Министерство образования и науки Российской Федерации

САНКТ–ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

С. А. МОЛОДЯКОВ

ЭВМ И ПЕРИФЕРИЙНЫЕ УСТРОЙСТВА

Часть I Основы организации ЭВМ

Учебное пособие

Санкт-Петербург

2012

УДК 004.2: 004.35: 004.431.4

Молодяков С.А. ЭВМ и периферийные устройства. Часть I. Основы организации ЭВМ. Учебное пособие. СПб.: СПбГПУ, 2012.- 367 с.

В учебном пособии представлен материал лекций, которые читаются по дисциплине «ЭВМ и периферийные устройства» на кафедре информационных и управляющих систем СПбГПУ. В первой части «Основы организации ЭВМ» рассмотрены вопросы, связанные с базовой организацией ЭВМ, а также ее программирования на языке Ассемблера.

Представлены задания к лабораторным работам, которые проводятся на персональных компьютерах Pentium с использованием языков программирования Ассемблер и С++.

Учебное пособие предназначено для студентов, изучающих дисциплины «Архитектура ЭВМ», «ЭВМ и периферийные устройства» по специальностям 231000 Программная инженерия, 230100 Информатика и вычислительная техника.

© Молодяков С. А., 2012

© Санкт-Петербургский государственный политехнический университет

Оглавление

ГЛАВА 1. БАЗОВАЯ ОРГАНИЗАЦИЯ ЭВМ 5
1.1. Многоуровневая организация и эволюция ЭВМ. 5
1.2. Архитектурные принципы Фон-Неймана 10
1.3. Магистральная вычислительная структура........ 14
1.4. Процессор. Основные компоненты и их назначение. 17
1.5. Программная модель процессора Pentium. 20
1.6. Оперативная память (общая характеристика) 23
1.7. Типовая структура машинной команды в пространстве и во времени. Формат и конвейер команд........ 26
ГЛАВА 2. ВИДЫ ДАННЫХ, ИХ КОДИРОВАНИЕ, КОМАНДЫ........ 35
2.1. Числа и системы счисления 37
2.2. Представление и команды над целыми числами 39
2.3. Форматы чисел плавающей запятой, команды 50
2.4. Кодирование символов 57
2.5. Графические данные, их представление и кодирование 65
ГЛАВА 3. ПРОЦЕССОР 86
3.1. Процессор – аппаратный уровень. Операционные устройства 86
3.2. Устройство управления. Микропрограммный автомат. 95
3.3. Архитектуры систем команд........ 102
3.4. Ассемблер и система команд процессора на примере процессора Pentium. 111
3.5. Способы адресации........ 122
3.6. Управление вычислительным процессом 135
3.7. Кодирование команд в процессоре x86........ 149
ГЛАВА 4. ПАМЯТЬ. НИЖНИЙ УРОВЕНЬ 153
4.1. Методы доступа 153
4.2. Иерархия запоминающих устройств........ 155
4.3. Основная память. ОЗУ 159
4.4. Микросхемы памяти 162
4.5. Регенерация памяти 174
4.6. Обнаружение и исправление ошибок........ 176
4.7. Флэш-память........ 179
4.7. Кэш-память 180
4.8. Многоуровневая кэш-память и пакетный режим передачи данных 194
ГЛАВА 5. ПАМЯТЬ. ВЕРХНИЙ УРОВЕНЬ 198
5.1. Динамическое распределение памяти 198
5.2. Виртуальная память 199

5.3. Общие принципы защиты памяти ... 205
5.4. Мультизадачность ... 207
5.5. Дисковые массивы и уровни RAID ... 214
5.6. Организация памяти в процессорах Pentium ... 221
5.6.1. Трансляция адреса в реальном режиме ... 222
5.6.2. Трансляция адреса в защищенном режиме ... 223
5.6.3. Страничный механизм ... 233
5.6.4. Защита в процессоре Pentium ... 237
5.6.5. Аппаратная поддержка мультизадачности ... 245
5.6.6. Прерывания в защищенном режиме ... 250
5.7. Системные регистры процессоров Pentium ... 253
ГЛАВА 6. ВВОД-ВЫВОД ... 259
6.1. Подключение периферийных устройств к ЭВМ ... 259
6.2. Синхронизация выполнения программы с внешними процессами ... 274
6.3. Прерывания ... 277
6.4. Реализация механизма прерывания в процессорах Pentium ... 288
6.5. Аппаратная поддержка отладки ... 292
6.6. Прямой доступ к памяти ... 296
6.7. Шины. PCI. PCI Express ... 301
6.8. Процесс загрузки компьютера ... 313
Лабораторные работы ... 318
Лабораторная работа по теме №1. Введение в низкоуровневое программирование. Встроенный отладчик IDE. Встроенный Ассемблер. TurboDebugger ... 318
Лабораторная работа по теме N2 Система команд процессора XX86, ее связь с кодами команд. ... 322
Лабораторная работа по теме N3 Способы адресации и сегментная организация памяти ... 328
Лабораторная работа по теме N4 Подпрограммы и передача параметров ... 331
Лабораторная работа по теме N5 Подпрограммы, программные прерывания и особые случаи. ... 339
Лабораторная работа по теме N6 FPU. Кодирование чисел с плавающей запятой. Особые численные значения. Особые случаи. ... 343
Лабораторная работа по теме N7 Обмен ЭВМ с клавиатурой ... 351
Лабораторная работа по теме N8 Мультизадачность ... 359
Список контрольных вопросов ... 364
Рекомендуемая литература ... 367

ГЛАВА 1. БАЗОВАЯ ОРГАНИЗАЦИЯ ЭВМ

1.1. Многоуровневая организация и эволюция ЭВМ.

Изучение вычислительной техники и программирования немыслимо без рассмотрения организации ЭВМ. В данном курсе предполагается изучение, как основ функционирования, так и особенностей современных процессоров и периферийных устройств во взаимосвязи с обрабатываемыми видами данных. Рассмотрение проводится как на уровне "модели процессора для программиста", так и на уровне компонентов структурной иерархии, которые не являются впрямую программно-доступными. Рассмотрение аппаратуры неотделимо от программного обеспечения, так как граница между ними постоянно перемещается. Сегодняшнее программное обеспечение может быть завтрашним аппаратным обеспечением и наоборот. Не важно, на каком уровне выполняется команда. Программист, работающий на уровне архитектуры системы, может использовать команду умножения, как будто это команда аппаратного обеспечения, и даже не задумываясь об этом. То, что для одного человека - программное обеспечение, для другого аппаратное.

Развитие компьютерных технологий не останавливается, появляются новые идеи, часть старых забывается. В данном курсе рассматриваются основополагающие подходы к созданию высокопроизводительных процессоров, систем и ЭВМ, делаются непрерывные ссылки на наиболее распространенный в мире компьютер IBM PC.

Многоуровневая организация средств цифровой вычислительной техники [10]

Цифровые вычислительные системы имеют многоуровневую иерархическую организацию, которая проявляется в различных аспектах. В частности, чем ниже работаем - тем больший уровень эффективности доступен, но тем выше трудозатраты при разработке.

Устройство нижележащих уровней может быть скрыто от пользователя, и его для получения практического результата (например, работающего программного продукта) знать не обязательно. Но знание и использование свойств нижележащих уровней позволяет в том или ином смысле повысить эффективность разрабатываемой программы или устройства.

Уровень	Характеристика, примечание
проблемно-ориентированные языки	использование готовой прикладной программы (например, MS Word)
процедурно ориент. языки	программирование прикладной задачи
уровень языка макроассемблера	Позволяет поименовать часто повторяющиеся фрагменты кода и в дальнейшем использовать это имя
уровень функций ОС (BIOS)	Можем пользоваться готовыми типовыми подпрограммами для часто используемых или аппаратно зависимых действий
уровень машинных команд	Если пишем свои аналоги системных функций
микропрограммный уровень	Программно невидим, но знание его свойств позволяет улучшать эффективность кода
уровень аппаратуры	Доступен во встроенных применениях

При написании оптимизирующих трансляторов, при системном программировании, при разработке приложений, критичных к скорости выполнения, иногда приходится учитывать свойства иерархических уровней, вплоть до аппаратного, для достижения более высокой эффективности. Например, при оптимизации по скорости в процессорах с многопотоковыми конвейерами, надо учитывать времена выполнения команд и возможное влияние соседних команд друг на друга.

Сфера нашего рассмотрения - первые 5 уровней.

Эволюция развития ЭВМ

Разные авторы по-разному делят на этапы эволюцию вычислительной техники (ВТ). Мы отметим только ключевые этапы развития электронных вычислительных машин (ЭВМ).

- Нулевое поколение **–** механическая эра**.**

Первые счеты – абак в древнем Вавилоне 3000 лет до н.э.

Счеты с косточками на проволоке – Китай 500 лет до н.э.

1492 г. Леонардо да Винчи приводит рисунок тринадцатиразрядного десятичного сумматора на основе **зубчатых колес**.

1642 г. Блез Паскаль изготовил более 10 вычислителей, который суммировали и вычитали пятиразрядные десятичные числа.

1673 г. Г.В.Лейбниц создает вычислитель всех четырех операций над 12-ными десятичными числами. Результат умножения – 16 цифр.

1863 г. Чарльз Бэббидж механическая машина имела считыватель с перфокарт для ввода программ и данных, память (склад) имела объем пятьдесят 40-разрядных чисел и два аккумулятора для хранения промежуточных результатов. Имелись условные переходы. Суммирование занимает 3 с, а умножение 2-4 минуты. Позднее создает принтер.

1885 г. Дорр Фельт – первый калькулятор, в котором числа вводятся нажатием клавиш.

1937 г. Алан Тьюринг публикует статью, в которой излагает концепцию упрощенной вычислительной машины, получившей название машины Тьюринга.

1938 г. Конрад Цузе – механический программируемый вычислитель Z1 с памятью на 1000 бит. Позднее Z3 с программой на перфоленте. Умнож. 5с.

- Первое поколение – электронные лампы (1945-1955). Типичным представителем является ЭВМ ЭНИАК, которая описана ниже. Это поколение аккумуляторных ЭВМ с небольшой памятью (4 Кбайт).

- Второе поколение – транзисторы (1955-1965). Изобретение транзистора произошло в лаборатории Bell Laboratories, за что получена Нобелевская премия в 1956 г. Появились новые архитектурные элементы: блок обработки чисел с плавающей запятой, общая шина, память на магнитных сердечниках, суперкомпьютеры (Сеймур Крей, суперЭВМ CDC6600 имела быстродействие 1 MFLOPS). Широко распространился отечественный компьютер Минск-32 и миникомпьютер Электроника-100.
- Третье поколение – интегральные схемы (1965-1980). Изобретение кремниевой интегральной схемы в 1958 г. Робертом Нойсом. Архитектурные новшества: конвейерная и параллельная обработка, микропрограммирование, кэш-память, первые операционные системы. Во всем мире широко распространился компьютер IBM-360, который имел следующие характеристики (модель 50): время цикла 500 нс, объем памяти 256 Кбайт, за одно обращение к ОЗУ выбирал 4 байта..
- Четвертое поколение – сверхбольшие интегральные схемы (1980-). Large-scale integration LSI - до 1000 транзисторов на кристалле. Very large-scale integration VLSI - до 100 000 транзисторов на кристалле. Появился микропроцессор – ЭВМ на кристалле, а затем и персональные ЭВМ Intel и Apple. В 1981 году появился первый персональный компьютер IBM PC. Основная память стала полупроводниковой. Идея ЭВМ с сокращенным набором команд – RISC идеология. Язык программирования С.
- Пятое и шестое поколение – мультипроцессорные системы (1990-)

Закон Мура: Число транзисторов на микросхеме удваивается каждые 18 месяцев.

Гордон Мур (Gordon Moore) основал компанию Intel вместе с Робертом Нойсом (Robert Noyce) в 1968 году. Сначала Гордон Мур занимал должность исполнительного вице-президента корпорации. В 1975 г. он стал президентом и главным управляющим Intel и оставался генеральным

директором с 1975 по 1987 год. В 1997 г. ему было присвоено звание почетного председателя совета директоров.

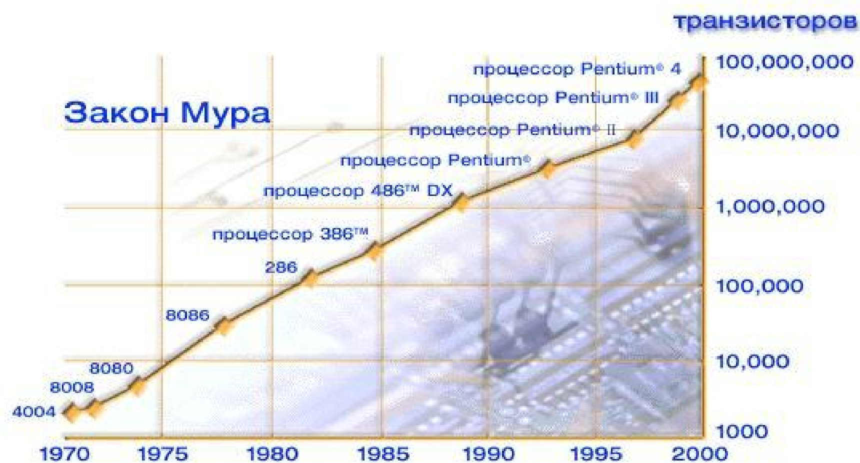


Рис. 1.1. Закон Мура. Развитие процессоров Intel.

Свой закон Мур сформулировал в 1965 году, работая над статьей для Electronics magazine (тогда он был сотрудником Fairchild Semiconductor). В то время закон гласил, что число транзисторов на кристалле удваивается каждый год. В 1975 году Мур внес поправку: ...удваивается каждые два года. В 1965 году, в процессе подготовки выступления, Гордон Мур сделал знаменательное наблюдение. Представив в виде графика рост производительности запоминающих микросхем, он обнаружил любопытную закономерность: новые модели микросхем разрабатывались спустя одинаковые периоды - 18-24 месяца - после появления их предшественников, а емкость их при этом возрастала каждый раз примерно вдвое. Если такая тенденция продолжится, заключил Мур, то мощность вычислительных устройств экспоненциально возрастет на протяжении относительно короткого промежутка времени.

Закон постоянно действует и многие считают, что это будет происходить, возможно, до 2020 года. В подтверждении приведем график (рис. 1.1.).

Другие компьютерные «законы»:

Закон Рока. Это всего лишь маленькое дополнение к закону Мура: "стоимость основных фондов, используемых в производстве полупроводников, удваивается каждые четыре года".

Закон Макрона. Закон Мура лежит в основе закона Макрона, действующего уже на протяжении долгих лет. Закон это гласит: "машина, которая бы вас полностью устроила, никак не может стоить меньше $5000".

Первый натановский закон. (Натан Мирвольд главный администратор компании Microsoft). Программное обеспечение – это газ. Оно распространяется и полностью заполняет резервуар, в котором находится.

1.2. Архитектурные принципы Фон-Неймана

В каждой области науки и техники существуют некоторые фундаментальные идеи или принципы, которые определяют ее содержание и развитие. В компьютерной науке роль таких фундаментальных идей сыграли принципы, сформулированные независимо друг от друга - американским математиком и физиком Джоном фон Нейманом и советским ученым Сергеем Лебедевым. (Джон фон Нейман (1903-1957), Сергей Лебедев (1902-1974).) Эти принципы определяют основы организации компьютера.

Считается, что первый электронный компьютер ЭНИАК был изготовлен в США в 1946г. (смотри линейку развития компьютеров) ЭНИАК состоял из 18000 электронных ламп и 1500 реле и весил 30 тонн. Он имел 20 регистров, каждый из которых мог содержать 10-разрядное десятичное число. Блестящий анализ сильных и слабых сторон проекта ЭНИАК был дан в отчете Принстонского института перспективных исследований "Предварительное обсуждение логического конструирования электронного вычислительного устройства" (июнь 1946 г.). Этот отчет, составленный выдающимся американским математиком Джоном фон Нейманом и его коллегами по Принстонскому институту Г. Голдстайном и А. Берксом, представлял проект нового электронного компьютера. Идеи, высказанные в этом отчете, известны под названием "Неймановских Принципов".

Говоря об основоположниках теоретической информатики, нельзя не упомянуть о двух научных достижениях, алгебре логики и теории алгоритмов. Алгебра логики была разработана в середине 19-го века английским математиком Джорджем Булем и рассматривалась им в качестве метода математизации формальной логики. Разработка электронных компьютеров на двухпозиционных электронных элементах создала возможным широкое использование "булевой логики" для проектирования компьютерных схем. В первой половине 30-х годов 20-го столетия появились математические работы, в которых была доказана принципиальная возможность решения с помощью автоматов любой проблемы, поддающейся алгоритмической обработке. Данное доказательство содержалось в опубликованных в 1936 г. работах английского математика А. Тьюринга и американского математика Э. Поста. (Джордж Буль (1815-1864) Алан Тьюринг (1912-1954))

В Советском Союзе работы по созданию электронных компьютеров были начаты несколько позже. Первый советский электронный компьютер был изготовлен в Киеве в 1953г. Он назывался МЭСМ (малая электронная счетная машина), а его главным конструктором был академик Сергей Лебедев, автор проектов компьютеров серии БЭСМ (большая электронная счетная машина). В проекте МЭСМ Сергей Лебедев независимо от Неймана пришел к тем же идеям конструирования электронных компьютеров, что и Нейман.

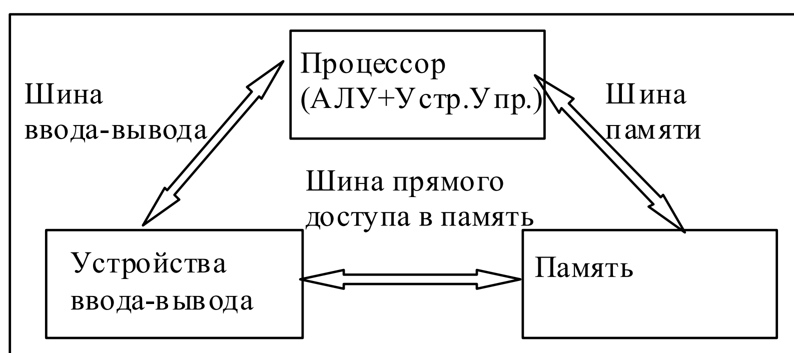


Рис. 1.2. Структурная схема типовой ЭВМ.

Сущность "Неймановских Принципов" состояла в следующем:

1. Компьютер включает связанные между собой Процессор (арифметическое устройство и устройство управления), Память и Устройства ввода-вывода (рис. 1.2).
2. Компьютеры на электронных элементах должны работать не в десятичной, а в двоичной системе счисления.
3. Программа, так же как и числа, с которыми оперирует компьютер, записываются в двоичном коде, то есть по форме представления команды и числа однотипны.
4. Программа должна размещаться в одном из блоков компьютера - в запоминающем устройстве, которое имеет произвольный доступ. Программа и данные могут находиться в общей памяти (принстонская архитектура).
5. Трудности физической реализации запоминающего устройства большого быстродействия, энергонезависимого и большой памяти требуют иерархической организации памяти. Программа выполняется из основной памяти, а сохраняется в энергонезависимой вторичной памяти (магнитных дисках). Файл – идентификационная совокупность экземпляров полностью описанного в конкретной программе типа данных, находящихся вне программы во внешней памяти и доступных программе посредством специальных операций (согласно стандарту ISO).
6. Арифметико-логическое устройство (АЛУ) компьютера конструируется на основе схем, выполняющих операцию сложения, сдвига, логическую операцию. Помимо результата операции АЛУ формирует ряд признаков результата (флагов), которые могут анализироваться при выполнении команд условной передачи управления.

7. В компьютере используется параллельный принцип организации вычислительного процесса (операции над двоичными кодами осуществляются одновременно над всеми разрядами).
8. Централизованное последовательное управление при выполнении команд. Нет конвейера, параллельности, внеочередного выполнения и прочего, что свойственно современным процессорам.
9. Линейная структура адресации памяти.
10. Низкий уровень машинного языка. Нет микропрограммируемости.

Существенно подчеркнуть, что центральное место среди "принципов Неймана " занимает предложение об использовании двоичной системы счисления, что было обусловлено рядом обстоятельств. Во-первых, несомненными арифметическими достоинствами двоичной системы счисления, ее "оптимальным" согласованием с "булевой" логикой и простотой технической реализации двоичного элемента памяти (триггера).

Выполнение команд по программе, хранимой в ЭВМ.

Программа представляет собой последовательность команд, хранимых в памяти компьютера. Команды в ЭВМ (в машине фон Неймана) располагаются в ячейках программной памяти подряд, одна за другой. Процессор по порядку считывает команды из памяти и выполняет их. Этот процесс сводится к последовательному выполнению этапов: считывание команды из памяти, дешифрация команды, обращение к памяти за операндом, выборка операнда, исполнение операции, запись результата в память. Операнд или результат могут быть взяты/записаны в устройства ввода-вывода. Следующая команда выполняется после завершения предыдущей команды. Ее адрес содержится в особом регистре процессора, называемом указателем (или счетчиком) команд. После считывания очередной команды процессор автоматически увеличивает содержимое счетчика команд, так, что он указывает на очередную команду.

Известны альтернативные пути построения компьютера:

- Потоковая машина – действиями управляют сами данные.
- Нейронные сети и др.

1.3. Магистральная вычислительная структура

Принципы фон Неймана применимы и к магистральной или шинной архитектуре. В этом случае ЭВМ включает четвертый элемент – магистраль, которая связывает отдельные элементы и влияет на выполнение команд.

Магистраль/канал – унифицированная подсистема связи структурных частей ЭВМ. Унификация магистрали состоит в том, что все устройства подключаются к магистрали одинаково (используют один и тот же набор сигналов, один и тот же алгоритм обмена). Унификация позволяет легко заменять, добавлять или удалять отдельные части, входящие в состав ЭВМ, без нарушения ее работоспособности. Основные конструктивные компоненты магистрали – линии связи (провода), которые можно подразделить на три группы (шины) – *адреса, данных* и *управления*.

Линии связи – провод, по которому передается логический сигнал.

Шина – группа линий однотипных сигналов.

Шина адреса предназначена для передачи из процессора в память параллельным кодом двоичного слова, представляющего собой начальный адрес участка памяти, к которому требуется обращение. Количество линий (ширина шины адреса) определяет *размер физического адресного пространства*, т.е. максимальное количество различных адресов в ОЗУ. Адрес по шине передается от процессора в память или во внешнее устройство. В процессор адрес передается только в мультипроцессорных системах, для поддержания правильной работы КЭШей.

Шина данных предназначена для передачи команд и данных между процессором, памятью и периферийными устройствами. Передача слов осуществляется также параллельным кодом, а «ширина» шины данных в реальных системах может составлять от 1 до 4 и более байтов. Шина данных является двунаправленной и имеет наибольшую пропускную способность.

В некоторых ЭВМ шина адреса и шина данных объединены в одну мультиплексируемую шину адреса/данных. Такая шина функционирует в режиме разделения времени: цикл шины разбит на временной интервал передачи адреса и на временной интервал передачи данных. Мультиплексирование позволяет сократить общее число линий, но требует усложнения логики связи с шиной. Кроме того, оно может привести к потере производительности.

Шина управления предназначена для передачи управляющих сигналов из процессора в прочие устройства, подключенные к магистрали.

Любое устройство, подключенное к магистрали, должно быть способно:

- а) распознать «свой адрес», формируемый процессором на адресной шине;

- б) распознать по сигналам на шине управления действие, которого ждет от устройства процессор;

- в) выполнить это действие: передать в процессор либо принять из процессора через шину данных двоичное слово.

Последовательность трех перечисленных шагов составляет «цикл магистрали» («**канальный цикл**»). Канальные циклы могут следовать на магистрали непрерывно, либо с интервалами. Они происходят под управлением процессора или внешних устройств, и, таким образом, обеспечивают обмен информацией между частями ЭВМ.

Канальный цикл обмена данными.

Типовая структура (временная диаграмма) канального **цикла обмена данными** изображена на рис.1.3. В ней можно выделить два этапа - этап передачи/декодирования адреса – фазу адреса (задается положением во времени строба адреса) и этап передачи данных – фазу данных (задается положением строба данных).

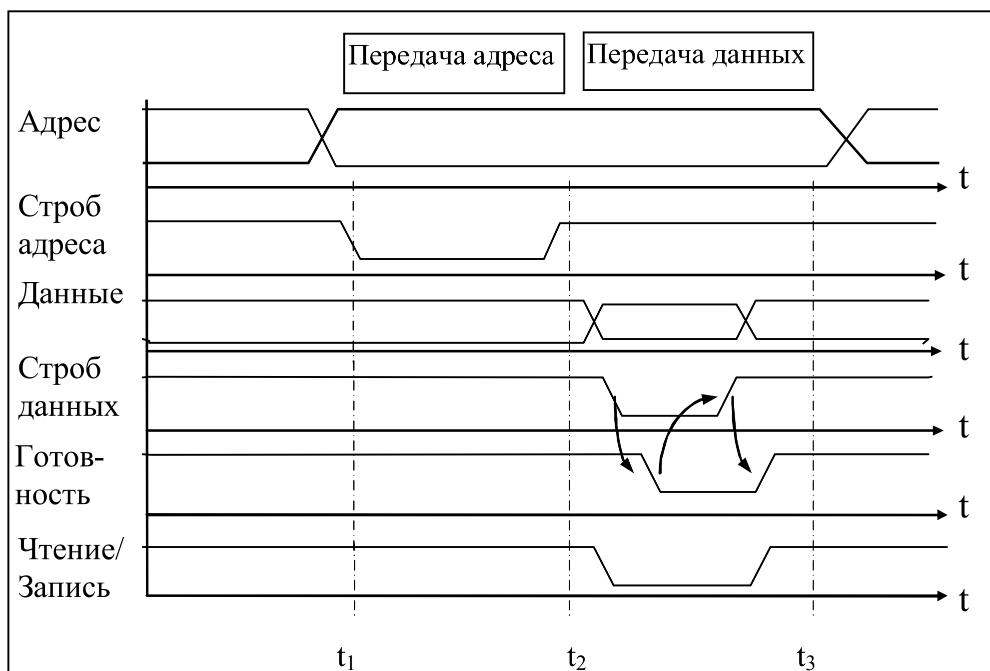


Рис. 1.3. Типовая временная диаграмма канального цикла
для раздельных шин адреса и данных.

Синхронный канал - время канального цикла постоянно, адрес/данные записываются синхронно со стробом. Обычно переключение сигналов шины происходит по импульсам тактовой частоты.

Асинхронный канал - время канального цикла зависит от приемного устройства, которое после записи данных вырабатывает сигнал Готовности.

В приведенном примере шины адреса и данных - раздельные, адрес передается синхронно, в то время как передача данных производится асинхронно - длительность этапа передачи данных определяется задержкой снятия сигнала Готовности.

Арбитраж шин. При любой передаче по шине всегда имеется инициатор передачи или ведущее устройство и имеется приемник данных или ведомое устройство. Ведущее устройство захватывает шину и начинает формировать временную диаграмму ее работы. Если несколько устройств претендуют на права ведущего, то должен быть механизм выбора одного из них. Этот механизм и называется арбитражем шин. Арбитраж построен на основе нескольких положений:

- Каждому из претендентов на шину присваивается определенный уровень приоритета, который может оставаться неизменным (статический приоритет) или изменяться по какому-то алгоритму (динамический приоритет). Шина отдается устройству с наивысшим приоритетом.
- Может использоваться централизованная схема арбитража или децентрализованная. Центральный арбитр, находящийся, например, в контроллере шины, принимает запросы параллельно или последовательно от устройств и решает задачу предоставления шины. В децентрализованной схеме каждый ведущий может иметь контроллер шины и самостоятельно принимать решение о ее захвате.
- Ограничение времени управления шиной. Вне зависимости от модели арбитража должно быть предусмотрено ограничение времени управления шиной. Может использоваться несколько вариантов. Например, алгоритм фиксированного кванта времени, который отводится каждому ведущему для захвата шины.

1.4. Процессор. Основные компоненты и их назначение.

(Central Processor Unit, CPU)

Осуществляет основные действия по выполнению команд. В нем можно выделить несколько составляющих частей:

1) декодер команд,

2) арифметико-логическое устройство АЛУ, выполняющее действия над операндами,

3) регистры для хранения данных, адресов и служебной информации,

4) устройство для формирования (вычисления) адресов операндов,

5) устройство управления.

Устройство управления – управляет процессом последовательной выборки, декодирования и исполнения команд программы, хранимой в памяти. Устройство управления формирует временную диаграмму работы всех узлов процессора. Часть регистров также можно отнести к устройству управления.

Устройство формирования адресов операндов – вычисляет адрес, по которому произойдет очередное обращение к участку памяти, содержащему операнд.

АЛУ – комбинационное логическое устройство, имеющее два (многоразрядных) входа (на которые подаются два слова входных операндов), на выходе АЛУ формируется результат операций, которые процессор выполняет над *операндами,* таких, как сложение, умножение и т.п. Минимальный набор операций (машина Фон Неймана), которые должно выполнять АЛУ, включает операции сложения, инверсии и логического «И», все остальные операции можно получить на базе этих.

Регистры. Минимальный набор регистров, необходимый для функционирования процессора включает следующие регистры:

аккумулятор – хранит результаты операций, часто имеет удвоенную длину по сравнением с разрядностью процессора (для хранения результатов операций умножения и сдвига);

счетчик команд – содержит адрес следующей команды;

регистр адреса – содержит адрес операнда, используется при косвенной адресации;

регистр флагов (состояния и управления) - содержит код, характеризующий результаты предыдущих операций, а также информацию о текущем состоянии процессора.

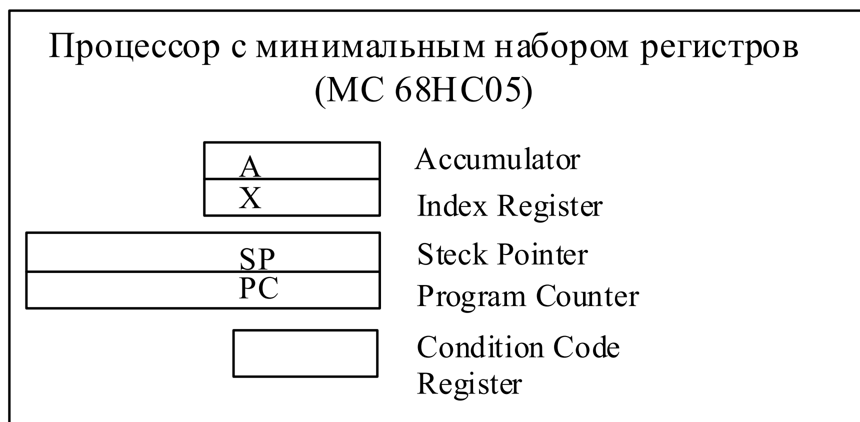


Рис. 1.4. Регистры процессора MC68HC05

Регистровый файл - набор однотипных регистров.

Каждый процессор имеет свой набор регистров. Так процессор – микроконтроллер MC68HC05 (рис. 1.4) имеет минимально возможный набор регистров. Можно выделить две большие группы процессоров: с *регистрами общего назначения* (регистровыми файлами) и со *специализированным набором регистров* (рис. 1.5). В первом случае все регистры регистрового файла одинаковы и их можно использовать произвольно в командах (упрощается программирование). Во втором случае за каждым регистром закреплена своя функция, и использование регистров в командах оговорено в формате каждой команды. Однако процессоры с регистровыми файлами требуется большие аппаратные затраты на организацию связи между регистрами. Так процессоры Motorola 68020 состояли из двух микросхем, в то время как процессоры Intel80386 только из одной.

Модель процессора для программиста – набор регистров, форматы команд, способы адресации, организация памяти и др. Можно рассматривать регистровую модель процессора - набор регистров, их форматы и способы работы с ними.

- *на пользовательском уровне* (регистры общего назначения и флагов)

- *на системном уровне* (регистры управления процессором и организации памяти, элементы организации прерываний и прямого доступа к памяти)

Вычислительное ядро (Core)– этим термином обозначают совокупность элементов процессора, необходимых для выполнения команды.

Периферийные устройства – устройства, внешние по отношению к связке процессор - память.

Устройства ввода-вывода – часть периферийных устройств, предназначенная для связи ЭВМ с «внешним миром». (Прочие периферийные устройства обслуживают внутренние потребности ЭВМ: таймеры, контроллер прерываний,…).

1.5. Программная модель процессора Pentium.

Под программной моделью на пользовательском уровне процессора Pentium мы будем понимать всю линейку процессоров IntelXX86 от 16-ти до 64-х разрядных процессоров. Программисту на уровне команд доступны четырнадцать регистров (рис. 1.5). Их удобно разбить на четыре группы: 1)Регистры данных, 2)адресные, 3)сегментные 4)указатель команд и регистр флагов (признаков). Для 64-х разрядных процессоров имеется еще одна группа из 8-ми регистров данных.

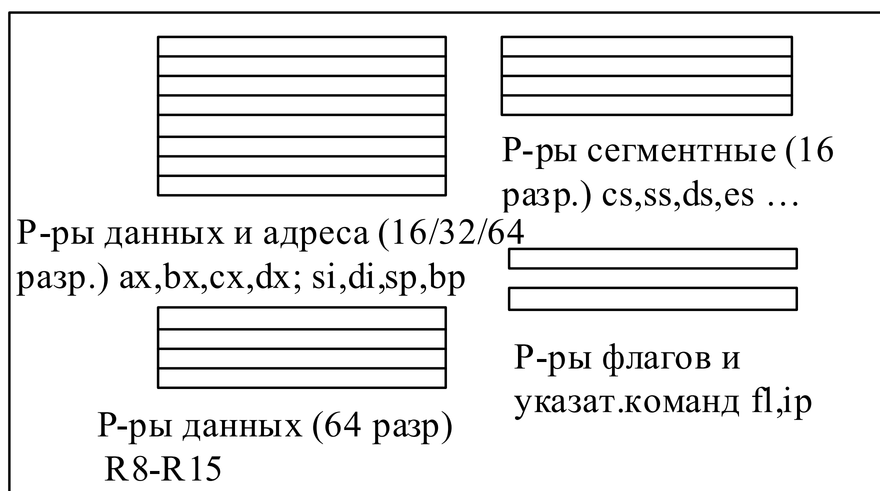


Рис. 1.5. Регистры процессора Pentium на пользовательском уровне.

1) Регистры данных (в некоторых книгах их называют регистрами общего назначения). Операнды в этих регистрах могут быть как слова, так и байты. Если операнд - байт, может быть указана любая половина регистра. Есть ряд команд, в которых функции отдельных регистров специализированы.

2) Указатели и индексные регистры (адресные регистры, используются для хранения 16-разрядных адресов).

3) Сегментные регистры (указывают начала четырех сегментов - участков по 64 К байт в 1М ОЗУ: сегмент команд CS, сегмент стека SS и два сегмента данных - DS и ES extra)

4) Указатель команд и регистр флагов.

Функции регистров i8086

AX Аккумулятор Умножение, деление и ввод-вывод слов
AL Аккумулятор(мл) Умножение, деление и ввод-вывод байтов
AH Аккумулятор(ст) Умножение и деление байтов
BX Регистр базы Базовый регистр, может быть использован как адресный
CX Счетчик Операции с цепочками, циклы
CL Счетчик (мл) Динамические сдвиги и ротации
DX Данные Умножение и деление слов, косвенный ввод-вывод
SP Указатель стека Стековые операции
BP Указатель базы Базовый регистр
SI Индекс источника Операции с цепочками, индексный регистр
DI Индекс приемника Операции с цепочками,

Примеры команд с регистрами:

```
mov ax,bx;      ax = bx
add ax,dx;      ax= ax + dx
```

Примеры команд с 8-ми, 16-ти, 32-х, 64-х разрядными регистрами:

```
add al,cl //  add ax,cx //  add eax,ecx // add rax,rcx
```

Регистр флагов процессора i8086

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

CF (Carry Flag) - флаг переноса;
PF (Parity Flag) - флаг четности;
AF (Auxiliary Carry Flag) - флаг вспомогательного переноса;
ZF (Zero Flag) - флаг нуля;
SF (Sign Flag) - флаг знака;
TF (Trap Flag) - флаг прерывания для отладки
IF (Interrupt-Enable Flag) - флаг разрешения прерывания;
DF (Direction Flag) - флаг направления цепочечных команд
OF (Overflow Flag) - флаг переполнения.

Сегментные регистры и организация памяти.

Для формирования физических адресов используется механизм сегментации памяти. Необходимость сегментной организации памяти

определяется недостаточным размером адресных регистров для адресации к памяти большей 64 Кбайт. Для организации адресации в процессоре используются четыре (в i80386 и выше шесть) 16-разрядных сегментных регистра, которые указывают на начала четырех сегментов памяти - участков по 64 К байт: сегмент команд CS, сегмент стека SS и два сегмента данных - DS и ES extra). Местоположение операнда в сегменте определяется смещением (offset).

Адресуемая память (адресное пространство для i8086) представляет собой область из 1M байт. Два смежных байта образуют слово. Адресом слова считается адрес младшего байта. Физический адрес памяти имеет длину 20 бит, однако, все обрабатываемые в регистрах процессора величины имеют длину 16 бит. Схема трансляции приведена на рис.1.6.

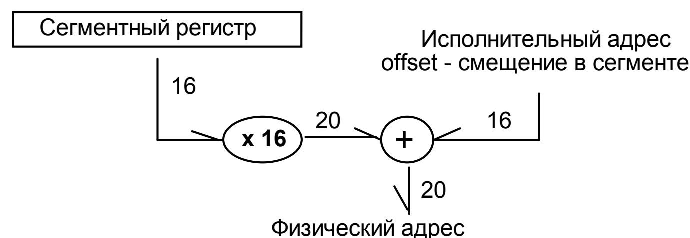


Рис. 1.6. Вычисление физического адреса в процессоре Intel X86.

Как было уже сказано, пространство памяти 1 М доступно процессору через 4 "окна" (сегмента). Начальный адрес каждого сегмента содержится в одном из четырех сегментных регистров (рис.1.7). Команды обращаются к байтам и словам в пределах сегментов, используя относительный (внутрисегментный) адрес. Таким образом, адрес операнда задается двумя компонентами: адресом сегмента и смещением. Адрес сквозной нумерации 0xb8000 задается в виде: **0xb800:0.**

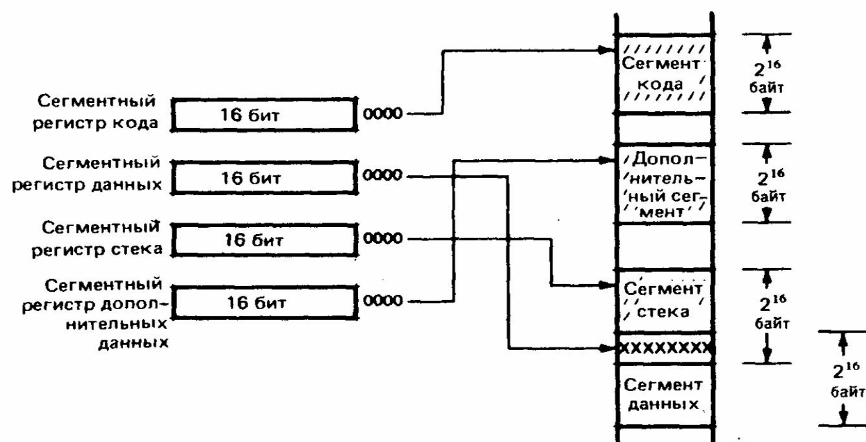


Рис. 1.7. Сегментная организация памяти в процессоре Intel X86.

1.6. Оперативная память (общая характеристика)

Оперативная память (оперативное запоминающее устройство ОЗУ) – часть компьютера, которая хранит команды выполняемых программ и элементы обрабатываемых данных и тесно взаимодействует с процессором в ходе выполнения программы.

Часть памяти, непосредственно взаимодействующая с процессором, имеет организацию с произвольным доступом. Эту часть называют *оперативной памятью (оперативным запоминающим устройством ОЗУ)*. Этому термину почти соответствует английский эквивалент *Random Access Memory RAM*.

Структурно физически реализованное устройство ОЗУ представляет собой (одномерный) упорядоченный массив запоминающих ячеек одинаковой разрядности (рис. 1.8), которые далее будем называть *минимальными адресуемыми единицами (МАЕ)*. Разрядность МАЕ в большинстве компьютеров в настоящее время равна одному байту (8 битов), хотя и сегодня есть модели, в которых адресуемая единица памяти имеет другую длину.

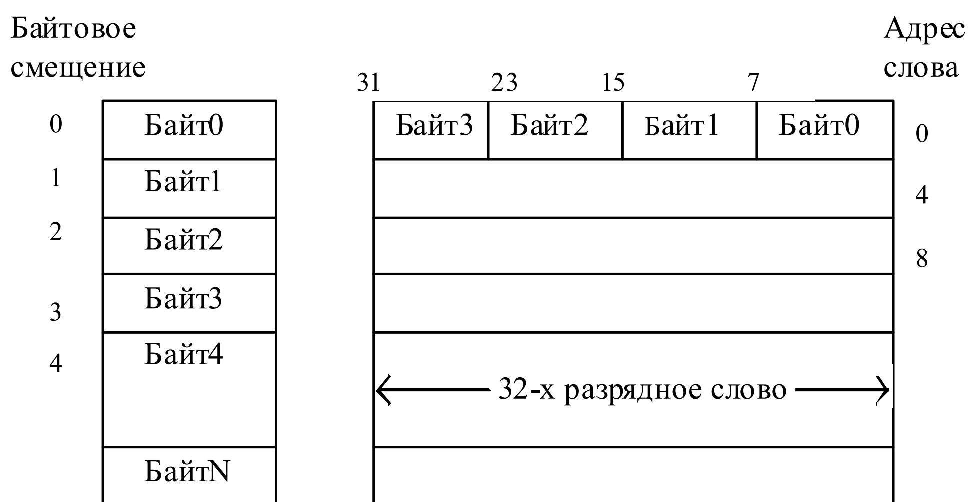


Рис. 1.8. Схема побайтной организации памяти.

Если команды и данные хранятся в одном и том же ОЗУ, такую архитектуру называют *принстонской* (машина Фон-Неймана).

Если команды и данные хранятся в разной памяти, такую архитектуру ЭВМ называют *гарвардской*.

Ячейки физической памяти пронумерованы подряд. Номер ячейки физической памяти называют ее ***физическим адресом***. Длина команды или элемента данных (операнда) нередко превышает длину МАЕ. Таким образом, каждый элемент программы (команда) или данных (операнд) хранится в ОЗУ, начиная с определенного адреса, и занимает там одну или несколько МАЕ. Когда говорят об адресе команды или об адресе элемента данных, имеют в виду адрес младшей МАЕ.

Большинство процессоров способно оперировать с двоичными словами (операндами) разной длины. Чаще всего, длина операндов измеряется в *байтах* и кратна степени числа 2. например, процессоры Pentium способны выполнять действия с операндами длиной в 1, 2, 4 или 8 байтов.

Во многих (но не во всех) процессорах разные команды имеют различную длину (но она всегда кратна длине минимальной адресуемой единицы, т.е. длина команды в адресах всегда выражается целым числом). В процессорах x86 длина команды может составлять от 1 до 15 байтов. Команда x86 может начинаться с произвольного адреса.

Бывают процессоры, в которых МАЕ имеют разную длину для команд и для данных. При этом для хранения команд и для хранения данных используются разные (как логически, так и физически, конструктивно) устройства ОЗУ. В таком случае говорят, о двух разных *адресных пространствах* – команд и данных.

Адреса элементов оперативной памяти, подобно командам и данным, во внутреннем представлении в компьютере также представляют собой двоичные слова.

Примеры команд обращения к памяти:

mov ax,[bx]; пересылка одного 16-ти разрядного слова из памяти в регистр ax

add al,[di+2]; суммирование байта из памяти с байтовым регистром и запись в регистр

Иерархическая организация памяти

Подсистема памяти имеет иерархическую, «многослойную» структуру: при переходе по слоям "сверху-вниз" (от процессора) - увеличивается объем и падает скорость.

Рис. 1.9. Иерархическая структура памяти.

Чем определяется соотношение объемов и скоростей «слоев»? Эти характеристики выбираются с целью получить наивысшую производительность при той же цене. Перечислим «слои» памяти ЭВМ:

1) **регистры процессора** – это составная часть процессора, которая, однако, выполняет функцию (временного) хранения программных объектов: элементов программного кода, обрабатываемых операндов и их адресов;

2) **кэш-память** (может быть, многоуровневая);

3) **оперативное запоминающее устройство** ОЗУ (Random Access Memory RAM) – память с произвольной адресацией;

4) **внешние ЗУ** (диски) – это память с последовательным доступом;

5) **сеть** – с точки зрения хранения данных, сеть ЭВМ, к которой подключен компьютер, может рассматриваться, как огромное, но медленное хранилище информации (Интернет).

С точки зрения понимания того, как проходит процесс выполнения программы, наиболее важное значение имеет часть 3) ОЗУ (основная или оперативная память).

1.7. Типовая структура машинной команды в пространстве и во времени. Формат и конвейер команд

Структура машинной команды «в пространстве»

Структура команды:

КОП	Адресная часть

Машинная команда во внутреннем представлении по форме представляет собой *двоичное слово*. Его можно представить состоящим из двух частей (полей): кода операции (КОП) и адресной части команды.

Код операции обозначает, что должна сделать команда. Длина битового поля, отведенного в команде под КОП, должна быть достаточной для кодирования всех команд процессора.

Поле КОП может быть переменной или постоянной длины. Некоторые кодовые комбинации поля КОП оставляют незадействованными, это *резервные коды* для будущего расширения системы команд.

Адресная часть команды содержит информацию о местоположении одного или нескольких операндов, используемых командой. Операнды или данные в ЭВМ, с которыми машинные команды выполняют действия, по форме также представляют собой двоичные слова. Эти битовые слова часто (но отнюдь не всегда) имеет смысл интерпретировать как числа.

Можно выделить несколько обобщенных форматов команд на основе рассмотрения количества операндов и адресных частей в команде (рис. 1.10).

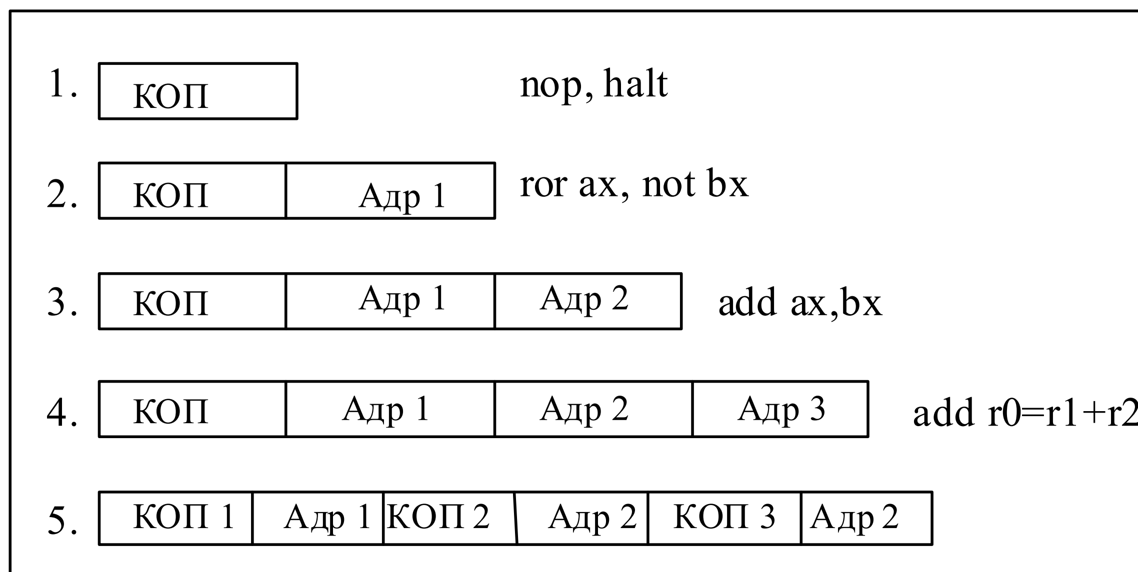


Рис.1.10. Обобщенные форматы команд.

1. В любой системе команд есть «пустая» команда, которая не делает никакого действия (NOP). Для нее просто не требуется указания операндов. Некоторые команды не требуют указания каких-либо операндов, так как их действие направлено на один заранее предопределенный операнд. Например, команда «разрешить прерывание» STI – эта команда устанавливает бит IF в регистре состояний EFLAGS.

2. Некоторые команды выполняют действие с одним операндом. Например, команда NEG – поменять знак операнда. Такие команды называют однооперандными. Передача управления – также однооперандная команда, в которой результат операции – изменение содержимого счетчика команд. В таких командах есть одно адресное поле для указания места расположения (или значения) операнда.
Пример такой команды: изменение знака операнда:

neg ax ; эта команда изменит знак целого числа в регистре процессора

3. В любом процессоре есть *команды пересылок*, которые копируют содержимое элементов памяти в другие элементы памяти. Такие команды

требуют указания двух операндов: *источника Source (Src)* и *приемника Destination (Dest, Dst)*. Пример такой команды:

mov bl, [esi] ; Содержимое однобайтового элемента данных из ячейки памяти, адрес которой хранится в регистре процессора esi копируется в регистр процессора bl.

4. Привычные нам арифметические действия (сложение, вычитание, умножение, …) оперируют с двумя операндами и формируют результат операции: C=A+B. Таким образом, в операции участвуют три элемента. В некоторых системах команд (процессоры Power, Alpha) соответствующие команды действительно позволяют программисту независимо указать места расположения всех трех элементов. Для уменьшения длины команды в других системах команд эти команды имеют только два адресных поля, при выполнении команды результат операции помещается на место одного из операндов, замещая («затирая») его.

5. В системах команд процессоров с длинным командным словом (VLIW) одновременно выполняются несколько команд и они должны быть записаны подряд – длинным словом. Формировать длинное слово может или программист (в сигнальных процессорах), или транслятор языка (Itanium).

Из приведенного рассмотрения должно стать понятным, что структура адресной части команды может существенно различаться для разных команд, как по количеству описываемых операндов, так и по способу кодирования информации о том, где расположен операнд.

Разработчики системы команд выбирают структуру адресной части команды, исходя из нескольких взаимно-исключающих требований, главные два из которых следующие:

а) Адресная часть команды должна быть по возможности короткой, чтобы не увеличивалась чрезмерно длина команды.

б) Должна обеспечиваться достаточная гибкость и универсальность кодирования параметров команды.

Адрес(а) операнда(ов), с которым(и) выполнит действие команда, определяются в процессе обработки этой команды (формируются *устройством вычисления адресов*). Такой адрес называют *исполнительным (executive)* или *эффективным (effective)* адресом EA. Адрес, который передается по магистрали из процессора в ОЗУ и управляет работой ОЗУ, будем называть *физическим адресом*.

Для того, чтобы получить значение *физического адреса* из значения *исполнительного адреса*, последний в процессоре должен быть подвергнут преобразованию, которое обозначают словосочетанием *трансляция адреса*.

Понятие *адресного пространства* является одним из фундаментальных в организации ЭВМ.

Отдельно следует сказать о *физическом адресном пространстве* – т.е. о пространстве реально формируемых адресов, которым могут соответствовать реально существующие (включенные в систему) элементы памяти. В системах на базе большинства представителей семейства Pentium может быть сформирован физический адрес длиной 32 или 64 бита, т.е. физическое адресное пространство равно 2^{32} или 2^{64}. В любой реальной системе *любое* адресное пространство используется чаще всего не полностью.

Можно сказать, что *трансляция адреса* это операция отображения одного адресного пространства (или может быть его части) на другое адресное пространство (или его часть). В простейших вычислительных системах трансляция адресов может представлять собой просто *тождественное отображение*, т.е. физический адрес равен исполнительному.

Процедура вычисления физического адреса по содержимому адресной части команды может быть достаточно сложной. Она включает два этапа:

1) вычисление *исполнительного*/*эффективного* адреса EA (executive/effective address) в соответствии со способом адресации;

2) переход от ЕА к физическому адресу (эта часть вычислений, какуже было сказано, называется "трансляцией адреса").

Структура команды во времени

Процесс выполнения одной машинной команды может быть разложен на несколько этапов. Разбиение на этапы имеет методическое назначение – оно помогает понять, как взаимодействуют отдельные узлы процессора при выполнении команд. Такое рассмотрение будет сделано непосредственно далее в данном подразделе.

Однако разбиение процесса выполнения команды на отдельные более мелкие этапы при конструировании внутренней структуры и схемотехники процессора позволяет использовать конвейеризацию для увеличения быстродействия процессора структурным путем. Фактически в разных процессорах количество этапов, на которые разбит процесс выполнения команд, может различаться как для разных процессоров, так и для разных команд в одном процессоре.

Процесс выполнения команды может содержать следующие этапы:

1.* Выборка команды(fetch F)	Обращение к памяти
2.* Дешифрация КОП(decode D)	
3. Вычисление адреса операнда	Обращение к памяти
4. Выборка операнда (read R)	Обращение к памяти
5.* Исполнение операции(execute E)	
6. Запись результата(retire/write W)	Обращение к памяти

Опишем более подробно процесс выполнения команд программы.

Выборка (F1 Fetch**).** Процессор байт за байтом считывает из ОЗУ порции программного кода (фрагмент программного кода, соответствующий команде процессора **обязательно** содержит КОП, и, возможно, но необязательно адресную часть) по адресу, который в данный момент содержится в счетчике команд.

После считывания очередной порции программного кода, содержимое счетчика автоматически увеличивается, так, что он всегда содержит адрес байта программы, который должен быть считан следующим.

Декодирование (D2 Decode**).** Процессор декодирует команду - анализирует считанный КОП, и по типу операции, закодированному в КОП, принимает решение о том, какая операция должна быть выполнена (управление АЛУ), а также о том, сколько байтов содержит данная команда и как их следует интерпретировать. Затем следует декодирование адресной части (если она есть). Если декодируемая команда предполагает расположение операнда в памяти, следует этап А3, в противном случае (если операнды расположены только в регистрах процессора) этап **А3** отсутствует.

Вычисление исполнительного адреса (А3 Address Calculation**)**, в ходе которого процессор формирует физический адрес операнда в памяти. Иногда (но далеко не всегда) для определения составных частей адреса операнда необходимо дополнительное обращение к ОЗУ. По вычисленному адресу происходит затем следующий этап R4:

Выборка операнда (R4 Read**)**, для чего (возможно) придется выполнить обращение к ОЗУ. Если адресная часть команды содержит информацию о нескольких операндах, находящихся в памяти, то этапы **А3** и **R4** будут повторены нужное количество раз. Если операнды располагаются в регистрах процессора, то на этапе **R4** обращений к памяти не требуется. За выборкой операндов следует этап Е5:

Исполнение операции (Е5 Execute**)** - не предполагает обращения к памяти, хотя время выполнения может для разных команд значительно различаться. По окончании этапа исполнения, может быть (но не всегда) придется произвести заключительное действие W6:

Запись результата (W6 Write**)** для чего (возможно) потребуется обращение к ОЗУ.

В зависимости от вида операции, количества операндов, способов адресации структура команды во времени (в частности, требуемое количество обращений к памяти) может сильно различаться для различных команд. Это означает, что для разных команд некоторые этапы могут отсутствовать, и для разных команд длительность во времени (даже одноименных) этапов может быть различной.

Конвейеризация

Конвейеризация - параллельное одновременное выполнение разных этапов разных команд. Чтобы такое запараллеливание стало возможным, в процессоре необходимо иметь для выполнения каждого из этапов отдельный исполнительный узел. На рис. 1.11 показаны узлы процессора и конвейер команд. Команды продвигаются из одного узла в другой.

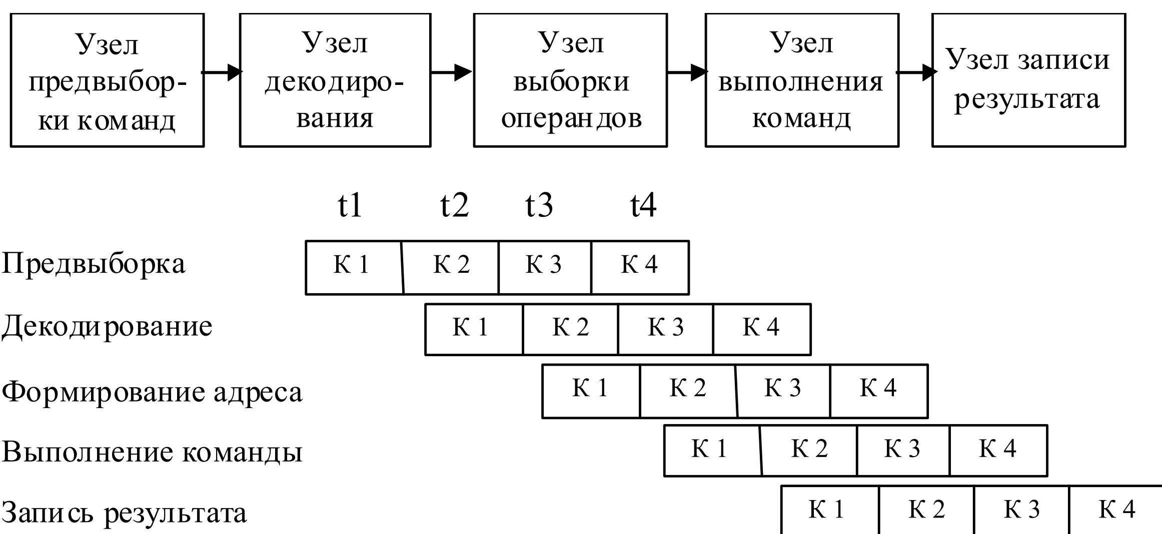


Рис.1.11. Конвейер команд.

Конвейеризация увеличивает пропускную способность процессора (количество команд, завершающихся в единицу времени), но она не сокращает время выполнения отдельной команды. Имеются некоторые накладные расходы на конвейеризацию, возникающие в результате несбалансированности задержки на каждой его ступени. Частота синхронизации (такт синхронизации) не может быть выше, чем время, необходимое для работы наиболее медленной ступени конвейера.

Конвейер не всегда представляет собой линейную цепочку этапов. В ряде ситуаций оказывается выгодным, когда функциональные блоки соединены между собой не последовательно, а в соответствии с логикой обработки. Отдельные блоки в цепочке могут пропускаться, а другие – образовывать циклические процедуры. Это позволяет с помощью одного конвейера вычислять более одной функции.

Поток команд - естественная последовательность команд, проходящая по конвейеру процессора. Процессор может поддерживать несколько потоков команд (суперпроцессоры 5 и 6 поколения), если для каждого потока и каждого этапа есть исполнительные элементы.

Суперконвейер команд – разбиение каждой ступени на подступени при одновременном увеличении тактовой частоты внутри конвейера; – включение в состав процессора многих конвейеров, работающих с перекрытием. Дробление ступеней позволяет поднять тактовые частоты процессора. К суперконвейерным относятся процессоры, в которых число ступеней больше шести.

Тип процессора	Количество ступеней конвейера
Pentium III	10
Itanium	10
UltraSPARC III	14
Pentium 4	20

Суперконвейер ведет не только к увеличению скорости вычислений, но и к возникновению дополнительных сложностей. Возрастает вероятность конфликтов. Дороже встает ошибка предсказания перехода – приходится сбрасывать весь длинный конвейер, на что требуется дополнительное время. Усложняется логика взаимодействия ступеней. Однако за счет использования новых архитектур, удается справиться с большинством проблем. При рассмотрении современных процессоров будут описаны новые идеи:

исполнения команд с изменением последовательности, переименования регистров, спекулятивного исполнения и другие.

Основные тенденции в развитии структур средств ВТ

Основная цель, к которой стремятся, совершенствуя структуру средств ВТ - это повышение производительности на единицу затраченных ресурсов.

Основные пути достижения этой цели связаны с распараллеливанием этапов вычислительного процесса. Современные процессоры и мультипроцессорные системы обеспечивают параллельное выполнение:

1. этапов соседних команд (конвейерная структура процессора)
2. этапов обращения к памяти (конвейерное обращение к памяти)
3. команд (векторный процессор, суперскалярный процессор, VLIW процессор)
4. потоков (нитей) команд (многоядерный и многопотоковый процессор)
5. программ (мультипроцессорная система, сеть)

ГЛАВА 2. ВИДЫ ДАННЫХ, ИХ КОДИРОВАНИЕ, КОМАНДЫ

Если вы прочитаете содержимое любой ячейки памяти, то вы увидите набор битов. Вы не скажете что это – команда, число, символ или др. На физическом уровне все виды данных и команды в ЭВМ представляются одинаково в виде битов (символов 0 и 1), которым соответствует два уровня напряжения 0В и 5В. Рассмотрим, как и какие данные могут быть закодированы (двоичным кодом), какие команды предназначены для работы с ними.

№	Вид	Команды	Форма представления	Параметры
1.	Числа	Арифметические: + - * / > <	Код числа	Диапазон и точность представления
2.	Тексты	Строковые и сравнения (совпадает или нет)	Строка символов	Алфавит
3.	Изображения	Обраб. фрагмента (перенос, заливка и др.)	Пиксел (полутон, цвет)	Формат

Числа. Этот вид данных был первым, с которым работали средства ВТ. Числа делятся на числа без знака, со знаком и с плавающей запятой. Основными операциями над числами являются арифметические операции.

Тексты. Как только было осознано, что компьютер может не только "вычислять", а является универсальным обработчиком данных, стали использовать ВТ для обработки текстов.. Основной объем данных у человечества накоплен был в символьной (текстовой) форме. До средств ВТ уже были разработаны способы кодирования текстов для передачи (телеграф). Основные операции, которые надо было выполнять с текстами – это:

- поиск в тексте фрагментов с заданным содержимым (вхождения подстроки в строку),

- сортировка (последовательности строк), по заданному критерию (например, по алфавиту),

- конкатенация (соединение) строк.

Изображения - человек получает через зрительный канал около 90% информации. До недавнего времени основной проблемой для обработки изображений был большой потребный объем ресурсов (1000 х 1000 пикселов - типовой размер). Операции над изображениями: фильтрация, заливка фрагмента, оконтуривание, перенос фрагмента и др.

Битовые поля - вид данных, когда отдельные элементы данных имеют разный размер (в битах), (нередко меньший, чем МАЕ) и напрямую не адресуются процессором. Характерный пример - отдельные поля в машинной команде (этот вид данных является выходным для трансляторов). Другой пример - входные данные, получаемые управляющими ЭВМ (микроконтроллерами) с объекта управления. Обращение к битовым полям требует дополнительных команд процессора для маскирования и сдвига. Операции над многобитовыми полями: извлечение битового поля, сборка одного числа из битовых полей, замена битового поля, проверка отдельного бита, переключение битов и др.

Например, для переключения битов (с 0 на 1, с 1 на 0) надо сложить х командой «Исключающее ИЛИ» с маской, у которой в соответствующих позициях единицы.

Прочие виды - .... (например, данные, описывающие звуки) и многое другое, с чем пока еще работают не очень много.

Видим, что перечисленные виды отличаются не только по содержанию, форме представления, параметрам, но и командам, которые необходимо выполнить при их обработке.

2.1. Числа и системы счисления

Характеристики чисел – диапазон представимых значений и точность представления.

Цифровое представление величин, когда определенно выбран формат представления (т.е. например выбрана разрядность, система счисления,.), имеет два присущих свойства :

а) ограничен диапазон представимых значений (их при N-битовом коде не более, чем 2^N штук);

б) ограничена точность (разрешающая способность) представления величины - нельзя отобразить изменение, меньшее некоторого кванта (единицы младшего разряда при целочисленном представлении);

в) представление чисел оказывается «циклическим» по отношению к арифметическим действиям.

Форма представления данных во всех современных средств ВТ - двоичный код. Это означает, что число изображается двоичным кодом. (Вначале пытались строить вычислительные устройства, работающие в десятичной системе счисления, поскольку десятичная система счисления привычна для человека. Но потом стало ясно, что технически проще реализовать элементы с малым количеством дискретных устойчивых состояний. Некоторые старые виды логических элементов использовали троичную систему счисления, но сегодня все работают в двоичной)

Системы счисления

Система счисления - совокупность правил представления и записи чисел.

Алфавит системы счисления - набор символов. Отдельные символы - буквы алфавита или цифры.

Основание СС - число символов в алфавите.

Слово или просто число - комбинация символов, изображающая число.

Виды систем счисления.

1. **Непозиционные** - значимость каждого символа в словах определяется только его начертанием и независит от его положения (позиции) в слове. Пример: Римская - Алфавит: I V X L C D M; Число LXXVII= L+X+X+V+I+I

2. **Позиционные** - значимость каждого символа определяется как символом, так и его местом (позицией). **Основание** - количество цифр в алфавите. Если основание N, то N-ричная сист. счисления. **Разрядная сетка** - совокупность возможных положений символов в словах. В разр.сетке веделяют место разделения слов на целую и дробную части, в этом месте ставится "запятая". Разряды располагаются и нумеруются вправо и влево от запятой. Каждому разряду придается "вес" $P_к$. **Весовая функция** - распределение весовых коэффициентов по разрядной сетке $\boldsymbol{P_к = f(к)}$.

3. **Однородная позиционная -** $\boldsymbol{P_к = n^k}$. Пример: восьмеричная, десятичная.

Восьмеричное изображение

$$A=a_{n-1}10^{n-1} + a_{n-2}10^{n-2} +...+ a_0 10^0+...+ a_{-m}10^{-m} =a_{n-1}8^{n-1} + a_{n-2}8^{n-2} +...+ a_0 8^0 + a_{-1}8^{-1}+...+ a_{-m}8^{-m}$$

Десятичное изображение

$N = 12345= 1*10^4 + 2*10^3 + \dots + 5*10^0$

Двоичное изображение

$N = 10011_b = (1*10^{100} + 0*10^{11} + 0*10^{10} + 1*10^1 + 1*10^0)_{bin} = (1*2^4 + 0*2^3 +0*2^2 + 1*2^1 + 1*2^0)_{dec}$

Шестнадцатиричное

$N = B4F1h = B*10^3 + 4*10^2 + F*10^1 + 1*10^0 = 11*16^3 + 4*16^2 + 15*16^1 + 1*16^0$

4. **Неоднородная позиционная** - весовая функция не представляет простую показательную функцию. Пример: время, двоично-десятичная.

5. **С непосредственным изображением чисел** - один знак - один символ.

6**. С кодированным изображением чисел** - число кодируется несколькими символами. *Пример*: **бинарно-кодированные системы -** каждая цифра исходного числа кодируется с использованием алфавита двоичной системы. Для кодирования десятичных чисел используются четыре двоичных разряда и веса разрядов могут быть: 8-4-2-1, 7-4-2-1, 5-4-2-1 и др. Одноразрядное число: $Y=\sum z_i p_i$; z_i - двоичная цифра, p_i - коэф-т веса.

Преобразование представления чисел из одной системы в другую

ЭВМ имеет устройства (регистры и АЛУ) с ограниченной разрядностью, поэтому числовая система ЭВМ конечна (может представить ограниченное количество чисел) и циклична, что наблюдается при последовательном прибавлении 1. Пример цикличности: 111111111+1=00000000 (теряется бит переноса).

Два способа:

Начиная с младших разрядов – последовательным делением на основание той системы счисления, в которую хотим преобразовать. Остатки от деления дают последовательно цифры кода, начиная с младшей.

13	2
6	1
3	0
1	1
0	1

Пример: - Из десятичной в двоичную - деление на 2 , запись остатка.

- Обратный перевод - сложение двоичных чисел с учетом их веса.

$1*2^3 + 1*2^2 + 0*2^1 + 1*2^0 = 8+4+0+1=13$

Начиная со старших разрядов – последовательным вычитанием степеней основания той системы счисления, в которую хотим преобразовать. Знак результата дает последовательно цифры кода, начиная со старшей.

2.2. Представление и команды над целыми числами

Целые числа без знака

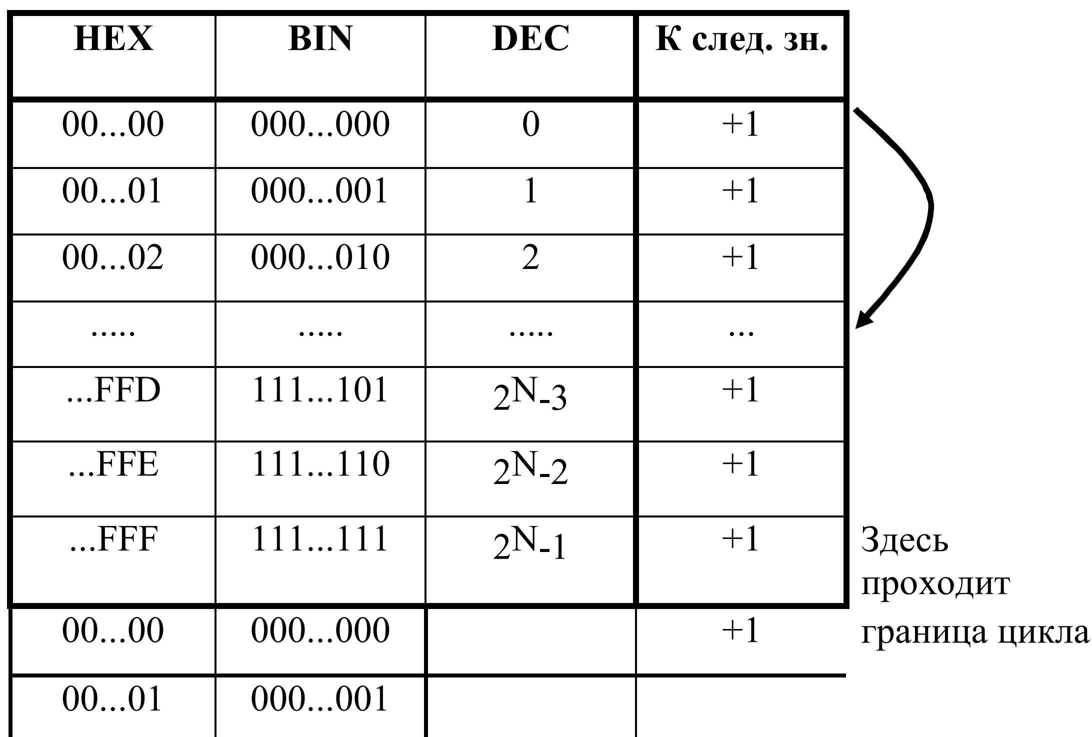
Целые числа используются для обозначения того, что можно считать "на штуки". Подсчитываемые "штуки" могут относиться как к вешним объектам

(данные вводятся извне), так и к внутренним, например количество циклов в программе. В частности это могут быть адреса элементов данных в памяти.

Представление чисел оказывается циклически повторяющимся. Цикличность представления и ограниченность диапазона приводят к тому, что итогом операции может быть либо **правильный результат, либо выход за границу** диапазона (этот выход называют термином "переполнение" (разрядной сетки)).

Диапазон представимых значений в N-битовой сетке можно записать таблицей в разных системах счисления: При возникновении переполнения его хотелось бы в процессоре обнаружить, а также определить, в какую сторону оно произошло: получился ли результат "больше максимального представимого числа" (при сложении) или "меньше минимального" (при вычитании).

Табл. 2.1. Диапазон представимых значений в *N*-битовой сетке



HEX	BIN	DEC	К след. зн.
00...00	000...000	0	+1
00...01	000...001	1	+1
00...02	000...010	2	+1
.....	.....	.....	...
...FFD	111...101	2^N-3	+1
...FFE	111...110	2^N-2	+1
...FFF	111...111	2^N-1	+1
00...00	000...000		+1
00...01	000...001		

При обнаружении переполнения (а также и других «экстраординарных», нештатных ситуаций) возможны различные способы реакции:

а) проигнорировать

б) остановить выполнение программы

в) предпринять заранее предусмотренные действия.

Замечание о разрядности: во многих процессорах возможно выполнение операций с операндами разной разрядности (обычно от минимально адресуемой единицы (МАЕ) до Слова).

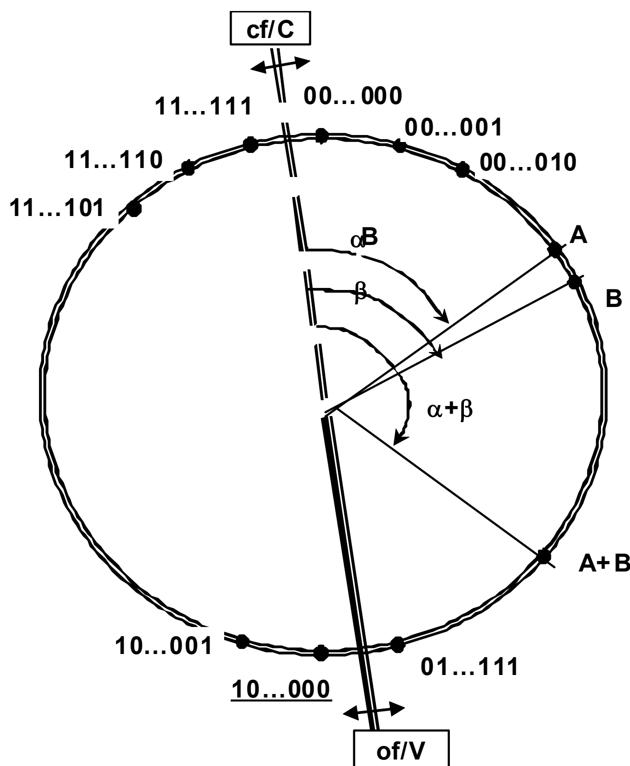


Рис. 2.1. Круговая диаграмма представления чисел.

Очень наглядно изображать цикличность представления целых чисел в ограниченной разрядной сетке с помощью круговой диаграммы, изображенной слева. Эта диаграмма хорошо иллюстрирует переход к следующему числу путем прибавления 1, переход к предыдущему путем вычитания 1, сложение чисел как сложение эквивалентных углов, а также явления переполнения для беззнаковых чисел и чисел со знаком.

Переполнение и расширение разрядности

В большинстве процессоров в случае переполнения разрядной сетки устанавливается специальный флаг переполнения - С-бит.

+	$a_{n-1}\ a_{n-2} \ldots\ldots a_1\ a_0$	первый операнд
	$\underline{b_{n-1}\ b_{n-2} \ldots\ldots b_1\ b_0}$	второй операнд
C←	$s_{n-1}\ s_{n-2} \ldots\ldots s_1\ s_0$	сумма (если возникает перенос – он устанавливает С-бит)

$_\ a_{n-1}\ a_{n-2}\ldots\ldots a_1\ a_0$ первый операнд

$\underline{\ b_{n-1}\ b_{n-2}\ldots\ldots b_1\ b_0\ }$ второй операнд

$C \leftarrow s_{n-1}\ s_{n-2}\ldots\ldots s_1\ s_0$ разность (если возникает заем – он устанавливает С-бит)

Пример переполнения чисел в 8-разрядном регистре.

254+7=5

1 1 1 1 1 1 1 0

0 0 0 0 0 1 1 1

0 0 0 0 0 1 0 1

Если диапазон представимых чисел в данной разрядной сетке мал, можно использовать представление чисел с повышенной разрядностью – перейти от *char* к *int* или к *double*. И далее использовать для представления операнда два или три или более слов.

В некоторых процессорах (команды *MMX-SSE*) может использоваться так называемая «арифметика с насыщением» (*saturation*) при выходе результата за границу представимого диапазона процессор подставляет в качестве результата ближайшее к нему представимое число. Приведем пример насыщения чисел на границах диапазона.

254+7=255

1 1 1 1 1 1 1 0

0 0 0 0 0 1 1 1

1 1 1 1 1 1 1 1 Насыщение на верхней границе

7-254=0

0 0 0 0 0 1 1 1

1 1 1 1 1 1 1 0

0 0 0 0 0 0 0 0 Насыщение на нижней границе

Целые со знаком: дополнительный код

Для кодирования знака используют следующие коды:

Прямой код. Знаковый разряд - единица, цифровые разряды без изменения.

Дополнительный код. Знаковый разряд - единица, цифровые разряды инвертируются и к младшему разряду добавляется единица. Доп.код позволяет производить замену операции вычитания сложением дополнительных кодов. Правило $[X]_{доп} + [Y]_{доп} = [X+Y]_{доп}$ выполняется всегда. Декодирование такое же, как и кодирования. Используется в ЭВМ.

Обратный код. Знаковый разряд - единица, цифровые разряды инвертируются. При сложении чисел единица переноса добавляется в младший разряд.

Модифицированные коды. Под знак отводят два разряда: старший - знаковый, младший - переполнения.

Число	Прямой код	Дополнительный код	Обратный код	Модифицированный код, обратный
-0,1101	1,1101	1,0011	1,0010	11,0010

Дополнительный код (английский эквивалент: two's complement) получается естественным образом - последовательным вычитанием 1 из нуля. Представление чисел со знаком в дополнительном коде также будет циклическим.

Дополнительный код хорош тем, что в нем действия сложения и вычитания над беззнаковыми числами и над числами со знаком выглядят одинаково. Это эквивалентно утверждению, что «дополнительный код суммы чисел равен сумме дополнительных кодов слагаемых».

Табл. 2.2. Диапазон представимых значений в N-битовой сетке.

HEX	BIN	DEC	К предыд. зн.	
10...00	100...000	-2^{N-1}	-1	Наименьшее отрицательное число
.....	.....	.....		

1..FFE		-2	-1	
1..FFF	111...111	-1	-1	
00...00	000...000	0	-1	Нуль
00...01	000...001	1	-1	
00...02	000...010	2	-1	
.....	.....			
0..FFF	011...111	$2^{N-1}-1$	-1	Наибольшее положительное число

Обратите внимание на то, что в выбранной разрядной сетке используются для представления чисел все кодовые комбинации, т.е. любой последовательности единиц и нулей в коде соответствует какое-то числовое значение. (При представлении чисел в форматах плавающей точки это будет не так !)

Граница на числовом круге выбирается так, чтобы диапазон был симметричен относительно нуля. Однако полной симметрии не получается. Наименьшее отрицательное число не имеет положительного эквивалента. Кроме того, при такой границе, **старший бит кода играет роль знакового**: 0 соответствует положительным числам, а 1 - отрицательным.

В дальнейшем, говоря о представлении отрицательных чисел, будем иметь в виду использование дополнительного кода, если специально не оговорено иное.

Действия над целыми числами и машинные команды

Какие операции надо производить с целыми числами? Интуитивно это в первом приближении понятно: Арифметические действия: + - * / .Сравнение целых чисел: > < =

Каков минимальный набор команд? Оказывается, он очень невелик. (Более сложные команды можно реализовать как комбинации простых). Некоторые такие комбинации со временем реализуют в уровне машинных команд.

Деление	последовательность вычитаний и сравнений
Умножение	последовательность сложений и сдвигов
Вычитание	сложение с отрицательным операндом

Смена знака	инвертирование и прибавление 1
Сложение	
Сдвиг	(что делать с битами, выходящими за пределы разрядной сетки ?)
Увеличение (/уменьшение) на 1	по честному со всеми переносами и, возможно, с переполнением

В этой таблице некоторые вышестоящие операции могут быть выполнены как комбинации некоторых нижестоящих.

Приведем пример сложения двух чисел разных знаков.

```
          -2+7=5
0  0  0  0  1  0  2
1  1  1  1  0  1  инверт
1  1  1  1  1  0   +1->-2
0  0  0  1  1  1  7
0  0  0  1  0  1  5
```

Приведем пример сложения двух отрицательных чисел.

```
             -2-7=-9
0  0  0  0  0  0  1  0  2
1  1  1  1  1  1  0  1  инверт

1  1  1  1  1  1  1  0   +1->-2
1  1  1  1  1  0  0  1  -7
1  1  1  1  0  1  1  1  -9

0  0  0  0  1  0  0  0  инверт
0  0  0  0  1  0  0  1  +1->-9
```

Переполнение при действиях над числами со знаком.

При сложении беззнаковых чисел - переполнение происходит, если при действии пересекается граница разрыва 11...11 <-> 00...00.

При сложении чисел со знаком: если знаки разные - переполнения быть не может.

Если знаки одинаковые, и знак результата - совпадает со знаками операндов - переполнения нет; противоположен знаку операндов - переполнение есть. При действиях над числами со знаком переполнение происходит, если пересекаем границу разрыва 011..11 - 10..00. Для

регистрации этих фактов в процессоре обычно делают два триггера (чаще всего они входят в состав регистра состояний). Содержимое этих триггеров называют битами признаков или флагами.

