность операндов: W=0 – операция с байтами; W=1 – операция со словами (16 или 32 разрядными). Бит D уточняет разрядность слов (при W=1): 16 – при D=0, 32 – при D=1. Эта разрядность и принимается. Ее можно изменить, если использовать специальные префиксы разрядности.

Для сегментов стека. Если производится обращение к стеку (т.е. выполняется команда push, pop, call или др. подобные), то при D=0 используется регистр SP (16-разрядный), а при D=1 – регистр ESP (32-разрядный).

Для сегментов кода. Процессор интерпретирует содержимое кодового сегмента, дескриптор которого имеет D=0, как 16-битый код процессора i286. Кодовый сегмент, дескриптор которого имеет D=1, считается содержащим 32-битый код процессора i386, 486, Pentium.

Бит 5 в байте 6 (бит 53) зарезервирован фирмой Intel для будущих процессоров и программисты не должны использовать этот бит. Он должен содержать 0.

Бит пользователя U (User) – бит 4 в байте 6 предназначен для использования системными программистами по их усмотрению, а процессор игнорирует этот бит. Такие «свободные» биты в зарубежной литературе обычно отмечаются как AVL (Available – доступный, свободный). Возможные применения бита U включают в себя отметку сегментов, предназначенных для сбора «мусора», или сегментов, базовые адреса которых нельзя модифицировать.

Байт 5 (байт доступа) дескриптора определяет права доступа к выбираемому сегменту (байт AR – Access Rights). Этот байт описывается в п. 6.1 при рассмотрении методов защиты памяти [1, 4, 9].

Слежение за байтами дескриптора целиком лежит на трансляторе и операционной системе, а не на программисте. Эти байты устанавливаются при формировании дескрипторов в дескрипторных таблицах.

Селектор в защищенном режиме. Обращение к дескриптору в таблице GDT или LDT осуществляется с помощью селектора, загружаемого в сегментный регистр: CS, DS, SS, ES, FS или GS. Селектор выбирает (селектирует) целевой дескриптор среди других в таблице. Он представляет собой 16-разрядный указатель, который имеет три поля: RPL, TI и Index (рис. 2.10).

Поле RPL (биты 0-1) определяет запрашиваемый уровень привилегий (Request Privilege Level). Это двухразрядный код, указывающий допустимый уровень защиты сегмента, который может быть выбран с помощью данного селектора. Но код RPL не может уменьшить значение текущего уровня привиле-

гий CPL (Current Privilege Level), содержащийся в регистре CS (т.е. не может увеличить привилегию сегмента). Уровни привилегий и методы защиты памяти по ним рассматриваются в п.6.3.

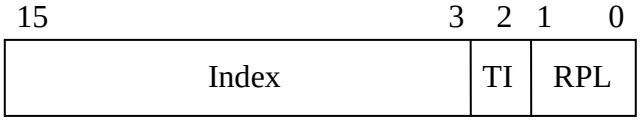


Рис. 2.10. Формат селектора

Поле TI (бит 2) служит индикатором таблицы (Table Indicator). Его значение указывает выбираемую таблицу: GDT при TI=0, LDT при TI=1. Местоположение таблиц GDT и LDT в памяти (их начальные адреса) задаются регистрами GDTR и LDTR. При каждой загрузке селектора в сегментный регистр процессор автоматически считывает дескриптор из заданной таблицы и заносит его в дескрипторный регистр.

Поле Index (биты 3-15) служит индексом (номером) для выбора одного из 2^{13}=8192 дескрипторов, содержащихся в таблице.

Формирование базового адреса и других характеристик сегмента в защищенном режиме. Как только селектор загружается в сегментный регистр (см. «Открытие сегментов» выше), процессор извлекает указываемый им дескриптор из дескрипторной таблицы – глобальной (GDT) (при TI=0) или локальной (LDT) (при TI=1) и заносит его в дескрипторный регистр, связанный с задействованным сегментным регистром (рис. 2.11). Далее, если правила доступа к сегменту соблюдены (гл.6), в последующих обращениях к памяти используется уже этот извлеченный из таблицы дескриптор для адресации сегмента. При нарушении правил доступа генерируется внутреннее прерывание защиты памяти, т.е. процессор отказывается открывать этот сегмент (гл.6.) Таким образом, дескриптор в дескрипторном регистре является быстро доступной копией дескриптора из таблицы, и эта копия дает полное описание сегмента. Иными словами, применяется кэширование (caching) дескрипторов в дескрипторных регистрах, а сами эти, теневые, регистры являются кэш-регистрами. Подчеркнем также, что извлечение дескриптора из таблицы происходит автоматически, в командном цикле тех же команд mov, jmp и пр., которые загружают селектор в сегментный регистр.

При обращении к таблице селектор используется как указатель на дескриптор, т.е. как номер дескриптора в таблице. Для определения положения дескриптора относительно начала таблицы его номер (Index, размещенный в старших 13 битах селектора, рис. 2.10) умножается на 8, то есть реально сдвигается на три разряда влево, так как длина дескриптора составляет 8 байт. Если селектор обращается к дескриптору, содержащемуся в таблице GDT (при TI = 0 в селекторе), то полученное смещение сравнивается с хранящейся в GDTR границей таблицы. Если нарушения границы нет, то смещение прибавляется к со-

держащемуся в GDTR базовому адресу, в результате чего образуется линейный адрес выбираемого дескриптора.

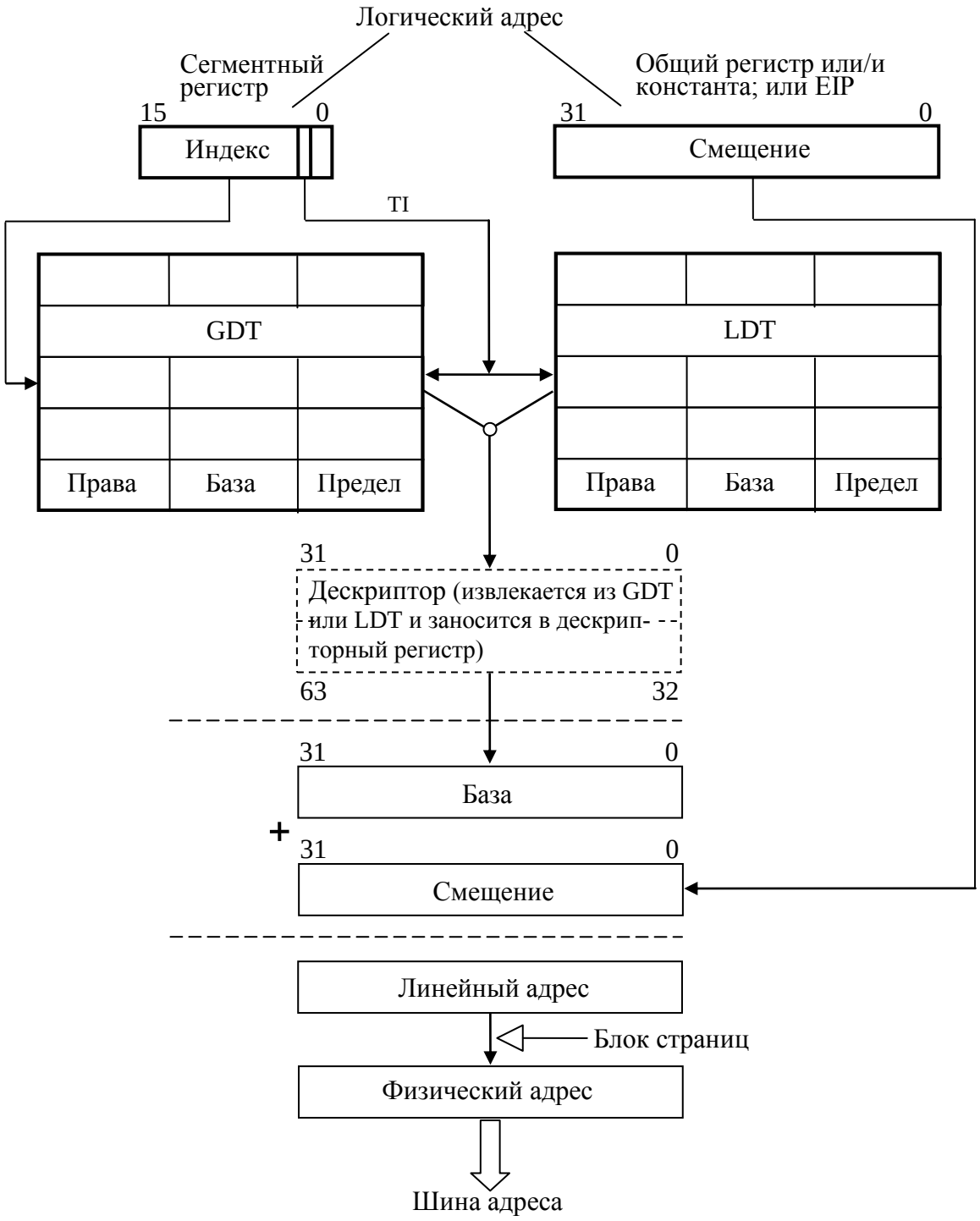


Рис. 2.11. Преобразование логического адреса
в физический в защищенном режиме

Нулевой дескриптор в GDT является пустым, не используемым. Селектор с нулевым значением разрядов 2….15 называется *нуль-индикатором*. Он обеспечивает обращение к нулевому дескриптору GDT. Так как этот дескриптор не используется, то при обращении к нему происходит прерывание. Одно из возможных применений пустых селекторов заключается

в следующем. Перед инициированием задачи операционная система может загрузить в регистры DS и ES пустые селекторы. Если в последующем не инициализировать эти регистры, то адресация памяти через них вызовет особый случай (прерывание).

При TI = 1 дескриптор выбирается из локальной таблицы LDT. Ее местоположение задается регистром LDTR, но не им самим непосредственно, а его 64-битным «теневым» расширением. Именно: для задания таблицы LDT надо загрузить ее 16-разрядный селектор в регистр LDTR. Селектор должен определять размещение в глобальной таблице GDT дескриптора используемой локальной таблицы, и этот дескриптор, в свою очередь, содержит уже всю информацию о LDT: базовый адрес, предел и другие характеристики. Процессор автоматически считывает назначенный дескриптор из глобальной таблицы и помещает его в свой «теневой регистр», ассоциируемый с регистром LDTR. С этого момента локальная таблица идентифицируется, распознается процессором, и ее можно использовать для решения текущей задачи: открывать локальные сегменты, описываемые дескрипторами задачи, занося их селекторы в сегментные регистры.

Загрузка в LDTR пустого селектора, для которого поле Index = 0, допустима. Такая операция сообщает процессору о том, что в задаче не будет использоваться локальная дескрипторная таблица. Это характерно для небольших однопользовательских систем.

В системе команд процессоров IA-32 есть всего шесть (привилегированных; см. п.6.3) команд, оперирующих регистрами дескрипторных таблиц: lgdt и lidt – для загрузки регистров GDTR и IDTR; sgdt и sidt – для сохранения этих регистров в памяти; lldt и sldt – для загрузки селектора в регистр LDTR и сохранения селектора из LDTR.

Мы пояснили, что селектор в защищенном режиме (его индекс Index вместе с индикатором TI) указывает на дескриптор в дескрипторной таблице, в котором находится начальный адрес сегмента. Селектор, стало быть, служит косвенным адресом сегмента, т.е. задает не сам адрес, а где его взять, из какой ячейки дескрипторной таблицы извлечь. Таким образом, *селектор в защищенном режиме является не прямым (как в реальном режиме), а косвенным адресом сегмента*.

Обращаясь к примеру выше, но отнесенному на этот раз к защищенному режиму:

```
mov AX, 73h  ; занести новый селектор 73h
mov DS, AX   ; в сегментный регистр DS через регистр AX
```

заключаем, что вторая команда (mov DS, AX) заносит селектор (73h) из аккумулятора AX в сегментный регистр DS; и она же извлекает дескриптор, указываемый селектором 73h, из дескрипторной таблицы, помещая его в дескрипторный регистр для DS.

Селектор (его Index – старшие 13 бит) определяет номер элемента (индекс) таблицы, бит 2 (индикатор TI) определяет саму таблицу (см. рис.2.11), а младшие 2 бита указывают допустимый уровень защиты сегмента RPL (см. п.6.3). В нашем примере селектор равен 73h=0000º0000º0111º0011b. (Здесь b, binary – указатель двоичной системы счисления.) Его старшие 13 бит есть 0000º0000º0111º0 = 1110 = 14. Значит, извлекается 1110b=14-й элемент (деск-

риптор) из дескрипторной таблицы. Поскольку TI = 0 (бит 2), то используется глобальная дескрипторная таблица GDT. Таким образом, открываемый сегмент памяти описывается в нашем примере 14-м дескриптором из таблицы GDT.

Общий формат команды загрузки селектора в сегментные регистры данных такой:

mov sr, r/m

где sr – сегментный регистр (segment register), r/m – общий регистр (register) или ячейка памяти (memory). Согласно сказанному эта команда заносит сначала новое значение селектора (из r или m) в сегментный регистр (sr), а затем извлекает назначенный этим селектором дескриптор из таблицы и помещает его в дескрипторный регистр. Следовательно, действие команды mov DS, AX в защищенном режиме можно описать выражениями

DS ← AX = 73h = селектор,

DS dr ← *m*(линейный адрес GDT + 14×8) = дескриптор.

Здесь:

DS dr – дескрипторный регистр для DS;

GDT – глобальная дескрипторная таблица (поскольку TI = 0), линейный адрес которой хранится в регистре GDTR (см. рис.2.5);

число 14×8 представляет собой относительный адрес 14-го элемента таблицы (дескриптора), поскольку длина дескрипторов составляет 8 байт;

выражение *m*(x) обозначает содержимое ячейки памяти (memory) с адресом x. В данном случае x = линейный адрес GDT + 14×8, а *m*(x) – 64-битный дескриптор, расположенный по этому адресу.

Определение линейного адреса памяти в защищенном режиме происходит, как и в реальном, путем сложения базы и смещения (см. рис. 2.11):

Line address = base + offset.

База (base) и смещение (offset) здесь 32-битные. База берется из дескриптора в дескрипторном регистре. Смещение задается общим регистром процессора или числовой константой в команде или их сочетанием; или регистром EIP для кодового сегмента. Их сумма определяет 32-битный линейный адрес памяти. Возможный перенос (из старшего разряда 31 в несуществующий разряд 32) теряется, так что после ячейки с максимальным адресом FFFFFFFF будет логически располагаться ячейка 00000000. Адресное пространство памяти замкнуто в кольцо (как и в реальном режиме).

Выбор процессором сегментного регистра для образования адреса. После установки сегмента (посредством загрузки его селектора в сегментный регистр) программа должна задавать только смещения. При задании одного смещения простые правила определяют, какой сегментный регистр необходимо привлекать для образования адреса. Эти правила приведены в табл. 2.4.

Согласно табл. 2.4 команды всегда выбираются из текущего сегмента кода: базовый адрес определяется регистром CS, а смещение находится в регистре EIP/IP. Стековые команды всегда обращаются к текущему стековому сегменту: базовый адрес в SS, смещение в ESP/SP. Для переменных принимается по умолчанию основной сегмент данных (база по DS), т.е. явная спецификация

этого сегмента необязательна. Однако программист может заставить МП обращаться к переменной, находящейся в другом сегменте. Для этого он использует явную спецификацию сегмента с помощью так называемого *префикса замены сегмента*.

Таблица 2.4 – Источники логического адреса

Тип обращения к памяти	База по умолчанию	Вариант	Смещение
Выборка команды	CS	Нет	EIP/IP
Стековые операции	SS	Нет	ESP/SP
Переменная	DS	CS, SS, ES	EA*
Цепочка-источник	DS	CS, SS, ES	ESI/SI
Цепочка-получатель	ES	Нет	EDI/DI
BP как базовый регистр	SS	CS, SS, ES	EA*
* EA (Effective Address) – так называемый *эффективный* или *исполнительный адрес*, т.е. смещение переменной, вычисляемое операционным устройством в соответствии с определенным в команде режимом адресации. При вычислении EA перенос из старшего бита игнорируется.			

На языке ассемблера префиксы сегментов записываются именами сегментных регистров, после которых ставится двоеточие:

CS: SS: DS: ES: FS: GS:

и эти префиксы надо помещать перед адресом/именем переменной.

Например, команда mov AL, ES: [SI] (префикс ES: помещен перед косвенным адресом [SI]) прочитает в регистр AL переменную из дополнительного сегмента данных (ES). А такая же команда mov AL, [SI] , но без префикса, прочитает в AL переменную, находящуюся по умолчанию в основном сегменте данных (DS).

При трансляции программы префиксы переводятся в однобайтные коды, размещаемые перед командой.

Для цепочек-источников в цепочечных командах применяется сегмент DS, а для цепочек-приемников – сегмент ES. И это правило тоже можно изменить с помощью префиксов сегментов.

Когда в команде в качестве источника внутрисегментного адреса указывается регистр BP, переменная по умолчанию находится в стековом сегменте. Однако с помощью сегментного префикса этот же регистр можно использовать для обращений к данным в любом другом из текущих сегментов.

Просмотр селекторов и дескрипторов сегментов. Прикладной программист, работающий на языке ассемблера, не может в условиях защищенного режима прямо адресовать физическую память. Любое обращение к памяти (коду, стеку или данным) производится относительно базовых адресов сегментов, а они больше не хранятся в сегментных регистрах, как в реальном режиме (или в процессоре 8086). Программисту остается только одно – обращаться к памяти по символическим адресам и загружать в сегментные регистры только их. Чтобы задавать физические адреса, необходимо решать сложную задачу, требующую знания тонкостей организации таблиц GDT и LDT.

В связи с этим вместе с процессорами IA-32 для их программной поддержки разработаны новые компиляторы, ассемблеры, редакторы связей и загрузчики. При редактировании связей (компоновке) символическим адресам назначаются действительные адреса памяти. Окончательную привязку адресов осуществляет загрузчик. Прикладному программисту не нужно теперь вникать в детали разбивки памяти на сегменты и знать их базовые адреса. Тонкости манипуляций селекторами, дескрипторами и дескрипторными таблицами оставлены разработчикам операционных систем и системным программистам.

При выполнении программы можно просмотреть фактические значения селекторов, которые редактор связей или загрузчик назначил сегментам. По селектору можно узнать, какую дескрипторную таблицу (бит TI) и какой дескриптор сегмента (поле IIndex) он выбирает. Имея права доступа к дескрипторным таблицам, можно даже считать и декодировать дескриптор. [9] Эти возможности поддерживаются с помощью машинных команд, приведенных в п. Б.5.

Состояние сегментных и дескрипторных регистров после пуска приведено в табл. 2.3 (п. 2.3).

2.5. Регистр флагов

Регистр флагов EFlags 32-битный и содержит 17 флажков (один из которых 2-битный и определяет уровень привилегий PL – privilege level, а остальные флажки 1-битные). Младшая половина (Flags) соответствует флаговому регистру 8086. В реальном режиме используется именно эта половина. Она имеет следующий формат:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
X	X	X	X	OF	DF	IF	TF	SF	ZF	X	AF	X	PF	X	CF

Здесь X обозначает биты, которые не используются.
Примечания:
команда popf – читает регистр Flags в AX;
имя EFlags/Flags не отражено в Ассемблере.

Флажки регистра Flags делятся на две группы: *арифметические и управляющие*. Дополнительные флажки EFlags мы изучать не будем.

Арифметические флажки:

CF (Carry flag) – флаг (признак) переноса;
PF (Parity flag) – флаг (признак) чётности;
AF (Auxiliary flag) – флаг (признак) вспомогательного переноса;
ZF (Zero flag) – флаг нуля;
SF (Sign flag) – флаг знака;
OF (Overflow flag) – флаг арифметического переполнения.

Управляющие флажки:

TF (Trace flag) – флаг прослеживания, трассировки;
IF (Interruption flag) – флаг разрешения прерываний;

DF (Direction flag) – флаг направления.

Арифметические флажки реагируют определенным образом на результат выполнения каждой арифметической и логической команды (операции). После выполнения каждой такой команды они устанавливаются в «1» или сбрасываются в «0» или остаются без изменения в зависимости от результата операции и типа команды. Другие команды не влияют на флажки (за исключением некоторых, таких как сдвиговые команды через флажок переноса; или команды выталкивания флажков из стека). Иначе, арифметические флажки хранят информацию, характеризующую результат выполнения последней команды. В свою очередь значения этих флажков проверяются командами условных переходов, и команда реализует или нет переход (разветвление в программе) в зависимости от значения флажка. Принцип передачи информации через эти флажки командам разветвления демонстрируется диаграммой на рис.2.12.

Рис. 2.12. Назначение арифметических флажков

Кроме того, флажок CF необходим для осуществления арифметических операций над двоичными одно- и многобайтными числами, а флажок AF – над двоично-кодированными десятичными числами.

Результаты выполнения арифметических и логических команд отражаются на состоянии арифметических флажков следующим образом.

Флажок CF. Реагирует на перенос при сложении и на заём при вычитании.

А именно, если при сложении чисел (происходящем по командам сложения add и др.) произошел перенос из старшего бита результата в следующий по старшинству, несуществующий бит, то признак переноса CF устанавливается в состояние «1»: CF←1. В противном случае, если переноса не возникло, признак CF сбрасывается в ноль: CF←0. Иными словами, флажок CF служит тем самым «несуществующим» битом при сложении, в который передается 1 при переносе.

При вычитании чисел (происходящем по командам вычитания sub и др.) бит CF сигнализирует не о переносе, а о заеме из несуществующего в старший бит уменьшаемого, т.е. устанавливается в этом случае в «1»: CF←1. Если же заема не было, признак CF сбрасывается в «0»: CF←0. При вычитании флажок CF служит, стало быть, признаком заема Br (Borrow).

В зависимости от размера операндов старшим битом может быть бит d7 (8-битные байты), d15 (16-битные слова) или d31 (32-битные двойные слова).

Например, по команде

add *x*, *y*,

где *x* и *y* – 8-битные операнды, сумма *x*+*y* заносится в 9-битный получатель CF_*x*:

CF_*x*←*x*+*y*,

т.е. 8 младших бит суммы заносится на место операнда *x*, служащего аккумулятором – приемником, накопителем результата, а 9-й бит – в CF (благодаря чему CF показывает наличие переноса). Выражение CF_*x* обозначает конкатенацию (сочленение) CF и аккумулятора *x*, т.е. число $CF \cdot 2^8 + x$.

Соответственно, по команде

sub *x*, *y*,

где *x* и *y* – 8-битные операнды, производится вычитание:

$CF_x \leftarrow 1_x - y$, CF← ¬CF,

где 1_x – конкатенация 1 и x, т.е. число $2^8 + x$, и y вычитается на самом деле не из x, а из $2^8 + x$. В данном случае единица «1», приписанная слева к x (эквивалентная прибавлению к x числа 2^8), нужна для заема, если он понадобится. После завершающего инвертирования CF показывает, был ли осуществлен этот заем.

Резюмируя, флаг переноса можно назвать признаком переполнения в арифметических операциях сложения и вычитания натуральных чисел. Он сигнализирует о выходе результата операции за диапазон от 0 до $2^n - 1$, где n – длина разрядной сетки* в битах для представления операндов и результата операции. При сложении он показывает, что сумма больше $2^n - 1$, а при вычитании – что разность меньше нуля, т.е. что уменьшаемое меньше вычитаемого.

* *Разрядная сетка* ЭВМ – это упорядоченный набор двоичных разрядов, используемых в ЭВМ для представления чисел.

Флажок CF фиксирует также значение выдвигаемого бита при сдвиге операнда с помощью команд сдвига.

Флажок PF. Индицирует четность результата.

При «четном» результате в младшем байте аккумулятора (получателя) (наличии четного числа единиц в младших 8 битах результата), флаг PF устанавливается в «1»: PF := 1. При нечетном числе единиц в младшем байте результата флаг PF сбрасывается: PF := 0. Подчеркнем, что на состояние этого флажка влияет только младший байт результата. Этот флаг предназначен для контроля правильности передач данных («контроля четности»).

Флажок AF. Индицирует наличие вспомогательного переноса.

Если произошел вспомогательный перенос – перенос из младшей тетрады (полубайта) в старшую, т.е. из разряда d3 в разряд d4 результата, то флаг AF устанавливается в единицу. Иначе он обнуляется: AF := 0. Это происходит как в командах сложения, так и вычитания, так как вычитание реализуется путем сложения уменьшаемого с дополнением вычитаемого с последующим инвертированием флага переноса CF (см. п. 2.6). На указанное сложение и реагирует AF. Флаг AF проверяется только командами десятичной коррекции.

Флажок ZF. Сигнализирует о нулевом результате операции.

В том случае, если результат операции равен нулю, флаг ZF устанавливается в «1»; если же хотя бы в одном бите результата имеется единица, ZF сбрасывается в ноль.

Флажок SF. Регистрирует знак результата.

Пусть получен отрицательный знак результата, т.е. произошла установка в «1» старшего, знакового бита результата: бита d7 в *байтных* операциях, бита d15 в *словных* операциях, или бита d31 в *двухсловных* операциях. Тогда устанавливается флаг SF: SF := 1. Если, наоборот, результат положительный (т.е. знаковый бит результата d7 или d15 или d31 обратился в нуль), то SF сбрасывается: SF := 0. Иными словами, бит SF просто копирует значение старшего бита результата, который при использовании дополнительного кода (п.2.6) соответствует знаку числа: SF := d7 (для байтов), d15 (для слов), d31 (для двойных слов).

Флажок OF. Сигнализирует об арифметическом переполнении при сложении и вычитании целых чисел со знаком, записанных в дополнительном коде. Если произошло арифметическое переполнение, то признак OF устанавливается в «1», если арифметического переполнения не было, OF сбрасывается в «0».

Арифметическим переполнением (целочисленным) называется *выход результата арифметической операции* (*сложения или вычитания целых чисел со знаком*) *за пределы диапазона чисел, представимых в данной разрядной сетке при помощи дополнительного кода*. Этот диапазон составляет:

для 8-битной разрядной сетки – от -2^7 до 2^7-1;

для 16-битной сетки – от -2^{15} до $2^{15}-1$;

для 32-битной сетки – от -2^{31} до $2^{31}-1$.

И вообще, для n-битной разрядной сетки допустимый диапазон знаковых целых чисел в дополнительном коде есть $-2^{n-1} \le x \le 2^{n-1}-1$ (см. п.2.6).

Коротко: *арифметическое переполнение* – это выход суммы или разности целых знаковых чисел из диапазона $-2^{n-1} \div 2^{n-1}-1$, где n – длина разрядной сетки в битах.

Примеры.

mov AL, 127; AL←127

add AL, 2; CF_AL←127+2. Здесь CF = 0, OF = 1, т.е. переноса нет, переполнение есть.

mov AL, 255; AL←255

add AL, 2; CF_AL←255+2, Здесь CF = 1, OF = 0, т.е. перенос есть, переполнения нет (см. п.2.6).

Сравнивая флажки CF и OF, заключаем, что CF *сигнализирует о переполнении при сложении и вычитании в натуральной арифметике*, а OF – *в знаковой*.

При выполнении команд умножения натуральных (по команде mul) и знаковых (по команде imul) чисел арифметического переполнения никогда не происходит, так как под произведение сразу отводится достаточно длинный аккумулятор, равный удвоенной длине сомножителей. Поэтому произведение всегда вмещается в аккумулятор. Но флажки CF и OF реагируют на обе эти ко-

манды, показывая, что результат вышел за пределы младшей половины произведения и содержит старшую половину.

При делении (по командам div и idiv) переполнение возможно (в смысле невмещаемости частного от деления в аккумулятор). Оно называется «ошибкой деления» или «делением на нуль» и вызывает внутреннее прерывание с номером $n = 0$ (гл.5). Содержимое арифметических флагов после выполнения обеих этих команд неопределенно.

Управляющие флажки предназначены для управления некоторыми режимами работы микропроцессора (рис. 2.13).

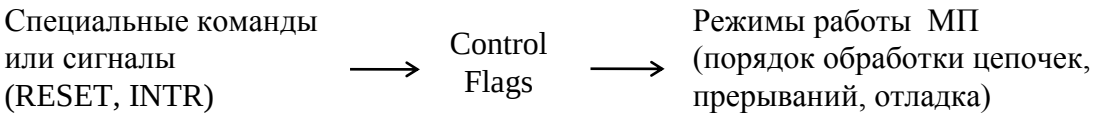


Рис. 2.13. Назначение управляющих флажков

Отдельные управляющие флажки выполняют следующие функции.

Флажок направления DF определяет порядок сканирования (направления обработки) цепочек (строк) данных в цепочечных командах: от меньших адресов к бóльшим (DF = 0) или наоборот (DF = 1). Установка флажка (в «1») производится командой STD (Set DF), сброс (в «0») – командой CLD (Clear DF).

Флажок прерывания IF управляет прохождением запросов в процессор по входу INTR (рис. 2.14). Если IF = 0, запросы не проходят в процессор, игнорируются им («прерывания запрещены»). Если IF = 1, то прерывания разрешены, сигнал INTR проходит в процессор, распознается и соответствующим образом обрабатывается (п. 5.3).

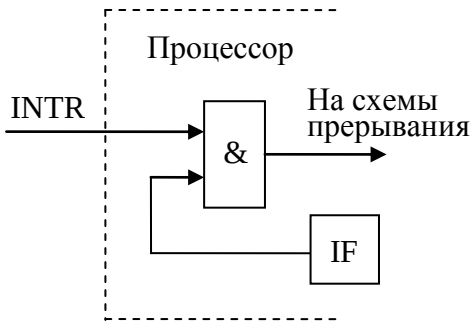


Рис. 2.14. Прохождение
запроса прерывания в процессор

Флажок IF устанавливается в «1» командой sti, а сбрасывается в ноль командой cli. Он сбрасывается также в «0» автоматически при запуске процессора (сигналом RESET) и всех видах прерываний (рис.2.15 и п.5.3). Поэтому сразу после пуска или прерывания по входу INTR дальнейшие прерывания по этому входу запрещены.

Следует иметь в виду, что команда sti разрешает прерывание только после завершения следующей за ней команды. Подобный порядок позволяет процессору вслед за командой sti выполнить команду возврата из прерывания iret (в частности, осуществить возврат из вложенных прерываний) и только потом реагировать на запрос INTR.

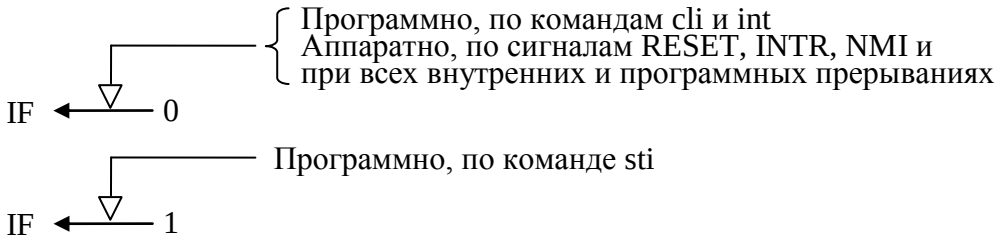


Рис. 2.15. Воздействие на флажок разрешения прерывания

Состояние флажка IF не влияет на восприятие других видов прерываний (внешних немаскируемых по входу NMI, внутренних и программных); оно влияет только на восприятие сигнала INTR.

Флажок трассировки TF. Установка в «1» флажка прослеживания (трассировки) TF переводит МП в одношаговый (однокомандный) режим работы, который применяется при отладке программ. В этом режиме МП автоматически генерирует внутреннее прерывание (с номером $n=1$) после выполнения каждой команды и переходит к соответствующей подпрограмме, которая, например, может просматривать содержимое внутренних регистров.

В процессорах Intel нет специальной команды для изменения флажка TF. Модификация TF производится только путем включения всех флажков в стек, изменения в них бита 8 и последующего извлечения из стека обратно в регистр EFlags/Flags.

Обозначение (имя) регистра флажков Flags не встроено в Ассемблер и не может использоваться в текстах программ для обращения к этому регистру. Обработка регистра Flags и его некоторых отдельных битов производится с помощью специальных команд: pushf – втолкнуть в стек содержимое Flags, popf – вытолкнуть из стека Flags и указанных выше команд модификации управляющих флажков. Эти команды адресуют регистр Flags не по имени, а неявно (самим кодом операции).

Остальные флажки расширенного регистра EFlags расположены в битах 31-16 регистра EFlags (вне Flags). Они используются системными программами и в настоящем пособии не рассматриваются.

2.6. Арифметические операции над двоичными числами

Целочисленные арифметические операции и команды. Процессор IA-32 может выполнять целочисленные операции и операции с плавающей точкой. Для этого в его архитектуре есть два отдельных блока:

устройство для выполнения целочисленных операций;

устройство для выполнения операций с плавающей точкой.

Каждое из этих устройств имеет свою систему команд. В принципе, целочисленное устройство может взять на себя многие функции устройства с плавающей точкой, но это потребует больших вычислительных затрат. Устройство с плавающей точкой мы рассматривать не будем. Поясним лишь,

что оно выполняет арифметические операции над числами, хранящимися в регистрах r0-r7 в формате, показанном на рис. 2.16, а названия всех его команд на языке ассемблера начинаются с буквы F (Floating): fadd, fsub, fmul, fdiv и т.д.

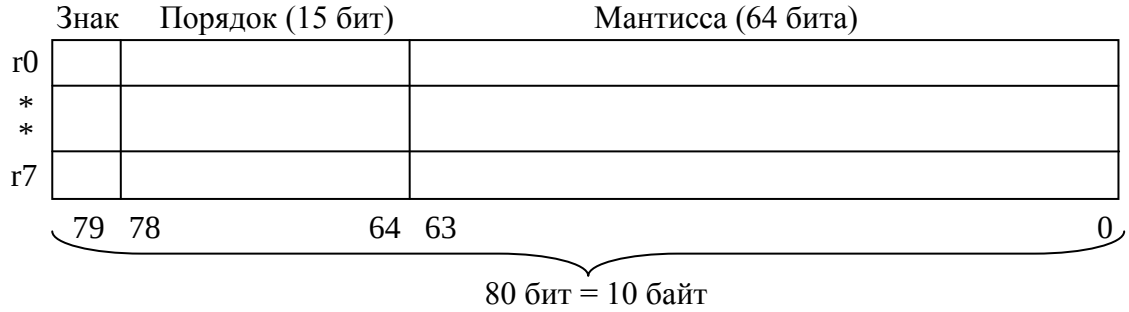


Рис. 2.16. Форматы двоичных чисел с плавающей запятой

Для большинства задач, использующих язык ассемблера, достаточно целочисленной арифметики.

Целочисленное вычислительное устройство поддерживает чуть больше десятка арифметических команд:

Преобразования типов: cbw, cwd, cwde, cdq, movsx, movzx.

Двоичной арифметики: сложения: add, adc, inc; вычитания: sub, sbb, dec; умножения: imul, mul; деления: idiv, div; изменения знака: neg.

Десятичной арифметики: коррекции сложения: aaa, daa; коррекции вычитания: aas, das; коррекции умножения: aam; коррекции деления: aad.

Вспомогательные команды: bswap.

Прочие команды с арифметическим принципом действия: cmp, cmpxchg, setcc, xadd.

Целые двоичные числа. *Целое двоичное число* – это целое число, закодированное в двоичной системе счисления. Целое число называют также числом с фиксированной точкой (или запятой) (мысленно помещаемой обычно правее младшего разряда числа). Размерность целого двоичного числа, обрабатываемого одной командой процессора IA-32, может составлять 8, 16 или 32 бита.

Как описать числа с фиксированной точкой в программе? Это делается с помощью директив описания данных db, dw и dd. При этом отдельные байты директив dw и dd размещаются в памяти по принципу «младший байт по младшему адресу».

Листинг 1. Примеры описания целых чисел

```
Var1  db  23      ; переменная Var1 равна 23=17h=00010111b
Var2  db  101b  ; Var2=5
Var3  dw  517   ; Var3=517=0205h: байт по адресу Var3 содержит
        ; 05h = 00000101b, байт по адресу Var3+1 содержит 02h=00000010b
        ; Эту же переменную Var3 = 517=0205h можно описать и по байтам:
Var3  db  05h
        db  02h
Var4  dd  9875645  ; Var4=9875645
```

Форматы целых двоичных чисел. Для представления целых чисел в компьютерах применяют натуральный, прямой, обратный и дополнительный коды. Обратный код не поддерживается аппаратно, на уровне команд процессора, принципиально мало отличается от дополнительного, и мы его рассматривать не будем. Прямой код тоже не встроен в какие-либо машинные команды, реализующие знаковую арифметику. Однако с его помощью удобно умножать и делить числа программно [11]. Кроме того, он применяется в цифровых приборах, например, в аналого-цифровых преобразователях, схема управления которых формирует код знака, а результат преобразования выражается числом со знаком в прямом коде.

Двоичные коды целых чисел для случая однобайтной разрядной сетки показаны на рис. 2.17.

Натуральный код – это просто двоичная запись целого неотрицательного числа без знака – *натурального числа* на всю ширину разрядной сетки (см. рис. 2.17). В математике натуральными называются числа 1, 2, 3… . В информатике к ним причисляют еще и число 0.

Максимальный диапазон чисел, представимых 8-битным натуральным кодом, составляет $0 \div 2^8 - 1$, или $0 \div 255$.

Прямой код служит для записи чисел со знаком. Для обозначения знака отводят старший разряд (разряд 7 для 8-разрядной сетки; см. рис. 2.17). В него заносят 0, если число неотрицательное, и 1 – если отрицательное. В остальные, цифровые, разряды (разряды 6-0) помещают само число – его модуль в двоичном представлении. Число нуль можно записать и со знаком «минус» – с 1 в знаковом разряде (так называемый «отрицательный» нуль). Соответственно нуль со знаком «плюс» (0 в знаковом разряде) – это «положительный» нуль.

Максимальный диапазон чисел в 8-битном прямом коде занимает пределы от $-(2^7 - 1) = -127$ до $2^7 - 1 = 127$.

Дополнительный код (англ. two's complement, иногда twos-complement) – наиболее распространённый способ представления знаковых чисел в компьютерах. Он позволяет заменить операцию вычитания на операцию сложения и сделать операции сложения и вычитания одинаковыми для знаковых и беззнаковых чисел, чем упрощает архитектуру ЭВМ.

n-битный дополнительный код отрицательного числа можно получить вычитанием модуля числа из нуля с заемом единицы из несуществующего, воображаемого, $n+1$-го разряда или, что эквивалентно, вычитанием модуля из 2^n: так называемое «дополнение до 2^n» (отсюда название этого кода: «дополнительный»).

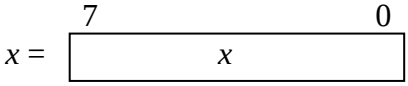
Определение 1: полное определение дополнительного кода. Дадим его, без ограничения общности, для случая 8-битной разрядной сетки: $n = 8$. Именно: дополнительный 8-битный код отрицательного числа $x < 0$ содержит в знаковом бите (бите 7) единицу, а в цифровых битах (6-0) – дополнение модуля числа $|x|$ до 2^7, т.е. разность $2^7 - |x|$, которая обязана оставаться неотрицательной ($2^7 - |x| \geq 0$):

$$x_{\text{доп}} = 1_(2^7 - |x|) \qquad (2.1)$$

(см. рис. 2.17).

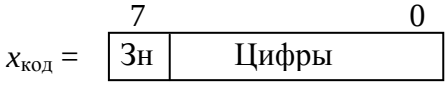
Форматы целых двоичных чисел

Целое без знака *x* (так называемый натуральный код числа):



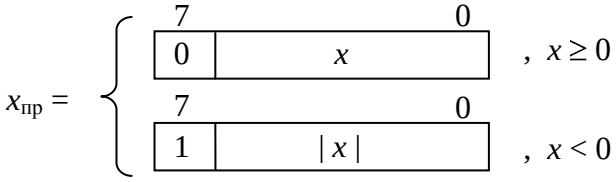
все разряды 7 – 0 цифровые,
диапазон представления чисел
$0 \le x \le 2^8\text{-}1 = 255$

Целое со знаком *x*:



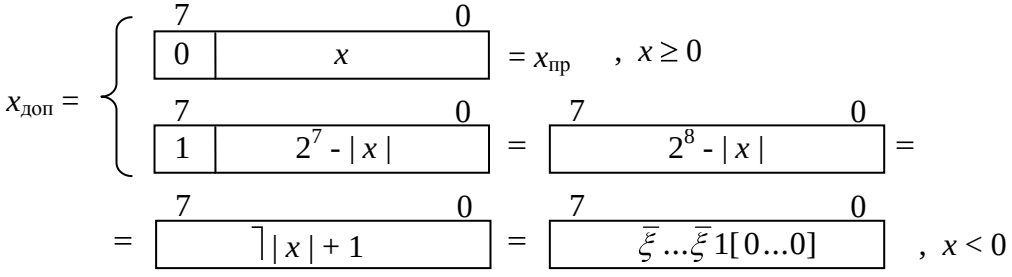
разряд 7 знаковый: 0, если $x \ge 0$, и 1, если $x < 0$; разряды 6 – 0 цифровые

Целое со знаком *x* в прямом коде:



$-127 = -(2^7\text{-}1) \le x \le 2^7\text{-}1 = 127$

Целое со знаком *x* в дополнительном коде:



где 2^7 - | *x* | – дополнение модуля *x* до 2^7; 2^8 - | *x* | – дополнение модуля *x* до 2^8; ⌉ обозначает поразрядную инверсию числа, например: ⌉°1101°0010 = 0010°1101; $\bar{\xi}\ldots\bar{\xi}1[0\ldots0]$ – побитная инверсия числа $|x| = \xi\ldots\xi1[0\ldots0]$, не затрагивающая самой младшей единицы и всех нулей справа от нее (здесь [0…0] – необязательные нули).

$-128 = -2^7 \le x \le 2^7\text{-}1 = 127$

Рис. 2.17. Форматы целых двоичных чисел
(на примере 8-битной разрядной сетки)

Выражение $1_(2^7 - |x|)$ обозначает конкатенацию* 1 (единицы) и 7-битного двоичного числа $2^7 - |x|$, т.е. 8-битное число, образованное путем сочленения цифровых записей этих чисел в одно число.

* (англ. concatenate – сцеплять) – операция сочленения, склеивания объектов линейной структуры, обычно строк, добавление одной строки в конец другой. Например, конкатенация слов «микро» и «мир» дает слово «микромир». Конкатенация чисел 5 и 2 дает число 52.

Поскольку в (1) $2^7 - |x| \geq 0$, или $|x| \leq 2^7$, то закодированное по формуле (1) отрицательное число находится в диапазоне $-2^7 \leq x < 0$.

Знаковую единицу (в бите 7) можно трактовать как число 2^7. Поэтому полный 8 битный дополнительный код отрицательного числа можно получить вычитанием модуля числа из $2^7 + 2^7 = 2^8$ (дополнением до 2^8):

$$x_{\text{доп}} = 2^8 - |x| = \text{ДОП}(|x|), \qquad (2.2)$$

где по-прежнему $-2^7 \leq x < 0$, а выражение ДОП($|x|$) обозначает дополнение модуля числа до 2^8, т.е. разность $2^8 - |x|$.

Если число неотрицательное: $x \geq 0$, то его дополнительный код принимается по определению равным прямому коду: в знаковом бите (7) записывается нуль, а в цифровых (6-0) – само число в двоичном представлении:

$$x_{\text{доп}} = x.$$

x в этом случае лежит в диапазоне $0 \leq x \leq 2^7 - 1$. Иными словами, специального дополнительного кода для положительных чисел нет, а в качестве него используется прямой код. По этому поводу говорят иногда, что положительные числа записываются не в дополнительном, а в прямом коде. Тем не менее, для единства терминологии мы будем придерживаться нашего определения, т.е. говорить о дополнительном коде, как отрицательных, так и положительных чисел, понимая под последним просто прямой код.

Согласно сказанному полный диапазон чисел, представимых 8-битным дополнительным кодом, составляет от $-2^7 = -128$ до $2^7 - 1 = 127$.

В общем случае *n*-битной разрядной сетки всюду в приведенных здесь и ниже выражениях 8 разрядов надо заменить на n разрядов, а 7 на $n-1$. Числа могут лежать соответственно в пределах от -2^{n-1} до $2^{n-1} - 1$.

Для формул (1) и (2) получим более простые представления, не содержащие операции вычитания. Для этого вернемся сначала к понятию дополнения и дадим ему общее определение (дефиницию).

Определение 2. Операция вычисления «дополнения» может применяться к любому коду. Пусть s – некоторый 8-разрядный двоичный код. Тогда дополнение s до 2^8 по определению есть 8-битная разность

$$\text{ДОП}(s) = 2^8 - s, \qquad (2.3)$$

где код s при вычитании интерпретируется как натуральное число (число без знака).

При этом дополнение нуля ($s = 0$) равно нулю:

$$\text{ДОП}(0) = 0,$$

так как $2^8 - 0 = 1°0000°0000$, но при вычислении 9-й по счету бит (бит 8), содержащий 1, отбрасывается (сетка 8-битная!). В результате $\text{ДОП}(0) = 0000°0000$.

Простые способы вычисления дополнения (теорема).

Способ 1. *Дополнение легко вычисляется путем побитной инверсии кода с последующим прибавлением* 1:

$$\text{ДОП}(s) = 2^8 - s = \neg s + 1, \qquad (2.4)$$

где символ $\neg$ обозначает инверсию всех битов числа s. Например, если $s = 1001°1000$, то $\neg s = \neg°1001°1000 = 0110°0111$.

Действительно, $2^8 - s = 2^8 - 1 - s + 1 = 11111111 - s_7 s_6 s_5 s_4 s_3 s_2 s_1 s_0 + 1 =$
$= \bar{s}_7 \bar{s}_6 \bar{s}_5 \bar{s}_4 \bar{s}_3 \bar{s}_2 \bar{s}_1 \bar{s}_0 + 1 = \neg s + 1$.

Отдельно для $s = 0$ по формуле (4) имеем $\text{ДОП}(0) = 0$. В самом деле, $\neg 0000°0000 + 1 = 1111°1111 + 1 = 1°0000°0000$, и $\text{ДОП}(0) = 0000°0000$, так как единица из 9-го бита отбрасывается (см. пояснение к (3)).

Способ 2 (еще один простой способ вычисления дополнения). Из (4) вытекает, что поскольку любой ненулевой код ($s \neq 0$) всегда имеет один из двух видов $s = \sigma...\sigma 10...0$ или $s = \sigma...\sigma 1$, то его дополнение можно найти так:

$$\text{ДОП}(s) = \bar{\sigma}...\bar{\sigma}10...0 \text{ или } \text{ДОП}(s) = \bar{\sigma}...\bar{\sigma}1, \qquad (2.5)$$

где $\bar{\sigma} = \neg\sigma$ – инверсия бита σ.

В самом деле, $\text{ДОП}(s) = \neg\sigma...\sigma 10...0 + 1 = \bar{\sigma}...\bar{\sigma}01...1 + 1 = \bar{\sigma}...\bar{\sigma}10...0$,

т.е. *дополнение можно получить путем инверсии всех битов кода, находящихся слева от самой младшей единицы*.

Из (5) опять-таки имеем: $\text{ДОП}(0) = 0$, так как инвертировать (слева от единицы) нечего (поскольку нет самой единицы).

Мы показали, что согласно формулам (3)-(5) дополнение 8-битного кода s до 2^8 можно получить любым из 3-х следующих способов:

$$\text{ДОП}(s) = 2^8 - s = \neg s + 1 = \bar{\sigma}...\bar{\sigma}1[0...0], \qquad (2.6)$$

где $s = \sigma...\sigma 1[0...0]$, $[0...0]$ – необязательные нули,

т.е., дополнение можно найти,

вычитая код s из 2^8; или

инвертируя код побитно с последующим прибавлением 1; или

инвертируя все биты кода, расположенные слева от младшей единицы.

Согласно формулам (1), (2) и (6) получаем

четыре эквивалентных определения дополнительного кода отрицательного числа $-2^7 \le x < 0$ (см. рис. 2.16):

$$x_{\text{доп}} = 1_(2^7 - |x|) = 2^8 - |x| = \neg|x| + 1 = \bar{\xi}...\bar{\xi}1[0...0], \qquad (2.7)$$

где $|x| = \xi...\xi 1[0...0]$.

Таким образом, для нахождения дополнительного кода отрицательного числа надо:

в знаковый бит (7) *записать* 1, *а в цифровые* (6-0) – *разность* $2^7 - |x|$; или

во все восемь бит (7-0) *записать разность* $2^8 - |x|$; или

инвертировать (*8-битный*) *модуль числа побитно и прибавить единицу*; или

инвертировать все биты (*8-битного*) *модуля числа, находящиеся слева от младшей единицы*.

Таблица представлений целых знаковых чисел в прямом и дополнительном кодах.

Десятичное представление	Двоичное представление (8 бит)	
	прямой	дополнительный
127	01111111	01111111
5	00000101	00000101
4	00000100	00000100
3	00000011	00000011
2	00000010	00000010
1	00000001	00000001
0	00000000	00000000
-0	10000000	- - -
-1	10000001	11111111
-2	10000010	11111110
-3	10000011	11111101
-4	10000100	11111100
-5	10000101	11111011
-126	11111110	10000010
-127	11111111	10000001
-128	- - -	10000000

Операция расширения знака. При необходимости увеличения разрядности числа (например, с 8 до 16 разрядов) выполняется операция расширения знака. *Расширение знака* (англ. Sign extension) – это операция над двоичным числом, которая позволяет увеличить разрядность числа с сохранением знака и значения. Выполняется добавлением цифр со стороны старшего значащего разряда. Применительно к дополнительному коду, добавляются нули, если число положительное (старший разряд равен 0), и единицы, если число отрицательное (старший разряд равен 1). Например:

Десятичное число	Двоичное число в дополнительном коде (8 разрядов)	Двоичное число в дополнительном коде (16 разрядов)
10	0000 1010	0000 0000 0000 1010
-15	1111 0001	1111 1111 1111 0001

Арифметические операции над целыми двоичными числами. Процессор выполняет все целочисленные арифметические операции (команды) над двоичными кодами операндов как над целыми числами без знака (натуральными числами). Это означает, что он обрабатывает знаковый разряд чисел (для 8-битных чисел – это разряд 7, для 16-битных – 15, для 32-битных –

31) как и остальные, цифровые разряды (6-0, 14-0, 30-0), т.е. трактует этот разряд (7, 15, 31) как цифровой, а не знаковый. Например, если в регистре AL находится число -5 в прямом коде, т.е. 1000°0101, а в регистре AH – число 6, т.е. 0000°0110, то команда add AL,AH сложит эти числа так:

1000°0101
+
0000°0110
=
1000°1011,

т.е. даст в ответе натуральное число 139 (1000°1011). Это происходит потому, что процессор на самом деле складывает не числа -5 и 6, а числа 133 (двоичный код 1000°0101) и 6 (0000°0110) и получает 139 (1000°1011). Если же трактовать результат (1000°1011) как число со знаком, то в прямом коде это -11. Но это неправильный ответ. Правильный ответ должен был бы быть +1.

Мы видим, что когда числа со знаком записываются в прямом коде, их нельзя правильно сложить с помощью одной команды, поскольку процессор воспринимает их как натуральные, а не знаковые числа. Процессор вообще не подозревает о существовании отрицательных чисел и «думает», что все числа являются натуральными, числами без знака. Процессор просто складывает, вычитает, делит и умножает числовые коды как натуральные числа, какой бы смысл они ни несли, как бы ни трактовались человеком. Исключение (для процессоров Intel) составляют лишь две команды: умножения imul и деления idiv, по которым процессор оперирует именно со знаковыми числами, записанными в дополнительном коде. Правильный результат (произведение или частное) получается тоже в дополнительном коде.

Для сложения на компьютере двух чисел со знаком, заданных в **прямом коде**, надо поступать так, как это делает человек. Надо проанализировать сначала знаковые разряды чисел и оперировать далее с модулями чисел, принимая во внимание их знаки. Если знаки одинаковые, надо складывать модули чисел и приписывать сумме одинаковый знак чисел. Если знаки разные, надо из большего модуля вычитать меньший модуль и приписывать разности знак большего по модулю числа. Указанные шаги приводят к алгоритму с двумя ветвями (в зависимости от одинаковости или различия знаков), каждая из которых вновь разделяется на две ветви (по отдельности складывающие или вычитающие модули и определяющие знак конечного результата): всего 4 ветви в алгоритме. Подобный алгоритм требует привлечения нескольких команд (операций) процессора, т.е. выполнения небольшой программы.

Аналогичные соображения можно высказать и в отношении вычитания. Умножение и деление в прямых кодах менее громоздки, однако и они осуществляются с помощью нескольких команд.

В связи с громоздким характером сложения и вычитания знаковых целых в прямом коде эти операции целесообразно выполнять в **дополнительном коде**. Дополнительный код обладает полезным свойством (которое мы приводим без доказательства, хотя оно сразу вытекает из определения самого кода):

Свойство 1: сложение и вычитание чисел в дополнительном коде.

Сложение. Пусть x и y – целые числа со знаком, лежащие в диапазоне от -2^7 до 2^7-1, а $x_{\text{доп}}$ и $y_{\text{доп}}$ – их 8-битные дополнительные коды. Тогда при условии отсутствия арифметического переполнения (определение которого дано в п.2.5 и воспроизводится чуть ниже)

$$x_{\text{доп}} + y_{\text{доп}} = (x+y)_{\text{доп}}, \qquad (2.8)$$

где $x_{\text{доп}} + y_{\text{доп}}$ – 8-битная сумма дополнительных кодов, складываемых как натуральные числа, а возможный перенос из старшего, 8-го, бита (бита 7) в несуществующий 9-й бит (бит 8), если он возник, отбрасывается; $(x+y)_{\text{доп}}$ – правильный 8-битый дополнительный код суммы чисел.

Если же арифметическое переполнение имеет место, то для записи дополнительного кода суммы $(x+y)_{\text{доп}}$ требуется 9 бит. Тогда, при переполнении, правильный код суммы получится в 9-битной сумме кодов $x_{\text{доп}} + y_{\text{доп}}$, включающей бит 8, который в данном случае будет знаковым битом.

Коротко:

$\text{d8 d7 ... d0} \leftarrow x_{\text{доп}} + y_{\text{доп}}$.

В отсутствие переполнения

$(x+y)_{\text{доп}} = \text{d7 ... d0}$,

в присутствие переполнения

$(x+y)_{\text{доп}} = \text{d8 d7 ... d0}$.

Вычитание. Аналогичное свойство соблюдается и для разности двух целых знаковых чисел x и y. А именно, в отсутствие переполнения

$$x_{\text{доп}} - y_{\text{доп}} = (x-y)_{\text{доп}}, \qquad (2.9)$$

где $(x-y)_{\text{доп}}$ – правильный 8-битный дополнительный код разности чисел, и при вычислении разности $x_{\text{доп}} - y_{\text{доп}}$ в случае $x_{\text{доп}} < y_{\text{доп}}$ производится заем единицы («1») из несуществующего 9-го бита (бита 8).

В случае переполнения дополнительный код разности $(x-y)_{\text{доп}}$ должен быть 9-битным, и он получается расширением 8-битной разности $x_{\text{доп}} - y_{\text{доп}}$ слева нулем (0), если заем из бита 8 был, и единицей (1) – если не был. Можно сказать условно, что бит 8, расширяющий разность $x_{\text{доп}} - y_{\text{доп}}$, сигналит о «не заеме».

Коротко:

$\text{d8 d7 ... d0} \leftarrow 1_x_{\text{доп}} - y_{\text{доп}}$, где $1_x_{\text{доп}}$ – конкатенация 1 и $x_{\text{доп}}$ (именно так понимается операция $x_{\text{доп}} - y_{\text{доп}}$).

Тогда в отсутствие переполнения

$(x-y)_{\text{доп}} = \text{d7 ... d0}$,

в присутствие переполнения

$(x-y)_{\text{доп}} = \text{d8 d7 ... d0}$.

Таким образом,

Сумма/разность двух дополнительных кодов целых чисел со знаком (складываемых/вычитаемых как натуральные числа в пределах принятой разрядной сетки) равна правильному дополнительному коду их суммы/разности при условии, что не произошло арифметического переполнения.

Если же переполнение имело место, то правильный дополнительный код суммы/разности представляется суммой/разностью дополнительных кодов, расширенной битом переноса/«не заема».

Представленное свойство позволяет вместо того, чтобы складывать/вычитать знаковые числа, складывать/вычитать их дополнительные коды как натуральные числа – числа без знака.

Заметим, что применительно к операциям умножения и деления данное свойство дополнительных кодов не имеет места.

Другое полезное свойство дополнительного кода.

Свойство 2: изменение знака числа в дополнительном коде. Пусть x – целое число со знаком, находящееся в диапазоне $-2^7 \le x \le 2^7 - 1$, а $x_{\text{доп}}$ – его 8-битный дополнительный код. Тогда в отсутствие арифметического переполнения

$$(-x)_{\text{доп}} = \text{ДОП}(x_{\text{доп}}), \qquad (2.10)$$

где $\text{ДОП}(x_{\text{доп}})$ – 8-битное дополнение $x_{\text{доп}}$ до 2^8,

т.е. *дополнительный код числа $-x$ («минус» x) равен дополнению дополнительного кода $x_{\text{доп}}$ самого числа x, если нет арифметического переполнения*; или, иначе: *для изменения знака числа в дополнительном коде надо взять дополнение его дополнительного кода*.

Арифметическое переполнение, т.е. неправильный результат, наступает здесь в единственном случае $x = -128$, так как у противоположного числа $-x = +128$ нет 8-разрядного дополнительного кода. В этом случае правильный, 9-битный, код получается расширением 8-битного дополнения $\text{ДОП}(x_{\text{доп}})$ нулевым по значению битом 8:

$$\text{при } x = -128 \ (-x)_{\text{доп}} = 0_\text{ДОП}(x_{\text{доп}}) = 0°1000°0000. \qquad (2.11)$$

Кроме того, при $x = 0$ имеем: $x_{\text{доп}} = 0$, $(-x)_{\text{доп}} = (-0)_{\text{доп}} = \text{ДОП}(x_{\text{доп}}) =$ $= \text{ДОП}(0) = 2^8 - 0 = 2^8 = 1°0000°0000$, и далее надо брать в этом результате опять-таки только 8 младших разрядов, умещающихся в разрядную сетку, отбрасывая 9-й разряд, содержащий единицу: $(-0)_{\text{доп}} = 0000°0000$.

