МИНИСТЕРСТВО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
КАБАРДИНО-БАЛКАРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ЗАЩИЩЕННЫЙ РЕЖИМ МИКРОПРОЦЕССОРОВ PENTIUM

Пособие для самостоятельной работы

Для специальности: 220200 – Автоматизированные системы обработки информации и управления

НАЛЬЧИК 2003

УДК.681.3.06
ББК 32.973

Рецензент:
кандидат физико-математических наук, ученый секретарь
Института информатики и проблем регионального управления КБНЦ РАН
Г.В. Чернышев

Составитель: **Солодовникова О.С.**
Защищенный режим микропроцессоров Pentium: Пособие для самостоятельной работы. – Нальчик: Каб.-Балк. ун-т., 2003. – 92 с.

В работе рассмотрены вопросы, связанные с такими особенностями микропроцессоров Pentium, как защищенный режим и обработка прерываний в нем. Приводятся примеры программ, демонстрирующие работу микропроцессора в этом режиме.

Издание предназначено для углубленного изучения архитектуры микропроцессоров Intel Pentium в рамках соответствующих дисциплин специальности 220200 – АСОИиУ, а также всем, кто профессионально занимается компьютерными технологиями и программированием для микропроцессоров Intel на низком уровне. Эти знания полезны не только студентам и разработчикам, но и любознательным пользователям персональным компьютером.

Рекомендовано РИСом университета

УДК.681.3.06
ББК 32.973.

ВВЕДЕНИЕ

Данное пособие охватывает ключевые аспекты, связанные с развитием современной вычислительной техники. В нем рассматриваются особенности микропроцессоров Pentium с точки зрения системотехника, причем как разработчика аппаратных средств, так и системного программиста.

В пособии рассматриваются вопросы, которые являются частью дисциплины «Организация ЭВМ и систем», читаемой автором в высшем учебном заведении. Это наложило отпечаток не только на методику изложения материала, но и позволило расставить необходимые акценты на тех вопросах, которые обычно вызывают трудности у студентов. Пособие поможет разобраться в режимах работы современных процессоров – защищенном, виртуальном и режиме системного управления. Изучив его, вы поймете, что делается «за кулисами» операционной системы, откуда берется сообщение «Нарушение общей защиты» и какие «недопустимые операции» пыталось выполнить приложение, снятое операционной системой защищенного режима.

Впервые защищенный режим появился в микропроцессоре i80286 фирмы Intel. Именно этот режим позволяет полностью использовать все возможности, предоставляемые микропроцессором. Все современные многозадачные операционные системы работают только в этом режиме. Такие операционные системы стали стандартом. Поэтому так важно для понимания всех вопросов, происходящих в компьютере во время работы многозадачной операционной системы, знать основы функционирования микропроцессора в защищенном режиме.

Любой современный микропроцессор, находясь в реальном режиме, очень мало отличается от «старого, доброго» i8086. Это лишь его более быстрый аналог с увеличенным (до 32 бит) размером всех регистров, кроме сегментных. Чтобы получить доступ ко всем остальным архитектурным и функциональным новшествам микропроцессора, необходимо перейти в защищенный режим. Если бы мы могли проникнуть внутрь компьютера после перехода в защищенный режим, то увидели бы, что микропроцессор совершенно преобразился. Прежде всего, это стало бы заметно по изменениям принципов работы микропроцессора с памятью. Она по-прежнему остается сегментированной, но изменяются функции и номенклатура программно-аппаратных компонентов, участвующих в сегментации. В реальном режиме аппаратные средства контроля доступа к сегменту отсутствовали. Если и можно было организовать такой контроль, то только со стороны операционной системы. Реальный режим поддерживал выполнение всего одной программы. Не возникала потребность в защите программ от их взаимного влияния. Для обес-

печения совместной работы нескольких задач необходимо защитить их от взаимного влияния, а если возникает потребность во взаимном действии между ними, то оно должно обязательно регулироваться. В защищенном режиме микропроцессор поддерживает два типа защиты – по привилегиям и доступу к памяти. Эти и другие ключевые моменты представлены для изучения.

Для изучения примеров программ необходимо иметь компьютер с процессором Intel Pentium. Из программного обеспечения необходим редактор текстов, хотя бы самый простой, и пакет транслятора ассемблера фирм Microsoft или Borland. Желательно отдать предпочтение транслятору Turbo Assembler (TASM) фирмы Borland, потому что именно он использовался при разработке примеров программ пособия. Ну и конечно, необходимы большое желание, терпение и тяга к познанию. Сложность еще и в том, что восприятие данного пособия неразрывно связано с множеством других вопросов, рассматриваемых в курсе дисциплины «Организация ЭВМ и систем». С этой точки зрения следует воспринимать данное пособие как описание конкретного режима микропроцессора и изучение приемов работы с ним для решения конкретных задач.

1. ЗАЩИЩЕННЫЙ РЕЖИМ

Защищенный режим *Protected Mode,* более торжественно называемый *Protected Virtual Address Mode* (защищенный режим виртуальной адресации), является основным режимом работы 32-разрядных процессоров. В этом режиме процессор позволяет адресовать до 4 Гбайт физической памяти, через которые при использовании механизма сегментации могут отображаться до 64 Тбайт виртуальной памяти каждой задачи. Режим виртуального процессора 8086 – *Virtual 8086 Mode* или *V86* – является особым состоянием задачи защищенного режима, в котором процессор функционирует как 8086 с возможностью использования 32-разрядных адресов и операндов.

Защищенный режим появился еще в процессоре 80286, но имел не все возможности, доступные в 32-разрядных процессорах. Приведенное ниже описание защищенного режима применимо и к процессору 80286, но с учетом следующих ограничений:

• регистры CRn, DRn и TRn отсутствуют;

• вход в защищенный режим осуществляется только по загрузке *MSW* с PE=1, выход в реальный – только по аппаратному сбросу;

• режима виртуального 8086 нет;

• форматы дескрипторов имеют 16-битные поля лимита и 24-битные поля базового адреса, что ограничивает размер сегмента до 64 Кбайт, объем физической памяти не превосходит 16 Мбайт, виртуальной – 1 Гбайт;

• обращение к памяти за границей 16 Мбайт приводит к кольцевому “сворачиванию” адреса, – обращение к 17-му мегабайту физически адресует первый (что справедливо и для 386SX);

• режимы 32-битных адресов и данных отсутствуют;

• блок страничной переадресации отсутствует (физический адрес памяти эквивалентен логическому);

• ограничения на операции ввода/вывода накладываются только через *IOPL*, битовой карты разрешения ввода/вывода нет.

1.1. Основные понятия защищенного режима

Защищенный режим предназначен для обеспечения независимости выполнения нескольких задач, что подразумевает защиту ресурсов одной задачи от возможного воздействия другой.

Основным защищаемым ресурсом является память, в которой хранятся коды, данные и различные системные таблицы (например, таблица прерываний). Защищать требуется и совместно используемую аппаратуру, обращение к которой обычно происходит через операции ввода/вывода и прерывания. В защищенном режиме процессор аппаратно реализует многие функции защиты, необходимые для построения супервизора многозадачной ОС, в том числе механизм виртуальной памяти.

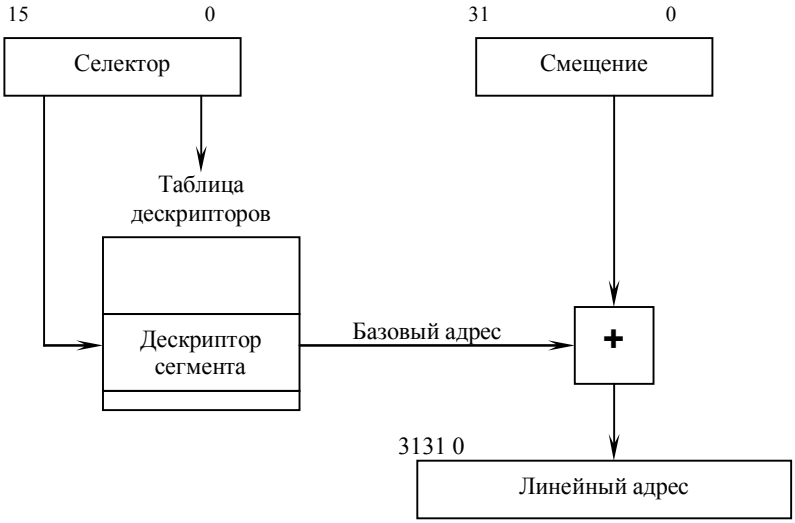

Рис. 1. Формирование линейного адреса в защищенном режиме

Защита памяти основана на *сегментации. Сегмент* – это блок пространства памяти определенного назначения. К элементам сегмента возможно обращение с помощью различных инструкций процессора, использующих разные режимы адресации для формирования адреса в пределах сегмента. Максимальный размер сегмента – 4 Гбайт (для процессоров 8086 и 80286 предел был всего 64 Кбайт). Сегменты памяти выделяются задачам операционной системой, но в реальном режиме любая задача может переопределить значение сегментных регистров, задающих положение сегмента в пространстве памяти, и "залезть" в чужую область данных или кода. В защищенном режиме сегменты тоже распределяются операционной системой, но прикладная программа сможет использовать только разрешенные для нее сегменты памяти, выбирая их с помощью *селекторов* из предварительно сформированных *таблиц дескрипторов сегментов.*

Процессор может обращаться только к тем сегментам памяти, для которых имеются дескрипторы в таблицах. Механизм сегментации формирует линейный адрес по схеме, приведенной на рис.1. Дескрипторы выбираются с помощью 16-битных селекторов, программно загружаемых в сегментные регистры; формат селекторов приведен на рис.2. Индекс, указывающий номер дескриптора в таблицы, совместно с индикатором таблицы TI позволяет выбрать дескриптор из локальной (TI=1) или глобальной (TI=0) таблицы дескрипторов. Для неиспользуемых сегментных регистров предназначен нулевой селектор сегмента, формально адресующийся к самому первому элементу глобальной таблицы. Попытка обращения к памяти по такому сегмент-

ному регистру вызовет исключение. Исключение возникнет и при попытке загрузки нулевого селектора в регистр *CS* или *SS*. Поле *RPL* указывает требуемый уровень привилегий.

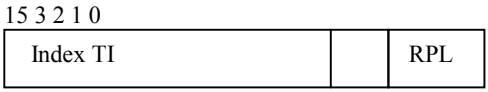

Рис. 2. Формат селектора

Дескрипторы представляют собой 8-байтные структуры данных, используемые для определения свойств программных элементов (сегментов, вентилей и таблиц). Дескриптор определяет положение элемента в памяти, размер занимаемой им области (лимит), его назначение и характеристики защиты. Все дескрипторы хранятся в таблицах, обращение к которым поддерживается процессором аппаратно.

Защита памяти с помощью сегментации не позволяет:

• использовать сегменты не по назначению (например, пытаться трактовать область данных как коды инструкций);

• нарушать права доступа (пытаться модифицировать сегмент, предназначенный только для чтения, обращаться к сегменту, не имея достаточных привилегий, и т. п.);

• адресоваться к элементам, выходящим за лимит сегмента;

• изменять содержимое таблиц дескрипторов (то есть параметров сегментов), не имея достаточных привилегий.

Защищенный режим предоставляет средства *переключения задач.* Состояние каждой задачи (значение всех связанных с ней регистров процессора) может быть сохранено в специальном сегменте состояния задачи *TSS(Task State Segment),* на который указывает селектор в регистре задачи *TR*. При переключении задач достаточно загрузить новый селектор в регистр задачи, и состояние текущей задачи автоматически сохранится в ее *TSS,* а в процессор загрузится состояние новой (возможно, ранее прерванной) задачи, и начнется (продолжится) ее выполнение.

Четырехуровневая иерархическая система привилегий предназначена для управления использованием привилегированных инструкций и доступом к дескрипторам. Уровни привилегий нумеруются от 0 до 3, нулевой уровень соответствует максимальным (неограниченным) возможностям доступа и отводится для ядра операционной системы. Уровень 3 имеет самые ограниченные права и обычно предоставляется прикладным задачам. Систему защиты обычно изображают в виде концентрических колец соответствующих уровням привилегий рис.3, а сами уровни привилегий иногда называют кольцами защиты. Сервисы, предоставляемые задачам, могут находиться в разных кольцах защиты. Передача управления между задачами контролируется *вентилями* (Gate), называемыми также *шлюзами,* проверяющими правила

использования уровней привилегий. Через вентили задачи могут получить доступ только к разрешенным им сервисам других сегментов.

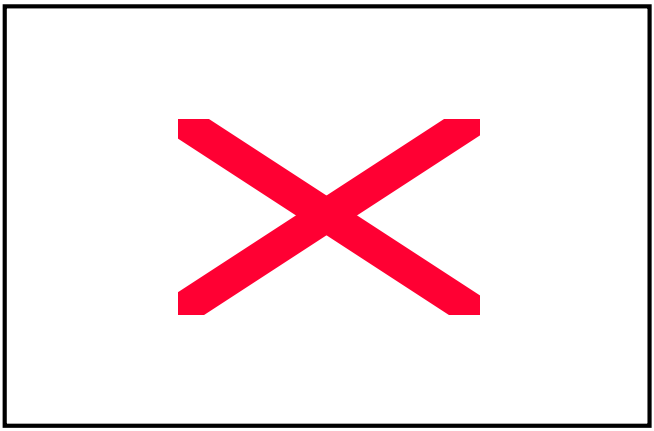

Рис. 3. Уровни привилегий

Уровни привилегий относятся к дескрипторам, селекторам и задачам. Кроме того, в регистре флагов имеется *поле привилегий ввода/вывода, с* помощью которого обеспечивается управление доступом к инструкциям ввода/вывода и управлению флагом прерываний.

Дескрипторы и привилегии являются основой системы защиты, дескрипторы определяют структуры программных элементов (без которых эти элементы невозможно использовать), а привилегии определяют возможность доступа к дескрипторам и выполнения привилегированных инструкций. Любое нарушение защиты приводит к возникновению специальных исключений, обрабатываемых ядром операционной системы.

Механизм виртуальной памяти позволяет любой задаче использовать логическое адресное пространство размером до 64 Тбайт (16К сегментов по 4 Гбайт). Для этого каждый сегмент в своем дескрипторе имеет специальный бит, который указывает на присутствие данного сегмента в оперативной памяти в текущий момент времени. Неиспользуемый сегмент может быть выгружен из оперативной памяти во внешнюю (например, дисковую), о чем делается пометка в его дескрипторе. На освободившееся место из внешней памяти может восстанавливаться содержимое другого сегмента (этот процесс называется свопингом или подкачкой), и в его дескрипторе делается пометка о присутствии в памяти. При обращении задачи к отсутствующему сегменту процессор вырабатывает соответствующее исключение, обработчик которого и заведует виртуальной памятью в операционной системе. Механизм страничной переадресации обеспечивает виртуализацию памяти, адресуемой ло-

гическим адресом, на уровне страниц фиксированного размера. После подкачки сегмента (страницы) выполнение задачи продолжается, так что виртуализация памяти для прикладных задач прозрачна (если не принимать во внимание задержку, вызванную подкачкой).

Процессор предоставляет только необходимые аппаратные средства поддержки защиты и виртуальной памяти, а их реальное использование и устойчивость работы программ и самой операционной системы защищенного режима, конечно же, зависят от корректности построения ОС и предусмотрительности ее разработчиков. Хорошо спроектированная операционная система защищенного режима может обеспечить устойчивость ОС даже при некорректном поведении прикладных задач.

1.2. Дескрипторы и таблицы

Существуют *три типа таблиц дескрипторов* – локальная таблица дескрипторов *LDT* (Local Descriptor Table), глобальная таблица дескрипторов *GDT* (Global Descriptor Table) и таблица дескрипторов прерываний *IDT* (Interrupt Descriptor Table). Размеры таблиц могут находиться в пределах 8 байт – 64 Кбайт, что соответствует числу элементов в таблице от 1 до 8 К.

С каждой из этих таблиц связан соответствующий регистр процессора. Регистры *GDTR* и *IDTR* имеют программно-доступное 16-битное поле лимита, задающее размер таблицы, и 32-битное (у 80286 – 24-битное) поле базового адреса, определяющее положение таблицы в пространстве линейных (у 80286 – физических) адресов памяти. У регистра *LDTR* программно доступно только 16-битное поле селектора, по которому из *GDT* автоматически загружаются программно недоступные и невидимые поля базового адреса и лимита.

Команды загрузки регистров таблиц *LGDT, LIDT* и *LLDT* являются привилегированными (выполняются только на уровне привилегий 0). Команды *LGDT* и *LIDT* загружают из памяти 6-байтное поле, содержащее базовый адрес и лимит локальной таблицы. Команда *LLDT* загружает только селектор, ссылающийся на дескриптор, содержащий базовый адрес и лимит локальной таблицы дескрипторов.

Глобальная таблица (GDT) содержит дескрипторы, доступные всем задачам. Она может содержать дескрипторы любых типов, кроме дескрипторов прерываний и ловушек. Нулевой элемент этой таблицы процессором не используется. *Локальная таблица (LDT)* может быть собственной для каждой задачи и содержит только дескрипторы сегментов, вентилей задачи и вызовов. Сегмент недоступен задаче, если его дескриптора нет в текущий момент ни в *GDT*, ни в *LDT*.

Выбор таблицы (локальная или глобальная) определяется по значению бита *TI* селектора, а положение (номер) дескриптора задается 13-битным полем *INDEX* селектора. При ссылке на дескриптор, выходящий за лимит таблицы, возникает исключение *#GP*.

Таблица дескрипторов прерываний, используемая в защищенном режиме, может содержать описания до 256 прерываний. Таблица может содержать

только вентили задач, прерываний и ловушек. Базовый адрес и лимит таблицы загружается привилегированной командой *LIDT* (аналогично *LGDT).* Размер *IDT* должен быть не менее 256 байт, для того, чтобы в нее поместились все зарезервированные прерывания процессора. Ссылка на элементы *IDT* происходит по командам *INT,* аппаратным прерываниям и исключениям процессора. При возникновении прерывания или исключения, дескриптор которого выходит за лимит таблицы, вырабатывается исключение #DF.

Дескрипторы имеют 8-байтный формат как для 16-разрядных (80286), так и для 32-разрядных процессоров. Назначение дескриптора определяется полями *байта управления доступом* (Acce*SS* Rights Byte) – байта со смещением 5. Дескрипторы 16-и 32-разрядных процессоров отличаются разрядностью поля базового адреса (24 и 32 бит) и трактовкой поля лимита, которое должно обеспечивать размер сегмента до 64 Кбайт или 4 Гбайт соответственно. Два старших байта у дескрипторов 80286 всегда нулевые (из-за требований совместимости с последующими процессорами, объявленными при выпуске 80286), что позволяет их отличать и корректно использовать, выполняя 16-битные приложения защищенного режима на 32-разрядных процессорах.

Два старших байта дескрипторов 32-разрядных процессоров содержат расширения полей *BASE* и *LIMIT,* бит дробности *G* (Granularity), определяющий, в каких единицах задан лимит: *G*=0 – в байтах, *G*=l – в страницах по 4 Кбайт (что и обеспечивает максимальную длину в 4 Гбайт).

В дескрипторах сегментов, отсутствующих в физической памяти *(P=0),* процессор интересуется только байтом управления доступом. Остальные байты могут использоваться по усмотрению ОС.

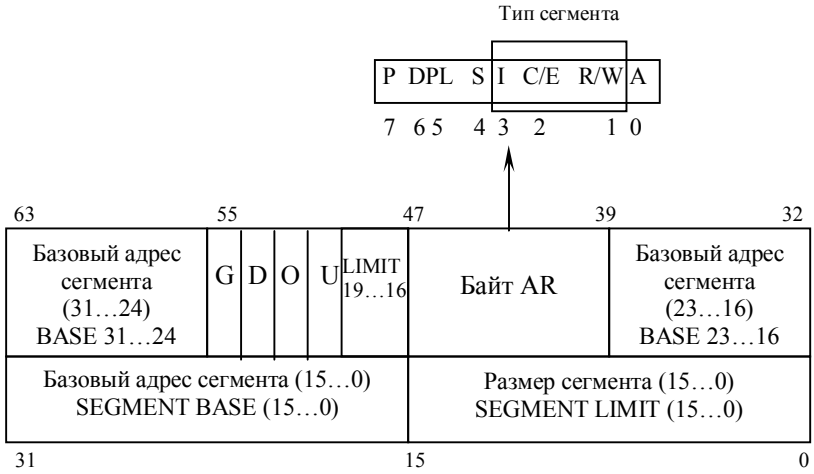

Рис.4. Структура дескриптора сегмента защищенного режима микропроцессора

Дескрипторы сегментов кода и данных определяют базовый адрес, размер сегмента, права доступа (чтение, чтение/запись, только исполнение кода или исполнение/чтение), а для систем с виртуальной памятью еще и присутствие сегмента в физической памяти (рис.4).

В байте управления доступом поля имеют следующее назначение:

• Бит 7 – P(Present) присутствие в памяти. При P=1 сегмент отображен в физической памяти, при *P=0* отображения нет и поля базового адреса и лимита не используются.

• Биты 6, 5 – *DPL* (Descriptor Privilege Level) – атрибуты привилегий сегмента.

• Бит 0 – A (Acce*S*Sed) – обращение. *A=0 – к* сегменту не было обращения, A=1 – селектор данного сегмента загружался в регистр сегмента или для него выполнялась команда тестирования.

Для дескриптора сегмента данных (включая и стек):

• Бит 2 – E (Expand Down) – контролируемое направление расширения: E=0 – расширяемый вверх (смещение не должно превышать значения лимита), E=1 – расширяемый вниз (стек, у которого смещение должно превышать значение лимита).

• Бит 1 – W (Writeable) – разрешение (W=1) или (W=0) записи данных в сегмент.

Для сегментов данных (включая и стек), расширяемых вниз (E=1), бит D в предпоследнем байте дескриптора определяет верхнюю границу сегмента: при D = 0 – FFFFh (максимальный размер сегмента – 64 Кбайт), при D=1 – FFFFFFFFh (максимальный размер сегмента – 4 Гбайт). Этот же бит в дескрипторе *сегмента стека* определяет и разрядность используемого указателя стека: D=0 – используется 16-битный SP, разрядность данных при операциях *PUSH, POP* – 16 бит; при D=1 используется 32-битный *ESP,* разрядность данных при операциях *PUSH, POP* – 32 бит.

В *сегмент кода* запись невозможна, лимит указывает на его последний байт, а биты типа имеют следующее назначение:

• Бит 2 – C (Conforming, конформность или подчиненность): при C=1 код может исполняться, если текущий уровень привилегий (*CPL*) не ниже уровня привилегий дескриптора *(DPL);* при C=0 (неподчиненный сегмент) управление к данному сегменту может передаваться только, если *CPL=DPL.*

• Бит 1 – *R* (Readable) – разрешение (R=l) или запрет (R=0) чтения сегмента.

Запись в сегмент кода возможна только через псевдоним (Alias) – сегмент данных с разрешенной записью, имеющий те же значения базы и лимита.

Бит D (Default Operation Size) в предпоследнем байте дескриптора сегмента кода определяет разрядность адресов и операндов по умолчанию: D=0 – 16 бит, D=1 – 32 бит.