С (или cf -от CARRY - перенос) регистрирует при выполнении многих операций перенос/заем из старшего (знакового) разряда за границу разрядной сетки

V (или of от OVerFlow - переполнение) - регистрирует перенос/заем в старший (знаковый) разряд.

Если диапазона представимых значений не хватает, можно искусственно увеличить разрядность - хранить величины в нескольких машинных словах - тогда действия над ними, даже такие простые, как сложение и вычитание потребуют большего количества действий.

Команды для действий с целыми числами

Далее описывается действие основных команд, имеющихся в большинстве процессоров, примеры мнемоник приводятся для системы команд *i*XX86.

Сложение и вычитание

Обе операции трехоперандные: два слагаемых и результат. Формат команды может быть:

3-х операндным –

2-х операндным – результат помещается на место одного из слагаемыых (так в *i*XX86)

1-операндным – второй операнд и сумма находятся в предопределенном месте (в регистре-аккумуляторе)

нуль-операндным – оба операнда находятся в предопределенных местах (чаще всего это стек).

add a, b **a ← a + b** так принято в Intel – ассемблере, т.е. результат сложения помещается на место операнда **a**. Аналогично для вычитания : **sub a, b**

Приятная **особенность дополнительного кода** для представления отрицательных чисел:

Результат действий сложения и вычитания при использовании дополнительного кода имеет один и тот же вид, независимо от того, интерпретируются ли операнды как беззнаковые или как знаковые. Для действий умножения и деления это не так.

Глядя на двоичное представление целого числа, невозможно сказать, знаковое оно или беззнаковое.

При выполнении операции схемотехника процессора изменяет значения битов **cf** и **of**. Анализируя эти биты, можно узнать, не выходит ли результат сложения или вычитания за пределы представимого диапазона. (Как это сделать, узнаем позже).

Если не хватает разрядности, поддерживаемой АЛУ, то можно искусственно увеличить разрядность - хранить величины в нескольких машинных словах. Рассмотрим, как можно организовать сложение двух «длинных» операндов, используя «короткое» сложение.

$$
\begin{array}{rr}
 & + a_{n-1}\ a_{n-2}\ \ldots\ldots\ a_1\ a_0 \\
 & +\mathbf{cf} \leftarrow b_{n-1}\ b_{n-2}\ \ldots\ldots\ b_1\ b_0 \\
+ a_{2n-1}\ a_{2n-2}\ \ldots\ldots\ a_{n-1}\ a_n & \\
+ b_{2n-1}\ b_{2n-2}\ \ldots\ldots\ b_{n-1}\ b_n & \\
\hline
\mathbf{cf} \leftarrow b_{2n-1}\ b_{2n-2}\ \ldots\ldots\ b_{n-1}\ b_n & b_{n-1}\ b_{n-2}\ \ldots\ldots\ b_1\ b_0
\end{array}
$$

Результат переноса при сложении младших частей регистрируется в флаге **cf**. Его значение надо учитывать при сложении старших частей.

Для учета при сложении значения флага переноса **cf** в систему команд добавляют **команду сложения с переносом**: **adc a, b** (работает так: **a ← a + b + cf**)

Последовательность команд при выполнении сложения с двойной точностью может быть такой

```
add  b0, a0        ; b0 ← b0 + a0  (перенос в cf)
adc  b1, a1        ; b1 ← b1 + a1 + cf
```

В некоторых системах команд (например DEC16) команда **adc** однооперандная : **a=a+cf** и тогда так:

```
add b0, a0      ; b0 ← b0 + a0
adc b1          ; b1 ← b1 + cf
add b1, a1      ; b1 ← b1 + a1           // Вопрос о тройной точности
```

Как перейти от представления с одинарной точностью к представлению с двойной ? Для положительных чисел просто - старшее слово - нулевое.

Для отрицательных - **операция расширения знака** (Sign Extention). Ее действие состоит в заполнении старшего слова знаковыми битами младшего. В системе команд *i*XX86 есть команды **cbw** преобразования 8 бит→16 бит, **cwd** преобразования 16 бит → 32 бит и **cdq**.

Умножение и деление

При умножении *n*-разрядных операндов, результат может иметь разрядность до 2*n*. Поэтому переполнение разрядной сетки при умножении будет возникать гораздо чаще, чем при сложении (т.е. при большей доле комбинаций сомножителей). В системах команд не очень примитивных процессорах есть команды умножения и деления, однако для знаковых и для беззнаковых операндов команды должны быть разные.

В большинстве процессоров АЛУ устроено так, что соотношение разрядностей операндов и результата в командах умножения и деления следующее:

$(n) * (n) = (2n)$, $(2n) / (n) = (n)_q(n)_r$, т.е. разрядность произведения и делимого вдвое больше. Что делать с «длинным» результатом – проблема программиста.

Как ведут себя флаги при выполнении команд умножения и деления, следует смотреть в полном описании команд.

Частный случай умножения – умножение на степень 2-х, (т.е. на степень основания системы счисления) можно реализовать сдвигом операнда на

соответствующее число разрядов влево для положительной степени или вправо для отрицательной. Для этого в системе команд может иметься **команда арифметического сдвига**. При арифметическом сдвиге вправо (деление на степень двойки) знаковый разряд числа сохраняет свое значение неизменным, а при сдвиге влево (умножение на степень двойки) правый конец операнда заполняется нулями. Если при сдвиге влево знаковый разряд операнда изменял свое значение – это означает, что при умножении произошло переполнение (выход за пределы диапазона представимых значений), это вызывает изменение флагов **cf** и **of**

Сравнение чисел, флаги и набор команд ветвления

Сравнение в ЭВМ используется для организации последующего ветвления алгоритма (условного перехода в программе) в зависимости от результата сравнения.

Сравнение **можно производить** по условиям: а) равно / неравно; б) больше / меньше.

Условие а) всегда осмысленно и формально означает, что все биты сравниваемых операдов одинаковы.

Условие б) - его семантика понятна для данных, которые неким образом упорядочены, таких как числа или символы алфавита. Для такого вида данных, как битовые поля - не всегда понятно, что такое "больше/.меньше"

Сравнение в процессоре происходит по одной из двух схем:

1) Сравнение операнда с нулем (его можно произвести специальной командой **проверка** *test*)

2) Сравнение двух операндов между собой (вычитанием и последующим сравнением результата с нулем)

Для сравнения двух чисел можно использовать команды вычитания, кроме того обычно в системе команд есть специальные команды для сравнения. В *i*XX86 это команда

cmp a,b - она делает вычитание **a – b**, по результату операции устанавливаются флаги, после чего результат вычитания теряется, а операнды сохраняются неизменными. Обычно в процессоре имеются еще два триггера (флага), которые меняют свое состояние при выполнении команд в зависимости от получившегося результата:

Z (zf - от Zero) - флаг нулевого результата: **zf ← 1**, если результат равен нулю.

N (sf - от Negatve или от Sign) - флаг отрицательного результата (знака) - значение этого флага совпадает со знаковым разрядом результата, т.е. 1 соответствует знаку "минус".

После выполнения команды, осуществляющей сравнение (и переустанавливающей флаги) надо осуществить ветвление. Для этого в процессоре обычно существует большая **группа команд условного ветвления**, которое происходит (или нет) в зависимости от состояния тех или иных флагов.

2.3. Форматы чисел плавающей запятой, команды

Логика перехода к формату ПТ

Отметим снова некоторые свойства целочисленного формата представления величин:

а) разрешающая способность (минимальное изменение величины, которое можно отобразить), равна 1 (при любом числе разрядов **n**).

б) диапазон представимых значений однозначно определяется количеством разрядов **n** в разрядной сетке: 2^n, т.е. разрешающая способность и диапазон связаны однозначно при данном количестве битов в представлении числа.

В формате чисел с плавающей запятой точность и диапазон развязываются. **Точность** – задается количеством знаков (битов) в поле мантиссы. **Диапазон** – количеством битов в поле порядка (оно определяет, насколько можем сдвигать разделитель).

Неоднозначность представления и нормализованная форма

Число в формате ПТ можно представить как

$$\pm \text{\textit{Мантисса}} * \text{\textit{основание}}^{(\pm \text{\textit{порядок}})}.$$

Однако такое представление неоднозначно:

$$1.2345 * 10^2 = 12.345 * 10^1 = 0.12345 * 10^0$$

Выбирают одно из представлений как стандартное – такое представление называют *нормализованным*.

Двоичное представление - возможность неявного бита. Нормализация.

Если определенным образом выбрано нормализованное представление, то при его записи в двоичном виде старший разряд мантиссы всегда двоичная 1 - ее можно не запоминать - сэкономим бит !!!

В чем состоит процедура нормализации: двигаем мантиссу, пока не получим нормализованного представления, и при этом корректируем порядок, для того, чтобы сохранить величину числа неизменной.

Пример: число $101101.110001000..._b * 2_d^{1010}$

Пусть выбрано нормализованное представление, в котором мантисса содержит один значащий разряд в целой части (всегда = 1).

При нормализации сдвигаем мантиссу вправо на 5 разрядов (это эквивалентно делению мантиссы на 2^5=32, и корректируем порядок (увеличиваем его на 5_d=101_b).

Нормализованное представление: $1.011011100010..._b * 2_d^{1111}$.

Опуская неявный бит, получим мантиссу: 011011100010..., порядок ..01111.

Важное замечание 1: при нормализации, когда приходится сдвигать мантиссу вправо, возможна потеря точности при выходе младших битов мантиссы за границу ее разрядной сетки.

Важное замечание 2: нормализация не всегда возможна, если при коррекции порядка происходит выход за границу его разрядной сетки. Если получившееся число слишком велико, его просто нельзя представить в выбранном формате ПТ. Если число слишком мало, то его можно представить, но лишь в ненормализованной форме

Нормализация нужна для представления с максимальной точностью - использования всех разрядов мантиссы. Неявный бит возможен только в нормализованном представлении.

Типовой формат представления числа ПТ

Знак	Порядок	Мантисса

Знак относится ко всему числу.

Поле мантиссы содержит ее значение в прямом (не в дополнительном) коде с опущенным неявным битом.

Поле порядка содержит сумму истинного порядка и положительной константы, называемой смещением (см. примеры далее).

Действия с числами ПТ

Сложение и вычитание

$9.87 * 10^3 + 4.56 * 10^2 = 9.87 * 10^3 + 0.456 * 10^3 = (9.87 + 0.456) * 10^3 = 10.326 * 10^3 = 1.0326 * 10^4$

1) Проверить, выровнены ли порядки, и если нет – то выровнять
2) Сложить мантиссы (одна из них возможно денормализована)
3) Проверить, нормализован ли результат, если нет – то нормализовать.

Умножение и деление

$(3.5 * 10^{2)} * (4.5 * 10^{-4}) = 3.5 * 10^2 * 4.5 * 10^{-4} = (3.5 * 4.5) * (10^2 * 10^{-4}) = 12.25 * 10^{2-4} = 1.225 * 10^{-1}$

1) Сложить порядки
2) Перемножить мантиссы

3) Проверить, нормализован ли результат, если нет – то нормализовать.

Замечание о влиянии ограниченной разрядной сетки

Если ограничена разрядная сетка, то а) при перемножении мантисс возможно произойдет потеря точности из-за того, что «длинное» произведение не влезет в разрядную сетку для мантиссы, б) нормализация не всегда возможна.

Стандарт на числа ПТ ANSI/IEEE 754-1985

При выборе формата можно выбирать:
- разрядности мантиссы и порядка
- использовать ли формат с неявным битом
- нормализованное представление
- величину смещения при изображении порядка
- расположение полей в коде ПЗ

Форматы: одинарная, двойная, расширенная точность

Русское наименование	Короткое вещественное (КВ)	Длинное вещественное (ДВ)	Временное вещественное (ВВ)
Английское наименование	Short Real (SR)	Long Real (LR)	Temporary Real (TR)
Всего битов	32	64	80
Поле порядка	8	11	15
Смещение порядка	$2^7 - 1 = 127$	$2^{10} - 1 = 1023$	$2^{14} - 1 = 16383$
Диапазон десятичных порядков	-37 ÷ +38	-307 ÷ +308	-4931 ÷ +4932
Поле мантиссы	23	52	64
Точных знач. десятичных цифр	6	15	19
Неявный бит	Опущен	Опущен	Изображается

Параметры временного вещественного выбраны так, чтобы при действиях с ДВ по возможности не происходило потери точности.

Диапазоны значений и использование кодов

При кодировании целых чисел в выбранной разрядной сетке обычно используются все кодовые комбинации. При кодировании ПЗ есть кодовые комбинации, которым не соответствует никакое значение !

В таблице перечислены типы значений, кодовые комбинации, используемые при представлении чисел ПЗ

	Наименование	Знак	Порядок	Мантисса	Значение
1	Положительные "не-числа" (NotANumber NAN)	0 0	11...111 .... 11...111	(1.)**1**1...11 .... (1.)**0**0...01	+ QNAN ..... + SNAN
2	+ Бесконечность	0	11...111	(1.)00...00	+ ∞
3	Положительное нормализованное число	0 ... 0	11...110 ..... 00...001	(1.)11...11 ..... (1.)00...00	КВ ДВ ВВ
4	Положительное ненормализованное число (только ВВ)	0 ... **0**	11…110 **00...001**	01...111 **00...000**	Только ВВ (неподдерживаемое) **+ Псевдонуль**
5	Положительное денормализованное число	0 ... 0	00...000 (..001) 00...000	(0.)11...11 ..... (0.)00...01	
6	+ нуль - нуль	0 1	00...000 00...000	(0.)00...00 (0.)00...00	+ 0 - 0
7	Отрицательное денормализованное число	1 ... 1	00...000 (..001) 00...000	(0.)00...01 ..... 0.)11...11	
8	Отрицательное ненормализованное число (только ВВ)	**1** ... 1	**00...001** ..... 11…110	**00...000** ..... 01...111	**- Псевдонуль** (неподдерживаемое - только ВВ)
9	Отрицательное нормализованное число	1 ... 1	00...001 ..... 11...110	(1.)00...00 ..... (1.)11...11	
10	- Бесконечность	1	11...111	(1.)00...00	
11	Отрицательные "не-числа" (NotANumber NAN) **Неопределенность**	1 ... 1 **1**	11...111 ... 11...111 **11...111**	(1.)**0**0...01 ... (1.)**1**1...11 **110...00**	- SNAN ... - QNAN

В таблице перечислены специальные численные значения:

Денормализованное вещественное число - малое, которое не поддается нормализации.

Нули - положительные и отрицательные.

Со знаком и без знака бесконечности (аффинная и проективная бесконечность).

Сигнальные и спокойные нечисла NaN - Not-a-Number.

Особые случаи при действиях над числами ПЗ

При выполнении операций над числами ПЗ могут наступать особые случаи. Особый случай - неправильное выполнение действия процессором. Численный особый случай - выход числа из формата представления. При возникновении особого случая процессор или а) сам обрабатывает его в сопроцессоре (FPU), формируя приемлемый результат (специальное численное значение), или б) осуществляет вызов программного обработчика прерывания (Особый случай - источник внутреннего прерывания). Перечислим особые случаи в порядке убывания приоритета. Может наступить и несколько особых случаев одновременно.

Особый случай	Ответ FPU по умолчанию
1) недействительная операция	QNaN (особ. случ. стека) SNaN (недейств операция)
2) деление на нуль	Бесконечность
3) денормализованный операнд	Денормализ. операнд (правила действия – стремится к нормализации)
4)численное переполнение	+Беск., -Беск., Nmax, Nmin В зависимости от режимов округления
5) численное антипереполнение	Денормализ. операнд или 0.
6) неточный результат	Результат округления

Неточный результат, округление и его виды

Неточный результат возникает при действиях с мантиссой (например, при умножении) или записи из временного вещественного. При этом в сопроцессоре устанавливается соответствующий флаг и происходит округление результата в соответствии с одним из четырех правил:

1) к ближайшему значению; 2) по направлению к +∞; 3) по направлению к -∞; 4) по направлению к 0 (усечение – выполняется легко и быстро).

При реализации численных методов использование разных способов округления по-разному влияет на точность метода.

Численное антипереполнение, денормализованные и ненормализованные числа, псевдонуль.

Результат умножения двух малых чисел может быть слишком мал для представления нормализованным числом (появление такого результата называют антипереполнением). Это означает, что при нормализации скорректированный порядок выходит за границу представимого диапазона порядков. В рамках стандарта такой результат представляется денормализованным числом. Для представления денормализованных чисел в диапазоне порядков выбирается одно значение 00……00 - оно свидетельствует, что число – денормализованное. При этом предполагается, что порядок равен 00……01, а мантисса не содержит 1 в целой части.

Однако вычисления при получении денормализованного результата можно (с определенной потерей точности) продолжать (в старых реализациях вместо денормализованного результата возникал «машинный нуль»).

Кодовые комбинации, содержащие ненулевое поле порядка и нулевой старший бит мантиссы (такое возвожно только в формате ВВ) называют ненормализованными числами.

Если перемножаются два денормализованных числа, содержащие в мантиссах суммарно более 64 нулей, то ненулевой результат не может представлен даже в формате ВВ. Такой результат называется **псевдонулем**.

Действия с бесконечностями

Деление на нуль или на очень малое число вызывает выход “вверх” за диапазон представимых значений формата ПЗ (переполнение). Для результатов таких операций используется специальное значение “бесконечность”, которое отмечается особым кодом.

Знак бесконечности, получаемой при переполнении разрядной сетки часто бывает ненадежен:

$\pm\infty = X / \pm\alpha = X / (a_1y_1 - a_2y_2)$ в знаменателе малая разность двух больших чисел, на знак влияет игра результатов округления. Поэтому для работы с бесконечностями использовались два режима: *проективное замыкание (projective closure)*, когда значения $+\infty$ и $-\infty$ не различаются, и *аффинное замыкание (affine closure)*.

Бесконечность в аффинном смысле допускает большее количество операций, в частности сложение «бесконечностей» с одинаковыми знаками дает результат «бесконечность», тогда как в бесконечность в проективном смысле дает результат «неопределенность».

Недействительные операции и их результаты.

Недействительная операция	Результат
Любая арифметическая операция над SNAN или Unsupported	QNAN
Сложение ∞ с разными знаками или вычитание ∞ с одинаковыми знаками	Неопределенность
Умножение 0 * ∞ или деление ∞ / ∞ или 0 / 0	Неопределенность
Сравнение или тестирование, если хотя бы один операнд NAN	Устанавливает код условия «не сравнимы»
Операция вычисления остатка, когда делитель =0 или делимое = ∞ .	Неопределенность
Тригонометрические операции, когда арг = ∞	Неопределенность
Корень или логарифм отрицательного числа	Неопределенность
Пересылка в целый операнд NAN или ∞	Целочисленная неопределенность (100…0)

2.4. Кодирование символов

Общие соображения.

Текст есть (изначально) – материализованная человеческая речь. Поэтому в первом приближении структура текста сходна со структурой речи: фразы – слова – фонемы для речи превращаются в фразы – слова – буквы для

фонетических систем или в фразы – лексемы – знаки (иероглифы) для нефонетических. Таким образом, элементарной информационной единицей для представления текста является *символ* текста (буква, фонема, иероглиф).

Хочется быть в состоянии изображать и обрабатывать любой текст из тех, что печатаются в книгах. Что именно хочется изображать/кодировать

1) Собственно символы, входящие в состав данного текста

2) Конфигурацию символов, используемых в данном тексте

3) Расположение фрагментов текста на странице (колонки, таблицы, направление текста, и т.п.)

4) Расположение символов внутри фрагмента текста, например надстрочные и подстрочные индексы, символы в математических или в химических формулах

5) Составные символы, включающие несколько отдельно кодируемых частей (интеграл или сумма с пределами, …)

Каково общее количество символов, которые хочется изображать:

Буквы латинские - 26 * 2 = 52, так как строчные и прописные

Буквы русские - 32 * 2 = 64, строчные и прописные

Цифры - 10 штук

Знаки препинания -тоже штук 10

Знаки математических операций - тоже более 10

Греческие буквы - используются в математических текстах

Дополнительные знаки, используемые в текстах разного вида, такие как $,%,#,@,§,...

Специальные символы, используемые в различных специфических областях деятельности, например: нотные знаки в музыке, типографские (корректорские) знаки, специальные математические знаки (интегралы, кванторы,

Если текст многоязычный, то дополнительно буквы национальных алфавитов.

Всего для исчерпывающего кодирования фонетических систем письменности требуется изображать (кодировать) несколько сотен (а может быть, и более 1000) символов.

ASCII-код. Структура кодовой таблицы

American Standard Code for Information Interchange - Использовали 7 битов. Всего можно было изобразить до 128 символов, но из них печатных было 96. 8-й бит использовали по-разному: - для контроля четности; - как признак "нет/есть данные"; - для расширения набора изображаемых символов; - никак не использовали.

OCT	DEC	HEX	Содержимое
000...037	0...31	0...1F	Управляющие символы (ConTRoL-коды)
040...057	32...47	20...2F	Знаки (20h - пробел)
060...071	48...57	30...39	Цифры 0,1,...9
072...100	58...64	3A...40	Знаки
101...132	65...90	41...5A	Буквы A,B,...Z
133...140	91...96	5B...60	Знаки
141...172	97...122	61...7A	Буквы a,b,...z
173...177	123...127	7B...7F	Знаки

Алфавитно-цифровые символы и символ пробела

Они занимают часть кодов, начиная с 48, и упорядочены, что позволяет легко:

а) производить преобразование из числового представления в символьное:

код_цифры = число + 30h

б) переходить от порядкового номера в алфавите к коду буквы - для латинского алфавита:

код_прописной_буквы = порядковый_номер + 40h

код_строчной_буквы = порядковый_номер + 60h

Кроме того, упорядочение позволяет легко программировать сортировку записей, включая и левые пробелы, по алфавиту или по возрастанию чисел,.

Арифметические знаки и знаки препинания расположены с алфавитно-цифровыми вперемешку.

Управляющие символы

В технике передачи символьной информации (телеграфии) надо было передавать не только печатные символы, но и дополнительную управляющую информацию, такую, как

- переход к следующей строке
- переход к началу строки
- отмена последнего переданного символа, ... и еще многие другие.

Для этого в таблице ASCII была выделена часть кодов с 0 по 31. Некоторые наиболее употребительные управляющие коды с их обозначениями, пришедшими из техники телеграфной связи, даны в таблице:

OCT	DEC	HEX	Ctrl/	Сокр.	Название, пришедшее из телеграфии
000	0	00	^@	NUL	Пустой символ (это не пробел!)
003	3	03	^C	ETX	Конец текста (прерывание)
007	7	07	^G	BEL	Звуковой сигнал
010	8	08	^H	BS	Возврат на шаг
011	9	09	^I	HT	Табуляция
012	10	0A	^J	LF	Перевод строки
014	12	0C	^L	FF	Перевод формата
015	13	0D	^M	CR	Возврат каретки
016	14	0E	^N	SO	Верхний регистр
017	15	0F	^O	SI	Нижний регистр
021	17	11	^Q	DC1	Упр.устр.1 (возобновление работы)
023	19	13	^S	DC3	Упр.устр.3 (приостановка работы)
032	26	1A	^Z	SUB	Замена (конец текста, строки, файла)
033	27	1B	^[	ESC	Переключение кода (начало управл. послед.)

Управляющие символы называют Control-кодами от английского слова "управление".

На старых примитивных устройствах символьного ввода-вывода - телетайпах - с каждой клавишей был однозначно связан формируемый код. Для формирования же управляющих кодов на телетайпах делалась специальная клавиша Control, удержание которой "обнуляло" три старших бита в коде, формируемом при нажатии "обычной" алфавитно-цифровой

клавиши. Например, для ввода управляющего символа ESC с кодом 1Bh надо было нажать клавишу с символом **;** (код 3Bh) или клавишу с символом **[** (код 5Bh). Любой из этих кодов при обнуленных трех старших битах давал код 1Bh (символа ESC). Отсюда название клавиши Ctrl на современных клавиатурах.

Символ ESC (код 27). С этого кода начинаются последовательности символов, называемые управляющими. Эти последовательности также используются для переключения кодовых таблиц, а, кроме того, для изменения многих других свойств символьных устройств (клавиатур, дисплеев, принтеров и т.п.). Управляющие языки принтеров (фирм Epson, Hewlett-Paccard) основаны на таких Esc-последовательностях. Принцип формирования Esc-последовательности состоит в следующем: *если в потоке кодов встретился код ESC, то несколько следующих кодов не являются кодами изображаемых символов (хотя и могут совпадать с ними), а несут информацию о команде для устройства.*

Расширения кодовой таблицы

Коды	Назначение
128...175	Иностранные символы (для европейских языков с диакритическими знаками) и
176...213	Символы псевдографики
224...254	Научные символы

Русскоязычные кодировки

Использовали символы переключения регистров для замены части символов - латинские буквы заменяли русскими. Так делалось в стандартном ASCII. В поток символов вставляли, например SI и SO, которые трактовались как РУС и ЛАТ. В тексте, где русские и латинские буквы чередуются -до двух байтов на символ.

Существует несколько способов расположения русских символов.

1. **КОИ-8** – Кириллица перекрывает псевдографику, символы расположены не в алфавитном порядке. (с 192 кириллица заменяет латинские буквы: ю,а,б,с,д…) Электр. почта.

2. **CP1251** стандарт для Microsoft Windows (с 192 по 256 прописные и строчные)

3. **CP866** DOS (Альтернативная кодировка)- А-Я: 128-159; а-п: 160-175; р-я: 224-239.

Кодовые страницы

Этот подход - расширение идеи с переключением регистров или кодировок. Могут быть выбраны управляющие последовательности для смены кодовых страниц. Кодовые страницы - стандартизованы международно.

Стандарт Unicode

Стандарт Unicode использует два байта для изображения символа. Количество кодов - 65536. Текстовые файлы становятся **вдвое длиннее**. Однако многие текстовые редакторы сочетают текст с графикой и используют свои форматы с большим количеством управляющей информации - реальное увеличение объема - около 25%.

Требуется **значительный объем ОЗУ для хранения таблицы фонтов**. ArialCyr в Windows - занимает немного больше 50 кБайт на 256 символов. Фонт для 50000 символов - в 200 раз больше - 10 МБайт ???

Во многих алфавитах **символы составные**. Немецкий, французский - буквы с диакритическими знаками.

Иврит и арабский - **тексты справа-налево, а числа слева направо**. В арабском не два набора символов (строчные - прописные), а четыре.

В китайском, японском, корейском - **десятки тысяч иероглифов** ....

Структура таблицы Unicode

Коды	Описание
0...127	Нынешний ASCII

... 8191	Различные алфавиты: -латинский, - европейские- фонетический- кириллица - армянский, иврит, арабский, эфиопский,...
8192...12287	Знаки пунктуации, математические, технические символы, орнаменты
12288...16383	Фонетические и др. специальные символы китайского, корейского, японского языков
16384...59391	Китайские, корейские, японские иероглифы
59392...65024	Блок для частного использования
65025...65536	Блок обеспечения совместимости

Символьное представление чисел - универсальный формат межкомпьютерного общения.

Числа представляются кодами символов соответствующих цифр и букв:
-32.655E-3 - 10 байтов с кодами символов.
Такой формат одинаково воспринимается любой вычислительной системой: перед использованием числового значения, оно формируется системой путем преобразования из символьной строки во внутреннее представление.

Разметка текста

Для **разметки** текста могут использоваться два способа. Первый состоит в использовании специальных «не символьных» кодировок, в пределе приводящих к кодированию образа документа, как растрового или векторного изображения. Другой способ состоит в добавлении внутрь текста специальных последовательностей символов для разметки, которые должны интерпретироваться **не как символы**. Примеры языков/систем кодирования: SGML (Standard Generic Markup Language); HTML (HyperText Markup Language); T_EX (L^AT_EX); TROFF (Unix); PostScript; PDF.

Postscript

Postscript был разработан Джоном Уорноком и Чаком Гешке из Adobe Systems в начале 80-х гг. Исходно Postscript использовался как ядро механизма печати компьютеров Apple, но вскоре стал широко распространенным стандартом для большинства компьютерных систем. Интерпретаторы Postscript (в виде программных или аппаратных

компонентов) для печати документов присутствуют практически во всех современных компьютерных системах.

В Postscript используется модель изображения текста (или рисунков) на чистой странице. Когда страница готова, она выводится на печать и начинается «прорисовка» изображения очередной страницы. Это есть не что иное, как метод компиляции. Каждый документ Postscript включает в себя программу, которая печатает на принтере (или отображает на экране монитора) следующие друг за другом страницы.

Программа Postscript состоит из четырех компонентов:

1. Интерпретатор для выполнения вычислений. Основной моделью такого интерпретатора является простой стек постфиксного выполнения.

2. Синтаксис языка. Он основан на синтаксисе языка Forth.

3. Расширения для раскрашивания. Расширение языка Forth командами закрашивания для управления процессом отображения текста и рисунков на листе бумаги.

4. Соглашения. Набор соглашений, не входящих в официальный язык Postcript; которые используют различные принтеры для согласования представле документов. Использование этих соглашений упрощает передачу документов Postscript из одной системы в другую.

Postscript был разработан как архитектура виртуальной машины, предназначенной для создания печатных документов. В большинстве приложений не предполагается, что программист будет читать текст документа Postscript. Тем не менее, синтаксис Postscript достаточно прост и легок для восприятия. Существуют образовательные программы для обучения этому языку программирования. Его синтаксис и семантика отличаются простотой, а доступность программ для отображения на экране документов Postscript означает, что у любого пользователя имеется возможность доступа к интерпретатору виртуальной машины, на которой можно тестировать свои Postscript-программы. Следующим этапом развития

Postscript стало создание фирмой Adobe формата PDF (Portable Document Format — формат переносимых документов). PDF -это форма сжатия файлов Postscript. Программы чтения PDF-файлов свободно распространяются по Интернету, а большинство web-браузеров могут отображать PDF-файлы.

HTML

HTML вовсе не является языком программирования. HTML – это язык разметки. Вы используете HTML для разметки текстового документа, точно так же, как это делает редактор при помощи жирного красного карандаша. Эти пометки служат для определения формата (или стиля), который будет использован при выводе текста на экран монитора.

Если у вас есть желание выделить часть текста на Web-странице жирным шрифтом, вы отметите ее следующим образом:

<B>this text appears bold</B>

Символы <B> "включают" жирный шрифт, а </B> "выключают" его. Они называются тэгами (от англ. tag – ярлык, признак.) и не отображаются на экране. Они лишь предписывают выводить заключенный между ними текст жирным шрифтом.

2.5. Графические данные, их представление и кодирование

Задача кодирования изображений. Изображение, воспринимаемое зрительным анализатором человека (глазами), представляет собой образ динамической трехмерной сцены. С течением времени изменяются как свойства предметов и источников (такие, как яркость и отражающие свойства) так и их взаимное положение в пространстве. Глаз воспринимает световое поле E(x,y,z,t,f); x,y,z - координаты, t - время, f - частота света. Наиболее сложная задача, которую решают современные средства ВТ при отображении графики – это моделирование динамических трехмерных сцен.

Трехмерная (3D – образ) – означает, что в качестве исходных данных используется информация о положении объектов сцены и источников света в трехмерном пространстве.

Динамическая – означает, что объекты сцены могут перемещаться, и для последовательных отображаемых кадров значения яркостей / цветов некоторых (или всех) отображаемых точек придется пересчитывать.

Способы кодирования - зависят от того, для чего кодируем. Кодируют для следующих целей:

Обработка - доступ к пикселам, соответствие формы представления и свойств системы команд

Хранение - компактность, совместимость, независимость от конкретных типов устройства на котором изображение было сформировано и устройства, на котором оно может быть отображено.

Отображение - (как пишем в ВидеоRAM) - должно быть соответствие между размерностями выводимого изображения и размерностями устройства отображения во всех трех координатах.

Для перехода от непрерывного представления 2D изображения должны быть выполнены две операции:

1) Пространственная дискретизация по x и по y – (размер изображения - сколько писелов в строке/сколько строк). Параметры цвета в пределах одного пиксела принимаются постоянными.

2) Дискретизация цветовых компонент по уровню – представление непрерывной величины мощности одним из конечного ряда значений. Для представления цветовых компонент (см. далее) можно в зависимости от требуемой точности (верности воспроизведения цвета) использовать различное количество значений (и следовательно битов для их кодирования). Во многих используемых вариантах для кодирования одного цвета используется менее одного байта – кодирование битовыми полями.

Отметим, что цифровые цветные/полутоновые 2D изображения, полученные пространственной дискретизацией являются *растровыми*, т.е. в них описываются индивидуально характеристики каждого пиксела изображения.

Упрощение растрового многоуровневого можно проводить в двух направлениях:

1) Уменьшать количество уровней дискретизации цветов/тона – вплоть до двух, когда характеристика пиксела: “светится – не светится” – переход к бинарному изображению.

2) Индивидуально описывать не все пикселы изображения, а только принадлежащие “объектам” – переход к векторному описанию изображения.

Классы изображений

Бинарные изображения

По уровню - только два значения – (при преобразовании из полутонового – существенным будет выбор порога). В данном случае битовое поле, соответствующее одному пикселу имеет длину один бит.

Описываемые кривыми

Это есть переход к векторному описанию. Не кодируем каждый пиксел, а описываем только точки, принадлежащие элементам графических объектов (например, для задания окружности указываем координаты центра, радиус и атрибуты рисования – цвет линии и т.п.). Выбирается некоторый набор элементарных объектов (примитивов) – отрезок прямой, окружность, прямоугольник, …. Изображение «строится» из этих примитивов. Переход от растрового к векторному представлению (векторизация) является нетривиальной задачей.

Описываемые точками (многоугольниками)

Это частный случай векторного представления (достаточно простой). Только угловые точки фигур, в промежутках линейная интерполяция – т.е. изображение строится только из отрезков прямых.

Векторное и растровое представления - достоинства и недостатки

Когда говорят о векторном или растровом представлениях, имеют в виду 2D - образ.

Растровое представление получается в результате последовательного выполнения над непрерывным 2D изображением двух операций:

1) *Пространственная дискретизация*, в результате чего изображение делится на элементы - *пиксели* (*h* строк по *w* элементов в строке). Каждый пиксел характеризуется цветом, который получается путем усреднения цвета по площади пикселя и может быть представлен, например тремя компонентами R, G, B.

2) Квантование характеристики пиксела по уровню, в результате чего цветовая характеристика может быть представлена в заданной разрядной сетке *n* битов на пиксел.

После этого данные, описывающие изображение можно рассматривать как двумерный массив *n*-битовых элементов размером $w * h$. В частности, можно задать конфигурацию такого объекта, как символ, задав соответствующий массив маленькой размерности, описывающий прямоугольную область размером в знакоместо (например, 7 x 5 пикселов).

Векторное представление получается в результате выделения на непрерывном изображении *объектов* (точек, отрезков прямых, многоугольников, отрезков кривых, их комбинаций, конфигураций символов и т.п.). Каждый такой объект можно описать относительно небольшим набором величин (например, отрезок прямой - координатами концов, толщиной линии, цветом, ...). Например, конфигурацию буквы Р, можно задать, описав отрезок и полуокружность (полуэллипс).

Операция по выделению объектов на естественных изображениях (фотографиях, ...) достаточно сложна, неоднозначна, зависит от многих факторов. Такую операцию называют *сегментацией, векторизацией*. Векторизацию искусственно получаемых изображений проводить, как правило, легче, часто это делается непосредственно во время создания изображения. Многие графические редакторы описывают создаваемое изображение сразу в векторной форме (например, CorelDraw), в то время как другие формируют растровое представление (например, PaintBrush).

Векторное представление позволяет легче выполнять операции по обработке, такие как масштабирование, повороты,.... При масштабировании (уменьшении) растровых изображений можно «потерять» тонкие фрагменты. Но сложно делать векторизацию для естественных изображений, для "пестрых" картинок может оказаться, что векторное представление более объемное, чем растровое.

Кодирование цвета

В общем случае это распределение мощности светового потока по частотам. Свет представляет собой смесь гармонических (синусоидальных) электромагнитнных колебаний разных частот. На каждой частоте f колебание можно охарактеризовать интенсивностью (амплитудой) и фазой колебания: $y(t)=A sin(2\pi ft+\varphi)$. Суммарное колебание можно описать, задав зависимости амплитуды и фазы от частоты: $A(f)$ и $\varphi(f)$. Обратите внимание: чтобы исчерпывающим образом описать свет, излучаемый точечным источником, надо использовать две непрерывные (т.е. характеризуемые бесконечным множеством значений) функции амплитудного и фазового спектра (при этом мы еще опускаем возможность поляризации света).

В то же время известно, что в компьютерной технике характеристику света, характеризуют всего лишь тремя составляющими (например красной, зеленой и синей). Чтобы понять, почему это возможно, надо рассмотреть особенность человеческого зрения.

Трехкомпонентная гипотеза цветового зрения

Человеческий глаз имеет три типа цветовых анализаторов (R,G,B) Каждый из них имеет свою (непрерывную) частотную характеристику, с достаточно широкой полосой пропускания, а на выходе каждого цветового анализатора - сигнал, пропорциональный интегральной интенсивности излучения в полосе.

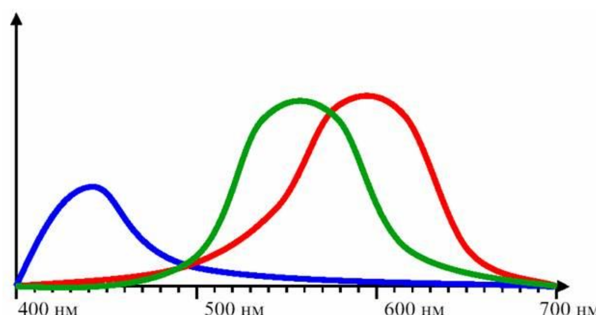


Рис. 2.2. Спектральная чувствительность элем. человеческого глаза.

Отсюда понятно, что получить данную величину сигнала интенсивности на выходе цветового рецептора можно, подавая на вход свет с разным спектральным составом, и в частности монохроматический свет. Только поэтому данный (широкополосный) свет воспринимается глазом субьективно так же, как сумма трех (не обязательно R,G,B) компонент. Таким образом, трехкомпонентное цветовое представление предназначено прежде всего для визуализации (т.е. для восприятия глазом) и может не годиться для других технических целей. В рамках трехккомпонентного представления цвета надо задавать три независимых величины интенсивности для каждой цветовой компоненты.

Насколько точно надо представлять каждую компоненту? Это зависит от назначения системы. Если она предназначена для субъективного восприятия - человек способен различать яркость соседних участков в монохромном изображении, когда она отличается на величину около 1%...0,5%. - **достаточно 8 битов на каждую цветовую компоненту**.

(R,G,B) - аддитивное представление

Цвет получается сложением компонент - как в кинескопе дисплея. В нем каждая из цветовых компонент R, G, B управляет интенсивностью отдельного электронного луча, определяющего яркость свечения пятна люминофора соответствующего цвета - яркости отдельных (соседних) пятен R, G, B **складываются**. При сложении всех компонент одинаковой интенсивности получаем белый цвет.

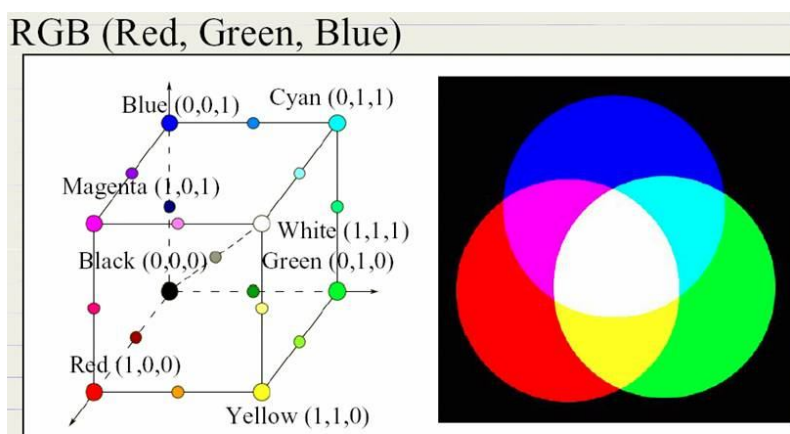


Рис. 2.3. Аддитивное кодирование цветов.

(C,M,Y,K) - субтрактивное представление

Цвет получается **вычитанием** компонент CYAN, MAGENTA, YELLOW (дополнительных к цветам R,G,B) из чисто белого. Такая система представления цвета соответствует формированию цвета в полиграфии: краски, нанесенные на белую бумагу, поглощают некоторые частоты падающего белого света, а остальная часть отражается и достигает глаза. При сложении всех компонент одинаковой интенсивности получаем черный цвет. Компонента K несет информацию о количестве черного, она теоретически избыточна, но практически используется (в типографских приложениях), поскольку реальные красители C,M,Y в сумме не дают черного цвета, а только серый.

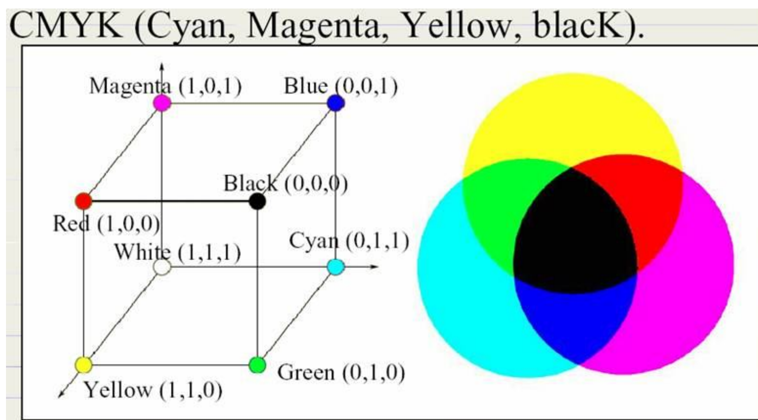


Рис. 2.4. Субтрактивное кодирование цветов.

Группа способов кодирования, в которых разделена информация о яркости (интенсивности) и о цвете (*Y,Cb,Cr*) - *Y* - luminosity яркость, *Cb* и *Cr* доли синего и красного. Такие представления позволяют с разной точностью кодировать яркость (более точно, т.е. большим количеством битов на пиксел) и цвет (менее точно). Это разумно, так как глаз менее чувствителен к относительному изменению интенсивностей отдельных цветов и более чувствителен к суммарной интенсивности.

$Y = 0.299R \quad +0.587G \ +0.114B$

$Cb = 0.1687R \quad -0.3313G +0.500B$

$Cr = 0.500R \quad -0.4187G +0.0813B$

Есть и другие способы кодирования, относящиеся к этой группе.

Способы и точность кодирования цвета для отображения

Способы и точность кодирования цвета имеет две взаимосвязанных стороны: 1) Количество представляемых цветов; 2) Как эти цвета выбраны во всем множестве цветов (в *RGB* - пространстве). Используются два способа:

1) **непосредственное** представление цветовых компонент

2) представление **с использованием таблицы цветов** Color Look-Up Table (CLUT)

При непосредственном представлении цветов каждый пиксел изображения представлен элементом данных, включающим три битовых поля *R, G, B*. Каждое поле содержит число, характеризующее интенсивность одной цветовой компоненты. Код этой интенсивности непосредственно используется для управления яркостью луча при отображении на дисплее. Размеры полей n_r, n_g, n_b определяют общее количество представимых цветов: $N = 2^n$, где $n=n_r+n_g+n_b$ - количество битов на один пиксел.

При длине элемента данных, описывающего пиксел, равной **одному байту** (8 битов) длины цветовых полей *R, G, B* обычно составляют соответственно 3, 3 и 2 бита, при этом общее количество представимых цветов равно 2^8=256. Меньшее количество битов для интенсивности синего в данном и в следующем представлениях объясняется меньшей чувствительностью глаза к изменениям синего.

При длине элемента данных пиксела в **два байта** (16 или 15 битов) цветовые компоненты обычно имеют длины 6(R), 6(G) и 4(B) бита, или 5(R), 6(G) и 5(B), или 5(R), 5(G) и 5(B), или 4(R), 4(G) 4(B и 4(Атрибут), а общее количество представимых цветов равно 2^{16}=65536 либо 2^{15}=32768. Это представление называют **HiColor**.

При большей длине пиксела используют для каждой цветовой компоненты поле длиной в байт, при этом количество градаций по каждому цвету составляет 2^8=256, а общее количество представимых цветов равно 2^{24} = 16777216 (около 16 миллионов). Это представление носит название **True Color**. Длина элемента данных, описывающего пиксел может составлять три, либо четыре байта, в последнем случае "лишний" байт используется для какой-либо дополнительной информации о пикселе.

При непосредственном представлении цветов точность представления цвета, набор и количество представимых цветов однозначно связаны. Это означает, например, что в формате "два байта на пиксел" 6(R), 6(G), 4(B)

количество градаций красного m_r и зеленого m_g равно по 2^6=64, количество градаций синего m_b равно2^4=16. а общее количество представимых на одном изображении цветов равно $m_r * m_g * m_b = 2^6 * 2^6 * 2^4 = 2^{16}$=65536/

При использовании представления с таблицей цветов код пиксела не содержит информации об интенсивности цветовых компонент, а лишь указывает строку в таблице цветов.

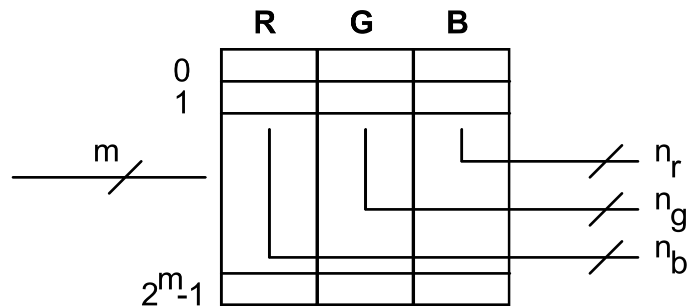


Рис. 2.5. Схема таблицы цветов.

Таблица цветов представляет собой память небольшого объема: *m* строк (называемых обычно регистрами) в каждой строке (регистре) $n_r+n_g+n_b$ битов. Обычно делают $n_r=n_g=n_b=n$. В памяти (на диске, в массиве изображения в ОЗУ, в видеопамяти и т.п.) одному пикселу соответствует элемент данных *m* битов длиной (чаще всего один байт). При этом на изображении возможно одновременно отображать не более 2^m цветов. Информация о цветовых компонентах этих цветов хранится в строках таблицы цветов. При этом "точность" задания цвета определяется длинами n_r, n_g, n_b полей. Регистры таблицы цветов программно доступны, таким образом набор отображаемых цветов (палитру) можно в любой момент переопределить программно. Всего возможно задать 2^{3n} разных оттенков, однако в каждый данный момент отображаются только 2^m из них.

Наиболее известная реализация системы с таблицей цветов: видеоадаптеры типа VGA. В них использована таблица размером 256 регистров x 3 * 6 бит/цвет. Одномоментно доступны 256 цветов из набора в 262144 цвета.

Форматы для хранения графических данных

Свойства изображений, которые хочется отобразить в графическом файле:

- чтобы система работала с разными устройствами ввода и с разными устройствами вывода и была инвариантна к различию их свойств;

- чтобы объем данных при хранении или передаче был минимальным – поэтому для хранения наиболее объемной части – собственно изображения используют разнообразные алгоритмы сжатия (компрессии) данных. (На компрессию / декомпрессию будет уходить дополнительное время.);

- чтобы формат представления собственно изображения требовал минимума действий для преобразования из входного формата в формат обработки, а затем в формат отображения;

- чтобы описывалось более чем одного изображение в одном файле - и тогда необходимо сохранять индивидуальные характеристики для каждого изображения, такие как таблица цветов, позиционирование,...

- кроме того, размещение изображения на "листе" (привязка к координатам вывода), interlacing чередование строк и др.

Имеется множество графических форматов, которые отличаются разными способами решения указанных задач (см. табл.). В целом графические форматы – это набор правил для стандартного хранения и представления графической информации. Графический файл содержит информацию не только о самом изображении, но также информацию о том, как эти данные должны интерпретироваться. Обычно графический файл содержит заголовок и блок данных. В заголовок включается информация об устройстве получения изображения (таблица цветов, количество бит на пиксел, используется ли сжатие и др.) и общие данные о самом изображении (размерность, смещение на экране и др.).

Табл. Графические форматы

Растровые для хранения и отображения	GIF	Graphics Interchange Format	Изображение до 16000x16000, сжимающий LZW. Interlessin (черезстрочность хранения)
	TIFF	Tagged Image File Format	Aldus и Microsoft Для издательских систем, сканеров сжимающий
	PCX	PC Paintbrush	Формат редактора изображений фирмы Z-Soft
	BMP	Bitmap for Windows	Сжатие RLE
Векторные для хранения и отображения	DXF	Drawing Interchange Format	AutoCad Секции: Заголовка. Таблиц. Блоков. Элементов
Сжимающие для хранения	MPEG	Motion Pictures Expert	Правила кодирования и декодирования потоков изображений и звука
	JPEG	Joint Photographic Experts Group	Статические изображения. Схема кодировки: RGB->YUV, Блоки, Дискретное косин. преобразование, Усечение, Сжатие.

Краткая характеристика наиболее распространенных растровых форматов

1) Формат PCX (PCExchange), разработан PCPaintBrush, является одним из самых известных и старых. Практически любое приложение легко импортирует его. Он не позволяет хранить цветоделенные CMYK-изображения и цветовые профили, что делает невозможным его применение при создании цветных публикаций. Является устаревшим, вытеснен усовершенствованными форматами GIF и TIFF.

2) Формат BMP (Bitmap) предназначен для Windows, и поддерживается всеми приложениями, работающими в этой среде. Позволяет хранить

полноцветные изображения в цветовой модели RGB и индексированные изображения. Не применяется в издательской деятельности, но широко используется в оформлении прикладных программ.

3) Формат JPEG (JointPhotographicExpertsGroup) предназначен для сохранения растровых файлов со сжатием. Сжатие по этому методу уменьшает размер файла от десятых долей процента до ста раз (практический диапазон - от 5 до 15), но при этом происходит потеря качества (в большинстве случаев эти потери находятся в пределах допустимых). Распаковка JPEG-файла происходит автоматически во время его открытия. Формат поддерживает только полутоновые и полноцветные изображения в моделях RGB и CMYK. Допускается сохранение контуров обтравки и цветовых профилей. Формат не позволяет использовать анимацию и прозрачность. Обычно формат JPEG применяется для хранения высококачественных фотографий. Формат JPEG позволяет использовать до 16 миллионов цветов.

4) Формат GIF (GraphicsInterchangeFormat) в издательских целях не применяется,однако очень широко распространен на Web. Допускает хранение в одном файле нескольких изображений. Чаще всего такая возможность используется на страницах Web. Web-браузер демонстрирует изображения, находящиеся в файле GIF последовательно. Если каждое изображение представляет собой фазу мультипликации, то вы увидите маленький мультфильм. Формат способен хранить только индексированные изображения. Стандартный фильтр экспорта в. формат GIF поддерживает единственную особенность формата- чересстрочную развертку. Чересстрочная развертка используется браузерами: по мере загрузки в изображении появляется все больше деталей. Это дает возможность пользователю еще в процессе загрузки изображений решить, стоит ли дожидаться ее завершения или перейти к следующей странице.

5) Формат PNG (PortableNetworkGraphics) предназначен для передачи изображений в сетях. Поддерживает полноцветные изображения RGB и индексированные изображения. Возможно использование единственного дополнительного канала для хранения маски прозрачности. Имеет эффективный алгоритм сжатия без потери информации. Этот формат тоже применяется на Web

6) Формат PCD (Photo CD). Изображения запоминаются всегда в альбомной ориентации. Дает при импорте определять разрешение изображения.

7) Формат PSD (Adobe Photoshop Document) является внутренним форматаом программы AdobePhoroshop. Удобен для общения с другими продуктами фирмы Adobe. Поддерживает все сведения о документе, но пока недостаточно распространен.

8) Формат TIFF (Tagged Image File Format) создан как универсальный формат для сканированных изображений. Переносим на разные платформы. Импортируется парктически во все издательские системы. Поддерживает алгоритмы сжатия без потерь.

9) Формат EPS (Encapsulated PostScript) описывает изображение на универсальном языке PostScript. Описывает не только растровые, но и векторные изображения, а также текст. Предпочтителен для полиграфических целей. Имеет большой размер файла.

10) Формат DCS позволяет вставлять изображения, разделенные на плашечные цвета. Является вариантом формата EPS.

11) Формат PDF (Portable Document Format) предложен фирмой Adobe как независимый от платформы формат, в котором могут быть сохранены иллюстрации (векторные и растровые) и текст, причем с множеством шрифтов и гипертекстовых ссылок. Для достижения продекларированной

в названии переносимости (portable), размер PDF-файла должен быть малым. Для этого используется компрессия - к каждому виду объектов применяется свой способ. Для работы с этим форматом компания Adobe выпустила пакет Acrobat. Acrobat Distiller переводит в PDF PostScript-файлы, Acrobat Exchange позволяет их редактировать: устанавливать внутренние ссылки, ссылки на внешние звуковые и видеофайлы, Web-ссылки. Ряд программ также позволяют создавать PDF'ы. Первоначальная задача PDF - передача по сети в сжатом виде проиллюстрированных и отформатированных документов - сегодня значительно расширена. Кроме того, в PDF можно быстро передавать клиенту полноценные эскизы. PDF позволяет не заботиться о наличии необходимых шрифтов у получателя - все подгружается прямо в файл.

Алгоритмы сжатия

Алгоритмы сжатия без потерь

Групповое кодирование RLE

Серия повторяющихся величин заменяется двумя: *значением* и *количеством*. Цепочка из *n* одинаковых байтов (при *байт/пиксел*) будет заменена двумя байтами. Хорошо кодируются изображения с большими областями постоянной закраски (из-под "рисовалок" типа PaintBrush).

Кодирование по Хаффману (Huffman, 1952)

Используются коды переменной длины, причем более короткие комбинации для более часто встречающихся величин. Для эффективного кодирования надо иметь статистику: как часто встречаются разные значения - поэтому кодирование "в два прохода". Степень сжатия также зависит от типа изображения - плохо работает для файлов, содержащих длинные последовательности одинаковых пикселов. Процессы кодирования и декодирования - сравнительно медленные процессы. Алгоритм очень чувствителен к "потере" битов в закодированных данных.

Алгоритм LZW (Lempel-Ziv-Welch - 1984)

Используется в архиваторах. Подобно алгоритму Хаффмана заменяет длинные последовательности более короткими, но не требует предварительно собирать статистику, он формирует все более эффективную таблицу кодирования по мере продвижения по шифруемому массиву. Более "шумные" изображения кодируются хуже. Поэтому иногда рекомендуется подавить низкочастотной фильтрацией высокие пространственные частоты на изображении Типичные коэффициенты сжатия между 1:1 и 1:3, хотя иногда м.б. и 1:10. Этот алгоритм используется в графических форматах GIF и TIFF.

Арифметическое сжатие (1979...1989).

Подобен алгоритму Хаффмана - использует более длинные кодовые последовательности для более редких входых последовательностей

Алгоритмы сжатия с потерями

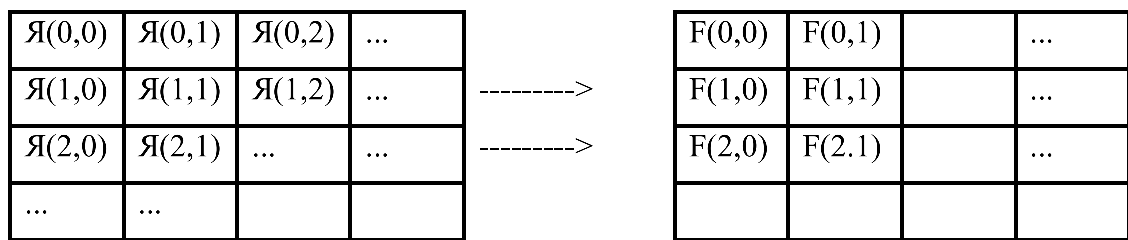
В отличие от алгоритмов для архивации программ, числовых и текстовых данных при сжатии изображений допустимы искажения, которые незаметны или малозаметны для глаза. Применение - для сжатия фото и кино-изображений.

Алгоритм JPEG Joinnt Photographic Experts Group - комитет ISO

Метод сжатия включает следующие шаги:

1) Представление RGB преобразуется в представление YCbCr (формулы были даны ранее в этом тексте)

2) Выполняется дискретное косинутсное преобразование для квадратных фрагментов изображения (обычно 8x8) отдельно для яркости и отдельно с меньшей пространственной дискретностью для цветовых компонент.



Я(0,0)	Я(0,1)	Я(0,2)	...
Я(1,0)	Я(1,1)	Я(1,2)	...
Я(2,0)	Я(2,1)	...	...
...	...		

--------->

--------->

F(0,0)	F(0,1)		...
F(1,0)	F(1,1)		...
F(2,0)	F(2.1)		...

3) Полученные коэффициенты округляются (это позволяет представить их меньшим количеством битов (на этом этапе - основные потери). Коэффициенты при высоких частотах при этом могут оказаться нулевыми - их можно отбросить.

4) Последовательность коэффициентов кодируется по Хаффману или алгоритмом арифметического кодирования.

При декодировании этапы выполняются в обратном порядке. Достижимые коэффициенты сжатия - 1:10...1:50 при ухудшающемся качестве.

MPEG - международный комитет Motion Pictures Expert ISO в 1993г

MPEG - разработан международным комитетом Motion Pictures Expert и принят в окончательной редакции ISO только в 1993г. Хотя MPEG-стандарт определяет правила кодирования и декодирования цифровых потоков как изображений, так и связанного с ними звука, в этом материале мы остановимся только на изображении. Стандарт JPEG отличается от MPEG тем, что проводит независимое сжатие каждого кадра изображения.

Компрессия использует следующие основные идеи:

- устранение временной избыточности видео, учитывающее тот факт, что в пределах коротких интервалов времени большинство фрагментов сцены оказываются неподвижными или незначительно смещаются по полю (на нижнем рис. 2.6 показаны два соседних кадра из фильма Терминатор-2 и разностный кадр – кодируем разностный кадр);

- устранение пространственной избыточности изображений подавлением мелких деталей сцены, несущественных для ее визуального

восприятия человеком (цвет большинства точек неба одинаков – кодируем их одинаково);

- использование более низкого цветового разрешения при YUV-представлении изображений (Y -яркость, U и V - цветоразностные сигналы) - установлено, что глаз менее чувствителен к пространственным изменениям оттенков цвета по сравнению с изменениями яркости; повышение информационной плотности результирующего цифрового потока путем выбора оптимального математического кода для его описания (например, использование более коротких кодовых слов для наиболее часто повторяемых значений).

Рис. 2.6. Два соседних кадра и разностный кадр.

На первой ступени поток видео разделяется на кадры изображения:

- I (Intra), независимо сжатые; опорные при восстановлении остальных изображений по их разностям;

- P (predicted), сжатые с использованием ссылки на одно изображение – содержащие разность текущего изображения с предыдущим I или P с учетом смещений отдельных фрагментов;

- B (bidirertlonally predicted), сжатые с использованием ссылки на два изображения – содержащие разность текущего изображения с предыдущим и последующим изображениями типов I или P с учетом смещений отдельных фрагментов;

Изображения объединяются в группы (GOP Group Of Pictures), представляющие собой минимальный повторяемый набор последовательных изображений, которые могут декодированы независимо от других изображений в последовательности.

Типичной является группа вида (I0 B1 B2 P3 B4 B5 P6 B7 B8 P9 B10 B11) (I12 B13 B14 P15 B16 B17 P18...), в которой I-тип повторяется каждые полсекунды. Обратим внимание, что в изображении P3 основная часть фрагментов сцены предсказывается на основании соответствующих смещенных фрагментов изображения I0. Собственно кодированию подвергаются только разности этих пар фрагментов. Аналогично P6 строится на базе P3, P9 - на базе P6 и т.д. В то же время большинство фрагментов B4 и B2 предсказываются как полусумма смещенных фрагментов из I0 и P3, B4 и B5 - из P3 и P6, B7 и B8 - из P6 и P9 и т.д. В то же время B-изображения не используются для предсказания никаких других изображений.

Отдельные изображения состоят из макроблоков. Макроблок - это основная структурная единица фрагментации изображения. Он соответствует участку изображения размером 16x16 пикселей. Именно для них определяются вектора смещения относительно I- или P-изображений. В свою очередь каждый макроблок состоит из шести блоков, четыре из которых несут информацию о яркости Y, а по одному определяют цветовые U- и V-компоненты. Каждый блок представляет собой матрицу 8x8 элементов.

Блоки являются базовыми структурными единицами, над которыми осуществляются основные операции кодирования, в том числе выполняется дискретное косинусное преобразование (DCT discrete cosine transform) и квантование полученных коэффициентов. Кодирование блоков похоже как в алгоритме JPEG.

Основные пути повышения степени сжатия

Изменение алгоритма сжатия I-кадров. Выше приведен алгоритм, основанный на ДКП. Сегодня все чаще используются алгоритмы, основанные на вэйвлетах (см. JPEG-2000).

Изменение алгоритма сжатия без потерь.

Изменение алгоритма работы с векторами смещения блоков. Подбор векторов смещений блоков по наименьшему среднеквадратичному смещению не является оптимальным. Существуют алгоритмы дающие лучший результат при некоторых дополнительных затратах времени при сжатии.

Применение обработки коэффициентов. Можно пытаться получить больше информации об изображении из сохраненных коэффициентов. Например, возможно быстрое сравнение коэффициентов после ДКП в соседних блоках и их усреднение по достаточно сложным алгоритмам.

Применение обработки получающихся кадров. Известная беда алгоритмов сжатия изображений - неизбежная "блочность" (хорошо заметные границы макроблоков, эффект Гиббса). Однако можно построить быстрые алгоритмы постобработки, которые уберут видимые границы между блоками, не внося существенных помех в само изображение В принципе существуют алгоритмы, работающие совсем без применения блоков (даже без векторов смещения блоков), но такой подход пока не оправдывает себя ни по степени сжатия, ни по скорости работы декодера.

Улучшение алгоритмов масштабирования изображений. Как правило, видео на компьютере просматривают во весь экран. Появляется возможность использовать более сложные и качественные алгоритмы масштабирования на весь экран.

Применение предварительной обработки видео. Если мы хотим получить достаточно высокую степень сжатия, то можно заранее предсказать, что в нашем изображении пострадают высокие частоты. Оно станет сглаженным, пропадут многие мелкие детали. При этом, как правило, появляются дополнительные артефакты в виде полосок, ореолов у резких границ, волн. Значительно повысить качество изображения после кодирования позволяет предобработка с удалением высоких частот. При этом существуют алгоритмы, обрабатывающие поток таким образом, что визуально качество изображения не изменяется, однако после декодера мы получаем существенно более качественное изображение.

Motion-JPEG

Motion-JPEG (или M-JPEG) является наиболее простым алгоритмом сжатия видео. В нем каждый кадр сжимается независимо алгоритмом JPEG. Этот прием дает высокую скорость доступа к произвольным кадрам, как в прямом, так и в обратном порядке следования. Соответственно легко реализуются плавные "перемотки" в обоих направлениях, аудио-визуальная синхронизация и, что самое главное - редактирование. Типичные операции JPEG сейчас поддерживаются на аппаратном уровне большинством видеокарт и данный формат позволяет легко оперировать большими объемами данных при монтаже фильмов. Независимое сжатие отдельных кадров позволяет накладывать различные эффекты, не опасаясь, что взаимное влияние соседних кадров внесет дополнительные искажения в фильм.

ГЛАВА 3. ПРОЦЕССОР

3.1. Процессор – аппаратный уровень. Операционные устройства

Процессор – это ядро и мотор компьютера, обеспечивает как выполнение операций, так и функционирование всех узлов. Обычно простой процессор включает в себя декодер команд, операционное устройство (АЛУ), регистры, транслятор адресов операндов и устройство управления (рис.3.1). Отдельные элементы связаны внутренней шиной. Связь с внешними устройствами осуществляется через внешнюю шину.

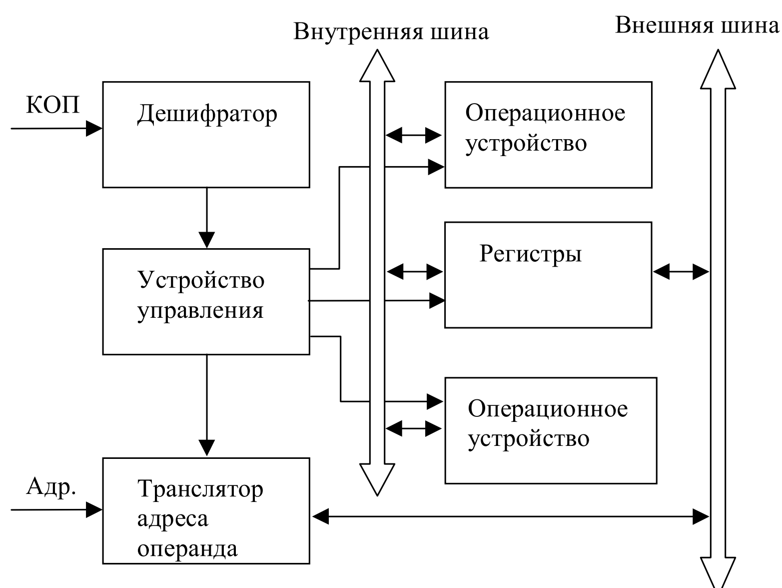


Рис.3.1. Структура процессора.

Все перечисленные элементы могут помещаться в одной микросхеме - микропроцессоре. Они доступны через выводы микросхемы при выполнении команд, загружаемых из ОЗУ. На рис. 3.2. показаны выводы типичного микропроцессора. Рассмотрим как работает процессор изнутри, на аппаратном и микропрограммном уровне. Автор считает, что в данном курсе не ставится цель подробного описания и анализа операционных устройств и устройств управления, хотя ключевые элементы будут отмечены.

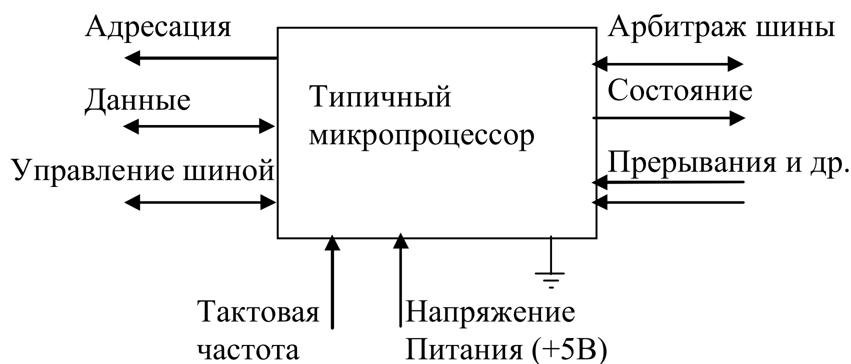


Рис. 3.2. Типичный процессор

Аппаратный уровень процессора в настоящее время не является ни программно доступным, ни даже наблюдаемым, однако представление о нем помогает пониманию таких вопросов, как скорость выполнения команд, возможность конвейеризации, ...

Для того чтобы понять как выполняются команды в процессоре рассмотрим работу операционных устройств. В машине фон Неймана в качестве операционного устройства применяется АЛУ. Учитывая разнообразие выполняемых операций и обрабатываемых данных, можно говорить не об одном устройстве, а о целом комплексе специализированных операционных устройств, каждое из которых выполняет свое подмножество операций (команд). Следует выделить операционные устройства:

- целочисленной арифметики
- логических операций
- десятичной арифметики
- чисел с плавающей запятой

Кроме указанных устройств процессоры в зависимости от их назначения могут иметь и другие операционные устройства: управления потреблением, графических операций, упаковки/распаковки изображений и др. В минимальном варианте операционное устройство целочисленной арифметики (АЛУ) должно содержать аппаратуру для реализации лишь основных логических операций, сдвигов, инвертирования, а также сложения

чисел в формате с фиксированной запятой. Опираясь на этот набор, можно программным способом обеспечить выполнение остальных арифметических и логических операций как для чисел с фиксированной запятой, так и для других форм представления информации. Следует отметить, что подобный вариант не позволяет добиться высокой скорости вычислений, поэтому по мере расширения технологических возможностей доля аппаратных средств в составе АЛУ постоянно возрастает.

Набор элементов, на основе которых строятся структуры различных операционных устройств, называется *структурным базисом*. Структурный базис операционных устройств включает в себя:

- регистры, обеспечивающие хранение слов данных;
- шины, связывающие регистры и предназначенные для передачи слов данных;
- комбинационные схемы, реализующие вычисления по управляющим сигналам от устройства управления.

Выполнение команды может быть сведено к нескольким (одновременно, либо последовательно в времени) операциям пересылки из регистра в регистр возможно с промежуточным преобразованием пересылаемых двоичных слов на комбинационных логических схемах.

Регистры. Хранят двоичное слово. Регистр это линейка триггеров, которые имеют входы для изменения состояния, которое влияет на выходы. Регистры могут быть:

- программно-видимые явно
- видимые косвенно, такие как "теневые" регистры дескрипторов сегментов
- внутренние для специальных целей, например регистр адреса памяти и регистр данных для обмена с памятью, про них покрайней мере известно, для чего они...
- внутренние для хранения внутренних промежуточных результатов...

Кроме того, в процессоре могут быть "псевдо-регистры", хранящие константы: 0, 1, -1, и др. (м.б. такие, как π, *e,...*).

Шины. Это просто группа параллельных проводов, связывающих между собой два адресата и позволяющих передать слово данных параллельным кодом.

Вентили (Gates). Это логические элементы (коньюнкторы), разрешающие передачу данных (например на вход триггера).

Будем в дальнейшем изображать вентиль схематически следующим образом:

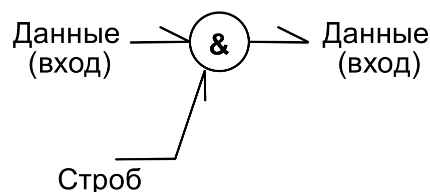


Рис.3.3. Структура схема вентиля.

АЛУ - это комбинационная схема, (т.е. она не содержит внутри элементов памяти)

- принимающая на два входа два операнда (например, содержимое двух регистров) и
- формирующая на выходе результат операции.

Большинство компьютеров содержат одну общую схему для выполнения операций И, ИЛИ и сложения над двумя машинными словами. Обычно такая схема для n-битных слов состоит из n идентичных схем для индивидуальных битовых позиций. На рис. 3.4 изображена схема [11] одноразрядного АЛУ. Это устройство может вычислять одну из 4 следующих функций: А И В, А ИЛИ В, В- и А + В. Выбор функции зависит от того, какие сигналы поступают на линии F_0 и F_1: 00, 01,10 или 11 (в двоичной системе счисления). Отметим, что здесь А+В означает арифметическую сумму А и В, а не логическую операцию И.

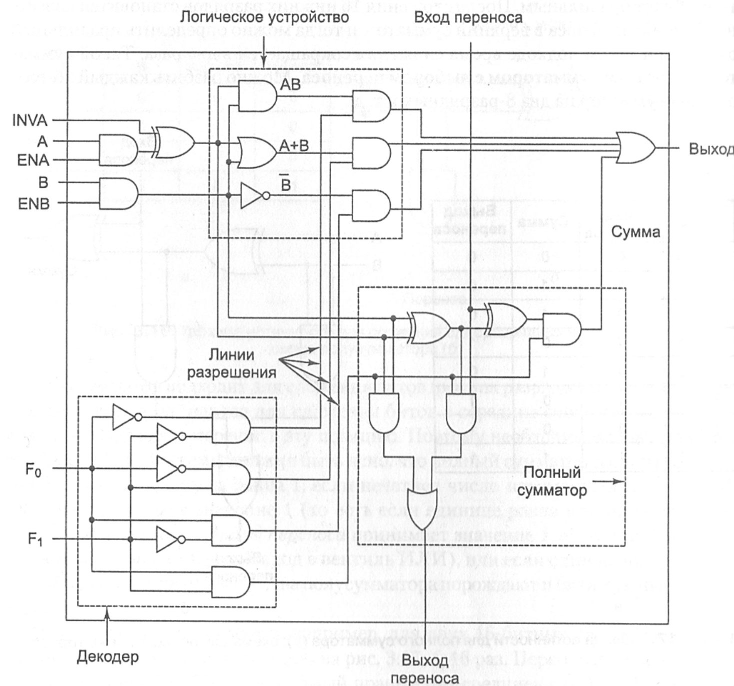


Рис. 3.4. Схема одноразрядного АЛУ

В левом нижнем углу схемы находится двухразрядный декодер, который порождает сигналы включения для четырех операций. Выбор операции определяется сигналами управления F_0 и F_1. В зависимости от значений Fo и F1 выбирается одна из четырех линий разрешения, и тогда выходной сигнал выбранной функции проходит через последний вентиль ИЛИ.

В нижнем правом углу находится полный сумматор для подсчета суммы A и B и для осуществления переносов. Переносы необходимы, поскольку несколько таких схем могут быть соединены для выполнения операций над целыми словами. Одноразрядные схемы, подобные той, которая изображена на рис. 3.4, называются разрядными микропроцессорными секциями. Они позволяют разработчику сконструировать АЛУ любой желаемой ширины.

Примеры выполнения элементарных действий

Комбинирование перечисленных узлов (шины, вентили, АЛУ, мультиплексоры) позволяет выполнять различные операции.

Пересылка

Это действие (этап "исполнение" команды mov R2, R1) реализует структура, изображенная на рисунке 3.3. Изобразим ее проще.

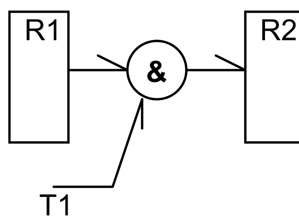


Рис. 3.5. Схема устройства исполнения команды *mov*.

Для исполнения этого действия достаточно одного стробового сигнала T1.

Сдвиг

Приведенная схема реализует логический сдвиг влево rol R1. Линии данных с выхода вентиля &3 поданы на вход регистра R1 со смещением в сторону старших разрядов, а на линию младшего разряда подан 0.

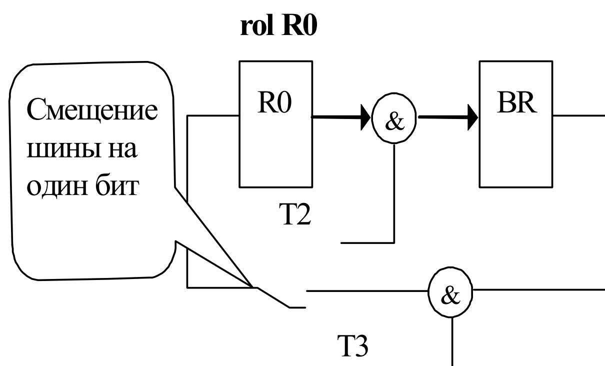


Рис. 3.6. Схема устройства исполнения команды сдвига.

Для исключения "гонок" операция проводится за два такта: сначала строб T2 переписывает содержимое регистра R1 во внутренний регистр Ri, а затем строб T3 возвращает операнд в R1 (со смещением на один разряд). Таким образом, исполнение команды "сдвиг" происходит за два такта.

Сложение

В состав операционного устройства входят три регистра со своими логическими схемами - это регистр первого слагаемого RA, регистр второго слагаемого RB и регистр результата RC. Этап исполнения команды **add R0, R1**. Содержимое операндов из регистров R0 и R1 надо подать на входы сумматора - это выполняют стробы T4 и T5, подаваемые одновременно. Операнды из регистров R0, R1 передаются во внутренние буферные регистры АЛУ RA, RB. На управляющий вход АЛУ после этого надо подать

строб T6, разрешающий АЛУ операцию сложения. Через время, не меньшее, чем задержка АЛУ, подается строб T7, разрешающий запись результата с выхода АЛУ в регистр R1.

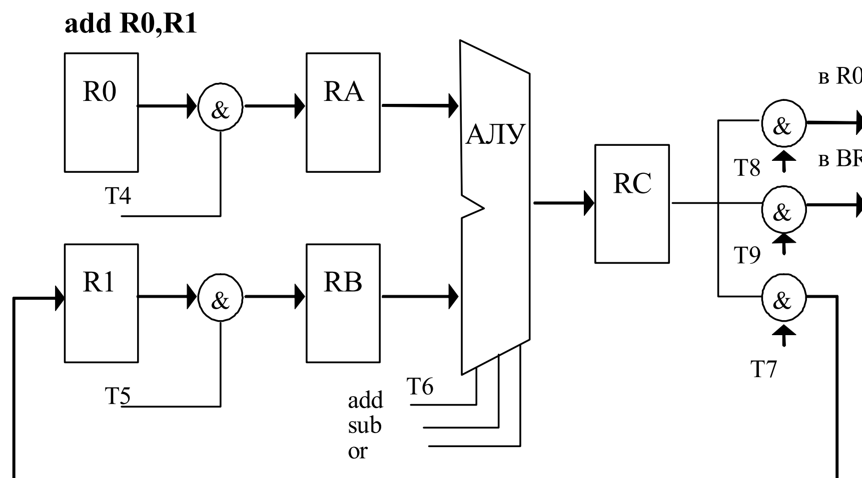
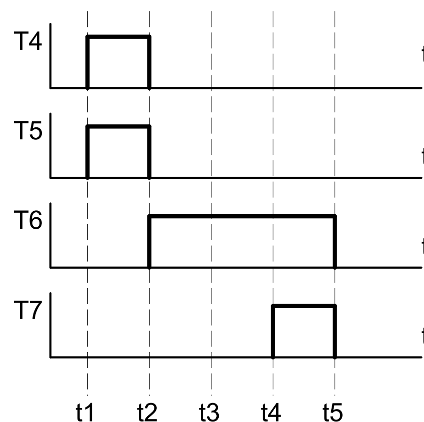


Рис. 3.7. Структурная схема АЛУ

Временная диаграмма стробов может иметь следующий вид:



В момент t1 подаются стробы T4 и T5, длительность их (t2-t1) должна быть достаточной для надежного переключения триггеров в регистрах RA, RB. В момент t2 подается разрешение на АЛУ, время задержки которого (предположим) равно t4-t2. В момент t4 строб T7 переписывает результат сложения с выхода АЛУ в регистр-приемник R1.

Отметим, что каждое из перечисленных действий требует некоторого минимального времени, зависящего от быстродействия логических элементов, причем для разных действий это время, вообще говоря, различно.

Система формирования стробов в процессоре основана на дискретной периодической сетке, формируемой тактовым генератором.

Операционные устройства (ОПУ) с магистральной структурой

Описанное выше устройство является с жесткой структурой. Более широкое применение находят ОПУ с магистральной структурой. В них все внутренние регистры объединены в отдельный узел регистров общего назначения (РОН), а все комбинационные схемы в операционный блок (ОПБ), который часто ассоциируют с термином арифметико-логическое устройство. Операционный блок и узел регистров сообщаются между собой с помощью магистралей – отсюда и название “магистральное ОПУ”.

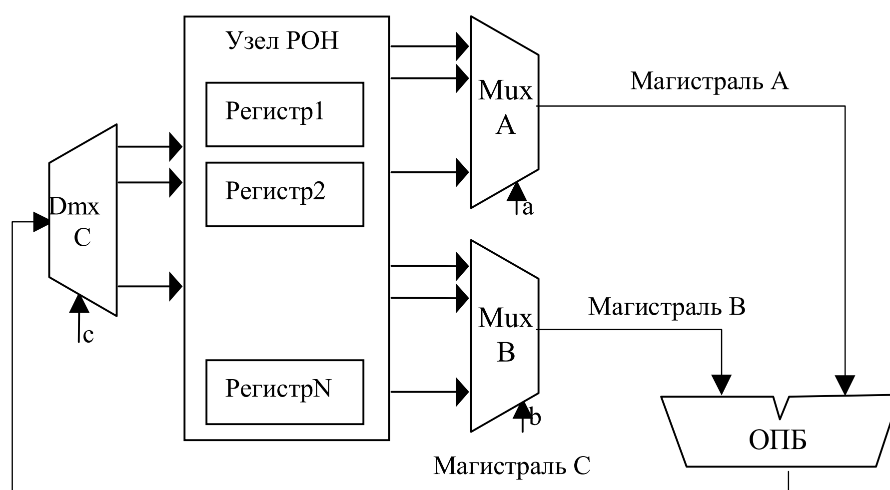


Рис. 3.8. Магистральное операционное устройство

В состав РОН входят N регистров общего назначения, подключенных к магистралям А и В через мультиплексоры MuxA и MuxB и к магистрали С через демультиплексор DmxC. Каждый из мультиплексоров является управляемым коммутатором, соединяющий выход одного из регистров с соответствующей магистралью. Демультиплексор коммутирует один информационный вход на n информационных выходов. Номер подключаемого регистра определяется адресом a или b, подаваемым на адресные входы мультиплексора (демультиплексора) из устройства управления. По магистралям операнды поступают на входы операционного

блока, где выполняется операция. Результат по магистрали С заносится через демультиплексор в конкретный регистр узла РОН.