К свойству 2 (как и к свойству 1) легко прийти исходя из определения дополнительного кода (но этот приход мы тоже не доказываем).

Для вычисления дополнения $\text{ДОП}(x_{\text{доп}})$, а значит, для изменения знака числа, можно воспользоваться формулами (6). В результате получаем, что $(-x)_{\text{доп}}$ при $x \ne -128$ определяется любым из 3-х следующих способов:

$$(-x)_{\text{доп}} = \text{ДОП}(x_{\text{доп}}) = 2^8 - x_{\text{доп}} = \overline{x_{\text{доп}} + 1} = \overline{\xi}...\overline{\xi}1[0...0], \qquad (2.12)$$

где $x_{\text{доп}} = \xi...\xi 1[0...0]$.

Итак, *для изменения знака числа, записанного в дополнительном коде, надо:*

вычесть этот дополнительный код из 2^8; или

инвертировать этот код побитно и прибавить единицу; или

инвертировать все биты этого кода, находящиеся слева от младшей единицы,

за исключением числа $x = -2^7 = -128$, дающего переполнение в этой операции. В таком случае противоположное к -128 число, т.е. (знаковое) число 128 представляется 9-битным дополнительным кодом (11).

Свойство 3: Замена операции вычитания на операцию сложения. Свойства 1 и 2 дополнительного кода *позволяют производить вычитание двух чисел путем сложения уменьшаемого с дополнением вычитаемого*:

$$x - y = x + (-y) = x + \text{ДОП}(y),$$

поскольку взятие дополнения эквивалентно изменению знака числа.

Строгая формулировка данного свойства звучит так. Пусть x и y – целые числа со знаком, а $x_{\text{доп}}$ и $y_{\text{доп}}$ – их дополнительные коды. Тогда в отсутствие арифметического переполнения имеем

$$(x-y)_{\text{доп}} = (x+(-y))_{\text{доп}} = x_{\text{доп}} + (-y)_{\text{доп}} = x_{\text{доп}} + \text{ДОП}(y_{\text{доп}})$$

– дополнительный код разности двух чисел равен сумме дополнительного кода уменьшаемого и дополнения дополнительного кода вычитаемого.

Такая операция дает правильный ответ – разность двух знаковых чисел x и y в 8-битном дополнительном коде, если не произошло арифметического переполнения.

Если же переполнение произошло, то правильный дополнительный код суммы является 9-битным, равным сумме кодов $x_{\text{доп}} + \text{ДОП}(y_{\text{доп}})$, в которой надо учесть бит номер 8.

Коротко:

$\text{d8 d7 ... d0} \leftarrow x_{\text{доп}} + \text{ДОП}(y_{\text{доп}})$.

В отсутствие переполнения

$(x-y)_{\text{доп}} = \text{d7 ... d0}$,

в присутствие переполнения

$(x-y)_{\text{доп}} = \text{d8 d7 ... d0}$.

Благодаря возможности заменить вычитание сложением в процессорах не делают устройств-вычитателей, а ограничиваются сумматорами и схемами побитной инверсии. А поскольку, к тому же, знаковые разряды дополнительных кодов обрабатываются подобно обычным, цифровым, достаточно иметь сумматоры только для натуральных чисел. Эти два обстоятельства существенно упрощают аппаратную реализацию арифметических блоков цифровых приборов.

Сложение натуральных и знаковых чисел с помощью команды add.

Формат команды на ассемблере:

add x, y

Здесь x – n-битная переменная; y – n-битная переменная или константа (непосредственный операнд); переменные x и y могут находиться в регистрах процессора, а одна из них – в ячейке памяти; $n = 8,16,32$.

Действие команды:

$CF_x \leftarrow x + y$

В случае, например, 8-битных операндов x и y по этой команде в 9-битный получатель CF_x заносится сумма двух 8-битных чисел x и y, двоичные коды которых складываются как натуральные числа (числа без знака). Выражение CF_x обозначает конкатенацию (сочленение) флажка CF и байта-аккумулятора, в качестве которого используется регистр или ячейка переменной x. Таким образом, 8 бит суммы $x + y$ (биты 7-0) заносятся на место первого операнда x, а ее самый старший, 9-й бит (бит 8) – во флажок переноса CF.

Интерпретация результата работы команды add.

1. Если x и y – натуральные числа, то в CF_x получается их правильная сумма.
2. Если x и y – дополнительные коды чисел со знаком, то в x получается правильный дополнительный код суммы при условии, что не произошло арифметического переполнения. Если же переполнение произошло, то правильный 9-битный дополнительный код суммы образуется в CF_x.

Утверждение 1 справедливо ввиду назначения флажка CF, а 2 следует из свойства 1 дополнительных кодов.

Пример.

```
mov  BH, -5 ; занести в регистр BH число -5
add  BH, 9 ; сложить -5 и 9 и занести сумму в CF_BH
```

Первая команда (mov) заносит в регистр BH число -5. При этом -5 транслируется ассемблером в дополнительный код по одной из формул (7). Например, по второй формуле имеем:

$$(-5)_{доп} = 2^8 - |-5| = 2^8 - 5 = 1°0000°0000 - 0000°0101 = 1111°1011$$

по третьей:

$$(-5)_{доп} = \overline{5}+1 = 1111°1010 + 1 = 1111°1011$$

Этот код (1111°1011) и поступает в BH по команде mov: BH←1111°1011.

Вторая команда (add) складывает коды как натуральные числа и помещает результат в CF_BH:

$1111°1011 = (-5)_{доп}$

\+

$0000°1001 = 9$

=

$1°0000°0100 \rightarrow$ CF_BH : CF = 1, BH = 0000°0100 = 4

Таким образом, во флажок CF поступает 1, а в BH – число 0000°0100 = 4, т.е. правильная сумма чисел -5 и 9. Это и есть ответ для случая 2 – сложения 8-битных знаковых чисел в дополнительном коде.

Вычитание натуральных и знаковых чисел с помощью команды sub.

Формат команды на ассемблере:

sub *x*, *y*

где операнды имеют тот же смысл, что и в команде add.

Действие команды с точки зрения программиста (программное действие) в случае 8-битных операндов *x* и *y* осуществляется следующим образом:

$$\mathrm{CF_}x \leftarrow 2^8 + x - y \equiv 1_x - y,\ \mathrm{CF} \leftarrow \neg\mathrm{CF} \qquad (2.13)$$

– вычисляется разность между 9-битным числом $2^8 + x = 1_x$ (1_x – конкатенация 1 и x) и 8-битным числом y. Эта разность запоминается в 9-битном получателе CF_*x*, после чего флажок CF инвертируется. Числа вычитаются как натуральные (числа без знака), т.е. вычитаются их коды. Инвертированный флажок CF сигналит о заеме, т.е. о том, что вычитаемое y (как число без знака) больше уменьшаемого x (как числá без знака). Единица («1») приписывается слева к операнду x перед вычитанием с целью заема в операции $x - y$, если $y > x$. Если же $y \le x$, то заем не нужен, флажок CF остается равным 1 и после инвертирования обнуляется, сигналя об отсутствии заема.

Мы пояснили действие команды sub с точки зрения программиста, соответствующее естественному содержанию операции вычитания. В действительности же по команде sub процессор не вычитает, а складывает уменьшаемое с дополнением вычитаемого. Поэтому аппаратная реализация команды выглядит так:

$$\mathrm{CF_}x \leftarrow x + \mathrm{ДОП}(y),\ \mathrm{CF} \leftarrow \neg\mathrm{CF}$$

Согласно свойству 3 дополнительного кода это эквивалентно вычитанию по формуле (13). Действительно, $x + \mathrm{ДОП}(y) = x + 2^8 - y = 2^8 + x - y$.

Поясним еще раз, что инвертирование флажка CF в командах вычитания введено в архитектуру Intel для того, чтобы он сигналил о заеме, а не о переносе при вычитании.

Интерпретация результата работы команды sub:

1. Если x и y – натуральные числа, то в $\mathrm{CF_}x$ получается их правильная разность в дополнительном коде.

   Это утверждение можно перефразировать иначе:

   если x и y – натуральные числа, то:

   а) при $x \ge y$ в x получается их положительная разность в натуральном коде;

   б) при $x < y$ в $\mathrm{CF_}x$ – отрицательная разность в дополнительном коде.
2. Если x и y – дополнительные коды чисел со знаком, то в x получается правильный дополнительный код их разности при условии, что нет арифметического переполнения. Если переполнение есть, то правильный

9-битный дополнительный код разности равен $\overline{CF}_x$, где CF – состояние флажка переноса после выполнения команды sub, $\overline{CF} = \neg CF$ – инверсия флажка.

Утверждения 1 и 2 доказываются путем простых рассуждений, опирающихся на свойства 1-3. Эти рассуждения мы здесь опускаем.

Пример.

```
mov  DL, 3 ; записать в регистр DL число 3
sub  DL, 4 ; вычесть 4 из содержимого DL, т.е. из 3, запомнить ответ в CF_DL
и инвертировать CF.
```

Первая команда заносит в DL число 3: DL←0000°0011

Вторая вычитает 4 из $2^8 + 3 = 1°0000°0000 + 0000°0011 = 1°0000°0011$, заносит ответ в CF_DL и инвертирует CF:

```
1°0000°0011 =  2^8 + 3
–
0000°0100 =4
=
0°1111°1111 → CF_DL, CF←¬CF: CF = 1, DL = 1111°1111 = (-1)доп
```

Таким образом, в CF оказывается 1, в DL – 1111°1111, т.е. -1 в дополнительном коде. В результате мы имеем в DL правильную разность чисел 3 и 4 в дополнительном коде – число -1 (случай 2, переполнения нет), а флажок CF сигналит о заеме из 9-го разряда, т.е. о том, что вычитаемое больше уменьшаемого.

Приведенный пример для команды sub не исчерпывает всех вариантов сочетания значений исходных операндов. Вы можете проверить эти варианты сами и убедиться на практике в справедливости утверждений 1 и 2.

Напомним, что факт арифметического переполнения фиксируется процессором с помощью **флажка OF** (Overflow Flag). Этот флажок устанавливается в «1», если арифметическое переполнение произошло, и сбрасывается в «0», если переполнения не было (п. 2.5).

Можно доказать (доказательство мы не приводим) следующий **критерий арифметического переполнения**, удобный для аппаратной реализации. Переполнение (а значит, установка OF в «1») имеет место, когда перенос в старший, знаковый, разряд n-1 (n = 8, 16, 32) и из него не совпадают. В противном случае, когда они совпадают, т.е. имеются оба переноса или нет ни того, ни другого, переполнения нет (производится сброс OF в «0»). Говоря условно, если число указанных переносов нечетное, то переполнение есть (OF=1), а если четное – нет (OF=0).

Таким образом, значение флага OF при сложении/вычитании в процессоре определяется логическим выражением $OF = C_{n-1} \oplus C_n$, где C_{n-1} – перенос в знаковый разряд, C_n – перенос из знакового разряда.

Этот критерий встроен в электронную схему процессора, обеспечивая правильную автоматическую реакцию флажка OF.

Выше был назван еще один флажок, помогающий правильно организовать операции сложения и вычитания. Это **флажок CF** (Carry Flag) (п. 2.5). В операциях сложения и вычитания натуральных чисел, а также знаковых чисел в случае арифметического переполнения правильный результат всегда образуется в совокупном получателе – сочленении этого флажка и аккумулятора CF_Ac.

Особенности сложения и вычитания многобайтных чисел. При необходимости сложения многобайтных целых чисел первые (младшие) байты (или слова) чисел складываются с помощью команды add, а все последующие – с помощью команды adc, которая складывает два операнда, а также значение флажка CF, т.е. значение переноса от сложения предыдущего байта. Если предварительно обнулить флажок CF, то и первые байты чисел можно складывать по команде adc. При вычитании многобайтных целых чисел действия аналогичны, но используются команды sub (вычитание) и sbb (вычитание операнда и флажка CF).

Преобразование чисел из одной системы счисления в другую. После ввода с клавиатуры исходные операнды должны быть преобразованы в двоичную систему счисления (если они были введены в другой системе счисления). Для этого отдельные разряды введенных операндов умножаются на соответствующие этим разрядам веса с последующим сложением результатов. Например, пусть введено десятичное число $x = 7132$. Его перевод в двоичное число можно выполнить с помощью соотношения: $x = 7 \cdot 10^3 + 1 \cdot 10^2 + 3 \cdot 10^1 + 2 \cdot 10^0$ (где все числа задаются, разумеется, в двоичной системе). Поэтому для перевода надо написать программу, реализующую вычисление этого соотношения.

Перед выводом на экран результат операции надо, наоборот, перевести из двоичной системы в целевую. Например, для перевода двоичного числа *x* в десятичную систему воспользуемся тем же соотношением $x = x_3 \cdot 10^3 + x_2 \cdot 10^2 + x_1 \cdot 10^1 + x_0 \cdot 10^0$, где x_3, x_2, x_1, x_0 – десятичные цифры, подлежащие определению. Записав это выражение как $x = ((x_3 \cdot 10 + x_2) \cdot 10 + x_1) \cdot 10 + x_0$, получаем правило для определения искомых цифр x_3, x_2, x_1, x_0:

x_0 – остаток от деления *x* на 10;

x_1 – остаток от деления первого частного $(x_3 \cdot 10 + x_2) \cdot 10 + x_1$ на 10;

x_2 – остаток от деления второго частного $x_3 \cdot 10 + x_2$ на 10;

и, наконец,

x_3 – последний остаток (когда последнее частное, которое мы собираемся разделить на 10, оказывается меньше 10).

2.7. Стек

Стеком (от англ. stack – кипа, стопка) называют часть оперативной памяти (часть ОЗУ), реализующую стековую дисциплину доступа (записи и чтения слов) по принципу: последний записанный в стек элемент выбирается из него первым (LIFO – Last-Input-First-Output, последний в очереди первым обслуживается, «последним вошел – первым вышел»). То есть, последнее сохраненное в стеке значение будет первым значением, которое вы получите при чтении из стека. Стек можно представить себе как некий ящик без крышки (рис. 2.18), в который могут помещаться однотипные объекты [24].

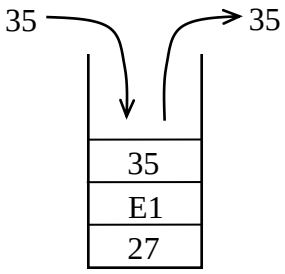


Рис. 2.18. Принцип действия стека

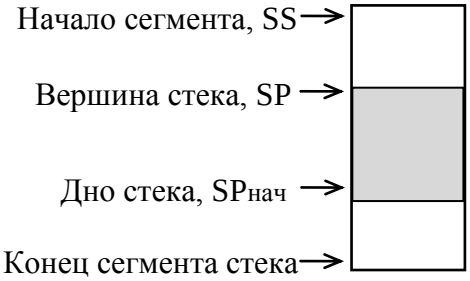


Рис. 2.19. Стек процессоров Intel расположен в стековом сегменте памяти

Стек состоит из смежных ячеек ОЗУ и, в общем, может располагаться в любом месте памяти. Однако стек процессоров Intel должен обязательно располагаться в стековом сегменте, задаваемом регистром SS (рис.2.19 [3]). Стек используется для временного хранения данных – промежуточных операндов и адресов возврата из подпрограмм.

Стеки всех развитых процессоров (Intel, Motorola и т.п.) реализованы в ОЗУ. Такие стеки называются *внешними* или *программными*. Некоторые процессоры (PIC16F, 8248) оснащены не внешним (расположенным в ОЗУ), а *внутренним*, или так называемым *аппаратным* стеком, находящимся внутри самого процессора. Аппаратный стек представляет собой группу специальных, последовательно пронумерованных регистров процессора и функционирует по тому же принципу, что и стек в ОЗУ. Далее будем рассматривать программный стек.

Последний элемент, записанный в стек, и, равным образом, ячейка, в которую он записан, называется *вершиной стека* (TOS – Top Of Stack). В процессоре есть регистр, указывающий на вершину стека – SP (Stack Pointer) – *указатель стека*. Он всегда содержит адрес вершины стека (см. рис. 2.19). Указатель стека процессоров Intel носит название ESP в защищенном режиме (32-битный регистр) и SP в реальном режиме (16-битный регистр).

Со стеком разрешены **три операции**: 1) инициализация (начальная установка), 2) запись в стек и 3) чтение из стека (рис. 2.20).

1) Инициализация стека. Прежде чем начать работу со стеком, программист должен инициализировать стек, т.е. произвести его начальную уста-

новку. Инициализация заключается в занесении в регистр SP начального адреса стека (начального значения адреса «вершины» стека). При помощи этой операции в памяти ЭВМ отводится место под стек начиная с установленного в SP адреса (в сторону младших адресов). Для МП Intel инициализация стека требует еще задания стекового сегмента с помощью регистра SS, при этом в защищенном режиме вместо SP используется расширенный регистр ESP.

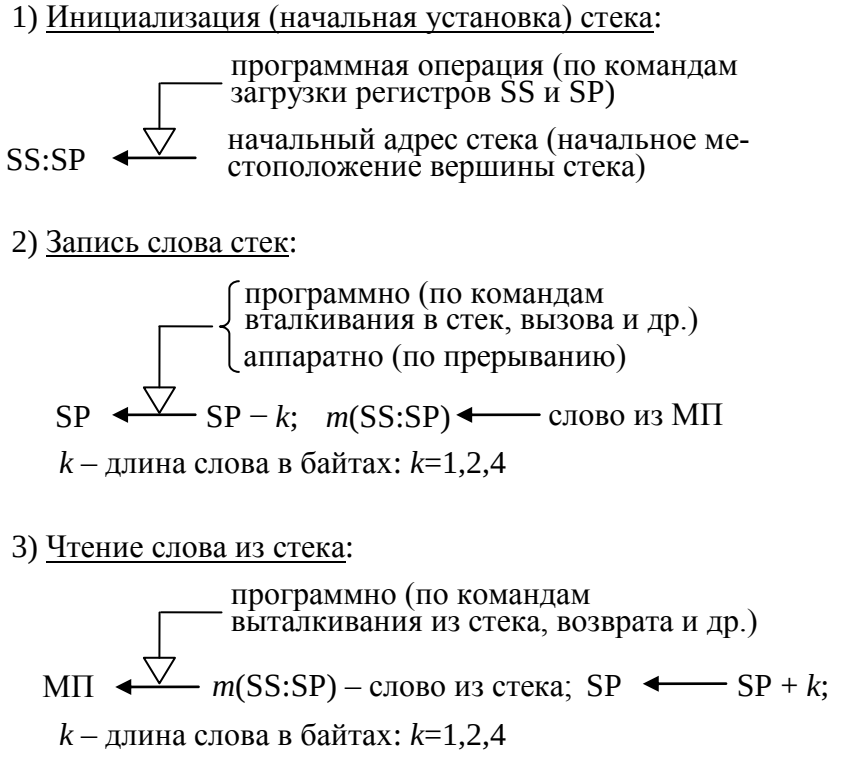


Рис. 2.20. Три операции со стеком процессора Intel в реальном режиме. В защищенном режиме вместо SP используется регистр ESP

Для МП Intel инициализация стека осуществляется при помощи трех команд (возьмем для определенности вариант реального режима и соответствующий ему регистр SP):

```
mov AX, Stack_Seg   ; Инициализировать
mov SS, AX                ; регистры SS
mov SP, Stack_Top   ; и SP
```

Первые две команды выделяют в памяти новый стековый сегмент, в котором будет располагаться новый стек, путем занесения базового адреса (селектора) сегмента (Stack_Seg) в регистр SS. Третья команда определяет, с какой ячейки внутри этого сегмента будет начат сам стек, т.е. определяет начальный внутрисегментный адрес вершины стека (Stack_Top). Полный, сегментированный адрес стека будет Stack_Seg:Stack_Top. После выполнения этих трех команд в памяти Intel отводится место под стек начиная с ячейки Stack_Seg:Stack_Top в сторону уменьшения адресов.

Приняв для наглядности численные адреса, возьмем те же команды, которые на этот раз открывают стек глубиной 100 внутри стекового сегмента с базовым адресом 12340h (рис.2.21):

```
mov AX, 1234h    ; AX←1234h
mov SS, AX       ; SS←AX – открыть стековый сегмент с ячейки 12340h
mov SP, 100      ; SP←100 – начать стек с 100-й ячейки сегмента.
```

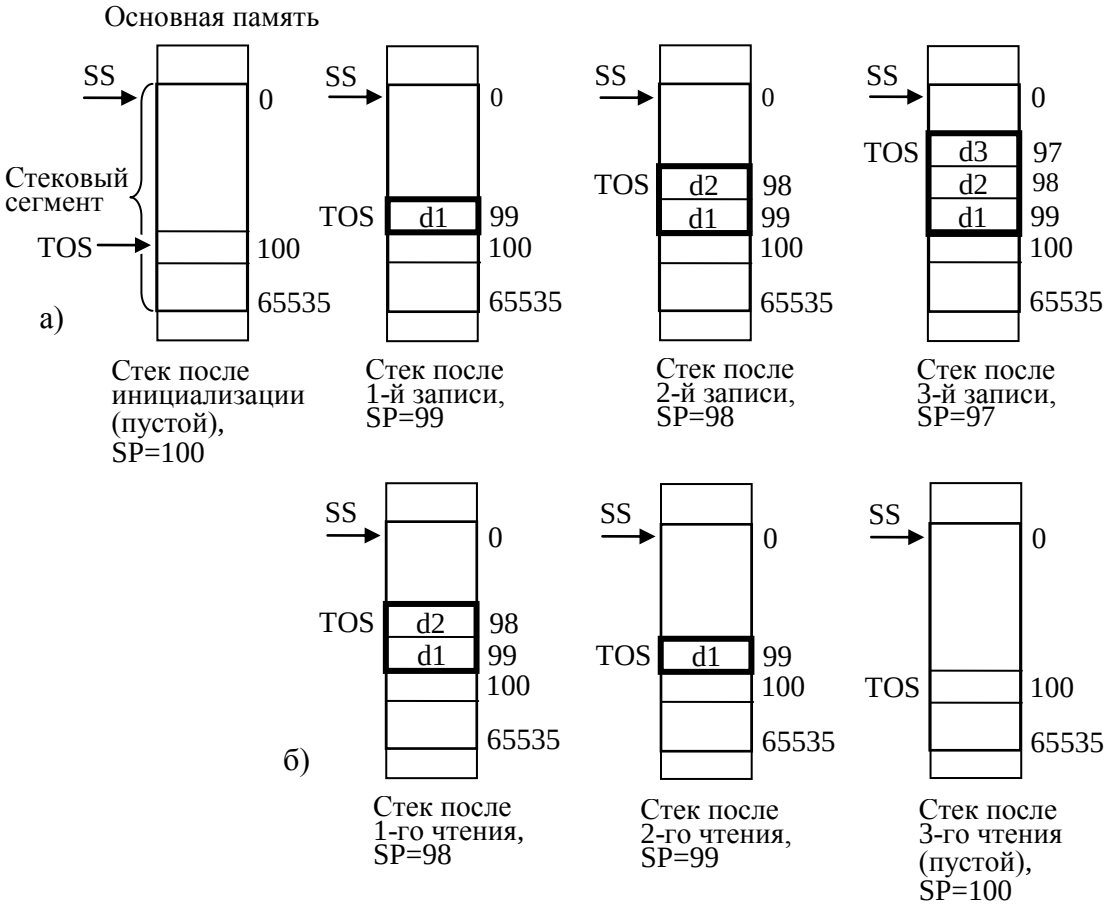


Рис. 2.21. Стек «растет» снизу вверх при записи в него (а) и «укорачивается» сверху вниз при чтении из него (б)

После инициализации можно обращаться к стеку – производить запись и чтение из него информации. Обращение производится при помощи специальных – стековых команд. Запись в стек – при помощи команды push – «втолкнуть» операнд в стек, а также команд вызова подпрограмм call и вызова прерываний int (сохраняющих в стеке адрес возврата из подпрограмм). Чтение информации из стека выполняется обратными командами: pop – «вытолкнуть» операнд из стека, возврата из подпрограмм ret и возврата из прерываний iret (извлекающими из стека ранее запомненный там адрес возврата). (Команды int и iret заносят и извлекают из стека еще и флажки.)

Все стековые команды адресуют стек через регистр-указатель стека SP (для МП Intel – через пару регистров SS:ESP или SS: SP). Вовлечение регистра SP этими командами происходит автоматически, без непосредственного указания программиста. Сам регистр SP в формате этих команд не указывается, ни-

где в явном виде не фигурирует. Он подразумевается самой командой (кодом операции). Такая адресация называется неявной или подразумеваемой или «адресацией по умолчанию» (п. 4.2).

Кроме программных обращений к стеку, осуществляемых по указке программиста (с помощью машинных команд), существуют и чисто аппаратные, не предусмотренные программой обращения к стеку (по записи), происходящие во время прерываний. Порядок записи в стек во время прерываний происходит по тем же правилам, что и при обычной, программной записи (гл. 5).

2) Запись каждого элемента (одной порции данных) в стек всегда производится по адресу на k меньшему, чем содержимое SP, а k – это размер элемента в байтах, и при каждой записи содержимое SP автоматически *уменьшается* на k (см. рис.2.20). Для процессора Intel k =1, 2, 4. Соответственно, при каждой записи содержимое указателя стека ESP/SP автоматически *уменьшается*: на 1 – при записи байта, на 2 – слова, на 4 – двойного слова.

Таким образом, при записи в стек сначала производится декремент SP на число записываемых байтов, а затем по полученному адресу записывается данное:

$$\mathrm{SP} \leftarrow \mathrm{SP} - k;\ m(\mathrm{SP}) \leftarrow \text{данное из МП}$$

в случае Intel:

$$\mathrm{ESP/SP} \leftarrow \mathrm{ESP/SP} - k;\ m(\mathrm{SS:ESP/SP}) \leftarrow \text{данное из МП}$$

где $m(\mathrm{SP})$ или $m(\mathrm{SS:ESP/SP})$ – ячейка памяти (memory) с адресом SP или SS:ESP/SP.

Если в предыдущем примере принять Stack_Top = 100 и записывать в стек байты (k=1), то первая запись произойдет в 8-битную ячейку с номером 99 внутри стекового сегмента, вторая – в ячейку 98, третья – в 97 и т.д. (см. рис.2.20).

Запись в стек процессора Intel осуществляется, например, по команде push. В реальном режиме это осуществляется следующим образом:

push AL ; SP←SP-1, m(SS:SP)←AL – записать («втолкнуть») в стек содержимого аккумулятора AL. Если команда «push» следует сразу после команд инициализации стека на ячейку 100, то содержимое AL будет записано в ячейку 100-1=99.

3) Чтение каждого элемента (порции данных) из стека происходит из ячейки, адрес которой размещается в SP (в случае Intel – в ESP/SP), и при каждом чтении указатель стека автоматически *увеличивается* на k, где k – длина элемента в байтах (см. рис.2.20). Процессор сначала читает по указанному адресу, а потом увеличивает его на k:

$$\text{МП} \leftarrow m(\mathrm{SP});\ \mathrm{SP} \leftarrow \mathrm{SP} + k$$

в случае Intel:

$$\text{МП} \leftarrow m(\mathrm{SS:ESP/SP});\ \mathrm{ESP/SP} \leftarrow \mathrm{ESP/SP} + k$$

Для процессора Intel k=1, 2, 4.

Чтение из стека Intel осуществляется, например, по команде pop. В реальном режиме это выглядит так (см. рис.2.20):

pop BH ; BH←*m*(SS:SP), SP←SP+1 – прочитать («вытолкнуть») элемент из стека в регистр BH. Если команду «pop» поставить вслед за «push» из последнего примера, то будет прочитана 99 ячейка, после чего указатель стека SP продвинется на 1: SP←99+1=100.

Благодаря автоматической коррекции в регистре SP (для Intel – в ESP/SP) всегда оказывается адрес последней занятой в стеке ячейки. Эту ячейку мы назвали вершиной стека: SP (ESP/SP) *всегда указывает на вершину стека* (TOS – Top Of Stack). Таким образом, запись и чтение из стека всегда производится *на* и *из* вершины стека, т.е. стек «растет» снизу вверх (в сторону младших адресов) при записи в него, и «укорачивается» сверху вниз (в сторону старших адресов) при чтении их него. Это означает, что только вершина стека, где находится последний его элемент, непосредственно доступна извне (программно доступна) при чтении из стека, а ячейка над вершиной стека программно доступна при записи в стек.

Последнее ограничение относится, конечно, только к упомянутым специальным командам обращения к стеку. Обычные команды имеют доступ к любым ячейкам стека, как и вообще к любым областям памяти.

Роль стека. Использование стека позволяет простыми аппаратными средствами осуществить:

а) удобное временное хранение промежуточных результатов счета, не задумываясь об адресах хранения;
б) переход к подпрограммам и возврат к основной программе (в т.ч. в случае нескольких вложенных подпрограмм);
в) переход к прерывающей подпрограмме и возврат к прерванной программе (раздел 5).

Любая команда перехода на подпрограмму (например, команда вызова call) прежде чем передать управление команде, расположенной по адресу перехода, заносит в стек *адрес возврата* – значение программного счетчика IP, которое было непосредственно перед переходом. Это значение показывает на следующую после call команду. Таким образом, *адрес возврата – это адрес команды после* call.

Всякая подпрограмма заканчивается командой возврата ret – return, извлекающей из стека адрес возврата и заносящей его обратно в IP. Благодаря этому возврат происходит всегда в то место основной программы, откуда в последний раз был произведен вызов (т.е. на команду, расположенную после call). Поэтому можно осуществлять вызовы подпрограмм из разных мест основной программы, возврат будет гарантирован в эти же места. Аналогично происходят вызовы (по команде int) и возвраты (по команде iret – interruption return) из подпрограмм прерываний: команда int запоминает в стеке содержимое регистров CS, IP и Flags, а команда iret восстанавливает их из стека, после чего работа основной, прерванной программы возобновляется с команды, следующей за int.

Если стек может хранить несколько адресов (значений IP), то мы получаем возможность выполнять вложенные подпрограммы, т.е. из подпрограммы можно сделать переход на другую подпрограмму или даже на саму себя – это

называется рекурсией. Очевидно, что глубина вложений подпрограмм зависит от емкости стека.

Итак:

Инициализация стека производится командами mov.

Запись в стек производится командами push, call и int. Запись каждого байта производится в ячейку над вершиной, т.е. по адресу SP-1, для этого перед каждой записью SP уменьшается на 1: SP←SP-1. Вершина TOS поднимается снизу вверх, *стек растет при записи*.

Чтение из стека производится командами pop, ret и iret. Чтение происходит из вершины, после чего SP увеличивает на 1 на каждый прочитанный байт: SP←SP+1. TOS опускается сверху вниз, *стек укорачивается при чтении*.

Подпрограммы. Вызов и возврат из подпрограмм. *Подпрограммой* (англ. subroutine) называют поименованную последовательность команд, которую можно вызывать – запускать на счет (с помощью команды call) из разных мест программы, при этом возврат из подпрограммы происходит всегда в ту точку программы, на ту ее команду, которая следует за командой последнего вызова («на после call»). Для возврата служит команда возврата ret (return), которую помещают в конце подпрограммы (рис.2.22). Возврат в правильную точку программы происходит благодаря тому, что команда call запоминает адрес возврата (содержимое программного счетчика) в стеке, а команда ret извлекает его из стека и заносит обратно в программный счетчик (см. п.Б.4).

Подпрограмма имеет имя (метку, заканчивающуюся двоеточием), которым, на языке ассемблера, помечается точка входа* в подпрограмму – команда, с которой подпрограмма запускается. Имя (без двоеточия) надо указывать в адресной части команды call при вызове подпрограммы:

call <имя>.

* *Точка входа* – это адрес оперативной памяти, с которого начинается выполнение программы, а также и сама команда, расположенная по этому адресу.

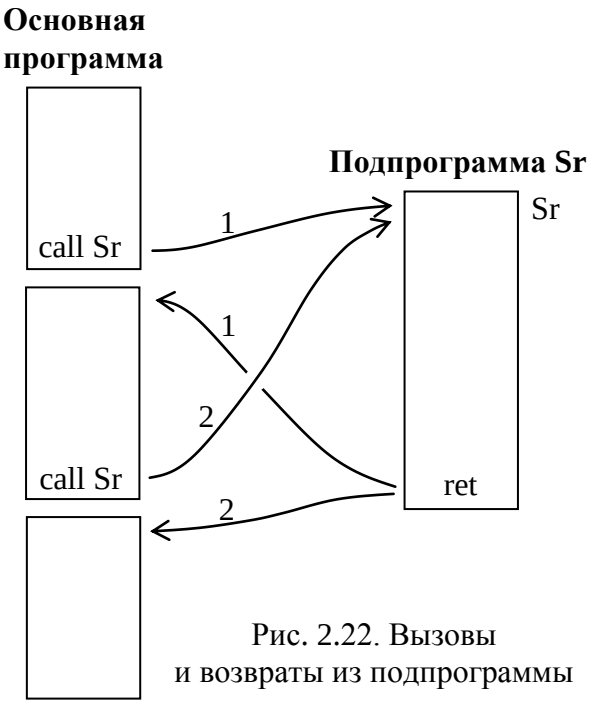


Рис. 2.22. Вызовы и возвраты из подпрограммы

По отношению к подпрограмме остальную часть программы принято называть основной программой. Подпрограммы служат для многократного выполнения однотипных действий, они значительно экономят память, так как позволяют описать эти действия однократно, а затем вызывать их по мере необходимости.

Подпрограмму можно размещать (в основной программе), в принципе, где угодно [17]. Но при этом надо понимать, что сама по себе подпрограмма не

должна выполняться, а может выполняться лишь тогда, когда к ней обратятся. Поэтому размещать ее надо так, чтобы на нее случайно не попало управление. Если же подпрограмму просто вставить в общий поток команд, то микропроцессор будет воспринимать команды подпрограммы как часть этого потока. Учитывая это обстоятельство, подпрограммы размещают обычно либо в конце сегмента команд (за условной директивой FINISH, или ENDS), либо в самом начале этого сегмента – перед той командой, с которой должно начинаться выполнение программы. В больших программах подпрограммы нередко размещают в отдельном сегменте команд. Если имеется несколько подпрограмм, то их обычно размещают рядом.

Второй вопрос – как описывать подпрограмму? В общем случае группу команд, образующих подпрограмму, можно никак не выделять в тексте программы. Однако в языке ассемблера (как и в языках программирования высокого уровня) подпрограммы принято оформлять специальным образом – в виде процедур или в виде функций [18].

Подпрограммы-функции и подпрограммы-процедуры. *Функция* – это подпрограмма, способная возвращать некоторое значение, т.е. ее вызов является, с точки зрения языка программирования, выражением. Поэтому он может использоваться только в других выражениях или в качестве правой части присваивания. *Процедура* – это независимая подпрограмма, которая осуществляет управление какими-то устройствами или выполняет какие-либо вычисления. Если производятся вычисления, то результат помещается в глобальную переменную, чтобы этим результатом могла воспользоваться другая подпрограмма или основная программа.

Пример использования функции вычисления синуса угла *x*:

$y := \sin(x)$;

Пример вызова процедуры поиска максимального из чисел *a* и *b*:

$Poisk(a,b,res)$;

Никак не оформленная подпрограмма имеет вид

```
; M – подпрограмма с именем M.
M: …
   ret
```

Ее вызов осуществляется командой

```
call M
```

Для описания последовательности команд в виде процедуры в языке ассемблера используются две директивы: PROC (procedure) и ENDP (end of procedure). Описание подпрограммы в виде процедуры выглядит так:

```
<имя процедуры> PROC <параметр (тип процедуры)>
    <тело процедуры>
<имя процедуры> ENDP
```

Как видно, перед телом процедуры (ее командами) ставится директива PROC, а за ним – директива ENDP. В обеих этих директивах указывается одно и то же имя – имя, которое мы дали подпрограмме-процедуре. Имя процедуры

считается меткой, считается, что оно метит первую команду процедуры (хотя за ним и не ставится двоеточие). Например, имя процедуры можно указать в команде вызова, и тогда будет осуществлен переход на первую команду процедуры.

Так как программа может занимать более одного сегмента, то процедуры, вызывающие друг друга, могут находиться в различных сегментах:

Если вызываемая процедура находится в текущем сегменте, то используется ближний вызов команды call:

call M ; SP← SP-2, *m*(SS:SP)←IP, IP←M.

При этом параметр, определяющий тип процедуры, должен принимать значение NEAR.

Если же вызываемая процедура находится в другом сегменте, то используется дальний вызов команды call, а параметр типа процедуры FAR:

call M ; SP← SP-2, *m*(SS:SP)←IP, SP← SP-2, *m*(SS:SP)←CS, CS:IP←M.

Точно так же осуществляется вызов и никак не оформленных подпрограмм [19].

Макрокоманда. *Макрокомандой* (макросом, macro command) называют поименованную группу команд. Для именования служит парная директива MACROC – ENDM [19]. Ниже приведена структура макроса с именем ABC:

```
ABC MACROC <формальные параметры через запятую>
    <тело макроса>
ENDM
```

После объявления макроса его имя (в нашем примере ABC) можно записывать в разных точках программы любое число раз:

```
ABC
…
ABC
…
ABC
```

Каждый раз транслятор вставляет вместо имени всю группу команд макроса (тело макроса).

Макрокоманда в сравнении с подпрограммой. Макрокоманда похожа на подпрограмму, но при трансляции ее тело вставляется в программу столько раз, сколько раз макрокоманда вызывается, а подпрограмма содержится в программной памяти в единственном экземпляре. Поэтому использование подпрограмм экономит место в памяти. Однако вызов и возврат из подпрограмм занимает дополнительное время процессора, затрачиваемое на выполнение команд call и ret, которых нет в макрокомандах. Коротко: подпрограмма экономит место, а макрокоманда – время.

Глава 3. ОБМЕН ИНФОРМАЦИЕЙ В ЭВМ

3.1. Структура аппаратуры и потоки информации ЭВМ

Разработка компьютерной системы предполагает использование готовых, покупных изделий – микрочипов процессора, устройств памяти, контроллеров ввода-вывода и других устройств, которые подлежат соединению в целостно функционирующую систему. Для решения возникающих при этом задач разработчику необходимо располагать сведениями об архитектуре устройств на уровне программно доступных элементов и о функциях их внешних сигналов. Внутренняя организация устройств скрыта от пользователя, им не контролируется, а ее ресурсы вовлекаются в функционирование системы только путем активизации их через внешний интерфейс и программно доступные элементы. Поскольку функции комплектующих изделий предопределены и предоставляются в распоряжение пользователя в неизменном виде, то разработка аппаратной части системы сводится по существу к разработке интерфейса – средств сопряжения готовых изделий между собой. Поэтому ключевую роль при создании и эксплуатации компьютерных систем играет знание принципов построения интерфейса ЭВМ и обмена информацией по системной магистрали.

На рис. 3.1 представлена **структурная схема и потоки информации вычислительной системы** [15]. Как видно из рисунка, для ввода данных в компьютер могут понадобиться обычные периферийные устройства: диск, клавиатура, флэш-карты и пр. Данные от процесса или объекта управления могут быть введены непосредственно через порты ввода. Если данные имеют аналоговую форму, например, для управления температурой на промышленном предприятии, они сначала должны быть преобразованы в цифровую форму, так как компьютер может работать только по двоичной системе. Это преобразование осуществляется с помощью аналого-цифрового преобразователя (АЦП). Аналогично данные для управления процессом, полученные на выходе компьютера, могут быть преобразованы в аналоговую форму с помощью цифро-аналогового преобразователя (ЦАП).

Адреса, данные и сигналы управления передаются по шинам. По *шине адреса* процессор выдает адреса команд и операндов на память или адреса операндов на порты устройств ввода-вывода (УВВ). *Селекторы (дешифраторы) адреса* находят по этому адресу нужную информацию в памяти или УВВ. По *шине данных* происходит обмен информацией между процессором и памятью или УВВ: процессор получает команды и операнды из памяти или выдает операнды в память, а также вводит и выводит данные от/на устройства ввода-вывода через порты. По *шине управления* процессор выставляет отдельные сигналы, управляющие работой памяти и портов ввода-вывода.

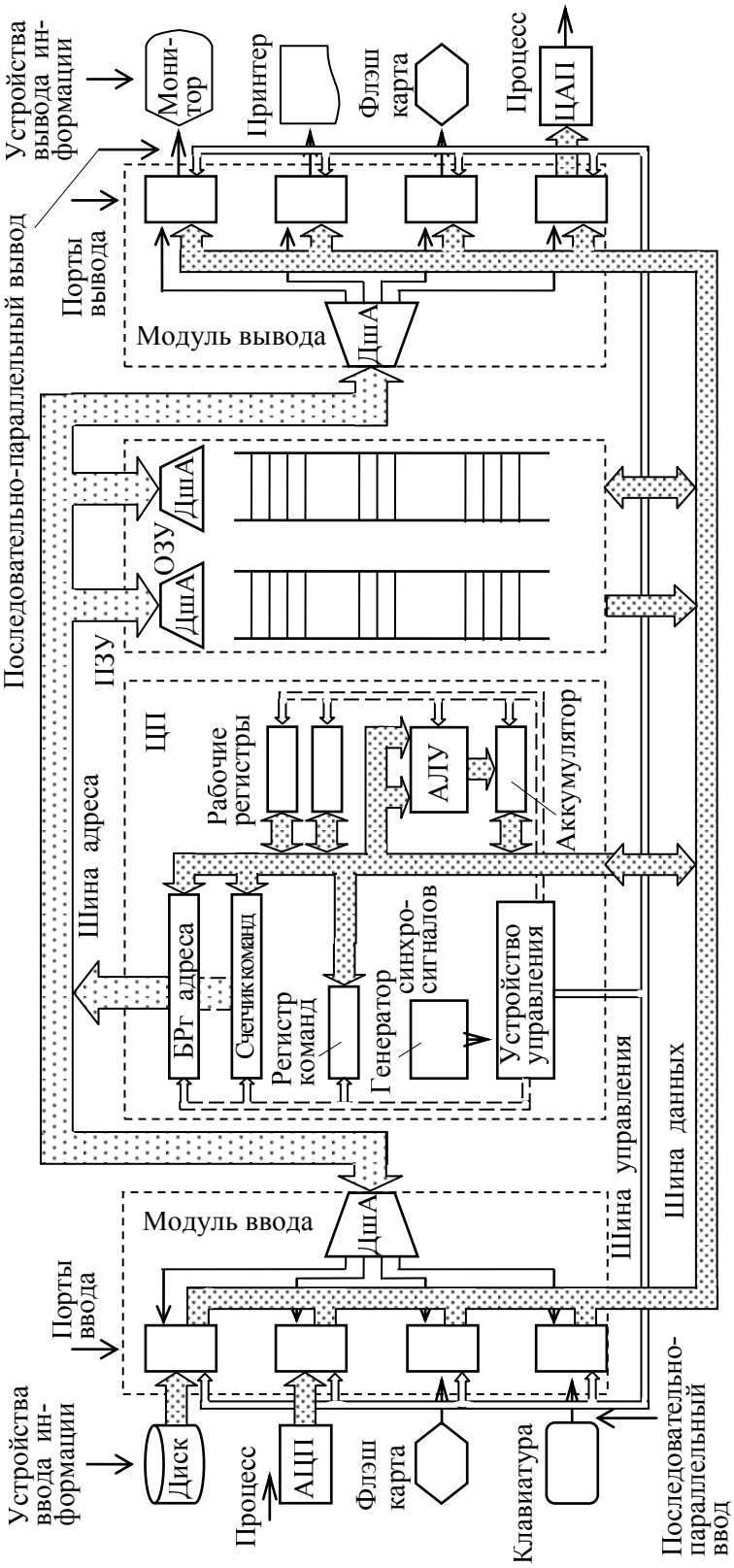


Рис. 3.1 – Потоки информации в ЭВМ
АЛУ – арифметико-логическое устройство; ДшА – дешифратор адреса;
БРг – буферный регистр

Основная память состоит из ПЗУ и ОЗУ (п. 1.2). Содержимое ПЗУ не может быть стерто. Поэтому ПЗУ используется как память программ, которую изготовитель заранее запрограммировал в соответствии с требованиями пользователя. Для различных программ используют различные ПЗУ или их части. Постоянное запоминающее устройство можно сравнить со складом, с которого можно получать какие-то вещи (предметы).

Памятью данных в компьютере является ОЗУ. Его можно сравнить со складом, на который можно сдавать вещи и получать. Информация, хранящаяся в ОЗУ, стирается, когда прерывается подача напряжения питания. Это можно предотвратить, установив резервные батареи или источники бесперебойного питания, содержащие электрические аккумуляторы.

В современных компьютерах программы содержатся преимущественно в ОЗУ, а не в ПЗУ, после их загрузки с диска, а в ПЗУ хранятся обычно только программы базовой системы ввода-вывода (ROM BIOS), а также некоторые константовые параметры.

Данные, поступающие в ОЗУ, обрабатываются в центральном процессоре (ЦП) в соответствии с программой, хранящейся в ОЗУ или ПЗУ. Результаты операций в ЦП хранятся в *регистре аккумуляторе* или ОЗУ. Они могут быть выведены по команде через один из портов вывода на устройства вывода, подсоединенные к этому порту. Требуемый порт вывода выбирается через схему селектора (дешифратора) адреса.

Система соединения блоков ЭВМ. Информация может поступать на данную шину только от одного блока ЭВМ. Например, информация может поступить на шину данных только от одного из следующих блоков: ЦП, ОЗУ, ПЗУ, одного из портов ввода. Для достижения этого в каждую выходную линию шины каждого блока включена так называемая тристабильная буферная схема (рис.3.2), которая аналогична выключателю, управляемому соответствующим сигналом.

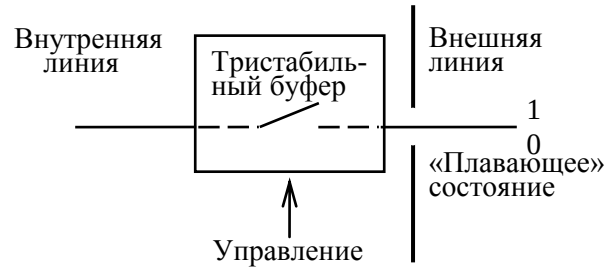


Рис. 3.2. Тристабильная буферная схема

Название тристабильная (three-state) происходит от того, что для данного блока внешняя линия может принимать три разных состояния: 0, 1 и отключенное.

Когда под воздействием сигнала управления тристабильная буферная схема срабатывает, то соединения между внутренней и внешней линиями нет. В этом (отключенном) состоянии сопротивление между внутренней и внешней

линиями высоко, и поэтому такое состояние называется высокоимпедансным. Его также часто называют плавающим.

Сигналы, которые переключают тристабильные буферные схемы в требуемое состояние, поступают от устройства управления ЦП (УУ). Таким образом, УУ выбирает блок ЭВМ, который передает данные в шину.

На рис. 3.3 показана **логика формирования системных шин процессора Intel 80x86 старших моделей,** а на рис. 3.4 – **логика подключения к шинам устройств памяти и периферийных регистров.** На схемах:

1) *Центральный процессор* (модуль ЦП; см. рис. 3.3):

CPU – центральное процессорное устройство – собственно процессор Intel.

ABF и DBF (Address Bus Former and Data Bus Former) – формирователи шин адреса и данных, построенные на буферах BUF (Buffer) – двунаправленных усилителях-формирователях шин. Сигнал на входе TF (Transmit Formation) буфера определяет направление передачи данных: TF = 1 – с шины A на B; TF = 0 – с шины B на A. Сигнал OE# = 0* (Output Enable) разрешает передачу, при OE# = 1 выходные буферы формирователя переводятся в отключенное состояние, передача запрещена.

* Значок # (решетка) означает, что сигнал имеет низкий активный уровень.

DC (Decoder) – дешифратор кода, выполняющий функцию системного контроллера SC (System Controller), или формирователя шины управления CBF (Control Bus Former). Вырабатывает сигналы управления передачей данных, индицирующие тип шинного цикла (см. далее).

2) Участок *оперативной памяти* (см. рис. 3.4):

Две микросхемы RAM емкостью по 2^{11} = 2Кбайта каждая. Запись/чтение данных в/из ОЗУ производится по двунаправленным линиям. Входные линии AD10-AD0 служат для адресации ячеек внутри ОЗУ. Сигнал на входе R/W# определяет выполняемую операцию: R/W# = 1 – чтение из адресованной ячейки, R/W# = 0 – запись в ячейку. Линии выборки кристалла CS# (Chip Select) разрешают выполнение операции записи или чтения. Операция возможна только при активных (низких) уровнях напряжений на всех входах CS#. Если хотя бы на одном входе сигнал пассивен (5 В), внутренние линии данных отключены от внешних DT7-DT0.

DC – дешифратор адреса.

3) *Порты ввода-вывода* (см. рис. 3.4):

Восемь однобайтных портов ввода и восемь вывода реализованы на микросхемах информационных регистров RG (Register). Запись кода данных в регистр происходит по входным линиям DI7-DI0 (Data Input) при активном (высоком) уровне напряжения на стробирующем входе STB (Strobe) (при STB = 0 регистр отключен от шины DI7-DI0 и данные в него не могут быть занесены). Чтение данного из регистра – по выходным линиям DO7-DO0 (Data Output) при

активном (0 В) сигнале на входе разрешения выдачи OE#. При OE# = 5 В данные не выдаются, регистр отключен от шины данных DO7-DO0.

4) *Системная магистраль* (см. рис. 3.3 и 3.4):

DB (Data Bus) – *шина данных*. Представлена сигналами D63-D0. D63 соответствует старшему биту данных, D0 – младшему, линии D63-D56 определяют старший байт, а линии D7-D0 – младший байт. ШД используется для передачи данных (команд и операндов) между процессором и другими устройствами системы, а также для ввода номера прерывания «*n*» из контроллера прерывания по линиям D7-D0.

AB (Address Bus) – *шина адреса*. Содержит линии A31-A3, BE7#-BE0#. A31 – старший бит, A3 – младший. Сигналы на линиях A31-A3 определяют адрес 8-байтной ячейки памяти, к которой процессор будет обращаться в текущем цикле, а сигналы адресации байтов BE7#-BE0# прямо указывают выбираемые

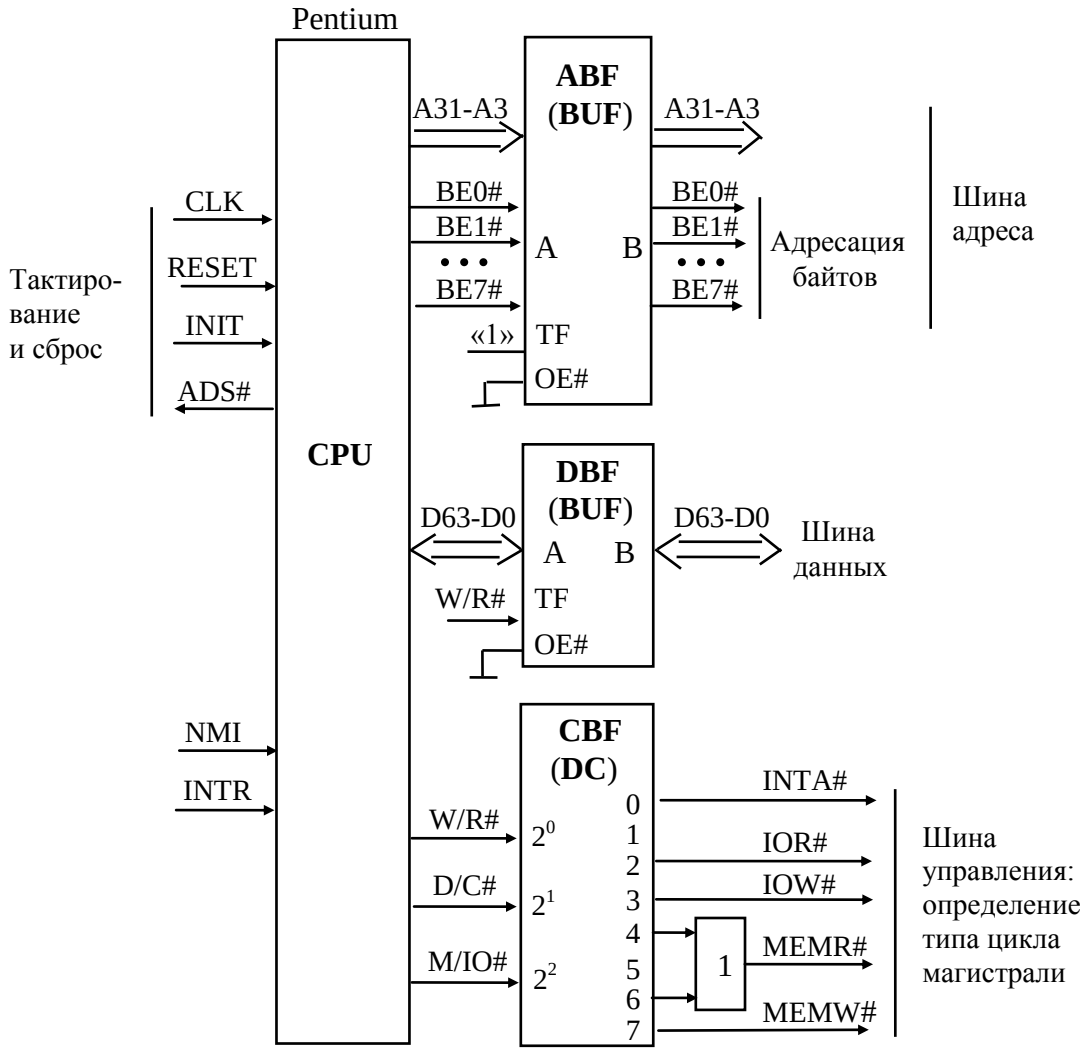


Рис. 3.3. Логика формирования системных шин процессора Intel 80x86

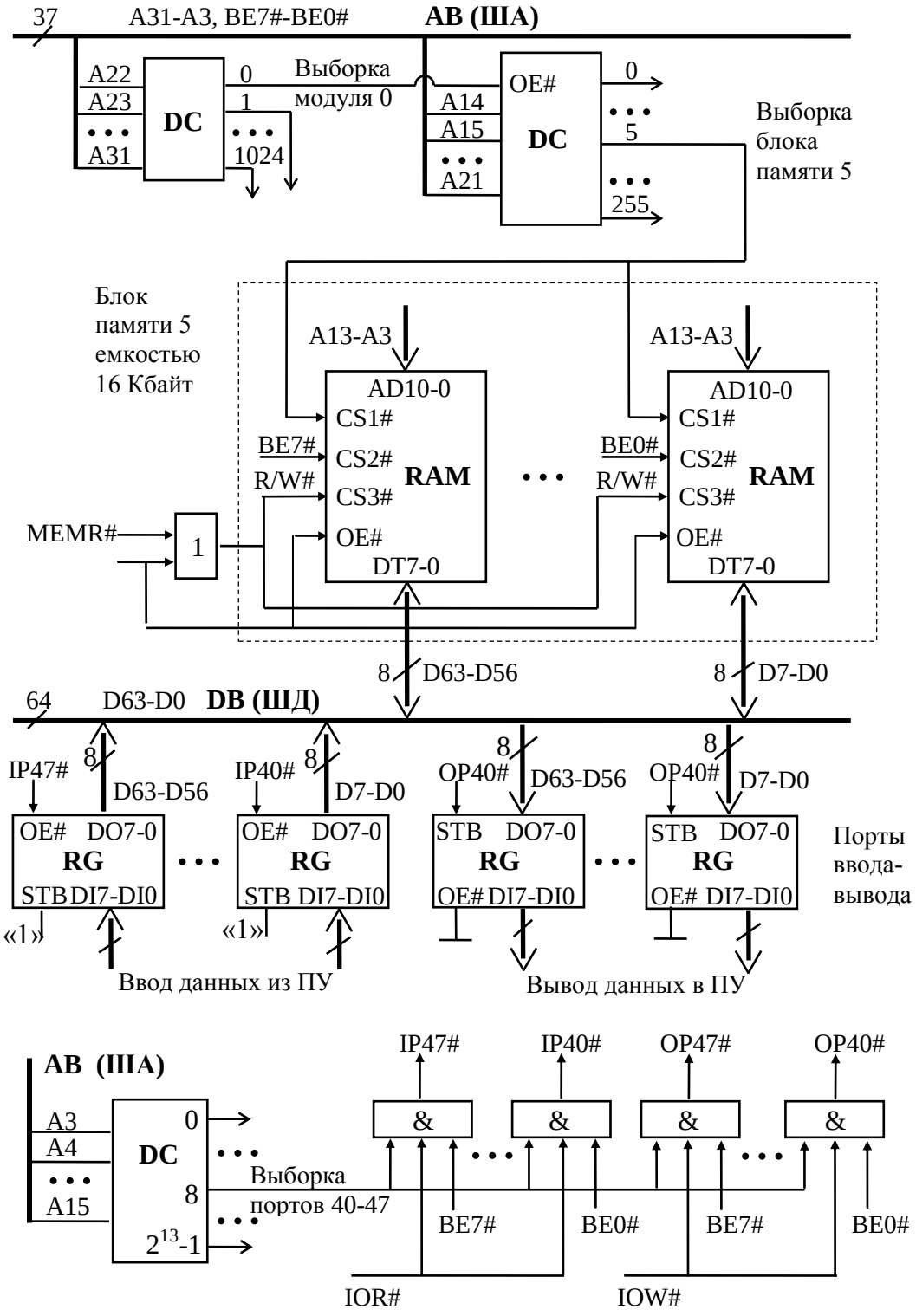


Рис. 3.4. Участок оперативной памяти и портов ввода-вывода Intel-компьютера

байты внутри этой ячейки: BE7# адресует байт D63-D56, ... , BE0# – байт D7-D0. Для формирования нужных сигналов BEx# процессор использует биты A2-A0 адреса, задаваемого программой, и размер в байтах адресуемого данного. Порты адресуются по линиям A15-A3, а линии A31-A16 пассивны.

