Архитектура микропроцессорных систем

Лекция 1. Микропроцессор и его архитектура

Основные понятия и характеристики архитектуры микропроцессоров

Микропроцессор (МП) - это программно *управляемое устройство*, которое предназначено для обработки *цифровой информации* и управления процессом этой обработки и выполнено в виде одной или нескольких больших *интегральных схем* (БИС).

Понятие **большая интегральная схема** в настоящее время четко не определено. Ранее считалось, что к этому классу следует относить микросхемы, содержащие более 1000 элементов на кристалле. И действительно, в эти параметры укладывались первые микропроцессоры. Например, 4-разрядная процессорная секция микропроцессорного комплекта К584, выпускавшегося в конце 1970-х годов, содержала около 1500 элементов. Сейчас, когда микропроцессоры содержат десятки миллионов *транзисторов* и их количество непрерывно увеличивается, под БИС будем понимать функционально сложную *интегральную схему*.

Микропроцессорная система (МПС) представляет собой функционально законченное изделие, состоящее из одного или нескольких устройств, основу которой составляет *микропроцессор*.

Микропроцессор характеризуется большим количеством параметров и свойств, так как он является, с одной стороны, функционально сложным вычислительным устройством, а с другой - электронным прибором, изделием электронной промышленности. Как средство вычислительной техники он характеризуется прежде всего своей **архитектурой**, то есть совокупностью программно-аппаратных свойств, предоставляемых пользователю. Сюда относятся *система команд*, типы и форматы обрабатываемых данных, режимы адресации, количество и распределение регистров, принципы взаимодействия с оперативной памятью и внешними устройствами (характеристики системы прерываний, *прямой доступ* к памяти и т. д.). *По* своей архитектуре микропроцессоры разделяются на несколько типов (рис. 1.1).

Универсальные микропроцессоры предназначены для решения задач цифровой обработки различного *типа информации* от инженерных расчетов до работы с базами данных, не связанных жесткими ограничениями на *время выполнения* задания. Этот *класс* микропроцессоров наиболее широко известен. К нему относятся такие известные микропроцессоры, как МП ряда *Pentium* фирмы Intel и МП семейства Athlon фирмы *AMD*.

Рис. 1.1. Классификация микропроцессоров

Характеристики универсальных микропроцессоров:

- разрядность: определяется максимальной разрядностью целочисленных данных, обрабатываемых за 1 такт, то есть фактически разрядностью *арифметико-логического устройства* (*АЛУ*);
- виды и форматы обрабатываемых данных;
- система команд, режимы адресации операндов;
- емкость прямоадресуемой оперативной памяти: определяется разрядностью *шины адреса*;
- частота внешней синхронизации. Для частоты синхронизации обычно указывается ее максимально возможное значение, при котором гарантируется работоспособность схемы. Для функционально сложных схем, к которым относятся и микропроцессоры, иногда указывают также минимально возможную частоту синхронизации. Уменьшение частоты ниже этого предела может привести к отказу схемы. В то же время в тех применениях МП, где не требуется высокое быстродействие, снижение частоты синхронизации - одно из направлений энергосбережения. В ряде современных микропроцессоров при уменьшении частоты он переходит в *<спящий режим>*, при котором сохраняет свое состояние. Частота синхронизации в рамках одной архитектуры позволяет сравнить производительность микропроцессоров. Но разные архитектурные решения влияют на производительность гораздо больше, чем частота;
- производительность: определяется с помощью специальных тестов, при этом совокупность тестов подбирается таким образом, чтобы они по возможности покрывали различные характеристики микроархитектуры процессоров, влияющие на производительность.

Универсальные микропроцессоры принято разделять на **CISC** - и **RISC-микропроцессоры**. **CISC-микропроцессоры** (Completed *Instruction Set Computing* - вычисления с полной системой команд) имеют в своем составе весь классический набор команд с широко развитыми режимами адресации операндов. Именно к этому классу относятся, например, микро процессоры типа *Pentium*. В то же время **RISC-микропроцессоры** (*reduced instruction set computing* - вычисления

с сокращенной системой команд) используют, как следует из определения, уменьшенное количество команд и режимов адресации. Здесь прежде всего следует выделить такие микропроцессоры, как *Alpha* 21x64, *Power PC*. Количество команд в системе команд - наиболее очевидное, но на сегодняшний день не самое главное различие в этих направлениях развития универсальных микропроцессоров. Другие различия мы будем рассматривать *по* мере изучения особенностей их архитектуры.

Однокристальные микроконтроллеры (ОМК или просто МК) предназначены для использования в системах промышленной и бытовой автоматики. Они представляют собой большие интегральные схемы, которые включают в себя все устройства, необходимые для реализации цифровой системы управления минимальной конфигурации: *процессор* (как правило, целочисленный), ЗУ команд, ЗУ данных, *генератор* тактовых сигналов, программируемые устройства для связи с внешней средой (*контроллер прерывания*, таймеры-счетчики, разнообразные порты ввода/вывода), иногда аналого-цифровые и цифро-аналоговые преобразователи и т. д. В некоторых источниках этот *класс* микропроцессоров называется однокристальными микро-ЭВМ (ОМЭВМ).

В настоящее время две трети всех производимых микропроцессорных БИС в мире составляют МП этого класса, причем почти две трети из них имеет *разрядность*, не превышающую 16 *бит*. К классу однокристальных *микроконтроллеров* прежде всего относятся микропроцессоры серии *MCS*-51 фирмы Intel и аналогичные микропроцессоры других производителей, *архитектура*которых де-факто стала стандартом.

Отличительные особенности архитектуры однокристальных микроконтроллеров:

- физическое и логическое разделение памяти команд и памяти данных (гарвардская архитектура), в то время как в классической неймановской архитектуре программы и данные находятся в общем *запоминающем устройстве* и имеют одинаковый механизм доступа;
- упрощенная и ориентированная на задачи управления система команд: в МК, как правило, отсутствуют средства обработки данных с плавающей точкой, но в то же время в систему команд входят команды, ориентированные на эффективную работу с датчиками и исполнительными устройствами, например, команды обработки битовой информации;
- простейшие режимы адресации операндов.

Основные характеристики микроконтроллеров (в качестве примера численные значения представлены для МК-51):

1. Разрядность (8 бит).
2. Емкость внутренней памяти команд и памяти данных, возможности и пределы их расширения:

- внутренняя память команд - 4 Кбайт (в среднем команда имеет длину 2 байта, таким образом, во внутренней памяти может быть размещена программа длиной около 2000 команд); возможность наращивания за счет подключения внешней памяти до 64 Кбайт;
- *память данных* на кристалле 128 байт (можно подключить *внешнюю память* общей емкостью до 64 Кбайт).

3. *Тактовая частота*:
   - внешняя частота 12 МГц;
   - частота машинного цикла 1 МГц.
4. Возможности взаимодействия с внешними устройствами: количество и назначение *портов ввода-вывода*, характеристики системы прерывания, программная поддержка взаимодействия с внешними устройствами.

Наличие и характеристики встроенных *аналого-цифровых преобразователей* (*АЦП*) и цифро-аналоговых преобразователей (*ЦАП*) для упрощения согласования с датчиками и исполнительными устройствами системы управления.

Секционированные микропроцессоры (другие названия: микропрограммируемые и разрядно-модульные) - это микропроцессоры, предназначенные для построения специализированных процессоров. Они представляют собой микропроцессорные секции относительно небольшой (от 2 до 16) разрядности с пользовательским доступом к микропрограммному уровню управления и средствами для объединения нескольких секций.

Такая организация позволяет спроектировать *процессор* необходимой разрядности и со специализированной системой команд. Из-за своей малой разрядности микропроцессорные секции могут быть построены с использованием быстродействующих технологий. Совокупность всех этих факторов обеспечивает возможность создания процессора, наилучшим образом ориентированного на заданный *класс* алгоритмов как *по* системе команд и режимам адресации, так и *по* форматам данных.

Одним из первых комплектов секционированных микропроцессоров были МП БИС семейства Intel 3000. В нашей стране они выпускались в составе серии К589 и 585. *Процессорные элементы* этой серии представляли собой двухразрядный *микропроцессор*. Наиболее распространенным комплектом секционированных микропроцессоров является Am2900, основу которого составляют 4-разрядные секции. В нашей стране аналог этого комплекта выпускался в составе серии К1804. В состав комплекта входили следующие БИС:

- разрядное секционное *АЛУ*;
- блок ускоренного переноса;
- разрядное секционное *АЛУ* с аппаратной поддержкой умножения;
- тип схем микропрограммного управления;
- контроллер состояния и сдвига;

- *контроллер приоритетных прерываний*.

Основным недостатком микропроцессорных систем на базе секционированных микропроцессорных БИС явилась сложность проектирования, отладки и программирования систем на их основе. Использование специализированной системы команд приводило к несовместимости разрабатываемого *ПО* для различных микропроцессоров. Возможность создания оптимального *по*многим параметрам специализированного процессора требовала труда квалифицированных разработчиков на протяжении длительного времени. Однако бурное развитие электронных технологий привело к тому, что за время проектирования специализированного процессора разрабатывался универсальный *микропроцессор,* возможности которого перекрывали гипотетический выигрыш от проектирования специализированного устройства. Это привело к тому, что в настоящее время данный *класс* микропроцессорных БИС практически не используется.

Процессоры цифровой обработки сигналов, или **цифровые сигнальные процессоры**, представляют собой бурно развивающийся *класс* микропроцессоров, предназначенных для решения задач цифровой *обработки сигналов* - обработки звуковых сигналов, изображений, распознавания образов и т. д. Они включают в себя многие черты однокристальных микро-контроллеров: гарвардскую архитектуру, встроенную *память* команд и данных, развитые возможности работы с внешними устройствами. В то же время в них присутствуют черты и универсальных МП, особенно с *RISC*-архитектурой: конвейерная организация работы, программные и *аппаратные средства* для выполнения операций с *плавающей запятой*, аппаратная *поддержка* сложных специализированных вычислений, особенно умножения.

Как электронное изделие *микропроцессор* характеризуется рядом параметров, наиболее важными из которых являются следующие:

1. Требования к синхронизации: максимальная частота, стабильность.
2. Количество и номиналы источников питания, требования к их стабильности. В настоящее время существует тенденция к уменьшению напряжения питания, что сокращает тепловыделение схемы и ведет к повышению частоты ее работы. Если первые микропроцессоры работали при напряжении питания+-15В, то сейчас отдельные схемы используют источники менее 1 В.
3. **Мощность рассеяния** - это мощность потерь в выходном каскаде схемы, превращающаяся в тепло и нагревающая выходные транзисторы. Иначе говоря, она характеризует показатель тепловыделения БИС, что во многом определяет требования к конструктивному оформлению *микропроцессорной системы*. Эта характеристика особенно важна для встраиваемых МПС.
4. Уровни сигналов логического нуля и логической единицы, которые связаны с номиналами источников питания.
5. Тип корпуса - позволяет оценить пригодность схемы для работы в тех или иных условиях, а также возможность использования новой БИС в качестве замены существующей на плате.

6. Температура окружающей среды, при которой может работать схема. Здесь выделяют два диапазона:
   - коммерческий (0 ^{0}C … +70^{0}C);
   - расширенный (-40 ^{0}C … +85 ^{0}C).
7. **Помехоустойчивость** - определяет способность схемы выполнять свои функции при наличии помех. Помехоустойчивость оценивается интенсивностью помех, при которых нарушение функций устройства еще не превышает допустимых пределов. Чем сильнее помеха, при которой устройство остается работоспособным, тем выше его помехоустойчивость.
8. **Нагрузочная способность**, или коэффициент разветвления по выходу, определяется числом схем этой же серии, входы которых могут быть присоединены к выходу данной схемы без нарушения ее работоспособности. Чем выше нагрузочная способность, тем шире логические возможности схемы и тем меньше таких микросхем необходимо для построения сложного вычислительного устройства. Однако с увеличением этого коэффициента ухудшаются помехоустойчивость и быстродействие.
9. **Надежность** - это способность схемы сохранять свой уровень качества функционирования при установленных условиях за установленный период времени. Обычно характеризуется *интенсивностью отказов* (час-1) или средним временем наработки на отказ (час). В настоящее время этот параметр для больших интегральных схем обычно не указывается изготовителем. О надежности МП БИС можно судить по косвенным показателям, например, по приводимой разработчиками средств вычислительной техники надежности изделия в целом.
10. **Характеристики технологического процесса**. Основной показатель здесь - разрешающая способность процесса. ***В настоящее время (сентябрь 2017 г.) она достигает 7 – 14 - 32 нм, то есть около 140-70- 30 тыс. линий на 1 мм. Более совершенный технологический процесс позволяет создать микропроцессор, обладающий большими функциональными возможностями.***

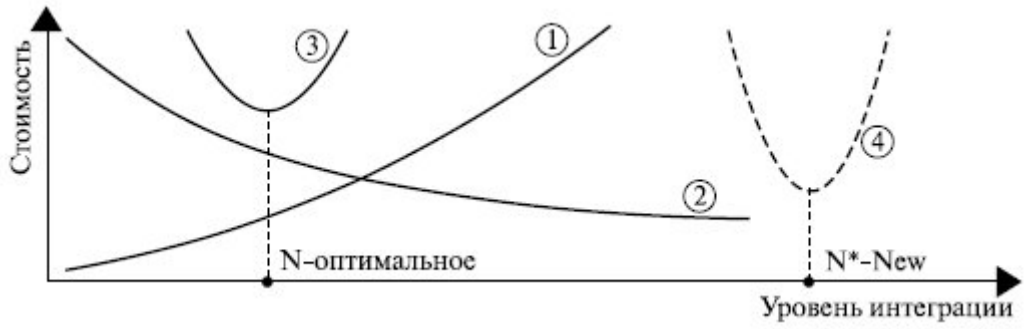

Рис. 1.2. Затраты на производство микропроцессорной системы

Затраты на изготовление устройств, использующих микропроцессорные БИС, представлены на рис. 1.2. Здесь:

1. затраты на изготовление БИС (чем больше степень интеграции элементов на кристалле, тем дороже обходится производство схемы);
2. затраты на сборку и наладку *микропроцессорной системы* (с увеличением функциональных возможностей МП потребуется меньше схем для создания МПС);
3. общая стоимость *микропроцессорной системы*, которая складывается из затрат (1) и (2). Она имеет некоторое оптимальное значение для данного уровня развития технологии;
4. переход на новую технологию (оптимальным будет уже другое количество элементов на кристалле, а общая стоимость изделия снижается).

В 1965 году Гордон Мур сформулировал гипотезу, известную в настоящее время как <закон Мура>, согласно которой каждые 1,5-2 года число *транзисторов* в расчете на одну *интегральную схему* будет удваиваться. Это обеспечивается непрерывным совершенствованием технологических процессов производства микросхем.

Наиболее развитая в технологическом отношении *фирма* Intel в жизненном цикле полупроводниковых технологий, создаваемых и применяемых в корпорации, выделяет шесть стадий.

Самая ранняя стадия проходит за пределами Intel - в университетских лабораториях и независимых исследовательских центрах, где ведутся поиски новых физических принципов и методов, которые могут стать основой научно-технологического задела на годы вперед. *Корпорация* финансирует эти исследования.

На второй стадии исследователи Intel выбирают наиболее перспективные направления развития новых технологий. При этом обычно рассматривается 2-3 варианта решения.

Главная задача третьей стадии - полная черновая проработка новой технологии и демонстрация ее осуществимости.

После этого начинается четвертая стадия, главная цель которой - обеспечить достижение заданных значений таких ключевых технических и экономических показателей, как *выход* годных изделий, *надежность, стоимость* и некоторые другие. Завершение этапа подтверждается выпуском первой промышленной партии новых изделий.

Пятая стадия - промышленное освоение новой технологии. Эта проблема не менее сложна, чем разработка самой технологии, поскольку необычайно трудно в точности воспроизвести в условиях реального производства то, что было получено в лаборатории. Обычно именно здесь возникают задержки со сроками выпуска новых изделий, с достижением запланированного объема поставок и себестоимости продукции.

Последняя, шестая стадия жизненного *цикла* технологии (перед отказом от ее применения) - зрелость. Зрелая технология, подвергаясь определенному совершенствованию с целью повышения производительности оборудования и снижения себестоимости продукции, обеспечивает основные объемы производства. *По* мере внедрения новых, более совершенных технологий <старые> производства ликвидируются.

Но не сразу: сначала они переводятся на выпуск микросхем с меньшим быстродействием или с меньшим числом *транзисторов*, например, периферийных БИС.

Этапы развития архитектуры универсальных микропроцессоров

Первый *микропроцессор* был разработан фирмой Intel в 1971 году. Он получил название I-4004, имел 4-разрядную структуру и был ориентирован на использование в калькуляторах. Впоследствии этой же фирмой был выпущен еще один 4-разрядный *микропроцессор* - I-4040.

На протяжении многих лет крупнейшими разработчиками и производителями универсальных микропроцессоров в мире являются компании Intel (70-75 % мирового производства) и *AdvancedMicro* Devices (*AMD*), занимающая 20-25 % рынка. Их разработки идут во многом параллельными путями. В нашем курсе мы будем рассматривать развитие архитектуры универсальных микропроцессоров на примере микропроцессоров фирмы Intel.

В 1972 году на рынке появился 8-разрядный МП I-8008, а вслед за ним, в **1974 году,- I-8080.** Последний *микропроцессор* сыграл значительную роль в развитии микропроцессорной техники. Во многом он заложил основы архитектуры для всех последующих поколений микропроцессоров. Он имеет раздельные 8-разрядную *шину данных* и 16-разрядную *шину адреса*, возможность подключения памяти емкостью до 64 Кбайт и до 256 внешних устройств. *Микропроцессор* содержит 16-разрядные **указатель команд** (*Instruction Pointer - IP*) и **указатель стека** (*Stack Pointer - SP*), шесть 8-разрядных регистров общего назначения (РОН), которые могут использоваться как три 16-разрядные. *Система команд* состоит из 78 базовых команд. При загрузке операнда из памяти применяется прямая, косвенная регистровая или стековая *адресация*. В общем случае программист может использовать регистровую, прямую, косвенную, непосредственную, индексную, прямую и косвенную автоинкрементную и автодекрементную адресации.

Микропроцессор содержит входные и выходные интерфейсные сигналы, обеспечивающие реакцию на сигналы запросов внешних прерываний, организацию прямого доступа к памяти, а также согласование своего *цикла работы* с медленными внешними устройствами (ВУ).

Его отличительной чертой стало создание **микропроцессорного комплекта** или семейства, то есть набора БИС, совместимых между собой *по* интерфейсным

сигналам и функционально дополняющих друг друга. В нашей стране этот микропроцессорный комплект выпускался в составе серии К580, в которую вошли следующие микросхемы:

- КР580ВМ80А - однокристальный 8-разрядный микропроцессор;
- КР580ВВ51А - программируемый *последовательный интерфейс*;
- КР580ВИ53 - программируемый таймер;
- КР580ВВ55А - программируемый параллельный интерфейс;
- КР580ВТ57 - контроллер прямого доступа к памяти;
- КР580ВН59 - *контроллер прерываний*;
- КР580ВВ79 - интерфейс клавиатуры и дисплея;
- КР580ВГ75 - контроллер ЭЛТ;
- КР580ВК91А - интерфейс МП - канал общего пользования;
- КР580ГФ24 - генератор тактовых сигналов и некоторые другие схемы, предназначенные в основном для согласования работы отдельных частей *микропроцессорной системы*.

БИС данного микропроцессорного комплекта вследствие хороших архитектурных решений, широкой номенклатуры и совместимости до сих пор можно встретить в некоторых цифровых устройствах, не требующих высокого быстродействия и разрядности, а идеи, заложенные в таких схемах, как *контроллер прерываний* и *контроллер* прямого доступа к памяти, используются в современных наборах системной логики - чипсетах.

Очередным крупным шагом в развитии микропроцессорной техники стало появление в 1978 году 16-разрядных универсальных микропроцессоров. Здесь прежде всего следует выделить *микропроцессор* I-8086,

выпускавшийся отечественной электронной промышленностью в составе семейства К1810. Эти микропроцессоры, заложившие основы архитектуры *x86,* использовались при производстве первых *персональных ЭВМ*.

Основными отличительными чертами в архитектуре этого микропроцессора стали:

- увеличение разрядности регистров общего назначения до 16 бит;
- увеличение количества регистров общего назначения до 8;
- увеличение количества режимов адресации операндов;
- расширение количества флагов в регистре признаков, в том числе за счет введения флагов управления, обеспечивающих, например, возможность запрета внешних *маскируемых прерываний*;
- появление сегментного механизма обращения к памяти, который обеспечил возможность обращения к памяти емкостью до 1 Мбайт при использовании 16-разрядных регистров.

Появившийся вслед за этим в 1982 году *микропроцессор* i286 явился переходной ступенью к 32-разрядным универсальным микропроцессорам. В процессоре i286 было реализовано два режима работы - защищенный и реальный. В **реальном режиме** работы *процессор* был полностью совместим с выпускавшимися ранее 16-разрядными микропроцессорами с архитектурой *x86*. В формировании адреса участвовали только 20 линий, поэтому максимальная емкость адресуемой памяти в этом режиме осталась прежней - 1 Мбайт. В **защищенном режиме** *процессор* мог адресовать до 1 Гбайт виртуальной памяти. *Шина* адреса была увеличена до 24 *бит*, поэтому емкость адресуемой памяти составляла 16 Мбайт. Для защиты от несанкционированного доступа к программам и данным и выполнения привилегированных команд, которые могут кардинально изменить состояние всей системы, в процессоре i286 была введена защита *по* привилегиям. С этой целью *микропроцессор* поддерживал 4 уровня привилегий. Для выполнения операций над числами с плавающей точкой была разработана отдельная БИС - *математический сопроцессор* 80287.

В 1985 году был выпущен 32-разрядный универсальный *микропроцессор* i386 - первый полноценный представитель архитектуры *IA-32* (Intel *Architecture*-32). Развитие этой архитектуры продолжалось вплоть до последних моделей микропроцессора *Pentium* 4. Данную архитектуру отличает ряд изменений, некоторые из которых имеют чисто количественное *значение*, а другие носят принципиальный характер.

Главным внешним отличием является увеличение разрядности *шины данных* и *шины адреса* до 32 *бит*. Это, в свою *очередь*, связано с изменениями в разрядности внутренних элементов микропроцессора.

Большие качественные изменения произошли на уровне работы микропроцессора в **защищенном режиме**, который был существенно развит *по* сравнению с i286. Отметим основные черты этого режима.

1. Принципиально меняется механизм формирования физического адреса. Прежде всего, изменяется механизм использования *сегментированной* памяти. Сегменты в защищенном режиме могут иметь произвольную длину и располагаться в памяти начиная с произвольного адреса. Каждый сегмент снабжается рядом атрибутов (базовый адрес, длина сегмента, его тип, уровень защиты и т. п.), которые хранятся в специальной структуре, называемой **дескриптором сегмента**, и используются блоком управления памятью микропроцессора при формировании *физических адресов* операндов и команд. Появляется возможность использования **страничного механизма** *организации памяти*. **Страница** - это раздел памяти, который, в отличие от сегмента, имеет фиксированную длину. *Страничная организация* памяти служит основой виртуальной памяти и обеспечивает более эффективное, по сравнению с сегментной, использование памяти.
2. Организуется аппаратная поддержка **мультипрограммного режима работы**, при котором в памяти одновременно содержатся программы и данные для

выполнения нескольких задач. Каждой задаче предоставляется свой <виртуальный процессор>. В каждый момент времени реальный процессор предоставляется одному из виртуальных процессоров, выполняющему свою задачу.

3. С целью обеспечения защиты информации и упрощения организации мультипрограммного режима работы микропроцессор снабжается специальными механизмами, определяющими, какие операции и обращения к памяти разрешается производить процессору при выполнении текущей задачи.

За время, прошедшее после появления первого 32-разрядного микропроцессора, только фирмой Intel было выпущено несколько десятков модификаций 32-разрядных МП. Изменения в некоторых моделях носили принципиальный характер, а ряд моделей содержали в основном лишь количественные изменения отдельных параметров (частота, емкость кэш-памяти и т. п.). Основные этапы развития этой архитектуры, которые, на наш взгляд, носят принципиальный характер, представлены в <u>табл. 1.1.</u>

ица 1.1. Этапы развития архитектуры IA-32								
дель	начала выпуска	ло транзисторов на кристале	симальная тактовая частота, МГц	ма обработки данных	личие кэш-памяти на кристале	стры	че-ство команд в системе команд	че-ство конвей-еров/ступеней конвей-ера
	5	тыс.		, ФТ		азрядные с ФТ***		
	9	млн.		, ФТ, ПТ		' --- разрядные с ПТ***	' ---	
ium	8	млн.		' ---	' ---	' ---	' ---	
iumMMX	7	млн.		' --- MD, ФПБ	' ---	' ---		
iumIII	9	млн. (28,1 млн.) *	0	' --- MD, П3	' --- ы L2	' --- 8-разрядные SSE		
ium4	0	лн.	0	' ---	' ---	' ---	4	

Остановимся вкратце на их рассмотрении.

К основным нововведениям микропроцессора i486, выпущенного в 1989 году, относятся два, которые связаны с расширившимися технологическими возможностями. Это *размещение* непосредственно на кристалле БИС двух важных блоков, которые раньше выполнялись в виде отдельных микросхем: кэш-памяти и блока процессора обработки чисел с плавающей точкой (**floating point unit - FPU**). Кэш-*память* имела объем 8

Кбайт и предназначалась для хранения программ и данных. *FPU* имел внутренний *файл* из восьми 80-разрядных регистров, свой *регистр состояния* и управления.

Главной отличительной чертой нового продукта в линейке 32-разрядных микропроцессоров - МП *Pentium* - явилась возможность *конвейерной обработки* информации. Хотя некоторые авторы считают, что конвейер появился уже в i486, это не является общепринятым мнением.

Высокая скорость выполнения команд в МП *Pentium* достигалась благодаря двум 5-ступенчатым конвейерам, позволявшим одновременно исполнять несколько инструкций. *Обмен информацией* с памятью через *кэш данных* осуществлялся независимо от *процессорного ядра*, а *буфер* инструкций был связан с ним через высокоскоростную 256-разрядную внутреннюю шину. Несмотря на то что новый кристалл был спроектирован как 32-разрядный, для связи с остальными компонентами системы использовалась внешняя 64-разрядная *шина данных*. Появление конвейера обусловило необходимость введения еще одного блока - схемы предсказания переходов. Эффективная работа данной схемы чрезвычайно важна для повышения производительности микропроцессора. Все последующие модификации микропроцессоров непременно связаны с улучшением ее работы.

Основным нововведением разработанного в 1997 году микропроцессора *Pentium MMX* стал блок, обеспечивавший новую схему обработки целочисленной информации - **SIMD (Single Instruction - Multiple Data**: одна *команда* - множество данных). До этого обработка велась *по* классической схеме ***SISD*: каждая *команда* выполняла действия над своей парой операндов.** Введение *SIMD*-операций позволило обрабатывать одновременно несколько операндов с использованием одной команды, что дало возможность существенно поднять *производительность*микропроцессора на тех задачах, где над большими массивами однородной информации выполнялись одинаковые *операции*, например, в мультимедийных приложениях. Появление таких возможностей потребовало введения в систему команд 57 новых инструкций, но регистровая структура микропроцессора не изменилась.

Микропроцессор Pentium III, появившийся в 1999 году, позволил обрабатывать *по* схеме *SIMD* не только целочисленные операнды, но и числа с плавающей точкой. Для этого *система команд*была расширена на 70 инструкций, а в структуре микропроцессора появился специальный блок *SSE*, содержащий, в

частности, отдельный регистровый *файл* из восьми 128-разрядных регистров. Еще одной новинкой, использованной в *Pentium* III, было *размещение* на кристалле кэш-памяти второго уровня (начиная с ядра Coppermine), работающей на частоте ядра. Но это носило скорее количественный характер и не внесло существенных изменений в архитектуру.

Микропроцессор Pentium 4 завершает линейку 32-разрядных микропроцессоров. Основным вкладом этого микропроцессора в развитие архитектуры *IA-32* стало еще большее увеличение глубины конвейера - до 31 стадии, что позволило сильно нарастить частоту процессора. Количество конвейеров возросло до 9. Кроме поддержки ставших традиционными инструкций *MMX* и *SSE*, в *Pentium* 4 добавили еще 144 команды *SSE2*, затем и *SSE3*, ориентированные в первую *очередь* на работу с потоковыми данными.

В 2001 году фирмой Intel был выпущен *микропроцессор Itanium*, положивший начало новой 64-разрядной архитектуре - *IA-64*, которая сменила архитектуру 32-разрядных микропроцессоров *IA-32*, господствовавшую на протяжении более 15 лет.

Данное учебное пособие в части универсальных микропроцессоров будет базироваться в основном на рассмотрении базовой архитектуры 32- разрядного микропроцессора, которая сложилась в микропроцессоре i486. Основные моменты, касающиеся развития этой архитектуры (конвейерная организация работы, обработка информации *по* схеме *SIMD* и т. д.), будут рассмотрены отдельно. Также отдельно будут рассмотрены современные направления развития архитектуры универсальных микропроцессоров и, в качестве примера, *архитектура* 80-ядерного микропроцессора фирмы Intel и микропроцессора *Itanium*.

Структура 32-разрядного универсального микропроцессора

Рассмотрение архитектуры ***IA-32*** начнем с микропроцессора i486. В нем впервые появились те блоки, которых не было на кристалле первого 32-разрядного микропроцессора i386, - кэш-*памятьи процессор* обработки чисел с плавающей точкой. Именно его архитектуру можно рассматривать как базовую для *IA-32*. Структура микропроцессора i486 представлена на рис. 1.3.

Рассмотрим состав и назначение основных блоков этого микропроцессора.

Процессор обработки чисел с фиксированной точкой содержит 32-разрядное *АЛУ* и блок **регистров общего назначения**. *АЛУ* предназначено для обработки двоичных чисел длиной 1, 2 или 4 байта без знака или со знаком, а также двоично-десятичных чисел, не превышающих 99. Двоичные числа со знаком представляются в *дополнительном коде*. Блок регистров общего назначения содержит восемь 32-разрядных регистров, часть из которых допускает 16- и 8-разрядное обращение.