Магистральные ОПУ различаются по виду и количеству магистралей, организации узла РОН, типу ОПБ. По функциональному назначению выделяют: магистрали внешних связей, соединяющие ОПУ с памятью и каналами ввода-вывода; внутренние магистрали ОПУ, отвечающие за связь между узлом РОН и операционным блоком. Количество магистралей зависит от архитектуры ЭВМ и обычно не превышает двух для внешних связей и трех – для внутренних.

Операционные устройства с плавающей запятой.

Операции над числами с плавающей запятой имеют существенные отличия от операций целочисленной арифметики, поэтому их реализуют с помощью самостоятельного операционного устройства (ОП). Минимальный набор операций этого устройства включает четыре действия: сложение, вычитание, умножение и деление. Операции выполняются над числами в описанном ранее (глава 2) формате IEEE 754.

Особенностью ОП является то, что операции над тремя составляющими чисел с ПЗ (знаками, мантиссами и порядками операндов) выполняются раздельно: блоком обработки знаков, : блоком обработки порядков, блоком обработки мантисс. Для хранения операндов и результата предусмотрены соответствующие регистры.

В арифметике с плавающей запятой сложение и вычитание – более сложные операции, чем умножение и деление. Это обусловлено тем, что необходимо выравнивать порядки операндов. Алгоритм сложения включает следующие этапы:

1. Подготовительный этап.
2. Определение операнда, имеющего меньший порядок, и сдвиг его мантиссы вправо на число разрядов, равное разности порядков операндов.

3. Приравнивание порядка результата большему из порядков операндов.
4. Сложение мантисс.
5. Проверка на переполнение
6. Заключительный этап.

Подготовительный этап осуществляет “распаковку” чисел с ПЗ, загрузку трех составляющих. В старший разряд мантиссы восстанавливается единица. Здесь может быть выполнена проверка на равенство нулю одного или обоих операндов. Это позволяет исключить ненужные операции. Действия на заключительном этапе сводятся к выявлению нулевого значения мантиссы, нормализации мантиссы, выявлению отрицательного переполнения порядка (появляется денормализованный операнд), “упаковке” составляющих результата.

В отличие от целочисленной арифметики, в операциях с ПЗ сложение и вычитание производятся приближенно, так как при выравнивании порядков происходит потеря младших разрядов одного из слагаемых. В этом случае погрешность всегда отрицательна и может доходить до единицы младшего разряда.

3.2. Устройство управления. Микропрограммный автомат.

Устройство управления (УУ) ЭВМ реализует функции управления ходом вычислительного процесса, обеспечивая автоматическое выполнение команд программы. Входной информацией для УУ служат:

- тактовые импульсы – с каждым импульсом УУ выполняет одну или несколько микроопераций;
- код операции – поступает из регистра команды и используется, чтобы определить, какие микрооперации должны выполняться;
- флаги – требуются для оценки состояния процессора, что необходимо для выполнения команд условного перехода;

- сигналы из системной шины – обслуживание прерывания, прямого доступа в память.

В свою очередь УУ, а точнее микропрограммный автомат, формирует следующую выходную информацию:

- внутренние сигналы управления – перемещают операнды из регистра в регистр инициируют работу операционных устройств;
- сигналы в системную шину – управляющие сигналы в память или в устройства ввода-вывода.

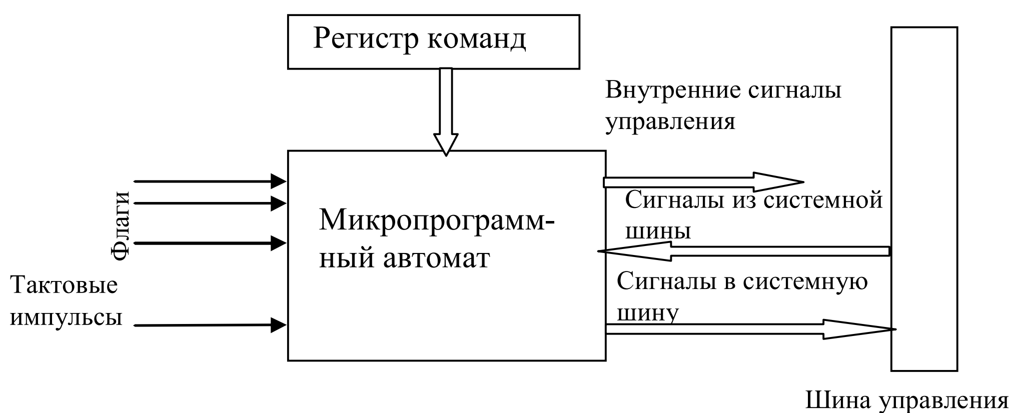


Рис. 3.12. Модель устройства управления

Как отмечалось, процесс функционирования ЭВМ состоит из последовательности элементарных действий в ее узлах. Такие элементарные действия, выполняемые в течение одного такта сигналов синхронизации, называются *микрооперациями*. Совокупность одновременно выполняемых микроопераций образует *микрокоманду*. Последовательность микрокоманд, определяющая порядок реализации машинного цикла, составляет *микропрограмму*. Микропрограммный автомат определяет микропрограмму как последовательность выполнения микроопераций.

В общей структуре УУ можно выделить две части: управляющую и адресную. Управляющая часть предназначена для координирования работы операционного блока, адресной части, основной памяти и др. Адресная часть обеспечивает формирование адресов команд и адресов операндов в основной памяти.

Состав управляющей части: регистр команды, микропрограммный автомат, узел прерываний и приоритетов. Микропрограммный автомат на основе декодирования команды вырабатывает определенную последовательность микрокоманд. В зависимости от способа формирования микрокоманд различают микропрограммные автоматы с жесткой и программируемой логикой.

Адресная часть УУ включает в себя: операционный узел устройства управления (ОПУУ), регистр адреса, счетчик команд. Регистр адреса используется для хранения исполнительных адресов операндов, а счетчик команд – для выработки и хранения адресов команд. Их содержимое посылаются в регистр адреса основной памяти. ОПУУ обрабатывает адресные части команд, формируя исполнительные адреса операндов, а также подготавливает адрес следующей команды при выполнении команд перехода. Иногда ОПУУ совмещают с основным операционным устройством.

Микропрограммный автомат с жесткой логикой.

Тип микропрограммного автомата определяет название всего УУ. В микропрограммном автомате с жесткой логикой выходные сигналы управления реализуются за счет однажды соединенных логических схем.

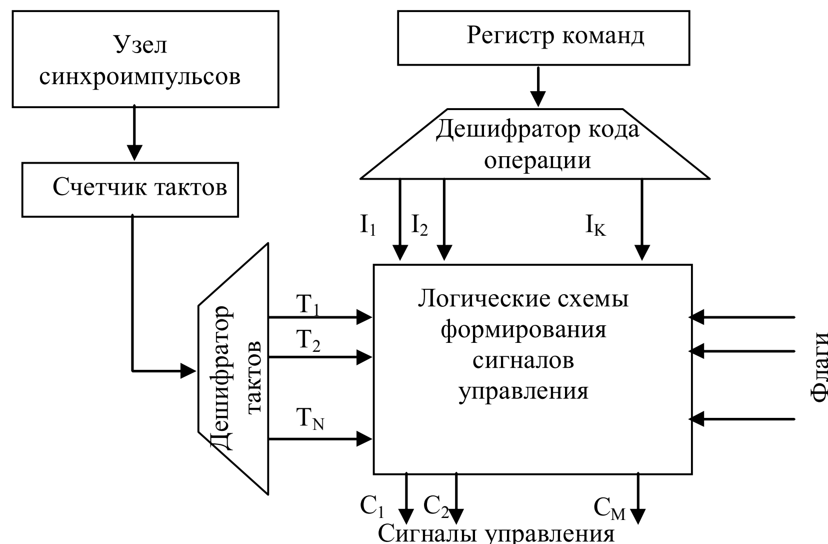


Рис. 3.13. Микропрограммный автомат с жесткой логикой

Код операций (КОП), хранящийся в регистре команды, используется для определения того, какие сигналы управления и в какой последовательности должны формироваться. При этом, желательно иметь в УУ отдельный логический сигнал для каждого кода операции (I_K). Это может быть реализовано с помощью дешифратора. Сигналы управления, по которым выполняется микрооперация, должны вырабатываться в строго определенные моменты времени, поэтому все сигналы управления “привязаны” к импульсам синхронизации. Счетчик тактов сбрасывается (устанавливается в состояние T_1) по окончании цикла очередной команды. Цикл команды может потребовать разного количества тактов. На каждом такте вырабатывается своя микрокоманда, состоящая из нескольких сигналов управления. Дополнительным фактором, влияющим на выработку сигналов управления, являются флаги.

Микропрограммный автомат с программируемой логикой.

Отличительной особенностью микропрограммного автомата с программируемой логикой является наличие памяти микропрограмм. Каждой команде вычислительного устройства в этой памяти соответствует микропрограмма.

Идею микропрограммирования сигналов управления предложил в 1951 г. Морис Уилкс (Кембриджский университет, Британия). ЭВМ стала иметь три уровня выполнения команд: между командами и сигналами управления появилась микропрограмма. Команда ЭВМ интерпретировалась в микропрограмму. Аппаратное обеспечение должно было выполнять только микропрограммы с ограниченным набором микрокоманд, отсюда существенно уменьшались аппаратные затраты. К 70-м годам идея о том, что написанная программа сначала должна интерпретироваться микропрограммами, а не выполняться непосредственно электроникой, стала преобладающей. Однако в современных процессорах, когда аппаратные затраты стали менее существенны, отказались от идеи

микропрограммирования, так как она стала сдерживать рост производительности.

Типичная схема микропрограммного автомата приведена на рисунке 3.14.

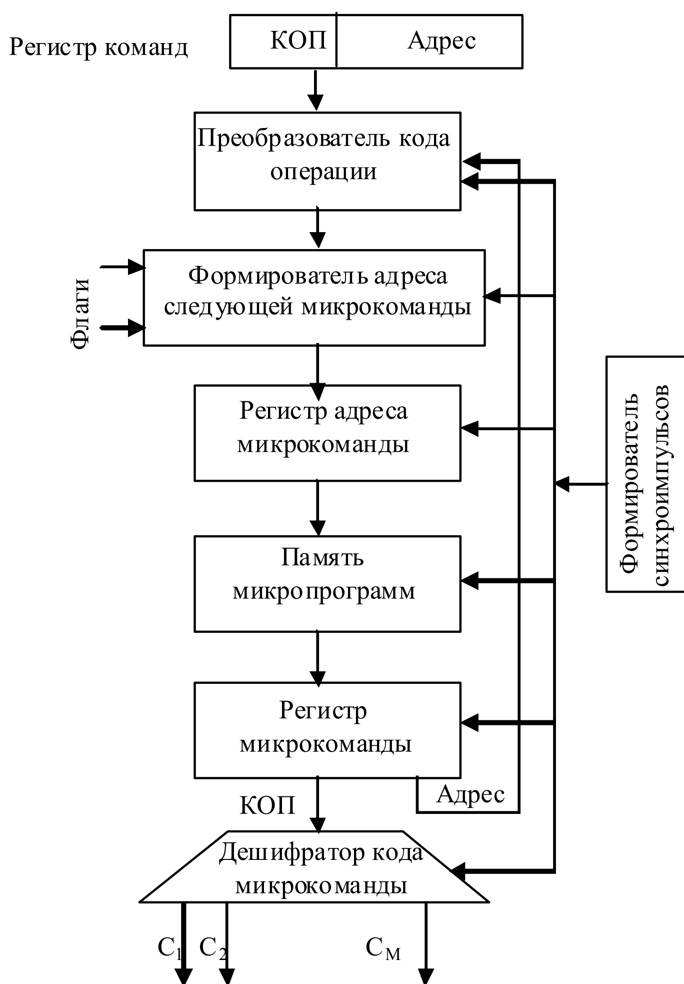


Рис. 3.14. Микропрограммный автомат с программируемой логикой

Запуск микропрограммы выполнения операции осуществляется путем передачи кода операции из регистра команды на вход преобразователя, в котором код операции (КОП) преобразуется в начальный адрес микропрограммы. Выбранная по этому адресу из памяти микропрограмм микрокоманда заносится в регистр. Микрокоманда содержит КОП и адресную часть. КОП поступает на дешифратор и формирует управляющие сигналы, адрес передается для формирования адреса следующей

микрокоманды. Этот адрес может зависеть от флагов, КОП, внешних устройств.

Пример процессора с тремя шинами и его микропрограммирования. Типовая реализация микропрограммного уровня содержит микрокоманды всего двух типов: стробирования GATE и анализа TEST.

Микрокоманда GATE

1	0	0	1	1	....	1	1	1	0	0	1

Биты соответствуют стробам (вентилям) Признак GATE

Микрокоманды считываются из ПЗУ микрокоманд с тактовой частотой, каждый бит команды GATE подается на свой вентиль, и если в считанном бите "единица", то вентиль в данном такте открывается.

Микрокоманда TEST

Адрес	0/1	000...010...000	N регистра	0
На какой адрес микрокода перейти	В	Какой бит проверить на = В (только один бит д.б.=1)	В каком регистре	Признак TEST

Количество битов в слове д.б. ≥ (числа вентилей+1) и в нем должны помещаться все поля микрокоманды TEST. Обычно количество битов в микрокоманде 40...60...100.

Количество микрокоманд должно быть достаточно для кодирования всех команд процессора, обычный объем микрокода 256...1000 слов.

Однако часто оказывается, что количество используемых комбинаций битов (т.е. разных микрокоманд) значительно меньше количества слов в микрокоде. Тогда можно расщепить микропрограммный уровень на два.

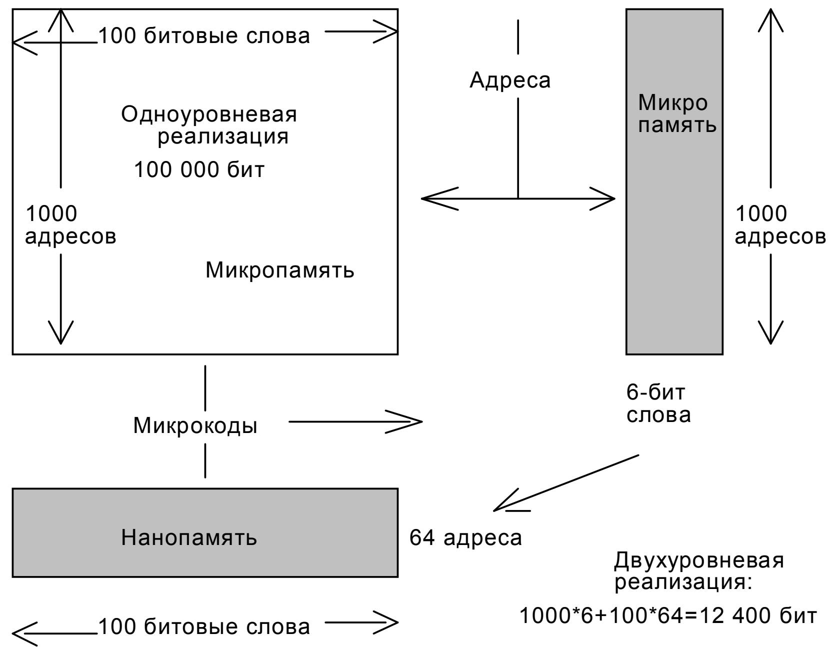


Рис. 3.15. Организации микропрограммной памяти.

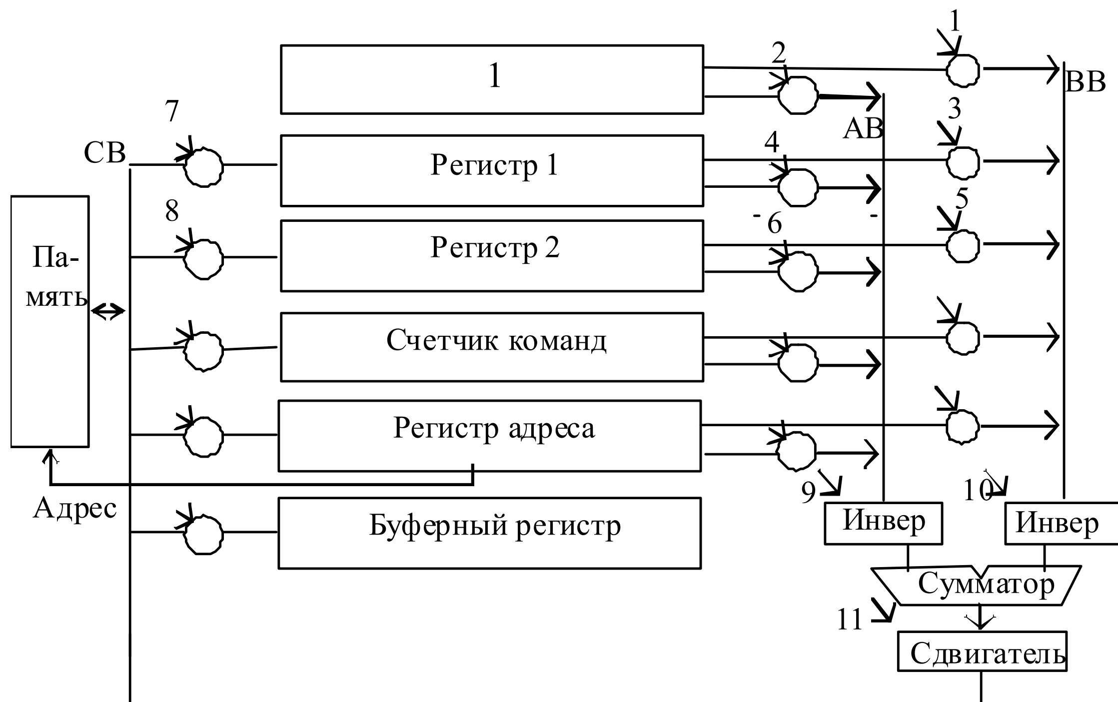


Рис. 3.16. Процессор с тремя внутренними шинами

Приведем пример "аппаратной и микропрограммной" реализации. Пусть надо реализовать этап исполнения команды sub R2, R1 ; R2-R1 -> R2 Если АЛУ умеет выполнять вычитание, тогда процесс исполнения может быть таким:

1) R1 -> AB и R2 -> BB // Помещаем операнды на входы АЛУ

2) АЛУ - вычитание // Делаем вычитание

3) Выход АЛУ -> R2 // Записываем результат

Если же АЛУ не умеет делать вычитание, а только складывает, тогда делаем так:

1) R1 -> AB и Инв BB и -1 -> BB

2) АЛУ - сложение // Получаем доп. код R1

3) Выход АЛУ - > Rбуф // и сохраняем его во временном регистре

4) Rбуф -> AB и R2 -> BB //

5) АЛУ - сложение // Складываем с доп. кодом

6) Выход АЛУ - > R2 // Записываем результат

В таблице показана микропрограммная реализация команд add и not, указаны биты команд GATE, которые приводят к открытию вентилей в схеме.

	Вент1	Вент2	Вент3	Вент4	Вент5	Вент6	Вент7	Вент8	Вент9	Вент10
add1:	0	0	1	0	0	1	0	0	0	0
	0	0	0	0	0	0	1	0	0	0
add2:	0	0	1	0	0	1	0	0	0	0
	0	0	0	0	0	0	0	1	0	0
not r1:	1	0	0	1	0	0	0	0	1	1
	0	0	0	0	0	0	1	0	0	0

3.3. Архитектуры систем команд

С развитием вычислительной техники появлялись различные архитектуры системы команд (АСК), некоторые из них становились основными на каком-то этапе. Сложившуюся на настоящий момент ситуацию в области архитектуры команд иллюстрирует рис. 3.19 .

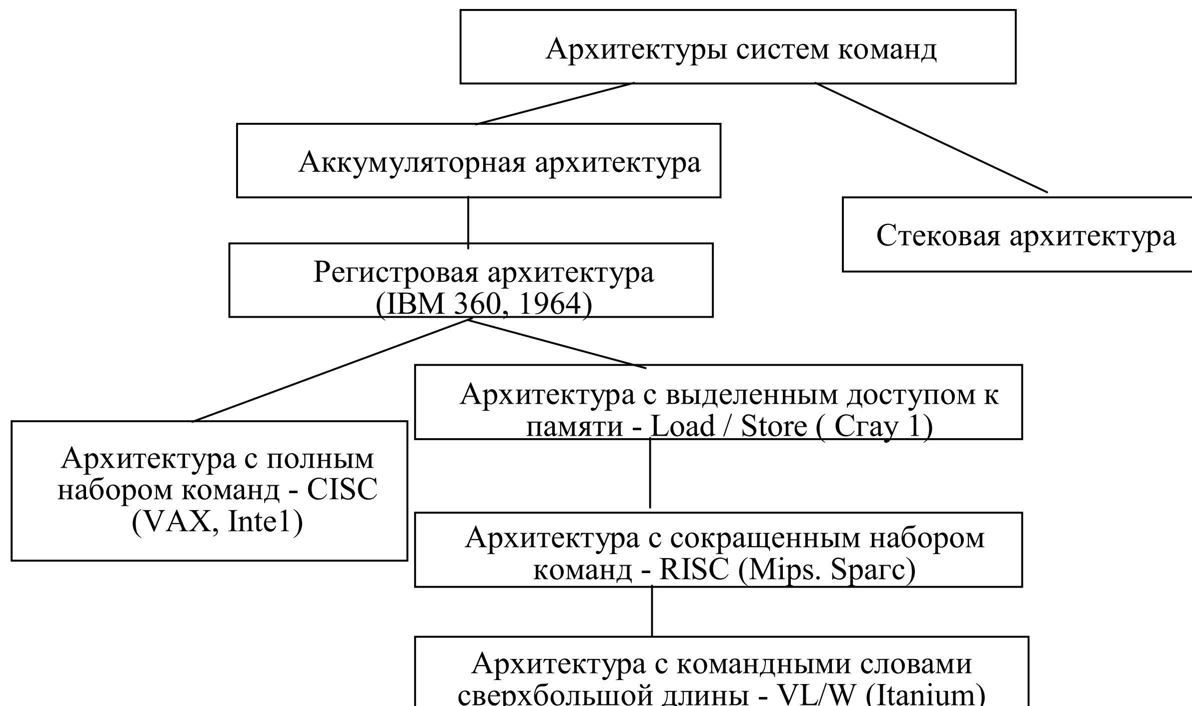


Рис. 3.19. Архитектуры систем команд с примерами

При анализе/выборе архитектуры команд существенны два момента. Первый - это состав операций, выполняемых вычислительной машиной, и их сложность. Второй - место хранения операндов, что влияет на количество и длину адресов в адресной части команды.

Современная технология программирования ориентирована на языки высокого уровня (ЯВУ), главная цель которых - облегчить процесс программирования. Переход к ЯВУ, однако, породил серьезную проблему: сложные операторы, характерные для ЯВУ, существенно отличаются от простых машинных операций, реализуемых в большинстве вычислительных машин. Проблема получила название *семантическою разрыва*, а ее следствием становится недостаточно эффективное выполнение программ на ЭВМ. Пытаясь преодолеть семантический разрыв, разработчики вычислительных машин в настоящее время выбирают один из трех подходов и, соответственно, один из трех типов АСК:

- архитектуру с полным набором команд: CISC (Complex Instruction Set Computer);

- архитектуру с сокращенным набором команд: RISC (Reduced Instruction Set Compllter);
- архитектуру с командными словами сверхбольшой длины: VLIW (Very Long Instruction Word)

В вычислительных машинах типа CISC проблема семантического разрыва решается за счет расширения системы команд, дополнения ее сложными командами, семантически аналогичными операторам ЯВУ.

Основоположником CISC-архитектуры считается компания IBM, которая начала применять данный подход с семейства машин IBM 360 и продолжает его в своих мощных современных универсальных ЭВМ, таких как IBM ES/9000. Аналогичный подход характерен и для компании Intel в ее микропроцессорах серии 8086 и Pentium. Эта архитектура является практическим стандартом для рынка микрокомпьютеров. Для CISC-архитектуры типичны:

- наличие в процессоре сравнительно небольшого числа регистров общего назначения;
- большое количество машинных команд, некоторые из них аппаратно реализуют сложные операторы ЯВУ;
- широкое использование микропрограммирования;
- разнообразие способов адресации операндов;
- разная временная структура выполнения команд;
- множество форматов команд различной разрядности (Pentium – длина команды 1-10 байт);
- преобладание двухадресного формата команд;
- наличие команд, где обработка совмещается с обращением к памяти.

Рассмотренный способ решения проблемы семантического разрыва вместе с тем ведет к усложнению аппаратуры ЭВМ, главным образом устройства управления, что, в свою очередь, негативно сказывается на

производительности ЭВМ в целом. Это заставило более внимательно проанализировать программы, получаемые после компиляции с ЯВУ. Был предпринят комплекс исследований, в результате которых обнаружилось, что доля дополнительных команд, эквивалентных операторам ЯВУ, в общем объеме программ не превышает 10-20%, а для некоторых наиболее сложных команд даже 0,2%. В то же время объем аппаратных средств, требуемых для реализации дополнительных команд, возрастает весьма существенно. Так, емкость микропрограммной памяти при поддержании сложных команд может увеличиваться на 60%.

Детальный анализ результатов упомянутых исследований привел к появлению RISC архитектуры. Термин RISC впервые был использован Д. Паттерсоном и Д. Дитцелем в 1980 году. Идея заключается в ограничении списка команд ЭВМ наиболее часто используемыми простейшими командами, оперирующими данными, размещенными только в регистрах процессорах. Сложные команды ЯВУ выполняются набором простых, число которых больше чем в CISC, но время их выполнения меньше. Основные особенности RISC архитектуры:

- небольшое число команд с простым декодированием;
- резко уменьшено количество форматов команд;
- почти все команды выполняются за один такт;
- ограничено количество способов адресации;
- длинный регистровый файл, что позволяет большему объему данных храниться в регистрах на процессорном кристалле большее время и упрощает работу компилятора по распределению регистров под переменные;
- обращение к памяти допускается лишь с помощью специальных команд чтения и записи;
- используются трехадресные команды, что помимо упрощения дешифрации дает возможность сохранять большее число переменных в

регистрах без их последующей перезагрузки

- эффективно загружен конвейер команд.

Элементы RISC-архитектуры впервые появились в вычислительных машинах CDC 6600 и суперЭВМ компании Cray Research. Однако окончательно понятие RISC в современном его понимании сформировалось на базе трех исследовательских проектов компьютеров: процессора 801 компании IBM, процессора RISC университета Беркли и процессора MIPS Стенфордского университета.

Разработка экспериментального проекта компании IBM началась еще в конце 70-х годов, но его результаты никогда не публиковались и компьютер на его основе в промышленных масштабах не изготавливался. В 1980 году Д.Паттерсон со своими коллегами из Беркли начали свой проект и изготовили две машины, которые получили названия RISC-I и RISC-II. Главными идеями этих машин было отделение медленной памяти от высокоскоростных регистров и использование регистровых окон. В 1981году Дж.Хеннесси со своими коллегами опубликовал описание стенфордской машины MIPS, основным аспектом разработки которой была эффективная реализация конвейерной обработки посредством тщательного планирования компилятором его загрузки.

Достаточно успешно реализуется RISC архитектура и в современных ВМ, например в процессорах Alpha, серии PA фирмы Hewlett-Packard, семействе PowerPC и т. п. Отметим, что в последних микропроцессорах фирмы Intel и AMD широко используются идеи, свойственные RISC архитектуре, так что многие различия между CISC и RISC постепенно стираются.

Концепция архитектуры команд со словами сверхбольшой длины (VLIW) базируется на RISC-архитектуре, где несколько простых RISC - команд объединяются в одну сверхдлинную команду и выполняются параллельно. См. процессоры Itanium, Transmeta.

Важную роль в выборе АСК играет ответ на вопрос о том, где могут храниться операнды и каким образом к ним осуществляется доступ. С этих позиций различают следующие виды архитектур системы команд:

- стековую;
- аккумуляторную;
- регистровую;
- с выделенным доступом к памяти.

Стековая архитектура

Стеком называется память, состоящая из взаимосвязанных ячеек, взаимодействующих по принципу «последним вошел, первым вышел» (LIFO, Last In First Out).

Верхнюю ячейку называют вершиной стека. Для работы со стеком предусмотрены две операции: push (проталкивание данных в стек) и pop (выталкивание данных из стека). Запись возможна только в верхнюю ячейку стека, при этом вся хранящаяся в стеке информация предварительно проталкивается на одну позицию вниз. Чтение допустимо также только из вершины стека. Извлеченная информация удаляется из стека, а оставшееся его содержимое продвигается вверх. В вычислительных машинах, где реализована АСК на базе стека (их обычно называют стековыми), операнды перед обработкой помещаются в две верхних ячейки стековой памяти. Результат операции заносится в стек.

В большинстве процессоров стек (т.е. память со стековым доступом) организован в участке обычной памяти с адресной организацией. Для этого в процессоре имеется специальный регистр – указатель стека (Stack Pointer SP). Этот регистр содержит адрес памяти того участка, в который будет осуществляться стековый доступ, а, говоря более точно, адрес «верхушки стека». Указатель стека обычно программно доступен, то есть к нему можно производить обращение как к любому другому регистру.

При описании вычислений с использованием стека обычно используется иная форма записи математических выражений, известная как обратная польская запись (обратная польская нотация), которую предложил польский математик Я. Лукашевич. Особенность ее в том, что в выражении отсутствуют скобки, а знак операции располагается не между операндами, а следует за ними (постфиксная форма). Последовательность операций определяется их приоритетами. Выражение $a = a + b + a \times c$ в постфиксной форме будет записано в виде: $a = a\ b + \ a\ c \times +$.

АСК на основе стека долгое время считался неперспективным. Однако возрождается интерес к стековой архитектуре ВМ, связано с популярностью языка Java и расширением сферы применения языка Forth, семантике которых наиболее близка именно стековая архитектура.

Аккумуляторная архитектура

Архитектура на базе аккумулятора исторически возникла одной из первых. В ней для хранения одного из операндов арифметической или логической операции в процессоре имеется выделенный регистр - *аккумулятор. В этот* же регистр заносится и результат операции. Поскольку адрес одного из операндов предопределен, в командах обработки достаточно явно указать местоположение только второго операнда. Изначально оба операнда хранятся в основной памяти, и до выполнения операции один из них нужно загрузить в аккумулятор. После выполнения команды обработки результат находится в аккумуляторе и, если он не является операндом для последующей команды, его требуется сохранить в ячейке памяти.

Для загрузки в аккумулятор содержимого ячейки *x* предусмотрена команда загрузки *load x.* По этой команде информация считывается из ячейки памяти *x,* выход памяти подключается к входам аккумулятора и происходит занесение считанных данных в аккумулятор. Запись содержимого аккумулятора в ячейку *x* осуществляется командой сохранения *store x,* при выполнении которой выходы аккумулятора подключаются шине, после чего

информация записывается в память. Для выполнения операции в АЛУ производится считывание одного из операндов из памяти в регистр данных. Второй операнд находится в аккумуляторе. Выходы регистра данных и аккумулятора подключаются к соответствующим входам АЛУ. По окончании предписанной операции результат с выхода АЛУ заносится в аккумулятор.

Достоинствами аккумуляторной АСК можно считать короткие команды и простоту декодирования команд. Однако наличие всего одного регистра порождает многократные обращения к основной памяти.

Регистровая архитектура

В машинах данного типа процессор включает в себя массив регистров (регистровый файл), известных как регистры общего назначения (РОН). Эти регистры, в каком-то смысле, можно рассматривать как явно управляемый КЭШ для хранения недавно использовавшихся данных.

Размер регистров обычно фиксирован и совпадает с размером машинного слова. К любому регистру можно обратиться, указав его номер. Количество РОН в архитектурах типа CISC обычно невелико (от 8 до 32), и для представления номера конкретного регистра необходимо не более пяти разрядов, благодаря чему в адресной части команд обработки допустимо одновременно указать номера двух, а зачастую и трех регистров (двух регистров операндов и регистра результата). RISC-архитектура предполагает использование существенно большего числа РОН (до нескольких сотен), однако типичная для таких ВМ длина команды (обычно 32 разряда) позволяет определить в команде до трех регистров.

Регистровая архитектура допускает расположение операндов в одной из двух запоминающих сред: основной памяти или регистрах. С учетом возможного размещения операндов в рамках регистровых АСК выделяют три подвида команд обработки: регистр-регистр; регистр-память; память-память. Вариант «регистр-регистр» характеризуется простотой реализации,

фиксированной длиной команды, быстрым выполнением команды, но большой длиной кода; он является основным в вычислительных машинах типа RISC. Команды типа «регистр-память» имеют компактный код, простое декодирование, но длинное место адреса в коде, потеря операнда при записи; характерны для CISC машин. Вариант «память-память» считается неэффективным, хотя и остается в некоторых моделях класса CISC.

К достоинствам регистровых АСК следует отнести: компактность получаемого кода, высокую скорость вычислений за счет замены обращений к памяти на обращения к регистрам. С другой стороны, данная архитектура требует более длинных инструкций по сравнению с аккумуляторной архитектурой. В наши дни этот вид архитектуры является преобладающим.

Архитектура с выделенным доступом к памяти

В архитектуре с выделенным доступом к памяти обращение к основной памяти возможно только с помощью двух специальных команд: *load* и *store.* По команде *load* информация считывается из ячейки памяти в регистр процессора. Запись из регистра в память происходит по команде *store*. Операнды во всех командах обработки могут находиться только в регистрах процессора (в регистрах общего назначения). Результат операции также заносится в регистр. В архитектуре отсутствуют команды обработки, допускающие прямое обращение к основной памяти. АСК с выделенным доступом к памяти характерна для всех вычислительных машин с RISC-архитектурой. Команды в таких ЭВМ, как правило, имеют длину 32 бита и трехадресный формат. Примеры процессоров: Sun SPARC, MIPS R10000, DEC Alpha, PowerPC и др.

3.4. Ассемблер и система команд процессора на примере процессора Pentium.

Хотя наборы команд, реализованных в разных процессорах, различаются по количеству и перечню команд, по способам кодирования, по длине команд и по времени их выполнения, в системах команд разных процессоров есть весьма много общего. Знание этих общих свойств помогает быстрее освоить программирование нового процессора.

Разработчики процессора стремятся включить в систему команд, прежде всего, те действия, которые чаще требуются программистам. При этом наиболее часто требуемые действия стремятся реализовать в более коротких и быстрых командах. Перечень и свойства операций, выполняемых процессорными командами, тесно связаны со свойствами разных видов данных, которые обрабатываются на ЭВМ. Общего в разных системах команд достаточно много. Далее будем рассматривать главным образом это общее, часто упоминая, чем вызваны те или иные различия.

Мнемоники Ассемблера

Для описания команд и их действия будем использовать мнемоники, принятые в языке Ассемблера. Язык Ассемблера специфичен для каждого типа процессора, так как включает в себя совокупность символических обозначений процессорных команд и способов адресации. Несмотря на специфичность, в языках Ассемблера для разных процессоров достаточно много общего, как в форме (в синтаксисе) так и в содержании отображаемых конструкциями языка понятий, поскольку и в различных процессорах также имеется много одинаковых, либо похожих свойств.

В некоторых учебных изданиях даже не делается различия между изучением процессора и изучением языка Ассемблера для него (наверное, с точки зрения программиста на Ассемблере это так и есть). Однако изучение программирования на языке ассемблера включает в себя три компоненты.

1) Собственно синтаксис Ассемблера, мнемоники команд процессора и способов адресации.
2) Управление процессом трансляции (директивы Ассемблера), позволяющее программисту получить программу с нужными свойствами, например, задать требуемое расположение частей программы в памяти и т.п.
3) Изучение набора и свойств сервисов используемой операционной системы (стандартных подпрограмм ОС, доступных прикладному программисту).

Нам в данном курсе будет нужен только п.1) Ассемблерные мнемоники и фрагменты программ призваны показать, как можно использовать возможности, предоставляемые аппаратурой процессора.
Если вы хотите подробно ознакомиться с техникой программирования на языке Ассемблера для процессоров семейства x86, рекомендуем вам воспользоваться книгами в списке литературы.

Для облегчения понимания кратко опишем основные правила записи команд на Ассемблере (они справедливы для многих известных автору Ассемблеров).

1) Как правило, ассемблерная строка однозначно соответствует одной процессорной команде.

2) Команда языка Ассемблера имеет следующую структуру

Метка1: КОП Оп1,Оп2,… ; Комментарий

Вот пример команды на языке ассемблера

L1: mov r1, #12A9h ; Загрузка регистра константой

Пояснение: Данный оператор содержит команду пересылки, которая загружает константу 12A9h в регистр процессора r1. Константа задана программистом в виде шестнадцатиричного числа.
Оператор включает в свой состав следующие поля:

Метка – это символическое обозначение адреса. В мнемонике команды, приведенной выше, метка обозначает адрес, начиная с которого байты данной команды будут расположены в ОЗУ после загрузки программы в

память. Имя метки часто используется как операнд в командах переходов. (**Замечание 1:** метки могут обозначать любой адрес, в том числе и тот, с которого расположен операнд. **Замечание 2:** конкретное значение физического адреса, соответствующего метке будет определено только после загрузки оттранслированной программы в память. При разных запусках этот физический адрес может получиться различным, если программист не принимает специальных мер по заданию определенного значения для этого адреса.).

КОП – мнемоническое обозначение кода операции, выполняемой данной командой, например mov – переслать

Оп1, Оп2,… - символические обозначения операндов, обычно они разделяются запятыми (хотя в некоторых Ассемблерах для разделения операндов используется пробел). Количество операндов в команде может быть различным, в большинстве современных процессоров от 0 до 3. Если операндов больше, чем один, некоторые из них являются «источниками», а некоторые другие – «приемниками». Например, команда сложения

```
add     sum,op1,op2
```

содержит указания на два операнда-приемника (слагаемые) – op1 и op2, а также указание на элемент данных (sum), куда команде следует поместить результат.

Количество операндов, которые программист может указать в команде, определено отдельно для каждой команды конкретного процессора его разработчиками. Для многих команд делают допустимым несколько форматов (см. напр. команду целочисленного умножения *imul* для процессоров семейства x86).

В Ассемблерах процессоров фирмы Intel по большей части, операнды-источники записываются левее операндов-приемников. В Ассемблерах других фирм это может быть наоборот. В качестве операнда можно

использовать метку, в этом случае метка будет обозначать адрес, с которого располагается в памяти соответствующий операнд.

Комментарий позволяет программисту записать пояснение к строке, и как правило, игнорируется транслятором.

3) Хотя каждый процессор имеет свой Ассемблер (соответствующий его системе команд), многие мнемонические обозначения в разных Ассемблерах одинаковы для одинаковых операций. Этот факт сильно облегчает изучение следующего Ассемблера (и процессора), после того, как хотя бы один уже изучен.

4) При записи обозначений операндов используются условные обозначения выбранного программистом способа адресации. Обозначения различных способов адресации в разных ассемблерах также имеют много общего. Мы познакомимся с конкретными обозначениями при рассмотрении способов адресации.

Количество команд для разных типов ЭВМ колеблется от малых десятков до сотен. В таком множестве разобраться достаточно трудно, поэтому для рассмотрения разобьем все команды на группы (проклассифицируем). В разных книгах эта классификация тоже сделана по-разному. Выделяют от 3 до >10 групп. (Наиболее обозримой для человека является классификация, содержащая на нижележащем уровне от 3 до 8 подклассов).

1. Команды пересылки

1.1. Пересылки общего назначения MOV,L**,LD*,LOD* (от Load - загрузить), ST* (от Store – сохранить). Передают слово/байт данных из одной части ЭВМ в другую без изменения. Иногда в эту группу включают также и команды ввода-вывода для ЭВМ, у которых область адресов внешних устройств включена в общее адресное пространство.

1.2. Пересылки из/в стек: PUSH (втолкнуть),POP (вынуть). Обычно отличаются тем, что используют стековую адресацию (задаваемую неявно).

1.3. Пересылки двоичных слов, представляющих собой адреса операндов или части (компоненты) адресов. Для операций с адресами нередко в процессор вводят специальные команды. Это связано с тем, что разрядность адреса в процессоре не всегда совпадает с разрядностью АЛУ и регистров.

1.4. Пересылки между элементами вычислительного ядра (регистры процессора, элементы памяти) и периферийными устройствами. Хотя эти команды выполняют простую передачу двоичного слова, соответствующая группа команд (называемая командами ввода-вывода) обычно рассматривается отдельно.

2. Команды обработки

2.1.Арифметические.

Минимальный набор арифметических команд очень мал. Это (например)

сложение - ADD

инвертирование -COM / NOT (такая ЭВМ действительно была: PDP-8 (DEC))

прибавление «единицы» - INC

Все остальное можно сделать, комбинируя эти команды.

Однако в современном микропроцессоре арифметических команд обычно больше: -

SUB вычитание

CMP сравнение операндов – эта команда выполняет вычитание операндов, по результату изменяет флаги, после чего результат теряется (эта команда предназначена для проверки условий)

NEG, смена знака операнда

ASR,ASL, SAR,SAL арифметические сдвиги операнда (Arithmetic Shift to Right/Left)

INC, DEC увеличение или уменьшение операнда на 1

ADC,SBC/SBB - операции с переносом C (carry-bit) - для выполнения действий с повышенной точностью, когда операнд занимает несколько слов.

SXT, SEX, CBW, CWD, CDQ- расширение знака (преобразование в формат с повышенной разрядностью)

MUL,DIV – умножение и деление беззнаковых и знаковых чисел

Набор операций с плавающей точкой, включающий обычно значительное количество команд (несколько десятков), в которые часто входят команды вычисления элементарных функций – Sin, Cos, Log, Exp и т.п.

2.2.Логические (это команды побитовой обработки)

OR - поразрядное логическое сложение. Фактически это команда побитовой установки (т.е. записи «единицы» в заданные биты операнда). Пример:

or opr, 0Ch ; Установка в 1 битов в позициях 2 и 3 в операнде **al**

Если исходно в **al** содержался, например операнд 1101 1001_2, то после выполнения команды в **al** будет содержаться 1101 **11**01 (отмечены установленные биты). Второй операнд это обычно константа, задавая которую, программист указывает, какие биты следует установить, он на жаргоне программистов носит название «маска».

Можно описать действие команды OR следующим образом: *команда безусловно устанавливает в «единицу» биты в тех позициях первого операнда, которые отмечены «единицами» во втором операнде (в маске), оставляя прочие биты первого операнда неизменными.*

AND - поразрядное логическое умножение. Это команда побитового сброса (записи в заданные биты «нулей»). Пример:

and bh, 0Fh ; Сброс старшей тетрады (старшего ниббла) в байтовом операнде.

Если, например, исходно в регистре **bh** содержался операнд 1101 1001_2, то после выполнения команды в регистре **bh** будет содержаться **0000** 1001_2 (отмечены очищенные биты). Действие команды AND можно описать следующим образом: *команда безусловно сбрасывает в 0 биты в тех позициях первого операнда, которые отмечены «нулями» в маске, оставляя прочие биты первого операнда неизменными.*

XOR - поразрядное исключающее ИЛИ, eXclusive OR (иногда на русском эту операцию называют «ЛИБО»). Эта двухоперандная команда фактически выполняет выборочное инвертирование битов. Например:

xor cl,0F0h ; Инвертирование битов старшей тетрады

Если исходно в **cl** содержалось $1101\ 1001_2$, то после выполнения команды в **cl** будет содержаться **0010** 1001_2 (отмечены проинвертированные биты).

Действие команды **xor** можно описать так: команда инвертирует в первом операнде биты в позициях, которые отмечены в маске «единицами», оставляя прочие биты неизменными.

TEST – проверка битовых полей – команда чаще всего двухоперандная, выполняет поразрядное логическое умножение операндов, и по результату операции изменяет состояние флагов «нуля» **zf** и «знака» **sf**, после чего результат операции теряется.

NOT , **COM**– инвертирование операнда (замена значения каждого бита на противоположное). Это однооперандная команда.

2.3.Сдвиги

Операция сдвига состоит в одновременном перемещении содержимого операнда в разрядной сетке. Существует 3 разновидности операции сдвигов, которые различаются тем, что происходит с битами, выходящими за пределы разрядной сетки с одного «конца» операнда, и с освобождающимися позициями на другом его «конце».

1) **ROR, ROL, RCR, RCL** - циклические сдвиги. При циклическом сдвиге то, что выходит за границу разрядной сетки, помещается в освобождающуюся позицию на другом конце операнда.

2) **ASR,ASL / SAR,SAL** - арифметические сдвиги. Эта разновидность сдвига осуществляется таким образом, что результат оказывается эквивалентен умножению (при сдвиге влево) или делению (при сдвиге вправо) операнда на основание системы счисления, т.е. на 2. Сравните: если «сдвинуть» цифры в десятичном числе на разряд влево, результат будет эквивалентен исходному

числу, умноженному на 10 (12 и 120 после сдвига). Более подробно особенности арифметического сдвига обсуждаются при рассмотрении системы команд процессоров x86.

3) **SHL,SHR** - логические сдвиги. При выполнении логических сдвигов биты, «выдвигаемые» из разрядной сетки, теряются, а противоположный конец операнда заполняется «нулями».

(Каждая из упомянутых разновидностей может также иметь варианты, в составе системы команд x86 различают пять видов сдвигов).

3. Проверки и передача управления

Эти команды позволяют реализовать конструкции

IF ... THEN ... ELSE

REPEAT ... UNTIL

WHILE ... DO

FOR ... DO

GOTO ...

Для выполнения этих функций в каждом процессоре есть регистр флагов (регистр состояния), разряды которого устанавливаются или сбрасываются в зависимости от свойств результата предыдущей операции. Анализируются:- равенство нулю; - знак; - перенос; - арифметическое переполнение; - перенос между тетрадами и др......

3.1. Команды проверки

TEST - проверка отдельных битов (логическим умножением).

CMP - сравнение операндов.

Кроме того, признаки устанавливаются после выполнения многих команд. Обратите внимание на то, что нет единых правил поведения признаков. (В процессорах фирмы Motorola при пересылке флаги «нуля» и «знака» изменяются, а в процессорах Intel - нет).

3.2. Команды ветвления по условию.

B** / J*** от слов (branch / jump). Их может быть 10...30 штук. (например: BNE - переход, если не равно 0). Набор команд ветвления обсуждается при рассмотрении системы команд процессоров x86.

Адресация в командах ветвления может быть разного типа:

а) полный адрес перехода

б) относительная (смещение)

в) пропуск команды.

Но чаще всего короткая относительная. Короткая, потому что участки, которые надо обходить, имеют не очень большую длину. Если надо перейти далеко, то надо комбинировать условный переход с безусловным JMP.

3.3. Команда безусловной передачи управления

JMP Адресация делается такой, чтобы можно было "прыгнуть" в любое место программной памяти ("длинная" адресация).

3.4. Команда организации цикла

Позволяет организовать в программе структуры FOR ... DO более простым способом, чем с помощью команд ветвления.

LOOP*

3.5. Команда обращения к подпрограмме (вызов процедуры).

JSR, JMS, CALL

Для экономии памяти, занимаемой программой, делают возможность записать эту последовательность команд только в одном месте памяти и передавать управление к ней из разных мест вызывающей программы.

При обращении / возврате надо обеспечить:

а) передачу управления в любое место памяти, поэтому "длинная" адресация;

б) возврат в то место, откуда был вызов (т.е. место вызова должно автоматически запоминаться)

в) запоминание промежуточных результатов, имеющихся к моменту вызова (содержимое регистров процессора, а при рекурсивном вызове процедуры надо запоминать и промежуточные результаты работы самой процедуры).

Для запоминания всего этого чаще всего используется стек (участок ОЗУ или специальное ОЗУ со стековой адресацией). Часть вышеперечисленной информации запоминается автоматически при выполнении команды CALL, а сохранение оставшегося - дело программиста.

4. Команды ввода – вывода (обмена с периферийными устройствами)

Фактически, команды это тоже команды пересылки. Основное отличие между пересылками «регистр-память» и пересылками из/в периферийное устройство ПУ состоит в том, что скорость функционирования ПУ может существенно отличаться от скорости работы процессора, и поэтому ПУ далеко не всегда бывает готово к обмену.

- Устройств ввода-вывода (УВВ) много, надо как-то адресовать
- Устройства ввода-вывода разные, надо как-то унифицировать правила обмена.
- Скорость работы УВВ отличается от скорости работы процессора, надо как-то синхронизировать работу процессора и УВВ.

Варианты структуры взаимодействия процессора и УВВ:

а) УВВ имеют свою систему нумерации (адресации), и в системе команд есть отдельные команды I/O, (как в IBM PC)

IN - команда ввода из ВНУ

OUT - команда вывода на ВНУ

В этом случае в формате команды УВВ предусматривается своя система адресации.

б) Обращение к УВВ такое же, как к памяти, часть адресов использована для ВНУ (ввод / вывод, отображенный на память).

Достоинства: -можно использовать для обмена с ВНУ всю систему команд, в том числе команды обработки, что сокращает программу.

Недостатки:

-команда вв/выв длиннее и дольше выполняется, чем в а)

-сложнее устройство декодирования адреса в ВНУ.

5. Системные команды (команды управления процессором)

Некоторые функции этой группы команд общие для многих типов процессоров:

- разрешение/запрещение прерываний
- работа с регистром состояний (с флагами)

Другие могут быть специфическими для разных процессоров

- работа с расширенной памятью
- взаимодействие с другими процессорами в многопроцессорной системе.....

6. Расширения системы команд по мере развития микропроцессорной техники

Кроме упомянутых команд и их групп в отдельных моделях как универсальных, так и специализированных процессоров могут быть реализованы специальные команды, предназначенные для выполнения более сложных действий, часто используемых в применениях, для которых разрабатывалась конкретная модель. Приведем несколько примеров:

1) В специализированных графических процессорах используются специальные команды, облегчающие выполнение операций линейной интерполяции, которые очень часто используются при преобразованиях графических данных.
2) В специализированных графических процессорах могут быть команды, выполняющие достаточно сложные операции с группами простых элементов данных, например с двумерными блоками в памяти (это фрагменты изображений). Более подробно этот вопрос затронем, обсуждая графические данные.
3) В процессорах Pentium MMX-SSE реализована группа новых команд. Каждая из команд MMX способна выполнять одно и то же действие одновременно над несколькими «короткими» операндами, расположенными в «длинных» MMX-регистрах. Здесь реализована

идеология SIMD (Single Instruction – Multiple Data – «одна команда – несколько элементов данных), позволяющая получать несколько результатов при выполнении каждой MMX-команды. Это частный случай более обширной группы команд, упомянутой в предыдущем пункте.

4) В разновидности процессоров, известной под названием «цифровые сигнальные процессоры» (DSP – Digital Signal Processors) часто реализованы команды, облегчающие вычисление «цифровой свертки» ($Y=Sum(X_i*K_i)$). Операция цифровой свертки очень часто используется в алгоритмах обработки сигналов. (Сигнальные процессоры нередко имеют также своеобразную архитектуру, программную модель, хорошо приспособленную для действий с отсчетами сигналов.)
5) В однокристальных микроконтроллерах M68HC12 фирмы Motorola реализованы специальные команды, облегчающие программирование алгоритмов «нечеткой логики» (Fuzzy Logic).

3.5. Способы адресации

Простейшая структура адресной части команды:

КОП	ФизАдрес1	ФизАдрес2	...

Адресные поля содержат физические адреса

1) Для ЭВМ 1...2 поколений были характерны небольшой объем ОЗУ (4...32 Кслов) и значительная длина машинного слова (40...64 и более бит). Длина слова выбиралась обычно достаточной для представления числа с инженерной точностью в плавающем формате. При этом в поле адреса мог поместиться полный физический адрес (для маленького объема ОЗУ он был коротким).

2) В ходе развития ЭВМ

- увеличивался объем ОЗУ, а следовательно и длина требуемого физического адреса;

- желательно было вычислять/модифицировать адреса. Это давало

возможность одному участку программы обрабатывать данные, расположенные в разных адресах.

Перешли к структуре:

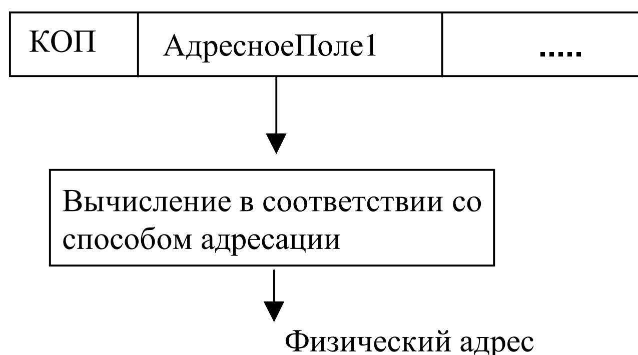


Рис. 3.20. Двухуровневая схема трансляции адреса

Простейший вариант: физический адрес содержится в адресном регистре, а адресное поле содержит имя (код, номер) адресного регистра плюс код, обозначающий способ адресации (в данном случае *косвенно-регистровую адресацию*). Длина такого поля адреса, могла быть гораздо меньше длины адреса. Например, если процессор содержит восемь регистров общего назначения и использует не более 8 разных способов адресации, длина номера регистра = 3 бита, длина кода способа адресации также 3 бита. Адресное поле в команде будет содержать всего 6 бит.

3) При дальнейшем развитии ЭВМ - увеличивается размер адресного пространства

При 32-разрядном адресе размер адресного пространства = 4 Мбайт. В то же время реальный объем ОЗУ составляет 16...128 Мбайт (длина физического адреса: 24...27 разрядов). Таким образом, имеет место несовпадение диапазона логических адресов (в частности, тех чисел, которые могут храниться в адресных регистрах процессора) и диапазона физических адресов, нумерующих реально существующие ячейки ОЗУ. Поэтому все более широко используется следующая схема преобразования адресов:

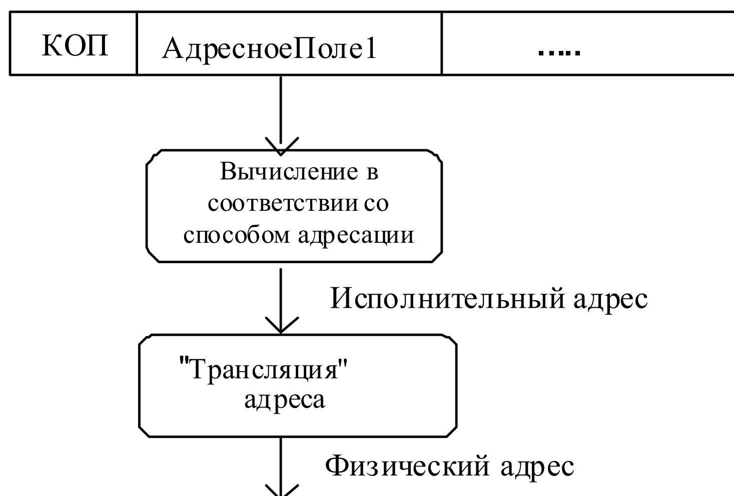


Рис. 3.21. Трехуровневая схема трансляции адреса

В результате вычисления в соответствии со способом адресации формируется объект, называемый *исполнительным (executive)* или *эффективным* (так переводят английский термин *effective*, хотя лучше *исполнительный* или *действующий адрес)*. Это понятие уровня языка ассемблера почти эквивалентно используемому в языках высокого уровня понятию *указатель (pointer)*. Все перечисленные в данном абзаце понятия прежде всего суть понятия логические (логические адреса).

Логические адреса требуется отображать на физические адреса фактически имеющейся памяти. В простейшем случае можно это делать «один в один», начиная (размещая, загружая) всю программу целиком в определенное место памяти. В то же время, как увидим далее, такое взаимно-однозначное отображение адресов не всегда удобно, а иногда даже и невозможно.

Понятие «Способ адресации» включает:

1) Способ кодирования адреса в адресном поле команды
2) Условное обозначение (синтаксис) способа адресации при записи команды на языке ассемблера
3) Алгоритм вычисления исполнительного адреса по информации, содержащейся в адресном поле, а также в других элементах процессора,

имеющих отношение к вычислению адреса (хранящих адресную информацию, компоненты адреса)

Функции способов адресации (и механизма трансляции адреса)

1. Обеспечить удобство вычисления логических адресов при отображении на память компонентов сложных структур данных (массив, структура и поля ее записей, список и т.п.) Это одно из свойств, которые обычно имеют в виду, когда говорят, что "микропроцессор оснащен средствами для программирования на языках высокого уровня"

2. Обеспечить переход от содержимого адресного поля команды к логическому адресу и отображение пространства логических адресов на пространство физических адресов.

3. Обеспечить перемещаемость программных модулей (для легкости компоновки из этих модулей большой программы) или обеспечения позиционной независимости программы. Под термином «перемещаемость» могут иметь в виду одно из двух различных свойств программы. *Статическая перемещаемость* состоит в том, что оттранслированную программу можно, без модификации адресных частей команд, загружать, начиная с различных адресов, при этом программа сохраняет работоспособность. *Динамическая перемещаемость* – более сильное свойство, состоящее в том, что загруженную и выполняющуюся программу можно в любой точке остановить, переместить (как одно целое) в другое место памяти и затем успешно продолжить выполнение с точки останова.

4. Для настройки на реально существующую в системе физическую память.

5. Обеспечить возможность многозадачного режима работы, когда в памяти одновременно находятся и выполняются несколько программ: надо разрешить этим программам обмениваться информацией, но защитить их друг от друга (чтобы одна программа не имела возможности испортить другую).

Что такое "многозадачность"?

В простейшем случае две задачи - это программа пользователя ПрП и операционная система ОС. ПрП не должна иметь возможности испортить ОС. Другой пример: кроме ОС - две ПрП, из которых одна работает в фоновом режиме. Например, основная задача - текстовый процессор, с которым работает человек, редактируя текст. В паузах, когда человек думает, какую клавишу нажать, работает фоновая задача, например, распечатывается на принтере другой текст. Еще более сложный случай - многопользовательская система, когда на одном процессоре работает несколько пользователей одновременно, они разделяют процессорное время, пространства памяти и другие ресурсы.

Механизм трансляции адреса позволяет отображать логические адреса программ, выполняемых на вычислительной системе именно в тот диапазон физических адресов, который соответствует реально установленной памяти.

Способы адресации при адресной организации памяти

Понятие косвенной адресации.

Пусть какой-нибудь способ адресации (1) указывает место положения операнда (неважно, регистр процессора или адрес в памяти). Можно представить себе другой способ адресации (2), во всем аналогичный способу (1) за исключением того, что в указываемом месте расположен не сам операнд, а его адрес, т.е. номер ячейки в ОЗУ, где находится операнд. Тогда будем говорить, что способ (1) - прямой, а соответствующий ему способ (2) – косвенный по отношению к способу (1).

В символике языков ассемблера для обозначения косвенности используются скобки или символ @.

1) add ax,loc эта ассемблерная запись означает "прибавить к содержимому регистра AX содержимое ячейки памяти, адрес которой обозначен символическим именем LOC.

2) ADD AX,(LOC) или ADD AX,@LOC в этой записи используется косвенная адресация, запись означает "прибавить к содержимому регистра AX содержимое ячейки памяти, адрес которой хранится в ячейке памяти с символическим именем LOC.

Можно провести аналогию между использованием косвенной адресации в языке ассемблера и использованием указателей в языке Си. В строке iA += iB ; обращение за операндом iB произойдет с использованием прямой адресации. Строка iA += *piB; использует косвенную адресацию переменной через указатель piB = &iB. По индукции можно представить себе двойную, тройную косвенность и т.п. Существовали компьютеры, в которых была реализована косвенность произвольной глубины.

Рассмотрим теперь конкретно различные способы адресации. При рассмотрении использованы мнемоники ассемблера для процессоров i*86. Используемый в команде способ адресации кодируется в *адресном поле операнда*. За исключением *непосредственной* и *абсолютной* адресации, адресное поле указывает на адресный регистр (может быть на несколько) и задает способ вычисления адреса с участием содержимого этого регистра (ов).

Неявная (inherent) адресация.

Этот термин относится не к способу вычисления адреса (местоположения) операнда, а к способу обозначения этого местоположения в синтаксисе ассемблера.

Например, команда “поместить в стек: push ax

Непосредственная адресация.

Если операнд является заранее известной константой, которая используется "только здесь" и однократно, то его значение удобно помещать прямо в команде за кодом операции

add ax,**12** ; прибавить константу 12 к содержимому регистра ax

КОП	Адресное поле операнда	Непосредственный операнд

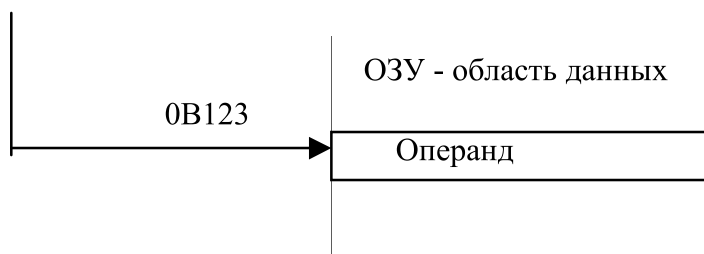
Никаких вычислений для определения адреса операнда делать не надо, операнд выбирается из ОЗУ вслед за командой. Операнд может быть разной длины (занимать разное число ячеек/байтов).

Абсолютная или прямая адресация.

В команде за КОП помещается заранее известный адрес операнда. Обратите внимание, что абсолютная адресация является косвенным вариантом для непосредственной адресации.

add cx, **[123]**, ; содержимое регистра складывается с содержимым ячейки, абсолютный адрес которой входит в состав команды.

КОП	Адресное поле операнда	
0B123 (абсолютный адрес операнда)		



Для выборки операнда нужно дополнительное обращение к ОЗУ

Регистровая адресация.

Операнд находится в одном из регистров процессора.

sub **bx**,**si**; вычитаются два операнда, находящиеся в регистрах процессора

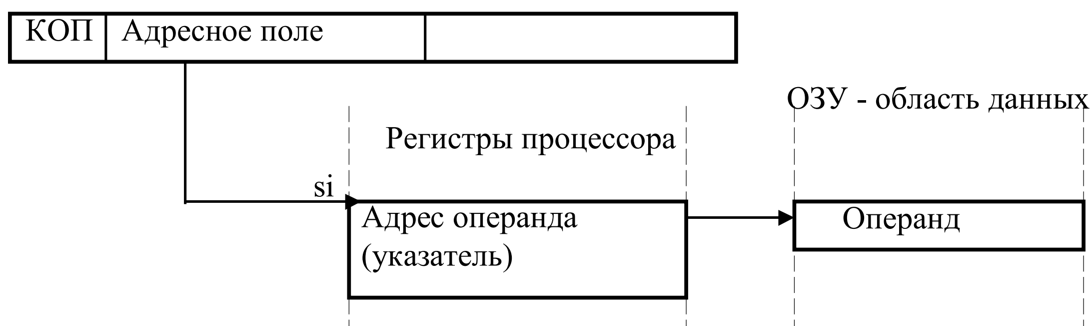
КОП	Адресное поле	Адресное поле

Обращение к регистру процессора гораздо быстрее, чем к ячейке ОЗУ. Код регистра занимает более короткое поле в команде, чем операнд или адрес.

Косвенно-регистровая адресация.

Адрес операнда находится в одном из регистров процессора. Во многих процессорах регистры специализированы, и адрес может находиться не в любом из них.

mov **[si]**, 12



Указатель регистра Ri занимает более короткое поле, чем адрес. Тот факт, что адрес операнда находится в регистре, позволяет вычислять или модифицировать этот адрес, при этом один и тот же участок программы может обрабатывать разные элементы данных (находящиеся в разных адресах).

Вот фрагмент, суммирующий в регистре ax элементы массива слов:

```
      mov    si,  "начальный адрес массива"
      mov    cx,  "количество элементов массива"
ckl   add    ax, [si]    ;  прибавляем следующий элемент
      add    si, 2       ;  модифицируем адрес в массиве
      loop ckl           ;  организуем цикл
```

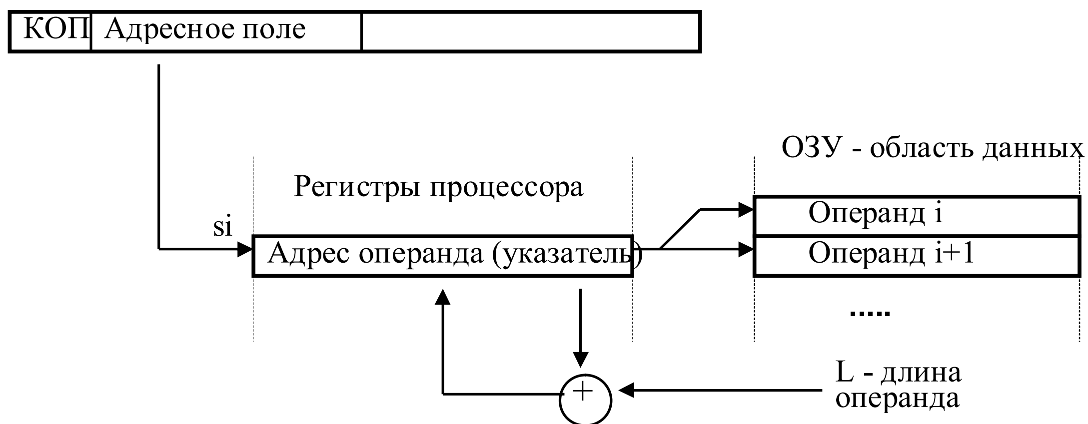
Выделенные команды эквивалентны Си-фрагменту sum += *(piMas++); .

Автоинкрементная (автодекрементная) адресация.

Модификация адреса, находящегося в адресном регистре - частая и типичная операция (см. предыдущий пример). Во многих процессорах существует усовершенствованная разновидность косвенно-регистровой адресации, которая совмещает обращение к памяти по адресу,

содержащемуся в регистре и модификацию адреса (содержимого регистра). Такая адресация называется *автоинкрементной*, если адрес, хранящийся в регистре, автоматически увеличивается при обращении к операнду, или *автодекрементной*, если адрес автоматически уменьшается. Общее название для этих двух типов адресации – ***автоиндексная (или адресация с автоиндексацией)***.

В ходе выполнения команды с данным типом адресации после (либо до) выборки операнда из ОЗУ содержимое регистра Ri модифицируется: к его старому содержимому прибавляется число L - длина операнда в байтах (в минимальных адресуемых единицах). Таким образом, после этой модификации в Ri оказывается адрес следующего элемента данных. Автоинкрементная адресация позволяет обращаться к элементам данных, расположенных в памяти подряд по возрастанию адресов (например, к элементам массива). При автодекрементной адресации сложение с L заменяется на вычитание, и элементы данных перебираются в порядке убывания их адресов.



Величина L, на которую происходит увеличение или уменьшение адреса, в простейшем случае равна минимальной адресуемой единице (байту). Однако, если процессор поддерживает операции с операндами разной длины (байт, слово, двойное слово,...), то и авто-ин/де-кремент может быть реализован на 1. на 2, на 4, ...

Если модификация адресного регистра происходит после обращения к операнду, такую разновидность называют *постинкрементной* адресацией, если модификация адреса выполняется перед обращением к операнду - *преинкрементной*. Наиболее часто в процессоре реализуется сочетание *постинкрементной* и *предекрементной* адресации.

В процессорах i*86 автоиндексная адресация в общем виде (т.е. для использования в произвольной команде) не реализована. Однако в неявном виде она использована в командах операций со строками. Например, при выполнении команды LODS происходит загрузка в регистр AL (или AX или EAX) содержимого ячейки памяти, на которую указывает адресный регистр SI, после чего содержимое регистра SI автоматически модифицируется.

Многокомпонентные способы формирования адреса

Они тесно связаны с идеей *относительной адресации*. Если отсчитывать адреса объектов программы от места, жестко привязанного к программе (например, от ее начала), то эти относительные значения не меняются, куда бы ни была загружена программа. То же можно сказать и про любую часть программы (например, про участок памяти с данными). Как же можно осуществить такую относительную адресацию.

Для этого можно формировать адрес **из двух слагаемых**:

Одно (базовый адрес) зависит от того, куда загружаем программу, задается перед загрузкой, и в течение всего времени выполнения программы остается неизменным (но от загрузки к загрузке может изменяться).

Второе (относительный адрес) представляет собой разность базового адреса и абсолютного адреса объекта.

Многокомпонентное формирование адреса связано с идеей относительной адресации, когда адрес задается не как абсолютное значение, а как смещение относительно какого-либо базового значения.

Хочется обеспечить позиционную независимость (без пересчета адресных частей команд). Если вспомним, что после трансляции

загружаемый (двоичный) образ программы не меняется, можем высказать идею отсчитывать адреса относительно чего-то в самой программе. Идея формировать адрес как комбинацию (сумму или конкатенацию) нескольких компонент используется как в способах адресации, так и при трансляции адресов.

Между понятиями «способ адресации» и «процедура трансляции адреса» непросто провести более-менее четкую границу. Та часть преобразования адресной информации, которая изменяется от одной команды к другой – способ адресации. Трансляция адреса - часть преобразования адреса, одинаковая для разных команд. Трансляцией адреса тоже можно управлять (например, включить или выключить трансляцию страниц), однако это переключение выполняется редко, после чего значительная часть программы выполняется в заданном единожды режиме.
Рассмотрим далее некоторые частные случаи.

Страничная адресация

Это, пожалуй, простейший из многокомпонентных способов. Адрес разбивается на две части, которые объединяются не сложением, а конкатенацией (соединением двух частей числа). Старшая часть адреса указывает *номер страницы*, она либо хранится в специальном *регистре страницы* (общий случай), либо берется из счетчика команд (адресация на текущую страницу), либо фиксирована (часто так делается адресация на нулевую страницу, которая используется для специальных целей). В команде (или в регистре) указывается только младшая часть адреса, таким образом, удается уменьшить длину команды.

Относительная адресация

В качестве одной из компонент при формировании адреса используется счетчик команд.

При этом способе адресации операнд отстоит в памяти от команды на фиксированную величину. Иными словами, операнд указывается путем

задания его смещения (displacement) в памяти относительно обращающейся к операнду команды. Использование только относительной адресации делает программы позиционно-независимыми, т.е. их можно перемещать в памяти без каких бы то ни было изменений в тексте программы. Относительная адресация используется во многих процессорах в командах условного ветвления, причем чаще всего смещение имеет формат "байт со знаком" и способно задать переход в диапазоне +127 ÷ -128 байтов.

Стековый доступ к памяти

Стековый доступ к памяти происходит в большинстве процессоров в следующих ситуациях:

1) При выполнении команд стекового доступа: «поместить в стек» **push** или «извлечь из стека» **pop** .
2) При выполнении команды вызова подпрограммы и при возврате из нее
3) При входе в прерывание и при *возврате из прерывания*.

Положение стека в адресуемой памяти определяется содержимым *указателя стека*. Поскольку *указатель стека* программно доступен, программист может задать его значение обычной командой пересылки, например, команда **mov sp,#1000h** задает положение стека, начиная с адреса 1000h.

При стековом доступе содержимое *указателя стека* используется как адрес операнда-приемника при записи в стек или как адрес операнда-источника при считывании из стека. При обращении к стеку автоматически модифицируется и содержимое указателя стека, чтобы обеспечить следующее обращение к очередной ячейке стека. Используя ранее введенную терминологию, можно сказать, что обращение к стеку происходит с использованием косвенно-регистровой адресации через указатель стека с автоиндексацией (см. таблицу далее, которая поясняет, какие допустимы варианты автоиндексации).

В разных процессорах стековый доступ может быть реализован по-разному в нескольких аспектах:

1. Направление роста стека. Если при записи в стек, содержимое *указателя стека* автоматически увеличивается (и соответственно, при считывании автоматически уменьшается), то говорят, что стек растет в сторону увеличения адресов. В противоположном случае говорят, что стек растет в сторону уменьшения адресов.
2. Если модификация *указателя стека* выполняется до записи и соответственно после считывания, то *указатель стека* всегда указывает на последнюю занятую ячейку стека. Наоборот, если модификация производится после записи и до считывания, *указатель стека* всегда указывает на первую свободную ячейку стека.

Если запись в стек происходит с преиндексацией, то считывание должно происходить с постиндексацией (и наоборот).

Распространение различных видов адресации зависит от типа АСК. Достаточно ясная ситуация с RISC архитектурой. Из самой идеи подхода вытекает, преимущественный способ адресации – регистровая. Для аккумуляторной архитектуры главные способы адресации – прямая и непосредственная. В таблице показана частота применения разных способов адресации на двух тестовых программах для процессоров типа Intel 80x86. Видно, что наиболее активно используется прямая и базово-регистровая адресации, хотя интенсивность использования различных способов тесно связана с решаемой задачей.

Тип адресации	GCC	Spice
Косвенно- регистровая [bx]	10%	3%
Индексная с масштабированием [2*ebx+5]	20%	33%
Прямая [123]	19%	51%
Базово-регистровая [bp+5]	48%	14%

3.6. Управление вычислительным процессом

В машине со структурой фон Неймана команды расположены в ячейках памяти программ (в адресном пространстве команд) подряд. При рассмотрении принципа работы ЭВМ мы считали, что команды выбираются из памяти программ и выполняются подряд (имеет место свойство локальной сериальности). В процессорах имеются возможности нарушать естественный порядок следования команд при их выполнении. Благодаря этому программы могут приобрести ряд полезных свойств. Рассмотрим ситуации, в которых желательно нарушение естественного порядка выполнения команд.

1. *Условное ветвление.* Необходимость разветвления в алгоритме: в зависимости от того, каким получился некий промежуточный результат, следует выполнить одно из двух различных действий. Пример: поиск заданной фамилии в списке студентов. Если очередная фамилия совпала с искомой, поиск следует прекратить, если нет – поиск надо продолжить.
2. *Программный цикл*. Выполнение одной и той же последовательности действий с несколькими экземплярами данных. Для этого можно написать *последовательность команд* так, чтобы она могла обрабатывать элементы данных, находящиеся в задаваемых адресах. В конце такой последовательности следует организовать разветвление. Если еще не все элементы данных обработаны, следует (нарушив порядок команд «подряд») перейти к началу *последовательности команд*, одновременно изменив адреса обрабатываемых данных. Если же все требуемые данные обработаны – продолжить выполнение программы далее.
3. *Модульная иерархическая структура программы*.

Цель	Средство
«Интеллектуализация» программы	Организация условных ветвлений
Экономия объема программы	Использование итерационных/циклических фрагментов
Борьба со сложностью	Организация компьютерной программы как многоуровневой иерархической структуры
Повышение полезной загрузки ресурсов	Реакция на асинхронные события
Повышение полезной загрузки ресурсов	Выполнение одновременно нескольких программ

Проверка условий, флаги и набор команд ветвления

Сравнение в ЭВМ используется для организации последующего ветвления алгоритма (условного перехода в программе) в зависимости от результата сравнения. Сравнение **можно производить** по условиям: а) равно / неравно; б) больше / меньше.

Условие а) всегда осмысленно и формально означает, что все биты сравниваемых операндов одинаковы. Условие б) - его семантика понятна для данных, которые неким образом упорядочены, таких как числа или символы алфавита. Для такого вида данных, как битовые поля - не всегда понятно, что такое "больше/.меньше"

Сравнение в процессоре происходит по одной из двух схем:

1) Сравнение операнда с нулем (его можно произвести специальной командой **проверка** – в системе команд x86 – команда **test a, b**)

2) Сравнение двух операндов между собой (вычитанием и последующим сравнением результата с нулем)

После выполнения команды, осуществляющей сравнение (и переустанавливающей флаги) надо осуществить ветвление. Для этого в процессоре обычно существует большая **группа команд условного ветвления**, которое происходит (или нет) в зависимости от состояний тех или иных флагов. В группе команд ветвления можно выделить три подгруппы:

Ветвление **по простому условию**:

Описание	Условие	Альтернативное	Семантика
По знаку	$sf = 0$	$sf = 1$	Результат положителен/отрицателен
По нулю	$zf = 0$	$zf = 1$	Результат равен/не равен нулю
По переносу	$cf = 0$	$cf = 1$	Установлен/нет флаг **cf** (по разным причинам)
По переполнению	$of = 0$	$of = 1$	Установлен/нет флаг **of** (по разным причинам)

При сравнении двух чисел в результате вычитания может быть получен как правильный результат, так и переполнение разрядной сетки. Однако, оказывается, что и при переполнении можно определить, какой из операндов был больше, анализируя комбинацию флагов.

Ветвление **по результату сравнения беззнаковых чисел**

Описание	Условие
Если больше	$cf \vee zf = 0$
Если меньше или равно	$cf \vee zf = 1$
Если больше или равно	$cf = 0$
Если меньше	$cf = 1$

Ветвление **по результату сравнения чисел со знаком**

Описание	Условие
Если больше или равно	$sf \oplus of = 0$
Если меньше	$sf \oplus of = 1$
Если больше	$(sf \oplus of) \vee zf = 0$
Если меньше или равно	$(sf \oplus of) \vee zf = 1$

Например, если требуется организовать программный фрагмент со структурой, изображенной в левом столбце таблицы, фрагменты ассемблерного кода будут использовать различные команды условного ветвления в зависимости от того, являются ли сравниваемые операнды знаковыми или нет.

	a и **b** беззнаковые	**a** и **b** знаковые
Да **a ≥b** Блок1 Нет	Для ветвления используется команда ветвления **jnae** по результату сравнения беззнаковых чисел	Для ветвления используется команда ветвления **jnge** по результату сравнения чисел со знаком

	cmp a,b **jnae L2** ; команды блока1 **L2: jmp L3** ; команды блока1 **L3:**	**cmp a,b** **jnge L2** ; команды блока1 **L2: jmp L3** ; команды блока1 **L3:**

Организация в программе циклических конструкций

В языках высокого уровня используется несколько разновидностей циклов. Все они предусматривают следующие (общие) свойства:

1) Если цикл не предусматривает специально бесконечного повторения, в последовательности команд должен быть фрагмент, проверяющий условие выхода из цикла.
2) Если условие выхода из цикла не выполнено, производится передача управления на начало последовательности команд, образующих цикл.

Один из вариантов цикла предусматривает заранее (до начала выполнения цикла) определенное количество повторений. В этом случае

1) заводят специальную переменную
2) перед началом цикла заносят в нее требуемое количество повторений
3) выполняют команды цикла
4) в конце цикла уменьшают ее на 1
5) проверяют ее на равенство нулю, и если не равно, повторяют с п.3

В системе команд процессоров x86 есть команда организации цикла **loop.** Ее недостаток состоит в том, что она в качестве переменной-счетчика цикла позволяет использовать только регистр процессора ECX. Это ее свойство затрудняет организацию вложенных циклических конструкций.

Организация иерархической структуры программы

Современные «большие» программы содержат 10^6 и более команд. Для того чтобы можно было в такой программе разобраться (не говоря уже о том, что ее надо перед этим написать и отладить), программа должна быть *структурирована*. Основное средство структурирования на уровне системы команд процессора – это поддержка аппаратным уровнем (система команд, способы адресации) *организации подпрограмм*.

В 60...70 годах развилась манера, называемая "структурным программированием", которая широко использовала подпрограммную структуру и еще ряд договоренностей. Структурное программирование позволило значительно увеличить производительность труда при написании сложных программ и повысить их надежность. Сущность понятия *подпрограмма* (ПП) состоит в следующем.

Некоторые «законченные» функции требуется выполнять в разных местах программы. Примеры: а) определить сумму элементов массива; б) напечатать какой-либо текст на экране или принтере. После того, как написан и отлажен программный фрагмент, выполняющий подобную функцию, этот фрагмент можно рассматривать как новую мощную команду. Использовать написанный фрагмент можно двумя способами.

Первый состоит в том, чтобы подставлять текст этого фрагмента в те места программы, где требуется выполнение соответствующей функции (этот способ обычно реализуется при использовании приема, называемого в терминологии языков программирования *макроподстановкой*). Недостаток этого способа состоит в том, что один и тот же фрагмент кода повторяется многократно, увеличивая общий размер программы. (Попутно отметим, что достоинство макроподстановки – меньшее время выполнения, так как не тратится время на *вызов-возврат* и *сохранение-восстановление контекста* – см. далее).