Системные сегменты предназначены для хранения локальных таблиц дескрипторов *LDT* (Local Descriptor Table) и состояния задач *TSS* (Task State

Segment). Их дескрипторы определяют базовый адрес, лимит сегмента (1-64 Кбайт), права доступа (чтение, чтение/запись, только исполнение кода или исполнение/чтение) и присутствие сегмента в физической памяти (рис. 4).

В байте управления доступом у этих дескрипторов бит *P* определяет действительность (P=1) или недействительность (P=0) содержимого сегмента. Поле уровня привилегий *DPL* используется только в дескрипторах сегментов состояния задач. Поскольку обращение к локальным дескрипторам возможно только по привилегированным командам, поле DPL для дескрипторов таблиц не используется. Поле *Тип сегмента* определяет тип сегмента:

000	Сегмент данных только для чтения
001	Сегмент данных с разрешением чтения и записи
010	Не определена
011	Сегмент стека с разрешением чтения и записи
100	Сегмент кода с разрешением только выполнения
101	Сегмент кода с разрешением выполнения и чтения из него
110	Подчиненный сегмент кода с разрешением выполнения
111	Подчиненный сегмент кода с разрешением выполнения и чтения из него

Межсегментная передача управления непосредственно (командами *JMP, CALL, INT, RET* и IRET) возможна только к сегментам кода с тем же уровнем привилегий либо к подчиненным сегментам, уровень привилегий которых выше, чем *CPL* (при этом *CPL* не изменяется). Для переходов с изменением уровня привилегий используются *вентили* (Gate), иногда называемые шлюзами. Для каждого способа косвенной межсегментной передачи управления имеются соответствующие вентили, их использование позволяет процессору автоматически выполнять контроль защиты. *Вентили вызова* (Call Gates) используются для вызовов процедур со сменой уровня привилегий, *вентили задач* (Task Gates) используются для переключения задач, *вентили прерываний* (Interrupt Gates) и *ловушек* (Trap Gates) определяют процедуры обслуживания прерываний. Вентили вызова позволяют автоматически копировать заданное число слов из старого стека в новый. Вентили прерываний отличаются от вентилей ловушек только тем, что они запрещают прерывания (сбрасывают *IF),* а вентили ловушек – нет. Для каждого типа вентилей используются соответствующие *дескрипторы вентилей* (Gate Descriptors). Формат дескрипторов вентилей приведен на рис.5.

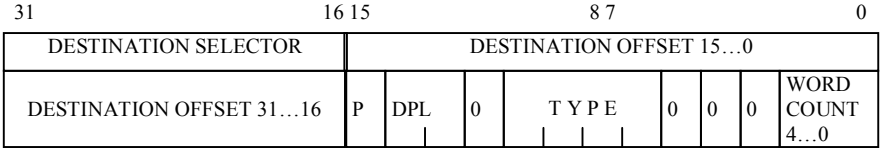

Рис. 5. Дескрипторы вентилей

В байте управления доступом у этих дескрипторов бит P определяет действительность (P=1) или недействительность (P=0) содержимого сегмента. Поле *DPL* задает уровень привилегий. Поле Type определяет тип вентиля:

- 4 – вентиль вызова 80286 (Call Gate);
- 5 – вентиль задачи 80286 (Task Gate);
- 6 – вентиль прерывания 80286 (Interrupt Gate);
- 7 – вентиль ловушки 80286 (Trap Gate);
- C – вентиль вызова 386+ (Call Gate);
- D – вентиль задачи 386+(Task Gate);
- E – вентиль прерывания 386+ (Interrupt Gate);
- F – вентиль ловушки 386+ (Trap Gate).

Поле *Word Count* используется только в вентилях вызовов и определяет число слов из стека вызывающего процесса, автоматически копируемых в стек вызываемой процедуры. Для сегментов 80286 слова 16-битные, для 386+ – 32-битные.

Слово *Destination Selector* для вентилей вызова, прерываний и ловушек задает селектор целевого сегмента кода, а для вентиля задачи – селектор целевого *TSS*.

Слово *Destination Offset* задает смещение (адрес) точки входа в целевом сегменте.

При использовании вентилей может возникнуть исключение *#GP*, которое означает, что селектор указывает на некорректный тип дескриптора. При попытке использования недействительного вентиля (P=0) возникает исключение *#NP*.

1.3. Привилегии

В защищенном режиме процессор имеет четырехуровневую систему привилегий, которая управляет использованием привилегированных инструкций и доступом к дескрипторам (и связанным с ними сегментам). Уровни привилегий нумеруются от 0 до 3, высшие привилегии соответствуют нулевому уровню. Уровни привилегий обеспечивают защиту задач, изолируемых друг от друга локальными таблицами дескрипторов. Сервисы операционной системы, обработчики прерываний и другое системное обеспечение могут включаться в виртуальное адресное пространство каждой задачи и защищаться системой привилегий. Каждая часть системы работает на своем уровне привилегий. Задачи, дескрипторы и селекторы имеют свои атрибуты привилегий.

Привилегии задач (Task Privilege) оказывают влияние на выполнение инструкций и использование дескрипторов. Текущий уровень привилегии задачи *CPL* (Current Privilege Level) определяется двумя младшими битами регистра *CS.CPL* задачи может изменяться только при передаче управления к новому сегменту через дескриптор вентиля. Задача начинает выполняться с уровня *CPL*, указанного селектором кодового сегмента внутри *TSS*, когда задача инициру-

ется посредством операции переключения задач. Задача, выполняемая на нулевом уровне привилегий, имеет доступ ко всем сегментам, описанным в *GDT*, и является самой привилегированной. Задача, выполняемая на уровне 3, имеет самые ограниченные права доступа. Текущий уровень привилегии может изменяться только при передаче управления через вентили.

Привилегии дескриптора (Descriptor Privilege) задаются полем *DPL* байта управления доступом. *DPL* определяет наибольший номер уровня привилегий (фактически, наименьшие привилегии), с которым возможен доступ к данному дескриптору. Самый защищенный дескриптор имеет *DPL*=0, к нему имеют доступ только задачи с *CPL=0.* Самый беззащитный дескриптор имеет *DPL=3,* его могут использовать задачи с *CPL=0,* 1, 2, 3. Это правило применимо ко всем дескрипторам, за исключением дескриптора *LDT.*

Привилегии селектора (Selector Privilege) задаются полем *RPL* (Requested Privilege Level) двумя младшими битами селектора. С помощью *RPL* можно урезать *эффективный уровень привилегий EPL* (Effective Privilege Level), который определяется как максимальное из значений *CPL* и *RPL.* Селектор с *RPL=0* не вводит дополнительных ограничений.

Контроль доступа к сегментам данных производится при исполнении команд, загружающих селекторы в *SS*, *DS*, *ES, FS* и *GS.* Команды загрузки *DS*, *ES, FS* и *GS* должны ссылаться на дескрипторы сегментов данных или сегментов кодов, допускающих чтение. Для получения доступа эффективный уровень привилегий *EPL* должен быть равным или меньшим (арифметически) уровня привилегий *DPL* дескриптора. Исключением из этого правила является читаемый подчиненный сегмент кода, который может быть прочитан задачей с любым *CPL.* Если эффективный уровень привилегий не разрешает доступ или ссылка производится на некорректный тип дескриптора (на дескриптор вентиля или на дескриптор только исполняемого кодового сегмента), вырабатывается исключение *#GP*. При ссылке на несуществующий дескриптор вырабатывается исключение *#NP*.

Команды загрузки *SS* должны ссылаться на дескриптор сегмента данных, допускающий запись. При этом *DPL* и *RPL* должны быть равны *CPL.* Нарушение этого условия и ссылка на дескриптор другого типа порождают исключение *#GP*, при ссылке на несуществующий дескриптор вырабатывается исключение *#SS.*

Контроль типов и привилегий при передаче управления производится при загрузке селектора в регистр *CS.* Тип дескриптора, на который ссылается данный селектор, должен соответствовать выполняемой инструкции. Нарушение типа (например, ссылка инструкции *JMP* на вентиль вызова) порождает исключение *#GP.* При передаче управления действуют следующие правила привилегий, нарушение которых также приводит к исключению *#GP:*

• команды *JMP* или *CALL* могут ссылаться либо на подчиненный сегмент кода с *DPL,* большим или равным *CPL,* либо на неподчиненный сегмент с *DPL* равным *CPL;*

• прерывания внутри задачи или вызовы, которые могут изменить уровень привилегий, могут передавать управление кодовому сегменту с уровнем привилегий, равным или большим уровня привилегий *CPL,* только через вентили с тем же или меньшим уровнем привилегий, чем *CPL;*

• инструкции возврата, которые не переключают задачи, могут передать управление только кодовому сегменту с таким же или меньшим уровнем привилегий;

• переключение задач может выполняться с помощью вызова, перехода или прерывания, которые ссылаются на вентиль задачи или сегмент состояния задачи (*TSS*) с тем же или меньшим уровнем привилегий.

Смена уровня привилегий, происходящая при передаче управления, автоматически вызывает переопределение стека. Начальное значение указателя стека *SS:SP* для уровня привилегий 0, 1, 2 содержится в *TSS.* При передаче управления по командам *JMP* или *CALL* в *CS:SP* загружается новое значение указателя стека, а старые значения помещаются в новый стек. При возврате на прежний уровень привилегий его стек восстанавливается (как часть инструкции *RET* или *IRET*). Для вызовов подпрограмм с передачей параметров через стек и сменой уровня привилегий из предыдущего стека в новый копируется фиксированное число слов, заданное в вентиле. Команда межсегментного возврата *RET c* выравниванием указателя стека при возврате корректно восстановит значение предыдущего указателя.

Привилегии и битовая карта разрешения ввода/вывода контролируют возможность выполнения операций ввода/вывода и управления флагом прерываний *IF*. Уровень привилегий ввода/вывода определяется полем *IOPL* (Input/Output Privelege Level) регистра флагов. Значение *IOPL* можно изменить только при *CPL=0.*

При $CPL \leq IOPL$ на операции ввода/вывода и управление флагом *IF* никаких ограничений не накладывается. При $CPL > IOPL$ попытка ввода/вывода, выполненная задачей с *TSS* класса 80286, вызывает исключение *#GP* (отказ). Если $CPL > IOPL$, а с задачей связан *TSS* 386+, инструкции ввода/вывода могут выполняться только по адресам портов, для которых установлены нулевые биты в карте разрешения ввода/вывода, имеющейся в *TSS*. Попытки обращения к портам, которым соответствуют единичные биты карты или которые не попали в карту (ее размер может усекаться), вызывают исключение *#GP*.

При $CPL > IOPL$ попытка выполнения инструкций *CLI* и *STI* вызывает исключение *#GP*. Неявное управление флагом прерываний инструкциями загрузки или восстановления регистра флагов блокируется без генерации исключений.

1.4. Защита

Для надежной работы многозадачных систем необходима защита задач друг от друга. Защита предназначена для предотвращения несанкционированного доступа к памяти и выполнения критических инструкций – команды *HLT*, которая останавливает процессор, команд ввода/вывода, управления флагом разрешения прерываний и команд, влияющих на сегменты кода и данных. Механизмы защиты вводят следующие ограничения:

• ограничение *использования* сегментов (например, запрет записи в сегменты данных, предназначенные только для чтения или попытки исполнения данных как кода). Для использования доступны только сегменты, дескрипторы которых описаны в *GDT* и *LDT*;

• ограничение *доступа* к сегментам через правила привилегий;

• ограничение набора инструкций – выделение *привилегированных инструкций* или операций, которые можно выполнять только при определенных уровнях *CPL* и *IOPL*.

• ограничение возможности межсегментных вызовов и передачи управления.

В защищенном режиме при выполнении инструкций процессор выполняет *проверку условий,* порождающих исключения:

Проверка при загрузке сегментных регистров

• Превышение лимита таблицы дескрипторов – *#GP*.

• Несуществующий дескриптор сегмента – *#NP* или *#SS*.

• Нарушение привилегий – *#GP*.

• Загрузка неверного дескриптора или типа сегмента – *#GP*:

• загрузка в *SS* сегмента кода или сегмента данных только для чтения,

• загрузка управляющих дескрипторов в *DS*, *ES* или *SS*,

• загрузка только исполняемых сегментов в *DS*, *ES* или *SS*,

• загрузка сегмента данных в *CS*.

Проверка ссылок операндов

• Запись в сегмент кода или сегмент данных только для чтения – *#GP*.

• Чтение из только исполняемого сегмента кодов – *#GP*.

• Превышение лимита сегмента – *#SS* или *#GP*.

Проверка привилегий инструкций

• *CPL*≠0 при выполнении инструкций *LIDT, LLDT, LGDT, LTR, LMSW, CTS, HLT, INVD, INVLPG, WBINVD,* операции с регистрами *DRn, TRn, CRn – #GP*.

• *CPL > IOPL* при выполнении инструкций *STI, CLI,* а для 80286 еще и инструкции *LOCK – #GP*.

• *CPL > IOPL* при выполнении инструкций *IN, INS, OUT, OUTS* с портами, не разрешенными битовой картой ввода/вывода – *#GP*.

При выполнении команд *IRET* и POPF с недостаточным уровнем привилегий биты *IF* и *IOPL* в регистре флагов не изменяются, исключения не порождаются:

• *IF* не меняется, при *CPL* > *IOPL*;

• *IOPL* не меняется, если *CPL* > 0.

Проверки при передаче управления по инструкциям *JMP, CALL, RET, INT* и *IRET* включают как проверку ссылок по лимиту (в "ближних" формах *JMP, CALL* и *RET* выполняются только эти проверки), так и проверку правил привилегий при межсегментных передачах через вентили.

Для того, чтобы задачи не "нарывались" на срабатывание защиты, в систему команд введены специальные *инструкции тестирования указателей.* Они позволяют быстро удостовериться в возможности использования селектора или сегмента без риска порождения исключения:

• *ARPL* – выравнивание *RPL.* При ее исполнении *RPL* селектора приравнивается максимальному значению из текущего *RPL* селектора и поля *RPL* в указанном регистре. Если при этом *RPL* изменился, устанавливается *ZF*=1;

• *VERR* – проверка возможности чтения: если сегмент, на который указывает селектор, допускает чтение, устанавливается *ZF*=1;

• *VERW* – проверка возможности записи: если сегмент, на который указывает селектор, допускает запись, устанавливается *ZF*=1;

• *LSL* – чтение лимита сегмента в регистр, если позволяют привилегии. При успехе устанавливается *ZF=1;*

• *LAR* – чтение байта доступа дескриптора в регистр, если позволяют привилегии. При успехе устанавливается *ZF*=1.

Некоторые функции защиты выполняются и механизмом страничной переадресации, однако, в отличие от "непробиваемой" сегментной защиты, существуют способы обхода страничной защиты на уровне пользователя (*CPL*=3).

1.5. Переключение задач

Для многозадачных и многопользовательских операционных систем важна способность процессора к быстрому переключению выполняемых задач. Операция переключения задач процессора (Task Switch Operation) сохраняет состояние процессора и связь с предыдущей задачей, загружает состояние новой задачи и начинает ее выполнение. Переключение задач выполняется по инструкции межсегментного перехода *(JMP)* или вызова *(CALL),* ссылающейся на *сегмент состояния задачи TSS* (Task State Segment) или *дескриптор вентиля задачи* в *GDT* или *LDT.* Переключение задач может происходить также по аппаратным и программным прерываниям и исключениям, если соответствующий элемент в *IDT* является дескриптором вентиля задачи. *Дескриптор TSS* указывает на *сегмент,* содержащий полное состояние процессора, а *дескриптор вентиля задачи* содержит *селектор,* указывающий на дескриптор *TSS.*

Каждая задача должна иметь связанный с ней *TSS*. 32-разрядные процессоры допускают и 16-битный (в стиле 80286) формат *TSS*. Оба типа сегментов содержат образы регистров процессора, раздельные указатели стеков для уровней привилегий 0, 1 и 2, а также обратную ссылку на селектор *TSS* вызвавшей задачи. Свободное поле *TSS* может использоваться по усмотрению ОС. *TSS* для процессоров 386+ содержит элементы, отсутствующие в 80286: битовые карты разрешения ввода/вывода и перенаправления прерываний, а также бит отладочной ловушки *T* (при *T*=1 переключение в данную задачу вызывает исключение отладки). Последним элементом *TSS*-386+ должен быть байт 0FFh. Карту перенаправления прерываний поддерживают только процессоры с расширением *VME*. Значение поля лимита дескриптора для *TSS*-286 должно превышать 002Bh, а для *TSS*-386+ – 0064h.

Карта разрешения ввода/вывода (I/O Permi*SS*ion Bit Map), расположенная в конце *TSS*-386+, имеет по одному биту на каждый адрес портов ввода/вывода. Разрешению обращения соответствует нулевое значение бита. Максимальный размер таблицы (2000h), соответствующий всем 64K адресов, может быть урезан лимитом *TSS*, но байт-терминатор 0FFh должен обязательно вписываться в лимит *TSS*. Порты с адресами, не попавшими в усеченную таблицу, считаются недоступными.

Текущий *TSS* идентифицируется специальным *регистром задачи TR* (Task Register). Этот регистр содержит селектор, ссылающийся на дескриптор текущего *TSS*. Программно-невидимые регистры базового адреса и лимита, связанные с *TR,* загружаются при загрузке в *TR* нового селектора.

Для возврата управления задаче, вызвавшей текущую задачу или ей прерванной, используется инструкция *IRET*. В регистре флагов имеется флаг вложенной задачи *NT* (Nested Task), который управляет функцией инструкции *IRET*. При *NT=0 IRET* работает обычным образом, оставаясь в текущей задаче. При *NT=1* (текущая задача – вложенная) *IRET* выполняет переключение в предыдущую задачу.

Когда инструкции *CALL, JMP* или *INT* выполняют переключение задач, старый (кроме случая *JMP)* и новый *TSS* помечаются занятыми (меняется значение *TYPE* в их дескрипторах), и в поле обратной ссылки в новом *TSS* устанавливается значение селектора старого *TSS*. Инструкции *CALL* и *INT,* переключающие задачи, устанавливают в новой задаче бит *NT*. Прерывание, не вызывающее переключения задач, сбросит бит *NT*. Этот бит может устанавливаться и сбрасываться инструкциями *POPF* и *IRET*.

Смена контекста сопроцессора при переключении задач автоматически не производится, поскольку новой задаче сопроцессор может и не понадобиться. Однако процессор обнаруживает первое использование сопроцес-

сора после переключения задач и вырабатывает исключение *#NM*. Обработчик этого исключения сам определит, необходима ли смена контекста. Всякий раз при переключении задач процессор устанавливает бит *TS* (Task Switched) в *MSW*. Это указывает на то, что контекст процессора может относиться к другой задаче. При выполнении инструкций *ESC* или *WAIT,* если *TS*=1 и *MP=1* (сопроцессор присутствует), вырабатывается исключение *#NM* (отсутствующий сопроцессор).

1.6. Страничное управление памятью

Страничное управление (Paging) является средством организации виртуальной памяти с подкачкой страниц по запросу (Demand-Paged Virtual Memory). В отличие от сегментации, которая организует программы и данные в модули различного размера, страничная организация оперирует с памятью, как с набором страниц одинакового размера. В момент обращения страница может присутствовать в физической оперативной памяти, а может быть выгруженной на внешнюю (дисковую) память. При обращении к выгруженной странице памяти процессор вырабатывает исключение *#PF – отказ страницы,* а программный обработчик исключения (часть ОС) получит необходимую информацию для *свопинга* – "подкачки" отсутствующей страницы с диска. Страницы не имеют прямой связи с логической структурой данных или программ. В то время как селекторы можно рассматривать как логические имена модулей кодов и данных, страницы представляют части этих модулей. Учитывая обычное свойство локальности (близкого расположения требуемых ячеек памяти) кода и ссылок на данные, в оперативной памяти в каждый момент времени следует хранить только небольшие области сегментов, необходимые активным задачам. Эту возможность (а следовательно, и увеличение допустимого числа одновременно выполняемых задач при ограниченном объеме оперативной памяти) как раз и обеспечивает страничное управление памятью. В первых 32-разрядных процессорах (начиная с 80386) размер страницы составлял 4 Кбайт. Начиная с Pentium, появилась возможность увеличения размера страницы до 4 Мбайт, одновременно с использованием страниц размером 4 Кбайт. В P6 имеется возможность расширения физического адреса до 36 бит (64 Гбайт), при котором могут использоваться страницы размером 4 Кбайт и 2 Мбайт.

Базовый механизм страничного управления использует двухуровневую табличную трансляцию линейного адреса в физический (рис. 6).

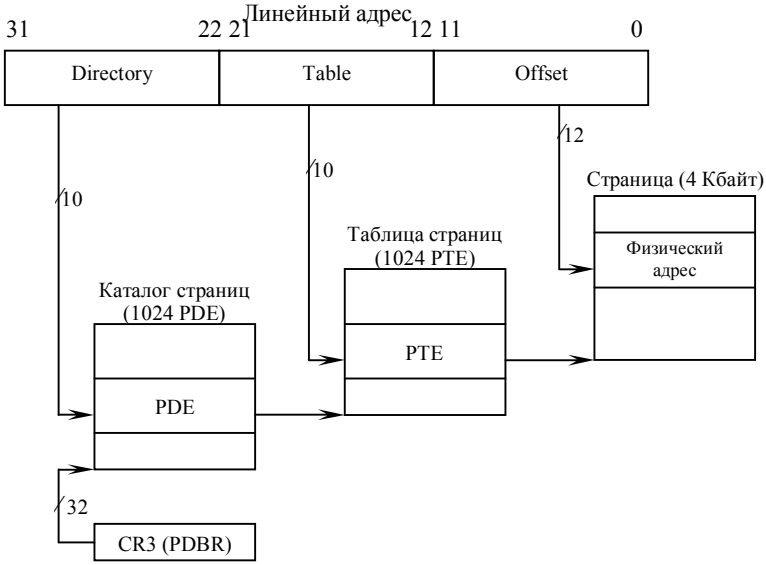

Рис. 6. Базовый механизм страничной переадресации

Механизм имеет три части: *каталог страниц* (Page Directory), *таблицы страниц* (Page Table) и *собственно страницы* (Page Frame). Механизм включается установкой бита *PG*=1 в регистре *CR0*. Регистр *CR2* хранит *линейный адрес отказа* (Page Fault Linear Addre*SS*) – адрес памяти, по которому был обнаружен последний отказ страницы. Регистр *CR3* хранит *физический адрес каталога страниц* (Page Directory Physical Base Addre*SS*). Его младшие 12 бит всегда нулевые (каталог выравнивается по границе страницы).

Каталог страниц размером 4 Кбайт содержит 1024 32-битных строки *PDE* (Page Directory Entry). Каждая строка (рис. 7*)* содержит 20 старших бит адреса таблицы следующего уровня (младшие биты этого адреса всегда нулевые) и признаки (атрибуты) этой таблицы. Индексом поиска в каталоге страниц являются 10 старших бит линейного адреса (A22-A31).