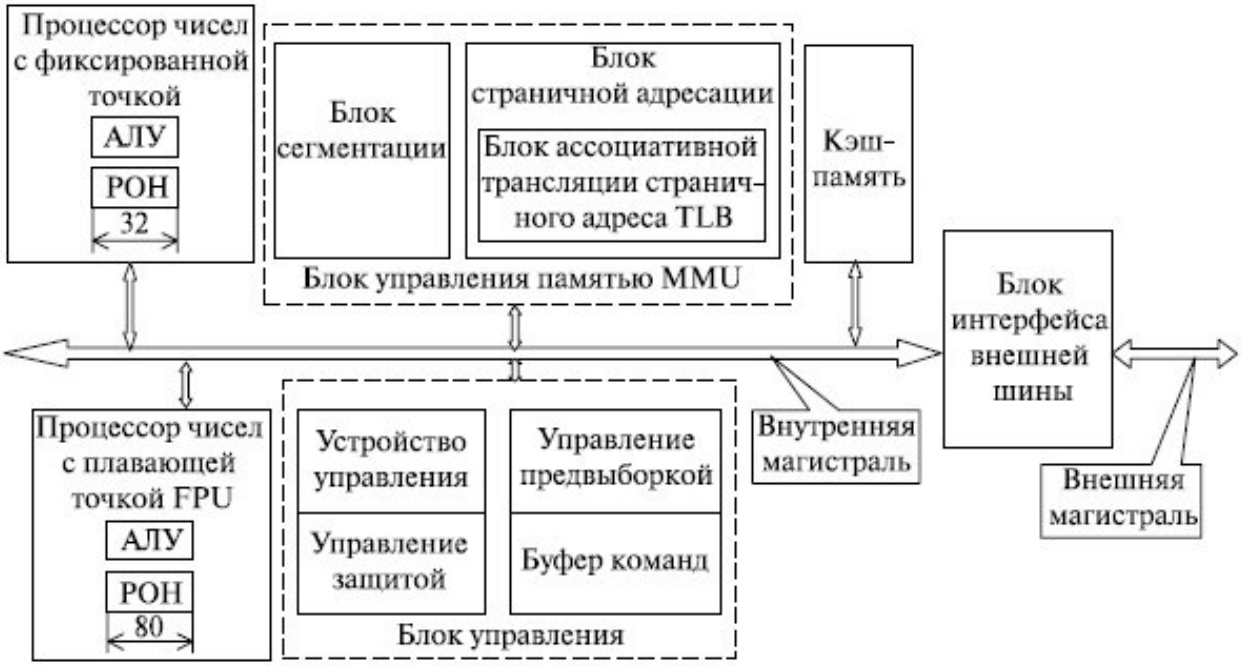

Рис. 1.3. Структура универсального микропроцессора

Процессор обработки чисел с плавающей точкой состоит из 80-разрядного *АЛУ*, блока из восьми 80-разрядных регистров общего назначения, а также управляющих регистров. Главным образом он предназначен для обработки чисел с плавающей точкой, но также используется для обработки целых чисел со знаком длиной 8 *байт* и двоично-десятичных чисел величиной от 100 до 99…9 (18 цифр). На первых этапах развития *SIMD*-обработки регистры **FPU** использовались для хранения операндов, представленных в новых форматах.

Блок управления памятью (*Memory Management* Unit - *MMU*) состоит из двух основных блоков в соответствии с *организацией памяти*.

В общем случае *память* в микропроцессоре делится на *сегменты*, которые, в свою *очередь*, делятся на страницы. В соответствии с этим, *MMU* содержит блок *сегментации* (или блок сегментного преобразования адреса) и блок *страничного преобразования*, в состав которого входит так называемый **буфер ассоциативной трансляции адресов стра- ниц** (*TLB*).

Кэш-память представляет собой промежуточную ступень между оперативной памятью и регистрами микропроцессора и предназначена для хранения наиболее часто используемой информации.

В состав **блока управления** входят:

- собственно *устройство управления*, то есть та классическая схема, которая под действием кода команды вырабатывает набор управляющих сигналов, поступающих на разные узлы как самого микропроцессора, так и на блок интерфейса внешней шины;

- управление *защитой памяти*: обеспечивает аппаратную защиту программ и данных при управлении памятью и по привилегиям;
- блок управления предвыборкой команд: реализует опережающее заполнение буфера команд, представляющего собой некоторую *буферную память*. Буфер команд имеет емкость 32 байта и заполняется командами из следующих ячеек памяти команд по мере своего освобождения. Этим обеспечивается ускорение обработки микропроцессором следующей команды. Данный блок подвергался, пожалуй, наиболее существенным переработкам по мере развития архитектуры *IA-32* - причина в широком последующем использовании конвейерной организации работы МП и связанной с этим необходимости постоянного совершенствования блока предсказания адреса следующей команды.

Блок интерфейса внешней шины осуществляет электрическое согласование параметров внутренней магистрали с сигналами внешних *магистралей*, формирование необходимых сигналов на внешнюю *магистраль* и прием сигналов извне. Внешняя *магистраль* микропроцессора состоит из *шины адреса*, *шины данных* и сигналов управления:

- *шина данных* имеет ширину 32 разряда;
- 32-разрядный адрес передается по 34-разрядной шине A31...A2+(B3,B2,B1,B0). Чтобы с минимальными потерями согласовывать 32-разрядную *шину данных* с передачей данных меньшей разрядности, младшие разряды адреса (A1 и A0) передаются в дешифрированном виде (B3, B2, B1, B0). Они показывают, какие байты из 32-разрядной *шины данных* в данный момент реально востребованы: 1 байт, 2 младших байта, 2 старших байта либо все 32 разряда данных;
- шина управления - 32-разрядная. По ней передаются сигналы записи и чтения содержимого оперативной памяти и внешних устройств, сигналы *запросов прерываний*, прямого доступа к памяти и т. д.

Особый интерес представляют три режима работы микропроцессора: реальный, защищенный и режим виртуального МП i8086. В **реальном режиме** обеспечивается совместимость на уровне объектных кодов с микропроцессором i8086 и микропроцессором i286, работающем в реальном режиме. В этом режиме *архитектура* 32-разрядного микропроцессора почти полностью идентична архитектуре 16-разрядного МП. Для программиста же он вообще представляется как МП i8086, выполняющий написанные программы с большей скоростью и обладающий расширенной системой команд и регистрами. Благодаря этим качествам *фирма* Intel сохранила прежних клиентов, которые хотели модернизировать свои системы, не отказываясь от имевшегося задела в области программного обеспечения, и привлекла тех, кому изначально требовалась высокая скорость обработки информации.

Одно из основных ограничений реального режима было связано с предельной емкостью адресуемой памяти, равной 1 Мбайт. От него свободен **защищенный режим**, позволяющий воспользоваться всеми преимуществами архитектуры нового МП. Размер адресного пространства в этом случае увеличивается до 4 Гбайт, а

общий объем поддерживаемого адресного пространства - до 64 терабайт (1 Тбайт = 2^{40} *байт*). МП, работающие в защищенном режиме, обладают более высоким быстродействием и возможностями организации истинной *многозадачности*.

Наконец, **режим виртуального МП** открывает возможность одновременного исполнения программ, написанных для МП i8086, i286 и i386.

Поскольку *емкость памяти*, адресуемой микропроцессором, не ограничена значением 1 Мбайт, этот режим позволяет формировать несколько виртуальных сред i8086.

Краткие итоги. В лекции даны определения микропроцессора и *микропроцессорной системы*, архитектуры микропроцессора. Приведена классификация микропроцессоров *по* их архитектуре, представлены параметры, которые характеризуют микропроцессоры каждого класса как вычислительное устройство и как электронное изделие. Рассмотрены этапы развития архитектуры универсальных микропроцессоров на примере МП БИС фирмы Intel, занимающей *доминирующее положение* в этом секторе рынка. Описаны структура и основные блоки микропроцессора i486, являющегося базовым микропроцессором для этой архитектуры.

Лекция 2. Регистровая структура универсального микропроцессора

Регистровая структура универсального микропроцессора

В универсальном 32-разрядном микропроцессоре выделяют следующие **группы регистров**:

- основные функциональные регистры;
- регистры процессора с плавающей точкой;
- системные регистры;
- регистры отладки и тестирования.

Первые две группы регистров используются прикладными программами, последние две группы - *системными программами*, имеющими наивысший *уровень привилегий*.

Рассмотрим каждую из этих групп подробнее.

Основные функциональные регистры

В состав регистров этой группы входят:

- регистры общего назначения;
- *регистр указателя* команд;
- *регистр флагов*;
- *сегментные регистры*.

Состав и структура **регистров общего назначения** представлены на рис. 2.1.

Блок состоит из восьми 32-разрядных регистров. К каждому из них можно обращаться как к одному двойному слову (32 разряда).

Отметим, что понятие "*слово*" в данной архитектуре не идентично разрядности микропроцессора. Исторически сложилось так, что под **словом** понимается *единица* информации длиной 2 байта, или 16 двоичных разрядов. К младшим 16 разрядам регистров общего назначения можно обращаться так же, как и в 16-разрядном микропроцессоре (AX, BX…*SP*). Четыре 16-разрядных регистра AX, BX, CX, DX допускают обращение отдельно к своему старшему и младшему байту. Тем самым регистры позволяют на программном уровне работать либо с восемью 32-разрядными, либо с восемью 16-разрядными, либо с восемью 8-разрядными регистрами.

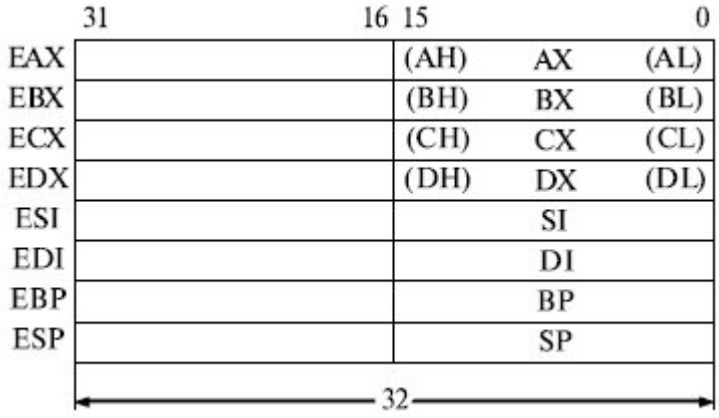

Рис. 2.1. Регистры общего назначения

Все эти регистры используются для хранения промежуточных результатов вычислений и составных частей адреса при различных режимах адресации операндов, расположенных в памяти.

Кроме того, ряд регистров этого блока имеют свое, присущее только им назначение:

- EAX/AX/AL - регистр-аккумулятор, используется для сокращения длины команды при работе с непосредственными операндами;
- AX/AL - приемник (источник) данных в командах ввода (вывода) данных из (в) внешнего устройства;
- DX - определяет адрес ВУ в командах ввода (вывода) данных;
- ECX - используется в качестве счетчика циклов в *командах циклов*;
- BP, *SP* - используются при работе со стеком;
- *ESI*, *EDI* (DI, *SI*) - определяют положение строк в памяти в командах обработки строк.

Регистр указателя команд и регистр флагов имеют длину 32 разряда.

Младшее *слово* каждого из этих регистров (разряды 0-15) функционально соответствует аналогичным разрядам в 16-разрядном микропроцессоре (рис. 2.2).

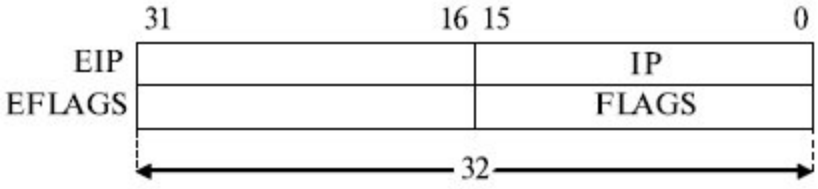

Рис. 2.2. Регистр указателя команд и регистр флагов

Регистр указателя команд *EIP* хранит смещение адреса команд относительно начала сегмента кода (сегмента команд).

Регистр флагов EFLAGS содержит признаки результата выполненной команды, а также разряды, *управляющие* работой микропроцессора: обработкой

маскированных прерываний, последовательностью вызываемых задач, вводом-выводом и рядом других действий. Из этих флагов рассмотрим только наиболее значимые и интересные с точки зрения дальнейшего изучения работы микропроцессора.

К **битам состояния** регистра флагов относятся:

- ZF - признак нуля результата (ZF = 1, если все разряды результата равны 0);
- SF - знак результата (SF = 1, если старший разряд результата равен 1, то есть если результат отрицательный);
- OF - признак переполнения (OF = 1, если при выполнении арифметических операций над числами со знаком происходит переполнение разрядной сетки);
- CF - флаг переноса (CF = 1, если выполнение операции сложения приводит к переносу за пределы разрядной сетки), устанавливается также в некоторых других операциях;
- PF - признак четности (дополняет до нечетного числа единиц младший байт результата);
- AF - флаг полупереноса (используется при операциях над двоичнодесятичными числами);
- DF - устанавливается пользователем и определяет порядок обработки строк символов в соответствующих командах: *декремент* (при DF = 1) или *инкремент* (при DF = 0) содержимого индексных регистров *ESI*, *EDI* (*SI*, DI) после обработки одного символа.

В состав **флагов управления** входят:

- IF - **флаг прерываний** (при IF = 1 разрешается обработка маскированных *аппаратных прерываний*);
- TF - флаг *ловушки*, или трассировки (при TF = 1 после выполнения каждой команды возникает прерывание, используемое отладчиками;);
- NT - бит вложенной задачи (показывает, что данная задача была вызвана из другой программы, аналогично подпрограмме, и возврат из этой задачи должен проводиться по механизму переключения задач);
- IOPL - 2-разрядное поле **уровня привилегий ввода/вывода** (определяет *уровень привилегий* программ, которым разрешено выполнение операции ввода-вывода);
- VM - режим виртуального микропроцессора i8086 (при работе микропроцессора в защищенном режиме установка VM = 1 вызывает переключение в режим виртуального микропроцессора i8086; в этом случае микропроцессор функционирует как быстрый МП i8086, но реализует механизмы *защиты памяти*, страничной адресации и ряд других дополнительных возможностей; бит VM может быть установлен только в защищенном режиме).

Блок **сегментных регистров** состоит из шести 16-разрядных регистров, которые указывают на различные *сегменты*, расположенные в памяти компьютера:

- CS (*Code Segment*) - сегмент кода программы;
- *DS* (*Data Segment*) - сегмент данных;
- SS (*Stack Segment*) - сегментстека;
- ES, FS, GS - дополнительные сегменты данных.

При работе микропроцессора в реальном режиме в сегментном регистре содержатся старшие 16 разрядов 20-разрядного базового адреса сегмента.
Физический *адрес* начала сегмента получается умножением этой величины на 16:

$$A_{\text{баз сегм}} = (\text{сегментный регистр}) * 16$$

Получающийся 20-разрядный *адрес* позволяет адресовать *память* емкостью 2^{20} *байт* = 1 Мбайт. При этом *сегменты* имеют постоянную длину 2^{16} *байт*. Разработчики первых персональных компьютеров полагали, что оперативная *память*, большая чем 1 Мбайт, никогда не потребуется пользователю, поэтому вся *архитектура* строилась исходя именно из этого положения.

При переходе к 32-разрядной архитектуре стало необходимым обеспечить возможность адресации памяти емкостью до 2^{32} *байт*. Кроме того, введение защищенного режима работы микропроцессора потребовало хранения большого количества дополнительной информации о сегменте: его длине, которая стала переменной, уровне привилегий, его типе и т. д. Простое увеличение разрядности сегментных регистров до 32 *бит* не обеспечило бы возможности хранения всей этой информации. Поэтому все данные о сегменте стали размещаться в специальных структурах - **дескрипторах (описателях) сегментов**, которые хранятся в **таблицах дескрипторов**, расположенных в памяти, а *сегментные регистры*, сохранив свою первоначальную длину в 16 разрядов, содержат так называемый селектор (*указатель*), который используется для того, чтобы найти нужный **дескриптор** в этих таблицах.

Регистры процессора с плавающей точкой

К этой группе регистров относятся (рис. 2.3):

- регистры данных;
- регистры тегов;
- *регистр состояния*;
- *указатели команд* и данных *FPU*;
- *регистр управления FPU*.

Рис. 2.3. Структура регистров процессора с плавающей точкой

Блок регистров данных. Доступен либо как *стек* (его *вершина* TOP определена в регистре состояний *FPU*), либо как набор пронумерованных регистров.

Старший разряд 80-разрядного регистра данных кодирует знак *мантиссы* хранящегося в нем числа с плавающей точкой. Следующее *поле* отведено под *кодирование* порядка. Порядок представлен в виде так называемого машинного, или смещенного, порядка (Псм) без знака:

$$\text{Псм} = \text{П} + \Delta$$

где П - истинный порядок числа, а величина Δ определяется следующим образом: $\Delta = |-\text{Пмакс}|$ При этом самый большой *по* модулю отрицательный истинный порядок преобразуется в нулевой смещенный, а все остальные истинные порядки преобразуются в положительные. Тем самым упрощаются *операции* обработки чисел с плавающей точкой.

В последнем *поле* регистра данных записывается *мантисса* числа.

Количество разрядов, отводимых под *поле* порядка и *поле мантиссы*, определяется регистром управления *FPU*.

Данный блок может также использоваться для хранения двоичнодесятичных чисел. Они представляются в упакованной форме и содержат 18 тетрад, каждая из которых соответствует одному десятичному разряду.

Для представления знака такого числа используется старший разряд старшего байта (*бит* 79), в остальных разрядах этого байта устанавливаются нули.

Микропроцессор может обрабатывать числа следующих типов (табл. 2.1):

Таблица 2.1. Типы чисел 32-разрядного микропроцессора			
Тип	**Размер, байт**	**Диапазон**	**работка**
ые без знака		255 65535 $4,3*10^9$	*У* ФТ
ые со знаком		8...+127 768...+32767 $*10^9$...+2,1*10^9	*У* ФТ
		$*10^{18}$...+9*10^{18}	*U*
лавающей точкой	+8+23)*мантисса*	$37*10^{35}$	*U*
	+11+52)	$67*10^{308}$ 308	
	1+15+64)	$1*10^{4932}$	
ично- десятичные числа	спакованный	9	*У* ФТ
	акованный	99	*У* ФТ
	упакованных	…9(18 цифр)	*U*

Помимо этого *микропроцессор* может обрабатывать символьные данные, данные типа "строка" и типа"*указатель*".

Регистр тегов. Определяет содержимое регистра данных с целью оптимизации обработки:

- 00 - достоверное значение;
- 01 - нуль (нулевое значение);
- 10 - не-числа (например, бесконечность);
- 11 - пусто (содержание регистров не определено).

Операции с плавающей точкой требуют довольно много времени.

Использование тегов позволяет в определенных случаях сократить *время выполнения команды*. Например, если известно, что один из сомножителей равен нулю, то произведению можно присвоить нулевое *значение* без выполнения каких-либо действий.

Регистр состояния содержит *указатель* вершины блока данных, работающего в режиме стека (TOP), признаки результата и ошибок, возникающих при

выполнении *операции* в *FPU*, а также флаг переполнения и антипереполнения стека регистров данных.

Регистр управления управляет округлением (к ближайшему значению, вниз, вверх, к нулю), точностью (*длина мантиссы* 23, 52 или 64 бита), а также содержит маску признаков ошибок, фиксируемых в *регистре состояния*.

Указатели команд и данных содержат *адрес* команды, вызвавшей ошибку, и *адрес* использованного операнда. Эти регистры имеют 48-разрядный формат: 16 разрядов содержат селектор соответствующего сегмента, а остальные 32 разряда - смещение в нем.

Системные регистры

Системные регистры управляют функционированием микропроцессора в целом и режимами работы отдельных его блоков. Эти регистры доступны только в защищенном режиме для программ, имеющих максимальный *уровень привилегий*. Они включают в свой состав 2 группы регистров:

- *регистры управления* (CR0...CR4);
- регистры системных адресов и системных сегментов.

Регистр управления CR0 содержит биты, определяющие режим работы процессора:

- PE - разрешение защиты: установка PE = 1 переводит микропроцессор в *защищенный режим*;
- PG - включение страничной адресации памяти (при PG = 1 страничный механизм включен);
- CD, NW - управление режимами работы внутренней кэш-памяти (CD = 1 - запрещение заполнения кэш-памяти; NW = 1 - запрет сквозной записи).

Ряд *бит* (*MP*, EM, TS, NE) управляют режимами работы *FPU*.

Регистр CR1 был зарезервирован для последующего развития. Однако начиная с МП Pentium в микропроцессорах появился *регистр управления* CR4, а *регистр* CR1 так и остался зарезервированным.

Регистр CR2 содержит линейный *адрес*, который вызвал страничную ошибку, например, отсутствие страницы в оперативной памяти или недостаточный *уровень привилегий*.

В регистре CR3 находится базовый *адрес* каталога *таблицы страниц* (старшие 20 разрядов), а также биты *PCD* и PWT, *управляющие* работой кэш-памяти при страничной адресации (при *PCD* = 1 *загрузка* содержимого страницы в кэш-*память* запрещена; при PWT = 1 реализуется режим сквозной записи, а при PWT = 0 - обратной записи).

Регистр CR4 содержит биты, обеспечивающие расширение функциональных возможностей микропроцессора, начиная с Pentium. В частности, он содержит следующие *управляющие* разряды:

- *VME*, PVI - управляют работой виртуальных прерываний;
- *PAE* - обеспечивает расширение физического адреса до 36 разрядов (при *PAE* = 1); PGE - определяет некоторые страницы (часто используемые или используемые несколькими процессорами) как глобальные (при PGE = 1);
- *PSE* - расширяет размер адресуемых страниц до 4 Мбайт (при *PSE* = 1), при *PSE* = 0 сохраняет размер страницы 4 Кбайт.

Регистры системных адресов и системных сегментов представлены на рис. 2.4.

Рис. 2.4. Структура регистров системных адресов и системных сегментов

В их число входят GDTR - **регистр глобальной таблицы дескрипторов** и IDTR - **регистр таблицы дескрипторов прерываний**. В этих регистрах определяются базовый *адрес* и размер соответствующей таблицы. К

К этой группе относятся также LDTR - **регистр локальной таблицы дескрипторов** и TR - **регистр задач**. Регистры LTDR и TR представляют собой селекторы, которые указывают на положение дескрипторов, описывающих соответственно сегмент, содержащий локальную таблицу дескрипторов, и сегмент состояния задачи (*Task State Segment* - TSS).

Использование этих регистров в дальнейшем будет рассмотрено более подробно.

Регистры отладки и тестирования

32-разрядные **регистры отладки** (DR0...DR7) имеют следующее назначение:

- DR0...DR3 - содержат линейные адреса 4 контрольных точек останова при отладке;
- DR4 и DR5 зарезервированы;
- DR6 - *регистр состояния*: показывает текущее состояние МП при останове в этих точках;

- DR7 - *регистр управления*: задает условия останова в контрольных точках. Регистры DR4 и DR5 не используются.

Регистры тестирования (TR3...TR7) используются при тестировании кэш-памяти и **буфера ассоциативной трансляции адресов страниц** (TLB).

По мере развития архитектуры микропроцессора их количество расширилось и дополнилось новым содержанием. В частности, с помощью регистра TR12 можно запретить предсказание и трассировку ветвлений, параллельное выполнение инструкций и выполнить некоторые другие действия.

Краткие итоги. В лекции рассмотрены назначение и состав регистровой структуры универсального микропроцессора, во многом определяющие архитектурные особенности микропроцессора.

Лекция 3. Физическая и логическая организация адресного пространства

Логическое адресное пространство

Для адресации операндов в физическом адресном пространстве программы используют логическую адресацию. *Процессор* автоматически транслирует логические адреса в физические, выдаваемые затем на системную шину.

Архитектура компьютера различает физическое *адресное пространство* (ФАП) и логическое *адресное пространство* (ЛАП). **Физическое адресное пространство** представляет собой простой *одномерный массив* байтов, *доступ* к которому реализуется аппаратурой памяти *по* адресу, присутствующему на *шине адреса микропроцессорной системы*. **Логическое адресное пространство** организуется самим программистом исходя из конкретных потребностей. Трансляцию логических адресов в физические осуществляет блок управления памятью *MMU*.

В архитектуре современных *микропроцессоров* ЛАП представляется в виде набора элементарных структур: байтов, сегментов и страниц. В микропроцессорах используются следующие варианты организации **логического адресного пространства**:

- **плоское (линейное) ЛАП**: состоит из массива байтов, не имеющего определенной структуры; трансляция адреса не требуется, так как *логический адрес* совпадает с физическим;
- **сегментированное ЛАП**: состоит из сегментов - непрерывных областей памяти, содержащих в общем случае переменное число байтов; *логический адрес* содержит 2 части: идентификатор сегмента и смещение внутри сегмента; *трансляцию адреса* проводит блок *сегментации MMU*;
- **страничное ЛАП**: состоит из страниц - непрерывных областей памяти, каждая из которых содержит фиксированное число байтов. *Логический адрес* состоит из номера (идентификатора) страницы и смещения внутри страницы; *трансляция логического адреса в физический* проводится блоком *страничного преобразования MMU*;
- **сегментно-страничное ЛАП**: состоит из **сегментов**, которые, в свою очередь, состоят из страниц; *логический адрес* состоит из идентификатора сегмента и смещения внутри сегмента. Блок сегментного преобразования *MMU* проводит трансляцию *логического адреса* в номер страницы и смещение в ней, которые затем транслируются в *физический адрес* блоком *страничного преобразования MMU*.

Таким образом, основой получения *физического адреса* памяти служит *логический адрес*. В какой-то степени логическое *адресное пространство*, с которым имеет дело программист, можно сравнить со структурой книги, где аналогом сегмента выступает рассказ, страница книги соответствует странице ЛАП, а

искомая *информация* - это некоторое *слово*. При этом если *память* организована как линейная, то номер искомого слова задается в явном виде и просто отсчитывается от начала книги. При сегментном представлении памяти искомое *слово* определяется его номером в заданном рассказе.
Страничное *представление* памяти предполагает задание информации о слове в виде номера страницы в книге и номера слова на указанной странице. При сегментно-страничном представлении *логический адрес* слова задается номером слова в определенном рассказе. В этом случае *по* оглавлению книги определяется номер страницы, с которой начинается указанный рассказ. Затем, зная количество слов на странице и положение слова в рассказе, можно вычислить страницу книги и положение искомого слова на этой странице.

Формирование физического адреса в универсальном микропроцессоре при различных режимах работы

Микропроцессор способен работать в двух режимах: реальном и защищенном.

При работе в **реальном режиме** возможности процессора ограничены: емкость адресуемой памяти составляет 1 Мбайт, отсутствует *страничная организация* памяти, **сегменты** имеют фиксированную длину 2^{16} *байт*.

Этот режим обычно используется на начальном этапе загрузки компьютера для перехода в **защищенный режим**.

В **реальном режиме** *сегментные регистры* процессора содержат старшие 16 *бит физического адреса* начала **сегмента**. Сдвинутый на 4 разряда влево **селектор** дает 20-разрядный базовый *адрес* сегмента. *Физический адрес* получается путем сложения этого адреса с 16-разрядным значением смещения в сегменте, формируемого *по* заданному режиму адресации для операнда или извлекаемому из регистра *EIP* для команды (рис. 3.1). *По* полученному адресу происходит *выборка* информации из памяти.

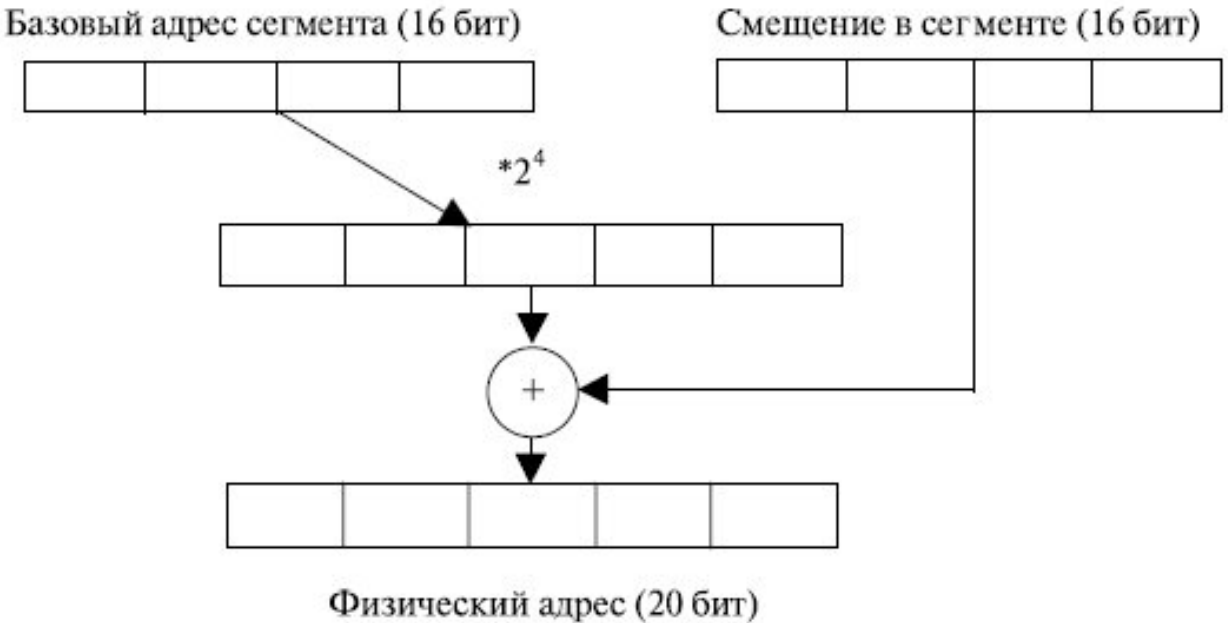

Рис. 3.1. Схема получения физического адреса

Наиболее полно возможности *микропроцессора по* адресации памяти реализуются при работе в **защищенном режиме**. Объем адресуемой памяти увеличивается до 4 Гбайт, появляется возможность страничного режима адресации. **Сегменты** могут иметь переменную длину от 1 байта до 4 Гбайт.

Общая схема формирования *физического адреса микропроцессором*, работающим в **защищенном режиме**, представлена на рис. 3.2.

Как уже отмечалось, основой формирования *физического адреса* служит *логический адрес*. Он состоит из двух частей: **селектора** и **смещения в сегменте**.