Второй способ предполагает наличие нужного фрагмента лишь в одном экземпляре и передачу ему управления (вызов из разных мест программы)

всякий раз, когда требуется выполнение реализуемой фрагментом функции. Используемый таким образом фрагмент кода и называют подпрограммой (ПП). При этом необходимо обеспечить несколько дополнительных действий:

1) После выполнения ПП управление надо вернуть в место вызова (каждый раз оно может быть разным, т.е. требуется при каждом вызове запоминать адрес возврата) – будем называть это действие *связью по управлению*.
2) При передаче управления ПП надо передать ей также и исходные данные (при суммировании элементов массива это могут быть начальный адрес массива и число элементов), а при возврате управления в место вызова надо туда также вернуть и результат (в приведенном примере это сумма массива) – это действие будем называть *связью по данным*.
3) Вызываемый фрагмент кода (подпрограмма) может использовать при своей работе ресурсы, такие как регистры процессора и т.п. К моменту вызова эти регистры могут содержать результаты работы предшествующих команд, которые будут нужны в дальнейшем. Совокупность переменных величин (содержимых регистров процессора, ячеек памяти), полностью характеризующих состояние программы, будем называть контекстом *программы*. При вызове подпрограммы нужно сохранить ту часть контекста вызывающей программы, которую подпрограмма при своей работе может изменить (испортить).
4) Подпрограмме для работы могут потребоваться локальные переменные величины. Часть из них может располагаться в регистрах, другая часть – в ячейках памяти. Эти переменные нужны только до окончания работы подпрограммы (до возврата в вызывающую программу). При возврате занятую память хорошо бы освободить.
5) В некоторых случаях одновременно может существовать несколько экземпляров одной подпрограммы, каждая со своим контекстом. Соответственно нужно несколько экземпляров локальных переменных.

Простейший пример – рекурсивный вызов, когда подпрограмма вызывает сама себя.

В составе системы команд есть обычно инструкции, обеспечивающие в той или иной мере выполнение только что перечисленных действий и требований или, по крайней мере, некоторых из них.

Обращение к подпрограммам - передача управления

Для работы с подпрограммами в составе системы команд любого процессора есть две команды:

1) **Команда обращения к подпрограмме ОП** (для этой команды используются мнемоники **call, jsr**). Эта команда должна автоматически запоминать адрес возврата (т.е. адрес команды, следующей за командой **call**). В современных процессорах при выполнении команды обращения к ПП адрес возврата запоминается автоматически в стеке.
2) **Команда возврата из ПП** (используются мнемоники **ret**, **rts**).

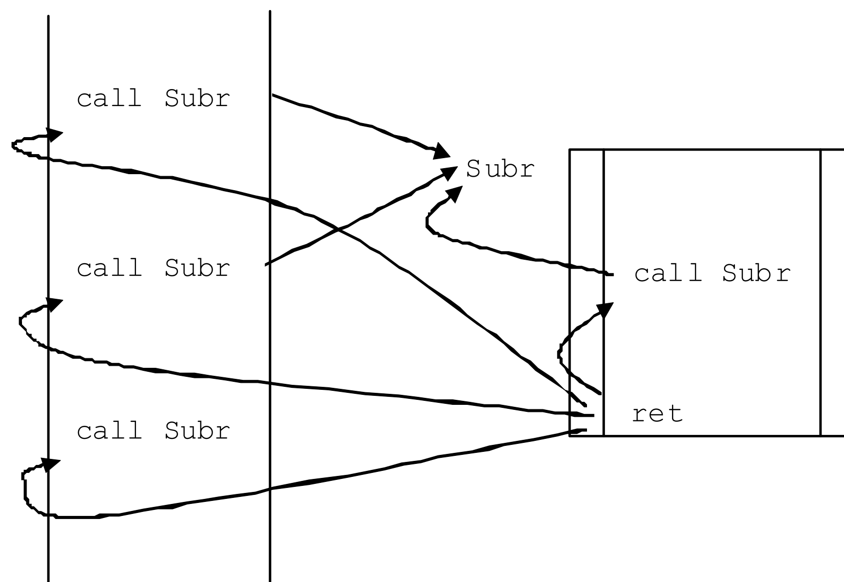


Рис. 3.22. Связь с подпрограммой по управлению.

Обращение к подпрограммам - сохранение/восстановление контекста

Для обращения к подпрограмме используется специальная *команда вызова подпрограммы*. (В системе команд x86 она имеет мнемонику **call**). Минимальный набор действий, которые выполняет такая команда, это:

1) сохранение адреса возврата (т.е. адреса команды, следующей за **call**)
2) загрузка в счетчик команд адреса первой исполняемой команды ПП.

Для возврата из ПП в системе команд есть специальная команда (в i*86 ее мнемоника **ret**). Эта команда загружает ранее сохраненный адрес возврата в счетчик команд.

В x86 имеются две разновидности команды **call** –ближний near вызов (внутри текущего сегмента кода) и дальний far вызов (в другой сегмент кода). В случае использования дальнего вызова в стеке сохраняется наряду со счетчиком команд **(e)ip** и сегментный регистр **cs**. Соответственно имеются и две команды возврата **ret near** и **ret far**. Состояние стека при ближнем и дальнем вызовах иллюстрируется следующим рисунком.

Понятие контекста (вектора состояния): - это в каждой конкретной точке выполнения программы *содержимое всех переменных элементов (регистров, ячеек памяти), которое требуется установить/восстановить, чтобы стало возможным запустить выполнение программы с этой точки*.

Подпрограмма при своей работе использует какие-либо ресурсы процессора (регистры), а, кроме того, меняет состояние отдельных ячеек памяти, флагов и т.п. - **т.е. ПП изменяет контекст**. Естественный способ исключить влияние изменения контекста – непосредственно перед вызовом ПП или сразу после входа в ПП сохранять текущий контекст, а при выходе из ПП все восстанавливать назад. Было бы удобно, если бы программисту не надо было задумываться о сохранении-восстановлении контекста. Проблема сохранения-восстановления контекста возникает и в других ситуациях

(прерывания, переключение задач - это будет позже). Чаще всего для автоматического сохранении-восстановлении контекста используется стек.

Рассмотрим типовую последовательность команд, используемую при вызове подпрограммы. Пусть подпрограмма использует два входных параметра (два аргумента) типа **int**, возвращает значение тоже типа **int** и использует две локальные переменные. Две команды **mov** в вызывающей части помещают в стек параметры, после чего выполняется команда вызова подпрограммы **call**.

Первые две команды подпрограммы сохраняют старое значение **ebp** (элемент контекста) и устанавливают **ebp** на адрес внутри стекового кадра. Следующая команда (**sub esp,8**) занимает в стеке место под две локальные переменные

Теперь подпрограмма может, используя адресацию *косвенно-регистровую со смещением* через регистр **ebp**, иметь доступ к параметрам, используя положительные значения смещения (в примере для доступа к первому параметру смещение должно быть равно 12), и к локальным переменным, используя отрицательные смещения (в примере смещение –8 позволяет обращаться к локальной переменной 2).

Вызывающая часть	Подпрограмма
push Param1; Передача **push Param2**; параметров **call Subr** **add esp, 8** ; Освобождаем место параметров	**Subr:** **push ebp** ; сохраняем контекст **mov ebp, esp** ; указатель на стековый кадр **push esi** ; сохраняем контекст (esi) **sub esp,8** ; место для двух локальных перем. **mov eax, [ebp+12]** ;. доступ к параметру1 **... ...** **mov esi, [ebp-8]** ; доступ к локальной перем. **... ...** **mov eax, Result** ; Результат – в регистр **pop esi** ; восстанавливаем контекст (esi) **mov esp,ebp** ; Освобождаем место локальных пер. **pop ebp** ; восстанавливаем контекст **ret**

Результат, вычисляемый подпрограммой можно передавать в вызывающую часть разными способами, но чаще всего это делают, помещая

результата в регистр перед возвратом из подпрограммы, как сделано в примере. Для освобождения места, занятого локальными переменными, следует вернуть указателю стека то значение, которое он имел в начале подпрограммы (оно хранится в **ebp**). После этого выполняется команда возврата **ret**, использующая сохраненный в стеке адрес возврата в вызывающую часть программы.

После возврата следует освободить место в стеке, занятое параметрами. В нашем примере это делает команда **add esp, 8** , однако в архитектуре x86 это может быть сделано с помощью разновидности команды возврата из подпрограммы **ret n**, она не только осуществляет возврат, но и извлекает из стека (в «никуда») число байтов, указываемое параметром **n**.

Для облегчения сохранения-восстановления контекста в разных реализациях процессоров могут быть использованы:

1) Специальные команды:

В ix86+ команды pusha, popa - они сохраняют в стеке - восстанавливают регистры данных и адресные в таком порядке: (e)ax, (e)cx, (e)dx, (e)bx, (e)sp, (e)bp, (e)si, (e)di. При этом значение (e)sp берется то, которое было до начала выполнения команды pusha.

В Motorola 60xxx есть команды movem (MOVE Multiple), сохраняющие/восстанавливающие регистры в/из памяти. Особенность - можно указать (битовой маской), какие регистры сохранять. Место сохранения указывается с использованием стандартных способов адресации.

2) Несколько экземпляров наборов регистров

TMS9900 - регистров данных и адресов не было вообще. В качестве регистров использовалось несколько ячеек памяти, положение которых в адресном пространстве указывал специальный регистр - указатель рабочей области (аналогично в **Transputer** фирмы Inmos).

Transputer (семейство процессоров фирмы Inmos) - регистров данных/адресов всего 3, и они образуют стек. Переход (и к подпрограмме)

производится только, когда стек пуст, поэтому сохранять надо только адрес возврата.

IA-64 (Itanium) динамическое переименование регистров в регистровом файле при вызове процедуры (см. Intel IA-64 Architecture Software Developer's Manual, vol.2, IA-64 System Architecture, раздел 6 – IA-64 Register Stack Engine).

Обращение к подпрограммам - обмен данными

При обращении к подпрограмме могут передаваться: сами данные; адрес участка памяти, где находятся данные. Данные могут находиться: а) в регистре/ax, если данных мало или если в процессоре регистров много; б) в стеке; в) в оговоренном известном месте памяти.

В большинстве процессоров для а) сохранения-восстановления контекста, б) связи по данным (передачи параметров и возврата значений) в) выделения памяти под локальные переменные используется фрагмент стека, называемый ***стековым кадром***.

Пример на Си:

```
int Calc (int x, int y) {
      int iX, iY;
      iX=2*x+y;
      iY=3*y-x;
      return iX*iY;
}
```

Эта процедура принимает два параметра **x** и **y** и возвращает результат вычисления несложного арифметического выражения в глобальную переменную **z**. В процедуре объявлены две локальные переменные. Далее приведен результат компиляции этой процедуры после его дизассемблирования полноэкранным отладчиком TurboDebugger.

AB1_3.CALC: begin	
cs:0000 55 push bp	Сохранение контекста
cs:0001 89E5 mov bp,sp	В bp базовый адрес стекового кадра
cs:0003 83EC04 sub sp,0004	Выделение места под локальные

	переменные
AB1_3.15: iX:=2*x+y;	
cs:0006 8B460A mov ax,[bp+06]	Доступ к первому параметру
cs:0009 D1E0 shl ax,1	
cs:000B 034608 add ax,[bp+08]	Доступ ко второму параметру
cs:000E 8946FE mov [bp-02],ax	Доступ к локальной переменной
AB1_3.16: iY:=3*y-x;	
cs:0011 8B4608 mov ax,[bp+08]	Доступ ко второму параметру
cs:0014 8BF0 mov si,ax	
cs:0016 D1E0 shl ax,1	
cs:0018 01F0 add ax,si	
cs:001A 2B460A sub ax,[bp+06]	Доступ к первому параметру
cs:001D 8946FC mov [bp-04],ax	Доступ к локальной переменной
AB1_3.17: z:=iX*iY;	
cs:0020 8B46FE mov ax,[bp-02]	Доступ к локальной переменной
cs:0023 F766FC imul word ptr [bp-04]	Результат оставляем в AX
cs:0026 C47E04 les di,[bp+0A]	Берет третий параметр – адрес результата
cs:0029 268905 mov es:[di],ax	Запись результата процедуры
AB1_3.18: end;	
cs:002C 89EC mov sp,bp	Освобождение локальных переменных
cs:002E 5D pop bp	Восстановление контекста
cs:002F C20800 ret 0008	Возврат из процедуры с освобождением места

Поддержка возможности рекурсивного вызова

Рекурсивный вызов (когда ПП вызывает сама себя) иногда бывает полезен. Классический пример - вычисление факториала с использованием соотношения

N ! = N * (N-1) !

Другой пример - решение задачи о "Ханойской башне".

Рекурсия при вызове подпрограммы может быть как прямая (ПП вызывает сама себя) так и цепная (ПП-а А вызывает ПП-у В, та в свою очередь вызывает ПП-у С, а затем ПП-а С вызывает ПП-у А).

Проблем при рекурсивном вызове несколько:

1) Запоминание адресов возврата - каждый экз. ПП имеет свой адрес возврата

2) Каждый экземпляр ПП должен иметь собственные наборы внутренних переменных

3) Иногда требуется, чтобы экземпляры ПП, вызванные позднее, имели доступ к внутренним переменным ранее вызванных экземпляров.

Решение всех этих проблем обеспечивается, когда при вызове ПП в стеке выделяется блок ячеек (стековый кадр) для параметров и временных переменных очередного экземпляра ПП (как в примере, рассмотренном выше). Адресацию внутри стекового кадра удобно делать относительно какого-либо его адреса. Для этого перед вызовом ПП параметры помещаются в стек::

```
        push    par1
        push    par2
        call    subr
```

в начале ПП применяется (как мы уже увидели) типовая последовательность команд:

```
subr:   push    bp      ;  запоминание старого значения регистра базы bp
        mov     bp,sp   ;  передача в bp адреса начала стекового кадра
```

Теперь можно к локальным переменным ПП обращаться, используя адресацию относительно регистра базы:

```
        mov     ax,[bp+4]
        mov     bx,[bp+6]
```

Для обеспечения возможности использовать параметры и переменные "внешних" экземпляров ПП при рекурсивном вызове, стековый кадр должен содержать ссылки на "предыдущие" стековые кадры.

В системе команд процессора могут быть команды, облегчающие формирование стекового кадра. Например, в x86 есть команды ENTER и LEAVE.

enter   frmsiz, level

Первый параметр команды определяет количество байтов, требуемых в стеке для временных (локальных) переменных. Второй параметр показывает уровень "рекурсивности" (для самого внешнего уровня level=1). Для доступа к элементам стекового кадра используется адресный регистр BP, который указывает на начало (самый старший адрес) стекового кадра.

Структура стекового кадра в x86 включает три компонента:
- указатель на начало предыдущего стекового кадра (*динамическая связь*)
- таблица указателей на начала каждого из ранее созданных кадров (*индикатор*)
- локальные переменные.

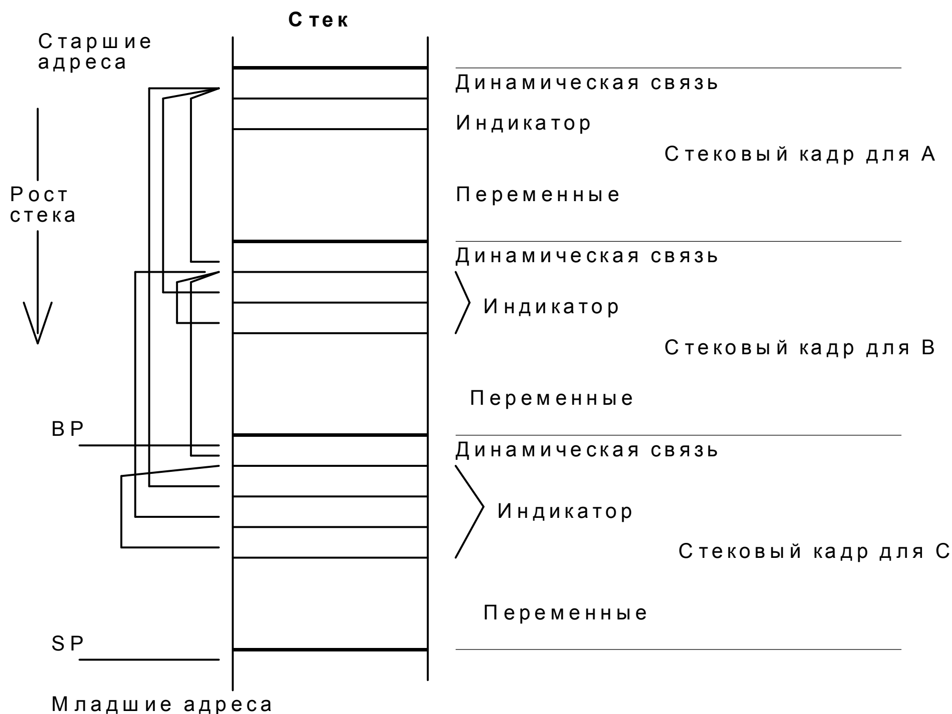


Рис. 3.23. Структура стековых кадров, создаваемых командой **enter**.

Выполнение команды ENTER:

При выполнении команды делаются следующие действия (описание - в предположении, что происходит рекурсивный вход в подпрограмму):

1. TmpREG = SP   ; Запомнить адрес начала следующего стекового кадра
2. BP -> в стек   ; Создание динамической связи

3. Повторить (level-1) раз: ; Создание индикатора

   BP -=2

   [BP] -> в стек ; Копирование элементов индикатора из предыдущего кадра

4. TmpREG -> в стек ; Добавление в индикатор ссылки на данный кадр
5. BP = TmpREG ; Установить BP на начало создаваемого кадра
6. SP -= frmsiz ; Выделение в стеке блока под временные переменные

3.7. Кодирование команд в процессоре x86

Организация памяти

Адресуемая память (адресное пространство) представляет собой область из 1M байт. Физический адрес памяти имеет длину 20 бит (рис.3.24.). Для формирования физических адресов используется механизм сегментации памяти. Пространство памяти 1 М доступно процессору через 4 "окна" (сегмента) каждый размером 64 Кбайт. Начальный адрес каждого сегмента содержится в одном из четырех сегментных регистров. Команды обращаются к байтам и словам в пределах сегментов, используя относительный (внутрисегментный) адрес.

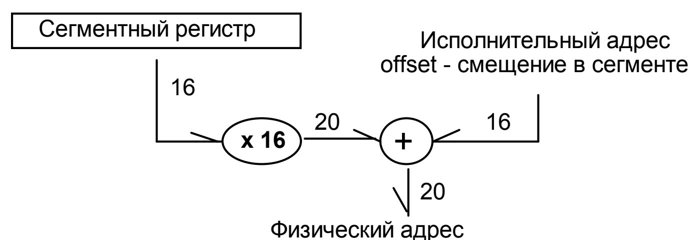


Рис. 3.24. Схема трансляции адреса в процессоре i8086

Для того, чтобы не увеличивать чрезмерно длину команды, большая часть команд в системе команд x86 описывает не более двух операндов. Ниже представлены основные форматы команд.

Форматы команд

№						
1)	КОП	w	reg	Данные	Данные 16	Операнд в регистр
2)	КОП	s	w	ПостбайтО	Данные	Данные 16
3)	КОП	d	w	Постбайт	Смещение	Смещен.16
4)	КОП	w	Мл.байт адр	Ст.байт адр.		Кор.форм. прям. адресации
5)	КОП	v	w	Постбайт		Сдвиги
6)	КОП	Условие	Смещение			Условн.переход
7)	КОП					Неявн. адес. и управл.
8)	КОП	Порт				Ввод-вывод из порта

Кодирование двухоперандной команды

[Префикс] КОП [постбайт адресации] [смещение] [непоср.операнд]

В этой сложной структуре КОП может занимать один или два байта. Адресная часть команды может либо вообще отсутствовать, либо включать в свой состав от одного (ModR/M) до 12 байтов. Адресная информация, закодированная в такой сложной команде, может содержать следующие сведения

а) сколько операндов использует команда (два, один или ни одного);

б) расположен ли операнд в регистре или в памяти

в) если в регистре, то в каком

г) если операнд в команде, сколько для него там отведено места (immediate)

д) если операнд в памяти, то по какой схеме следует определять его адрес

е) при многокомпонентной адресации, в адресной части может находиться поле для части адреса (displacement)

ж) и др.…

Префикс Длина 1 байт. (Всего 5 префиксов для X86)

1.	Переназначения сегмента	**add es:[bx],ax**	
2.	Повторения действия для строковых команд	**REP, REPZ, REPNZ, REPNE**	rep movsb
3.	Блокировки	**Lock**	lock rep movsb

4.	Размера адреса		
5.	Размера операнда		

КОП - код операции. Длина 1 байт. 0-й бит КОП во многих (но не во всех) командах показывает, производится ли операция со словом (=1) или с байтом (=0). 1-й бит КОП в двухадресных командах указывает, какой из операндов является приемником.

Постбайт адресации. Длина 1 байт. Постбайт адресации показывает, где находятся операнды. Один из операндов (первый) может быть расположен в регистре (регистровая адресация) или в произвольной ячейке ОЗУ (все способы адресации кроме непосредственной). Второй операнд может находиться в теле команды (непосредственная адресация) или в регистре (регистровая адресация). Каждый из операндов может быть как источником, так и приемником (за исключением непосредственной адресации: непосредственный операнд может быть только источником). Структура системы адресации несимметрична.

```
   7    6     5    4    3     2    1    0
 !   mod   !     reg     !     r/m     !
 !-----!------!----!----!----!----!-----!----!
```

Поля **mod** и **r/m** задают место расположения первого операнда. Поле **reg** задает положение второго операнда.

Значения поля mod:

11 - операнд в регистре (при остальных mod операнд в ОЗУ, а регистры, на которые указывают поля mod и r/m, содержат компоненты адреса операнда)

10 - смещение два байта (без знака)

01 - смещение один байт (со знаком)

00 - смещение в команде отсутствует

Смещение. Длина 1 байт (при mod-01) или 2 байта(при mod=10).

Непосредственный операнд. Длина 1 или 2 байта

Кодирование регистров полями reg и r/m при mod=11(т.е. при

Кодирование способа вычисления адреса при адресации в память с

регистровой адресации)
Табл.1

reg или r/m	Байт	Слово	
000	AL	AX	
001	CL	CX	
010	DL	DX	
011	BL	BX	
100	AH	SP	
101	CH	BP	
110	DH	SI	
111	BH	DI	

использованием полей mod и r/m)
Табл.2

r/m	mod=00	mod=01 или 10
000	BX+SI	BX+SI+смещение
001	BX+DI	BX+DI+смещение
010	BP+SI	BP+SI+смещение
011	BP+DI	BP+DI+смещение
100	SI	SI+смещение
101	DI	DI+смещение
110	direct	BP+смещение
111	BX	BX+смещение

В 32-х разрядных процессорах для кодирования расширенных регистров появился в формате после постбайта **SIB байт**. Байт SIB включает в себя следующие поля:

- Поле ss, которое занимает 2 старших бита байта, определяет масштабный коэффициент.
- Поле index, которое занимает следующие 3 бита за полем ss, задает для индексного регистра номер регистра.
- Поле base, которое занимает младшие три бита байта, задает номер базового регистра.

В 64-х разрядных процессорах для кодирования номеров 64-х разрядных регистров появился в формате REX- байт (рис. 3.25).

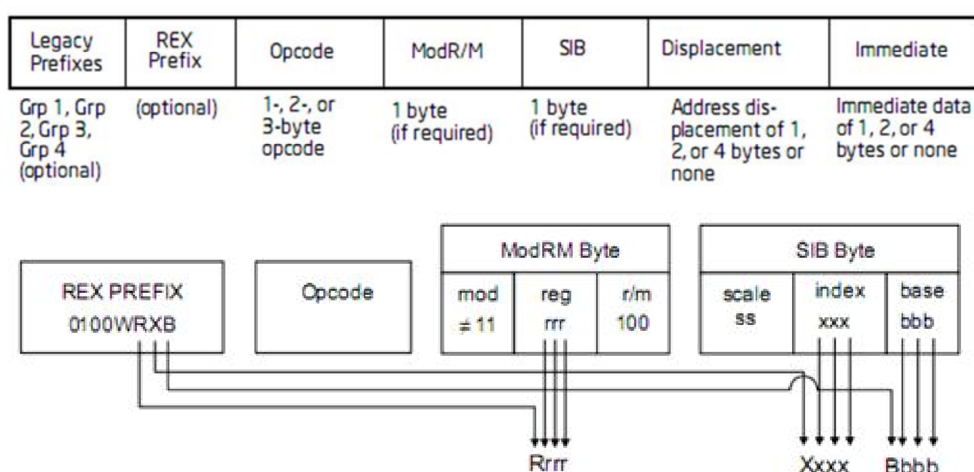


Рис. 3.25. Формат 64-х разрядных команд.

Более подробную информацию можно получит в [1, 2, 13, 14].

ГЛАВА 4. ПАМЯТЬ. НИЖНИЙ УРОВНЬ

Память предназначена для фиксации, хранения и выдачи информации в процессе работы ЭВМ. Процессы чтения и записи информации определяются как процессы обращения к запоминающему устройству (ЗУ). Емкость ЗУ характеризуют числом битов либо байтов, которое мажет храниться в запоминающем устройстве. На практике применяются более крупные единицы, а для их обозначения к славам «бит» или «байт» добавляют приставки: кило, мега, гига, тера, пета, экза (kila, mega, giga, tera, peta, exa). Важной характеристикой ЗУ является единица пересылки. Для основной памяти единица пересылки определяется шириной шины данных, то есть количеством битов, передаваемых по линиям шины параллельно. Обычно единица пересылки равна длине слова, на не обязательно. Применительно к внешней памяти данные часто передаются единицами, превышающими размер слова, и такие единицы называются блоками.

Говоря о физическом типе запоминающего устройства, необходимо выделить три наиболее распространенных технологии ЗУ - это полупроводниковая память, память с магнитным носителем информации, используемая в магнитных дисках; и память с оптическим носителем - оптические диски.

4.1. Методы доступа

Различают четыре основных метода доступа. С каждым из них связана своя организация памяти, их необходимо учитывать при оценке быстродействия ЭВМ.

Последовательный доступ. ЗУ с последовательным доступам ориентированных на хранение информации в виде последовательности блоков данных, называемых записями. Для доступа к нужному элементу *(слову* или байту) необходимо прочитать все предшествующие ему данные. Время доступа зависит от положения требуемой записи в последовательности записей на носителе информации и позиции элемента

внутри данной записи. Примером может служить ЗУ на магнитной ленте или расположение данных на оптическом диске (рис. 4.1).

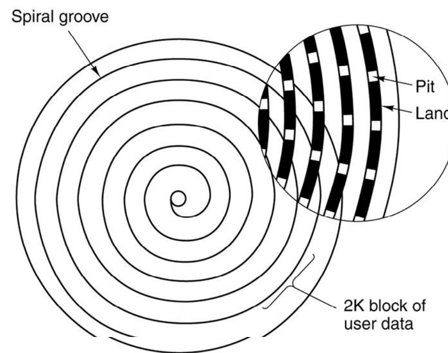


Рис. 4.1. Схема доступа к данным оптического диска.

Прямой доступ. Каждая запись имеет уникальный адрес, отражающий ее физическое размещение на носителе информации. Обращение осуществляется как адресный доступ к началу записи, с последующим последовательным доступом к определенной единице информации внутри записи. В результате время доступа к определенной позиции является величиной переменной. Такой режим характерен для магнитных дисков (рис. 4.2).

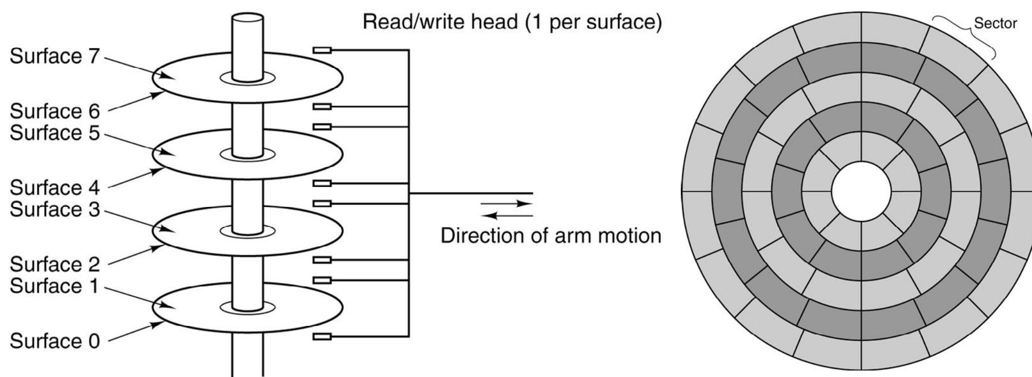


Рис. 4.2. Схема доступа к данным винчестера.

Произвольный доступ. Каждая ячейка памяти имеет уникальный физический адрес. Обращение к любой ячейке занимает одно и то же время и мажет проводиться в произвольной очередности. Примерам могут служить запоминающие устройства основной памяти.

Ассоциативный доступ. Этот вид доступа позволяет выполнять поиск ячеек, содержащих такую информацию, в которой значение отдельных битов

совпадает с состоянием одноименных битов в заданном образце. Сравнение осуществляется параллельно для всех ячеек памяти, независимо от ее емкости. По ассоциативному принципу построены блоки кэш-памяти.

Параметры быстродействия ЗУ

Время доступа. Для памяти с произвольным доступам оно соответствует интервалу времени от момента поступления адреса до момента, когда данные заносятся в память или становятся доступными. В ЗУ с подвижным носителем информации - это время, затрачиваемое на установку головки записи/считывания (или носителя) в нужную позицию.

Длительность цикла памяти или период обращения ($T_Ц$). Понятие применяется к памяти с произвольным доступом, для которой оно означает минимальное время между двумя последовательными обращениями к памяти. Период обращения включает в себя время доступа плюс некоторое дополнительное время. Дополнительное время может требоваться для затухания сигналов на линиях, а в некоторых типах ЗУ, где считывание информации приводит к ее разрушению - для восстановления считанной информации.

Скорость передачи. Эта скорость, с которой данные могут передаваться в память или из нее. Для памяти с произвольным доступом она равна $1/T_Ц$. Для других видов памяти скорость передачи определяется соотношением: $T_N = T_A + N/R$, где T_N - среднее время считывания или записи N битов; T_A - среднее время доступа; R - скорость пересылки в битах в секунду.

4.2. Иерархия запоминающих устройств

Память часто называют «узким местом» фон-Неймановских ЭВМ из-за ее серьезного отставания по быстродействию от процессоров, причем разрыв этот неуклонно увеличивается. Так, если производительность процессоров ежегодно возрастает вдвое примерно каждые 1,5 года, то для микросхем памяти прирост быстродействия не превышает 9% в год (удвоение за 10 лет),

что выражается в увеличении разрыва в быстродействии между процессором и памятью приблизительна на 50% в год.

При создании системы памяти постоянно приходится решать задачу обеспечения требуемой емкости и высокого быстродействия за приемлемую цену. Наиболее эффективным решением является создание иерархической памяти. Иерархическая память состоит из ЗУ различных типов (рис. 4.3.), которые, в зависимости от характеристик, относят к определенному уровню иерархии. Более высокий уровень меньше по емкости, быстрее и имеет большую стоимость в пересчете на бит, чем более низкий уровень. Уровни иерархии взаимосвязаны: все данные на одном уровне могут быть также найдены на более низком уровне, и все данные на этом более низком уровне могут быть найдены на следующем нижележащем уровне и т. д.

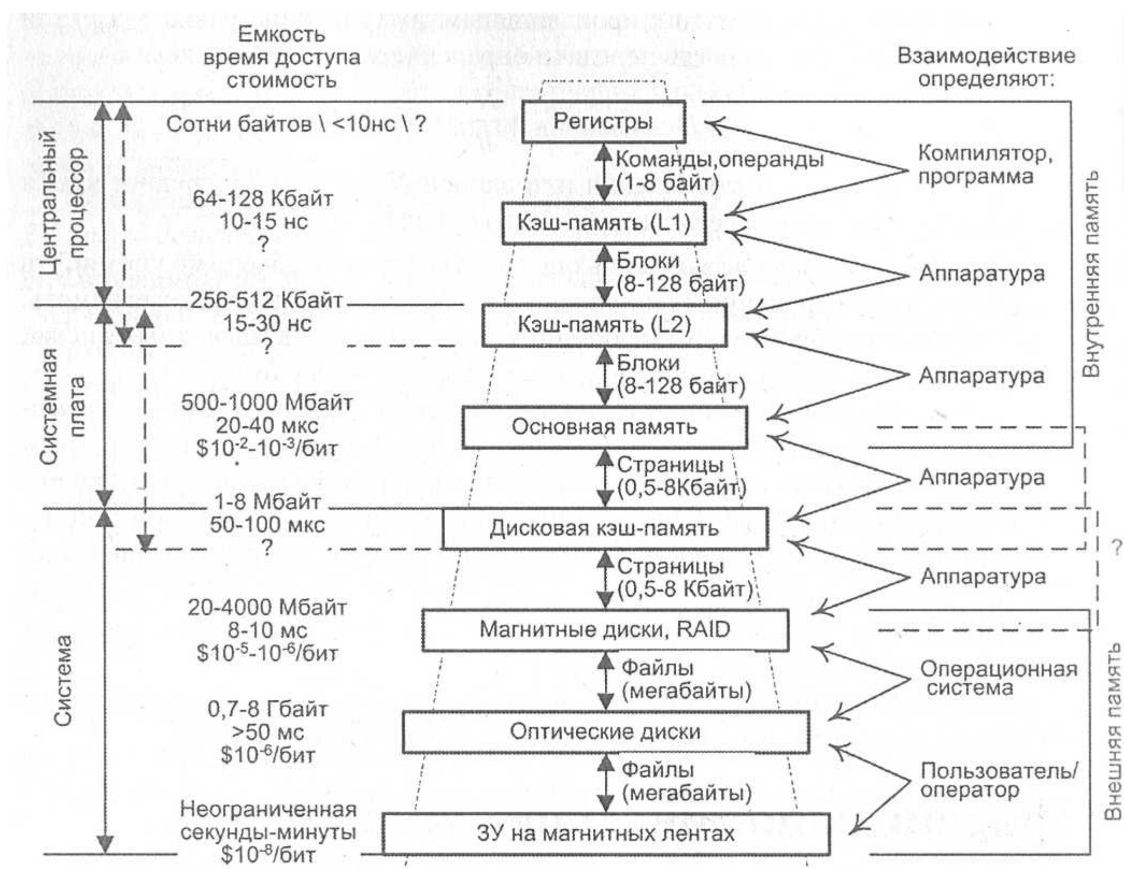


Рис. 4.3. Схема иерархической памяти [11, стр.200]

Четыре верхних уровня иерархии образуют внутреннюю память ЭВМ, а все нижние уровни - это внешняя или вторичная память. По мере движения вниз по иерархической структуре:

I. Уменьшается соотношение «стоимость/бит».

2. Возрастает емкость:

3. Растет время доступа.

4. Уменьшается частота обращения к памяти со стороны центрального процессора.

Если память организована в соответствии с пунктами 1-3, а характер размещения в ней данных и команд удовлетворяет пункту 4, иерархическая организация ведет к уменьшению общей стоимости при заданном уровне производительности.

Справедливость этого утверждения вытекает из принципа *локальности по обращению*. Если рассмотреть процесс выполнения большинства программ, то можно заметить, что с очень высокой вероятностью адрес очередной команды программы либо следует непосредственно за адресом, по которому была считана текущая команда, либо расположен вблизи него. Такое расположение адресов называется *пространственной локальностью программы*. Обрабатываемые данные, как правило, структурированы, и такие структуры обычно хранятся в последовательных ячейках памяти. Данная особенность про грамм называется *пространственной локальностью данных*. Кроме того, программы содержат множество небольших циклов и подпрограмм. Это означает, что небольшие наборы команд могут многократно повторяться в течение некоторого интервала времени, то есть имеет место *временная локальность*. Все три вида локальности объединяет понятие *локальность* по обращению. Принцип локальности часто облекают в численную форму и представляют в виде так называемого правила «90/10»: 90% времени работы программы связано с доступом к 10% адресного пространства этой программы.

Из свойства локальности вытекает, что программу разумно представить в виде последовательно обрабатываемых фрагментов. Помещая такие фрагменты в более быструю память, можно существенно снизить общие задержки на обращение, поскольку команды и данные, будучи один раз переданы из медленного ЗУ в быстрое, затем могут использоваться многократно, и среднее время доступа к ним в этом случае определяется уже более быстрым ЗУ.

На каждом уровне иерархии информация разбивается на блоки, которые и пересылаются между уровнями. При доступе к командам и данным, например, для их считывания, сначала производится поиск в памяти верхнего уровня. Факт обнаружения нужной информации называют попаданием (hit), в противном случае говорят о промахе (miss). При промахе производится поиск в ЗУ следующего более низкого уровня, где также возможны попадание или промах. После обнаружении необходимой информации выполняется последовательная пересылка блока, содержащего искомую информацию, с нижних уровней на верхние. Следует отметить, что независимо от числа уровней иерархии пересылка информации может осуществляться только между двумя соседними уровнями.

При оценке эффективности подобной организации памяти обычно используют следующие характеристики:

- *коэффициент попаданий* (hit rate) - отношение числа обращений к памяти, при которых произошло попадание, к общему числу обращений к ЗУ данного уровня иерархии;
- *коэффициент промахов* (miss rate) - отношение числа обращений к памяти, при которых имел место промах, к общему числу обращений к ЗУ данного уровня иерархии;
- *время обращения при попадании* (hit time) - время, необходимое для поиска нужной информации в памяти верхнего уровня, плюс время на фактическое считывание данных;

- *потери на промах* (miss penalty) - время, требуемое для замены блока в памяти более высокого уровня на блок с нужными данными, расположенный в ЗУ следующего (более низкого) уровня.

Описание некоторого уровня иерархии ЗУ предполагает конкретизацию четырех моментов:

- размещения блока - допустимого места расположения блока на примыкающем сверху уровне иерархии;
- идентификации блока - способа нахождения блока;
- замещения блока - выбора блока, заменяемого при промахе с целью освобождения места для нового блока;
- согласования копий (когерентность данных) - обеспечения согласованности копий одних и тех же блоков, расположенных на разных уровнях.

4.3. Основная память. ОЗУ

Основная память представляет собой единственный вид памяти, к которой ЦП может обращаться непосредственно. Основную память образуют запоминающие устройства с произвольным доступом. Каждая ячейка имеет уникальный адрес, позволяющий различать ячейки при обращении к ним для выполнения операций записи и считывания.

Основная память может включать в себя два типа устройств: оперативные запоминающие устройства (ОЗУ) и постоянные запоминающие устройства (ПЗУ). Преимущественную долю основной памяти образует ОЗУ, называемое оперативным, потому что оно допускает как запись, так и считывание информации, причем обе операции выполняются однотипно, практически с одной и той же скоростью. В англоязычной литературе ОЗУ соответствует аббревиатура RAM - Random Access Memory (рис. 4.4). Для большинства типов полупроводниковых ОЗУ характерна энергозависимость - даже при кратковременном прерывании питания хранимая информация теряется. Микросхема ОЗУ должна быть

постоянно подключена к источнику питания и поэтому может использоваться только как временная память.

Memory Type	Category	Erasure	Write Mechanism	Volatility
Random-access memory (RAM)	Read-write memory	Electrically, byte-level	Electrically	Volatile
Read-only memory (ROM)	Read-only memory	Not possible	Masks	Nonvolatile
Programmable ROM (PROM)			Electrically	
Erasable PROM (EPROM)	Read-mostly memory	UV light, chip-level		
Electrically Erasable PROM (EEPROM)		Electrically, byte-level		
Flash memory		Electrically, block-level		

Рис. 4.4. Типы полупроводниковой памяти.

Вторую группу полупроводниковых ЗУ основной памяти образуют энергонезависимые микросхемы ПЗУ (ROM – Read Only Memory). ПЗУ обеспечивает считывание информации, но не допускает ее изменения (в ряде случаев информация в ПЗУ может быть изменена, но этот процесс сильно отличается от считывания и требует значительно большего времени).

Энергозависимые ОЗУ можно подразделить на две основные подгруппы: динамическую память (DRAM - Dynamic Random Access Memory) и статическую память (SRAM -,-- Static Random Access Memory).

В статических ОЗУ запоминающий элемент может хранить записанную информацию неограниченно долго (при наличии питающего напряжения). Запоминающий элемент динамического ОЗУ способен хранить информацию только в течение достаточно короткого промежутка времени, после которого информацию нужно восстанавливать заново, иначе она будет потеряна. Динамические ЗУ, как и статические, энергозависимы.

Роль запоминающего элемента в статическом ОЗУ исполняет триггер. Триггер представляет собой схему с двумя устойчивыми состояниями, обычно состоящую из четырех или шести транзисторов (рис. 4.5).

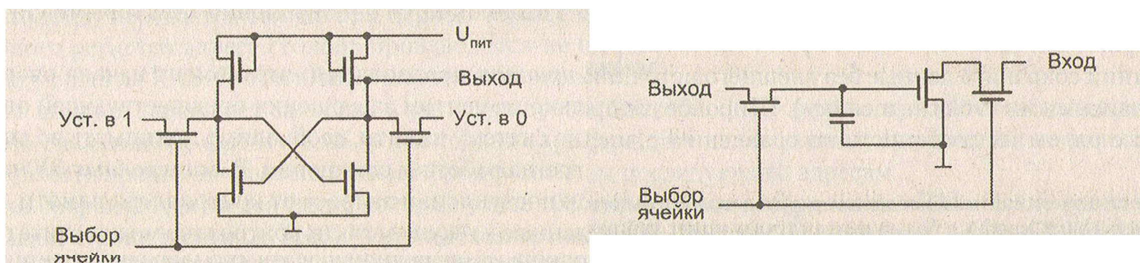


Рис. 4.5. Схема элементов статического и динамического ЗУ.

Запоминающий элемент (ЗЭ) динамической памяти значительно проще. Он состоит из одного конденсатора и запирающего транзистора (рис. 4.5). Простота схемы позволяет достичь высокой плотности размещения, в итоге, снизить стоимость. Главный недостаток подобной технологии связан с тем, что накапливаемый на конденсаторе заряд со временем теряется. Среднее время утечки заряда ЗЭ динамической памяти составляет сотни или даже десятки миллисекунд, поэтому, заряд необходимо успеть восстановить в течение данного отрезка времени, иначе информация будет утеряна. Периодическое восстановление заряда ЗЭ называется регенерацией и осуществляется каждые 2-100 мс.

Блочная организация основной памяти

Адресное пространство памяти разбито на группы последовательных адресов, каждая такая группа обеспечивается отдельным банком памяти. Для обращения используется 9-разрядный адрес, семь младших разрядов которого (A_6-A_0) поступают параллельно на все банки памяти и выбирают в каждом из них одну ячейку. Два старших разряда адреса (A_8, A_7) содержат номер банка. Выбор банка обеспечивается либо с помощью дешифратора номера банка памяти, либо путем мультиплексирования информации (на рис. 4.6 показаны оба варианта). В функциональном отношении такая память может рассматриваться как единое ЗУ, емкость которого равна суммарной емкости отдельных банков, а быстродействие - быстродействию отдельного банка.

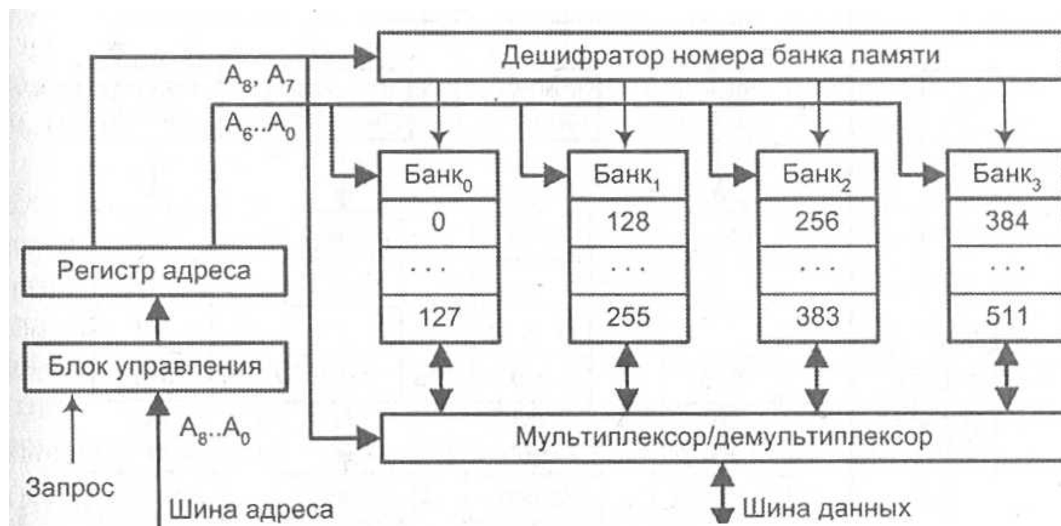


Рис. 4.6. Блочная организация памяти.

4.4. Микросхемы памяти

Интегральные микросхемы памяти организованы в виде матрицы ячеек, каждая из которых, в зависимости от разрядности, состоит из одного или более запоминающих элементов ЗЭ и имеет свой адрес (рис. 4.7). Каждый ЗЭ способен хранить один бит информации.

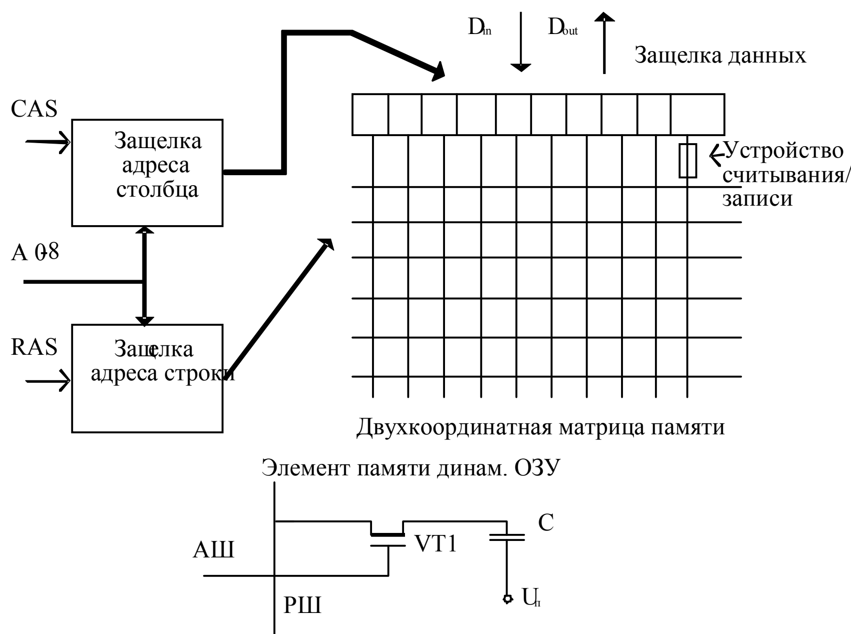


Рис. 4.7. Адресная схема организации памяти.

При матричной организации микросхем памяти реализуется координатный принцип адресации ячеек. Адрес ячейки, поступающий по шине адреса ЭВМ, пропускается через логику выбора, где он разделяется на

две составляющие: адрес строки и адрес столбца. Адреса строки и столбца запоминаются соответственно в регистре защелки адреса строки и регистре защелки адреса столбца микросхемы. Регистры соединяются каждый со своим дешифратором. Выходы дешифраторов образуют систему горизонтальных и вертикальных линий, к которым подсоединены запоминающие элементы матрицы, при этом каждый ЗЭ расположен на пересечении одной горизонтальной и одной вертикальной линии.

ЗЭ, объединенные общим горизонтальным проводом, принято называть строкой (row). Запоминающие элементы, подключенные к общему вертикальному проводу, называют столбцом (column). Фактически вертикальных проводов в микросхеме должно быть, по крайней мере, вдвое больше, чем это требуется для адресации, поскольку к каждому ЗЭ необходимо подключить линию, по которой будет передаваться считанная и записываемая информация.

Совокупность запоминающих элементов и логических схем, связанных с выбором строк и столбцов, называют ядром микросхемы памяти. Помимо ядра в микросхеме имеется еще интерфейсная логика, обеспечивающая взаимодействие ядра с внешним миром. В ее задачи, в частности, входят коммутация нужного столбца на выход при считывании и на вход - при записи.

На физическую организацию ядра, как матрицы однобитовых ЗЭ, накладывается логическая организация памяти, под которой понимается разрядность микросхемы, то есть количество линий ввода/вывода. Разрядность микросхемы определяет количество ЗЭ, имеющих один и тот же адрес (такая совокупность запоминающих элементов называется ячейкой), то есть каждый столбец содержит столько разрядов, сколько есть линий ввода/вывода данных.

Для уменьшения числа контактов микросхемы адреса строки и столбца в большинстве микросхем подаются через одни и те же контакты

последовательно во времени (мультиплексируются) и запоминаются соответственно в регистре адреса строки и регистре адреса столбца микросхемы. Мультиплексирование обычно реализуется внешней логикой.

Для синхронизации процессов фиксации и обработки адресной информации адрес строки (RA) сопровождается сигналом RAS (Row Address Strobe - строб строки), а адрес столбца (CA) - сигналом CAS (Column Address Strobe - строб столбца). Чтобы стробирование было надежным, эти сигналы подаются с задержкой, достаточной для завершения переходных процессов на шине адреса в адресных цепях микросхемы. Сигнал выбора микросхемы CS (Crystal Select) разрешает работу схемы и используется для выбора определенной микросхемы в системах, состоящих из нескольких микросхем. Вход WE (Write Enable - разрешение записи) определяет вид выыполняемой операции (считывание или запись). На все время пока микросхемы памяти не использует шину данных, информационные выходы микросхемы переводятся в третье (высокоимпедансное) состояние. Управление переключением в третье состояние обеспечивается сигналом OE (Output Enable - разрешение выдачи выходных сигналов). Этот сигнал активизируется при выполнении операции чтения. На рис. 4.8 представлено изображение ОЗУ на принципиальных схемах.

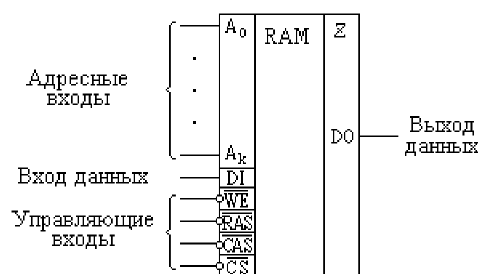


Рис. 4.8. Условное изображение ОЗУ

Структурная схема БИС динамического ОЗУ с четырьмя банками памяти показана на рис.4.9. Основными ее компонентами являются четыре банка памяти, представляющих собой матрицы элементов памяти с

дешифраторами строк и столбцов и усилителями чтения-записи. Кроме собственно банков памяти в состав ОЗУ входят:

- буфер адреса, фиксирующий адреса строки и столбца;

- счетчик регенерации, формирующий адрес строки, в которой должна выполняться очередная регенерация;

- дешифратор команд, определяющий, какое действие (команду) должна выполнить микросхема в соответствии с поданными управляющими сигналами (и сигналом A10);

- схемы управления, формирующие управляющие сигналы для остальных узлов микросхемы;

- схемы коммутации данных, передающие читаемые или записываемые данные из/в банки памяти;

- буфер ввода/вывода данных, обеспечивающий связь микросхемы памяти с шиной данных.

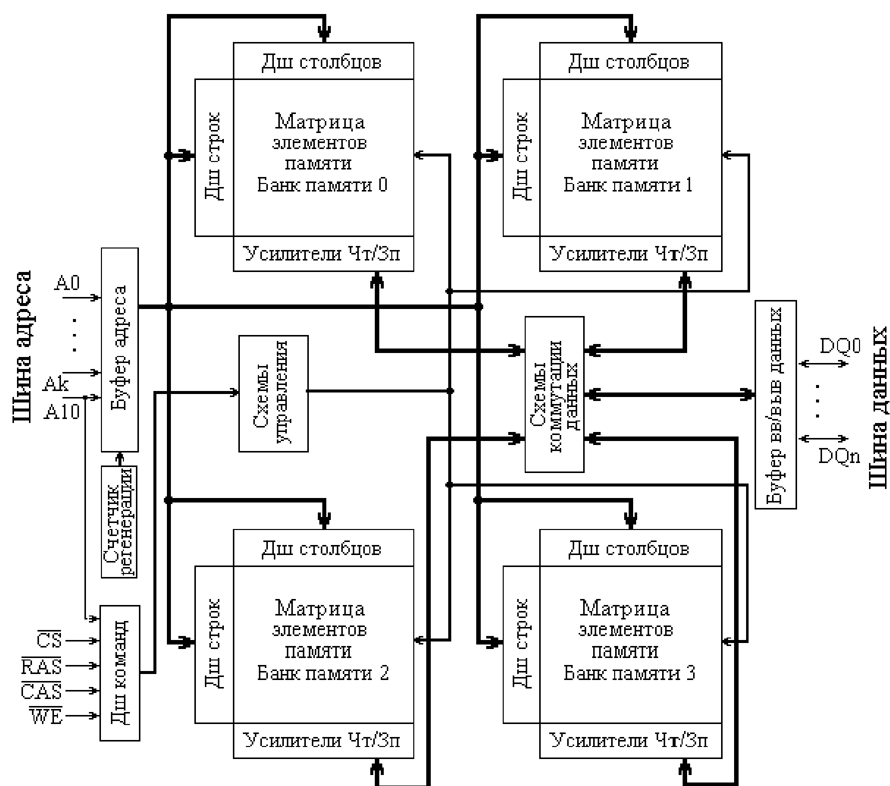


Рис. 4.9. Структурная схема БИС динамического ОЗУ.

Управление операциями с основной памятью осуществляется

контроллером памяти. Обычно этот контроллер входит в состав центрального процессора либо реализуется в виде внешнего по отношению к памяти устройства. В последних микросхемах памяти часть функций контроллера возлагается на микросхему. В общем случае на каждую операцию (считывание или запись) требуется как минимум пять тактов, которые используются следующим образом:

1. Указание типа операции (чтение или запись) и установка адреса строки.
2. Формирование сигнала RAS.
3. Установка адреса столбца.
4. Формирование сигнала CAS.
5. Возврат сигналов RAS и CAS в неактивное состояние. Обратная запись.

Типовую процедуру доступа к памяти рассмотрим на примере чтения с мультиплексированием адресов строки столбцов. Сначала на входе WE устанавливается уровень, соответствующий операции чтения, а на адресные контакты микросхемы подается адрес строки, сопровождаемый сигналом RAS. По заднему фронту этого сигнала адрес запоминается в регистре адреса строки, после чего дешифрируется. После стабилизации процессов, вызванных сигналом RAS, выбранная строка подключается к усилителям считывания/записи (УСЗ). Далее на вход подается адрес столбца, который по заднему фронту сигнала CAS заносится в регистр адреса столбца. Одновременно подготавливается выходной регистр данных, куда после стабилизации сигнала CAS загружается информация с выбранных УСЗ.

Разработчики микросхем памяти тратят значительные усилия на повышение быстродействия микросхем. Возможности «ускорения» ядра микросхемы ЗУ весьма ограничены и связаны в основном с миниатюризацией запоминающих элементов. Наибольшие успехи достигнуты в интерфейсной части, касаются они, главным образом,

операций чтения, то есть способов доставки содержимого ячейки на шину данных. Наибольшее распространение получили следующие шесть фундаментальных подходов: последовательный; конвейерный; регистровый; страничный; пакетный; удвоенной скорости. Они будут рассмотрены при обсуждении типов микросхем.

Типы микросхем динамических ОЗУ

Динамическая память состоит из ядра (массива ЗЭ) и интерфейсной логики (буферных регистров, усилителей чтения данных, схемы регенерации и др.). Хотя количество видов DRAM уже превысило два десятка, ядро у них организовано практически одинаково. Главные различия связаны с интерфейсной логикой, причем различия эти обусловлены также и областью применения микросхем - помимо основной памяти ЭВМ, микросхемы памяти входят, например, в состав видеоадаптеров. Классификация микросхем динамической памяти показана на рис. 4.10.

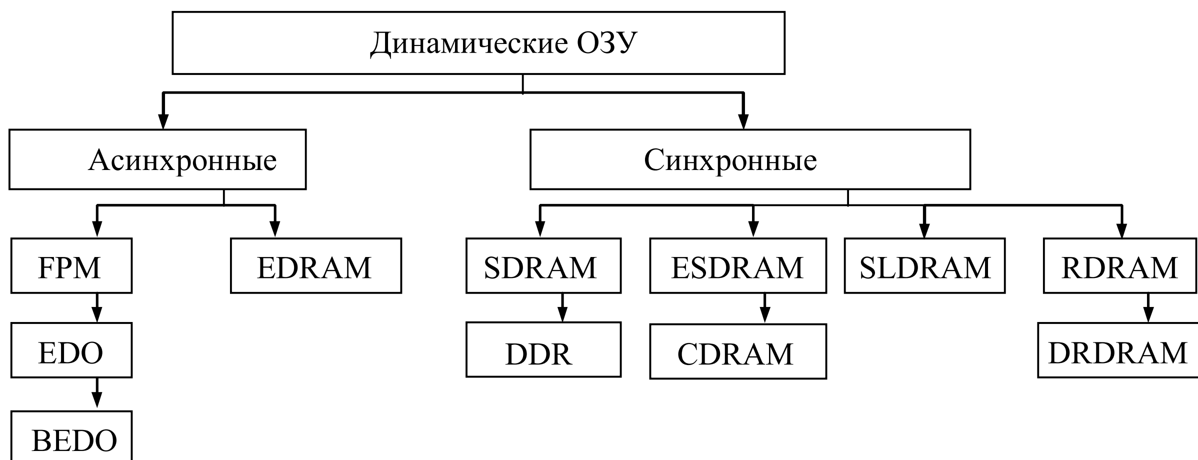


Рис. 4.10. Типы микросхем динамического ОЗУ

Теперь рассмотрим различные типы микросхем динамической памяти DRAM. На начальном этапе это были микросхемы асинхронной памяти, работа которых не привязана жестко к тактовым импульсам системной шины. Асинхронной памяти свойственны дополнительные затраты времени на взаимодействие микросхем памяти и контроллера, Так, в асинхронной схеме сигнал RAS будет сформирован только после поступления в

контроллер тактирующего импульса и будет воспринят микросхемой памяти через некоторое время.

Микросхемы DRAM. В первых микросхемах динамической памяти применялся наиболее простой способ обмена данными. Он позволял считывать и записывать строку памяти только на каждый пятый такт (рис. 4.11). Этапы такой процедуры были описаны ранее. Традиционной DRAM соответствует формула 5-5-5-5. Микросхемы данного типа могли работать на частотах до 40 МГц и из-за своей медлительности (время доступа составляло около 120 нс) просуществовали недолго.

Микросхемы FPM DRAM. Микросхемы динамического ОЗУ, реализующие режим FPM (Fast Page Mode), также относятся к ранним типам DRAM. В основе лежит следующая идея. Доступ к ячейкам, лежащим в одной строке матрицы, можно проводить быстрее. Для доступа к очередной ячейке достаточно подавать на микросхему лишь адрес нового столбца, сопровождая его сигналом CAS. Полный же адрес (строки и столбца) передается только при первом обращении к строке. Сигнал RAS остается активным на протяжении всего страничного цикла и позволяет заносить в регистр адреса столбца новую информацию не по спадающему фронту CAS, а как только адрес на входе стабилизируется, то есть практически по переднему фронту сигнала CAS. Схема чтения для FPM DRAM (рис. 4.11) описывается формулой 5-3-3-3 (всего 14 тактов). Применение схемы быстрого страничного доступа позволило сократить время доступа до 60 нс.

Микросхемы EDO DRAM. Следующим этапом в развитии динамических ОЗУ стали микросхемы с гиперстраничным режимом доступа (HPM, Hyper Page Mode), более известные как EDO (Extended Data Output - расширенное время удержания данных на выходе). Главная особенность технологии - увеличенное по сравнению с FPM DRAM время доступности данных на выходе микросхемы. В микросхемах FPM DRAM выходные

данные остаются действительными только при активном сигнале CAS, за счет чего во втором и последующих доступах к строке нужно три такта: такт переключения CAS в активное состояние, такт считывания данных и такт переключения CAS в неактивное состояние. В EDO DRAM по активному (спадающему) фронту сигнала CAS данные запоминаются во внутреннем регистре, где хранятся еще некоторое время после того, как поступит следующий активный фронт сигнала. Это позволяет использовать хранимые данные, когда CAS уже переведен в неактивное состояние. Схема чтения у EDO DRAM уже 5-2-2-2, что на 20% быстрее, чем у FPM. Время доступа составляет порядка 30-40 нс.

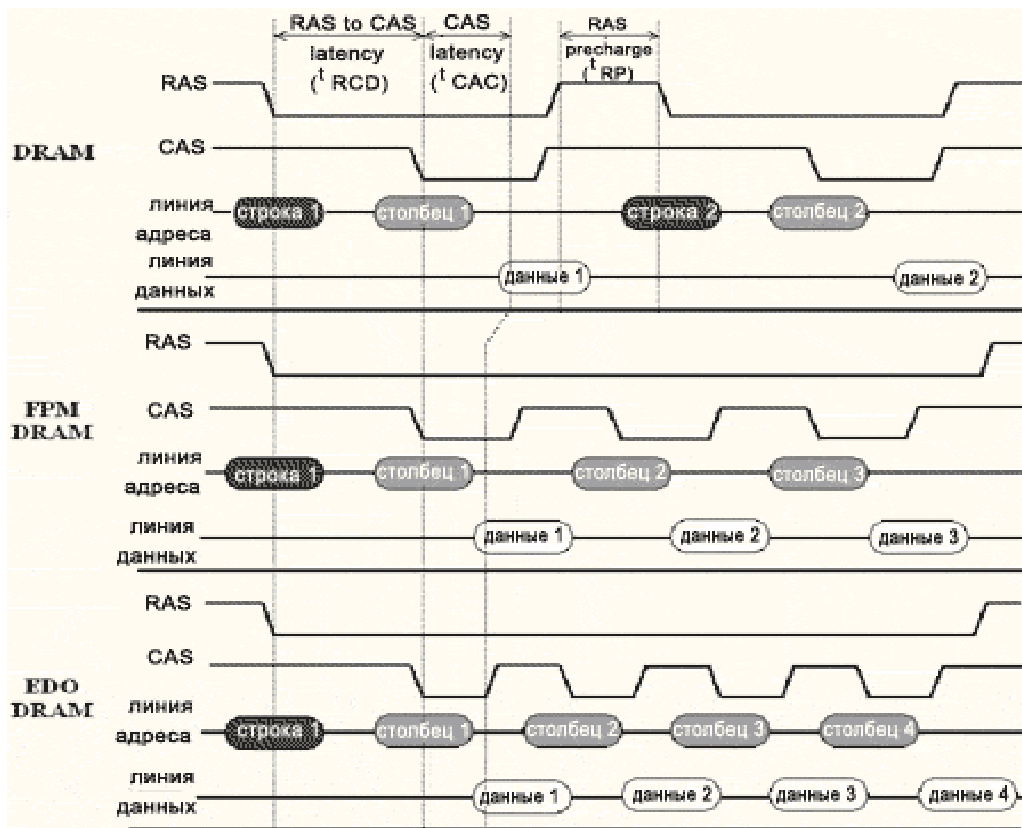


Рис. 4.11. Временные диаграммы работы микросхем памяти DRAM, FPM DRAM, EDO DRAM.

Микросхемы BEDO DRAM. Технология EDO, была усовершенствована компанией VIA Technologies. Новая модификация EDO известна как BEDO (Burst EDO - пакетная EDO). Новизна метода в том, что при первом обращении считывается вся строка микросхемы, в которую

входят последовательные слова пакета. За последовательной пересылкой слов (переключением столбцов) автоматически следит внутренний счетчик микросхемы. Это исключает необходимость выдавать адреса для всех ячеек пакета, но требует поддержки со стороны внешней логики. Способ позволяет сократить время считывания второго и последующих слов еще на один такт (рис. 4.12), благодаря чему формула приобретает вид 5-1-1-1.

Микросхемы SDRAM. Аббревиатура SDRAM (Synchronous DRAM - Синхронная DRAM) используется для обозначения микросхем «обычных» синхронных динамических ОЗУ. Кардинальные отличия SDRAM от рассмотренных выше асинхронных динамических ОЗУ можно свести к четырем положениям:

- синхронный метод передачи данных на шину
- применение нескольких (двух или четырех) внутренних банков памяти
- конвейерный механизм пересылки пакета
- передача части функций контроллера памяти логике самой микросхемы

Синхронность памяти позволяет контроллеру памяти «знать» моменты готовности данных, за счет чего снижаются издержки циклов ожидания и поиска данных. Так как данные появляются на выходе микросхемы одновременно с тактовыми импульсами, упрощается взаимодействие памяти с другими устройствами ЭВМ. В отличие от BEDO конвейер позволяет передавать данные пакета по тактам, благодаря чему ОЗУ может работать бесперебойно на более высоких частотах, чем асинхронные ОЗУ.

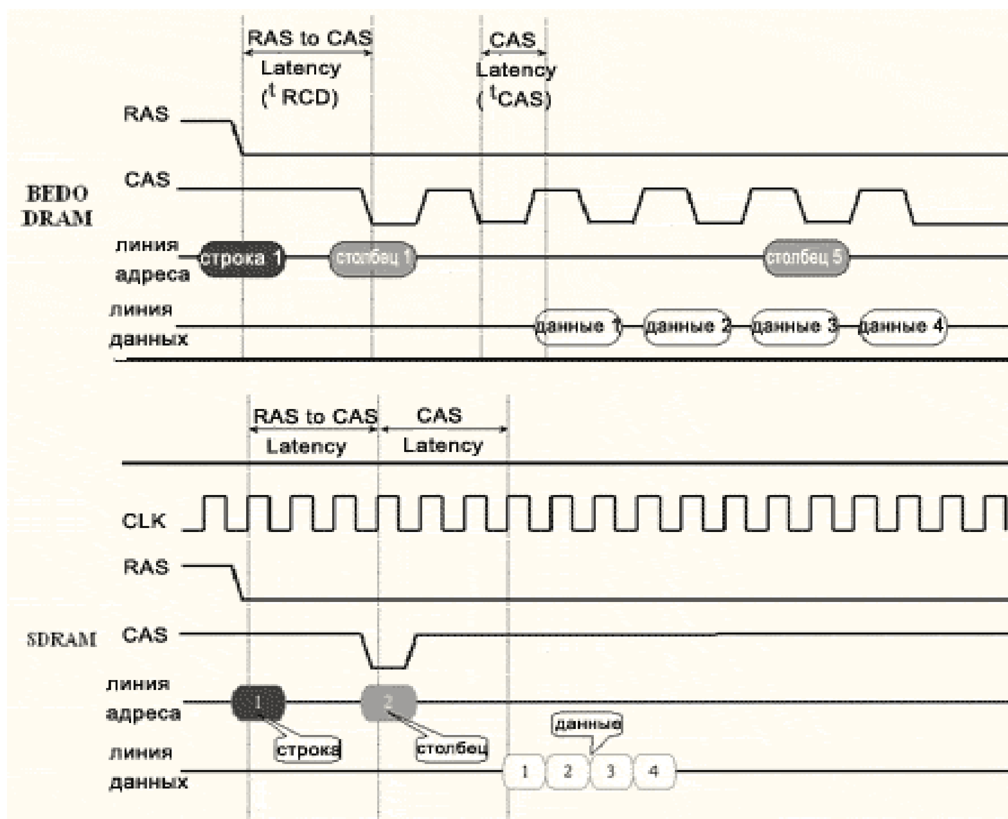


Рис. 4.12. Временные диаграммы работы микросхем памяти BEDO DRAM, SDRAM.

Микросхемы DDR SDRAM. Важным этапом в дальнейшем развитии технологии SDRAM стала DDR SDRAM (Double Data Rate SDRAM - SDRAM с удвоенной скоростью передачи данных). В отличие от SDRAM новая модификация выдает данные в пакетном режиме по обоим фронтам импульса синхронизации, за чего пропускная способность возрастает вдвое.

Микросхемы RDRAM, DRDRAM. Принципиально отличный подход к построению DRAM был предложен компанией Rambus в 1997 году. В нем используется оригинальная система обмена данными между ядром и контроллером памяти. Ведутся работы по повышению быстродействия, в частности, связанные с применением КЭШ в микросхемах (CDRAM).

Модули динамической памяти

Модули динамической полупроводниковой памяти прошли эволюцию от набора микросхем, устанавливаемых на системной плате и заметных по своему регулярному расположению (несколько смежных рядов одинаковых

микросхем) до отдельных небольших плат, вставляемых в стандартный разъем (слот) системной платы. Первенство в создании таких модулей памяти обычно относят к фирме IBM.

Основными разновидностями модулей динамических оперативных ЗУ с момента их оформления в виде самостоятельных единиц были:

- 30-контактные однобайтные модули SIMM (DRAM)
- 72-контактные четырехбайтные модули SIMM (DRAM)
- 168-контактные восьмибайтные модули DIMM (SDRAM)
- 184-контактные восьмибайтные модули DIMM (DDR SDRAM)
- 184-контактные (20 из них не заняты) двухбайтные модули RIMM RDDRAM).

Сокращение SIMM означает Single In-Line Memory Module – модуль памяти с одним рядом контактов, так как контакты краевого разъема модуля, расположенные в одинаковых позициях с двух сторон платы, электрически соединены. Соответственно DIMM значит Dual In-Line Memory Module – модуль памяти с двумя рядами контактов. А вот RIMM означает Rambus Memory Module – модуль памяти типа Rambus. Кроме этих модулей имеются также варианты для малогабаритных компьютеров, для графических карт и некоторые другие.

Если микросхемы памяти физически располагаются только с одной стороны платы, то такой модуль называют односторонним, а если с двух сторон – то двухсторонним. При равной емкости модулей, у двухстороннего модуля количество микросхем больше, поэтому на каждую линию шины данных приходится большая нагрузка, чем при использовании одностороннего. С этой точки зрения односторонние модули предпочтительней двусторонних. Однако количество банков в двусторонних модулях вдвое больше, чем в односторонних, поэтому при определенных условиях и хорошем контроллере памяти двусторонний модуль может обеспечить несколько большую производительность.

Помимо собственно конструктивной организации и типа памяти, модули имеют также и некоторые другие различия. Одним из таких различий является возможность (или ее отсутствие) контроля хранимых данных.

Контроль может основываться на использовании дополнительных (по одному на каждый хранимый байт) битов четности (Parity bits), т.е. в этом случае каждый байт занимает в памяти по 9 бит. Такой контроль позволяет выявить ошибки при считывании хранимой информации из памяти, но не исправить их. Более сложный контроль предполагает использование кодов, корректирующих ошибки – ECC (Error Correcting Codes), которые позволяют обнаруживать ошибки большей кратности, чем одиночные, а одиночные ошибки могут быть исправлены. Подобные схемы используются в серверных конфигурациях, когда требуется повышенная надежность. Память, устанавливаемая в настольные ПЭВМ, обычно не имеет никакого контроля.

Кроме того, известны также различные модификации схем контроля, вплоть до просто имитирующих контрольные функции, но не осуществляющие их, например, с генерацией всегда верного бита четности.

Модули DIMM также различаются по наличию или отсутствию в них буферных схем на шинах адреса и управляющих сигналов. Не буферизованные (unbuffered) модули больше нагружают эти шины, но более быстродействующие и дешевые. Их обычно применяют в настольных ЭВМ. Буферизованные (registered) имеют буферные регистры и, обеспечивая меньшую нагрузку на шины, позволяют подключить к ней большее количество модулей. Однако эти регистры несколько снижают быстродействие памяти, требуя лишнего такта задержки. Применяют буферизованные модули обычно в серверных системах.

Еще одной особенностью, различающей модули динамической памяти, является способ, посредством которого после включения компьютера определяется объем и тип установленной в нем памяти.

В первых персональных ЭВМ объем и быстродействие установленной памяти задавались переключателями (джамперами – jumpers), расположенными на системной плате. С появлением модулей SIMM (существовали также похожие на них модули SIPP) стал использоваться так называемый параллельный метод идентификации (parallel presence detect), при котором краевой разъем модуля имел дополнительные контакты, используемые только для целей указания присутствия модуля в том слоте, где он установлен, его объема и времени обращения. В самых первых (30-контактных) модулях таких дополнительных контактов было только два, в 72-контактных модулях их стало четыре: два указывали на объем модуля и два – на время обращения. Эти контакты могли заземляться непосредственно на модуле, что позволяло различить четыре вида модулей по объему и четыре – по времени доступа.

Попытки использовать этот же прием в последующих модулях потребовали увеличения количества таких контактов, но решить все проблемы идентификации не смогли. Поэтому, начиная с модулей DIMM, используют так называемый последовательный способ идентификации (Serial Presence Detect - SPD), при котором на плату модуля устанавливается специальная дополнительная микросхема, так называемый SPD-чип, представляющая собой небольшую постоянную память на 128 или 256 байт с последовательным (I2C) интерфейсом доступа. В этой микросхеме записана основная информация об изготовителе микросхемы и ее параметрах. Формат этих данных стандартный, определенный советом JEDEC (Joint Electron Devices Engineering Council), стандартов которого придерживаются все изготовители полупроводниковой памяти.

4.5. Регенерация памяти

В различных типах микросхем динамической памяти нашли применение три основные метода регенерации:

1. одним сигналом RAS (ROR - RAS Only Refresh);

2. сигналом CAS, предваряющим, сигнал RAS (CBR - CAS Before RAS);
3. скрытая и автоматическая регенерация (SR - Self Refresh).

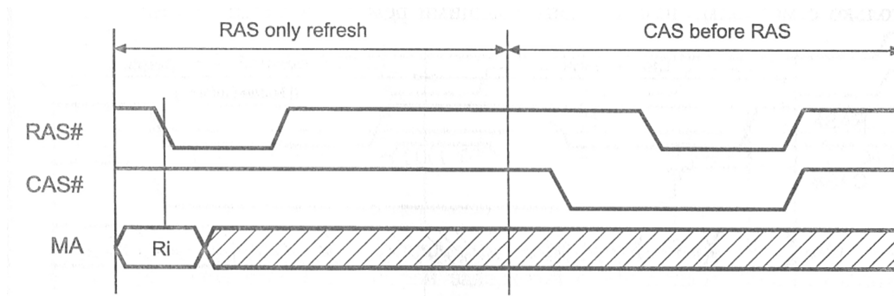


Рис. 4.13. Временные диаграммы регенерации памяти в режимах ROR, CBR [11].

Регенерация одним RAS использовалась еще в первых микросхемах DRAM. На шину адреса выдается адрес регенерируемой строки, сопровождаемый сигналом RAS. При этом выбирается строка ячеек и хранящиеся там данные поступают во внутренние цепи микросхемы, после чего записываются обратно. Так как сигнал CAS не появляется, цикл чтения/записи не начинается. В следующий раз на шину адреса подается адрес следующей строки и т. д., пока не восстановятся все строки, после чего цикл повторяется. К недостаткам метода можно отнести занятость шины адреса в момент регенерации, когда доступ к другим устройствам блокирован.

Особенность метода CBR в том, что если в обычном цикле чтения/записи сигнал RAS всегда предшествует сигналу CAS, то при появлении сигнала CAS первым начинается специальный цикл регенерации. В этом случае адрес строки не передается, а микросхема использует свой внутренний счетчик, содержимое которого увеличивается на единицу при каждом очередном CBR-цикле. Режим позволяет регенерировать память; не занимая шину адреса, то есть более эффективен.

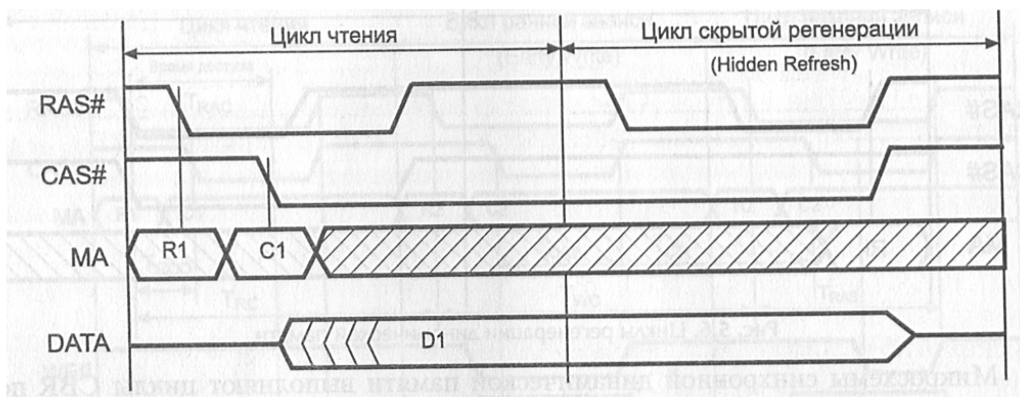


Рис. 4.14. Временные диаграммы регенерации памяти в режиме CBR – скрытой регенерации.

4.6. Обнаружение и исправление ошибок

При работе с полупроводниковой памятью не исключено возникновение различного рода отказов и сбоев. Причиной отказов могут быть производственные дефекты, повреждение микросхем или их физический износ. Проявляются отказы в том, что в отдельных разрядах одной или нескольких ячеек постоянно считывается 0 или 1, вне зависимости от реально записанной туда информации. Сбой - это случайное событие, выражающееся в неверном считывании или записи информации в отдельных разрядах одной или нескольких ячеек, не связанное с дефектами микросхемы. Сбои обычно обусловлены проблемами с источником питания или с воздействием альфа-частиц, возникающих в результате распада радиоактивных элементов, которые в небольших количествах присутствуют практически в любых материалах. Как отказы, так и сбои крайне нежелательны, поэтому в большинстве систем основной памяти содержатся схемы, служащие для обнаружения и исправления ошибок.

Вне зависимости от того, как именно реализуется контроль и исправление ошибок, в основе их всегда лежит введение избыточности. Это означает, что контролируемые разряды дополняются контрольными разрядами, благодаря которым и возможно детектирование ошибок, а в ряде методов - их коррекция. Общую схему обнаружения и исправления ошибок

иллюстрирует рис. 4.15.

На рисунке показано, каким образом осуществляются обнаружение и исправление ошибок. Перед записью М-разрядных данных в память производится их обработка, обозначенная на схеме функцией f, в результате которой формируется добавочный К-разрядный код. В память заносятся как данные, так и этот вычисленный код, то есть (М + К)-разрядная информация. При чтении информации повторно формируется -К-разрядный код, который сравнивается с аналогичным кодом, считанным из ячейки. Сравнение приводит к одному из трех результатов

- Не обнаружено ни одной ошибки. Извлеченные из ячейки данные подаются на выход памяти
- Обнаружена ошибка, и она может быть исправлена. Биты данных и добавочного кода подаются на схему коррекции. После исправления ошибки данные поступают на выход памяти
- Обнаружена ошибка, и она не может быть исправлена. Выдается сообщение о неисправимой ошибке.

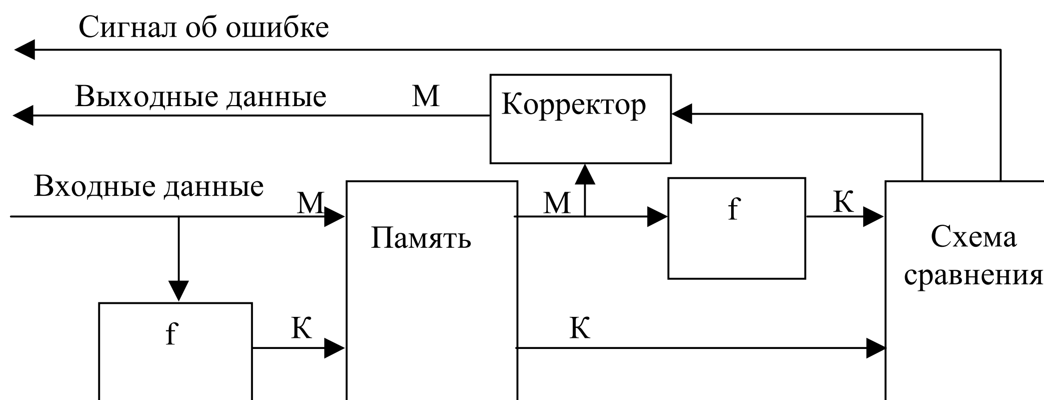


Рис. 4.15. Схема памяти с обнаружением и исправлением ошибок.

Коды, используемые для подобных операций, называют *корректирующими кодами* или кодами с исправлением ошибок.

Простейший вид такого кода основан на добавлении к каждому байту информации одного бита паритета. Бит паритета - это дополнительный бит, значение которого устанавливается таким, чтобы суммарное число единиц в данных, с учетом этого дополнительного разряда, было четным (или нечетным). В ряде систем за основу берется четность, в иных - нечетность.