Рис.7. Структура 32-битных элементов страничного преобразования: строка каталога

Каждая *таблица страниц* также имеет 1024 строки *PTE* (Page Table Entry) аналогичного формата (рис. 8), но эти строки содержат базовый физический адрес (Page Frame Addre*SS*) и атрибуты самих страниц. Индексом поиска в таблице являются биты A12-A21 линейного адреса. *Физический адрес* получается из адреса страницы, взятого из таблицы, и младших 12 бит линейного адреса. Строки каталога и таблиц имеют следующие биты атрибутов:

• *P* (Present) – бит присутствия. *P*=1 означает возможность использования данной строки для трансляции адреса. Бит присутствия вхождений в таблицы, используемые текущим исполняемым кодом, должен быть установлен. Программный код не должен его изменять "на ходу". Если *P*=0, то все остальные биты доступны операционной системе и могут использоваться для получения информации о местонахождении данной страницы.

• *A* (Acce*SS*ed) – признак доступа, который устанавливается перед любым чтением или записью по адресу, в преобразовании которого участвует данная строка.

• *D* (Dirty) – признак, который устанавливается перед операцией записи по адресу, в преобразовании которого участвует данная строка. Таким образом, помечается использованная – "грязная" страница, которую в случае замещения необходимо выгрузить на диск. Признак D всегда равен нулю для элементов каталога страниц.

Рис.8. Структура 32-битных элементов
страничного преобразования (строка таблицы)

Биты *P, A, D* модифицируются процессором аппаратно в заблокированных шинных циклах. При их программной модификации в многопроцессорных системах должен использоваться префикс *LOCK,* гарантирующий сохранение целостности данных.

Поле *OS Reserved* программно используется по усмотрению ОС и может хранить, например, информацию о "возрасте" страницы, необходимую для реализации замещения по алгоритму *LRU* (Least Recently Used – наиболее долго не использовавшаяся страница замещается первой).

Бит *PWT* (Page Write Through) определяет политику записи при кэшировании, а бит *PCD* (Page Cache Disable) запрещает кэширование памяти для обслуживаемых страниц или таблиц (используются на процессорах 486+).

Бит *PS* (Page Size) задает размер страницы (только в *PDE).* При *PS=0* страница имеет размер 4 Кбайт, *PS*=1 используется в расширениях *PAE* и *PSE*.

Механизм защиты страниц различает два уровня привилегий: *пользователь* (User) и *супервизор* (Supervisor). Пользователю соответствует уровень привилегий 3, супервизору – уровни 0, 1 и 2. Строки таблиц имеют атрибуты защиты страниц – биты *U* (User) и *W* (Writable – возможна запись); в некоторых описаниях те же биты называются *U/S* (User/Supervisor) и R/W (Read/Write). Эти атрибуты в строке каталога страниц относятся ко всем страницам, на которые ссылается данная строка через таблицу второго уровня. Атрибуты защиты в строке таблицы страниц относятся к конкретной странице памяти, которую она обслуживает. Права доступа к странице приведены в табл.1. Если атрибуты защиты в *PDE* и *PTE* различаются, то результирующие атрибуты определяются согласно табл.2. Защита на уровне страниц включается установкой бита *WP* (Write Protect) в регистре *CR0,* по аппаратному сбросу он обнуляется.

Таблица 1

Защита на уровне страниц

U (U/S)	W(R/W)	Разрешено при PL=3	Разрешено при PL=0, 1, 2
0	0	Нет	Чтение/запись
0	1	Нет	Чтение/запись
1	0	Только чтение	Чтение/запись
1	1	Чтение/запись	Чтение/запись

Бит G (Global), появившийся в P6, определяет глобальность страницы. Он анализируется только в строке, указывающей на страницу физической памяти (в *PTE* для страниц в 4 Кбайт, в PDE – для страниц 2 Мбайт или 4 Мбайт). Этот бит, управляемый только программно, позволяет пометить страницы глобального использования (например, ядра ОС). При установленном бите *PGE* в регистре *CR4* строки с указателями на глобальные таблицы не будут аннулироваться в *TLB* при загрузке *CR3* или переключении задач, что снижает издержки обслуживания виртуальной памяти.

Таблица 2

Комбинация атрибутов защиты

PDE		PTE		Результат	
U	W	U	W	U	W
1	0	1	0	1	0
1	0	1	1	1	0
1	1	1	0	1	0
1	1	1	1	1	1
1	0	0	0	0	1
1	0	0	1	0	1
1	1	0	0	0	1
1	1	0	1	0	1
0	0	1	0	0	1
0	0	1	1	0	1
0	1	1	0	0	1
0	1	1	1	0	1
0	0	0	0	0	1
0	0	0	1	0	1
0	1	0	0	0	1
0	1	0	1	0	1

Механизм страничного управления при обращении к памяти может порождать исключение *#PF*. Оно возникает при обращении к отсутствующей странице и при нарушении прав доступа, определяемых уровнем привилегий и битами *U* и W. Для идентификации причины отказа в стек помещается 16-битный код ошибки, формат которого приведен на рис. 9. Хотя названия бит совпадают с атрибутами строк, их назначение отличается. Бит *U/S* указывает на уровень привилегий, при котором произошел отказ (1 пользователь, 0 – супервизор). Бит *W/R* указывает на операцию, при которой произошел отказ (0 – чтение, 1 – запись). Бит *P* указывает на причину отказа (*P*=1 – отсутствие страницы в памяти, *P*=0 – нарушение защиты). Биты *U* не используются. Проверка защиты на уровне страниц выполняется после проверок защиты сегментов. Если при попытке доступа к памяти сработала защита сегментов, то проверка на уровне страниц уже не выполняется.

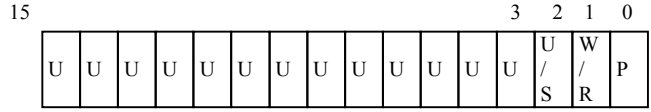

Рис. 9. Формат кода ошибки при отказе страницы

Обращение при каждой операции доступа к памяти к двум таблицам, расположенным в памяти, существенно снижает производительность. Для предотвращения этого замедления в процессор введен буфер ассоциативной трансляции *TLB* (Translation Look aside Buffer) для хранения интенсивно используемых строк таблиц. В процессорах 80386 и 486 буфер представляет собой четырехканальный наборно-ассоциативный кэш на 32 строки таблиц трансляции. Такой размер позволяет хранить информацию о трансляции 128 Кбайт памяти, что в большинстве случаев мультизадачного использования дает коэффициент кэш-попаданий 98 %, то есть только 2 % обращений к памяти требуют дополнительных обращений к таблицам. В процессоре Pentium имеются раздельные *TLB* для инструкций и данных, а в P6 буферы разделены еще и по размеру обслуживаемых страниц (4 Кбайт и 2 Мбайт/4 Мбайт).

Когда страничное управление разрешено (бит *PG*=1 в *CR0*), блок страничной переадресации получает 32-битный линейный адрес от блока сегментации. Его старшие 20 бит сравниваются со значениями из *TLB*, и, в случае попадания, физический адрес вычисляется по начальному адресу страницы, полученному из *TLB*, а затем выводится на шину адреса. Если соответствующей строки в *TLB* нет, производится чтение строки из страничного каталога. Если строка имеет бит *P*=1 (таблица присутствует в памяти), в ней устанавливается бит доступа *A* и производится чтение указанной ею строки из таблицы второго уровня. Если и в этой строке *P*=1, процессор обновляет в ней биты *A* и *D*, вычисляет физический адрес и, наконец, производит обращение по этому адресу. Если на этих этапах встречается *P*=0, вырабатывается исключение *#PF,* обработчик которого должен принять меры по загрузке затребованной страницы в оперативную память. Поскольку это исключение классифицируется как отказ, после его обработки (успешной) повторяется доступ к затребованной ячейке памяти. Во время его обработки может еще раз возникнуть исключение *#PF*, но это не приведет к двойному отказу.

Обработчик исключения *#PF*, поддерживающий подкачку страниц по запросу, должен скопировать страницу с внешней (дисковой) памяти в оперативную, загрузить адрес страницы в строку таблицы и установить бит присутствия *P*. Поскольку в *TLB* могла оставаться старая некорректная копия строки, необходимо объявить содержимое *TLB* недействительным (произвести очистку). После этого процесс, породивший исключение, может быть продолжен.

Буферы *TLB* для прикладных задач (*CPL*>0) программно невидимы, с ними работает только ОС с *CPL*=0. Операционная система должна корректно сгенерировать начальные таблицы трансляции и обрабатывать исключения отказов. В случае изменения таблиц (и при изменении значений бита *P* в любых таблицах) она должна очищать буферы *TLB* (целиком или конкретные вхождения). Очистка всех буферов (кроме глобальных вхождений при установленном бите *PGE)* происходит при загрузке регистра *CR3*, выполняемой явно или по переключении задач. При изменении отображения одиночной страницы очистка может выполняться по инструкции *INVLPG,* которая, по

возможности, очистит только конкретное вхождение в *TLB*, но в ряде случаев может обновить и весь буфер.

Процессоры Pentium и старше кроме стандартных страниц 4 Кбайт могут оперировать и *страницами размером 4 Мбайт,* что позволяет уменьшить накладные расходы на обслуживание страничного режима при возросших потребностях программ в памяти. *Расширение размера страницы* (Page Size Extension) разрешается установкой бита *PSE* в регистре *CR4.* При *CR4.PSE=0* страничное преобразование работает по базовой схеме (рис. 7). При *CR4.PSE*=l процессор анализирует бит 7, определенный теперь как *PS* (Page Size – размер страницы) строки каталога страниц (POE). Если *PDE.PS=0,* эта строка ссылается на таблицу 4-килобайтных страниц, и обработка идет по схеме рис. 10. Если *PDE.PS*=1, то биты 31:12 этой строки являются базовым физическим адресом страницы размером 4 Мбайт – здесь ступень таблицы страниц исключена. Формат строки каталога (*PDE*) для страницы с расширенным размером приведен на рис. 11.

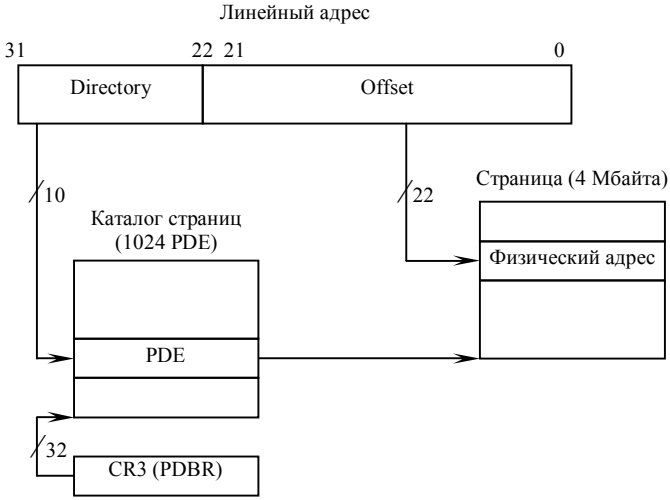

Рис. 10. Страничная переадресация в режиме PSE

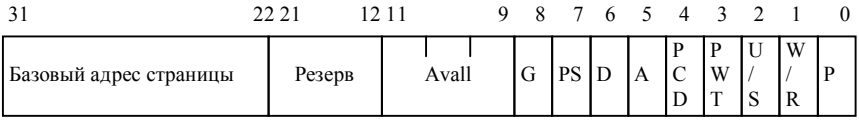

Рис. 11. Строка каталога для страницы размером 4 Мбайта

Процессоры P6 также поддерживают *расширение физического адреса* (Physical Addre*SS* Extensions) до 64 Гбайт. Это расширение включается установкой бита *PAE* в регистре *CR4,* при этом расширение *PSE* становится недоступным (бит *PS*E игнорируется). У процессоров 6-го поколения разрядность внешней шины адреса составляет 36 бит. Расширенная разрядность адреса возможна лишь в режиме *PAE*, при *PAE*=0 биты внешней шины адреса A[35:32] принудительно обнуляются. Поскольку архитектура предполагает разрядность линейного адреса только 32 бит, старшие 4 бита могут появиться только в результате работы блока страничной переадресации. Здесь блок страничной переадресации оперирует уже 64-битными элементами, 32-битный регистр *CR3* хранит указатель (Page Directory Base Pointer) на маленькую табличку 64-битных указателей, находящуюся в первых 4 Гбайт памяти. Два старших бита [31:30] линейного адреса выбирают из этой таблицы указатель на одну из 4 таблиц каталогов. Следующие 9 бит [29:21] линейного адреса выбирают элемент из этой таблицы, который, в зависимости от бита *PS,* может быть как ссылкой на таблицу страниц (*PS*=0), так и базовым адресом страницы памяти (*PS*=1). При *PS*=0 биты [20:12] линейного адреса выбирают страницу размером 4 Кбайт из таблицы, а биты [11:0] являются смещением в этой странице. При *PS*= 1 биты [20:0] линейного адреса являются смещением внутри страницы размером 2 Мбайт. Схемы страничного преобразования для режима *PAE* приведены на рис.12, а структура элементов – на рис. 13.

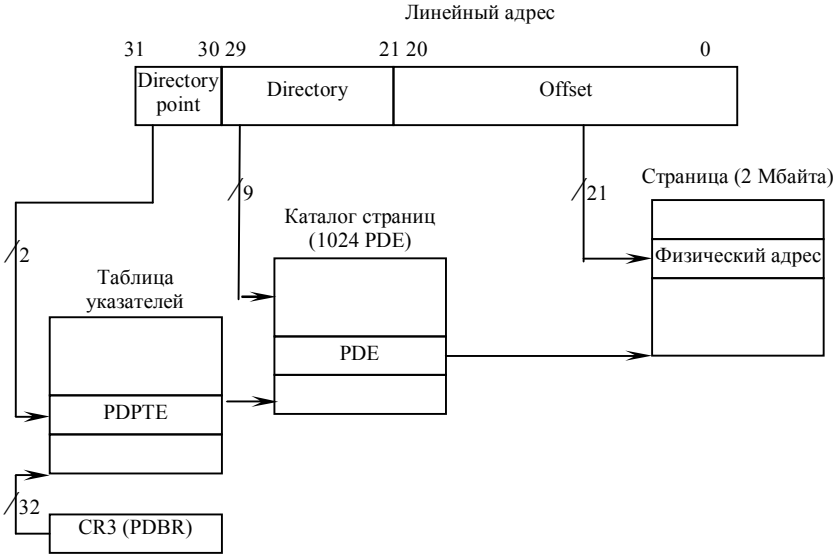

Рис.12. Страничная переадресация в режиме PAE
для страниц размером 2 Мбайта

Наличие у процессора расширений размеров страниц и физического адреса можно обнаружить с помощью инструкций *CPUID*. Если процессор эту инструкцию не поддерживает, то у него нет и этих средств.

PDPTE

Строка таблицы указателей на каталоги

PDE4K

Строка каталога для страницы размером 4 Кбайт

PTE4K

Строка таблицы для страниц размером 4 Кбайт

PDE2M

Строка каталога для страницы размером 4 Кбайт

Рис. 13. Структура 64-битных элементов страничного преобразования

1.7. Виртуализация прерываний

Согласно базовой архитектуре 32-разрядных процессоров, в защищенном режиме инструкции *CLI* и *STI*, управляющие флагом прерывания *IF,* являются *чувствительными* к уровню привилегий. Если уровень привилегий задачи позволяет *(CPL ≤ IOPL),* они будут воздействовать на флаг прерываний, как и инструкции, воздействующие на этот флаг неявно. Попытка их использования при *CPL>IOPL* вызовет исключение-отказ *#GP*. Инструкции, управляющие флагом прерывания неявно, исключения не вызывают, но и не изменяют состояния флага.

Прикладные программы должны иметь возможность защитить себя от воздействия внешних прерываний. Для защиты критичных участков кода обычно применяют инструкции *CLI* и *STI,* которые могут встречаться в прикладных задачах довольно часто. Отработка исключений каждый раз при их появлении значительно снижает производительность вычислительного процесса. В то же время позволять прикладным программам напрямую управлять флагом прерывания процессора далеко не всегда допустимо, поскольку это может приводить к потере управления операционной системой. В многозадачной системе с разделяемыми устройствами внешние прерывания первоначально обрабатываются операционной системой, которая определяет, к какой задаче (задачам) относится каждое конкретное прерывание. ОС должна иметь возможность сообщить соответствующей задаче (или всем) о его возникновении, но только в тот момент, когда задача имеет состояние с разрешенными прерываниями.

Задача, которая должна обрабатывать аппаратные прерывания, может получать их как в реальном виде (как внешние прерывания), так и в виртуальном. Когда приложению передаются *реальные прерывания,* для него станет проблематичным использование виртуальной памяти с подкачкой страниц по запросу. Если страница с обработчиком окажется выгруженной, то по возникновении аппаратного прерывания потребуется подкачка страницы, занимающая значительное время. Для обработчиков прерываний такое время реакции окажется неприемлемым.

Чтобы избежать подкачки страниц по аппаратным прерываниям, обработчики прерываний должны располагаться в ядре ОС, постоянно присутствующем в памяти. Если задача-приемник прерывания находится в прерываемом состоянии, ОС сигнализирует ей о возникновении прерывания сразу. Если задача находится в непрерываемом состоянии, ОС сигнализирует ей о прерывании только после перехода в прерываемое состояние. Таким образом, задаче передаются *виртуальные прерывания*. В этом случае возможно использование виртуальной памяти, поскольку обработчики, выполняющие критические по времени действия, всегда находятся в памяти.

Виртуализация прерываний на основе вышеупомянутых базовых свойств процессора выполняется ОС чисто программно, однако обработка потока частых исключений весьма неэффективна.

В процессорах Pentium и последних моделях 486 появились новые аппаратные средства виртуализации прерываний. В регистре *EFLAGS* появились флаги виртуального разрешения прерывания *VIF* и ожидающего прерывания *VIP*. В регистре *CR4* бит *PVI* (Protected-Mode Virtual Interrupt) разрешает виртуализацию флага прерываний в защищенном режиме для задач с *CPL*=3. При *VIP=0,* а также при *VIP=1* и *CPL*<3 инструкции *CLI* и *STI* будут вести себя вышеописанным способом – либо управлять флагом *IF,* либо вызывать исключение *#GP*. При *VIP=1* и *CPL=3* их поведение меняется:

• Если *IOPL=3,* то *CLI* и *STI* переключают флаг *IF*.

• Если *IOPL*<3 и *VIP=0* (нет ожидающего прерывания), то *CLI/STI* просто управляют флагом *VIF*.

• Если *IOPL*<3 и *VIP=1* (присутствует ожидающее прерывание), то попытка разрешения прерывания (установки *VIF),* используя *STI*, вызовет исключение *#GP*. Его обработчик и донесет задаче весть об имевшем место внешнем прерывании.

Итак, флаг виртуального разрешения прерывания *VIF* указывает на прерываемое или не прерываемое состояние задачи, выполняемой на *CPL*=3; этот флаг не затрагивает всего процессора. Флаг ожидающего прерывания *VIP* устанавливается в образе регистров задачи операционной системой, если внешнее прерывание пришло в момент, когда задача находилась в непрерываемом состоянии. Мониторинг перехода задачи в состояние с разрешенными прерываниями осуществляется процессором автоматически без лишних затрат времени. Флаги, участвующие в виртуализации прерываний:

• *VI P* – модифицируется инструкцией *IRETD,* если *CPL*=0.

• *VIF* – модифицируется инструкцией *IRETD,* если *CPL=0;* инструкциями *CLI* и *STI* как было показано выше.

• *IOPL* – модифицируется инструкцией *IRET(D)* и *POPF(D),* если *CPL*=0.

• *IF* – модифицируется инструкцией *IRET(D)* и *POPF(D),* если *CPL*≤*IOPL;* инструкциями *CLI* и *STI* как было показано выше.

Наличие у процессора средств виртуализации прерываний можно обнаружить с помощью инструкции *CPUID.* Если процессор эту инструкцию не поддерживает, то у него нет и этих средств.

1.8. Режим виртуального 8086 (V86 и EV86)

Прикладные программы для 8086 могут исполняться на 32-разрядных процессорах как в реальном режиме, так и в режиме виртуального 8086 (V86), который является особым состоянием задачи защищенного режима. Назначение этого режима – формирование *виртуальной машины,* эмулирующей процессор 8086. Виртуальная машина формируется программными

средствами операционной системы – монитором V86, который поддерживается специальными аппаратными средствами процессора. Режим V86 позволяет пользоваться аппаратными средствами поддержки многозадачности. В этом режиме работает защита и механизм страничной переадресации, позволяющий адресоваться к любой области 4-гигабайтного пространства физической памяти. Выполнение приложений 8086 в среде V86 возможно параллельно с приложениями защищенного режима. Страничная переадресация позволяет параллельно выполняться нескольким задачам V86 с возможностью совместного использования общих областей кода операционной системы и разделения реальных аппаратных ресурсов компьютера. Режим V86 появился с процессором 80386, в процессорах Pentium и последних моделях 486 появилось его расширение – EV86 (Enhanced Virtual 8086). Цель этого расширения заключалась в переносе ряда функций формирования виртуальной машины с программного обеспечения на аппаратные средства процессора, что существенно повышает ее производительность.

Монитор V86 представляет собой модуль 32-битного программного кода, исполняющийся с *CPL=0.* Он содержит обработчики прерываний и исключений, средства инициализации задач V86 и эмуляции операций ввода/вывода. Монитор тесно связан с обработчиком исключения *#GP*, через который в основном и происходит общение с приложениями 8086.

Приложение 8086 "привыкло" работать в среде своей ОС (например, MS-D*OS*) и пользоваться ее сервисами. В V86 ОС реального режима может работать на той же виртуальной машине, что и приложение, а может и эмулироваться средствами ОС защищенного режима. Первый способ проще – здесь в V86 может быть запущена традиционная ОС (например, MS-D*OS*). Второй способ требует затрат на разработку эмуляции, но только в этом случае можно получить все преимущества реальной многозадачности.

В режиме V86 программе доступны все регистры 8086, а с помощью префикса изменения разрядности операндов – и их 32-битные расширения. В качестве сегментных регистров через префиксы замены сегментов возможно использование и регистров *GS* и *FS*, которых в реальном 8086 нет. По умолчанию адресация 16-битная, но с помощью префикса изменения разрядности адреса возможно использование 32-битной адресации. Набор команд включает все команды 8086 и последующие расширения, реализованные в 80286, 80386, 486 и Pentium. Попытка выполнения инструкций, ориентированных на использование операционной системой и допустимых только для защищенного режима – *LTR, STR, LLDT, SLDT, LAR, LSL, ARPL, VERR* и *VERW,* – вызовет исключение *#UD*.

Модель памяти в режиме V86 имеет некоторые особенности. Одним из основных различий реального и защищенного режимов является трактовка содержимого сегментных регистров. В режиме V86, как и в реальном, для получения линейного адреса содержимое сегментных регистров сдвигается на четыре разряда влево и суммируется с эффективным адресом. Один мега-

байт (точнее, область 0-l0FFEFh) адресуемого таким образом пространства с помощью страничной трансляции может отображаться в любую область 4 Гбайт физической памяти. С помощью страничной трансляции можно добиться “сворачивания в кольцо” памяти размером 1 Мбайт, свойственного 8086 (отобразив адреса выше FFFFFh на ту же физическую память, что и начинающиеся с нуля). Поскольку таблица трансляции задается управляющим регистром *CR3*, в многозадачном окружении при переключении задач автоматически загрузится и требуемая таблица трансляции. Превышение исполнительным адресом границы 64 Кбайт (в том числе при 32-битной адресации) вызывает исключение *#SS* или *#GP*. Настоящий 8086 в таком случае переходит к нулевым адресам того же сегмента, как бы свернувшегося в кольцо.