Селектор содержится в *сегментном регистре микропроцессора* и позволяет найти описание сегмента **(дескриптор)** в специальной таблице дескрипторов. **Дескрипторы** сегментов хранятся в специальных *системных объектах* глобальной (*GDT*) и локальных (*LDT*) таблицах дескрипторов. **Дескриптор** играет очень важную роль в функционировании *микропроцессора*, от формирования *физического адреса* при различной организации адресного пространства и до организации мультипрограммного режима работы. Поэтому рассмотрим его структуру более подробно.

Сегменты *микропроцессора*, работающего в **защищенном режиме**, характеризуются большим количеством параметров. Поэтому в универсальных 32-разрядных микропроцессорах *информация* о сегменте хранится в

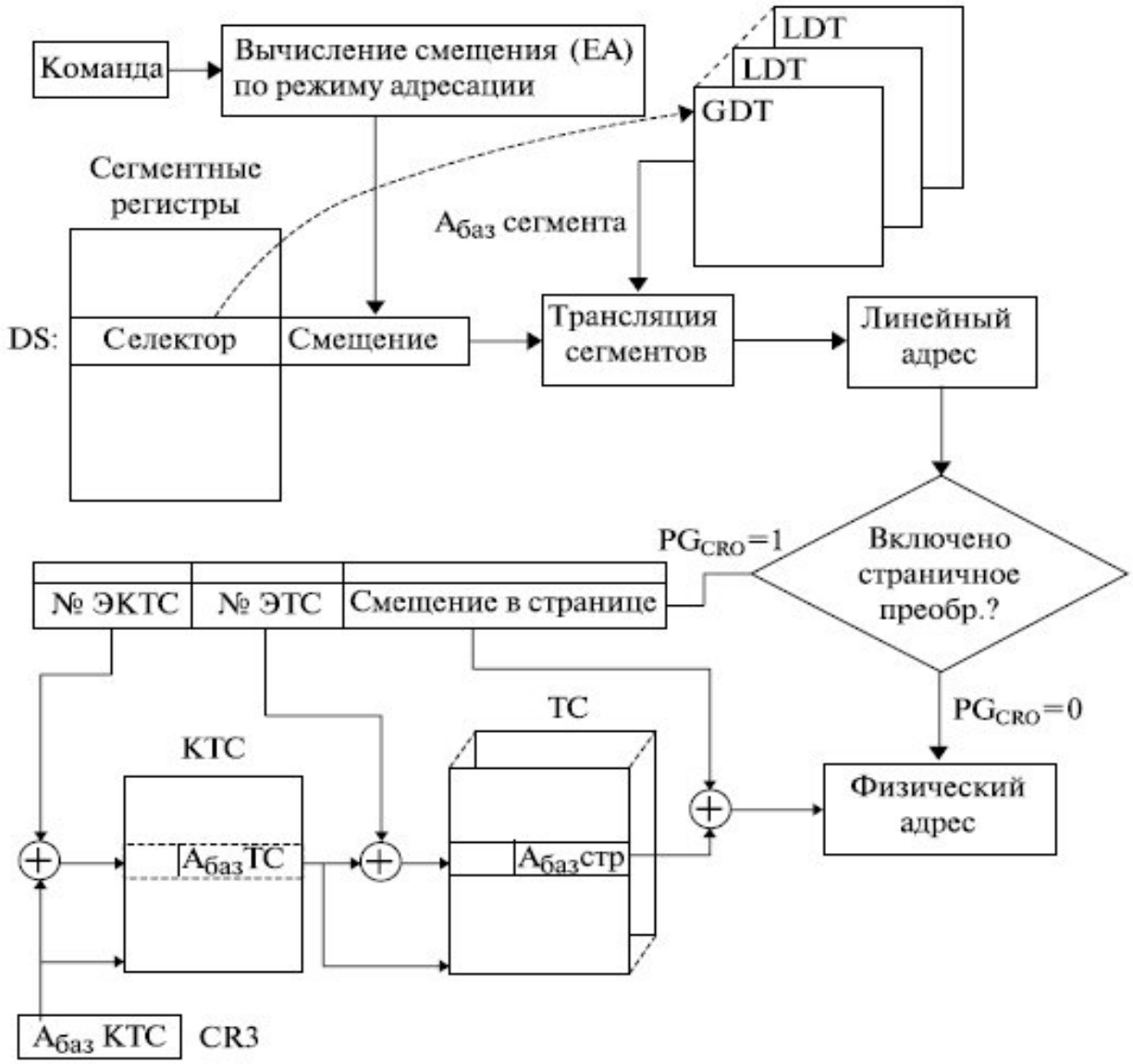

Рис. 3.2. Формирование физического адреса при сегментно-страничной организации памяти

специальной 8-байтной структуре данных, называемой **дескриптором**, а за *сегментными регистрами* закреплена основная *функция* - *определение* местоположения дескриптора.

Структура **дескриптора сегмента** представлена на рис. 3.3.

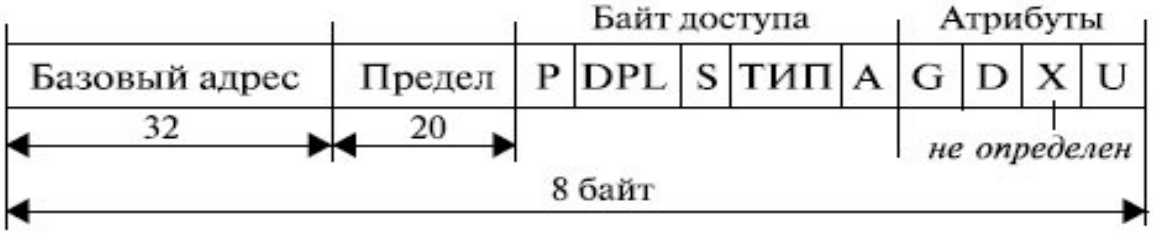

Рис. 3.3. Структура дескриптора сегмента

Мы будем рассматривать именно структуру, а не формат дескриптора, так как при переходе от *микропроцессора* i286 к 32-разрядному МП расположение отдельных полей дескриптора потеряло свою стройность и частично стало иметь вид "заплаток", поставленных с целью механического увеличения разрядности этих полей.

32-разрядное *поле* базового адреса позволяет определить начальный *адрес* сегмента в любой точке адресного пространства в 2^{32} *байт* (4 Гбайт).

Поле предела (limit) указывает длину сегмента (точнее, длину сегмента минус 1: если в этом *поле* записан 0, то это означает, что сегмент имеет длину 1) в адресуемых единицах, то есть максимальный размер сегмента равен 2^{20} элементов.

Величина элемента определяется одним из атрибутов дескриптора битом G (*Granularity* - *гранулярность*, или дробность):

$$G = \begin{cases} 0\text{-длина в байтах} \\ 1\text{-длина в страницах} \end{cases}$$

Таким образом, сегмент может иметь размер с точностью до 1 байта в диапазоне от 1 байта до 1 Мбайт (при G = 0). При объеме страницы в 2^{12} = 4 Кбайт можно задать объем сегмента до 4 Гбайт (при G = l):

$$V_{\text{сегм макс}} = 2^{20}_{\text{стр}} * 2^{12}_{\text{байт}} = 2^{32}\text{байт}$$

Так как в архитектуре *IA-32* сегмент может начинаться в произвольной точке адресного пространства и иметь произвольную длину, *сегменты* в памяти могут частично или полностью перекрываться.

Бит размерности (Default size) определяет длину адресов и операндов, используемых в команде *по* умолчанию:

$$D = \begin{cases} 0\text{-16 разрядов} \\ 1\text{-32 разряда} \end{cases}$$

Конечно, этот *бит* предназначен не для обычного пользователя, а для системного программиста, применяющего его, например, для отметки сегментов для сбора"мусора" или сегментов, базовые адреса которых нельзя модифицировать. Этот *бит* доступен только программам, работающим на высшем уровне привилегий. *Микропроцессор* в своей работе его не меняет и не использует.

Байт доступа определяет основные правила обращения с сегментом.

Бит присутствия P (Present) показывает возможность доступа к сегменту. *Операционная система* (ОС) отмечает сегмент, передаваемый из оперативной во *внешнюю память*, как временно отсутствующий, уставливая в его дескрипторе P = 0. При P = 1 сегмент находится в физической памяти. Когда выбирается *дескриптор* с P = 0 (сегмент отсутствует в *ОЗУ*), поля базового адреса и предела игнорируются. Это естественно: например, как может идти речь о базовом адресе сегмента, если самого сегмента вообще нет в оперативной памяти? В этой ситуации *процессор* отвергает все последующие попытки использовать **дескриптор** в командах, и определяемое **дескриптором** *адресное пространство* как бы"пропадает".

Возникает особый случай неприсутствия сегмента. При этом *операционная система* копирует запрошенный сегмент с диска в *память* (при этом, возможно, удаляя другой сегмент), загружает в **дескриптор** базовый *адрес* сегмента, устанавливает P = 1 и осуществляет *рестарт* той команды, которая обратилась к отсутствовавшему в *ОЗУ* сегменту.

Двухразрядное *поле DPL* (*Descriptor Privilege Level*) указывает один из четырех возможных (от 0 до 3) **уровней привилегий дескриптора**, определяющий возможность доступа к сегменту со стороны тех или иных программ (уровень 0 соответствует самому высокому уровню привилегий).

Бит обращения A (Accessed) устанавливается в"1" при любом обращении к сегменту. Используется операционной системой для того, чтобы отслеживать *сегменты*, к которым дольше всего не было обращений.

Пусть, например, 1 раз в секунду *операционная система* в дескрипторах всех сегментов сбрасывает *бит* A. Если *по* прошествии некоторого времени необходимо загрузить в оперативную *память* новый сегмент, места для которого недостаточно, *операционная система* определяет"кандидатов" на то, чтобы очистить часть оперативной памяти, среди тех сегментов, в **дескрипторах** которых *бит* A до этого момента не был установлен в"1", то есть к которым не было обращения за последнее время.

Поле типа в байте доступа определяет назначение и особенности использования сегмента. Если *бит* S (System - *бит* 4 байта доступа) равен 1, то данный **дескриптор** описывает реальный сегмент памяти. Если S = 0, то этот *дескриптор* описывает специальный системный *объект*, который может и не быть сегментом памяти, например, *шлюз* вызова, используемый при переключении задач, или *дескриптор* локальной таблицы дескрипторов *LDT*. Назначение битов <3...0> байта доступа определяется типом сегмента (рис. 3.4).

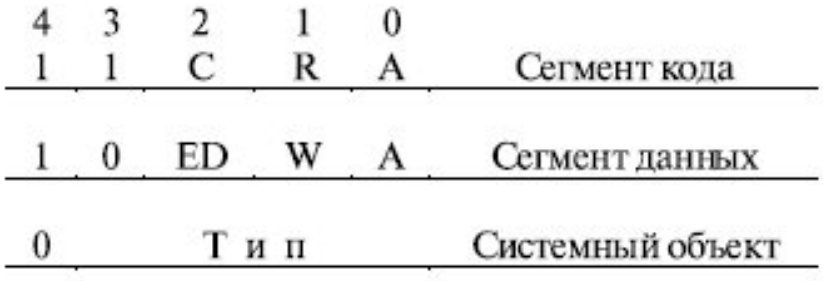

Рис. 3.4. Формат поля типа байта доступа

В сегменте кода: *бит* подчинения, или согласования, C (*Conforming*) определяет дополнительные правила обращения, которые обеспечивают защиту сегментов программ. При *C* = 1 данный сегмент является подчиненным сегментом кода. В этом случае он намеренно лишается защиты *по* привилегиям. Такое средство удобно для организации, например, подпрограмм, которые должны быть доступны всем выполняющимся в системе задачам. При *C* = 0 - это обычный сегмент кода; *бит* считывания R (*Readable*) устанавливает, можно ли обращаться к сегменту только на *исполнение* или на *исполнение* и считывание, например, констант как данных с помощью префикса замены сегмента. При R = 0 допускается только *выборка* из сегмента команд для их выполнения. При R = 1 разрешено также *чтение данных* из сегмента.

Запись в сегмент кода запрещена. При любой попытке записи возникает *программное прерывание*.

В сегменте данных:

- ED (*Expand* Down) - бит направления расширения. При ED = 1 этот сегмент является сегментом стека и смещение в сегменте должно быть больше размера сегмента. При ED = 0 - это сегмент собственно данных (смещение должно быть меньше или равно размеру сегмента);
- бит разрешения записи W(Writeable). При W = 1 разрешено изменение сегмента. При W = 0 запись в сегмент запрещена, при попытке записи в сегмент возникает *программное прерывание*.

В случае обращения за операндом **смещение в сегменте** формируется *микропроцессором по* режиму адресации операнда, заданному в команде. Смещение в сегменте кода извлекается из **регистра - указателя команд** *EIP*.

Сумма извлеченного из дескриптора начального адреса сегмента и сформированного смещения в сегменте дает **линейный адрес** (ЛА).

Если в *микропроцессоре* используется только сегментное *представление* адресного пространства, то полученный линейный *адрес* является также и физическим.

Если помимо сегментного используется и страничный механизм *организации памяти*, то **линейный адрес** представляется в виде двух полей: старшие разряды содержат номер *виртуальной страницы*, а младшие смещение в странице. Преобразование номера *виртуальной страницы* в номер физической проводится с помощью специальных системных таблиц: **каталога таблиц страниц** (КТС) и **таблиц страниц** (ТС). Положение каталога *таблиц страниц* в памяти определяется системным регистром CR3. *Физический адрес* вычисляется как сумма полученного из *таблицы страниц* адреса *физической страницы* и смещения в странице, полученного из линейного адреса.

Рассмотрим теперь все этапы преобразования *логического адреса* в физический более подробно.

Структура кода команды и формирование смещения в сегменте

Как отмечалось выше, смещение в сегменте кода команд извлекается из регистра *EIP* и поэтому не требует дополнительных пояснений.

Механизм же формирования **смещения в сегменте** данных проводится на основе режима адресации операнда и требует отдельного изучения. Рассмотрим сначала структуру кода команды универсального 32-разрядного *микропроцессора*. Команды в архитектуре *IA-32* имеют большое разнообразие форматов, которые зависят от типа *операции*, режимов адресации операндов, длины используемых непосредственных операндов и смещений и ряда других факторов. Они имеют длину от 1 до 15 *байт*. Все это существенно затрудняет их *декодирование* в МП с данной архитектурой. На рис. 3.5 представлен формат двухоперандной команды общего вида.

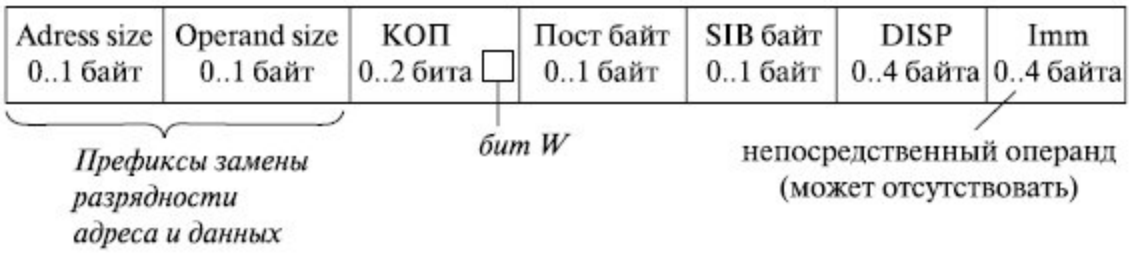

Рис. 3.5. Формат команды 32-разрядного микропроцессора

Команда может начинаться с нескольких необязательных *байт* (префиксов), которые определяют особенности выполнения команды. Префиксы размеров длины адреса и операндов позволяют изменить их значения,
установленные *по* умолчанию **битом размерности** D в дескрипторе сегмента. Для операндов совместно с битом w, содержащимся в коде команды, *префикс* размера позволяет определить *операнд* длиной 8, 16 или 32 разряда. *Префикс* размера адреса определяет 16- или 32-разрядное смещение в сегменте (табл. 3.1).

лица 3.1. Использование префиксов переопределения размеров адресов и операндов								
размерности D в дескрипторе сегмента								
фикс размерности Операнда*								
фикс размерности адреса*								
ядность операнда (бит) **								
ядность адреса (бит)								

Примечания:

* + - *префикс* присутствует;

- - *префикс* отсутствует.

** w = 1/0

В коде команды могут использоваться также дополнительные байты для префикса замены сегментного регистра, установленного *по* умолчанию, префикса повторения *операции* или префикса, предотвращающего прерывания *операции* перемещения данных.

Поле* КОП** содержит код выполняемой команды, а также ***бит* w** размерности используемых операндов. Для команд, применяющих непосредственный *операнд, код операции* может также занимать также часть ***постбайта.

***Постбайт* (рис. 3.6)** определяет местоположение операндов. Основная часть команд *микропроцессора* с архитектурой *IA-32* позволяет работать только с одним операндом, находящимся в оперативной памяти. Его *режим адресации* кодируется полями *md* и r/m *постбайта*. Второй *операнд* либо извлекается из **регистров общего назначения** *микропроцессора* (его номер указывается в *поле* reg *постбайта*), либо кодируется в *поле* Imm самой команды (непосредственный *операнд*).

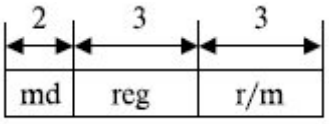

Рис. 3.6. Формат постбайта

Байт масштабируемого индекса базы (*SIB*) служит для представления сложных структур памяти. На его наличие указывает код 100 в *поле* r/m *постбайта*. *SIB-байт* имеет следующую структуру (рис. 3.7):

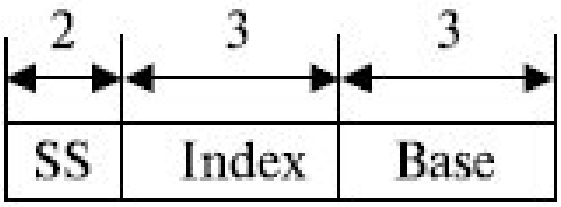

Рис. 3.7. Формат SIB-байта

Здесь SS - *поле* масштаба, Index задает номер одного изРОН, используемого в качестве индексного регистра (*регистр ESP* не может быть регистром индекса). *Поле Base* в комбинации с полем *mod постбайта* указывает *регистр* базы и смещение для индексированных операндов.

Применение *SIB*-байта позволяет формировать смещение в сегменте, иногда называемое *эффективным адресом* (ЭА), для операндов следующим образом:

$$\text{лещение в сегменте}) = [base] + [index]^{ss} + disp \quad 1)$$

где [base] - *значение* базового регистра, [index] - *значение* индексного регистра, ss - величина масштабного множителя, disp - *значение* смещения, закодированного в самой команде. В качестве базы или индекса может быть использован любой **регистр общего назначения** *микропроцессора*. Величина индекса может быть умножена на масштабный коэффициент (1, 2, 4 или 8), что дает возможность ссылки на элемент массива или записи соответствующей длины.

Смещение disp кодируется как величина со знаком в *дополнительном коде*. Его *длина* определяется значением бита D в **дескрипторе** сегмента, битом w в первом байте команды и наличием или отсутствием префикса разрядности адреса согласно табл. 3.1.

Этот механизм отражает основные усовершенствования в *способах адресации* операндов для 32-разрядной архитектуры *IA-32 по* сравнению с архитектурой *x86*. Различные комбинации слагаемых в выражении (3.1) дают следующие способы адресации памяти:

- прямая (только смещение),
- косвенная (только база),

- базовая относительная (база + смещение),
- индексная (индекс с масштабом),
- индексная со смещением (индекс с масштабом + смещение),
- базовая индексная (база + индекс с масштабом)
- относительная базовая индексная (база + индекс с масштабом + смещение).

Главные особенности формата команд МП с архитектурой *IA-32 по* сравнению с 16-разрядным *микропроцессором*:

- возможность использования любого из **регистров общего назначения** в любом из режимов адресации;
- возможность использования 32-разрядных непосредственных операндов и смещений при относительных режимах адресации наряду с имевшимися ранее 8- и 16-разрядными;
- добавление еще одного режима адресации - относительного базового индексного с масштабированием.

Сегментная организация памяти в защищенном режиме

В основе сегментной модели памяти лежит разделение ее на независимые адресные пространства переменной длины - **сегменты**. Для программы *адресное пространство* разделено на блоки смежных адресов, называемых сегментами, а *программа* может обращаться только к данным, находящимся в этих сегментах. Внутри сегментов применяется линейная *адресация*, то есть *программа* может обращаться к байту 0, байту 1 и т. д. Такая *адресация* осуществляется относительно начала сегмента, и *физический адрес*, ассоциируемый, например, с программным адресом 0, *по* существу, скрыт от программиста. Этот подход к управлению памятью опирается на тот факт, что программы обычно логически разделяются на области (*сегменты*) кода, данных и стека. При этом упрощается изоляция программ друг от друга в мультипрограммном режиме работы.
Каждый **сегмент** имеет свое целевое назначение. Каждая задача имеет непосредственный *доступ* к трем основным сегментам: кода, данных и стека, определяемых *сегментными регистрами* CS, *DS* SS соответственно, и к трем дополнительным сегментам данных, определяемых *сегментными регистрами* ES, FS, GS. Описания этих сегментов содержатся в их **дескрипторах**.
Любая *программа*, независимо от уровня ее привилегий, не может обращаться к сегменту до тех пор, пока он не описан с помощью дескриптора, а сам *дескриптор* не помещен в таблицу дескрипторов.

Дескрипторы хранятся либо в **глобальной таблице дескрипторов** (Global *Descriptor* Table - *GDT*), либо в локальных таблицах дескрипторов **(Local Descriptor Table - LDT)**. В **GDT** содержатся дескрипторы сегментов, которые доступны всем активным задачам, имеющимся в системе на данный момент. *GDT* может содержать любые дескрипторы сегментов, за исключением дескрипторов прерываний и ловушек. Обычно *GDT* включает

дескрипторы сегментов кодов и данных операционной системы, сегментов состояния задач и дескрипторы сегментов, содержащих локальные таблицы дескрипторов. *Микропроцессорная система* имеет единственную глобальную таблицу дескрипторов.

Локальная таблица дескрипторов *LDT* используется для хранения дескрипторов, доступных только данной задаче. Их количество определяется количеством активных задач в системе.

С точки зрения расположения в памяти, локальные таблицы дескрипторов представляют собой обычные *сегменты*. Они могут накладываться друг на друга, частично пересекаться. Это приводит к тому, что отдельные *сегменты*, описанные дескрипторами в своих *LDT*, могут разделяться несколькими задачами (рис. 3.8).

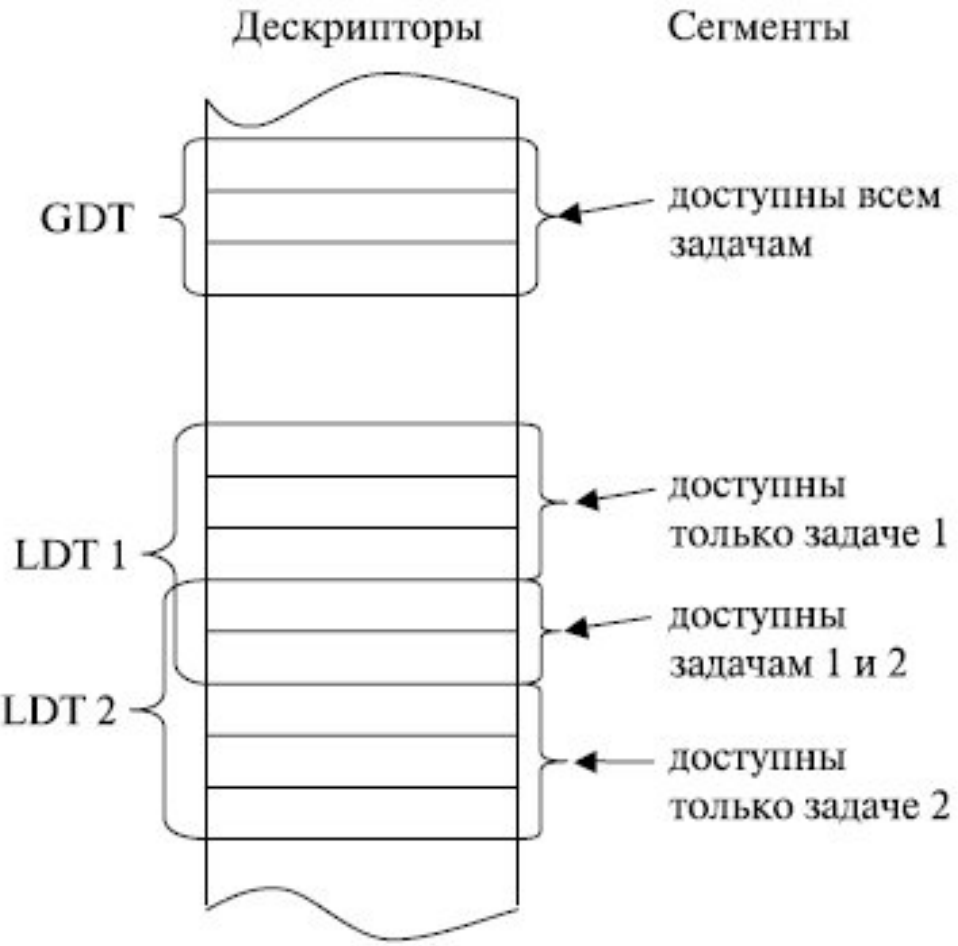

Рис. 3.8. Описание сегментов в таблицах дескрипторов

Для нахождения **дескриптора** в таблице дескрипторов используется **селектор**, который содержится в одном из сегментных регистров. **Селектор** представляет собой 16-разрядое *слово*, которое разбито на 3 поля (рис. 3.9):

- TI (Table *Indicator* - **индикатор таблицы**) показывает, к какой таблице идет обращение: TI = 0 - дескриптор находится в глобальной таблице дескрипторов *GDT*, TI = 1- в локальной таблице *LDT*;

- Index: **поле индекса** - номер дескриптора в соответствующей таблице дескрипторов;
- *RPL* (Request *privilege level* - **уровень привилегий запроса**). При обращении сравнивается с полем *DPL* в байте доступа дескриптора.

Обращение разрешается, если *уровень привилегий* запроса не ниже, чем *уровень привилегий* дескриптора.

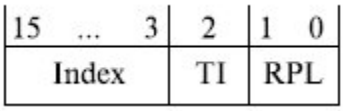

Рис. 3.9. Формат селектора

Максимальное количество дескрипторов, находящихся в таблице дескрипторов, определяется длиной поля Index селектора и равно 2^{13}. Так как каждый *дескриптор* имеет длину 8 *байт*, максимальный объем любой таблицы дескрипторов составляет 2^{16} *байт*. Каждая из таблиц дескрипторов имеет *регистр* (GDTR для глобальной таблицы и LDTR для локальной), определяющий ее положение в памяти. *Регистр* GDTR содержит 48 разрядов, из которых 32 задают базовый *адрес* **глобальной таблицы дескрипторов**, а 16 указывают ее объем в байтах (границу таблицы). Для определения положения **дескриптора** относительно начала таблицы его номер (*поле* Index селектора) 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 содержащемуся в GDTR базовому адресу, в результате чего образуется *физический адрес* выбираемого дескриптора (рис. 3.10).

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*). *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.

Для обращения к **локальной таблице дескрипторов** предназначен 16-разрядный *регистр* LDTR. Он содержит **селектор**, определяющий размещениев *GDT* дескриптора используемой локальной таблицы дескрипторов.

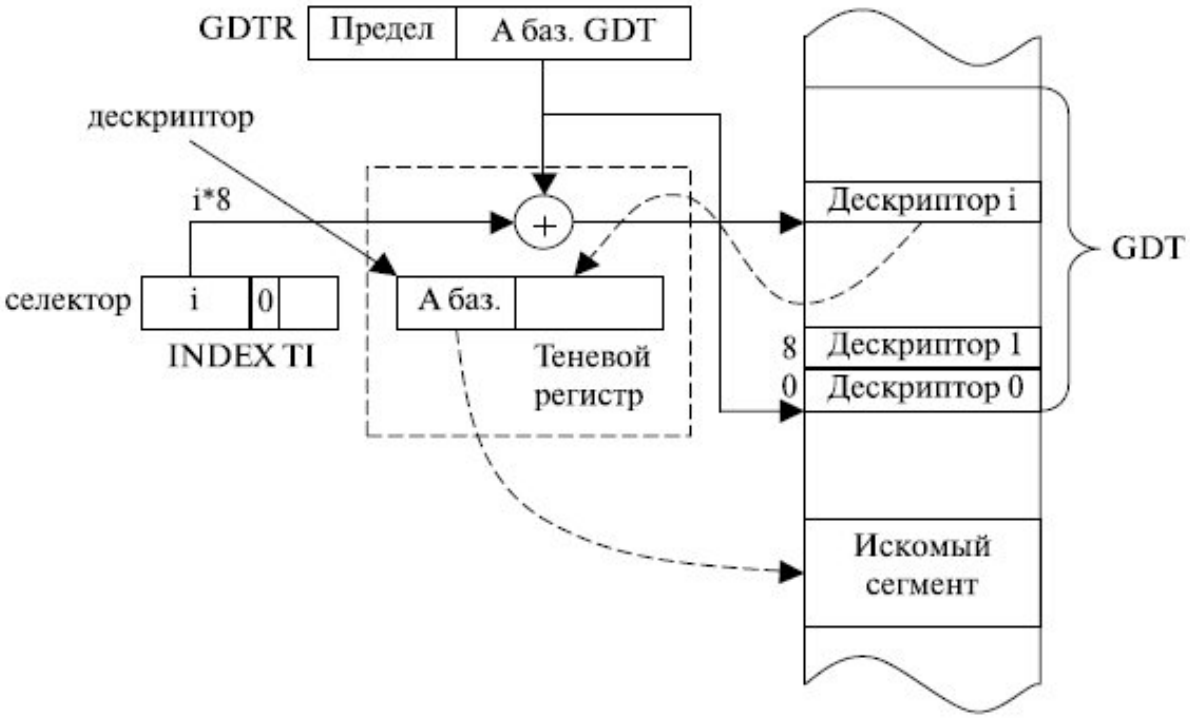

Рис. 3.10. Получение дескриптора, находящегося в глобальной таблице дескрипторов GDT

Такая структура упрощает работу с таблицами *LDT*. Благодаря описанию *LDT* с помощью селектора эти таблицы превращаются в обычные *сегменты* памяти и, в частности, могут размещаться в любых областях памяти, участвовать в *свопинге* и т. п. Внутри процессора с регистром LDTR ассоциируется так называемый "теневой *регистр*", в котором и хранится *дескриптор LDT* текущей задачи. Это ускоряет в последующем обращение к локальной таблице дескрипторов текущей задачи. При переключении с одной задачи на другую для замены используемой *LDT* достаточно загрузить в *регистр* LDTR селектор новой *LDT*, а *процессор* уже автоматически загрузит в теневой *регистр дескриптор* новой *LDT* при первом обращении к нему.