Простейший вариант корректирующего кода также может быть построен на базе битов паритета. Для этого биты данных представляются в виде матрицы, к каждой строке и столбцу которой добавляется бит паритета. Для 64-разрядных данных этот подход иллюстрирует таблице. Здесь D - биты

	0	1	2	3	4	5	6	7	
0	D0	D1	D2	D3	D4	D5	D6	D7	C0
1	D8	D9	D10	D11	D12	D13	D14	D15	C1
2	D16	D17	D18	D19	D20	D21	D22	D23	C2
3	D24	D25	D26	D27	D28	D29	D30	D31	C3
4	D32	D33	D34	D35	D36	D37	D38	D39	C4
5	D40	D41	D42	D43	D44	D45	D46	D47	C5
6	D48	D49	D50	D51	D52	D53	D54	D55	C6
7	D56	D57	D58	D59	D60	D61	D62	D63	C7
	K0	K1	K2	K3	K4	K5	K6	K7	P

данных, С – столбец битов паритета строк, К - строка битов паритета столбцов, Р - бит паритета, контролирующий столбец С и строку К. Таким образом, к 64 битам данных нужно добавить 17 бит паритета: по 8 бит на строки и столбцы и один дополнительный бит для контроля строки и столбца битов паритета. Если в одной строке и одном столбце обнаружено нарушение паритета, для исправления ошибки достаточно просто инвертировать бит на пересечении этих строки и столбца. Если ошибка паритета выявлена только в одной строке или только одном столбце либо одновременно в нескольких строках и столбцах, фиксируется многобитовая ошибка и формируется признак невозможности коррекции.

Недостаток рассмотренного приема в том, что он требует большого числа дополнительных разрядов. Более эффективным представляется код, предложенный Ричардом Хэммингом и носящий его имя (код Хэмминга).

4.7. Флэш-память

Флэш-память, появившаяся в конце 1980-х годов (Intel), является представителем класса перепрограммируемых постоянных ЗУ с электрическим стиранием. Однако стирание в ней осуществляется сразу целой области ячеек: блока или всей микросхемы. Это обеспечивает более быструю запись информации или, как иначе называют данную процедуру, программирование ЗУ. Для упрощения этой процедуры в микросхему включаются специальные блоки, делающие запись "прозрачной" (подобной записи в обычное ЗУ) для аппаратного и программного окружения.

Флэш-память строится на однотранзисторных элементах памяти (с "плавающим" затвором), что обеспечивает плотность хранения информации даже несколько выше, чем в динамической оперативной памяти. Существуют различные технологии построения базовых элементов флэш-памяти, разработанные ее основными производителями. Эти технологии отличаются количеством слоев, методами стирания и записи данных, а также структурной организацией, что отражается в их названии. Наиболее широко известны NOR и NAND типы флэш-памяти, запоминающие транзисторы в которых подключены к разрядным шинам, соответственно, параллельно и последовательно. Первый тип имеет относительно большие размеры ячеек и быстрый произвольный доступ (порядка 70 нс), что позволяет выполнять программы непосредственно из этой памяти. Второй тип имеет меньшие размеры ячеек и быстрый последовательный доступ (обеспечивая скорость передачи до 16 Мбайт/с), что более пригодно для построения устройств блочного типа, например "твердотельных дисков".

Способность сохранять информацию при выключенном питании, малые размеры, высокая надежность и приемлемая цена привели к широкому ее распространению. Этот вид памяти применяется для хранения BIOS, построения так называемых "твердотельных" дисков (memory stick, memory drive и др.), карт памяти различного назначения и т.п. Причем устройства на

основе флэш-памяти используются не только в ЭВМ, но и во многих других применениях.

Элементы памяти флэш-ЗУ организованы в матрицы, как и в других видах полупроводниковой памяти. Разрядность данных для микросхем составляет 1-2 байта. Операция чтения из флэш-памяти выполняется как в обычных ЗУ с произвольным доступом (оперативных ЗУ или кэш). Однако запись сохраняет в себе некоторые особенности, аналогичные свойствам постоянных ЗУ. Перед записью данных в ЗУ ячейки, в которые будет производиться запись, должны быть очищены (стерты). Стирание заключается в переводе элементов памяти в состояние единицы и возможно только сразу для целого блока ячеек (в первых микросхемах предусматривалось стирание только для всей матрицы сразу). Выборочное стирание невозможно. Фактически при операции записи производится два действия: запись и считывание, но управление этими операциями производится внутренним автоматом и “прозрачно” для процессора.

4.7. Кэш-память

Как уже отмечалось, в качестве элементной базы основной памяти в большинстве ЭВМ служат микросхемы динамических ОЗУ, на порядок уступающие по быстродействию центральному процессору. В результате процессор вынужден простаивать несколько тактовых периодов. Если ОЗУ выполнить на быстрых микросхемах статической памяти, стоимость ЭВМ возрастет весьма существенно. Экономически приемлемое решение этой проблемы было предложено *М. Уилксом в 1965* году в процессе разработки ЭВМ Atlas и заключается оно в использовании двухуровневой памяти, когда между ОЗУ и процессором размещается небольшая, но быстродействующая буферная память. В процессе работы такой системы в буферную память копируются участки ОЗУ, к которым производится обращение со стороны процессора. Выигрыш достигается за счет ранее рассмотренного свойства локальности.

Уилкс называл рассматриваемую буферную память подчиненной (slave). Позже распространение получил термин кэш-память (от английского слова cache -;- убежище, тайник), поскольку такая память обычно скрыта от программиста в том смысле, что он не может ее адресовать и может даже вообще не знать о ее существовании. Впервые кэш-системы появились в машинах семейства IBM 360.

В общем виде использование кэш-памяти поясним следующим образом. Когда ЦП пытается прочитать слово из основной памяти, сначала осуществляется поиск копии этого слова в кэше. Если такая копия существует обращение к ОП не производится, а в ЦП передается слово; извлеченное из кэш-памяти. Данную ситуацию принято называть успешным обращением или попаданием (hit).При отсутствии слова в кэше, то есть при неуспешном обращении - промахе (miss), требуемое слово передается в ЦП из основной памяти, но одновременно из 0П в кэш-память пересылается блок данных, содержащий это слово.

Выигрыш в скорости при использовании кэш получается только в том случае, если перенесенные один раз в кэш фрагменты затем используются многократно. Причем, целесообразно помещать в кэш одновременно несколько разных участков ОЗУ, так как процессор одновременно работает с разными адресами, например, программой и данными. Согласно описанной выше локальности (локальной серийности) достаточно в кэш помещать 10 часть объема программы, чтобы 90% времени процессор работал с кэш.

Кэш-память в вычислительной системе оказывается «двойником» участков основной памяти с адресной организацией. При обращении к основной памяти формируется физический адрес, размер которого соответствует разрядности адресной шины (реальный объем физической памяти обычно меньше, а размер кэша в 20…100 раз меньше объема основной памяти). Как в этих условиях схемотехника управления памятью

может «понять», что элемент, к которому происходит обращение, находится в кэше? Кэш представляет собой ***ассоциативную память.***

Поскольку при наличии кэш-памяти адресуемый объект может существовать в нескольких экземплярах (В двухпроцессорной системе с двухуровневым кэшем, выполняющей самомодифицируемую программу (изменяющую собственные команды) один и тот же фрагмент кода может существовать в 9 экземплярах: 1) в основном ОЗУ, 2) и 3) в кэшах второго уровня двух процессорах, 4) и 5)в кэшах команд первого уровня обоих процессоров, 6) и 7) в кэшах данных первого уровня обоих процессоров, 8) и 9)в буферах предвыборки обоих процессоров). При записи (в кэш) результата операции может оказаться, что содержимое кэша и соответствующего элемента ОЗУ становится различным (нарушение когерентности памяти). Другой bus-master может в такой ситуации считать недействительные (старые) данные. Поэтому такой ситуации нельзя допустить. Для обеспечения когерентности используется механизм отслеживания достоверности строк кэша и объявления содержимого этих строк недостоверными в случае нарушения когерентности – после этого данными из кэша пользоваться нельзя, и потребуется обращение к основной памяти для замены значения на обновленное.

На эффективность применения кэш-памяти в иерархической системе памяти влияет целый ряд моментов. К наиболее существенным из них можно отнести:

- емкость кэш-памяти;
- размер строки;
- способ отображения основной памяти на кэш-память;
- алгоритм замещения информации в заполненной кэш-памяти;
- алгоритм согласования содержимого основной 'и кэш-памяти;
- число уровней кэш-памяти.

Ассоциативная память

В ассоциативной памяти элементы выбираются не по адресу, а *по содержимому*. Поясним последнее понятие более подробно:

Для памяти с адресной организацией было введено понятие *минимальной адресуемой единицы* (МАЕ), как порции данных, имеющей индивидуальный адрес. Введем аналогичное понятие для ассоциативной памяти, и будем эту минимальную единицу хранения в ассоциативной памяти называть *строкой ассоциативной памяти (СтрАП)*.

Каждая СтрАП содержит два поля: поле *тега* (англ. **tag** – ярлык, этикетка, признак) и поле *данных*. Запрос на чтение к ассоциативной памяти словами можно выразить следующим образом*: выбрать строку (строки), у которой (у которых) тег равен заданному значению*.

Особо отметим, что при таком запросе возможен один из трех результатов:

1) имеется в точности одна строка с заданным тегом
2) имеется несколько строк с заданным тегом
3) нет ни одной строки с заданным тегом

Поиск записи по признаку – это действие, типичное для обращений к базам данных, и поиск в базе это очень часто ассоциативный поиск. Для выполнения такого поиска следует просмотреть все записи и сравнить заданный тег с тегом каждой записи. Это можно сделать и при использовании для хранения записей обычной адресуемой памяти (и понятно, что это потребует достаточно много времени – пропорционально количеству хранимых записей !). **Об ассоциативной памяти говорят тогда, когда ассоциативная выборка данных из памяти поддержана аппаратно**.

При записи в ассоциативную память элемент данных помещается в СтрАП вместе с присущим этому элементу тегом. Для этого можно использовать любую свободную СтрАП. Рассмотрим разновидности структурной организации кэш-памяти или способы отображения оперативной памяти на кэш.

Полностью ассоциативный кэш

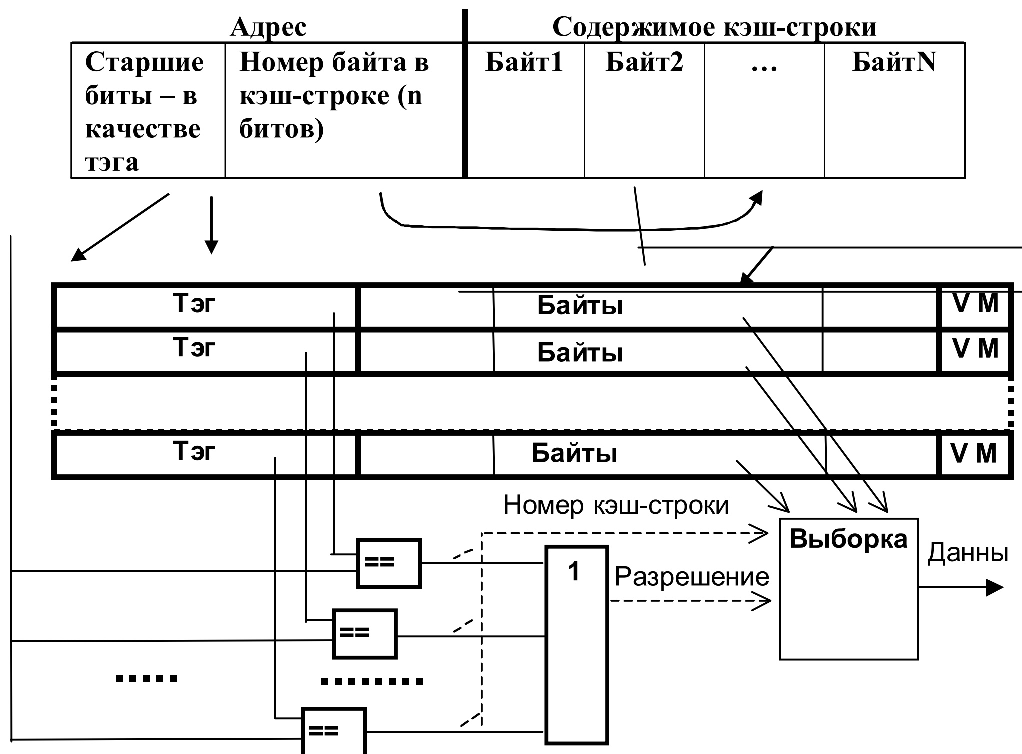


Рис. 4.16. Схема полностью ассоциативной КЭШ-памяти.

Опишем алгоритм работы системы с кэш памятью. В начале работы кэш память пуста. При выполнении первой же команды во время выборки ее код, а также еще несколько соседних байтов программного кода будут перенесены (медленно) в одну из строк кэша, и одновременно старшая часть адреса будет записана в соответствующий тег. Так происходит заполнение кэш-строки.

Если следующие выборки возможны из этого участка, они будут сделаны уже из кэша (быстро) «кэш-попадание». Если же окажется, что нужного элемента в кэше нет «кэш-промахом», то обращение происходит к ОЗУ (медленно), при этом одновременно заполняется очередная кэш-строка.

Обращение к кэшу происходит следующим образом. После формирования исполнительного адреса, его старшие биты, образующие тег, аппаратно (быстро) и одновременно сравниваются с тегами всех кэш-строк. При этом возможны только две ситуации из трех, перечисленных ранее: либо все сравнения дадут отрицательный результат (кэш-промах), либо

положительный результат сравнения будет зафиксирован в точности для одной строки (кэш-попадание).

При считывании, если зафиксировано кэш-попадание, младшие разряды адреса определяют, позицию в кэш-строке, начиная с которой следует выбирать байты, а тип операции определяет количество байтов. Очевидно, что если длина элемента данных превышает один байт, то возможны ситуации, когда этот элемент (частями) расположен в двух (или более) разных кэш-строках, и время на выборку такого элемента увеличится. Противодействовать этому можно, выравнивая операнды и команды по границам кэш-строк, что и учитывают при разработке оптимизирующих трансляторов или при ручной оптимизации кода.

Если произошел кэш-промах, а в кэше нет свободных строк, необходимо заменить одну строку кэша на другую строку.

Основная цель стратегии замещения - это удерживать в кэш-памяти строки, к которым наиболее вероятны обращения в ближайшем будущем, и заменять строки, доступ к которым произойдет в более отдаленном времени или вообще не случится. Очевидно, что оптимальным будет алгоритм, который замещает ту строку, обращение к которой в будущем произойдет позже, чем к любой другой строке кэш, К сожалению, такое предсказание практически нереализуемо, и приходится привлекать алгоритмы, уступающие оптимальному. Вне зависимости от используемого алгоритма замещения для достижения высокой скорости он должен быть реализован аппаратными средствами.

Среди множества возможных алгоритмов замещения наиболее распространенными являются четыре, рассматриваемые в порядке уменьшения их относительной эффективности. Любой из них может быть применен в полностью ассоциативном кэш.

Наиболее эффективным является алгоритм замещения на основе наиболее давнего использования **(LRU - Least Recently Used),** при котором

замещается та строка кэш-памяти, к которой дольше всего не было обращения. Проводившиеся исследования показали, что алгоритм LRU, который «смотрит» назад, работает достаточно хорошо в сравнении с оптимальным алгоритмом, «смотрящим» вперед.

Наиболее известны два способа аппаратурной реализации этого алгоритма. В первом из них с каждой строкой кэш-памяти ассоциируют счетчик. К содержимому всех счетчиков через определенные интервалы времени добавляется единица. При обращении к строке ее счетчик обнуляется. Таким образом, наибольшее число будет в счетчике той строки, к которой дольше всего не было обращений и эта строка - первый кандидат на замещение. Второй способ реализуется с помощью очереди, куда в порядке заполнения строк кэш-памяти заносятся ссылки на эти строки. При каждом обращении к строке ссылка на нее перемещается в конец очереди. В итоге первой в очереди каждый раз оказывается ссылка на строку, к которой дольше всего не было обращений. Именно эта строка прежде всего и заменяется.

Другой возможный алгоритм замещения - алгоритм, работающий по принципу «первый вошел, первый вышел» (**FIFO - First In First Out**). Здесь заменяется строка, дольше всего находившаяся в кэш-памяти. Алгоритм легко реализуется с помощью рассмотренной ранее очереди, с той лишь разницей, что после обращения к строке положение соответствующей ссылки в очереди не меняется.

Еще один алгоритм - замена наименее часто использовавшейся строки (**LFU - Least Frequently Used**). Заменяется та строка в кэш-памяти, к которой было меньше всего обращений. Принцип можно воплотить на практике, связав каждую строку со счетчиком обращений, к содержимому которого после каждого обращения добавляется единица. Главным претендентом на замещение является строка, счетчик которой содержит наименьшее число.

Простейший алгоритм - **произвольный выбор строки для замены**. Замещаемая строка выбирается случайным образом. Реализовано это может быть, например, с помощью счетчика, содержимое которого увеличивается на единицу с каждым тактовым импульсом, вне зависимости от того, имело место попадание или промах. Значение в счетчике определяет заменяемую строку.

Кроме тега и байтов данных в кэш-строке могут содержаться дополнительные служебные поля, среди которых в первую очередь следует отметить бит достоверности V (от valid – действительный имеющий силу) и бит модификации M (от modify – изменять, модифицировать). При заполнении очередной кэш-строки V устанавливается в состояние «достоверно», а M в состояние «не модифицировано». В случае, если в ходе выполнения программы содержимое данной строки было изменено, переключается бит M, сигнализируя о том, что при замене данной строки ее содержимое следует переписать в ОЗУ. Если по каким-либо причинам произошло изменение копии элемента данной строки, хранимого в другом месте (например в ОЗУ), переключается бит V. При обращении к такой строке будет зафиксирован кэш-промах (несмотря на то, что тег совпадает), и обращение произойдет к основному ОЗУ. Кроме того, служебное поле может содержать биты, поддерживающие алгоритм LRU.

Оценка объема оборудования

Типовой объем кэш-памяти в современной системе – 8…1024 кбайт, а длина кэш-строки 4…32 байт. Дальнейшая оценка делается для значений объема кэша 256 кбайт и длины строки 32 байт, что характерно для систем с процессорами Pentium и PentiumPro. Длина тега при этом равна 27 бит, а количество строк в кэше составит 256К/32=8192. Именно столько цифровых компараторов 27 битных кодов потребуется для реализации вышеописанной структуры. Приблизительная оценка затрат оборудования для построения цифрового компаратора дает значение 10 транз/бит, а общее количество

транзисторов **только** в блоке компараторов будет равно 10∗27∗8192= 2 211 840, что приблизительно в полтора раза меньше **общего** количества транзисторов на кристалле Pentium. Таким образом, ясно, что описанная структура полностью ассоциативной кэш-памяти реализуема только при малом количестве строк в кэше, т.е. при малом объеме кэша (практически не более 32…64 строк). Кэш большего объема строят по другой структуре.

КЭШ с прямым отображением памяти

Кэш-памяти с прямым отображением требует минимального объема оборудования. При этом всю основную память можно представить в виде двухмерного массива блоков (кэш строк), в котором количество рядов равно числу строк в кэш-памяти, а в каждом ряду последовательно находятся блоки, переадресуемые на одну и ту же строку кэш-памяти. В приведенном примере количество строк соответствует разрядности индекса, а количество блоков в строке – разрядности тега. Общий объем кэш-памяти составляет 64К байт.

Адрес		
Старшие биты – в качестве тега A16-31	Индекс A2-15	Номер байта в кэш-строке A0-1

Рис. 4.17. Схема КЭШ памяти с прямым отображением.

Логика работы кэш сводится к следующему. При поступления адреса анализируется поле «индекс». Оно указывает на одну из 16К строк, а именно на ту, где могут быть данные (например при чтении). Затем сравниваются

старшие 16 бит адреса с тегом, который хранится в строке. При совпадении – кэш попадание, иначе – кэш промах. В этом случае требуется всего лишь один многоразрядный компаратор, на вход которого подается тег с той единственной кэш-строки, которая оказывается выбранной полем «индекс». Именно так была устроена одна из первых реализаций кэша - накристальная кэш-память в микропроцессоре фирмы Motorola MC68020: 32 строки по 8 байтов образуют 256-байтовый кэш с прямым отображением.

Кэш ассоциативный по множеству (set-associative)

Синонимы: наборно-ассоциативный, множественно-ассоциативный.

Множественно-ассоциативное отображение относится к группе методов частично-ассоциативного отображения. Оно сочетает в себе достоинства предыдущих двух методов. Кэш включает несколько страниц кэш памяти с прямым отображением (см. рис. 4.18). Строки с одинаковым индексом образуют набор. При такой организации элемент данных с заданным адресом можно поместить не в любую кэш-строку, а только в строку, принадлежащую набору строк, выбираемых частью адреса, обозначенной на рисунке как «индекс». По количеству строк в каждом наборе говорят о «двухвходовой, четырехвходовой и т.д..» наборно-ассоциативной кэш-памяти. Еще раз подчеркнем, что за уменьшение количества компараторов кодов в наборно-ассоциативной памяти приходится платить усложнением входной схемотехники компараторов и увеличением относительной частоты кэш-промахов.

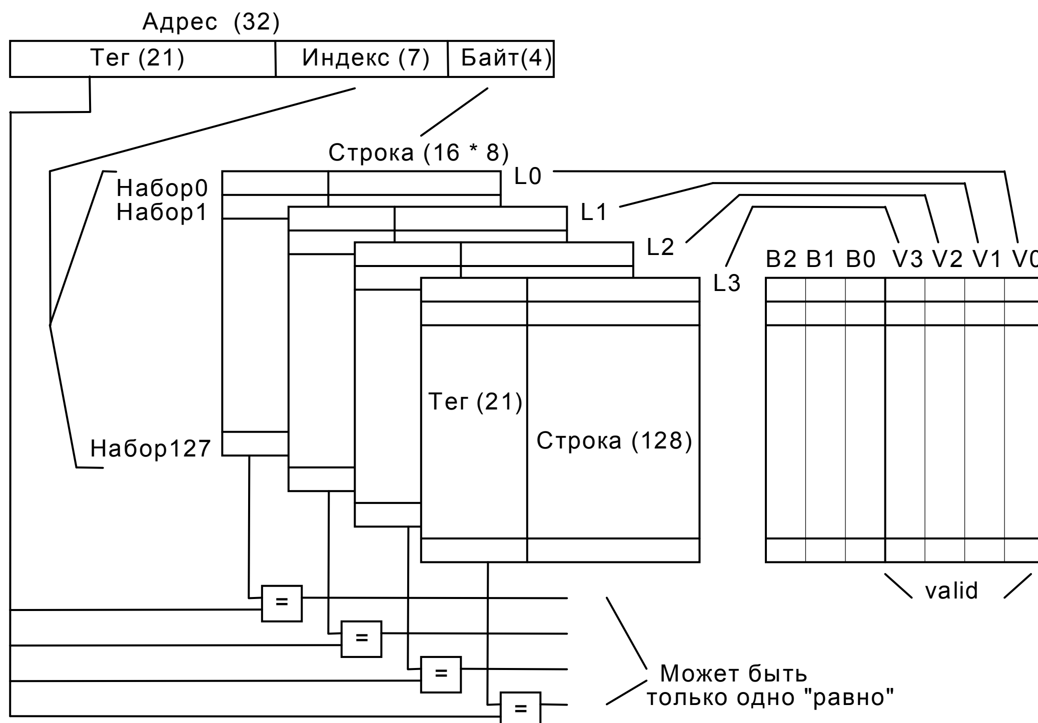


Рис. 4.18. Схема КЭШ памяти ассоциативной по множеству.

Алгоритмы обеспечения когерентности системы памяти

Проблема когерентности состоит в том, что при наличии в системе кэш-памяти одному исполнительному адресу соответствует более чем одно место хранения данных. Проблема возникает тогда, когда имеется более чем одно устройство, способное модифицировать содержимое памяти. В этом случае одно устройство может изменить значение элемента (например контроллер ПДП записывает блок данных в основную память), в то время как другое устройство (процессор), работая с другим экземпляром того же элемента данных, находящимся в кэше, будет использовать старое (неизмененное) значение. Понятно, что такие ситуации недопустимы.

Рассмотрим, что следует сделать, если когерентность нарушается при модификации кэш-строки. В этом случае требуется:

а) записать (может быть с некоторой задержкой) измененное значение в основную память

б) сбросить признаки достоверности в прочих копиях (кэш-памяти других процессоров системы, если таковые имеются).

Для записи модифицированного данным процессором значения из его кэша в основную память используются две стратегии: *сквозная запись* (write-through) и *обратная запись* (write-back).

При **сквозной записи** перенос модифицируемого значения из кэша в ОЗУ происходит одновременно с изменением в кэше. Достоинство этой стратегии состоит в автоматическом обеспечении когерентности данного кэша и ОЗУ, а недостаток – в заметном снижении скорости (каждая запись безусловно идет в ОЗУ, даже если следующая команда снова будет модифицировать тот же элемент данных). На время обращения к ОЗУ дальнейшее выполнение программы приостанавливается.

Существует модификация этой стратегии, называемая **буферизованной сквозной записью**, при которой копирование модифицированной кэш-строки в ОЗУ происходит не сразу, а через промежуточный буфер, работающий по схеме FIFO. Копирование в этот буфер происходит так же быстро, как и в кэш, при этом в многопроцессорной системе сразу же происходит сброс битов достоверности в кэш-памяти других процессоров. Перенос содержимого буфера в ОЗУ осуществляется затем параллельно с продолжением выполнения программы в те промежутки, когда данный процессор освобождает шину, связывающую его с основной памятью. В этом случае также возможны задержки (например, если следующее обращение к кэшу тоже вызовет кэш-промах), однако это будет происходить реже, чем в отсутствии буфера. Существуют определенные правила работа с указанным буфером. В частности, если буфер занят и идет чтение ячейки ОЗУ, то чтение происходит в обход буфера (изменяется последовательность обращения к памяти).

При использовании стратегии **обратной записи** перенос модифицированной кэш-строки в ОЗУ происходит только при ее замене, что в среднем случается реже, чем обращения к ней (в кэш). Как следствие, эта стратегия обеспечивает меньшую по сравнению со сквозной записью, долю

обращений к ОЗУ, т.е. более высокую эффективность кэширования. Недостатки обратной записи: а) более сложная аппаратная реализация, б) сложнее обеспечить когерентность, в) при замене модифицированной строки возникает задержка, обусловленная необходимостью обращения к ОЗУ. Последний недостаток можно частично устранить, используя **упреждающую обратную запись**: строка-кандидат на замену определяется заранее, переносится из кэша в ОЗУ в интервалах, когда шина свободна и затем объявляется в данном кэше недостоверной. В результате в кэш-памяти почти всегда имеется свободная строка, которую можно без задержки использовать при кэш-промахе.

Для обеспечения когерентности в сложных системах с несколькими кэшами разработан и стандартизован протокол обеспечения когерентности, называемый **MESI** по названиям состояний, в которых может находиться кэш-строка: **M**odified, **E**xclusive, **S**hared, **I**nvalid.

Invalid – Это состояние означает, что строка недостоверна в данном кэше. При обращении к этому элементу будет зафиксирован кэш-промах и произойдет внешнее обращение в память следующего уровня (в кэш L2 или в основную память)

Shared – строка в кэше содержит достоверные данные и совпадает с содержимым основной памяти. Обращение будет внутренним (в данный кэш) и не приведет к активизации внешней шины.

Exclusive – строка достоверна, находится в кэше, совпадает с содержимым основной памяти, а экземпляры этой строки, находящиеся в других кэшах, являются недостоверными.

Modified – Строка находится в кэше, достоверна, но содержит более поздний (модифицированный) экземпляр данных, нежели остальные экземпляры в основной памяти и в других кэшах.

Замечание 1: строка достоверна, если она находится в состояниях S, E или M.

Замечание 2: если строка находится в состоянии I, это означает, что она свободна.

Собственно протокол состоит из набора правил, по которым меняется состояние кэш-строки в результате возникновения разных событий, могущих быть причинами нарушения когерентности. В основе этих правил лежит принцип объявления строки недостоверной в случае подозрения в нарушении когерентности, что при обращении к этой строке вызовет кэш-промах и обновление содержимого этой строки новым значением из основной памяти. Таблица 3.1 и рисунок 3.1, взятые из технического описания процессора Pentium (Pentium ® Processor Family Developer's Manual, Volume 2, стр.94, 95), показывают эти правила.

Table 3-1. Basic MESI State Transitions

Current State	Action	New State	Memory Bus Activity
M	Read	M	None
	Write	M	None
	Snoop	E,S,I	Writeback
E	Read	E	None
	Write	M	None
	Snoop	E,S,I	None
S	Read	S	None
	Write	M,E,S	Writethrough
	Snoop	S, I	None
I	Read	M,E,S,I	Linefill
	Write	M,E,S,I	Writethrough
	Snoop	I	None

Figure 3-1. State Changes

Рис. 4.19. Схема изменения MESI состояний.

4.8. Многоуровневая кэш-память и пакетный режим передачи данных

Современные технологии позволяют разместить кэш-память и ЦП на общем кристалле. Такая внутренняя кэш-память строится по технологии статического ОЗУ и является наиболее быстродействующей. Емкость ее обычно не превышает 64 Кбайт. Попытки увеличения емкости обычно при водят к снижению быстродействия, главным образом из-за усложнения схем управления и дешифрации адреса. Общую емкость кэш-памяти ЭВМ увеличивают за счет второй (внешней) кэш-памяти, расположенной между внутренней кэш-памятью и ОЗУ. Такая система известна под названием двухуровневой, где внутренней кэш-памяти отводится роль первого уровня (L1), а внешней - второго уровня (L2). Емкость L2 может быть значительной (до 1 МБ).

При доступе к памяти ЦП сначала обращается к кэш-памяти первого уровня. В случае промаха производится обращение к кэш-памяти второго уровня. Если информация отсутствует и в L2, выполняется обращение к ОЗУ и соответствующий блок заносится сначала в L2, а затем и в L1. Благодаря такой процедуре часто запрашиваемая информация может быть быстро восстановлена из кэш-памяти второго уровня. Для ускорения обмена информацией между ЦП и L2 между ними часто вводят специальную шину, так называемую шину заднего плана, в отличие от шины переднею плана, связывающей ЦП с основной памятью.

Количество уровней кэш-памяти не ограничивается двумя. В некоторых ЭВМ можно встретить кэш-память третьего уровня (L3) и ведутся активные дискуссии о введении также и кэш-памяти четвертого уровня (L4). Характер взаимодействия очередного уровня с предшествующим аналогичен описанному для L1 и L2. Таким образом, можно говорить об иерархии кэш-памяти. Каждый последующий уровень характеризуется большей емкостью, меньшей стоимостью, но и меньшим быстродействием, хотя оно все же

выше, чем у ЗУ основной памяти.

Большинство современных процессоров имеют разделенную кэш первого уровня на кэш команд и кэш данных (кэш гарвардской архитектуры). Такое разделение (рис. 4.20) позволяет повысить эффективность работы кэша по следующим соображениям:

- Многие процессоры имеют конвейерную архитектуру, при которой блоки конвейера работают параллельно. Выборка команды и доступ к данным осуществляется на разных этапах конвейера. Использование раздельной кэш-памяти позволяет выполнять эти операции параллельно.
- Кэш команд может быть реализован только для чтения, следовательно, не требует реализации никаких алгоритмов обратной записи, что делает этот кэш проще, дешевле и быстрее.

Согласно результатам исследований часто используемых целочисленных задач, у Intel Pentium III 16 Кбайт четырехканального D-cache было достаточно для покрытия около 93% запросов, а 16-Кбайт четырехканального I-cache - 99% запросов.

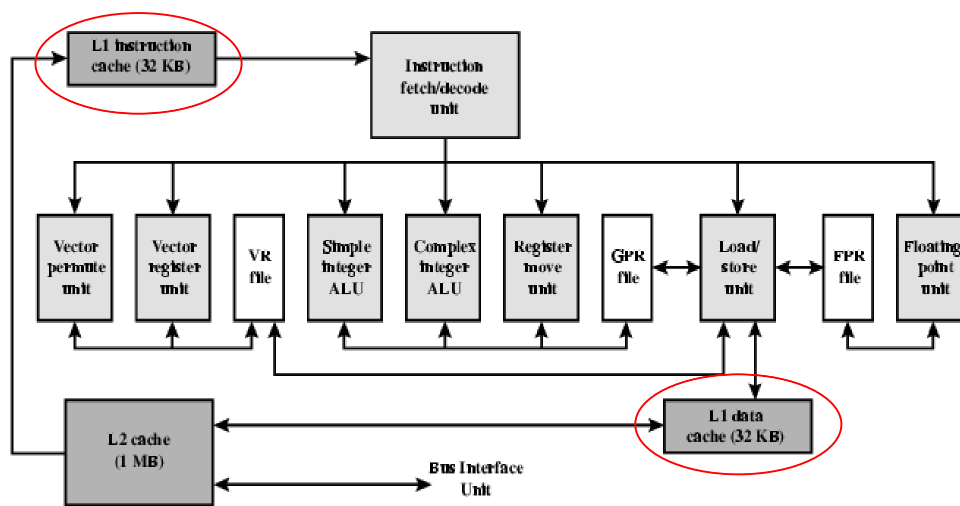


Рис. 4.20. Схема вычислительного ядра процессора PowerPC G4.

Пакетный режим передачи данных

Режим *пакетной передачи* (Burst Mode) предназначен для быстрых операций со строками КЭШа. Строка КЭШа процессора, например, имеет длину 16 байт, следовательно, для ее пересылки требуется четыре 32-

разрядных шинных цикла. В этом режиме адрес и сигналы идентификации типа шинного цикла выдаются только в первом такте пакета. В каждом из последующих тактов могут передаваться данные, адрес для которых уже не передается по шине, а вычисляется из первого по правилам, известным и процессору, и внешнему устройству. В пакетный цикл процессор может преобразовать любой внутренний запрос на множественную передачу.

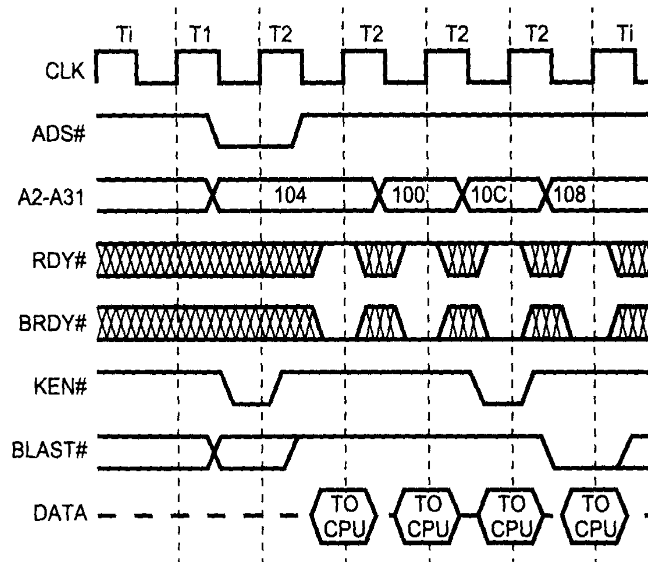


Рис. 4.21. Временная диаграмма пакетного режима загрузки КЭШ памяти

Пакетный цикл (рис. 4.21) начинается процессором так же, как и обычный: на внешней шине устанавливается адрес, сигналы идентификации типа цикла и формируется строб ADS#. В следующем такте передается первая порция данных, и, если она не единственная, сигнал BLAST# имеет пассивное значение. Если адресованное устройство поддерживает пакетный режим, оно должно ответить сигналом BRDY# вместо сигнала RDY# по готовности данных в первой же передаче данных цикла. В этом случае процессор продолжит цикл как пакетный, не вводя такта адресации-идентификации (с сигналом ADS#), а сразу перейдет к передаче следующей порции данных. Нормально о завершении пакетного цикла процессор сообщает устройству сигналом BLAST#, который выдается в такте последней передачи пакета. Если у процессора есть намерения собрать пакет, а устройство отвечает сигналом RDY#, данные будут передаваться обычными

циклами. Введением сигнала RDY# вместо BRDY# внешнее устройство может в любой момент прервать пакетную передачу, и процессор ее продолжит обычными циклами. В идеальном варианте (без тактов ожидания) для передачи 16 байт в пакетном режиме требуется всего 5 тактов шины вместо восьми, которые потребовались бы при обычном режиме обмена

Пакетный режим предполагает соблюдение одних и тех же правил формирования последующих адресов, как процессором, так и внешним устройством (памятью). Во время пакетного цикла процессора старшие биты адреса A[3l:4] остаются неизменными (как и сигналы идентификации M/IO#, D/C#, W/R#. Изменяться могут только биты A[3:2] и сигналы BE[3:O] (у процессоров с 64-битной шиной данных неизменны A[3l:5], меняются только A[4:3] и BE[7:0]). Таким образом, один пакетный цикл не может пересекать границу строки КЭШа. Кроме того, имеется специфический порядок следования адресов в пакетном цикле, который определяется начальным адресом пакета (задается процессором) и разрядностью передач.

ГЛАВА 5. ПАМЯТЬ. ВЕРХНИЙ УРОВЕНЬ

Современный компьютер состоит из ряда уровней, каждый из которых добавляет дополнительные функции к уровню, который находится под ним. Рассмотрим верхний уровень организации памяти, который связан с уровнем операционной систмы (уровень 3 в многоуровневой организации ВТ). На этом уровне могут появлятся новые команды и механизмы, которые расширяют возможности ВТ.

5.1. Динамическое распределение памяти

Распределение памяти в вычислительной системе предполагает ее выделение для разных видов команд и данных с целью обеспечить нужный объем и требуемое быстродействие памяти. Такое выделение проводится статически (команды и директивы транслятора) или динамически. Статическое выделение связано с резервированием памяти например под массивы. Если эта память не используется, то она всеравно занята и недоступна программам. Динамическое выделение памяти предполагает, что память занята только на короткое время, а затем она освобождается. Выделим три уровня динамического распределения памяти: стек, виртуальная память, команды операционной системы.

Стек. Все переменные и массивы объединяются в заголовке функции. Они хранятся в стеке и освобождают память только при окончании выполнения функции. Во время выполнения функции стек может только увеличиваться, например при выполнении рекурсивных обращений к другим функциям. Возможно неконтролируемое увеличение стека.

Виртуальная память – это механизм распределения памяти по страницам и загрузки этих страниц из внешней памяти. Он является основным методом динамического распределения памяти в ЭВМ и будет рассмотрен ниже. Механизм виртуальной памяти требует аппаратной и программной поддержки. В ряде случаев этот механизм неэффективен или им сложно воспользоваться. Наприме, в ситуации работы с большими

кадрами изображений при ограниченной памяти или при отсутствии аппаратной поддержки.

Команды операционной системы. Вводятся две функции: выделение памяти и освобождения памяти. Функция, выделяющая память, резервирует необходимый объем памяти с заданными свойствами, а функция освобождения памяти освобождает память и она может быть использована снова при выделении.

При запросе на выделение памяти указывается требуемый размер участка. Для поиска нужного участка необходим список (таблица) свободных участков с указанием их размеров. Обычно на уровне операционной системы поддерживается таблица блоков, на которые разбивается память с указанием занятости блока. Эта таблица упорядочена по возрастанию объемов блоков с целью быстрого поиска нужного объема. Если выделенный объем больше требуемого, то часть памяти не используется – эта ситуация называется *фрагментация памяти*.

5.2. Виртуальная память

Технология виртуализации касается разных сторон вычислительной системы. Это разделение сервера на несколько виртуальных машин, каждая из которых способна выполнять собственную операционную систему и прикладную среду. Это виртуализация внешних устройств. Наконец это виртуализации памяти.

Виртуализация памяти (виртуальная память) – метод автоматического управления иерархической памятью, при котором программисту кажется, что он имеет дело с единой памятью большой емкости и высокого быстродействия. Нижний слой памяти может включать память на магнитных дисках, сеть и другие элементы.

Идея виртуальной памяти возникла из желания выполнять программы большего размера чем ОЗУ. Вначале стали использовать **оверлейную структуру** программ, когда большая программа разбивается на отдельные

модули перекрытия (оверлеи overlays). Первый модуль загружается в память и работает. Если ему нужен код или данные, содержащиеся в других модулях, то работающий модуль вызывает функции, загружающие другие модули. При этом программист сам должен отследить: 1) вызов функций загрузки, 2) передачу управления вновь загруженным модулям, 3) контроль суммарного объема загруженных модулей (объем не должен превышать имеющегося объема памяти), 4) размещение модулей в памяти, и другие подобные вопросы. Эта работа оказывается весьма трудоемкой.

При этом отметим, что поскольку вновь загружаемые модули "ложатся" в память на место находившихся там ранее, то 1) разные объекты программы, находящиеся в разных модулях, но попадающие в одно и то же место физической памяти получают одни и те же физические адреса, 2) расходуется время на перезагрузку модулей 3) фрагментируется память, поскольку модули имеют разные размеры, 4) для работы в другой конфигурации памяти требуется перекомпоновка программы.

Как альтернатива оверлейной структуре, возникла концепция **виртуальной памяти** (*1961 г, Манчестер*). Фактически используется та же идеология перезагрузки модулей (свопинг swapping), однако замена осуществляется аппаратурой и операционной системой автоматически, без какого-либо участия программиста и незаметно для него (рис.5.1.).

При этом программист оперирует адресами в (линейном) **виртуальном адресном пространстве задачи**, размер которого не меньше размера программы (включающей как исполняемый код так и данные). Каждый объект программы в пространстве виртуальных адресов имеет уникальный адрес. Программа пишется в виртуальных адресах.

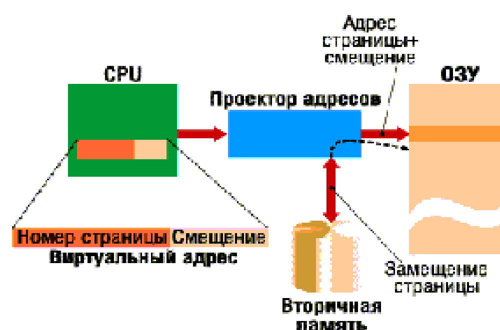


Рис. 5.1. Схема организации виртуальной памяти.

Принципы организации виртуальной памяти

Пространство *линейных (виртуальных)* адресов программы и диапазон физических адресов реально существующей памяти разбиваются на страницы (блоки, сегменты). Старшие адреса виртуального адреса определяют номер страницы, а младшие смещение на странице (см. рис.5.1). Схема трансляции (проектор адресов), основанная на таблице страниц, обеспечивает загрузку новых и удаление старых страниц.

Термин *линейный адрес* подразумевает, что все адреса, используемые программой, расположены в едином одномерном (линейном) адресном пространстве, и каждый программный объект (элемент данных или команда программного кода) имеет свой уникальный адрес в этом пространстве. В противоположность этому, при использовании оверлейной структуры, один и тот же адрес может соответствовать нескольким объектам из разных модулей (оверлеев), загружаемых в одну область памяти в разные интервалы времени.

Рассмотрим организацию таблицы страниц и механизма замещения. Блоки памяти в рассматриваемом примере имеют одинаковую длину, но это не обязательно. Простейший вариант: размеры блоков (страниц) одинаковы, равны $2^К$ и выровнены по $2^К$ границам адресов (см. рис.5.2.). Пусть физическая память имеет ограниченный размер 2^{k+3} и разбита на 8 страниц, а выполняемая программа содержит N страниц размером $\leq 2^k$. Для установления соответствия операционная система формирует в системной

области памяти таблицу страниц, которая содержит информацию о всех блоках выполняемой программы.

Внешнее ЗУ (диск)

ОЗУ (системная область)

Таблица сегментов (страниц)

Виртуальные адреса

A0 Стр 0

A1 Стр 1

An

AN СтрN

БП	Блок физ. памяти	Место на диске	БИ	......
1	110		1	
1	101		0	
0	xxx			

(эти страницы в ОЗУ)

(этой страницы в ОЗУ нет)

ОЗУ (физ.память)

000 001 010 011 100 101 110 111

Линейный (виртуальный) адрес

Адрес внутри страницы

101

Физический адрес

Рис. 5.2. Схема виртуальной памяти на основе страниц.

Запись для каждого блока содержит обычно следующие поля:

1) **Место на внешнем носителе**, где расположен блок. Туда ОС обращается, когда требуется загрузить данный блок в физическую память. При этой загрузке ОС должна выбрать свободную страницу физической памяти, загрузить туда страницу с диска, а затем заполнить в строке таблицы страниц два поля:

2) записать **номер стсраницы физической памяти**, куда записана страница, а также установить

3) **"бит присутствия" БП**, который говорит о том, что данная страница находится в ОЗУ , в соответствующей строке таблицы автоматически устанавливается

4)"**бит изменения" БИ** - этот бит устанавливается, если при работе программы в данную страницу происходит запись, т.е. страница подвергается

модификации**.** Этот бит будет использован, если данная страница удаляется из ОЗУ, поскольку на его место требуется поместить в память новую страницу. Старую страницу требуется сохранять на диске только в том случае, если бит изменения был установлен, (т.е. страница была изменена).

Обычно для записи информации о местоположении страницы на диске (номер блока/сектора) используется специальная таблица – карта диска.

Рассмотрим более подробно, как работает виртуальная память. Линейный (виртуальный) адрес рассматривается состоящим из двух частей: номера страницы и адреса внутри страницы. По номеру страницы выбирается строка таблицы страниц. Если бит присутствия установлен в “1”, то страница в ОЗУ есть. Виртуальный номер страницы замещается реальным номером страницы в ОЗУ, взятым из строки таблицы. Происходит обращение к странице ОЗУ с учетом смещения. Если бит присутствия установлен в “0”, то это значит данной страницы в ОЗУ нет. Происходит прерывание и загрузка нужной страницы с установкой бита присутствия в “1”. Затем формируется реальный адрес и считывается/записывается операнд. Загрузку страниц определяют три группы правил:

Правила выборки страниц (fetch policy).

Методы выборки определяют правила выбора загружаемой в память страницы. Известны три правила:

1. *Подкачка по требованию*. Очередная страница в физическую память загружается только при страничной ошибке – только при отсутствии нужной страницы. При начальной загрузке программы или массива вызывается множество страничных ошибок, а затем интенсивность потока страничных ошибок снижается.
2. *Подкачка по требованию с кластеризацией.* При страничной ошибке загружается не только страница, вызвавшая ошибку, но и нескольео последующих страниц. Эта стратегия обеспечивает минимизацию

числа операций ввода-вывода, связанных с подкачкой. Размер кластера зависит от объема цизической памяти и составляет 2-8 страниц.

3. *Упреждающая подкачка.* Страница в ОЗУ загружается еще до того момента, когда она потребуется, т.е. по предположению. Эта стратегия обеспечивает максимальное быстродействие в системе. Однако, трудно определить какая страница понадобиться в будующем и возможны случаи, когда загружается не нужная страница.

Правила размещения (placement policy). Определяют местоположения подкачиваемой страницы в памяти с учетом размеров кэш-ей и стрмления свести нагрузку на них к минимому. В мультипроцессорных ситемах правила размещения страниц по узлам системы определяются программным обеспечением.

Правила замены (replacement policy).

Политика замещения определяет какую страницу можно убрать из памяти при вызове новой страницы. Выбирать такую страницу наугад нельзя. Если, например, убрать из памяти страницу с командой, которая вызвала ошибку, то при повторном выполнении команды или при выполнении следующей команды произойдет еще одна ошибка из-за отсутствия страницы. Необходимо удалять из памяти те страницы, которые не будут нужны долгое время. Однако будующее использование страниц предсказать не возможно, поэтому используют алгоритмы основываясь на истории обращиний к страницам. Эти алгоритмы очень похожи на метода замещения кэш-строк.

LRU (Least Recently Used) Замещается та страница, к которой дольше всего не было обращения.

FIFO (First In First Out). Здесь заменяется страница, дольше всего находившаяся в памяти.

Правила замены могут быть **глобальными или локальными**. При глобальной замене выгружается самая старая страница всех процессов, а при

локальной – только текущего, при раьоте которого произошла страничная ошибка.

За счет виртуального механизма решаются вопросы:

1) Выполнение программы, объем которой превышает объем физической памяти.
2) Независимость работоспособности программы от места ее расположения в физической памяти и от конфигурации этой памяти.
3) Динамическое распределение памяти.
4) Независимость (легкая сопрягаемость) разрядности компонентов логических адресов в программе (указателей), разрядности линейного адреса, разрядности физического адреса и реального объема памяти, установленной в системе.

5.3. Общие принципы защиты памяти

Механизм защиты позволяет ограничить влияние неправильно работающей программы на другие выполняемые программы и их данные. Защита представляет собой ценное свойство при разработке программных продуктов, поскольку она обеспечивает сохранность в памяти при любых ситуациях средств разработки программного обеспечения (операционной системы, отладчика). При сбое в прикладной программе в полной исправности сохраняется программное обеспечение, позволяющее выдать диагностические сообщения, а отладчик имеет возможность произвести "посмертный" анализ содержимого памяти и регистров сбойной программы. В системах, эксплуатирующих готовое программное обеспечение, защита позволяет повысить его надежность и дает возможность инициировать восстановительные процедуры в системе. Можно назвать следующие цели защиты:

1. Средства защиты могут быть использованы для предотвращения взаимного влияния одновременно выполняемых задач. Например, может выполняться защита от затирания одной задачей команд и данных другой

задачи в частности операционной системы.

2. При разработке программы механизм защиты поможет получить более четкую картину программных ошибок. Когда программа выполняет неожиданную ссылку к недопустимой в данный момент области памяти, механизм защиты может блокировать данное событие и сообщить о нем.

3. В системах, предназначенных для конечных пользователей механизм защиты может предохранять систему от программных сбоев, вызываемых невыявленными ошибками в программах.

Защита в ЭВМ может быть построена на следующих принципах.

1. Расщепление адресных пространств при трансляции адресов.
2. Проверки разрешено/запрещено при работе с памятью при записи или чтении операндов.
3. Защита отдельных ячеек памяти, каждая из которых имеет специальный разряд защиты. Этот разряд проверяется при обращении к памяти.
4. Метод граничных регистров. Защищаемая область непрерывна и связана с двумя регистрами - начало и конец защищаемой области, и признак защиты (чтение, запись). При обращении адрес сравнивается со значениями регистров. Требуется по 2 регистра на каждую защищаемую область.
5. Кольца защиты. Задачи, выполняемые на ЭВМ, разделяют по уровню привилегий. Минимально используют два уровня привилегий: супервизор, пользователь. Такое разделение используется в процессоре PowerPC 603. В процессорах Pentium, Itanium механизм защиты распознает четыре уровня привилегированности, нумеруемые от 0 до 3. Чем больше номер, тем ниже уровень привилегированности. При удовлетворении всех прочих проверок защиты исключение общей защиты генерируется при попытке программы доступа к сегменту с большим уровнем привилегированности (т.е. с меньшим числом, задающим уровень), чем

имеет данный сегмент. В ЭВМ VAX-11. В системе есть 4 уровня привелегий (кольца защиты):

00 kernel режим ядра

01 executer режим управления

10 supervisor контроль

11 user пользователь

6. Метод ключей защиты. Память разбивается на блоки фиксированной длинны (страницы). Каждому блоку ставится в соответствие некий код, называемый ключ защиты памяти. Каждой программе в свою очередь присваивается код защиты программы. Условием доступа программы к блоку памяти служит совпадение ключа и кода защиты или равенство их нулю. Нулевой код защиты имеет операционная система. Память разбита на страницы. Составляется таблица страниц, которая содержит адрес страницы, ключ защиты и признак защиты. При обращении к памяти производится проверка прав доступа. Использован в IBM 360/370.

7. Задание точек входов в подпрограммы. (см. механизм шлюзования в процессорах Pentium)

5.4. Мультизадачность

Что такое мультизадачность (многозадачность) и зачем она нужна

Понятие «задача» (task) в первом приближении аналогично понятию «программа». Когда говорят о многозадачном режиме, имеют в виду возможность одновременного (параллельного) выполнения нескольких программ. Истинная параллельность возможна лишь в многопроцессорной системе, когда каждая программа выполняется на своем ресурсе.

В однопроцессорной системе несколько программ могут разделять процессорное время. При переключении с одной задачи на другую требуется сохранять контекст прерываемой задачи, чтобы можно было впоследствии продолжить ее выполнение. Это обычно делает операционная система, причем для этого совсем не обязательно наличие какой-либо специальной

аппаратной поддержки. Например в DOS 5 есть программная оболочка DosShell, содержащая в своем составе подобный переключатель задач, работающий в реальном режиме процессора 8086 и сохраняющий (достаточно медленно) образ прерываемой задачи на жестком диске.

Чем может быть полезна многозадачность

В большей части случаев достигается повышение производительности труда человека, например в следующих ситуациях:

1) Выполнение длинных рутинных операций (форматирование текста, долгая передача через модем, копирование длинных файлов и т.п.)
2) Работа одновременно с несколькими приложениями, когда выходные данные одних служат входными для других. Например, связка FineReader (сканирование и распознавание текста) + PhotoEditor (сканирование и редактирование изображений) + AcrobatReader (чтение документов в формате .pdf) --> MsWord (окончательное оформление документа, содержащего выдержки из документации, из сканированного текста и рисунки).
3) Использование процессорного времени фоновой задачей в том случае, когда текущая задача ожидает предоставления ресурса.

Многозадачность и операционная система

Большинство операционных систем (ОС) поддерживают многозадачность. Они последовательно переключают задачи одну на другую.В каждый момент времени процессор выполняет только одну задачу. В многопоточных процессорах одновременно могут выполняться несколько задач. ОС планирует какая из задач будет выполняться следующей, выбирает эту задачу и переключает контексты задач. Методы переключения зависят от стратегии, выбранной ОС. Рассмотрим методы планирования и диспетчерирования.

Различают несколько видов планирований.

Долгосрочное планирование	Решение, какую из поступивших программ следует

	поставить в очередь на обслуживание.
Среднесрочное планирование	Решение, загружать ли в ОЗУ задачу (частично или полностью). Относится к функции swapping.
Краткосрочное планирование - диспетчер	Вызывается каждый раз, когда нужно принять решение, какой задаче предоставить ресурс процессора.
Планирование ввода-вывода	Решение, какой из запросов операций вводы-вывода передать на обработку доступному устройству.

Задача (поток) в процессе жизни в ОС может находиться в одном из состояний:

- Ready (готова) – готова к выполнению;
- Standby (простаивает) – выбрана следующей для выполнения на конкретном процессоре, диспетчер переключит контекст на эту задачу;
- Running (выполняется) – задача выполняется до момента:
    - Завершения;
    - Вытеснения задачей с более высоким приоритетом;
    - Самостоятельного перехода в состояние ожидания;
    - Окончания выделенного кванта времени.
- Waiting (ожидает) – задача переходит в это состояние самостоятельно на синхронизирующем объекте или по команде ОС, по окончании ожидания задача переходит в состояние Running или Ready;
- Transition (переходное состояние) – стек задачи выгружен из памяти, при загрузке переходи в состояние Ready;
- Terminated (завершена) – заканчивается выполнение, может удаляться из памяти;
- Initialized (инициализирована) – загужена из ROM или винчестера.

Обычная последовательность прохождения задачи по состояниям может быть такой:

Initialized – Ready – Standby – Running – Terminated. Более полно схема прохождения задачи по состояниям показана на рис.5.3 [9].

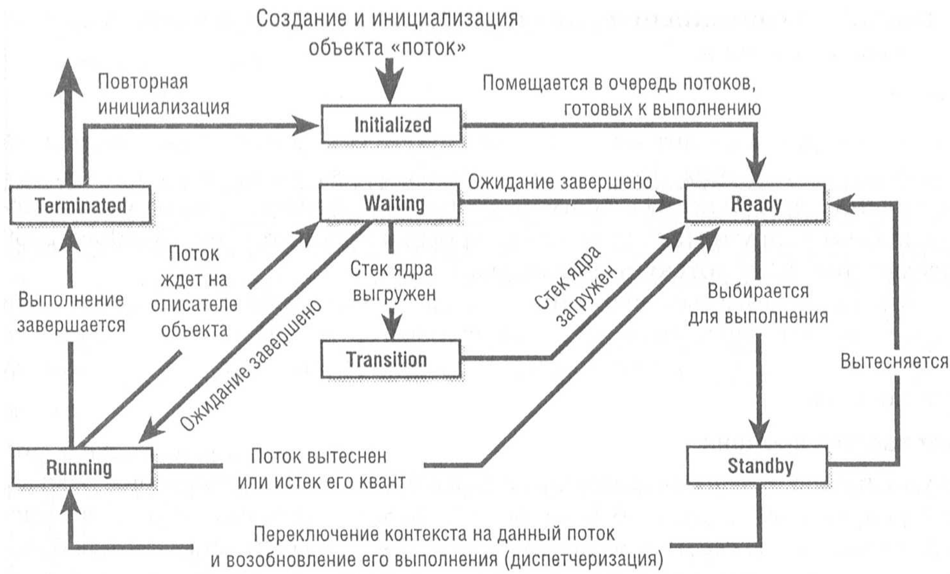
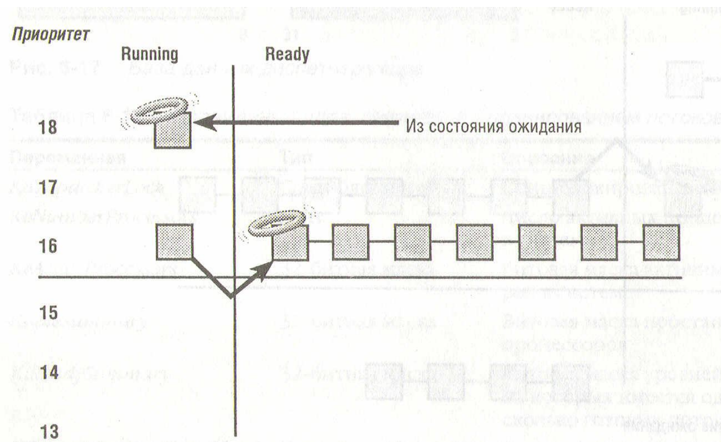


Рис. 5.3. Схема продвижения задачи по состояниям

Способы планирования.

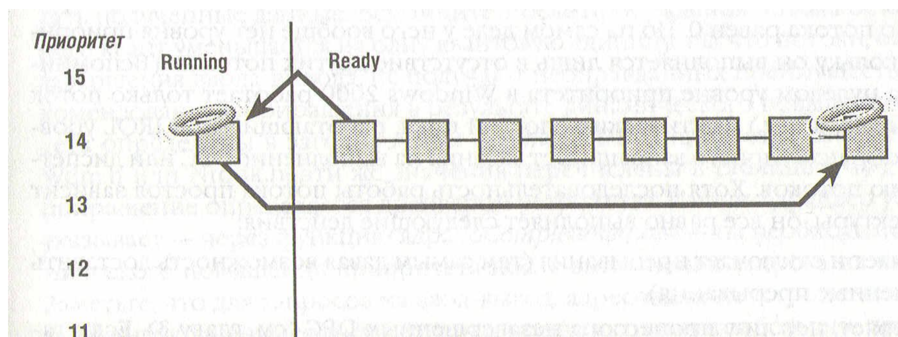
1. Невытесняемое планирование. Новая задача переходит в состояние Running, если предыдущая задача закончила свое выполнение или перешла в сотояние Waiting.

2. Посностью вытесняемое планирование. Задача вытесняется задачей с более высоким приоритетом. Така ситуация может быть в двух случаях: пришла задача большего приоритета или приоритет текущей задачи понизился. Когда задача вытесняется она встает в очередь готовых задач соответствующего уровня приоритета. На рисунке задача (поток) с приоритетом 18 выходит из состояния ожидания и захватывает процессор, вытесняя выполняемую в этот момент задачу (с приоритетом 16) в очередь готовых потоков. Заметьте, что вытесненная задача помещается не в конец, а в начало очереди. После завершения вытеснившей задачи вытесненная сможет отработать остаток своего времени.

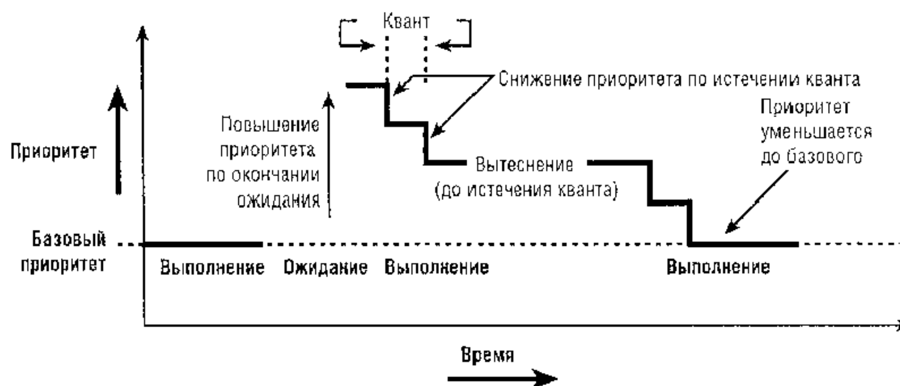


3. Круговое планирование. Используется операция уступить новой задаче. Такое планирование может применяться для задач одного приоритета.

4. Завершение кванта. Каждой задаче выделяется квант времени. По окончании выполняется другая задача с текущим уровнем приоритета, если такой нет, то второй квант времени выполняется первая задача. Вытесненная задача ставится в очередь задач данного приоритета (см. рис.) и переводится из состояния Running в состояние Ready. В Win 2000 у каждой задачи (потока) свое значение кванта. Это значение выражается не в единицах времени, а целым числом в так называемых квантовых единицах. Минимальный квант (6 единиц) по умолчанию равен двум интервалам системного таймера. Максимальный соответствует 36 единиц. Длительный квант в ряде случаев позволяет свести к минимому переключения контекста. Получая больший квант программы обслуживания клиентского запроса (в серверах) имеют больше шансов выполнить запрос и вернуться в состояние ожидания до истечения выделенного кванта. Длина временного интервала таймера зависит от аппаратной платформы и на большинстве однопроцессорных x86систем составляет 10 мс, а на бльшинстве многопроцессорных систем - 15 мс. Величиной выделенного кванта можно управлять, в частности динамически приращивая квант при круговом планировании.



5. Адаптивное планирование. Основано на динамическом изменении приоритетов задач. По истечении выделенного кванта приоритет задачи снижается. Если с текущим уровнем приоритета нет задач, задача выполняется следующий квант, и приоритет снова снижается и так далее до базового. Если задача долгое время не выполнялась, то ее приоритет возрастает до тех пор, пока она не начнет выполняться. На рис. показан этот процесс. Техника назначения приоритетов и их изменения полностью определяется уровнем ОС.



6. Планирование по расписанию. Данное планирование предполагает использование комбинации разных способов.

Распределение приоритетов.

Приоритеты (в Win 2000 32 уровня приоритетов) в ОС распределяются на статические и динамически изменяемые. Статические имеют задачи обработки прерываний. Динамические - задачи приложений. Приоритеты задач назначаются, ОС может их повысить, в процессе выполнения они снижаются. Приоритеты потоков Windows 2000 повышает в следующих случаях:

- после завершения операций ввода-вывода;
- по окончании ожидания на каком-либо событии исполнительной системы или на семафоре;
- по окончании операции ожидания потоками активного процесса;
- при пробуждении GUI-потоков из-за операций с окнами;
- если поток, готовый к выполнению, задерживается из-за нехватки процессорного времени.

Динамическое повышение приоритета предназначено для оптимизации общей пропускной способности и отзывчивости системы, а также для устранения потенциально (нечестных) сценариев планирования.

Контекст. Переключение контекста.

Контекст – полная информация о текущем состоянии задачи и процессора, которая позволяет в любой момент продолжить вычисления. Контекст зависит от архитектуры процессора, типа задачи и работы ОС. В типичном случае полный контекст включает следующие данные:

- счетчик команд,
- регистр состояния процессора (регистр флагов),
- содержимое остальных регистров процессора,
- указатели на стеки ядра ОС и пользователя,
- указатели на адресное пространство задачи (см. каталог таблиц страниц CR3 для x86),
- промежуточные результаты работы программы.

Переключение контекста позволяет загрузить в процессор из памяти новую задачу на выполнение, а информацию о старой сохранить. Объем контекста 32-разрядного процессора составляет 64 байта и более. Объем контекста RISC процессора значительно больше, так в процессоре Itanium число 64-разрядных регистров около 500. Ряд задач не требуют сохранения полного контекста, достаточно сохранить/ восстановить счетчик команд и

регистр состояния процессора. Такими задачами могут быть задачи обслуживания прерываний клавиатуры, таймера и др. В этом случае размер контекста 8 байт (два слова).

Многозадачность в результате сводится к планированию прохождения задач, диспетчерированию и переключению контекста. Аппараная поддержка возможна только операций по переключению контекста. Остальные слишком сложны и реализуются ОС.

Планирование потоков в системах с симметричной мультипроцессорной обработкой

В многопроцессорной системе диспетчер при планировании рассматривает дополнительные условия, связанные с выбором процессора для выполнения текущей задачи, при этом учитывается:

- поток уже выполнялся в прошлый раз на данном процессоре;
- данный процессор должен быть идеальным для этого потока.

Если в момент прихода задачи (переход в состояние Ready) все процессоры заняты, проверяется, нельзя ли вытеснить задачу на одном из процессоров. Какой процессор рассматривается в первую очередь? Сначала - идеальный, затем – остальные.

5.5. Дисковые массивы и уровни RAID

В 1987 г. Паттерсон, Гибсон и Катц, американские исследователи из Калифорнийского университета в Беркли, описали в своей статье A Case for Redundant Arrays of Inexpensive Disks (RAID) несколько типов дисковых массивов, обозначив их аббревиатурой RAID. Основная идея RAID состояла в объединении нескольких небольших и недорогих дисков в массив, который по производительности не уступал бы одному большому диску (Single Large Expensive Drive, SLED), использовавшемуся обычно с компьютерами типа мэйнфрейм. Заметим, что для компьютера этот массив дисков должен был выглядеть как одно логическое устройство. Увеличение количества дисков в массиве, как правило, означало повышение производительности, по крайней

мере при чтении информации. Слово "недорогой" (inexpensive) в названии RAID характеризует стоимость одного диска в массиве по сравнению с большими дисками мэйнфреймов. Кстати, некоторое время спустя после выхода вышеупомянутой статьи из Беркли пришла новая расшифровка аббревиатуры RAID - Redundant Arrays of Independent Disks. Дело в том, что из-за низкой надежности недорогих дисков в массивах первоначально пришлось использовать достаточно дорогие дисковые устройства мэйнфреймов.

Одним из способов повышения производительности ввода/вывода является использование параллелизма путем объединения нескольких физических дисков в матрицу (группу) с организацией их работы аналогично одному логическому диску. К сожалению, надежность матрицы любых устройств падает при увеличении числа устройств. Полагая интенсивность отказов постоянной, т.е. при экспоненциальном законе распределения наработки на отказ, а также при условии, что отказы независимы, получим, что среднее время безотказной работы (mean time to failure - MTTF) матрицы дисков будет равно:

MTTF одного диска / Число дисков в матрице

Для достижения повышенного уровня отказоустойчивости приходится жертвовать пропускной способностью ввода/вывода или емкостью памяти. Необходимо использовать дополнительные диски, содержащие избыточную информацию, позволяющую восстановить исходные данные при отказе диска. Отсюда получают акроним для избыточных матриц недорогих дисков RAID (redundant array of inexpensive disks). Существует несколько способов объединения дисков RAID. Каждый уровень представляет свой компромисс между пропускной способностью ввода/вывода и емкостью диска, предназначенной для хранения избыточной информации.

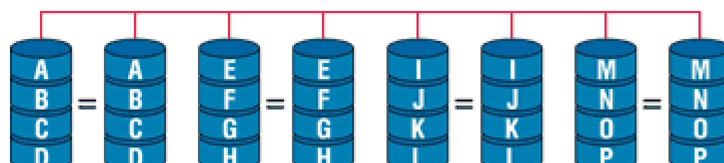
Когда какой-либо диск отказывает, предполагается, что в течение короткого интервала времени он будет заменен и информация будет

восстановлена на новом диске с использованием избыточной информации. Это время называется средним временем восстановления (mean time to repair - MTTR). Этот показатель можно уменьшить, если в систему входят дополнительные диски в качестве "горячего резерва": при отказе диска резервный диск подключается аппаратно-программными средствами. Периодически оператор вручную заменяет все отказавшие диски. Четыре основных этапа этого процесса состоят в следующем:

1. определение отказавшего диска,
2. устранение отказа без останова обработки;
3. восстановление потерянных данных на резервном диске;
4. периодическая замена отказавших дисков на новые.

RAID1: Зеркальные диски

Зеркальные диски представляют традиционный способ повышения надежности магнитных дисков. Это наиболее дорогостоящий из рассматриваемых способов, так как все диски дублируются и при каждой записи информация записывается также и на проверочный диск. Таким образом, приходится идти на некоторые жертвы в пропускной способности ввода/вывода и емкости памяти ради получения более высокой надежности. Зеркальные диски широко применяются многими фирмами. В частности компания Tandem Computers применяет зеркальные диски, а также дублирует контроллеры и магистрали ввода/вывода с целью повышения отказоустойчивости. Эта версия зеркальных дисков поддерживает параллельное считывание.



Дублирование всех дисков может означать удвоение стоимости всей системы или, иначе, использование лишь 50% емкости диска для хранения

данных. Повышение емкости, на которое приходится идти, составляет 100%. Такая низкая экономичность привела к появлению следующего уровня RAID.

RAID 2: матрица с поразрядным расслоением

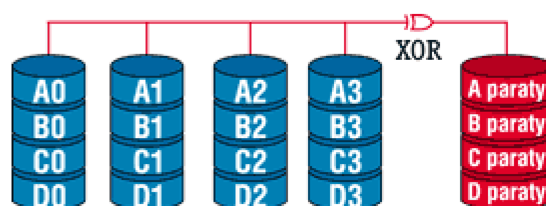
Один из путей достижения надежности при снижении потерь емкости памяти может быть подсказан организацией основной памяти, в которой для исправления одиночных и обнаружения двойных ошибок используются избыточные контрольные разряды. Такое решение можно повторить путем поразрядного расслоения данных и записи их на диски группы, дополненной достаточным количеством контрольных дисков для обнаружения и исправления одиночных ошибок. Один диск контроля четности позволяет обнаружить одиночную ошибку, но для ее исправления требуется больше дисков.

Такая организация обеспечивает лишь один поток ввода/вывода для каждой группы независимо от ее размера. Группы большого размера приводят к снижению избыточной емкости, идущей на обеспечение отказоустойчивости, тогда как при организации меньшего числа групп наблюдается снижение операций ввода/вывода, которые могут выполняться матрицей параллельно.

При записи больших массивов данных системы уровня 2 имеют такую же производительность, что и системы уровня 1, хотя в них используется меньше контрольных дисков и, таким образом, по этому показателю они превосходят системы уровня 1. При передаче небольших порций данных производительность теряется, так как требуется записать либо считать группу целиком, независимо от конкретных потребностей. Таким образом, RAID уровня 2 предпочтительны для суперкомпьютеров, но не подходят для обработки транзакций. Компания Thinking Machine использовала RAID уровня 2 в ЭВМ Connection Machine при 32 дисках данных и 10 контрольных дисках, включая 3 диска горячего резерва.

RAID 3: аппаратное обнаружение ошибок и четность

Большинство контрольных дисков, используемых в RAID уровня 2, нужны для определения положения неисправного разряда. Эти диски становятся полностью избыточными, так как большинство контроллеров в состоянии определить, когда диск отказал при помощи специальных сигналов, поддерживаемых дисковым интерфейсом, либо при помощи дополнительного кодирования информации, записанной на диск и используемой для исправления случайных сбоев. По существу, если контроллер может определить положение ошибочного разряда, то для восстановления данных требуется лишь один бит четности. Уменьшение числа контрольных дисков до одного на группу снижает избыточность емкости до вполне разумных размеров. Часто количество дисков в группе равно 5 (4 диска данных плюс 1 контрольный). Подобные устройства выпускаются, например, фирмами Maxtor и Micropolis. Каждое из таких устройств воспринимается машиной как отдельный логический диск с учетверенной пропускной способностью, учетверенной емкостью и значительно более высокой надежностью.



RAID 4: внутригрупповой параллелизм

RAID уровня 4 повышает производительность передачи небольших объемов данных за счет параллелизма, давая возможность выполнять более одного обращения по вводу/выводу к группе в единицу времени. Логические блоки передачи в данном случае не распределяются между отдельными дисками, вместо этого каждый индивидуальный блок попадает на отдельный диск.

Достоинство поразрядного расслоения состоит в простоте вычисления кода Хэмминга, что необходимо для обнаружения и исправления ошибок в

системах уровня 2. В RAID уровня 3 обнаружение ошибок диска с точностью до сектора осуществляется дисковым контроллером. Следовательно, если записывать отдельный блок передачи в отдельный сектор, то можно обнаружить ошибки отдельного считывания без доступа к дополнительным дискам. Главное отличие между системами уровня 3 и 4 состоит в том, что в последних расслоение выполняется на уровне сектора, а не на уровне битов или байтов.

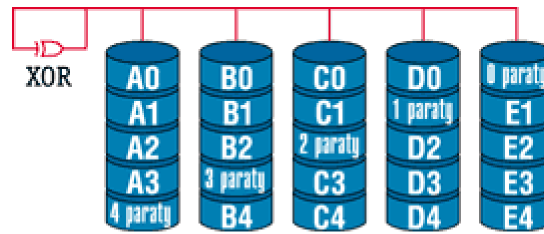
В системах уровня 4 обновление контрольной информации реализовано достаточно просто. Для вычисления нового значения четности требуются лишь старый блок данных, старый блок четности и новый блок данных:

новая четность = (старые данные xor новые данные) xor старая четность

В системах уровня 4 для записи небольших массивов данных используются два диска, которые выполняют четыре выборки (чтение данных плюс четности, запись данных плюс четности). Производительность групповых операций записи и считывания остается прежней, но при небольших (на один диск) записях и считываниях производительность существенно улучшается. К сожалению, улучшение производительности оказывается недостаточной для того, чтобы этот метод мог занять место системы уровня 1.

RAID 5: четность вращения для распараллеливания записей

RAID уровня 4 позволяли добиться параллелизма при считывании отдельных дисков, но запись по-прежнему ограничена возможностью выполнения одной операции на группу, так как при каждой операции должны выполняться запись и чтение контрольного диска. Система уровня 5 улучшает возможности системы уровня 4 посредством распределения контрольной информации между всеми дисками группы.

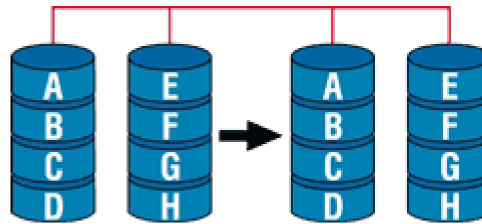


Это небольшое изменение оказывает огромное влияние на производительность записи небольших массивов информации. Если операции записи могут быть спланированы так, чтобы обращаться за данными и соответствующими им блоками четности к разным дискам, появляется возможность параллельного выполнения N/2 записей, где N - число дисков в группе. Данная организация имеет одинаково высокую производительность при записи и при считывании как небольших, так и больших объемов информации, что делает ее наиболее привлекательной в случаях смешанных применений.

RAID 6: Двумерная четность для обеспечения большей надежности

Этот пункт можно рассмотреть в контексте соотношения отказоустойчивость/пропускная способность. RAID 5 предлагают, по существу, лишь одно измерение дисковой матрицы, вторым измерением которой являются секторы. Теперь рассмотрим объединение дисков в двумерный массив таким образом, чтобы секторы являлись третьим измерением. Мы можем иметь контроль четности по строкам, как в системах уровня 5, а также по столбцам, которые, в свою очередь, могут расслаиваться для обеспечения возможности параллельной записи. При такой организации можно преодолеть любые отказы двух дисков и многие отказы трех дисков. Однако при выполнении логической записи реально происходит шесть обращений к диску: за старыми данными, за четностью по строкам и по столбцам, а также для записи новых данных и новых значений четности. Для некоторых применений с очень высокими требованиями к отказоустойчивости такая избыточность может оказаться приемлемой,

однако для традиционных суперкомпьютеров и для обработки транзакций данный метод не подойдет.



В общем случае, если доминируют короткие записи и считывания и стоимость емкости памяти не является определяющей, наилучшую производительность демонстрируют системы RAID уровня 1. Однако если стоимость емкости памяти существенна, либо если можно снизить вероятность появления коротких записей (например, при высоком коэффициенте отношения числа считываний к числу записей, при эффективной буферизации последовательностей считывания-модификации-записи, либо при приведении коротких записей к длинным с использованием стратегии кэширования файлов), RAID уровня 5 могут обеспечить очень высокую производительность, особенно в терминах отношения стоимость/производительность.

5.6. Организация памяти в процессорах Pentium

Общие сведения по преобразованию адреса

Организация памяти в процессорах семейства Pentium (x86) позволяет использовать следующие режимы работы

1. Реальный
2. Защищенный (сегментации)
3. Страничный
4. Системного управления

Реальный режим служит для обеспечения преемственности предыдущих процессоров и программ, а также для некоторых применений.

Защищенный режим служит для того, чтобы дать каждой программе несколько независимых, защищенных адресных пространств.

Страничный используется прежде всего для поддержки виртуальной памяти, т.е. среды, в которой большие адресные пространства моделируются на базе небольшой области оперативной памяти и некоторой дисковой памяти. Разработчики систем могут использовать как оба этих механизма (2 и 3), так и любой из них. При одновременном выполнении нескольких программ для защиты программ от влияния на них других программ можно использовать любой механизм.

Режим системного управления прежде всего предназначен для обеспечения перехода процессора в состояние пониженного энергопотребления (будет рассмотрен позднее).

5.6.1. Трансляция адреса в реальном режиме

В реальном режиме сегментный регистр содержит непосредственно компоненту адреса.

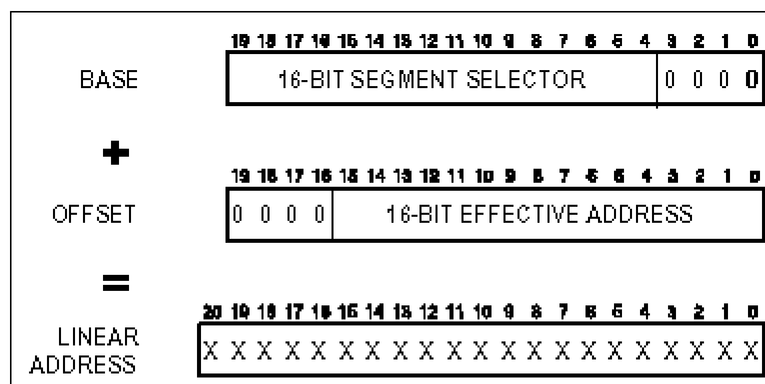


Рис. 5.5. Схема троансляции адреса в реальном режиме

Для получения *линейного адреса* в реальном режиме, содержимое сегментного регистра сдвигается на четыре позиции влево (т.е. умножается на 16) и полученный 20-битовый результат складывается с 16-битовым offset (внутрисегментным смещением), вычисленным в соответствии с используемым способом адресации). Эти действия выполняются в процессоре аппаратно блоком формирования адреса и незаметны для программиста. В каждый данный момент времени активны/доступны

программе (для 386+) шесть сегментов по числу имеющихся в процессоре сегментных регистров.

В реальном режиме сформированный *линейный адрес* подается непосредственно на физическую адресную шину, т.е. в реальном режиме понятия *линейного* и *физического* адресов совпадают.

Таким образом, в реальном режиме размер сегмента фиксирован. Максимальный адрес, который можно сформировать по приведенной схеме, равен 0xFFFF0 + 0xFFFF = 0x10FFEF, что дает общее количество адресов 0x10FFF0 = 0x10000 + 0x1000 – 0x10 = $2^{20} + 2^{16} - 2^{4}$.

В реальном режиме содержимое сегментных регистров доступно прикладному программисту и может быть им изменено. Задавая содержимое сегментного регистра, программист может расположить сегмент в физическом адресном пространстве с шагом в 16 байт.

5.6.2. Трансляция адреса в защищенном режиме

В *защищенном режиме* схема вычисления линейного адреса аналогична схеме реального режима в том отношении, что *линейный адрес*, как и в реальном режиме, получается сложением *базового адреса сегмента* и значения offset. Разница состоит в следующем.