Адресные линии используются согласно следующему правилу (рис. 3.4): по старшим линиям A31-A22 процессор выбирает модуль памяти, к которому он будет обращаться в текущем шинном цикле (например, модуль 0), по средним линиям A21-A14 определяется блок внутри модуля (например, блок 5), а по младшим линиям A13-A3 – 8-байтная ячейка в этом блоке. Наконец, при помощи сигналов BE7#-BE0# процессор указывает целевые байты в ячейке, которые и передаются в данном цикле.

Шина управления. Локальные управляющие сигналы M/IO# , D/C# и W/R# указывают тип выполняемого цикла магистрали (табл.3.1). Сигнал W/R# разделяет циклы записи (Write) и циклы чтения (Read): при W/R# = 1 процессор выполняет запись данного во внешнее устройство (память или порт), при W/R# = 0 – чтение данного из внешнего устройства (памяти или порта). Сигнал D/C# разделяет циклы данных (Data), т.е. передачи операндов (D/C# = 1), и циклы управления (Control) – считывание команд из памяти или ввода номера прерывания «*n*» из контроллера прерывания (D/C# = 0). Наконец, сигнал M/IO# показывает будет ли обращение к памяти (Memory): M/IO# = 1, или к порту ввода-вывода (Input-Output): M/IO# = 0.

Таблица 3.1 – Типы циклов магистрали процессора Intel 80x86

M/IO#	D/C#	W/R#	Тип цикла
0	0	0	Подтверждение прерывания
0	0	1	Специальные циклы (Останов и др.) *
0	1	0	Ввод (Чтение устройства ввода-вывода)
0	1	1	Вывод (Запись в УВВ)
1	0	0	Чтение кода (команды) из памяти
1	0	1	Зарезервировано
1	1	0	Чтение данных (операнда) из памяти
1	1	1	Запись данного (операнда) в память

* Специальные циклы различаются дополнительно сигналами на выходах BE7# - BE0# и здесь не рассматриваются.

На основе локальных сигналов M/IO#, D/C# и W/R# формируются системные сигналы MEMR# (Чт), MEMW# (Зп), IOR# (Вв), IOW# (Выв), описываемые в п.3.3.

Сигнал состояния адреса ADS# (Address State) определяет начало цикла обмена по шине, цикла шины (п.3.4). Сообщает о том, что в шину адреса в данном цикле уже выставлен из процессора истинный адрес ячейки или порта. RESET и INIT (Initialization) – сигналы сброса и инициализации процессора. CLK (Clock) – общий сигнал синхронизации, или тактирования. INTR (Interruption) и NMI (No Masked Interruption) – сигналы прерывания.

Другие управляющие сигналы процессоров Intel 80x86 мы рассматривать не будем. С ними можно ознакомиться по литературе [4, 9, 10, 19, 23].

3.2. Базовые машинные операции

Компоненты программного процесса. Работа МП заключается в выполнении *машинной программы*. Процесс выполнения любой программы состоит из последовательного выполнения отдельных ее команд. Отработку процессором одной команды, а также интервал времени, требующийся для этого, называют *командным циклом, циклом команды*.

Цикл любой команды сводится к одному или нескольким обращениям процессора к устройствам шины – памяти и «периферии». Обращаться к шине процессору требуется, например, для чтения команд, а также для чтения, записи, ввода или вывода данных выполняемой команды. Всякое такое обращение есть один *машинный цикл* (*цикл шины, цикл магистрали*). При этом внутренние операции процессора, следующие сразу после обращения процессора к шине и до нового такого обращения, также входят в состав данного машинного цикла. К ним относятся декодирование считанной команды, сложение операндов, их сдвиги и т.п. Машинный цикл содержит, следовательно, однократную *внешнюю пересылку*, к которой могут добавляться еще *операции* по расшифровке команды и обработке данных, происходящие *внутри процессора*. В зависимости от типа и формата команды, способов адресации и числа операндов, число машинных циклов в одном командном цикле может быть различным – от одного до нескольких. Так же и в одном машинном цикле может выполняться различное число операций внутри процессора над байтами/словами (обычно не более одной-двух).

Процессор – динамическое устройство. Для того чтобы он функционировал, на него должны поступать побуждающие толчки – импульсы электрического напряжения от специального устройства – тактового генератора (п.1.2). С помощью импульсов, проходящих через все блоки процессора, генератор подталкивает их на выполнение очередных операций над данными и синхронизирует ход операций отдельных блоков. Период следования тактовых импульсов (промежуток между двумя соседними импульсами) называют *машинным тактом*. Длительность машинных тактов всегда одинакова. Выполнение различных операций – внешних пересылок и внутренних операций процессора может занимать от долей такта до многих тактов в зависимости от команды и процессора. Общая тенденция заключается в уменьшении количества тактов, затрачиваемых на выполнение операций.

Таким образом, просматривается иерархия процедур при работе микропроцессора:

Машинная программа ® Командный цикл ® Машинный цикл ® Машинная операция ® Машинный такт.

Сначала мы познакомимся с *базовыми, элементарными машинными операциями*, представляющими собой минимальные, функционально законченные

операции над байтами и словами, не расчленяемые далее на более простые подобные операции. Затем, в п.3.3, мы покажем, как из базовых операций образуются машинные циклы, а в п.4.6, 4.8 – как машинные циклы, следуя один за другим, компонуются в командные циклы и всю машинную программу. Таким способом мы проследим компоновку программно-аппаратных действий процессора и ЭВМ в виде последовательности базовых операций. Большинство этих операций производится над одной порцией данных – байтом или словом в один прием. Некоторые состоят из нескольких одинарных операций.

Семь типов машинных операций. МП работает в составе компьютерной системы и постоянно обменивается информацией с памятью и периферийными устройствами. Для обмена он использует системную магистраль. Знание принципов обмена информацией по системной магистрали необходимо для успешного решения задач, возникающих при организации интерфейса разрабатываемой вычислительной системы.

Основными шинами системной магистрали, используемыми при описании передачи цифровых сигналов в ЭВМ, являются шины данных, адресов и управления (см. рис. 1.6, 1.7). Модель системы с тремя шинами позволяет описать все *семь типов базовых машинных операций вычислительной системы* [14]:

1) **Чтение** центральным процессором одной порции данных из памяти. В этой операции процессор извлекает порцию данных из ячейки основной памяти. Порция данных пересылается под управлением процессора по шине данных и заносится в один из его регистров:

Регистр процессора (Рг) ← ШД ← ячейка памяти (ЯП).

2) **Запись** процессором одной порции данных («данного») в память. В этой операции «данное» передается из процессора в основную память:

Рг → ШД → ЯП.

3) **Ввод** процессором данного из периферийного устройства (устройства ввода), которое пересылается по шине данных из порта ввода ПУ и заносится в регистр процессора:

Рг ← ШД ← порт ввода.

4) **Вывод** данного из регистра процессора в ПУ (в порт вывода, устройство вывода):

Рг → ШД → порт вывода.

5) **Прерывание**, или «подтверждение прерывания» процессором – автоматическую реакцию процессора на сигнал прерывания ЗПр (запроса прерывания, или INTR – Interruption). Этот сигнал поступает на процессор от периферийного устройства (обычно – через контроллер прерываний) и вызывает следующие ответные действия процессора:

а) ввод из ПУ (чаще из контроллера прерываний) адреса перехода на подпрограмму обслуживания прерывания (подпрограмму прерывания ПП, ISR, от англ. – Interruption SubRoutine):

Пр ← ШД ← адрес перехода на ПП

в результате чего процессор «узнаёт», куда, по какому адресу памяти ему надо перейти, чтобы выполнить подпрограмму прерывания, и

б) запоминание в памяти (в стеке) адреса возврата из ПП:

Пр: адрес возврата из ПП → ШД → стек

и процессор сможет потом, после завершения подпрограммы, найти ту точку основной программы, в которую ему следует вернуться, чтобы продолжить выполнение основной программы, извлекши адрес этой точки из стека.

По окончании этих действий процессор переходит на ПП.

Развитые процессоры вводят в операции 5а не прямой, а косвенный адрес перехода, например, (как процессоры Intel) так называемый *номер прерывания n*. Получив значение *n*, процессор окончательно считывает из памяти (из определяемой этим номером ячейки) пусковой адрес ПП.

Режим прерываний описывается в главе 5.

6) **Прямой доступ** периферийного устройства (ПУ) к основной памяти компьютера (ПДП, DMA – Direct Memory Access). Прямой доступ к памяти заключается в пересылке данных между ПУ и памятью напрямую, минуя центральный процессор. Этот режим осуществляется под управлением контроллера ПДП (DMA Controller). Процессор на время ПДП вообще отключается от системной магистрали и не принимает никакого участия в пересылке данных, для чего контроллер ПДП подает на вход процессора сигнал захвата HLD (Зх), отсоединяющий внутренние блоки процессора от захваченных внешних шин.

7) Все **внутренние операции** процессора по обработке и преобразованию данных**,** происходящие на его собственных, внутренних узлах:

а) дешифрация команд, поступающих на процессор из памяти;

б) передача операндов из регистра на регистр процессора (внутренние, межрегистровые пересылки);

в) арифметические и логические действия над операндами;

г) другие внутренние операции.

Пересылка – это копирование данных. При выполнении любых операций по пересылке данных, как по системной магистрали, так и внутри процессора, приемное устройство копирует в своем регистре (регистре-приемнике) данное, содержащееся в выдающем устройстве (регистре-источнике). Содержимое регистра-источника при этом не уничтожается, остается без изменения.

Примеры операций и команд процессоров Intel 80x86:

Оп1 – чтение: mov AL,[BX]; AL←*m*(DS:BX) – скопировать в регистр-аккумулятор AL содержимое ячейки памяти, расположенной в сегменте DS по адресу, находящемуся в регистре BX.

Оп2 – запись: mov ES:[DI], BX; *m*(ES:DI) ←BX – передать слово из регистра BX в ячейку по адресу из DI, расположенную в дополнительном сегменте данных ES.

Оп3 – ввод: in AL, 60h; AL←*p*(60h) – ввести байт из порта с адресом 60 h в аккумулятор AL.

Оп4 – вывод: out DX, AX; *p*(DX) ←AX – ввести слово из AX в порт, косвенно адресуемый регистром DX.

Оп7 – преобразование: dec byte ptr [2000h]; *m*(DS:2000) ← *m*(DS:2000) - 1 – уменьшить (декрементировать, decrement) на 1 содержимое 2000-й однобайтной ячейки памяти (расположенной по умолчанию в сегменте данных).

Состав машинных операций отдельных команд. Каждая команда сопровождается всегда проведением не одной, а нескольких базовых машинных операций. Кроме *основной, «целевой»* операции, для осуществления которой команда предназначена (например, сложить два числа), проводится ряд *вспомогательных* операций, поддерживающих основную и не отраженных в названии команды. К таким, вспомогательным, операциям относятся всегда *чтение кода* команды из памяти и его *дешифрация*, являющиеся обязательными для всех команд. Для большинства команд выполняются также *вспомогательные пересылки* операндов из регистра на регистр или на ячейку памяти и др. Полный состав машинных операций любой команды мы научимся определять в гл.4. Однако какова бы ни была машинная команда, она всегда выполняется процессором при помощи операций, относящихся к названным выше базовым типам 1-4 и 7. Состав операций каждой команды зависит от ее назначения, размера в байтах, формата, режима адресации операнда и др. характеристик. Но любая команда генерирует всегда один и только один состав машинных операций из числа 1-4 и 7.

Функциональная полнота машинных операций. Мы перечислили семь типов машинных операций. Пять из них (1-4 и 7) генерируются программой (машинными командами), а две (5 и 6) – аппаратурой: по сигналу прерывания (INTR) (операция 5) или при участии сигнала захвата (HLD) (операция 6).

Любую выполняемую системой операцию можно отнести к одному из семи названных типов. Однако вовсе не обязательно, чтобы система выполняла все эти операции; существуют весьма эффективные специализированные системы, выполняющие только три или четыре операции. Обязательными для любой компьютерной системы являются операции 1, 2 (чтение, запись) и 7 (дешифрация команд и обработка данных внутри процессора). Эти операции выполняются любым процессором.

3.3. Операции внешних пересылок

В процессе работы компьютера пересылка информации по одной и той же магистрали в один и тот же момент времени происходит только между двумя устройствами **по принципу «точка-точка», «активный-пассивный».** Одно из этих устройств является активным (*ведущим, задатчиком, управляющим*), другое – пассивным (*исполнителем, ведомым, управляемым*).

Активное устройство начинает процедуру обмена и управляет ею. Пассивное устройство выполняет предписания активного. Обычно активным является процессор (за исключением начальной фазы перехода на подпрограмму

прерывания и режима прямого доступа к памяти, в которых активно периферийное устройство, а процессор пассивен; п.4.7). Оперативная память – всегда пассивное устройство.

Передача данных происходит между процессором, с одной стороны, и памятью или ПУ, с другой; или, в режиме ПДП, – между ПУ и памятью. Возможна передача между двумя ПУ, тогда одно – задатчик, другое – исполнитель.

Физическим средством проведения внешних пересылок между органами ядра ЭВМ – процессором, памятью и контроллерами ПУ служит **системная магистраль** ЭВМ. Она состоит из трех шин (см. рис. 1.1, 1.5, 1.6):

шины данных (ШД),
шины адресов (ША) и
шины управления (ШУ).

Системные шины формируются на основе внешних (*собственных, локальных, «первичных»*) шин микрочипа процессора при помощи устройства шинного интерфейса, или системного контроллера. Он вырабатывает все необходимые, унифицированные сигналы системной магистрали, используя собственные, порой уникальные сигналы микрочипа процессора, выставляемые тем на своих внешних выводах («ножках»).

У персональных компьютеров *системной* считают чаще собственную, или так называемую «переднюю» шину процессора (от англ. Front Side Bus, FSB). Она соединяет процессор и набор микросхем (chipset, «чипсет»), формирующий, в свою очередь, другие системные шины компьютера, которые в этом случае именуют *шинами расширения* (п. 3.6, рис. 3.14).

Физически *шина* представляет собой несколько проводников (электрических линий, цепей), объединенных в единую магистраль. По каждому проводнику осуществляется передача одного двоичного разряда (бита) информации. Высокий уровень электрического потенциала соответствует обычно логической «1», низкий уровень – логическому нулю «0».

Проводники *шины данных* и передаваемые по ним сигналы обозначаются буквами D*n*-1 . . . D0, где *n* – число проводников, разрядов. Для разных процессоров величина n может принимать значения 8, 16, 32, 64.

Сигналы и проводники *шины адреса* обозначаются A*m*-1 . . . A0, *m* – число разрядов шины адреса: *m* = 16, 20, 24, 32, 64.

У процессоров Pentium разрядность шины данных *n* = 64, адреса *m* = 32. У обновленных типов процессоров Xeon и Pentium 4 (так называемой архитектуры IA-64), процессоров семейства K8 (архитектуры AMD64, от названия фирмы-разработчика Advanced Micro Devices) и ряда других *m* = 64.

Сигналы (линии) *шины управления* разнотипны и носят индивидуальные имена, отражающие функции сигналов и зависящие от типа магистрали (AGP, PCI, VESA, IESA). Типичными управляющими сигналами являются:

1. Чт, или MEMR – сигнал управления чтением памяти (Memory Read);
2. Зп, или MEMW –сигнал записи в память (Memory Write);
3. Вв, или IOR – сигнал ввода из порта (Input-Output Read);
4. Выв, или IOW – сигнал вывода в порт (Input-Output Write);
5. ЗПр, или INTR – сигнал прерывания, запрос прерывания (Interrupt);

6. ППр, или INTA – сигнал подтверждения прерываний (Interruption Acknowledge).

Существует и множество других сигналов, передаваемых по шине управления, – сигналы тактирования от кварцевого генератора, синхронизации, арбитража (захвата шин) и т.д.

Шина данных служит физическим средством для передачи данных между выбранным устройством и центральным процессором. Под *данными* понимаются *команды* и данные, подлежащие обработке, т.е. *операнды*. (В узком смысле *данными*, *собственно данными* называются только операнды.) Сигналы, указывающие на источник или приемник информации (адреса ячеек памяти и портов в/в), в системе с раздельными ША и ШД по шине данных не передаются. Единственная задача шины данных – обеспечить прохождение данных (команд и операндов) между ЦП и устройством, адрес которого задан шиной адреса (или, вообще, между активным и пассивным устройством).

Порция данных передается по ШД в параллельном формате, одновременно по всем задействованным для этого проводникам шины данных, например по всей ширине шины D*n*-1 . . . D0, если данное имеет *n* разрядов. Таким образом, число проводников в шине данных равно максимальной разрядности двоичного числа или кода, который может быть передан по ШД. Такая, *максимальная,* единица информации, которая может быть передана по шине данных в параллельном формате и обработана процессором в один прием, называется *машинным* или *физическим словом* или просто «словом». Длина машинного слова в битах равна числу линий (разрядности) шины данных процессора и обычно совпадает с разрядностью общих регистров процессора, ячеек памяти и регистров ввода-вывода ПУ. Слово данных МК PIC16F состоит из 8 бит, слово команд – из 14 бит, слово процессора Intel 8086 16 бит, процессора Intel Pentium – 64 бита. Итак, за одну операцию обмена данными по ШД может быть передано в параллельном формате одно машинное слово.

Процессоры (16-, 32- и 64-разрядные) могут адресовать в памяти и обрабатывать в один прием и меньшие, чем машинное слово, порции информации, и такие порции данных тоже называются *словами*. Например, 64-разрядный процессор может обрабатывать 32-, 16- и 8-битные слова. *Наименьшую* адресуемую процессором порцию информации (наименьшее слово) в памяти или ПУ принято называть *байтом* (англ. byte). Стандартный байт содержит 8 бит.

Итак, для 8-битных процессоров машинное слово равно байту, для 16-битных – двум байтам, для 32-битных – четырем байтам и т.д. В любом случае длина (разрядность) слова кратна байту. Обычно она составляет целую степень двойки: 2^k.

Существуют и дополнительные трактовки термина «слово» (п. 4.2).

Имеются процессоры, у которых внутренние регистры и внешняя шина данных имеют разную ширину. Таким был процессор Intel 8088. Он обладал 16-битными внутренними регистрами и 8-битной внешней шиной данных. Об этом процессоре говорили, что он 16/8-битный. Размер слова относился к числу битов его внутренних регистров процессора (16). Процессор Pentium является

32/64-битным: 32-битные внутренние регистры и 64-битная внешняя ШД, а размер его физического слова 64-бита. Процессоры R4000 фирмы MIPS, Alpha фирмы DEC, архитектур IA-64, AMD64 и ряд других – 64-битные (64-битные внутренние регистры и 64-битная внешняя ШД).

Таким образом, *базовыми единицами информации* (*типами данных*) процессора служат: биты, байты и слова. Из них байты и слова могут обрабатываться процессором в один прием, посредством одной машинной операции. Биты таким свойством не обладают. Для обработки отдельных битов приходится применять последовательность из нескольких машинных команд, хотя ряд процессоров имеют и бит-ориентированные команды – команды, обрабатывающие отдельные биты внутри многобитного данного.

В зависимости от выполняемой операции информация по ШД может передаваться в двух направлениях: от внешних устройств к ЦП (чтение, ввод, подтверждение прерывания) и от ЦП к внешним устройствам (запись, вывод). Следовательно, шина данных процессора является двунаправленной. Этим она отличается от шины адреса, по которой адрес передается всегда в одном направлении: от ЦП к дешифратору адреса внешних по отношению к ЦП устройств.

По **шине адреса** центральный процессор передает адрес той ячейки памяти или того порта периферийного устройства, с которым он будет обмениваться порцией данных – принимать или выдавать данное. Каждой ячейке и каждому порту присвоен определенный адрес, известный программисту. Этот адрес записывается в выполняемой ЦП программе и поступает в ЦП при выборке программы из памяти. По шине адреса информация передается всегда в виде кодовой комбинации двоичных сигналов (нулей и единиц). Она передается в параллельном формате (одновременно по всем проводникам ША). Таким образом, число проводников в ША равно разрядности двоичного адреса, передаваемого по ША. Передаваемый из ЦП адрес поступает на входы дешифраторов памяти и периферийных устройств, которые выбирают нужную ячейку или порт. В системе с раздельными шинами адреса и данных данные, которые будут пересылаться между адресованной ячейкой (или портом) и ЦП, по шине адреса не передаются. По шине адреса не передаются также электрические сигналы, определяющие тип внешней пересылки (чтение или запись, в память или в ПУ), начало и окончание пересылки и другие управляющие сигналы. Передача адреса памяти или ПУ и обеспечение тем самым правильного места пересылки данных является основной функцией системной шины адреса.

По **шине управления** ЦП передает электрические сигналы, определяющие тип и моменты начала и окончания нужной операции обмена информацией с внешними устройствами. Эти сигналы включают устройства на выполнение нужных операций. Например: сигнал Чт (чтение), выдаваемый процессором, включает память на выполнение операции чтения ячейки – открывает ячейку по чтению; сигнал Зп (запись) открывает целевую ячейку по записи в нее информации; сигнал Вв (ввод) служит для управления вводом данного из порта в процессор; сигнал Выв (вывод) – выводом данного из процессора в порт; ППр –

активизирует контроллер прерываний на выдачу им номера прерывания в процессор; и т.д. По ШУ процессор принимает также сигналы от внешних устройств, несущие запросы на обслуживание этих устройств процессором (запросы прерываний) или на другие действия (например, на переключение процессора в режим ПДП). По шине управления передаются также общие (стартовые и синхронизирующие) сигналы.

Управляющие сигналы передаются по шине управления, как правило, одиночными битами – электрическими потенциалами по отдельным линиям (цепям) шины управления. Этим ШУ отличается от шин данных и адреса, информация по которым передается байтами и словами.

Набор управляющих сигналов, передаваемых по ШУ, их временные диаграммы, электрические характеристики и наименования могут отличаться у разных ЭВМ. Однако перечисленные функции указанных выше сигналов одинаковы и не зависят от вида процессора и ЭВМ.

Способы назначения устройства обмена. Для проведения пересылки одной порции данных между ЦП и внешними по отношению к нему устройствами (памятью и ПУ) процессор должен назначить это устройство. Возможны два способа назначения устройства: *адресный и безадресный*. Адресное назначение осуществляется путем подачи процессором адреса устройства в системную шину адреса; оно используется при выполнении операций *чтения, записи, ввода и вывода* (1 - 4, вторая часть операции 5а – чтение и 5б). *Безадресное назначение* устройства происходит при *подтверждении прерывания процессором* (первая часть операции 5а – ввод номера прерывания).

Потоки данных, проходящие по системной магистрали во время проведения *адресных внешних пересылок* (чтения, записи, ввода и вывода), показаны на рис.3.5, а во время *безадресной пересылки* (подтверждения прерывания) – в нижней части рис.3.5. **Временные диаграммы сигналов** в этих операциях изображены на рис.3.6-3.8. [14]

Чтение данного из памяти (операции 1 и 5а – чтение). На рис.3.5 показаны направления потоков информации при выполнении операции чтения из памяти системы, а на рис.3.6 – временная диаграмма цикла чтения. Эта диаграмма должна соблюдаться при выполнении операции чтения данных из полупроводникового запоминающего устройства (ЗУ). Формировать временную диаграмму должен ЦП (или другое активное устройство, например контроллер ПДП – в режиме ПДП). На рис.3.6 показаны электрические сигналы, которые должно получать ЗУ при считывании данных независимо от того, микропроцессор какого типа используется в составе ЦП.

Шина адреса обеспечивает передачу правильного адреса той ячейки памяти, из которой должны считываться данные. Затем на шину управления подается сигнал Чт (Чтение). Пока он имеет активный уровень (низкий), данные поступают из адресованной ячейки на шину данных и остаются стабильными, а микропроцессор считывает двоичную информацию с шины данных и записывает ее во внутренний регистр. Через определенное время сигнал Чт сбрасывается и передача данных прекращается.

Запись данных в память. На рис. 3.5 показаны потоки данных, а на рис. 3.7 – временная диаграмма цикла записи данных в полупроводниковое ЗУ (операции 2 и 5б). При записи должна строго выдерживаться временная последовательность подаваемых сигналов, определяемая типом используемого ЗУ. Формировать временную диаграмму должен ЦП (активное устройство). На рис. 3.7 показаны электрические сигналы, которые должно получать ЗУ при записи данных независимо от того, микропроцессор какого типа использован для построения ЦП.

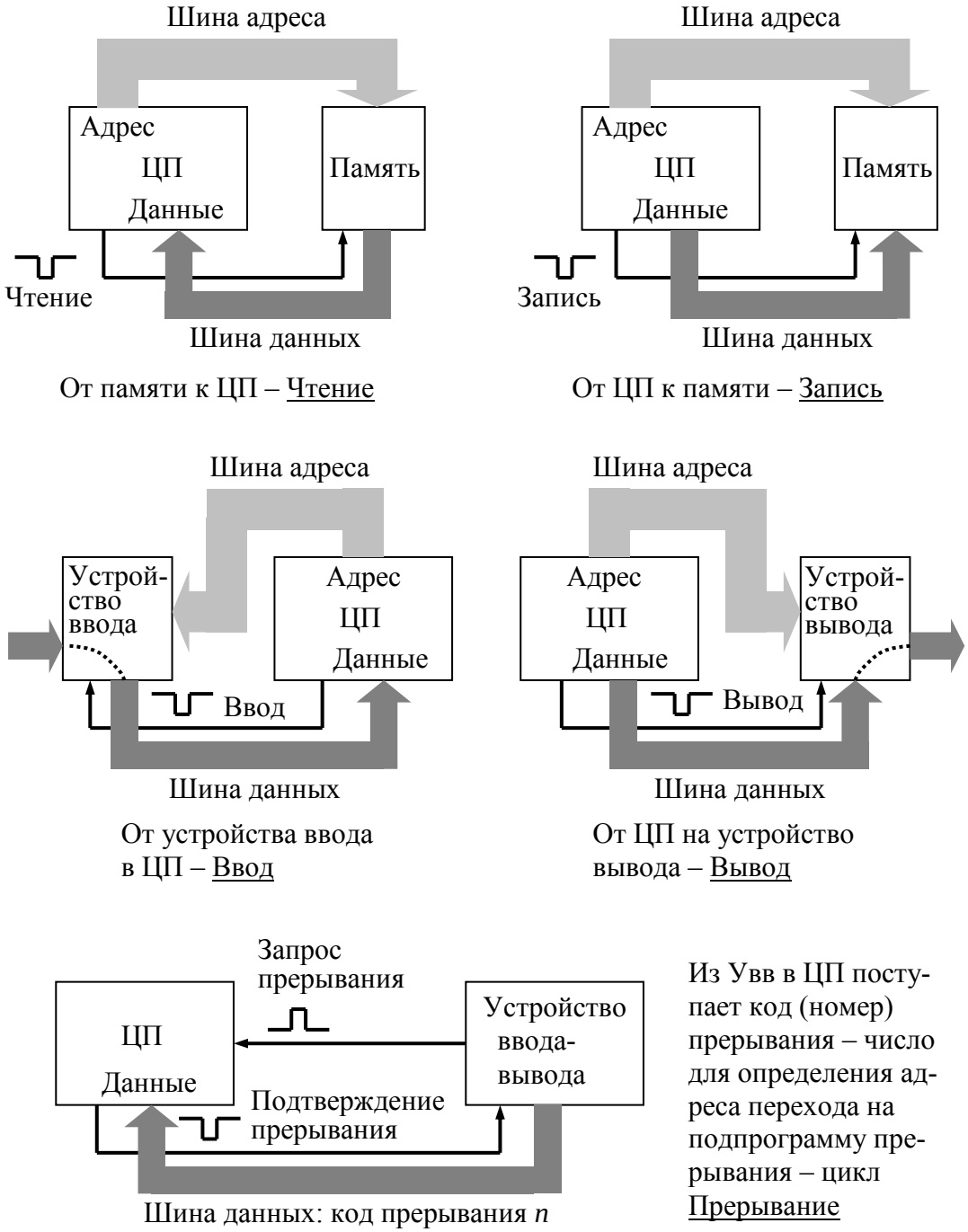


Рис. 3.5. Пять операций обмена данными в компьютере [13]

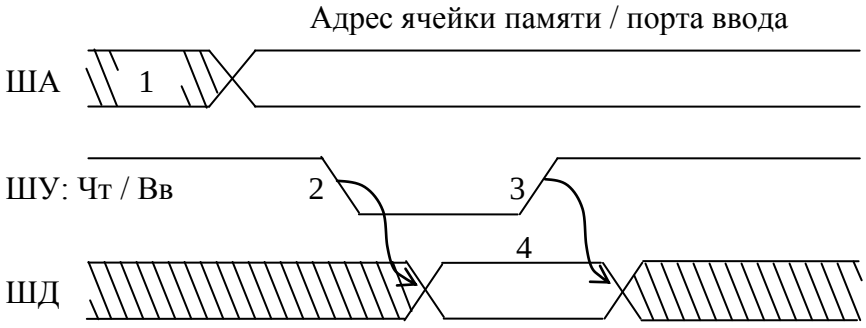


Рис. 3.6. Временная диаграмма циклов чтения данных из памяти и ввода из ПУ.
Для цикла чтения из памяти: 1 – адрес поступает в память; 2 – сигнал управления установлен (разрешена выдача данных из памяти на шину данных); 3 – сигнал управления сброшен; 4 – данные на выходе памяти стабильны.
Для цикла ввода из ПУ: 1 – на шине адреса установлен адрес порта ввода данных; 2 – на шине управления установлен сигнал разрешения ввода данных на системную шину данных (начата передача данных); 3 – передача данных завершена; 4 – стабильная входная информация.

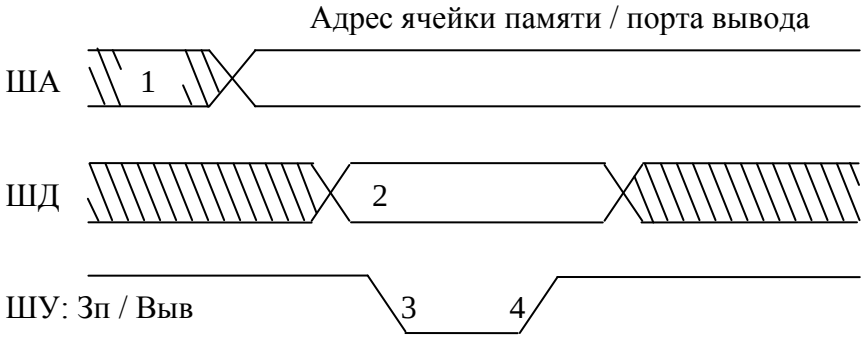


Рис. 3.7. Запись данных в память и вывод в ПУ.
Для цикла записи в память: 1 – адрес поступает на схемы памяти; 2 – на входы данных памяти подана стабильная информация; 3 – выдача сигнала Зп (данные на шине данных стабильны); 4 – сброс сигнала Зп (передача данных завершена).
Для цикла вывода в ПУ: 1 – на адресной шине установлен адрес порта вывода; 2 – на шине данных установлена передаваемая в ПУ информация; Выв – сигнал управления передачей данных в выбранный порт вывода

При выполнении операции записи по ША передается адрес той ячейки памяти, в которую ЦП запишет данные. Затем ЦП выдает в ШД данные. Во время выполнения операции записи сигнал управления записью Зп не подается на ШУ до тех пор, пока ЦП не подаст стабильные данные на ШД. Через фиксированный промежуток времени сигнал Зп сбрасывается. Запоминающее устройство использует сигнал Зп для инициирования операции приема данных из ШД в адресованную ячейку. Сигнал на шине управления остается стабильным лишь ограниченное время, меньшее, чем время выполнения передачи данных по шине данных.

Ввод данных из ПУ (из порта ввода) (операция 3). При вводе данных из порта ввода периферийного устройства Пр принимает данные из любого уст-

ройства, кроме ЗУ, например с оптического диска, клавиатуры, флэш-карты, аналого-цифрового преобразователя или из другого устройства, которое может пересылать данные в центральный процессор (см. рис.3.5). Различие между операциями ввода данных из ПУ и чтения из памяти заключается в том, что для этих операций используются разные сигналы системной шины управления.

Временная диаграмма цикла ввода данных из ПУ аналогична диаграмме цикла чтения из памяти (рис. 3.6). Функцией адресной шины является обеспечение правильного электрического соединения процессора с внешним устройством. В цикле ввода шина адреса обеспечивает подачу адреса порта ввода на адресные входы дешифратора периферийных устройств. Затем на системную шину управления подается сигнал ввода Вв. Пока его уровень остается активным, данные поступают из ПУ (из назначенного порта) на системную шину данных. Пр принимает эту двоичную информацию с шины данных и записывает ее в свой внутренний регистр. Через определенное время сигнал Вв сбрасывается, и передача данных заканчивается.

Вывод данных в ПУ (в порт вывода) (операция 4). Устройством вывода данных может быть любое устройство кроме ЗУ, воспринимающее передаваемую микропроцессором информацию, например экран монитора, оптический диск, флэш-карта, цифро-аналоговый преобразователь, др. устройства.

Показанная на рис. 3.7 временная диаграмма описывает взаимодействие Пр с устройством вывода данных (см. рис. 3.5). Оно аналогично операции записи в память. ША обеспечивает подачу правильного адреса порта. ШД используется для передачи данных с выхода Пр на информационный вход устройства вывода. Затем на ШУ подается сигнал управления выводом (активный уровень) и начинается передача данных.

Некоторые ЭВМ, например, на базе процессоров 1839ВМ1Ф (одного из ведущих российских разработчиков и производителей микроэлектроники Ангстрем) или процессоров J11, Dec MicroVAX-11 (корпорации Digital Equipment Corporation) выполняют ввод подобно чтению из ячейки, и для них эти операции абсолютно идентичны. Так же идентичны для них операции вывода и записи.

Подтверждение прерывания. В этой операции (относящейся к типу 5а, п. 3.3) периферийное устройство сообщает центральному процессору код адреса перехода на подпрограмму обслуживания прерывания (например, номер прерывания *n* в первой части операции 5а). Потоки информации в этой операции показаны на рис. 3.5, а временная диаграмма сигналов – на рис. 3.8. Формирует эту диаграмму центральный процессор.

Данная операция – «цикл подтверждения прерывания» (или просто цикл «Прерывание») генерируется процессором по сигналу запроса прерывания ЗПр (INTR – Interruption), поступающим на процессор от периферийного устройства. Сначала, после поступления ЗПр, процессор закачивает текущую команду, а потом, если прерывания разрешены, информирует периферийное устройство (чаще – контроллер прерываний) о переключении в режим прерывания, выставляя на линию подтверждения прерывания ППр (INTA – Interruption

Acknowledge) активный уровень напряжения. По переднему фронту сигнала ППр периферийное устройство (контроллер прерываний) сообщает процессору по шине данных код адреса перехода на подпрограмму прерывания (номер прерывания *n*), после чего процессор снимает сигнал ППр, а ПУ прекращает передачу по шине данных. На этом пересылка в цикле прерывания заканчивается. Дальнейшие действия процессора по обработке прерывания состоят в определении адреса перехода на подпрограмму прерывания на основании его кода (номера *n*), сохранении контекста программы в стеке, переходе на подпрограмму и ее выполнении (глава 5).

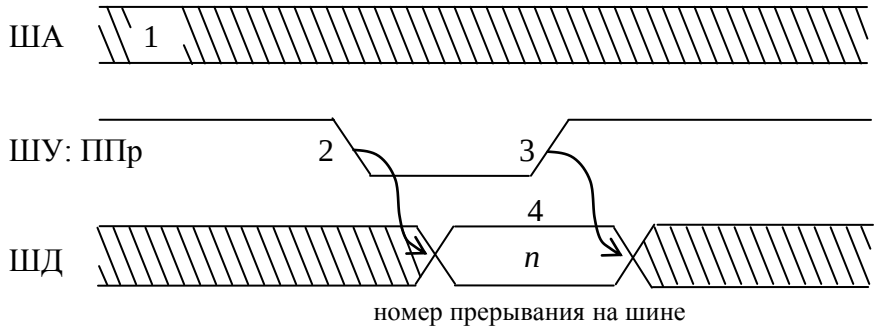


Рис. 3.8. Временная диаграмма цикла обработки прерывания.
1 – никакой адрес в шину адреса не поступает (безадресное назначение устройства процессором); 2 – сигнал подтверждения прерывания установлен; 3 – сигнал ППр сброшен; 4 – код адреса перехода (номер прерывания *n*) на выходе периферийного устройства (контроллера прерываний) стабилен

Обратите внимание, что процессор назначает устройство обмена в цикле прерывания *безадресным способом*, т.е. не выставляет никакого истинного адреса в шину адреса.

3.4. Машинные циклы

Следующим по иерархии компонентом рабочего процесса, после машинной операции, является **машинный цикл.** Он состоит из одной или нескольких операций. Каждый МЦ соответствует одному обращению процессора к памяти или периферийным устройствам и некоторым (но не обязательно) внутренним операциям процессора над данными, выполняемым после этого обращения к шине и до начала следующего. К внутренним операциям могут относиться: расшифровка считанного из памяти кода команды, сложение операндов, сдвиг числа на определенное количество бит и т.п. Если внутренние операции в машинном цикле отсутствуют, то он сводится к одной пересылке по системной магистрали. Вместо термина «машинный цикл» используются также термины «цикл обращения к магистрали», «цикл обращения к каналу ЭВМ», «цикл магистрали», «цикл шины», «магистральный цикл», «шинный цикл». Длительность машинного цикла может варьироваться в определенных пределах в зависимости от модели процессора и выполняемой операции, но она всегда равна

целому числу периодов (или долей периода) следования синхроимпульсов тактового генератора – тактов *Т*.

Из машинных циклов складываются, в свою очередь, командные циклы, затрачиваемые на выполнение отдельных команд, а из командных циклов составляется законченный программный процесс (п.3.2 и 4.6).

В соответствии со сказанным весь интервал времени функционирования ЭВМ от пуска до останова подразделяется на целое число дискретных промежутков времени – машинных циклов, как показано на рис.3.9.

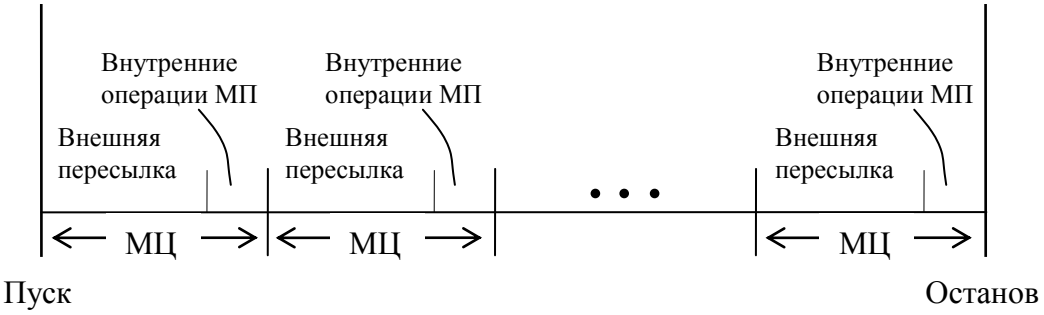


Рис. 3.9. Последовательность машинных циклов при работе ЭВМ

Итак, *машинный цикл* – это действия процессора:

а) по выполнению *одной внешней пересылки* по шине данных, к которой могут добавляться

б) *внутренние операции* процессора по обработке данных, выполняемые на его собственных узлах после завершения данной пересылки и до начала следующей.

Машинным циклом называют также дискретный промежуток времени выполнения этих действий.

Понятие машинного цикла, в отличие от базовых операций, включает в себя не только содержание, но и форму проведения операций. Оно отражает тот факт, что обработка одной порции данных в компьютере организована в виде унифицированной последовательности событий, распределенных по шинам и блокам процессора и компьютера, разбитых на дискретные такты и синхронизированных с помощью тактовых импульсов. Машинный цикл представляет собой единообразный способ взаимодействия устройств компьютера между собой и реализации ими машинных операций.

Поскольку внутренние операции протекают скрыто от пользователя и не отражаются на состоянии внешних выводов микросхемы, то для пользователя машинный цикл выглядит как одна внешняя пересылка, т.е. как один цикл шины, шинный цикл. Соответственно весь процесс функционирования процессора от пуска до останова представляется на системной магистрали в виде последовательности шинных циклов – внешних пересылок данных между процессором и памятью или периферийными устройствами.

Внутренние операции требуют для своего проведения определенного времени. Для этого процессор может вводить в машинный цикл дополнительные временные задержки («холостые» такты). В холостых тактах никакие

внешние пересылки по ШД процессором не производятся, и он вообще отключается от внешних шин. Холостые такты обычно добавляются в конце машинных циклов. У современных высокопроизводительных процессоров, располагающих параллельно работающими операционным, управляющим и интерфейсным устройствами, внутренние операции по обработке ранее введенных команд и данных осуществляются одновременно с выборкой последующих команд или пересылкой данных обрабатываемых команд (так называемая «конвейерная обработка команд»). Поэтому необходимость во введении холостых тактов у таких процессоров возникает крайне редко и фактически не влияет на производительность процессора.

По существу термин «цикл канала», «цикл шины» эквивалентен понятию машинного цикла процессора. Однако в многопроцессорных системах этот термин несколько шире в том смысле, что цикл канала может инициировать не только центральный процессор, но и, например, процессор ввода-вывода или арифметический сопроцессор.

На основе семи типов машинных операций, рассмотренных в п.3.2, процессор формирует *пять типов машинных циклов*, отличающихся производимыми в них *внешними пересылками*:

1) машинные циклы чтения одной порции данных из памяти – циклы типа «Чтение»;
2) машинные циклы записи данного в память – циклы «Запись»;
3) циклы ввода данного в Пр из порта ПУ – циклы «Ввод»;
4) циклы вывода данного из Пр в порт ПУ – «Вывод»;

4') некоторые модификации указанных типов, например, циклы типа «чтение данного из памяти - преобразование данного - запись результата по тому же адресу». Подобные циклы выполняются процессорами 1839ВМ1Ф, J11, MicroVAX-11 и др. В них адрес выставляется в шину адреса один раз, а данные передаются по шине данных дважды – при чтении, и при записи.

5) машинные циклы подтверждения прерывания, предназначенные для ввода в процессор из ПУ (из контроллера прерываний) кода пускового адреса подпрограммы прерывания или номера прерывания *n* – циклы «Прерывание», или «Подтверждение прерывания».

Все внутренние операции процессора по обработке данных (операции типа 7) выполняются в течение названных типов машинных циклов: в холостых тактах этих циклов или одновременно с выполнением внешних пересылок следующих циклов.

В режиме ПДП (операция 6) данные передаются между основной памятью и периферийным устройством, минуя ЦП; ЦП на это время отключается от системной шины и не участвует ни в каких пересылках. Следовательно, можно считать, что ПДП проводится в промежутках *между* машинными циклами процессора.

Машинные циклы можно различать дополнительно по смыслу проводимых в них *внутренних операций* над данными. Например: цикл «Чтение команды», в котором команда выбирается из памяти и расшифровывается процессо-

ром; цикл «Чтение операнда», используемый для чтения операнда из памяти и, например, сложения его с содержимым регистра процессора; цикл «Запись операнда в стек», вовлекающий в работу специальный операционный регистр процессора – указатель стека для адресации стековой памяти с последующей записью в нее операнда; и т.д.

Циклы можно различать также по *размеру передаваемой в них порции данных*, например: «Ввод байта», «Запись слова» и т.д.

Временные диаграммы машинных циклов выглядят так же, как и диаграммы соответствующих внешних пересылок (см. рис.3.6-3.8), но с разбиением на такты. Внутренние операции не проявляются на состоянии внешних выводов микросхемы и на временных диаграммах не отображаются. К диаграммам могут добавляться лишь холостые такты, в которых процессор выполняет внутренние операции.

Наряду с временными диаграммами сигналов действия процессора в машинных циклах удобно описывать при помощи условных символьных выражений – **логических диаграмм**. Они представлены на рис.3.10-3.12.

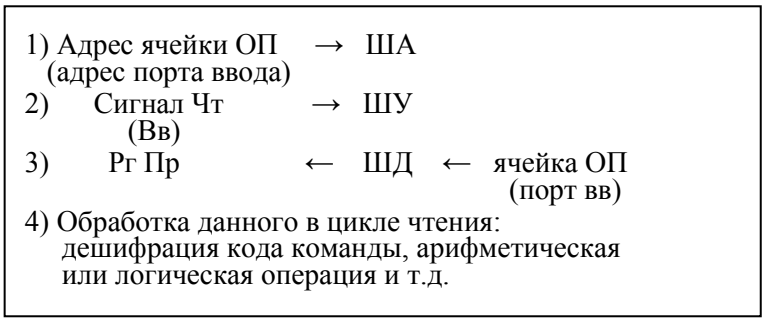


Рис. 3.10. Действия МП в машинных циклах «Чтение» и «Ввод»

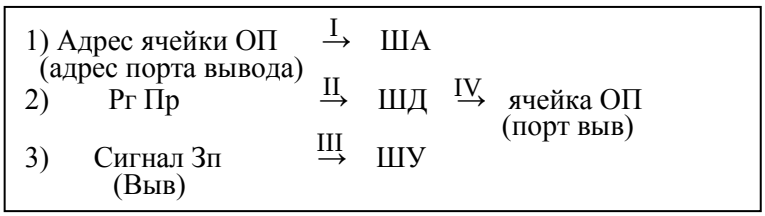


Рис. 3.11. Действия МП в машинных циклах «Запись» и «Вывод».
I – первая операция, II – вторая, III – третья, IV – четвертая

Согласно диаграммам на рис.3.10 и 3.11 и сказанному ранее каждое обращение МП к ячейке памяти (по чтению или записи) или к порту в/в (по вводу или выводу), т.е. *каждый машинный цикл адресного типа* (1 - 4) всегда начинается с того, что:

1) микропроцессор выставляет адрес ячейки (или порта), к которой он будет производить обращение в этом цикле, в шину адреса и
2) обращается к памяти (или периферийному устройству) управляющим сигналом, формируемым по одной из линий шины управления для инициирования нужной операции чтения или записи (ввода или вывода);

3) после этого происходит передача данного по шине данных между процессором и адресованной ячейкой (портом) в нужном направлении: при чтении и вводе – из устройства в процессор, при записи и выводе – из процессора в устройство, причем при записи и выводе процессор сначала выдает данное, и только потом – сигнал записи или вывода (это делается с целью стабилизации данного на шине к моменту занесения его во внешнее устройство);
4) при необходимости в машинном цикле процессор производит внутренние операции над данными. Для этого он может вводить в цикл дополнительные, «холостые» такты.

Подобные действия производятся в *адресных* машинных циклах (типов 1-4). При выполнении же *безадресного* цикла (типа 5) шина адреса не используется, а передача данного происходит под управлением одного только сигнала подтверждения прерывания ППр (см. рис.3.12).

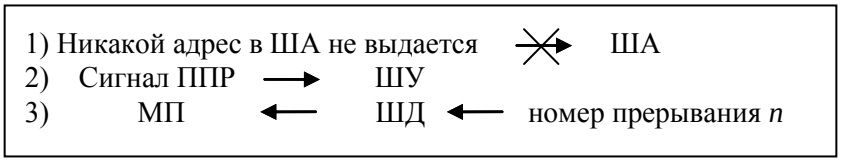


Рис. 3.12. Действия МП в машинном цикле «Прерывание»

3.5. Важное заключение

В заключение необходимо сказать, что основное внимание было уделено нами последовательности событий; временные диаграммы при этом можно описывать в общем виде, поскольку важнее понять состав и назначение вырабатываемых сигналов и общую последовательность их формирования, чем углубляться в рассмотрение конкретных временных характеристик и их точных значений. Понимание принципа действия позволит сравнивать различные системные сигналы; при необходимости можно использовать детальный анализ временных соотношений.

При разработке большинства схем интерфейса микропроцессорной системы не обязательно проводить детальный анализ временных диаграмм. Если придерживаться некоторых простых правил, интерфейсы МП систем можно разрабатывать и отлаживать, зная общую последовательность действий, выполняемых при работе интерфейса. Эти правила были рассмотрены в приведенных положениях по организации интерфейса МП систем.

Возвращаясь к приведенным временным диаграммам, можно отметить, что выполнением всех временных соотношений управляет центральный процессор, т.е. при выполнении каждой операции микропроцессор будет вырабатывать в соответствующие моменты времени сигналы, подаваемые на каждую из трех шин. Поэтому, при разработке внутримашинного интерфейса можно

просто использовать шины, полагая, что правильность выполнения временной диаграммы работы системы обеспечивается микропроцессором [14].

3.6. Эволюция шинной архитектуры

Наиболее наглядно и полно можно проследить проблемы и тенденции развития систем передачи данных при рассмотрении ретроспективы эволюции интерфейсов и структур этих систем на примере персональных компьютеров типа IBM PC (рис. 3.13) [7].

В начале эры современных персональных компьютеров, приходящейся на начало 80-х годов 20 века, частота работы процессора составляла 10 МГц, при этом на выполнение даже самых простейших операций процессор затрачивал несколько тактов. В таких условиях для обеспечения бесперебойной работы процессора, например i8088, было достаточно всего 4 миллионов обращений к памяти в секунду, что соответствовало циклу работы в 250 нс. Этим условиям удовлетворяла одношинная структура систем передачи данных, когда все устройства компьютера, включая ОЗУ, общались с процессором через общую шину (рис. 3.13, *а*), которую называли системной. Все интерфейсы ПУ подключались к этой шине. Наиболее распространенной системной шиной в этот период стала сначала 8-разрядная, затем 16-разрядная шина ISA, работающая на частоте 8 МГц. Впоследствии с появлением 32-разрядных процессоров была разработана новая шина MCA (полностью несовместимая с ISA), а также версия расширения шины ISA – EISA.

С ростом частоты работы ПК и изменения времени доступа к ОЗУ пропускная способность шин ISA, MCA, EISA стала тормозить работу процессора. Решение проблемы нашли в выделении канала передачи данных Пр - ОЗУ в отдельную шину, построенную на базе внешнего интерфейса Пр, и изолированную от медленной шины ISA/MCA/EISA посредством контроллера шины данных. Это повысило производительность работы центрального процессора. Все ПУ продолжали взаимодействовать с центральным процессором через системную шину (рис. 3.13, *б*).

С дальнейшим ростом частоты работы Пр тормозом в работе стало ОЗУ. Тогда ввели дополнительную высокоскоростную кэш-память, что уменьшило простои Пр. На определенном этапе развития компьютеров стали широко использовать мультимедиа. Сразу выявилось узкое место во взаимодействии центрального процессора и видеокарты. Имеющиеся системные шины ISA, MCA, EISA не удовлетворяли этим условиям.

Выход был найден с разработкой и внедрением высокоскоростных локальных шин, посредством которых можно было связаться с памятью, на этой же шине работали жесткие диски, что также повышало качество вывода графической информации. Первой такой шиной была шина VLB (VL-bus), практически повторявшая интерфейс процессора i486. Затем появилась локальная шина PCI. Она была процессорно независимой и поэтому получила наибольшее распространение для последующих типов процессоров. Эта шина имела частоту работы 33 МГц и при 32-разрядных данных обеспечивала пропускную способность в 132 Мбайт/с (рис. 3.13, *в*). Системная шина ISA/MCA/EISA по-прежнему использовалась в компьютерах, что позволяло применять в новых компьютерах огромное количество ранее разработанных аппаратных и программных средств.

В такой системе передачи данных различные ПУ подключались к разным шинам. Медленные – к ISA/MCA/EISA, а высокоскоростные – к PCI. С появление шины PCI стало целесообразным использовать высокоскоростные параллельные и последовательные интерфейсы ПУ (SCSI, ATA, USB). На этом этапе системной стали называть шину процессора, через которую он взаимодействовал с ОЗУ. Шина PCI и ISA/MCA/EISA и подобные другие назвали шинами ввода-вывода или шинами расширения. Действительно, эти шины как бы расширяли число устройств, работающих с ЦП, и их основной функцией стало обеспечение процессов ввода и вывода информации.

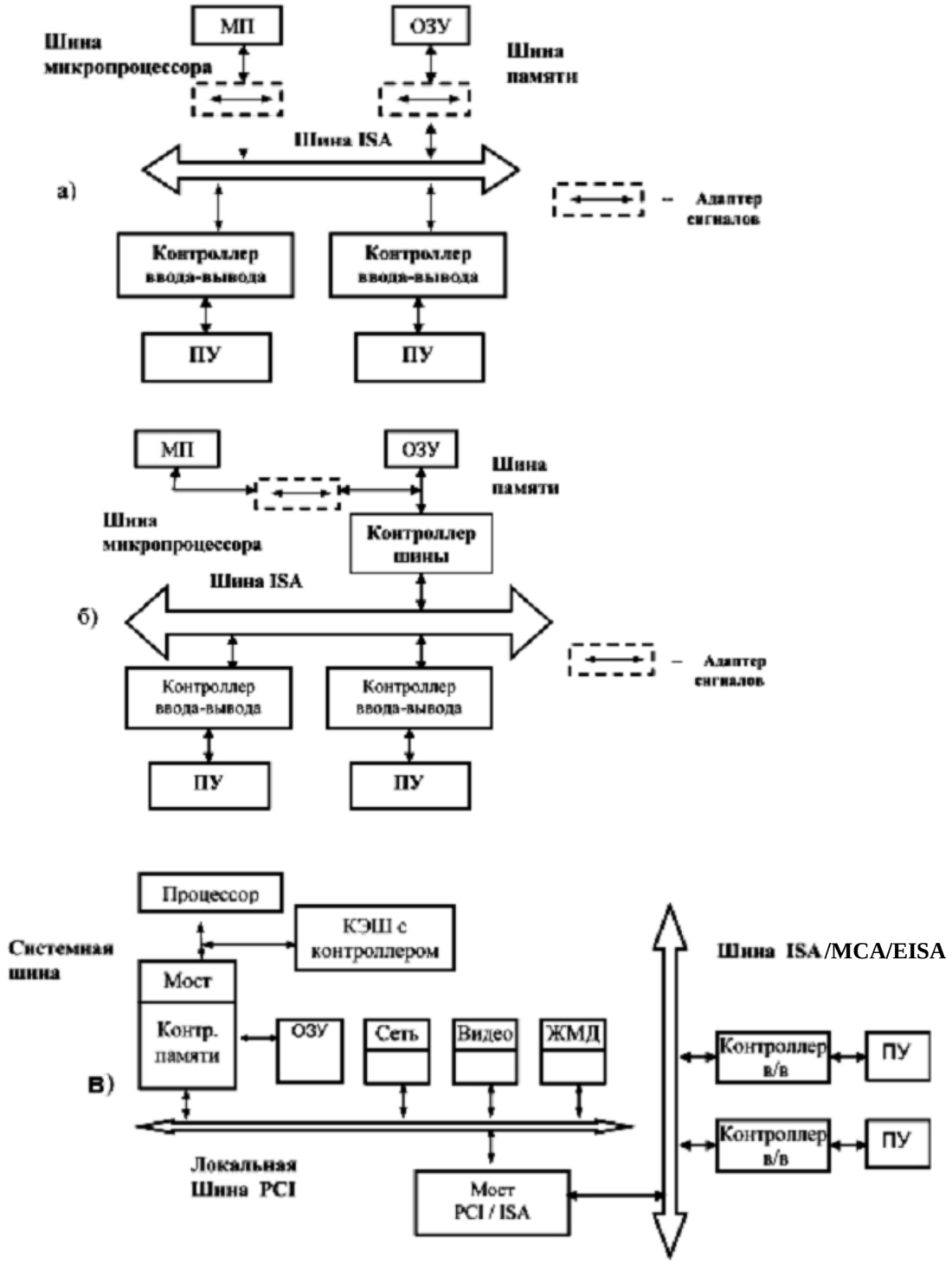


Рис. 3.13. Эволюция шинной архитектуры

Появление шины PCI не сняло всех проблем по качественному выводу визуальной информации для 3-мерных изображений, «живого» видео. Здесь уже требовались скорости в сотни Мбайт/с. В 1996 г. фирма Intel разработала новую шину AGP, предназначенную только для связи ОЗУ и процессора с видеокартой монитора. Эта шина обеспечивает пропускную способность в сотни Мбайт/с. Она непосредственно связывает видеокарту с ОЗУ минуя шину PCI.

Таким образом, спустя годы снова пришли к многомагистральной структуре ввода-вывода с радиально-магистральными интерфейсами ПУ. Все шины систем передачи данных объединяются в единую транспортную среду передачи информации с помощью специальных устройств – мостов.

Мост – связующий компонент, устройство, применяемое для объединения шин, использующих разные или одинаковые протоколы обмена (рис. 3.14). Мост – это сложное устройство, которое осуществляет не только коммутацию каналов передачи данных, но и производит управление соответствующими шинами.

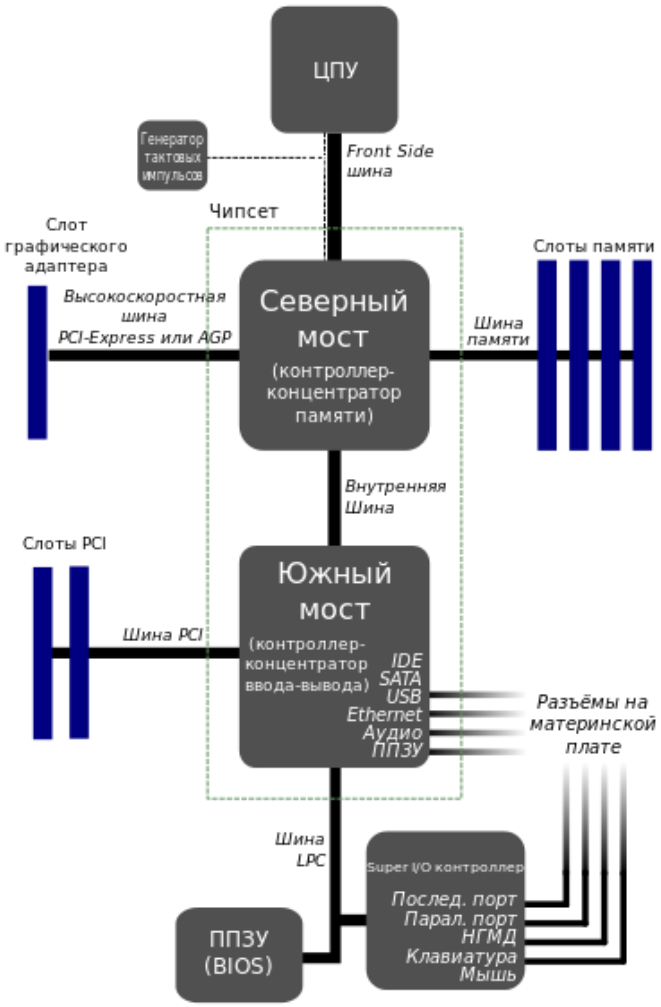


Рис. 3.14. Схематическое изображение материнской платы.
Front Side Bus (FSB) – собственная, или, буквально, «передняя» шина процессора

В общем случае, для обеспечения выполнения функций интерфейсов, входящих в систему ввода-вывода, применяются специальные контроллеры и схемы. К ним можно отнести контроллеры прерываний и прямого доступа к памяти, таймер, часы реального времени, буферы шин данных, дешифраторы, мультиплексоры, регистры и другие логические устройства. В первых компьютерах, построенных с использованием микропроцессоров, контроллер и другие устройства строились на базе набора интегральных схем малой, средней и большой степени интеграции. Адаптеры, таймер и др. выпускались в виде отдельных микросхем (8250, 8255, 8259, 8237 и т.д.).