Если в селекторе **индикатор таблицы** TI = 1, то *дескриптор* сегмента выбирается из **локальной таблицы дескрипторов**. Процесс определения адреса сегмента в этом случае представлен на рис. 3.11.

Он более сложен *по* сравнению с получением дескриптора из глобальной таблицы дескрипторов и проходит следующие этапы:

1. Анализируем, к какой из двух возможных таблиц дескрипторов идет обращение. Если в селекторе TI = 1, то обращение идет к локальной таблице дескрипторов.

2. Находим дескриптор локальной таблицы дескрипторов в глобальной таблице дескрипторов.

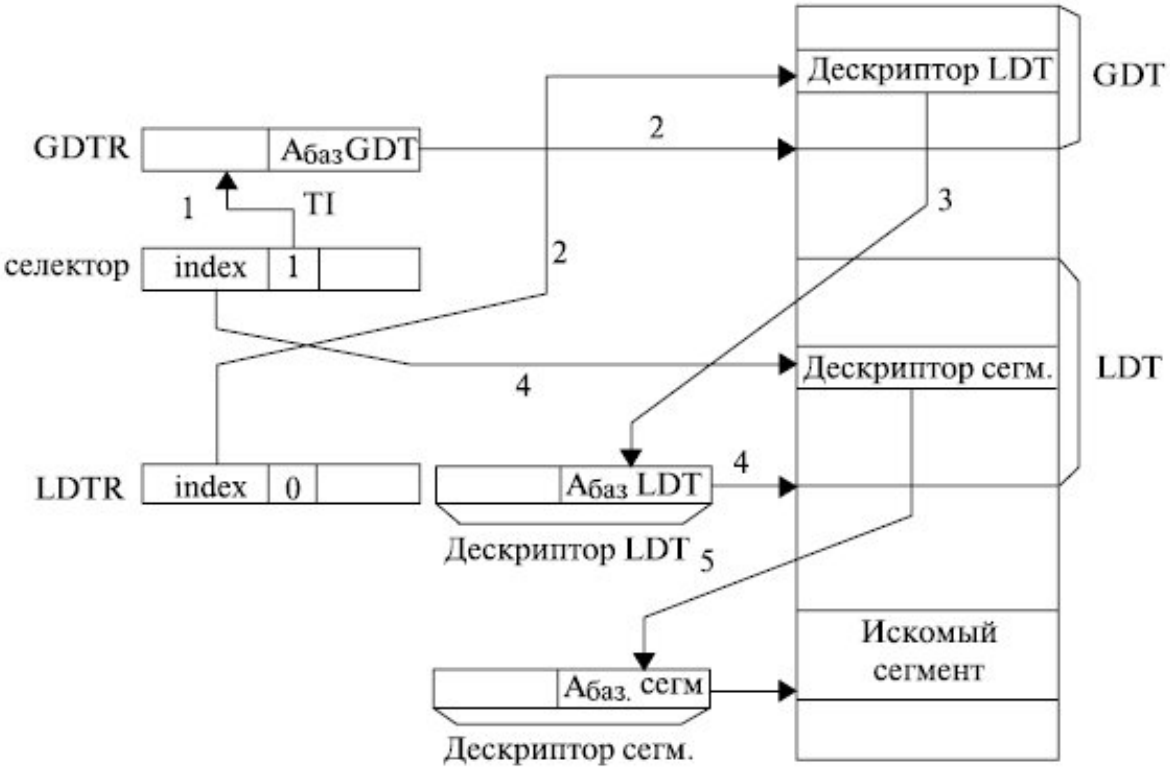

Рис. 3.11. Получение дескриптора, находящегося в локальной таблице дескрипторов LDT

3. Считываем дескриптор *LDT* в "теневой" регистр регистра LDTR *микропроцессора*. После считывания дескриптора в "теневой" регистр дальнейшее обращение к сегменту аналогично обращению к *GDT*, где вместо *GDT* R используются поля базового адреса и предела из дескриптора LDTR, находящегося в "теневом" регистре.
4. По адресу локальной таблицы дескрипторов, находящемуся в ее дескрипторе, и номеру дескриптора из обрабатываемого селектора находим дескриптор сегмента в локальной таблице дескрипторов.
5. Записываем дескриптор сегмента в "теневой" регистр сегментного регистра *микропроцессора* для ускорения последующих обращений к искомому сегменту.

Таким образом, при обращении к сегменту через таблицу *LDT* появляется дополнительный *уровень вложенности*, снижающий *быстродействие микропроцессора*. "Теневые" регистры *микропроцессора* частично обеспечивают решение этой проблемы.

Поле адреса дескриптора, полученного из локальной или глобальной таблицы дескрипторов, определяет начало искомого сегмента. При суммировании полученного базового адреса сегмента и смещения в сегменте получается линейный *адрес* искомой ячейки памяти.

В случае если режим страничной адресации выключен (в регистре CR0 *бит* PG = 0), полученный линейный *адрес* равен *физическому адресу* искомого операнда или команды.

Рассмотрим подробнее процесс получения адреса операнда на примере команды

MOV EAX, [ECX+*ESI*+20h].

В этой команде нет специальных указаний об использовании сегмента, поэтому она обращается к текущему сегменту данных, селектор которого *по* умолчанию находится в *сегментном регистре DS*. Пусть (*DS*) = 0000000000011.0.XXb.

Формирование *физического адреса* операнда включает следующие действия (для *сегментированного* ЛАП):

1. Образовать *эффективный адрес* (вычислить смещение в сегменте):

   EA = (ECX)+(*ESI*)+20h.

2. Выбрать 3-й дескриптор (Index = 3) из *GDT* (TI = 0).

   Для этого:

   - считать базовый адрес глобальной таблицы дескрипторов ($\text{А}_{\text{баз}GDT}$) из GDTR ;
   - вычислить $\text{А}_{\text{баз}GDT} + (Index) * 8$;
   - обратиться по полученному адресу в память и считать нужный дескриптор.
3. Получить **линейный адрес**: $\text{ЛА} = EA + A_{\text{баздескр.3}}$, где $A_{\text{баздескр.3}}$ - базовый адрес сегмента из считанного дескриптора с номером 3.
4. Так как при сегментной организации адресного пространства **линейный адрес** равен физическому, следует обратиться к памяти по сформированному адресу и передать двойное **слово** в EAX.

При TI = 1 потребовалось бы еще одно обращение к памяти для считывания дескриптора *LDT* из *GDT*.

Чтобы сократить число обращений к памяти (а такой процесс должен проходить и при считывании кода каждой команды), в микропроцессорах с архитектурой *IA-32* применяется так называемое *кэширование* дескрипторов. *Кэширование* опирается на тот факт, что обращение к памяти производятся гораздо чаще, чем изменение используемых **сегментов** и переключение задач. Поэтому с каждым регистром, содержащим **селекторы** тех или иных сегментов (*сегментные регистры*, а также регистры **локальной таблицы дескрипторов** LDTR и *регистр* задач TR), ассоциируются "теневые", или *кэш*-регистры.

При первом считывании **дескриптора**, определяемого данным **селектором**, *процессор* автоматически считывает (кэширует) нужный *дескриптор* в соответствующий "теневой" *регистр*. Поскольку теперь *дескриптор* находится внутри МП, для получения линейного адреса памяти потребуется только сформировать *эффективный адрес* и просуммировать его с базовым адресом сегмента из нужного "теневого" регистра.

Так как *программа* обычно редко модифицирует регистры с **селекторами**, в **защищенном режиме** она будет выполняться примерно с такой же скоростью, как и в реальном режиме.

Помимо локальной и глобальной таблиц дескрипторов в *микропроцессорной системе* используется также **дескрипторная таблица прерываний** (*IDT*). Она содержит дескрипторы специальных *системных объектов*, которые определяют точки *входа в процедуры* обработки прерываний.

IDT служит заменой *таблицы векторов прерываний* 16-разрядного *микропроцессора*. Обращение к ней проводится только аппаратными средствами МП при возникновении *аппаратных прерываний* или особых случаев при выполнении программы. Программы самостоятельно не могут обратиться к *IDT*, так как единственный *бит* **индикатора таблицы** в **селекторе**сегмента идентифицирует только *GDT* или *LDT*.

До перевода процессора в **защищенный режим** необходимо создать таблицы *GDT* и *IDT* и соответственно инициализировать регистры GDTR и *IDT* R. Таблицы *GDT* и *IDT*определяются при загрузке в соответствующие регистры GDTR и *IDT* R базового адреса и предела. Это действие осуществляется только один раз в ходе подготовки к переходу в **защищенный режим**, и в дальнейшем содержимое GDTR и IDTR не изменяется. Это значит, что местонахождение таблиц *GDT* и *IDT* в известном смысле фиксировано, и они не могут участвовать в *свопинге*.

Страничная организация памяти

Страничная организация памяти применяется только в **защищенном режиме**, если в *регистре управления* CR0 *бит* PG = 1.

Основное применение *страничного преобразования адреса* связано с реализацией виртуальной памяти, которая позволяет программисту использовать большее *пространство* памяти, чем физическая *основная память*.

Принцип **виртуальной памяти** предполагает, что *пользователь* при подготовке своей программы имеет дело не с физической ОП, действительно работающей в составе компьютера и имеющей некоторую фиксированную емкость, а с виртуальной (кажущейся) одноуровневой памятью, емкость которой равна всему адресному пространству, определяемому размером адресной шины ($L_{ша}$) компьютера:

$$V_{\text{вирт}} >> V_{\text{физ}} V_{\text{вирт}} = 2^{L\text{ша}}$$

Для 32-разрядного *микропроцессора*:

$$V_{\text{вирт}} = 2^{32} = 4\text{Гбайт}$$

Программист имеет в своем распоряжении *адресное пространство*, ограниченное лишь разрядностью адресной шины, независимо от реальной емкости оперативной памяти компьютера и объемов памяти, которые используются другими программами, параллельно обрабатываемыми в мультипрограммной ЭВМ.

Виртуальная память, обеспечивая возможность программисту обращаться к очень большому объему непрерывного адресного пространства, предоставляемого в его монопольное распоряжение, обладает обычными свойствами: побайтовая *адресация*, *время доступа*, сравнимое со временем доступа к оперативной памяти.

На всех этапах подготовки программ, включая загрузку в *память, программа* представляется в *виртуальных адресах*, и лишь при выполнении машинной команды *виртуальные адреса*преобразуются в физические. Для каждой программы, выполняемой в мультипрограммном режиме, создается своя *виртуальная память*. Каждая *программа* использует одни и те же *виртуальные адреса* от нулевого до максимально большого в данной архитектуре.

Для преобразования виртуальных адресов в физические физическая и *виртуальная память* разбиваются на блоки фиксированной длины, называемые **страницами**. Объемы виртуальной и физической страниц совпадают. Страницы виртуальной и физической памяти нумеруются. Отсутствующие в физической памяти страницы обычно хранятся во внешней памяти. Фиксированный размер всех страниц позволяет загрузить любую нужную виртуальную страницу в любую физическую.

Как отмечалось выше, при страничном представлении памяти виртуальный (*логический) адрес* представляет собой номер *виртуальной страницы* и смещение внутри этой страницы. В свою *очередь, физический адрес* - это номер *физической страницы* и смещение в ней.

Правила перевода номеров виртуальных страниц в номера *физических страниц* обычно задаются в виде таблицы *страничного преобразования*. Такие таблицы формируются системой управления памятью и модифицируются каждый раз при перераспределении памяти. *Операционная система* постоянно отслеживает состояние виртуальных страниц той или иной программы и определяет, находится ли она в оперативной памяти, и если находится, то в каком конкретно месте. Прикладные программы не касаются процесса *страничного преобразования адреса* и могут использовать все *адресное*

пространство. *Процессор* автоматически формирует особый случай неприсутствия, когда *программа* обращается к странице, отсутствующей в физической памяти. При обработке этого особого случая ОС загружает затребованную страницу из внешней памяти, при необходимости отправляя некоторую другую страницу на *диск*(процесс свопинга).

Перевод виртуальных адресов в физические проиллюстрирован на рис. 3.12.

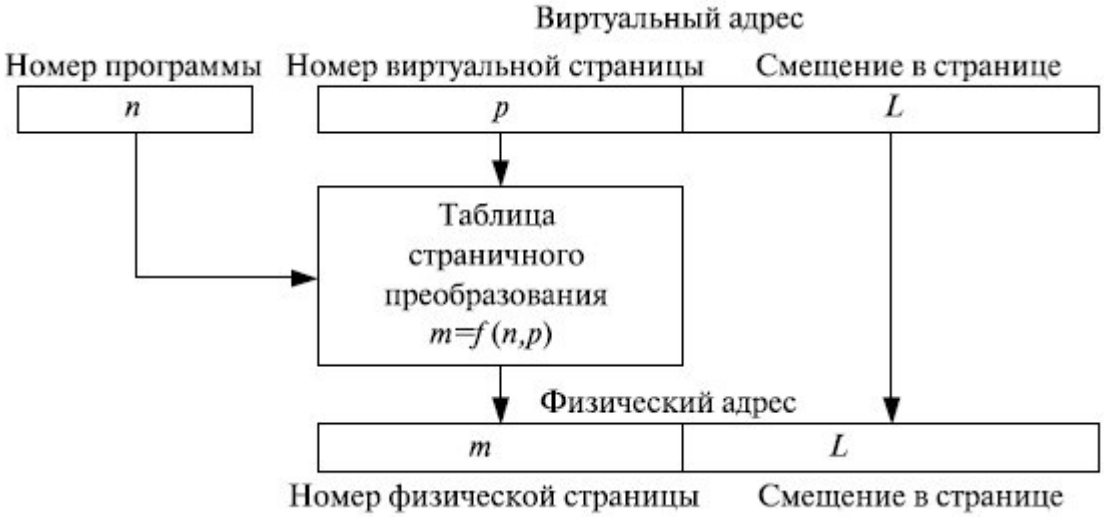

Рис. 3.12. Принцип преобразования виртуального страничного адреса в физический

Рассмотрим пример преобразования адреса *виртуальной страницы* в *адрес физической страницы*. Пусть *компьютер* использует *адресное пространство*, предполагающее *разбиение* на страницы объемом $V_{стр}$ = 1I, и имеет оперативную *память* $V_{ОЗУ}$ = 3 страницы. Пусть на компьютере одновременно выполняются четыре программы, имеющие следующее количество страниц: V^A = 2, V^B = 1, V^C = 3, V^D = 2. Переключение между программами происходит через время кванта t^k = 1. *Время выполнения* каждой страницы любой программы составляет $t = 2t^k$. Полагаем, что страницы программ загружаются в оперативную *память по* мере необходимости и *по* возможности в свободные области *ОЗУ*. Если вся *память* занята, то новая страница замещает ту, к которой дольше всего не было обращений.

При таких условиях *таблица* загрузки оперативной памяти и таблицы *страничного преобразования* для каждой программы будут иметь вид, представленный в табл. 3.2.

В таблице распределения оперативной памяти выделены номера активных в данном такте страниц. В таблицах *страничного преобразования* прочерками отмечены ситуации, когда данная *виртуальная страница* отсутствует в оперативной памяти.

Таблица 3.2. Пример страничного распределения памяти в мультипрограммной ЭВМ

ница	ы
	амическое распределение оперативной памяти
0	
	ица *страничного преобразования* для программы A
	ица *страничного преобразования* для программы B
	ица *страничного преобразования* для программы C
	ица *страничного преобразования* для программы D

Если каждая страница имеет объем 1000 адресуемых ячеек, то, например, в такте 9 обращение *по виртуальному адресу* 1100 программы А (*виртуальная страница* 1, смещение в странице равно 100) приведет к обращению *по физическому адресу* 2100 (*физическая страница* 2, смещение в *физической странице* такое же, как и в виртуальной, то есть 100).

Рассмотрим теперь применение этих общих принципов *страничного преобразования адреса* в *микропроцессоре* с архитектурой *IA-32* при объеме страницы в 4 Кбайт.

Основой *страничного преобразования* служит 32-разрядный **линейный адрес**, полученный на этапе сегментного преобразования **логического адреса**. *Страничное преобразование*выполняется при значении бита PG = 1 в управляющем регистре CR0.

В этом случае старшие 20 разрядов **линейного адреса** фактически представляют собой номер *виртуальной страницы*. Однако при прямом одноступенчатом преобразовании этого номера в номер *физической страницы* необходима *таблица* из 2^{20} элементов длиной 4 байта каждый (20-разрядный номер страницы плюс некоторая дополнительная *информация*), т. е. 4 Мбайт. В мультипрограммной среде такая *таблица* может потребоваться для каждой задачи. Эта *таблица* должна постоянно храниться в оперативной памяти, чтобы существенно не увеличивать время формирования *физического адреса*. Для этих целей потребуется постоянное резервирование существенной части емкости *ОЗУ*, что на этапе появления первых ЭВМ на основе МП с архитектурой *IA-32* было практически невозможно.

Вместо этого *микропроцессор* использует двухступенчатое *страничное преобразование адреса*. Корневая страница, называемая **каталогом таблиц страниц** (КТС), содержит 1024 32-разрядных **элемента каталога таблиц страниц** (ЭКТС - PDE page *directory entry*). Каждый из них адресует подчиненную **таблицу страниц** (ТС), то есть всего допускается до 1024 *подчиненных таблиц* страниц. Каждая из *таблиц страниц* содержит 1024 32-разрядных **элемента таблицы страниц** (ЭТС - PTE page *table entry*), каждый из которых и адресует физическую страницу. Таким образом, общее количество адресуемых *физических страниц* равно 2^{20}, то есть все *виртуальное адресное пространство* (4 Кбайт * 2^{20} элементов = 2^{32} *байт*). Каждая *таблица* занимает 1024 * 4 = 4 Кбайт, то есть ровно 1 страницу. Общий объем таблиц, используемых для *страничного преобразования*, не уменьшился, а даже несколько возрос за счет использования каталога *таблиц страниц*. Однако, во-первых, практически всегда в системе этот размер можно существенно уменьшить за счет того, что некоторые **линейные адреса**никогда не будут сформированы (а эту информацию дают **таблицы дескрипторов** сегментов), и для них не нужно создавать таблицу страниц. А во-вторых, в оперативной памяти должны постоянно находиться лишь **каталог таблиц страниц и таблица страниц** выполняемой в настоящее время программы. Остальные **таблицы страниц** могут временно храниться во внешней памяти.

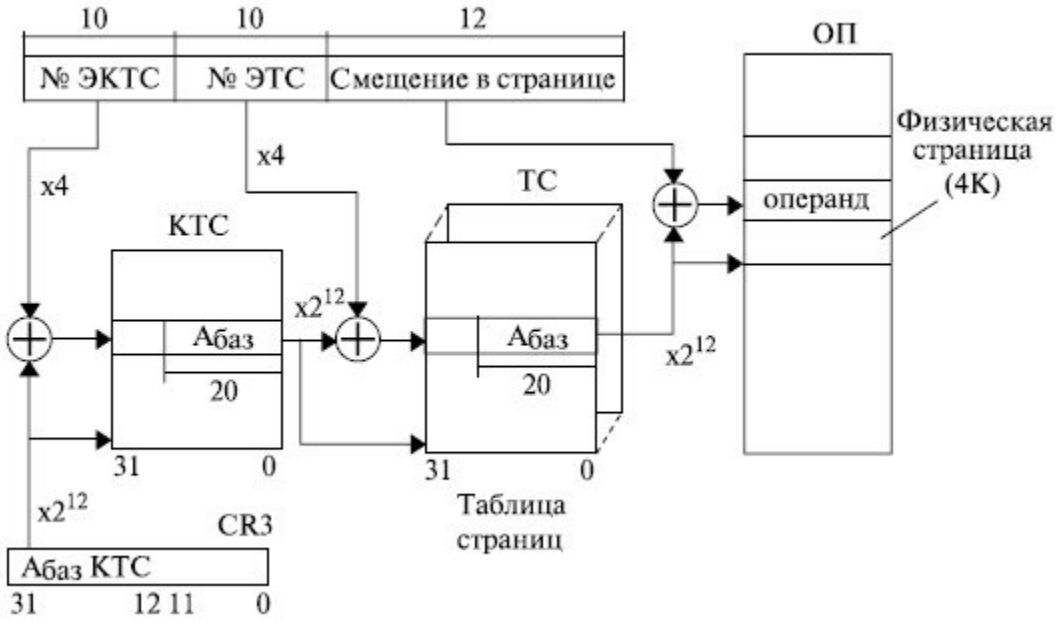

Рис. 3.13. Страничное преобразование линейного адреса в физический

Таким образом, преобразование линейного адреса в физический имеет вид, представленный на рис. 3.13.

Старшие 20 разрядов **линейного адреса** разбиваются на два 10-разрядных поля: *поле* **номера элемента каталога таблиц страниц** и *поле* **номера элемента таблицы страниц**. Так как и **каталог таблиц страниц**, и каждая **таблица страниц** занимают ровно 1 страницу и выровнены *по* границе страницы, то младшие 12 разрядов их базового адреса равны нулю, и для определения их *физического адреса* достаточно 20-разрядного поля.

Для **каталога таблиц страниц** его 20-разрядный *адрес* находится в *регистре управления* CR3. КТС постоянно находится в памяти и не участвует в *свопинге*.

Старшие 20 разрядов *физического адреса* **таблицы страниц** извлекаются из ЭТС. Структуры элемента КТС и элемента ТС схожи (рис. 3.14).

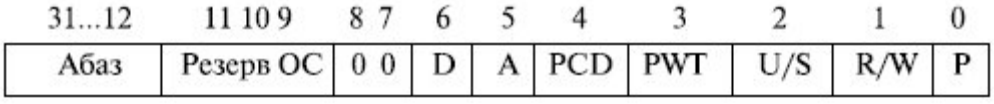

Рис. 3.14. Структура элементов каталога таблиц страниц и таблицы страниц

Старшие 20 разрядов элемента дают базовый *адрес* **таблицы страниц** (в ЭКТС) или *физической страницы* (в ЭТС). Биты P, A, R/W и U/S имеют определенное сходство с аналогичными атрибутами дескриптора сегмента, другие биты имеют специфическое назначение.

Бит присутствия P показывает, отображается ли *адрес* страничного кадра (*таблицы страниц* или *страницы памяти*) на страницу в физической памяти. При P =

1 страница присутствует в *ОЗУ*. При P = 0 страницы в памяти нет, и обращение к этой странице вызывает *прерывание* типа "*страничное нарушение*".

Бит доступа A устанавливается *микропроцессором* в состояние A = 1 при обращении к данному страничному кадру для записи или чтения информации.

Бит модификации D (Dirty - "грязный") устанавливается процессором равным 1 в элементе ЭТС при записи на данную страницу. Для элементов каталога *таблиц страниц значение*бита D является неопределенным. При загрузке страницы в *память операционная система* сбрасывает *бит* D. Если при необходимости выгрузки страницы во *внешнюю память* оказывается, что для нее D = 0, это означает, что к странице в памяти не было обращений на *запись*, во внешней памяти есть ее точная копия, и реально передавать страницу из памяти на *диск* не нужно. Тем самым экономится время при *свопинге*.

Бит чтения-записи R/W и *бит* U/S (user/*supervisor* - *пользователь*/супервизор) определяют *права доступа* к *таблице страниц* или к странице для программ с различными уровнями привилегий. Для страниц существует только 2 уровня привилегий: уровень *супервизора* (U/S = 0), соответствующий значению *DPL* сегмента 0, 1, 2, и уровень пользователя (U/S = 1), соответствующий *DPL* = 3. Если к странице осуществляется *запрос* с уровнем привилегий 3 (программы пользователя), то при значении U/S = 0 ему запрещается *доступ* к соответствующей таблице или странице. Если U/S = 1, то при значении R/W = 0 разрешается только чтение таблицы или страницы, а при R/W = 1 - и чтение, и *запись*.

При запросах с большими привилегиями (системные программные уровни 0, 1, 2) допускается *запись* и чтение таблиц и страниц при любых значениях U/S, R/W (табл. 3.3).

Таблица 3.3. Допустимые действия со страницами на различных уровнях привилегий

W	Допустимо для уровня 3	Допустимо для уровней 0, 1, 2
	чего	ение/запись
	ение	ение/запись
	ение/запись	ение/запись

Биты PWT и *PCD* используются для управления работой *кэш*-памяти при страничной адресации. *Бит PCD* - запрещение кэширования страницы. При *PCD* =

1 *кэширование* запрещено. *Бит* PWT - *бит* обратной записи страниц. Определяет метод обновления внешней *кэш*-памяти (*кэш* 2-го уровня). При PWT= 1 - обновление проводится методом сквозной записи (как для внутреннего кэша), при PWT = 0 - методом обратной записи.

Биты 9…11 в ЭКТС и ЭТС зарезервированы за операционной системой. *Процессор* никогда не использует и не изменяет эти биты. Разработчики ОС могут привлечь эти биты для хранения информации о "старении" страниц, чтобы определять страницы, подлежащие замене из внешней памяти, и для других целей.

Старшие 10 разрядов **линейного адреса** совместно с содержимым *регистра управления* CR3 определяют необходимый **элемент каталога таблиц страниц**. Следующие 10 разрядов линейного адреса содержат номер элемента в выбранной *таблице страниц*.

Так как и ЭКТС, и ЭТС имеют длину 4 байта, для получения смещения начала элемента относительно начала соответствующей таблицы необходимо его номер умножить на 4.

Последние 12 разрядов **линейного адреса** содержат смещение в странице. Таким образом, сумма смещения в странице и базового адреса страницы, извлеченного из ЭТС, дает *физический адрес* искомого байта.

Буфер ассоциативной трансляции страничного адреса

Страничная организация памяти требует двух дополнительных обращений к памяти: для считывания ЭКТС и ЭТС. Чтобы не ухудшать *производительность* процессора, в схемы управления *страничным преобразованием адреса* встроен **буфер ассоциативной трансляции страничного адреса** (Translation Lookaside Buffer - *TLB*).

Когда *программа* формирует **линейный адрес**, который отображен на находящийся в *TLB* элемент PTE, преобразование выполняется без дополнительных обращений к памяти.

TLB представляет собой *память* с ассоциативной выборкой, которая содержит 20-разрядные базовые адреса 32 страниц. Каждый из базовых

Рис. 3.15. Структура буфера TLB ассоциативной трансляции страничного адреса

адресов имеет свой признак (*тег*), в качестве которого используются старшие разряды **линейного адреса**.

Программы не могут управлять кэшированием элементов PTE. *Диспетчер* памяти *MMU* кэширует каждый используемый элемент PTE до заполнения буфера. При заполненном *TLBпроцессор* может найти страничную информацию для 128 Кбайт физической памяти (32 страницы *по* 4 Кбайт). При такой емкости *TLB* доля *кэш*-попаданий составляет в среднем 98 %.

TLB состоит из модуля основной памяти, блока *LRU*, используемого при ее обновлении, и логики обслуживания (рис. 3.15).

Основной *модуль* памяти содержит 8 блоков, каждый из них содержит информацию о 4 страницах, для которых ранее производилось преобразование страничного адреса. Таким образом, основной *модуль* содержит 32 строки, позволяющие непосредственно, без обращения к КТС и *таблице страниц*, определить базовый *физический адрес* для одной из 32 страниц, параметры которой загружены в *TLB*.

Каждая строка *TLB* содержит информацию, необходимую для ее выбора (*тег*, биты, определяющие *доступ* к странице), и информацию о выбираемой странице (базовый *адрес*, атрибуты). Ее структура представлена на рис. 3.16.

Рис. 3.16. Формат строки модуля основной памяти TLB

Поля базового адреса *физической страницы* и атрибуты D, U/S, R/W, PWT, *PCD* аналогичны соответствующим полям в ЭТС. *Поле* тега содержит старшие разряды линейного адреса, для которого номер *виртуальной страницы* преобразовывался в номер *физической страницы*. *Бит* V определяет *достоверность* хранимой в данной строке информации (V = 0 -незаполненная строка, V = 1 - достоверная *информация*).

После формирования **линейного адреса** 3 младших разряда поля номера *виртуальной страницы* (биты 14….12 линейного адреса) определяют номер одного из 8 блоков *TLB*. Старшие 17 разрядов (биты 31…15) ЛА сравниваются одновременно ассоциативным образом с 17 битами тегов, содержащихся в каждой из 4 строк выбранного блока, с учетом бита достоверности V каждой строки. Если для некоторой строки сравнение прошло успешно, значит, эта строка содержит информацию *по* искомой *физической странице*. То есть для нее преобразование *виртуального адреса* в физический уже выполнялось, и результат этого преобразования содержится в найденной строке. Указанный в найденной строке базовый *адрес* обеспечивает выборку нужной *физической страницы*.

Если совпадения не было, то базовый *адрес* нужной страницы отсутствует в *TLB*, и преобразование страничного адреса проводится обычным путем с обращением к КТС и ТС. Полученная из *таблицы страниц информация* о *физической странице* вместе с 17 старшими разрядами **линейного адреса** (*тег*) заносится в строку одного из блоков *TLB*, номер которого задается битами 14…12 линейного адреса. Выбор одной из 4 строк адресованного блока, в которую заносится новое содержимое (*тег*, базовый *адрес* и др.), определяется принятым механизмом замещения.

Для *TLB* принят механизм замещения наиболее долго неиспользуемой строки (least recently used - *LRU*). При этом выбор замещаемой строки в блоке определяется битами B0, B1, B2 (биты *LRU*), которые хранятся в дополнительном модуле памяти *TLB*. Этот *модуль* содержит 8 строк *по* 3 разряда каждая. Строка модуля соответствует одному из блоков основной памяти *TLB*. Логика обслуживания *TLB* переопределяет биты строки *LRU по* мере обращения к соответствующим строкам блока *TLB*.

В любой момент сочетание разрядов B0…B2 указывает, к какой из строк данного блока дольше всего не было обращения. Именно эта строка и замещается при

необходимости записи новой строки, если все 4 строки блока уже заполнены. Заполненность строки определяется значением бита ее достоверности V.