Все программы, запущенные в режиме V86, выполняются со всеми *проверками защиты*. Они автоматически получают уровень привилегий 3, то есть минимальные привилегии (реальный режим подразумевает уровень привилегий 0). Попытка выполнения привилегированных инструкций вызывает исключение *#GP*. *К* этим инструкциям относятся *LIDT, LGDT, LMSW, CTS, HLT,* а также операции с регистрами *DRn, TRn, CRn, MSR*.

В режиме V86 понятие чувствительности к уровню привилегий при вводе/выводе (*IOPL*-sensitive) имеет особую трактовку. Инструкции ввода/вывода *IN, OUT, (REP) INS, (REP) OUTS* в режиме V86 *не чувствительны* к *IOPL,* а управление доступом к портам осуществляется только через битовую карту ввода/вывода в сегменте состояния задачи. Попытка обращения к запрещенным портам вызовет исключение *#GP*. Битовая карта может быть усеченной (это достигается сочетанием лимита *TSS* и смещения карты разрешения), тогда “отрезанные” адреса будут соответствовать запрещенным портам.

Особую проблему в режиме V86 составляет *обработка прерываний* – как программных, так и аппаратных. Здесь *чувствительными* являются инструкции *INTn, PUSHF, POPF, STI, CLI* и *IRET,* которые могут воздействовать на флаг прерываний *IF*. Однако инструкции *INT3, INTO* и *BOUND* не чувствительны к *IOPL*.

Если установить *IOPL=3,* то задача, исполняемая в V86, будет выполняться с максимальной производительностью, имея возможность непосредственно управлять флагом прерываний *IF*. Однако для многозадачных систем защищенного режима такие “вольности” не допустимы, поскольку ОС может быть легко лишена возможности управления системой. Чтобы этого не происходило, ОС должна сделать виртуальным флаг *IF* для задач V86, для чего устанавливают *IOPL<3*. Это приводит к тому, что все чувствительные инструкции задачи V86, приводящие к изменениям флага *IF,* будут вызывать исключения-отказы, по которым будет производиться переключение в монитор виртуальных машин (часть многозадачной ОС защищенного режима). Однако частая обработка этих исключений значительно снизит производительность виртуальной машины V86.

Инструкции *INT n,* которые широко ис*пользуются, например, в сервисах* D*OS* и BI*OS*, приводят к выходу из режима V86. Все прерывания и *исключения* влекут за собой смену уровня привилегий на уровень операционной системы защищенного режима. Если *IOPL*<3, то все прерывания через исключения-отказы приводят к выходу в монитор виртуальной машины. При *IOPL*<3 прерывания вызывают исполнения соответствующих процедур защищенного режима, которые задаются ОС (приложения 8086 "не понимают" таких обработчиков). Обработчик, заданный ОС, может распознать, что прерывание пришло из V86, по образу регистра *EFLAGS* в стеке. Далее ОС может либо обработать это прерывание самостоятельно, эмулируя выполнение требуемых функций, либо переслать его к ОС реального режима, работающей в V86 (reflecting interrupt – отражение прерывания). Частые переключения режимов (задач), которые происходят при выполнении прерываний, снижают производительность.

Приложение 8086, которое должно обрабатывать аппаратные прерывания, может получать их как в реальном виде (как внешние прерывания), так и в виртуальном. В режиме V86 виртуализация прерываний выполняется программно монитором ОС.

В конечном итоге ОС защищенного режима может прозрачно для приложения 8086, работающего в режиме V86, эмулировать окружение обычной машины 8086, включая прерывания и перехват обращения к портам. Однако при *IOPL<3,* обеспечивающем устойчивость системы с полной виртуализацией, производительность виртуальной машины будет низкой.

Проблему виртуализации прерываний позволяет разрешить *расширенный режим* – EV86. При *IOPL=3* приложение 8086 по-прежнему имеет возможность управления флагом *IF* (инструкциями *CLI, STI).* Изменения касаются работы при *IOPL<3.* Теперь чувствительные инструкции не вызывают безусловного исключения-отказа, а воздействуют на виртуальную версию флага прерываний *VIF* в регистре *EFLAGS.* Этот флаг не влияет на восприятие процессором внешних (маскируемых) прерываний, а лишь указывает на состояние задачи EV86 – разрешила или запретила она обработку прерываний. При этом, во-первых, повышается производительность – инструкции *CLU* и *STI* теперь не приводят к отказам, а во-вторых, упрощается монитор, обеспечивающий программную виртуализацию флага прерываний (в V86 монитор должен был отслеживать все инструкции, влияющие на *IF – CLI, STI, PUSHF, POPF, INT* и *IRET*). Аппаратная виртуализация флага приводит к значительному повышению производительности.

Режим EV86 включается установкой бита V*ME* в регистре *CR4.* В этом режиме сегмент состояния задачи *TSS*-386 приобретает новую 32-байтную структуру – *карту перенаправления прерываний* (interrupt redirection bitmap). По структуре она напоминает карту разрешения портов ввода/вывода и располагается в *TSS* прямо перед ней – слово *IO_BITMAP_OFFSET* является указателем на ее конец. Каждый бит карты перенаправления соответствует од-

ному из 256 (32x8) программных прерываний, вызываемых инструкцией *INT n*. На программные прерывания, вызываемые иным образом, исключения и аппаратные прерывания карта перенаправления действия не оказывает. Если бит установлен, то соответствующее прерывание вызовет исключение-отказ с выходом из EV86 в монитор. Если бит сброшен, прерывание обрабатывается процедурой реального режима без выхода из EV86.

Для работы с *IOPL<3* предназначены новые флаги – *VIF* и *VIP*. Эти флаги может анализировать и модифицировать только монитор, работающий на уровне *CPL=0.* Теперь инструкции задачи EV86, связанные с флагом *IF,* не приводят к выходу в монитор по исключению-отказу, а воздействуют только на флаг *VIF,* не затрагивая реального флага управления прерываниями *IF*. Однако этот факт от приложения EV86 скрывается – везде вместо *IF* подставляется *VIF*. Флаг *VIF* является указателем для монитора на состояние задачи EV86 – запрещены или разрешены прерывания. Если монитор должен сообщить задаче EV86 о внешнем прерывании, то в случае, если прерывания задачей разрешены, он это может сделать сразу. Если прерывания запрещены *(VIF=0),* монитор установит флаг ожидающего прерывания *VIP.* Теперь, как только задача EV86 попытается разрешить прерывания, исполнив инструкцию *STI* или иным способом, до фактической установки флага *VIF* выработается исключение-отказ *#GP,* по которому монитор получит управление и вызовет процедуру обслуживания ожидающего прерывания. Флаги *VIF* и *VIP* позволяют существенно упростить виртуализацию прерываний монитором и повысить производительность.

Новые возможности виртуализации прерываний работают при *IOPL<3,* однако инструкция *PUSHF* в режиме EV86 симулирует *IOPL=3,* так что задача EV86 не сможет определить свой реальный *IOPL* по образу регистра *EFLAGS* в стеке. Поведение процессора в режиме EV86 при *IOPL*=3 иначе как удивительным не назовешь, но для такого “беззащитного” варианта этот режим, пожалуй, и не нужен.

Вход в режим V86 – установка бита *VM* в регистре *EFLAGS* – возможен одним из двух способов;

• выполнение инструкции IRET в 32-битном режиме, когда образ *EFLAGS* сохранен в стеке с установленным битом *VM* (при *CPL=0,* иначе бит *VM* не установится);

• переключение на задачу-386, у которой в *TSS* образ *EFLAGS* имеет установленный бит *VM.*

При использовании режима *EV86* необходимо также установить бит *VME* в регистре *CR4.*

Выход из режима V86 (EV86) возможен только при обработке прерывания. Если вызываемая процедура имеет *CPL*=0, то бит *VM* будет сброшен, и она будет выполняться в защищенном режиме. Если ее *CPL*>0, произойдет исключение *#GP* – нарушение защиты. Если прерывание вызывает переключение задач, состояние регистров с установленным флагом *VM* сохранится в

TSS старой задачи, к которой можно будет вернуться. Новый режим (защищенный или V86) установится в соответствии с *TSS* новой задачи.

Значение бита *VM* не может быть изменено никакими другими способами; кроме того, его значение не может быть прочитано – при любом программном сохранении регистра флагов значение *VM* всегда показывается нулевым. Так что приложение, выполняемое в среде V86, никак не может ни переключить режим процессора, ни распознать, в каком режиме – реальном или виртуальном – оно исполняется. Это, конечно же, справедливо при корректно построенном мониторе виртуальной машины V86, являющимся частью ОС защищенного режима. Процессор для этого предоставляет все необходимые аппаратные средства, позволяющие выполнить полную эмуляцию 8086.

1.9. Переключение "реальный-защищенный режим"

Переключение процессора в защищенный режим из реального осуществляется загрузкой в *CR0* слова с единичным значением бита *PE* (Protect Enable). Для совместимости с ПО для 80286 бит *PE* может быть установлен также инструкцией *LMSW*. До переключения в памяти должны быть проинициализированы необходимые таблицы дескрипторов *IDT* и *GDT*. Сразу после включения защищенного режима процессор имеет *CPL*=0. Для всех 32-разрядных процессоров рекомендуется выполнять следующую последовательность действий для переключения в защищенный режим.

1. Запретить маскируемые прерывания сбросом флага *IF,* а возникновение немаскируемых прерываний блокировать внешней логикой. Программный код на время "переходного периода" должен гарантировать отсутствие исключений и не использовать программных прерываний. Это требование вызвано сменой механизма вызова обработчиков прерываний.

2. Загрузить в *GDTR* базовый адрес *GDT* (инструкцией *LGDT).*

3. Инструкцией *MOV CR0* установить флаг *PE,* а если требуется страничное управление памятью, то и флаг *PG*.

4. Сразу после этого должна выполняться команда межсегментного перехода *(JMP Far)* или вызова *(CALL Far)* для очистки очереди инструкций, декодированных в реальном режиме, и выполнения сериализации процессора. Если включается страничное преобразование, то коды инструкций *MOV CR0* и *JMP* или *CALL* должны находиться в странице, для которой физический адрес совпадает с логическим (для кода, которому передается управление, этого требования не предъявляется).

5. Если планируется использование локальной таблицы дескрипторов, инструкцией *LLDT* загрузить селектор сегмента для *LDT* в регистр *LDTR*.

6. Инструкцией *LTR* загрузить в регистр задач селектор *TSS* для начальной задачи защищенного режима.

7. Перезагрузить сегментные регистры (кроме *CS*), содержимое которых еще относится к реальному режиму, или выполнить переход или вызов другой

задачи (при этом перезагрузка регистров произойдет автоматически). В неиспользуемые сегментные регистры загружается нулевое значение селектора.

8. Инструкцией *LIDT* загрузить в регистр *IDTR* адрес и лимит *IDT* – таблицы дескрипторов прерываний защищенного режима.

9. Разрешить маскируемые и немаскируемые аппаратные прерывания.

Переключение процессора из защищенного режима в реальный возможно не только через аппаратный сброс, как это было у 80286, но и с помощью сброса бита *PE* в *CR0*. При этом для корректного перехода, согласно документации на процессоры, должны выполняться следующие действия:

1. Запретить маскируемые прерывания флагом *IF,* немаскируемые – внешней схемой.

2. Если включена страничная трансляция, то необходимо обеспечить равенство линейных и физических адресов для текущего исполняемого кода (перейти на такую страницу), а также для таблиц *GDT* и *IDT*. Обнулить бит *PG* в регистре *CR0* и загрузить нули в *CR3* для очистки кэш-буфера *TLB*.

3. Передать управление читаемому сегменту с лимитом 64 Кбайт.

4. Загрузить в сегментные регистры *SS*, *DS*, *ES, FS* и *GS* селектор дескриптора (ненулевой), в котором установлен лимит 64 Кбайт, байтовая дробность (*G*=0), расширяемость вверх (*E*=0), доступность записи (*W*=1) и присутствие *(P=1)*. Если сегментные регистры не перезагружать, исполнение будет продолжаться с атрибутами, унаследованными от защищенного режима.

5. Инициализировать таблицу векторов прерываний реального режима (в пределах первого мегабайта) и указать на нее инструкцией *LIDT*.

6. Сбросить бит *PE* для перехода в реальный режим.

7. Выполнить дальний переход на программу реального режима, что сбросит очередь инструкций, декодированных в защищенном режиме, и загрузит соответствующие права доступа к сегменту кода.

8. Загрузить корректное значение в сегментные регистры и указатель стека.

9. Разрешить прерывания.

После этого загружаются остальные регистры. Процессор теперь работает в реальном режиме, по умолчанию с 16-разрядными адресами и данными.

Шаги 3 и 4 предназначены для загрузки программно-недоступных регистров дескрипторов сегментов параметрами стандартного реального режима. Однако вместо них можно создать и "нереальный" режим, отличающийся от реального возможностью доступа к сегментам большого (до 4 Гбайт) размера. Правда, у процессоров 80286 и 80386 лимит кодового сегмента принудительно ограничивается размером 64 Кбайт, но у старших процессоров большой размер допустим для всех сегментов. "Нереальный режим" часто используется менеджерами памяти для DOS и игровыми программами, требующими большого объема памяти.

1.10. Примеры программ защищенного режима

Программа перевода микропроцессора в защищенный режим

Действия, необходимые для обеспечения функционирования этой программы:

1. Подготовка в оперативной памяти таблицы глобальных дескрипторов *GDT*.
2. Инициализация необходимых дескрипторов в таблице *GDT*.
3. Загрузка в регистр gdtr адреса и размера таблицы *GDT*.
4. Запрет обработки аппаратных прерываний.
5. Переключение микропроцессора в защищенный режим.
6. Организация работы в защищенном режиме:
   - настроить сегментные регистры;
   - выполнить собственно содержательную работу программы; в нашем случае мы просто обозначим сам факт успешного перехода в защищенный режим;
   - подготовиться к возврату в реальный режим;
   - запретить аппаратные прерывания;
7. Переключение микропроцессора в реальный режим.
8. Настройка сегментных регистров для работы в реальном режиме.
9. Разрешение прерываний.
10. Стандартное для MS-DOS завершение работы программы.

Подготовка таблиц глобальных дескрипторов GDT. Для того, чтобы собрать информацию о всех программных объектах, находящихся в данный момент в памяти и в системе в целом, их дескрипторы собираются в таблицы, которые представляют собой массивы 8-байтовых элементов.

Микропроцессор аппаратно поддерживает три типа дескрипторных таблиц:

1. Таблица *GDT* (Global Descriptor Table) – *глобальная дескрипторная таблица.* Это основная общесистемная таблица, к которой допускается обращение со стороны программ, обладающих достаточными привилегиями. Расположение таблицы *GDT* в памяти произвольно; оно локализуется с помощью специального регистра *gdtr*. В таблице *GDT* могут содержаться следующие типы дескрипторов:

- дескрипторы сегментов кодов программ;
- дескрипторы сегментов данных программ;
- дескрипторы стековых сегментов программ;
- дескрипторы TSS (Task Segment Status) – специальные системные объекты, называемые сегментами состояния задач;
- дескрипторы для таблиц LDT;
- шлюзы вызова;
- шлюзы задач.

2. Таблица *LDT* (Local Descriptor Table) – *локальная дескрипторная таблица.* Для любой задачи в системе может быть создана своя дескрипторная таблица подобно общесистемной *GDT*. Тем самым адресное пространство задачи локализуется в пределах, установленных набором дескрипторов таблицы *LDT*. Для связи между таблицами *GDT* и *LDT* в таблице *GDT* создается дескриптор, описывающий область памяти, в которой находится *LDT*. Расположение таблицы *LDT* в памяти также произвольно и локализуется с помощью специального регистра ldtr. В таблице *LDT* могут содержаться следующие типы дескрипторов:

- дескрипторы сегментов кодов программ;
- дескрипторы сегментов данных программ;
- дескрипторы стековых сегментов программ;
- шлюзы вызова;
- шлюзы задач.

3. Таблица *IDT* (Interrupt Descriptor Table) – *дескрипторная таблица прерываний.* Данная таблица также является общесистемной и содержит дескрипторы специального типа, которые определяют местоположение программ обработчиков всех видов прерываний. В качестве аналогии можно привести таблицу векторов прерываний реального режима. Расположение таблицы *IDT* в памяти произвольно и локализуется с помощью специального регистра idtr. Элементы данной таблицы называются *шлюзами.* Это элементы особого рода, и мы их рассмотрим на следующем уроке при изучении обработки прерываний в защищенном режиме. Сейчас только отметим, что эти шлюзы бывают трех типов:

- шлюзы задач;
- шлюзы прерываний;
- шлюзы ловушек.

Каждая из дескрипторных таблиц может содержать до 8192 (2^{13}) дескрипторов. Данное значение определяется размерностью поля в сегментном регистре. Роль дескрипторов в защищенном режиме изменяется по сравнению с реальным режимом, и это отражается даже на их названии – в защищенном режиме они называются селекторами сегментов.

Определим в программе только одну дескрипторную таблицу – глобальную дескрипторную таблицу *GDT*. Прерывания в программе не обрабатываются. Таблицу *LDT* есть смысл применять, когда в системе работают несколько задач и необходимо изолировать их друг от друга.

Дескрипторы, описывающие сегменты некоторой программы, могут содержаться как в глобальной (*GDT*), так и в локальной (*LDT*) дескрипторных таблицах. Сегментные регистры содержат селекторы, которые являются указателями на дескрипторы, описывающие соответствующие области памяти. Но как микропроцессор узнает о том, в какой из двух дескрипторных таблиц находится дескриптор, на который указывает селектор?

Структура сегментного регистра в защищенном режиме представляется тремя полями. Поле *RPL*, занимающее два младших бита 0 и 1 (Request Privilege Level – запрашиваемый уровень привилегий), используется в механизме ограничения доступа по привилегиям. А вот состояние однобитового поля *TI*, занимающего бит 2 сегментного регистра, как раз и определяет, с какой именно таблицей идет работа:

- если *TI* = 0, то сегментный регистр содержит селектор на дескриптор в глобальной дескрипторной таблице *GDT*;
- если *TI* = 1, то сегментный регистр содержит селектор на дескриптор в локальной дескрипторной таблице *LDT*.

А также сегментный регистр содержит поле селектора. Оно определяет число, кратное восьми (так как три младшие бита заняты под поля *RPL* и *TI*, являющееся указателем на дескриптор в одной из дескрипторных таблиц в соответствии со значением бита в поле *TI*. Управлять состоянием этого бита может либо сама программа, но тогда она должна обладать достаточным уровнем привилегий, либо операционная система, обеспечивающая ее работу. Для самой же программы ничего не меняется. По прежнему базовые адреса ее сегментов определяются с помощью сегментных регистров, хотя и по-иному, чем в реальном режиме, принципу. В каждый момент времени микропроцессор может работать только с одной дескрипторной таблицей: *GDT* или *LDT*. В нашем случае мы будем считать, что все сегменты программы находятся в глобальной дескрипторной таблице *GDT*, то есть *TI* = 0 во всех используемых сегментных регистрах.

Определимся теперь с набором дескрипторов в таблице *GDT*, которые понадобятся для нашей программы:

- дескриптор для описания сегмента самой таблицы *GDT*;
- дескриптор для описания сегмента данных программы;
- дескриптор для описания сегмента команд программы;
- дескриптор для описания сегмента стека программы;
- дескриптор для описания сегмента, в котором будет находиться процедура для выдачи сигнала сирены;
- дескриптор для описания видеопамяти.

Формат дескриптора сегмента показан на рис. 4. В программе его удобнее описать в виде структуры:

```
descr     struc
limit      dw     0
base_1    dw     0
base_2    db     0
atrdb             0
lim_atr   db     0
base_3    db     0
 ends
```

Используя этот шаблон структуры, опишем таблицу GDT как массив структур:

```
gdt_seg segment para
gdt_0           descr <0,0,0,0,0,0>
gdt_gdt_8       descr <0,0,0,0,0,0>
gdt_ldt_10      descr <0,0,0,0,0,0>
gdt_ds_18       descr <0,0,0,0,0,0>
gdt_es_vbf_20   descr <0,0,0,0,0,0>
gdt_ss_28       descr <0,0,0,0,0,0>
gdt_cs_30       descr <0,0,0,0,0,0>
gdt_size=$-gdt_0-1 ;определение размера таблицы GDT gdt_seg ends
```

Инициализация дескрипторов в таблице GDT. После того, как мы определились с набором дескрипторов и даже сформировали глобальную дескрипторную таблицу, нужно разобраться с тем, где брать информацию для их заполнения и как это делать. Давайте обсудим этот вопрос для каждого поля в отдельности:

- поле limit – размер сегмента. В это поле нужно поместить точный размер сегмента за вычетом единицы, так как адресация идет от нуля. Наиболее оптимальный способ заключается в использовании оператора $ – извлечь текущее значение счетчика адреса. Пример использования этого оператора для определения размера приведен выше при описании таблицы *GDT*;
- поля base_1, base_2, base_3 – поля 32-разрядного базового адреса. Этот адрес будет известен после загрузки программы в оперативную память, и поля придется загружать на этапе выполнения при подготовке к переходу в защищенный режим. Фрагмент программы заполнения поля базового адреса для дескриптора gdt_gdt_8 может выглядеть так:

```
xor eax, eax
mov ax,gdt_seg ; адрес сегмента в ax сдвигом на 4 разряда получим физический
               ; 20-разрядный адрес сегмента gdt_seg;
 shl eax, 4
 mov base_1,ax
 rol eax, 16 ; меняем для получения оставшейся части адреса
 mov base_2,al
```

Так как эту операцию придется проделывать несколько раз для каждого из сегментов программы, то для удобства работы и повышения наглядности оформим этот фрагмент в виде макроса load_addr:

```
load_addr macro descr,seg_addr,seg_size
 mov descr.limit,seg_size
 xor eax,eax
 mov ax,seg_adr
 shl eax, 4
```

```
mov descr.base_1,ax
rol eax,16
mov  descr.base_2,al
endm
```

Заодно добавляем к операциям, выполняемым макросом, и инициализацию поля предела;

• поле atr – байт атрибутов. Структура этого байта представлена в приложении А табл.2. Понятно, что для конкретного сегмента его значение будет всегда константой. Чтобы не запутывать себя окончательно, на мой взгляд, значение этого байта в целом удобнее формировать как логическую сумму нужных значений его полей для определения типа конкретного сегмента. Сформируем элементарные константы полей этого байта в соответствии с битами приложения А табл.2:

```
; биты 0, 4, 5, 6, 7 – постоянная часть байта AR для всех типов сегментов
const equ 10010000b ;бит 1 – доступность сегментов по чтению/записи
code_r_n equ 00000000b ; сегмент кода: чтение запрещено
code_r_y equ 00000010b ; сегмент кода: чтение разрешено
data_wm_n equ 00000000b ; сегмент данных: модификация запрещена
data_wm_y equ 00000010b ; сегмент данных: модификация разрешена,
                          ;бит 2 – тип сегмента
code_n equ 00000000b ; обычный сегмент кода
code_p equ 00000100b ; подчиненный сегмент кода
_data equ 00000000b    ; сегмент данных
_stackequ 00000100b    ; сегмент стека, бит 3 – предназначение
_code equ 00001000b    ; сегмент кода
data_stk equ 00000000b ; сегмент данных или стека
```

Теперь для получения значения атрибута для нужного сегмента достаточно подобрать нужные константы по битам и выполнить подсчет суммы, как, например, для дескриптора gdt_gdt_8:

```
atr=const or data_wm_y or _data or data_stk
art=
```

Значение atr для дескриптора сегмента gdt_gdt_8 будет равно 10010010 = = 92h. Для удобства использования можно создать макрос, который будет создавать константу с именем, состоящим из двух частей приставки atr и имени дескриптора, для которого формируется байт атрибута (к примеру, для дескриптора gdt_gdt_8 это будет atr_gdt_gdt_8):

```
atr macro descr,bit1, bit2, bit3 atr&descr=const or bit"! or bit2 or bit3 endm
```

Имена формируемых констант нужно указывать при инициализации структур для каждого дескриптора, к примеру для gdt_gdt_8:

```
gdt_seg segment para
gdt_0 descr <0,0,0,0,0,0>
```

```
 atr gdt_gdt_8,data_wni_y,_data,data_stk
 gdt_gdt_8 descr <0,0,0,atr_gdt_gdt_8,0,0>
;…
 gdt_seg ends
```

• поле lim_atr – байт, состоящий из четырех старших битов размера сегмента и четырех атрибутов (приложение А табл.1). В нашем случае размер сегмента небольшой, то есть четыре старших бита размера равны 0. И оставшиеся биты атрибутов для нашего случая также нулевые;

• поле base_3 содержит старший байт 4-байтового физического адреса сегмента. Так как мы начинаем работу в реальном режиме, где размер максимального физического адреса не превышает 20 бит, то этот байт также будет нулевым.