С повышением производительности компьютеров и увеличением степени интеграции все вышеперечисленные устройства и схемы стали объединяться в микросхемы со сверхбольшой степенью интеграции. Набор таких сверхбольших многофункциональных микро-

схем или одна такая микросхема называется «чипсет» (chipset). Мосты и контроллеры, входящие в chipset, осуществляют управление всеми потоками передаваемых данных. Именно chipset определяет основные особенности общей архитектуры компьютера и, соответственно, достигаемый уровень производительности в условиях, когда лимитирующим фактором становится не процессор, а его окружение – память и система ввода-вывода. Типичный чипсет персонального компьютера состоит из двух мостов – северного и южного (п. 1.4), размещаемых на материнской плате (см. рис. 3.14).

Глава 4. ПОРЯДОК РАБОТЫ КОМПЬЮТЕРА

4.1. Программный процесс

В предыдущей главе мы познакомились с базовыми машинными операциями, выполняемыми процессором на своих внутренних узлах и внешней шине при взаимодействии с другими устройствами ЭВМ. Мы убедились, что из базовых операций процессор формирует машинные циклы, представляющие собой способ дискретизации его работы во времени. За один машинный цикл процессор совершает *одну внешнюю пересылку* по системной магистрали ЭВМ и, возможно, некоторые *внутренние операции* по обработке данных на собственных узлах. Машинные циклы формализуют проведение внешних пересылок. Они сводят их к однотипным действиям на системной магистрали. Машинные циклы унифицированы и понятны для всех устройств ЭВМ. Они являются универсальным средством обмена информацией между ними и выступают своего рода элементарными строительными кирпичиками, системными операционными квантами, из которых строится вся работа ЭВМ. В настоящей главе мы обсудим порядок упомянутого строительства. Мы увидим, как при работе ЭВМ под управлением машинной программы элементарные *машинные операции* и формируемые из них *машинные циклы* (строительные кирпичики) складываются в *командные (рабочие) циклы* процессора, затрачиваемые на выполнение отдельных машинных команд, а те, в свою очередь, следуя один за другим, выстраиваются в целостную процедуру выполнения *машинной программы* целиком. Это означает, что машинная программа выполняется в ЭВМ в виде последовательности простых событий – машинных операций, т.е. процесса. Соответственно, *процессом* в информатике называют и саму программу в стадии выполнения.

4.2. Информация в ЭВМ. Программа и данные. Структура команды

Представление информации в компьютере. Любая информация (числа, команды, тексты и т.д.) представляются в компьютере в виде двоичных кодов (двоичных чисел, двоичных слов) фиксированной или переменной длины. Отдельные, минимальные элементы двоичного кода, имеющие значение 0 или 1 (двоичных цифр), называют *разрядами* (двоичными) или *битами* (от англ. сокращения Binary digiT, bit – двоичная цифра). В компьютере слова часто разбивают на части, называемые *слогами* или *байтами*. (Слово «байт», англ. «byte» представляет собой сокращение словосочетания BinarY TErm — «двоичный терм».) Байт понимается как *наименьшая* порция информации, адресуемая процессором в памяти или периферийных устройствах, передаваемая по шине данных в параллельном формате и обрабатываемая процессором одномоментно, в один прием (п.3.3). В современных компьютерах используется байт, состоящий из 8 битов. Однако в истории компьютерной техники существовали решения с иными размерами байта (например, 6, 32 или 36 битов), поэтому иногда в ком-

пьютерных стандартах и официальных документах для однозначного обозначения группы из 8 битов используется термин «октет» (лат. *octet*).

Понятие слова является менее определенным, чем байт. В содержательном отношении *словом* называют любую единицу информации в компьютере, имеющую законченный логический смысл и воспринимаемую как целостное, не расчленяемое значение. Например: многобайтное число, машинную команду, слово русского языка в тексте, записанное с помощью ASCII-кодов* и т.д. Обычно подобное слово состоит из целого числа байтов, в частном случае из одного байта, хотя никому не возбраняется разместить, например, какое-нибудь двоичное число по нецелому числу байтов, но такое число-слово неудобно обрабатывать программно.

***** ASCII (англ. American Standard Code for Information Interchange) – двоичные коды символов внешнего алфавита (латинских и национальных букв, десятичных цифр, знаков препинания) и управляющих символов (возврат каретки, перевод строки, звонок и др.).

Иную трактовку понятия слова связывают не с содержательным аспектом, а с разрядностью процессора. *Словом* (*машинным*, *физическим словом*) называют в этом случае *наибольшую* порцию информации, передаваемую по шине данных в параллельном формате и обрабатываемую процессором в один прием (п.3.3). Поэтому размер машинного слова равен ширине шины данных, и в байтах составляет обычно целую степень 2. В IA-32 компьютерах машинное слово состоит из восьми байт (64 бита). Таким образом, если *байт* – это *наименьшая адресуемая* порция информации в памяти или ПУ, то *машинное слово* – *наибольшая*. *Словом* называют также и вообще любую порцию информации, передаваемую по ШД и обрабатываемую в один прием (например, байт) (п.3.3).

Иногда размер слова определяют некоторым условным стандартом, единым для целого класса или семейства процессоров. Так, для процессоров IA-32 *словом* называют не 64-битное, а 16-битное значение. Соответственно 32-битное значение называют *двойным словом*, а 64-битное – *учетверенным словом*, чтобы обеспечить совместимость терминологии для всего семейства. 80-битное (10-байтное) данное (поддерживаемое встроенными в Pentium и в последующие модели арифметическими сопроцессорами FPU) – tbyte (ten byte – десять байт). Такая терминология удобна для программистов, не заботящихся об особенностях физического представления данных в компьютере.

Физические носители информации. Вся информация в компьютере представляется двоичными числами – комбинациями нулей и единиц. Позиция в двоичном числе, отводимая для записи нуля или единицы, называется двоичным разрядом или битом. Физическим носителем двоичного разряда в компьютере служит некоторое техническое устройство, двум различным состояниям которого приписываются значения 0 и 1. К такого рода устройствам относятся триггеры, конденсаторы, магниты, оптические элементы. В триггере – электронном элементе с двумя устойчивыми состояниями логический «0» представляется чаще всего сигналом от 0 до 1 В, а логическая «1» – сигналом от 2 до 5 В. Напряжение за пределами указанных величин недопустимо. В конденсаторе наличие заряда предполагает «1», а отсутствие – «0». На самом деле, отдельного конденсатора нет, а его роль выполняет обычно электрическая емкость между затвором и истоком полевого транзистора. Но для простоты изложения емкость выделена нами как бы в отдельный элемент. В магните «1» выражается его

намагниченностью, а «0» – размагниченностью или противоположной намагниченностью. Роль магнита выполняет крошечный участок тонкого ферромагнитного слоя, нанесенного на поверхность ленты или диска. «Нули» и «единицы» записываются на этот слой при помощи магнитного поля, создаваемого сложной микроскопической конструкцией (магнитной «головкой»), находящейся на самом кончике механической держателя, и располагаются по прямой дорожке ленты и по спиральной дорожке диска. В оптических дисках «0» и «1» записываются на поверхность диска (тоже вдоль спиральной дорожки) при помощи лазерного луча, формируемого оптической головкой: «0» – в виде мельчайшей выемки, впадины – «пита» (от англ. *pit* – «ямка», «углубление»), а «1» – в виде ровного участка на поверхности диска, или «ленда» (*land* – ровная поверхность) – промежутка между питами. Чтение производится с помощью считывающего лазерного луча, направляемого на вращающийся диск. При сканировании лазерный луч по-разному отражается от углубления и от ровной основной поверхности. Если от основной поверхности (land) свет отражается полностью, то в углублениях (pits) свет поглощается (из-за интерференции).

Поскольку оба состояния 0 и 1 указанных устройств устойчивы, т.е. мало подвержены воздействию различного рода помех, то эти устройства могут надежно сохранять один двоичный разряд числа, и по этой причине они используются в качестве *элементов памяти* компьютера. Для представления многоразрядного двоичного числа (байта или слова) используется упорядоченный набор (блок, линейка) таких устройств (элементов памяти), называемый *ячейкой памяти* (п.1.3). В частности, линейка триггеров образует так называемый *регистр* – электронное устройство для запоминания двоичного числа. Регистры используются в качестве собственных регистров процессора, портов ввода-вывода и, наряду с емкостными устройствами, ячеек основной памяти ЭВМ. Магнитные и оптические элементы применяются для создания устройств внешней, периферийной памяти ЭВМ – дисков и лент. Съемные накопители флэш-карты представляют собой разновидность полупроводниковой технологии электрически стираемой, перепрограммируемой, энергонезависимой памяти (EEPROM, англ. Electrically Erasable Programmable Read-Only Memory), информация в которой хранится в массиве транзисторов с плавающим затвором.

Байтная организация памяти. Ячейки основной памяти имеют порядковые *номера*, по которым их можно различать. Эти номера называются также *адресами ячеек*. Задавая их, можно обращаться к ячейкам посредством машинных команд. Собственные, уникальные, номера присваиваются однобайтным ячейкам – ячейкам размером в 1 байт (8 бит). Номера однобайтных ячеек принимают последовательные целочисленные значения от 0 до $2^m - 1$, где m – разрядность номера (адреса). В этом смысле говорят, что *память компьютера организована по байтам*. У процессора IA-32 $m = 32$. Поэтому количество однобайтных ячеек памяти может достигать 2^m = 4 Г, а их номера (адреса) изменяются от 00000000h до FFFFFFFFh.

Соседние однобайтные ячейки могут объединяться в более крупные – многобайтные ячейки, которые, как и однобайтные ячейки, могут адресоваться и обрабатываться процессором по одной машинной команде. Размер таких ячеек в байтах составляет обычно целую степень двойки: 2 байта, 4 байта, 8 байт. Многобайтные ячейки отдельно не нумеруются, а идентифицируются по уже назначенным адресам однобайтных ячеек. *Адресом многобайтной ячейки* считается младший из адресов составляющих ее однобайтных ячеек. Например, если число 1234h (длиной в 16 бит) хранится в двух однобайтных ячейках памяти с адресами 75 и 76, то адресом всего числа, всей двухбайтной ячейки считается 75. При этом многобайтное число хранится в памяти по правилу: *млад-*

шие байты по младшим адресам: «младшее по младшему адресу». В нашем примере байт 34h хранится по адресу 75, а байт 12h – по адресу 76: |34h| |12h| (рис. 4.1).

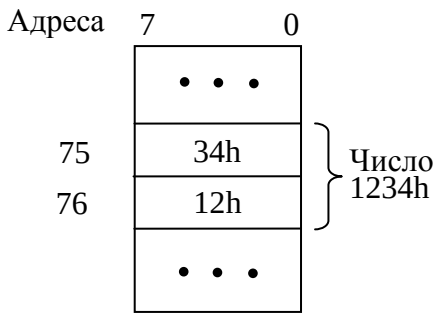


Рис. 4.1. Размещение длинного числа по байтам памяти: «младшее по младшему адресу»

Современные запоминающие устройства обычно имеют размер физической ячейки (максимальной по размеру ячейки, содержимое которой передается по шине данных ЗУ в один прием) 8 бит, 16 бит, 32 бита, 64 бита. В ранних ЭВМ использовались и более экзотические размеры, например 39 (БЭСМ-1) или 48 (БЭСМ-6). Длина ячеек микросхем ЗУ может оказаться короче машинного слова процессора. Тогда для приема/выдачи слова они соединяются параллельно по две, четыре или более микросхемы, каждая из которых подключается к своей группе линий шины данных (как это имеет место, например, в схеме на рис. 3.4).

Аналогично ячейкам памяти организованы и регистры ввода-вывода периферийных устройств. Они тоже имеют стандартную длину 1 байт (8 бит) и объединяются в более крупные, многобайтные, регистры по тем же правилам, что и ячейки памяти. Что касается собственных регистров процессора, то их длина составляет чаще всего $1, 2, 4, 8$ байт.

Программы и данные. Любую информацию, хранящуюся в памяти компьютера и обрабатываемую процессором, можно подразделить по смыслу на команды (программы) и операнды (данные). Команды указывают процессору, какие операции он должен выполнять, а операнды – это те числа, над которыми эти операции выполняются, команды отвечают на вопрос «что делать», а операнды – «над чем». Говоря коротко, МП обрабатывает *данные* (собственно данные – операнды) в соответствии с *программой*. Понятие программы и данных является центральным в ЭВМ. Вся цифровая информация, находящаяся в памяти ЭВМ, интерпретируется процессором либо как программы, либо как данные. Программы и данные – это тот материал, с которым работает процессор. Для процессора очень важно отличать программы от данных, так как первые (т.е. программы) диктуют процессору порядок его действий, а последние (т.е. данные) служат теми числами, над которыми эти действия выполняются. Понятия программы и данных проходят через всю идеологию построения ЭВМ и, в конечном счете, определяют принципы ее архитектуры и организации.

Программа, непосредственно воспринимаемая машиной, представляет собой последовательность машинных команд, записанных в двоичных кодах и расположенных в памяти в том порядке, в каком они должны выполняться процессором – подряд, одна за другой, по смежным ячейкам памяти, а в случае переходов по программе – по адресам перехода. Вся программа занимает в памяти один или несколько непрерывных массивов информации. Такую общую

форму имеет любая машинная программа. Содержание программы определяется совокупным смыслом составляющих ее команд, которые будучи расположенные в нужном порядке, реализуют требуемый алгоритм обработки данных по решению поставленной задачи. Программа выступает в качестве задания процессору на работу.

Перед запуском на счет программа должна быть загружена в основную память компьютера. Часть ОП, где размещаются программы, называется **программной памятью**. Программная память располагается в ПЗУ и ОЗУ. Управляющие ЭВМ, встраиваемые в технологическое оборудование, (например, однокристальные микро ЭВМ – микроконтроллеры) содержат в ПЗУ, как правило, все необходимые управляющие и системные программы. Универсальные ЭВМ хранят в ПЗУ лишь стартовую, инициирующую часть программного оснащения, служащую для начальной загрузки операционной системы с жесткого магнитного диска в ОЗУ. В ПЗУ общих ЭВМ находятся также некоторые вспомогательные программы: тестовые программы для проверки работоспособности устройств и объема памяти, программы-драйверы периферийных устройств, подпрограммы обработки прерываний и др. Совокупность упомянутых программ ПЗУ составляет, например, так называемую базовую систему ввода-вывода ROM BIOS. В ПЗУ могут храниться, кроме того, некоторые константовые данные, например марка процессора, год выпуска и пр.

Данные (цифровая информация, интерпретируемая процессором как **операнды**, т.е. как числа, над которыми производятся операции) записаны в другой части ОП, называемой **памятью данных**, обычно в ОЗУ, а также могут находиться в программной памяти (когда они входят в состав команд). Данные (операнды), размещенные в памяти данных, называются **переменными**, так как они могут изменяться в ходе работы программы. Они тоже объединяются обычно в непрерывные массивы информации (как и команды в программной памяти). Данные (операнды), находящиеся в программной памяти, т.е. входящие в состав команд (встроенные в коды команд в качестве непосредственных операндов или адресов) называются **константами**. Их нельзя изменить во время работы программы (программа не изменяет саму себя).

Заметим, что *данными* называют зачастую любую информацию, хранящуюся в компьютере, т.е. не только операнды, но и команды, поскольку и те и другие служат отправным материалом для работы компьютера. К слову сказать, и шина компьютера, предназначенная для считывания процессором команд из программной памяти и обмена операндами между устройствами ЭВМ, именуется шиной данных.

У процессоров Intel программная память организована в форме программных, кодовых сегментов, адресуемых при помощи регистра CS, а **память данных** – в форме сегментов данных и стека, адресуемых при помощи регистров DS, ES, FS, GS, SS (п.2.3, 2.4, 4.5). Кроме того следующие **области памяти** этих процессоров **зарезервированы** для специальных операций:

FFFFFFF0-FFFFFFFF

– для пуска после системного сброса;

00000000-000003FF

– для хранения таблицы векторов прерываний (гл.5).

Источники программ и данных. Программы и данные, накопленные в основной памяти, поступают в нее от пользователя через устройства ввода (с клавиатуры, манипуляторов типов «мышь» и «джойстик», сканера и др. устройств), от технических объектов (датчиков физических величин через аналого-цифровые преобразователи или др. сопрягающие устройства) или из внешней памяти (с магнитного диска, магнитной ленты, флэш-карты и т.д.). Запись программ в ПЗУ производится с помощью специальных приборов – программаторов ПЗУ, или в процессе изготовления.

Данные формируются в основной памяти также в процессе счета программы. Результаты счета выводятся через устройства вывода пользователю (на экран монитора, принтер и др. устройства) или на технические объекты (на исполнительные механизмы через цифро-аналоговые преобразователи и усилители) или во внешнюю память (на диски, ленты, флэш-карты).

Команда (**машинная инструкция**) служит приказом процессору на выполнение какой-либо простой операции – пересылки порции данных, преобразования данного или другой операции. Структура команды имеет общий вид, показанный на рис. 4.2. Команда содержит код выполняемой ею операции (КОП) и информацию о данных, участвующих в операции. КОП сообщает процессору, что нужно делать: переслать информацию, сложить два числа, осуществить переход. Поэтому КОП называют часто командой, как и полный формат команды. Информация о данных определяет операнды или адреса операндов в памяти или портах в/в (для команд обработки данных и изменения состояния процессора); или адреса переходов (для команд передачи управления). Команда содержит также информацию об адресе, по которому помещается результат операции.

Рис. 4.2. Общая структура команды

Часть команды, содержащая КОП, называется *операционной*, а часть, содержащая операнды или адреса – *адресной*. Операционная часть обязательна для любой команды, а адресная имеется не у всех команд. Например, команда останова hlt (halt) не имеет адресной части по определению, так как она не оперирует ни с какими операндами, а останавливает работу процессора. Команда возврата ret (return), хотя и оперирует с операндами, но адресует их неявно (при помощи регистра указателя стека), поэтому тоже не имеет адресной части.

Команды обработки данных содержат в своей адресной части *операнды* или *их адреса*. Примерами таких команд являются команды пересылки mov, ввода и вывода in и out, сложения add, вычитания sub и многие другие. Коман-

ды передачи управления содержат в своей адресной части не адреса операндов, а адреса следующих для исполнения команд – *адреса переходов*. К ним принадлежат команды перехода, вызова и возврата из подпрограмм и прерываний: jmp addr – перейти по адресу addr, call addr – вызвать подпрограмму с адреса addr, ret – вернуться из подпрограммы, int *n* – вызвать прерывание с номером *n*, iret – вернуться из прерывания и т.д.

Напомним, что числа, входящие в адресную часть команд, – непосредственные операнды, адреса операндов или адреса переходов – это и есть то, что мы называем *константами*. А числа, располагающиеся вне команд, вне программы (вне программной памяти), называются *переменными*. Они находятся в памяти данных.

Режимы (способы) адресации. Способ задания операнда в команде (или адреса следующей команды в команде передачи управления) называется *режимом адресации* (операнда или перехода соответственно). В зависимости от наличия или отсутствия адресной части в команде режимы адресации бывают *явные* и *неявные*, а в зависимости от ранга (от степени вложенности) адреса – *непосредственные*, *прямые* и *косвенные*.

Явная и неявная адресация. Если команда в своей адресной части содержит буквальные сведения об операнде или его адресе или адресе результата операции (или об адресе перехода) или о месте, где хранится адрес (об адресе адреса), то такая адресация называется *явной*. Если же адресное поле для операнда или адреса в команде отсутствует, а операнд/адрес подразумевается, т.е. заранее известен, предопределен, принимается по умолчанию, то такая адресация называется *неявной*. При неявной адресации операнд/адрес фактически задается кодом операции команды.

Например, команда rol AL, 1 – циклически сдвинуть содержимое регистра-аккумулятора AL влево на один бит использует явную адресацию аккумулятора и явное задание непосредственного операнда (единицы). Команда inc BX – инкрементировать (увеличить) содержимое BX на единицу адресует регистр BX явно, а единицу – неявно, по умолчанию. В ранее приведенной команде ret адресация перехода (через стек) неявная, так как в адресной части команды нет указания об адресе перехода (как и вообще нет самой адресной части).

Непосредственная, прямая и косвенная адресация. Адресация называется *непосредственной* («ссылкой, адресацией 0-го ранга»), если адресная часть команды содержит сам операнд (при явной адресации); или этот операнд подразумевается, принимается по умолчанию (при неявной адресации). Соответственно такой операнд называется *непосредственным операндом*. При непосредственной адресации не требуется обращения к памяти для выборки операнда и ячейки памяти для его хранения. Это способствует уменьшению времени выполнения программы и занимаемого ею объема памяти. Непосредственная адресация удобна для хранения различного рода констант.

В иных случаях адресная часть содержит не операнд, а его адрес (или адрес перехода); или этот адрес подразумевается кодом операции при неявной адресации. Такая адресация называется *прямой* («ссылкой 1-го ранга»).

И, наконец, если код команды определяет (явно или неявно) адрес ячейки памяти или имя регистра процессора, в котором находится не операнд, а его адрес (называемый *указателем*) или адрес перехода, то такая адресация называется *косвенной* («ссылкой 2-го ранга»). При косвенной адресации задается, как мы видим, не адрес операнда (или перехода), а *где* его взять, в *каком* регистре или ячейке памяти он находится, т.е. задается *адрес адреса* («адрес 2-го ранга»).

Примеры.

mov BX, 10 – «занести число 10 в регистр BX». В данной команде задается сам операнд-источник – число 10, являющийся, следовательно, непосредственным операндом, а способ его задания – *непосредственной адресацией*.

mov BX, [10] – «передать в регистр BX содержимое 10-й ячейки памяти из сегмента данных DS». Здесь операнд-источник адресуется прямо – указанием его адреса в памяти (числа 10). Это – *прямая адресация* операнда.

mov BX, [SI] – «прочитать в BX содержимое ячейки, адрес которой (внутри сегмента данных DS) содержится в регистре SI». Эта команда адресует источник (ячейку памяти) косвенно: она указывает, где взять адрес ячейки – в регистре SI. Тут мы имеем дело с *косвенной адресацией* операнда-источника. Регистр SI (или его содержимое) является указателем на операнд. Если предварительно занести в SI число 10: mov SI, 10, то будет прочитана 10-я ячейка из сегмента данных, как и в предыдущем примере.

И во всех приведенных примерах операнд-приемник – регистр BX адресуется прямо – заданием его имени (*прямая адресация* операнда-приемника).

Косвенная адресация широко используется в ЭВМ, имеющих короткое машинное слово, (например, в микроконтроллерах) для преодоления ограничений короткого формата команд (обычно используется косвенная регистровая адресация, при которой адрес задается регистром процессора). Кроме того, при помощи косвенной адресации легко организовывать обработку больших массивов данных. Занеся один раз начальный адрес массива (адрес его первого элемента) в регистр-указатель, можно переходить к следующим элементам, инкрементируя этот адрес. Инкрементация производится в цикле программы с числом повторений, равным числу элементов массива. Это дает возможность обрабатывать все элементы с помощью короткого программного цикла. Пример подобного цикла можно увидеть далее в программе на рис.4.3 начиная с метки Summa и заканчивая командой ветвления jnz Summa. Если же применять прямую адресацию элементов, то пришлось бы записывать в программе столько команд обращения к памяти, сколько элементов содержится в массиве, указывая каждый раз адрес нового элемента. Если массив содержит, к примеру, 1000 элементов, то в программе пришлось бы записать 1000 команд с тысячью разными адресами. Без применения косвенной, «динамической» адресации обработать такой длинный массив практически невозможно. Прямая, «статическая» адресация не годится для этой цели.

Компоновка команд. Команды могут состоять из одного или нескольких машинных слов. Все слова одной команды размещаются подряд, по смежным ячейкам памяти. Первое слово всегда содержит КОП и, возможно, укороченные

адреса операндов (или перехода). Другие слова, если они имеются, содержат сами операнды или их адреса в памяти или портах, или адрес перехода, или являются расширением кода операции.

Адресом команды считается адрес ее первого (с наименьшим адресом) слова, содержащего КОП.

Пословная компоновка команд применяется в процессорах с архитектурой DEC VAX-11.

Побайтная компоновка команд. У многих процессоров (включая Intel) команды компонуются не из слов, а из байтов (тоже размещаемых один за другим по смежным – 8-битным ячейкам памяти). При таком – побайтном – построении команд КОП находится всегда в первом байте команды, по самому младшему адресу. Этот адрес считается **адресом всей команды**.

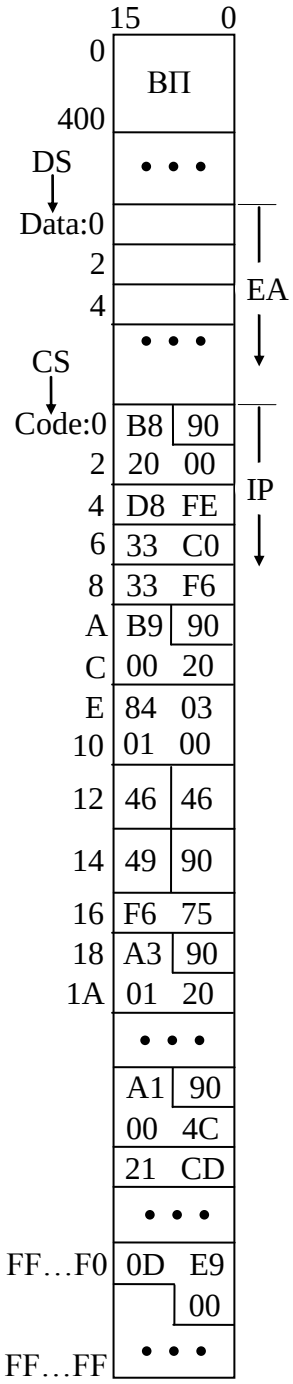
Команды МП IA-32 имеют длину от 1 до 11 байтов (рис.4.3). 1-й байт всегда содержит КОП, который в некоторых командах расширяется на 2-й байт. В большинстве команд за КОП следуют 1 или 2 специальных байта, определяющих режим адресации операнда (или перехода), при этом первый из этих байтов может содержать дополнительное расширение КОП. Остальные байты, если они имеются, содержат числовые константы, определяющие адрес операнда (внутрисегментный) или перехода (0, 1, 2 или 4 байта) и/или сам операнд или числовые сдвиговые добавки к адресам, заданным регистрами (при смешанной адресации) (0, 1, 2 или 4 байта). Режимы адресации бывают явными и неявными, непосредственными, прямыми и косвенными или их сочетаниями (смешанными).

Рис. 4.3. Структура команды МП Intel

Перед кодом команды в ряде случаев вводятся один или несколько префиксных байтов, модифицирующих выполняемую операцию: префикс замены сегмента CS:, SS:, DS:; FS: или GS:; префикс повторения rep; префикс блокировки lock; префикс разрядности операнда; префикс разрядности адреса. Всего перед командой может стоять четыре префикса. Префиксы считаются частью команды. Под этим понимается тот факт, что прерывание, поступившее в момент обработки префикса, будет обслужено лишь по окончании выполнения всей команды. Поэтому можно считать, что вместе с префиксами длина команд процессоров Intel составляет от 1 до 15 байтов.

Проведенные оценки показывают, что в среднем длина команд составляет 4-5 байтов. Поэтому каждая из двух очередей команд (по 32 байта) в процессоре Pentium содержит обычно 6-8 команд, а в обработке на конвейере находится еще 4-5 команд.

На рис. 4.4 дан **пример программы МП IA-32** в кодах и на языке ассемблера (в мнемокодах) [18] и показано размещение ее в памяти. На рисунке




```
Метка  КОП  Операнды  Комментарий
; Программа суммирования элементов массива Array
Name Example        ; Начало исходного модуля
ASSUME CS:Code, DS:Data; Базовые адреса сегментов
Data  SEGMENT    ; Начало сегмента данных
        ORG  100h
Array DW 20h DUP(?)  ; Зарезервировать 32 слова
  ; памяти под массив Array начиная с адреса 100h
Sum   DW 0  ; Определить и инициировать
                ; переменную SUM
Data  ENDS    ; Конец сегмента данных
Code  SEGMENT    ; Начало сегмента кода
Start:  nop              ; Инициализация
        mov AX, Data  ; регистра DS на
        mov DS, AX    ;  Data=2000
        xor AX, AX    ;  Сбросить аккумулятор
        xor SI, SI        ;  и индекс массива
        nop
        mov CX, 20h   ;  Число элементов в CX
Summa:  add AX, Array[SI] ; Прибавить элемент

        inc SI            ;  Продвинуть
        inc SI            ;  индекс массива
        nop
        dec CX            ;  Это последний элемент?
        jnz Summa         ;  Нет, повторить суммирование
        nop
        mov Sum, AX  ;  Да, запомнить сумму в
                          ;  переменной Sum
        • • •
        nop
        mov AX, 4Ch  ;  Выход из
        int 21h           ;  программы
Code  ENDS          ; Конец кодового сегмента
      END Start     ; Конец исходного модуля
      jmp 0000      ;  Стартовая команда компьютера:
      ; перейти на ячейку 0000 стартового сегмента
      ; кода с базой FFFF0000
```

Рисунок 4.4 – Программа МП IA-32 и ее размещение в памяти

На рисунке обозначено: ВП – таблица векторов прерываний; EA (Effective Address) – адрес операнда внутри сегмента данных (эффективный, исполнительный адрес операнда); сегмент данных адресуется регистром DS, сегмент кода – регистром CS. Программа получает в аккумуляторе AX сумму элементов массива Array и запоминает ее в переменной Sum. Программа состоит из 13 команд, расположенных с базового адреса Code. Область данных начинается от базового адреса Data. По адресу FFFFFFF0 (в трех ячейках с адресами FFFFFFF0...FFFFFFF2) находится стартовая команда

компьютера – команда прямого безусловного перехода jmp (код операции E9), передающая управление на стартовую программу компьютера, расположенную начиная с адреса FFFF0000.

Команды в программном коде для наглядности выровнены по четным адресам, для чего все команды, содержащие нечетное число байтов, дополнены спереди однобайтной «холостой» командой nop (no operation) – нет операции (код 90h). Совместно с nop они теперь начинаются с четных адресов памяти.

4.3. Размещение программы в основной памяти

Существенное условие размещения программы в основной памяти. Программист, работающий на языке машинных команд или ассемблера, должен следить за правильным размещением программы в основной памяти ЭВМ:

1. Все команды программы должны быть размещены подряд в последовательных ячейках памяти (для МП IA-32 – в последовательных 8-битных ячейках) в том порядке, в каком они записаны в программе (см. рис. 4.3). Такое размещение программы называется *естественным*. И только команды, на которые передается управление с помощью команд перехода, *принудительно* размещаются по адресам, указанным в этих командах перехода – адресам перехода (например, команда add в программе на рис.4.3 находится по мнемоадресу перехода (метке) Summa). Поэтому команды выполняются, как правило, в порядке возрастания адресов.
2. Команды МП Intel могут начинаться как с четных, так и нечетных адресов памяти.
3. Первая команда программы помещается в некоторую исходную ячейку памяти, начиная с которой и располагается вся программа. Обычно адрес этой ячейки задается автоматически операционной средой, в которой выполняется программа (операционной системой). В этом случае его численное значение скрыто от программиста. Программист идентифицирует свою программу не по адресу, а по имени (мнемоническому адресу). При необходимости программист может «вручную» установить начальный адрес программы на определенное числовое значение. Необходимость в этом возникает, например, при программировании встраиваемых микроконтроллеров. Если же программы создаются для запуска и исполнения в операционной среде, то использование числовых адресов памяти не только не обязательно, но и нежелательно из-за возможности допустить ошибку и занять в памяти место, отводимое под другие программы, разрушив тем самым их работу. В этом случае при написании программ на языке ассемблера программист оперирует в терминах не числовых, а мнемонических адресов – меток команд, имен переменных, имен подпрограмм. Применение директив ассемблера резко облегчает соблюдение перечисленных правил 1-3. [1, 2, 8, 9, 17]

Стартовая команда компьютера (запускаемая самой первой по сигналу системного сброса/пуска RESET) должна находиться во вполне определенной

ячейке памяти – ячейке, с которой запускается процессор по сигналу RESET. Для большинства процессоров адрес пусковой ячейки строго фиксирован. Для МП Intel он равен FFFFFFF0, для микроконтроллеров PIC16F – 0000, для МК семейства AVR – 000. Для некоторых микропроцессоров пусковой адрес задается пользователем, для чего перед пуском он заносится в специальный внешний регистр и считывается из него процессором в момент пуска. В пусковую ячейку памяти помещают, как правило, команду безусловного перехода на начало стартовой программы компьютера. Такая команда jmp расположена в ячейке FFFFFFF0 для программы, приведенной на рис. 4.3, и она передает управление на ячейку FFFF0000.

Пусковая ячейка должна входить в состав ПЗУ – энергонезависимого ЗУ. В ПЗУ находится и вся стартовая программа компьютера. Поэтому после включения питания компьютер всегда готов к запуску стартовой программы.

4.4. Продвижение процессора по программе

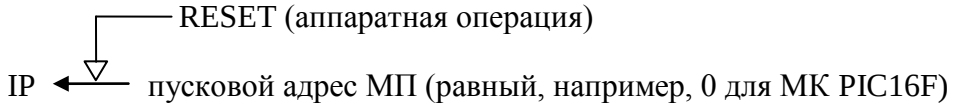
Порядок выполнения команд. Команды программы выполняются, как правило, в порядке возрастания адресов и начиная с некоторого начального, стартового адреса: S, S+1, S+2, ... (В программе на рис. 4.4 этот адрес обозначен именем Start.) Исключение составляют лишь команды, принимающие на себя управление по указке команд перехода и выполняемые с нарушением естественного порядка. Например, команда add AX, Array[SI], помеченная меткой Summa на рис. 4.4, принимает управление от команды перехода jnz Summa.

Назначение программного счетчика (указателя инструкций IP). Для поддержания указанного порядка выполнения команд служит специальный собственный регистр процессора, называемый *программным счетчиком* PC (Program Counter) или *указателем инструкций* IP (Instruction Pointer) или *счетчиком команд* СчК. *Этот регистр всегда содержит адрес следующей по порядку, очередной, еще не исполненной команды, которая будет выбрана из памяти и исполнена вслед за текущей командой; или адрес очередного слова выбираемой команды (если команда длинная и выбирается за несколько циклов шины)*. Благодаря регистру IP процессор всегда «знает», откуда получить следующую команду или ее очередное слово.

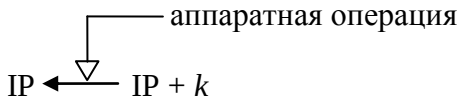
Правила модификации программного счетчика. Представленное свойство программного счетчика достигается с помощью простого логического механизма, встроенного в аппаратуру процессора и схематично поясняемого диаграммой на рис. 4.5.

1) *При пуске* процессора (осуществляемого по сигналу системного сброса/пуска RESET) в IP заносится определенный, обычно фиксированный, стартовый адрес, например, нулевой: IP ← 0000. Поэтому процессор запускается всегда с заранее известной ячейки памяти (в нашем примере – с нулевой).

1) При пуске МП происходит инициализация (начальная установка) IP:



2) При естественном продвижении по программе (при считывании очередного слова команды) содержимое программного счетчика увеличивается на число выбранных из памяти байтов *k*:



3) При принудительном продвижении по программе в программный счетчик загружается адрес перехода (передачи управления):

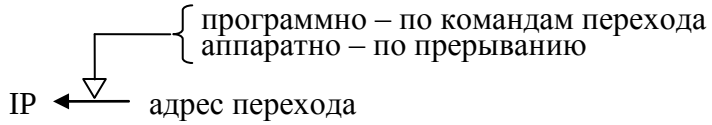


Рис. 4.5. Три типа операций, выполняемых с программным счетчиком

2) *Естественное продвижение по программе*. После пуска процессор приступает к выборке и выполнению команд программы. Команды выбираются и выполняются им последовательно, одна за другой, в том порядке, в каком они записаны в программе; или в порядке переходов по программе. Первый способ продвижения по программе, когда команды выбираются в порядке возрастания их адресов, называется естественным, второй, когда происходит «скачок» вперед или назад по программе из текущей точки программы в заданную – принудительным (п. 1.2). После считывания процессором каждого слова команды из памяти содержимое IP автоматически и немедленно (еще до выполнения внутренних операций процессора) увеличивается на:

1 – после считывания (выборки) однобайтного слова,

2 – после выборки 2-байтного слова, и т.д.

И, вообще, IP *увеличивается всегда на число выбранных из памяти байтов команды* (см. рис.4.4). Поэтому он снова указывает на следующий за только что выбранным байт программной памяти, и этот байт (вместе с содержащим его словом) будет считан из памяти в следующем цикле чтения команды (если, конечно, выбранная команда не окажется командой перехода).

3) *Переходы по программе*. По командам перехода (передачи управления) в IP загружается адрес перехода: «естественный» адрес в IP, продвинутый после выборки команды перехода, стирается и на его место записывается новое значение, задающее местоположение следующей команды для исполнения. Насильственная загрузка нового адреса в IP производится также во время прерывания программы (гл. 5) (и, как уже было сказано, по сигналу системного сброса RESET) (см. рис. 4.5). В результате выбирается не команда из ближайшей по

ходу программы ячейки, а команда, размещенная по адресу перехода, т.е. происходит «скачок», переход в новую точку программной памяти.

Благодаря соблюдению представленных правил программный счетчик содержит всегда адрес очередной команды. Он всегда указывает на следующую по порядку, очередную, еще не исполненную команду программы, подлежащую выборке и исполнению вслед за текущей командой; или на очередное слово выбираемой команды.

Посредством команд передачи управления программный счетчик доступен программисту. Это – программно-доступный, «видимый» регистр процессора. При помощи команд передачи управления в него можно загрузить любое новое значение – *адрес перехода*. Другие команды не могут принудительно изменить содержимое IP.

Отметим, что обозначение программного счетчика (IP, PC или какое-либо еще) не встроено в язык ассемблера процессоров Intel. Нельзя, например, написать: mov IP, 1234h. Регистр IP (или EIP) адресуется командами передачи управления (jmp, call, ret и др.) неявно (по умолчанию).

Заметим, что некоторые процессоры (микроконтроллеры, например PIC16F) допускают явную адресацию программного счетчика по имени, распознаваемому ассемблером: МК PIC16F – по имени PCL – младший байт программного счетчика PC, и по имени PCLath – старшие биты PC.

4.5. Особенности продвижения по программе процессора IA-32

Указатель инструкций EIP/IP и указатель кодового сегмента CS. Согласно п.2.4 особенность процессоров архитектуры IA-32 состоит в том, что процессор использует два числа для назначения адреса памяти. Одно из них (селектор) определяет (прямо или косвенно) базовый адрес, другое – смещение относительно базы. База и смещение суммируются в процессоре и дают абсолютный адрес ячейки памяти. Совокупность селектора и смещения принято называть логическим или сегментированным адресом ячейки, а сумму базы и смещения – линейным (или абсолютным) адресом. Подобный способ адресации называют сегментацией адреса, сегментацией памяти.

Механизм сегментации применяется, в том числе, и по отношению к программной памяти. Он позволяет процессору IA-32 формировать линейный адрес команды, используя двухкомпонентный логический адрес. Одна из компонент логического адреса – 16-битный селектор – находится в регистре-указателе кодового сегмента CS. Селектор задает (прямо – в реальном режиме и косвенно – в защищенном) начальный, базовый, адрес CS base кодового сегмента – сегмента, хранящего исполняемую в данный момент программу. Другая компонента логического адреса – 16-битная в реальном режиме и 32-битная в защищенном – находится в регистре-указателе инструкций IP или EIP соответственно. Она определяет местоположение команды относительно начала кодового сегмента, задавая ее смещение (offset) относительно базы. Путем суммирования базы и смещения процессор находит линейный адрес команды.

В общем случае регистр EIP/IP указывает не на команду, а на слово программной памяти – группу последовательных байтов, выбираемых процессором из программной памяти за один цикл шины. Эти байты могут принадлежать сразу нескольким смежным командам; или являться частью одной длинной команды; или быть одной целой командой. Подобную группу байтов назовем *командным словом, словом программной памяти*. Размер командного слова может составлять от 1 до 8 байтов (он не превышает ширины шины данных). Итак, программный счетчик EIP/IP определяет местоположение команды или командного слова относительно начала кодового сегмента.

Коротко: *регистр* CS *адресует кодовый сегмент, а регистр* EIP/IP – *очередное командное слово внутри этого сегмента*.

Адресация программы МП IA-32 в реальном режиме. В реальном режиме регистр CS работает в паре с IP. *Содержимое этих двух 16-битных регистров определяет 20-битный физический адрес очередной команды или очередного командного слова для следующей выборки из памяти*. Для вычисления искомого физического адреса процессор использует сдвинутое на 4 бита влево содержимое CS, которое фиксируется в дескрипторном регистре, связанном с CS, и прибавляет к нему содержимое IP (рис. 4.6):

Physical instruction address = CS_0 + IP.

Здесь CS_0 = CS_0000 – сдвинутое содержимое CS, т.е. CS с пристроенными к нему справа четырьмя двоичными нулями (одним шестнадцатеричным нулем: «умножение на 16»): CS_0 = CS×16.

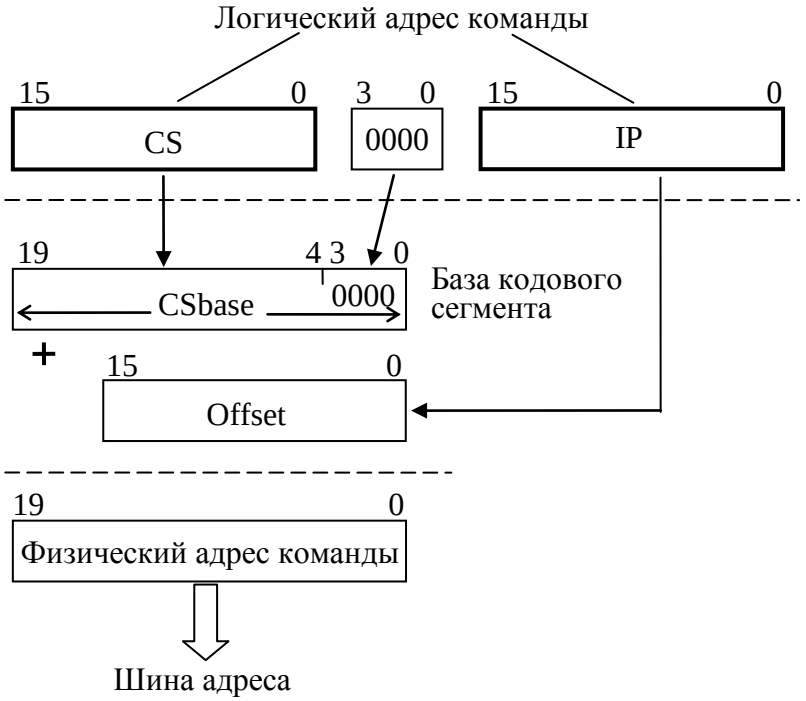


Рис. 4.6. Вычисление физического адреса команды в реальном режиме процессора Intel

Вычисленный таким образом 20-битный код и выставляется процессором в системную шину адреса A19..0 в качестве адреса очередного командного слова. В остальные линии шины адреса A31..20 сразу после пуска выдаются единицы, а после первого межсегментного перехода – нули. Возникающий при суммировании перенос из старшего бита (A19) игнорируется, нигде не запоминается.

20-битное число CS_0 (конкатенация CS и 4 нулевых битов) – это, по терминологии п.2.4, *базовый адрес* кодового сегмента: CS base = CS_0, а содержимое IP – *смещение* команды относительно базы: offset = IP. Сумма базы и смещения определяет абсолютный, физический адрес ячейки для выборки команды. Выражение CS_0 + IP кратко записывается как CS:IP.

Для перехода на другой кодовый сегмент надо загрузить его селектор (с помощью команды передачи управления; или по прерыванию) в регистр CS (см. п.2.4 и абзац «Правила модификации регистров CS и EIP/IP» ниже в этом параграфе).

Адресация программы МП IA-32 в защищенном режиме. Для адресации программной памяти в защищенном режиме служат регистр CS (включая его теневое расширение – дескрипторный регистр) и регистр EIP (расширенный указатель инструкций). *Два числа, которые они содержат, – 16-битный селектор в* CS *и 32-битное смещение в* EIP *являются логическим (сегментированным) адресом очередной команды или очередного командного слова, подлежащего выборке процессором из памяти в следующем цикле шины. На основе логического адреса определяется 32-битный линейный адрес этой команды (слова).*

При загрузке селектора в CS (по команде передачи управления или прерыванию) процессор автоматически считывает указываемый им дескриптор из дескрипторной таблицы и заносит его в связанный с CS дескрипторный регистр. Считанный дескриптор полностью описывает кодовый сегмент. Он содержит 32-битный базовый адрес сегмента CS base, 20-битный предел сегмента и другие параметры. Поскольку CS указывает не на сам кодовый сегмент, а на дескриптор в памяти, который и содержит всю информацию о сегменте, то CS адресует кодовый сегмент в защищенном режиме не прямо (как в реальном режиме), а косвенно.

Смещение команды (командного слова) (относительно базы CS base) задается указателем команд EIP. CS base и EIP суммируются в процессоре и дают линейный адрес команды (слова):

Line instruction address = CS base + EIP

(рис. 4.7).

Данная операция кратко записывается с помощью выражения CS:EIP. Здесь также при возникновении переноса из старшего бита (бита A31) в несуществующий бит A32 происходит сброс адреса в окрестность нулевого значения, поскольку перенос нигде не запоминается, так что за ячейкой с максимальным адресом FFFFFFFF следует ячейка с нулевым адресом.

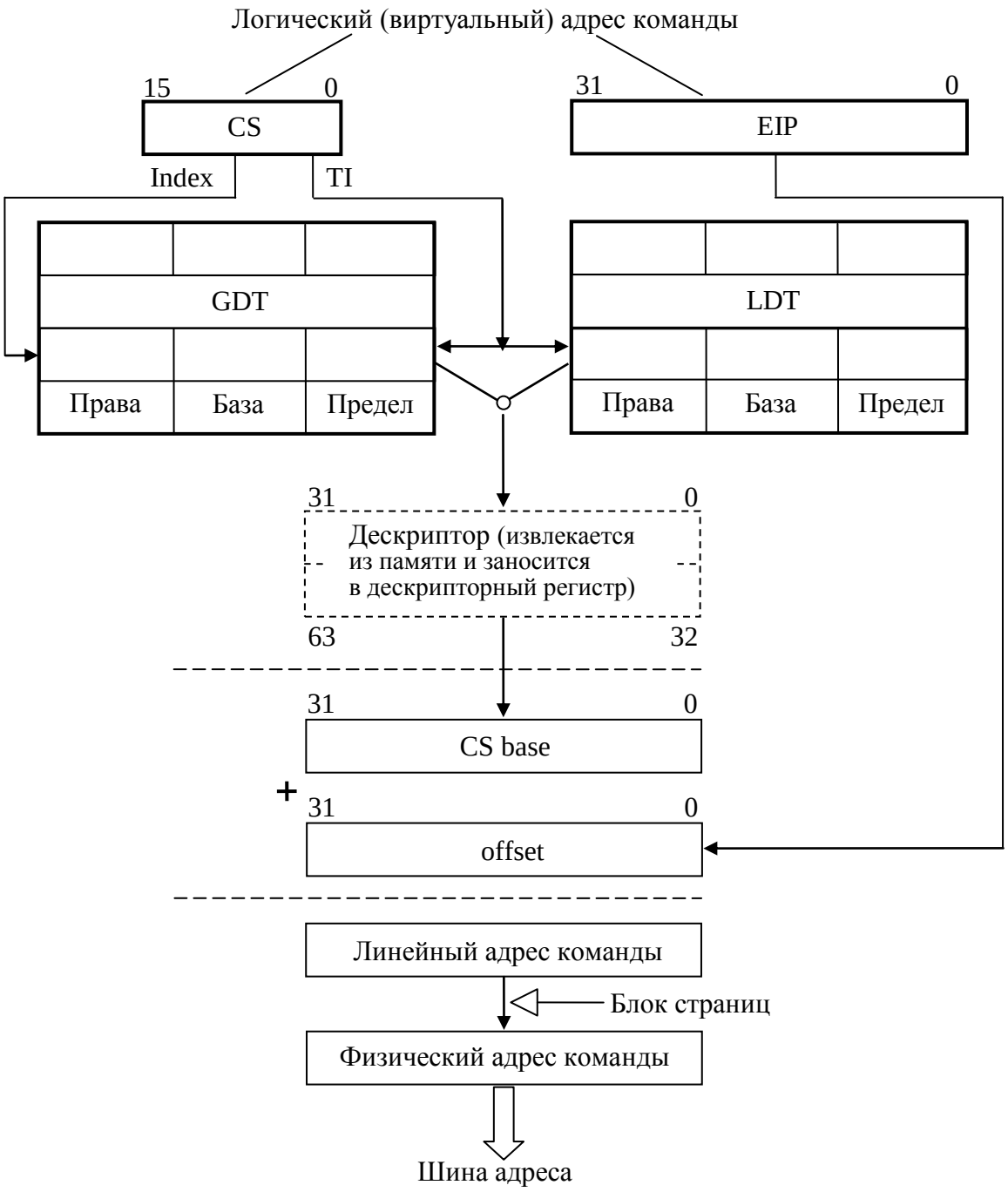


Рис. 4.7. Вычисление линейного адреса команды
в защищенном режиме процессора IA-32

Кодовый сегмент. Адресация программной памяти по принципу CS:EIP/IP = база:смещение (сегмент:смещение, code:offset) позволяет логически выделить в пространстве памяти ЭВМ кодовый сегмент, предназначенный для хранения текущей программы, и указать в этом сегменте на очередную команду (слово) программы. Кодовый сегмент представляет собой способ логической организации программной памяти. В реальном режиме общий объем пространства памяти процессоров IA-32 составляет 1 Мбайт, и в нем можно выделять кодовые сегменты объемом 64 Кбайта каждый. В защищенном режиме общий

объем памяти 4 Гбайт, а длина кодовых сегментов может составлять от 1 до 4 Гбайт.

После пуска кодовый сегмент занимает адреса FFFF0000-FFFFFFFF, имеет длину 64 Кбайта, и процессор запускается с ячейки FFFFFFF0 этого сегмента (см. п. 2.3 и табл.2.3, а также п. 4.6).

Правила модификации регистров CS и EIP/IP. Регистры CS и EIP/IP обеспечивают адресацию очередных команд в логическом пространстве памяти и продвижение процессора по программе путем модификации их содержимого по определенным правилам, представленным на рис. 4.8.

1) При пуске МП Intel происходит инициализация (начальная установка) процессора:

RESET (аппаратная операция)

CS:EIP ← F000:0000FFF0

CS base ← FFFF0000

После сброса и инициализации в первом цикле магистрали выборка кода команды всегда производится из ячейки памяти с адресом CS base + EIP = FFFFFFF0

2) При естественном продвижении по программе (при считывании очередного командного слова) содержимое программного счетчика увеличивается на число выбранных из памяти байтов *k*:

аппаратная операция

EIP/IP ← EIP/IP + *k*

k = 8 при выборке выровненного слова – слова с адресом, кратным восьми; *k* < 8 при выборке невыровненного слова после перехода процессора по не кратному 8-ми, адресу.

3) При принудительном продвижении по программе в программный счетчик загружается адрес перехода (передачи управления)

При далеких переходах (типа Far):

программно – по командам далекого перехода
аппаратно – по прерыванию

CS:EIP/IP ← новая база : новое смещение

При близких (NEAR) и коротких (SHORT) переходах:

программно – по командам близкого и короткого перехода

EIP/IP ← новое смещение = EIP/IP + знаковая добавка

Рис. 4.8. Модификация содержимого регистров CS и EIP/IP при продвижении процессора IA-32 по программе

1) *Пуск* процессора IA-32 происходит всегда с ячейки памяти, имеющей адрес CS base + IP = FFFFFFF0 (п. 2.2, 2.3, 4.6, табл. 2.3).

2) *Естественное продвижение по программе*. После пуска процессор приступает к выборке и выполнению команд программы. Выборка (считывание) команд производится из программной памяти (из текущего кодового сегмента) по адресу CS:IP в реальном режиме и, после переключения в защищенный режим, по адресу CS:EIP. Считывание команд (слов) происходит 8-байтными порциями по линиям D63...D0 системной шины данных из 8-байтных ячеек памяти, имеющих кратные восьми адреса. При каждом считывании содержимое EIP/IP автоматически увеличивается на восемь:

EIP/IP ← EIP/IP + 8

и регистр EIP/IP вновь указывает на следующую, соседнюю, 8-байтную ячейку программной памяти (кодового сегмента). Длина командного слова, выбираемого из памяти, составляет в этом случае 8 байт.

Из этого правила имеется исключение. Оно наступает при переходе по невыровненному адресу – адресу, не кратному восьми. Если процессор под действием команды передачи управления переходит по такому адресу, то в цикле выборки он считывает не 8, а k байт ($k < 8$), вплоть до границы следующего выровненного (физического) слова, и продвигает программный счетчик на k:

EIP/IP ← EIP/IP + k

Соответственно, длина командного слова в подобном случае короче 8-ми байтов. Далее считывание команд 8-байтными словами возобновляется. Считанные команды сортируются внутри процессора побайтно (в регистрах очереди команд) и выполняются процессором в порядке очередности.

Таким образом, после инкрементации на число считанных байт регистр EIP/IP снова указывает на следующую, соседнюю ячейку программной памяти (внутри кодового сегмента), задавая ее смещение (offset) относительно базы сегмента (base), определяемой регистром CS. Этим он обеспечивает *естественное* продвижение по программе.

3) *Переходы по программе*. Регистры CS и EIP/IP доступны программисту посредством команд передачи управления: безусловного перехода, перехода по условию – ветвления, вызова и возврата из подпрограмм, вызова и возврата из прерываний (Приложение, п. Б.4, и литература [1, 2, 4, 9]). Под действием этих команд в указанные регистры загружаются новые значения: селектор (selector) – в CS и смещение (offset) – в EIP/IP. Имеются команды перехода, которые изменяют только содержимое программного счетчика EIP/IP (смещение), оставляя селектор в CS без изменения. Такие команды называются *командами близкого* или *внутрисегментного перехода* (типов NEAR – *близкого* и SHORT – *короткого*). Команды же, воздействующие на оба регистра CS и EIP/IP, называются *командами далекого* или *межсегментного перехода* (типа FAR). Команды перехода заставляют процессор нарушить естественное продвижение по программе и совершить переход на новую ячейку программной памяти, заданную адресом перехода. Они вызывают *принудительное* продвижение процессора по программе.

Команды далекого перехода (FAR) открывают новые сегменты и реализуют переход на любую ячейку всего адресного пространства памяти объемом в 4 Гбайт. Команды близкого перехода ограничены пределами текущего кодового сегмента. В случае собственно близкого перехода (NEAR) осуществляется доступ к любой ячейке кодового сегмента, а в случае короткого (SHORT) – только к ближайшим 127 байтам вперед, по ходу программы, и к 128 байтам назад, против хода. Команды типов NEAR и SHORT используют относительную адресацию перехода. Они указывают, на сколько байтов вперед или назад по программной памяти надо перейти от данной точки программы. Для этой цели они содержат в своей адресной части знаковую числовую добавку, прибавляемую к текущему значению программного счетчика, получившемуся после выборки команды перехода из очереди команд процессора, т.е. указывающему на следующую в памяти, смежную (с командой перехода) команду (п. Б.4).

При переходах по программной памяти, вызванных прерываниями (аппаратными или программными), в регистры CS и EIP/IP принудительно загружается адрес перехода на подпрограмму прерывания: в CS – новый селектор, в EIP/IP – новое смещение (эти два слова в совокупности называются вектором прерывания; гл.5). При аппаратных прерываниях данная операция совершается автоматически, в ответ на поступление в процессор сигнала запроса прерывания INTR или сигнала немаскируемого прерывания NMI, а не по указке машинных команд, как при программных прерываниях (по командам int) или вызовах подпрограмм (по командам call).

Мы видим, следовательно, что независимо от способа продвижения по программе регистры CS и EIP/IP всегда содержат правильный сегментированный адрес очередной команды или очередного командного слова программной памяти. На основе сегментированного формируется линейный адрес. В реальном режиме линейный адрес командного слова выставляется процессором в шину адреса A19...0 как физический, а в защищенном дополнительно преобразуется в физический блоком страниц (если блок включен) и тогда уже выставляется в шину адреса A31...0. Если блок страниц отключен, то в ША выставляется линейный адрес (и тогда он является физическим).

Вывод. С помощью команд передачи управления (jmp, call, ret и т.д.) в регистры CS и EIP/IP могут быть занесены любые новые значения – сегментированный адрес перехода. Иными словами, *данные регистры доступны программисту*. Кроме того, *регистр* EIP/IP *автоматически продвигает свое значение на* 1 *на каждый выбранный из программной памяти байт*, или, в целом, на число выбранных байтов, указывая на следующее, соседнее слово программной памяти. Содержимое CS и EIP/IP модифицируется также во время прерываний (гл.5) и устанавливается на определенные значения при пуске процессора (п.2.3, табл.2.3).