При инициализации все биты B0, B1, B2 для всех блоков сбрасываются в "0". В ходе работы биты B0, B1, B2 принимают значения в соответствии с алгоритмом *LRU* следующим образом. Если последнее обращение в текущем блоке производилось к строкам L0 или L1, устанавливается B0 = 1. Если же оно осуществлялось к L3 или L4, то B0 = 0. При проверке пары строк L0:L1 в случае последнего обращения к L0 устанавливается B1 = 1, в противном случае B1 = 0. При проверке пары строк L2:L3 в случае последнего обращения к L2устанавливается B2 = 1, в противном случае B2 = 0 (табл. 3.4).

Блица 3.4. Порядок изменения бит в строке *LRU*

Бит *LRU*	Последнее обращение
= 1	или L1
= 0	или L3
= 1	
= 0	
= 1	
= 0	

При поиске подлежащей замене строки в блоке, если все 4 строки достоверны, вначале проверяется, к какой из пар строк L0:L1 или L2:L3 производилось последнее обращение. Затем производится *анализ* внутри пары, отобранной на предыдущем этапе. Таким образом, замена строки в блоке *TLB* в случае, когда они все достоверны, проводится в соответствии с табл. 3.5.

Блица 3.5. Порядок замены строк в блоке *TLB*

	Заменяемая строка

В принципе, для того чтобы задать номер одной из 4 строк, достаточно 2-разрядного кода. Но при этом механизм, определяющий строку, к которой дольше всего не было обращений, может оказаться достаточно сложным. Здесь же путем добавления всего лишь одного бита получаем прозрачный механизм как *по* установке этих разрядов, так и *по* определению строк, подлежащих замене. Хранение **элементов таблиц страниц** в *TLB* таит в себе опасность, связанную с модификацией ЭТС "на лету", то есть в ходе выполнения программы. Предположим, что в программе предусматривается вносить изменения в **каталог таблиц страниц** или в **таблицы страниц**. После подготовки необходимых таблиц и разрешения *страничного преобразования процессор* будет загружать ЭТС в *TLB* до заполнения. Затем *по* мере необходимости *процессор* заменяет старые элементы новыми. Пусть теперь *программа* изменила некоторый ЭТС, хранящийся в памяти. Если измененный элемент уже находится в *TLB*, *процессор* будет пользоваться его старым значением, так как он не может узнать о том, что этот элемент в памяти был модифицирован.

Для преодоления этой коллизии программисту необходимо после всей подготовки к изменению ЭТС сразу же перезагрузить *регистр* CR3, например, с помощью такой пары команд:

```
MOV EAX, CR3
MOV CR3, EAX
```

При любой перезагрузке регистра CR3 все биты достоверности в *TLB* сбрасываются (V = 0), и МП будет вынужден проводить новые преобразования страничного адреса для всех страниц, загружая при этом в *TLB* и модифицированный ЭТС.

Аналогичные *операции* выполняются при отсутствии страницы в оперативной памяти (P = 0), но уже операционной системой. *Страничное нарушение* при отсутствии страницы в оперативной памяти вызывает следующие действия:

1. операционная система копирует запрошенную страницу с диска в оперативную память;
2. ОС загружает адрес страничного кадра в элемент *таблицы страниц* или каталога *таблиц страниц* и устанавливает P = 1. При этом могут быть установлены и другие биты, например, R/W ;
3. так как в *TLB* может оставаться копия старого ЭКТС или ЭТС, операционная система очищает его;
4. осуществляется *рестарт* команды, вызвавшей *страничное нарушение*.

Расширение объемов обрабатываемой информации, особенно мультимедийной, вместе с увеличением функциональных возможностей *микропроцессоров* и микропроцессорных систем в целом, привело к существенному изменению в параметрах страниц. Современные *микропроцессоры*, сохраняя главную особенность *страничной организации* постоянство объема страницы, обеспечивают возможность использования широкого диапазона их размеров. Так, в *микропроцессоре Itanium* на аппаратном уровне поддерживаются страницы емкостью 4/8/16/64/256 Кбайт, 1/4/16/64/256 Мбайт.

Краткие итоги. В лекции рассмотрено физическое и логическое *представление* памяти. Показан общий механизм формирования *физического адреса* при сегментно-страничном представлении адресного пространства. Рассмотрен порядок формирования адреса блоком *сегментации* и блоком *страничного преобразования микропроцессора*. Показаны способы, обеспечивающие сокращение времени такого преобразования.

Лекция 4. Организация и принципы работы кэш-памяти

Общие принципы функционирования кэш-памяти

Кэш-память (КП), или **кэш**, представляет собой организованную в виде ассоциативного *запоминающего устройства* (АЗУ) быстродействующую *буферную память* ограниченного объема, которая располагается между регистрами процессора и относительно медленной основной памятью и хранит наиболее часто используемую информацию совместно с ее признаками (тегами), в качестве которых выступает часть адресного кода.

В процессе работы отдельные блоки информации копируются из основной памяти в *кэш-память*. При обращении процессора за командой или данными сначала проверяется их наличие в КП. Если необходимая *информация* находится в кэше, она быстро извлекается. Это **кэш-попадание**. Если необходимая *информация* в КП отсутствует (**кэш-промах**), то она выбирается из основной памяти, передается в *микропроцессор* и одновременно заносится в *кэш-память*. Повышение быстродействия вычислительной системы достигается в том случае, когда **кэш-попадания реализуются намного чаще, чем кэш-промахи**.

Зададимся вопросом: "А как определить наиболее часто используемую информацию? Неужели сначала кто-то анализирует ход выполнения программы, определяет, какие команды и данные чаще используются, а потом, при следующем запуске программы, эти данные переписываются в *кэш-память* и уже тогда *программа* выполняется эффективно?" Конечно нет. Хотя в современных микропроцессорах имеется определенный механизм, который позволяет в некоторой степени реализовать этот принцип. Но в основном, конечно, *кэш-память* сама отбирает информацию, которая чаще всего используется. Рассмотрим, как это происходит.

Механизм сохранения информации в кэш-памяти

При включении микропроцессора в работу вся информация в его кэш-памяти недостоверна.

При обращении к памяти микропроцессор, как уже отмечалось, сначала проверяет, не содержится ли искомая информация в кэш-памяти.

Для этого сформированный им физический адрес сравнивается с адресами ячеек памяти, которые были ранее кэшированы из ОЗУ в КП.

При первом обращении такой информации в *кэш*-памяти, естественно, нет, и это соответствует **кэш-промаху**. Тогда *микропроцессор* проводит обращение к оперативной памяти, извлекает нужную информацию, использует ее в своей работе, но одновременно записывает эту информацию в *кэш*.

Если бы в *кэш-память* заносилась только востребованная микропроцессором в данный момент *информация*, то, скорее всего, при следующем обращении вновь произошел бы **кэш-промах**: вряд ли следующее обращение произойдет к той же самой команде или к тому же самому операнду. **Кэш-попадания** происходили бы лишь после того, как в КП накопится достаточно большой фрагмент программы, содержащий некоторые циклические участки кода, или фрагмент данных, подлежащих повторной обработке. Для того чтобы уже следующее обращение к КП приводило как можно чаще к **кэш-попаданиям**, передача из оперативной памяти в *кэш-память* происходит не теми порциями (байтами или словами), которые востребованы микропроцессором в данном обращении, а так называемыми **строками**. То есть *кэш-память* и оперативная *память* с точки зрения кэширования организуются в виде строк. *Длина строки*превышает максимально возможную длину востребованных микропроцессором данных. Обычно она составляет от 16 до 64 *байт* и выровнена в памяти по границе соответствующего раздела (рис. 4.1).

Рис. 4.1. Организация обмена между оперативной и кэш-памятью

Высокий *процент* **кэш-попаданий** в этом случае обеспечивается благодаря тому, что в большинстве случаев программы обращаются к ячейкам памяти, расположенным вблизи от ранее использованных. Это свойство, называемое **принципом локальности ссылок**, обеспечивает эффективность использования КП. Оно подразумевает, что при исполнении программы в течение некоторого относительно малого интервала времени происходит обращение к памяти в пределах ограниченного диапазона адресов (как по коду программы, так и по данным).

Например, микропроцессору для своей работы потребовалось 2 байта информации. Если строка имеет длину 16 *байт*, то в *кэш* переписываются не только нужные 2 байта, но и некоторое их окружение. Когда *микропроцессор* обращается за новой информацией, в силу локальности ссылок, скорее всего, обращение произойдет по соседнему адресу. Затем опять по соседнему, опять по соседнему и т. д. Таким образом, ряд следующих обращений будет происходить непосредственно к *кэш*-памяти, минуя оперативную *память* **(кэш-попадания)**. Когда очередной сформированный микропроцессором *физический адрес* выйдет за пределы строки *кэш*-памяти (произойдет **кэш-промах**), будет выполнена подкачка в *кэш* новой строки, и вновь ряд последующих обращений вызовет **кэш-попадания**.

Чем длиннее используемая при обмене между оперативной и кэшпамятью строка, тем больше *вероятность* того, что следующее обращение произойдет в пределах этой

строки. Но в то же время чем длиннее строка, тем дольше она будет перекачиваться из оперативной памяти в *кэш*. И если очередная *команда* окажется *командой перехода* или *выборка* данных начнется из нового массива, то есть следующее обращение произойдет не по соседнему адресу, то время, затраченное на передачу длинной строки, будет использовано напрасно. Поэтому при выборе длины строки должен быть разумный *компромисс* между соотношением времени обращения к оперативной и *кэш*-памяти и вероятностью достаточно удаленного перехода от текущего адреса при выполнении программы. Обычно *длина строки* определяется в результате моделирования аппаратно-программной *структуры системы*.

После того как в КП накопится достаточно большой объем информации, увеличивается *вероятность* того, что формирование очередного адреса приведет к **кэш-попаданию**. Особенно велика *вероятность* этого при выполнении циклических участков программы.

Старая *информация* по возможности сохраняется в *кэш*-памяти. Ее замена на новую определяется емкостью, организацией и стратегией обновления кэша.

Типы кэш-памяти

Если каждая строка *ОЗУ* имеет только одно фиксированное *место*, на котором она может находиться в *кэш*-памяти, то такая *кэш-память* называется памятью **с прямым отображением**.

Предположим, что *ОЗУ* состоит из 1000 строк с номерами от 0 до 999, а *кэш-память* имеет емкость только 100 строк. В *кэш*-памяти с прямым отображением строки *ОЗУ* с номерами 0, 100, 200, ..., 900 могут сохраняться только в строке 0 КП и нигде иначе, строки 1, 101, 201, …, 901

ОЗУ - в строке 1 КП, строки *ОЗУ* с номерами 99, 199, …, 999 сохраняются в строке 99 *кэш*-памяти (рис. 4.2). Такая организация *кэш*-памяти обеспечивает быстрый *поиск* в ней нужной информации: необходимо проверить ее наличие только в одном месте. Однако емкость КП при этом используется не в полной мере: несмотря на то, что часть *кэш*-памяти может быть не заполнена, будет происходить *вытеснение* из нее полезной информации при последовательных обращениях, например, к строкам 101, 301, 101 *ОЗУ*.

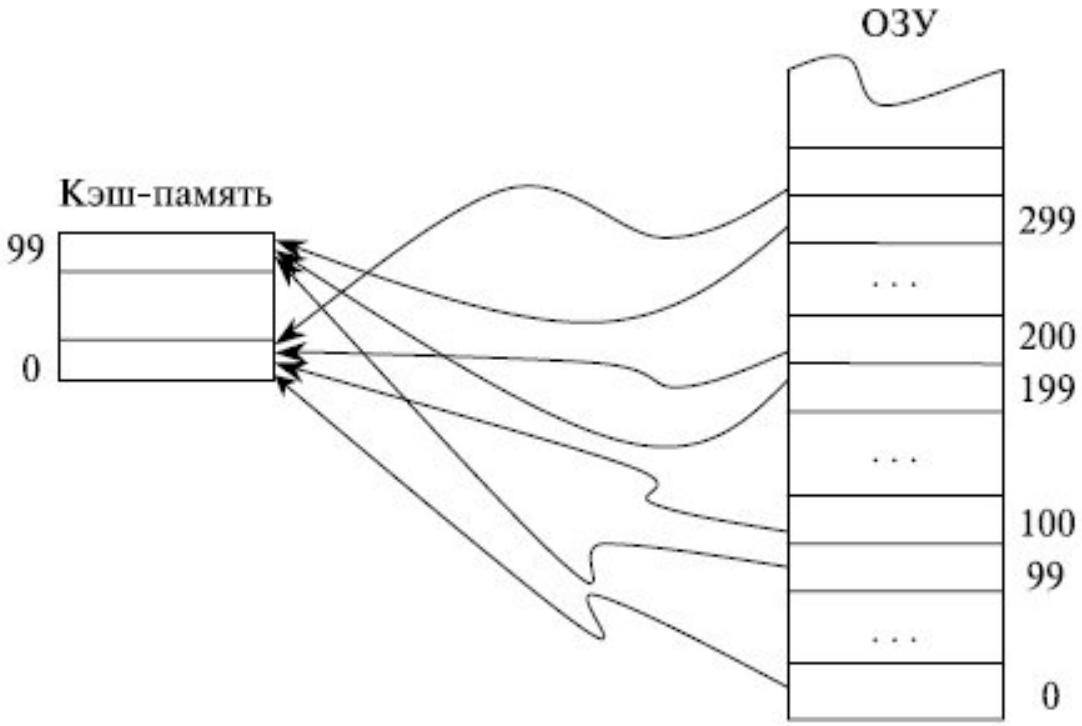

Рис. 4.2. Принцип организации кэш-памяти с прямым отображением

Кэш-память называется **полностью ассоциативной**, если каждая строка *ОЗУ* может располагаться в любом месте *кэш*-памяти.

В полностью ассоциативной *кэш*-памяти максимально используется весь ее объем: *вытеснение* сохраненной в КП информации проводится лишь после ее полного заполнения. Однако *поиск*в *кэш*-памяти, организованной подобным образом, представляет собой трудную задачу.

Компромиссом между этими двумя способами организации *кэш*-памяти служит **множественно-ассоциативная** КП, в которой каждая строка *ОЗУ* может находиться по ограниченному множеству мест в *кэш*-памяти.

При необходимости замещения информации в *кэш*-памяти на новую используется несколько **стратегий замещения**. Наиболее известными среди них являются:

1. *LRU* - замещается строка, к которой дольше всего не было обращений;
2. FIFO - замещается самая давняя по пребыванию в кэш-памяти строка;
3. Random - замещение проходит случайным образом.

Последний вариант, существенно экономя *аппаратные средства* по сравнению с другими подходами, в ряде случаев обеспечивает и более эффективное использование *кэш*-памяти. Предположим, например, что КП имеет объем 4 строки, а некоторый циклический участок программы имеет длину 5 строк. В этом случае при стратегиях *LRU* и FIFO *кэш-память* окажется фактически бесполезной ввиду отсутствия *кэш*-попаданий. В то же время при использовании стратегии

случайного замещения информации часть обращений к КП приведет к *кэш*-попаданиям.

Некоторые эвристические оценки вероятности *кэш*-промаха при разных *стратегиях замещения* **(в процентах)** представлены в табл. 4.1.

Таблица 4.1. Вероятность кэш-промаха для различной кэш-памяти

азмер кэша,Кбайт	**Организация кэш-памяти**					
	2-канальная ассоциативная		**4-канальная ассоциативная**		**8-канальная ассоциативная**	
	RU	**Random**	***RU***	**Random**	***RU***	**Random**
	5	7	3	3	2	2

Анализ таблицы показывает, что:

- увеличением емкости кэша, естественно, уменьшается вероятность **кэш-промаха**, но даже при незначительной на сегодняшний день емкости кэш-памяти в 16 Кбайт около 95 % обращений происходят к КП, минуя оперативную память;
- чем больше степень ассоциативности кэш-памяти, тем больше вероятность **кэш-попадания** за счет более полного заполнения КП (время поиска информации в КП в данном анализе не учитывается);
- механизм *LRU* обеспечивает более высокую вероятность **кэш-попадания** по сравнению с механизмом случайного замещения Random, однако этот выигрыш не очень значителен.

Соответствие между данными в оперативной памяти и в *кэш*-памяти обеспечивается внесением изменений в те области *ОЗУ*, для которых данные в *кэш*-памяти подверглись изменениям. Существует два основных способа реализации этих действий: со сквозной записью (writethrough) и с обратной записью (write-back).

При считывании оба способа работают идентично. При записи **кэширование со сквозной записью** обновляет *основную память* параллельно с обновлением информации в КП. Это несколько снижает *быстродействие* системы, так

как *микропроцессор* впоследствии может вновь обратиться по этому же адресу для записи информации, и предыдущая пересылка строки *кэш*-памяти в *ОЗУ* окажется бесполезной. Однако при таком подходе содержимое соответствующих друг другу строк *ОЗУ* и КП всегда идентично. Это играет большую роль в мультипроцессорных *системах с общей* оперативной памятью.

Кэширование с обратной записью модифицирует строку *ОЗУ* лишь при *вытеснении* строки *кэш*-памяти, например, в случае необходимости освобождения места для записи новой строки из *ОЗУ* в уже заполненную КП. *Операции* обратной записи также инициируются механизмом поддержания согласованности *кэш*-памяти при работе мультипроцессорной системы с общей оперативной памятью.

Промежуточное положение между этими подходами занимает способ, при котором все строки, предназначенные для передачи из КП в *ОЗУ*, предварительно накапливаются в некотором буфере. Передача осуществляется либо при *вытеснении* строки, как в случае **кэширования с обратной записью**, либо при необходимости согласования *кэш*-памяти нескольких микропроцессоров в мультипроцессорной системе, либо при заполнении буфера. Такая передача проводится в пакетном режиме, что более эффективно, чем передача отдельной строки.

Организация внутренней кэш-памяти микропроцессора

Внутренний *кэш* 32-разрядного универсального микропроцессора является общим при обращении как к командам, так и к данным. Обращение ведется по физическим адресам.

Кэш-память обычно реализуется в виде ассоциативного ЗУ, в котором для каждой строки сохраняются дополнительные сведения, называемые тегом, или признаком, в качестве которого выступает адресный код или его часть. Когда в АЗУ подается *адрес*, с ним одновременно сравниваются все теги.

Внутренняя *кэш-память* в микропроцессоре i486 реализует **сквозную запись**. Начиная с МП Pentium используется **сквозная** или **обратная запись**.

Во внешней КП применяется любой способ записи или их комбинация.

Внутренняя *кэш-память* МП i486 имеет емкость 8 Кбайт и организована в виде 4-канальной ассоциативной памяти. Это означает, что данные из какой-либо строки *ОЗУ* могут храниться в любой из 4 строк *кэш*-памяти.

КП состоит из следующих блоков (рис. 4.3):

- блока данных,
- блока тегов,
- блока достоверности и *LRU*.

Блок данных содержит 8 Кбайт данных и команд. Он разделен на 4 массива (направления), каждый из которых состоит из 128 строк. Строка

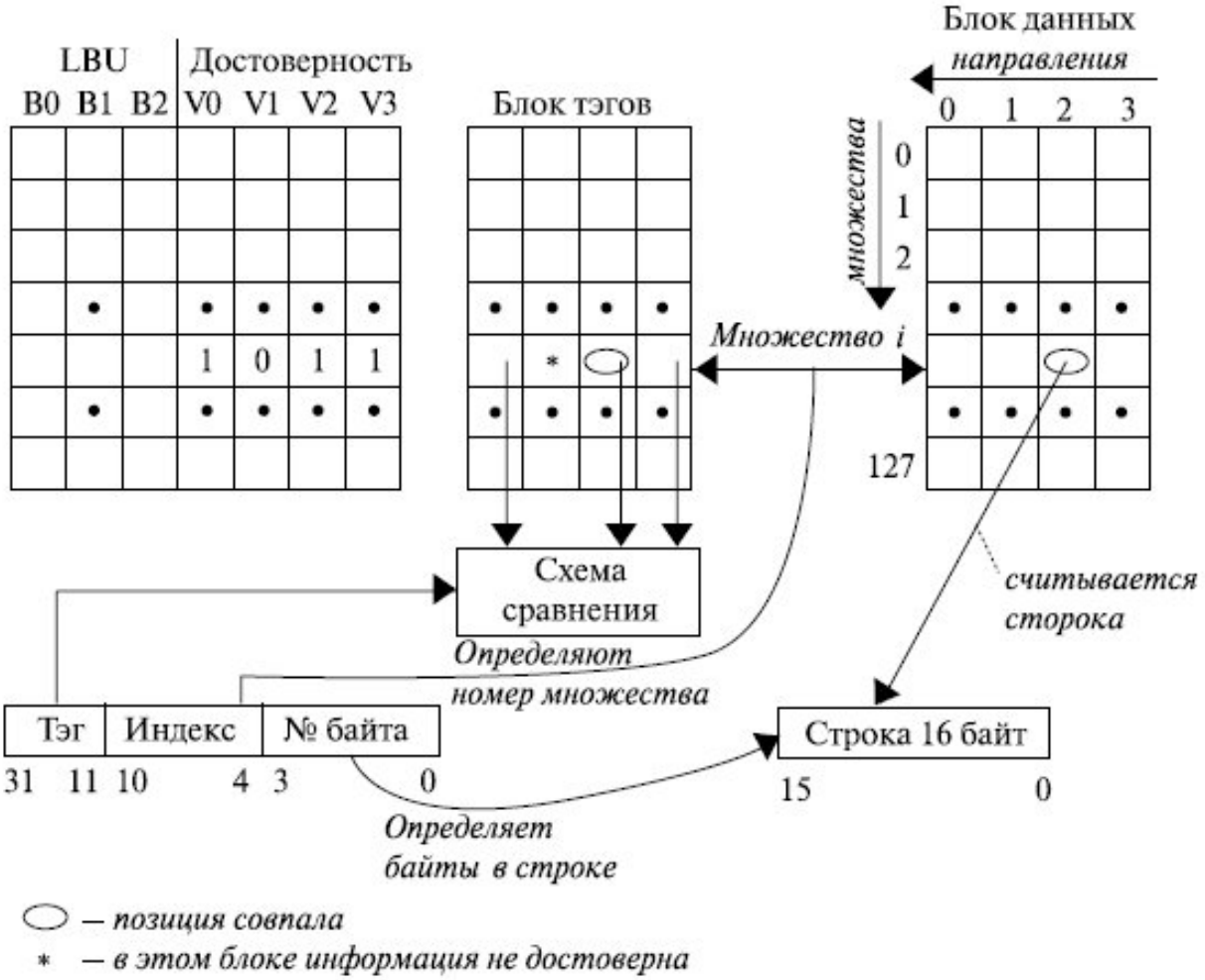

Рис. 4.3. Структура внутренней кэш-памяти МП i486

содержит данные из 16 последовательных адресов памяти начиная с адреса, кратного 16. *Индекс массивов* блока данных, состоящий из 7 *бит*, соответствует 4 строкам КП, по одной из каждого массива. Четыре строки КП с одним и тем же индексом называются множеством.

В **блоке тегов** имеется один *тег* длиной 21 *бит* для каждой строки данных в КП. Блок тегов также разделен на 4 массива по 128 тегов. *Тег* содержит старшие 21 *бит* физического адреса данных, находящихся в соответствующей строке КП.

В **блоке достоверности** и *LRU* содержится по одному 7-разрядному значению для каждого из 128 множеств строк КП: 4 бита достоверности (V) по одному на каждую строку *множества*и 3 бита (B0 ... B2), *управляющие* механизмом *LRU*. Биты достоверности показывают, содержит ли строка достоверные (V = 1) или недостоверные (V = 0) данные. При программной очистке КП и аппаратном сбросе процессора все биты достоверности сбрасываются в 0.

Адресация кэш-памяти осуществляется путем разделения старших 28 *бит* физического адреса на 2 части. Младшие 7 *бит* из этих разрядов (разряды 10…4 физического адреса) образуют *поле*индекса и определяют множество, в котором могут храниться данные. Старшие 21 *бит* (разряды 31…11 физического адреса) служат полем тега и применяются для определения того, находится ли *информация* с данным физическим адресом в какой-либо строке выбранного *множества*.

Поиск в *кэш*-памяти информации с заданным физическим адресом выполняется следующим образом:

1. Физический адрес, по которому происходит обращение, разбивается на 3 поля: Тег, Индекс, № байта. 7 разрядов A10…A4 поля индекса определяют одно из 128 множеств.
2. В выбранном множестве содержатся 4 строки с информацией.

Чтобы определить, присутствует ли нужная информация в одной из строк этого множества, проводится сравнение старших 21 бита физического адреса (поле Тег) с тегами строк выбранного множества. Сравнение проводится только для достоверных строк, то есть тех, у которых в блоке достоверности установлен бит достоверности V = 1.

3. Если для одной из строк ее тег и разряды A31...A11 физического адреса совпали, то это означает, что произошло **кэш-попадание** и необходимая информация есть в кэш-памяти.
4. Считывается найденная строка из 16 байт. Искомый байт в ней определяется 4 младшими разрядами физического адреса (A3…A0).
5. Если на этапе 3 совпадения не произошло или все строки множества недостоверны, эта ситуация определяется как **кэш-промах**. В этом случае по сформированному микропроцессором физическому адресу выполняется обращение к оперативной памяти. Из *ОЗУ* извлекается нужная информация, и содержащая ее строка записывается в свободную строку выбранного множества. Старшие 21бит физического адреса записываются в поле тега этой строки. Если все строки в выбранном множестве достоверны, то замещается строка, к которой дольше всего не было обращений согласно механизму *LRU*. Этот механизм действует точно так же, как и при *вытеснении* строк из **буфера ассоциативной трансляции** *TLB*.

Режим работы *кэш*-памяти определяется программно установкой разрядов CD (запрет кэширования) и NW (запрет сквозной записи) в управляющем регистре CR0. *Кэширование* можно разрешить (это состояние после инициализации при сбросе), можно запретить при наличии достоверных строк (в этом режиме КП действует как быстрое внутреннее *ОЗУ*) или, наконец, *кэширование* может быть полностью запрещено.

Управление работой кэш-памяти на уровне страниц

В **элементах каталога страниц и таблиц страниц** имеются 2 бита, которые применяются для управления выходными сигналами процессора и участвуют в кэшировании страниц.

Бит PCD запрещает (*PCD* = 1) или разрешает (*PCD* = 0) *кэширование* страницы. Запрещение кэширования необходимо для страниц, которые содержат порты ввода/вывода с отображением на *память*. Оно также полезно для страниц, *кэширование* которых не дает выигрыша в быстродействии, например, страниц, содержащих программу инициализации.

Бит PWT определяет метод обновления *ОЗУ* и внешней *кэш*-памяти (*кэш* 2-го уровня). Если PWT = 1, то для данных в соответствующей странице определяется *кэширование* со сквозной записью, при PWT = 0 применяется способ обратной записи. Используется в микропроцессорах начиная с Pentium. Так как внутренняя *кэш-память* в МП i486 работает со сквозной записью, состояние бита PWT на нее не влияет. *Бит* PWT в этом случае действует только на внешнюю КП.

Обеспечение согласованности кэш-памяти микропроцессоров в мультипроцессорных системах

Рассмотрим особенности работы *кэш*-памяти в том случае, когда одновременно несколько микропроцессоров используют общую оперативную *память* (рис. 4.4). В этом случае могут возникнуть проблемы, связанные с кэшированием информации из оперативной памяти в *кэш-память* микропроцессоров.

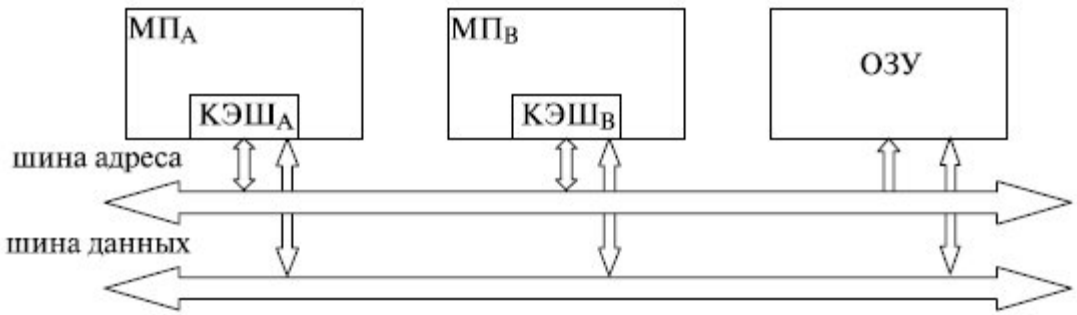

Рис. 4.4. Структура мультимикропроцессорной системы с общей оперативной памятью

Предположим, что МП А считал некоторую строку данных из *ОЗУ* в свою внутреннюю КП и изменил данные в этой строке в процессе работы.

Мы отмечали, что существует два основных механизма обновления оперативной памяти:

1. **сквозная запись**, которая подразумевает, что как только изменилась информация во внутренней кэш-памяти, эта же информация копируется в то же место оперативной памяти, и

2. **обратная запись**, при которой микропроцессор после изменения информации во внутреннем кэше отражает это изменение в оперативной памяти не сразу, а лишь в тот момент, когда происходит *вытеснение* данной строки из кэш-памяти в оперативную. То есть существуют определенные моменты времени, когда информация, предположим, по адресу 2000 имеет разные значения: микропроцессор ее обновил, а в оперативной памяти осталось старое значение. Если в этот момент другой микропроцессор (МП В), использующий ту же оперативную память, обратится по адресу 2000 в *ОЗУ*, то он прочитает оттуда старую информацию, которая к этому времени уже не актуальна.

Для обеспечения согласованности (*когерентности*) памяти в мультипроцессорных системах используются аппаратные *механизмы*, позволяющие решить эту проблему. Такие *механизмы*называются **протоколами когерентности кэш-памяти**. Эти протоколы призваны гарантировать, что любое считывание элемента данных возвращает последнее по времени записанное в него *значение*.

Существует два класса протоколов *когерентности*:

- **протоколы на основе справочника** (directory based): информация о состоянии блока физической памяти содержится только в одном месте, называемом справочником (физически справочник может быть распределен по узлам системы);
- **протоколы наблюдения** (*snooping*): каждый кэш, который содержит копию данных некоторого блока физической памяти, имеет также соответствующую копию служебной информации о его состоянии; централизованная система записей отсутствует; обычно кэши расположены на *общей шине*, и контроллеры всех кэшей наблюдают за шиной (просматривают ее), чтобы определять, какие обращения по адресам в пределах этого блока происходят со стороны других микропроцессоров.

В мультипроцессорных *системах с общей* памятью наибольшей популярностью пользуются **протоколы наблюдения**, поскольку для опроса состояния кэшей они могут использовать уже существующее физическое соединение - шину памяти.