Таким образом, в нашей программе инициализации будут подлежать три поля:

limit, три первых байта адреса base_1 и base_2 и байт атрибута atr.

Загрузка регистра gdtr. Дескрипторные таблицы являются ключевыми для организации работы в защищенном режиме. Их местоположение в памяти может быть любым. Микропроцессор узнает о том, где находятся эти таблицы, по содержимому определенных системных регистров. Так, после того, как была сформирована таблица GDT, ее адрес нужно поместить в регистр gdtr. Но этого мало, так как в этот же регистр нужно поместить и размер этой таблицы. Для загрузки именно этого регистра в системе команд микропроцессора есть специальная команда:

Igdt *адрес_48-битного_поля* (Load GDT register) – загрузить регистр gdtr. Команда Igdt загружает системный регистр gdtr содержимым 6-байтового поля, адрес которого указан в качестве операнда.

Из описания команды следует, что вначале необходимо сформировать поле из шести байт со структурой, аналогичной формату регистра gdtr, а затем указать адрес этого поля в качестве операнда команды Igdt.

Для резервирования поля из шести байт (48 бит) TASM поддерживает специальные директивы резервирования и инициализации данных – dp и df. После выделения – с помощью одной из этих директив – области памяти в сегменте данных необходимо сформировать в этой области указатель на начало таблицы GDT и ее размер. Но удобнее использовать структуру. Пример ее использования показан в следующем фрагменте программы:

```
point struc
lim dw 0
adr dd 0
ends
data segment
point_gdt point <gdt_size,0>
;…
code segment
;…
```

```
xor eax, eax
mov ax,gdt_seg
shl eax,4
mov dword ptr point_gdt.adr,eax
lgdt pword point_gdt
```

Запрет обработки аппаратных прерываний. Как только микропроцессор переключится в защищенный режим, первое же прерывание от таймера, которое происходит 18,2 раза в секунду, "подвесит" компьютер. В данном примере прерывания как программные, так и внешние, мы запретим.

Переключение микропроцессора в защищенный режим. Теперь все готово для того, чтобы корректно перейти в защищенный режим. Специальных команд микропроцессора для выполнения такого перехода нет. О том, что микропроцессор находится в защищенном режиме, говорит лишь состояние бита *PE* в регистре *CR0*. Установить этот бит можно двумя способами:

- непосредственной установкой бита *PE* в регистре *CR0*. Состояние этого бита управляет режимами работы микропроцессора: если *PE* = 0, то микропроцессор работает в реальном режиме;
- если *PE* = 1, то микропроцессор работает в защищенном режиме;
- использованием функции 89h прерывания 15h BIOS. Опишем оба способа, но использовать будем первый.

Регистр *CR0* программно доступен, поэтому установить бит *PE* можно, используя обычные команды ассемблера:

```
mov eax, cr0
or eax, 0001h
mov cr0, eax
```

Последняя команда mov переводит микропроцессор в защищенный режим.

Функция 89h прерывания 15h выполняет это и некоторые другие действия неявно. Прежде чем вызывать это прерывание, необходимо посредством регистров сообщить ему следующее:

- ah = 89h;
- bl = новый номер для аппаратного прерывания уровня irq0. Уровни irql...7 будут иметь следующие по порядку номера;
- bh = новый номер для аппаратного прерывания уровня irq8. Уровни irq9...f будут иметь следующие по порядку номера;
- ds:si = адрес *GDT* для защищенного режима;
- cx – адрес первой выполняемой команды в защищенном режиме.

Эта функция предполагает, что дескрипторы в таблице *GDT* расположены в определенной последовательности:

- 0h – пустой дескриптор;
- 8h – дескриптор таблицы *GDT*;
- 10h – дескриптор таблицы *LDT*;
- 18h – дескриптор сегмента данных, на него указывает селектор в регистре ds;

• 20h – дескриптор дополнительного сегмента данных, на него указывает селектор в регистре es;

• 28h – дескриптор сегмента стека, на него указывает селектор в регистре ss;

• 30h – дескриптор сегмента кода, на него указывает селектор в регистре es;

• остальные дескрипторы.

В нашей таблице *GDT* этот порядок следования соблюден, поэтому фрагмент программы перевода микропроцессора в защищенный режим может быть следующим:

```
code segment
;…
  mov ah, 89h
  mov bl, 20h
  mov bh, 28h
  mov ax, gdt_seg
  mov ds, ax
  mov si, 0
 lea ex, protect
 int 15h
protect:
…;работа в защищенном режиме
```

Настройка сегментных регистров. После выполнения всех вышеописанных действий состояние микропроцессора можно сравнить с состоянием человека, которого после пребывания на ярком солнце завели в темную комнату. Что нужно сделать в такой ситуации для того, чтобы микропроцессор не уподобился слону в посудной лавке?

Содержимое сегментных регистров в реальном и защищенном режимах интерпретируется микропроцессором по-разному. Как только микропроцессор оказывается в защищенном режиме, первую же команду он пытается выполнить традиционно: по содержимому пары cs:ip определить ее адрес, выбрать ее и т. д. Но содержимое es должно быть индексом, указывающим на дескриптор сегмента кода в таблице *GDT*. Но пока это не так, так как в данный момент es все еще содержит физический адрес параграфа сегмента кода, как этого требуют правила формирования физического адреса в реальном режиме. То же самое происходит и с другими регистрами. Но если содержимое других сегментных регистров можно подкорректировать в программе, то в случае с регистром cs этого сделать нельзя, так как он в защищенном режиме программно недоступен. Нужно помочь микропроцессору сориентироваться в этой затруднительной ситуации. Действие команд перехода основано как раз на изменении содержимого регистров cs и ip. Команды ближнего перехода изменяют только содержимое eip\ip, а команды дальнего перехода – оба регистра cs и eip\ip. Воспользуемся этим обстоятельством, вдобавок су-

ществует и еще одно свойство команд передачи управления – они сбрасывают конвейер микропроцессора, подготавливая его тем самым к приему команд, которые сформированы уже по правилам защищенного режима. Это же обстоятельство заставляет нас впрямую моделировать команду межсегментного перехода, чтобы поместить правильное значение селектора в сегментный регистр cs. Это можно сделать так:

```
code segment
;…
  db 0eah              ;машинный код команды jmp
  dw offset protect    ;смещение метки перехода ;в сегменте команд
  dw 30h               ;селектор сегмента кода в таблице GDT
protect:               ;загрузить селекторы для остальных дескрипторов
   mov ax,18h
   mov ds,ax           ;сегментный регистр данных
   mov ax,28h
   mov ss,ax           ;сегментный регистр стека
   mov ax,20h
   mov es,ax;          ;дополнительный сегмент данных: для указания на
                       ;видеобуфер
```

Но за кадром остался один момент, на который нужно обязательно обратить внимание. В микропроцессоре каждому сегментному регистру соответствует свой теневой регистр дескриптора. Этот регистр имеет размер 64 бит и формат дескриптора сегмента. Смена содержимого теневых регистров производится автоматически всякий раз при смене содержимого соответствующего сегментного регистра. Последние наши действия по изменению содержимого сегментных регистров привели к тому, что мы неявно записали в теневые регистры микропроцессора содержимое соответствующих дескрипторов из *GDT*. Программисту теневые регистры недоступны, с ними работает только микропроцессор. Если есть необходимость изменить их содержимое, то для этого нужно сначала изменить сам дескриптор, а затем загрузить соответствующий ему селектор в нужный сегментный регистр.

Подготовка к возврату в реальный режим. Здесь возникает примерно та же проблема с сегментными регистрами, что была при входе в защищенный режим. Мы упоминали уже о теневых регистрах микропроцессора, но не сказали того, что микропроцессор использует их, даже работая в реальном режиме. При этом поля этих регистров заполнены, конечно, в соответствии с требованиями реального режима:

- предел должен быть равен 64 К = 0ffffh;
- бит G = 0 (значение размера в поле предела) – это значение в байтах;
- байт атрибута равен 10010010 = 92h;
- базовый адрес значения не имеет.

Следовательно, перед переходом в реальный режим нужно сформировать эти значения в соответствующих дескрипторах и сделать актуальными эти изменения в теневых регистрах, для чего нужно перезагрузить сегментные регистры. После этого можно смело переходить в реальный режим. В программе все эти действия могут выглядеть так:

```
code segment
    ;…
    ;мы в защищенном режиме и начинаем переход в реальный режим
    ;загрузим значение 0ffffh в поле предела
   mov gdt_ds_18.limit,0ffffh ;для регистров cs, ss, es, fs и gs аналогично
    ;селектор – в регистр данных, чтобы обновить теневой регистр для ds
   mov ax, 18
   mov ds, ax
     ;для регистров ss, es, fs и gs аналогично для регистра cs загрузку
     ;селектора проведем особым способом – командой jmp far
   db 0eah
   dw offset jump
   dw 30h
   jump: ;можно переходить в реальный режим
```

Переключение микропроцессора в реальный режим. Нужно всего лишь сбросить нулевой бит регистра *CR0*. Это можно сделать несколькими способами. К примеру:

```
mov eax, cr0
and al, 0feh
mov cr0, eax
```

После сброса бита *PE* микропроцессор снова оказался в реальном режиме.

Настройка сегментных регистров. После перехода в реальный режим опять возникает проблема с содержимым сегментных регистров. Решается она уже известным вам способом:

```
;... моделирование команды дальнего перехода для загрузки cs
db 0eah
dw real_mode
dw code
real_mode: mov ax, data
           mov ds, ax
           mov ax, stk
           mov ss, ax
```

Разрешение прерываний. Теперь можно разрешить прерывания. Так как в системе прерываний мы ничего не меняли, то действия наши тоже были простейшими: а) запрещение аппаратных прерываний для того, чтобы на

время смены режима работы микропроцессора нас не беспокоило прерывание от таймера, б) последующее разрешение прерываний после возврата в реальный режим. На практике, конечно, без прерываний не обойтись.

```
cli ;флаг IF в 0 – запретить прерывания от аппаратуры
```

и

```
sti ;флаг IF в 1 – разрешить прерывания от аппаратуры.
```

Стандартное для MS-DOS завершение работы программы. Если все было сделано правильно, то окончание работы программы выглядит стандартно.

Теперь нам осталось собрать все вместе в одну программу, ввести ее в машину и приступить к экспериментам. Нужно отметить один момент, с которым вы столкнетесь при трансляции исходного модуля. Он связан с вычислением константы code_size. Фрагмент листинга с этой ошибкой будет выглядеть так:

```
184 load_descr gdt_cs_30, CODE,code_size
1 185 0072 C706 0030r 0197 mov gdt_cs_30.limit, code_size
**Error** prg_16_1.asm(94) LOAD_DESCR(1) Forward reference needs override
```

Природа этой ошибки понятна – опережающая ссылка на объект, о котором компилятор еще не имеет информации. Объектом в данном случае является константа code_size. Она вычисляется в конце сегмента кода, а применяется – в середине. Чтобы устранить эту ошибку, нужно заставить компилятор выполнить два прохода программы. Тогда за первый проход он получит представление обо всех объектах программы, а на втором проходе сформирует правильные команды. Чтобы указать компилятору на необходимость выполнения двух проходов, используется опция командной строки /m2:

```
tasm /m2 prg16_1, asm…
```

ЛИСТИНГ

Программа перевода микропроцессора в защищенный режим

```
.586P       ;разрешение инструкций Pentium
.MODEL large
include mac.inc ;структура для описания дескрипторов сегментов
descr STRUC
            limit dw 0
            base_1 dw 0
            base_2 db 0
            attr db 0
            lim_atr db 0
            base_3 db 0
ENDS
;макрос инициализации дескрипторов
load_descr MACRO des,seg_addr, seg_size
            mov des.limit, seg_size
            xor eax, eax
            mov ax, seg_addr
            shl eax, 4
            mov des.base_1, ax
            rol eax,16
            mov des.base_2,al
ENDM
atr MACRO descr,bit 1,bit2,bit3
;можно использовать условные директивы
;для проверки наличия параметров
atr_&descr=constp or bit1 or bit2 or bit3
ENDM
;структура для описания псевдодескриптора gdtr
point STRUC
            lim         dw          0
            adr         dd          0
ENDS
;атрибуты для описания дескрипторов сегментов постоянная часть
;байта AR для всех сегментов – биты 0, 4, 5, 6, 7
constp equ 10010000b
;бит 1
code_r_ n equ    00010000b ;кодовый сегмент: чтение запрещено;
code_r_ y equ    00000010b ;кодовый сегмент: чтение разрешено;
data_wm_n equ 00000000b ;сегмент данных: модификация запрещена;
```

```
data_wm_y equ 00000010b ;сегмент данных: модификация разрешена;
;бит 2
code_n equ 00000000b ;обычный сегмент кода
code_p equ 00000100b ;подчиненный сегмент кода
data_ equ    00000000b ;для сегмента данных
stack equ    00000000b ;для сегмента стека
;бит 3
code_ equ 00001000b ;сегмент кода
data_stk equ 00000000b       ;сегмент данных или стека
stks egment stack "stack" use16
            db 256 dup (0)
stk  ends
;таблица глобальных дескрипторов
gdt_seg segment para public "data" use16
gdt_0 descr <0,0,0,0,0,0>                ;никогда не используется
            atr gdt_gdt_8,data_wm_y,data_,data_stk
;ниже описываем саму gdt
gdt_gdt_8 descr <0,0,0,atr_gdt_gdt_8,0,0>
;не используем
gdt_ldt_10 descr <0,0,0,0,0,0>
atr gdt_ds_18, data_wn_y, data_, data_stk
;дескриптор сегмента данных
gdt_ds_18 descr <0,0,0,atr_gdt_ds_18,0,0>
            atr gdt_vbf_20, data_wm_y, data_, data_stk
 gdt_es_vbf_20 descr <0,0,0,atr_gdt_vbf_20,0,0>        ;видеобуфер
            atr gdt_ss_28,data_wm_y,stack_,data_stk
            gdt_ss_28 descr <0,0,0,atr_gdt_ss_28,0,0>   ;сегмент стека
            atr gdt_cs_30,code_ r_y,code_n,code_
            gdt_cs_30 descr <0,0,0,atr_gdt_cs_30,0,0>   ;сегмент кода
            gdt_size=$-gdt_0-1                          ;размер GDT минус 1
GDT_SEG ENDS
;данные программы
data segment para public "data" use16          ;сегмент данных
point_gdt point <gdt_size,0>
hello db “Вывод сообщения в защищенном режиме”
hel_size=$-hello
tonelow dw 2651              ;нижняя граница звучания 450 Гц
cnt    db 0                  ;счётчик для выхода из программы
temp dw ?                    ;верхняя граница звучания
data_size=$-point_gdt-1
data ends
code segment byte public "code" use16
;сегмент кода с 16-разрядным режимом адресации
```

```
assume cs:code,ss:stk
main proc
        mov ax,stk
        mov ss,ax
; заполняем таблицу глобальных дескрипторов
assume ds:gdt_seg
        mov ax,gdt_seg
        mov dsax
        load_descr gdt_gdt_8,GDT_SEG,gdt_size
        load_descr gdt_ds_18,DATA,data_size
        load_descr gdt_es_vbf_20,0b800h,3999
        load_descr gdt_ss_28,STK,255
        load_descr  gdt_cs_30,CODE,0ffffh ;code_size
assume ds:data
        mov ax,data
        mov ds,ax
;загружаем gdtr
        xor eax,eax
        mov ax, gdt_seg
        shl eax, 4
        mov point_gdt.adr, eax
        Igdt point_gdt
;запрещаем прерывания
        cli
        mov al,80h
        out 70h,al
; переключаемся в защищенный режим
        mov eax, cr0
        or al, 1
        mov cr0, eax        ;настраиваем регистры
        db 0eah ;машинный код команды jmp
        dw offset protect ;смещение метки перехода в сегменте команд
        dw 30h ;селектор сегмента кода в таблице GDT
protect:
;загрузить селекторы для остальных дескрипторов
        mov ax,18h
        mov ds,ax
        mov ax,20h
        mov es,ax
        mov ax,28h
        mov ss,ax           ;работаем в защищенном режиме:
; выводим строку в видеобуфер
        mov cx,hel_size ;длина сообщения
        mov si,offset hello ;адрес строки сообщения
```

```
        mov     di,1640   ;начальный адрес видеобуфера для вывода
        mov     ah, 07h   ;атрибут выводимых символов
outstr:
        mov     al,[si]
        mov     es:[di],ax
        inc     si
        inc     di
        inc     di
        loop    outstr
;запускаем сирену
go:
;заносим слово состояния 110110110b (0B6h):
;выбираем второй канал порта 43п – динамик
        mov     ax,0B06h
        out     43h,ax    ;в порт 43h
        in      al,61h    ;получим значение порта 61h в a1
        or      a1,3      ;инициализируем динамик – подаем ток
        out     61h,al    ;в порт 61п
        mov     cx,2083 ;количество шагов
musicup:
;в ax значение нижней границы частоты 1193000/2651= 450 Гц,
;где 1193000- частота динамика
        mov ax, tonelow
        out 42h,al         ;al в порт 42h
        mov al,ah          ;обмен между a1 и ah
        out 42h,al         ;старший байт ax (ah) в порт 42h
        add tonelow, 1     ;увеличиваем частоту
        delay 1            ;задержка на 1 мкс
        mov dx,tonelow     ;текущее значение частоты в dx
        mov temp, dx       ;temp – верхнее значение частоты
        loop musicup       ;повторить цикл повышения
        mov cx, 2083
musicdown:
        mov ax, temp       ;верхнее значение частоты в ax
        out 42h, al        ;младший байт ax (a1) в порт 42h
        mov al, ah         ;обмен между a1 и ah
        out 42h, al        ;в порт 42h уже старший байт ax (ah)
        sub temp, 1        ;уменьшаем значение частоты
        delay 1            ;задержка на 1 мкс
        loop musicdown     ;повторить цикл понижения
nosound:
        in al,61h          ;получим значение порта 61h в a1
;слово состояния – выключение динамика и таймера
        and     al, 0FCh
        out     61h, al   ;в порт 61h
        mov     dx,2651 ;для последующих циклов
```

```
        mov     tonelow,dx ;увеличиваем счётчик проходов –
                           ;количество звучаний сирены
        inc     cnt
        cmp     cnt, 5     ;5 раз
        jne     go         ;если нет – идти на метку go
;формирование дескрипторов для реального режима
        assume  ds:gdt_seg
        mov     ax, 8h
        mov     ds.ax
        mov     gdt_ds_18.limit,0ffffh
        mov     gdt_es_vbf_20.limit,0ffffh
        mov     gdt_ss_28.limit,0ffffh
        mov     gdt_cs_30.limit,0ffffh
        assume  ds:data
;загрузка теневых дескрипторов
        mov     ax, 18h
        mov     ds, ax
        mov     ax, 20h
        mov     es, ax
        mov     ax, 28h
        mov     ss, ax
        db      0eah
        dw      offset jump
        dw      30h
jump:
        mov     eax, cr0
        and     al, 0feh
        mov     cr0, eax
        db      0eah
        dw      offset r_mode
        dw      code
r_mode:
        mov     ax, data
        mov     ds, ax
        mov     ax, stk
        mov     ss, ax
;разрешаем прерывания
        sti
        xor     al, al
        out     70h, al
        ;окончание работы программы (стандартно)
        mov     ax,4c00h
        int     21 h
main    endp
code    ends
        end     main
```

Переключение задач

Переключение задач осуществляется, если:

- текущая задача выполняет дальний JMP или CALL на шлюз задачи или прямо на TSS;
- текущая задача выполняет IRET, если флаг NT равен 1;
- происходит прерывание или исключение, в качестве обработчика которого в IDT записан шлюз задачи.

При переключении процессор выполняет следующие действия:

1. Для команд CALL и JMP проверяет привилегии (CPL текущей задачи и RPL селектора новой задачи не могут быть больше, чем DPL шлюза или TSS, на который передается управление);
2. Проверяется дескриптор TSS (его бит присутствия и лимит);
3. Проверяется, что новый TSS, старый TSS и все дескрипторы сегментов находятся в страницах, отмеченных как присутствующие;
4. Сохраняется состояние задачи;
5. Загружается регистр TR. Если на следующих шагах происходит исключение, его обработчику придется доделывать переключение задач, вместо того, чтобы повторять ошибочную команду;
6. Тип новой задачи в дескрипторе изменяется на занятый и устанавливается флаг TS в CR0;
7. Загружается состояние задачи из нового TSS: LDTR, CR3, EFLAGS, EIP, регистры общего назначения и сегментные регистры;

Если переключение задачи вызывается командами JMP, CALL, прерыванием или исключением, селектор TSS предыдущей задачи записывается в поле связи новой задачи и устанавливается флаг NT. Если флаг NT установлен, команда IRET выполняет обратное переключение задач.

При любом запуске задачи ее тип изменяется в дескрипторе на занятый. Попытка вызвать такую задачу приводит к #GP, сделать задачу снова свободной можно, только завершив ее командой IRET или переключившись на другую задачу командой JMP.