1) *Базовый адрес* каждого из шести сегментов хранится в не видимом программисту (теневом) «продолжении» соответствующего сегментного регистра и имеет длину 32 бита (4 байта). Теневая часть сегментного регистра имеет длину 8 байтов (64 бита).

2) Логический адрес (offset), как для кода, так и для данных имеет длину 32 бита (4 байта).

3) Диапазон значений линейных адресов определяется длиной линейного адреса, которая также составляет 32 бита. Таким образом, базовый адрес задает положение начала сегмента, которое может располагаться в любом месте диапазона линейных адресов.

4) Длина сегмента в защищенном режиме не фиксирована, а задается 20-битовым полем Limit (предел), которое вместе с базовым адресом находится в теневой части сегментного регистра. Длина сегмента может иметь величину от 1 байта до 4 Гбайт.

5) При выполнении команды, процессор сравнивает значение offset с длиной сегмента. Если величина offset превышает размер сегмента (т.е. программа пытается обратиться за границу сегмента), выполнение данной команды останавливается и генерируется прерывание по исключительной ситуации.

6) Если предел не превышен, то вычисляется линейный адрес, и выполнение команды продолжается (и завершается).

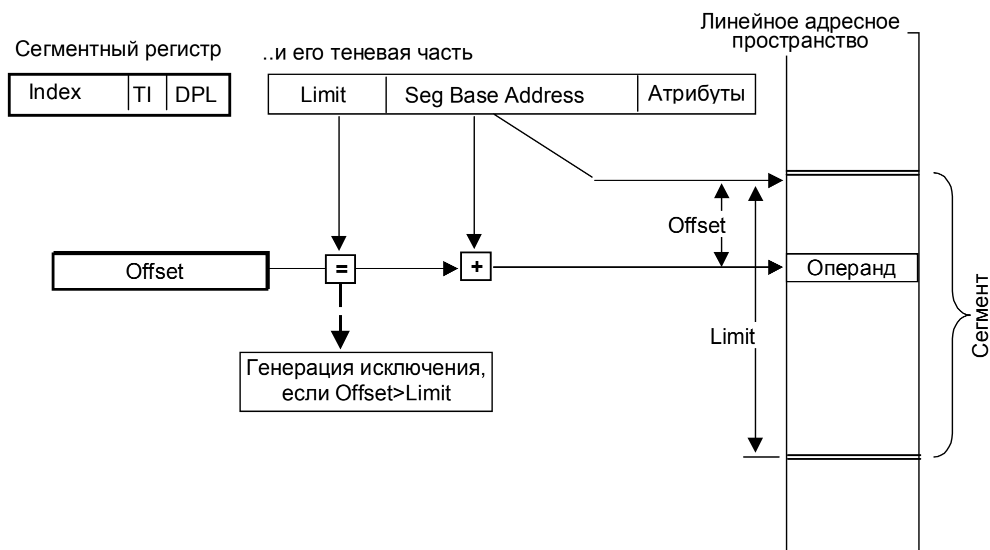


Рис. 5.6. Схема использования теневой части сегментного регистра.

Кроме базового адреса (32 бита) и длины сегмента (20 битов), в теневой части каждого сегментного регистра хранится еще ряд характеристик (атрибутов) данного сегмента. Они в совокупности с базовым адресом и длиной образуют *дескриптор* (описатель) сегмента.

Для работы одной программы необходимо определить хотя бы четыре сегмента:

- сегмент кода **cs**,

- сегмент стека **ss** и
- два сегмента данных **ds** и **es**.

Однако защищенный режим процессоров x86 предназначен для многозадачной работы, когда в памяти ЭВМ одновременно находятся и попеременно (во времени) выполняются несколько программ. Дескрипторы сегментов программ формируются операционной системой в специальных системных структурах данных – *дескрипторных таблицах* при подготовке программы к выполнению. Дескрипторные таблицы постоянно хранятся в ОЗУ.

При загрузке ОС вначале работают программы реального режима, котрые создают дескрипторные таблицы и определяют дескрипторы для компонент ОС, после чего процессор переводится в защищенный режим.

При последующей работе ОС для каждой загружаемой программы определяет несколько сегментов, для каждого сегмента заводит дескриптор и задает для сегмента длину и атрибуты.

В ходе (попеременной) работы нескольких программ часть этих сегментов (или даже все) оказываются загруженными в ОЗУ. Для всех сегментов, загруженных в ОЗУ, операционная система определяет и записывает в дескриптор базовый адрес в линейном адресном пространстве.

Каждая задача, выполняемая на компьютере, может иметь собственную *локальную дескрипторную таблицу Local Descriptor Table (LDT)*, содержащую описания сегментов (а также и других объектов), доступных только этой задаче. Кроме того, имеется одна *глобальная дескрипторная таблица Global Descriptor Table (GDT)*, которая содержит описания сегментов, потенциально доступных любой задаче.

Каждый из шести сегментных регистров содержит двоичное слово, называемое селектором сегмента. Селектор сегмента однозначно задает дескриптор сегмента, который используется процессором в данный момент.

Схема трансляции адреса с использованием GDT приведена на рис 5.7.

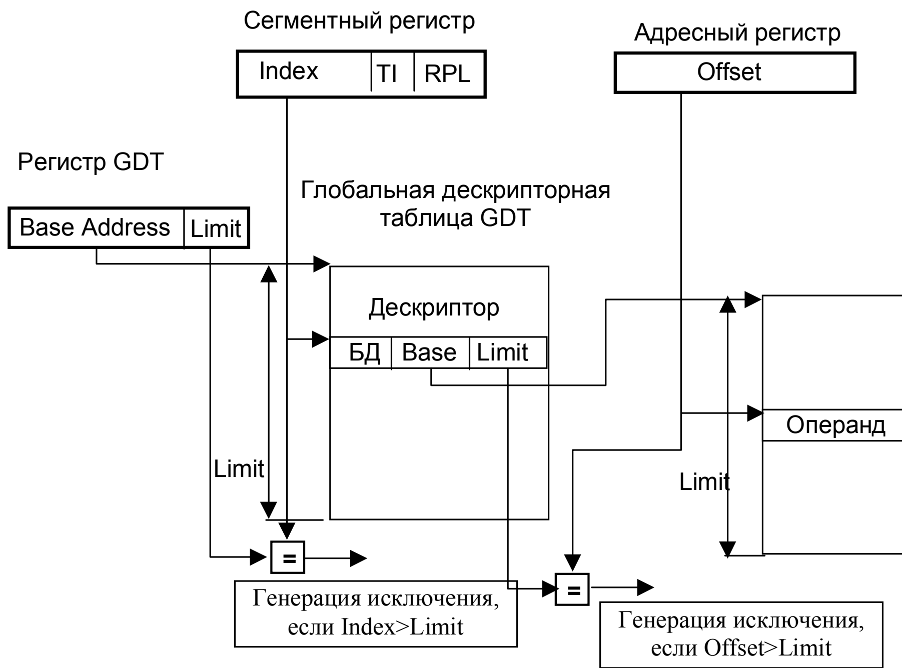


Рис. 5.7. Схема трансляции адреса с использованием GDT.

Схема трансляции адреса с использованием LDT приведена на рис.5.8.

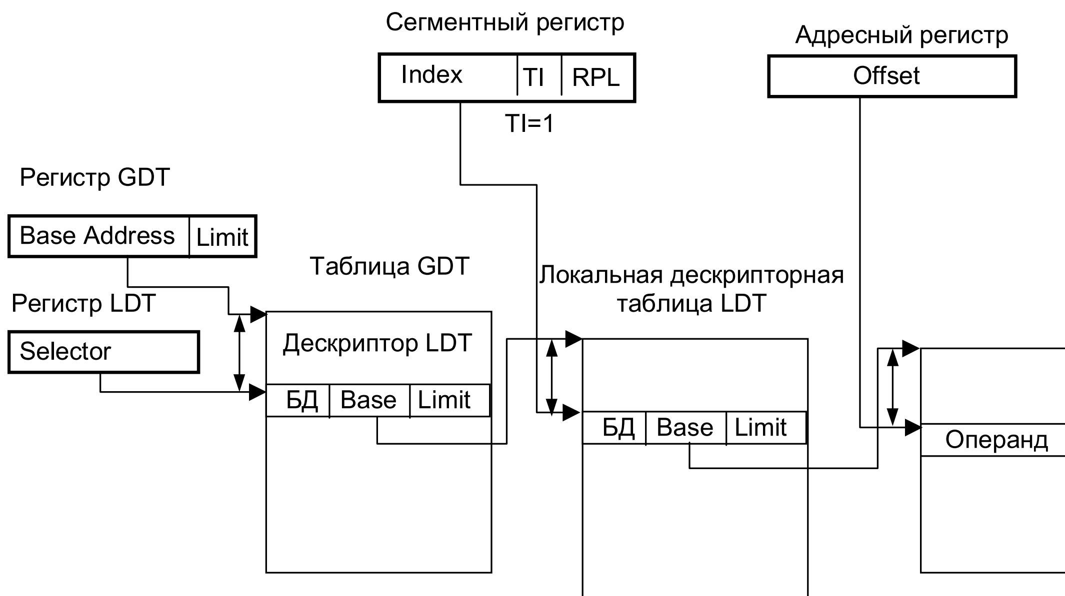


Рис.5.8. Схема трансляции адреса с использованием LDT.

Рассмотрим форматы отдельных элементов. Формат селектора сегмента представлен на рис. .

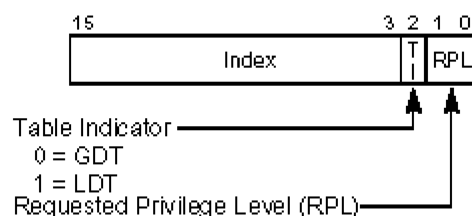


Рис. 5.9. Формат селектора сегмента

Поле *INDEX* указывает номер дескриптора в таблице дескрипторов. Таблица дескрипторов также представляет собой сегмент размером до 64 Кбайт, и может содержать до 8192 элементов.

Поле *TI (Table Indicator)* выбирает одну из двух дескрипторных таблиц: *глобальную GDT* или *локальную LDT*. Каждой задаче могут быть доступны две дескрипторные таблицы, а общее максимальное количество сегментов составляет 16384. (Следует помнить, что часть дескрипторов всегда занята под служебные нужды, сами дескрипторные таблицы тоже представляют собой сегменты, нулевой дескриптор в таблице не допускается использовать, и т.п.)

Двухбитовое поле *RPL* указывает запрошенный программой *уровень привилегий* (см. далее) и используется при защите памяти.

Рассмотрим подробнее формат дескриптора сегмента (изображен на рис 5.10.).

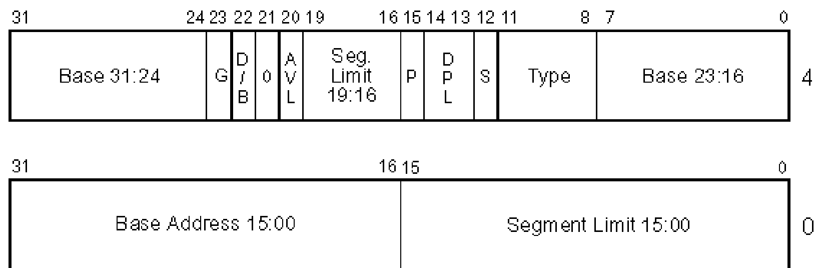


Рис. 5.10. Формат дескриптора сегмента

32-битовое **поле *базового адреса*** используется при вычислении *линейного* адреса в соответствии с вышеприведенной схемой.

20-битовое поле предела *Segment Limit* содержит размер сегмента. В зависимости от значения бита G (*Granularity)* этот размер может быть указан в байтах (при G=0 – максимальный размер сегмента 1Мбайт), либо в 4К-*страницах* (при G=1 – максимальный размер сегмента $2^{12} \cdot 2^{20}$=4Гбайт). Таким образом, размер сегмента может быть задан в пределах от 1 байта до 4 Гбайт

(в отличие от реального режима, где размер сегмента равен 2^{16} байт и фиксирован).

Однобитовый **признак размера операнда *D/B*** задает размер операнда по умолчанию, используемого командой при обращении к данному сегменту.

Двухбитовое поле **уровень привилегий сегмента *DPL*** *Descriptor Privilege Level* используется механизмом защиты памяти.

Бит **присутствия *P*** *Segment Present.* Бит присутствия может быть установлен, либо сброшен программно, аппаратно только анализируется его состояние. Нулевое значение этого бита при обращении к сегменту вызывает прерывание по исключительной ситуации. Этот бит может быть использован разработчиком операционной системы для «маркировки» сегментов, загруженных в данный момент в физическую память. Этот бит, в частности, может быть использован для органиации виртуальной памяти на уровне замены сегментов.

Бит ***S*** равен 0 в дескрипторах сегментов, содержащих код или данные и равен 1 для системных дескрипторов (описывающих сегменты, содержащие дескрипторные таблицы а также еще ряда других – они будут рассмотрены позже).

Поле ***TYPE*** в дескрипторах системных сегментов обозначает один из 12 возможных типов, а в дескрипторах сегментов кода или данных на этом месте содержатся дополнительные битовые поля, которые будут далее рассмотрены.

Байт, включающий поля ***P, DPL, S, TYPE*** носит название *байт доступа*.

При рассмотрении механизма формирования линейного адреса при трансляции сегмента возникают следующие вопросы:

1) Откуда берутся дескрипторные таблицы и конкретные значения полей в дескрипторах

2) Базовые адреса сегментов хранятся в дескрипторах, находящихся в дескрипторной таблице в основной памяти. Если в ходе трансляции

сегментов при выполнении каждой команды требуется (возможно неоднократно) обращаться за значениями базовых адресов в основную память, команды в защищенном режиме должны выполняться гораздо медленнее, нежели в реальном. Так ли это?

Ответим на эти вопросы.

Дескрипторные таблицы создаются и заполняются при инициализации операционной среды компонентами операционной системы. Это означает, что если в реальном режиме процессор способен немедленно после включения выбирать (fetch) и выполнять команды, то для того, чтобы выполнить хотя бы одну команду в защищенном режиме, должна быть создана хотя бы одна дескрипторная таблица (GDT), содержащая хотя бы один дескриптор, описывающий сегмент с кодом программы, а если в программе содержатся команды обращения к памяти, то требуется наличие еще одного дескриптора, описывающего сегмент данных.

Для ускорения процедуры трансляции сегментов, как уже было отмечено, дескрипторы сегментов, селекторы которых загружены в сегментные регистры, скопированы в невидимые, «теневые» 8-байтовые «продолжения» сегментных регистров. Именно из теневых частей (быстро !) извлекается значение базового адреса при трансляции сегмента.

VISIBLE PART	INVISIBLE PART	
SELECTOR	BASE ADDRESS, LIMIT, ETC.	CS
		SS
		DS
		ES
		FS
		GS

Рис. 5.11. Теневые части сегментных регистров.

Когда происходит загрузка дескриптора в теневой регистр? Это происходит при выполнении команд, изменяющих содержимое «обычных» сегментных регистров (команды **mov seg_reg, reg; lds**; **les**; **lfs, lgs, lss**, **jmp far, call far, int n,…**). При выполнении подобных команд в

защищенном режиме, процессор не только помещает новое значение (селектор сегмента) в видимую часть сегментного регистра, но и загружает из соответствующей дескрипторной таблицы восьмибайтовый дескриптор в теневую часть сегментного регистра. Вследствие этого команды, изменяющие содержимое сегментных регистров в защищенном режиме выполняются значительно дольше, чем в реальном.

Как процессор находит дескриптор? Для этого ему надо знать адрес начала (базовый адрес) дескрипторной таблицы и расположение в ней нужного дескриптора (смещение). Индекс дескриптора содержится в селекторе, загружаемом в сегментный регистр. Для хранения базовых адресов дескрипторных таблиц в архитектуру процессоров было введено несколько новых регистров GDTR, LDTR, IDTR, TR. Рассмотрим их назначение.

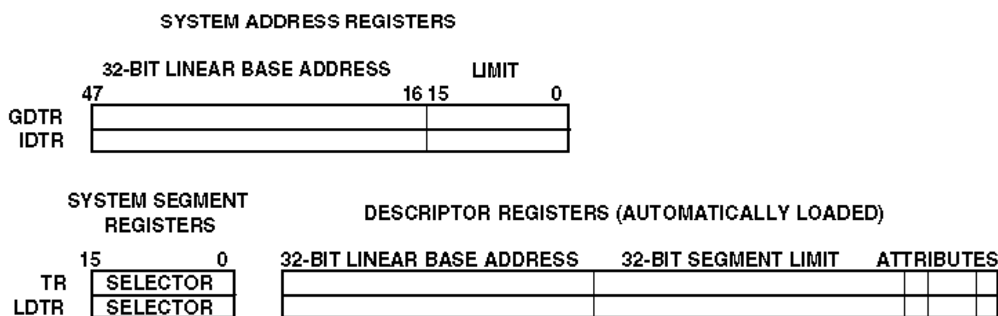


Рис. 5.12. Системные адресные регистры.

Глобальная дескрипторная таблица хранится в отдельном сегменте памяти. 32-битовый базовый адрес этого сегмента и его 16-битовая длина хранятся в **регистре GDTR (Global Descriptor Table Register)**. Таким образом, максимальная длина таблицы составляет 64 кБайт, в ней может максимально содержаться 8192 дескриптора.

Локальные дескрипторные таблицы могут создаваться индивидуально для каждой задач. Каждая локальная дескрипторная таблица также хранится в отдельном сегменте, который описывается дескриптором, хранимым в GDT. При запуске данной задачи ее локальная дескрипторная таблица

активизируется путем записи селектора ее дескриптора в 16-битовый ***регистр LDTR* (Local Descriptor Table Register)**. При этом автоматически загружается теневая часть этого регистра, куда переносятся из GDT базовый адрес LDT и ее предел, а также битовые поля атрибутов сегмента.

Регистр *IDTR* (Interrupt Descriptor Table Register) хранит базовый адрес и длину сегмента, содержащего *таблицу дескрипторов прерываний*. Дескриптор прерывания выполняет в защищенном режиме те же функции, что вектор прерывания в реальном режиме.

Наконец, **регистр задачи *TR* (Task Register)** содержит селектор **сегмента задачи TSS (Task State Segment)**. Его назначение рассматривается в разделе, посвященном поддержке многозадачности.

Рассмотрим более подробно схему обращения к памяти в защищенном режиме.

1) После получения команды на запуск программы, операционная система загружает в память первый сегмент кода программы, заполняет в соответствующем дескрипторе поля базы и предела, устанавливает в 1 бит присутствия, а затем передает управление на первую команду программы. При этом перезагружается новым значением селектора сегментный регистр **cs**. Используя поле **index** этого селектора, процессор находит в дескрипторной таблице дескриптор соответствующего сегмента кода и переносит его в теневую часть регистра **cs**.

2) Положение объекта (команды или элемента данных) в памяти задается парой значений seg:offset. Значение seg – селектор сегмента – находится в сегментном регистре. Значение offset для выборки команды берется из счетчика команд, а для элемента данных аппаратно вычисляется процессором в соотвествии с используемым способом адресации. При обращении к памяти, процессор вычисляет линейный адрес. Для этого он складывает значение базового адреса сегмента (из теневой части сегментного регистра) со значением offset и производит обращение к памяти.

3) Если при выполнении команды оказывается, что сегмент, индекс которого используется в команде, отстутствует в памяти (процессор “узнает” об этом, проверяя значение бита присутствия), происходит прерывание по исключительной ситуации. Управление передается операционной системе, и она переносит недостающий сегмент в память с диска, заполняет поля базы и предела в дескрипторе, устанавливает в 1 бит присутствия, а затем (возвратом из прерывания) возвращает управление на ту же команду. Теперь эта команда может выполниться до конца

4) При выполнении каждой команды аппаратура процессора проверяет выполнение ряда условий (наличие сегмента в памяти – одно из этих условий). Если условие не выполняется, то выполнение команды прекращается и происходит прерывание по исключительной ситуации – это общий принцип срабатывания механизмов защиты.

Сегментный механизм можно использовать для организации виртуальной памяти. Однако это оказалось очень неудобным, так как обычно сегменты имеют разный размер и располагаются в памяти с произвольного адреса. При свопинге (замене) сегментов пространство памяти вследствие этого фрагментируется и, тем самым, расходуется нерационально. Сегментный механизм появился в семействе x86, начиная с модели i80286. Однако работоспособной ОС с виртуальной памятью на базе этого процессора, которая стала бы сколько-нибудь популярной, так и не было разработано. В процессорах в дополнение к сегментному механизму был разработан *страничный механизм трансляции адресов*.

Оценим, каков максимальный размер логического адресного пространства, которым может обладать задача.

Задаче могут быть доступны две дескрипторные таблицы: GDT и LDT, каждая из которых может содержать до 8192 (2^{13}) дескрипторов сегментов. Каждый сегмент может иметь размер до 4 Гбайт (2^{32} байт). Таким образом, ограничение сверху на размер логического адресного пространства

составляет

$2 * 2^{13} * 2^{32} = 2^{46}$ байт. Естественно, такой объем еще весьма долго не сможет целиком поместиться в физической памяти компьютера (для процессоров x86 ее размер ограничен величиной 2^{32} байт). Однако при наличии виртуальной памяти ограничение размера физической памяти лишь замедлит выполнение очень большой программы (за счет потерь на обмен сегментов между памятью и диском).

5.6.3. Страничный механизм

Для реализации виртуальной памяти очень желательно, чтобы заменяемые блоки имели одинаковый размер и были в памяти выровнены (т.е. лежали с адресов, кратных этому размеру) – это позволит избежать фрагментации памяти при свопинге. Такой механизм имеется в процессорах Pentium.

Если включен страничный механизм, то все физическое адресное пространство разбивается на блоки одинакового размера – страницы. В процессорах семейства x86 страницы могут иметь размер 4 кбайт либо 4 Мбайт. Существующие операционные системы используют в основном маленькие 4 Кбайтные страницы. Дальнейшее рассмотрение идет применительно именно к этому варианту.

Линейный адрес (результат трансляции сегмента) интерпретируется процессором, как состоящий из трех частей (рис. 5.13). Младшие 12 разрядов адреса определяют относительный адрес внутри страницы.

Старшие 20 разрядов линейного адреса (*индекс страницы)* задают одну из 2^{20} страниц. Это делается с помощью двухступенчатого табличного преобразования. Старшие 20 разрядов линейного адреса интерпретируются как два 10-битовых поля. Разряды 12…20 задают номер *дескриптора страницы* (Page Table Entry) в *таблице страниц* (Page Table).

Дескриптор страницы имеет длину 4 байта. Одна таблица содержит 1024 дескриптора.

Старшие 20 разрядов дескриптора страницы содержат ее базовый адрес. Процессор формирует физический адрес операнда с помощью конкатенации базового адреса (из дескриптора) и внутристраничного смещения (младшая часть линейного адреса).

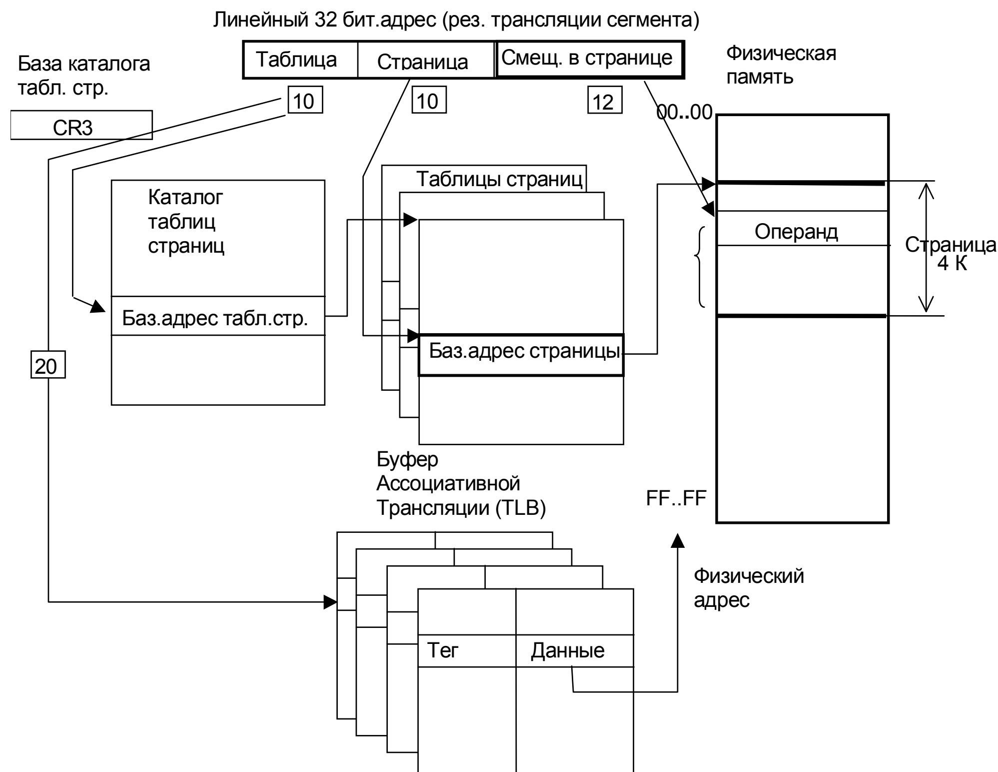


Рис. 5.13.Страничный механизм трансляции адреса.

Табличное преобразование адреса при включенном страничном механизме выполняется один или несколько раз при выполнении каждой команды. Для того, чтобы страничная трансляция присходила быстро, для хранения индексов недавно использовавшихся страниц и их базовых адресов используется небольшая память с ассоциативным доступом – *Буфер Ассоциативной Трансляции (Translation LookAside Buffer TLB*).

Таблица страниц сама представляет собой 4 кбайт страницу. Одна программа использует много страниц (иногда больше, чем 1024), т.е. ей может требоваться больше, чем одна таблица страниц. В схеме трансляции страниц может одновременно использоваться до 1024 таблиц страниц.

Каждая таблица страниц в свою очередь описывается дескриптором в еще одной таблице – в *каталоге таблиц страниц (Page Directory)*.

Старшие десять битов линейного адреса задают индекс дескриптора в каталоге таблиц страниц, т.е. используемую при данном обращении таблицу страниц.

Рассмотрим структуру дескриптора страницы более подробно. Дескрипторы страниц кроме базового адреса, содержат набор битовых полей, облегчающих реализацию виртуальной памяти и защиты.

P – бит присутствия страницы в памяти

R/W – бит разрешения записи на данную страницу – позволяет ограничить доступ к отдель- ным диапазонам адресов

U/S – бит уровня привилегий – пользовательская/системная страница

PWT – алгоритм работы данной страницы с КЭШ-памятью – разрешение «сквозной» записи (Write-Throuhg)

PCD – бит запрета кэширования страницы (Page Cache Disabled)

A – признак того, что к элементам страницы производился доступ (Accessed)

D – признак того, что на страницу производиласть запись (флог модификации – Dirty)

G – признак «глобальной» (Global) страницы, используется для страниц, постоянно находящихся в памяти (например, чдля ядра операционной системы. Это признак запрещает удалять дескриптор такой страницы из *TLB*.

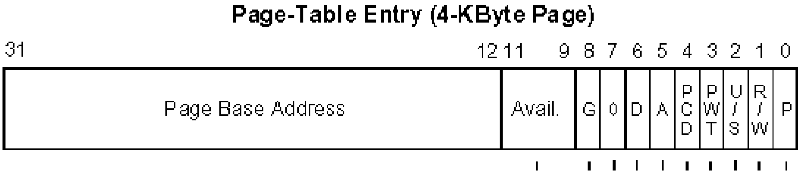


Рис. 5.14. Структура дескриптора страницы

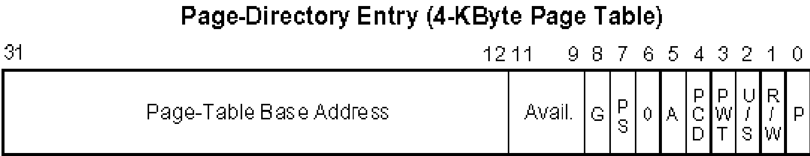


Рис. 5.15. Структура дескриптора таблицы страниц

Реализация виртуальной памяти на уровне страниц:

- Обращение к странице (P=0) > особ. случ. страничная ошибка (№=14)
- Вызывается обработчик страничной ошибки.
- Из CR2 линейный адрес стр. ошибки. Из стека код стр. ошибки.
- Операц. система по табл. ищет нужную страницу на винчестере.
- Из ОЗУ выбрасывается стр. (используется бит A и алгоритм LRU), Она записыв. при D=1
- В ОЗУ записывается нужная страница. Повторно выполняется команда.

В процессорах Pentium Pro и старше реализован механизм расширения физического адресного пространства.

Таким образом, весь процесс преобразования адреса можно изобразить схемой рис.5.16.

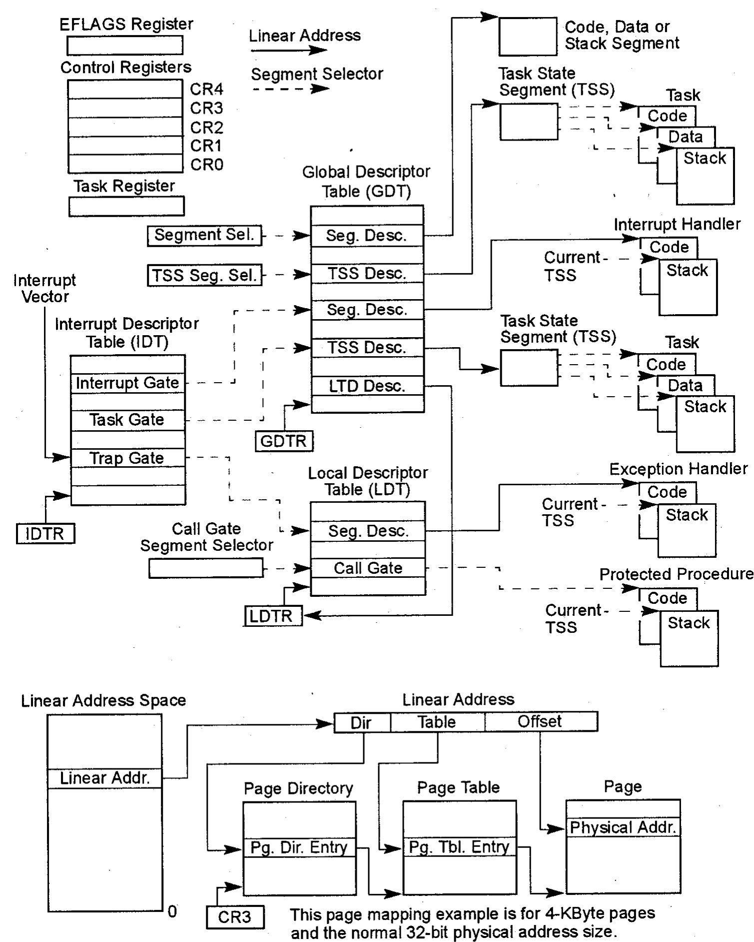


Рис. 5.16. Полная схема трансляции адресов.

(заимствовано из документа «Intel Architecture Software Developer’s Manual, Volume 3: System Programming Guide». стр. 2-2 или 38 по сквозной нумерации.)

5.6.4. Защита в процессоре Pentium

При выполнении программой тех или иных действий, в частности обращений к памяти, процессор аппаратно проверяет также некоторые условия. При их несоблюдении вместо продолжения выполнения команды генерируется прерывание по исключительной ситуации (и затем как правило, передача управления операционной системе).

Проверки , выполняемые процессором при выполнении команд, можно подразделить на следующие категории:

• Limit checks – проверки превышения границы сегмента.

• Type checks – проверки допустимости типа дескриптора при обращении, возможности чтения или записи в сегмент.

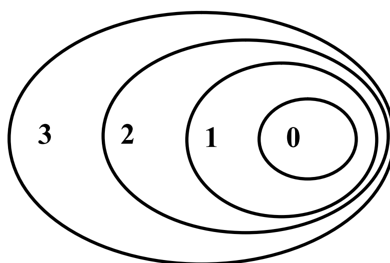
• Privilege level checks – проверки уровня привилегий.

• Restriction of addressable domain – ограничение доступа к отдельным диапазонам адресов.

• Restriction of procedure entry-points – проверки доступности точек входа в процедуры (механизм шлюзования).

• Restriction of instruction set – проверки привилегированных команд.

Рассмотрим более подробно ситуации, в которых возможно срабатывание механизма защиты.



0 – ядро операционно системы
0,1,2 - супервизор; 3 - пользователь

Рис. 5.16. Кольца защиты.

В защищенном режиме каждой программе может быть присвоен один из четырех уровней привилегий (0 – наиболее привилегированный, 3 – наименее).

Операционная система **при формировании дескрипторов** и заполнении дескрипторной таблицы, назначает для каждого сегмента уровень привилегий (обычно 0 – для сегментов операционной системы, и 3 – для программ – приложений), и записывает соответствующий код в поле **DPL** дескриптора.

При загрузке сегмента кода, его DPL помещается в младшие биты регистра **CS**, это значение называется **CPL** Current Privilege Level – текущий уровень привилегий, т.е. уровень привилегий выполняемого в данный момент кода.

Прикладная программа не может напрямую установить значения базы и предела для сегмента, а может лишь выбрать один из имеющихся в дескрипторной таблице (которую заполнила ОС).

Байт доступа и типы сегментов

Поле **TYPE** байта доступа в дескрипторах кода или данных содержит четыре однобитовых поля:

Бит11 служит признаком вида сегмента: 0 – сегмент данных, 1 – сегмент кода

Бит8 – **А** - бит обращения (access). При создании дескрипторов операционная система должна установить значение этого бита для всех сегментов нулевым. При загрузке сегмента в память процессор автоматически устанавливает этот бит в 1, сигнализируя тем самым, что к данному сегменту уже было обращение..

Биты 9 и 10 имеют разное значение для сегментов данных и кода.

Для сегмента данных:

Бит9 – **W** (Write) разрешение записи: 0- только чтение, 1- чтение/запись. При попытке записи в сегмент «только для чтения» произойдет прерывание по исключительной ситуации.

Бит10 – **D** (Direction) - направление расширения сегмента. Если этот бит установлен, то сегмент предназначен для стека, который растет в направлении младших адресов. Этот бит влияет на способ контроля процессором превышения размера сегмента.

Для сегмента кода:

Бит9 - **R** (Read) – разрешение чтения, если этот бит установлен, то сегмент кода можно читать.

Бит10 - C (Conforming) – бит «подчинения» (будет рассмотрен далее).

Системных дескрипторов имеется 13 типов, они перечислены в приводимой таблице.

Type Field					Description
Decimal	11	10	9	8	
0	0	0	0	0	Reserved
1	0	0	0	1	16-Bit TSS (Available)
2	0	0	1	0	LDT
3	0	0	1	1	16-Bit TSS (Busy)
4	0	1	0	0	16-Bit Call Gate
5	0	1	0	1	Task Gate
6	0	1	1	0	16-Bit Interrupt Gate
7	0	1	1	1	16-Bit Trap Gate
8	1	0	0	0	Reserved
9	1	0	0	1	32-Bit TSS (Available)
10	1	0	1	0	Reserved
11	1	0	1	1	32-Bit TSS (Busy)
12	1	1	0	0	32-Bit Call Gate
13	1	1	0	1	Reserved
14	1	1	1	0	32-Bit Interrupt Gate
15	1	1	1	1	32-Bit Trap Gate

Для системного дескриптора поле **TYPE** содержит код типа объекта, описываемого дескриптором, сегментами являются только некоторые из этих объектов; значения кодов также приведены в таблице.

Привилегированные команды

1. Нулевого приоритета

2. Чувствительные команды

- команды ввода.вывода

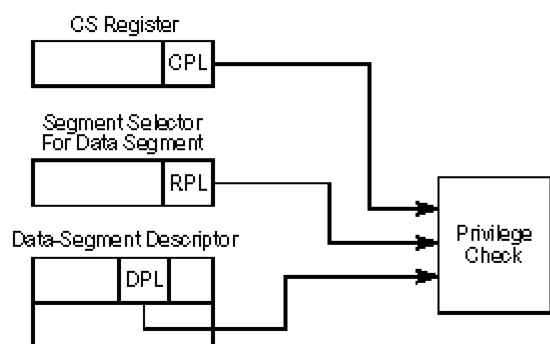
– работы с прерываниями

3. Команды работы с флагами IOPL IF

Ряд команд может выполняться только программами, имеющими 0-й уровень приоритета (т.е. операционной системой). Вот эти команды:

• LGDT—Load GDT register.
• LLDT—Load LDT register.
• LTR—Load task register.
• LIDT—Load IDT register.
• MOV (control registers)—Load and store control registers.
• LMSW—Load machine status word.
• CLTS—Clear task-switched flag in register CR0.
• MOV (debug registers)—Load and store debug registers.
• INVD—Invalidate cache, without writeback.
• WBINVD—Invalidate cache, with writeback.
• INVLPG—Invalidate TLB entry.
• HLT—Halt processor.
• RDMSR—Read Model-Specific Registers.
• WRMSR—Write Model-Specific Registers.
• RDPMC—Read Performance-Monitoring Counter.
• RDTSC—Read Time-Stamp Counter.

При загрузке сегмента данных (это происходит, когда программа перезагуржает один из сегментных регистров) анализируются три значения уровня привилегий:



1) CPL – уровень привилегий текущего сегмента кода

2) DPL – уровень привилегий сегмента, к которому происходит обращение

3) RPL – запрашиваемый уровень привилегий; это поле в селекторе операнда программист может установить сам, но не

выше CPL, т.е. $RPL \geq CPL$. Это позволяет понизить уровень привилегий.

Условие доступа: max (CPL,RPL=<DPL)

При обращении за операндами:

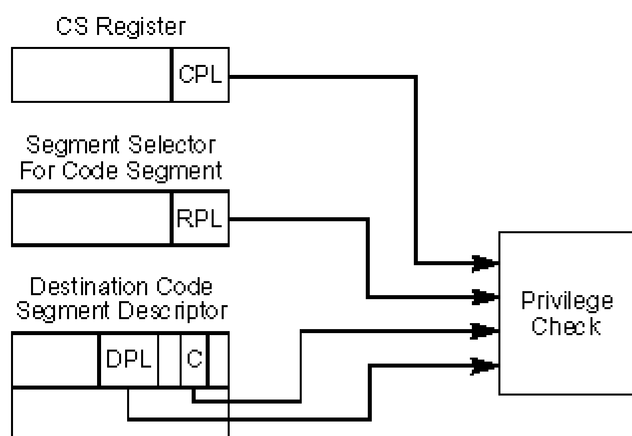
1) Если обращение за данными не в сегменты данных или стека, а к сегменту кода, для которого запрещено чтение (контроль типа) – исключение

2) Сравнивается offset и предел сегмента, если превышение – исключение

3) Если команда пытается записать в Read-Only сегмент данных (бит R=1) или в страницу, для которой запрещена запись – генерируется исключение.

При передаче управления посредством ближнего перехода или ближнего вызова подпрограммы в команде указывается логический адрес перехода, содержащий только смещение, при этом процессор осуществляет только контроль выхода за границу сегмента.

При передаче управления посредством дальнего перехода, дальнего вызова или при программном прерывании адрес перехода задан (в команде или в векторе прерывания) в формате *селектор_сегмента:смещение*. При этом происходит перезагрузка сегментного регистра CS новым значением селектора. Обычно это возможно только при выполнении условия $CPL \geq DPL$ (т.е. переход на более привилегированный сегмент кода невозможен и вызовет исключение). Однако это лишило бы возможности использования прикладной программой с низким уровнем привилегий сервисов ОС, работающих на более высоком уровне привилегий. Для разрешения этого противоречия реализованы **два разных механизма**.



Первый состоит в том, что для такого рода сервисов (компонентов ОС, выполненных в виде подпрограмм и имеющих высокий уровень привилегий, создается отдельный сегмент кода, в дескрипторе (в байте доступа) которого устанавливается *бит подчинения С*. При обращению к такому сегменту исключения не происходит, но CPL сохраняется равным уровню привилегий вызывающего сегмента. Это позволяет вызывать системную функцию с любого уровня привилегий.

Однако этот способ не всегда годится. Иногда сервису ОС требуется обладать высоким уровнем привилегий, чтобы выполнить требуемую функцию (например динамически выделить память – это часто означает – создать новый сегмент с заполнением дескрипторной таблицы).

Второй способ. Такой вызов можно безопасно допустить, если обеспечить возможность входа в вызываемый модуль только в предопределенной точке, и при этом контролировать процесс передачи параметров. Это делается путем указания в дальнем адресе вызова не селектора сегмента кода, а селектора так называемого **шлюза вызова (Call Gate)**. При таком переходе указываемое в адресе перехода смещение игнорируется (никак не используется) потому что «настоящие» селектор сегмента, куда будет выполнен вызов, содержатся в дескрипторе шлюза вызова, на который указывает селектор сегмента, заданный в команде **call far**.

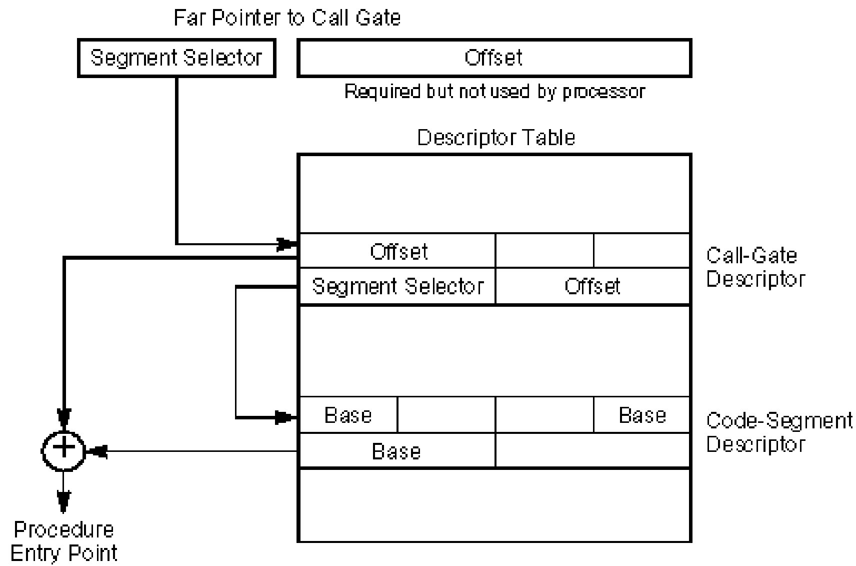
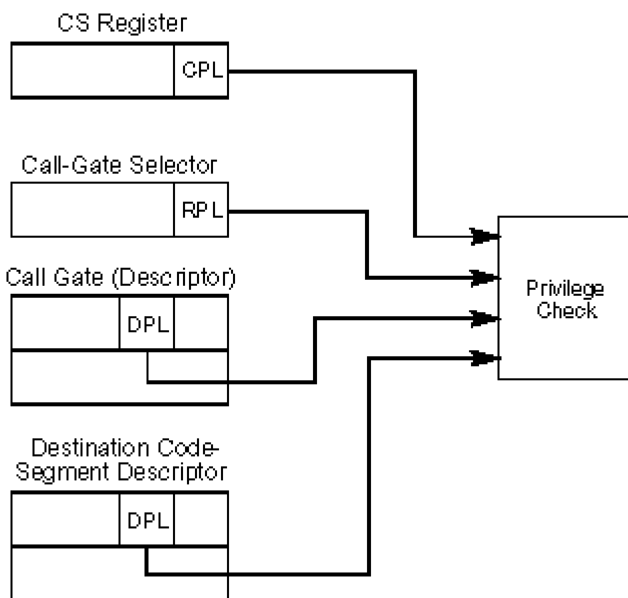


Рис. 5.19. Схема трансляции через шлюз.

Функции шлюза:

1. Определение точки входа в процедуру
2. Задание уровня привилегированности, требуемого для входа в процедуру.
3. Переключение стеков.



При проверке привилегий анализируются:

1) **RPL ≥ CPL**

2) Текущий уровень привилегий должен позволить обратиться к «шлюзу вызова»:

CPL ≥ CallGate-DPL

Имеется четыре разновидности шлюзов:

Шлюз вызова подпрограммы	Call gates
Шлюз программного прерывания или исключения	Trap gates

Шлюз аппаратного прерывания Interrupt gates

Шлюз задачи Task gates

Проверки, порождающие исключения, можно тоже разделить на группы (номера прерываний, которые возникнут по исключительной ситуации, указаны в скобках):

Проверка при *загрузке* сегментных регистров

• Превышение лимита таблицы дескрипторов (13).

• Несуществующий дескриптор сегмента (11 или 12).

• Нарушение привилегий (13).

• Загрузка неверного дескриптора или типа сегмента (13);

- загрузка в SS сегмента данных только чтения или сегмента кода,

- 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 (13).

• Чтение из только исполняемого сегмента кодов (13).

• Превышение лимита сегмента (12 или 13).

Проверка привилегий инструкций

• CPL $\neq$ 0 при выполнении инструкций LIDT, LLDT, LGDT, LTR, LMSW, CTS, HLT, операций с регистрами DRn, TRn, CRn (13).

• CPL > IOPL при выполнении инструкций STI, CLI, а для 80286 еще и инструкции LOCK (13).

• CPL > IOPL при выполнении инструкций IN, INS, OUT, OUTS с портами, не разрешенными битовой картой ввода-вывода (13).

При выполнении команд IRET и POPF с недостаточным уровнем привилегий не изменяются биты IF и IOPL в регистре флагов, что не порождает исключений:

• IF не меняется, при CPL>IOPL;

• IOPL не меняется, если CPLO.

Команды тестирования условий защиты

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, он приравнивается максимальному значению из текущего RPL селектора и поля RPL в указанном регистре. При изменении RPL устанавливается ZF=1;

VERR — верификация чтения: если сегмент, на который указывает селектор, допускает чтение, устанавливается ZF=1;

VERW — верификация 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.

5.6.5. Аппаратная поддержка мультизадачности

В защищеном режиме имеется аппаратная поддержка переключения задач, что облегчает работу создателя многозадачных операционных систем. Переключение на новую задачу во многом напоминает вызов подпрограммы, но при переключении задач автоматически сохраняется гораздо больше информации о состоянии процессора в специальных системных структурах данных. Далее перечислены эти структуры, а также регистры процессора, поддерживающие многозадачность.

1) Сегменты состояния задачи (Task State Segment TSS)
2) Дескрипторы сегмента состояния задачи
3) Дескрипторы шлюзов вызова задачи

4) Регистр задач.

Рассмотрим эти элементы подробнее.

Сегмент состояния задачи TSS

Его структура изображена на рис. 5.20. Каждая задача имеет собственный TSS.

31 … 16	15 … 0	
I/O MAP BASE ADDRESS	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 T	64
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	SELECTOR FOR TASK'S LDT	60
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	GS	5C
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	FS	58
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	DS	54
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	SS	50
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	CS	4C
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	ES	48
EDI		44
ESI		40
EBP		3C
ESP		38
EBX		34
EDX		30
ECX		2C
EAX		28
EFLAGS		24
EIP		20
CR3 (PDBR)		1C
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	SS2	18
ESP2		14
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	SS1	10
ESP1		C
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	SS0	8
ESP0		4
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	LINK (OLD TSS SELECTOR)	0

Рис. 5.20. Формат сегмента TSS.

Поля TSS можно разделить на две категории:

❖ **Динамические**, процессор обновляет их значения при каждом переключении данной задачи. В эту категорию входят следующие поля:

➢ регистры общего назначения EAX, ECX, EDX, EBX, ESP, EBP, ESI и EDI

➢ сегментные регистры ES, CS, SS, DS, FS и GS

- ➢ регистр флагов EFLAGS
- ➢ счетчик команд IP
- ➢ селектор сегмента TSS предыдущей задачи.

❖ **Статические**, при переключении задачи процессор только читает, но не обновляет эти поля. Их значения устанавливаются операционной системой при создании задачи. К этим полям относятся следующие:

- ➢ Селектор локальной дескрипторной таблицы задачи (каждая задача может иметь одну собственную LDT, но может и не иметь).
- ➢ Управляющий регистр CR3 (это BDPR – регистр базового адреса таблицы каталогов страниц.
- ➢ Логические адреса стеков для уровней привилегий 0, 1 и 2
- ➢ Значение Т-бита (включения режима пошаговой отладки)
- ➢ Базовый адрес карты разрешения портов ввода/вывода и карты переназначения прерываний (необязательные элементы). Если эти элементы присутствуют, то они сохраняются в верхних адресах TSS.

Дескриптор TSS

Сегмент состояния задачи, подобно другим сегментам, описывается дескриптором, формат которого следующий:

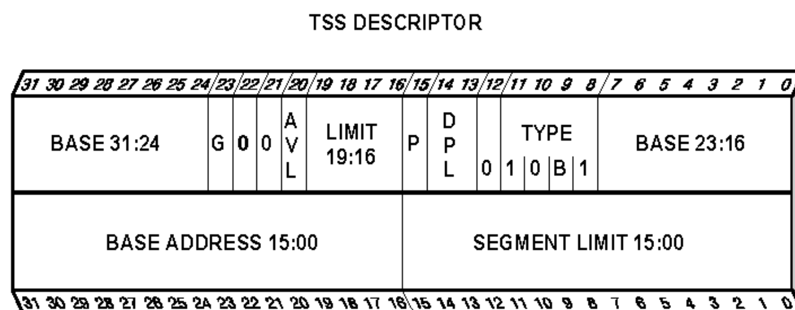


Рис. 5.21. Формат дескриптора TSS.

Рассмотрим особенности полей дескриптора. Поля базы, предела и DPL имеют то же назначение, что и для сегментов данных. Предел сегмента TSS должен быть равен или больше величины 67h (на 1 меньше минимального размера TSS – 68h). Размер TSS может быть и больше, при этом добавочное пространство может быть использовано операционной системой по своему усмотрению).

Задача не реентерабельна (т.е. не допускает рекурсивных вызовов). Для выявления попыток рекурсивного вызова служит бит занятости **B** (Busy

Bit). Если задача была вызвана (и затем, возможно, прервана, например вызовом другой задачи), бит занятости устанавливается и остается установленным, что и будет признаком рекурсивного вызова при попытке вызвать эту задачу.

Переключение задачи происходит вследствие передачи управления через дескриптор TSS командами JMP или CALL, а также при прерываниях или исключениях через этот дескриптор.

Регистр задачи

Его содержимое показывает местонахождение в памяти текущего TSS. Этот регистр имеет видимую (т.е. программно доступную) часть, содержащую селектор TSS выполняемой в данный момент задачи и «теневую», которая автоматически заполняется процессором при переключении задачи, и которая содержит базу и предел сегмента TSS. Селектор TSS является индексом дескриптора TSS, который может находиться только в GDT.

Привилегированные команды LTR и STR позволяют читать и писать этот регистр.

Дескриптор шлюза задачи

Дескриптор TSS обычно имеет высокий уровень привилегий, что исключает его использование «обычной» программой. Для обеспечения защищенного доступа к задаче используется шлюз задачи – дескриптор, имеющий низкий уровень привилегий, но обеспечивающий «косвенный» вызов на дескриптор TSS с высоким уровнем привилегий.

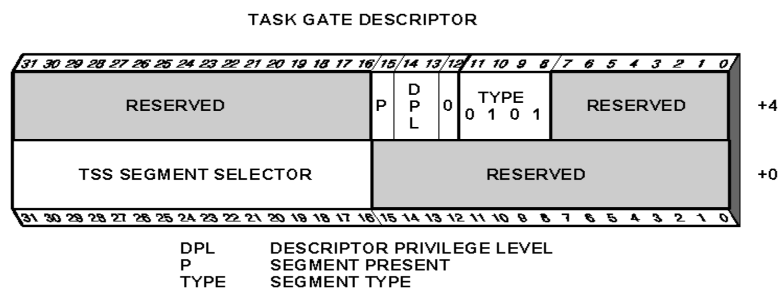


Рис. 5.22. Формат шлюза TSS.

Обратите внимание, что при использовании шлюза задачи защита анализирует DPL дескриптора шлюза, а не DPL дескриптора TSS.

Схема переключения задачи

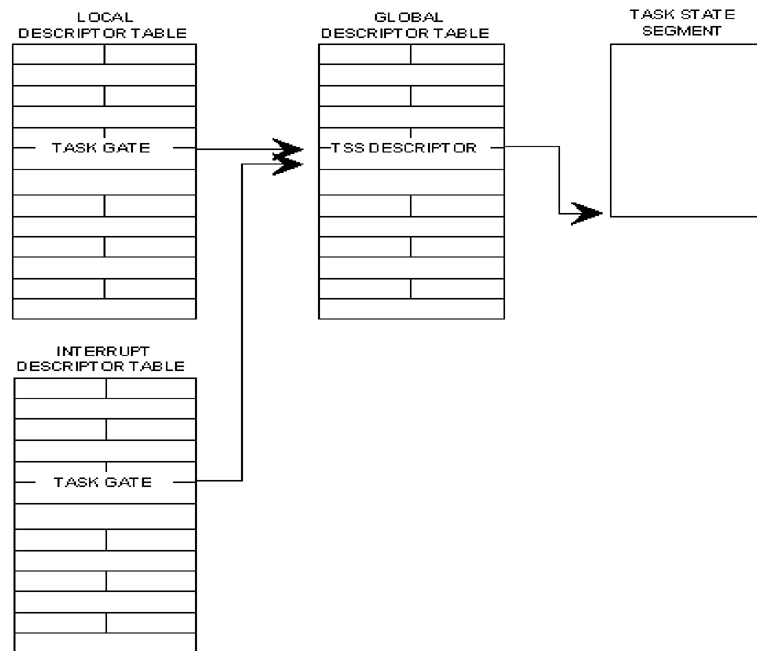


Рис. 5.23. Схема переключения задачи.

Процессор передает управление другой задаче в одном из четырех случаев:

1) Текущая задача выполняет JMP или CALL на дескриптор TSS
2) Текущая задача выполняет JMP или CALL на шлюз задачи
3) Прерывание или исключение происходит через шлюз задачи, находящийся в IDT
4) Текущая задача выполняет команду IRET при установленном флаге NT (в CR0)

Вложенные задачи

При переключении задачи в ее TSS происходит заполнение ссылки на TSS предыдущей задачи. Эта информация используется для возврата на прерванную задачу, после окончания прервавшей. Операционная система может модифицировать эти связи, изменяя записи в TSS. Каждая задача может иметь собственную LDT и систему трансляции страниц. Регистр

PDBR (Page Directory Base Register) (CR3) загружается новым значением из TSS при переключении задачи.

5.6.6. Прерывания в защищенном режиме

Источниками запросов, «запускающих» процесс входа в прерывание могут быть события трех типов: **аппаратные прерывания**, **исключения** и **программные прерывания**. В свою очередь, исключения подразделяются на: **отказы** (fault), **ловушки** (trap) и **выходы из процесса** (abort). Они различаются по способу реакции процессора.

Отказ регистрируется на границе (во времени) между командами **до** начала выполнения команды, вызвавшей прерывание (например, исключение по защите памяти или по делению на нуль). Это означает, что в стеке сохраняется адрес команды, вызвавшей исключение (а не адрес следущией команды). В ряде случаев после адекватной реакции на прерывание есть возможность заново вернуться к выполнению прерванной программы именно с команды, вызвавшей отказ. Например, по прерыванию «страничная ошибка» (возникающему при обращении к странице, остутствующей в памяти) управление передается операционной системе, она производит подгрузку в память отсутствующей страницы, после чего можно возобновить выполнение команды, вызвавшей страничную ошибку.

Ловушка регистрируется также на границе между командами, но после окончания команды, вызвавшей прерывание, при этом в стеке сохраняется адрес следующей команды. Примеры ловушек – программные прерывания INTO или INT3.

Выход из процесса – это исключение, после которого не только невозможно продолжить выполнение программы, вызвавшей прерывание, но и зачастую невозможно установить адрес команды, вызвавшей исключение.

Вектора прерываний в защищенном режиме

Vector Number	Description
0	Divide Error

1	Debug Exception
2	NMI Interrupt
3	Breakpoint
4	INTO-detected Overflow
5	BOUND Range Exceeded
6	Invalid Opcode
7	Device Not Available
8	Double Fault
9	CoProcessor Segment Overrun (reserved)
10	Invalid Task State Segment
11	Segment Not Present
12	Stack Fault
13	General Protection
14	Page Fault
15	(Intel reserved. Do not use.)
16	Floating-Point Error
17	Alignment Check
18	Machine Check*
19-31	(Intel reserved. Do not use.)
32-255	Maskable Interrupts

Таблица дескрипторов прерываний

Переход на обработчик происходит через дескрипторы, содержащиеся в таблице дескрипторов прерываний Interrupt Descriptor Table IDT. Каждый дескриптор связывает источник прерывания или исключения с подпрограммой (шлюз подпрограммы-обработчика) или с задачей (шлюз задачи) которая будет вызвана в ответ на событие, связанное с прерыванием.

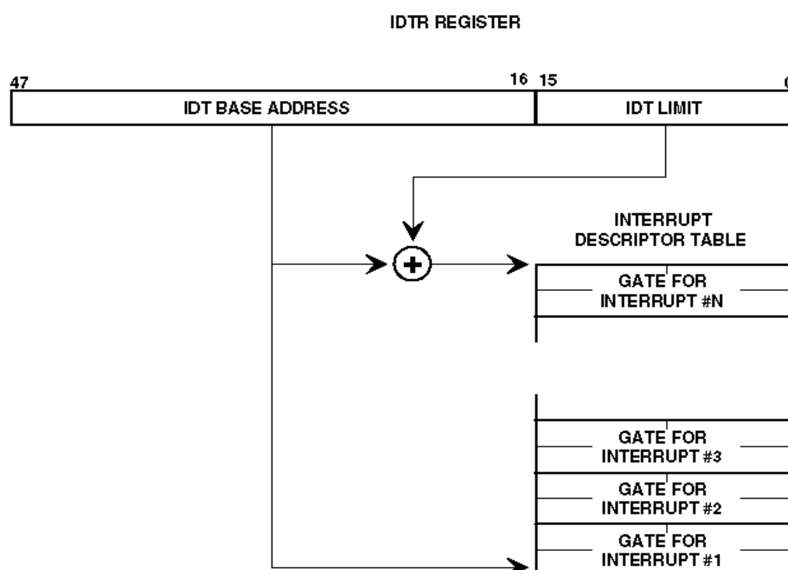


Рис. 5.24. Схема вызова обработчика прерывания.

Начало и предел IDT содержатся в **регистре IDT (RIDT).** Таким образом, таблица шлюзов прерываний может находиться в любом месте физической памяти (в отличие от реального режима, где таблица векторов находится в младших адресах). Более того, ничто не мешает иметь системе несколько IDT. Для переключения на новую IDT всего лишь требуется изменить значение RIDT, для чего имеются привилегированные команды LIDT и SIDT.

Далее приведен формат дескрипторов шлюзов, которые могут содержаться в IDT.

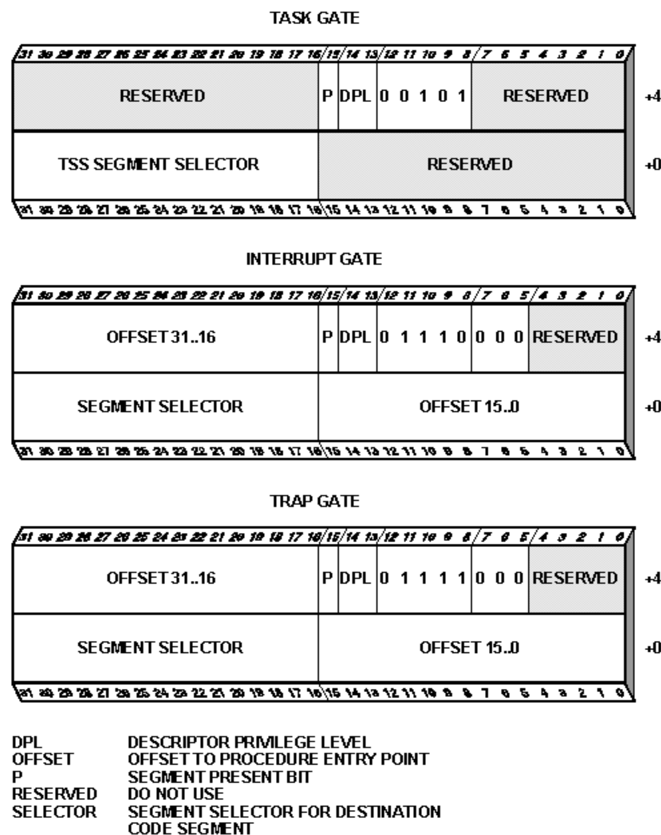


Рис. 5.25. Формат дескрипторов шлюзов.

Подобно тому, как команда CALL может вызвать подпрограмму или задачу, прерывание (или исключение) может вызвать обработчик, оформленный в виде подпрограммы обработки прерывания или в виде задачи (рис. 5.26).

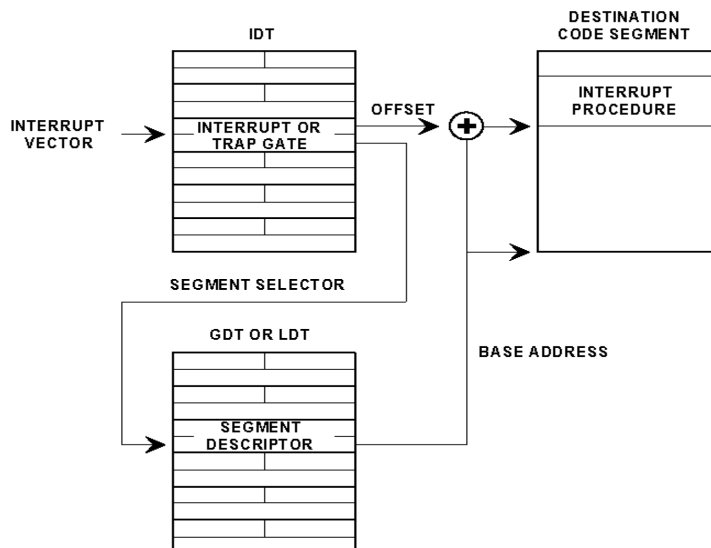


Рис. 5.26. Схема вызова программы через шлюз прерывания.

5.7. Системные регистры процессоров Pentium

На рис. 5.27 показан формат управляющих регистров CR0, CR2, CR3 и CR4. В большинстве систем загрузка управляющих регистров из прикладных программ невозможна (хотя в незащищенных системах такая загрузка разрешается). Прикладные программы имеют возможность считывать эти регистры, например для определения наличия математического сопроцессора. Некоторые разновидности команды MOV позволяют загружать управляющие регистры из регистров общего назначения, и наоборот. Например,

```
MOV  EAX,  CR0
MOV  CR3,  EBX
```

Назначения регистров: CRO, CR4 – управляющие. CR1 – резерв. CR2 – линейный адрес страничной ошибки. CR3 – база каталога таблиц страниц

Регистры CR0, CR4 содержат системные управляющие флаги, которые управляют режимами или указывают на состояние процессора в целом, а не относительно выполнения конкретных задач. Программа не должна пытаться изменить состояние каких-либо битов в зарезервированных позициях. Эти зарезервированные биты всегда должны устанавливаться в то состояние, которое они имели ранее при считывании.

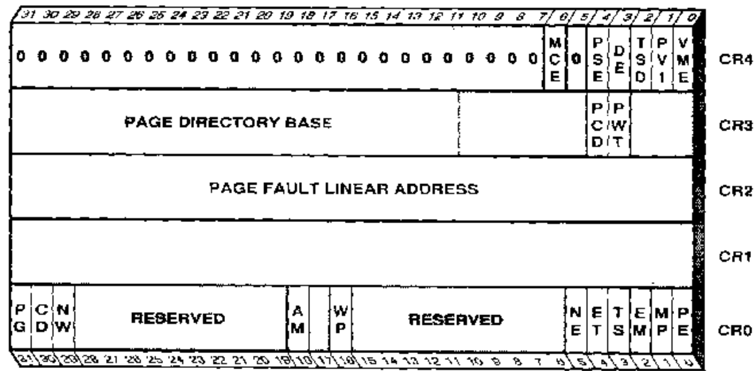


Рис. 5.27. Формат регистров управления

Назначение большинства битов управляющих регистров приведено в нижеследующей тавлице.

CRO	CR4
CD - Cache Disable (запрет внутр. КЭШа) - разрешает внутреннее кеширование, если он очищен NW - Not Write-through - разрешает сквозную запись и циклы аннулирования кеша, если он очищен AM - Alignment Mask WP - Write Protect (защита страниц супервизора) Если этот бит установлен, то он защищает от записи страницы уровня пользователя при доступе в режиме супервизора. Если этот бит очищен, то страницы уровня пользователя, предназначенные только для чтения, могут быть затерты записью процессом супервизора. NE - Numeric Error ET - Extension Type (индикация наличия 387) TS - Task Switched - Процессор устанавливает бит TS при каждом переключении задачи EM - Emulation MP - Monitor coprocessor PE - Protection Enable Установка бита PE разрешает защиту на уровне сегмента.	VMP - Virtual-8086 Mode Extensions PVI - Protected-Mode Virtual Interrupts TSD - Time Stamp Disable (команда RDTSC становится привилег-ной) DE - Debugging Extensions PSE - Page Size Extensions (4Мб страницы) MCE – Machine Check Enable PCD (Запрещение кеширования на уровне страниц, бит 4 регистра CR3) PWT (Прозрачность записи на уровне страниц, бит 3 регистра CR3)

Отладка и отладочные регистры

Процессор обеспечивает расширенные отладочные средства, которые в частности полезны для разработки сложных программных систем, таких как мультизадачные операционные системы. Условия сбоя для этих программных систем могут быть очень сложными и зависимыми от времени.

Отладочные средства процессора предоставляют системному программисту мощный инструмент просмотра динамического состояния процессора.

Отладка поддерживается посредством отладочных регистров. Они содержат адреса точек памяти, называемых контрольными точками (точками останова), которые запускают отладочное программное обеспечение. При выполнении операции с памятью по одному из этих адресов генерируется исключение. Контрольная точка задается для конкретной формы доступа к памяти, например выборки команды или операции записи двойного слова. Отладочные регистры поддерживают как контрольные точки команд, так и контрольные точки данных.

Для других процессоров контрольные точки команд устанавливаются заменой нормальных команд командами контрольных точек. При выполнении команды контрольной точки вызывается отладчик. Однако в случае отладочных регистров процессора 386+ это не обязательно. Исключая необходимость записи в пространство кода, процесс отладки упрощается (не требуется установка отображения сегментов данных в ту же область памяти, что и кодовый сегмент), и контрольные точки могут устанавливаться в ПЗУ – резидентном программном обеспечении. Кроме того, контрольные точки могут быть установлены для чтения и записи данных, что позволяет контролировать состояние данных в режиме реального времени. Доступ к этим регистрам имеют только программы с наивысшим уровнем привилегированности.

В число средств архитектуры, поддерживающих отладку, входят:

- - Отладочные адресные регистры - задают адреса до четырех контрольных точек.
- Отладочный управляющий регистр - задает формы доступа к памяти для контрольных точек.
- Отладочный регистр состояния - сообщает об условиях, которые существовали в момент генерации исключения.

- Бит ловушки в TSS (T-бит) - генерирует отладочное исключение при попытке выполнить переключение задачи на задачу, для которой этот бит был установлен.
- Флаг возобновления (RF) - подавляет возобновление множественных исключений для одной и той же команды.
- Флаг трассировки (TF) - генерирует отладочное исключение после каждого выполнения команды.
- Команда контрольной точки - Вызывает отладчик (генерирует отладочное исключение). Данная команда является альтернативным способом установки контрольных точек в коде. Она особенно полезна, когда желательно иметь более четырех контрольных точек, либо при помещении контрольных точек в исходный код.
- Резервируемый вектор прерывания для исключения контрольной точки - вызывает процедуру или задачу при выполнении команды контрольной точки.

Эти средства позволяют вызывать отладчик как отдельную задачу или процедуру в контексте текущей задачи. Для вызова отладчика могут быть использованы следующие условия:

- Переключение на конкретную задачу.
- Выполнение команды контрольной точки.
- Выполнение любой команды.
- Выполнение команды по заданному адресу.
- Чтение или запись байта, слова или двойного слова по конкретному заданному адресу.
- Запись в байт, слово или двойное слово по заданному адресу.
- Попытка изменить содержимое отладочного регистра.

Рассмотрим назначение отладочных регистров.

DR7: LEN 3 | R/W 3 | LEN 2 | R/W 2 | LEN 1 | R/W 1 | LEN 0 | R/W 0 | 0 0 | GD | 0 0 1 | GE | LE | G3 | L3 | G2 | L2 | G1 | L1 | G0 | L0

DR6: 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 | BT | BS | BD | 1 1 1 1 1 1 1 1 1 | B3 | B2 | B1 | B0

DR5: RESERVED

DR4: RESERVED

DR3: BREAKPOINT 3 LINEAR ADDRESS

DR2: BREAKPOINT 2 LINEAR ADDRESS

DR1: BREAKPOINT 1 LINEAR ADDRESS

DR0: BREAKPOINT 0 LINEAR ADDRESS

Рис. 5.28. Формат регистров отладки

Отладочные регистры (DR0-DR3)

Каждый из этих регистров содержит линейный адрес одной из четырех контрольных точек. При разрешенной подкачке страниц эти адреса транслируются в физические адреса по алгоритму подкачки страниц. Условие каждой контрольной точки определяется содержимым регистра DR7.

Отладочный управляющий регистр (DR7)

Отладочный управляющий регистр задает вид доступа к памяти, связанный с каждой контрольной точкой. Каждый адрес в регистрах DR0 - DR3 соответствует полю R/W0 - R/W3 в регистре DR7. процессор интерпретирует их следующим образом:

00 - Прерывание только при выполнении команды

00 - Прерывание только при записи данных

10 - не определено

11 - прерывание при чтении и записи данных, но не при выборке команды

Поля LEN0 - LEN3 регистра DR7 задат размер позиции памяти, для которой установлены контрольные точки. Может быть задан размер в 1, 2

или 4 байта. Поля длины интерпретируются следующим образом: 00 - доина один байт; 01 - длина два байта;10 - не определено;11 - длина четыре байта.

Младшие восемь битов регистра DR7 (поля от L0 до L3 и G0 до G3) по отдельности разрешают условия контрольных точек в четырех адресах. Существует два уровня их разрешения: локальный (от L0 до L3) и глобальный (от G0 до G3). Локальные биты разрешения автоматически очищаются процессором при каждом переключении задачи, чтобы избежать нежелательных условий контрольных точек в новой задаче. Они используются для установки контрольных точек в одной отдельной задаче. Глобальные биты разрешения при переключении задачи не очищаются. Они используются для разрешения условий контрольной точки, применимых ко всем задачам.

ГЛАВА 6. ВВОД-ВЫВОД

6.1. Подключение периферийных устройств к ЭВМ

Периферийные устройства

Термином периферийное устройство (ПУ) будем называть устройства самой разнообразной природы, которые являются внешними по отношению к вычислительному ядру (процессор + память) и которые по скорости своей работы не согласованы с процессором. (Таким естественным образом согласована по скорости работы с процессором основная память. Для лучшего согласования используются дополнительные структурные элементы, такие как кэш-память, дополнительные локальные шины для доступа к памяти и др. – см. более подробно в предыдущих разделах.). Как правило, периферийные устройства «работают» медленнее процессора, т.е. передают или принимают данные медленнее, чем это в состоянии делать процессор, обмениваясь по магистрали (например, с памятью). Но с точки зрения передачи информации по магистрали в рамках упрощенной схемы ЦВМ, периферийное устройство, подключенное к магистрали ОЗУ, мало отличается от памяти.

Приведем два примера простых периферийных устройств персонального компьютера.

1. **Таймер.** Его основная функция – обеспечить программе возможность дать возможность программе, читая содержимое таймера, получать информацию о истекшем времени, подобно тому, как человек получает ее, поглядывая на часы. Таймер в том или ином виде реализован практически в любой вычислительной системе.

Основной элемент таймера – счетчик, на счетный вход которого постоянно поступает последовательность импульсов от генератора стабильной частоты. Вследствие этого, содержимое счетчика таймера постоянно изменяется во времени. Программа может прочитать содержимое счетчика таймера (скопировать это значение в переменную).

Таймер персонального IBM PC – совместимого компьютера содержит три почти одинаковых канала. Счетчик каждого канала имеет 16 двоичных разрядов, таким образом, программа может считать из канала таймера значение

2. **Последовательный (коммуникационный) интерфейс** (по-другому называемый COM-портом). Это периферийное устройство на самом деле включает два отдельных независимых узла: передатчик и приемник. Байт, записанный программой в регистр данных передатчика, немедленно после записи начинает бит за битом передаваться через последовательный интерфейс.

Требования к организации и решения систем ввода-вывода

Организация взаимодействия и обмена информации в вычислительной(информационной) системе требует решения целого ряда проблем, среди которых выделим следующие:

- **Возможность изменять конфигурацию**: необходимо обеспечить возможность реализации ЭВМ с переменным составом оборудования, в первую очередь, с различным набором устройств ввода-вывода, с тем, чтобы пользователь мог выбирать конфигурацию машины в соответствии с ее назначением, легко добавлять новые устройства и отключать те, в использовании которых он не нуждается;
- **Параллельная работа ЭВМ и ПУ**: для эффективного и высокопроизводительного использования оборудования компьютера следует реализовать параллельную во времени работу процессора над вычислительной частью программы и выполнение периферийными устройствами процедур ввода-вывода;
- **Унификация программирования**: необходимо упростить для пользователя и стандартизовать программирование операций ввода-вывода, обеспечить независимость программирования ввода-вывода от особенностей того или иного периферийного устройства;

- **Синхронизация работы ЭВМ и ПУ:** в ЭВМ должно быть обеспечено автоматическое распознавание и реакция процессора на многообразие ситуаций, возникающих в УВВ (готовность устройства, отсутствие носителя, различные нарушения нормальной работы и др.).

Указанные требования решаются за счет:

- Унифицированных (независящего от типа внешнего устройства) интерфейсов.
- Унифицированных соединений
- Унифицированного программирования

Интерфейс- это совокупность программных и аппаратных средств, предназначенных для передачи информации между компонентами ЭВМ и включающих в себя электронные схемы, линии, шины и сигналы адресов, данных и управления, алгоритмы передачи сигналов и правила интерпретации сигналов устройствами. Интерфейсы характеризуются следующими параметрами:

- пропускная способность - количество информации, которая может быть передана через интерфейс в единицу времени;
- максимальная частота передачи информационных сигналов через интерфейс;
- максимально допустимое расстояние между соединяемыми устройствами;
- общее число проводов (линий) в интерфейсе;
- информационная ширина интерфейса - число бит или байт данных, передаваемых параллельно через интерфейс.

Основным назначением интерфейса является унификация внутрисистемных и межсистемных связей и устройств сопряжения с целью эффективной реализации прогрессивных методов проектирования функциональных элементов вычислительных систем.

Основными архитектурами соединений устройств являются:

Каскадное соединение. Информация передается только в одном направлении от предыдущего функционального элемента (ФЭ) к последующему. Если сообщение не адресовано данному ФЭ, она транслируется без изменений. Такая структура характеризуется малой скоростью обмена, но имеет минимальное число шин.

Радиальное соединение. Центральный ФЭ связывается с каждым ФЭ. Такая система обладает наибольшей надежностью и производительностью, но ограничена по числу ФЭ, в ней невозможен непосредственный обмен между ФЭ.

Магистрально-модульное соединение. Все устройства, составляющие компьютер, включая и микропроцессор, организуются в виде модулей, которые соединяются между собой общей магистралью. Обмен информацией по магистрали удовлетворяет требованиям некоторого общего интерфейса, установленного для магистрали данного типа. Каждый модуль подключается к магистрали посредством специальных интерфейсных схем (Иi).

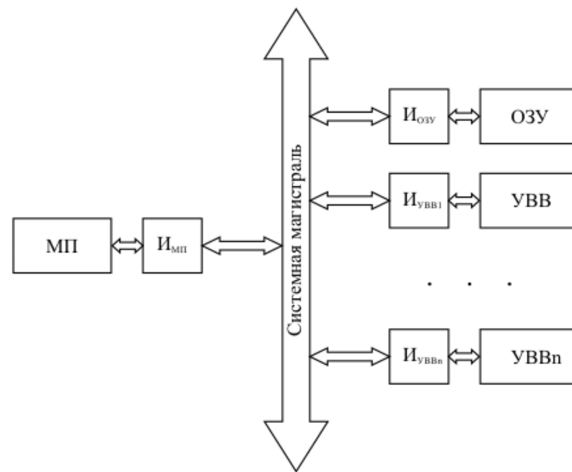


Рис. 6.1. Схема магистрально-модульного соединения.

На интерфейсные схемы модулей возлагаются следующие задачи: обеспечение функциональной и электрической совместимости сигналов и протоколов обмена модуля и системной магистрали; преобразование внутреннего формата данных модуля в формат данных системной

магистрали и обратно; обеспечение восприятия единых команд обмена информацией и преобразование их в последовательность внутренних управляющих сигналов. Эти интерфейсные схемы могут быть достаточно сложными и по своим возможностям соответствовать универсальным микропроцессорам. Такие схемы принято называть контроллерами.

Унификация программирования достигается за счет унифицированных форматов команд в ЭВМ и унифицированных форматов регистров ПУ. На уровне операционной системы унификация достигается путем использования драйверов внешних устройств.

Модель внешнего устройства для программиста.

Каждое периферийное устройство имеет в своем составе один или несколько регистров, которые можно прочитать или записать командами программы. Сложные периферийные устройства могут содержать несколько десятков регистров. Таким образом, общее количество регистров периферийных устройств в системе может быть значительным.

Словосочетанием модель периферийного устройства для программиста будем называть набор регистров в периферийном устройстве, которые можно прочитать-записать программно.

Простейшая модель для программиста внешнего устройства может содержать весьма мало регистров: - **регистр состояния (статуса), регистр управления, регистр (буфер) данных**.

Применяют два основных способа доступа к регистрам ПУ:

1. **Произвольной адресации регистров.**
2. **С индексацией регистров (SB).**

В первом случае процессор по команде может записать/считать данные в любой регистр ПУ. Если ПУ имеет большое количество регистров, например более 100 как в sound blaster, то используют второй способ. В этом случае ПУ имеет два регистра: индексный регистр и регистр данных. При записи в

индексный регистр заносим номер регистра ПУ, в который хотим записать, затем в регистр данных заносим записываемое число.

Адресация регистров внешнего устройства:

- изолированный ввод-вывод (порты)
- отображение на память.

В некоторых процессорах доступ к регистрам периферийных устройств осуществляется аналогично доступу к ячейкам памяти. Каждому регистру присвоен адрес в адресном пространстве памяти. В этом случае для обращения к регистрам ПУ можно использовать те же команды, что и для доступа к ячейкам памяти. Такая организация носит название «**ввод-вывод, отображаемый на память**». Обычно разработчики вычислительной системы выделяют для адресации регистров ПУ какой-либо фиксированный диапазон адресов.

В других процессорах регистры ПУ имеют свою систему адресации, никак не связанную с адресацией ячеек памяти. Для обращения к регистрам ПУ в системе команд имеются специальные команды ввода-вывода. Такая организация обмена с ПУ носит название «**изолированный ввод-вывод**». Для обозначения программно-доступных регистров периферийных устройств в компьютерной литературе используют термин «порты ввода-вывода».

Пример 1

Нулевой канал таймера имеет порт данных с адресом 40h и порт управления с адресом 43h

Пример 2

Приемник последовательного интерфейса COM1 имеет порт данных с адресом 0378h и порт состояния (статуса) с адресом 03FDh. Младший бит в порте состояния (флаг окончания приема) устанавливается в 1, если приемник принял извне байт.

Оба вида адресации аппаратно поддержаны со стороны процессора. В первом случае в адресном пространстве процессора выделяется область

адресов для ПУ. Адреса из этой области транслируются напрямую, без использования схемы преобразования (логический адрес равен физическому). В процессоре Motorola 68040 этот механизм называется - прозрачная трансляция адресов. Для указания выделенной области регистров ПУ используются специальные регистры. В этих регистрах дополнительно имеются поля, определяющие разрешение/запрещение КЭШ-ния, тип доступа r/w и др. На рис.6.2. представлена схема страничной трансляции.

Регистры прозрачной трансляции TTO и TT1 (рис.6.3.) определяют блоки адресного пространства, которые транслируются прозрачно (прямо): в этих блоках логические адреса являются физическим.