Для поддержания *когерентности* применяется два основных метода.

Один из методов заключается в том, чтобы гарантировать, что *процессор* должен получить исключительные *права доступа* к элементу данных перед выполнением записи в этот *элемент данных*. Этот тип протоколов называется протоколом **записи с аннулированием** (write invalidate protocol), поскольку при выполнении записи он аннулирует другие копии. Это наиболее часто используемый протокол как в **схемах на основе справочников**, так и в **схемах наблюдения**. Исключительное *право доступа* гарантирует, что во *время выполнения*записи не существует никаких других копий элемента данных, в которые можно писать или из которых можно читать: все другие кэшированные копии элемента данных аннулированы.

Альтернативой протоколу записи с *аннулированием* является обновление всех копий элемента данных в случае записи в этот *элемент данных*.

Этот тип протокола называется протоколом *записи с обновлением* (write update protocol), или протоколом *записи с трансляцией* (write *broadcast* protocol).

Эти две схемы во многом похожи на схемы работы *кэш*-памяти со сквозной и с обратной записью. Ключевым моментом реализации в *многопроцессорных системах* с небольшим числом процессоров как схемы записи с *аннулированием*, так и схемы записи с обновлением данных, является использование для выполнения этих операций механизма шины. Для выполнения *операции* обновления или аннулирования *процессор* просто захватывает шину и транслирует по ней *адрес*, по которому должно производиться обновление или *аннулирование* данных. Все процессоры непрерывно наблюдают за шиной, контролируя появляющиеся на ней адреса.

Процессоры проверяют, не находится ли в их *кэш*-памяти *адрес*, появившийся на шине. Если это так, то соответствующие данные в кэше либо аннулируются, либо обновляются в зависимости от используемого протокола.

Рассмотрим один из наиболее распространенных протоколов, обеспечивающих согласованную работу *кэш*-памяти нескольких микропроцессоров и основной памяти в мультимикропроцессорных системах, **протокол MESI**, который относится к группе **протоколов наблюдения с аннулированием**. Будем знакомиться с ним на примере двухпроцессорной системы, состоящей из микропроцессоров A и B.

Этот протокол использует 4 признака состояния строки *кэш*-памяти микропроцессора, по первым буквам которых и называется протокол:

- измененное состояние (Modified): информация, хранимая в кэш-памяти микропроцессора A, достоверна только в этом кэше; она отсутствует в оперативной памяти и в кэш-памяти других микропроцессоров;
- исключительная копия (Exclusive): информация, содержащаяся в кэше A, содержится еще только в оперативной памяти;
- разделяемая информация (Shared): информация, содержащаяся в кэше A, содержится в кэш-памяти по крайней мере еще одного МП, а также в оперативной памяти;
- недостоверная информация (*Invalid*): в строке кэш-памяти находится недостоверная информация.

Таким образом, состояние признаков потокола *MESI* отражает следующие состояния (по отношению к МПА) строки *кэш*-памяти (табл. 4.2):

Таблица 4.2. Формирование признаков состояния протокола *MESI*

Состояние признака протокола	стояние строки памяти		
	Кэш А	Кэш В	ЗУ
dified			
red			
lusive			
alid			

При работе микропроцессора А с точки зрения обеспечения *когерентности* памяти возможны следующие ситуации:

- RH (Read *Hit*) - **кэш-попадание** при чтении;
- WH (Write *Hit*) - **кэш-попадание** при записи;
- RME (Read Miss Exclusive) - **кэш-промах** причтении;
- *RMS* (Read Miss Shared) - **кэш-промах** при чтении, но соответствующий блок есть в кэш-памяти другого микропроцессора;
- WM (Write Miss) - **кэш-промах** при записи;
- SHR (Snoop *Hit* Read) - обнаружение копии блока при прослушивании операции чтения другого кэша;
- SHW (Snoop *Hit* Write) - обнаружение копии блока при прослушивании операции записи другого кэша.

Наибольший интерес здесь представляют две последние позиции.

Современные микропроцессоры имеют двунаправленную шину адреса.

Выдавая информацию на эту шину, *микропроцессор* адресует ячейки оперативной памяти или устройства ввода-вывода. В силу того, что в рассматриваемой мультипроцессорной системе микропроцессоры связаны общей шиной, в том числе и шиной адреса, принимая информацию по адресным линиям, *микропроцессор* определяет, было ли обращение по адресам, содержащимся в его *кэш*-памяти, со стороны других микропроцессоров. При обнаружении такого обращения меняется состояние строки *кэш*-памяти микропроцессора.

Изменения признака состояния блока *кэш*-памяти МП в зависимости от различных ситуаций в его работе и работе мультимикропроцессорной системы в целом представлены на рис. 4.5.

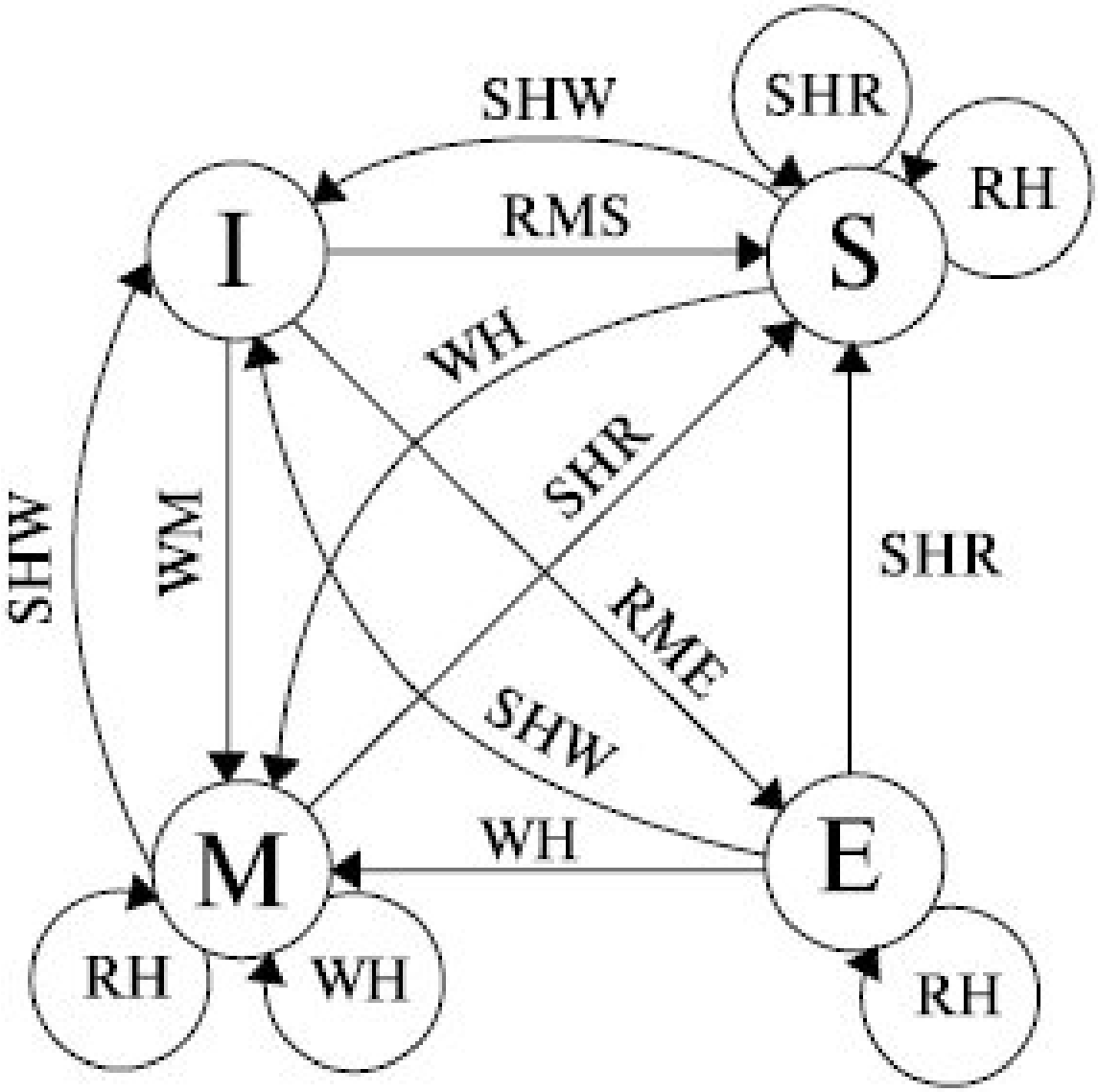

Рис. 4.5. MESI-диаграмма обеспечения когерентности кэш-памяти

Проиллюстрируем некоторые из представленных переходов.

Пусть блок *кэш*-памяти находится в состоянии Modified, то есть достоверная *информация* находится только в *кэш*-памяти данного МП. Тогда в случае обнаружения при прослушивании адресной шины обращения со стороны другого микропроцессора для чтения информации по входящим в данную строку адресам *микропроцессор* должен передать эту строку кэшпамяти в *ОЗУ*, откуда она уже будет прочитана другим микропроцессором.

При этом состояние строки в *кэш*-памяти рассматриваемого микропроцессора изменится с модифицированного на разделяемое (Shared).

Если строка *кэш*-памяти находилась в состоянии *Invalid*, то есть *информация* в ней была недостоверной, то по отношению к этой строке следует рассматривать только ситуации, связанные с **кэш-промахами**. Так, если произошел **кэш-промах** при выполнении *операции* записи, то необходимая строка будет занесена в *кэш-память* данного МП, в эту строку будут записаны измененные данные, и она приобретет статус исключительного владельца новой информации (Modified).

Краткие итоги. В лекции рассмотрены общие принципы функционирования *кэш*-памяти микропроцессора, организация *кэш*-памяти с прямым отображением, полностью ассоциативной и множественно-ассоциативной КП. Рассмотрены основные *механизмы* обновления оперативной памяти: *кэширование* со сквозной и с обратной записью. Представлена организация внутренней *кэш*-памяти микропроцессора. Разобраны способы обеспечения согласованности *кэш*-памяти микропроцессоров в мультипроцессорных системах.

Лекция 5. Аппаратные средства защиты информации в микропроцессоре

Аппаратные средства защиты информации в микропроцессоре

Если в памяти одновременно могут находиться несколько независимых программ, необходимы специальные меры по предотвращению или ограничению обращений одной программы к областям памяти, используемым другими программами. Программы могут содержать такие ошибки, которые, если этому не воспрепятствовать, приводят к искажению информации, принадлежащей другим программам. Последствия таких ошибок особенно опасны, если разрушению подвергнутся программы операционной системы. Другими словами, надо исключить несанкционированное воздействие программы пользователя на работу программ других пользователей и программ операционной системы.

Чтобы воспрепятствовать разрушению одних программ другими, достаточно защитить область памяти данной программы от попыток записи в нее со стороны других программ, а в некоторых случаях и своей программы (*защита от записи*), при этом допускается обращение других программ к этой области памяти для считывания данных.

В других случаях, например, при ограничениях на *доступ* к информации, хранящейся в системе, необходимо иметь возможность запрещать другим программам производить как *запись*, так и считывание в данной области памяти. Такая *защита от записи* и считывания помогает отладке программы, при этом осуществляется *контроль* каждого случая выхода за область памяти своей программы.

Для облегчения отладки программ желательно выявлять и такие характерные ошибки в программах, как попытки использования данных вместо команд или команд вместо данных в собственной программе, хотя эти ошибки могут и не разрушать информацию.

Средства *защиты памяти* должны предотвращать:

- неразрешенное взаимодействие пользователей друг с другом;
- несанкционированный доступ пользователей к данным;
- повреждение программ и данных из-за ошибок в программах;
- намеренные попытки разрушить *целостность системы*;
- случайные искажения данных.

Средства защиты микропроцессора делятся на 2 группы:

- **защиту при управлении памятью** и
- **защиту по привилегиям**.

Средства управления памятью обнаруживают большинство программных ошибок.

До загрузки селектора в *сегментный регистр* и кэширования дескриптора осуществляется несколько контро-льных проверок:*процессор* проверяет, что *поле* Index **селектора** находится в пределах таблицы, определяемой его битом TI ;

- при загрузке **селектора** в сегментный регистр данных (*DS*, ES, FS, GS) **тип дескриптора** должен разрешать считывание из сегмента.

Только выполняемые сегменты для этих регистров не допускаются, но сегменты с разрешенными операциями выполнения/считывания допустимы;

- в случае сегментного регистра стека (SS) в сегменте должны быть разрешены операции считывания и записи;
- при загрузке регистра CS сегмент должен быть обязательно исполняемым;
- в регистр LDTR можно загружать только селектор, указывающий на дескриптор сегмента типа *LDT* ;
- в регистр TR можно загружать только селектор, указывающий на дескриптор сегмента состояния задачи.

Если хотя бы одна проверка дала отрицательный результат, то формируется особый случай и *загрузка* **селектора** не производится.

После загрузки селектора при фактическом обращении к памяти *процессор* контролирует, чтобы запрашиваемая операция (чтение/*запись*)для этого сегмента была разрешена. На этом этапе обнаруживаются и отвергаются попытки записи в сегмент кода или в только считываемые *сегменты* данных, а также считывание из сегмента кода, для которого разрешено только выполнение. Здесь же проводится проверка превышения сформированного **смещения в сегменте** длины сегмента, указанной в **поле предела дескриптора**. Такие ситуации невозможно выявить при загрузке селектора.

Защита по привилегиям фиксирует более тонкие ошибки и намеренные попытки нарушить *целостность системы*.

Под **привилегиями** понимается свойство, определяющее, какие *операции* и обращения к памяти разрешается производить процессору при выполнении текущей задачи.

На аппаратном уровне в процессоре поддерживаются 4 **уровня привилегий**. Распознаваемым процессором объектам назначается *значение* от 0 до 3, причем 0 соответствует высшему, а 3 - низшему уровню привилегий. С помощью указания уровня привилегий и правил защиты обеспечивается *управляемый доступ* к процедурам и данным операционной системы и других задач.

Привилегии устанавливаются значениями соответствующих полей в следующих основных *системных объектах* микропроцессора:

- *DPL* - **уровень привилегий сегмента** (находится в **байте доступа** дескриптора сегмента);
- *RPL* - биты <0,1> **селектора**, хранящегося в *сегментном регистре*;

текущий *уровень привилегий* программы *CPL* задается полем *RPL* селектора, хранящегося в *сегментном регистре* CS ;

- IOPL - поле **регистра флагов**, которое указывает, на каком уровне привилегий разрешено выполнять операции ввода/вывода, а также в некоторых других объектах, используемых, например, при переключении задач и обработке прерываний.

Так как число программ, которые могут выполняться на более высоком **уровне привилегий**, уменьшается к уровню 0 и так как программы уровня 0 действуют как *ядро* системы, **уровни привилегий** обычно изображаются в виде четырех **колец защиты** (*Protection Rings*) (рис. 5.1).

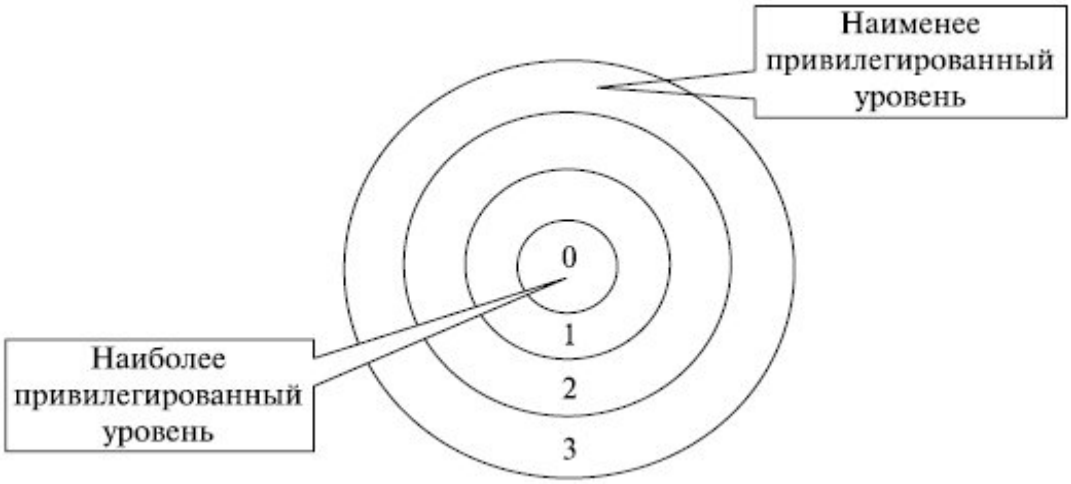

Рис. 5.1. "Кольца защиты"

Типовое распределение программ по кольцам защиты выглядит следующим образом:

- **уровень 0**: ядро ОС, обеспечивающее инициализацию работы, управление доступом к памяти, защиту и ряд других жизненно важных функций, нарушение которых полностью выводит из строя процессор;
- **уровень 1**: основная часть программ ОС (утилиты);
- **уровень 2**: служебные программы ОС (драйверы, СУБД, специализированные подсистемы программирования и т. д.);
- **уровень 3**: прикладные программы пользователя.

Аппаратные средства процессора, работающего в **защищенном режиме**, постоянно контролируют, что текущая *программа* достаточно привилегированна для того, чтобы:

- выполнять некоторые команды, называемые привилегированными;

- выполнять операции ввода/вывода на том или ином внешнем устройстве;
- обращаться к данным других программ;
- передавать управление внешнему (по отношению к самой программе) коду командами межсегментной передачи управления.

Привилегированные команды - это те команды, которые влияют на *механизмы* управления памятью, защиты и некоторые другие жизненно важные функции. Это, например, команды загрузки таблиц дескрипторов *GDT*, *IDT*, *LDT*, команды обмена с *регистрами управления* CRi. Они могут выполняться только программами, имеющими наивысший (нулевой)

уровень привилегий. Это приводит к тому, что простую незащищенную систему можно целиком реализовать только в кольце 0, так как в других **кольцах защиты** не будут доступны все команды.

Операции ввода/вывода разрешено выполнять программам, *уровень привилегий* которых не ниже значения, установленного в *поле* IOPL **регистра флагов**. То есть должно выполняться соотношение: $CPL \leq$ IOPL.

Обращение к данным других программ разрешается только на своем и менее привилегированном уровнях защиты (рис. 5.2).

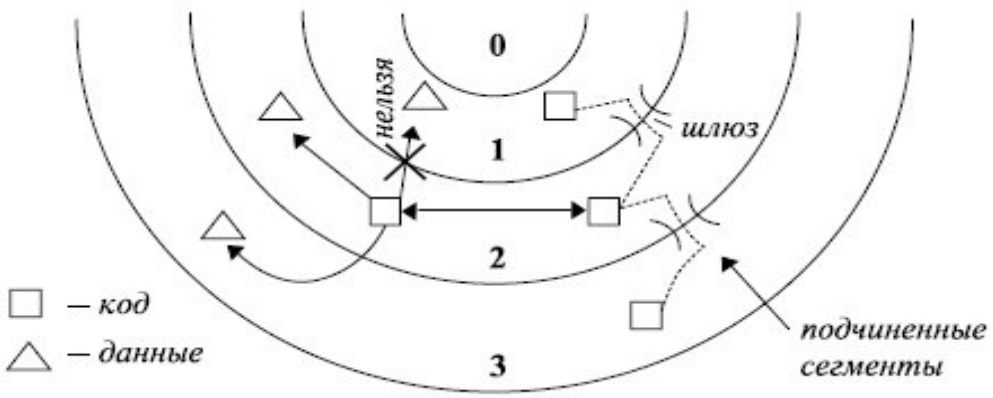

Рис. 5.2. Порядок взаимодействия программ и данных на разных уровнях привилегий

Передачи управления между программами ограничиваются только текущим **кольцом защиты**. В то же время в процессе выполнения любой программы необходимо обращаться к программам, находящимся на более высоком уровне привилегий, например, к драйверам или *СУБД*. Для этих целей используются специально установленные точки входа в эти программы (шлюзы). Передача управления на более низкий *уровень привилегий* осуществляется с помощью механизма подчиненных сегментов.

При передаче управления подчиненному сегменту действует правило: *DPL*>=max (*CPL*, *RPL*). Однако при этом подчиненный код будет выполняться на том же уровне привилегий, что и вызвавший его код (*CPL* не изменится). Ограничивая

передачу управления в пределах одного **кольца защиты**, *процессор* предотвращает *произвольное* изменение **уровней привилегий**. Ели бы *значение CPL* можно было легко изменить, все остальные средства защиты по привилегиям потеряли бы смысл.

Использование шлюзов вызова

В процессе работы программам постоянно приходится обращаться к программам, находящимся на более высоком уровне привилегий, например, к драйверам внешних устройств, системам программирования. *Прямой* бесконтрольный вызов таких программ запрещается средствами защиты. Однако если к какой-либо системной программе предусматривается обращение со стороны менее привилегированных программ, то для нее создается специальный *объект* - **шлюз вызова**.

Здесь наиболее интересный момент связан с реализацией этого шлюза. С одной стороны, пользователю необходимо предоставить возможность выполнить необходимую ему более привилегированную программу. Но, с другой стороны, бесконтрольный вызов таких программ, например, *запуск* драйвера с его середины вследствие ошибок программирования или злого умысла, может привести к непредсказуемым последствиям.

Таким образом, необходимо дать возможность обращаться к системным программам, но обращаться только начиная с определенной фиксированной точки кода. **Шлюзы вызова** - это некоторые системные объекты, которые обеспечивают вход в строго определенную точку программы, находящейся на более высоком уровне привилегий.

Шлюз должен быть предварительно помещен в таблицу дескрипторов. Дескрипторы шлюзов не определяют никакого адресного пространства, поэтому в них нет полей базы и предела, то есть они фактически не являются дескрипторами. Обращение к более привилегированным программам производится командами, аналогичными командам обращения к подпрограммам в другом кодовом сегменте (команды типа FAR CALL).

То есть нельзя перейти в более привилегированный сегмент командой с полной передачей управления: "пришел на уровень с более высокими привилегиями и там остался", а переход возможен только с помощью команд с возвратом. Эти команды должны адресовать **шлюз вызова**, а не сегмент кода назначения. **Шлюз вызова** определяет сегмент кода, которому передается управление, и точное смещение в этом сегменте, где начинается выполнение процедуры.

Формат **шлюза вызова** представлен на рис. 5.3.

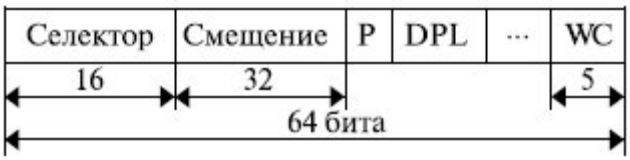

Рис. 5.3. Формат шлюза вызова

Шлюз вызова содержит селектор того сегмента, куда передается управление, и смещение в этом сегменте. Это, с одной стороны, позволяет найти данную программу, но, с другой стороны, строго определяет точку входа в программу, чтобы можно было запустить ее только со строго определенного места.

Другими важными параметрами, определяемыми шлюзом вызова, являются: P - **бит присутствия** ; WC - количество параметров, передаваемых из стека текущей программы в *стек*вызываемой программы; *DPL* - **уровень привилегий**.

При использовании шлюза вызова проводится следующий *анализ* **уровней привилегий**:

- значение *DPL* шлюза вызова должно быть больше или равно значению текущего уровня привилегий *CPL* и значению *RPL* селектора, вызывающего шлюз;
- значение *DPL* шлюза вызова должно быть больше или равно значению *DPL* целевого сегмента кода;
- значение *DPL* целевого сегмента кода должно быть меньше или равно значению текущего уровня привилегий *CPL*.

Порядок использования **шлюза вызова** представлен на рис. 5.4.

1. Как любая команда межсегментного перехода, команда FAR CALL содержит селектор сегмента и смещение в этом сегменте. Смещение, которое указано в команде, микропроцессор игнорирует: положение вызываемого кода в более привилегированном сегменте определяется не им, а шлюзом вызова. По селектору, определенному в команде, идет обращение к **таблице дескрипторов**. По **типу дескриптора** определяется, что это системный объект типа "шлюз вызова".
2. **Селектор** из шлюза вызова заносится в регистр CS микропроцессора, а смещение - в **регистр - указателя команд** EIP.
3. По полученному **селектору** обращаемся к **дескриптору** сегмента более привилегированной программы.

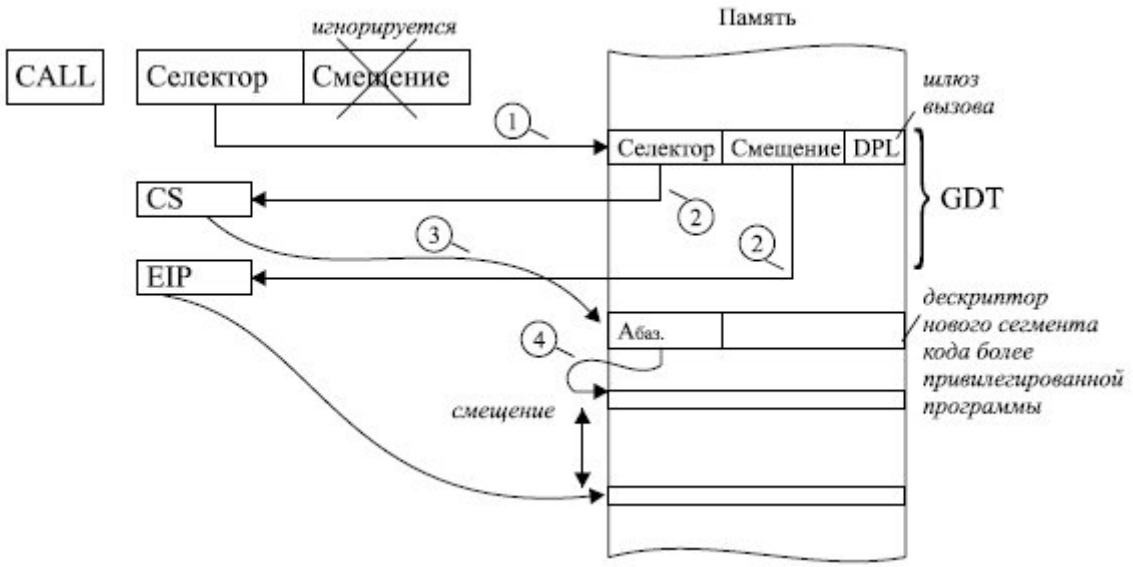

Рис. 5.4. Использование шлюза вызова для обращения к программам на более высоком уровне привилегий

4. Из дескриптора извлекается базовый адрес нового сегмента. Его суммирование со значением смещения из шлюза вызова, занесенного в EIP, определяет физический адрес начала новой программы.

Иначе говоря, с помощью такой многоступенчатой обработки команды перехода мы получаем *доступ* к более привилегированной программе. Но для этого *поле DPL* шлюза должно быть установлено таким, чтобы к нему могла обратиться менее привилегированная *программа*. То есть если мы хотим, чтобы пользовательские программы могли вызывать некоторую программу, находящуюся, например, на **уровне привилегий** 1, то *DPL* шлюза этой программы должен быть равен 3.

Предположим теперь, что *пользователь* хочет воспользоваться некоторыми системными утилитами. Пусть пользовательская *программа* имеет *уровень привилегий* 3, *ядро* ОС - уровень 0, утилиты ОС - уровень 1 (рис. 5.5).

В этом случае *шлюз* утилиты должен иметь *DPL* = 3. Это позволит пользовательской программе вызвать утилиты ОС, но не ее *ядро*. При необходимости к ядру операционной системы могут обратиться утилиты.

Механизм вызова:

- шлюзу утилит присваивается **уровень привилегий** 3, обеспечивая его доступность пользовательским программам;
- шлюзу ядра присваивается **уровень привилегий** 1, что делает его доступным для программ-утилит, но обращение пользовательских программ к шлюзу ядра невозможно.

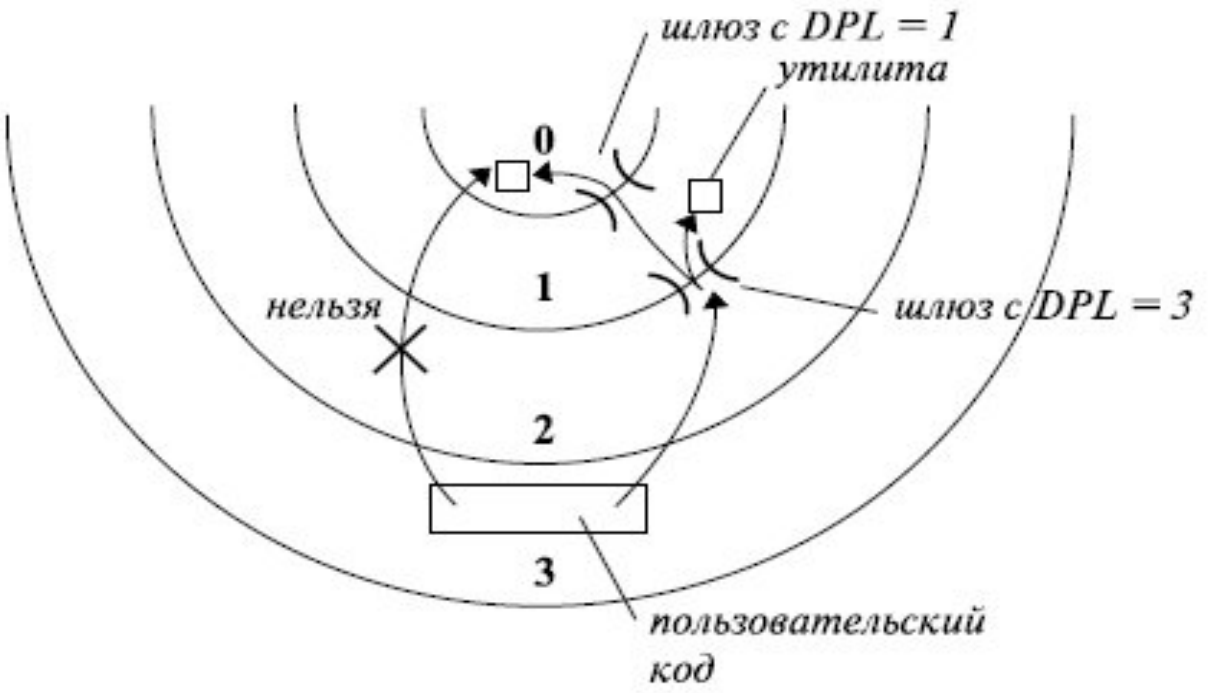

Рис. 5.5. Последовательное обращение к более привилегированным программам

Таким образом, *программа* может последовательно обратиться к ряду более привилегированных программ.