ЛИСТИНГ

Программа переключения задач

```
.386p
RM_seg segment para public "CODE" use16
assume cs:RM_seg,ds:PM_seg,ss:stack_seg
start:
; подготовить сегментные регистры
push PM_seg
pop ds
;проверить, не находимся ли мы уже в PM
mov eax,cr0
test al,1
jz no_V86
;сообщить и выйти
mov dx,off set v86_msg
err_exit:
push cs
pop ds
mov ah, 9
int 21h
mov         ah,4Ch
int         21h
; убедиться, что мы не под Windows
no_V86:
mov         ax,1600h
int         2Fh
test        al,al
jz          no_windows
;сообщить и         выйти
mov         dx, off set win_msg
jmp         short err_exit
;сообщения об ошибках при старте
v86Jnsg db    "Процессор в режиме V86 – нельзя переключиться в PM$"
win_msg db "Программа запущена под Windows – нельзя перейти в кольцо 0$"
;итак, мы точно находимся в реальном режиме no_windows:
; очистить экран
mov         ax,3
int         10h
;вычислить базы для всех дескрипторов сегментов данных
xor         eax, eax
mov         ax, RM_seg
```

```
shl         eax, 4
mov         word ptr GDT_16bitCS+2, ax
shr         eax, 16
mov         byte ptr GDT_16bitCS+4, al
mov         ax, PM_seg
shl         eax, 4
mov         word ptr GDT_32bitCS+2, ax
mov         word ptr GDT_32bitSS+2, ax
shr         eax, 16
mov         byte ptr GDT_32bitCS+4, al
mov         byte ptr GDT_32bitSS+4, al
;вычислить линейный адрес GDT
xor         eax, eax
mov         ax, PM_seg
shl         eax, 4
push        eax
add         eax, off set GDT
mov dword ptr gdtr+2, eax
;загрузить GDT
 lgdt f     word ptr gdtr
;вычислить линейные адреса сегментов TSS наших двух задач
pop eax
push eax
add eax, offset TSS_0
mov word ptr GDT_TSS0+2, ax
shr         eax, 16
mov         byte ptr GDT_TSS0+4, al
pop         eax
add         eax,offset TSS_1
mov         word ptr GDT_TSS1+2, ax
shr         eax, 16
mov         byte ptr GDT_TSS1+4, al
;открыть A20
mov         al, 2
out         92h, al
cli ;запретить прерывания
;запретить NMI
in          al, 70h
or          al, 80h
out         70h, al
;переключиться в PM
mov         eax, cr0
or          al, 1
```

```
mov         cr0, eax
;загрузить CS
db          66h
db          0EAh
dd          offset PM_entry
dw          SEL_32bitCS
RM_return:
; переключиться    в реальный режим RM
mov         eax, cr0
and         al, 0FEh
mov         cr0, eax
;сбросить очередь предвыборки и загрузить CS
db          0EAh
dw          $+4
dw          RM_seg
;настроить сегментные регистры для реального режима
mov         ax, PM_seg
mov         ds, ax
mov         es, ax
mov         ax, stack_seg
mov         bx, stack_l
mov         ss, ax
mov         sp, bx
;разрешить NMI
in          al, 70h
and         al, 07FH
out         70h, al
;разрешить прерывания
sti
;завершить программу
mov ah,4Ch
int 21h
RM_seg ends
PM_seg segment para public "CODE"           use32
assume cs:PM_seg
; таблица глобальных дескрипторов
GOT label byte
 db         8 dup(0)
GDT_flatDS db      0FFh, 0FFh, 0, 0, 0, 10010010b, 11001111b, 0
GDT_16bitCS db     0FFh, 0FFh, 0, 0, 0, 10011010b, 0,0
GDT_32bitCS db     0FFh, 0FFh,0, 0, 0, 10011010b, 11001111b, 0
GDT_32bitSS db     0FFh, 0FFh, 0, 0, 0, 10010010b, 11001111b,0
; сегмент TSS задачи 0       (32-битный свободный TSS)
```

```
GDT_TSS0 db       067h, 0, 0, 0, 0, 10001001b, 01000000b, 0
; сегмент TSS задачи 1       (32-битный свободный TSS)
GDT_SS1 db        067h,0, 0, 0, 0, 10001001b, 01000000b, 0
gdt_size = $-GDT
gdtr dw    gdt_size-1 ;размер GDT
   dd                 ? ;адрес GDT ;используемые селекторы
SEL_flatDS equ    001000b
SEL_16bitCS equ   010000b
SEL_32bltCS equ   011000b
SEL_32bitSS equ   100000b
SEL_TSS0 equ      101000b
SEL_TSS1 equ      110000b
```

;сегмент TSS_0 будет инициализирован, как только мы выполним переключение

```
;из нашей основной задачи. Конечно, если бы мы собирались использовать
;несколько уровней привилегий, то нужно было бы инициализировать стеки
TSS_0 db 68h dup(0)
;сегмент TSS_1. В него будет выполняться переключение, так что надо
;инициализировать все, что может потребоваться:
TSS_1 dd 0,0,0,0,0,0,0,0 ;связь, стеки, CR3
dd offset task_1 ;EIP
;регистры общего назначения
dd 0,0,0,0,0,stack_l2, 0, 0, 0B8140h ;(ESP и EDI)
;сегментные регистры
dd SEL_flatDS,SEL_32bitCS,SEL_32bitSS,8EL_flatOS,0,0
dd 0 ; LDTR
dd 0 ; адрес таблицы ввода-вывода
; точка входа в 32-битный защищенный режим
PM_entry:
;подготовить регистры
xor eax, eax
mov ax, SEL_flatDS
mov ds, ax
mov es, ax
mov ax, SEL_32bitSS
mov ebx, stack_l
mov ss, ax
mov esp, ebx
;загрузить TSS задачи 0 в регистр TR
mov ax, SEL_TSS0
Ltr ax
```

;только теперь наша программа выполнила все требования к переходу в защищенный режим

```
xor eax, eax
mov edi, 0B8000h   ;DS:EDI – адрес начала экрана
task_0:
mov byte ptr ds:[edi],al         ;вывести символ AL на экран
                                 ; дальний переход на TSS задачи 1
db 0EAh
dd 0
dw SEL_TSS1
add edl, 2  ;DS:EDI – адрес следующего символа
inc al                   ;AL – код следующего символа;
cmp a1,80 ;если это 80,
jb task_0   ;выйти из цикла
;дальний переход на процедуру выхода в реальный режим
db 0EAh
dd offset RM_return
dw SEL_16bitCS
;задача 1
task_1:
mov byte ptr ds:[edi],al ;вывести символ на экран
inc al                    ;увеличить код символа
add edi, 2                ;увеличить адрес символа, переключиться на задачу 0
db 0EAh
dd 0
 dw SEL_TSS0
;сюда будет приходить управление, когда задача 0 начнет выполнять переход
;на задачу 1 во всех случаях, кроме первого
 mov ecx, 02000000h          ;небольшая пауза, зависящая от скорости
 loop $                      ;процессора
 jmp task_1
PM_seg ends
stack_seg segment para stack "STACK"
stack_start db 100h dup(?)              ;стек задачи0
stack_l = $-stack_start
stack_task2 db 100h dup(?) ;стек задачи 1
stack_12 = $-stack_start
stack_seg ends
end start
```

Чтобы реализовать многозадачность в реальном времени в нашем примере, достаточно создать обработчик прерывания системного таймера IRQ0 в виде отдельной (третьей) задачи и поместить в IDT шлюз этой задачи. Текст обработчика для примера мог быть крайне простым:

```
task_3: ;это отдельная задача – не нужно сохранять регистры!
mov al,20h
out 20h,al
jmp task_0
ror al,20h
out 20h,al
jmp task_1
jmp task_3
```

Но при вызове обработчика прерывания старая задача помечается как занятая в GDT и повторный JMP на нее приведет к ошибке. Вызов задачи обработчика прерывания, так же как и вызов задачи командой CALL, подразумевает, что она завершится командой IRET. Именно команду IRET оказывается проще всего вызвать для передачи управления из такого обработчика – достаточно только подменить селектор вызвавшей нас задачи в поле связи и выполнить IRET.

```
task_3:          ;при инициализации DS должен быть установлен на PM_seg
mov al, 20h
out 20h, al
mov word ptr TSS_3, SEL_TSS0
iret
mov al, 20h
out 20h, al
mov word ptr TSS_3, SEL_TSS1
iret
jmp task_3
```

Единственное дополнительное изменение, которое нужно внести – это инициализировать дескриптор TSS задачи task_1 уже как занятый, так как управление на него будет передаваться командой IRET, что, впрочем, не составляет никаких проблем.

Помните, что во вложенных задачах команда IRET не означает конца программы – следующий вызов задачи всегда передает управление на следующую после IRET команду.

2. ОБРАБОТКА ПРЕРЫВАНИЙ В ЗАЩИЩЕННОМ РЕЖИМЕ

Обработка прерываний в защищенном режиме отличается от обработки в реальном режиме так же сильно, как и сам защищенный режим отличается от реального. Причины здесь в том, что, во-первых, в защищенном режиме несколько иное распределение номеров векторов прерываний и, во-вторых, здесь принципиально иным является сам механизм обработки прерываний.

Для того, чтобы разобраться с первой причиной, посмотрим на распределение векторов прерываний в защищенном режиме. В табл. 3 приложения А для удобства сравнительного анализа приведено распределение номеров прерываний в реальном и защищенном режимах.

Некоторых прерываний в столбце для реального режима наблюдается двойственность источников прерываний (два возможных источника прерывания приведены через косую черту). Это объясняется тем, что изначально в микропроцессоре 18086/88 были жестко определены всего 5 прерываний с номерами 0...4. В последующих моделях микропроцессора разработчики зарезервировали номера в таблице прерываний 0...31. Но программы BIOS тоже настроены на обработку аппаратных прерываний с определенными номерами, которые, в свою очередь, накладываются на номера исключений из диапазона 0...31. Это может быть источником конфликтов, суть которых в том, что непонятно, от какого источника пришло прерывание. Это обстоятельство нужно учитывать программисту, который самостоятельно пишет какой-либо из обработчиков прерываний для замены системного. Системные обработчики прерываний для предотвращения таких конфликтов выполняют дополнительные действия по идентификации источника прерывания и лишь затем осуществляют непосредственно обработку прерывания.

Для того чтобы разобраться со второй причиной несовместимости механизмов обработки прерываний в реальном и защищенном режимах, рассмотрим схему обработки прерывания в защищенном режиме (рис. 14).

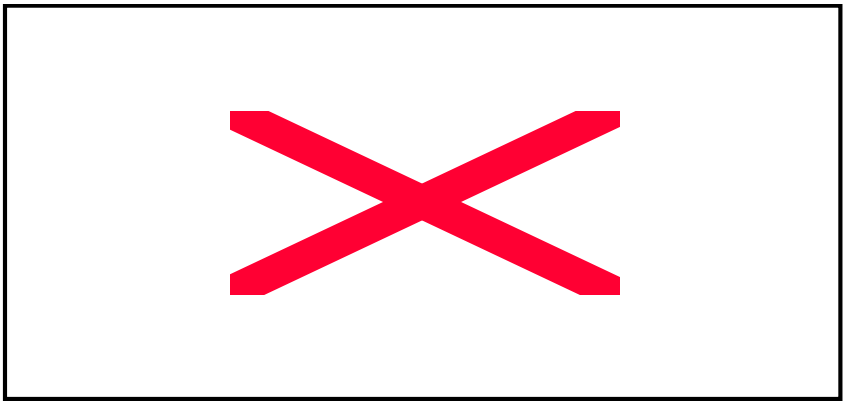

Рис. 14. Схема обработки прерывания в защищенном режиме

Ключевыми компонентами в этой схеме являются *дескрипторная таблица прерываний IDT* и системный регистр idtr. При возникновении прерывания от источника с номером *n* микропроцессор, находясь в защищенном режиме, выполняет следующие действия (рис. 14):

1. Определяет местонахождение таблицы *IDT*, адрес и размер которой содержится в регистре idtr.
2. Складывает значение адреса, по которому размещена *IDT*, и значение n*8. По данному смещению в таблице *IDT* должен находиться 8-байтовый *дескриптор,* определяющий местоположение процедуры обработки прерывания.
3. Переключается на процедуру обработки прерывания.

Это очень обобщенная схема. На практике все гораздо глубже, интереснее и сложнее.

Начнем с того, что в защищенном режиме прерывания и исключения можно разделить на несколько групп:

- сбой;
- ловушка;
- аварийное завершение.

Это деление производится в соответствии со следующими признаками:

- какая информация сохраняется о месте возникновения прерывания или исключения?
- возможно ли возобновление прерванной программы?

Исходя из этих признаков, можно дать следующие характеристики вышеперечисленным группам:

- *Сбой (ошибка)* – прерывание или исключение, при возникновении которого в стек записываются значения регистров cs:ip, указывающие на команду, вызвавшую данное прерывание. Это позволяет, получив доступ к сегменту кода, исправить ошибочную команду в обработчике прерывания и, вернув управление программе, фактически осуществить ее рестарт (вспомните, что в реальном режиме при возникновении прерывания в стеке всегда запоминается адрес команды, следующей за той, которая вызвала это прерывание).
- *Ловушка* – прерывание или исключение, при возникновении которого в стек записываются значения регистров cs:ip, указывающие на команду, следующую за командой, вызвавшей данное прерывание. Так же, как и в случае ошибок, возможен рестарт программы. Для этого необходимо лишь исправить в обработчике прерывания соответствующие код или данные, послужившие источником ошибки. После этого перед возвратом управления нужно скорректировать значение ip в стеке на длину команды, вызвавшей данное прерывание. Как видим, механизм ловушек похож на механизм прерываний в реальном режиме, хотя не во всем. Здесь есть один тонкий момент. Если прерывание типа ловушки возникло в команде передачи управления jmp, то содержимое пары cs:ip в стеке будет отражать результат этого перехода, то есть соответствовать команде назначения.

• *Аварийное завершение* – прерывание, при котором информация о месте его возникновения недоступна или неполна и поэтому рестарт практически невозможен, если только данная ситуация не была запланирована заранее.

Микропроцессор жестко определяет, какие прерывания являются ошибками, ловушками и авариями. Из приведенных выше описаний этих типов прерывания видно, что соответствующие программы-обработчики будут отличаться алгоритмами работы. По этой причине полезной является информация, приведенная в четвертом столбце табл. 3 (приложение А). В ней каждое из прерываний защищенного режима классифицировано в соответствии с приведенной выше схемой.

Но в табл. 3 (приложение А) присутствует еще один столбец. Его наличие объясняется существованием еще одной особенности: некоторые прерывания при своем возникновении дополнительно генерируют и записывают в стек так называемый *код ошибки*. Этот код может впоследствии использоваться для установления источника прерывания. Код ошибки, если он генерируется для данного прерывания, записывается в стек вслед за содержимым регистров eflags, cs и eip (рис. 15).

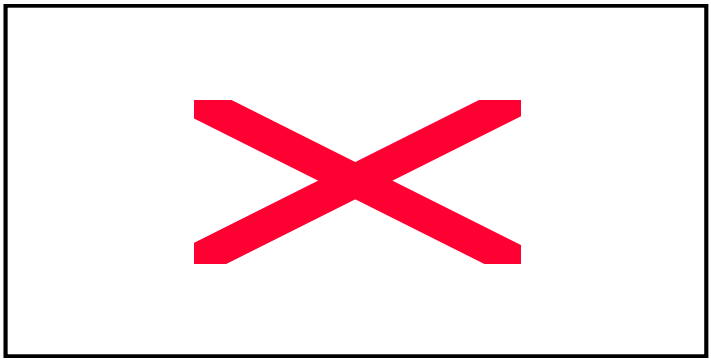

Рис. 15. Стек для прерываний с кодом и без кода ошибки

Какую информацию можно извлечь из кода ошибки? Код ошибки структурно представляет собой совокупность четырех полей и имеет размерность 16 бит. Если установлен режим адресации в сегменте use32, то при помещении в стек код ошибки расширяется нулями до 32 бит в сторону старших разрядов. Поля кода ошибки перечислены в табл. 3.

Таблица 3

Структура кода ошибки

Номер бита	Обозначение	Назначение и содержимое
0	EXTENT	Если EXTENT = 1, ИС вызвана аппаратным прерыванием; если EXTENT = 0, ИС вызвана последней выполнявшейся командой
1	*IDT*	Если *IDT* = 1, в поле INDEX содержится индекс дескриптора в таблице *IDT*; если *IDT* = 0, в поле INDEX содержится индекс дескриптора в таблице *GDT* или *LDT*
2	TI	Если TI = 0, значение в поле INDEX соответствует номеру дескриптора в таблице *GDT*; если TI = 1, значение в поле INDEX соответствует номеру дескриптора в таблице *LDT*
3...15	INDEX	Номер дескриптора в одной из таблиц *IDT*, *GDT* или *LDT* в соответствии с состояниями полей EXTENT, *IDT* или TI; или 0 для некоторых исключений

Что представляет собой *дескрипторная таблица прерываний IDT*?

На рис. 14 показаны роль и, соответственно, функциональное назначение таблицы прерываний защищенного режима *IDT*. Она связывает каждый вектор прерывания с дескриптором процедуры или задачи, которая будет обрабатывать это прерывание. Формат таблицы *IDT* подобен формату *GDT* и *LDT*, то есть представляет собой совокупность 8-байтовых дескрипторов. Отличия таблицы *IDT* от указанных таблиц состоят в следующем:

- нулевой дескриптор, в отличие от таблицы *GDT*, используется; он описывает шлюз для программы обработки исключительной ситуации 0 (ошибка деления);
- дескрипторы в таблице *IDT* строго упорядочены в соответствии с номерами прерываний. В таблицах *GDT* и *LDT*, как помните, порядок описания дескрипторов роли не играет, хотя и допускается наличие некоторых соглашений по их упорядоченности;
- размерность таблицы *IDT* – не более 256 элементов размером по восемь байт, по числу возможных источников прерываний. В отдельных случаях есть смысл описывать все 256 дескрипторов этой таблицы, формируя для неиспользуемых номеров прерываний шлюзы-заглушки. Это позволит корректно обрабатывать все прерывания, даже если они и не планируются к использованию в данной задаче. Если этого не сделать, то при незапланированном преры-

вании с номером, превышающим пределы *IDT* для данной задачи, будет возникать исключительная ситуация общей защиты (с номером 13 (ODh));

- при работе с прерываниями микропроцессор всегда определяет местоположение таблицы *IDT* по полю базового адреса в регистре idtr (табл. 3). Это он делает независимо от режима, в котором работает. В реальном режиме содержимое поля базового адреса в регистре idtr равно нулю, поэтому работа с таблицей векторов прерываний выполняется без проблем (структура ее в данном случае не имеет значения). Из вышесказанного следует, что возможно произвольное размещение в памяти этой таблицы не только в защищенном режиме, но и реальном.

Шлюзом обычно называют дескрипторы в таблице прерываний *IDT* для того, чтобы подчеркнуть их специфику по сравнению с дескрипторами в таблицах *GDT* и *LDT*. Шлюзы предназначены для указания точки входа в программу обработки прерывания. В дескрипторной таблице прерываний *IDT* могут содержаться шлюзы трех типов. Их существование объясняется наличием разных по характеру источников прерывания и особенностями передачи управления в программу обработки. По формату шлюзы похожи на дескрипторы в таблицах *GDT* и *LDT*. Физически микропроцессор отличает их по значению в поле типа и содержимому остальных полей. Возможны три типа шлюзов:

- шлюз ловушки;
- шлюз прерывания;
- шлюз задачи.

Дадим им более подробную характеристику.

2.1. Шлюз ловушки

Шлюз ловушки – элемент таблицы *IDT*, имеющий формат, показанный на рис. 16.

В табл.4 приведены назначения полей в формате шлюза ловушки.

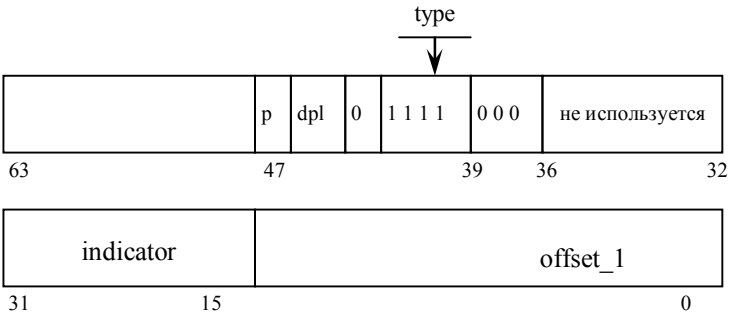

Рис. 16. Формат шлюза ловушки

Таблица 4

Поля шлюза ловушки

Номера битов	Обозначение	Значение	Назначение
0...15	offset_l	nn...nn	Смещение в сегменте (первая половина)
16...31	indicator	mm...mm	Селектор, указывающий на дескриптор в *LDT* или *GDT*
32...36		-	Не используется
37...39		000	Постоянное значение
40...43	type	1111	Тип шлюза – ловушка
44		0	Постоянное значение
45...46	dpl	nn (обычно dpl = 112)	Определение минимального уровня привилегий задачи, которая может передать управление обработчику прерываний через данный шлюз
47	P	0 или 1	Бит присутствия
48...63	offset_2	nn..nn	Смещение в сегменте (вторая половина)

Когда возникает прерывание и его вектор выбирает в таблице *IDT* дескриптор шлюза с типом ловушки, микропроцессор сохраняет в стеке информацию о месте, где он прервал работу текущей программы (рис. 15). После этого он передает управление в соответствии с полями indicator и offset. Поле indicator представляет селектор одной из таблиц, *GDT* или *LDT*, в зависимости от состояния бита TI в нем. Поле offset определяет смещение в сегменте кода. Этот сегмент кода описывается дескриптором, на который указывает селектор в поле indicator. После того как управление было передано обработчику прерывания, он выполняет свою работу до тех пор, пока не встретит команду iret. Эта команда восстанавливает из стека состояние регистров eflags, cs и eip на момент возникновения прерывания, и работа приостановленной программы продолжается. При подготовке выхода из программы обработки прерывания имейте в виду, что команда iret ничего не знает о возможности наличия в стеке кода ошибки, поэтому для корректного возврата управления не забудьте при необходимости предварительно удалить командой pop код ошибки из стека.

2.2. Шлюз прерывания

Шлюз прерывания – элемент таблицы *IDT*, имеющий формат, показанный на рис. 17.

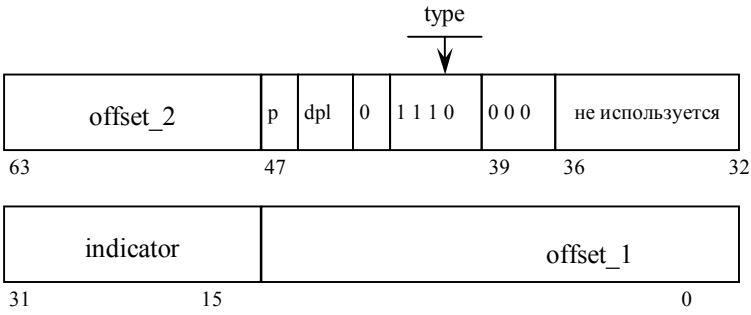

Рис.17. Формат шлюза прерывания

Если внимательно посмотреть на этот формат, то можно сделать вывод, что его отличие по сравнению с форматом шлюза ловушки только в поле типа. Это имеет свою причину. При возникновении прерывания, которому соответствует шлюз прерывания, микропроцессор выполняет те же действия, что и для шлюза ловушки, но с одним важным отличием. Когда микропроцессор передает управление обработчику прерывания через шлюз прерывания, то он сбрасывает флаг прерывания в регистре eflags в 0, запрещая тем самым обработку аппаратных прерываний. Этот факт имеет важное значение для программирования обработчиков аппаратных и программных прерываний. Если у вас есть сомнение в том, какой из двух рассмотренных типов шлюзов использовать – применяйте шлюз прерывания.