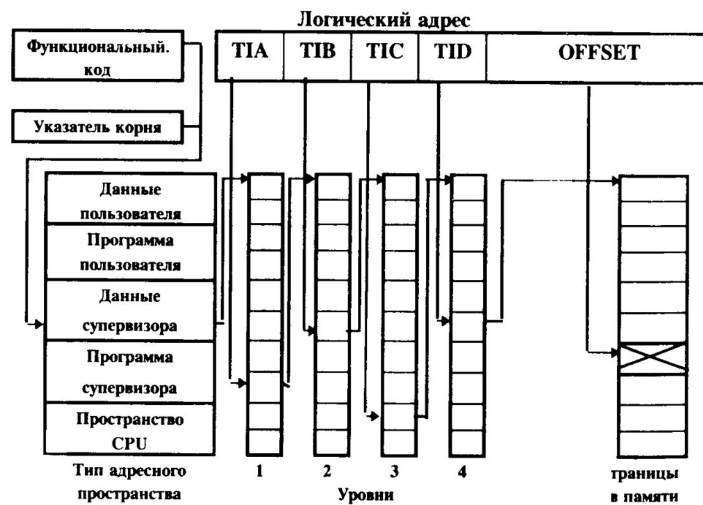


Рис. 6.2. Схема страничной трансляции процессора MC68040

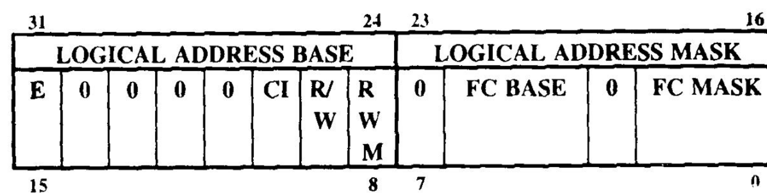


Рис. 6.3. Формат регистров прозрачной трансляции.

Значения полей регистров TTO и TT1:

LOGICAL ADDRESS BASE - значение A31-A24, определяют блок прозрачной трансляции;

LOGICAL ADDRESS MASK - игнорируемые разряды A31-A24;

E (Enable) - разрешение; определяет возможность прозрачной трансляции (E=1 разрешена).

Cl (Cashe Inhibit) - кэш запрещен; определяет возможность использования кэша: C1=0 используется;

R/W - чтение/запись: определяет тип доступа в транслируемом блоке: R/W=0 запись; R/W=1 чтение.

RWM (Read/Write Mask) - маска чтения - записи; определяет достоверность поля R/W (RWM=O поле используется)

FC BASE (Function Code Base) - значение функционального кода блока прозрачной трансляции;

FC MASK (Function Code Mask) - игнорируемые биты функциональною кода.

При изолированном вводе-выводе процессор должен иметь: специальные команды работы с ПУ, специальные сигналы на шине, различающие обращение к памяти и к ПУ.

В процессорах семейства x86 использован *изолированный ввод-вывод*. На шине процессора имеются специальные сигналы. При обращении к памяти вырабатываются стробы MEMR, MEMW , а к ПУ стробы I/OR, I/OW. Схема подключения приведена на рис 6.4.

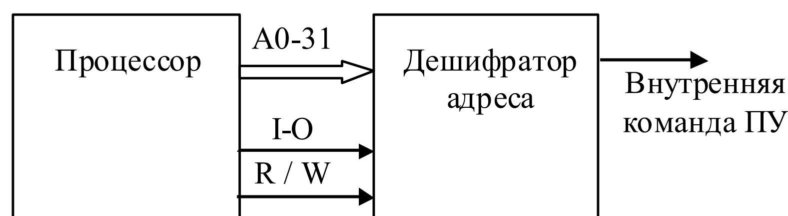


Рис. 6.4. Схема подключения ПУ

В составе системы команд имеются следующие две команды ввода-вывода:

1) **in** - ввод 1/2/4-байтового элемента данных из порта в регистр процессора **al/ax/eax**

2) **out** – вывод 1/2/4-байтового элемента данных из регистра

al/ax/eax в порт.

Номер порта операнда-источника в команде **in** или операнда-приемника в команде можно задать двумя способами – непосредственной адресацией или косвенно-регистровой с использованием (только) регистра dx.

Примеры:

```
in    al,040h    ;  ввести байт в al из порта 40h
mov   dx,0378h   ;  записать номер порта 0378h в регистр dx
in    al,dx      ;  ввести байт в al из порта 0378h
mov   al,049h    ;  записать байт 49h в регистр al
out   dx,al      ;  вывести байт из al в порт, номер которого содержится в dx
```

Кроме того, в системе команд есть две строковые (цепочечные) команды ввода-вывода:

1) **insb / insw / insd** – ввод из порта цепочки (строки) байтов/слов/двойных слов в последовательные адреса памяти

2) **outsb / outsw /outsd** - вывод цепочки (строки) байтов/слов/двойных слов из последовательных адресов памяти в порт

Схема подключения внешнего устройства к ISA шине

ОПИСАНИЕ СИГНАЛОВ ШИНЫ

OSC Генератор: Высокочастотные импульсы с периодом 70 нс(14.31818 МГц).

CLOCK Системная частота (от процессора)

RESET DRV Этот сигнал используется для сброса или инициализации системной логики при включении питания или при низком уровне напряжения на линии. Он синхрони- зирован с задним фронтом CLOCK и имеет активный вы- сокий уровень. (от процессора)

SA0-SA19 Биты адреса с 0 по 19: Эти линии используются для адресации памяти и устройств ввода/вывода в системе. 20 адресных линий позволяют адресовать до 1 Мбайта памяти. SA0 - это младший значащий разряд, а SA19 - старший значащий разряд. Сигналы генерируются либо процессором, либо устройством ПДП. Они имеют актив-ный высокий уровень. (от процессора)

SD0-SD15. Биты данных с 0 по 15 эти сигналы служат для передачи данных между процессором, памятью и внешними устройствами. D0 - это младший разряд, а D15- старший. Они имеют активный высокий уровень. (двунаправлен)

BALE Разрешение селекции адреса: этот сигнал вырабатывается контроллером шины 82288 и используется на системной плате для защелкивания верного значения адреса от процессора. Он доступен на канале ввода/ вывода как индикатор того, что значение адреса на магистрали верное(когда используется вместе с AEN). Адрес защелкивается по заднему фронту сигнала. (двунаправлен)

I/O CH CK Проверка канала: этот сигнал обеспечивает процессор информацией об ошибках четности памяти или внешних устройств в канале. Когда этот сигнал пере- ходит в низкое состояние, регистрируется ошибка четности. (в процессор)

I/O CHRDY Готовность канала: этот сигнал, обычно высокий, переводится в низкое состояние памятью или внешним устройством для продления цикла обращения. Он дает возможность присоединять к системе устройства с низким быстродействием с минимальными затратами. Любое медленное устройство, используя этот сигнал, должно держать его в низком состоянии до тех пор, пока оно не проведет операцию распознавания адреса и не выполнит команду чтения или записи. Однако, этот сигнал не должен оставаться в низком состоянии дольше 10 циклов синхронизации системы. Цикл обращения к памяти или внешнему устройству увеличивается на целое число циклов синхронизации. (в процессор)

IRQ3- IRQ15 Запрос на прерывание 3-15 эти сигналы используются для сообщения процессору, что устройство требует обслуживания. Они имеют разный приоритет. IRQ3 - с наивысшим приоритетом, а IRQ15- с низшим. Запрос на прерывание вырабатывается при переходе сигнала из низкого состояния в высокое и удержании его до распознавания процессором.(в процессор)

IOR Команда чтения из устройства: данный сигнал указывает внешнему устройству на необходимость выставить свои данные на шину данных. Он может вырабатываться процессором или устройством ПДП. Активный уровень сигнала - низкий. (от процессора)

IOW Команда записи в устройство: этот сигнал сообщает устройству о необходимости ввода данных с магистра ли. Он может вырабатываться как процессором, так и внешним устройством. Активный уровень сигнала - низкий (от процессора)

SMEMR Команда чтения памяти из пространства 1 Мбайт Этот сигнал указывает памяти, что она должна выставить свои данные на шину . Он может

вырабатываться как процессором, так и устройством ПДП. Активный уровень сигнала - низкий (от процессора).

SMEMW Команда записи в память из пространства 1 Мбайт. данный сигнал указывает памяти на необходимость прочитать данные, выставленные на шину данных. Он может вырабатываться как процессором, так и устройством ПДП . Активный уровень сигнала - низкий (от процессора).

MEMR Команда чтения памяти: Этот сигнал указывает памяти, что она должна выставить свои данные на шину. Он может вырабатываться как процессором, так и устройством ПДП. Активный уровень сигнала - низкий (от процессора).

MEMW Команда записи в память: данный сигнал указывает памяти на необходимость прочитать данные, выстав- ленные на шину данных. Он может вырабатываться как процессором, так и устройством ПДП. Активный уровень сигнала - низкий (от процессора).

DRQ0-DRQ3 DRQ5-DRQ7Запрос ПДП 0-7: данные сигналы являются асинхронными запросами канала периферийными устройствами для выполнения операций ПДП. Они имеют различный приоритет. DRQ7 - низший, а DRQ0 - высший. Запрос генерируется переводом соответствующего сигнала в активное(высокое) состояние. Сигнал должен удерживаться в высоком состоянии до тех пор, пока не станет активной соответствующая линия DACK (в процессор).

DACK0 - DACK7 Подтверждение ПДП 0-7: эти сигналы используются для ответа на соответствующие запросы ПДП(0-7) и Они имеют низкий активный уровень (от процессора).

AEN Разрешение адреса: Данный сигнал используется для отключения процессора и других устройств от канала для проведения цикла ПДП. Когда этот сигнал активен (высокий), контроллер ПДП получает шину адреса, шину данных, а также линии чтения и записи (от процессора).

T/C Счетчик завершения: на этой линии появляется импульс когда достигнуто состояние счетчика завершения какого-либо устройства ПДП (от процессора).

REFRESH Запрос на регенерацию динамической памяти (от процессора).

LA17-LA23 Незащелкиваемые адресные линии А17-А23 (двунаправлен).

SBHE Показывает что старший байт данных находится на старшей шине данных SD8-SD15 (двунаправлен).

0WS Сигнал показывает процессору что текущий цикл шины может быть выполнен без дополнительных тактов ожидания (в процессор).

MASTER Сигнал перехвата управления внешним устройством системной магистрали (в процессор).

-MEM CS16 Сигнал подтверждающий то , что процессор может работать с этой памятью 16-разрядными словами без побайтовой распаковки(в процессор).

-IO CS16 Аналогично, только с устройствами ввода/вывода (в процессор).

Помимо описанных сигналов в канале ввода/вывода имеется ряд линий питания для устройств, подключенных к каналу.

Временные диаграммы.

Чтение из порта ввода

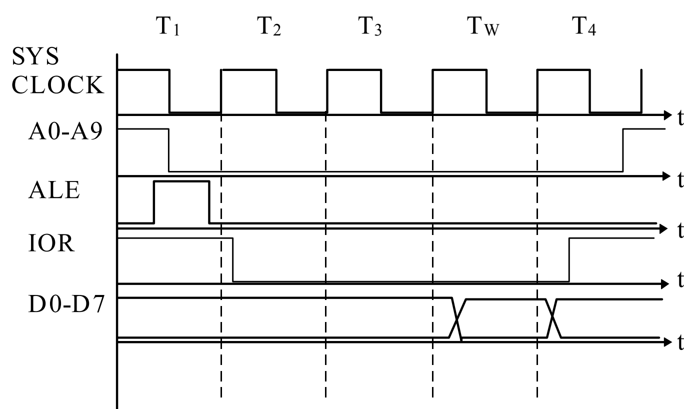


Рис. 6.5. Временная диаграмма чтения из порта

Временная диаграмма операции чтения из порта (in port) занимает четыре такта опорной частоты и один такт ожидания. Само чтение данных происходит в начале четвертого периода. При необходимости удлинения времени чтения подают сигнал I/O CH RDY готовности канала, который увеличивает такт ожидания. Максимальное время ожидания определяется необходимостью регенерации памяти.

Запись в порт вывода

Временная диаграмма операции записи в порт (out port) занимает тоже время, что и чтения из порта. Данные на шине находятся в течение всей длительности синхросигнала IOW.

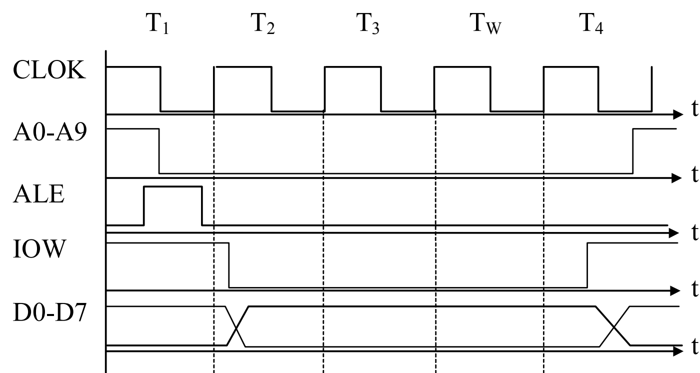


Рис. 6.6. Временная диаграмма записи в порт

В порт можно записывать и соответственно считывать как байт, так и слово (два байта). Второй байт может появиться на старших разрядах шины данных или вслед первого на младших разрядах. В первом случае внешнее устройство должно сформировать сигнал IO CS16, а процессор записать в (считать из) четный адрес порта. Если сигнал IO CS16 отсутствует, то запись первого и второго байта происходит в последовательные адреса портов, причем увеличение адреса порта на шине адреса происходит автоматически. При обращении к нечетным адресам считывается и записывается младший байт.

Структурная схема модуля на базе шины ISA.

Подробная структурная схема модуля Plug in Data Acquisition Board приведена на рис.6.7. На рисунке показаны не только составляющие модуль элементы, но и перечислены микросхемы для их реализации. В модуле можно выделить следующие элементы: буфер системной шины, схему декодирования адреса порта, параллельный ввод-вывод КР580 ВВ55 (Intel 8255), микросхемы АЦП и ЦАП.

Буферизация содержимого системной шины.

Необходимость буферизации определяется следующими причинами:
- Защита канала от больших напряжений, замыканий на землю.
- Защита от большой нагрузки при подключении к каналу большого числа потребителей (более 5), которые увеличивают емкостную нагрузку и вследствие чего изменяют временные параметры.

В качестве буфера можно использовать шинные приемо-передатчики, подключенные к соответствующим линиям канала. В модуле используются следующие микросхемы: однонаправленные приемо-передатчики - 74LS245 (КР580 ВА 86), двунаправленные приемо-передатчики - 74LS244 (КР580 ИР 82). Последние применены для буферизации шины данных и включаются по сигналу AEN, а направление передачи определяет сигнал IOW.

Схема декодирования адреса порта.

Элемент декодирования адреса порта предназначен для определения порта модулем. Номер порта в модуле задан аппаратно и при совпадении в процессе декодирования этого номера с пришедшим (по шине адреса) модуль откликается на команду ввода-вывода (происходят те или иные действия).

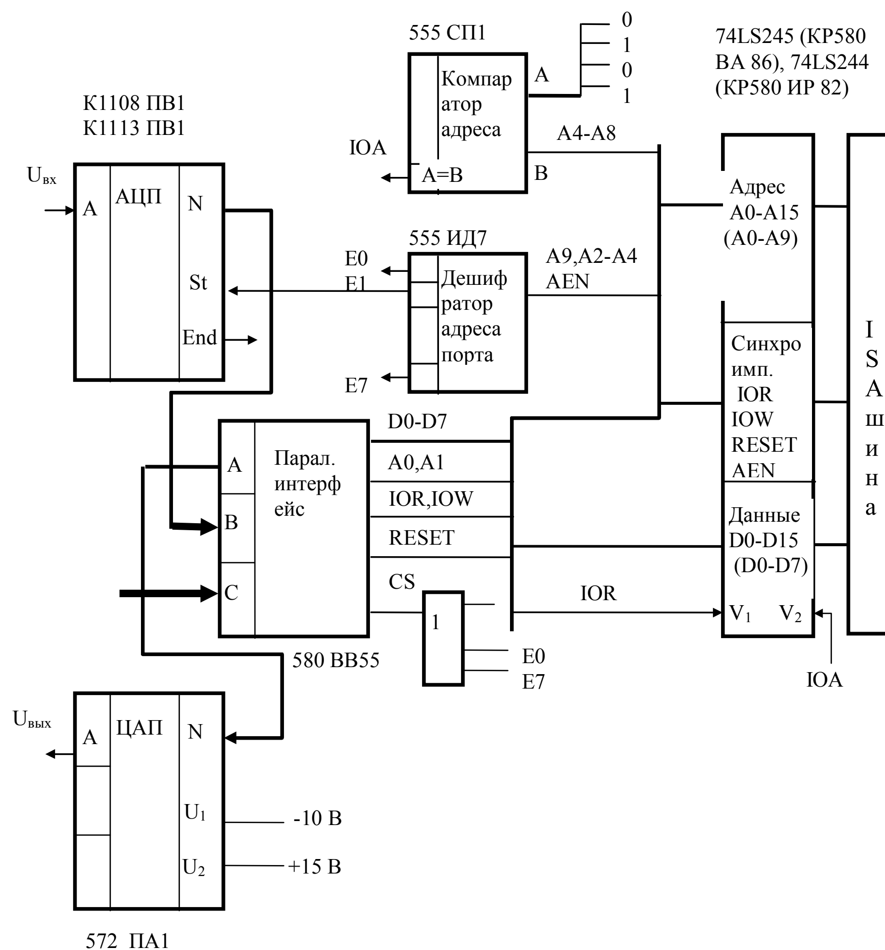


Рис. 6.7. Схема подключения АЦП к шине.

На рис.6.7. представлена двухкаскадная схема декодирования адреса. Она включает цифровой компаратор (555 СП1) и дешифратор (555 ИД7). На входе А компаратора жестко задан код 0101, а код адреса дешифратора определяется номером используемого выхода. Разряд А9 включает дешифратор. Младшие разряды А0, А1 задают номер канала параллельного интерфейса. Адрес порта можно определить в соответствии с приведенной ниже схемой.

Схема формирования адреса портов

9	8	7	6	5	4	3	2	1	0
1	0	1	0	1	0	0	0		
Задано	Компаратор				Дешифратор			Парал. интерф.	

Схема интерфейса параллельного ввода-вывода.

Схема предназначена для подключения нескольких источников (приемников) данных к шине, а также для организации двунаправленного и стробируемого ввода-вывода. Таблица сигналов управления микросхемой (Intel 8255, 580 ВВ55):

A1	A2	RD	WR	CS	Функция
0	0	0	1	0	Из порта А сигнал поступает на шину
0	1	0	1	0	Из порта В сигнал поступает на шину
1	0	0	1	0	Из порта С сигнал поступает на шину
0	0	1	0	0	С шины сигнал поступает в порт А
0	1	1	0	0	С шины сигнал поступает в порт В
1	0	1	0	0	С шины сигнал поступает в порт С
1	1	1	0	0	С шины поступает управляющее слово

Микросхема параллельного интерфейса поддерживает несколько режимов работы по каждому выходу (А, В, С), выбор режима проводится подачей управляющего слова. Ниже приведен формат управляющего слова.

Разряд	Значение	Функция
0	1	Ввод из порта С 2 часть
0	0	Вывод в порт С 2 часть
1	1	Ввод из порта В
1	0	Вывод в порт В
2	1	Стробируемые операции в порту В
2	0	Основные операции в порту В

3	1	Ввод из порта С
3	0	Вывод в порт С
4	1	Ввод из порта А
4	0	Вывод в порт А
5,6	00	Основные операции в порту А
5,6	01	Стробируемые операции в порту А
5,6	10	Двунаправленные операции в порту А
7	1	Флаг выбора режима

Для того, чтобы порт А работал на вывод, а порты В,С на ввод необходимо записать управляющее слово:

7	6	5	4	3	2	1	0
1	0	0	0	1	0	1	1

В результате при работе со схемой рис.6.7. необходимо первоначально запрограммировать конфигурацию внутренних портов, а лишь затем считывать из АЦП или выводить в ЦАП информацию. Команды чтения и вывода аналогового сигнала будут иметь вид:

Port [$2A3]:=$8B; A:= Port[$2A1]; Port[$2A0]:=A;

6.2. Синхронизация выполнения программы с внешними процессами

Существует много ситуаций когда требуется синхронизация выполнения программы с внешним событием. Такие ситуации возникают прежде всего при обмене данными с периферийными устройствами, в частности с *устройствами ввода-вывода*.

Примеры:

- Прежде, чем прочитать принятый байт из регистра данных COM-порта, программа должна убедиться, что процесс приема байта завершился.
- При выводе данных на принтер программа печати должна проверить, что предыдущая порция данных напечатана (печать происходит довольно долго), и принтер способен принять новую порцию.
- При считывании данных с диска сектор аппаратно (гораздо медленнее, чем способен оттуда читать процессор) читается в буфер

контроллера, программа должна "узнать", что буфер заполнен, прежде чем читать оттуда.

В многозадачной операционной среде одна из выполняемых программ может ожидать данных, подготавливаемых другой программой. В этом случае требуется синхронизация продолжения первой программы с завершением второй. Хотя обе программы работают на одном процессоре под одной ОС, они могут ничего не «знать» друг о друге. В этом случае событие, состоящее в завершении программы2 ничем не отличается для программы1 от внешнего события.

Возможны другие (обратные) ситуации, когда наоборот, при наступлении события нельзя продолжать выполнение текущей программы. Примеры:

- Произошло деление на нуль, и следующую операцию, которая должна использовать результат деления, выполнить невозможно.
- Пользователь нажал Ctrl-Break, и выполнять программу дальше не следует.

Общим в приведенных примерах является то, что в определенные моменты программа должна прореагировать на некоторое событие, причем во всех приведенных примерах реакция есть передача управления по условию наступления внешнего события.

Рассмотрим далее, каким образом программа может «узнать» о наступлении события.

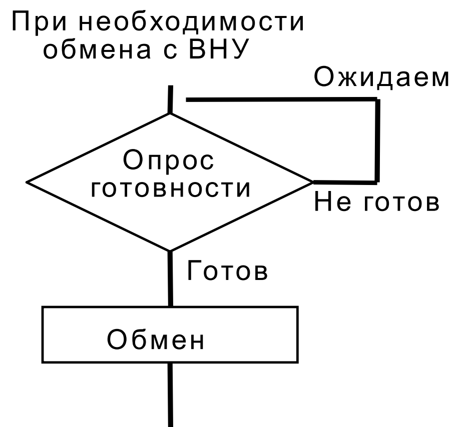
Способы синхронизации: программный опрос состояния versus прерывания

Перечислим способы синхронизации:

1. Программный безусловный обмен.
2. Программный опрос готовности
3. Прерывание
4. Прямой доступ в память

5. Удлинение канального цикла ввода-вывода.

Простейший способ - программный опрос готовности (polling).



Опрос готовности состоит чаще всего в анализе состояния определенных битов в регистре статуса (состояния) – при достижении состояния готовности к обмену Вну устанавливает этот(эти) бит(ы) в определенное состояние. Обмен с ВнУ состоит в чтении или записи в регистр данных.

Так, если приемник последовательного интерфейса принял извне байт, автоматически устанавливается флаг готовности – младший байт в порте 03FDh. Фрагмент программы, ожидающий приема байта, может выглядеть так:

```
    mov  dx, 03FDh     ; адрес регистра статуса – в dx
L1: in   al, dx        ; читаем содержимое регистра статуса
    test al, 01h       ; проверяем в прочитанном значении младший байт
    jz   L1            ; если он=0 – повторяем чтение и проверку
    mov  dx, 03F8h     ; в противном случае – заносим в dx адрес регистра данных
    mov  al, dx        ; и читаем принятый байт
```

Недостаток: при ожидании готовности нерационально расходуется время процессора. Можно опрашивать не постоянно, а периодически, при этом появляется задержка реакции на готовность. Поэтому был разработан механизм **прерывания**.

6.3. Прерывания

Термин *прерывание* в русскоязычной компьютерной литературе многозначен и употребляется для обозначения трех различных вещей. В англоязычной литературе используются три разных термина:

hardware interrupt - *аппаратное прерывание*

exception - *исключение*, прерывание по исключительной (экстраординарной) внутренней ситуации

software interrupt - *программное прерывание*.

Прерывание аналогично обращению к подпрограмме, только это обращение происходит не по команде вызова подпрограммы, а как следствие наступления некоторого события. Основная особенность (в отличие от использования вызова call) рассматриваемой ситуации состоит в том, что момент наступления этого события может быть никак не связан с текущим состоянием программы.

Будем далее называть подпрограмму, обрабатывающую факт наступления события *обработчиком прерывания (interrupt handler* или *exception handler).*

Термин *обработка прерывания* может использоваться для обозначений двух различных вещей: 1) действия, которые автоматически выполняет процессор при возникновении запроса, они реализованы аппаратно, и 2) действия, которые выполняет программа-обработчик прерывания (handler). Мы будем использовать термин *обработка прерывания* только во втором смысле, а для первой группы действий будем использовать термин *вход в прерывание*.

Аппаратные прерывания и проблема приоритетов.

Термином аппаратное прерывание называют ситуацию, когда источник прерывания - устройство, внешнее по отношению к вычислительному ядру, а событие, вызвавшее прерывание, не синхронизировано с процессом выполнения программы. Иногда источник запроса может быть интегрирован

в одну микросхему с процессором (например таймер в однокристальном микроконтроллере), но логически процессор и таймер независимы (и даже могут тактироваться от разных генераторов).

Типичный пример использования внешнего прерывания: по сигналу готовности принтера (говорящей о том, что предыдущая порция текста напечатана) выдать на него следующую порцию текстовой информации.

Используется две основные схемы подключения источников запросов к процессору: **радиальная и магистральная.**

При магистральной схеме запросы от всех источников поступают на одну и ту же линию запроса. После приема запроса процессор проводит с устройствами обмен сигналами с целью выяснить источник запроса, при этом усложняются схемы обслуживания прерываний и увеличивается время реакции на запрос (что в некоторых случаях оказывается нежелательно).

При радиальной схеме каждый источник запроса использует индивидуальный вход запроса, в результате чего процессор может определить источник запроса и осуществить переход на соответствующую подпрограму обработки.

В том случае, когда несколько запросов могут требуют обработки одновременно, может оказаться, что некоторые из запросов более важны и должны быть обработаны в первую очередь. Для этого система управления прерываниями должна давать программисту возможность определять приоритеты, т.е. порядок реакции на запросы

Механизм прерывания, его общие свойства (как это обычно делается).

При возникновении события, на которое надо отреагировать (запроса на прерывание), обычно процессор автоматически (без участия программиста) выполняет следующие действия:

1. Заканчивается выполнение текущей команды (иногда прерывается, если команда длинная, а иногда выполняется еще одна или несколько команд).

2. Анализируется, разрешено ли прерывание. (Если нет, то переход к выполнению следующей команды).

3. Если запросов несколько, принимается решение, который запрос обслуживать (разрешение приоритета priority resolving).

Если система запросов радиальная или источник запроса - внутреннее событие процессора (exception), то переход к .п.6.

4. При магистральной схеме запросов:

Процессор передает источникам запросов подтверждение приема запроса (этот сигнал должен достигнуть только того источника запроса, который имеет наивысший приоритет).

5. Источник запроса передает процессору идентифицирующую его информацию (каждый источник запроса может иметь собственную программу обработки, и процессор должен узнать, какой обработчик использовать).

6. Процессор сохраняет информацию о текущем контексте (текущий вектор состояния - почти всегда неполностью).

7. Адрес перехода на программу обработки прерывания хранится в определенной для каждого источника запроса прерывания области памяти, называемой вектором прерывания. Процессор загружает начальный адрес программы обработки прерывания из вектора прерывания в счетчика команд.

8. Для возврата из прерывания в системе команд обычно есть специальная команда "возврат из прерывания" (мнемоника iret или reti или rti). По этой команде восстанавливается контекст прерванной программы в том объеме, в котором он быь сохранен при входе в перывание (п.6). Эта команда должна быть последней исполняемой командой обработчика.

Приведенное описание соответствует обработке внешнего аппаратного прерывания. В случае, если причина прерывания - внутреннее событие процессора (исключительная ситуация - exception), то этапы 4 и 5 отсутствуют, как в случае радиальных прерываний.

Опишем перечисленные этапы более детально.

1. Внешний запрос прерывания приходит асинхронно (без какой-либо привязки во времени) по отношению к выполняемому потоку команд. Произвести переход на обработчик в большинстве процессоров можно только в промежутке между выполнением соседних команд, поэтому обычно выполнение текущей команды заканчивается. В некоторых процессорах, если время выполнения текущей команды велико, ее выполнение прерывается, а после выхода из прерывания команда начинает выполняться сначала (это, например, типично для команд плавающей точки в процессорах, где они реализованы микропрограммно). В процессорах x86 прерываются команды строковых операций, но после выхода из прерывания они не начинаются сначала, а продолжают работу с того места, где были прерваны. Некоторые комбинации команд используются совместно, и прерывание между ними может привести к фатальным результатам. Такова в x86 пара команд, переустанавливающих положение стека: для этого надо поменять содержимое регистра сегмента стека и затем содержимое указателя стека. Поскольку в x86 для сохранения контекста используется стек, то прерывание между указанными двумя командами вызовет сохранение контекста (якобы в стеке) в неверном месте памяти. Для исключения такой ситуации команда загрузки в регистр сегмента стека автоматически запрещает прерывание до окончания следующей команды.

2. В некоторых ситуациях прерывание недопустимо (например, при выполнении участков программы, критичных ко времени выполнения), поэтому во всех процессорах имеется возможность запретить прерывания (по крайней мере некоторые). Возникшие в этот период запросы могут быть

потеряны, либо могут ждать обслуживания, которое произойдет, когда прерывания будут разрешены - это зависит от устройства конкретного процессора и от свойств входов запроса прерываний. В некоторых процессорах можно программно управлять свойствами входа: запоминается или теряется запрос, который приходит в период, когда прерывания процессору запрещены. В системе команд обычно есть команды, запрещающие и разрешающие прерывание. Кроме того, внешние устройства-источники запросов нередко позволяют программно разрешить/запретить прерывание от данного устройства (прерывания от других устройств при этом будут обрабатываться).

3. При нескольких источниках запроса возможна ситуация, когда в процессор поступают одновременно несколько запросов. Это вовсе не значит, что запросы одновременно возникают, это крайне маловероятно. Однако нередка ситуация, когда в течение некоторого времени прерывание запрещено, и в этот период (не одновременно) возникает несколько запросов. Когда выполнится команда "разрешить прерывание", запросы поступают в процессор равноправно. Надо каким-нибудь способом определить порядок, в котором запросы будут обрабатываться (их приоритеты). Используется несколько разных способов:

а) приоритеты фиксированы и не могут быть изменены;

б) приоритеты определяются электрической схемой и для их изменения надо произвести переключения в схеме;

в) приоритеты изменяются циклически: последнему обслуженному устройству присваивается самый низкий приоритет;

г) приоритеты могут быть программно заданы в произвольной комбинации;

...

... и множество других и вариантов...

4. Далее происходит обмен сигналами между источником запроса и процессором. Сначала процессор посылает сигнал **подтверждения приема**

запроса. В частном случае сигнал подтверждения распространяется только до устройства, наиболее близкого к процессору из всех, выставивших запрос прерывания.

5. Используется несколько способов идентификации источника прерывания. В случае магистральной архитектуры устройство обычно передает по магистрали информацию о себе процессору. Эта информация обычно содержит условный код (номер) источника запроса, по которому процессор способен определить адрес памяти, содержащий информацию о местонахождении в памяти обработчика. В качестве условного кода может использоваться адрес *вектора прерывания* или стартовый адрес обработчика или даже полный код команды call перехода на обработчик (так было сделано в процессоре i8080). Термин *вектор прерывания* используют в двух разных значениях: а) контекст (вектор состояния) обработчика, автоматически загружаемый при выполнении прерывания, б) участок памяти, где хранится этот контекст. Обычно для хранения контекстов обработчиков разработчики процессора выделяют в адресном пространстве специальную *область векторов прерывания*. Для некоторых из прерываний разработчики процессора могут предопределить положение векторов и значения адресов перехода, в то время как другие программист может задавать по своему усмотрению программно.

6. Сохранение текущего контекста процессор чаще всего делает в стеке. В простейшем случае сохраняется содержимое счетчика команд и содержимое регистра состояния. Во многих процессорах в регистре состояния отдельные биты имеют отношение к управлению прерываниями, и автоматическое сохранение слова состояния предоставляет ряд дополнительных возможностей, а иногда является просто необходимым (см. например далее "пошаговую отладку" для процессоров x86).

7. Загрузка контекста обработчика (вектора прерывания). Результат этого действия - переход на первую команду обработчика. Используются различные способы указания того, в каком адресе памяти расположена эта первая команда. Чаще всего адрес обработчика содержится в соответствующем векторе прерывания. Существенным является вопрос о том, разрешено ли прерывание после перехода на обработчик. В большинстве процессоров при входе в прерывание повторное (вложенное) прерывание автоматически запрещается. Обработчик может разрешить прерывание соответствующей командой (например в процессорах x86 это команда **sti**). В этом случае возможно "вложенное" прерывание, в том числе и от этого же источника, но для этого обработчик прерывания должен быть *реентерабельным*, т.е. допускать рекурсивный вызов. Кроме того, это может привести к нарушению приоритетов – менее приоритетное событие будет обслужено раньше, чем более приоритетное.

8. Возврат из прерывания восстанавливает контекст прерванной программы. Существенно, что восстанавливается состояние "разрешено/запрещено прерывание" - после возврата из прерывания возможно следующее прерывание, если есть запрос.

Время реакции на запрос прерывания. Оно определяется двумя независимыми факторами.

1) Время входа в прерывание для одиночного запроса.

2) Интенсивность потока запросов - при значительном количестве источников и при интенсивном потоке запросов часто оказывается, что очередной запрос приходит еще до того, как обслужен предыдущий. В этом случае среднее время реакции на запрос (время от момента возникновения запроса до начала выполнения первой команды обработчика) может оказаться значительным

Способы задания точек входа в обработчик прерывания.

- **Короткое прерывание** - по прерыванию происходит выход на первую исполняемую команду. Адрес команды может быть фиксирован (TMS 320C10) или задаваться номером прерывания (ADSP 2181).

- **Длинное прерывание** - выход в обработчик через команду JMP или через вектор прерывания.

- **Фиксированное расположение векторов в памяти** (Каждый вектор определяет адрес обработчика прерываний) (Intel 80x86);

- **Программно задаваемое / перемещаемое** расположение векторов (MC 68060)

Источники прерываний: внешние и внутренние события.

До сих пор речь шла о **событиях, внешних** по отношению к вычислительному ядру (процессор + память).

В ходе выполнения программы могут произойти **внутренние события**, которые заранее трудно предсказуемы, например:

- арифметическое переполнение
- деление на нуль
- появление в программе недопустимой команды (из-за ошибки программиста, из-за сбоя, из-за того, что программа разрабатывалась для "старших" моделей семейства процессоров и использует команды, отсутствующие в младшей модели)
- обращение к несуществующему адресу памяти

и многие другие подобные.

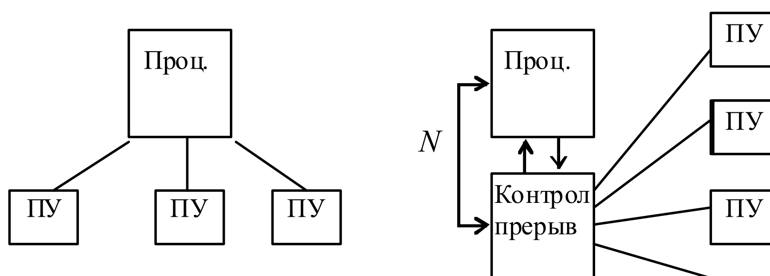
Такие внутренние события, делающие нормальное продолжение основного алгоритма невозможным, принято называть *прерываниями по внутренним причинам* или *исключительными ситуациями* или *исключениями (exceptions)*/

Почти все их можно выявить, вставляя в программу в подходящих местах соответствующие анализирующие фрагменты, Например, после выполнения действий по модификации адресов проверять результат на

нахождение в заданных границах, но это требует затрат дополнительного процессорного времени, иногда весьма значительных, а также дополнительных усилий программиста. Поэтому разработчики процессора могут предусматривать автоматическое выявление подобных ситуаций и генерацию запроса прерывания.

Радиальная схема и управление прерываниями. Контроллер прерываний.

Для реализации радиальной схемы подключения источников аппаратных запросов надо иметь отдельный вход запроса для каждого источника запроса. Большинство процессоров имеет малое количество входов запроса (1...3). Этого может быть достаточно в очень маленьких системах, однако весьма часто в системе требуется иметь больше входов запроса прерывания, чем имеет процессор. Это позволяет сделать дополнительный узел, называемый *программируемым контроллером прерываний* (*ПКП*). Обычно *ПКП* имеет несколько входов для подключения источников запросов, и один выход, подключаемый ко входу запроса прерывания процессора.



Радиальная и магистральная схемы организации внешних прерываний

ПКП выполняет несколько функций:

1) преобразовывает радиальную схему подключения периферийных устрой в магистральную (векторную),

2) обеспечивает обмен сигналами с процессором для идентификации источника прерываний,

3) позволяет управлять приоритетами отдельных запросов,

4) дает возможность программно разрешать/запрещать прерывания отдельно для каждого источника, а также выбирать фрагмент входного сигнала, инициирующий запрос (уровень или фронт, активная полярность).

В любой реальной вычислительной системе далеко не все векторы прерываний и даже не все входы запросов аппаратных прерываний бывают использованы. Разработчик программы или программно-аппаратного комплекса может использовать свободные ресурсы системы прерываний ля собственных нужд, например, для обеспечения реакции на события в «своем» (может быть, нестандартном) периферийном устройстве.

Действия, которые должен выполнить программист, чтобы прерывание было работоспособно.

В дальнейшем описании предполагается, что используется "свободное" прерывание.

1. Если используется аппаратное прерывание, предусмотреть соединение источника запроса с одним из свободных входов запроса. Вход надо выбирать с учетом требуемого приоритета добавляемого прерывания и возможностей контроллера прерываний по управлению приоритетами. Если используется программное прерывание, надо убедиться, что выбранный вектор свободен.

2. Написать подпрограмму реакции на запрос (обработчик, handler). Не забыть обеспечить в ней сохранение контекста на входе в обработчик и его восстановление на выходе из прерывания. (Иногда существуют соглашения о "свободных регистрах".). Выход надо делать командой возврата из прерывания. Примите решение, допускаете ли вы вложенные прерывания, если да - то в обработчике надо разрешить прерывания процессору (так как обычно при входе в прерывания повторные прерывания процессору автоматически запрещаются).

3. Занести начальный адрес обработчика в ячейки вектора для выбранного прерывания. Это лучше делать явными присваиваниями в инициализирующей части вашей программы.

4. Настроить контроллер прерываний на нужный режим: задать приоритет выбранного входа, задать форму сигнала запроса (для аппаратного прерывания).

5. Разрешить запрос прерывания от выбранного источника (для аппаратного прерывания).

6. Разрешить прерывания процессору.

Программные прерывания.

После введения в процессоры прерывания как способа реакции на асинхронные события, оказалось, что этот же механизм удобно использовать для обращения к подпрограммам наряду с обычной командой call.

Особенности вызова-возврата при прерывании: 1) сохранение большей части контекста, чем при выполнении call, а также 2) хранение адреса перехода в определенном месте (в векторе прерывания) - в некоторых случаях могут дать определенные преимущества.

В системах команд многих процессоров есть *команда (команды) программных прерываний* (**software interrupt**), позволяющие запустить механизм прерывания программно. Чаще всего для таких команд используются мнемоники **int** (от interrupt - прерывание) или **trap** (*англ.* ловушка). Команда обычно имеет один параметр, указывающий номер прерывания (т.е. фактически задающей адрес вектора - местоположение ячеек, хранящих адрес перехода на соответствующую подпрограмму). В виде программных прерываний оформляют во многих операционных системах обращения к стандартным функциям ОС. Преимуществом такого способа является то, что разработчики ОС могут свободно модифицировать как код, реализующий системную функцию, так и его расположение в адресном пространстве, оставляя неизменным лишь адрес вектора, через который

происходит обращение к этой системной функции. Таким образом, прикладная программа оказывается относительно независимой от версии ОС, если сохраняется интерфейс функции и адрес вектора.

6.4. Реализация механизма прерывания в процессорах Pentium

Общие сведения

В процессорах x86 имеется **три входа запроса** прерывания: немаскируемые входы RESET, NMI и маскируемый вход INT. По активному уровню сигнала по входу RESET в процессоре запускается микропрограмма начальной установки, после чего в пару регистров CS:IP загружаются значения **FFFF:0000**.

В **регистре состояний два бита** относятся к **управлению прерываниями**: флаг разрешения маскируемых прерываний **IF** и флаг пошаговой отладки **TF**. Флаг IF, будучи сброшен, запрещает запрос прерывания по входу INT. Использование флага TF для пошаговой отладки будет описано далее в подразделе "Аппаратная поддержка отладки".

Вектор прерывания в x86 содержит значения сегмента команд и счетчика команд, указывающие точку входа в обработчик, таким образом, вектор имеет длину 4 байта. Порядок байтов соответствует принципу Intel: старшие части по старшим адресам: младший и старший байты IP. затем младший и сташий байты CS.

Для векторов прерываний разработчики процессора i8086 отвели **область памяти** длиной 1 кБайт с 00000h до 003FFh. Прерывания идентифицируются *номером*, который равен адресу вектора, деленнному на 4 и который может иметь значения от 0 до 255_{10} = 0FFh.

В реальном режиме элемент таблицы - *вектор прерывания* – представляет собой начальный адрес программы обработки прерывания, состоящий из 16 битового значения сегментного регистра **cs** и 16-битового значения счетчика команд **ip**.

Начальный адрес таблицы векторов в физическом адресном пространстве для процессоров 386+ задается содержимым специального системного регистра IDTR (Interrupt Descriptor Table Register). Особенности прерываний в защищенном режиме процессора рассмотрены в главе 5.

В процессорах семейства Pentium имеется всего один вход аппаратного запроса прерывания, т.е. схема подключения источников запросов к процессору - векторная (магистральная). Максимальное количество независимых источников прерываний равно 256.

В составе вычислительной системы имеется два программируемых контроллера прерываний (ПКП, PIC – Programmable Interrupt Controller). ПКП соединены каскадно: второй соединен с первым, а первый с ЭВМ (рис.). Каждый ПКП имеет один выход запроса прерывания, подключенный к входу запроса процессора, и 8 входов, к которым могут подключаться сигналы запросов от устройств. Таким образом, ПКП преобразует векторную (магистральную) схему запросов прерываний в радиальную.

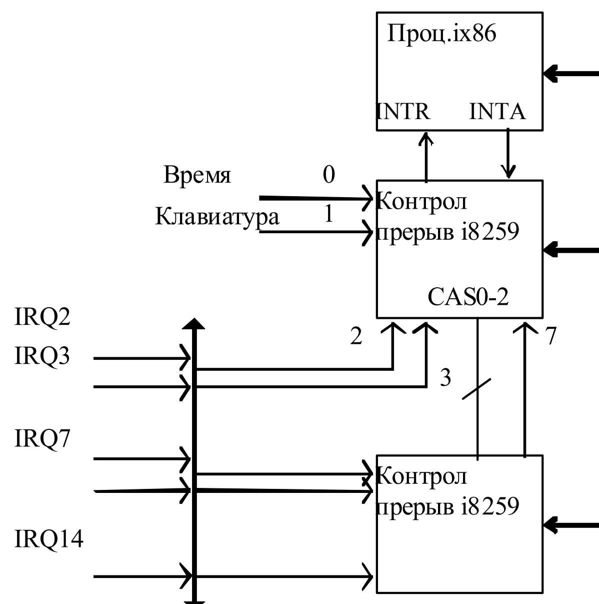


Рис. 6.8. Каскадная схема включения контроллеров прерываний

Внутренние регистры ПКП:

РЗПР (IRR) – регистр запросов.

РОЗПР (ISR) – регистр обслуживания запросов. Биты запросов переписываются из РЗПР.

РМЗ (IMR) – регистр маскирования. "1" – блокирует прерывание

Этапы обработки запросов в ПКП:

1. При запросе устанавливается соответствующий бит в "1" регистра РЗПР.

2. По первому импульсу INTA:

- бит в РЗПР сбрасывается;
- соответствующий бит в РОЗПР устанавливается, блокируя другие запросы с меньшим приоритетом;
- вырабатывается сигнал CAS0-2.

3. По второму импульсу INTA передается номер (вектор) прерывания в ЭВМ.

Ряд параметров работы ПКП может быть настроен программно через порты ПКП: ведущий 20h, 21h; ведомый – A0h, A1h. В частности ПКП позволяет устанавливать:

1. Начальный адрес векторов прерывания

2. Дисциплину приоритетов обслуживания

– фиксированный

– автоматички сдвигаемый

- программно управляемый (задается Nmax)

3. Форму сигнала запроса прерывания (фронт, уровень)

4. Разрешать/запрещать отдельные прерывания (программирование РМЗ)

5. Автоматически завершать обслуживание прерываний (сброс РОЗПР)

6. Опрашивать готовность (чтение РЗПР)

7. Режим специальной маски с отменой приоритетов.

Эти действия по настройке ПКП должны выполняться на этапе загрузки и инициализации операционной системы.

Действия x86 при входе в прерывание.

При входе в прерывание процессор x86 делает следующие шаги:

1) заканчивает выполнение текущей команды (а если текущая команда - запись в регистр сегмента стека SS, то выполняет еще и следующую команду - предполагается, что это команда записи нового значения в указатель стека SP);

2) проверяет, разрешено ли прерывание процессору (т.е. IF=1), если нет, то запоминает запрос и продолжает выполнение текущей программы, если прерывания разрешены, то:

3) сохраняет в стеке (в порядке перечисления) слово состояния процессора, содержимое регистра сегмента команд CS и счетчика команд IP, младшие байты по младшим адресам;

4) сбрасывает флаг разрешения прерывания IF, т.е. запрещает вложенное прерывание;

5) считывает с магистрали номер вектора прерывания (который ему должно передать ВнУ);

6) загружает из вектора новые значения счетчика команд IP и сегмента команд CS, в результате чего происходит переход на обработчик.

Команды управления прерываниями в x86.

cli запретить маскируемые прерывания (сбрасывается бит IF в регистре состояний)

sti разрешить маскируемые прерывания (бит IF устанавливается)

iret возврат из прерывания - восстанавливается из стека основной контекст прерванной программы (регистр состояний, сегмент команд и счетчик команд)

wait ожидание сигнала готовности (прерывания) от сопроцессора

hlt ожидание прерывания: процессор останавливается и продолжает работу после прихода запроса прерывания - переходит на обработчик, по завершении которого продолжает выполнение с команды, следующей за hlt.

int n программное прерывание через вектор номер **n.**

into прерывание при переполнении (т.е. если установлен флаг OF) через вектор 4. Последние две команды могут вызвать переход на любой обработчик, в том числе и на те, которые соответствуют исключения и аппаратным прерываниям.

int3 однобайтовая команда для отладочных остановов. Ее использование описано далее в подразделе "Аппаратная поддержка отладки"

6.5. Аппаратная поддержка отладки

При отладке программ желательно иметь следующие возможности:

1) выполнять отлаживаемую программу до заранее заданного места (задавать точки останова)

2) выполнять отдельные участки по шагам (по одной команде)

3) наблюдать текущее состояние отлаживаемой программы (все ее переменные, включая и вспомогательные, которые не фигурируют явно в программе, если пишем на ЯВУ)

4) иметь возможность принудительно завершить "зависшую" программу

5) средство отладки хочется защитить от некорректных действий отлаживаемой программы.

Уже давно в **качестве инструмента отладки** используют специальные вспомогательные **программы-отладчики**, которые взаимодействуют с отлаживаемой программой.

Функции отладчика:

1) показать - изменить содержимое заданных областей памяти или регистров

2) передать управление отлаживаемой программе

3) организовать выполнение программы по частям (до заданных точек, по одной команде и т.п.)

4) привязать выполняемые машинные команды к фрагментам исходного текста на ЯВУ

5) ... и др. и пр.

Задание точек останова (breakpoints).

Хочется, чтобы после заданной команды не выполнялась следующая, а управление было передано отладчику.

команда i-1

команда i ее хотим выполнить последней

команда i+1 вместо нее **(переход на отладчик)**,

команда i+2

команда i+3

Для организации останова (breakpoint) можно использовать следующую последовательность действий:

1) отладчик запоминает в своей внутренней области данных байты кода отлаживаемой программы, которые будут затерты командой перехода на отладчик

2) записывает команду перехода

3) передает управление отлаживаемой программе

4) когда будет достигнуто место останова, сработает команда перехода, и управление вернется в отладчик,

5) отладчик восстанавливает отлаживаемую программу, возвращая затертые байты.

Когда-то давно так и делали, используя для перехода команду jmp. Так можно делать, если команда перехода не длиннее самой короткой команды в системе команд. Если это не так, то рано или поздно столкнемся с ситуацией, когда команда перехода на отладчик затирает более, чем одну команду исходного кода (команды (i+1), (i+2),...в приведенном примере а логика выполнения отлаживаемой программы такова, что в ней происходит переход на команду (i+2). После установки точки останова здесь "хвост" команды перехода на отладчик (например jmp...??). В системе команд x86 минимальная длина команды - 1 байт.

Вторая проблема: если **точек останова несколько**, как узнать, на какой остановились ?

В системе команд x86 для точки останова есть специальная команда int3 длиной в 1 байт. Эта команда вызывает прерывание через вектор 3. Отладчик для установки (нескольких) остановов:

1) запоминает для каждой точки останова затираемый один байт

2) записывает в эти точки однобайтовую команду **int3**

3) записывает в **вектор3** адрес перехода на себя (адрес возврата в отладчик)

4) передает управление отлаживаемой программе

Программа выполняется, а если натолкнется на **int3** - произойдет прерывание, в стек запишется адрес, на 1 больше точки останова (по нему можно идентифицировать, какой останов сработал).

5) отладчик должен вернуть на место все сохраненные байты отлаживаемой программы.

Поддержка пошаговой отладки.

В этом случае точкой останова (breakpoint - переход на отладчик) должна быть каждая команда отлаживаемой программы. Описанная выше техника задания точек останова в этом случае оказывается громоздкой, а иногда просто неприменима.

Пошаговую отладку обычно поддерживают в процессорах на аппаратном уровне его разработчики. Типовой прием для этого - наличие в процессоре особого режима, когда **прерывание происходит после выполнения каждой команды**. Если бы такой режим был включен постоянно, то он был бы неработоспособным. Для включения-выключения этого режима обычно используют один из битов регистра состояния процессора. В процессорах x86 - это бит 8 регистра состояния (TF Trap Flag). Если TF=1, то пошаговое прерывание разрешено, TF=0 соответствует обычному режиму работы.

Теперь необходимо организовать, чтобы TF был равен 1, когда выполняется команда отлаживаемой программы, и сбрасывался в 0 при переходе на отладчик. После того, как отладчик выполнит все действия по индикации состояния отлаживаемой программы, необходимо осуществить переход на очередную ее команду, одновременно установив TF в 1, чтобы после выполнения этой команды вновь произошло прерывание. Эти требования выполняются автоматически, если использовать следующую технику:

Отладчик перед передачей управления на отлаживаемую программу,
1) записывает в вектор пошагового прерывания (номер 01h, адрес 00004h) адрес перехода в отладчик после выполнения команды отлаживаемой программы;
2) записывает в стек (с соответствующим смещением указателя стека): а) слово состояния процессора с установленным TF-битом, б) адрес отлаживаемой программы (CS:IP - 4 байта), с которого надо начать пошаговую отладку;
2) выполняет команду возврата из прерывания **iret**.

```
        .....
    push        #psw
    push        #cs
    push        #ip
    .....
    iret
l1  .....       ; сюда можем вернуться по прерыванию после выполнения команды.
```

По команде **iret** из стека загружается в CS:IP адрес "возврата" в отлаживаемую программу, а в SR - слово состояния с установленным TF-битом. Выполняется команда отлаживаемой программы (одна !), после чего происходит прерывание. Теперь выполняются действия, как при любом

прерывании: слово состояния с установленным TF-битом и содержимое CS:IP (адрес следующей команды отлаживаемой программы) сохраняются в стеке, из вектора 01h загружается CS:IP (это адрес перехода в отладчик). Теперь отладчик может показать состояние отлаживаемой программы после сделанного шага.

6.6. Прямой доступ к памяти

В ЭВМ используются два основных способа организации передачи данных между памятью и периферийными устройствами: программно-управляемая передача и прямой доступ к памяти (ПДП).

Программно-управляемая передача данных осуществляется при непосредственном участии и под управлением процессора. Например, при пересылке блока данных из периферийного устройства в оперативную память процессор должен выполнить следующую последовательность шагов:

1. сформировать начальный адрес области обмена ОП;
2. занести длину передаваемого массива данных в один из внутренних регистров, который будет играть роль счетчика;
3. выдать команду чтения информации из УВВ; при этом на шину адреса из МП выдается адрес УВВ, на шину управления - сигнал чтения данных из УВВ, а считанные данные заносятся во внутренний регистр МП;
4. выдать команду записи информации в ОП; при этом на шину адреса из МП выдается адрес ячейки оперативной памяти, на шину управления - сигнал записи данных в ОП, а на шину данных выставляются данные из регистра МП, в который они были помещены при чтении из УВВ;
5. модифицировать регистр, содержащий адрес оперативной памяти;
6. уменьшить счетчик длины массива на длину переданных данных;
7. если переданы не все данные, то повторить шаги 3-6, в противном случае закончить обмен.

Программа чтения данных в память, реализующая описанные выше действия, может иметь следующий вид:

```
SETUP: MOV AX,SEGMENT       ; настройка сегментного регистра
       MOV DS,AX
       MOV DI,OFFSET        ; настройка адреса
       MOV CX,COUNT         ; количество байт
       MOV DX,IOPORT        ; DX = порт ввода/вывода
READ:  IN AL,DX             ; чтение байта из порта
       MOV [DI],al          ; сохранить данные
       INC DI               ; увеличить индекс
       LOOP READ            ; продолжить до тех пор, пока CX = 0
CONT:  ......               ; продолжение программного кода
```

Как видно, программно-управляемый обмен ведет к нерациональному использованию мощности микропроцессора, который вынужден выполнять большое количество относительно простых операций, приостанавливая работу над основной программой. При этом действия, связанные с обращением к оперативной памяти и к периферийному устройству, обычно требуют удлиненного цикла работы микропроцессора из-за их более медленной по сравнению с микропроцессором работы, что приводит к еще более существенным потерям производительности ЭВМ.

Альтернативой программно-управляемому обмену служит прямой доступ к памяти - способ быстродействующего подключения внешнего устройства, при котором оно обращается к оперативной памяти, не прерывая работы процессора. Такой обмен происходит под управлением отдельного устройства - контроллера прямого доступа к памяти (КПДП).

Структура ЭВМ, имеющей в своем составе КПДП, представлена на рис.6.10.

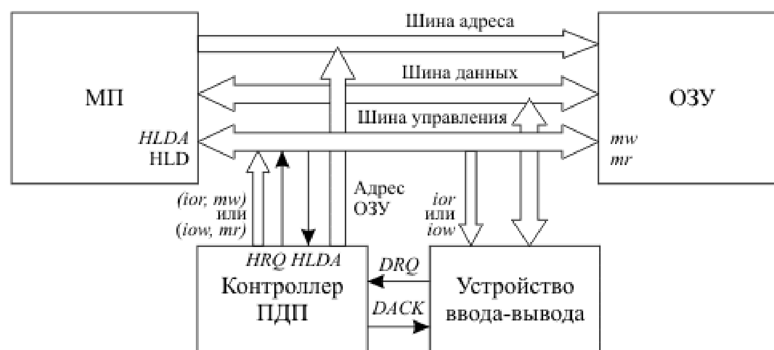


Рис. 6.10. Структура системы с контроллером ПДП

Процедура передачи данных в режиме ПДП состоит в следующем.

- Запрос DREQ (Dma REQuest) на начало передачи поступает в контроллер ПДП в виде электрического сигнала из внешнего устройства.
- КПДП посылает в процессор запрос канала HOLD
- Процессор заканчивает текущий канальный цикл и предоставляет канал, о чем сообщает сигналом HLDA (предоставление канала).
- КПДП сообщает устройству ввода-вывода о начале выполнения циклов прямого доступа к памяти (DACK).
- КПДП генерирует канальные циклы (т.е. нужные адреса и последовательности управляющих сигналов), в которых между памятью и внешним устройством происходит обмен байтами (или словами).

Временная диаграмма режима ПДП приведена на рис.6.11.

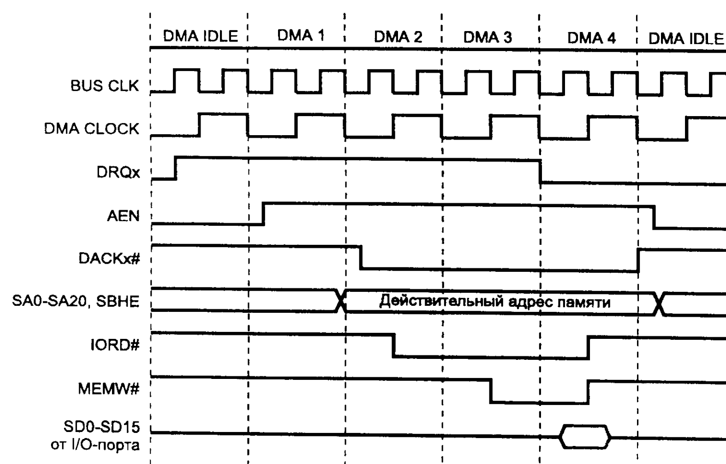


Рис. 6.11. Временная диаграмма режима ПДП

Перед началом обмена программа должна указать контроллеру ПДП:

- начальный адрес массива в памяти, куда (или откуда) будет передача
- направление передачи (в память или из нее)
- количество байтов (или слов), которые надо передать
- как реагировать на сигнал запроса......

и некоторые другие условия (см. далее описание КПДП для PC-совместимых компьютеров).

Передача массива данных по каналу состоит из последовательности так называемых циклов DMA. Цикл DMA для передач между памятью и устройством ввода-вывода представляет собой комбинацию одновременного выполнения шинных команды обращения к памяти (-MEMR или -MEMW) и команды обращения к порту (-IOW или -IOR, соответственно), во время которого на шине адреса активен адрес памяти. Активный порт в цикле DMA определяется косвенным образом по комбинации активных сигналов DRQ и -DACK. Реагировать на сигналы -IOW и -IOR в цикле DMA может только тот порт, который вызвал активизацию сигнала DRQ и получил подтверждение в виде соответствующего сигнала -DACK.

После того как один из каналов запрограммирован на необходимую передачу (запрограммирован канал контроллера DMA и регистр памяти страниц DMA), запускается устройство ввода-вывода, подключенное к этому каналу. С этого момента обслуживание обмена поручается подсистеме DMA, а микропроцессор может быть занят чем-либо другим. Когда устройство становится готовым к обмену, оно выдает запрос на обслуживание DRQ (рис.). Если данный канал в этот момент не замаскирован, подсистема DMA выдает запрос на захват шины у микропроцессора.

Когда микропроцессор готов освободить шину, он подтверждает запрос на захват шины сигналом HOLDA, и шина поступает в распоряжение подсистемы DMA. После этого выполняются действия по выдаче адреса ячейки памяти, с которой будет выполняться обмен по каналу контроллера DMA.

Начало передачи может происходить не только по внешнему сигналу, но и по команде (устанавливающей нужный бит в нужном порте) - по аналогии с прерыванием.

В ходе передачи КПДП может поддерживать три режима передачи:

1) **Одиночная передача** – на каждый фронт сигнала запроса

передается одно слово данных. DRQ должен быть активным, пока не активизируется соответствующий DACK. Если DRQ активен на протяжении одиночной передачи, контроллер переходит в неактивное состояние по выполнении одной передачи и освобождает шину системе.

2) **Передача по запросу** - после подачи сигнала запроса передача продолжается до тех пор, пока сигнал запроса активен, и прекращается, если ВНУ снимает сигнал запроса, хотя передача и не завершена (т.е. не передано количество слов, указанное при программировании КПДП

3) **Блочная передача** – после подачи сигнала запроса передаются все запрошенные слова, независимо от дальнейшего поведения сигнала запроса. DRQ должен быть активным, пока не появиться активный DACK.

В качестве примера отметим шину ISA. Магистраль обеспечивает подключение до семи внешних устройств, работающих в режиме прямого доступа к памяти, и до 11 запросов прерываний от УВВ. Еще четыре запроса прерываний зарезервированы за устройствами, входящими в состав стандартной конфигурации ЭВМ, и на магистраль не выведены.

Программирование

Перед началом работы необходимо запрограммировать все регистры представленные в таблице, часть из них программируются при инициализации системы. Программируются:

- начальный адрес памяти для обмена;

- уменьшенное на единицу число передаваемых байтов;

- направление обмена,

- требуемые режимы работы (разрешить или запретить циклическое изменение приоритетов, автоинициализацию, задать направление изменения адреса при обмене и т. д.).

- запрограммированный канал должен быть демаскирован (бит маски канала устанавливается при этом в 0), после чего он может принимать сигналы "Запрос на ПДП", генерируемые тем внешним устройством,

которое обслуживается через этот канал. Сигнал "Запрос на ПДП" может быть также инициирован установкой в 1 бита запроса данного канала в регистре запросов контроллера.

Внутренние регистры КПДП

Наименование	Число	Разрядность
Регистр начального адреса	4	16
Регистр начального счетчика циклов	4	16
Регистр текущего адреса	4	16
Регистр текущего счетчика циклов	4	16
Рабочий регистр адреса	1	16
Рабочий регистр счетчика циклов	1	16
Регистр состояния	1	8
Регистр команд	1	8
Регистр режима	4	6
Рабочий регистр	1	8
Регистр масок	1	4
Регистр запросов	1	4
Страничные регистры	4	

6.7. Шины. PCI. PCI Express

Шины дают возможность осуществлять обмен данными, и используют некоторый протокол, благодаря которому весь обмен происходит контролируемым образом. Однако шины имеют и другие выделяющиеся черты:

Стандартизированные электрические характеристики (например, количество проводников, уровни напряжения, скорости сигналов и т.д.)

Стандартизированные механические характеристики (например, тип соединения, размер платы, физическая конструкция и т.д.)

Стандартизированный протокол.

Типы шин:

- шины процессор - память (переднего плана Front-Side Bus, связывает процессор и ОЗУ; заднего плана Back-Side Bus, связывает процессор и КЭШ второго уровня)

- шины ввода - вывода (PCI, SCSI)
- системные шины (общая шина для памяти и устройств ввода-вывода).

Элементы шин:

Драйвер - возбудитель линии шины - схема меняющая напряжение.

- Задает три состояния на выходе:high, low, off.
- Типы драйверов: ТТЛ, МОП, ЭСЛ.
- Несколько драйверов могут быть подключены через монтажное ИЛИ.

Приемник - схемы сравнивающие уровень сигнала на входе со стандартным значением, по итогам определяется лог 0 или лог 1.

Трансивер - приемопередатчик, содержит приемник и драйвер, вход и выход которых сводятся в одну точку.

Арбитраж шин

В системах на роль ведущего вправе одновременно претендовать сразу несколько из подключенных к шине устройств, однако управлять шиной в каждый момент времени может только одно из них. **Чтобы исключить конфликты, шина должна предусматривать определенные механизмы арбитража запросов и правила предоставления шины одному из запросивших устройств.** Решение обычно принимается на основе приоритетов претендентов.

Схемы приоритетов

Каждому потенциальному ведущему присваивается определенный уровень приоритета, который может оставаться неизменным *(статический* или *фиксированный приоритет)* либо изменяться по какому-либо алгоритму *(динамический приоритет)*.

Основной недостаток статических приоритетов в том, что устройства, имеющие высокий приоритет, в состоянии полностью блокировать доступ к шине устройств с низким уровнем приоритета.

Алгоритмы динамического изменения приоритетов

- **простая циклическая смена приоритетов;**

После каждого цикла арбитража все приоритеты понижаются на один уровень, при этом устройство, имевшее ранее низший уровень приоритета, получает наивысший приоритет.

- **циклическая смена приоритетов с учетом последнего запроса;**

Все возможные запросы упорядочиваются в виде циклического, списка. После обработки очередного запроса обслуженному ведущему назначается низший уровень приоритета. Следующее в списке устройство получает наивысший приоритет.

- **смена приоритетов по случайному закону;**
- **схема равных приоритетов;**

При поступлении к арбитру нескольких запросов каждый из них имеет равные шансы на обслуживание. Возможный конфликт разрешается арбитром. Схема принята в асинхронных системах.

- **алгоритм наиболее давнего использования.**

В *алгоритме наиболее давнего использования* (LRU, Least Recently Used) после каждого цикла арбитража наивысший приоритет присваивается ведущему, который дольше чем другие не использовал шину.

- **алгоритм очереди (первым пришел - первым обслужен);**
- **алгоритм фиксированного кванта времени.**

Схемы арбитража

Централизованный арбитраж. При централизованном арбитраже в системе имеется специальное устройство центральный арбитр, - ответственное за предоставление доступа к шине только одному из запросивших ведущих. Это устройство, называемое иногда центральным контроллером шины, может быть самостоятельным модулем или частью ЦП.

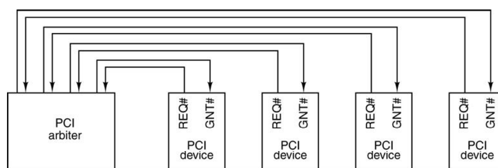


Схема централизованного арбитража PCI шины.

Децентрализованный арбитраж. Единый арбитр отсутствует, каждый ведущий имеет блок управления доступа к шине.

Опросные схемы арбитража. Запросы только фиксируются. Контроллер шины узнает о них опросив ведущих.

PCI - шина

PCI (Peripheral Component Interconnect) local bus - шина соединения периферийных компонентов. PCI это локальная шина, так как она занимает место между шиной процессора и периферийной шиной (например ISA). В отличие от ISA , шина PCI четко стандартизована.

В отличие от магистрали ISA, в которой управление магистралью поддерживается схемотехникой, сосредоточенной в устройствах-абонентах, магистраль PCI управляется индивидуальным контроллером, который имеет свою систему управляющих команд, и одной из основных функций которого является конфигурирование устройств, включенных в магистраль. Каждое устройство, включаемое в PCI, должно поддерживать возможности автоматической конфигурации.

Шина является синхронной - фиксация всех сигналов выполняется по положительному перепаду (фронту) сигнала CLK. В каждой транзакции (обмене по шине) участвуют два устройства - инициатор обмена (Initiator или Master, инициирующее устройство, ИУ) и целевое устройство (Target или Slave, ЦУ).

Размер адресного пространства памяти Соответствует разрядности 32-битовой шины адреса и равен 232 = 4 Гбайт. В стандарте предусмотрен вариант 64-разрядной шины адреса, т.е. возможна адресация с использованием 64-разрядного адреса. Реализация шины в PC-совместимых компьютерах не поддерживает 64-битовую адресацию.

Периферийные устройства могу использовать адресное пространство памяти так же, как и адресное пространство ввода-вывода.

Пространство ввода-вывода 232=4 Гбайт или 264≈6,4∗1019 адресов. Одно физическое/логическое устройство обычно занимает в адресном пространстве ввода-вывода более чем один адрес (некоторые могут занимать 32…64 адреса).

Разрядность шины данных. Может быть (в соответствии со спецификацией ver.2.1) 32 или 64 разряда.

Мультиплексирование адреса-данных использовано как в 32 так и в 64-разрядном варианте. Адрес и данные передаются по одним и тем же линиям в разные моменты времени.

Характеристики канального цикла. Тактовая частота шины IPC по спецификации 2.1 равна 33 или 66 МГц (т.е. один такт – 30 или15 нс). В PC-совместимых машинах используется 33 МГц. (В то же время частота внутренней системной шины, связывающей процессор с подсистемой оперативной памяти, может достигать 100 МГц)

Длительность канального цикла и возможность управления длительностью канального цикла. В шине PCI нет понятия фиксированного канального цикла.– в описании вместо понятия «канальный цикл» используется понятие «транзакция». Все транзакции трактуются как пакетные: в начале транзакции идет фаза адреса (длительностью в один такт), после чего следуют одна или несколько фаз данных. Длительность фазы данных минимально также один такт, но как Master (Initiator) так и Slave (Target) могут в каждой фазе данных вводить такты ожидания (подряд не более семи). Таким образом, максимальная скорость передачи приближается к одному слову за такт. А минимальная скорость обмена составляет 8 тактов/слово, т.е. около 4 слов/мкс. (Если устройство не в состоянии поддерживать такую скорость, обмен возможен только одиночными словами)

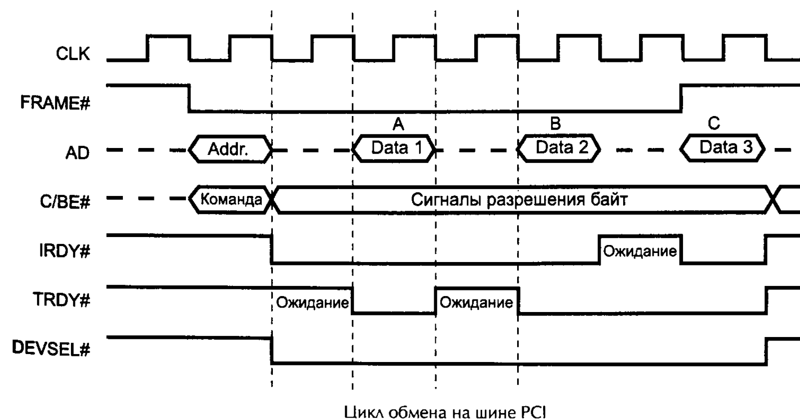


Рис. 6.12. Временная диаграмма шины PCI.

Порядок обмена:

1) Инициатор активирует сигнал FRAME# (кадр), отмечающий начало и конец транзакции и одновременно по шине AD передает адрес, а по линиям C/BE# - тип транзакции (команду)

2) Адресованое целевое устройство отвечает сигналом (Device Select – устройство выбрано)

3) Инициатор сигналом IRDY# (Initiator Ready - готовность инициатора) и независимо от него целевое устройство сигналом TRDY# (Target Ready – готовность целевого устройства) подтверждают свою готовность к обмену.

4) Обмен данными происходит в каждом такте только при условии наличия обоих сигналов, упомянутых в п.3). Если какое-либо из устройств «не успевает», оно может снять свой сигнал готовности, в результате чего вводятся такты ожидания (не более 7 тактов между соседними тактами данных), до тех пор, пока оба сигнала готовности вновь не станут активными.

5) Количество фаз данных в транзакции заранее не определено. Перед последним тактом обмена инициатор снимает сигнал FRAME#, что является сигналом окончания транзакции.

6) По окончании последнего такта оба устройства должны снять свои сигналы готовности, после чего транзакция считается завершенной

Завершение по тайм-ауту (Time-out) происходит, когда во время транзакции у инициатора отбирают право на управление шиной или когда истекает время, указанное в его таймере - медленное целевого устройства или слишком длинная транзакция.

Транзакция может быть завершена и по инициативе целевого устройства – для этого на шине PCI имеется сингнал STOP#.

Особенность временных диаграмм на магистрали PCI – протокол квитирования – инициатор всегда получает информацию об отработке транзакции целевым устройством благодаря наличию сигналов DEVSEL# и TRDY# от целевого устройства.

Из изложенного должно быть ясно, что при длинном пакете и при отстутствии тактов ожидания скорость обмена стремится к величине «одно слово за такт», что при 32-разрядном обмене и при частоте 33 МГц дает 33·4=132 Мбайт/с, а при 64-разрядной шине данных и при частоте 66 МГц – 66·8=528 Мбайт/с.

Контроль достоверности передачи

Для контроля используется дополнительная линия четности PAR, которая защищает сигналы AD (32 линии) и C/BE# (4 линии). Сигнал на линии четности формируется таким образом, чтобы общее количество единичных битов на контролируемых линиях (включая и линию четности PAR) было бы четным. При сбое на одной из линий, четность нарушается, что бызывает появление активного уровня на линии магистрали PERR#.

Команды шины PCI определяются значениями бит C/BE# в фазе адреса в соответствии с табл. :

C/BE[3:0]	Тип команды
0000	Interrupt Acknowledge - подтверждение прерывания
0001	Special Cycle - специальный цикл
0010	I/O Read - чтение порта ввода/вывода
0011	I/O Write - запись в порт ввода/вывода
0100	Зарезервировано
0101	Зарезервировано

0110 Memory Read - чтение памяти
0111 Memory Write - запись в память
1000 Зарезервировано
1001 Зарезервировано
1010 Configuration Read - конфигурационное считывание
1011 Configuration Write - конфигурационная запись
1100 Multiple Memory Read - множественное чтение памяти
1101 Dual Address Cycle - двухадресный цикл
1110 Memory Read Line - чтение строк памяти
1111 Memory Write and Invalidate - запись с инвалидацией

В команде подтверждение прерывания контроллер прерываний передает вектор прерывания по шине AD.

Команды конфигурационного чтения и записи адресуются к конфигурационному пространству и обеспечивают доступ к 256-байтным структурам. Обращение производится двойными словами. Структура содержит идентификатор устройства и производителя, состояние и команду, информацию об используемых ресурсах и ограничения на использование шины.

Управление магистралью

Любое устройство, подключенное к магистрали PCI может быть Bus-Master-ом. Но оно должно выполнять все требования режимов и временных диаграмм, поэтому оно должно включать в свой состав сложную схемотехнику управления. См далее раздел «Что должно быть в ВНУ для подключения».

Связь магистрали PCI с другими магистралями в системе

Данная магистраль PCI связана в системе с другими магистралями. Для этого применяются специальные узлы, называемые «мостами шины PCI» (PCI Bridge).

Главный мост (Host Bridge) – связывает PCI с магистралью процессора

Одноранговый мост (Peer-To-Peer Bridge) – для связи между собой двух или боле магистралей PCI

Специальный мост – для связи PCI с магистралью ISA, которая есть в большинстве современных системных плат, чтобы можно было использовать «старые» платы расширения (они кроме того, проще и дешевле).

«Мосты» программируются при инициализации и конфигурировании системы и «знают», какие адреса и запросы к какой магистрали относятся.

Автоконфигурирование устройств

Системные ресурсы, такие как порты ввода/вывода и линии запроса прерывания, на магистрали PCI распределяются автоматически. После системного сброса устройства, установленные на магистраль PCI не отвечают на обращения в адресное пространство памяти или ввода/вывода. Доступ к устройству возможен только через порты ввода/вывода контроллера PCI. Через эти порты системное ПО, записанное в BIOS системной платы с магистралью PCI после системного сброса: а) обращается к так называемому «конфигурационному пространству» каждого периферийного устройства, б) получает от него информацию о потребных ресурсах (адресах портов, запросах прерывания), в) производит распределение ресурсов, г) записывает параметры конфигурации в каждое из устройств. Только после этого к устройствам становится возможным доступ через порты ввода/вывода или через участки адресного пространства памяти.

Использование модулей стандарта PCI на других платформах

Поскольку PCI стандартизована и используется на разных платформах (в частности на Power PC), можно одну и ту же периферийную плату PCI использовать в разных компьютерах. Это поддерживается в BIOS многих периферийных плат – там есть несколько программных модулей, предназначенных для работы в разных компьютерах.

Поддержка обмена по прерываниям

Количество входов запроса определяется структурой магистрали PCI, на которой не может быть более 4 устройств. Магистраль имеет 4 радиальных входа запросов прерывания INTRA#, INTRB#, INTRC#, INTRD#,. Эти входы

перенаправляются на стандартные запросы IRQn. Соответствие перенаправления выбирается при автоконфигурации после включения или «холодного» рестарта системы.

Еще одна хорошая особенность запросов прерываний магистрали PCI – активный уровень сигналов запроса – низкий. Это позволяет использовать один вход запроса несколькими устройствами с объединением сигналов запроса по «монтажному ИЛИ»

На магистрали PCI распределение ресурсов выведено из-под опеки программиста или пользователя, и делается BIOS-ом автоматически во время инициализации

Поддержка прямого доступа к памяти

В спецификации PCI предусматривается возможность наличия в системе нескольких устройств-инициаторов. Каждое из таких устройств соединено парой сигналов REQ# (Request – запрос на управление магистралью) и GNT# (Grant – предоставление магистрали). Каждое устройство – инициатор должно иметь собственный программируемый таймер MLT (Master Latency Timer), определяющий максимально допустимое количество тактов в одной тразакции. При инициализации устройств на шине PCI можно задать этот параметр индивидуально для каждого устройства и тем самым задать распределение пропускной способности магистрали между абонентами. В компьютере с системной платой, имеющей магистраль PCI это делает программа POST при инициализации системы.

Конструктивное исполнение и возможности питания устройств

Разъем магистрали PCI – двухрядный и использует 62 пары контактов в 32-разрядном варианте или 94 пары в 64-разрядном. На разъеме имеются следующие напряжения питания: +5В, +3,3В, +12В, -12В.

Что (какая схемотехника) должно содержаться в периферийном устройстве, чтобы оно могло быть подключено к магистрали и обеспечивать обмен (при разных режимах, в частности по прерыванию и по ПДП).

Схемотехника для поддержки всех свойств магистрали PCI должна быть весьма сложной. Некоторые фирмы, в частности Altera, выпускают узлы для поддерки алгоритмов обмена по PCI.

На рисунке далее изображена структура подобного устройства фирмы Altera, выполненного на базе БИС Flex-логики. В ней можно выделить три функциональных части:

Блок конфигурационных регистров (64 байта)

Интерфейс целевого устройства

Встроенный контроллер поддержки прямого доступа к памяти, включающий буферное ОЗУ объемом 64 байта (16 четырехбайтовых слов), а также интерфейсную управляющую логику, поддерживающую взаимодействие с периферийным устройством.

Включение данной БИС в состав периферийного устройства в полной мере решает вопрос его подключения к магистрали PCI.

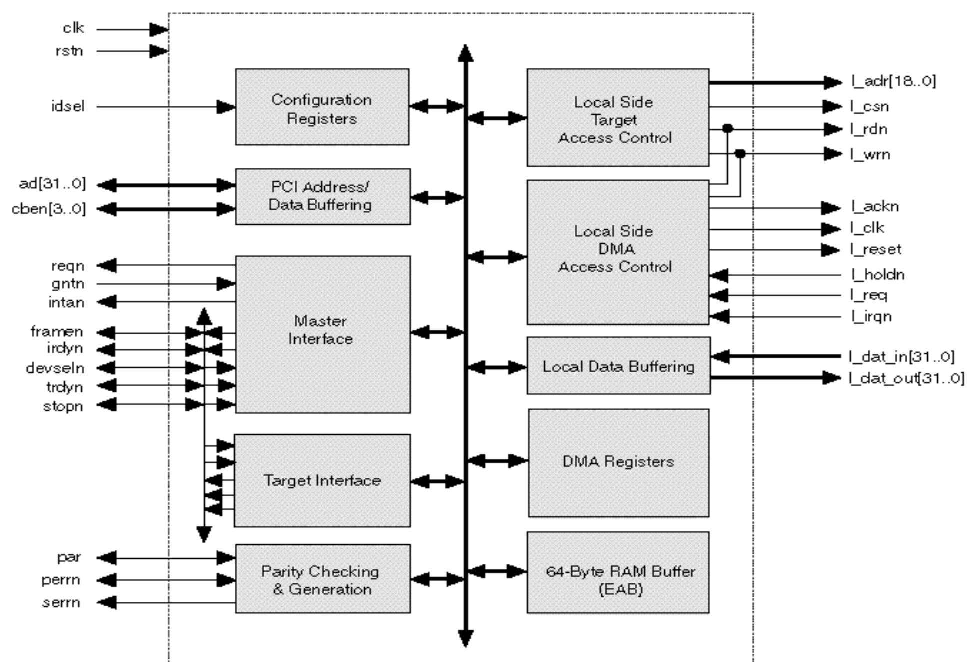


Рис. 6.13. Структурная схема контроллера PCI шины.

PCI Express

PCI Express - компьютерная шина, использующая программную модель шины PCI и высокопроизводительный физический протокол,

основанный напоследовательной передаче данных. В отличие от шины PCI, использовавшей для передачи данных общую шину, PCI Express, в общем случае, является пакетной сетью с топологией типа звезда.