Конкретная ОС необязательно должна поддерживать все 4 **уровня привилегий**. Так, ОС *UNIX* работает с двумя **кольцами защиты**: *супервизор* (уровень 0) и *пользователь* (уровни 1, 2, 3). *Операционная система OS/2* поддерживает 3 уровня: код Ос работает в кольце 0, специальные процедуры для обращения к устройствам ввода/вывода действуют в кольце 1, а прикладные программы выполняются в кольце 3. В *Windows* NT используются только два **уровня привилегий**: нулевое и третье кольцо. В нулевом кольце работает *ядро* системы и системные драйверы, а в третьем - все запущенные приложения. Привилегированные команды и ввод-ввод для третьего кольца запрещены. Для взаимодействия с аппаратной частью компьютера пользовательские программы вызывают системные сервисы ядра ОС, обращение к которым оформлено как **шлюзы**. При вызове такого **шлюза** процесс переходит в нулевое кольцо, и там *ядро* ОС и драйверы обрабатывают *запрос* и возвращают результаты приложению. После перехода в нулевое кольцо *приложение* не может как-либо контролировать свое *исполнение*, пока управление не будет возвращено коду третьего кольца. Это является необходимым условием защиты, обеспечивающим *безопасность* всей системы.

Защита по привилегиям начинает работать уже на этапе загрузки **селектора** в сегментные регистры. При загрузке селектора в сегментные регистры данных должно выполняться соотношение: $DPL < \max(CPL, RPL)$, а при загрузке **селектора** в *сегментный регистр* стека SS должно быть выполнено соотношение: $DPL = CPL$.

При страничном преобразовании адреса применяется простой двухуровневый механизм **защиты по привилегиям**: *пользователь* (соответствует уровню 3 привилегий сегмента) и *супервизор* (уровни 0, 1, 2), указываемый в бите U/S ЭТС.

При сегментно-страничной организации памяти производится *объединение* защиты сегментов и страниц: сначала реализуется защита сегментов, а затем защита страниц. Например, допускается определить большой сегмент данных, в котором некоторые части будут только считываемые, а другие допускают считывание и *запись*. В такой ситуации элементы каталога таблиц страниц и/или элементы таблиц страниц должны иметь соответствующие значения атрибута R/W.

Краткие итоги. В лекции рассмотрены основные требования, предъявляемые к средствам *защиты памяти, механизмы* защиты, используемые при управлении памятью и при защите по привилегиям, *доступ* к программам на более высоком уровне привилегий посредством использования шлюзов вызова.

Лекция 6. Мультипрограммный режим работы микропроцессора

Мультипрограммный режим работы микропроцессора

Многозадачностью (мультипрограммным режимом работы) называют такой способ организации работы системы, при которой в ее памяти одновременно содержатся программы и данные для выполнения нескольких процессов обработки информации (задач). В этом режиме должна обеспечиваться взаимная защита программ и данных, относящихся к различным задачам, а также возможность перехода от выполнения одной задачи к другой (переключение задач).

Под **задачей (процессом)** понимается последовательность взаимосвязанных действий, ведущих к достижению некоторой цели. В вычислительной технике под задачей понимается конкретная сущность, которая тесно связана с архитектурой процессора и обладает своим виртуальным адресным пространством и состоянием. В более простом смысле будем полагать, что **задача** - это *программа*, которая выполняется или ожидает выполнения, пока выполняется другая *программа*.

Процесс может находиться в следующих состояниях:

- порождение - подготавливаются условия для первого исполнения на процессоре;
- активное состояние (счет) - программа исполняется на процессоре;
- ожидание - программа не исполняется по причине занятости какого-либо ресурса;
- готовность - программа не исполняется, но для исполнения предоставлены все необходимые в текущий момент ресурсы, кроме *центрального процессора*;
- окончание - нормальное или аварийное завершение программы, после которого процессор и другие ресурсы ей не предоставляются.

Применительно к компьютерам в *определение* задачи обычно включаются ресурсы, необходимые для достижения цели. Понятие "*ресурс*" строго не определено. Будем считать, что всякий потребляемый *объект* (независимо от формы его существования), обладающий некоторой практической ценностью для потребителя, является **ресурсом**:

объем оперативной памяти, время счета на процессоре, дисковое *пространство* и т. д.

Основные черты мультипрограммного режима:

- оперативной памяти находятся несколько программ в состояниях *активности, ожидания* или готовности;
- время работы процессора разделяется между программами, находящимися в памяти в состоянии готовности;
- параллельно с работой процессора происходит подготовка и обмен с несколькими внешними устройствами (ВУ).

Мультипрограммирование предназначено для повышения пропускной способности вычислительной системы путем более равномерной и полной загрузки всего его оборудования, в первую *очередь* - процессора.

При этом скорость работы самого процессора и номинальная *производительность* ЭВМ не зависят от использования *мультипрограммирования*.

Мультипрограммный режим имеет в ЭВМ аппаратную и программную поддержку. Аппаратура, используемая при организации мультипрограммного режима, включает контроллеры *ОЗУ* и внешних устройств, которые могут работать параллельно с процессором, систему прерывания, *аппаратные средства* защиты программ и данных и т. д.

Программная составляющая содержит мультизадачную операционную систему, драйверы внешних устройств, обработчики прерываний и другие средства. *Операционная система*, реализуя **мультипрограммный режим**, должна распределять (в том числе динамически) между параллельно выполняемыми программами **ресурсы** системы (время процессора, оперативную и *внешнюю память*, устройства ввода-вывода) с целью увеличения пропускной способности и с учетом ограничений на **ресурсы** и требований *по* срочности выполнения отдельных программ.

Эффективность работы мультипрограммной ЭВМ можно оценить количеством задач, выполненных в единицу времени (*пропускная способность*), и временем выполнения отдельной программы.

Важное *значение* при *анализе работы* ЭВМ имеет *определение* степени использования ее ресурсов. Для этого широко применяются следующие показатели (рис. 6.1):

k_q коэффициент загрузки устройства:

$$k_q = \frac{T_q}{T}$$

где T_q - время занятости устройства q за общее время T работы ЭВМ;

L_q *средняя длина очереди* запросов к устройству q:

$$L_q = \frac{\sum_{j=1}^{n} L_{q_i} * \Delta t_i}{T}$$

где L_{qi} - *длина* очереди к устройству q на интервале времени Δt_i и $\sum_{j=1}^{n} \Delta t_i = T$

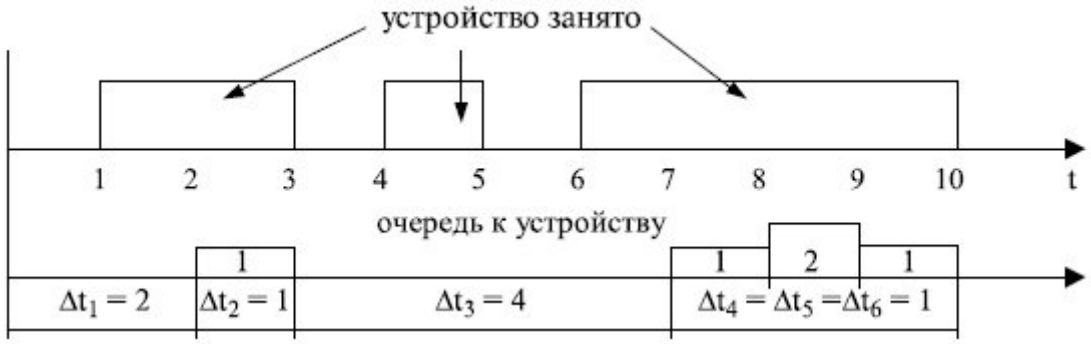

Рис. 6.1. Оценка показателей работы мультипрограммной ЭВМ

Для представленного на рис. 6.1 случая:

$$L_q = \frac{0 \times 1 + 1 \times 1 + 0 \times 4 + 1 \times 1 + 1 \times 2 + 1 \times 1}{10}$$

Помимо средней длины очереди важна также и динамика изменения ее длины.

По значениям k_q, L_q и изменениям во времени значения Lq можно определить наиболее дефицитный **ресурс** в системе, ее "узкое *место*".

Устранение этих "узких мест" может проводиться или за счет *увеличения производительности* соответствующего **ресурса**, или выбором такой смеси задач, которая обеспечивала бы более равномерное использование всех ресурсов (например, одни задачи более активно используют *процессор* (счетные задачи), другие - *жесткий диск* (работа с базами данных), третьи - устройства ввода-вывода (*вывод* большого объема графической информации)).

Рассмотрим пример выполнения четырех программ в мультипрограммном режиме при *коэффициенте мультипрограммирования* равном 2.

Коэффициент мультипрограммирования (КМ) - это количество программ, обрабатываемых одновременно в **мультипрограммном режиме**.

Полагаем, что ЭВМ имеет 3 устройства, которые могут работать параллельно: центральный *процессор* (*CPU*), устройство ввода (*IN*) и устройство вывода (OUT), а программы проходят следующий *цикл работы*:

счет1 - ввод - счет2 - *вывод*. Времена выполнения соответствующих блоков программ заданы в табл. 6.1.

Будем считать, что программы имеют относительный приоритет. То есть, во-первых, если на какой-либо *ресурс* при его освобождении одновременно претендует несколько программ, то он предоставляется программе с наименьшим номером. Во-вторых, *программа*, занявшая *ресурс*, не освобождает его до истечения

требуемого времени, даже если в этот период на *ресурс* станет претендовать *программа*, имеющая больший приоритет.

Таблица 6.1. Характеристики программ

Программа	PU1	PU2	OUT

Более приоритетной считаем программу с меньшим номером.

Очередь программ к процессору обозначим *Ready*, а общую *очередь* к внешним устройствам - Wait.

Полученная *диаграмма* выполнения программ представлена на рис. 6.2.

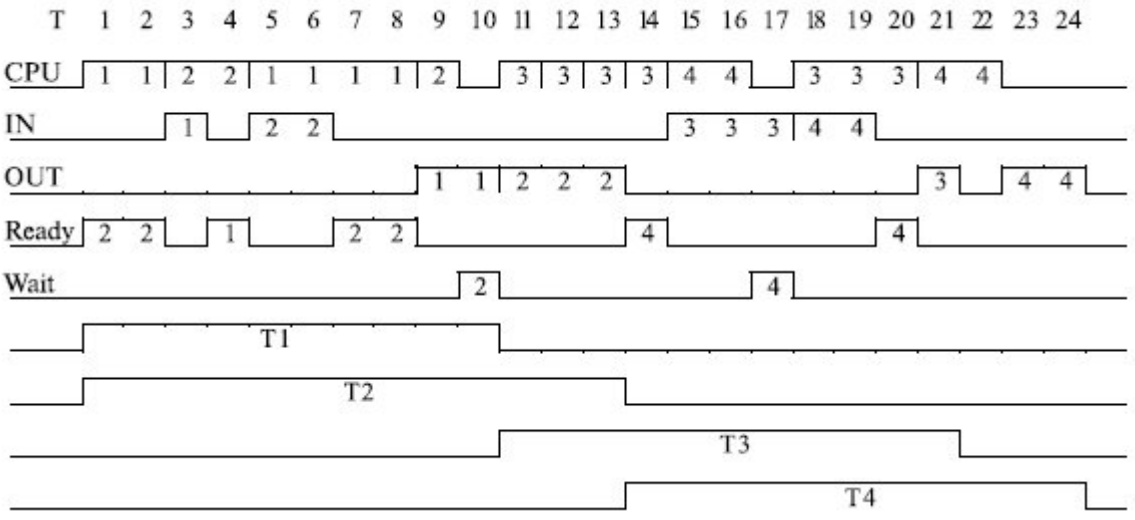

Рис. 6.2. Диаграмма выполнения программ в ЭВМ при Км = 2

Построив аналогичную диаграмму работы ЭВМ для Км = 3, получим результаты, представленные в табл. 6.2 (Км = 1 соответствует однопрограммной работе ЭВМ, и результаты для этого случая могут быть получены расчетными методами).

Анализ показывает, что с увеличением **коэффициента мультипрограммирования** *пропускная способность* ЭВМ будет увеличиваться, стремясь к некоторому пределу, определяемому характеристиками **ресурсов** ЭВМ. В то же время каждая *программа* будет выполняться в общем случае более длительное время из-за необходимости ожидания освобождения **ресурсов**, занятых другими программами. При увеличении **коэффициента мультипрограммирования** изменение значений *показателей эффективности* зависит от того, в каком состоянии находится система: перегрузки или недогрузки. Если какие-либо **ресурсы** ЭВМ задействованы достаточно интенсивно, то добавление новой программы, активно использующей эти ресурсы, будет малоэффективным.

Таблица 6.2. Характеристики работы ЭВМ при различных коэффициентах *мультипрограммирования*

Характеристика	**м = 1**	**м = 2**	**м = 3**
емя выполнения программы Т1			
Т2			
Т3			
Т4			
емя выполнения всех программ			
опускная способность (П)	1	7	3
	6	3	1
	2	3	6
	2	3	6

Микропроцессор для поддержки **мультипрограммного режима** работы использует определенный набор аппаратных средств.

В компьютере, работающем в **мультипрограммном режиме**, выделяется область памяти, доступная только операционной системе, в которой хранится вся *информация*, необходимая для *рестарта* задачи (контекстная *память*, или *кадр* состояния). При *переключении контекста компьютер* просто переходит от одного раздела к другому. В качестве такого раздела в универсальных микропроцессорах выступает **сегмент состояния задачи** TSS, который является небольшим сегментом данных с разрешенными операциями считывания и записи, - *доступ* к нему не разрешается никаким программам, даже на самом высоком **уровне привилегий**. К сегментам TSS может обращаться только сам *процессор*.

Для поддержки работы с сегментом состояния задачи служит 16-разрядный **регистр задачи** TR, в который заносится **селектор** дескриптора TSS, и связанный с TR программно недоступный 64-разрядный "теневой" *регистр*, в который загружается **дескриптор** TSS. Дескрипторы сегментов состояния задач хранятся только в **глобальной таблице дескрипторов***GDT*. Для переключения задач используется **шлюз задачи**.

Сегмент состояния задачи состоит из двух частей (рис. 6.3). Обязательная часть TSS объемом 104 байта содержит информацию, необходимую для *рестарта* данной задачи после ее вызова на *исполнение*, а также некоторую другую информацию. Дополнительная часть может содержать какую-либо информацию о задаче, используемую операционной системой (имя задачи, комментарии и т. д.), и **битовую карту ввода/вывода (БКВВ)**, определяющую устройства ввода/вывода, к которым разрешено обращение данной задачи при определенных ситуациях.

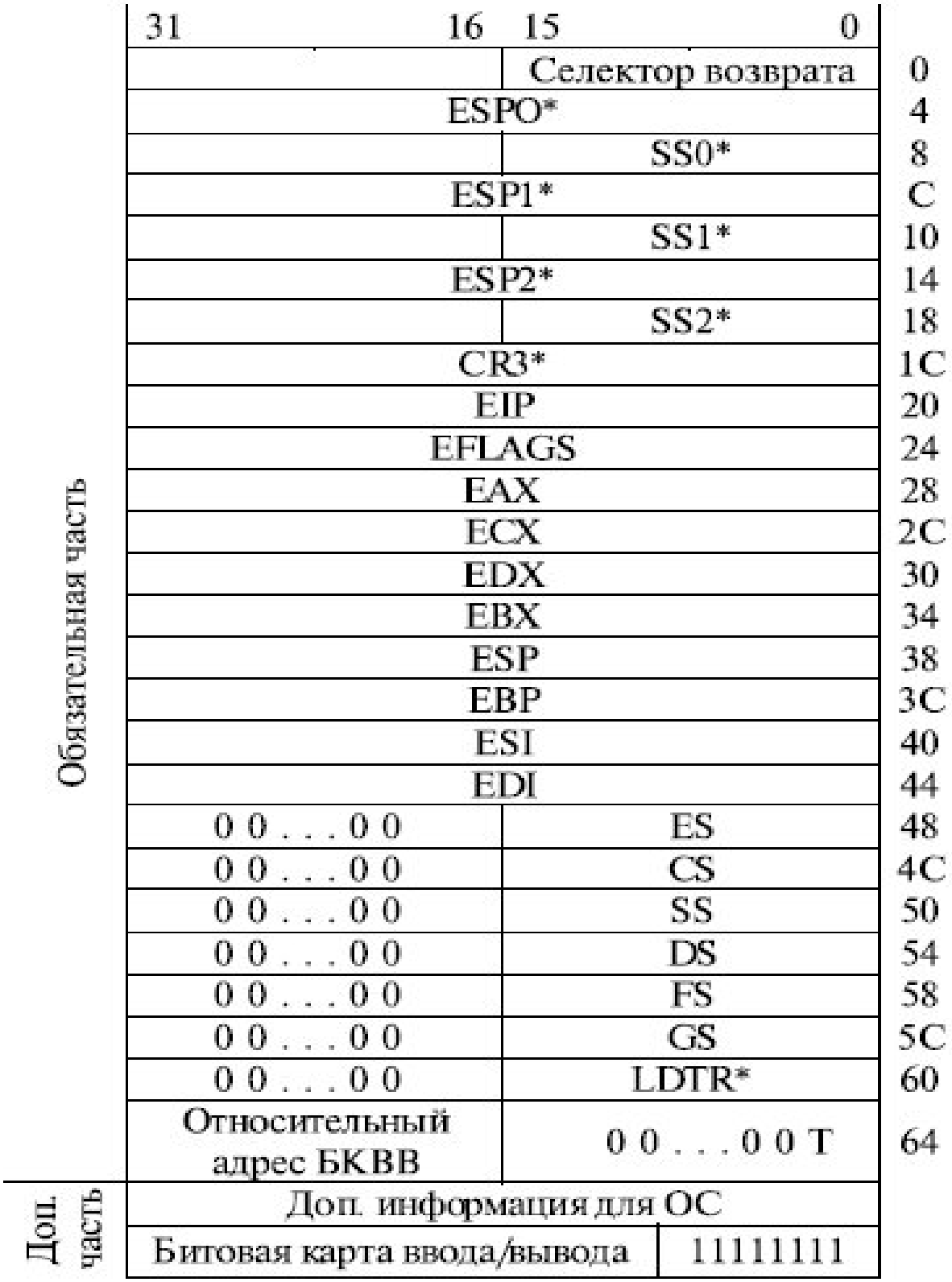

Рис. 6.3. Структура сегмента состояния задачи

Содержимое ряда полей TSS не изменяется при решении задачи (на рис. 6.3 они помечены звездочкой): селектор *LDT* данной задачи (то есть *регистр* LDTR), *регистр управления* CR3(базовый *адрес* **каталога таблиц страниц**), поля *SSi* и ESPi, которые определяют начальные адреса стеков при

переключении к задачам с более высоким **уровнем привилегий**, что обеспечивает их более надежную защиту.

При переключении задачи *процессор* может перейти к другой **локальной таблице дескрипторов** *LDT* и перезагрузить базовый *регистр* каталога страниц CR3. Это позволяет назначить каждой задаче свое *отображение логических адресов* на физические, что служит дополнительным средством защиты, так как задачи можно изолировать и предотвратить их взаимодействие друг с другом.

Поле селектора возврата обеспечивает *связь* данной задачи с вызвавшей ее программой. **Селектор возврата** - это **селектор** TSS предыдущей задачи, при выполнении которой произошел вызов данной задачи (TR предыдущей задачи, если предполагается возврат к ней).

Бит ловушки T используется при отладке программного обеспечения аналогично биту TF в регистре флагов: если T = 1, то при переключении на данную задачу возникает *прерывание*.

В TSS отсутствуют поля для хранения *регистров управления* CR0 и CR2. Это означает, что их содержимое не меняется при переключении задач.
Следовательно, *страничное преобразование* и условия работы с *FPU* являются глобальными для всех задач. Для каждой задачи может быть свой **каталог таблиц страниц**, но *страничное преобразование* может быть разрешено или запрещено только для *микропроцессорной системы* в целом.

Объем дополнительной части TSS зависит от количества служебной информации ОС, определяемой характером решаемой задачи, и размеров применяемой битовой карты ввода/вывода. Дополнительная часть TSS может вообще отсутствовать.

Относительный *адрес* БКВВ определяет положение битовой карты в TSS.
Каждый *бит* БКВВ соответствует однобайтовому порту ввода/вывода. Так как *микропроцессор* может обращаться к 216 портов, полная битовая карта ввода/вывода, определяющая возможность их обслуживания, -

это строка длиной до 64 Кбит. Когда для задачи определена БКВВ, ей предоставляется дополнительная возможность выполнения команд ввода/вывода. Если обычная защита *по* привилегиям запрещает ввод/*вывод* (*уровень привилегий* задачи меньше уровня, установленного в *поле* IOPL *регистра флагов*), *процессор* обращается к БКВВ для дополнительной проверки возможности ввода/вывода на конкретных устройствах. Если соответствующие этим адресам биты карты содержат нули, то *операции* ввода/вывода разрешаются. В противном случае формируется особый случай нарушения защиты. Таким образом, БКВВ обеспечивает задаче свободный *доступ* к незащищенным портам ввода/вывода без требования понижения значения в *поле* IOPL *регистра флагов*.

За последним байтом БКВВ в TSS должен следовать заключительный *байт*, состоящий из одних единиц. *Адрес* этого байта должен соответствовать границе сегмента, определенной дескриптором TSS.

После *поле* предела должна быть указана *длина* не менее 104 *байт* (*длина* обязательной части TSS). Если указанный размер сегмента меньше 104 *байт*, то происходит *прерывание*.

Байт доступа дескриптора TSS имеет следующий вид (рис. 6.4):

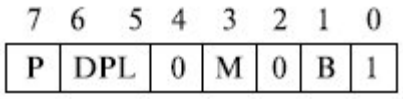

Рис. 6.4. Байт доступа дескриптора сегмента состояния задачи

У него есть ряд особенностей *по* сравнению с дескрипторами обычных сегментов.

Бит занятости B устанавливается в 1 при переключении на данную задачу. Используется для обнаружения попытки вызова задачи, выполнение которой прервано. Переключение задач производится, только если B = 0.

При B = 1 возникает *прерывание*. Установка B = 1 в дескрипторе TSS производится командами JMP и CALL, переключающими *микропроцессор* на выполнение данной задачи. При этом другие обращения в мультипрограммной системе к этой задаче будут запрещены.

В байте доступа дескриптора TSS *поле DPL* не означает *уровень привилегий* самого сегмента состояния задач, а аналогично функции поля *DPL* шлюза вызова показывает, какие программы могут обращаться к задаче, которая определяется данным дескриптором TSS.

Значение M = 0 обеспечивает совместимость с микропроцессором i286, в котором впервые появился **мультипрограммный режим работы**.

Переключение задач

Переключение задач осуществляется командами межсегментной передачи управления, при обработке прерываний и возврате из обработчиков. Если при этом управление передается **дескриптору сегмента состояния задачи** или **шлюзу задачи**, то происходит переключение задач. В соответствии с содержимым обязательной части TSS производится *загрузка*регистров МП, и он начинает выполнение поступившей задачи.

Шлюз задачи имеет следующий формат (рис. 6.5):

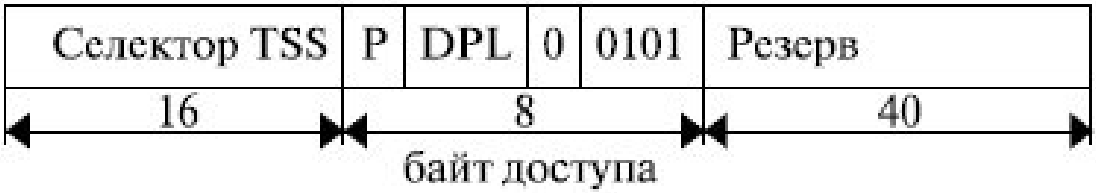

Рис. 6.5. Формат шлюза задачи

Он имеет статус *системного объекта*, и его тип, определенный в байте доступа, имеет *значение* 0101.

Переключение задач похоже на вызов процедуры, но при этом сохраняется больше информации. *Информация* о состоянии процессора сохраняется в TSS, а не в стеке.

Обращение к TSS осуществляется путем загрузки в *регистр* TR **селектора**, который адресует размещенный в *GDT* **дескриптор** TSS соответствующей задачи. Обычно *команда LTR*загрузки TR используется только при инициализации системы для установки начального содержимого TR.

В дальнейшем этот *регистр* загружается микропроцессором при выполнении команд, переключающих задачу.

При переключении задач проверяются привилегии, установленные для доступа к данным:

DPL' <= max (CPL, RPL),

т. е. допускается переключение на задачи, чья степень защиты меньше или равна **уровню привилегий** текущей программы и запроса.

Как правило, *поле DPL* дескриптора TSS равно 0, поэтому переключение задач могут производить только привилегированные программы.

Для переключения на задачи с большим приоритетом используется механизм **шлюзов задач**, аналогичный **шлюзам вызова**. При этом необходимо сохранение стеков более привилегированных задач, что и осуществляется с помощью сохранения *SSi,* ESPi в **сегменте состояния задачи**.

Для переключения задач в командах межсегментных переходов можно указать селектор шлюза задачи (косвенное переключение задачи) или селектор дескриптора TSS (прямое переключение задачи без привлечения шлюза задачи). Принципиальной разницы между этими переключениями нет. Но обычно цель использования шлюза задачи связана с мультизадачностью. Так как для каждой задачи имеется единственный **дескриптор** TSS, который должен находиться в *GDT*, все достаточно привилегированные задачи имеют к нему *доступ* и могут произвести переключение задачи. Но если определить для данного **дескриптора** TSS несколько шлюзов задачи и разместить их в

различных *LDT*, можно разрешить переключать задачи только тем задачам, которые имеют *шлюз* задачи в своих *LDT*. При этом сохраняется *доступ* к дескриптору TSS для программ с нулевым уровнем привилегий, которые могут прямо обращаться к нему в *GDT*.

Рассмотрим теперь переключение задач более подробно на примере механизма прямого переключения (рис. 6.6).

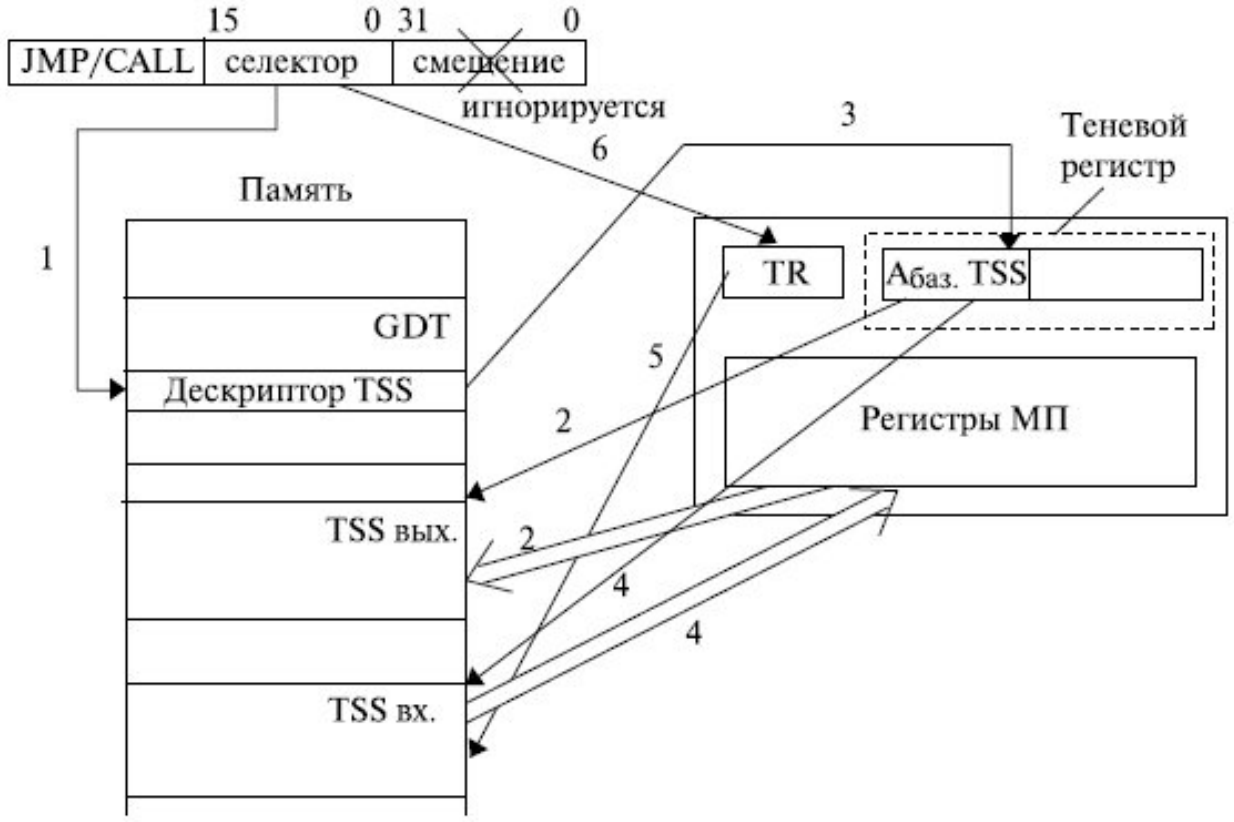

Рис. 6.6. Механизм прямого переключения задач

В общем случае *команда* межсегментного перехода содержит 3 поля:

- поле *кода операции* (JMP - *безусловный переход*, CALL - переход с возвратом),
- селектор нового сегмента команд, который заносится в *сегментный регистр* CS, и
- поле смещения, содержимое которого заносится в регистр - *указатель команд EIP*.

При переключении задач выполняется следующая последовательность действий:

1. В случае, когда тип дескриптора, определяемого вторым полем *команды перехода*, указывает, что данный дескриптор является дескриптором TSS и новая задача не занята (B = 0), запускается механизм переключения задач. В этом случае поле смещения в команде игнорируется.
2. В теневом регистре регистра задач TR микропроцессора к данному моменту содержится дескриптор TSS исполняемой задачи. Его поле адреса указывает на

область памяти, где должно быть сохранено состояние текущей задачи. Микропроцессор записывает в этот сегмент необходимую информацию.