2.3. Шлюз задачи

Шлюз задачи – элемент таблицы *IDT*, имеющий формат, показанный на рис. 18.

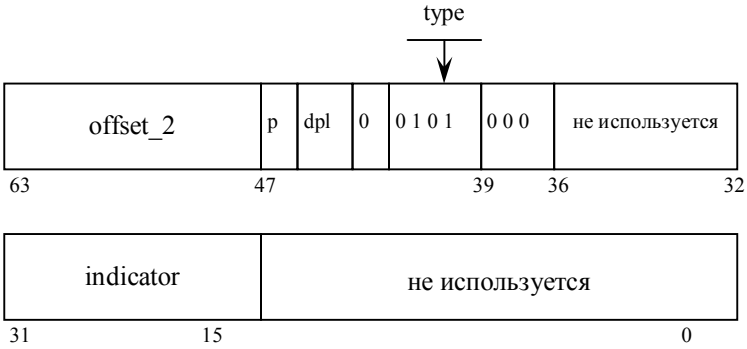

Рис. 18. Формат шлюза задачи

Когда микропроцессор при выполнении прерывания встречает в таблице *IDT* шлюз задачи, это означает, что он должен осуществить переход на особый объект (задачу). Задача является частью механизма многозадачности. На самом деле, многозадачность не означает того, что в некоторый момент времени микропроцессор может одновременно выполнять несколько командных потоков. Микропроцессор занят в каждый конкретный момент времени выполнением команд только одного потока. Поэтому многозадачность в случае нашего микропроцессора на самом деле является псевдомногозадачностью. Это означает, что микропроцессор, выполняя команды одного потока, по истечении некоторого времени должен переключиться на выполнение команд другого потока и т. д.

За счет высокого быстродействия микропроцессора создается иллюзия того, что несколько задач выполняются одновременно. Но как быть с ресурсами, которые эти потоки команд разделяют? Такими ресурсами являются, в частности, регистры. Их нужно где-то сохранять на то время, пока работает другая задача. В микропроцессорах Intel эта проблема решена следующим образом. Для каждой задачи определяется *сегмент состояния задачи* TSS (Task Segment Status) со строго определенной структурой. В этом сегменте есть поля для сохранения всех регистров общего назначения, некоторых системных регистров и другой информации. Всю совокупность этой информации называют *контекстом задачи.* Этот сегмент описывается, подобно другим сегментам, дескриптором в таблице *GDT* или *LDT*. Если с помощью некоторого селектора обратиться к такому дескриптору, то микропроцессор осуществит переключение на соответствующую задачу. Подобные переключения могут, в частности, осуществляться операционной системой, поддерживающей многозадачность, в соответствии с некоторой дисциплиной разделения времени между задачами. Переключение задач может производиться обычными командами межсегментной передачи управления либо по возникновению прерывания при переходе к обработчику прерывания через шлюз задачи.

Таким образом, наличие дескриптора с типом шлюза задачи в таблице *IDT* позволяет при возникновении соответствующего прерывания переключаться на некоторую задачу, которая будет осуществлять обработку прерывания. Поле indicator в шлюзе задачи содержит селектор, который определяет местонахождение дескриптора сегмента TSS – в таблице *GDT* или *LDT* (в зависимости от состояния бита TI селектора).

Итак, еще раз подчеркнем отличия перечисленных типов шлюзов. Шлюзы ловушки и прерывания с помощью полей indicator и offset определяют адрес, по которому находится точка входа в программу обработки прерывания. Шлюз задачи предназначен для реализации принципиально иного перехода к обработчику прерываний – с использованием механизма переключения задач.

После такого подробного обсуждения мы готовы к тому, чтобы рассмотреть, каким образом практически реализуется обработка прерываний в защи-

щенном режиме. Так как при этом возникает достаточно много нюансов, то мы рассмотрим задачу, которая бы отражала суть большинства из них. В нашей программе мы будем обрабатывать одно аппаратное прерывание (от таймера), две разнотипные исключительные ситуации и обычное пользовательское прерывание. На примере этой задачи будут показаны все аспекты обработки прерываний в защищенном режиме, за исключением использования задач в качестве обработчика прерываний (см. выше обсуждение шлюзов задач).

В общем случае для того, чтобы сделать возможным обработку прерываний в защищенном режиме, необходимо выполнить следующие действия:

- инициализировать таблицу *IDT*;
- составить процедуры обработчиков прерываний;
- запретить аппаратные прерывания;
- перепрограммировать контроллер прерываний i8259A;
- загрузить регистр idtr адресом и размером таблицы *IDT*;
- перейти в защищенный режим;
- разрешить обработку прерываний.

Далее микропроцессор, находясь в защищенном режиме, может производить обработку необходимых прерываний. По окончании работы для возврата в реальный режим нужно будет все "поставить на свои места", выполнив для этого последовательность следующих действий:

- запретить аппаратные прерывания;
- выполнить обратное перепрограммирование контроллера прерываний;
- перейти в реальный режим работы микропроцессора;
- разрешить обработку аппаратных прерываний.

Обсудим эти действия подробнее.

2.4. Пример программы "Обработка прерываний в защищенном режиме"

За основу возьмем программу *"Программа перевода микропроцессора в защищенный режим"* и дополним ее функционально, обрабатывая некоторые прерывания.

Инициализация таблицы IDT. Выше мы уже определились с тем, что так же, как и в реальном режиме, все прерывания защищенного режима имеют свои номера от 0 до 255. Отличие их от прерываний реального режима в том, что под цели микропроцессора (исключительные ситуации или просто исключения) отдано гораздо большее количество номеров – первые 32 вектора с номерами 0...31. Из этих 32 векторов реально используются только 0...17, остальные вектора 18...31 пока зарезервированы для будущих разработок. Для того, чтобы сформировать таблицу *IDT* в целом, необходимо определиться с описанием отдельных ее дескрипторов – шлюзов. Можно предло-

жить следующий, оформленный в виде структуры, шаблон для описания дескрипторов в таблице *IDT*:

```
descr_IDT struc
offs_1     dw 0    ;младшая часть адреса смещения обработчика прерывания
sel        dw 30   ;селектор на сегмент команд в таблице GOT
no_use     db  0
type_attr db  8eh ;по умолчанию шлюз прерывания, для ловушки – 8fh
offs_2     dw  0   ;старшая часть адреса смещения обработчика прерывания
ends
```

Теперь можно приступить к формированию таблицы *IDT*. Это можно делать как в отдельном сегменте, так и в сегменте данных. Чтобы не загромождать сегмент данных, оформим таблицу *IDT* в виде отдельного сегмента.

```
IDT_seg segment
        rept 5
        descr_IDT <dummy…>
        endm
int05h descr_IDT <int_05h…>
        rept 7
        descr_IDT <dummy_err… >
        endm
int0dh descr_IDT <int_0dh…>
        rept 3
        descr_IDT <dummy…>
        endm
int11h descr_IDT <dummy_err…>
        rept 14
        descr_IDT <dummy... >
        endm
int20h descr_IDT <new_08h…>
int21h descr_IDT <sirena,38…>
        rept 221
        descr_IDT <dummy…>
        endm
IDT_seg ends
```

Выше полностью описана вся таблица прерываний. При этом мы инициализировали все дескрипторы именем процедур обработки прерываний. Основная часть дескрипторов имеет в качестве обработчиков процедуру dummy, то есть они будут обрабатываться одной программой. Для некоторых прерываний мы ввели уникальные имена; эти прерывания будут иметь свои процедуры обработки в нашей программе.

Обработчики прерываний. Расположение и порядок следования процедур обработки прерываний в памяти может быть произвольным. Главное, не забывать поместить их адреса в соответствующие дескрипторы.

При написании самих программ обработки прерываний мы должны помнить о том, что при возникновении некоторых прерываний в стек вслед за содержимым регистров eip, cs и eflags может записываться еще и *код ошибки.* Поэтому в общем случае процедура обработки прерывания должна происходить по следующей схеме действий:

1. Снять со стека и проанализировать код ошибки (если он есть);
2. Сохранить в стеке используемые в обработчике регистры микропроцессора;
3. Выполнить необходимые действия, в том числе подготовить возможный рестарт команды, вызвавшей прерывание. Подобный рестарт подразумевает возобновление выполнения прерванной программы, начиная с команды, инициировавшей процесс прерывания;
4. Восстановить сохраненные на шаге 2 регистры;
5. Выдать команду iret.

Прерываний, для которых микропроцессор формирует код ошибки, немного – всего 8. В программе из листинга(см. ниже) приведены все возможные типы прерываний защищенного режима. Так, для исследования исключений типа ошибок взяты исключения 5 и 13 (0Dh), а для ловушек – 3 и обработка прерывания пользователя.

Программирование контроллера прерываний i8259A. При загрузке BIOS производит инициализацию контроллеров прерываний (ведущего и ведомого). При этом BIOS назначает им базовые номера: ведущему – 08h, ведомому – 70h. Из этого следует, что если мы разрешим обработку аппаратных прерываний в защищенном режиме, то первое же прерывание от таймера заставит микропроцессор через таблицу *IDT* обратиться к процедуре dummy. Что же делать? Ничего не остается, как перепрограммировать ведущий контроллер на другое значение базового вектора. Ведомый контроллер перепрограммировать не нужно – значение его базового вектора ничем нам не мешает, так как оно находится в области, отведенной под прерывания пользователя.

Загрузка регистра IDTR. Аналогично таблице *GDT* нужно сообщить микропроцессору, где находится таблица *IDT*. Это делается с помощью специально выделенного для этой цели регистра idtr Таким образом, в защищенном режиме любая задача, имея достаточный уровень привилегий, может создать свою среду для обработки прерываний Для загрузки регистра idtr можно использовать ту же структуру point, что и для регистра gdtr. Загрузка регистра idtr производится специальной командой lidt, которая имеет следующий формат:

lidt адрес_48-битного_поля (Load *IDT* register) – загрузить регистр idtr Команда idtr загружает системный регистр idtr содержимым 6-байтового поля, адрес которого указан в качестве операнда.

```
point struc
lim dw 0
adr dd 0
ends
data segment
point_IDT point <IDT_size,0>
…
data ends
code segment
…
xor eax, eax
mov ax, IDT_seg
shl eax, 4
mov dword ptr point_IDT adr, eax
igtr pword point_IDT
…
code ends
```

После этого можно перейти в защищенный режим и разрешить аппаратные прерывания. Выполнив работу в защищенном режиме, мы готовимся к обратному переходу в реальный режим. Для этого, кроме восстановления вычислительной среды этого режима, необходимо восстановить и соответствующий механизм прерываний. Для повышения наглядности программы вы можете процесс перепрограммирования контроллера оформить в виде процедуры или макрокоманды.

ЛИСТИНГ

```
.386P                   ;разрешение инструкций i386
.MODEL large
include show.inc
include mac.inc
;структура для описания дескрипторов сегментов
descr STRUC
limit dw 0
base_1      dw      0
base_2      db      0
attr        db      0
lim_atr     db      0
base_3      db      0
ENDS
; макрос инициализации дескрипторов
load_descrMACRO des, seg_addr, seg_size
mov         des.limit, seg_size
```

```
xor         eax, eax
mov         ax, seg_addr
shl         eax, 4
mov         des.base_1, ax
rol         eax, 16
mov         des.base_2, al
ENDM
atr MACRO           descr, bitl, bit2, bit3
atr_&descr=constp or bit1 or bit2 or bit3
ENDM
;структура для описания псевдодескриптора gdtr и idtr
point STRUC
lim         dw 0
adr          dd 0
ENDS
;структура для описания дескрипторов таблицы IDT
descr_LDT STRUC
offs_1 dw 0
sel      dw 30h ;селектор сегмента команд в таблице GDT
no_use db 0
type_attrdb 8eh ;шлюз прерывания
offs_2 dw 0
ENDS
;атрибуты для описания дескрипторов сегментов
constp              equ     10010000b
code_r_n            equ     00010000b
code_r_y            equ     00000010b
data_wm_n           equ     00000000b
data_wm_y           equ     00000010b
code_n              equ     00000000b
code_p              equ     00000101b
data_               equ     00000000b
stack_              equ     00000000b
code_               equ     00001000b
data_stk            equ     00000000b

stk         segment stack "stack"
db 256      dup (0)
stk         ends
;сегмент с таблицей глобальных дескрипторов
GDT_seg segment para public "data" use16
GDT_0 descr <0,0,0,0,0 0>
atrGDT_GDT_8, data_wm_y, data_, data_stk
```

```
;описывает саму GDT
GDT_GDT_8 descr <0,0,0, atr_GDT_GDT_8,0,0>
GDT_LDT_10 descr <0,0,0,0,0,0>     ;не используем
atrGDT_ds_18, data_wm_y, data_, data_stk
;дескриптор сегмента данных
GDT_ds_18 descr <0,0,0,atr_GDT_ds_18,0,0>
atrGDT_vbf_20, data_wm_y, data_, data_stk
GDT_es_vbf_20 descr <0,0,0,atr_GDT_vbf_20,0,0>     ;видеобуфер
atrGDT_ss_28, data_wm_y, stack_, data_stk
GDT_ss_28 descr <0,0,0, atr_GDT_ss_28,0,0> ;сегмент стека
atrGDT_cs_30, code_r_y, code_n, code_
GDT_cs_30 descr <0,0,0, atr_GDT_cs_30,0, 0> ;сегмент кода
atrGDT_sirena_38, code_r_y, code_n, code_
GDT_sirena_38 descr <0 0,0,atr_GDT_sirena_38,0,0>
GDT_size=$-GDT_0-1          ;размер GDT минус 1
GDT_seg ends

IDT_seg segment para public "data" use16
 int00h descr_IDT <dummy…>
 REPT 2
 descr_IDT <dummy…>
 ENDM
int03h descr_IDT <int_03h…>
 descr_IDT <dummy…>
int05h descr_IDT <int_05h…>
 REPT 7
 descr IDT <dummy_err…>
 ENDM
int0dh descr_IDT <int_0dh…>
 REPT 3
 descr_IDT <dummy…>
 ENDM
int11h descr_IDT <dummy_err…>
 REPT 14
 descr_IDT <dummy…>
 ENDM
int20h descr-IDT <new_08h…>
int21h descr_IDT <sirena,38h…>
 REPT 222
 descr_IDT <dummy…>
 ENDM
IDT_size=$-int00h-1
IDT_seg ends
```

```
;данные программы
data segment para public "data" usel6 ;сегмент данных
point_GDT point <GDT_size,0>
point_idt point <IDT_size,0>
char db "0"
maskf db 07h
position dw 2000
tonelow dw 2651 ;нижняя граница звучания 450 Гц
cnt db 0 ;счётчик для выхода из программы
temp dw ? ;верхняя граница звучания
min_index dw 0
max_index dw 99
data_size=$-point_GDT-1
data ends

sound segment byte private usel6
 assume cs:sound,ds:data,ss:stk
sirena PROC ;пользовательское прерывание
 push ds
 push ax
 push cx
 push dx
 go:
;заносим слово состояния 10110110b(0B6h) в командный регистр (порт 43h)
mov al, 0B6h
out 43h, al
in al, 61h ;получим значение порта 61h в al
or al, 3 ;инициализируем динамик и подаем ток в порт 61h
out 61h, al
mov cx, 2083 ;количество шагов ступенчатого изменения тона
musicup:
;в ax значение нижней границы частоты
mov ax, tonelow
out       42h, al    ;в порт 42h младшее слово ax :al
xchg      al, ah ;обмен между al и ah
out       42h, al ;в порт 42h старшее слово ax:ah
add       tonelow,1 ;повышаем тон
delay 1   ;задержка на 1 мкс
mov       dx, tonelow ;в dx текущее значение высоты
mov       temp, dx ;temp – верхнее значение высоты
loop      musicup ;повторить цикл повышения
mov       ex, 2083 ;восстановить счетчик цикла
```

```
musicdown:
mov         ax, temp ;в ax верхнее значение высоты
out         42h, al    ;в порт 42h младшее слово ax :al
mov         al, ah ;обмен между al и ah
out         42h, al ;в порт 42h старшее слово ax :ah
sub         temp, 1 ;понижаем высоту
delay 1     ;задержка на 1 мкс
loop musicdown ;повторить цикл понижения
nosound:
in al, 61h ;получим значение порта 61h в AL
and         al, 0FCh ;выключить динамик
out         61h, al ;в порт 61h
mov         dx, 2651 ;для последующих циклов
mov         tonelow, dx
inc         cnt ;увеличиваем счётчик проходов, то есть количество
            ;звучаний сирены
cmp         cnt,5 ;5 раз ?
jne         go ;если нет – идти на метку go
pop         dx
pop         cx
pop         ax
pop         ds
mov         bp, sp
mov         eax, [bp]
show        eax, 0
mov         eax, [bp+4]
show        eax, 160
mov         eax, [bp+8]
show        eax, 320
db 66h
iret
endp
sound_size=$-sirena-1
sound ends

code        segment byte public "code" use16
; сегмент кода с 16-разрядным режимом адресации
assume      cs:code,ss:stk
dummy       proc ;исключения без кода ошибки
mov         ax, 0ffffh
show        ax, 1600
db 66h
iret
```

```
endp
dummy_err proc ;исключения с кодом ошибки
pop eax ;снимаем со стека код ошибки и на экран
show eax,1760
db 66h
iret
endp
int_03h proc
;для этого исключения не формируется кода ошибки, поэтому
;анализируем содержимое eip в стеке и возвращаемся в программу
 pop eax
show eax, 0
push eax
db 66h
iret
endp
int_05h proc ;обработчик для 5-го исключения – команда bound код
;ошибки не формируется, поэтому корректируем индекс и выходим
mov a1, 5
mov ah,al
mov si,2
db 66h
iret
endp
int_0dh proc
pop eax ;снимаем со стека код ошибки
sub bx,4 ;исправляем адрес – причину возникновения исключения
;db 66h, возвращаемся обратно и рестарт виноватой команды
iret
endp
new_08h proc ;новое прерывание от таймера
assume ds:data
push ds
push es
push    ax
push    bx
mov     ax, 20h
mov     es, ax
scr:
mov     al, char
mov     ah, maskf
mov     bx, position
mov     es:[bx], ax
```

```
add         bx, 2
mov         position, bx
inc         char
pop         bx
pop         ax
pop         es
pop         ds
mov         al, 20h
out         20h, al
db 66h
iret
endp
new_8259a proc
;в al – значение нового базового вектора для ведущего контроллера
push        ax
mov         al,00010001b
out         20h, al ;ICW1 в порт 20h
jmp         $+2
jmp         $+2       ;задержка, чтобы успела отработать аппаратура
pop         ax
out         21h, al   ;ICW2 в порт 20h – новый базовый номер
jmp         $+2
jmp         $+2 ;задержка, чтобы успела отработать аппаратура
mov         al, 00000100b
out         21h, al ;ICW3 – ведомый подключается к уровню 2
jmp         $+2
jmp         $+2 ;задержка, чтобы успела отработать аппаратура
mov         al, 00000001b
out         21h, al ;ICW4 – EOI выдает программа пользователя
ret
endp

main        proc
 mov ax, stk
 mov ss, ax
 ;заполняем таблицу глобальных дескрипторов
 assume ds:GDT_SEG
 mov ax, GDT_SEG
 mov ds, ax
load_descr GDT_GDT_8, GDT_SEG, GDT_size
load_descr GDT_ds_18, DATA, data_size
load_descr GDT_es_vbf_20, 0b800h, 3999
load_descr GDT_ss_28, STK, 255
```

```
load_descr GDT_cs_30, CODE, code_size
load_descr GDT_sirena_38, SOUND, sound_size
assume ds:data
mov ax,data
mov ds,ax
;загружаем gdtr
xor eax, eax
mov ax, GDT_SEG
shl eax, 4
mov poi nt_GDT.adr, eax
lgdt point_GDT
;запрещаем прерывания
cli
mov al, 80h
out 70h, al
mov al, 20h ;новое значение базового вектора
call new_8259A
;загружаем idtr
xor eax, eax
mov ax, IDT_SEG
shl eax, 4
mov point_IDT.adr, eax
lidt point_IDT
;переключаемся в защищенный режим
mov eax, cr0
or al, 1
mov cr0, eax
;настраиваем регистры
db 0eah ;машинный код команды jmp
dw offset protect ;смещение метки перехода в сегменте команд
dw 30n ;селектор сегмента кода в GDT
protect:
;загрузить селекторы для остальных дескрипторов
mov ax, 18h
mov ds, ax
mov ax, 20h
mov es, ax
mov ax, 28h
mov ss, ax
;работаем в защищенном режиме:
;разрешаем прерывания от таймера, наблюдаем
sti
delay 3500
cli
```

```
;далее имитируем возникновение двух исключительных ситуаций (типа ошибки): 5 и 13
mov si, 130
bound si, dword ptr min_index
mov bx, data_size
mov ax, [bx]
;а теперь имитируем возникновение исключительной ситуации
;типа ловушки –3:
int 3
;обрабатываем их и в знак успеха запускаем из другого
;сегмента команд сирену
int 21h
;готовимся к переходу в реальный режим
;прерывания запрещены
;перепрограммируем контроллер
mov a1, 08h
call new_8259A
;формирование дескрипторов для реального режима
assume ds:GDT_SEG
mov ax, 8h
mov ds, ax
mov GDT_ds_18.limit, 0ffffh
mov GDT_es_vbf_20.limit, 0ffffh
mov GDT_ss_28.limit, 0ffffh
mov GDT_cs_30.limit, 0ffffh
assume ds:DATA
;загрузка теневых дескрипторов
mov ax, 18h
mov ds, ax
mov ax, 20h
mov es, ax
mov ax, 28h
mov ss, ax
db 0eah
dw offset jump
dw30h
jump: mov eax, cr0
and     al, 0feh
mov cr0, eax
db   0eah
dw   offset r_mode
dw   CODE
r_mode:
```

```
mov ax, DATA
mov ds, ax
mov ax, STK
mov ss, ax
mov ax, 3ffh
mov point_IDT.lim, ax
xor         eax, eax
mov point_IDT.adr, eax
lidt        point_IDT
;разрешаем прерывания
 sti
xor         al, al
out         70h, al
;окончание работы программы (стандартно)
mov ax, 4C00h
int 21h
main ENDP
code_size=$-dummy
code ends
end main
```

ПРИЛОЖЕНИЕ А

Дополнительные сведения

Таблица 1

Назначение полей дескриптора сегмента

Номер байта	Количество бит в поле	Символическое обозначение дескриптора	Назначение и содержание полей в дескрипторе
0...1	16	limit_1	Младшие биты 0...15 20-разрядного поля предела сегмента, который определяет размер сегмента в единицах, определяемых битом гранулярности g
2...4	23	base 1	Биты 0...23 32-разрядной базы сегмента, которая определяет значение линейного адреса начала сегмента в памяти
5	8	AR	Байт, поля которого определяют права доступа к сегменту (табл. 2 приложения А)
6	0..3	limit_2	Старшие биты 16...19 20-разрядного предела сегмента
6	1	U	Бит пользователя (User). Не имеет специального назначения и может использоваться по усмотрению программиста
6	1	–	0 – бит не используется
6	1	D	Бит разрядности операндов и адресов: 0 – в программе используются 16-разрядные операнды и режимы 16-разрядной адресации; 1 – в программе используются 32-разрядные операнды и режимы 32-разрядной адресации
6	1	G	Бит гранулярности: 0 – размер сегмента равен значению в поле limit в байтах; 1 – размер сегмента равен значению в поле limit в страницах
7	8	base 2	Биты 24...31 32-разрядной базы сегмента

Таблица 2

Байт AR дескриптора сегмента

Номер бита в байте AR	Символическое обозначение	Назначение и содержимое
0	A	Бит доступа (Accessed) к сегменту. Устанавливается аппаратно при обращении к сегменту
1	R	Для сегмента кода – это бит доступа по чтению (Readable); определяет, возможно ли чтение из сегмента кода при осуществлении замены префикса сегмента: 0 – чтение из сегмента запрещено; 1 – чтение из сегмента разрешено.
	W	Для сегментов данных — это бит записи: 0 – модификация данных в сегменте запрещена; 1 – модификация данных в сегменте разрешена
2	C	Для сегментов кода — это бит подчинения (Conforming): 1 — подчиненный сегмент кода; 0 — обычный сегмент кода.
	ED	Для многозадачного режима определяет особенности смены значения текущего уровня привилегий. Для сегментов данных — это бит расширения вниз (Expand Down); служит для различения сегментов стека и данных, а также определяет трактовку поля limit: 0 – сегмент данных; 1 – сегмент стека.
3	I	Бит предназначения (Intending): 0 — сегмент данных или стека; 1 – сегмент кода.
4	S	Если S=1 — то это бит сегмента (Segment). Для любых сегментов в памяти равен 1. Назначение и порядок использования сегмента уточняется битами C и R; если – 0 – то это бит системный (System). Такое состояние бита S говорит о том, что данный дескриптор описывает специальный системный объект, который может и не быть сегментом в памяти
5...6	dpl	Поле уровня привилегий сегмента (Descriptor Privilege Level). Содержит численное значение в диапазоне от 0 до 3 привилегированности сегмента, описываемого данным дескриптором
7	P	Бит присутствия (Present): 0 – сегмента нет в оперативной памяти в данный момент; 1 – сегмент находится в оперативной памяти в данный момент.