4.6. Пуск ЭВМ и ее функционирование после пуска. Командные циклы

Сброс, инициализация и пуск ЭВМ. После того как ЭВМ подготовлена к работе и в основную память занесена, тем или иным способом, программа в кодах и исходные данные, можно запускать ЭВМ на счет.

Обобщенный порядок функционирования ЭВМ представлен на рис. 4.9 [11]. Работа ЭВМ начинается с загрузки программного счетчика, инициализации (установки в начальное состояние) и пуска процессора с панели управления (блоки 1 и 2).

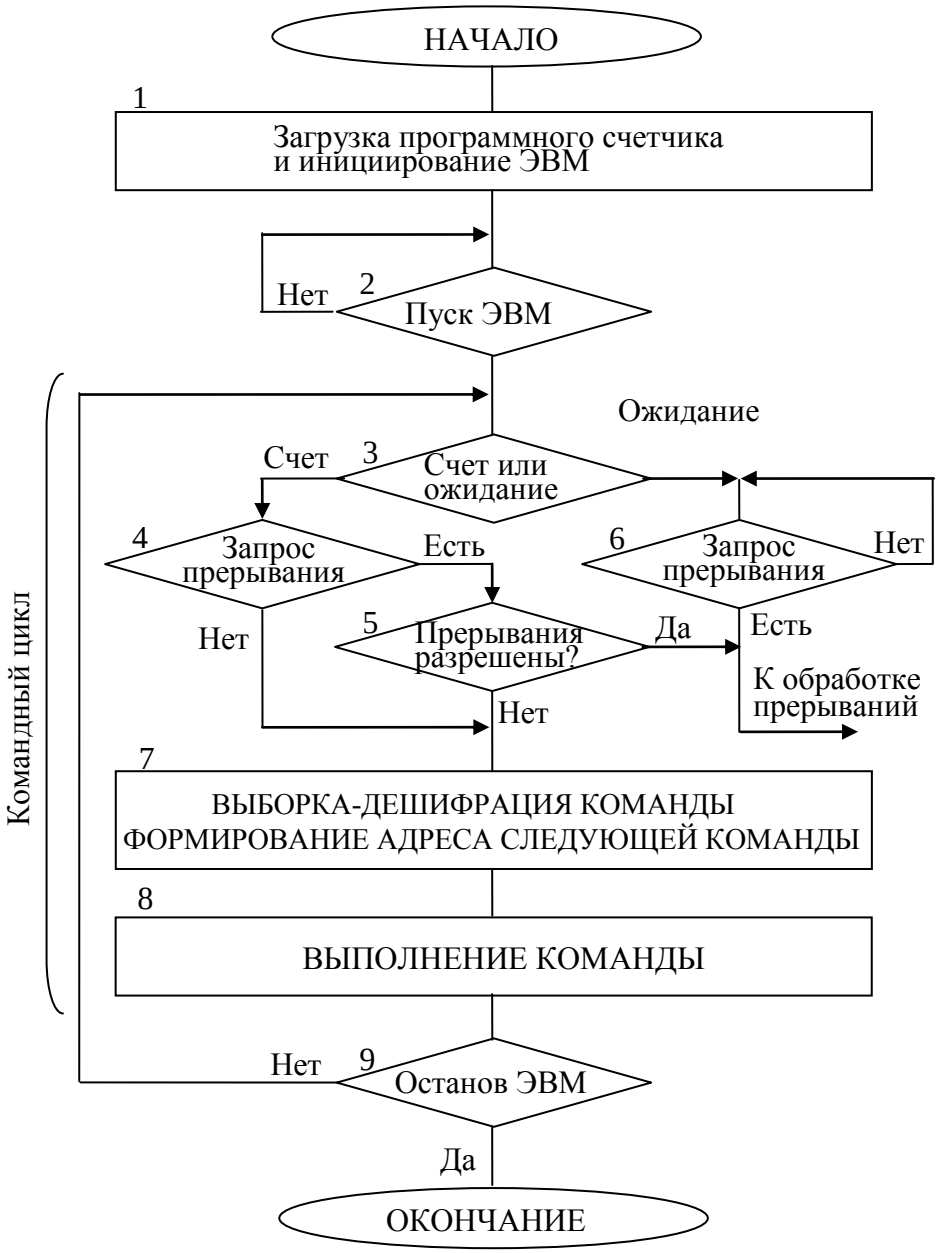


Рис. 4.9. Обобщенный порядок работы ЭВМ

Пуск процессора Intel производится при включенном питании и тактовых импульсах активизацией сигнала на входе RESET (так называемый «аппаратный сброс»). Он приводит к инициализации – установке внутренних блоков и внешних сигналов процессора в определенные начальные состояния [4, 9, 10, 19]. После снятия сигнала RESET процессор, по спадающему фронту RESET, запускается с конца физической памяти ЭВМ – ячейки FFFFFFF0, выполняя цикл чтения из нее машинной команды.

После пуска процессор Intel начинает работать в режиме реальных адресов при запрещенных прерываниях по входу INTR (IF=0). В этом режиме процессору доступен первый Mb памяти, в котором размещаются сегменты данных DS, SS, ES, FS, GS. Эти сегменты установлены на базовые адреса, равные нулю (п.2.3, табл.2.3), и далее в реальном режиме могут располагаться только в первом Мбайте памяти.

Кодовый сегмент CS после пуска занимает последние 64 Кбайт общего пространства памяти 4 Гб, входя в состав ROM BIOS (п. 2.3, табл. 2.3). Переход на другие адреса этого сегмента может быть осуществлен по командам близкой, внутрисегментной передачи управления jmp или call.

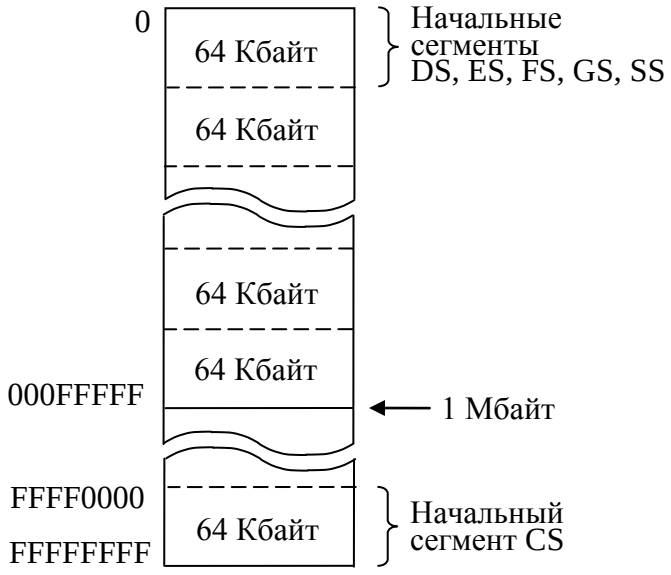


Рис. 4.10. Карта памяти после сброса

После первого же межсегментного перехода кодовый сегмент CS тоже переводится в первый Мбайт памяти. Это означает, что как только в программе появляется первая команда (например, jmp code:offset), обновляющая не только IP, но и CS (CS ← code, IP ← offset), база кодового сегмента устанавливается на значение CS base = code×16, и процессор переходит по 20-битному адресу code×16+ offset, относящемуся к первому мегабайту памяти. Он остается в этом мегабайте, не имея возможности выйти из него, до тех пор, пока он работает в реальном режиме. Для выхода в полное, 4-х гигабайтное, пространство памяти процессор должен быть переведен в защищенный режим работы (п. 2.2, 2.3).

Получающаяся после пуска (сброса) карта памяти представлена на рис. 4.10.

Выполнение команд после пуска начинается с фиксированной ячейки, адрес которой равен CS base + EIP = FFFFFFF0, где CS base = FFFF0000 – стартовый базовый адрес кодового сегмента, EIP = 0000FFF0 – стартовое значение указателя команд (табл. 2.3). Обычно в стартовой ячейке FFFFFFF0 «зашита»

команда безусловного перехода jmp на программу стартовой загрузки ОС, расположенную в ROM BIOS. Для Windows это программа Boot-Routine.

Файлы операционной системы изначально хранятся во внешней, долговременной памяти компьютера (на жестком, гибком или лазерном диске или флэшке). Диск, на котором находится операционная система и с которого производится ее загрузка, называется *системным* или *загрузочным*. На компьютере должен быть как минимум один системный диск. Загрузка ОС в оперативную память с системного диска происходит последовательно, в несколько этапов.

Сначала запущенный программный код (Boot-Routine) из ROM BIOS вызывает программу самопроверки POST (Power-On Self Test), проверяющую процессор, микросхемы ROM и RAM, вспомогательные элементы материнской платы, жесткий диск и другие компоненты, и производящую предварительную, так сказать, «грубую», настройку компонентов.

По окончании процедуры POST программа ROM BIOS вводит с диска C: (если дискеты в дисководе A: нет) так называемую главную загрузочную запись MBR (Master Boot Record) и передает ей управление. (Обычно это 512 байт первого сектора загрузочного диска.) Последняя загружает в ОЗУ ядро ОС, после чего в памяти размещается основная часть ОС.

Теперь, под управлением ОС, осуществляется инициализация (начальная установка) системы: регистров процессора, структур данных в памяти: дескрипторных таблиц, векторов прерываний и др., а также окончательная, «тонкая», настойка компьютерных составляющих под нужды данной ОС. После инициализации управление передается командному процессору или графическому интерфейсу или другим программам.

Переключение процессора в защищенный режим из реального осуществляется загрузкой в системный регистр CR0 слова с установленным битом PE (Protect Enable).

Как правило, большая часть действий по инициализации защищенного режима выполняется либо перед переключением в этот режим, либо непосредственно после переключения. Если инициализация осуществляется после переключения, выполняемые затем процедуры не должны использовать не инициализированные еще средства защищенного режима.

Итак, **после пуска ЭВМ** происходит тестирование компьютера, загрузка и передача управления ОС, после чего ЭВМ может выполнять все остальные программы – системные и прикладные по заданию пользователя или ОС. Любая программа выполняется последовательно, по командам, путем обращения процессора к очередным ячейкам памяти, в которых она размещена.

Командный, или рабочий, цикл процессора. Выполнение операций по извлечению из памяти и выполнению одной команды, а также интервал времени, требующийся для этого, носит название *командного или рабочего цикла процессора* (п. 3.2). Цикл команды состоит в свою очередь из одного или нескольких *машинных циклов*, каждый из которых соответствует одному обращению процессора к памяти или периферийным устройствам (п. 3.2, 3.4). Это означает, что общее число машинных циклов в командном цикле равно числу об-

ращений процессора к внешним устройствам (ОП и ПУ) за все время отработки команды.

Итак, командный цикл включает в себя (см. рис. 4.9):

- выборку и дешифрацию команды (блок 7) и
- выполнение команды (блок 8).

Каждое из этих действий может потребовать исполнения одного или нескольких машинных циклов. Выборка команды занимает ровно столько машинных циклов, сколько машинных слов она содержит (сколько ячеек памяти размером в ширину шины данных занимает). В каждом машинном цикле выбирается одно слово команды. Все машинные циклы выборки команды относятся к циклам типа «Чтение» («Чтение команды») (п. 3.3, 3.4).

При выполнении команды могут потребоваться дополнительные внешние пересылки – пересылки операндов или результатов операции, занимающие соответствующее число машинных циклов. Они реализуются процессором в форме циклов «Чтение» («Чтение операнда»), «Запись (операнда)», «Ввод» и «Вывод» (п. 3.3, 3.4). Если таких пересылок не требуется, то указанные машинные циклы отсутствуют, и тогда общее число машинных циклов в команде равно числу машинных циклов ее выборки.

Обычно процессор выбирает следующую команду только после того, как он завершил выполнение текущей. Развитые микропроцессоры, включая процессоры семейства Intel, производят опережающую выборку команд, когда, не дожидаясь окончания текущей команды (проведения связанных с ней внутренних операций), выбираются следующие. В этом случае между машинными циклами текущей команды (внешними пересылками ее операндов) могут встраиваться циклы опережающей выборки следующих команд. В остальном функционирование подобных процессоров не отличается от описанного выше порядка.

Закономерен вопрос, каким образом процессор различает код операции команды (КОП) и операнды (или их адреса), если и те и другие поступают в процессор по шине данных и представлены двоичными кодами. Дело в том, что в начальной ячейке любой команды всегда записан КОП, и каждый КОП содержит информацию, указывающую общее число слов (байт) данной команды. Таким образом, процессор точно знает, что он собирается получить в определенный момент времени. Если программист сделает ошибку, то МП может принять данные за команду или наоборот.

Другие действия процессора при выполнении программы. Во время выполнения программы процессор следит за внешними управляющими сигналами, а также идентифицирует свое внутреннее состояние.

В начале каждого машинного цикла (в случае Intel 80x86 – перед выборкой очередной команды из очереди команд) процессор распознает, какое из двух альтернативных состояний – *Счет* или *Останов* (*Ожидание прерывания*) имеет место (блок 3). В состоянии Останов (Ожидание прерывания) никакие программы не выполняются, а процессор ждет прихода сигнала прерывания INTR (ЗПр) (Intel 80x86 – сигналов INTR и NMI), после чего управление пере-

ходит к соответствующей прерывающей подпрограмме, переводящей процессор в состояние Счет.

МП Intel 80x86 переходит в состояние Останов под действием команды останова hlt, а выходит из него по сигналам пуска RESET и прерывания INTR и NMI. МК PIC16F – под действием команды sleep, переводящий микроконтроллер в режим энергосбережения SLEEP.

В состояние Счет процессор переходит:

- сразу после пуска ЭВМ (по сигналу RESET, воспринимаемому безусловно, в любом состоянии, в т.ч. в состоянии Останова);
- в ответ на запрос прерывания (по сигналам INTR или NMI; причем запросы обслуживаются и в момент Останова);
- и после завершения любой команды, не являющейся командой останова hlt или ожидания wait.

В состоянии Счет при отсутствии запросов прерываний или при условии, что они запрещены, (блоки 4 и 5) процессор выбирает и выполняет очередную команду. Если же имел место запрос прерывания, и при этом прерывания были разрешены, то происходит выход из нормального командного цикла и переход к подпрограмме обработки прерывания (блоки 4, 5 и 6; см. гл.5).

Отметим, что кроме состояния Останов (HALT – ожидание прерывания) МП Intel 80x86 может находиться еще и в состоянии Ожидание (WAIT – ожидание сигнала BUSY). В состояние Ожидание он переводится командой wait. Если на входе BUSY во время выполнения команды wait присутствует низкий уровень напряжения, то Pentium продолжает оставаться в состоянии ожидания, и никакие операции не выполняются. Если же на входе BUSY появляется сигнал высокого уровня и действует как минимум пять тактов синхронизации, то процессор выходит из состояния ожидания WAIT и переходит к выполнению следующей за wait команды.

Во время программного ожидания WAIT возможно прерывание работы процессора Intel 80x86 сигналами INTR и NMI. В этом случае процессор заканчивает исполнение команды wait и переходит к обслуживанию запроса прерывания в обычном порядке. Однако в отличие от прерывания во время HALT процессор опять возвращается, после выхода из подпрограммы прерывания, к выполнению команды wait (т.е. в состояние ожидания), а не к следующей за ней команде (как в случае hlt).

В течение командного и машинного циклов МП следит также за другими внешними сигналами: захвата HLD (Зх), готовности READY (Гт) и другими, генерирует сигналы состояния, синхронизации, ответные сигналы подтверждения прерывания INTA (ППр), подтверждения захвата HLDA (ПЗх), ожидания WAIT (Ож), другие управляющие сигналы, необходимые для нормальной работы устройств ввода-вывода, памяти, сопроцессора и других устройств. Таким образом, центральный процессор является «мозгом», определяющим действия ЭВМ.

Особенности работы процессоров Intel при выполнении программы. Большинство современных компьютеров выполняют команды последовательно. Этим свойством отли-

чаются и все процессоры Intel. Для каждой команды реализуются все действия, описанные в п.4.4-4.6. Команды выбираются из памяти, дешифрируются (преобразуются во внутренний код CPU) и выполняется ее микрокод. В результате получаются циклы выборки-дешифрации и исполнения.

Так как используется предвыборка команд (в очередь команд), а в процессоре Pentium – многоступенчатая конвейеризация команд, то циклы выборки команд и исполнения уже выбранных команд накладываются друг на друга, происходят одновременно. В многоступенчатых конвейерах Pentium частично перекрываются и циклы исполнения соседних команд. В результате время исполнения команды (в смысле выполнения внутренних операций) в большинстве случаев как бы исчезает из цикла команды, и остаются, с точки зрения пользователя, только циклы внешних пересылок – циклы магистрали.

Pentium разделяет выбранные команды на два потока, которые обрабатываются одновременно на двух его конвейерах U и V. Это еще более ускоряет обработку и позволяет увеличить ширину выборки до 64 битов (и до 256 битов из внутренней кэш-памяти), с которой конвейеры успешно справляются и не задерживают общий поток выбираемых команд.

При выборке команды производится инкремент программного счетчика EIP. EIP сообщает процессору, где искать следующую команду. При инкременте EIP команды выполняются в том порядке, в каком они хранятся в памяти. Только при «переходе» или «вызове» подпрограммы и «возврате» из нее в EIP загружается новое значение, и следующая команда выбирается из новой указанной ячейки.

Так как используется предвыборка команд, то значение EIP указывает на загружаемую команду в процессор, а не на ту, которая будет выполняться в процессоре.

Мы уже говорили, что, строго говоря, EIP указывает не на команду, а на командное слово, которое может быть частью длинной команды или содержать несколько команд, крайние из которых, возможно, разрезаны по границе слова. Это слово и загружается в процессор (в очередь команд, где его байты сортируются в порядке возрастания адресов) за один цикл выборки из программной памяти. Размер этого слова 8 байт – при выборке из внешней памяти, и 32 байта – при выборке из внутренней кэш-памяти (такие 32 байта называются строкой кэша). И только при переходах по программе EIP действительно указывает на следующую команду, подлежащую выборке и выполнению вслед за предыдущей. Эта команда и считывается из памяти по адресу EIP совместно с расположенными вслед за ней байтами до конца 8-байтного, физического, командного слова. При этом размер считываемой выборки может быть меньше 8 (если адрес перехода не кратен 8 – «не выровненный»).

Особенность регистра EIP заключается в том, что он отсчитывает команды (командные слова) не от начальной ячейки памяти с адресом 00000000, а от некоторой базовой точки в памяти, адрес которой может быть и ненулевым. Адрес этой точки (названный нами ранее базовым адресом кодового сегмента и обозначенный как CS base) устанавливается программно при помощи другого регистра процессора – указателя кодового сегмента CS (Code Segment) (п. 2.4, 4.5). Для определения полного, 32-битного, линейного адреса команды процессор суммирует базовый адрес CS base (тоже линейный и 32-битный) с 32-битным содержимым программного счетчика EIP. Вычисленный таким образом 32-битный код и выставляется процессором в шину адреса в качестве физического адреса очередной команды, если отсутствует страничное разделение памяти. Если страницы установлены (бит PG в регистре CR0 переведен в единичное состояние), линейный адрес дополнительно преобразуется блоком страниц в физический.

Сначала физический адрес очередной команды поступает на внутреннюю кэш-память. Если команда с этим адресом там имеется (т.е. ранее была загружена в кэш), то процессор выбирает ее оттуда (размер выборки 32 байта – строка кэш-памяти, а при переходе по невыровненному адресу размер выборки меньше). В этом случае внешний цикл выборки команды не требуется, и адрес в системную ША не поступает. Выборка команды производится из внутренней кэш-памяти по внутренней 256-битной ШД.

Если очередная команда во внутреннем кэше не обнаружена, то она выбирается из основной памяти. Размер одноразовой выборки 8 байт – физическое слово; или он меньше 8, если был переход по невыровненному адресу.

Перед выборкой из основной памяти проверяется наличие команды во внешней кэш-памяти, и при положительном ответе происходит выборка из внешнего кэша. Для проведения выборки из внешней (основной или кэш) памяти процессор выставляет адрес команды в системную шину адреса и выполняет цикл считывания команды по системной шине данных. При этом биты 31-3 адреса выставляются в линии A31 – A32 шины адреса, а биты 2-0 преобразуются в значения сигналов на линиях BE7# - BE0#, как показано на рис.3.3, 3.4. Биты A31 – A3 определяют 8-байтную ячейку в памяти, а сигналы BE7# - BE0# – нужные байты в этой ячейке.

4.7. Три режима ввода-вывода

Традиционно выделяют три основных варианта обмена процессора с внешними устройствами, **три режима ввода-вывода** [7]:

- программно управляемый (program polling – программный опрос);
- по прерываниям (interrupts);
- прямой доступ к памяти – ПДП (DMA, Direct Memory Access).

В программном режиме все операции по инициализации и управлению вводом и выводом производятся процессором по заданию программы и только ее. Процессор является *активным* устройством в этом режиме, а память и периферийные устройства – *пассивными*.

Существует три способа выполнения программного режима (рис.4.11). Интерфейсы должны учитывать возможность реализации всех 3-х способов. Первый способ – прямой, используется для синхронных ПУ, т.е. устройств, которые всегда готовы к работе и циклов ожидания не требуется. Второй – условный с занятием цикла, когда при не готовности ПУ, процессор ждет до тех пор, пока наступит его готовность. Третий – условный с совмещением. В отличие от предыдущего, процессор не ждет готовности ПУ, а переходит к продолжению программы с периодической проверкой готовности ПУ.

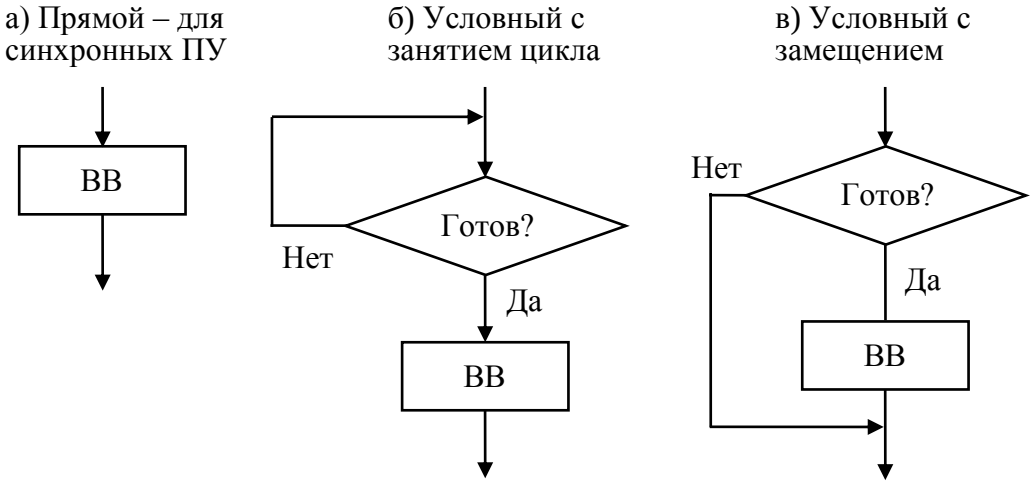


Рис. 4.11. Программный ввод-вывод

Режим прерываний. В этом режиме действия процессора определяются уже не только программой, но и чисто схемными событиями – на процессор поступают управляющие электронные сигналы, вырабатываемые периферийными устройствами, нуждающимися в немедленном обслуживании. Эти сигналы называются *сигналами прерываний*. Под их действием процессор переключается на обслуживание этих устройств и работает под управлением программы (подпрограммы прерывания), извлекаемой из памяти (рис.4.12). Поэтому режим прерываний можно считать *программно-аппаратным*. Прерывание текущей программы инициализируются периферийным устройством, которое ведет себя в этот момент *активно*, а процессор – *пассивно*. После переключения процессора на выполнение подпрограммы прерывания он снова становится *активным*, а ПУ – *пассивным*. Режим прерываний обсуждается в гл.5.

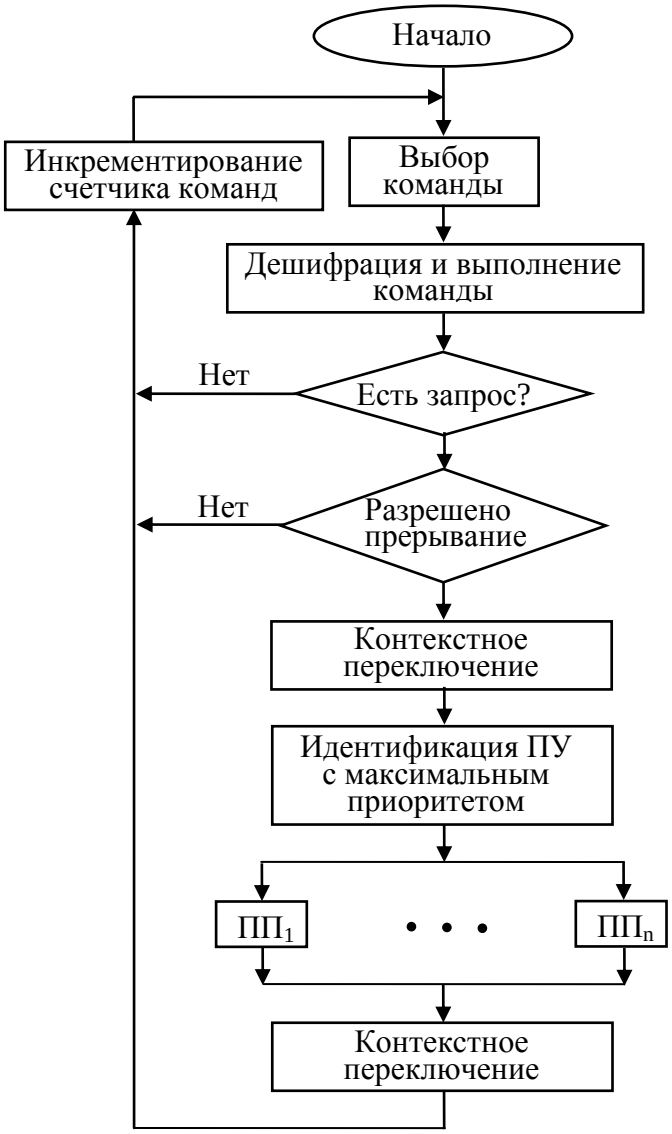


Рис. 4.12. Ввод-вывод в режиме прерывания

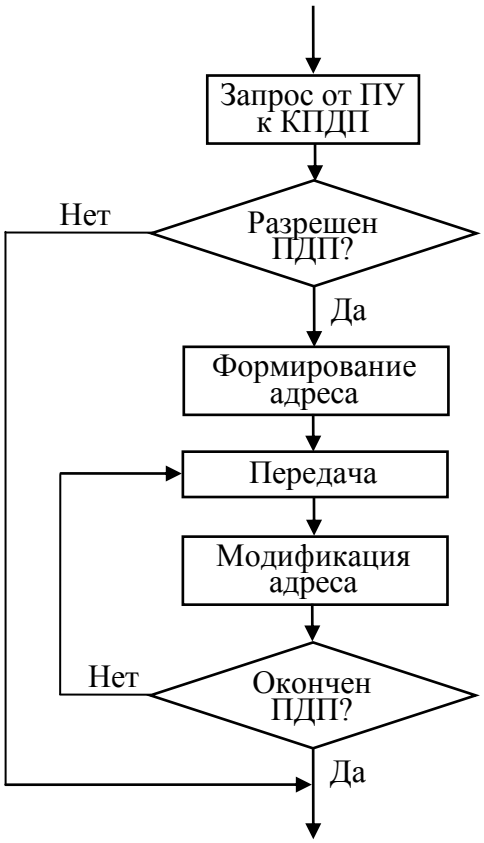


Рис. 4.13. Режим прямого доступа к памяти

Так называемый **прямой доступ периферийного устройства к памяти** ЭВМ представляет собой способ передачи данных между ПУ (например, магнитным диском; или экраном монитора) и основной памятью ЭВМ (способ ввода-вывода), минуя центральный процессор. Этот режим используется для высокоскоростных ПУ, например, для жесткого диска. ЦП на время передачи отключается от системной магистрали и не принимает никакого участия в ней. Управление передачей данных в режиме ПДП полностью перехватывает на себя особая микросхема – контроллер ПДП (КПДП, DMA controller). Работа КПДП ведется не по программе, извлекаемой из памяти ЭВМ, а проходит автономно, изолированно от программы, под управлением собственной электронной схемы (обычно предварительно установленной в нужный режим функционирования центральным процессором). Поэтому ПДП можно считать *аппаратным* режимом работы ЭВМ.

Для того чтобы приступить к ПДП контроллер посылает в процессор заявку на прямой доступ путем подачи на соответствующий вход процессора сигнала захвата магистрали 3х (HOLD) (рис.4.13). Получив от КПДП этот сигнал, процессор приостанавливает работу (завершает текущий шинный цикл), отключается от интерфейса (от системной магистрали) и выдает на контроллер ответный сигнал подтверждения захвата П3х (HLDA – Hold Acknowledge). С этого момента интерфейсом владеет задатчик, т.е. КПДП. Процессор при этом не выполняет контекстного переключения, а может продолжать свою работу, если она не требует интерфейса: выполняет команды, ранее считанные в очередь команд, или команды из внутреннего кэша. Если все подобные команды выполнены и требуется обращение к внешней (основной или кэш) памяти через интерфейс, процессор вводит «холостые» такты, во время которых никакие действия им не выполняются. Управление интерфейсом переходит к КПДП, который посредством выполнения операций чтения и записи передает информацию между ОЗУ и ПУ с соответствующим заданием адресов памяти. В этом режиме используется механизм задания уровня приоритетов для тех ПУ, которые работают с прямым доступом к памяти. Этот режим также должен быть предусмотрен в интерфейсах.

Во время ПДП процессор постоянно отслеживает значение сигнала 3х. Если он обнаруживает, что сигнал захвата снят, т.е. его уровень стал пассивным, он возобновляет прерванную работу по выполнению текущей программы с использованием системного интерфейса.

Средства организации ПДП составляют особую тему для обсуждения и в настоящем пособии не изучаются.

4.8. Разложение команд процессоров IA-32 по машинным циклам

В табл. 4.1 приведены некоторые команды процессоров архитектуры IA-32, их форматы и действие с точки зрения программиста. Ниже в тексте эти команды рассмотрены подробнее. Описаны машинные циклы процессора, генерируемые им при выполнении команд, и производимые в них базовые операции

по пересылке и преобразованию данных. Разложение команд по машинным циклам представлено в форме условных символьных выражений – логических диаграмм. При этом используются следующие обозначения:

← , : = – операция пересылки, копирования, присвоения;

ОчК – регистры очереди команд;

БРг – вспомогательные (буферные) регистры для временного хранения данных;

m(addr) – содержимое ячейки памяти (memory), имеющей адрес addr (addr = база:смещение);

p(port) – содержимое порта ввода-вывода с адресом port;

IPнов – новое значение программного счетчика в командах передачи управления (новое смещение команды);

CSнов – новое значение указателя кодового сегмента (новая база командного сегмента);

CSнов:IPнов – межсегментный адрес перехода (например, пусковой адрес подпрограммы, находящейся в другом сегменте);

CSстар:IPстар – адрес возврата из подпрограммы.

Таблица 4.1 – Описание команд для программиста

<Формат> (мнемокод и код)	<Действие> (псевдокод)
Пересылка: mov AX, [BX] ; 8B 07	AX ← *m*(DS:BX)
Пересылка : mov [DI], AL ; 89 05	*m*(DS:DI) ← AL
Ввод: in AL, 60h ; E4 60	AL ← *p*(60h)
Вывод : out DX, AX ; EF	*p*(DX) ← AX
Декремент: dec [2000] ; FF 0E 00 20	*m*(DS:2000) ← *m*(DS:2000) – 1
Вызов подпрограммы : call B0C0:1000 ; 9A 00 10 C0 B0	SP ← SP - 2; *m*(SS:SP) ← CS; IP ← 1000; CS ← B0C0
Здесь B0C0:1000 = CSнов:IPнов – пусковой адрес подпрограммы. По команде call в стеке сохраняются старые значения IPстар и CSстар, т.е. адрес возврата из подпрограммы (адрес команды после call), а на их место заносятся новые значения IPнов и CSнов, т.е. адрес перехода на подпрограмму.	
Возврат из подпрограммы : ret ; CB	*m*(SS:SP); SP ← SP + 2; CS ← *m*(SS:SP); SP ← SP + 2.
По этой команде в IP и CS заносятся старые значения, ранее запомненные в стеке командой call (адрес команды после call, т.е. адрес возврата).	

Сначала любая команда выбирается из памяти и ее код дешифруется. На выборку команды затрачивается ровно столько машинных циклов, сколько ячеек памяти она занимает. Все циклы выборки команды являются циклами чтения.

Затем команда выполняется. Для этого процессору могут потребоваться дополнительные внешние пересылки, реализуемые в форме машинных циклов

типов «Чтение», «Запись», «Ввод» или «Вывод», используемых для пересылки и преобразования операндов.

В приведенных примерах принято для определенности, что команды, состоящие из четного числа байтов, начинаются с четных адресов памяти, а команды, состоящие из нечетного числа байтов, – нечетных адресов.

При выполнении команд ввода и вывода с прямой адресацией 8-битный адрес порта выдается как в младшие (A7...0), так и в старшие (A15...8) линии шины адреса, т.е. дуплицируется в шину. Если, например, адрес порта равен 60h, то в ША будет выдано значение 6060h.

Выполнение команд процессором 8086 по машинным циклам. Логические диаграммы командных циклов.

Командный цикл команды mov AX, [BX].

M1 "Чтение команды"	M2 "Чтение операнда"
1) CS:IP → ША, IP ← IP + 2	1) DS:BX → ША
2) Чт → ШУ;	2) Чт → ШУ
3) ОчК ← ШД ← *m*(CS:IP) = 8B07	3) AX ← ШД ← *m*(DS:BX)
4) Дешифрация команды	

Командный цикл команды mov [DI], AL.

M1 "Чтение команды"	M2 "Запись операнда"
1) CS:IP → ША, IP ← IP + 2	1) DS:DI → ША
2) Чт → ШУ;	2) Зп → ШУ
3) ОчК ← ШД ← *m*(CS:IP) = 8905	3) AL ← ШД ← *m*(DS:DI)
4) Дешифрация команды	

Командный цикл команды in AL, 60h.

M1 "Чтение команды"	M2 "Ввод операнда"
1) CS:IP → ША, IP ← IP + 2	1) 6060 → ША
2) Чт → ШУ;	2) Вв → ШУ
3) ОчК ← ШД ← *m*(CS:IP) = E460	3) AL ← ШД ← *p*(6060)
4) Дешифрация команды	

Командный цикл команды out DX, AX.

M1 "Чтение команды"	M2 "Вывод операнда"
1) CS:IP → ША, IP ← IP + 2	1) DX → ША
2) Чт → ШУ	2) Выв → ШУ
3) ОчК ← ШД ← *m*(CS:IP) = xxEF	3) AX → ШД → *p*(DX)
4) Дешифрация команды	

Здесь и далее "xx" обозначает последний байт предыдущей команды

Командный цикл команды dec [2000h].

M1 "Чтение команды"	M2 "Чтение команды"
1) CS:IP → ША, IP ← IP + 2	1) CS:IP → ША, IP ← IP + 2
2) Чт → ШУ;	2) Чт → ШУ;
3) ОчК ← ШД ← *m*(CS:IP) = FF0E	3) ОчК ← ШД ← *m*(CS:IP) = 2000
4) Дешифрация команды	

М3 "Чтение операнда и увеличение на 1"

1) DS:2000 → ША
2) Чт → ШУ
3) БРг ← ШД ← *m*(DS:2000)
4) БРг ← БРг + 1

М4 «Запись результата"

1) DS:2000 → ША
2) Зп → ШУ
3) БРг ← ШД ← *m*(DS:2000)

Командный цикл команды call B0C0:1000. Команда call передает управление подпрограмме, расположенной по адресу B0C0:1000. Следующим за ней будет выполнен цикл чтения команды из ячейки CSнов:IPнов = B0C0:1000.

М1 "Чтение команды"

1) CS:IP → ША, IP ← IP + 2
2) Чт → ШУ;
3) ОчК ← ШД ← *m*(CS:IP) = xx9A
4) Дешифрация команды

М2 "Чтение IPнов "

1) CS:IP → ША, IP ← IP + 2
2) Чт → ШУ;
3) ОчК ← ШД ← *m*(CS:IP) = 1000
4) IPнов ← 1000 из ОчК

М3 "Чтение CSнов"

1) CS:IP → ША, IP ← IP + 2
2) Чт → ШУ;
3) ОчК ← ШД ← *m*(CS:IP) = B0C0
4) CSнов ← B0C0 из ОчК

М4 "Запись в стек CSстар"

1) SP → SP – 2, SS:SP → ША
2) Зп → ШУ
3) CS → ШД → *m*(SS:SP)

М5 "Запись в стек IPстар"

1) SP → SP – 2, SS:SP → ША
2) Зп → ШУ
3) IP → ШД → *m*(SS:SP)

Командный цикл команды ret. Команда ret возвращает управление на команду после call.

М1 "Чтение команды"

1) CS:IP → ША, IP ← IP + 2
2) Чт → ШУ;
3) ОчК ← ШД ← *m*(CS:IP) = xx9A
4) Дешифрация команды

М2 "Чтение IPстар из стека"

1) SS:SP → ША; SP ← SP + 2
2) Чт → ШУ;
3) IP ← ШД ← *m*(SS:SP) = IPстар

М2 "Чтение CSстар из стека"

1) SS:SP → ША; SP ← SP + 2
2) Чт → ШУ;
3) CS ← ШД ← *m*(SS:SP) = CSстар

Глава 5. ПРЕРЫВАНИЯ ПРОГРАММЫ

5.1. Понятие прерываний

Общие сведения о прерываниях [11, 12]. Во время выполнения процессором текущей программы в периферийных устройствах или во внешней среде, обменивающейся информацией с процессором через ПУ, а также внутри самого процессора могут возникать события, требующие немедленной реакции на них со стороны процессора.

Реакция состоит в том, что процессор прерывает обработку текущей программы и переходит к выполнению некоторой другой программы, специально предназначенной для данного события. По завершении этой программы процессор возвращается к выполнению прерванной программы.

Рассматриваемый процесс, называемый *прерыванием программы*, поясняется на рис. 5.1. Принципиально важным является то, что моменты возникновения событий, требующих прерывания программ, заранее неизвестны и поэтому не могут быть учтены при программировании.

Каждое событие, требующее прерывание, сопровождается сигналом, оповещающим процессор. Назовем эти сигналы *запросами прерывания*. Программу, затребованную запросом прерывания, называют *прерывающей программой* или (чаще) *подпрограммой* (так как она может запускаться многократно при повторных возникновениях прерывающих событий), противопоставляя ее *прерываемой программе*, выполнявшейся машиной до появления запроса.

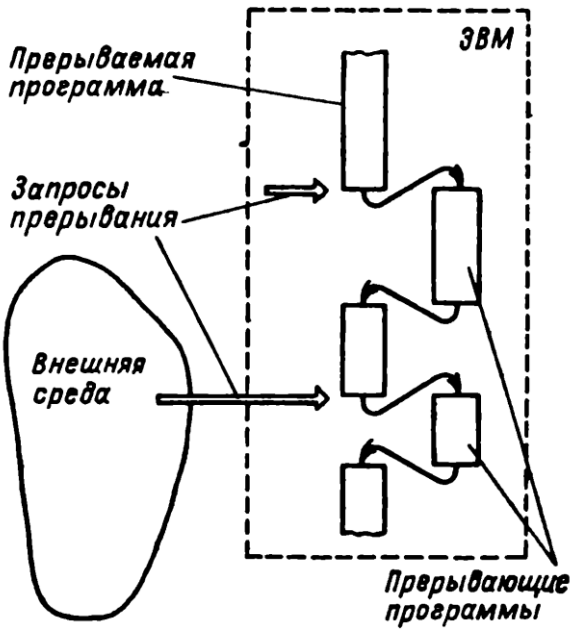


Рис. 5.1. Прерывание программы [1]

Виды прерываний. Различают два вида прерываний: *аппаратные* и *программные* (рис. 5.2). Выше речь шла об аппаратных прерываниях. Говорят также о программных прерываниях. Последние, однако, никакими прерываниями не являются, так как ничего не прерывают. Они представляют собой просто вызовы подпрограмм из основной программы, осуществляемые при помощи специальных команд (о которых речь пойдет ниже).

Аппаратные прерывания подразделяются на *внешние* и *внутренние*. Внешние инициируются запросами внешних устройств, требующих

обслуживания со стороны процессора, внутренние – внутренними средствами диагностики самого процессора.

Поскольку внутренние прерывания являются реакцией процессора на аномальные ситуации, возникающие при выполнении программы, их иногда условно относят не к аппаратным, а к программным прерываниям. Тогда внутренние и программные («чисто программные», происходящие по машинным командам) прерывания вместе называют *особыми случаями* или *исключениями* в программе, или, иногда, как и чисто программные, – просто «программными» (см. рис. 5.2).

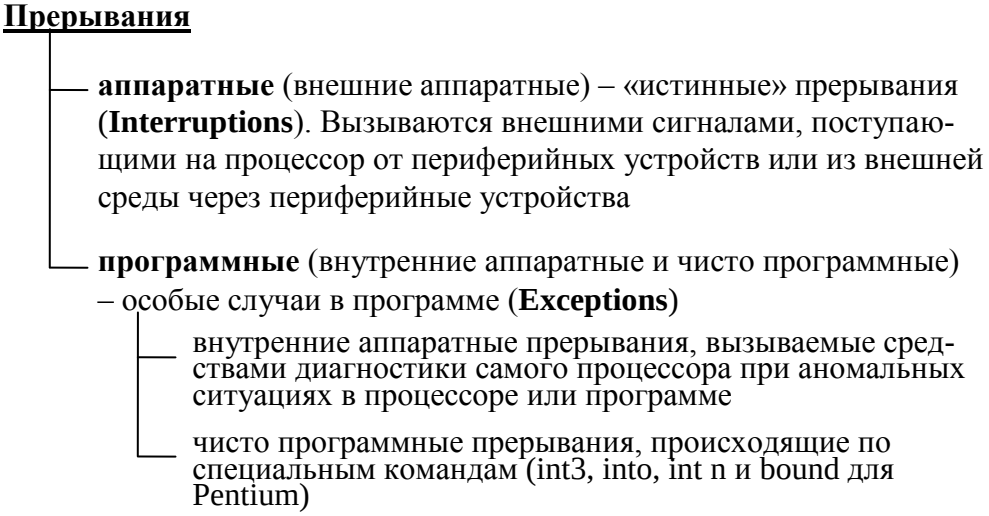


Рис. 5.2. Разновидности прерываний

Внешнее аппаратное прерывание служит мощным средством асинхронных переходов по программе. Как правило, переходы по программе происходят под управлением самой программы. Однако, в любой момент времени у того или иного внешнего устройства может возникнуть внеочередная, не предусмотренная этой программой необходимость выполнения совместных операций с процессором. Она может быть связана с такими событиями, как:

- нажатие клавиши на клавиатуре или кнопки «мыши» пользователем для ввода символов в редактируемый текст или команд в операционную систему; или отжатие клавиши / кнопки для завершения операции;
- подключение «флэшки» к USB-порту;
- сообщение о возникновении ошибок на шине или в памяти, об аварийном выключении сети;
- отладка программы пользователем;
- поступление сигналов от датчиков о возникновении аварийных и нештатных ситуаций на технологическом оборудовании;
- тактирование таймером выработки управляющих сигналов на технический объект в системах управления;
- другие причины, требующие немедленной реакции процессора.

В этом случае внешнее устройство может прервать выполняемую, основную, программу и заставить процессор перейти на другую, нужную ему программу (подпрограмму), которая и осуществит необходимое обслуживание периферийного устройства: введет данные с клавиатуры, выполнит команду от «мыши», организует обмен информацией с флэш-картой, запустит режим аварийного отключения оборудования, выработает управляющий сигнал на объект и т.п.

Прерывание текущей программы осуществляется с помощью специального сигнала – запроса прерывания ЗПр, подаваемого внешним устройством (или, чаще, специальной микросхемой – контроллером прерываний) на соответствующий вход процессора. Адрес передачи управления (*адрес перехода*) на программном счетчике процессора (прямой или косвенный*) задается тоже внешним устройством (контроллером прерываний). Это заставляет процессор перейти по нужному адресу, начиная с которого и располагается подпрограмма обслуживания внешнего устройства – *подпрограмма прерывания* (*подпрограмма обслуживания прерывания, подпрограмма обработки прерывания, обработчик прерывания* – ПП, Interruption SubRoutine – ISR, Interrupt Handler**) (рис. 5.3).

* Для процессоров IA-32 – это косвенный адрес. Он называется номером прерывания *n* (см. п.5.2, 5.5).

** Handle – ручка, рукоять, рукоятка; делать что-либо руками, регулировать. Handler – старший тренер; дрессировщик, укротитель; процедура программы, запущенная в результате определенного действия и обрабатывающая его последствия.

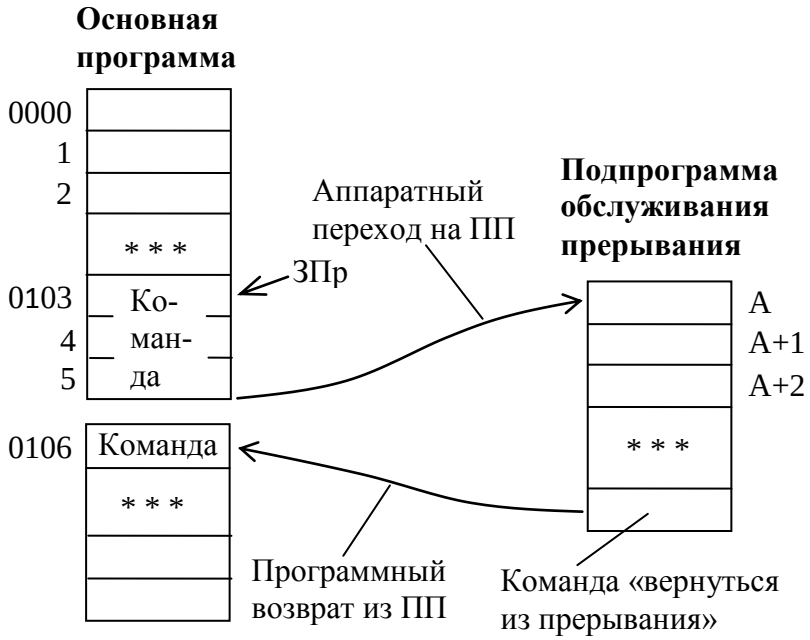


Рис. 5.3. Переход на подпрограмму обслуживания прерывания.
Адрес перехода равен A. Адрес возврата из прерывания 0106.
Запрос прерывания ЗПр поступил на процессор в момент выполнения команды, расположенной в ячейках 0103 - 0105

Такой режим называется *внешним аппаратным прерыванием*. Принципиально построение подпрограмм обслуживания прерываний не отличается от любых других подпрограмм, выполняемых процессором. Отличие состоит лишь в том, как организован переход к этим подпрограммам. Этот переход происходит не по указанию выполняемой в данный момент программы, поскольку, как уже отмечалось, он не предусмотрен ею и может понадобиться в любой момент, в любой точке основной программы, на любой ее команде. Данный переход осуществляется по требованию внешнего устройства, посылающего на процессор специальный сигнал – запрос прерывания и сообщающего процессору адрес перехода на подпрограмму.

В конце подпрограммы прерывания ставится команда «вернуться из прерывания» По ней процессор вновь переключается на исполнение основной, прерванной программы, возвращаясь в точку прерывания – на команду, перед которой прерывание произошло. Правильный возврат производится благодаря тому, что в фазе запроса прерывания адрес этой команды (*адрес возврата*) автоматически запоминается процессором в стеке, а по команде возврата извлекается из стека и заносится обратно в программный счетчик, заставляя процессор возобновить прерванную программу от точки прерывания. В процессорах IA-32 для возврата из прерываний служит команда iret (от англ. interruption return).

Внутренние аппаратные прерывания. Аналогичный режим – режим прерываний может быть вызван средствами диагностики самого процессора при обнаружении ошибки в его функционировании или функционировании программы, о чем мы говорили выше. Такие прерывания мы назвали внутренними аппаратными. Внутренние прерывания происходят при возникновении в процессоре таких событий, как появление ошибки в работе его аппаратуры, поступление неправильного кода команды, переполнение разрядной сетки, попытка деления на 0, выход из установленной для данной программы области памяти и др.

Адрес перехода на подпрограмму прерывания (для IA-32 – номер прерывания n) задается в этом случае тоже аппаратно, но изнутри процессора, а не извне (как при внешних прерываниях). Для большинства процессоров этот адрес (номер) предопределен, заранее назначен для каждого типа внутреннего прерывания при изготовлении процессора. Адресом возврата из внутренних прерываний является, как правило, адрес команды, на которой поступил внутренний запрос, а не адрес следующей за ней команды, как в случае внешних прерываний.

Заключаем, что **прерыванием (аппаратным прерыванием)** называется *временное прекращение выполнения текущей («основной») программы и переход процессора к другой программе – подпрограмме обслуживания прерывания (ПП, ISR), если этот переход не был предусмотрен текущей программой, а произошел асинхронно по отношению к ней, на произвольной, случайно подвернувшейся в этот момент команде и под управлением внешнего устройства (внешние прерывания) или собственных средств диагностики процессора (внутренние прерывания), а не по указке программы*.

Коротко, *прерывание* – это *асинхронный аппаратный переход по программе*.

Истинные прерывания (Interruptions) и особые случаи, или исключения, в программе (Exceptions). К прерываниям в собственном смысле слова следует отнести лишь аппаратные прерывания. В то же время так называемые «программные прерывания» (чисто программные) представляют собой просто вызовы подпрограмм из основной программы с помощью специальных команд (для процессоров IA-32 – команд int 3, into, int *n* и bound). Они именуются так лишь потому, что используют общие с аппаратными прерываниями вектора (адреса подпрограмм). При чисто программных прерываниях (как и при аппаратных) происходит также автоматическое запоминание не только адреса возврата, но и флажков в стеке (при обычных вызовах подпрограмм – по команде call – флажки не запоминаются).

Среди аппаратных внешние прерывания являются безусловно асинхронными, непредсказуемыми в отношении моментов их наступления. Внутренние прерывания не обладают таким свойством безоговорочно и по ряду свойств занимают промежуточное положение между внешними и чисто программными, имея признаки и тех и других. Многие внутренние прерывания могут быть выработаны в самых неожиданных местах программы: прерывания по ошибочному командному коду, нарушению правил доступа к сегментам памяти и т.д. Такие прерывания похожи на внешние в том отношении, что они, как и внешние, происходят асинхронно по отношению к выполняемой программе. Другие внутренние прерывания, наоборот, имеют сходство с программными, поскольку, как и программные, назначаются самим программистом: например, пошаговое прерывание, применяемое для отладки программы или проверки исправности оборудования; либо они, подобно программным, могут наступать только в определенных местах программы: прерывание по делению на «0» (генерируемое командами деления div и idiv при делении на нуль или малое число, если частное не помещается в разрядную сетку регистра-результата).

Тем не менее, общим для внутренних прерываний является то, что все они, в конечном итоге, порождаются программой, являются следствием программных событий. Запросы прерываний, поступающие от внутренних блоков процессора, свидетельствуют о возникновении исключительной ситуации в ходе работы программы: ошибках или затруднениях при выполнении команд, пошаговом прохождении программы, нарушении границ сегмента или правил защиты памяти и т.д. По этой причине внутренние прерывания нередко и причисляют к программным, называя их *особыми случаями* или *исключениями* в программе (*exceptions*). Для удобства терминологии исключениями называют и собственно «программные прерывания» (происходящие по командам int и др.).

В противоположность этому внешние прерывания не связаны с ходом работы программы. Они являются следствием исключительно аппаратных, внешних событий и происходят всегда в произвольные моменты времени, асинхронно по отношению к выполняемой программе. Поэтому они являются прерываниями в полном смысле этого слова, *истинными прерываниями* (*interruptions*).

Значения термина «прерывание». В технической литературе термин «прерывание» используется в нескольких близких, но не вполне совпадающих значениях. Прерыванием называют, во-первых, как было сказано, особый режим работы ЭВМ, заключающийся во внеочередном, не предусмотренном основной программой переходе процессора на другую программу – подпрограмму прерывания и последующем ее выполнении. Во-вторых, прерыванием называют саму подпрограмму обслуживания прерывания (ПП), или, как говорят, *процедуру прерывания*, *обработчик прерывания*, «прерывание». В-третьих, прерыванием или подтверждением прерывания называют первую фазу режима прерываний – автоматическую, аппаратную реакцию процессора на сигнал запроса прерывания (п. 3.3). И, наконец, в-четвертых, прерыванием именуют внешний сигнал, переводящий процессор в режим прерывания – сигнал запроса ЗПр, INTR (Interrupt).

Прерывания от нескольких устройств. Понятие внешнего прерывания можно обобщить для случая нескольких внешних устройств. Любое из устройств может прислать свой запрос на внимание. Обработка запросов следует по принципу обслуживания «первым вошел – первым вышел», либо по некоторой приоритетной схеме в соответствии с приоритетом запросов. Эта операция выполняется обычно с помощью специальных микросхем – блоков приоритетных прерываний, или контроллеров прерываний.

Резюме. При всех случаях способность обрабатывать прерывания может оказаться очень мощным средством во многих приложениях микропроцессоров. Поскольку при прерываниях нужно сохранять информацию о текущем состоянии МП, можно воспользоваться стеком, т.е. сохранить в нем внутреннее состояние МП в момент поступления запроса, включая программный счетчик, т.е. адрес команды, перед которой запрос поступил (адрес возврата из прерывания).

Отметим, что все МП реагируют на запросы только после завершения текущей команды, так как проверяют состояние входа ЗПр (INTR, NMI), а также наличие внутренних запросов в промежутке между командами.

5.2. Обработка внешних прерываний

Порядок обработки внешних прерываний. Внешнее устройство, нуждающееся во внеочередном обслуживании процессором, посылает в процессор асинхронный сигнал запроса (требования) прерывания ЗПр (ТПр, INTR) (см. рис. 5.3). Реагируя на этот сигнал, процессор завершает выполнение текущей команды и, при условии, что прерывания разрешены, прерывает свою работу по исполнению текущей программы и совершает далее автоматически следующие действия (см. также п. 3.3):

а) вводит из периферийного устройства (чаще – из контроллера прерываний) *адрес перехода* (прямой или косвенный) на подпрограмму обслуживания ПУ, с помощью которого процессор определяет местоположение этой подпрограммы в памяти (например, в случае процессоров Intel 80x86, процес-

сор вводит *номер прерывания n*, служащий косвенным адресом перехода на подпрограмму прерывания; см. п.5.5).

Эта действия реализуется в форме:

а') одного или нескольких (в зависимости от типа микропроцессора) машинных циклов типа «Прерывание» («Подтверждение прерывания») (см. рис. 3.8),

за которыми могут следовать (в случае косвенной адресации ПП);

а") один или несколько машинных циклов типа «Чтение» для окончательного считывания адреса ПП из ячейки памяти, указанной косвенным адресом (номером *n*);

б) запоминает *адрес возврата* из подпрограммы и, возможно, другую полезную информацию о своем состоянии, например флажки, в стеке. В результате процессор будет знать, куда потом, после завершения ПП, ему надо будет вернуться в память для того, чтобы продолжить выполнение прерванной программы.

Операция «б» реализуется в форме:

одного или нескольких циклов типа «Запись в стек».

Эти действия могут выполняться и в противоположном порядке – сначала б), потом а).

Адрес перехода на ПП, *пусковой адрес* ПП – адрес ее стартовой команды (прямой адрес), принято называть *вектором прерывания*. Вектор прерывания (ВП) задает ячейку, где хранится стартовая команда подпрограммы прерывания в памяти ЭВМ, как бы «показывает» на эту команду.

Адрес возврата – это адрес команды основной программы, на которую происходит возврат из ПП при помощи команды «вернуться из прерывания».

Мы пояснили, что сигнал прерывания ЗПр (INTR) принуждает процессор выполнить предопределенную последовательность машинных циклов, реализующих операции «а» и «б». Эти циклы не требуют управления со стороны программы. Они «встроены» в электронную схему процессора и генерируются ею автоматически в ответ на появление активного уровня напряжения на входе ЗПр. Эти циклы не могут быть изменены пользователем никакими программными или аппаратными средствами. Они обязательны, фиксированы для процессора данного типа и не зависят ни от работы программы, ни от действий внешних устройств. Они представляют собой, так сказать, «рефлексивную» реакцию процессора на сигнал прерывания.

После завершения указанной последовательности машинных циклов процессор переключается на подпрограмму, осуществляющую обслуживание периферийного устройства, запросившего прерывание.

5.3. Разрешение и запрещение прерываний

Процессор воспринимает сигнал прерывания ЗПр не при любых условиях, а только в том случае, если прерывание разрешено – при этом говорят, что процессор *не замаскирован* от прерываний. В противном случае, если прерыва-

ние запрещено, сигнал ЗПр не проходит в процессор – процессор «замаскирован» от прерываний. Прохождением запросов прерываний в процессор управляет специальный внутренний триггер процессора, называемым *триггером* или *флагом разрешения прерывания* IF (Interruption Flag). У МП IA-32 флаг IF входит в состав регистра флагов и может блокировать сигнал INTR (но не NMI; п.2.5, рис.2.14). У некоторых микропроцессоров этот триггер называется триггером приоритета. Обычно, в т.ч. у МП IA-32, в состоянии логической «1» данный триггер разрешает прохождение запросов в процессор, а в состоянии логического «0» – блокирует запросы (см. рис.2.13). Для ряда процессоров значения состояния триггера – триггера приоритета интерпретируются наоборот: при единичном состоянии триггера процессор не реагирует на сигнал прерывания, т.е. имеет высокий приоритет; при нулевом состоянии – воспринимает этот сигнал – низкий приоритет процессора.

Посредством специальных машинных команд триггер разрешения прерывания доступен программисту, и программист может изменять его нужным образом, разрешая или запрещая прерывания (п. 2.5, рис. 2.14). Кроме того, для многих процессоров (в том числе для Intel 80x86) этот триггер после восприятия очередного запроса автоматически переходит в состояние, соответствующее блокированию дальнейших запросов, выход из которого возможен посредством команды, разрешающей прерывание. Для этих процессоров данный триггер блокирует прерывания также сразу после пуска по сигналу сброса Сбр (RESET). Следовательно, сразу после пуска процессора прерывания запрещены.