3. Дескриптор TSS новой задачи переписывается из глобальной таблицы дескрипторов в теневой регистр регистра задач микропроцессора. Этот дескриптор определяет положение в памяти сегмента состояния новой задачи.
4. Информация о новой задаче переписывается из своего TSS в регистры микропроцессора.
5. Если команда, вызвавшая переключение задач, предполагает последующий возврат к старой задаче (CALL), то в поле **селектора возврата** TSS входящей задачи заносится содержимое регистра TR снимаемой с обработки задачи. При этом устанавливается значение бита вложенной задачи NT = 1 в регистре флагов EFLAGS.
6. В регистр TR микропроцессора из второго поля команды заносится селектор дескриптора TSS новой задачи.

Использование флага вложенной задачи NT и бита занятости B позволяет организовать корректную обработку вложенных задач. Порядок установки этих *бит* в зависимости от *типа команды*, вызвавшей переключение задач, указан в табл. 6.3.

блица 6.3. Модификация флагов занятости и вложенности при переключении задач

Тип команды	**Входящая задача**		**Выходящая задача**	
	Флаг NT в EFLAGS	**ит в дескрипторе TSS**	**Флаг NT в EFLAGS**	**ит в дескрипторе TSS**
CALL				
JMP				

Команда JMP при переключении на новую задачу не сохраняет в TSS **селектор возврата** и устанавливает NT = 0,а также B = 0 в дескрипторе старой задачи и B = 1 в дескрипторе новой задачи.

Команда CALL устанавливает B = 1 для новой задачи, но сохраняет B = 1 для предыдущей. Таким образом, каждая задача в цепи вызовов оказывается занятой, что запрещает применение рекурсивных процедур и реентерабельных программ.

Возврат из задачи осуществляется *по* команде IRET, которая анализирует флаг NT и при NT = 1 осуществляет переключение на задачу, задаваемую **селектором**

возврата в TSSтекущей задачи. При NT = 0 осуществляется обычная процедура возврата из процедуры с восстановлением из стека содержимого CS, *EIP*, EFLAGS.

Обычная *команда* RET возврата из подпрограммы не учитывает вложения задач.

Краткие итоги. В лекции рассмотрены принципы организации мультипрограммного режима обработки информации, представлены *аппаратные средства* микропроцессора, используемые для поддержки *мультипрограммирования*, разобран механизм переключения задач.

Лекция 7. Прерывания и особые случаи

Прерывания и особые случаи

Прерывание - это изменение естественного порядка выполнения программы, которое связано с необходимостью реакции системы на работу внешних устройств, а также на ошибки и особые ситуации, возникшие при выполнении программы. При этом вызывается специальная *программа* - **обработчик прерываний**, специфическая для каждой возникшей ситуации, после выполнения которой возобновляется работа прерванной программы.

Механизм прерывания обеспечивается соответствующими аппаратно-программными средствами компьютера.

Классификация прерываний представлена на рис. 7.1.

Рис. 7.1. Классификация прерываний

Запросы аппаратных прерываний возникают асинхронно по отношению к работе микропроцессора и связаны с работой внешних устройств.

Запрос от **немаскируемых прерываний** поступает на вход *NMI* микропроцессора и не может быть программно заблокирован. Обычно этот вход используется для запросов прерываний от схем контроля питания или неустранимых ошибок ввода/вывода.

Для запросов **маскируемых прерываний** используется вход INT микропроцессора. Обработка *запроса прерывания* по данному входу может быть заблокирована сбросом бита IF в **регистре флагов** микропроцессора.

Программные прерывания, строго говоря, называются исключениями или особыми случаями. Они связаны с особыми ситуациями, возникающими при выполнении программы (отсутствие страницы в оперативной памяти, нарушение защиты, *переполнение*), то есть с теми ситуациями, которые программист предвидеть не может, либо с наличием в программе специальной команды INT n, которая используется программистом для вызова функций операционной системы либо *BIOS*, поддерживающих работу с внешними устройствами. В дальнейшем при обсуждении работы системы прерываний мы будем употреблять единый термин

"*прерывание*" для аппаратных прерываний и исключений, если это не оговорено особо.

Программные прерывания делятся на следующие типы.

Нарушение (отказ) - особый случай, который *микропроцессор* может обнаружить до возникновения фактической ошибки (например, отсутствие страницы в оперативной памяти); после обработки нарушения *программа* выполняется с *рестарта* команды, приведшей к нарушению.

Ловушка - особый случай, который обнаруживается после окончания выполнения команды (например, наличие в программе команды INT n или установленный флаг TF в **регистре флагов**). После обработки этого прерывания выполнение программы продолжается со следующей команды.

Авария (*выход* из процесса) - столь серьезная ошибка, что некоторый *контекст* программы теряется и ее продолжение невозможно. Причину *аварии* установить нельзя, поэтому *программа*снимается с обработки. К *авариям* относятся аппаратные ошибки, а также несовместимые или недопустимые значения в системных таблицах.

Порядок обработки прерываний

Прерывания и особые случаи распознаются на границах команд, и программист может не заботиться о состоянии внутренних рабочих регистров и устройств конвейера.

Реагируя на запросы прерываний, *микропроцессор* должен идентифицировать его источник, сохранить минимальный *контекст* текущей программы и переключиться на специальную программу - обработчик прерывания. После обслуживания прерывания МП возвращается к прерванной программе, и она должна возобновиться так, как будто прерывания не было.

Обработка запросов прерываний состоит из:

- "рефлекторных" действий процессора, которые одинаковы для всех прерываний и особых случаев и которыми программист управлять не может;
- выполнения созданного программистом обработчика.

Для того чтобы *микропроцессор* мог идентифицировать источник прерывания и найти обработчик, соответствующий полученному запросу, каждому запросу прерывания присвоен свой номер (**тип прерывания**).

Тип прерывания для *программных прерываний* вводится изнутри микропроцессора; например, *прерывание* по отсутствию страницы в памяти имеет тип 14. Для прерываний, вызываемых командой INT n, тип содержится в самой команде. Для маскируемых аппаратных прерываний тип вводится из **контроллера**

приоритетных прерываний по *шине данных*. **Немаскируемому прерыванию** назначен тип 2.

Всего *микропроцессор* различает 256 **типов прерываний**. Таким образом, все они могут быть закодированы в 1 байте.

"Рефлекторные" действия микропроцессора по обработке *запроса прерывания* выполняются аппаратными средствами МП и включают в себя:

- определение **типа прерывания** ;
- сохранение контекста прерываемой программы (некоторой информации, которая позволит вернуться к прерванной программе и продолжить ее выполнение). Всегда автоматически сохраняются как минимум регистры *EIP* и CS, определяющие точку возврата в прерванную программу, и *регистр флагов* EFLAGS. Если вызов обработчика прерывания проводится с использованием шлюза задачи, то в памяти полностью сохраняется сегмент состояния TSS прерываемой задачи;
- определение адреса **обработчика прерывания** и передача управления первой команде этого обработчика.

После этого выполняется *программа* - **обработчик прерывания**, соответствующая поступившему запросу. Эта *программа* пишется и размещается в памяти прикладным или системным программистом. Обработчик прерывания должен завершаться командой I *RET*, по которой автоматически происходит переход к продолжению выполнения прерванной программы с восстановлением ее контекста.

Для вызова обработчика прерывания *микропроцессор* при работе в **реальном режиме** использует **таблицу векторов прерываний**, а в **защищенном режиме - таблицу дескрипторов прерываний**.

Таблица векторов прерываний (рис. 7.2) располагается в самых младших адресах оперативной памяти, имеет объем 1 Кбайт и содержит 4байтные элементы (**векторы прерываний**) для 256 обработчиков прерываний. Старшие 2 байта вектора загружаются в *сегментный регистр* команд CS, а младшие 2 байта - в **регистр указателя команд** IP. Обращение к элементам таблицы осуществляется по 8-разрядному коду - **типу прерывания**. Так как *таблица* всегда имеет нулевой начальный *адрес* и длину вектора в 4 байта, чтобы определить *адрес*вектора для прерывания типа i, достаточно просто умножить это *значение* на 4.

В **защищенном режиме** для вызова обработчика прерывания используется **таблица дескрипторов прерываний** *IDT*. Элементами таблицы являются 8-байтные *дескрипторы типа шлюз*-пециальные программные структуры, через которые происходит передача управления обработчику.

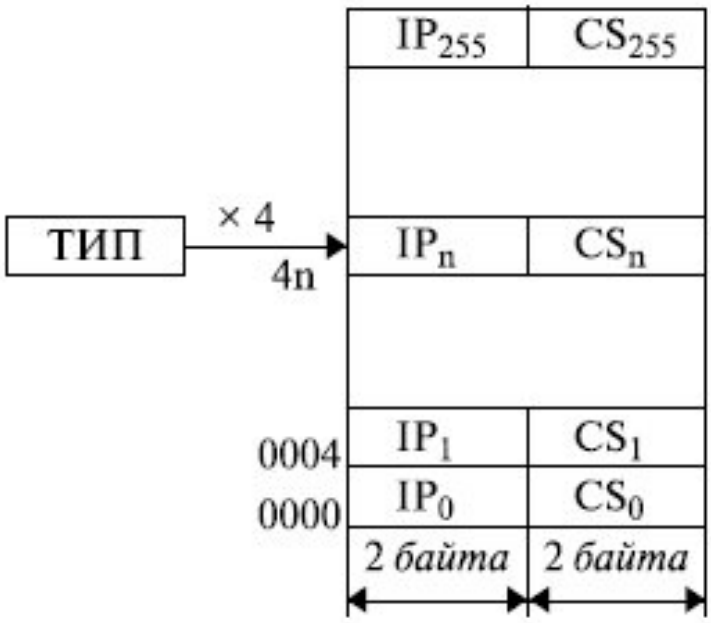

Рис. 7.2. Таблица векторов прерываний

Обращение к *IDT* аналогично обращению к **глобальной таблице дескрипторов**, где вместо системного регистра *GDT* R используется *регистр* IDTR, который определяет размер и базовый *адрес* таблицы в памяти.

Физический адрес дескриптора шлюза, находящегося в *IDT*, определяется как сумма базового адреса таблицы и умноженного на 8 типа прерывания (рис. 7.3).

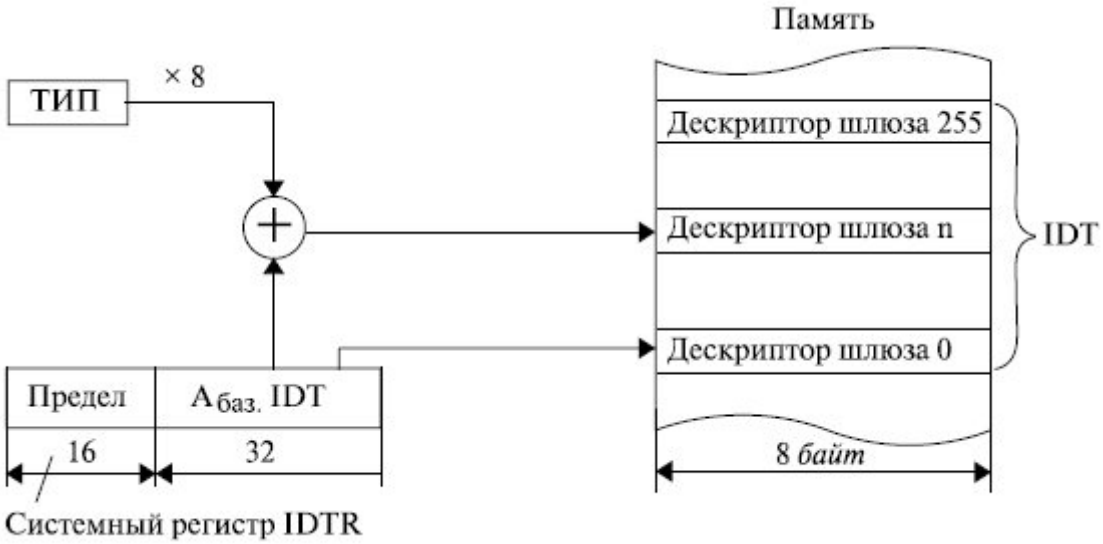

Рис. 7.3. Порядок обращения к таблице дескрипторов прерываний

Содержимое регистра IDTr не сохраняется в сегментах TSS и не изменяется при переключении задачи. Программы не могут обратиться к *IDT*, так как единственный *бит* TI**индикатора таблицы** в **селекторе** сегмента обеспечивает выбор только между таблицами *GDT* и *LDT*.

Максимальный *предел* таблицы дескрипторов прерываний составляет 256*8 - 1 = 2047.

Можно определить *предел* меньшим, но это не рекомендуется. Если происходит обращение к дескриптору вне пределов *IDT, процессор* переходит в режим отключения до получения сигнала по входу *NMI* или сброса.

В *IDT* могут храниться только дескрипторы следующих типов:

- шлюз *ловушки*,
- шлюз прерывания, шлюз задачи.

Шлюзы *ловушки* и прерывания сходны со **шлюзом вызова**, только в них отсутствует *поле* счетчика WC (рис. 7.4). Так как *прерывание* является неожиданным событием и не связано с текущей программой, говорить о *передаче параметров* их обработчику не приходится.

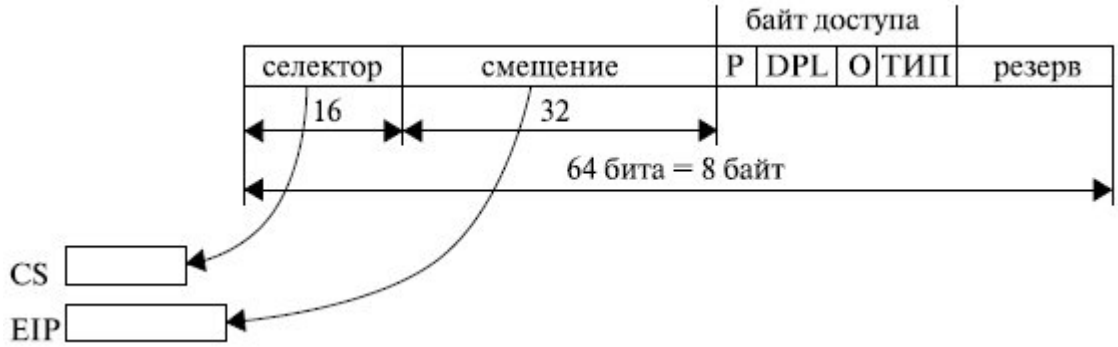

Рис. 7.4. Формат шлюзов ловушки и прерывания

Бит S = 0 в байте доступа определяет этот *дескриптор* как *системный объект*. Если *поле* ТИП в **байте доступа** равно 1110, то это **шлюз прерывания**, если 1111 - то **шлюз ловушки**.

Поле **уровня привилегий** дескриптора *DPL*, как правило, устанавливается равным 3 с тем, чтобы к **обработчику прерываний** могли обращаться программы с любого уровня привилегий.

Бит присутствия P может быть равен как 0, так и 1.

При входе вобработчик через *шлюз* прерывания в регистре флагов сбрасывается **бит разрешения прерываний** IF. В этом случае *микропроцессор* блокирует все **маскируемые аппаратные прерывания**. Поэтому в обработчике прерываний этот *бит* должен быть установлен в 1 как можно раньше с тем, чтобы не блокировать работу программ, которые вызываются, например, при обработке прерываний от *системного таймера*.

При входе вобработчик через *шлюз ловушки* флаг IF не меняется.

Вызов обработчика через *шлюз ловушки*, а не *шлюз* прерывания, чаще реализуют при *обработке исключений*, так как на период обслуживания прерывания нежелательно выключать механизм разделения времени, использующий прерывания таймера.

Вызов обработчика через **шлюз задачи** обычно осуществляется при обработке аппаратных прерываний, так как такая обработка не связана с текущей выполняемой задачей. При этом возможен механизм вложенных прерываний, если прерывания в задаче разрешены. Вызов обработчика прерывания через *шлюз* задачи осуществляется и при *обработке исключений*, например, "неразрешенный TSS ", когда поврежденная задача не может вызвать процедуру прерывания. Переключение задач требует примерно в 5 раз больше времени, чем вызов процедуры. Поэтому, если *приоритет запроса* высок, а *программа* обслуживания короткая, ее оформляют в виде процедуры.

Контроллер приоритетных прерываний

Прерывание - один из наиболее дефицитных ресурсов в *микропроцессорной системе*. *Микропроцессор* имеет только 2 входа для приема запросов прерываний: вход INT - по нему принимаются запросы, обработка которых может быть замаскирована сбросом флага IF в регистре флагов, - и вход немаскируемых прерываний *NMI*. Вход *NMI* фактически закреплен за запросами прерываний от схем контроля питания. Поэтому при такой архитектуре микропроцессора в *микропроцессорной системе* обязательно должны использоваться средства, которые позволяют предварительно обрабатывать и передавать на вход *маскируемых прерываний* INT микропроцессора запросы от многочисленных внешних устройств, входящих в состав микропроцессорной системы. В качестве такой схемы используется **контроллер приоритетных прерываний** (КПП).

Мы рассмотрим его функционирование на примере БИС i8259, которая, с одной стороны, имеет самостоятельное *значение*, а с другой стороны, фактически без изменений входит в состав современных чипсетов.

Его структура представлена на рис. 7.5.

Функции *контроллера приоритетных прерываний*:

- восприятие и фиксация до 8 запросов прерываний (IRQ0 - IRQ7), поступающих по внешним входам;
- выделение наиболее приоритетного из поступивших запросов, включая возможность маскирования отдельных запросов;
- выдача на *шину данных* (по требованию микропроцессора) **типа** выбранного **прерывания**.

При использовании КПП обработка запросов *немаскируемых прерывание* проходит следующие этапы:

1. Системная *периферия* на системной плате или устройство ввода/вывода на внешней шине активирует одну из линий IRQx.

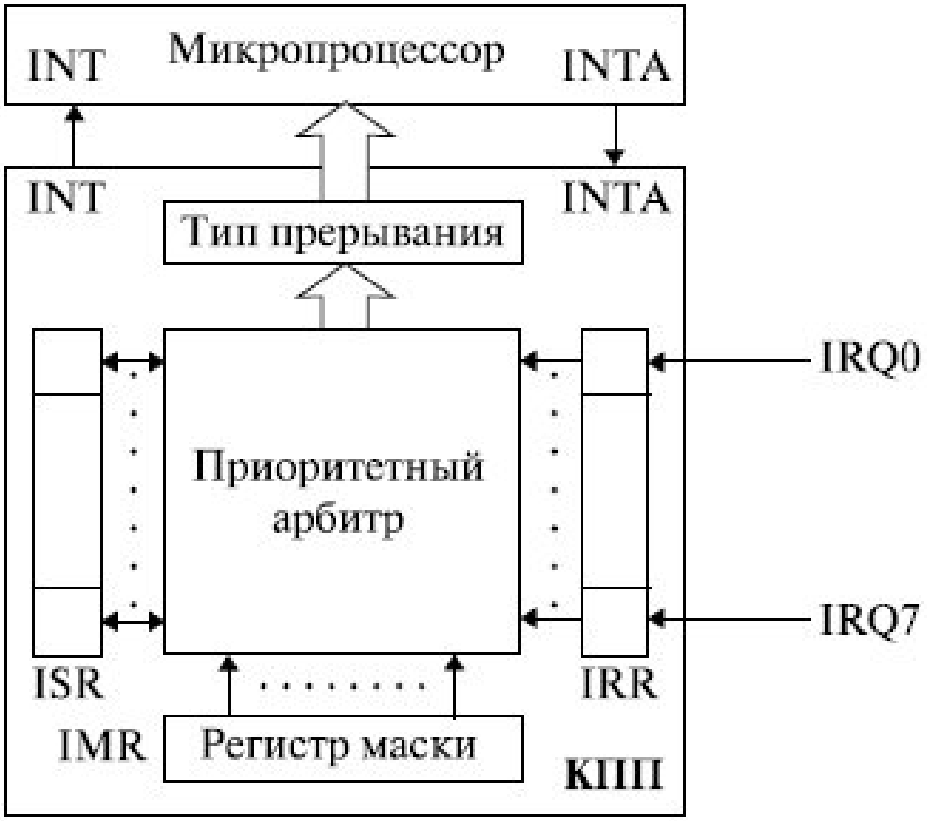

Рис. 7.5. Структура контроллера приоритетных прерываний

2. В **регистре запросов прерываний** *IRR*, который предварительно настраивается на восприятие запросов по спаду или низкому уровню сигнала, происходит установка соответствующих разрядов в "1".
3. Незамаскированные в **регистре маски** *IMR* запросы передаются в приоритетный арбитр, замаскированные блокируются.
4. В соответствии с выбранной в процессе инициализации дисциплиной обслуживания **приоритетный арбитр** выделяет наиболее приоритетный запрос. При системном сбросе контроллера самый высокий приоритет устанавливается для запроса, приходящего по входу IRQ0, а самый низкий - по входу IRQ7.
5. Приоритет выделенного запроса сравнивается с *приоритетом запроса*, который в данный момент может обрабатываться микропроцессором (его номер установлен в **регистре обслуживания прерываний** *ISR*). Если приоритет нового запроса выше либо в данный момент обслуживаемых запросов нет, то контроллер формирует *сигнал прерывания* INT в микропроцессор, в противном случае обработка запроса откладывается. В **регистре типа прерывания** формируется тип принятого к обработке *запроса прерывания*.
6. МП воспринимает *запрос прерывания*, и если флаг IF = 1, то по завершении текущей команды выполняет 2 цикла подтверждения прерывания, выдавая сигналы на выход INTA:

- в 1-м цикле запрещается запись в *IRR*. В *ISR* устанавливается разряд, соответствующий принятому к обработке запросу, и сбрасывается разряд в *IRR* ;
- во 2-м цикле **тип прерывания** передается в МП по разрядам D0D7 шины данных. Разрешается запись в *IRR* ;

7. МП принимает **тип прерывания** и использует его в качестве индекса при обращении к соответствующей **таблице прерываний** (**таблице векторов** или **таблице дескрипторов прерываний** в зависимости от режима работы МП).
8. В соответствии с установленным в микропроцессоре режимом работы (реальном или защищенном) и механизмом вызова программы - обработчика прерывания МП сохраняет необходимую информацию о прерываемой программе и переходит к выполнению **обработчика прерывания**.
9. Команда *IRET*, завершающая **обработчик прерываний**, восстанавливает прежнее состояние микропроцессора и передает управление прерванной программе.

Каскадное включение контроллеров приоритетных прерываний

Для расширения количества запросов прерываний, которые могут быть подключены к микропроцессору, в *микропроцессорной системе* может быть использовано несколько КПП. Схема каскадного подключения двух контроллеров представлена на рис. 7.6.

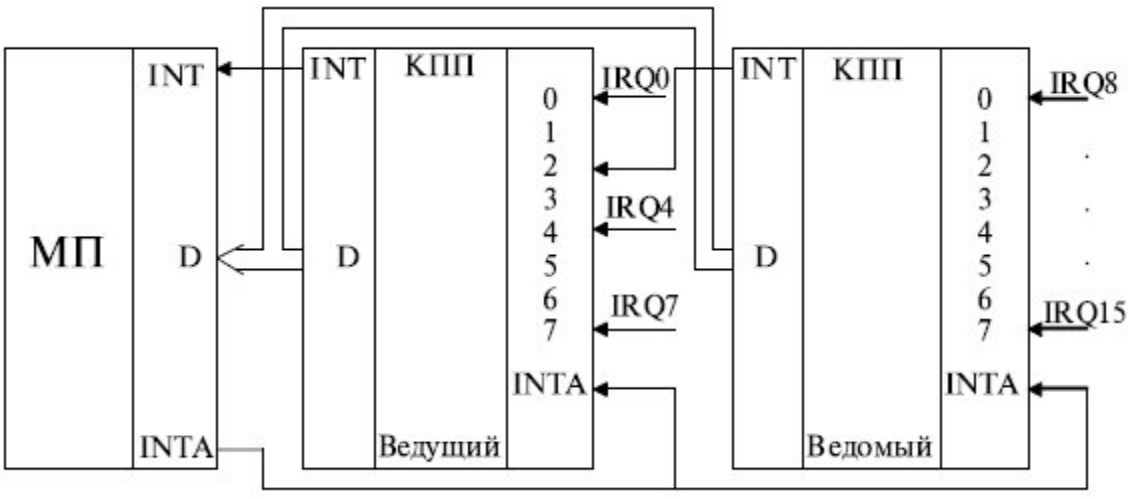

Рис. 7.6. Каскадное подключение контроллеров приоритетных прерываний к микропроцессору

К входу INT микропроцессора подключается *выход* INT ведущего контроллера. *Выход* INT ведомого контроллера подключается к одному из входов IRQi ведущего КПП на правах других запросов прерываний, поступающих на этот *контроллер*. В *персональной ЭВМ* всегда используются два *контроллера приоритетных прерываний*, причем ведомый КПП подключен к входу IRQ2 ведущего.

На рис. 7.5 было дано схематическое *представление контроллера приоритетных прерываний*. Для того чтобы лучше понять функционирование **контроллеров приоритетных прерываний**в реальных *микропроцессорных системах* и оценить все имеющиеся у них возможности, рассмотрим структуру КПП более подробно.

Регистры КПП делятся на 2 группы: регистры инициализации ICW1-ICW4 и операционные регистры OCW1-OCW3.

Регистры инициализации загружаются при инициализации контроллера и в процессе работы КПП не меняются.

Регистр ICW1 - управление микросхемой:

- настраивает контроллер на восприятие сигналов запроса по низкому уровню или заднему фронту;
- определяет, используется в МПС единственный КПП или применяется их каскадное включение;
- определяет порядок загрузки приказов инициализации. Необходимость этого обусловлена тем, что в пространстве ввода/вывода каждому контроллеру выделено всего 2 адреса. Так, в стандартной конфигурации *персональной ЭВМ* ведущему контроллеру выделены адреса 20h и 21h, а ведомому - A0h и A1h. В то же время каждый контроллер имеет в своем составе 7 регистров, к которым должен быть обеспечен программный доступ. В частности, при инициализации необходимо занести информацию в 4 регистра ICWi.

Регистр ICW2 - *регистр* **типа прерывания**. При инициализации в 5 его старших разрядов заносится некоторая константа. В процессе обработки запросов прерываний в 3 младшие разряда этого регистра заносится номер входа IRQi, по которому принят *запрос*, подлежащий обработке (рис. 7.7):

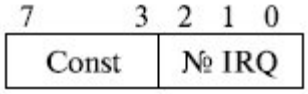

Рис. 7.7. Формат регистра типа прерывания ICW2

В *персональной ЭВМ* в *регистр* ICW2 ведущего контроллера при инициализации заносится константа 00001b, а в ведомый - константа 01110b. Поэтому типы всех прерываний, запросы от которых поступают через ведущий КПП, лежат в диапазоне 00001000b-00001111b (08h-0Fh), ачерез ведомый - в диапазоне 01110000b-01110111b (70h-77h). Распределение входов прерываний в стандартной конфигурации *персональной ЭВМ* представлено в табл. 7.1.

Таблица 7.1. Распределение входов запросов прерываний в *персональной ЭВМ*

Контроллер	Назначение	альное состояние маски
щий контроллер	ıер	
	иатура	
	мый КПП	
	едовательный порт 2 (COM2)	
	едовательный порт 1 (COM1)	
	едовательный принтер (*LPT2*)	
	ИД	
	ллельный принтер (*LPT1*)	
мый контроллер	ı реального времени	
	рв. Программно переназначен на IRQ2	
	рв	
	рв	
	ıь	
	бка сопроцессора	
	МД	
	рв	

Регистр ICW3 - *регистр* управления ведомым. Имеет различное назначение в ведущем и ведомом КПП. В ведущем КПП устанавливаются единицы в разрядах,

соответствующих линиям с подключенными ведомыми КПП. В *персональной ЭВМ* его *значение* имеет вид 00000100b. В ведомом КПП пять старших разрядов этого регистра установлены в 0, а в трех младших кодируется номер входа ведущего КПП, к которому подключен данный ведомый. В *персональной ЭВМ* его *значение* имеет вид 00000010b.

Регистр ICW4 - *регистр* управления режимом. Определяет, является данный КПП ведущим или ведомым, тип окончания прерывания, то есть кем должен сбрасываться *бит* запроса в регистре обслуживания прерывания *ISR*, и другие параметры работы.

Так как для КПП определено только 2 допустимых состояния (ведущий или ведомый), максимальная *конфигурация контроллеров приоритетных прерываний* состоит из 1 ведущего и 8 ведомых КПП. Это обеспечивает возможность подключения к входу INT микропроцессора до 64 запросов прерываний.

Содержимое **операционных регистров** изменяется в процессе работы КПП записью в них новой информации.

Регистр OCW1 (*IMR*) - **регистр маски прерывания**. Код 1 в разряде i запрещает, а код 0 - разрешает обработку *запроса прерывания* по входу IRQi.

Регистр OCW2 - определяет один из трех возможных порядков изменения приоритетов запросов прерываний:

- приоритеты не меняются в процессе работы КПП;
- приоритеты меняются циклически: после обработки очередного запроса его приоритет становится самым низким, а приоритеты остальных запросов циклически сдвигаются;
- процессе работы КПП какому-либо запросу можно задать наивысший приоритет, приоритеты остальных запросов при этом циклически сдвигаются.

В *персональной ЭВМ* установлено постоянство приоритетов запросов прерываний, при этом *запрос* IRQ0 имеет самый высокий приоритет.

Так как здесь используются два каскадно включенных контроллера (см.рис. 7.6), приоритеты запросов прерываний IRQi имеют вид, представленный на рис. 7.8.

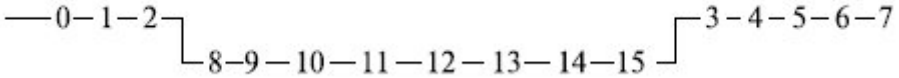

Рис. 7.8. Приоритетность запросов прерываний IRQi в персональной ЭВМ

Регистр OCW3 - управляет переводом контроллера в режим неприоритетного обслуживания и считыванием содержимого **регистра запросов** *IRR* и **регистра обслуживания** *ISR*. В режиме неприоритетного

обслуживания *микропроцессор* получает от КПП только сигнал *запроса прерывания*, после чего *микропроцессор* должен программно считать содержимое регистров *IRR* и *ISR* и по своим алгоритмам определить, какой из имеющихся запросов прерываний принять к обслуживанию.

Краткие итоги. В лекции рассмотрена классификация прерываний, порядок обработки прерываний в реальном и защищенном режимах работы микропроцессора, структура и функционирование *контроллера приоритетных прерываний*.