Таблица 3

Распределение прерываний в защищенном, и реальном режимах

Номер Вектора	Реальный режим	Защищенный режим	Тип прерывания	Код ошибки
0	Деление на ноль	Деление на ноль	Ошибка	Не формируется
1	Пошаговой работы	Пошаговой работы	Ошибка или ловушка	Не формируется
2	Сигнал на входе NMI	Сигнал на входе NMI	Ловушка	Не формируется
3	Контрольная точка	Контрольная точка	Ловушка	Не формируется
4	Переполнение	Переполнение	Ловушка	Не формируется
5	BIOS: печать экрана; нарушение границ массива	Нарушение границ массива	Ошибка	Не формируется
6	Недопустимая команда	Недопустимая команда	Ошибка	Не формируется
7	Обращение к отсутствующему сопроцессору	Обращение к отсутствующему сопроцессору	Ошибка	Не формируется
8	Микросхема таймера/двойная ошибка	Двойная ошибка	Авария	Формируется (всегда 0)
9	Клавиатура	Не используется	-	Формируется
10 (0Ah)	Сигнал на линии IRQ_2	Ошибочный TSS	Ошибка	Формируется
11 (OBh)	Сигнал на линии IRQ_3	Отсутствие сегмента	Ошибка	Формируется
12 (0Ch)	Сигнал на линии IRQ_4/выход за пределы стека	Выход за пределы стека или отсутствие стека	Ошибка	Формируется
13 (0Dh)	Сигнал на линии IRQ_5/нарушение общей защиты	Нарушение общей защиты	Ошибка	Формируется
14 (0Eh)	Контроллер НГМД сигнал на линии IRQ.6	Отсутствие страницы памяти	Ошибка	Формируется
15 (0Fh)	Сигнал на линии IRQ_7	Не используется	—	—
16 (10h)	BIOS: функции видеосистемы/ ошибка сопроцессора	Ошибка сопроцессора	Ошибка	Не формируется
17(llh)	Используется для прерываний BIOS и DOS	Ошибка выравнивания	Ошибка	Формируется (всегда 0)
18-31 (12h-19h)	Используется для прерываний BIOS и DOS	Зарезервировано	—	—
32-255 20h-0ffh)	Используется для прерываний BIOS и DOS	Аппаратные прерывания; прерывания, определяемые пользователем и ОС	Ловушка	Не формируется

Мультипроцессорные и избыточные системы

В данном приложении поясняется использование нескольких (двух или более) процессоров на одной системной шине, а именно описывается мультипроцессорная (SMP – Symmetric Multi-Processing) конфигурация и системы с избыточным контролем функциональности (FRC – Functional Redundancy Checking).

Симметричные многопроцессорные системы

Процессоры Pentium (начиная со второго поколения) имеют специальные интерфейсные средства для построения мультипроцессорных систем (с двумя и более процессорами). Целью объединения процессоров является использование *симметричной мультипроцессорной обработки SMP* (Symmetric Multi-Processing). В симметричной системе SMP каждый процессор выполняет свою задачу, порученную ему операционной системой. Поддержку SMP имеют такие ОС, как Novell NetWare, Microsoft Windows NT и различные диалекты UNIX. Процессоры, объединенные общей локальной шиной, разделяют общие ресурсы компьютера, включая память и внешние устройства. В каждый момент времени шиной может управлять только один процессор, по определенным правилам они меняются ролями.

Поскольку каждый из процессоров имеет свой внутренний первичный кэш, интерфейс обязан поддерживать *согласованность данных* во всех иерархических ступенях оперативной памяти – в первичном и вторичном кэшах и основной памяти (в Pentium вторичный кэш у процессоров общий). Эта задача решается с помощью локальных циклов слежения, воспринимаемых процессором, даже не управляющим шиной в данный момент.

Для обработки *аппаратных прерываний* в многопроцессорных системах традиционные аппаратные средства становятся непригодными, поскольку прежняя схема подачи запроса INTR и передачи вектора в цикле INTA# явно ориентирована на единственность процессора. Для решения этой задачи в процессоры, начиная со второго поколения Pentium, введен расширенный программируемый контроллер прерываний *APIC* (Advanced Programmable Interruption Controller). Этот контроллер имеет внешние сигналы локальных прерываний LINT[1:0] и трехпроводную интерфейсную шину (PICD[1:0] и PICCLK), по которым процессоры связываются с контроллером APIC системной платы. Для локальных запросов прерываний процессоры имеют линии LUNT0, LINT1. Локальные прерывания обслуживаются только тем процессором, на выводы которого поступают сигналы их запросов. Общие (разделяемые) прерывания (в том числе и SMI) приходят к процессорам в виде сообщений по интерфейсу APIC. При этом контроллеры предварительно программируются; тем самым определяются функции каждого из процессоров в случае возникновения того или иного аппаратного прерывания. Контроллеры APIC каждого из процессоров и контроллер системной платы, связанные ин-

терфейсом APIC, выполняют маршрутизацию прерываний (Interrupt Routing), причем, как статическую, так и динамическую. Внешне программный интерфейс обработки прерываний остается совместимым с управлением контроллером 8259A, что обеспечивает прозрачность присутствия APIC для прикладного программного обеспечения.

Симметричные системы имеют специальные механизмы *арбитража* доступа к локальной шине. Процессор – текущий владелец шины – отдаст управление шиной другому процессору по его запросу только по завершении операции. Сблокированные циклы не могут прерываться другим процессором, кроме случая, когда обращение к памяти попадает в область, модифицированный образ которой находится в кэше другого процессора. В таком случае этому процессору отдадут управление для выполнения обратной записи из кэша.

Мультипроцессорные системы в принципе могут использовать процессоры различного степпинга, но частоты ядра у них должны совпадать (шина, естественно, синхронизируется общим сигналом).

Возможности и способы реализации SMP зависят от модели процессора. Заметим, что средства поддержки SMP в семействе x86 имеются только у процессоров Intel.

Интерфейс *Pentium* (начиная со второго поколения) позволяет на одной локальной системной шине устанавливать два процессора, при этом почти все их одноименные выводы просто непосредственно объединяются. Роль конкретного процессора в системе фиксирована – она определяется внешними сигналами во время спада сигнала RESET. Один из процессоров назначается первичным *(Primary)* или загрузочным *(BSP* – Bootstrap Processor), другой – вторичным (DP – Dual Processor). После сигнала RESET сразу начинает функционировать только первичный процессор (BSP), выполняя программный код инициализации. Второй процессор начнет функционирование только после приема соответствующего сообщения по шине APIC, посланного под управлением программы инициализации.

Циклы слежения выполняются по сигналу ADS#, генерируемому другим процессором. Ответами на локальные циклы слежения являются сигналы PHIT# и PHITM#, а роль сигналов HIT# и HITM# остается прежней – они используются во внешних (по отношению к обоим процессорам) циклах слежения, инициируемых сигналами EADS#.

Арбитраж доступа выполняется с помощью “приватных” сигналов запроса (PBREQ#) и подтверждения передачи (PBGNT#) управления локальной шиной. Сигналы обычного системного арбитража (HOLD, HLDA, BOFF#) в двухпроцессорной системе действуют обычным образом, но воспринимаются и управляются поочередно текущим владельцем локальной шины.

Режим обработки прерываний посредством APIC разрешается сигналом APICEN по аппаратному сбросу, впоследствии он может быть запрещен программно.

В *процессорах P6* заложены более развитые возможности SMP. Системная шина P6, в отличие от локальной шины Pentium, изначально ориентирована на разделяемое управление множеством симметричных (до четырех на шине) и несимметричных (до восьми) агентов. Сокет 8 (Pentium Pro) и слот 2 (Pentium II Xeon) позволяют объединять до четырех процессоров, слот 1 (Pentium II) допускает объединение не более двух процессоров. Процессоры Pentium II OverDrive для сокета 8 тоже допускают объединение не более двух процессоров. Фирма Intel объясняет это ограничение большой паразитной индуктивностью контактов сокета (на первый взгляд это неубедительно, поскольку внешняя шина работает на тех же частотах, что и у Pentium Pro). Какой из процессоров станет первичным (BSP), определяется по загрузочному протоколу, – здесь нет жесткой аппаратной привязки роли процессора к его “географическому” адресу. Это позволяет повысить надежность SMP-системы, поскольку любой процессор может без механического вмешательства во время инициализации взять на себя роль BSP. Протокол мультипроцессорной инициализации работает на шине APIC, он позволяет управлять инициализацией до 15 процессоров. Арбитраж процессоров выполняется с помощью сигналов BREQ#[0:3] согласно общему протоколу. Процессоры могут пользоваться содержимым "чужого" кэша без его предварительной выгрузки в основную память.

Системы с избыточным контролем функциональности

В *конфигурации с избыточным контролем функциональности FRC* (Functional Redundancy Checking) два процессора (пара Master/Checker) выступают как один логический. Основной процессор (Master) работает в обычном однопроцессорном режиме. Проверочный процессор (Checker) выполняет все те же операции “про себя”, не управляя шиной, и сравнивает выходные сигналы основного (проверяемого) процессора с теми сигналами, которые он генерирует сам, выполняя те же операции без выхода на шину. В случае обнаружения расхождения вырабатывается сигнал ошибки (IERR# в Pentium, FRCERR в P6), который может обрабатываться как прерывание.

Поддержка FRC появилась, начиная с процессоров Pentium (но в Pentium MMX ее нет); она имеется и у процессоров фирмы AMD.

Ошибки процессоров Pentium

Процессоры Pentium, как и любые сложные изделия, кроме штатных свойств имеют недокументированные возможности, а также, к сожалению, ошибки и опечатки. Фирма Intel публикует перечни обнаруженных отклонений своих изделий от заявленных спецификаций (Erratum) на Web-сервере. Эти ошибки учитываются разработчиками ПО и системных плат и в большей части нейтрализуются средствами ОС и BIOS "в рабочем порядке". Тем не менее некоторые ошибки получили широкую огласку.

Первая из серьезных обнаруженных ошибок относилась к FPU процессора *Pentium* первого поколения и известна под названием "Floating point flaw". Эта ошибка выражалась в потере точности при выполнении деления при некотором сочетании операндов. Ошибка могла появляться от 4 до 19 разряда после десятичной точки. С начала 1995 года процессоры выпускались уже без ошибок. Статистические исследования показывают, что ошибка может проявляться раз в несколько лет. Тем не менее фирма Intel до сих пор обеспечивает бесплатную замену уже проданных процессоров с ошибкой на исправленные версии (но без апгрейда на более современные модели). Процессоры с ошибкой выявляются утилитой CPUIDF.EXE, которую можно получить на Web-сервере компании.

Весной 1997 года в процессорах *Pentium Pro* и *Pentium II* была обнаружена ошибка формирования флагов FPU при преобразовании форматов больших отрицательных чисел, в результате чего поведение FPU в некоторых случаях не подчиняется требованиям стандарта IEEE для обработки чисел с плавающей точкой. Этой ошибке, называемой в прессе "Dan-0411" по имени ее первооткрывателя и дате обнаружения, фирма Intel дала официальное название "Flag Erratum".

Осенью 1997 года обнаружилась более серьезная ошибка, которая поначалу поставила под сомнение возможность применения процессоров Pentium (включая MMX) в многопользовательских системах. Суть ошибки заключается в том, что любая задача может остановить процессор, независимо от уровня привилегий, что противоречит принципам защиты, провозглашенным для 32-разрядных процессоров. Штатным образом процессор останавливается по инструкции *HALT,* которая в защищенном режиме является привилегированной и не исполняется на уровне привилегий пользователя. Однако к блокировке процессора, оказывается, приводит и исполнение недопустимой инструкции, задаваемой последовательностью кодов F0 0F C7 C8. Этот код расшифровывается как инструкция *LOCK: CMPXCHG8B EAX* – инструкция некорректна, поскольку пытается сравнить 32-битный операнд с 64-битным. Тем не менее она доступна на любом уровне привилегий. Последний байт инструкции может принимать любое значение в диапазоне C8-CF, что будет соответствовать другим регистрам. При исполнении этой инструкции во вре-

мя вырабатывания исключения неверного кода инструкции (#UD) процессор блокируется. Таким образом, любой пользователь многопользовательской системы может умышленно остановить компьютер-сервер. Эта ошибка, названная “Pentium F0 bug”, имеется во всех процессорах Pentium и Pentium MMX. В процессорах Pentium Pro и Pentium II ее нет, как нет и в процессорах пятого и шестого поколений от AMD и Cyrix. Впоследствии оказалось, что все не так трагично и есть способы предотвращения блокировки на уровне ОС. Фирма Intel предложила “вышибать клин клином” – опережать блокировку вызовом исключения *#PF*. Для этого таблицу дескрипторов прерываний (IDT) выравнивают относительно границы 4-килобайтных страниц так, что ее первые 56 байт (обработчики исключений 0-6, включая *#UD)* находятся в одной странице, а остальные – в другой. Эту первую страницу не включают в системную область памяти ОС, так что каждое обращение к ней будет вызывать исключение отказа страницы (#PF). В обработчик этого исключения включается перехватчик, который по адресу отказа должен отлавливать исключение *#UD,* по контексту проверять его на наличие приведенного выше недопустимого кода и выполнять нейтрализующие действия. Другой способ вызова *#PF* до *#UD* заключается в том, чтобы для страницы с обработчиком *#UD* установить *PTE.W=0* и *CR0.WP*=1 (запрет записи на уровне пользователя, что допустимо не для всех ОС). В любом из этих двух способов обработке исключений (и прерываний) 0-6 будет предшествовать обработка исключения *#PF,* что по многим причинам весьма неудобно. Однако есть безобидный способ борьбы с F0 bug. Обнаружено, что если обработчик исключения *#UD* находится в некэшируемой области памяти, блокировки не происходит. Используя тот же принцип выравнивания границы *IDT,* первая ее страница объявляется некэшируемой (установкой *PTE.PCD=1)* или не допускающей обратной записи *(PTE.PWT=1).* При этом аномалия поведения процессора не приводит к его блокировке и никаких модификаций обработчиков исключения не требуется.

"Заплатки" на ошибку F0 bug имеются для всех ОС защищенного режима. Для ОС реального режима, где страничное управление и обработка отказа страниц не применяются, единственным способом борьбы с ошибкой является запрет первичного кэша опциями BIOS Setup.

ПРИЛОЖЕНИЕ Г

Список сокращений и имен регистров, структур данных и флагов

ОЗУ	Оперативное запоминающее устройство (RAM)
ОС	Операционная система (OS)
ПЗУ	Постоянное запоминающее устройство (ROM)
ПК	Персональный компьютер (PC)
ПО	Программное обеспечение (Software)
#AC	Alignment Check, исключение контроля выравнивания (вектор 17)
#BP	Breakpoint, прерывание отладки *INT 3* (вектор 3)
#BR	BOUND Range Exceeded, прерывание по контролю диапазона *BOUND* (вектор 5)
#DB	Debug, исключение отладки (вектор 1)
#DE	Divide Error, исключение деления на 0 (вектор 0)
#DF	Double Fault, исключение “двойной отказ” (вектор 8)
#GP	General Protection, общее исключение защиты (вектор 13)
#MC	Machine Check, исключение машинного контроля (вектор 18)
#MF	Math Fault, исключение сопроцессора (вектор 16)
#NM	No Math Coprocessor, исключение “сопроцессор недоступен” (вектор 7)
#NP	Segment Not Present, исключение отсутствия сегмента (вектор 11)
#OF	Overflow, прерывание по переполнению *INTO* (вектор 4)
#PF	Page Fault, исключение по отказу страницы (вектор 14)
#SS	Stack-Segment Fault, исключение по сегменту стека (вектор 12)
#TS	Invalid TSS, исключение по недопустимому TSS (вектор 10)
#UD	Undefined Opcode, исключение по недопустимому коду операции (вектор 6)
APIC	Advanced Peripheral Interrupt Controller, усовершенствованный контроллер прерываний
ASCII	American Standard Code for Information Interchange, американский стандартный код обмена информацией
AT	Advanced Technology, передовая технология; PC с процессорами 80286 и выше
B	Big, флаг разрядности сегмента стека (в дескрипторе)
BASE	Поле базового адреса сегмента, 32 бит (в дескрипторе)
BCD	Binary Coded Decimal, двоично-десятичный код
BEDO DRAM	Burst EDO, динамическая память с фиксацией данных в выходном регистре и внутренним счетчиком адреса для пакетного цикла
C	Conforming, флаг конформности (подчиненности) сегмента (в дескрипторе)
CD	Cache Disable, флаг запрета кэширования (CR0.30)

CF Carry Flag, флаг переноса (заема) *(EFLAGS.O)*

CPL Current Privilege Level, поле текущего уровня привилегии (CS[1:0])

CPU Central Processor Unit (центральный процессор)

CR0 Управляющий регистр, 32 бит

CR1 Управляющий регистр, не используется

CR2 Регистр линейного адреса отказа страницы, 32 бит

CR3 Регистр базового адреса каталога страниц, 32 бит

CR4 Управляющий регистр, 32 бит

DPL Descriptor Privilege Level, поле уровня привилегии сегмента (в дескрипторе)

DR0… 3 Регистры линейного адреса точек останова

DR4, 5 Регистры отладки, не используются

DR6 Регистр состояния контрольных точек

DR7 Регистр управления отладкой

GDT Global Descriptor Table, глобальная таблица дескрипторов

GDTR Регистр глобальной таблицы дескрипторов, 48 бит

GS Регистр селектора сегмента данных 16 бит

ID Id Flag, флаг доступности инструкций *CPUID* (*FLAGS.21*)

IDT Interrupt Descriptor Table, таблица дескрипторов прерываний

IDTR Interrupt Descriptor Table Register, регистр таблицы дескрипторов прерываний 48бит

INDEX Поле выбора селектора (*Seg*[15:3])

IP Регистр указателя инструкции

LDT Local Descriptor Table, Локальная таблица дескрипторов

LDTR Регистр селектора локальной таблицы дескрипторов, 16 бит

LIMIT Поле лимита сегмента 20 бит (в дескрипторе)

NT Nesterd Task Flag, флаг вложенности задач

PS Page Size, флаг размера страницы

PSE Page Size Extension, флаг расширения размера страницы (*CR*4.4)

PTE Page Table Entry, строка таблицы страниц

SMM System Management Mode, режим системного управления

SMP Symmetrical Multiprocessing, симметричная мультипроцессорная обработка

SMRAM System Management RAM, память, доступная в режиме SMM

SP Регистр указателя стека (младшие 16 бит *ESP)*

SPGA Staggered PGA, корпус ИС с шахматным расположением выводов

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Юров В. Assembler: учебный курс. – СПб: Питер Ком, 2002.
2. Pentium ® II Processor Developer's Manual, 1997.
3. P6 Family of Processors Hardware Developer's Manual. Intel Corporation, 1998.
4. Intel Architecture Software Developer's Manual, Vol. 1: Basic Architecture, Vol.2: Instraction Set Reference, Vol.3: System Programming Guide. Intel Corporation, 1997
5. Брам П., Брам Д. Микропроцессор 80386 и его программирование. – М.: Мир, 1990
6. Гук М. Процессоры Intel: от 8086 до Pentium II. – СПб: Питер, 1997
7. Гук М. Аппаратные средства IBM PC. Энциклопедия. – СПб: Питер Ком, 1998
8. Гук М. Процессоры Pentium II, Pentium Pro и просто Pentium. – СПб: Питер Ком, 1999.

Интернет
http://www.intel.ru
http://www.sysdoc.pair.com

СОДЕРЖАНИЕ

УЧЕБНОЕ ИЗДАНИЕ

Солодовникова Ольга Сергеевна

ЗАЩИЩЕННЫЙ РЕЖИМ МИКРОПРОЦЕССОРОВ PENTIUM

Пособие для самостоятельной работы

Для специальности: 220200 – Автоматизированные системы обработки информации и управления

Редактор ***Л.П. Кербиева***
Компьютерная верстка ***Е Х. Гергоковой***
Корректор ***Е.Г. Скачкова***

Изд. лиц. Серия ИД 06202 от 01.11.2001.
В печать 21.11.2003. Формат 60x84 1/16. Печать трафаретная.
Бумага газетная. 5.35 усл.п.л. 4.0 уч.-изд.л.
Тираж 250 экз. Заказ № _____ .
Кабардино-Балкарский государственный университет.
360004, г. Нальчик, ул. Чернышевского, 173.

Полиграфическое подразделение КБГУ.
360004, г. Нальчик, ул. Чернышевского, 173.