При использовании конкретного процессора необходимо внимательно анализировать все ситуации, которые приводят к маскированию и демаскированию прерываний.

5.4. Обработка внутренних и программных прерываний

Прерывания от источников внутри процессора обрабатываются в том же порядке, что и внешние прерывания, но не требуют обращения процессора к периферийному устройству (контроллеру прерываний) для считывания адреса/номера прерывания; этот адрес/номер задается внутренней схемой диагностики самого процессора. Поэтому у внутренних прерываний отсутствуют машинные циклы типа «Прерывание». В других отношениях реакция процессора на эти прерывания аналогична таковой на внешние.

Внутренние прерывания воспринимаются процессором безусловно, т.е. их обработку нельзя запретить. Внутренние прерывания имеют, как правило, более высокий приоритет, чем внешние, т.е. при возникновении одновременно внутреннего и внешнего запросов будет обслужен внутренний.

«Программные» (чисто программные) прерывания вызываются специальными машинными командами, помещаемыми в нужные места программы. Это коренным образом отличает их от аппаратных прерываний, поскольку делает их плановыми, наступающими в заранее намеченных точках программы. Они предусмотрены самим программистом для лучшей организации работы про-

граммы. Понятно также, что их нельзя запретить, как нельзя запретить выполнение ни одной команды в программе.

В случае программных прерываний (как и внутренних) тоже отсутствуют машинные циклы типа «Прерывание», поскольку адрес перехода на ПП задается командой, а не периферийным устройством. Вместо циклов прерывания выполняются циклы выборки команды.

5.5. Прерывания процессоров IA-32

Виды прерываний IA-32. Процессор архитектуры IA-32 обслуживает все упомянутые выше виды прерываний (рис. 5.4).

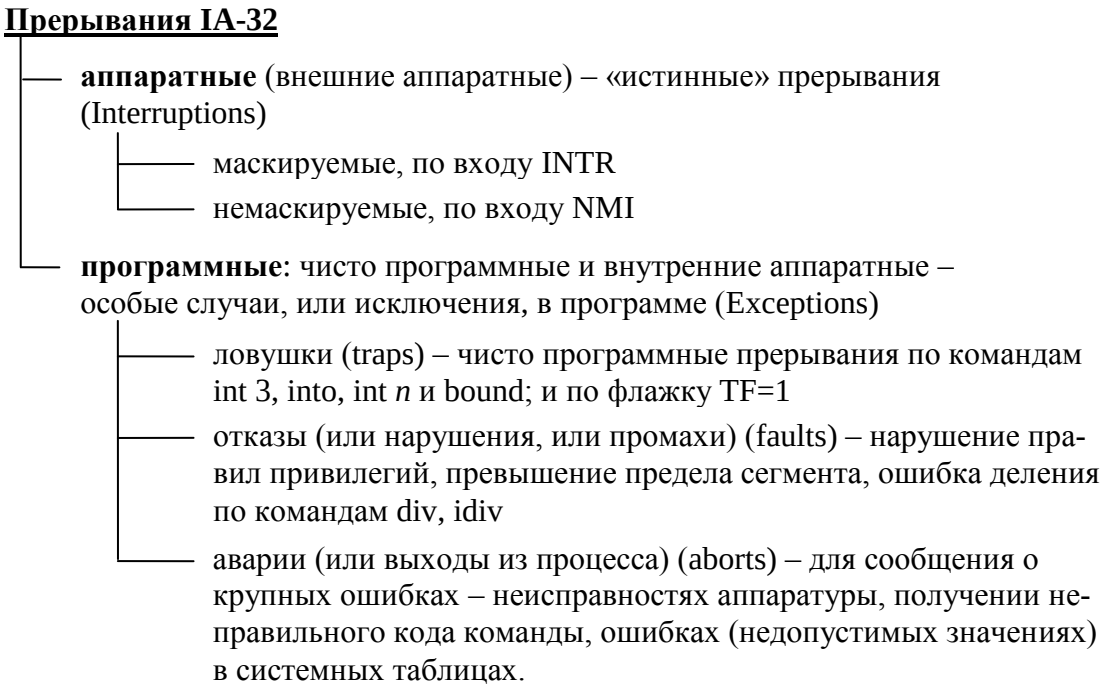


Рис. 5.4. Виды прерываний IA-32

Причинами прерываний текущей программы процессора IA-32 могут быть:

- внешний сигнал по входу маскируемых (INTR, Interruption) или немаскируемых (NMI, No Masked Interruption) прерываний (*внешнее аппаратное прерывание*, *истинное прерывание – Interruption*);
- аномальная ситуация, сложившаяся при выполнении конкретной команды и препятствующая нормальному продолжению программы (*внутреннее аппаратное прерывание*, или *исключение – Exceptions*).
- находящаяся в программе команда прерывания int 3, into, int *n* или bound (собственно *программное прерывание*).

Для краткости исключениями называют не только особые случаи в программе, но и чисто программные прерывания (по командам int, .. , bound). И об-

ратно, поскольку внутренние прерывания являются следствием программных событий, их нередко условно квалифицируют как программные

Адрес перехода на подпрограмму прерывания определяется через 8-битный номер прерывания *n*, который задается процессору аппаратно внешним устройством (контроллером прерываний) – при внешних прерываниях; внутренними средствами диагностики – при внутренних; и командами int и bound – при собственно программных (п. 5.6). Адрес перехода во всех случаях является сегментированным. Он подлежит загрузке в пару регистров CS:EIP/IP.

Адрес возврата из прерываний (тоже сегментированный), а также флажки автоматически запоминаются процессором в стеке сразу после восприятия прерывания.

Возврат из подпрограмм прерываний процессоров Intel, как аппаратных, так и программных, производится по команде iret, устанавливаемой в конце ПП. Под ее действием адрес возврата и флажки извлекаются из стека, заносятся в регистры CS, EIP/IP и EFlags/Flags, и процессор в следующем командном цикле возобновляет выполнение основной программы от точки прерывания.

Исключения (внутренние и программные прерывания) делятся на *ловушки* (traps), *отказы* (нарушения, промахи, faults) и *аварии* (ошибки, выходы из процесса, aborts). Это деление проводится в зависимости от способа сообщения о них и возможности перезапуска процессора с вызвавшей их команды (см. далее табл. 5.1). *Ловушка* – это исключение, которое возникает непосредственно после выполнения команды, на которой оно поступило, при этом возврат из прерывания происходит на следующую за ней команду. Примером ловушки является прерывание/исключение по команде int, устанавливаемое пользователем. *Отказы* – это исключения, которые выявляются и обслуживаются перед выполнением команды, их вызвавшей (в том смысле, что возврат происходит на эту же команду, а не следующую). Они позволяют перезапустить эту команду или всю выполняющуюся программу. При перезапуске те аномальные ситуации в программе, которые вызвали отказы, могут быть устранены, т.е. отказы не приводят к нарушению работы всей системы. Отказы могут иметь место, например, в виртуальной системе памяти, когда процессор обращается к несуществующим странице или сегменту. В процессе обработки такого исключения операционная система обращается к странице или сегменту на диске, а микропроцессор перезапускает команду. В отличие от отказа, *авария* или *выход из процесса* является исключением, которое не позволяет точно локализовать вызвавшую его команду. Поэтому при аварии нарушается работа всей системы и требуется ее перезапуск. Аварии используются для сообщения о крупных ошибках, таких как неисправность аппаратуры или ошибки в системных таблицах. С примерами исключений можно познакомиться далее по табл. 5.1.

Адресом возврата из процедуры обработки исключений (кроме команд int и других ловушек) всегда является команда, вызвавшая исключения, возможны префиксы.

Источники прерываний IA-32 и их номера. Возможные источники прерываний IA-32 показаны на рис.5.5. Источники различаются своими номе-

рами, идентифицирующими данный источник для микропроцессора. Номер источника *n* называется *номером* (или *типом* – type) *прерывания*, или *номером вектора прерывания* (НВП). Поскольку номер *n* состоит из 8 двоичных разрядов, допускается обработка до $2^8 = 256$ типов прерываний/исключений с номерами $n = 0, 1, 2, \ldots, 255$.

Рис. 5.5. Источники прерываний процессоров Intel

Внешние устройства – источники прерываний сообщают процессору о необходимости обслуживания подачей сигналов по специальным линиям, которые называются IRQ-линиями (Interruption ReQuire – запрос не прерывание). Эти линии заводятся на входы специального устройства – контроллера прерываний (см. рис. 5.5). Контроллер запоминает запросы и формирует на их основе сигнал INTR на процессор, соответствующий источнику с наибольшим приоритетом в данный момент. Если поступает одновременно несколько запросов IRQ, контроллер пропускает на процессор запрос с бóльшим приоритетом. Остальные запросы запоминаются контроллером и обслуживаются позже. Приоритеты устанавливаются при включении компьютера по умолчанию в последовательности, указанной ниже в «Раскладке прерываний», но могут быть и перепрограммированы. В современных компьютерах контроллер прерываний встроен в микросхему системного контроллера в качестве одного из ее блоков.

Раскладка прерываний, существующая в каждом персональном компьютере на сегодняшний день. Линии IRQ расположены по убыванию приоритета:

IRQ0 – прерывание от системного таймера, поступающее с некоторой заданной частотой. Используется для смены процессов и задач в мультипрограммной среде и др. целей;

IRQ1 – прерывание от клавиатуры;

IRQ2 – cascade (вывод на вторую микросхему контроллера линий прерываний);
IRQ8 – прерывание от часов реального времени, отсчитывающих реальное (оно же – системное) время;
IRQ9 – прерывание от контроллера графической платы;
IRQ10 – зарезервированная линия (свободно);
IRQ11 – прерывание от USB порта;
IRQ12 – прерывание от «мыши»;
IRQ13 – прерывание от математического сопроцессора;
IRQ14 – прерывание от контроллера жестких дисков;
IRQ15 – прерывание от IDE-контроллера жестких и оптических дисков;
IRQ3 – прерывание от последовательного порта COM1 (мышь);
IRQ4 – прерывание от последовательного порта COM2 (модем);
IRQ5 – зарезервированная линия (свободно);
IRQ6 – прерывание от контроллера гибких дисков (Floppy);
IRQ7 – прерывание от параллельного порта LPT (от принтера).

Нерегулярная нумерация линий IRQ связана с тем, что в первых машинах IBM AT использовалось **каскадирование двух контроллеров прерываний 8259**, один из которых был ведущим, другой – ведомым (рис. 5.6). Каждый из контроллеров обслуживал до 8 приоритетных уровней прерываний, а оба вместе – 15 уровней. Для каскадирования использовалась линия IRQ2 ведущей микросхемы. В результате номера линий следовали не подряд. Эта нумерация сохранена в современных компьютерах, чтобы избежать переделки программного обеспечения.

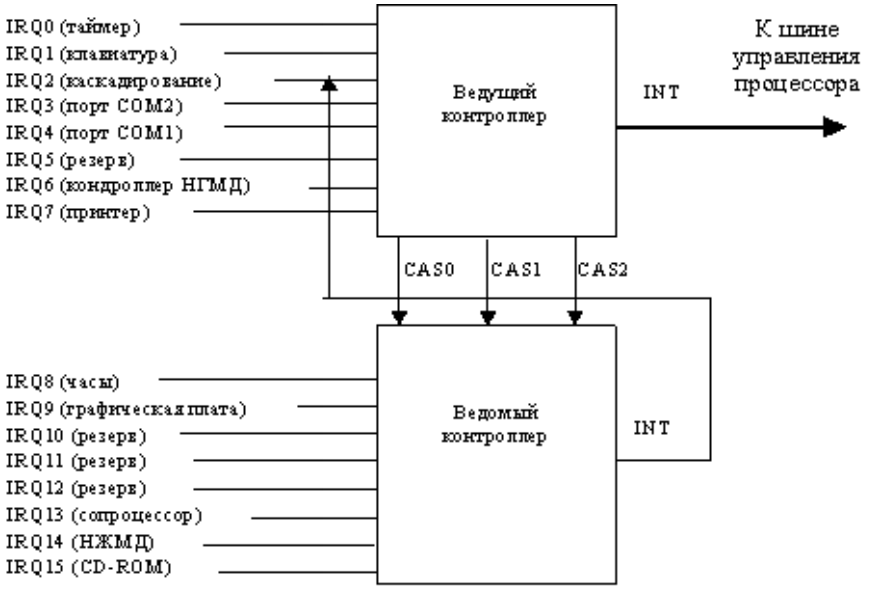


Рис. 5.6. Каскадирование прерываний

Перечень типов прерываний. Различные типы прерываний и их номера приведены в табл. 5.1. Первые 32 типа зарезервированы фирмой Intel для системных целей, остальные предоставлены пользователю.

Таблица 5.1 – Перечень прерываний/исключений

Номер *n*	Назначение	Источник	Тип исключения
00	Ошибка деления («деление на 0»)	Команды div, idiv	Отказ
01	Исключение для отладки (пошаговый режим)	Флажок трассировки TF	Ловушка
02	Немаскируемое прерывание	Сигнал NMI	Прерывание
03	Контрольная точка	Однобайтная команда int3	Ловушка
04	Арифметическое переполнение	Однобайтная команда into при условии OF=1	Ловушка
05	Превышение границы массива; или печать экрана (Print Screen)	команда bound; или клавиша Print Screen	Отказ
06	Недействительный код команды	Любая команда	Отказ
07	Процессор FPU недоступен	esc, wait	Отказ
08	Двойная ошибка; или системный таймер IRQ0	Любая команда, вызывающая исключение; или сигнал IRQ0	Авария
09	Клавиатура	Сигнал IRQ1	Прерывание
0Ah (10)	Недействительный TSS (сегмент задач)	jmp, call, iret, int	Отказ
0Bh (11)	Отсутствие сегмента	Команды, использующие сегментные регистры	Отказ
0C (12)	Нарушение стека	Команды, использующие стек	Отказ
0Dh (13)	Нарушение общей защиты	Любое обращение к памяти	Отказ
0Eh (14)	Отсутствие страницы	Любое обращение к памяти или выборка команды	Отказ
0Fh (15)	Зарезервировано		Отказ
или 0A-0Fh		IRQ2-IRQ7	Прерывание
10h (16)	Видеофункции BIOS	Обращение к видеопамяти	
11h (17)	Ошибка выравнивания	Обращение к невыровненному адресу	Отказ
12-1Fh (18-31)	Зарезервировано		
33h	Поддержка мыши	IRQ12	Прерывание
70h	Часы реального времени	IRQ8	Прерывание
20-FFh (32-255)	Для пользователя	Внешние маскируемые прерывания, int	
00-FFh (0-255)	Программные вызовы	Двухбайтные команды int *n*	Ловушка

Некоторые номера (почти все первые номера от 0 до 31) закреплены за прерываниями аппаратно, фиксированы, «встроены» в схему процессора и не

могут быть изменены никаким способом. Это имеет место для всех особых случаев и внешнего немаскируемого прерывания NMI, которому присвоен номер *n*=2. Другие номера формируются внешними схемами – контроллером прерываний (это все внешние аппаратные прерывания по входу INTR), еще одни зарезервированы фирмой Intel для вызова служб операционной системы и для дальнейшего расширения, а все остальные находятся в распоряжении программиста.

5.6. Обработка прерываний и исключений процессорами IA-32

Общая структура процесса обработки прерываний и исключений IA-32. Обработка всех прерываний и исключений осуществляется, в общем, одинаково, и состоит из двух основных этапов. На первом из них процессор выполняет некоторые «рефлексивные» операции, которые однотипны для всех видов прерываний и которыми программист управлять не может (п. 5.2, 5.4). Их результатом является переход на подпрограмму обслуживания прерываний. На втором этапе действует созданный программистом обработчик прерываний или исключений. Важно отметить, что запросы внешних прерываний и внутренних исключений распознаются на границах команд, поэтому не разрывают командные циклы и не нарушают их выполнение. Т.е. при поступлении запроса процессор сначала завершает выполнение текущей команды и потом только реагирует на запрос. Все служебные действия процессор производит автоматически. С каждым источником прерывания ассоциируется номер *n* от 0 до 255, идентифицирующий данный источник для процессора (п. 5.5).

Примечание. Имеется несколько случаев, когда сигналы внешних прерываний не распознаются и до завершения следующей команды. Префиксы повторения, блокировки шины и замены сегмента считаются частью команды, перед которой находится любой из них; поэтому прерывание не воспринимается между префиксом и командой. Команды пересылки (mov) в сегментный регистр и извлечения (pop) из стека в сегментный регистр рассматриваются аналогично: прерывание не распознается до завершения следующей за ними командой. Такой механизм защищает программу, которая переходит к новому стеку путем изменения содержимого регистров SS и ESP/SP. Если бы прерывание распознавалось после модификации SS, но до модификации ESP/SP, процессор включал бы содержимое регистров флажков, CS и EIP/IP в неправильную область памяти [8].

Имеется также два случая (команда wait – ожидание до поступления сигнала BUSY=1 и цепочечная команда с повторением), когда запрос прерывания распознается в середине команды. В этих случаях прерывания воспринимаются после любой законченной цепочечной операции или цикла проверки сигнала на входе BUSY [8].

Обработка прерываний и исключений в реальном режиме. При работе в реальном режиме (R-режиме, Real) процессор IA-32 обрабатывает все те же прерывания/исключения, что и процессор 8086, и несколько дополнительных исключений. По номеру *n*, посылаемому источником прерываний, процессор выбирает один из 256 обработчиков прерываний, обращаясь к таблице, находящейся в памяти. Эта таблица состоит из 256-ти 4-байтных элементов («векторов») и называется *таблицей векторов прерываний* (ТВП). Она занимает в памяти 4×256 = 1024 байт = 1 Кбайт, и после пуска процессора располагается с

нулевого адреса (диапазон адресов 00000-003FF). Каждый элемент таблицы (*вектор прерывания*, ВП) представляет собой полный, сегментированный адрес (селектор:смещение) запуска подпрограммы, которая обслуживает данный источник прерывания. 16 старших разрядов вектора (селектор) подлежат загрузке в регистр CS, а другие 16 (младших) разрядов (смещение) – в регистр IP (рис. 5.7).

Когда возникает прерывание, процессор включает в стек текущее содержимое регистров CS и IP и флажков Flags, запрещает прерывания (сбрасывает флажок IF) и передает управление ячейке, указанной вектором в таблице. Для нахождения вектора процессор умножает номер *n* на четыре (двойной сдвиг влево), определяя тем самым *адрес вектора прерывания*:

$$АВП = 4 \times n, \qquad (5.1)$$

причем этот адрес является линейным, не сегментированным. По этому адресу процессор считывает сам вектор, загружает его в пару регистров CS:IP и переходит на обработчик (п.5.7).

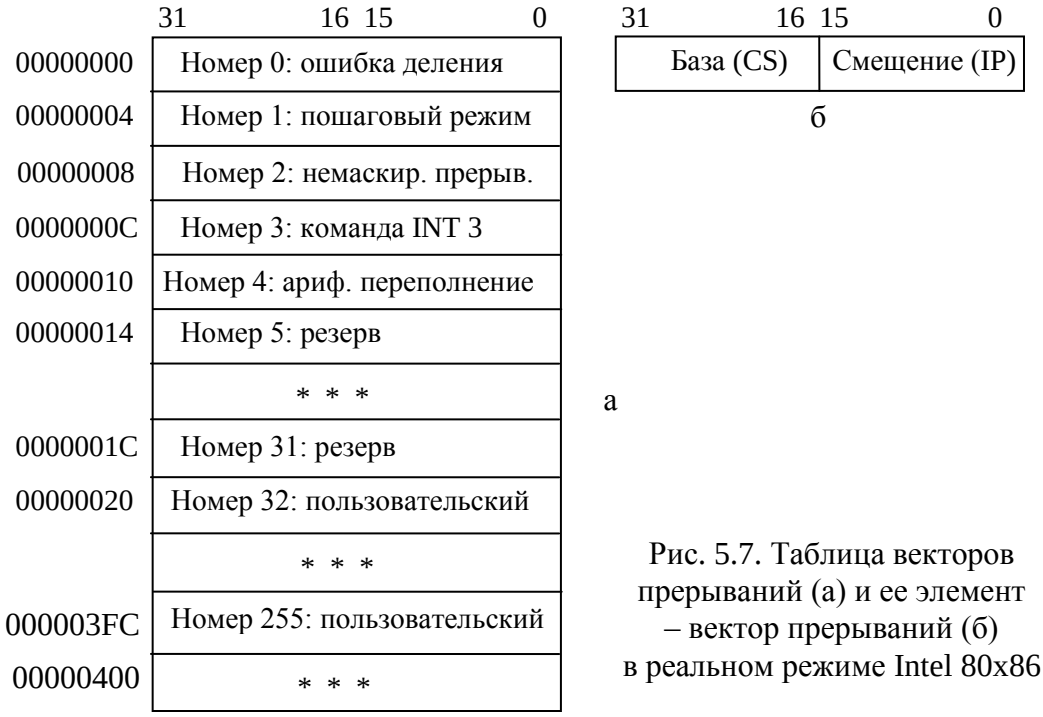


Рис. 5.7. Таблица векторов прерываний (а) и ее элемент – вектор прерываний (б) в реальном режиме Intel 80x86

Перед возвратом в прерванную программу команда iret в конце процедуры прерывания выполняет обратные действия со стеком – восстанавливает CS, IP и флажки. В R-режиме особые случаи (исключения) не формируют код ошибки.

Обработка прерываний в реальном режиме происходит в целом, как и у процессора 8086. Основное отличие заключается в том, что в случае IA-32 местоположение и размер таблицы прерываний зависит от содержимого регистра IDTR дескрипторной таблицы прерываний. Обычно этот факт не очевиден программистам, так как после инициализации при сбросе регистр IDTR содержит базовый адрес 0 и предел 3FF, что соответствует процессору 8086. Однако в R-режиме с помощью команды lidt загрузки регистра IDT можно изменить значе-

ние базового адреса и предела в регистре IDTR. Если возникает прерывание и его элемент в таблице прерываний находится вне предела, находящегося в регистре IDTR, формируется особый случай двойного нарушения [4, 9, 10, 19].

Обработка прерываний и исключений в защищенном режиме. Дескрипторная таблица прерываний. Обработка прерываний и исключений в защищенном режиме (Р-режиме, Protected) аналогична обработке их в реальном режиме, но переход на подпрограмму обслуживания осуществляется через таблицу 64-битных дескрипторов, а не 32-битных векторов.

Таблица прерываний в защищенном режиме, а также в V86-режиме является прямой заменой таблицы векторов прерываний. Она представляет собой массив 8-байтных дескрипторов и называется *дескрипторной таблицей прерываний* (ДТП, IDT) (рис. 5.8). Как и ТВП, таблица IDT должна определять 256 обработчиков прерываний и особых случаев. Поэтому ее максимальный размер равен 8×256=2048 байт = 2 Кбайт (или 0800h байт), а предел 2048-1=2047 (или 07FF). Конечно, всегда можно определить таблицу IDT меньшего размера, но это не рекомендуется. Если возникает особый случай и его номер выбирает дескриптор за пределом таблицы IDT, генерируется внутреннее прерывание «нарушение общей защиты».

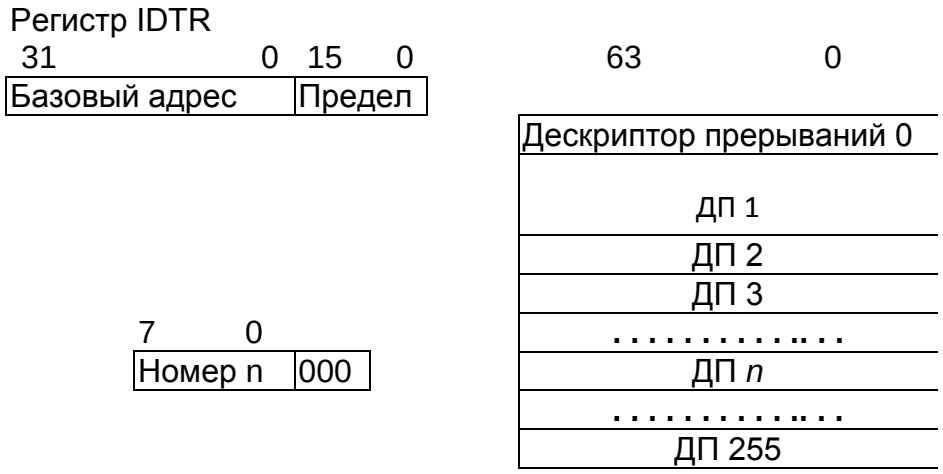


Рис. 5.8. Дескрипторная таблица прерываний IDT

Элемент таблицы IDT – *дескриптор прерывания* (ДП) – представляет собой 64-битную структуру данных, содержащую сегментированный виртуальный пусковой адрес ПП, а также информацию о правилах перехода на ПП. ДП осуществляет связь между устройством или программой – источником прерываний и программой – приемником и позволяет процессору автоматически проверять условия защиты. Коротко, *дескриптор прерывания* – это *допустимый* (*контролируемый, охраняемый*) *вход в подпрограмму прерывания*.

По-прежнему каждому аппаратному прерыванию и программному исключению присваивается номер *n*, по которому процессор распознает данное прерывание. На основании *n* он обращается к IDT. Для этого он сначала вычисляет адрес элемента IDT путем умножения номера *n* на 8 (по числу байт в дескрипторе) (тройной сдвиг влево) и сложения полученного значения с базовым

(начальным) адресом таблицы IDT, взятым из регистра IDTR. Это дает линейный *адрес* затребованного *дескриптора прерывания*:

$$\text{АДП} = \text{базовый адрес IDT} + 8\times n \qquad (5.2)$$

Теперь из ячейки с этим адресом процессор считывает дескриптор, извлекает из него пусковой адрес подпрограммы прерывания и заносит его в пару CS:EIP, что обеспечивает в следующем шинном цикле выборки команды переход на подпрограмму. Кроме того, в самом начале указанных действий, сразу после поступления прерывания, в стеке сохраняется содержимое CS, EIP и EFlags, что позволяет процессору впоследствии осуществить возврат в прерванную программу.

Различные источники прерываний/исключений задают номер *n* по-разному: аппаратное маскируемое прерывание INTR вводит *n* посредством шинного цикла «Подтверждение прерывания» из контроллера прерываний по шине данных D7-D0 (см. рис. 5.5); немаскируемое аппаратное прерывание NMI и исключения вводят его изнутри процессора и эти номера фиксированы (для NMI *n* = 2, и т.д.; см. табл. 5.1); команды int содержат (int *n*) или предполагают (into, int 3) этот номер. Если возникает исключение *n* и его номер выбирает дескриптор за пределом таблицы IDT, генерируется нарушение общей защиты. Нужно учитывать также, что при аппаратном прерывании из-за неисправности шины данных процессор может считать как номер прерывания байт 0FF (все единицы) и обратиться к соответствующему дескриптору. Когда таблица IDT короткая, при этом возникает особый случай общей таблицы. В отличие от глобальной дескрипторной таблицы GDT первый элемент таблицы IDT (с индексом равным нулю) содержит действительный дескриптор.

Таблица IDT может располагаться в любой области памяти. Как показано на рис. 5.8, процессор находит ее с помощью регистра IDTR, который содержит 32-битный линейный базовый адрес и 16-битный предел для таблицы IDT. Команды lidt загрузки и sidt сохранения IDTR позволяют загрузить в регистр новое значение и запомнить текущее в памяти. Обе команды имеют один операнд, являющийся адресом восьми байтов в памяти.

Элементы таблицы IDT являются системными дескрипторами типа шлюз, т.е. представляют собой полные сегментированные виртуальные адреса перехода на подпрограммы прерывания, и содержат также дополнительные правила перехода. Шлюз осуществляет связь между программой или устройством-источником и программой-приемником и позволяет процессору автоматически проверять условия защиты задач. Или, отметим еще раз, шлюз прерывания (как и шлюз вызова) – это допустимый (контролируемый, охраняемый) вход в подпрограмму прерывания.

В таблице IDT могут размещаться дескрипторы шлюзов трех типов: шлюза прерывания, шлюза ловушки и шлюза задачи. Формат системных дескрипторов типа шлюз приведен на рис. 5.9. Шлюзы прерывания и ловушки используются для вызова соответствующих процедур обслуживания, а шлюз задачи – для переключения задач. Таким образом, запрос прерывания может обрабатываться процедурой прерывания или задачей прерывания.

Для шлюза задачи поле смещения (байта 0,1,6,7) игнорируется и зарезервировано для будущего применения.

Шлюзы прерывания и ловушки содержат селектор кодового сегмента и смещение внутри него; совместно они однозначно определяют точку входа в обработчик прерывания/ловушки. Формат шлюза задачи аналогичен формату такого шлюза в таблицах GDT и LDT. Отметим единый формат и значение шлюзов прерывания/ловушки с шлюзами вызовов; единственное различие касается того, что в этих шлюзах нет 5-битного поля счетчика WC. По определению прерывание и исключение являются неожиданными событиями и не связаны с текущей программой, поэтому говорить о передаче параметров их обработчику не приходится.

Поле уровня привилегий DPL шлюза задачи, ловушки и прерывания определяет тот минимальный уровень привилегий, который необходим для передачи управления через шлюз. В большинстве систем рекомендуется устанавливать поле DPL=3, чтобы дескрипторы из таблицы IDT были доступны всем задачам. В противоположном случае возможно следующее неблагоприятное развитие событий. Пусть элемент 13 таблицы (вектор нарушения общей защиты) имеет значение DPL<3. Если в PL3-программе фиксируется превышение предела сегмента, процессор попытается вызвать обработчик нарушения общей защиты и обнаружит, что шлюз слишком привилегирован. Правило защиты по привилегиям не соблюдается, и процессор сформирует второе нарушение общей защиты. Вновь обнаружив, что шлюз слишком привилегирован, процессор приходит в состояние отключения и система перестает функционировать.

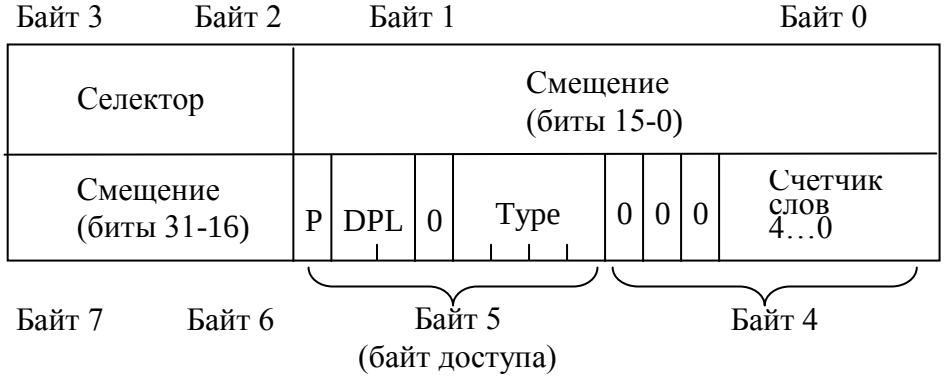


Рис. 5.9 Формат дескриптора типа шлюз

Type	Значение
5	шлюз задачи 80286 и Pentium
6	шлюз прерывания 80286
7	шлюз ловушки 80286
E	шлюз прерывания Pentium
F	шлюз ловушки Pentium

Так как обращение к шлюзу в таблице IDT является последним и «рефлексивным» действием процессора до передачи управления обработчику особого случая, необходимо правильно определить все дескрипторы в таблице IDT.

Отметим, что при обращении к шлюзам МП 80286 подпрограммы обслуживания работают с 16-разрядными операндами, а при обращении к шлюзам ix86 (x ≥ 3) используются 32-разрядные операнды.

Как и при использовании шлюза вызова, переход через шлюз ловушки или прерывания, приводящий к изменению уровня привилегий, вызывает и изменение стека. Процессор считывает селектор и смещение нового стека из полей указателя стека текущего сегмента TSS. Если, например, обработчик действует на уровне привилегий 0, то 48-битный указатель стека берется из полей SS0 и ESP0. После определения нового стека процессор включает в стек указатель старого стека SS:ESP, а затем содержимое регистров EFlags, CS и EIP.

Если при переходе через шлюз ловушки или прерывания уровень привилегий не изменяется, новый стек не создается и обработчик использует тот же стек текущей программы. В него заносится содержимое EFlags, CS и EIP.

Когда обработка особых случаев ведется отдельной задачей через шлюз задачи, уровень привилегий сегмента кода обработчика не играет роли. Процессор контролирует только уровень привилегий дескриптора шлюза задачи.

Отметим, что понятие задачи нами не изучается. Его описание можно найти в [4, 9, 10, 11].

Последовательность машинных циклов процессора IA-32 при обработке прерываний/исключений в реальном и защищенном режимах.

`Прерывания по входу INTR`. Когда процессор распознает активный сигнал (высокий уровень напряжения) на входе *внешнего маскируемого прерывания* INTR, и прерывания разрешены (IF = 1), он заканчивает выполнение текущей команды (а в некоторых случаях и следующей за ней; см. примечание выше), запрещает прерывания (сбрасывает флажок IF) и генерирует далее следующие машинные циклы:

1) Цикл или циклы (в зависимости от режима работы и соотношения адреса из ESP/SP и выровненного адреса) «Запись в стек» содержимого регистров EFlags/Flags, CS и EIP/IP (см. рис. 3.7, 3.11). Для ряда типов исключений в стек после EIP/IP заносится 32-разрядный код ошибки, значение которого помогает вызываемой процедуре обслуживания выявить причину ошибки.

2) Цикл «Подтверждение прерывания» (см. рис. 3.8, 3.12), в котором процессор вводит 8-разрядный номер прерывания *n* из контроллера прерываний.

Дальнейшие действия зависят от режима работы процессора.

В реальном режиме в этом же цикле процессор умножает номер *n* на 4 (двойной сдвиг влево, формула (5.1)), что дает линейный адрес вектора прерывания, после чего выполняет циклы

3) «Чтение памяти» (рис. 3.6, 3.10) для считывания по этому адресу вектора прерывания (ВП) и загрузки его в пару регистров CS:IP и

4) «Чтение стартовой команды» запрошенной процедуры.

В защищенном режиме в цикле «Подтверждение прерывания» процессор умножает номер *n* на 8 (тройной сдвиг влево) и складывает его с базовым адресом таблицы прерываний, взятым из регистра IDTR (формула (5.2)). Результат

сложения определяет линейный адрес дескриптора прерывания. Затем процессор генерирует

3) Цикл «Чтение памяти», в котором считывает по найденному адресу дескриптор, извлекает из него пусковой адрес процедуры (вектор прерывания), загружает его в пару регистров CS:EIP, и затем еще два цикла

4) «Чтение памяти», в котором на основании селектора из CS считывает дескриптор сегмента запрошенной процедуры из глобальной или локальной таблицы и вычисляет на его основе адрес перехода на процедуру путем сложения базы сегмента и смещения из EIP,

5) «Чтение стартовой команды» процедуры.

Прерывания по входу NMI и внутренние исключения. В случае запроса *внешнего немаскируемого прерывания* NMI и *внутренних исключений* выполняются все те же действия кроме цикла «Подтверждение прерывания», поскольку номер *n* поступает изнутри процессора.

При обработке чисто программных прерываний по командам int и bound действия аналогичны, но сначала производится цикл выборки команды из памяти, а потом уже приведенные выше циклы (но тоже без «Подтверждения прерывания», так как номер *n* задается самими командами).

Отметим, что большинство типов исключений обеспечивают сохранение в стеке адреса вызвавшей их команды (табл. 5.1). Это исключения типа отказов (типа 0, 5-7, 10-17) и отладочные исключения (тип 1). В результате имеется возможность повторного запуска (рестарта) программы после того, как процедура обработки устранит причины исключения. Например, в случае отсутствующего сегмента (P=0) выполнение программы может возобновиться после установки P=1. Эта возможность реализуется и после выполнения процедуры обслуживания исключения (ловушки) отладки. Другие исключения, например вызванные командами into, int3, int *n*, сохраняют в стеке адрес следующей команды. В результате после обработки этих исключений выполнение программы продолжается.

5.7. Приоритеты прерываний и исключений

Прерывания являются реакцией на внешние события, поэтому сигналы запросов по входам NMI и INTR проверяются и обрабатываются между выполнением команд программы. Немаскируемое прерывание NMI имеет более высокий приоритет, т.е. при одновременном выявлении запросов по обоим этим входам процедура его обслуживания выполняется первой. После ее завершения процессор IA-32 вызывает процедуру обработки маскируемого прерывания, если оно является разрешенным.

Запросы на обработку исключений генерируются внутренними блоками процессора IA-32, когда при выполнении команды возникают какие-либо проблемы. МП перезапускает команду, вызвавшую исключение, после того, как это исключение будет обработано соответствующей процедурой. Одна и та же команда может служить причиной нескольких исключений, однако при каждой

попытке выполнить эту команду будет возникать только одно из них. Процедура обработки каждого исключения исправляет вызвавшую его причину и осуществляет перезапуск. Таким образом, исключения обрабатываются последовательно до тех пор, пока не произойдет успешное завершение команды.

При дешифрации и выполнении каждой команды процессор IA-32 осуществляет следующую последовательность проверок возникновения прерываний/исключений:

1. Проверяются ловушки исключения 1 (отладки) только что завершенной команды (пошагового режима – через установленный флаг ловушки, прерывания по данным – установкой бита в регистрах отладки).
2. Проверяются отказы исключения 1 следующей команды (установка бита прерывания выполнения команды в регистрах отладки).
3. Проверяются внешние сигналы на входах NMI и INTR.
4. Проверяются отказы сегментации, мешающие выборке следующей команды (исключения 11 и 13).
5. Проверяются отказы страничной организации, мешающие выборке следующей команды (исключение 14).
6. Проверяются отказы дешифрации следующей команды (исключение 6 при недопустимом коде операции; исключение 6, если в реальном режиме или виртуальном режиме 8086 делает попытка выполнить команду, относящуюся исключительно к защищенному режиму; исключение 13, если команда длиннее 15 байтов или имеет место нарушение уровня привилегий в защищенном режиме).
7. Если идентифицировать код команды wait, проверяются условия TS = 1 и MP = 1 (исключение 7, если оба бита равны 1) (TS – бит переключения задачи в CR0; MP – бит управления сопроцессором в CR0, используется вместе с битом TS для генерации исключения 7 при идентификации кода команды wait. Если TS = 1 и MP = 1, код команды wait вызывает прерывание).

   TS и MP управляют режимами FPU.

   TS – Task Switched – задача переключена. Этот бит устанавливается процессором при каждом переключении задачи.

   Если TS установлен (т.е. после переключения задачи) и выполняется команда FPU, то возникает прерывание 7 – FPU-блок отсутствует. Обычно процедура обработки этого прерывания сохраняет состояние сопроцессора, относящееся к предыдущей задаче, загружает в сопроцессор состояние текущей задачи, очищает бит TS перед возвратом к команде IRET.

   MP – Math Present – сопроцессор присутствует. В процессорах i286 и i386 бит MP управлял функцией команды ожидания wait, которая применялась для синхронизации действий с сопроцессором. При выполнении программы в процессорах начиная с i486 этот бит должен быть установлен в 1. Если MP = 1 и TS = 1, код команды wait вызывает прерывание 7.
8. Если идентифицирован код команды с плавающей точкой, проверяются условия EM = 1 и TS = 1 (исключение 7, если один из битов равен 1).

9. Если идентифицирован код команды с плавающей точкой, проверяется флаг ошибки FPU (исключение 16, если этот сигнал активен).
10. При каждом обращении к памяти сначала проверяются ошибки сегментации (исключения 11, 12, 13), затем ошибки страничной организации (исключение 14). Эта последовательность определяет приоритеты обслуживания при одновременном возникновении нескольких прерываний/исключений различных типов.

5.8. Прерывание – эффективный механизм функционирования ЭВМ

Возможность прерываний программ – важное свойство ЭВМ, позволяющее эффективно использовать производительность процессора при наличии нескольких, протекающих параллельно во времени процессов, требующих управления со стороны процессора в произвольные моменты времени. В первую очередь это относится к организации параллельной во времени работы процессора и периферийных устройств машины, а также к использованию ЭВМ для управления техническими объектами и технологическими процессами в реальном масштабе времени (в темпе с процессом). Механизм прерываний обеспечивает немедленную реакцию ЭВМ на события, происходящие как во внешней среде (например, в управляемой компьютером технологической установке), так и внутри машины (при сбое питания, делении на нуль и др. событиях). Этот механизм превращает компьютер из машины для обычных вычислений в эффективное средство управления и обслуживания объектов самого широкого спектра – от современных технологий до бытовых приборов и компьютерных игр.

6. ЗАЩИТА ПАМЯТИ

Сейчас компьютеры широко применяются в банковском деле, управляют технологическим и медицинским оборудованием, ведут базы данных и знаний и т.п. По мере роста нашей зависимости от компьютеров требуется всё более надежные, живучие и защищенные системы, которые могут выполнять несколько задач одновременно (в режиме разделения времени), но эти задачи должны быть изолированы друг от друга так, чтобы ошибки в одной из них не влияли на другие и на операционную систему в целом. Именно на реализацию таких методов рассчитан Р-режим процессоров архитектуры IA-32. Когда программа осуществляет неожиданное обращение к неверному пространству памяти, механизм защиты блокирует обращение и сообщает о его возникновении.

6.1. Методы защиты

Для облегчения обнаружения и идентификации ошибок МП IA-32 сконструированы со **встроенными методами защиты** [4, 9, 19]. Основной единицей защиты является сегмент, и сегментные дескрипторы хранят необходимые параметры защиты. На рис. 6.1 показаны имеющие отношение к защите поля сегментных дескрипторы. К ним относятся: поле границы сегмента (его размера в байтах или страницах минус 1) и большинство полей байта доступа.

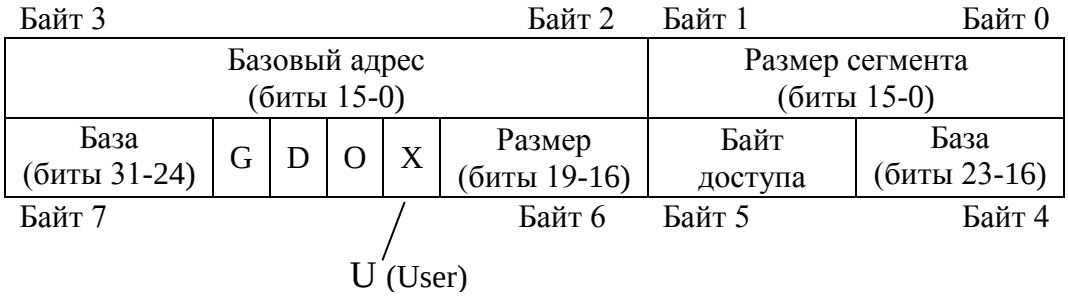
Перед началом каждого цикла обмена с памятью аппаратным путем проверяется, удовлетворяет ли ссылка в память критерию защиты. Любое нарушение предотвращает начало цикла обмена и приводит к прерыванию – исключению (n=10-13; чаще всего n=13). Так как эти проверки и трансляции адреса осуществляются одновременно друг с другом, то потери производительности малы или отсутствуют.

Параметры защиты запоминаются в сегментных дескрипторах во время их создания системными программами в дескрипторных таблицах. Когда программа загружает селектор в сегментный регистр, то МП IA-32 загружает весь дескриптор, содержащий не только базовый адрес, но также и информацию о защите: биты границы, типа и уровня привилегий. Каждый сегментный регистр хранит в своей невидимой части весь дескриптор сегмента, содержащий всю необходимую информацию о его защите.

Защита имеет несколько аспектов, которые можно условно разбить на две группы:

1) защита одной задачи от другой («прикладная» защита, защита приложения);
2) защита операционной системы от прикладных задач («системная» защита, защита системы).

а)



б)

				Тип (Type)			
7	6	5	4	3	2	1	0
P	DPL		S=1	E=1	C	R	A

в)

7	6	5	4	3	2	1	0
P	DPL		S=1	E=0	ED	W	A

г)

7	6	5	4	3			0
P	DPL		S=0	X	X	X	X

Рис. 6.1. Общий формат дескриптора (а) и форматы байта доступа сегмента:
б) – для кодового сегмента;
в) – для сегмента данных;
г) – для системного дескриптора

S – System
DPL – Descriptor Privilege Level
P и A – Presence и Access (для виртуальной памяти)
E – Executive
R – Read
W – Write
C – Conforming (подчинение)
ED – Expand Down
XXXX – поле типа

В первую группу входят проверки правил операций с уже открытыми сегментами:

- проверка границы сегмента (при этом используется поле границы, предела сегмента Limit);
- проверка типа операций с сегментами (используется поле типа Type в сочетании битом S–System).

Во вторую группу аспектов защиты входит проверка права доступа программы к новым сегментам, или защита по привилегиям (участвуют: поле DPL – в дескрипторе, поле CPL – в CS и поле IOPL – в Flags). Эта защита памяти (по привилегиям) имеет несколько аспектов:

- ограничение доступа к новым сегментам данных;
- ограничение точек входа в процедуру, находящуюся в новом кодовом сегменте;
- ограничение набора команд;
- проверку достоверности указателя (селектора) и дескриптора;
- защиту страниц и каталога.

Защита приложений рассматривается в п.6.2, защита системы (по привилегиям) в п.6.3.

Рассмотрим поля дескриптора, имеющие отношение к защите памяти. К ним относятся поля предела (размера) и байта доступа (5 байта) (см. рис. 6.1). Предел уже рассмотрен в п.2.4. Осталось познакомиться с байтом 5.

Байт 5 дескриптора определяет **права доступа** к выбираемому сегменту (AR – Access Rights). Кодирование байта доступа AR показано на рис.6.1. Его формат зависит от того, какого типа сегмент выбирается: кодовый (исполняемый, содержащий команды; рис. *б*), данновый (неисполняемый, содержащий операнды; рис. *в*) или системный (в этом случае дескриптор описывает системный объект – сегмент, содержащий локальную дескрипторную таблицу LDT, сегмент состояния задачи TSS или шлюз; рис. *г*).

Системный бит S определяет роль дескриптора в системе. При S=1 дескриптор обеспечивает обращение к сегменту программ (кодов) или данных (включая стек). Дескрипторы, имеющие S=0, служат для обращения к таблицам LDT, сегментам состояния задач TSS или являются шлюзами для входа в другие задачи или программы, включая программы обслуживания прерываний. Такие дескрипторы называются *системными*.

Рассмотрим сначала **обычные дескрипторы** (S=1) и общие поля тех и других (S=0 или S=1), а потом – системные дескрипторы (S=0).

Каждый байт доступа (как обычных, так и системных дескрипторов) имеет двухбитное поле уровня привилегий дескриптора DPL (Descriptor Privilege Level). Оно позволяет процессору предотвратить доступ обычных пользовательских программ к сегментам операционной системы. Поле DPL подробно рассматривается в п.6.3, посвященном кольцам защиты (системной защите).

Два бита присутствия P (Presence) и обращения A (Access) помогают операционной системе реализовывать виртуальную память. В системе виртуальной памяти при выполнении программы в памяти одновременно находятся некоторые ее сегменты, а остальные хранятся на диске. Когда программа пытается обратиться к сегменту, которого в памяти нет, возникает особый случай (прерывание 11) и процедура его обработки считывает («вталкивает») в память нужный сегмент. Если места в памяти для сегмента недостаточно, процедура обработки должна вначале освободить требуемое место, пересылая («выталкивая») один или несколько сегментов из памяти на диск. Весь этот процесс называется

свопингом («подкачкой») (от англ. swop=swap – замена). После передачи нужного сегмента в память процедура обработки особого случая осуществляет возврат в программу, и она продолжается, не подозревая о прерывании. Следовательно, операционная система создает иллюзию того, что все сегменты программы все время находятся в памяти.

Бит присутствия P показывает, находится ли сегмент в памяти или на диске. Если в дескрипторе P=0 (сегмент отсутствует), то любая попытка использовать этот дескриптор (например, загрузкой селектора в сегментный регистр) вызывает особый случай *отсутствия*. Операционная система виртуальной памяти отмечает все сегменты, находящиеся в памяти, установкой P=1, а сегменты, которых в памяти нет (они на диске), установкой P=0. Следовательно, особый случай «неприсутствия» (отсутствия) возникает, когда программа обращается к сегменту, которого нет в памяти. Процедура обработки этого случая может считать нужный сегмент с диска, установить поле базы дескриптора на адресацию базы сегмента и сделать P=1. После возврата в программу она продолжается так, как будто ничего не произошло.

Коротко: бит присутствия P определяет наличие сегмента в памяти: при P=1 сегмент присутствует в памяти, при P=0 – отсутствует (находится на диске). Если P=0 (сегмент отсутствует), то при обращении к нему (при загрузке его селектора в сегментный регистр), МП переходит к обработке особого случай (исключение 11).

Бит обращения A. В отличие от бита P, который сообщает операционной системе, когда передавать сегмент в память, бит обращения A помогает ей выбрать сегмент, который передается на диск, когда в памяти нет достаточно места для нового сегмента. При выборе сегмента операционная система попытается избежать передачи на диск тех сегментов, которые потребуются в ближайшем будущем. Поскольку она не может точно предсказать будущее поведение программы, операционная система должна сделать обоснованное предположение. Оно сводится к тому, что не используемый долгое время сегмент, вероятно, не потребуется и в будущем. Поэтому приемлемая стратегия при необходимости освобождения пространства памяти заключается в том, чтобы обменять (swap) сегмент, к которому не было обращения самое продолжительное время. Эта стратегия называется LRU (Least Recent Used) – заменяется дольше всего не использовавшийся сегмент.

Бит A помогает операционной системе ассоциировать с каждым сегментом приблизительное время его последнего использования. Для этого биты A всех сегментов первоначально устанавливаются в 0. При загрузке селектора в сегментный регистр бит A автоматически устанавливается процессором в 1 (в дескрипторной таблице). Через регулярные интервалы времени ОС сканирует все дескрипторы и если A=1 (т.е. к ним были обращения после предыдущего сканирования), то инкрементирует время последнего использования таких сегментов, и тут же сбрасывает A в ноль: A=0. В дальнейшем, когда необходимо выбрать сегмент для удаления из памяти при недостатке места (передачи на

диск), им будет сегмент с наибольшим временем последнего использования (т.е. ОС попросту ведет подсчет времени использования сегментов).

Бит исполняемости E (Executive). Дескрипторы программ и данных (S=1; см. рис.6.1) отличаются значениями бита исполняемости E: E=1 для программных сегментов и E=0 для данновых сегментов.

Бит считывания R вводится для программного сегмента. Если R=1, то наряду с обращением к такому сегменту для выполнения (через регистр CS) разрешаются и обращения для считывания (при помощи команд). Если же R=0, любая попытка считать из сегмента вызывает особый случай защиты (прерывание типа 13). Любая попытка записать в исполняемый сегмент тоже приводит к особому случаю типа 13.

Бит разрешения записи W имеется для сегментов данных. Если W=1, то помимо считывания из сегмента данных разрешена и запись в него. Если же W=0, то любая попытка записать в сегмент вызывает прерывание 13 (особый случай защиты). Кроме того, любая попытка выполнить сегмент данных также приводит к указанному особому случаю защиты.

Запись в сегмент программ, как мы сказали, всегда запрещена. Если возникает необходимость внести изменения в этот сегмент, то можно создать сегмент данных с разрешением записи (W=1), занимающий то же адресное пространство, что и модифицируемый сегмент программ. После внесения изменений в созданный сегмент данных можно обратиться к нему как к сегменту программ (запустить на исполнение).

Бит подчинения C (Conforming) (Conform – подчиняться, сообразовываться) определяет дополнительные правила обращения, которые обеспечивают защиту программных сегментов (п.6.4).

Бит расширения вниз ED (Expand Down) различает сегменты стека (ED=1) и обычные сегменты данных (ED=0) и интерпретирует поле предела. В сегментах данных (ED=0 – расширение «вверх», в сторону старших адресов) адресуемый диапазон простирается от базового адреса до базового адреса плюс предел, т.е. смещения варьируются от 0 до предела. Смещения, которые больше предела, вызывают особый случай защиты (прерывание 13).

В сегментах стека с ED=1 (расширение «вниз», в сторону младших адресов) ситуация противоположна в том смысле, что сегмент (предел) располагает в направлении уменьшения адресов от базы до границы сегмента. Здесь все смещения должны быть строго больше предела, а если смещение меньше или равно пределу, фиксируется особый случай нарушения стека (исключение 12).

Отметим, что термин «расширение вниз», т.е. к меньшим адресам, относится к такой графической иллюстрации адресного пространства, в которой адрес 0 находится внизу рисунка, а максимальный адрес вверху.

Рассмотрим теперь **байт доступа для системного дескриптора** (S=0). Поля P и DPL имеют тот же смысл, что и для обычных дескрипторов.

Поле типа системного дескриптора XXXX содержит число от 0 до F, определяющее тип системного дескриптора (табл. 6.1)

Таблица 6.1 – Типы системных дескрипторов

Тип	Определение	Тип	Определение
0	Зарезервирован	8	Зарезервирован
1	Доступный TSS процессора 80286	9	Доступный TSS Pentium
2	Таблица LDT	A	Зарезервирован
3	Занятый TSS 80286	B	Занятый TSS Pentium
4	Шлюз вызова 80286	C	Шлюз вызова Pentium
5	Шлюз задачи (80286/ Pentium)	D	Зарезервирован
6	Шлюз прерывания 80286	E	Шлюз прерывания Pentium
7	Шлюз ловушки 80286	F	Шлюз ловушки Pentium

Можно выделить три основных класса системных дескрипторов:

Дескрипторы таблиц LDT (тип 2) адресуют таблицы LDT, используемые при выполнении текущей задачи. Хранятся в таблице GDT и загружаются в теневое расширение LDTR при открывании LDT.

Дескрипторы задачных сегментов TSS (типы 1, 3, 9, B) используются при переключении задач в многозадачном режиме.

Дескрипторы типа шлюзов, или просто «шлюзы» (типы 4-7, C, E, F) используются для обращения к программам и задачам, а также при обработке прерываний. Назначение шлюзов рассматривается в п.6.4.

6.2. Защита приложений (прикладная защита).

Проверка границы (предела). Во всех типах сегментов, за исключением сегментов данных с ED=1, поле границы содержит длину сегмента (в байтах при G=0 и в страницах при G=1) минус 1, т.е. максимально допустимый внутрисегментный адрес. Если бит G=0, границей является значение 20–битного поля границы дескриптора. Граница в этом случае может изменяться от 0 до FFFFF (20^{20}–1), т.е. сегмент может содержать от 1 байта до 1 Мбайт. Если бит G=1, то МП присоединяет 12 младших единичных битов к значению поля границы (умножение на 2^{12}=4К минус 1). В этом случае реальная граница может быть принята от FFF (2^{12}-1, или 4Кб в сегменте) до FFFFFFFF (2^{32}-1, или 4Гбайт в сегменте). При обращениях к памяти процессор контролирует, чтобы используемые смещения были равны или меньше предела; для контроля применяется 32-битовые беззнаковые сравнения. Контроль осуществляется параллельно с преобразованием адреса, поэтому дополнительное время на него не требуется, Если смещение оказывается больше предела, фиксируется особое случай (n = 13). Отметим, что в случае многобайтных операндов указанному условию должно удовлетворять смещение самого старшего байта операнда.

Сегменты данных с ED = 1 (т. е. сегменты с расширением «вниз») предназначены для стеков. В процессоре Pentium (i286) стеки начинаются со смещения FFFF = 64k – 1 (а не FFFFF!) (при D = 0) и FFFFFFFF = 4Г –1 (при D = 1) и «растут вниз» к меньшим адресам до границы (предела) (которая определяется с учетом бита гранулярности G). Поэтому ситуация с проверкой оказывается противоположной. Здесь все смещения должны быть строго

больше предела, а если смещение меньше или равно пределу, фиксируется особый случай нарушения стека.

Заметим, что в зарубежной литературе термин «расширение вниз/вверх» относится к графическому изображению адресного пространства, в котором адрес 0 находится внизу, а максимальный адрес вверху. Мы изображаем его наоборот: адрес 0 – вверху, а максимальный адрес – внизу, как показано на рис.6.2. Поэтому для нас «расширение вверх», означая по-прежнему расширение в сторону старших адресов, на рисунке соответствует расширению к низу изображения памяти; и наоборот – при «расширении вниз», как показывают стрелки на рис.6.2.

На этом рисунке 6.2.1 изображены сегменты стека и данных, которые оба имеют предел 1FFF (G = 0). Для сегмента данных допустимыми будут все смещения, меньше или равные 1FFF. Первый адресуемый байт находится по смещению 0, а последний по смещению 1FFF; размер сегмента составляет 2000h или 8 Кбайт. Для сегмента стека, наоборот, допустимы все смещения, которые больше предела; первый адресуемый байт находится по смещению 2000h, а последний по смещению FFFF. Размер стека составляет E000 или 56k байт.

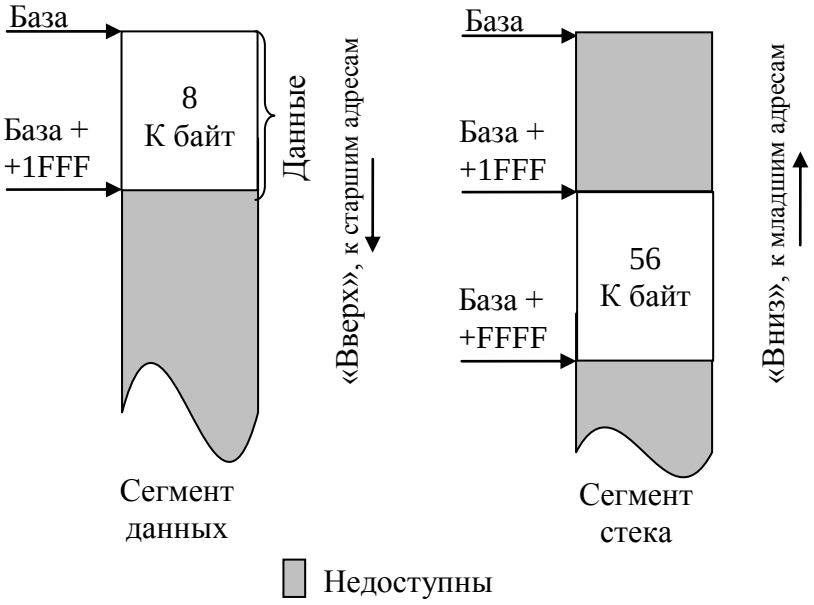


Рис. 6.2. Интерпретация поля предела
для сегментов данных и стека

В сегменте данных минимальный адрес задается полем базового адреса дескриптора, а максимальный – полем предела. Для сегментов стека минимальный адрес определяется полем предела, а чему же равен максимальный адрес? Здесь имеется два выбора. Если бит D = 0, то максимальный адрес равен базовому адресу плюс константа FFFF (64k-1). Когда же бит D = 1, максимальный адрес равен базовому адресу плюс FFFFFFFF. (4Г – 1). Так как все смещения должны быть больше предела,

максимальный размер стека получится, когда предел равен 0. Сегмент стека имеет минимальный размер, когда предел равен либо FFFF (D = 0) либо FFFFFFFF (D = 1), что соответствует пустому сегменту с нулевым размером.

Напомним, что, если в дескрипторе сегмента стека бит D=1, то во всех стековых операциях участвует 32-битный указатель стека ESP и все включения и извлечения оперируют 32-битными двойными словами. Если же бит D=0, то для стековых операций привлекается указатель стека SP, и в них участвуют 16-битные слова. Следовательно, для использования 32-битного стека, что рекомендуется при разработке новых прикладных программ, необходимо задать в дескрипторе сегмента стека D=1, а это приводит к максимальному адресу стека 4Г-1. Более того, процессор Intel 80x86 требует, чтобы при D=1 бит гранулярности G также находился в состоянии 1. Следовательно, необходимо не только определить 32-битный стековый сегмент с максимальным адресом 4Г-1, но и установить предел в единицах 4Кбайт.

В общем случае для использования стека необязательно определять сегмент стека. Вполне допустимо заставить указатель стека ESP адресовать середину обычного сегмента данных (разумеется, с разрешенной операцией записи). Но т.к. указатель стека обычно инициализируется на наибольший адрес в сегменте, а затем по мере включений в стек содержание указателя стека уменьшается, при нехватке стекового пространства возникают трудности с увеличением размера стека. Для расширения обычного сегмента данных достаточно увеличить значение в поле предела дескриптора и пользоваться большими адресами. Однако стек «растет» в направлении уменьшения адресов, поэтому при расширении сегмента данных придется сдвигать в область больших адресов весь стековый кадр. Если же с самого начала определить сегмент как стек, то для его расширения достаточно уменьшить значение предела.

Отметим, что бит ED не влияет на то, каким образом поле базы дескриптора участвует в процессе преобразования адреса. Изменяется только контроль нахождения в пределах.

Контроль необходим для изолирования задач, т.к. без него задача может «выйти» из своих сегментов и исказить другую задачу. Такая ситуация может возникнуть, например, если программа индексирует элементы, находящиеся за концом массива. Разработчики процессоров могли бы обойтись без пределов, потребовав, чтобы длина всех сегментов была точно 64 Кбайт. Однако это привело бы к неэффективному использованию памяти и не позволило бы реализовать очень малые системы с большим числом небольших задач.

Особенно важен контроль пределов сегментов с расширением вниз, т.е. сегментов стеков. Операционная система может запустить задачу с небольшим стеком. Если задача превышает свой стек, возникает особый случай и процедура его обработки может распределить задаче большее стековое пространство. Т.о., память не расходуется впустую на стеки, размеры которых определились бы с запасом.

Проверка типа. Поле Type (тип) байта доступа различно для дескрипторов разного формата и определяет желаемый способ использования сегмента. Проверка типа может быть использована для обнаружения программных ошибок, которые представляют собой попытки использовать сегмент способом, неожиданным для программиста. Процессор проверяет информацию о типе в двух случаях:

1. Если команда явно или неявно обращается к сегменту. Некоторые сегменты могут использоваться командами только предопределёнными способами. Например:

- если бит разрешения записи W не установлен, то ни одна команда не может записывать в сегмент данных;
- до тех пор, пока не установлен бит разрешения чтения R, ни одна команда не может читать из исполнительного сегмента;
- ни одна команда ни при каких условиях не может записывать в исполнительный сегмент. (Чтобы записать в эту область памяти надо загрузить селектор из CS в обычный сегментный регистр, например в DS, т.е. объявить эту область как сегмент данных).

2. Если новый селектор загружается в сегментный регистр. Некоторые сегментные регистры могут содержать дескрипторы только определённых типов. Например:

- в регистр SS могут быть загружены селекторы сегментов данных только с разрешением записи;
- регистр CS может загружаться селектором только исполнительного сегмента;
- селекторы исполнительных сегментов с защитой от чтения не могут быть загружены в сегментные регистры данных.

6.3. Защита по привилегиям (системная защита). Кольца защиты.

В предыдущем разделе мы рассмотрели, как процессор защищает одну задачу от другой, а теперь обратимся к тому, как он помогает операционной системе защищать саму себя от прикладных задач. Предполагается, что процессор работает в виртуальном (P-) режиме.

Практически во всех компьютерах для целей защиты предусматриваются, как минимум, два режима работы – системный режим, называемый также режимом супервизора (Supervisor), и пользовательский режим (User). Основное различие между ними состоит в том, что программам, работающим в режиме супервизора (а ими обычно являются процедуры операционной системы), доступны все ресурсы системы. В пользовательском режиме программам запрещается выполнение некоторых команд, влияющих на системные ресурсы, К таким привилегированным командам относятся команды ввода-вывода, манипуляций системными регистрами, управления прерываниями и др. В старших процессорах IA-32 этот принцип расширен и применительно к сегментам. Для них введены и аппаратно поддерживаются четыре уровня

привилегий PL (Privilege Level). Однако в страничном преобразовании адреса применяется упомянутый выше простой двухуровневый механизм защиты пользователь/супервизор.

Защита требует наложения на обычные программы (не входящие в операционную систему) трех типов ограничений:

1) обычным программам запрещается выполнять некоторые команды.
2) обычным программам должны быть недоступны определенные сегменты данных, доступные операционной системе
3) должно быть невозможным получение привилегий операционной системы, кроме как входом в нее в разрешенной точке входа.

Уровни привилегий. Для защиты сегментов в процессоре IA-32 применяется схема, в которой действуют *четыре уровня привилегии* PL (Privilege Level) с номерами 0-3. Чем меньше номер уровня, тем более привилегирован этот уровень (в отличие от того, что вы ожидали). *Степень защиты* сегмента также имеет четыре уровня, которые схематически представляются в виде вложенных колец защиты (рис.6.3).

Рис. 6.3. Уровни привилегий и защиты процессора IA-32

Соответствующие уровни защищенности иллюстрируются на примере программных сегментов. Наименее защищенными (привилегированными) являются прикладные программы пользователя, для которых выделяется уровень с номером 3. Уровни с номерами 0, 1, 2, отводятся для системных программ (супервизора), которые можно разделить на три уровня в зависимости от требований к их защищенности. Наиболее защищенная часть – ядро операционной системы (ОС) имеет уровень 0. В ядро входит часть ОС, обеспечивающая инициализацию работы, управление доступом к памяти, защиту и ряд других жизненно важных функций, нарушение которых полностью выводит из строя МП систему. Основная часть ОС (утилиты) должна иметь уровень 1. К уровню 2 обычно относятся ряд служебных программ ОС, например драйверы внешних устройств, инструментальные системы (системы программирования), системы управления базами данных и др. Наконец прикладные программы находятся на уровне 3.

В соответствии с уровнями привилегий и защищенности устанавливаются следующие **правила выполнения команд и доступа к сегментам** программ и данных:

1. Некоторые команды запрещено выполнять программам, имеющим низкий уровень привилегий.
2. Сегмент данных, имеющий уровень защиты PL, может быть открыт (т.е. его селектор загружен в сегментный регистр DS, ES, FS, GS или SS) программой, имеющей такой же или более высокий уровень привилегий. Точно так же и данные могут быть считаны (записаны) программой только из одинаковых с ней или менее привилегированных сегментов (в такие сегменты). *Иначе, программам запрещается «движение» к данным внутрь колец защиты*.
3. Программный сегмент (процедура), имеющий уровень защиты PL, может быть вызван (его селектор загружен в регистр CS) программой, имеющей такой же уровень привилегий. Программа может вызвать и более привилегированную программу (процедуру), но только через специальные точки входа – шлюзовые дескрипторы, или шлюзы, (или из специальных – подчиненных – сегментов). *Коротко, передачи управления ограничиваются занимаемым кольцом защиты*, и это правило имеет исключение, но *только для вызовов через шлюзы внутрь колец защит*ы (и возвратов наружу колец).

Уровни привилегий находятся в трех различных местах:

1. Каждый дескриптор имеет *уровень привилегий дескриптора* DPL. Этот уровень характеризует степень защиты сегмента. DPL определяет право программы выбрать этот дескриптор. DPL определяется битами 6 и 5 в байте доступа дескриптора. (Для записываемых или считываемых сегментов дескриптор DPL можно считать «классом» сегмента, а для исполняемых сегментов DPL характеризует «благонадежность» процедур в этом сегменте.)
2. Процессор автоматически поддерживает текущий уровень привилегий CPL. (Current PL) – уровень, на котором в текущий момент выполняется программа – «уровень благонадежности текущей программы». Процессор хранит значение CPL в двух младших битах регистра CS. Определить этот уровень бита можно, прочитав эти два младших бита. Уровень CPL равен содержимому поля DPL сегмента, который адресуется регистром CS.
3. Каждый подключаемый селектор, т.е. новый селектор, заносимый в селекторный регистр, содержит запрашиваемый уровень привилегий RPL (Request PL). Этот уровень может только понизить текущий уровень (увеличить значение PL в селекторном регистре по сравнению с CPL) на время доступа к данным или выполнения программы (см. далее). Заметим, что уменьшить CPL при помощи поля RPL нельзя. Но если RPL < CPL, то никакая защита не включается, а просто CPL не изменяется.

Напомним, что меньшее значение DPL, CPL, RPL соответствует более высокому уровню привилегий (защиты). Наиболее защищенный сегмент имеет

значение DPL = 0, наименее защищенный – DPL = 3. Для наименее привилегированных программ пользователя CPL = 3, наиболее привилегированные программы ядра ОС имеют CPL = 0.

Защищенные команды.

Обычным программам запрещается выполнять некоторые команды (системные команды). Подобные команды можно поделить на

– привилегированные и

– чувствительные (табл.6.2)

Привилегированные команды выполняются только на уровне PL = 0. (Здесь PL = CPL – текущий уровень привилегий). Если процессор обнаружит эту команду, когда CPL > 0, то он вызовет исключение по общей защите (*n* = 13). Этими командами являются:

clts – сброс флажка переключения;

hlt – останов;

lgdt – загрузка GDTR;

lidt – загрузка IDTR;

lldt – загрузка LDTR;

ltr – загрузка TR;

lmsw – загрузка MSW (слова состояния машины);

mov to/from CRn – передача с управляющими регистрами CRn;

mov to/from DRn – передача с отладочными регистрами DRn;

mov to/from TRn – передача с тестовыми регистрами TRn.

Таблица 6.2 – Защищенные команды

Уровни команд	Мнемоники команд
Привилегированные: команды, разрешенные только на уровне 0	lgdt, lidt, lldt, ltr, lmsw, clts, hlt, mov to/from CRn, DRn, TRn
Чувствительные: команды, разрешенные на уровне IOPL и выше	in, ins, out, outs, sti, cli, lock
Чувствительные: команды, модифицируемые в соответствии с уровнем	iret, popf

Чувствительные команды – это команды ввода in и вывода out, ввода строки ins и вывода строки outs, а также команды iret и popf, модифицируемые в соответствии с уровнем привилегии. На команды ввода/вывода налагаются ограничения для того, чтобы защищать функционирование адаптеров ПУ. Они выполняются с учетом CPL выполняемой программы и поля IOPL в регистре EFlags [8]. CPL сравнивается с IOPL. Ввод-вывод производится только при выполнении условия

$CPL \le IOPL$

Нарушение этого условия вызывает прерывание 13, за исключением случаев, когда при решении задачи доступ к порту разрешается специальной битовой картой ввода-вывода (она находится в сегменте задачи).

Флажки содержат важную информацию. Один из них мы уже назвали – это IOPL. Еще одним столь же важным элементом является флажок разрешения прерывания IF. Хотя обычным пользовательским программам нельзя изменять эти поля, команды загрузки флажков имеют и допустимые применения и не должны запрещаться для менее привилегированных колец. Поэтому, если команда загрузки флажков (а это команды iret и popf) выполняется в кольце, отличающемся от нулевого, поле IOPL остается неизменным; если такая команда выполняется в кольце, менее привилегированном, чем определяется IOPL, флажок IF тоже не изменяется. В любой из этих ситуаций особый случай не возникает.

Обращение к сегментам программ и данных в защищенном режиме производится с учетом описанной выше системы привилегий. При этом правила обращения зависят от типа сегмента – данных, стека, кода.

Защита сегментов данных.

Обращение к новым сегментам данных производится с помощью селекторов, загружаемых в регистры DS, ES, FS, GS. При обращении анализируются значения RPL селектора и CPL текущей программы. Эффективный уровень привилегий для запроса данных в этом случае определяется как максимальный из значений RPL и CPL: EPL = max (RPL, CPL). Обращение к запрашиваемому сегменту данных разрешается, если эффективный уровень меньше или равен уровню сегмента:

max (RPL, CPL) ≤ DPL

Нарушение этого правила вызывает прерывание 13 – нарушение защиты.

Переход к новому стековому сегменту происходит по более ужесточенному правилу: при загрузке селектора в стековый сегментный регистр SS процессор несколько ужесточает правило защиты: загрузка разрешается , если только значение CPL (RPL) равно значению DPL стекового сегмента:

RPL = CPL = DPL

Другими словами, не разрешается использовать стек даже с меньшими привилегиями. Кроме того, загрузка селектора в регистр SS производится , если только выбираемый сегмент допускает операции считывания и записи (бит W (бит 1) в байте доступа должен иметь значение W = 1), а также присутствует в памяти (бит P присутствия (бит 7) в байте доступа должен находится в состоянии 1). Нарушение этих правил (использование селекторов и дескрипторов, у которых RPL ≠ DPL и CPL ≠ DPL; обращение к сегменту, для которого W = 0) приводит к прерыванию 13. При обращении к отсутствующему сегменту (P = 0) возникает прерывание типа 11.

Защита кодовых сегментов. Общее правило таково: процессор запрещает передачу управления кодовому сегменту, находящемуся на другом уровне привилегий, но в определенных случаях разрешает вызов подпрограмм из сегментов более высокого уровня. Иначе, значение CPL нельзя так просто изменить. Это самый важный и самый сложный момент механизма защиты по

привилегиям, который, в дополнение ко всему, имеет несколько исключений. Ограничивая передачу управления пределами одного кольца защиты, процессор предотвращает произвольное изменение уровня привилегий. Если бы значение CPL можно было легко изменить, все остальные средства защиты оказались бы бессмысленны.

Передачи управления в другой сегмент осуществляют межсегментные (типа far) команды безусловного перехода jmp, вызова процедур call и возврата из процедур ret, вызова прерываний int и возврата из прерываний iret. Здесь мы рассмотрим обычные команды, а в п.6.4 – команды прерываний.

Адрес передачи управления задается с помощью 48-битного указателя селектор:смещение, который содержится либо в самой команде (прямая передача управления) либо берется из регистра или памяти (косвенная передача управления). При выполнении таких команд изменяются регистры CS и EIP, поэтому с точки зрения процессора контроль межсегментной передачи управления заключается в достоверности селектора, загружаемого в регистр CS.

При загрузке селектора в регистр CS процессор предпринимает следующие проверки. Прежде всего, проверяется, что целевой дескриптор (дескриптор сегмента, которому передается управление) определяет кодовый сегмент, т.е. имеет атрибут исполняемого сегмента: E = 1. Разрешения на считывание из этого сегмента не требуется. Затем процессор контролирует, что значение CPL равно значению DPL целевого дескриптора. Если при этом RPL > CPL, возникает особый случай защиты. Поэтому значение поля RPL должно быть меньше или равно текущему значению CPL: RPL ≤ CPL. В сомнительных ситуациях следует устанавливать RPL = 0 и тогда это поле не действует. Наконец, целевой кодовый сегмент должен быть отмечен битом присутствия (P = 1 в дескрипторе), а новое значение для EIP должно находиться в пределах нового кодового сегмента. Только при условии прохождения всех этих проверок процессор продолжает выполнение по указанному адресу передачи управления. Когда какая-либо проверка дает отрицательный результат, формируется нарушение общей защиты (прерывание 13) или нарушение неприсутствия (прерывание 11).

Определение текущего уровня привилегий. Иногда программе удобно или необходимо определить тот уровень привилегий (т.е. значение CPL), на котором она выполняется. Для этого можно прочитать два младших бита регистра CS, воспользовавшись, например, такими командами:

Первый способ

```
mov AX, CS      ; Передать селектор в регистр AX
and AX,03h      ; Выделить биты поля RPL
```

Второй способ

```
push CS         ; Включить код селектора в стек
pop AX          ; Передать селектор в AX
and 03h         ; Выделить биты поля RPL
```

Следует отметить, что МП имеет специальную команду ARPL r/m, reg , используемую для коррекции RPL селектора. При этом селектор загружается в регистр или ячейку памяти, адресуемую байтом ModRM команды ARPL. Селектор служит первым операндом команды. В качестве второго операнда в регистр, адресуемый байтом ModRM, загружается текущее значение CS:

```
arpl r/m, reg  ; В r/m должен быть селектор, в reg – CS
```

Действие команды:

```
; сравниваются RPL и CPL
; и в поле RPL заносится максимальное из значений RPL и CPL
```

При выполнении команды сравниваются значения двух младших битов операндов, которые содержат RPL и CPL. Поле RPL селектора (первого операнда) принимает максимальное из значений RPL и CPL. Если значение RPL при этом изменилось (возросло до величины CPL), то устанавливается признак ZF = 1.

6.4. Передача управления между уровнями привилегий

Напомним основные моменты защиты по привилегиям, рассмотренные в предыдущих параграфах.

- процессор Intel 80x86 аппаратно поддерживает работу на любом из четырех уровней привилегий;
- текущий уровень привилегий CPL определяется выполняемым кодом, который, в свою очередь, задается полем DPL дескриптора кодового сегмента;
- процессор ограничивает выполнение определенных команд для обычных программ;
- процессор запрещает доступ к более защищенным сегментам данных;
- процессор жестко ограничивает передачи управления пределами текущего уровня привилегий.

Теперь возникает вопрос о том, как же осуществить передачу управления между уровнями привилегий. На первый взгляд кажется, что ради поддержки защиты процессор должен запрещать менее привилегированным процедурам вызывать более привилегированные. Именно поэтому он предотвращает выполнение пользовательской программой тех операций, которые должны выполняться только операционной системой. Но при этом пользовательской программе запрещались бы и допустимые обслуживания, которые нужны ей от операционной системы. Об объемах услуг операционной системы свидетельствует, например, тот факт, что операционная система MS DOS через прерывание int 21h выполняет 150 разнообразных функций для пользователя. Поэтому процессор разрешает менее привилегированной процедуре вызывать более привилегированную, но ограничивает доступ разрешенными точками входа или (другой способ) так называемыми подчиненными сегментами. Следовательно, пользовательская программа получает от ОС только те

обслуживания, которые системе разрешено оказывать пользовательской программе.

Здесь мы рассмотрим, как осуществляется передача управления между уровнями привилегий. Применяются два способа передачи управления между уровнями привилегий. Первый (простой) заключается в применении *подчиненных кодовых сегментов*, а второй (сложный) – в использовании специальных дескрипторов, называемых *шлюзами вызова*.

Подчиненные кодовые сегменты (согласованные, conforming) (от англ. conform – согласоваться, подчиняться, соответствовать). Представим себе, что в системе имеется процедура (т.е. кодовый сегмент) для преобразования двоичных целых чисел в символьный код ASCII, которую используют программы на каждом уровне привилегий. Так как вызовы менее привилегированных процедур процессор запрещает, процедура должна находиться в кольце защиты 0, чтобы ее можно было вызывать из этого кольца. Но, в общем случае, процедура преобразования требует достаточно привилегий только для доступа к своим аргументам и возвращению результата, т.е. только такие же привилегии, как и вызывающая ее программа. Именно для таких ситуаций в процессоре IA-32 предусмотрены подчиненные (точнее, подчиняемые), или согласованные сегменты кода.

Кодовый сегмент определяется как подчиненный установкой бита 2 (бит C - Conforming) в байте доступа AR дескриптора сегмента – типы сегментов 6 и 7. В таких сегментах может быть разрешена или запрещена операция считывания. Если бит C установлен в 1, то обычные правила относительно CPL и DPL не действуют, и вместо них вводятся другие.

Процедуры в подчиненном сегменте можно вызывать или переходить к ним прямо из любой программы с таким же или меньшим уровнем привилегий, т.е. когда

CPL ≥ DPL подчиненного сегмента.

При этом, когда производится передача управления подчиненному сегменту, поле CPL не изменяется; оно остается на уровне вызывающей программы. Другими словами, подчиненные кодовые сегменты никогда не требуют больше привилегий, чем код, передавший им управление. Если, например, PL3-программа передает управление подчиненному кодовому сегменту, то он работает с CPL=3; если же этот сегмент вызывает PL0-программа, то он выполняется с CPL=0. Следовательно, подчиненный сегмент автоматически «перемещается» или «подчиняется» кольцу вызывающей программы. Это правило действует даже в том случае, если подчиненный сегмент вызывается через шлюз. Из-за такой «подвижности» уровня привилегий подчиненного кодового сегмента в нем не должны содержаться приведенные выше привилегированные команды, например, команды ввода-вывода. (По-другому. Когда управление передается подчиненному кодовому сегменту, биты поля RPL регистра CS не изменяются на поле DPL дескриптора нового кодового сегмента, как это обычно бывает, а сохраняют значение DPL последнего выполнявшегося неподчиненного кодового сегмента. Только в этой

единственной ситуации биты RPL регистра CS не соответствуют битам DPL дескриптора текущего выполняющегося кодового сегмента).

Наконец, передача управления подчиненному сегменту во внешнее кольцо защиты, т.е. когда CPL < DPL, никогда не разрешается.

Это ограничение необходимо, чтобы не нарушился весь механизм защиты при возврате из подчиненного сегмента, т.е. когда процедура в подчиненном сегменте закончилась и необходимо возвращать управление на уровень привилегий вызывающего кода. Если для перехода к подчиненному сегменту кода применялась команда jmp (или команда call), то возвращающая управление команда jmp (или команда ret) должна передавать управление более привилегированному и, главное, неподчиненному кодовому сегменту. Такие передачи управления запрещены, что, естественно, лежит в основе механизма защиты по привилегиям в процессоре IA-32.

Шлюзы вызовов. Менее привилегированная процедура может вызвать более привилегированную, обращаясь к ней через дескриптор шлюза вызова, определяющий доступную точку входа (рис.6.5). Этот способ обращения позволяет программам пользователя обращаться за обслуживанием к операционной системе. При этом допускаются обращения только к определенным процедурам, которые санкционируются путем введения в систему соответствующего шлюза. Тем самым исключается возможность несанкционированного обращения менее привилегированных процедур к более привилегированным, что позволяет защитить их от возможных искажений.

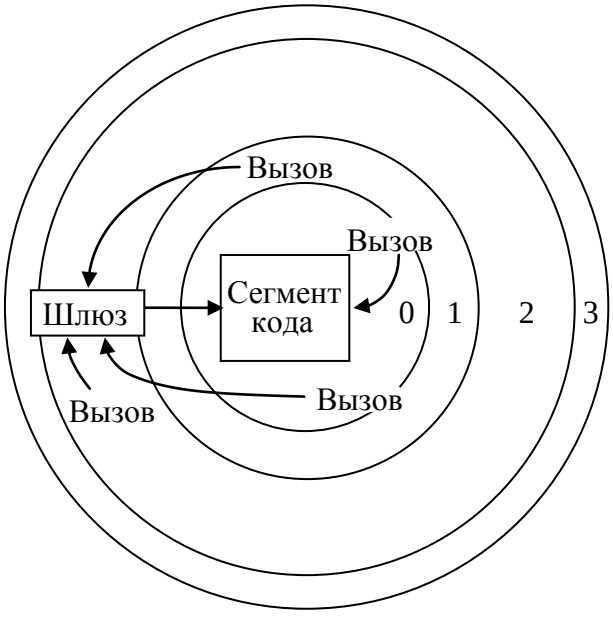


Рис. 6.5. Пример использования шлюза вызова

Для вызова более привилегированной программы команда call должна обратиться к хранящемуся в LDT дескриптору шлюза вызова, формат которого был показан на рис.6.1 и повторен здесь на рис.6.6.

В байтах 2 и 3 дескриптора шлюза содержится селектор вызываемого сегмента программы, а байты 0, 1, 6, 7 задают относительный адрес шлюза – точки входа в эту программу. Байт доступа (рис. б) содержит поля P и DPL, имеющие такое же значение, как и дескрипторах сегментов (раздел 2). Кроме того, байт доступа шлюза вызова имеет бит типа процессора T (бит 3), который имеет значение T = 0, если вызываемая программа написана для процессоров старше 80286, и T = 1, если программа написана для 80286. Этот бит определяет механизм формирования линейного адреса, который зависит от типа микропроцессора.

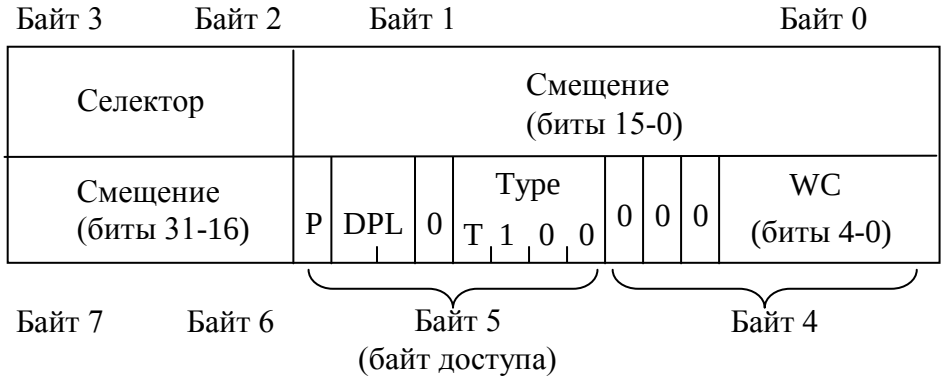


Рис. 6.6. Формат дескриптора типа шлюз

Пятибитовое поле WC в байте 4 указывает количество параметров, которые переносятся из стека текущей программы в стек новой программы. Параметры представляют собой 32-разрядные слова. Число параметров, переносимых из старого стека в новый, может составлять от 0 до 31.

При вызове программ через шлюз должны выполняться следующие правила. Значение RPL селектора, вызывающего шлюз, CPL текущей программы должны быть меньше или равны значению DPL в байте доступа шлюза:

$$\max (CPL, RPL_{\text{селектора шлюза}}) \le DPL_{\text{шлюза}} .$$

Таким образом, дескриптор шлюза должен иметь меньший уровень привилегий, чем запрос и текущая программа. Вызванный сегмент программ, в котором находится шлюз вызова (на который шлюз указывает), должен иметь такой же или более высокий уровень привилегий, чем текущая программа, т.е. для этого сегмента

$$CPL \ge DPL_{\text{целевого сегм. кода}}.$$

Если эти правила выполняются, то после вызова командой call дескриптора шлюза в регистр CS загружается селектор – байты 2 и 3 этого дескриптора. Этот селектор выбирает из LDT дескриптор вызываемого сегмента программы. При этом младшие два бита (поле PL) селектора игнорируются, а вместо них в регистр CS в качестве CPL заносится значение DPL из дескриптора вызываемого сегмента. В регистр EIP из дескриптора шлюза загружается относительный адрес входа в программу – байты 0, 1, 6, 7.

Если вызываемая программа имеет другой (более высокий) уровень привилегий, чем текущая, то при выполнении команды call для нее создается новый стек. При этом в регистры SS и ESP из сегмента состояния задачи TSS загружается новое значение, определяющее начальный адрес нового стека. В этот стек последовательно вводятся старые значения SS и ESP; параметры, переносимые из старого стека; старые значения CS и EIP. Число переносимых параметров определяется полем WC (рис.6.6), причем выбираются последние из загруженных в старый стек параметров. Последующие ячейки стека используются для хранения новых параметров.

По команде ret производится восстановление из стека старого содержимого регистров CS, EIP (IP) и SS, ESP (SP). Таким образом, одновременно с возвратом к исходной программе происходит и возвращение к старому стеку. Команда ret проверяет значение CPL и два младших разряда извлекаемого из стека старого содержимого CS, определяющие уровень привилегий программы, к которой осуществляется возврат. Команда выполняется, только если значение этих битов больше или равно CPL, т.е. если возврат осуществляется к программе с таким же или меньшим уровнем привилегий.

Нарушение командами jmp, call, ret указанных выше правил обращения к сегментам программ вызывает прерывание типа 13 (нарушение защиты).

Реализация ввода-вывода с помощью команд in, out, ins, outs в защищенном режиме производится с учетом CPL выполняемой программы. Величина CPL сравнивается со значением поля IOPL в регистре EFlags. Ввод-вывод производится только при выполнении условия

$CPL \le IOPL$.

Нарушение этого условия вызывает прерывание 13, за исключением случаев, когда при решении задачи доступ к порту разрешается специальной битовой картой ввода-вывода.

Резюмируя, правила использования шлюзов можно записать неравенствами

1) $DPL_{\text{шлюза}} \ge \max (CPL, RPL_{\text{селектора шлюза}})$

2) $CPL \ge DPL_{\text{целевого сегм. кода}}$

СПИСОК ЛИТЕРАТУРЫ

1. Ассемблер. Учебник для вузов / В.И. Юров. – СПб.: Питер, 2010. – 637 с.
2. Ассемблер. Язык и программирование для IBM PC: пер. с англ. / Питер Абель. – М.: Изд-во Корона-Век, Энтроп, 2007. – 736 с.
3. Баула В.Г. Введение в архитектуру ЭВМ и системы программирования / В.Г. Баула. – М.: МГУ, 2003. – 133 с.
4. Бродин В.Б. Микропроцессор i486 / В.Б. Бродин, И.И. Шагурин. – М.: «Диалог-МИФИ», 1993. – 240 с.
5. Брусенцов Н.П. Микрокомпьютеры / Н.П. Брусенцов. – М.: Наука, 1985. – 208 с.
6. Буза М.К. Архитектура компьютеров: учебник / М.К. Буза. – Минск: Новое знание, 2006. – 559 с.
7. Вычислительные машины и системы: учеб. пособие / А.П. Ремонтов, А.П. Писарев. – Пенза: Изд-во Пензенского государственного университета, 2006. – 96 с.
8. Григорьев В.Л. Программирование однокристальных микропроцессоров / В.Л. Григорьев. – М.: Энергоатомиздат, 1987. – 288 с.
9. Григорьев В.Л. Микропроцессор i486. Архитектура и программирование / В.Л. Григорьев. – В 4-х кн. – М.: Гранал, 1993.
10. Гук М.Ю. Аппаратные средства IBM PC. Энциклопедия / М.Ю. Гук. – СПб.: Питер, 2006. – 1072 с.
11. Жмакин А.П. Архитектура ЭВМ / А.П. Жмакин. – СПб.: БХВ-Петербург, 2006. – 320 с.
12. Каган Б.М. Электронные вычислительные машины и системы: учеб. пособие для вузов / Б.М. Каган. – М.: Энергоатомиздат, 1985. – 552 с.
13. Компьютерные сети: учеб. пособие / В. Чернега, Б. Платтнер. – Севастополь: Изд-во СевНТУ, 2006. – 500 с.
14. Коффрон Дж. Расширение микропроцессорных систем / Пер. с англ.; Под ред. П.В. Нестерова / Дж. Коффрон, В. Лонг. – М.: Машиностроение, 1987. – 320 с.
15. Микро-ЭВМ. / Пер. с англ. под ред. А. Дирксена. – М.: Энергоиздат, 1982. – 328 с.
16. Нортон П. Программно-аппаратная организация IBM PC / П. Нортон. – К., 1987. – 416 с.
17. Паттерсон Д. Архитектура компьютера и проектирование компьютерных систем. / Д. Паттерсон, Дж. Хеннеси. – 4-е изд. – СПб.: Питер, 2012. – 784 с.
18. Пильщиков В.Н. Программирование на языке ассемблера IBM PC / В.Н. Пильщиков. – М.: «Диалог-МИФИ», 2006. – 288 с.
19. Процессоры Pentium III, Athlon и другие / М. Гук, В. Юров – СПб.: Питер, 2000. – 480 с.
20. Сафонов В. http://www.intuit.ru/studies/courses/641/497/lecture/11272?page=1#keyword12

21. Системное программное обеспечение / А.В. Гордеев, А.Ю. Молчанов. – СПб.: Питер, 2002. – 736 с.
22. Соппа И.В. Введение в архитектуру персонального компьютера / И.В. Соппа. – Издательство Дальневосточного университета, 2001.
23. Старков В.В. Компьютерное железо: архитектура, устройство и конфигурирование / В.В. Старков. – М.: Горячая линия-Телеком, 2004. – 424 с.
24. Степанов А.Н. Архитектура вычислительных систем и компьютерных сетей: учеб. пособие / А.Н. Степанов. – М. и др.: Питер, 2007. – 509 с.
25. Таненбаум Э. Архитектура компьютера: пер. с англ. / Э. Таненбаум; Ред. А. В. Гордеев. – 6-е изд. – М. и др.: Питер, 2013. – 816 с.
26. Харрис Д.М. Цифровая схемотехника и архитектура компьютера / Дэвид М. Харрис и Сара Л. Харрис. – Изд-во Morgan Kaufman English Edition, 2013. – 1621 с.
27. http://bit.samag.ru/archive/article/1310

ПРИЛОЖЕНИЕ А
СОСТАВ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ ЭВМ

А.1. Системное ПО

ПО делится на *системное* (*общее*) и *прикладное* (*специальное, функциональное*) [5] (п. 1.1).

Программы, относящиеся к системному ПО, называются *системными*, а к прикладному ПО – *прикладными*.

«Прикладные программы являются конечной целью программирования. Если сравнить их создание с изготовлением товаров народного потребления (продуктов питания, обуви, одежды), то системное программирование будет играть роль производства средств производства (машин, станков, инструментов и т.д.)» [5].

Это разделение настолько существенно, что и программисты делятся соответственно на системных и прикладных.

Назначение *системных программ* – обеспечить функционирование компьютерной системы как единого целого и создать надлежащую основу для: 1) разработки и 2) выполнения прикладных программ. Компьютерные (машинные) команды (инструкции) слишком элементарны для того, чтобы непосредственно из них можно было эффективно строить программы, реализующие практические приложения компьютеров (хотя в отдельных случаях идут и этим путем). Кроме того, не оснащенный программами компьютер воспринимает и выполняет только команды, представленные двоичным кодом, неблагоприятным для человека, даже если тройки цифр заменить восьмеричными или четверки – шестнадцатеричными цифрами. Впрочем, использование 8-ричного или 16-ричного кода, вводимого, например, с клавиатуры, уже предполагает наличие драйвера клавиатуры и какой-нибудь программы-монитора, обеспечивающей минимальные возможности управления самим компьютером, подобно тому, как драйверы обеспечивают управление внешними устройствами. Короче говоря, ЭВМ с дисплеем, клавиатурой и дисками становится той понятливой, предупредительной и исполнительной системой, с которой легко и удобно работать лишь после того, как системные программисты возведут над ее набором команд многоярусную программную надстройку. [5]

Системное ПО состоит из двух главных частей:

- Операционной системы. Например: Windows, UNIX, Linux; и
- Инструментальной системы (системы программирования, системы автоматизации программирования): C++, Pascal, Java и т.д.

Кроме этих компонентов в системное ПО входят

- Сервисные программы, к которым относят системы технического обслуживания и программы-утилиты.*

* Отнесение того или иного программного обеспечения к системному до некоторой степени условно и зависит от соглашений, используемых в конкретном контексте. Как правило, к системному ПО относятся операционные системы, сервисные программы, системы

программирования, и, иногда, системы управления базами данных, широкий класс связующего ПО.

Операционная система является важнейшей и центральной частью системного ПО. Она представляет собой комплекс взаимосвязанных программ, постоянно находящихся в основной памяти компьютера и предназначенных для управления работой всех других программ, а также устройств компьютера и организации взаимодействия с пользователем.

Основная функция всех ОС – посредническая (рис.А.1). Она заключается в обеспечении интерфейсов*:

- пользователя (между пользователем и программно-аппаратными средствами). Благодаря этому пользователь имеет возможность взаимодействовать с программами и аппаратурой компьютера, отдавая операционной системе содержательные приказы на языке операционной системы, вводимые с клавиатуры в системном приглашении для командного процессора или с манипулятора типа «мышь» в пиктографическом меню графического интерфейса;
- между программным и аппаратным обеспечением;
- между разными видами программного обеспечения.

* *Интерфейс* (англ. interface) – стык; сопряжение, взаимодействие; поверхность раздела, граница раздела. В компьютере *интерфейс* – это устройство и/или программа сопряжения двух объектов, обеспечивающая обмен информации между ними (п.1.7).

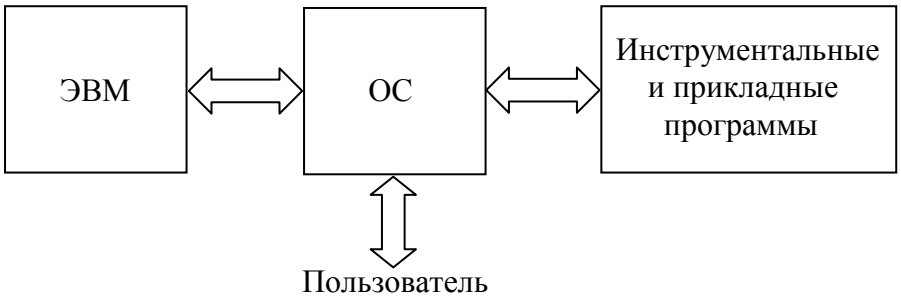


Рис. А.1. Назначение операционной системы

Операционная система связывает аппаратное обеспечение, прикладные программы и пользователя. Она взаимодействует с аппаратным обеспечением, прикладными программами и пользователем и может переносить информацию между ними. Прикладной программист не должен, например, беспокоиться о написании специальных программ для записи данных на все множество дисков, которые могут быть на персональном компьютере. Программист просто просит операционную систему записать данные на диск, а операционная система занимается зависящей от аппаратуры информацией. Операционная система получает предоставляемые прикладными программами данные и записывает их на физический диск. Такие детали, как где конкретно на физическом диске находится файл и когда диск готов к приему данных незаметно берет на себя операционная система. Пользователь и прикладное ПО не должны заботиться об этом.

Операционная система связывает также другие элементы аппаратуры (устройства), подключаемые к компьютеру. Когда вы устанавливаете аппаратное устройство, маленькая программа, называемая драйвером устройства, дает операционной системе инструкции по

работе с этим устройством. Типичными устройствами являются видеоадаптеры, клавиатура и мышь, а также последовательные и параллельные порты.

Кроме посреднической ОС выполняет и другие функции:

осуществление своего автоматического запуска;

организацию и обслуживание файловой системы;

управление приложениями (инструментальными и прикладными программами и утилитами);

быстрое обслуживание запросов извне с помощью механизма прерываний;

выполнение программ в реальном времени (в темпе с управляемым технологическим процессом или техническим объектом) – ОС реального времени (ОС РВ).

ОС содержит такие программы, как:

а) Базовая система ввода-вывода (BIOS – Basic Input-Output System) – это набор коротких программ, реализующих основные (низкоуровневые) операции ввода-вывода. Они хранятся в постоянной (энергонезависимой) памяти компьютера (в ПЗУ – постоянном запоминающем устройстве, или ROM – Read Only Memory – памяти только для чтения) и записываются туда при изготовлении материнской платы. Данная система, по сути, «встроена» в компьютер и является одновременно его аппаратной частью и частью операционной системы.

б) Загрузчик операционной системы – короткая программа, находящаяся в первом секторе любого загрузочного диска (диска с операционной системой). Функция этой программы заключается в считывании в память основных дисковых файлов ОС и передаче им дальнейшего управления ЭВМ. Загрузчик стартует из BIOS: завершая свою работу, программы BIOS дают команду на считывание загрузчика с диска и передают ему управление. Благодаря загрузчику и BIOS все ОС обеспечивают свой автоматический запуск.

в) Ядро ОС, или Супервизор, или Монитор, или базовый модуль. Супервизор образует основу операционной системы, ее ядро. Он связывает воедино все остальные части ОС: управляет работой программ и файловой системой, обеспечивает доступ к ней и обмен файлами между периферийными устройствами. Важнейшим свойством ядра является то, что код ядра исполняется с максимально высоким уровнем привилегий – ему доступны все команды процессора и разрешен доступ к любой аппаратуре компьютера. В конечном итоге ядро изолирует все остальные программы от аппаратуры компьютера. Все остальные программы получают доступ к аппаратным ресурсам только через посредничество ядра ОС. В MS DOS – дисковой операционной системы первых IBM-совместимых ЭВМ роль ядра, супервизора играла программа IBM.SYS, в операционной системе Windows эту функцию выполняет модуль KERNEL.

В ядре ОС выделяют несколько подсистем, каждая из которых отвечает за выполнение той или иной задачи:

- файловая система (предназначена для хранения данных на дисках и обеспечения доступа к ним);
- система управления памятью (размещает программы в памяти);
- система управления программами (осуществляет запуск и выполнение программ);
- система связи с драйверами устройств (отвечает за взаимодействие с внешними устройствами);
- система обработки ошибок;
- служба времени (предоставляет всем программам информацию о системном времени*).

* *Системное время* в компьютере – это просто текущая дата и время дня. Они отображаются в правом нижнем углу экрана. Системное время отсчитывается *системными часами*, или *часами реального времени*, представляющими собой энергонезависимое электронное устройство, работающее даже при выключенном компьютере.

Отметим, что кроме часов реального времени, любой компьютер (даже простейший IBM PC) содержит устройство, называемое *системным таймером*. Это устройство подключено к линии запроса на прерывание IRQ0 и вырабатывает прерывание номер 8h.

Наиболее важная часть работы ОС и ее ядра – работа с приложениями. С точки зрения управления приложениями различают однозадачные и многозадачные ОС. Однозадачные ОС передают все ресурсы компьютера одной задаче. Большинство современных ОС многозадачные. Они управляют распределением ресурсов между задачами и обеспечивают:

- возможность одновременной или поочерёдной работы нескольких приложений (мультипрограммный режим);
- возможность обмена данными между приложениями;
- возможность совместного использования ресурсов несколькими приложениями.

г) Командный процессор и графический интерфейс.

Командный процессор – это программа-интерпретатор, которая расшифровывает и исполняет команды пользователя, поступающие, прежде всего, через клавиатуру (через интерфейс командной строки) или путем последовательного исполнения пакетных командных файлов – файлов, в которых команды записаны друг за другом. Команды имеют вид текстовых предложений, содержащих имя затребованной операции, имя файла, имя каталога, символы ключей, уточняющих выполнение операции, и т.п. информацию. Для операционных систем MS DOS и Windows9X в качестве интерпретатора команд использовался командный процессор Command.com, для всех остальных ОС семейства Windows (NT/2k/XP/Vista/7/8 и старше) – Cmd.exe. Интерфейс подобного рода называют иногда *текстовым пользовательским интерфейсом* (TUI – Textual User Interface).

Графический интерфейс. Взаимодействие пользователя с компьютером через командный процессор ненаглядно и неудобно. Для его упрощения в состав современных операционных систем входят программные модули, создающие *графический пользовательский интерфейс* (GUI – Graphical User Interface). С его помощью пользователь может вводить команды через посредство мыши из пиктографического меню или манипулируя с графическими объектами. В этом случае достаточно «ткнуть» (кликнуть) мышью (курсором) в определенное место экрана, чтобы указать нужный файл, каталог и требуемое действие. Основными элементами графического интерфейса Windows являются рабочий стол с пиктограммами (значками, ярлыками, окнами – областями экрана, ограниченными прямоугольными рамками); панель задач – узкая полоска вдоль любой стороны экрана, на которой размещаются программные кнопки, индикаторы, главное меню (кнопка Пуск); контекстное меню (отображается при щелчке правой кнопкой мыши по выбранному объекту).

д) Планировщик заданий. Служит для планировки запуска программ – «запуска по расписанию» любой программы в определенное время с определенными условиями, например, вывода на экран сообщения, пуска работы технологического оборудования и т.п.;

е) Программы взаимодействия с аппаратным обеспечением и управления вводом-выводом: подпрограммы обработки прерываний, драйверы периферийных устройств, супервизор ввода-вывода и др.;

Средства аппаратного обеспечения отличаются гигантским многообразием. Существуют сотни различных моделей видеокарт, звуковых карт, принтеров, сканеров и т.д. Ни один разработчик программного обеспечения не может предусмотреть все варианты взаимодействия своей программы, например, с принтером. Поэтому каждый разработчик оборудования прикладывает к нему специальные программы-драйверы. Управление взаимодействием прикладных программ с драйверами – одна из функций ОС.

ж) Библиотекарь. Библиотека, упрощённо говоря, является своеобразным сборником ярлыков на файлы документов и мультимедиа: библиотека собирает файлы, а программа-библиотекарь управляет обращением к ним;

з) Программы вызовов трансляторов.

Есть приложения вычислительной техники, для которых ОС излишни. Например, встроенные микроконтроллеры (однокристальные микрокомпьютеры) содержатся сегодня во многих бытовых приборах, автомобилях (иногда по десятку в каждом), сотовых телефонах и т.п. Зачастую такой компьютер постоянно исполняет лишь одну программу, запускающуюся по включении. И простые игровые приставки – также представляющие собой специализированные микрокомпьютеры – могут обходиться без ОС, запуская при включении программу, записанную на вставленном в устройство «картридже» или компакт-диске. Тем не менее, некоторые микрокомпьютеры и игровые приставки всё же работают под управлением особых собственных ОС. В большинстве случаев, это UNIX-подобные системы (последнее особенно верно в отношении программируемого коммутационного оборудования: файрволов, маршрутизаторов*).

* *Файрвол*, *файервол* (англ. firewall) – межсетевой экран или сетевой экран – комплекс аппаратных или программных средств, осуществляющий контроль и фильтрацию проходящих через него сетевых пакетов. Основной задачей сетевого экрана является защита компьютерных сетей или отдельных узлов от несанкционированного доступа. Также сетевые экраны ч –

(нем. brandmauer) – заимствованный из немецкого языка термин, являющийся аналогом английского firewall в его оригинальном значении (стена, которая разделяет смежные здания, предохраняя от распространения пожара). Стандартный файрвол Windows находится в панели управления и называется брандмауэр Windows, но кроме него на компьютере могут быть установлены и файрволы сторонних производителей.

Маршрутизатор (то же, что *роутер*, от англ. router) – это специализированный сетевой компьютер, служащий для соединения сетей.

Система автоматизации программирования (*инструментальные средства*, *инструментальная система* ЭВМ) позволяет использовать ЭВМ для повышения эффективности программистского труда, для его автоматизации при разработке исходного текста программы, переводе программы в машинный код, ее отладке, тестировании и документировании. Говоря коротко, инструментальные системы (ИС) предназначены для создания новых программ.

ИС включает в себя [21]:

язык или языки программирования, на которых создаются исходные тексты программ, и

набор таких программ, как:

а) Транслятор;

б) Редактор;

в) Компоновщик (редактор связей);

г) Библиотеки прикладных программ;
д) Загрузчик;
е) Отладчик.

Не бывает самостоятельных (оторванных от ОС) систем программирования. Любая система программирования может работать только в соответствующей ОС, под которую она и создана, однако при этом она может позволять разрабатывать программное обеспечение и под другие ОС. Например, одна из самых популярных систем программирования на языке C/C++ от фирмы Watcom позволяет получать программы и для самой OS/2, и для DOS, и для Windows.

В современных ЭВМ инструментальные программы, как правило, не являются автономными (или, как говорят, портативными – portable), а встраиваются в единую программную среду (*интегрированную среду программирования, разработки* IDE – Integrated Development Environment), обеспечивающую быстрый и удобный доступ к любой из них из любой точки среды.

Языки программирования в свою очередь делятся на *языки низкого уровня* и *языки высокого уровня* (в зависимости от степени их близости к естественному, человеческому языку). К языкам низкого уровня, или машинно-ориентированным относятся машинные языки («машинные коды»), языки ассемблера, или автокоды, и языки макроассемблера. Среди языков высокого уровня выделяют *проблемно-ориентированные*, такие как Паскаль, Delphi, Фортран, Бэйсик, FoxPro, dBase, Java и др., и языки *широкой ориентации*, или *универсальные*: Модула, Си++, ПЛ/1 и др.

Транслятор (англ. Translator – переводчик) – это программа-переводчик. Она преобразует программу, написанную на одном из языков программирования, в программу, состоящую из машинных команд. Трансляторы бывают двух типов: компиляторы и интерпретаторы [24]. *Компилятор* (англ. Compiler – составитель, собиратель) читает всю программу *целиком*, делает ее перевод и создает законченный вариант программы на машинном языке, сохраняя его в виде так называемого объектного (object), или абсолютного, файла или файлов (кодов, модулей) (с расширением “obj”). *Интерпретатор* (англ. Interpreter – истолкователь, устный переводчик) осуществляет пооператорную трансляцию и выполнение исходной программы: переводит и выполняет программу *строка за строкой*.

После того, как программа откомпилирована, ни сама исходная программа, ни компилятор более не нужны. В то же время программа, обрабатываемая интерпретатором, должна заново переводиться на машинный язык при каждом очередном ее запуске. В отличие от компилятора, интерпретатор не порождает на выходе программу на машинном языке. Распознав команду исходного языка, он тут же выполняет ее. Интерпретатор позволяет начать обработку данных после написания даже одной команды. Это делает процесс разработки и отладки программ более гибким. Кроме того, отсутствие выходного машинного кода позволяет не "захламлять" внешние устройства дополнительными файлами, а сам интерпретатор можно достаточно легко адаптировать к любым машинным архитектурам, разработав его только один раз на широко распространенном языке программирования. Поэтому, интерпретируемые языки, типа Java Script, VB Script, язык MatLab получили широкое распространение. Недостатком интерпретаторов является низкая скорость выполнения программ. Обычно интерпретируемые программы выполняются в 50-100 раз медленнее программ, написанных в машинных кодах.

В итоге откомпилированные программы работают быстрее, но интерпретируемые проще исправлять и изменять.

Редактор (редактор исходного текста программы, текстовый редактор, Editor) – программа для создания и редактирования исходного текста программ. Редактор может быть отдельным приложением, или встроен в интегрированную среду разработки IDE.

Компоновщик (редактор связей, Linker) (программа) предназначен для связывания между собой объектных файлов, порождаемых компилятором, а также файлов библиотек прикладных программ, входящих в состав системы программирования. Объектный файл (или набор объектных файлов) не может быть исполнен до тех пор, пока все модули и секции в нем не будут увязаны между собой и все используемые в них адреса переменных и переходов согласованы. Это и делает редактор связей (компоновщик). Результатом его работы является единый файл (часто называемый «исполнимым файлом», executive, файл с расширением “exe”), который содержит весь текст результирующей программы на языке машинных кодов. Компоновщик может порождать сообщение об ошибке, если при попытке собрать объектные файлы в единое целое он не смог обнаружить какой-либо необходимой составляющей. Эта программа (exe файл) затем и выполняется.

Библиотеки прикладных программ содержат в себе наиболее часто используемые функции и подпрограммы в виде готовых объектных модулей.

Загрузчик (Loader) – программа, отвечающая за загрузку исполнимых файлов и запуск соответствующих новых процессов. При запуске новой программы загрузчик должен:

- считать данные из запускаемого *файла*;
- если необходимо – загрузить в *память* недостающие *динамические библиотеки*;
- заменить в коде новой программы относительные адреса и символические ссылки на точные с учётом текущего размещения в памяти, то есть выполнить *связывание, настройку адресов* (англ. Relocation, Address Relocation);
- создать в памяти образ нового процесса и *запланировать* его к исполнению.

(Загрузчик операционной системы действует по схожему принципу, но обычно является отдельной программой, поскольку решает специфическую задачу – запуск самой операционной системы.)

Отладчик (Debugger) – модуль интегрированной среды разработки или отдельная (портативная) программа, предназначенная для выполнения разрабатываемой программы в заданном режиме с целью поиска, обнаружения и локализации ошибок. Отладчик позволяет выполнять *трассировку* программы (выполнять программу по шагам – операторам или командам) или выполнять ее целиком, отслеживать, устанавливать или изменять значения переменных в процессе выполнения кода, устанавливать и удалять контрольные точки, или условия остановки, и т.д. В режиме трассировки программист видит последовательность выполнения команд и значения переменных на данном шаге выполнения программы, что позволяет легче обнаруживать ошибки. Трассировка может быть начата и окончена в любом месте программы, выполнение программы может останавливаться на каждой команде или на точках останова, трассировка может выполняться с заходом в процедуры и без заходов, а также осуществляться в обратном порядке (шаг назад).

Сервисные программы расширяют возможности ОС по обслуживанию системы и обеспечивают удобство работы пользователя. По мере эксплуатации любой операционной системы возникает необходимость в реализации некоторых функций, которые не входят в состав конкретной операционной системы. В этом случае пользователи, владеющие основами программирования, а часто и отдельные фирмы создают соответствующие программы. Иногда такие разработки впоследствии включаются в состав операционной системы и становятся ее частью. А наиболее часто они эксплуатируются пользователями самостоятельно, расширяя возможности операционных систем. Такие программы по-

лучили название сервисных программ, или программ обслуживания. К этой категории относят системы технического обслуживания и программы-утилиты.

Системы технического обслуживания – это совокупность программно-аппаратных средств, которые выполняют контроль, тестирование и диагностику устройств компьютера и обнаруживают неисправности в их работе. Они являются инструментом специалистов по эксплуатации и ремонту технических средств компьютера.

Программы-утилиты («утилиты», служебные программы) – это полезные специализированные программы, не включаемые обычно в серийные комплекты ОС и предназначенные для решения узкого круга вспомогательных задач. К ним относятся: диагностика, тестирование аппаратных и программных средств, оптимизация использования дискового пространства, восстановление разрушенной на магнитом диске информации, сжатие (архивирование) информации, запись информации на съемные диски (например, программа Nero Start Smart), защита от компьютерных вирусов (Dr.Web, Avast и т.п.) и др.

А.2. Прикладное ПО

Прикладное ПО представляет собой пакет прикладных программ, созданных пользователем и для пользователя и предназначенных для выполнения научных, проектных, производственных и учебных заданий, а также для бытового применения и развлечений. В большинстве операционных систем прикладные программы не могут обращаться к ресурсам компьютера напрямую, а взаимодействуют с оборудованием и другими программами посредством операционной системы.

Как правило, все пользователи предпочитают иметь набор прикладных программ, который нужен практически каждому. Их называют *программами общего назначения*. К их числу относятся:

- текстовые и графические редакторы, с помощью которых можно готовить различные тексты, создавать рисунки, строить чертежи; проще говоря, писать, чертить, рисовать;
- системы компьютерной верстки для создания макетов страниц для книг, журналов, рекламных объявлений, упаковки, дизайна выставочных стендов с целью последующей печати в типографии или на принтере;
- системы управления базами данных (СУБД), позволяющие превратить компьютер в справочник по любой теме (Clipper, Microsoft Access, Paradox, dBase, FoxPro, Visual FoxPro, Oracle, Informix);
- табличные процессоры, позволяющие организовывать очень распространенные на практике расчеты для электронных таблиц (например, программа Microsoft Excel);
- коммуникационные (сетевые) программы, предназначенные для обмена информацией с другими компьютерами, объединенными с данным в компьютерную сеть: веб-браузеры, или веб-обозреватели (средства просмотра Web-документов), вспомогательные браузеры (обозреватели) и др.;
- очень популярным видом прикладного программного обеспечения являются компьютерные игры. Большинство пользователей именно с них начинает свое общение с ЭВМ.

Кроме того, имеется большое количество *прикладных программ специального назначения* для профессиональной деятельности – *программы профессионального уровня*. Их часто называют *пакетами прикладных программ*. Это, например:

- бухгалтерские программы, производящие начисления заработной платы и другие расчеты, которые делаются в бухгалтериях (1 С: Бухгалтерия и др.);
- системы автоматизированного проектирования (САПР, CAE – Computer Aided Engineering) и конструирования, дизайна (CAD – Computer Aided Design, ACAD – Automatic CAD, Inventor), которые помогают инженерам и конструкторам разрабатывать проекты различных технических устройств;
- пакеты для инженерных и научных расчетов – программы, позволяющие выполнять сложные математические расчеты при решении научных и инженерных задач без составления программ или с составлением программ-сценариев – скриптов (MCAD – Mathematical Computer Aided Design, MatLab – Matrix Laboratory, ANSYS – Analysis of Systems – программа для расчета механических и электромагнитных полей);
- обучающие программы по разным школьным предметам;
- программы сбора и обработки данных для измерительных и информационных систем;
- программы управления технологическими процессами и техническими объектами в реальном масштабе времени (в темпе с процессом), используемые в компьютеризированных системах автоматического управления (САУ), в ИУВС;
- и многое другое.

ПРИЛОЖЕНИЕ Б
СВОДКА ПРИКЛАДНЫХ КОМАНД IA-32

Б.1. Функциональная классификация машинных команд

Набор команд процессоров IA-32 содержит более 300 машинных команд [1 - 4, 8, 9, 18]. С появлением каждой новой модели МП их количество, как правило, возрастало, отражая тем самым архитектурные нововведения, отличающие эту модель от ее предшественниц. Набор команд можно структурировать по функциональным группам. Очень полезно перед началом изучения работы отдельных команд получить общее представление о всей системе команд МП (рис. Б.1).

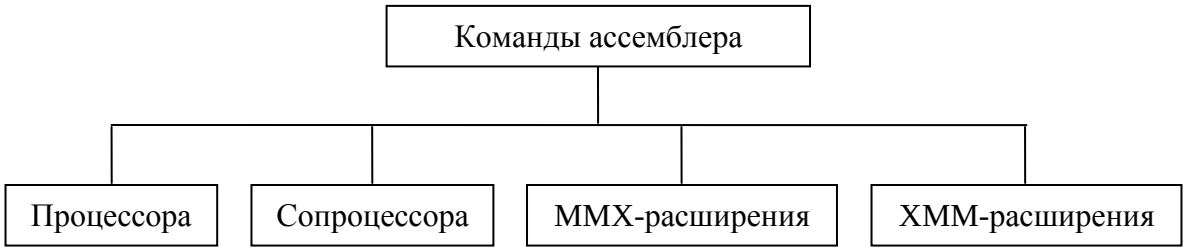


Рис. Б.1. Машинные команды МП IA-32

Как видно из этой структуры, МП IA-32 может выполнять целочисленные операции и операции с плавающей точкой. Для этого в его архитектуре есть два отдельных блока:

устройство для выполнения целочисленных операций («процессор» на рис. Б.1);

устройство для выполнения операций с плавающей точкой (FPU – Floating Processor Unit, или «сопроцессор» на рис. Б.1).

Кроме того, имеются специальные блоки для поддержки функций мультимедиа расширений («MMX-расширение» и «XMM-расширение»).

Каждое из этих устройств имеет свою систему команд. В настоящем пособии разбираются только команды целочисленного устройства – команды для операций над целыми числами.

Целочисленные команды процессора разбиты на *три основные группы по функциональному назначению* (рис. Б.2):

1. Команды *обработки данных*:
- пересылки и ввода-вывода данных. Команды пересылки содержат команды общего назначения (mov, …) и работы со стеком (push, pop), а команды ввода-вывода представлены двумя командами: in и out;
- арифметических операций, поддерживающих двоичную и десятичную арифметику (add, sub, neg, …);
- логических (булевых) операций и сдвигов (and, or, …);
- операций с битами (bt, bts, btc, bsf, bsr);
- цепочечные команды (movs, …).

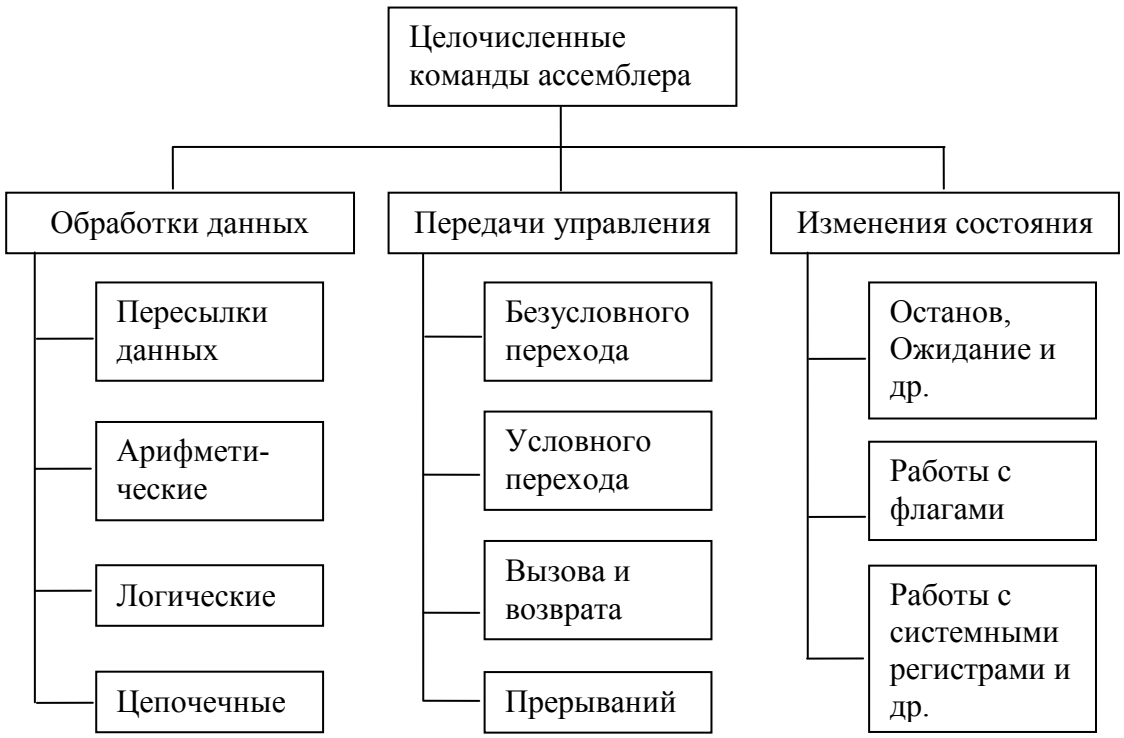


Рис. Б.2. Целочисленные команды IA-32

2. Команды *передачи управления*:
- простых переходов (jmp),
- переходов по условию, или ветвления (j*c*),
- вызовов (call) и возвратов (ret) из подпрограмм,
- прерываний (int) и возвратов из прерываний (iret) и др.

3. Команды *изменения состояния процессора и системных регистров*:
- останова (hlt). По команде hlt (halt) процессор прекращает выполнение программы и останавливается. Сигналы прерывания INTR, NMI и сброса RESET выводят процессор из состояния останова.
- ожидания (wait). Эта команда аналогична hlt, но возврат из прерываний происходит на команду wait, а не на следующую за ней команду, как в случае hlt,
- работы с флагами (sti, cli, pushf, popf, …),
- работы с системными регистрами и организации зашиты памяти (lgdt, sgdt, lidt, sidt, …),
- поддержки языка высокого уровня (bound – проверка границ массива, enter – обращение к процедуре, leave – выход из процедуры) и др.

Кроме команд используются *префиксы* – однобайтные коды, устанавливаемые перед командой и модифицирующие выполняемую операцию:
- префиксы замены сегментов (CS:, DS:, … , SS:),
- префиксы изменения разрядности операндов и адресов (OS, AS – два префикса),
- префикс блокировки шины (LOCK),

- префиксы повторения для цепочечных команд (REP, REPE/REPZ, REPNE/REPNZ – три префикса).

Префиксы считаются частью команды в том смысле, что прерывание не воспринимается между префиксом и командой.

Б.2. Обозначения

Служебные слова:

MOV, PUSH, POP, … , IN, OUT, ADD, SUB, … – названия команд;

AL, AH, AX, EAX, BL, BH, BX, EBX, …, SP, ESP, CS, … DS, … – имена или содержимое регистров процессора.

Специальные символы:

, (запятая) – разделение операндов, разделитель операндов;

[…] (квадратные скобки) – прямая или косвенная адресация.

Операнды:

s – операнд-источник (source) или поле в мнемокоде команды для его адресации;

d – то же для операнда-приемника (destination);

r – общий регистр процессора (general register) или его содержимое: AL, AH, AX, EAX, … , SI, ESI, DI, EDI, BP, EBP, SP, ESP;

ac – аккумулятор (accumulator) или его содержимое: AL, AX или EAX;

sr – сегментный регистр (segment register) или его содержимое: CS, DS, SS, ES, FS, GS;

m – ячейка/ее содержимое памяти (memory);

im – непосредственный операнд (immediate operand);

port – адрес порта в командах IN и OUT;

m(addr) – содержимое ячейки памяти, адрес которой равен addr;

m(r) – содержимое ячейки памяти, адрес которой находится в регистре r;

p – содержимое порта ввода-вывода;

p(port) или (port) – содержимое порта ввода-вывода, адрес которого равен port;

p(DX) и p(EDX) – содержимое порта ввода-вывода, адрес которого находится в регистре DX и EDX соответственно;

EA – эффективный адрес памяти (Effective Address) – адрес ячейки памяти внутри сегмента, т.е. ее смещение (Offset) в соответствии с указанным режимом адресации. Мнемоника EA в ассемблер не встроена;

EA(name) – эффективный адрес (смещение) переменной с именем name.

Действия:

← или := – переслать, присвоить. Например: d ← s – переслать значение операнда-источника s в приемник d (присвоить операнду d значение операнда s);

¬ или ⌉ или # – операция побитной инверсии двоичного числа. Например: ⌉1011=0100;

: (двоеточие) или _ (нижняя черта) – конкатенация (сочленение), т.е. образование одного операнда из двух путем их сочленения. Например, если d – m-битный операнд, а s – n-битный, то запись d:s или d_s обозначает (m+n)-битный операнд, у которого младшие n битов равны s, а старшие m битов – d (рис.Б.3)

Рис. Б.3. Конкатенация операндов

Целые числа. В языке ассемблера имеются средства записи целых и вещественных чисел. Вещественные числа мы рассматривать не будем, а ограничимся только целыми числами.

Целые числа могут быть записаны в десятичной, двоичной, восьмеричной и шестнадцатеричной системах счисления (другие системы не допускаются). Десятичные числа записываются как обычно, а вот при записи чисел в других системах в конце числа ставится суффикс – буква, которая указывает, в какой системе записано число: в конце двоичного числа ставится буква b (binary), в конце восьмеричного – буква o (octal) или буква q (чтобы не путать "o" с нулем), в конце шестнадцатеричного числа – буква h (hexadecimal). Ради общности суффикс, а именно букву d (decimal), разрешается ставить и в конце десятичного числа, но обычно этого не делают.

Примеры:

десятичные числа: 25, –386, +4, –386d

двоичные числа: 101b, –11000b

восьмеричные числа: 74q, –74q

шестнадцатеричные числа: 1AFh, –1AFh, 0AFh (но не AFh, так как это идентификатор, а не число)

Далее в разделе рассматриваются машинные команды только в мнемонической записи (на ассемблере). Двоичные коды команд, а также директивы ассемблера не обсуждаются. Не рассматриваются также команды для работы с системными регистрами (системные команды). Нужный материал можно найти в [1-4, 8, 9, 18].

Б.3. Команды обработки данных

Таблица Б.1 – Команды пересылки и ввода-вывода данных

Мнемоника	Действие
Пересылка данных	
MOV d, s (move, переслать)	d ← s
Варианты команды MOV в зависимости от кода команды: r/m ← r, r ← r/m, r/m ← im, r ← im, ac ← m*, m ← ac*, sr ← r/m, r/m ← sr. В защищенном режиме команда MOV sr, r/m вызывает следующие действия: sr ← selector of r/m, dr ← m(Line address of DT + index×8), где dr – дескрипторный регистр, DT – дескрипторная таблица, глобальная или локальная. В команде MOV sr, r/m нельзя указывать сегментный регистр кода CS, так как при этом результат операции неопределен.	
* Укороченные форматы команд	
PUSH s (push, втолкнуть в стек)	Стек ← s
Действие подробно: SP ← SP–n, где n – размер операнда-источника s в байтах; m(SS:SP) ← s. s: m, r, sr (кроме CS)	
PUSHF (втолкнуть флажки в стек)	Стек ← Flags
SP ← SP–2; m(SS:SP) ← Flags	
POP d (pop, вытолкнуть из стека)	d ← Стек
d ← m(SS:SP); SP ← SP+n, где n – размер операнда-приемника d в байтах. d: m, r, sr (кроме CS)	
POPF (вытолкнуть во флажки из стека)	Flags ← Стек
Flags ← m(SS:SP); SP ← SP+2	
XCHG r, s (exchange, обменять содержимое регистра и источника)	Буфер ← r, r ← s, s ← Буфер. Коротко: r ↔ s. s: r, m
XLAT (занести в AL из таблицы)	AL ← m(DS:BX+AL)
LEA r, m (load effective address, загрузить эффективный адрес памяти EA в r)	r ← EA(m)
При выполнении команды LEA вычисляется эффективный адрес EA памяти (в соответствии с указанным режимом адресации) и его значение (а не адресуемое им слово в памяти!) загружается в указанный общий регистр. Пример: BX=0400h, SI=003Ch; lea BX, [BX][SI]; BX=043Ch	
LDS r, m (загрузить слово из памяти в r, а следующее слово – в DS)	r ← m(DS_0+EA(m)), DS ← m(DS_0+EA(m)+2)
LES r, m (то же, но для ES)	то же, но для ES
Команды LDS и LES выполняют почти одни и те же действия: вычисляется эффективный адрес EA памяти, который суммируется с содержимым регистра DS; затем слово из памяти по полученному адресу загружается в адресуемый командой общий регистр, а следующее слово из памяти загружается в регистр DS (команда LDS) или ES (команда LES). Пример: DS=C000, DI=0010, m(C0010)=0180, m(C0012)=2000; LDS SI, [DI]; SI=0180, DS=2000	
LAHF (загрузить в AH младший байт Flags)	AH ← Flags<7:0>
SAHF (сохранить AH в младшем байте Flags)	Flags<7:0> ← AH

Ввод-вывод данных	
IN ac, port (input, ввести в аккумулятор из порта)	ac ← p(port) – содержимое порта с адресом port передается в аккумулятор ac: AL, AX, EAX
IN ac, DX (input, ввести в аккумулятор из порта)	ac ← p(DX) – содержимое порта, адрес которого находится в DX, передается в аккумулятор ac: AL, AX, EAX
IN ac, EDX (аналогично)	Ac: AL, AX, EAX
OUT port, ac (output, вывести в порт из аккумулятора)	p(port) ← ac – содержимое аккумулятора передается в порт с адресом port. Здесь ac: AL, AX, EAX
OUT DX, ac (output, вывести в порт из аккумулятора)	p(DX) ← ac – содержимое аккумулятора передается в порт, адресуемый регистром DX. Ac: AL, AX, EAX
OUT EDX, ac (аналогично)	Ac: AL, AX, EAX

Таблица Б.2 – Примеры команд пересылки и ввода-вывода

Мнемоника	Действие
Непосредственная адресация	
mov AL, 100	AL ← 100 – занести в регистр-аккумулятор AL число 100. Здесь операнд-источник (число 100) задается непосредственно, операнд-приемник (AL) – прямо
Прямая адресация	

mov AL, byte ptr [100]
mov AL, [100]
mov AL, DS:[100]
mov AL, [DS:100]
mov AL, DS:100
	AL ← m(DS:100) – занести в регистр-аккумулятор AL содержимое 100-й ячейки памяти (однобайтной) из сегмента данных DS.

mov byte ptr ES:[100], 14	m(ES:100) ← 14 – занести число 14 в 100-ю ячейку памяти (однобайтную) дополнительного сегмента данных ES. Здесь операнд-приемник (ячейка памяти) адресуется прямо, а операнд-источник (число 14) – непосредственно

Численное задание адреса операнда (смещения, Offset) (в данных примерах 100) применяется редко; обычно используемым ячейкам в программе присваиваются символические имена. В процессе трансляции ассемблер вычисляет и подставляет значения смещений этих имен в формируемую им машинную команду в поле смещения в команде. Например:

```
Data SEGMENT
ORG 100
Var1 DB 5
…
Data ENDS
Code SEGMENT
ASSUME CS: Code, DS: Data
    mov AX, Data
    mov DS, AX
…
    mov AL, Var1 ; записать байт Var1 (с логическим адресом DS:0100) в AL.
```

Получается тот же результат, что и при исполнении команды с мнемокодом mov AL, [100]

Косвенная регистровая адресация	
mov AL, [SI] mov AL, DS:[SI]	AL ← m(DS:SI) – занести в AL содержимое ячейки из сегмента памяти данных DS, внутрисегментный адрес которой находится в регистре SI. Если предварительно загрузить в SI число 100, то в AL будет передано содержимое 100-й ячейки сегмента DS
Косвенная регистровая адресация со смещением	
mov AX, [BP]10 mov AX, [BP+10]	AX ← m(SS:BP+10) – передать десятое слово массива, адресуемого регистром BP в стековом сегменте
mov CL, byte ptr [SI]1 mov CL, byte ptr [SI+1]	CL ← m(DS:SI+1)
mov [BX]Temp, CX	m(DS:BX+EA(Temp) ← CX – передать слово Temp в массиве, адресуемом регистром BX
mov Array[SI], DX	m(DS: EA(Array)+SI) ← DX – передать DX в элемент массива
Другие примеры	
mov AX, 1234h	AX ← 1234h – занести в регистр-аккумулятор AX непосредственный операнд 1234_{16}
mov AX, ES:[1234h]	AX ← m(ES: 1234h) – передать в дополнительный сегмент данных ES, в его ячейку с внутрисегментным адресом 1234_{16} , содержимое аккумулятора AX
mov [40h:200h], BX	m(40h:200h) ← BX – переслать в регистр BX слово из ячейки памяти, расположенной в сегменте 40h по (внутрисегментному) адресу 200h
mov Var, AH	Var ← AH – присвоить переменной Var значение из AH. Переменная Var находится в определенном сегменте памяти и должна быть предварительно объявлена в нем с помощью директивы DB
mov SI, BX	SI ← BX – занести в регистр SI число из регистра BX
mov [SI], BX	m(DS:SI) ← BX – занести число из BX в ячейку памяти, адрес которой (внутрисегментный) содержится в SI. Сама ячейка располагается в сегменте данных, базовый адрес которого задается через регистр DS. Сегмент памяти, в котором располагается данная ячейка, в команде явно не указан. В этом случае используется по умолчанию сегмент данных DS
Эту же команду можно записать эквивалентно, задавая сегмент данных явно, т.е. используя его префикс: mov [DS:SI], BX или mov DS:[SI], BX	
mov [FS:SI], BX mov FS:[SI], BX	m(FS:SI) ← BX – передать число из BX в ячейку сегмента FS, имеющую смещение, задаваемое SI.
mov AX, CS	AX ← CS – занести в аккумулятор селектор из сегментного регистра CS. С помощью этой команды программист может определить местоположение текущей программы в памяти – базу сегмента кода в реальном режиме и селектор сегмента кода в защищенном
push DH	SP ← SP-1, m(SS:SP) ← DH – занести в стек байт из DH
pop byte ptr ES:[BX]	m(ES:BX) ← m(SS:SP); SP ← SP+2 – вытолкнуть из стека байт в ячейку сегмента ES, адрес которой (внутрисегментное смещение) находится в BX
push DX	SP ← SP-2, m(SS:SP) ← DX – втолкнуть в стек слово из DX

in AL, 60h	AL ← p(60h) – содержимое порта 60h вводится в AL
out DX, AL	p(DX) ← AL – в порт, адрес которого записан в DX, выводится слово (16-битное) из AX
in EAX, DX	EAX ← p(DX) – в регистр EAX вводится двойное слово (32-битное число) из порта, адрес которого записан в DX
out DX, EAX	p(DX) ← EAX – в порт, адрес которого записан в DX, выводится двойное слово (32-битное) из EAX

Таблица Б.3 – Команды арифметических операций

Мнемоника	Действие
Команды двоичной арифметики	
Сложение	
ADD d, s (ADDition, Сложение)	CF ← 0, CF_d ← d+s
Варианты команды ADD в зависимости от кода команды: r ← r1+r2; r ← r+m; m ← m+r; r/m ← r/m+im; ac ← ac+im	
ADC d, s (Addition with Carry, сложение с переносом)	CF ← 0, CF_d ← d+s+CF
INC d (INCrement by 1, увеличить на 1)	d ← d+1
Варианты команды INC в зависимости от кода команды: r←r+1; r/m←r/m+1	
Вычитание	
SUB d, s (SUBtract, Вычитание)	Программно: CF ← 0, CF_d ← 1_d–s, CF ← ⌉CF Аппаратно: CF ← 0, CF_d ← d+⌉s+1, CF ← ⌉CF
Варианты команды SUB в зависимости от кода команды: r ← r−r; r ← r−m; m ← m−r; r/m ← r/m−im; ac ← ac−im	
SBB d, s (SuBtract with Borrow, вычитание с заемом)	Команда SBB из разности операндов вычитает еще значение флажка CF. Варианты команды аналогичны SUB
DEC d (DECrement by 1, уменьшение на единицу)	Программно: d ← 1_d–1 Аппаратно: d ← d+⌉1+1=d+111…1
Варианты команды DEC в зависимости от кода команды: r ← r−1; r/m ← r/m−1	
Умножение	
MUL s (MULtiply, целочисленное умножение натуральных чисел)	ext:ac ← ac×s: AX = AL × s (если операнд s – байт); DX:AX = AX × s (если s – слово); EDX:EAX = EAX × s (если s – двойное слово)
s: регистр (8,16,32) либо память (8,16,32)	
IMUL s (Signed Integer MULtiply, целочисленное умножение чисел со знаком)	то же, но числа знаковые и должны быть записаны в дополнительных кодах. Результат тоже получается в дополнительном коде

Деление	
DIV s (DIVide unsigned, беззнаковое деление – деление натуральных чисел)	ac ← quot((ext:ac)/s); quot – частное от деления ext ← rem((ext:ac)/s) ; rem – остаток от деления AL = AX /s (если s – байт), AL = quotient, AH = remainder; AX = DX:AX /s (если операнд s – слово), AX = quotient, DX = remainder
s: регистр либо память	
IDIV s (Signed Integer DIVide, целочисленное деление чисел со знаком)	то же, но числа в дополнительных кодах
Изменение знака	
NEG d (NEGate, изменить знак, two-complement)	CF_d ← 1_0–d = 2^n–d = Доп(d)= ¬d+1, CF ← ¬CF
Команды сравнения	
CMP d, s (CoMPare operands, сравнение операндов)	d–s без запоминания разности. Воздействует на все арифметические флажки, кроме AC
Варианты команды CMP в зависимости от кода команды: r–r; m–r; r–m; r/m–im; ac–im	
CMPXCHG ac, r/m (CoMPare and eXCHanGe, сравнение и обмен)	ac–r/m. Если ac≠r/m, то zf=0, r/m ← ac; если ac=r/m, то zf=1, ac ← r/m
Сравнивает содержимое аккумулятора (AL, AX или EAX) и r/m путём вычитания ac–r/m. Если ac≠r/m, то zf=0, r/m ← ac; если ac=r/m, то zf=1, ac ← r/m	
Команды десятичной арифметики	
Коррекция после сложения	
AAA (ASCII Adjust after Addition, ASCII-коррекция после сложения)	if (AL & 0Fh) > 9 or (AF = 1) then AL ← AL + 6; AH ← AH + 1; AF ← 1; CF ← 1 else AF ← 0; CF ← 0; AL ← AL & 0Fh
Команда AAA переводит число, записанное в младшем полубайте аккумулятора AL в число, представленное в неупакованном формате в двоично-десятичном коде (старший полубайт AL содержит нули)	

DAA (Decimal Adjust for Addition, десятичная коррекция после сложения)	if (AL & 0Fh) > 9 or (AF = 1) then AL ← AL + 6; AF ← 1 else AF ← 0;
	if (AL > 9Fh) or (CF = 1) then AL ← AL + 60; CF ← 1
	else CF ← 0

Команда DAA корректирует результат предшествующего ей сложения двух упакованных десятичных операндов (результат должен находиться в AL). Эта команда изменяет содержимое AL так, чтобы AL содержал пару упакованных десятичных цифр	
Коррекция после вычитания	
AAS (ASCII Adjust after Substraction, ASCII-коррекция после вычитания)	if (AL & 0Fh) > 9 or (AF = 1) then AL ← AL – 6; AH ← AH – 1; AF ← 1; CF ← 1 else AF ← 0; CF ← 0; AL ← AL & 0Fh
Команда AAS корректирует результат предшествующего вычитания двух операндов, представленных в неупакованном двоично-десятичном коде, за счет перевода содержимого AL в двоично-десятичный код. Операнд назначения команды вычитания должен быть специфицирован так же, как AL. Старший полубайт AL всегда равен нулю	

DAS (Decimal Adjust for Subtraction, десятичная коррекция после вычитания)	if (AL & 0Fh) > 9 or (AF = 1) then AL ← AL – 6; AF ← 1 else AF ← 0
	if (AL > 9Fh) or (CF = 1) then AL ← AL - 60h
	CF ← 1 else CF ← 0

Команда DAS корректирует результат предшествующего ей вычитания двух упакованных десятичных операндов (результат должен находиться в AL). Эта команда изменяет содержимое AL так, чтобы AL содержал пару упакованных десятичных цифр	
Коррекция после умножения	
AAM (ASCII Adjust after Multiply, ASCII-коррекция после умножения)	AH ← AX / 10; AL ← AX MOD 10. Корректирует результат предшествующего умножения двух операндов, представленных в неупакованном двоично-десятичном коде
Двузначное неупакованное число берется из AX, проводится корректировка, результат заносится в AX. Старшие полубайты обоих сомножителей должны быть равны нулю	
Коррекция перед делением	
AAD (ASCII Adjust before Division, ASCII-коррекция перед делением)	AL ← AH×10+AL; AH ← 0. Переводит двухзначное число, представленное в неупакованном формате в регистре AX, из двоично-десятичного кода в двоичный
Команды преобразования типов	
CBW (Convert Byte to Word/ Convert Word to Double Word Extended, преобразование байта в слово/слова в двойное слово)	if (AL < 80h) then AH ← 0 else AH ← FFh. Переводит байтовую величину со знаком в эквивалентное ей слово со знаком
CWD (Convert Word to Double word, преобразование слова в двойное слово)	if (AX < 8000h) then DX ← 0 else DX ← FFFFh. CWD расширяет бит знака регистра AX на весь регистр DX. Эта команда генерирует двойное слово, эквивалентное числу со знаком, находящемуся в регистре AX
CWDE (Convert Word to Double Word Extended, преобразование слова в двойное слово)	Используются только регистры AX и EAX: если знаковый бит AX=0, то установить старшее слово EAX=0000h; если знаковый бит AX=1, то установить старшее слово EAX=0FFFFh
CDQ (Convert Double word to Quad word)	Назначение: расширение двойного слова со знаком до размера учетверенного слова (64 бита) со знаком
Команда CDQ копирует значения старшего бита регистра EAX на все биты регистра EAX	
MOVSX d, s (MOVe and Sign eXtension, пересылка со знаковым расширением)	Преобразование элементов со знаком меньшей размерности в эквивалентные им элементы со знаком большей размерности
MOVZX d, s (MOVe and Zero eXtension, пересылка с нулевым расширением)	Преобразование элементов без знака меньшей размерности в эквивалентные им элементы без знака большей размерности
Вспомогательные команды	
BSWAP s (Byte SWAP, перестановка байтов)	Изменение порядка следования байтов; переход от одной формы адресации к другой

Прочие команды с арифметическим принципом действия	
SETcc im (byte SET on condition, установка байта по условию)	Установка операнда логическим значением (01h или 00h) в зависимости от истинности условия, заданного модификатором кода операции cc
XADD d, s (eXchange and ADD, обмен и сложение)	Обменять между собой и сложить s ← d; d ← d+s

Таблица Б.4 – Установка признаков при выполнении команд арифметических, логических операций и сдвигов

Операции	Команды	Флаги состояния					
		OF	CF	AF	SF	ZF	PF
Сложение, вычитание	ADD ADC SUB SBC	+	+	+	+	+	+
	CMP CMPS SCAS NEG	+	+	+	+	+	+
	INC DEC	+	–	+	+	+	+
Умножение	MUL IMUL	+	+	?	?	?	?
Деление	DIV IDIV	?	?	?	?	?	?
Десятичная коррекция	DAA DAS	?	+	+	+	+	+
	AAA AAS	?	+	+	?	?	?
	AAM AAD	?	?	?	+	+	+
Логика	AND TEST OR XOR	0	0	?	+	+	+
Сдвиги	SHL SHR > 1	+	+	?	+	+	+
	SHL SHR > n	?	+	?	+	+	+
	SAR	0	+	?	+	+	+
	ROL ROR RCL RCR >1	+	+	–	–	–	–
	ROL ROR RCL RCR >n	?	+	–	–	–	–
Восстановление флагов	POPF IRET	+	+	+	+	+	+
	SAHF	–	+	+	+	+	+
Управление флагом переноса	STC	–	1	–	–	–	–
	CLC	–	0	–	–	–	–
	CMC	–	#	–	–	–	–

Операции	Команды	Флаги управления		
		DF	IF	TF
Восстановление флагов	POPF IRET	+	+	+
Прерывания	INT INT0	–	0	0
Управление флагами	STD	1	–	–
	CLD	0	–	–
	STI	–	1	–
	CLI	–	0	–

Обозначения:

“+” – обозначает, что команда влияет на флаг;
“–” – не влияет на флаг;
“1” – флаг устанавливается в 1;
“0” – флаг устанавливается в 0;
“#” – инверсия флага;
“?” – неопределенное значение флага;
“>1” – одиночный сдвиг;
“>n” – многоразрядный сдвиг.

Таблица Б.5 – Примеры работы арифметических команд.

Состояние до команды	Команда/действие	Состояние после
CX=0049 SI=09EA	add SI, CX ; CF_SI←SI+CX	SI=0A33 OF,SF,ZF,CF=0 AF,PF=1
AX=F0F0 DI=3000 DS=1000 (13008)=B456	add AX,[DI+8] ; CF_AX←AX+m(DS: DI+8)	AX=A546 OF,ZF,AF,PF=0 SF,CF=1
CX=4567	add CX,0ABCDh ; CF_CX←CX+ABCDh	CX=F134 OF,ZF,PF,CF=0 SF,AF=1
AL=4E	add AL,0C5h ; CF_AL←AL+C5h	AL=13 OF,SF,ZF,PF=0 AF,CF=1
AX=89AB DI=0300 DS=1200 (12300)=3748 CF=1	add [DI],AX ; CF_m(DS:DI)←m(DS:DI)+AX	(12300)=C0F4 OF,ZF,PF,CF=0 SF,AF=1
DS=3000 CL=40h BX=0010 EA(Array)=3 byte(30013)=78h	add Array[BX], CL ; CF_m(DS: EA(Array)+BX) ← ← m(DS: EA(Array)+BX)+CL – сложить CL с десятым байтом массива Array, адресуемым регистром BX. Сумму занести в байт массива	CX=1000 BX=0010 EA(ARRAY)=3 (0013)=6233 ZF, CF, AF=0 OF, PF,SF=1
DI=FF00 DS=A000 (AFF00)=FF CF=0	inc [DI] ; m(DS:DI)←m(DS:DI)+1	(AFF00)=00 OF,SF,CF=0 ZF,AF,PF=1
CL=83 SS=1000 BP=FFFE (20006)=9A	sub CL, [BP+8] ; CF_CL←1_CL-m(SS:BP+8), CF←⌉CF	CL=E9 OF,ZF,PF=0 SF,AF,CF=1
SI=A000 DS=2500 (2F020)=3456	sub [SI+20h], 2AAh ;CF_m(DS:SI+20h)←1_m(DS:SI+20h)-2AAh; CF←⌉CF	[2F020]=31AC OF,SF,ZF,CF=0 AF,PF=1
AL=73	sub AL, 0B7h ; CF_AL←1_AL-B7h; CF←⌉CF	AL=BC ZF,PF=0 OF,SF,AF,CF=1
SI=6000 DS=2A00 (30000)=50 CF=1	dec [SI] ; m(DS:SI)←m(DS:SI)-1	DX=4F OF,SF,ZF,PF=0 AF,CF=1

Таблица Б.6 – Команды логических операций

Мнемоника	Действие
Команды логических операций	
AND d, s (logical AND, логическое И)	d ← d∧s Побитное логическое умножение операндов, результат в d
Возможные операнды команды AND и других логических команд: r1,r2; r,im; ac,im; r,m; m,r; m,im.	
OR d, s (logical OR, логическое ИЛИ)	d ← d∨s Логическое сложение побитно над операндами. Результат в d.
XOR d, s (Логическое исключающее ИЛИ, exclusive OR)	d ← d (exor) s Логическое исключающее ИЛИ, результат в d
NOT s (Побитное инвертирование операнда (invert each bit))	s ← ¬s Инвертирование всех битов операнда источника
TEST d, s (Логическое И)	d∧s Логическое побитное умножение d и s, результат нигде не запоминается, флажки реагируют в соответствии с табл.Б.4
Команды обработки бит	
Сканирования бит	
BSF d, s (Bit Scan Forward, побитное сканирование вперед)	Проверка наличия единичных битов в операнде источник начиная с бита 0. В d ← позиция первой единицы. Если все нули, то ZF=1
BSR d, s (Bit Scan Reverse, побитное сканирование назад)	Аналогично BSF, но проверяет со старшего бита
Проверки и модификации бит	
Двоичные цепочки (bit string), т.е. последовательности бит, как типы данных, широко применяются в компьютерной графике, для табличного представления распределения секторов дискового пространства и т.п. Команда проверки бита BT (Bit Test) выбирает определенный бит в двоичной цепочке и передает его значение во флажок переноса CF. Состояния других арифметических флажков после этой команды не определены. Очень похожие на нее команды BTC, BTS и BTR, после запоминания выбранного бита в CF, осуществляют еще его инвертирование, установку в 1 и сброс в 0. Поэтому ниже речь идет об одной «обобщенной» команде BTx. Ее обобщенный формат BTx s, index Операнд s задает двоичную цепочку в памяти или регистре (слово или двойное слово), а index выбирает конкретный бит в цепочке. Допустимые формы операндов команды BTx: BTx reg/mem16, reg16 BTx reg/mem32, reg32 BTx reg/mem16, im8 BTx reg/mem32, im8	
BT s, index	CF ← s(index)
BTC s, index (Bit Test and Complement, проверка бита с инверсией)	CF ← s(index), s(index) ← ¬s(index)

BTS s, index (Bit Test and Set, проверка бита с его установкой в 1)	CF ← s(index), s(index) ← 1
BTR s, index (Bit Test and Reset, проверка бита с его сбросом в 0)	CF ← s(index), s(index) ← 0

Команды сдвига

Команды сдвига перемещают биты в поле операнда влево или вправо, в зависимости от кода операции. Все команды сдвига имеют одинаковую структуру

```
OpCode d, cnt
```

Здесь `OpCode` – код операции (Operation Coge); `d` – сдвигаемый операнд, адресуемый стандартным образом; `cnt` (счет или счетчик, `count`) определяет число сдвигов.

Число сдвигов `cnt` может задаваться двумя способами:

статически – непосредственным числовым значением.

Например:

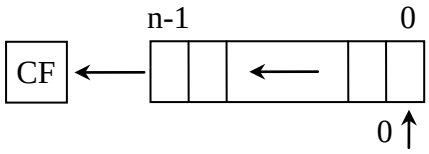
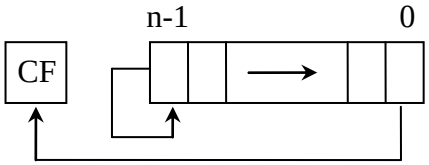
```
shl AH, 1;
```

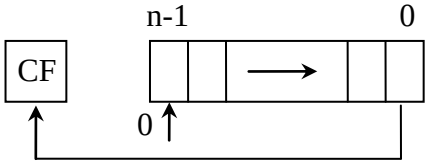
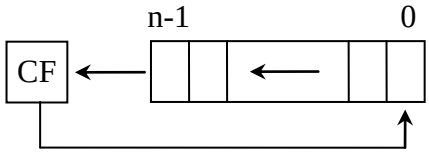
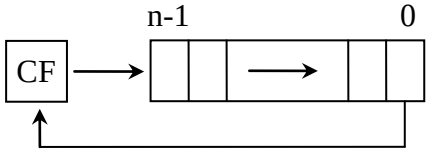
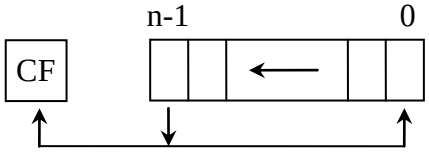
динамически – в регистре CL перед выполнением команды сдвига.

Например:

```
mov CL, 3
ror BX, CL
```

Линейный сдвиг

SAL d, cnt (Shift Arithmetic operand Left, Сдвиг арифметический операнда влево)	Арифметический сдвиг операнда d влево на cnt (count) позиций. CF принимает значение последнего вытесненного бита. В освободившиеся биты заносятся нули: 
В этой и всех других командах сдвига операнды: r,1; r,cl; m,1; m,cl.	
SAR d, cnt (Shift Arithmetic operand Right, Арифметический сдвиг операнда вправо)	Арифметический сдвиг операнда вправо на cnt (count) позиций. CF принимает значение последнего вытесненного бита. По мере сдвига вправо освобождающиеся места заполняются значением знакового разряда: 
SHL d, cnt (SHift logical Left, логический сдвиг операнда влево)	Аналогично команде SAL. Справа в операнд вдвигается cnt нулевых бит

SHR d, cnt (Логический сдвиг операнда вправо)	Слева в операнд вдвигается cnt нулевых бит: 
SHLD d, s, cnt (SHift Left Double word)	Сдвиг двойного слова влево. Пример: SHLD EBX, EAX, CL
Алгоритм работы SHLD: сдвинуть операнд приемник влево на cnt бит; сдвинуть операнд источник влево на cnt бит. Операнд источник только обеспечивает вдвигаемые в операнд приемник биты, сам он при этом не изменяется; выдвигаемые во время сдвига влево из операнда источник биты вдвигаются в операнд приемник с его правого края	
SHRD d, s, count (SHift Right Double word)	Сдвиг двойного слова вправо. Алгоритм как в SHLD, но для сдвига вправо
Циклический сдвиг	
RCL d, cnt (Rotate operand through Carry flag Left)	Сдвиг операнда влево (при этом старший бит становится младшим) cnt раз. (CF исп. как буфер для переноса): 
RCR d, cnt (Rotate operand through Carry flag Right)	Аналогично RCL, но вправо: 
ROL d, cnt (Rotate operand Left)	Циклический сдвиг операнда влево (старший бит становится младшим cnt раз). В CF копируется значение перенесенного бита: 
Отличие команд RCL и ROL: RCLиспользуется для циклического сдвига разрядов операнда влево. Этот сдвиг происходит с некоторой задержкой, так как очередной сдвигаемый бит оказывается на некоторое время вне операнда. В это время можно произвести его извлечение и (или) подмену	

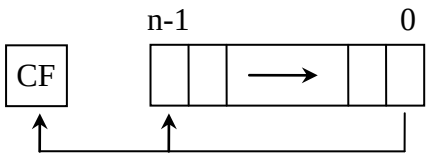
ROR im, cnt (Rotate operand Right)	Аналогично ROL, но вправо: 

Таблица Б.7 – Цепочечные команды пересылки

Под *цепочкой* (*строкой символов*) понимается последовательность любых контекстно-связанных (8-битных) байтов, (16-битных) слов или (32-битных) двойных слов, находящихся в смежных ячейках памяти. Коротко, цепочка – это одномерный массив данных. В системе команд Pentium имеется семь типов однобайтных команд, обрабатывающих 8-, 16- или 32-битные элементы цепочек:

команды пересылки цепочек MOVS d, s; MOVSB, MOVSW, MOVSD;
сравнения цепочек CMPS d,s; CMPSB, CMPSW, CMPSD;
сканирования цепочек SCAS d,s; SCASB, SCASW, SCASD;
загрузки элемента из цепочки LODS d,s; LODSB, LODSW, LODSD;
сохранения элемента в цепочке STOS d,s; STOSB, STOSW, STOSD;
ввода элементов цепочки из порта ввода-вывода INS d,s; INSB, INSW, INSD;
вывода элементов цепочки в порт ввода-вывода OUTS d,s; OUTSB, OUTSW, OUTSD.

Особенность всех цепочечных команд в том, что они, кроме обработки текущего элемента цепочки, осуществляют еще и автоматическое продвижение к следующему элементу данной цепочки.

Цепочечная команда сама по себе (без префикса повторения) выполняется только один раз, т.е. обрабатывает один элемент данных. Чтобы заставить команду выполниться в цикле – обработать всю цепочку, надо перед командой установить (в поле метки) префикс повторения. Благодаря такому префиксу цепочки данных обрабатываются значительно быстрее, чем при организации программного цикла. Число различных префиксов повторения три:

REP;
REPE или REPZ;
REPNE или REPNZ.

Ниже рассматриваются только цепочечные команды пересылки. Другие команды, а также префиксы, кроме REP, здесь не поясняются. О них можно прочитать в [1-4,8,9,18]

Мнемоника	Действие
Цепочечные команды пересылки Осуществляют пересылки данных из памяти в память (через процессор)	
MOVS d, s (передать цепочку (string) из s в d)	d ← s
MOVSB (передать байт)	m(ES:EDI/DI) ← m(DS: ESI/SI); if DF=0 then ESI/SI ← ESI/SI+1, EDI/DI ← EDI/DI+1; if DF=1 then ESI/SI ← ESI/SI-1, EDI/DI ← EDI/DI-1
MOVSW (передать слово)	то же, но ESI/SI и EDI/DI увеличиваются или уменьшаются на 2

MOVSD (передать двойное слово)	то же, но ESI/SI и EDI/DI увеличиваются или уменьшаются на 4
REP MOVSB (передать байт с повторением)	Repeat: m(ES: EDI/DI) ← m(DS: ESI/SI); if DF=0 then ESI/SI ← ESI/SI+1, EDI/DI ← EDI/DI+1; if DF=1 then ESI/SI ← ESI/SI-1, EDI/DI ← EDI/DI-1; CX ← CX-1, if CX≠0 then go to Repeat
REP MOVSW (передать слово с повторением)	то же, но ESI/SI и EDI/DI увеличиваются или уменьшаются на 2
REP MOVSD (передать двойное слово с повторением)	то же, но ESI/SI и EDI/DI увеличиваются или уменьшаются на 4

Цепочка-источник, адресуемая операндом s, может находиться в текущем сегменте данных, определяемом регистром DS. *Цепочка-приемник*, адресуемая операндом d, должна быть в дополнительном сегменте данных, адресуемом регистром ES. Допускается замена (с помощью префикса замены сегмента) только регистра DS, регистр ES подменять нельзя. Вторые части адресов – смещения цепочек – также находятся в строго определенных местах. Для *цепочки-источника* это регистр ESI/SI. Для *цепочки-получателя* это регистр EDI/DI. Таким образом, полные сегментированные адреса для операндов цепочечных команд следующие:

Адрес источника s – пара DS: ESI/SI; Адрес приемника d – пара ES: EDI/DI

Б.4. Команды передачи управления

Таблица Б.8 – Команды безусловного перехода

Мнемоника	Действие
Общее описание команд	
JMP [mod] Mark и JMP [mod] r/m — Безусловный переход (jump)	EIP/IP или CS: EIP/IP←Адрес перехода
Условие перехода: осуществляется всегда. Команда передает управление в место, определяемое операндами Mark (меткой) или r/m. **Метка – прямой мнемонический адрес перехода.** Mark – метка – это символическое имя, обозначающее то место (ту команду), куда необходимо передать управление. Транслируется ассемблером в адрес перехода. Поэтому метку можно считать мнемоническим адресом (мнемоадресом) перехода. Метка используется на языке ассемблера для *прямой* адресации перехода. Она устанавливается перед командой, на которую передается управление, и в адресной части самой команды передачи управления на ассемблере. [mod] – необязательный модификатор, определяющий тип перехода: SHORT, NEAR – прямой близкий переход, WORD PTR – косвенный близкий переход; или FAR – прямой далекий переход, DWORD PTR – косвенный далекий переход. **Косвенный адрес перехода** задается регистром или ячейкой в поле r/m. r/m – поле в мнемокоде команды передачи управления, указывающее на общий регистр процессора (r) или ячейку памяти (m), содержимое которых определяет адрес перехода. В случае m это поле может содержать также константу disp, прибавляемую к содержимому ячейки m. Этот способ используется на языке ассемблера для *косвенной* и *смешанной* (прямой - косвенной) адресации перехода. Транслятор вычисляет адрес перехода на основании значений операндов, записанных в поле r/m	

Какой конкретно регистр в записи EIP/IP будет использоваться, зависит от установленного режима адресации use16 или use32, определяемого битом D дескриптора (см. рис.2.8) [1-4,8,9,18]. Если указано use16 (D=0), то используется IP, если use32 (D=1) – то EIP. В реальном режиме процессора используется всегда IP

Близкие (внутрисегментные) переходы	
Прямой переход. Относительная адресация точки перехода	
JMP Mark JMP SHORT PTR Mark	Короткий (SHORT): EIP/IP ← EIP/IP+disp8
JMP Mark JMP NEAR PTR Mark	Близкий (NEAR): EIP/IP ← EIP/IP+disp16/32 (*)
Косвенный переход. Абсолютная адресация точки перехода	
JMP WORD PTR r	EIP/IP ← r
JMP WORD PTR m	EIP/IP ← m+[disp]
disp8/16/32 – сдвиговая добавка (displacement) – знаковое целое (8, 16 или 32-битное), записанное в дополнительном коде и размещенное в адресной части машинного кода команды перехода. Перед прибавлением знак disp расширяется до 32/16 бит (если длина disp короче). Значение EIP/IP в сумме (*) задает адрес команды, находящейся в программной памяти *сразу за* командой передачи управления. При вычислении нового адреса команды EIP/IP+disp возникающий перенос игнорируется, поэтому полученный адрес остаётся в пределах текущего сегмента кода	

Далекие (межсегментные) переходы Абсолютная адресация точки перехода	
Прямой переход	
JMP Mark JMP FAR PTR Mark	EIP/IP ← Новое смещение (Ofs) CS ← Новый селектор (Sel)
Новый селектор : Новое смещение – это сегментированный адрес перехода – адрес команды с меткой Mark	
Косвенный переход	
JMP DWORD PTR r	CS:EIP/IP ← r
JMP DWORD PTR m [disp]	CS:EIP/IP ← m+[disp]

Таблица Б.9 – Примеры команд безусловного перехода

Мнемоника	Действие
Прямой переход	
M1: mov AX, Var ; здесь M1 – это метка (Mark) ... jmp M1 ; после этой команды будет выполнена команда mov AX, Var	*Близкий* переход EIP/IP ← Ofs(M1)=EIP/IP+disp, где Ofs(M1) – смещение метки M1 (команды mov AX, Var) в этом сегменте. *Далекий* переход: EIP/IP ← Ofs(M1) CS ← Sel(M1) где Sel(M1) – новое значение селектора, определяемое меткой M1, т.е. селектор другого сегмента, в котором находится помеченная команда mov AX, Var; Ofs(M1) – смещение метки M1 в этом сегменте
Косвенный переход	
Близкий	
jmp BX	IP ← BX
jmp EBX	EIP ← EBX
jmp WORD PTR [BX]	IP ← m(DS:BX)
jmp WORD PTR ES:[BX+7]	IP ← m(ES:BX+7)
jmp WORD PTR Var[SI+7]	IP ← m(EA(Var)+SI+7)
Далекий	
jmp DWORD PTR [BX]	IP ← m(DS:BX) CS ← m(DS:BX+2)

Таблица Б.9 – Команды условного перехода (ветвления) J*c*

Типы операндов	Мнемокод	Критерий условного перехода	Значения флагов для осуществления перехода
Любые	JE/JZ	im_1 = im_2	ZF = 1
Любые	JNE/JNZ	im_1 ≠ im_2	ZF = 0
Со знаком	JL/JNGE	im_1 < im_2	SF ≠ OF
Со знаком	JLE/JNG	im_1 ≤ im_2	SF ≠ OF or ZF = 1
Со знаком	JG/JNLE	im_1 > im_2	SF = OF and ZF = 0
Со знаком	JGE/JNL	im_1 ≥ im_2	SF = OF
Без знака	JB/JNAE	im_1 < im_2	CF = 1
Без знака	JBE/JNA	im_1 ≤ im_2	CF = 1 or ZF=1
Без знака	JA/JNBE	im_1 > im_2	CF = 0 and ZF = 0
Без знака	JAE/JNB	im_1 ≥ im_2	CF = 0

Таблица Б.10 – Значение аббревиатур в названии команды J*c*

Мнемоническое обозначение	Английский	Русский	Тип операндов
E e	equal	Равно	Любые
N n	not	Не	Любые
G g	greater	Больше	Числа со знаком
L l	less	Меньше	Числа со знаком
A a	above	Выше, в смысле “больше”	Числа без знака
B b	below	Ниже, в смысле “меньше”	Числа без знака

Таблица Б.11 – Флаги

Название флага	Номер бита в EFLAGS/FLAGS	Команда условного перехода	Значение флага для осуществления перехода
Флаг переноса CF	1	JC	CF = 1
Флаг четности PF	2	JP/JPE	PF = 1
Флаг нуля ZF	6	JZ	ZF = 1
Флаг знака SF	7	JS	SF = 1
Флаг переполнения OF	11	JO	OF = 1
Флаг переноса CF	1	JNC	CF = 0
Флаг четности PF	2	JNP/JPO	PF = 0
Флаг нуля ZF	6	JNZ	ZF = 0
Флаг знака SF	7	JNS	SF = 0
Флаг переполнения OF	11	JNO	OF = 0

Таблица Б.12 – Команды вызова подпрограмм

Мнемоника	Действие
Общее описание команд вызова подпрограмм	
CALL [mod] Mark Вызов подпрограммы и CALL [mod] r/m	Стек ← Адрес возврата = [CS:] EIP/IP – это адрес команды, непосредственно следующей за CALL [CS:] EIP/IP ← Адрес перехода (формируется аналогично JMP)

Таблица Б.13 – Команды возврата из подпрограмм

Мнемоника	Действие
Команды возврата из подпрограмм	
RET (Возврат из подпрограммы)	[CS:]EIP/IP ← Адрес возврата из стека
При близком возврате (NEAR): EIP/IP ← Стек; При далеком (FAR): CS:EIP/IP ← Стек	
Таким образом, RET передает управление из вызванной подпрограммы команде, следовавшей непосредственно за CALL (возвращает управление «на после CALL»)	

Таблица Б.14 – Команды вызова подпрограмм прерываний

Мнемоника		Действие
Общее описание команд вызова прерываний		
INT *n* INT 3 INTO	Вызов подпрограммы обработки прерывания («обработчика прерывания»)	Стек ← Флаги и Адрес возврата, т.е. EFlags/Flags, CS и EIP/IP, где EIP/IP – внутрисегментный адрес команды после INT, CS – селектор сегмента, в котором находится INT; CS:EIP/IP ← Адрес вызова, т.е. CS ← Selector, EIP/IP ← Offset, где Адрес вызова = m(4×*n*) в реальном и V86 режимах, Адрес вызова = m(GDTR+8×*n*) в защищенном режиме
Условие вызова по командам INT *n*, INT 3: осуществляется всегда. Условие вызова по команде INTO: осуществляется в том случае, если установлен флажок переполнения OF; в противном случае, если OF=0, вызов не производится, а выполняется следующая за INTO команда. Сегментированный адрес вызова (Новый селектор : Новое смещение, Sel : Ofs), называемый вектором прерывания, считывается процессором из таблицы прерываний по адресу 4×*n* в реальном и V86 режимах и по адресу GDTR+8×*n* в защищенном режиме. Вызываемая команда должна входить в состав подпрограммы обработки прерывания. Она называется точкой входа в подпрограмму и не обязана быть первой по порядку командой ПП		

Таблица Б.15 – Команды возврата из подпрограмм прерываний

Мнемоника	Действие
Команды возврата из прерываний	
IRET (Возврат из подпрограммы прерывания)	CS:EIP/IP и EFlags/Flags ← Адрес возврата и старые флаги из стека
Передает управление из вызванной подпрограммы команде, следовавшей непосредственно за командой INT или за той командой, перед которой произошло внешнее аппаратное прерывание, или на ту команду, на которой произошло внутреннее аппаратное прерывание	

Таблица Б.16 – Команды циклов

Команды управления циклами. Три команды управления циклами (или итерациями) LOOP, LOOPNE и LOOPNZ применяются для организации программных циклов. В них предусматривается использование регистра ECX или CX в качестве счетчика цикла. Эти команды используют относительную адресацию перехода	
Мнемоника	Действие
LOOP short-label Переход по счетчику	CX ← CX – 1; if (CX <> 0) JMP short-label
LOOPE short-label Переход пока равно	CX ← CX – 1; if (CX <> 0) and (ZF = 1) JMP short-label
Команда LOOPE используется после команд CMP или SUB. Она уменьшает CX на 1, затем передает управление по метке short-label, если CX не равно нулю и если первый операнд команд CMP или SUB был равен второму операнду	
LOOPNE short-label Переход пока не равно	CX ← CX – 1; if (CX <> 0) and (ZF = 0) JMP short-label
Команда LOOPNE используется после команд CMP или SUB. Она уменьшает CX на 1, затем передает управление по метке short-label, если CX не равно нулю и если первый операнд команд CMP или SUB не равен второму операнду	
LOOPNZ short-label Переход пока не ноль	LOOPNZ – синоним LOOPNE
LOOPZ short-label Переход пока ноль	LOOPZ – синоним LOOPE
Примечание: недостаток команд организации цикла loop, loope/loopz и loopne/loopnz в том, что они реализуют только короткие переходы (от –128 до +127 байт). Для работы с длинными циклами придется использовать команды условного перехода и команду jmp	

Б.5. Команды изменения/проверки состояния процессора

Таблица Б.17 – Команды изменения состояния процессора

Мнемоника		Действие
CLC	(Clear CF)	Сбросить флажок CF в ноль
CMC	(Complement CF)	Инвертировать флажок
STC	(Set CF)	Установить флажок CF в единицу
CLD	(Clear DF)	Сбросить флажок DF в ноль
STD	(Set DF)	Установить флажок DF в единицу
CLI	(Clear IF)	Сбросить флажок IF в ноль
STI	(Set IF)	Установить флажок IF в единицу
HLT	(Halt)	Остановить процессор
WAIT	(Wait)	Ожидать внешнего события
LOCK	(Lock)	Выдать сигнал #LOCK
ESC	(Escape)	Переключиться на FPU – блок с плавающей точкой
NOP	(No Operation)	Команда пустой операции

Команды первой группы воздействуют на флажки, и смысл этого воздействия понятен из назначения флажков, описанного в п.2.5. Команды второй группы управляют другими функциями процессора.

Команда HLT останова заставляет процессор перейти в состояние останова. Из этого состояния процессор может быть выведен или сигналом сброса RESET, или аппаратным прерыванием на входе NMI, или запросом прерывания на входе INTR, если эти прерывания разрешены. При этом возврат из прерываний происходит на следующую за HLT команду. Данную команду можно использовать вместо бесконечного программного цикла, когда программа должна ожидать внешнего прерывания.

Команда WAIT ожидания переводит процессор в состояние ожидания, в котором он периодически через определенное количество тактов проверяет сигнал на входной линии BUSY. При появлении на ней активного уровня процессор переходит к выполнению следующей за WAIT команды. Из состояния ожидания процессор может быть выведен также сигналами RESET, NMI и INTR. В случае прерывания во время WAIT возврат из подпрограммы происходит снова на WAIT.

Команда LOCK, называемая также префиксом блокировки, заставляет процессор (работающий в максимальном режиме) выдать сигнал #LOCK на время выполнения следующей за префиксом команды. Этот сигнал блокирует запросы доступа к шине других подсистем. Префикс LOCK может находиться перед любой командой.

Команда ESC представляет собой средство, с помощью которого блок FPU арифметики с плавающей точкой может получать предназначенные для него команды и операнды.

Команда NOP (нет операции) не производит никаких действий. Она может применяться для удаления из программы ненужных байтов, а также в программных циклах задержки

Команды просмотра селекторов и дескрипторов сегментов. При выполнении программы можно просмотреть фактические значения селекторов и дескрипторов сегментов, которые редактор связей или загрузчик назначил сегментам. По селектору можно узнать, какую дескрипторную таблицу (бит TI) и какой дескриптор сегмента (поле IIndex) он выбирает. Имея права доступа к дескрипторным таблицам, можно даже считать и декодировать дескриптор. [9]

Чтобы упростить анализ селекторов, в процессоре IA-32 предусмотрено несколько специальных команд, сообщающих важную информацию. Команда LAR загрузки прав доступа (Load Access Rights) имеет следующий формат:

lar r16, r/m16

lar r32, r/m32

Она загружает в регистр-получатель (регистр общего назначения) права доступа селектора, находящегося в операнде-источнике r/m16 или r/m32. Эта команда отыскивает соответствующий селектору дескриптор и возвращает байт прав доступа AR и биты G, D, X и U (расширение прав).

Если операнд 16-битный, из дескриптора извлекаются биты 32…47, которые помещаются в 16-разрядный операнд-приемник, и выполняется операция

приемник ← приемник ∧ 00FF00h

Если операнд 32-битный, из дескриптора извлекаются биты 32…61, которые помещаются в 32-разрядный операнд-приемник, и выполняется операция

приемник ← приемник ∧ 00F?FF00h

(тетрада «?» не определена).

Другими словами, команда LAR сообщает о сегменте все, кроме базового адреса и предела. Выделяя нужные биты, можно определить, доступен ли данный сегмент программе, не вызывая нарушения защиты.

Предел сегмента сообщает команда загрузки предела сегмента LSL (Load Segment Limit) со следующим форматом

lsl r16, r/m16

lsl r32, r/m32

В 16- или 32-битный регистр-получатель загружается предел сегмента, который определяется селектором, находящемся в операнде-источнике r/m16 или r/m32. Если в дескрипторе бит гранулярности G=1 (т.е. сегмент имеет страничную гранулярность), предел расширяется на байтную гранулярность. Поэтому при использовании сегментов со страничной гранулярностью получателем следует указывать 32-битный регистр.

Еще две команды позволяют узнать, допускает ли сегмент, определяемый селектором, операции считывания или записи. Команды проверки на считывание VERR (VERify for Read) и запись VERW (VERify for Write) имеют одинаковый формат:

verr r/m16

verw r/m16

Операнд r/m16 задает в регистре или памяти селектор проверяемого сегмента. Обе эти команды устанавливают флажок ZF=1, если программа может загрузить этот селектор в сегментный регистр данных DS, ES, FS или GS, а также считывать (записывать) в сегмент без нарушения защиты. Отметим, что обе эти команды не контролируют бит P присутствия в дескрипторе.

При выполнении приведенных выше команд процессор учитывает уровень привилегий текущей программы и тех сегментов, которые определяются проверяемыми селекторами. При нарушении правил привилегий он может не возвратить нужной информации.

Команды работы с регистрами, управляющими режимами функционирования процессора, – системными регистрами не рассматриваются.

У ч е б н о е и з д а н и е

Козырев Владимир Георгиевич

АРХИТЕКТУРА КОМПЬЮТЕРНЫХ СИСТЕМ

Учебное пособие

Редактор – О.В. Дмитриева

Изд. № 102/17. Объем 15,75 п.л. Усл. печ. л. 14,65. Уч.-изд.л. 15,435.
РИИЦМ ФГАОУ ВО «Севастопольский государственный университет»