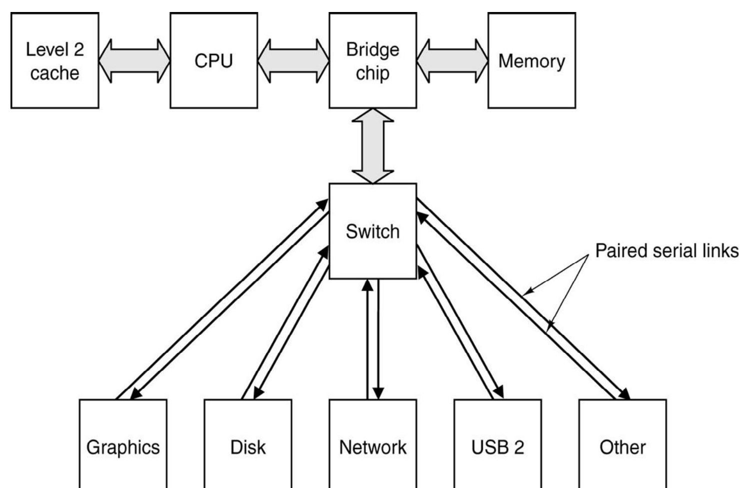


Рис. 6.14. Структурная схема подключения устройств через PCI-E.

Для подключения устройства PCI Express используется двунаправленное последовательное соединение типа точка-точка (рис. 6.14), называемое линией; это резко отличается от PCI, в которой все устройства подключаются к общей 32-разрядной параллельной двунаправленной шине. Соединение (link) между двумя устройствами PCI Express и состоит из одной (x1) или нескольких (x2, x4, x8, x12, x16 и x32) двунаправленных последовательных линий (рис. 6.15). Каждое устройство должно поддерживать соединение по крайней мере с одной линией (x1). На электрическом уровне каждое соединение использует низковольтную дифференциальную передачу сигнала (LVDS), приём и передача информации производится каждым устройством PCI Express по отдельным двум проводникам, таким образом, в простейшем случае, устройство подключается к коммутатору PCI Express всего лишь четырьмя проводниками.

PCI Express пересылает всю управляющую информацию, включая прерывания, через те же линии, что используются для передачи данных. Последовательный протокол никогда не может быть заблокирован. На физическом уровне, PCI Express использует метод канального

кодирования8b/10b (8 бит в десяти, избыточность — 20%) для устранения постоянной составляющей в передаваемом сигнале и для встраивания информации о синхронизации в поток данных. В PCI Express 3.0 используется более экономное кодирование 128b/130b с избыточностью 1,5%. Сигнальный уровень 0.8 вольт. Каждый канал состоит из двух дифференциальных сигнальных пар (необходимо только 4 контакта). Пропускная способность 2.5 Гигабита (250 МБ) в секунду для одного канала в каждом направлении одновременно (полный дуплекс), однако, следует учесть, что эффективная скорость передачи данных за вычетом избыточного кодирования составляет 2 Гигабита (200 МБ) ровно; Имеется возможность динамического подключения и конфигурации устройств.

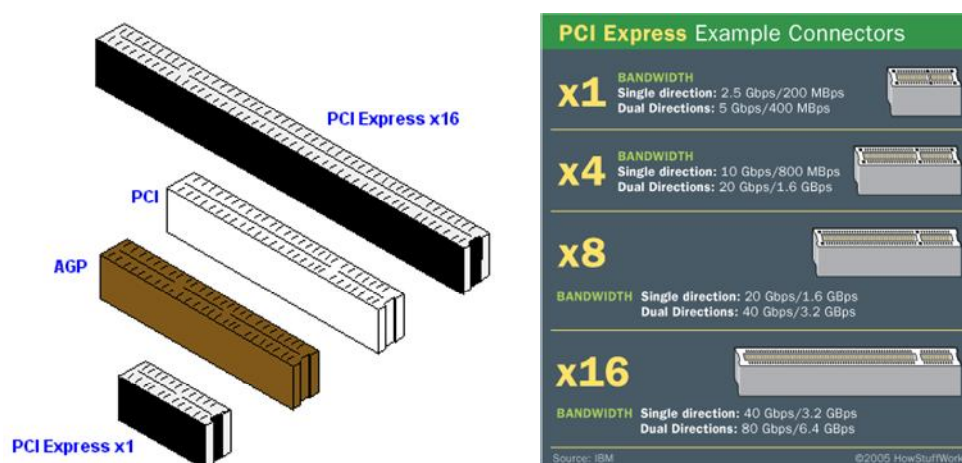


Рис. 6.15. Внешний вид разъемов шин.

6.8. Процесс загрузки компьютера

Что происходит при включении компьютера? С какого адреса выполняется программа? Как тестируется полная память? На эти и другие вопросы дает ответ описание процесса загрузки персонального компьютера .Перечислим последовательность этапов при включении компьютера по схеме: действие –результат.

1. После нажатия кнопки Power источник питания выполняет самотестирование. Вырабатывается сигнал PowerGood спустя 0,1...0,5 с и сигнал RESET.

2. Сигнал RESET снимает сигнал сброса с соответствующего входа микропроцессора

CS = FFFFh; IP = 0; DS = SS = ES = 0, сброс всех битов управляющих регистров, обнуление регистров АЛУ, все тристабильные буферные схемы переходят в высокоимпендансное состояние.

3. Микропроцессор начинает работу в реальном режиме с адреса FFFF:0000 здесь записана команда перехода на реально исполняемый код BIOS

4. Процессор реализует функцию начального самотестирования POST (Power-On Self Test). Тестируются процессор, память и системные средства ввода/вывода, а также производится конфигурирование программно-управляемых аппаратных средств материнской платы.

При выполнении каждой подпрограммы POST записывает её сигнатуру (код) в диагностический регистр, а также может выдавать диагностические сообщения в виде звуковых сигналов или свечения индикаторов. Краткие сообщения выводятся на экран.

5. Конфигурирование компьютера с использованием утилиты Setup, встроенной в код BIOS

Параметры конфигурирования запоминаются в энергонезависимой памяти, питаемой от миниатюрной батарейки, размещённой на материнской плате. (CMOS Memory, флэш, память для поддержки динамического конфигурирования системы Plug and Play).

6. Загрузка BOOT-сектора (INT19h). Сектор MBR (Master Boot Record – главная загрузочная запись) с координатами Cylinder:0 Head:0 Sector:1

Код MBR определяет расположение загрузочного (активного) раздела, считывая таблицу разделов, расположенную в конце MBR.

Master Boot Record передаёт управление коду загрузочного сектора в активном (загрузочном) разделе, который содержит загрузочную программу и таблицу параметров диска. Загрузочный сектор раздела просматривает

блок параметров BIOS в поисках расположения корневого каталога, а затем копирует из него в память системный файл IO.SYS

7. IO.SYS загружает драйверы некоторых устройств и выполняет ряд операций, связанных с загрузкой.

IO.SYS считывает информацию из системного реестра, а также исполняет файлы CONFIG.SYS и AUTOEXEC.BAT (при их наличии в корневом каталоге). При этом загружаются драйверы устройств, работающих в реальном режиме работы процессора, выполняются некоторые системные установки.

8. Загружается и запускается файл WIN.COM Он обращается к файлу VMM32.VXD. Виртуальные драйверы устройств, не загруженные с помощью файла VMM32.VXD, загружаются из файла SYSTEM.INI Затем из системного реестра загружается конфигурация с параметрами, установленными по умолчанию.

VMM32.VXD переключает процессор в защищённый режим, и начинается процесс инициализации виртуальных драйверов устройств согласно их параметру InitDevice. Процедура загрузки ОС заканчивается загрузкой файлов KRNL32.DLL, GDI.EXE, USER.EXE и EXPLORER.EXE.

Перечислим в порядке выполнения основные тесты POST для BIOS AWARD V4.51 и их сигнатуры.

C0 – осуществляется программирование регистров микросхемы Host Bridge для установки следующих режимов:

запрещается Internal и External Cache, а также операции с кэш-памятью;

перед запретом Internal Cache очищается;

Shadow RAM запрещается, вследствие чего происходит направление непосредственно к ROM циклов обращения к адресам расположения System BIOS.

далее программируются PIIX ресурсы: контроллер DMA, контроллер прерываний, таймер, блок RTC. При этом контроллер DMA переводится в пассивный режим.

C1 – с помощью последовательных циклов запись/чтение определяется тип памяти, суммарный объём и размещение по строкам. И в соответствии с полученной информацией настраивается DRAM-контроллер. На этом же этапе процессор должен быть переключён в Protected Mode (защищённый режим).

C3 – проверяются первые 256 кб памяти, которые в дальнейшем будут использованы как транзитный буфер, а также осуществляется распаковка и копирование System BIOS в DRAM.

C6 – по специальному алгоритму определяется наличие, тип и параметры External Cache.

CF – определяется тип процессора, а результат помещается в CMOS. Если по каким-либо причинам определение типа процессора закончилось неудачно, такая ошибка становится фатальной и система, а соответственно и выполнение POST, останавливается.

05 – осуществляется проверка и инициализация контроллера клавиатуры, однако на данный момент приём кодов нажатых клавиш ещё не возможен.

07 – проверяется функционирование CMOS и напряжение питания её батареи. Если фиксируется ошибка питания, выполнение POST не останавливается, однако BIOS запоминает этот факт. Ошибка при контрольной записи/чтении CMOS считается фатальной и POST останавливается на коде O7.

BE – программируются конфигурационные регистры Host Bridge и PIIX значениями, взятыми из BIOS.

0A – генерируется таблица векторов прерываний, а также производится первичная настройка подсистемы управления питанием.

0B – проверяется контрольная сумма блока ячеек CMOS, а также, если BIOS поддерживает PnP, выполняется сканирование устройств ISA PnP и инициализация их параметров. Для PCI-устройств устанавливаются основные (стандартные) поля в блоке конфигурационных регистров.

0C – инициализируется блок переменных BIOS.

0D/0E – определяется наличие видеоадаптера путём проверки наличия сигнатуры 55AA по адресу начала Video BIOS (C0000:0000h). Если Video BIOS обнаружен и его контрольная сумма правильная, включается процедура

инициализации видеоадаптера. С этого момента появляется изображение на экране монитора, высвечивается заставка видеоадаптера, инициализируется клавиатура. Далее по ходу POST тестируется контроллер DMA и контроллер прерываний.

30/31 – определяется объём Base Memory и External Memory, и с этого момента начинается отображаемый на экране тест оперативной памяти.

3D – инициализируется PS/2 mouse.

41 – производится инициализация подсистемы гибких дисков.

42 – выполняется программный сброс контроллера жёстких дисков. Если в Setup указан режим AUTO, производится детектирование устройств IDE, в противном случае параметры устройств берутся из CMOS.

45 – инициализируется сопроцессор FPU.

4E – настраивается клавиатура USB. На данном этапе становится возможен вход в CMOS Setup по нажатию клавиши DEL.

4F – осуществляется запрос на ввод пароля, если это предусмотрено установками CMOS Setup.

52 – производится поиск и инициализация ПЗУ дополнительных BIOS, а также картируется каждая из линий запросов прерывания PCI.

60 – если в Setup включён данный режим, устанавливается антивирусная защита BOOT Sector.

62 – осуществляется автоматический переход на зимнее или летнее время, для клавиатуры настраиваются состояние NumLock и режим автоповтора.

63 – корректируются блоки ESCD (только для PNP BIOS) и производится очистка ОЗУ.

B0 – это состояние записывается в регистр сигнатурного анализатора только в случае наличия ошибок, например, при тесте Extended Memory.

FF – последний этап, на котором подводится итог тестирования – успешная инициализация аппаратных средств компьютера сопровождается одиночным звуковым сигналом, после чего осуществляется передача управления загрузчику BOOT-сектора.

ЛАБОРАТОРНЫЕ РАБОТЫ ПЕРВОГО СЕМЕСТРА

Введение

Лабораторные занятия проводятся в течение двух семестров. В первом семестре лабораторные работы проводятся в классе персональных ЭВМ. Они посвящены практическому освоению программирования микропроцессора x86 на пользовательском и системном уровнях, а также изучению элементов ЭВМ. В данном документе даны описания.

Список лабораторных работ первого семестра.

1 Введение в низкоуровневое программирование.
Встроенный отладчик IDE, встроенный Ассемблер, TurboDebugger.
2 Система команд процессора XX86 ее связь с кодами команд.
3 Способы адресации и сегментная организация памяти.
4 Подпрограммы и передача параметров.
5 Подпрограммы, программные прерывания и особые случаи.
6 FPU. Кодирование чисел с плавающей запятой. Особые численные значения. Особые случаи.
7 Обмен ЭВМ с клавиатурой.
8 Мультизадачность.

Лабораторная работа по теме №1. Введение в низкоуровневое программирование. Встроенный отладчик IDE. Встроенный Ассемблер, TurboDebugger

В качестве инструмента при выполнении лабораторных работ можно использовать любую систему программирования, позволяющую иметь доступ к ресурсам компьютера на уровне физических адресов ячеек и портов, а также к регистрам процессора. Для программирования и отладки

проще всего использовать средства Borland C, встроенный ассемблер, экранный отладчик TD.

Дается шаблон программы, предлагается изучить использование средств Ассемблера в программе, а также отладку, как на уровне исходного Си-текста, так и на уровне отладчика.

Цель работы: 1. Знакомство с программными средствами для работы с аппаратурой.

2. Изучение программной модели процессора x86

Информация: Описание программной модели процессора. Организация памяти. Пример программы. Используйте описания работы элементов среды Borland C.

Задание на выполнение работы

1. Прочитайте текст программы и выделите логически связные части, поймите, что делают эти части. Разбирая текст, обратите внимание на то, какие функции Си используются и зачем. Делая это, научитесь пользоваться контекстным Help'ом. Познакомьтесь, как включать элементы встроенного Ассемблера в текст программы.

2. Скомпилируйте программу и убедитесь, что в ней отсутствуют ошибки.

3. Запустите программу, наблюдайте результат и убедитесь, что Вы понимаете, что происходит.

4. Научитесь, используя встроенный отладчик IDE:

- Выполнить программу до заданной точки останова. На уровне исходного Си-текста это можно сделать, используя пункты меню Run и Debug/Breakpoints.
- Проверить/изменить содержимое переменных после останова. Используйте пункт меню Debug/Evaluate/Modify, Debug/Inspect (горячая клавиша Alt/F4) либо Debug/Watches
- Измерить время выполнения заданной преподавателем команды так, как это сделано в программе LAB0 с командой MOV REG,MEM.

5. Средствами отладчика TD в окне CPU (пункт меню View/CPU) научитесь:

- \- наблюдать/изменять содержимое регистров процессора (подумайте, какие регистры можно менять безболезнено (изменения в некоторых регистрах могут привести к фатальным для нормального выполнения программы результатам)
- \- наблюдать/изменять ячейку памяти с известным физическим адресом:

(Прежде, чем писать куда-либо, хорошо было бы подумать: что это за адрес - адрес ОЗУ(не использует ли этот адрес еще какая-нибудь программа), адрес ПЗУ, существует ли физическое устройство, соответствующее данному адресу...

Варианты:

1. 0x46C
   0x80000
   0xF000:0xFFF0
2. 0x0040:0x6D
   0x8000:0x0010
   0xFFFFE
3. 0x41A
   0x80210
   0xFFFF:0x100
4. 0x0040:0x1E
   0xD00000
   0x8000:0x200
5. область памяти - 0x41A-0x43C
   0xD000:0x100
   0xFFFF:0xE

- \- читать/писать в произвольный порт ввода/вывода (например: считать порт 0x40 несколько раз - попробуйте объяснить наблюдаемый

результат; записать число 3 в 61 порт, потом быстро записать туда нуль);

6. **Определите в какие команды Ассемблера оттранслирован с языка СИ оператор цикла.** Напишите на уровне Ассемблера свой вариант цикла. Сравните, результаты представьте преподавателю.

Лабораторная работа по теме N2

Система команд процессора XX86 ее связь с кодами команд.

Система команд любого процессора должна удовлетворять таким требованиям как удобство программирования и эффективный доступ к ресурсам. Набор команд определяется организацией самого процессора. Вам предлагается изучить и практически освоить систему команд микропроцессора типа i8086. Оцените ее: какие Вы видите достоинства и недостатки.

Работа включает семь этапов: изучение кодирования команд с использованием экранного отладчика TD, знакомство со специальными (особенными) командами процессора, работающими с адресами; выполнение строковых команд при работе с логическими адресами; передача данных из одного места физической памяти в другое, написание программы на языке ассемблера и контрольный вопрос на перекодирование команд.

Основное внимание уделено тому, чтобы студент осознал связь мнемоники команды с кодом, который лежит в память, и с действием команды, которое может вызвать изменения указателя команд, флагов, регистров, указателя стека и самого стека.

Цель работы: 1. Изучение системы команд процессора x86

2. Изучение способов кодирования команд.

Информация: Система команд. Формат двухадресной команды.

Используйте один из учебников по Ассемблеру и шаблоны программ.

ЗАДАНИЕ НА РАБОТУ

ЧТО НУЖНО ОСОЗНАТЬ:

1. Связь мнемоники с кодом и действием команды
2. Все изменения, которые происходят при выполнении команды

- указатель команд (переход ?)
- вычисление адресов

- вычисление результата

- флаги

- указатель стека и сам стек - (... может быть что-нибудь еще ??)

3. Как кодируется команда (КОП, постбайт, адресная часть, префиксы)

ЧТО НУЖНО СДЕЛАТЬ, чтобы убедиться, что Ваше "осознание" пунктов 1-3 совпадает с действительностью:

1. С помощью СИ + TD, а также любой книги с системой команд запишите и выполните команды одной из колонок и поймите результат.

	1	2	3	4	5
- арифметическую операцию	add	inc	sub	cmp	mul
- логическую операцию	or	not	xor	and	test
- команду передачи управления/ветвления	je	jb	jl	loop	jg
- команду сдвига	shl	sar	rol	ror	rcl

Для каждой команды:

а) Запишите команду в СИ, используя встроенный ассемблер.

Оттранслируйте. Перейдите в TD.

б) Определите двоичный код, выделите поля команды и заполните таблицу.

Используя разные способы обращения к операнду, представьте все поля команды.

2. Выполните операции пересылки массива в массив с использованием команд: LOOP, LEA.

3. Напишите программу пересылки данных из одного массива в другой с одной из строковых команд: LODS, MOVS, STOS с использованием префикса повторения REP и подготовки нужных регистров.

4. Напишите программу пересылки данных из одного массива в другой в модели памяти LARGE. Обязательно используйте команду LDS (LES). Укажите, как используются регистры.

5. Напишите программу с использованием строковых команд, которая выполняет действия по переносу содержимого массива, начинающегося с физического адреса B8000 в массив - с адреса B9000.

6. Разработайте блок-схему и напишите программу поиска символа в видеопамяти на первых 4-х видеостраницах.

7. Напишите программу на Ассемблере пересылки данных из одного массива в другой с использованием строковых команд. Программа должна выводить данные на экран компьютера. Используйте среду разработки AsmTool.

8. Ответьте преподавателю на вопросы вида:
Какой код имеет команда: ADD EAX, [EBX+4*ECX+4].
Какая команда закодирована кодом: 0202h.

Таблица двоичных кодов команд

	Мнемоника	Префикс	КОП	Постбайт адресации	Смещение	Непоср.операнд
1						
2						

Система команд

Для каждого типа ЭВМ своя система команд. Так для PC команды подробно описаны в данном пособии и литературе [1, 2, 13, 14] и нет необходимости в их подробном рассмотрении. Каждый из изучающих компьютер должен иметь соответствующую книгу по системе команд. Однако имеется много общего в командах для разных компьютеров, и это общее будет рассмотрено ниже.

Количество команд для разных типов ЭВМ колеблется от малых десятков до сотен. В таком множестве разобраться достаточно трудно, поэтому для рассмотрения команд их разбивают на группы (классифицируют). В разных книгах классификация сделана по-разному. Выделяют от 3 до >10 групп команд.

Формат двухоперандной команды

[Префикс] КОП [постбайт адресации] [смещение] [непоср.операнд]

Префикс Длина 1 байт. (Всего 5 префиксов для X86)

1.	Переназначения сегмента	**add es:[bx],ax**	
2.	Повторения действия для строковых команд	**REP, REPZ, REPNZ, REPNE**	rep movsb
3.	Блокировки	**Lock**	lock rep movsb
4.	Размера адреса		
5.	Размера операнда		

КОП - код операции. Длина 1 байт. 0-й бит КОП во многих (но не во всех) командах показывает, производится ли операция со словом (=1) или с байтом (=0). 1-й бит КОП в двухадресных командах указывает, какой из операндов является приемником.

Постбайт адресации_.. Длина 1 байт. Постбайт адресации показывает, где находятся операнды. Один из операндов (первый) может быть расположен в регистре (регистровая адресация) или в произвольной ячейке ОЗУ (все способы адресации кроме непосредственной). Второй операнд может находиться в теле команды (непосредственная адресация) или в регистре (регистровая адресация). Каждый из операндов может быть как источником так и приемником (за исключением непосредственной адресации: непосредственный операнд может быть только источником). Структура системы адресации несимметрична.

```
  7   6    5   4   3    2   1   0
!  mod  !    reg    !    r/m    !
!-----!------!----!----!----!----!-----!----!
```

Поля **mod** и **r/m** задают место расположения первого операнда. Поле **reg** задает положение второго операнда.

Значения поля mod:

11 - операнд в регистре (при остальных mod операнд в ОЗУ, а регистры, на которые указывают поля mod и r/m, содержат компоненты адреса операнда)

10 - смещение два байта (без знака)

01 - смещение один байт (со знаком)

00 - смещение в команде отстутствует

Смещение. Длина 1 байт (при mod-01) или 2 байта(при mod=10).

Непосредственный операнд. Длина 1 или 2 байта

Кодирование регистров полями reg и r/m при mod=11(т.е. при регистровой адресации) Табл.1

Кодирование способа вычисления адреса при адресации в память с использованием полей mod и r/m) Табл.2

reg или r/m	Байт	Слово		r/m	mod=00	mod=01 или 10
000	AL	AX		000	BX+SI	BX+SI+смещение
001	CL	CX		001	BX+DI	BX+DI+смещение
010	DL	DX		010	BP+SI	BP+SI+смещение
011	BL	BX		011	BP+DI	BP+DI+смещение
100	AH	SP		100	SI	SI+смещение
101	CH	BP		101	DI	DI+смещение
110	DH	SI		110	direct	BP+смещение
111	BH	DI		111	BX	BX+смещение

Смещение. Длина 1 байт (при mod-01) или 2 байта(при mod=10).

Непосредственный операнд. Длина 1 или 2 байта

Таким образом, длина команды лежит в пределах от 1 до 10 байт.

Пример программы вывода текстовой строки на Asm.

```
   DOSSEG
   .MODEL TINY
   .STACK 100h
   .DATA
Message  DB 13,10,'Hi Привет! ',13,10,'$'
   .CODE
   mov    ax,@Data
   mov    ds,ax              ; установить регистр DS таким
                             ; образом, чтобы он указывал
                             ; на сегмент данных
   mov    ah,9               ; функция DOS вывода строки
   mov    dx,OFFSET Message  ; ссылка на сообщение "Привет!"
   int    21h                ; вывести "Привет!" на экран
   mov    ah,4ch             ; функция DOS завершения
                             ; программы
   int    21h                ; завершить программу
   END
```

Лабораторная работа по теме N3

Способы адресации и сегментная организация памяти

Для кодирования местоположения операнда используются различные способы адресации. Выбор какого либо способа определяется задачей минимизации длины команды, доступом к операнду и удобством программирования. В процессоре типа i8086 используются следующие способы адресации:

1.Непосредственная адресация add ax,12

2.Абсолютная адресация. call proc

3.Регистровая адресация. add ax,bx

4.Косвенно-регистровая адресация. add ax,[bx]

5.Базовая адресация (адресация по базе)add ax,[bx+disp]

6.Базово-индексная add ax,[bx+si]

7.Базово-индексная со смещением add ax,[bx+si+disp]

8.Относительная адресация. je next-line

Си поддерживает шесть моделей памяти: крошечную, малую, среднюю, компактную, большую и огромную. Модель устанавливается в опциях. Выбранная модель может влиять на результат выполнения программы.

Цель работы: 1. Изучение особенностей способов адресации процессора.

2. Связь способов адресации и времени выполнения команды.

3. Влияние моделей памяти на результат выполнения программы.

Информация: Способы адресации.

ЗАДАНИЕ НА РАБОТУ часть 1

ЧТО НУЖНО ОСОЗНАТЬ:

1. Как кодируется способ адресации
2. Какова схема вычисления адреса
3. Какие узлы процессора (адресные регистры и т.п.) содержат компоненты адреса

4. Как влияет выбор данного способа адресации на время выполнения команды.

5. Как эмулировать команду со сложным способом адресации последовательностью более простых.

ЧТО НУЖНО СДЕЛАТЬ, чтобы убедиться, что Ваше "осознание" пунктов 1-5 совпадает с действительностью:

1. На примере одной из команд реализуйте все способы адресации (регистровая, косв.-регистр., индексная/базовая, базово-индексная, базово-индексная со смещением)

2. На примере одной команды измерьте время выполнения команды при разных способах адресации(add ax,bx; add ax,[bx+si+5]; add ax,[bx]; add [bx],ax). Определите тактовую частоту процессора, частоту работы ОЗУ, влияние КЭШ. Учитывайте время пустого цикла.

3. Напишите программу заполнения четных строк двумерного массива, находящегося в сегменте данных, числами с использованием двух вариантов адресации. Программа должна показывать преимущества многокомпонентных способов адресации или строковых команд (stos).

ЗАДАНИЕ НА РАБОТУ часть 2

ЧТО НУЖНО ОСОЗНАТЬ:

1. Как расположены сегменты памяти при трансляции программы в СИ.

2. Какие модели памяти используются.

3. Как работать с разными сегментами и как пересылать данные из одного сегмента в другой.

4. Как используются префиксы переназначения сегментов

ЧТО НУЖНО СДЕЛАТЬ:

1. Оттранслируйте пример (п.3),написанный на СИ, в двух моделях памяти - поймите что изменилось, а если ничего не изменилось, то почему?

2. Напишите программу на Асс, результат работы которой меняется в зависимости от используемой модели памяти. Например, чтение данных из сегмента кода, сделайте так, чтобы читались одинаковые или разные данные.

Тексты программ с комментариями сохраните (для предъявления преподавателю)/

Возможности измерения интервалов времени в Win

1. Команда RDTSC (Read Time Stamp Counter)

Команда возвращает количество тактов, прошедших с момента подачи напряжения или сброса процессора.

Команда RDTSC состоит из двух байтов: $0F 31. Она возвращает в регистры EDX:EAX 64-битное значение time-stamp счетчика. Счетчик хранится в 64 битном регистре MSR.

Выполнение инструкции RDTSC доступно с любого уровня привилегий, при условии, что в регистре CR4 флажок "Time stamp disable" установлен на "0". В противном случае выполнение данной инструкции доступно только с нулевого уровня привилегий.

2. Функция API SetTimer().

Лабораторная работа по теме N4

Подпрограммы и передача параметров.

Модульность программирования на аппаратном уровне поддерживается организацией механизма подпрограмм. Этот механизм включает следующие элементы: команды обращения к ПП, передачу параметров, запоминание состояние процессора в стеке, программное прерывание и др. Аппаратные средства должны поддерживать вложенность ПП, быстрое обращение к ПП, обращение к ПП, расположенных в любых участках памяти ЭВМ. В работе предлагается ряд заданий, которые позволяют изучить и практически освоить работу с подпрограммами.

Цель работы: 1. Изучение особенностей команд работы с подпрограммами процессора x86.

2. Освоение методов передачи параметров в подпрограмму. Анализ использования стека.

Информация: Связь с ПП по управлению. Косвенный вызов процедуры. Передача параметров в ПП. Организация стека..

ЗАДАНИЕ НА РАБОТУ

ЧТО НУЖНО ОСОЗНАТЬ (дать ответы на вопросы себе и преподавателю):

1. Как организована связь с ПП по управлению:
   - команды call, ret;
   - внутрисегментная передача, межсегментная передача, косвенная и прямая адресация.
2. Как используется стек для сохранения адреса возврата и состояния процессора. Структура стекового кадра.
3. Передача параметров (связь по данным) с использованием:
   - регистров
   - стека
   - общей области памяти (как еще ?).

ЧТО НУЖНО СДЕЛАТЬ:

1. Оформите одно из заданий, приведенных ниже, в виде подпрограммы с параметрами на встроенном ассемблере в среде СИ. Операнды находятся в памяти (отведите для них место, используя средства языка Си). Разрядность операндов задайте как параметр.

Ваш фрагмент должен не только выполнять требуемое действие (формировать результат операции), но и обеспечивать верные значения флагов (такие же, как при выполнении соответствующей "короткой" команды из системы команд процессора). Вы можете "отследить" не все флаги, достаточно будет, если Вы обеспечите верные значения для двух флагов. Возьмите обязательно ZF и какой-нибудь второй по собственному выбору (но только такой, который ведет себя при выполнении команды нетривиально).

Смотрите в качестве прототипа шаблон программы эмуляции арифметического сдвига вправо.

2. Реализуйте обращение к ПП с передачей параметров через стек:
- внутрисегментное (модель памяти Tiny),
- межсегментное (модель памяти Large).

Приведите таблицу заполнения стека в обоих случаях. Отметьте отличия.

3. Реализуйте передачу параметров в ПП и результата из ПП с использованием:
- регистров,
- общей области памяти.

4. Переключите режим команд при трансляции программы с 8086 на 286. Определите с помощью TD какие дополнительные команды будут включены в программный код. В чем смысл использовагия этих команд. (enter - создание стекового кадра; leave - выход из процедуры высокого уровня)

Задания по теме ПОДПРОГРАММЫ:

1. Напишите фрагмент программы на встроенном ассемблере - сдвига арифметического вправо.
2. Напишите фрагмент программы на встроенном ассемблере. Сдвиг арифметический влево
3. Напишите фрагмент программы на встроенном ассемблере. Сдвиг логический вправо
4. Напишите фрагмент программы на ассемблере - инкремент двойного слова. Сравните Ваш способ с тем, как это делает компилятор.
5. Напишите фрагмент программы на ассемблере - декремент двойного слова. Посмотрите, как это делает компилятор.
6. Напишите фрагмент программы на ассемблере - смена знака двойного слова.
7. Напишите фрагмент программы на ассемблере - сложение двух двойных слов.
8. Напишите фрагмент программы на ассемблере - сравнение двух двойных слов.

Для шустрых:

9. Напишите фрагмент программы на ассемблере - умножение двух двойных слов.

Пример шаблона программы 1.

```
/*=============================================================*\
Эмуляция сложных (несуществующих) команд
   Студенты гр 000.0 Иванофф, Петрофф, & Сидорофф
Задание: эмуляция команды арифметического сдвига вправо операнда
с повышенной разрядностью. Операнд с повышенной разрядностью
находится в массиве iMas1, количество слов в длинном операнде содержится
в переменной iNWords, число разрядов, на которое надо сдвинуть длинный
операнд, содержится в переменной iNShift.
\* ============================================================ */
//  Глобальные переменные
int iNWords=3;        // Количество слов в операнде
```

```
int iNShift=4;          // На сколько битов сдвинуть
int iMas1[10]={0x1234, 0x2000,0x1000,0x800,0x400,0x200};
                    // Массив для  операнда

void main(void)
{       //  Используем регистр SI для перебора слов длинного операнда
        //  Заносим начальный адрес операнда в адресный регистр
    asm mov bx, iNShift   //  Заносим счетчик сдвигов
      //  Далее пойдет цикл, на каждой итерации которого выполняется
      // сдвиг длинного операнда на один бит
 cykl_po_sdwigam:    // ----------------------------------------
   asm{                    //                              |
      mov cx, iNWords             //  Заносим счетчик слов          |
      mov si, offset iMas1//  Прибавим к адресу младшего слова|
        add si, iNWords        // сдвигаемого операнда удвоенную   |
        add si, iNWords        // длину операнда                 |
                          //  Получили адрес старшего слова+2 |
                      //                              |
      test word ptr[si-2],0x8000//  Проверяем старшее слово (при  |
                      // этом С-бит всегда очищается      |
      jns cykl_po_slowam      //  Если отрицательно,            |
      stc                   // устанавливаем С-бит для верного  |
      }                   // расширения знака при сдвиге      |
                      //                              |
/* -------------------------------------------------------- *\   |
      mov ax, [si]   - Это другой вариант расширения знака в С    |
      rcl ax                                        |
\* -------------------------------------------------------- */
//  Теперь цикл по словам длинного операнда               |
  cykl_po_slowam:    // ---------------------------------------  |
    asm {                 //                             | |
          dec si; dec si        //  Модифицируем адрес          | |
          rcr word ptr[si],1  //  Сдвигаем циклически         | |
          loop cykl_po_slowam   //  Организуем цикл             | |
         }          // ---------------------------------------  |
    asm dec bx                       //  Считаем сдвиги               |
   asm jg cykl_po_sdwigam  // -----------------------------------
     //  Определяем признак "нуля"
   asm mov cx, iNWords          //  Заносим счетчик слов
   asm xor dx, dx             //  Для признака нуля
zero:
   asm {
       or dx, [si]          //  Признак "нуля"
```

```
      inc si; inc si        //  Модифицируем адрес
     loop zero
    }
  asm test dx,0xFFFF        //  Признак "нуля"
  asm nop}
```

Пример шаблона программы 2.

```
// Тема         - ПОДПРПОГРАММЫ. Передача параметров через стек.
#include <stdio.h>
#include<conio.h>
#include<iostream.h>
/*
При вызове функции в Си, она интерпретируется как команда Call в
Assembler. При этом передаваемые в функцию параметры, заносятся в СТЕК
по принципу справа налево. Затем заносится IP следующей команды, идущей
за Call. (внутрисегментный вызов процедуры)
Выход из функции осуществляется с помощью команды RET.
*/

//Функция, выполняющая сложение, параметры перед через стек
void addition(long sl_1,long sl_2, long &sum)
{
/*   asm{
      mov    AX,[BP+4]       //AX - младшие разряды первого слагаемого
      mov    DX,[BP+8]       //DX - младшие разряды второго слагаемого
      mov    SI,[BP+12]      //SI - сумма младших разрядов
      add    AX,DX          //сложение младших разрядов
      mov    [SI],AX        //результат косвенно в SI
      mov    AX,[BP+6]       //AX - старшие разряды первого слагаемого
      mov    DX,[BP+10]      //DX - младшие разряды второго слагаемого
                 //сложение старших разрядов
      mov    [SI+2],AX       //результат в SI
   }*/
}
void main ( void )
{long a,b;
long sum;
cout<<"\n Введите первое слагаемое:  ";
cin>>a;
cout<<" Введите второе слагаемое:  ";
cin>>b;
addition(a,b,sum);
cout<<"a + b =  "<<sum;
```

```
getch();
}
```

Связь с ПП по управлению

Обращение к подпрограмме и выход из нее выполняется двумя командами: CALL, RET. При обращении / возврате надо обеспечить:

а) передачу управления в любое место памяти, поэтому "длинная" адресация;

б) возврат в то место, откуда был вызов (т.е. место вызова должно автоматически запоминаться)

в) запоминание промежуточных результатов, имеющихся к моменту вызова (содержимое регистров процессора, а при рекурсивном вызове процедуры надо запоминать и промежуточные результаты работы самой процедуры).

Для запоминания всего этого используется стек (участок ОЗУ или специальное ОЗУ со стековой адресацией). Часть вышеперечисленной информации запоминается автоматически при выполнении команды CALL, а сохранение оставшегося - дело программиста.

Внутрисегментная команда CALL (NEAR) осуществляет передачу управления только внутри текущего 64-килобайтного командного сегмента. В стеке сохраняется содержимое только регистра IP.

Межсегментная команда CALL (FAR) может передавать управление любой процедуре в пределах мегабайтового адресного пространства микропроцессора. В стеке сохраняется содержимое регистров CS и IP.

Если процедура имеет (определена) атрибут NEAR , то команда CALL помещает смещение адреса следующей команды в стек. Если процедура имеет атрибут FAR, то команда CALL помещает в стек содержимое регистра CS, а затем смещение адреса.

После сохранения адреса возврата команда CALL загружает смещение адреса метки вызываемой процедуры в указатель команд IP. Если процедура имеет атрибут FAR, то загружается также номер блока метки вызываемой процедуры в регистр CS.

Команда RET заставляет микропроцессор возвратиться из процедуры в программу, вызывающую эту процедуру. RET извлекает адрес возврата из стека. Если процедура имеет атрибут NEAR, то команда RET извлекает из стека одно слово и загружает его в указатель команд IP. Если процедура имеет атрибут FAR, то команда RET извлекает из стека два слова: сначала смещение адреса для загрузки в указатель команд IP, а затем номер блока для загрузки в регистр CS.

Косвенный вызов процедуры.

При прямом вызове процедур операндом является метка начала процедуры. Но можно осуществить и косвенный вызов процедуры черед регистр или ячейку памяти. При косвенных вызовах черед ячейку памяти микропроцессор извлекает значение указателя команд IP для процедуры из сегмента данных, если только не используется регистр BP или нет замены сегмента. Если для адресации ячейки памяти используется регистр BP, то микропроцессор извлекает значение указателя команд из сегмента стека.

Процедуру с атрибутом NEAR можно вызвать косвенно, используя переменную размером в слово.

CALL WORD PTR [BX]

CALL WORD PTR VARIABLE-NAME

Процедуру с атрибугом FAR можно вызвать косвенно, используя переменную размером в двойное слово.

CALL DWORD PTR [BX]

CALL DWORD PTR VARIABLE-NAME

Передача параметров в ПП. Организация стека.

Передачу параметров в ПП и результата из ПП можно осуществить, через:

регистры;

стек:

общую область памяти.

Стек удобен для временного хранения данных (содержимого регистров и ячеек памяти). Вершина стека - ячейка в сегменте стека, адрес которой содержится в указателе стека SP. Т.к. стек "растет" по направлению к младшим адресам памяти, то первое помещаемое в стек слово запоминается в ячейке стека с наибольшим адресом, следующее на 2 байта ниже и т.д. Регистр SP всегда указывает на слово, помещенное в стек последним. Например, если ассемблерная ПП вызывается из СИ:

RES=AAA(a1,a2) // a1.a2 - имеют тип long

То стек будет выглядеть следующим образом:

для функции с атрибутом FAR	для функции с атрибутом NEAR	комментарий
sp+0A	sp+8	a2 старшее слово
sp+8	sp+6	a2 младшее слово 1
sp+6	sp+4	a1 старшее слово 1
sp+4	sp+2	a1 младшее слово 1
sp+2	нет	регистр CS вызывающей программы
sp	sp	регистр IP вызывающей программы

Лабораторная работа по теме N5

Подпрограммы, программные прерывания и особые случаи.

Действия при прерывании могут быть использованы и для перехода к ПП, предусмотренного программистом (программные прерывания). Для этого в систему команд вводится команда программного прерывания int. Можно вызвать ПП и через организацию особого случая. Прерывания и особые случаи подобны в том, что они заставляют процессор временно приостановить выполнение текущей программы для перехода к ПП. Прикладные программисты обычно не касаются обработки особых случаев, их обрабатывает операционная система. Однако некоторые типы особых случаев имеют отношение к прикладному программированию, и многие операционные системы предоставляют возможность прикладным программам их обрабатывать.

Предлагается реализовать вызов ПП через программное прерывание и особый случай, проанализировать заполнение стека.

Цель работы: 1. Изучение особенностей команд программного прерывания.

2. Освоение методов вызова ПП через программное прерывание и особый случай. Анализ использования стека.

Информация: Типы программных прерываний. Вызов прерывания.

ЗАДАНИЕ НА РАБОТУ

ЧТО НУЖНО ОСОЗНАТЬ

1. Программное прерывание как средство вызова ПП, достоинства и недостатки.

Команда int - возбуждает программное прерывание, позволяющее передать управление программе обработки прерывания;

Команда iret - возврат управления из программы обработки прерывания

2. Вектор прерывания. Разрешенные и запрещенные вектора.

3. Какие регистры запоминаются в стеке.

4. Отличие от аппаратного прерывания

(В аппаратном прерывании:

- разрешение / запрет прерывания в регистре флагов и во внешнем устройстве;

- необходимость запоминания в стековом кадре состояния всех используемых регистров;

- возможность программирования вложенности прерываний)

5. Отличия особых случаев от других источников прерываний.

6. Каковы источники особых случаев.

ЧТО НУЖНО СДЕЛАТЬ

1. Реализуйте выполнение вашей ПП через программное прерывание для чего: запомните старый вектор прерывания, установите новый, введите параметры ПП, выполните прогр. прерывание, обработайте данные в ПП (в программе обработке прерывания), выйдите в основную программу, выведите результат.

2. Проанализируйте заполнение стека.

3. Реализуйте выполнение вашей ПП путем создания особого случая (деление на нуль), соответствующему одному из первых десяти векторов.

4. Проанализируйте заполнение стека. При выходе из обработчика прерываний пропустите команду, вызывающую особый случай
(add [bp+18], 2).

5. Покажите использование команды int3, определите ее код с использованием TD

Типы программных прерываний.

Подобно вызову процедуры прерывание заставляет микропроцессор сохранить в стеке информацию для последующего возврата, а затем перейти к группе команд, находящейся в некоторой ячейке памяти. Но при вызове процедуры микропроцессор исполняет процедуру, а при прерывании программу обработки прерывания.

В то время как вызываемая процедура может иметь атрибуты NEAR и FAR, а ее вызов будет прямым или косвенным, прерывание всегда вызывает косвенный переход к своей программе обработки за счет получения ее адреса из вектора прерывания (32-битовой ячейки памяти). Кроме того, вызовы процедуры сохраняют в стеке только адрес, а прерывание еще и флаги.
У микропроцессора есть 3 команды прерывания INT, INTO, IRET.

При исполнении команды INT микропроцессор производит следующие действия:

1. Помещает в стек регистр флагов.
2. Обнуляет флаг трассировки TF и флаг включения-выключения прерываний IF для исключения пошагового режима исполнения команд и блокировки других маскируемых прерываний.
3. Помещает в стек значение регистра CS.
4. Вычисляет адрес вектора прерывания, умножая тип прерывания на 4.
5. Загружает второе слово вектора прерывания в регистр CS.
6. Помещает в стек значение указателя команд IP.
7. Загружает в указатель команд IP первое слово вектора прерывания.

После исполнения команды INT в стеке окажутся значения регистра флагов и регистров CS и IP, флаги TF и IF будут равны 0, а пара регистров CS:IP будут указывать на начальный адрес программы обработки прерывания. Затем микропроцессор начнет исполнение этой программы.

Имеется 256 векторов прерываний, которые размещаются в области памяти с младшими адресами. Т.к. каждый из них имеет длину 4 байта, то все они занимают первые 1К байтов (область памяти с абсолютным адресом от О до 3FFH).

Команда IRET по отношению к прерыванию выполняет ту же роль, что и команда RET для вызова процедуры. Она "откатывает" всю работу исходной операции и заставляет микропроцессор аккуратно выполнить возврат к основной программе. Команда IRET извлекает из стека 3 16-

битовых значения и загружает их в указатель команд IP, регистр сегмента команд CS и регистр флагов соответственно.

Вызов прерывания.

Для генерации внутреннего прерывания программы в СИ используется встроенная функция Geninterrupt(). Чтобы использовать программное прерывания как средство вызова ПП надо:

1. Сохранить старое значение вектора прерывания.

Old-f0=getvect(0xF0)

2. Установить новую функцию обработки прерывания для вектора прерывания. setvect(0xF0,add)

3. Вызвать процедуру генерации программного прерывания.

geninterrupt(0xF0)

4. Восстановить старое значение вектора. setvect(0xF0,Old-f0)

Пример программы с командой программного прерывания

```
*/#include<stdio.h>
#include<dos.h>
#include<iostream.h>
#include <conio.h>
void interrupt (*old)(...);     // здесь будем сохранять старый вектор
void interrupt cmp_int(...)     // а это наш обработчик
{
cout<<"Прерывание  ";
getch();
}
void main(void)
{ int aa;
  aa=1;
  old=getvect(0xf0);
  disable();
  setvect(0xf0,cmp_int);
  enable();
geninterrupt(0xf0);
  puts("v1=v2");
  setvect(0xf0,old);
  return;}
```

Лабораторная работа по теме N6

FPU. Кодирование чисел с плавающей запятой. Особые численные значения. Особые случаи.

Цель работы: 1. Изучение особенностей команд чисел с плавающей запятой.

2. Освоение методов вызова ПП через особый случай чисел с плавающей запятой.

Информация: Прерывания и особые случаи. Численные особые случаи при обработке команд FPU. Кодирование вещественных чисел с одинарной и двойной точностью. Формат слова управления FPU. Слово состояния FPU. Команды управления.

ЗАДАНИЕ НА РАБОТУ

ЧТО НУЖНО ОСОЗНАТЬ

1. Формат чисел с плавающей запятой.
2. Регистровая модель и команды работы с регистрами FPU
3. Как работает FPU при обработке особых случаев.

ЧТО НУЖНО СДЕЛАТЬ

1. Напишите программу на Асс (см пример): Введите в диалоге три числа, загрузите их в регистры FPU, просуммируйте, выведите результат. Проследите работу FPU в TD. Как используются регистры FPU ?

2. Получите в FPU нечисло (NaN), покажите его в TD.

3. Прочитайте и проанализируйте состояния регистров FPU в TD: регистра состояния и регистра управления.

4. Напишите программу, выполняющую команды FPU, вызывающую и реагирующую на особый случай FPU (обработчик прерывания сообщает о наступлении особого случая по результату анализа регистра состояния).

5. При сдаче работы покажите преподавателю механизм перевода числа, например 16.02, в формат с плавающей запятой.

Фрагмент программы работы с FPU как со стеком.

// Операции с плавающей запятой на уровне Ассемблера

```
asm {   finit
        fld ad
        fld bb
        fadd
        fstp bb
}
cout<<"   Acc  "<<bb;
```

Численные особые случаи при обработке команд FPU

FPU распознает следующие шесть классов условий численных особых случаев при выполнении численных команд:

I - недействительная операция: ошибка стека, недействительная операция для стандарта IEEE;

Z - деление на нуль;

D - денормализованный операнд;

O - численное переполнение;

U - численное антипереполнение;

P - неточный результат.

При возникновении численных особых случаев процессор выполняет одно из следующих действий:

-FPU может сам обрабатывать особый случай, формируя наиболее приемлемый результат и разрешая численной программе продолжать выполнение;

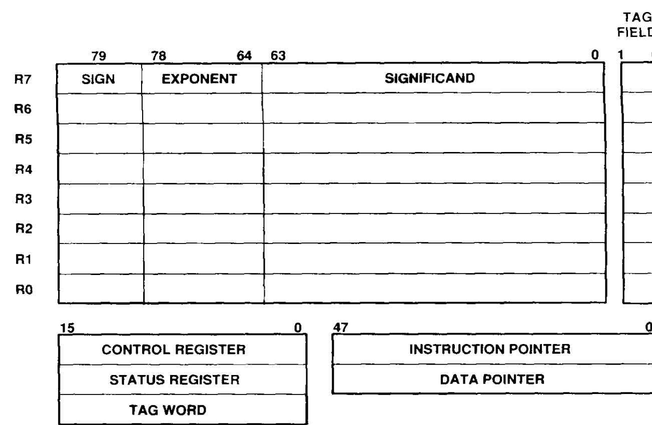
-осуществляется вызов программного обработчика особого программного случая.

Каждое из условий шести особых случаев, указанных выше, имеет соответствующий бит в слове состояния FPU и бит маски в слове управления. Если особый случай маскируется, то процессор по умолчанию выполняет соответствующее действие и продолжает работу. Если особый случай замаскирован (0 в маске регистра управления), то обработчик особого случая немедленно вызывается перед выполнением следующей команды

WAIT или неуправляющей команды FPU. В зависимости от значения бита NE в регистре управления CR0 процессора обработчик исключения вызывается или через вектор прерывания 16(NE=1), или посредством внешнего прерывания 0x75 (NE=0).

Когда особые случаи замаскированы, FPU может обнаружить множество особых случаев в одной команде, так как продолжает выполнение команды после завершения действий на маскируемый случай.

Архитектура FPU

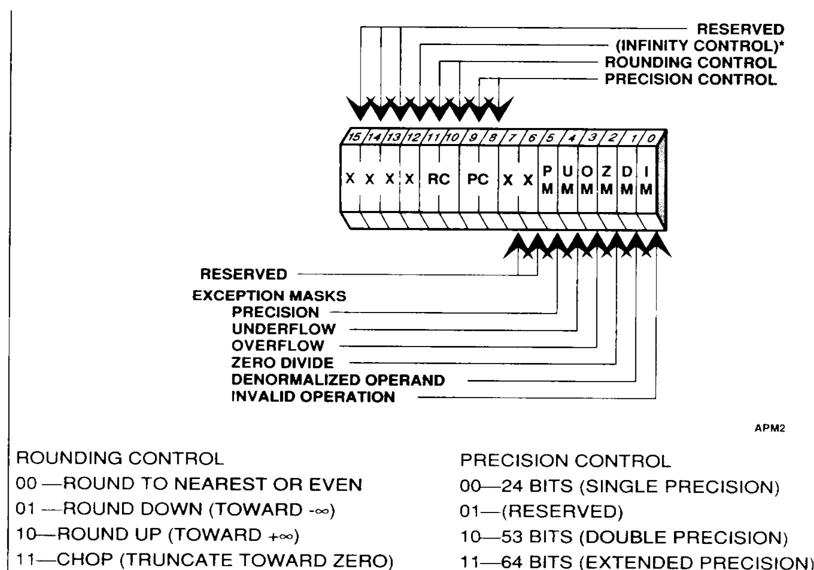


Процессор с плавающей точкой включает следующие регистры:

- восемь 80-битных регистров, организованных как регистровый стек;

- три 16-битных регистра, содержащих слова управления FPU, слово состояния FPU, слово тэгов; - указатели ошибок.

Формат слова управления FPU



FPU позволяет использовать несколько режимов обработки чисел с плавающей точкой (см. табл), которые выбираются при загрузке слова управления.

Команда FSTCW запоминает слово управления. Команда FLDCW загружает слово управления.

15,14,13 - Зарезервированные

12 - Управление бесконечностью*

11,10 - Управление округлением

9,8 - Управление точностью

7,6 - Зарезервированные

МАСКИ ИСКЛЮЧЕНИЙ

5 - Точность *

4 - Антипереполнение

3 - Переполнение -

2 - Деление на нуль

1 - Денормализованный операнд

0 - Недопустимая операция

Управление округлением

00 - округление до ближайшего или до четного

01 - округление вниз (к минус бесконечности)

10 - округление вверх (к плюс бесконечности)

11 - усечение (отбрасывание разрядов)

Управление точностью

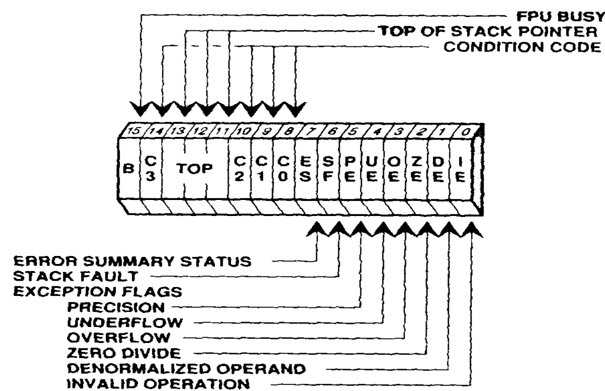
00 - 24 бита (обычная точность)

01 - (зарезервировано)

10 - 53 бита (двойная точность)

11 - 64 бита (расширенная точность)

Слово состояния FPU



Слово состояния FPU отражает полное состояние устройства с плавающей точкой. Команда FSTSW AX запоминает слово состояния непосредственно в регистре AX, разрешая процессору просматривать эти коды условия.

15 - FPU занято

13,12,11 - Вершина указателя стека

14,10,9,8 - Код условия

7 - Суммарное состояние ошибок Бит ES(7) устанавливается, если любой немаскированный бит особого случая установлен, иначе он очищается.

6 - Ошибка стека -*

ФЛАГИ ОСОБЫХ СЛУЧАЕВ

5 - Точность ---

4 - Отрицательное переполнение

3 - Переполнение ---

2 - Деление на нуль -*

1 - Денормализованный операнд

0 - Недействительная операция

Значения TOP (биты 13,12,11)таковы:

000 - регистр 0 - вершина стека

001 - регистр 1 - вершина стека

....

111 - регистр 7 - вершина стека

Четыре бита кода условия FPU (С3 - С0) подобны флагам Центрального процессора: процессор модифицирует эти биты, чтобы отразить результат арифметических операций. Эти биты кода условия используются для условного ветвления.

Команды управления

Команды управления FPU показаны в табл. Они обеспечивают управление FPU на системном уровне. Эти действия включают инициализацию FPU, обработку численных особых случаев и переключение задач. Команды, которые инициализируют FPU, сбрасывают особые случаи или запоминают всю или частично операционную среду FPU, можно представить в следующих двух модификациях:

-ожидать (WAIT) - мнемонике предшествует только один символ F . Эта форма команды проверяет немаскируемые численные особые случаи;

-не ожидать (NO-WAIT) - этой мнемонике предшествует два символа FN. Данная форма игнорирует немаскируемые численные особые случаи.

Таблица 2 Команды управления

Мнемоника	Операция
FINIT/FNINIT	Инициализация FPU
FLDCW	Загрузить слово управления
FSTCW /FNSTCW	Запомнить слово управления
FSTSW/FNSTSW	Запомнить слово состояния
FSTSW AX/FNSTSW AX	Запомнить слово состояния в регистре АХ
FCLEX/FNCLEX	Сбросить особые случаи
FSTENV/FNSTENV	Запомнить среду
FLDENV	Загрузить среду
FSAVE/FNSAVE	Запомнить полное состояние
FRSTOR	Восстановить полное состояние
FINCSTP	Инкремент указателя стека
FDECSTR	Декремент указателя стека
FFREE	Освободить регистр
FNOP	Нет операции

FWAIT Ожидание

Для процессора Pentium целочисленное устройство и устройство с плавающей точкой работают параллельно, возможно, что при возникновении особого случая для операций с плавающей точкой процессор разрушит информацию, необходимую для обнаружения особого случая прежде, чем обработчик подобных исключений будет вызван. Использование команды WAIT или FWAIT в требуемом месте программы может предотвратить такую ситуацию.

Таблица 3 Кодирование вещественных чисел с одинарной и двойной точностью

Класс	Знак	Смещенный порядок	Мантисса
Положительные нечисла			
QNaN	0	11..11	11..11
	O	11..11	10..00
SNaN	O	11..11	01..11
	0	11..11	00..01
Бесконечность	0	11..11	00..00
Положительные вещественные			
Нормализованные	0	11..10	11..11
	0	00..01	OO..OO
Денормализованные	O	00..00	11..11
	0	00..00	00..01
Отрицательные вещественные			
Денормализованные	1	00..00	00..01
	1	00..00	11..11
Нормализованные	1	00..01	00..00
Бесконечность	0	11..11	00..00
Отрицательные нечисла			
SNaN	1	11..11	01..11
	1	11..11	00..01
QNaN	1	11..11	11..11

```
     1      11..11     10..00

Одинарный:  8битое - 23 бита-
 Двойной:  11битов-  52 бита-
```

Лабораторная работа по теме N7

Обмен ЭВМ с клавиатурой

В данной работе предлагается изучить программную модель клавиатуры и экспериментально исследовать возможности, которые предоставляет клавиатура пользователю. Рассматриваются: коды, которые передаются из блока клавиатуры в процессорный блок; вопросы синхронизации во времени между нажатиями клавиш и действиями процессора; организация статусных байтов и буфера клавиатуры в ОЗУ; управление клавиатурой со стороны процессора.

Предлагается запустить программу scancode.com и наблюдать значения скан-кодов, которые соответствуют нажатиям и опусканиям клавиш. Запустить программу keyview.com, моделирующую поведение буфера клавиатуры, и наблюдать перемещение указателей головы и хвоста, а также скан-код клавиши и ASCII-код символа, которые заносятся в буфер. В итоге следует написать и отладить программу, которая выполняет следующую последовательность действий (не используя функции DOS или BIOS): ждет нажатия клавиши, читает скан-код нажатой клавиши, ASCII-код нажатой клавиши, адрес в буфере клавиатуры, с которого расположены эти коды. Программа должна показать, что происходит при нажатии клавиш Shift, Ctrl, Alt .

Цель работы: 1. Экспериментально исследовать возможности, которые предоставляет клавиатура пользователю.

Информация: Структура подключения клавиатуры. Порты для работы с клавиатурой. Описание команд.

ЗАДАНИЕ НА РАБОТУ

1. Вопросы:

1.Разберитесь, как организована связь клавиатуры с процессором в IBM PC (TechHelp, topic: AT Keyboard functions.) и выясните:

1.1. Что (какие коды) передается из блока клавиатуры в процессорный блок (и куда именно: в какие программно доступные узлы) ?

1.2. Как организована синхронизация во времени между нажатиями клавиши действиями процессора ?

1.2.1. Как процессор "узнает", что нажата очередная клавиша ?

1.2.2. Как контроллер клавиатуры узнает, что процессор считал предыдущий код, и можно посылать следующий ?

1.2.3. Что произойдет, если клавиши нажимаются быстро одна за другой, а процессор по каким-то причинам "не успевает" обрабатывать эти нажатия ?

1.3. Как организован буфер клавиатуры в ОЗУ ?

- Что записывается в буфер клавиатуры по нажатию клавиши ?
- Какая программа это делает ?
- Что происходит при нажатии клавиш Shift, Ctrl, Alt ?
- Что происходит при нажатии клавиш CapsLock, NumLock, ScrollLock ?

Каким образом (и кто?) "узнает" о том, что незадолго до этого была нажата одна из этих клавиш

1.4. Какие есть возможности по управлению клавиатурой со стороны процессора ?

2.Методическое задание:

1. Придумайте способ (методику), позволяющий наблюдать (с помощью экранного отладчика либо с помощью написанной Вами программы) "поведение" буфера клавиатуры при нажатии выбранных Вами клавиш (эти клавиши будем называть экспериментальными). Затруднение состоит в том, что для управления отладчиком тоже приходится манипулировать клавишами (управляющие нажатия).

2. Придумайте, как отличить управляющие нажатия от экспериментальных. Для нескольких экспериментальных нажатий

зафиксируйте соответствующие изменения в буфере клавиатуры и проинтерпретируйте их.

3. Практическое задание 1:

3.1. Запустите программу scancode.com и и наблюдайте значения скан-кодов, которые соответствуют нажатиям и опусканиям клавиш.

3.2. Запустите программу keyview.com, моделирующую поведение буфера клавиатуры и наблюдайте перемещение указателей головы и хвоста, а также скан-код клавиши и ASCII-код символа, которые заносятся в буфер.

3.3. Напишите и отладьте программу, которая выполняет следующую последовательность действий (не используя функции DOS или BIOS):

а) ждет нажатия клавиши (каким способом она может узнать, что нажата клавиша.)

б) по нажатию клавиши читает (откуда можно читать?)

-скан-код нажатой клавиши

-ASCII-код нажатой клавиши

-адрес в буфере клавиатуры, с которого расположены эти коды

в) выводит указанные значения на экран.

4. (Задание для шустрых) Напишите программу, которая выполняет ту же функцию, что и в п.3.3, но не читает буфер клавиатуры, а вызывается по прерыванию от клавиатуры. Программа вначале должна "перехватить" вектор прерывания типа 9, а перед завершением восстановить его. Если это задание вызовет трудности, Выполните его позже, когда подробнее познакомитесь с контроллером прерываний.

5. Практическое задание 2:

Напишите программу, которая запрещает прерывание от клавиатуры на заданное время (например на 10 с), при этом другие прерывания не должны быть запрещены, например, должны идти системные часы.

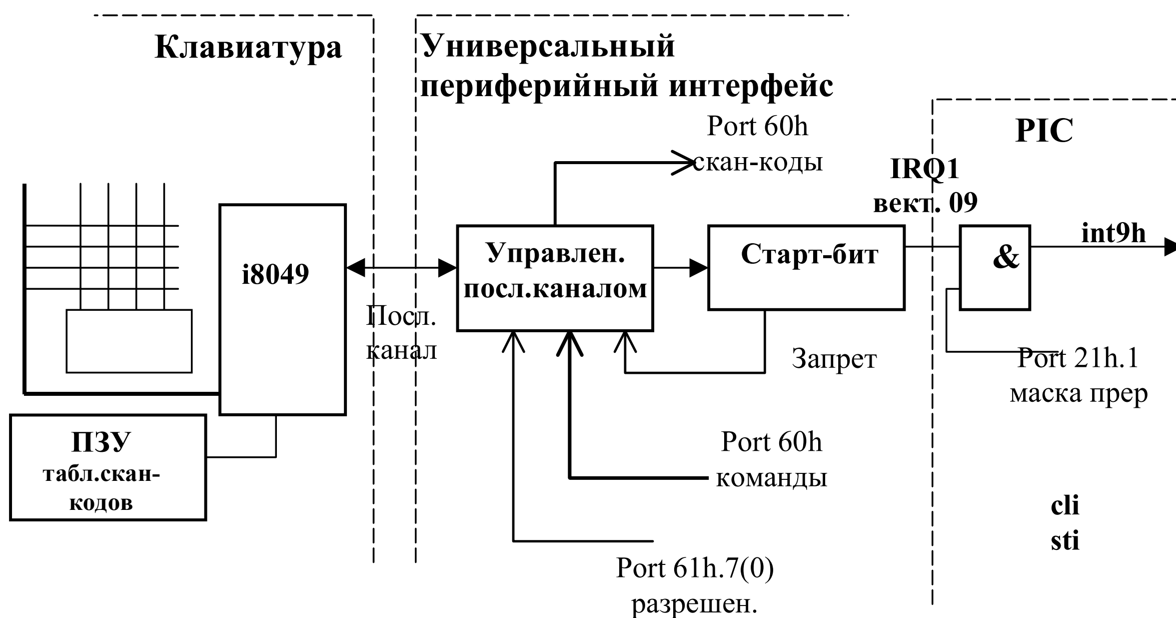
6. Практическое задание 3:

Напишите программу, которая по нажатию выбранных Вами клавиш зажигает/гасит индикаторы CapsLock, NumLock, ScrollLock.

7. Практическое задание 4:

Напишите программу, которая запрашивает с клавиатуры значения задержки и частоты автоповтора, а затем устанавливает соответствующие величины для клавиатуры.

Схема подключения клавиатуры



Все горизонтальные линии матрицы подключены через резисторы к источнику питания +5 В. Клавиатурный компьютер (однокристалльный контроллер 8049) имеет порта - входной и выходной. Входной порт подключен k горизонтальным линиям матрицы, а выходной - вертикальным. Устанавливая но очереди на каждой из вертикальных линий уровень напряжения, соответствующий логическому нулю клавиатурный компьютер опрашивает состояние горизонтальных линий. Если нажатых клавиш нет, уровень напряжений на всех горизонтальных линиях соответствует логической 1.

Если оператор нажмет на какую-либо клавишу, то соответствующие вертикальная и горизонтальная линии окажутся замкнутыми. Когда на этой вертикальной линии процессор установит значение логического нуля, то

уровень напряжения на горизонтальной линии также будет соответствовать логическому нулю, Как только на одной из горизонтальных линий появится уровень логического нуля, клавиатурный процессор фиксирует нажатие на клавишу. Он посылает в центральный компьютер запрос на прерывание и номер клавиши в матрице. Аналогичные действия выполняются, когда оператор отпускает нажатую ранее клавишу.

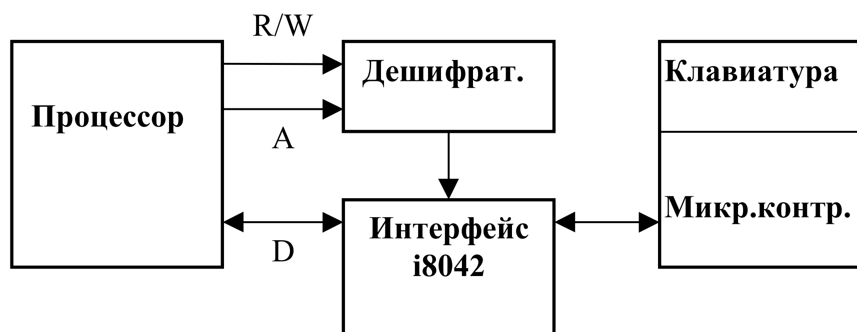
Номер клавиши, посылаемый клавиатурным процессором, однозначно связан с распайкой клавиатурной матрицы и не зависит от обозначений, нанесенных на поверхность клавиш. Этот номер называется скан-кодом (Scan Code). Слово scan ("сканирование") подчеркивает, что клавиатурный компьютер сканирует клавиатуру для поиска нажатой клавиши. Но программе нужен не порядковый номер нажатой клавиши, а соответствующий обозначению на этой клавише ASCII-код. Этот код не зависит однозначно от скан-кода, т. к, одной и той же клавише могут соответствовать несколько значений ASCII-кода. Это зависит от состояния других клавиш. Например, клавиша с обозначением '1' используется еще и для ввода сим вола '!' (если она нажата вместе с клавишей SHIFT). Поэтому все преобразования скан-кода в ASCII-код выполняются программным обеспечением. Как правило, эти преобразования выполняют модули BIOS. Для использования символов кириллицы эти модули расширяются клавиатурными драйверами.

Если нажать на клавишу и не отпускать ее, клавиатура перейдет в режим автоповтора. В этом режиме в центральный компьютер автоматически через некоторый период времени, называемый периодом автоповтора, посылается код нажатой клавиши. Режим автоповтора облегчает ввод с клавиатуры большого количества одинаковых символов.

Следует отметить, что клавиатура содержит внутренний 16- байтовый буфер, через который она осуществляет обмен данными с компьютером.

Клавиатура AT, благодаря наличию встроенного в нее контроллера может программироваться. Вы можете программно установить скорость автоповтора и величину задержки до его начала, а также управлять светодиодами клавиатуры.

Контроллер клавиатуры генерирует запрос прерывания (IRQ 1) при каждом нажатии или отпускании клавиши. Прерывание IRQ 1 вызывает переход по вектору INT 09H и обрабатывается подпрограммой BIOS.



Порты для работы с клавиатурой

Для работы с клавиатурой используются порты с адресами 60h,64h и 61h.

Порт 60h при чтении содержит скан-код последней нажатой клавиши.

Порт 61h управляет не только клавиатурой, но и другими устройствами компьютера, например работой встроенного динамика. Порт доступен как для чтения, так и для записи. Для нас важен самый старший бит этого порта. Если в старший бит порта 61h записать значение 1, клавиатура будет заблокирована, если 0 разблокирована.

Так как порт 61h управляет не только клавиатурой, при изменении содержимого старшего бита необходимо сохранить состояния остальных битов этого порта. Для этого можно сначала выполнить чтение содержимого порта в регистр, изменить состояние старшего бита, затем выполнить запись нового значения в порт: in al, 61h or al, 80h out 61h, al

Компьютер позволяет управлять скоростными характеристиками клавиатуры, а также зажигать или гасить сведиоды на лицевой панели клавиатуры - Scroll Lock, Num Lock, lps Lock.

Для команд, требующих посылки двух байтов, таких как установление скорости автоповтора, между двумя командами out следует сделать небольшую задержку

Описание команд

0FFh Сброс клавиатуры и запуск внутреннего теста

0FEh Повторить последнюю передачу

0FDh-0F7H (NOP)

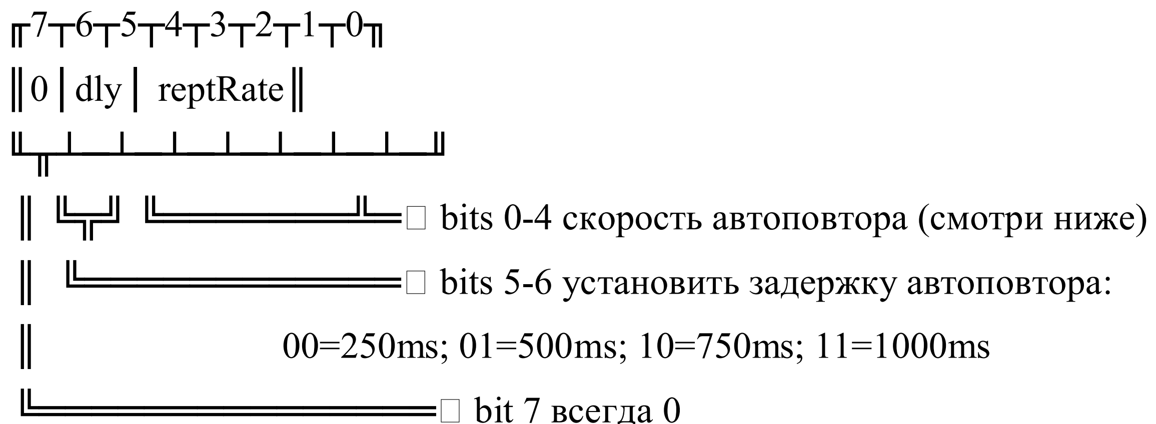
[0FDh - Вызов прерывания IRQ1]

0F6h Привести клавиатуру в исходное состояние и разрешить сканирование

0F5h Привести клавиатуру в исходное состояние и запретить сканирование

0F4h Сбросить буфер клавиатуры и начать сканирование

0F3h Задать скорость автоповтора и задержку до ее начала



This chart is a partial guide for the repeat rate (bits 0-4). You can interpolate for values not shown, but let's face it, you're only interested in the fastest rates.

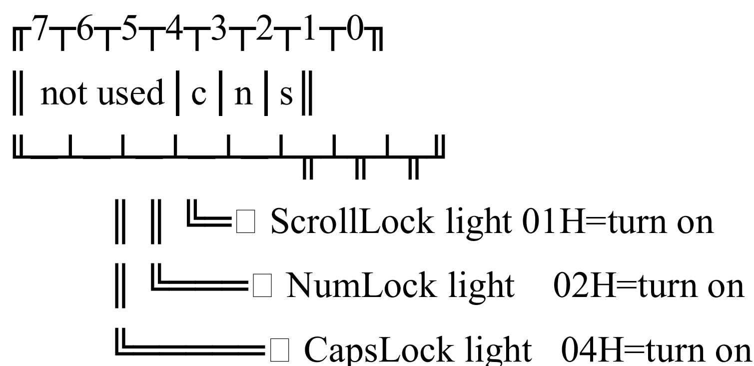
Код	Сим/сек	Код	Сим/сек
0	30.0	0Ah	10.0
1	26.7	0Dh	9.2
2	24.0	10h	7.5
4	20.0	14h	5.0
8	15.0	1Fh	2.0

Клавиатура первоначально установлена на задержкуr 1/2-секунды и скорость 10 символов/секунду.См.INT 16H 03H (установка скорости и задержки

0F2h-0EFh (NOP)

0EEh Эхо для целей диагностики. Просто возвращает 0EEh.

0EDh Управление светодиодами. Второй байт команды имеет формат:



Буфер клавиатуры

Указатели 0:041Ah - голова (последняя записанная)

0:041Ch - хвост (последняя несчитанная)

- Сам буфер находится в ячейках 0:041Eh...0:043Dh
- В буфер запись производит обработчик прерывания 09h.
- Считывание из буфера может осуществлять программа, используя функции DOS либо читая буфер напрямую.
- Как определить, пуст ли буфер? - сравнить указатели
- Как очистить буфер? - приравнять указатели

Лабораторная работа по теме N8

Мультизадачность

Мультипрограммирование предназначено для повышения пропускной способности вычислительной системы путем более равномерной и полной загрузки всего ее оборудования, в первую очередь процессора. При этом скорость работы самого процессора и номинальная производительность ЭВМ не зависят от мультипрограммирования.

Основные черты мультипрограммного режима:

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

В главе 5 рассмотрена организация мультизадачности в процессоре x86 в защищенном режиме работы. Процессор в этом случае аппаратно поддерживает мультизадачность. Микроконтроллеры не имеют средств аппаратной поддержки мультизадачности. В этом случае требуется на программном уровне обеспечить переключение задач при наступлении разных событий. В данной работе предлагается реализовать этот подход.

Цель работы: Получить опыт организации простой мультизадачности (без приоритетов).

ЗАДАНИЕ НА РАБОТУ

ЧТО НУЖНО ОСОЗНАТЬ

1. Виды контекста задачи (полный, короткий).
2. Методы сохранения и восстановления контекста.

ЧТО НУЖНО СДЕЛАТЬ

Организуйте три задачи. Выделите три области (статические) для сохранения контекста задач. Загрузите начальный контекст. Обеспечьте переключение между задачами по внешнему событию или таймеру, например по нажатию клавиши.

Пример двух программ, работающих в режиме мультизадачности (переключения контекста)

```
   masm
   DOSSEG
   .MODEL LARGE
   .STACK 100h
   .DATA
     CurrentContext dd 2
     Mes1 db '1',"$"
     Mes2 db '2',"$"
     Cont1 db 16d dup(0)
     Cont2 db 16d dup(0)
   .CODE
   jmp beg

   proc1 proc
     sti
t1:  mov ax,@Data
     mov ds,ax
     mov ah,09h
     mov dx,offset Mes1
     int 21h
     mov cx,2
     loop t1
     ret
   proc1 endp

   proc2 proc
     sti
t2:  mov ax,@Data
     mov ds,ax
     mov ah,09h
     mov dx,offset Mes2
     int 21h
```

```
  mov cx,2
  loop t2
  ret
proc2 endp

TaskManager proc
   mov es,ax
   mov ax,@Data
   mov ds,ax
   mov ax,es
   mov es,bp
   mov bp,sp
   mov si,[bp]
   mov di,[bp+2]
   mov bx,offset CurrentContext
   cmp word ptr ds:[bx],1
   jne c2  ;if now running Proc1
   ;SaveC1
   mov bx,offset Cont1
   mov ds:[bx+10d],ax
   mov ax,ds:[bp+4]
   mov ds:[bx+0d],ax
   mov ds:[bx+2d],di
   mov ds:[bx+4d],si
   ;mov ds:[bx+6d],sp
   mov ds:[bx+8d],es
   mov ds:[bx+12d],cx
   mov ds:[bx+14d],dx
   ;loadc2
   mov bx,offset CurrentContext
   mov word ptr ds:[bx],2
   mov bx,offset Cont2
   mov ax,ds:[bx+0d]
   mov word ptr [bp+4],ax
   mov di,ds:[bx+2d]
   mov si,ds:[bx+4d]
   ;mov sp,ds:[bx+6d]
   mov es,ds:[bx+8d]
   mov ax,ds:[bx+10d]
   mov cx,ds:[bx+12d]
   mov dx,ds:[bx+14d]

   jmp en ;if now running Proc2
```

```
  c2:
    ;SaveC2
    mov bx,offset Cont2
    mov ds:[bx+10d],ax
    mov ax,ds:[bp+4]
    mov ds:[bx+0d],ax
    mov ds:[bx+2d],di
    mov ds:[bx+4d],si
    ;mov ds:[bx+6d],sp
    mov ds:[bx+8d],es
    mov ds:[bx+12d],cx
    mov ds:[bx+14d],dx
    ;LoadC1
    mov bx,offset CurrentContext
    mov word ptr ds:[bx],1
    mov bx,offset Cont1
    mov ax,ds:[bx+0d]
    mov word ptr [bp+4],ax
    mov di,ds:[bx+2d]
    mov si,ds:[bx+4d]
    ;mov sp,ds:[bx+6d]
    mov es,ds:[bx+8d]
    mov ax,ds:[bx+10d]
    mov cx,ds:[bx+12d]
    mov dx,ds:[bx+14d]

  en:
    mov [bp+2],di
    mov [bp],si
    mov bp,es
    iret
  TaskManager endp

start proc
 beg:
  cli
  mov ax,@Data
  mov ds,ax
  mov cx,2
  mov ax,@CODE
  mov di,ax
  mov si,offset Proc2
  ;SaveC2
```

```
    xor ax,ax
    mov bx,offset Cont2
    mov ds:[bx+10d],ax
    pushf
    pop ax
    mov ds:[bx+0d],ax
    mov ds:[bx+2d],di
    mov ds:[bx+4d],si
    ;mov ds:[bx+6d],sp
    mov ds:[bx+8d],es
    mov ds:[bx+12d],cx
    mov ds:[bx+14d],dx

  mov ax,@CODE
  mov di,ax
  mov si,offset Proc1
  ;SaveC1
    xor ax,ax
    mov bx,offset Cont1
    mov ds:[bx+10d],ax
    pushf
    pop ax
    mov ds:[bx+0d],ax
    mov ds:[bx+2d],di
    mov ds:[bx+4d],si
    mov ds:[bx+6d],sp
    mov ds:[bx+8d],es
    mov ds:[bx+12d],cx
    mov ds:[bx+14d],dx

            ;set interrupt
  mov ax,0
  mov ds,ax
  mov ds:[32d+2],cs;press any key
  mov word ptr ds:[32d],offset TaskManager
  call Proc2
  ;mov ah,4ch
  ;int 21h
start endp
  END
```

Список контрольных вопросов

1. Организация фон-неймановской машины, ее отличия от современных ЭВМ.
2. Элементы процессора (схема). Операционные устройства. Типы операционных устройств с магистральной структурой.
3. Устройство управления. Микропрограммные автоматы с программируемой и жесткой логикой (схемы).
4. Структура процессора с тремя внутренними шинами. Микропрограммирование команд. Микрокоманды и микропрограмма
5. Организация шин. Синхронный и асинхронный протоколы. Методы повышения эффективности шин: расщепление транзакций, пакетный режим, конвейеризация, отказоустойчивость.
6. Форматы команд. Конвейер команд. Отличия CISC и RISC механизмов конвейера. Конвейерная адресация.
7. Кодирование данных. Целые числа: числа со знаком и без знака, повышенная точность, переполнение разрядной сетки.
8. Кодирование данных Изображения: классы изображений, кодирование цветных и полутоновых изображений, форматы графических файлов.
9. Кодирование данных Представление символьной информации: стандартные системы кодирования.
10. Форматы представления чисел с плавающей запятой в PC. Специальные численные значения. Особые случаи. Программная модель FPU .
11. Иерархия запоминающих устройств (схема). Методы доступа. Память адресная, стековая, ассоциативная (схемы). Локальность по обращению.
12. Организация динамической памяти (схема). Разбиение ОЗУ на банки.
13. Организация микросхем памяти DRAM с регенерацией (схема). Типы микросхем памяти FPM DRAM, EDO, BEDO, SDRAM, DDR, RDRAM и их временные диаграммы.
14. Регенерация памяти, типы и временные диаграммы. Обнаружение и исправление ошибок (схема).
15. Принципы организации КЭШ памяти: зачем она нужна, виды КЭШ памяти (схемы), как организован обмен между основной и КЭШ памятью, обновление КЭШ памяти: в чем проблема, какие алгоритмы. Примеры: КЭШ с прямым отображением (386 процессор), ассоциативный по множеству КЭШ (процессор Pentium).
16. Динамическое распределение памяти. Понятие и схема виртуальной памяти. Правила выборки страниц Алгоритмы замены страниц (сегментов). Логические и физические адреса. Трансляция адреса.
17. Общие принципы защиты памяти. Расслоение памяти. Защищенный режим виртуальной адресации в процессорах Pentium (обобщенная схема трансляции).
18. Схема вычисления физического адреса в защищенном режиме при обращении к сегментам, доступным для всех задач и только данной задаче. Схемы трансляции адреса с использованием GDT и LDT.
19. Виртуальная память. Аппаратная поддержка механизма виртуальной памяти на уровне сегментов и на уровне страниц. Формат селектора сегмента. Формат дескриптора сегмента. Типы дескрипторов.

20. Дескрипторные таблицы, их разновидности и содержание. Где они находятся? Какая информация содержится в дескрипторе? Прерывание в защищенном режиме.

21. Защита памяти на уровне сегментов. Механизмы проверок. Уровни привилегий (схемы проверок) и аппаратная поддержка защиты операционной системы при системных вызовах, обращениях к портам ввода-вывода и прерываниях. Привилегированные команды. Команды тестирования условий защиты

22. Механизм шлюзования. Формат дескриптора шлюза. Стеки.

23. Мультизадачный режим: в чем его смысл, в каких случаях он нужен? Переключение задач. Полный и короткий контекст задачи. Способы сохранения контекста. Какие механизмы аппаратной поддержки мультизадачности имеются в процессоре Pentium. (схема TSS …).

24. Мультизадачный режим. Состояния задач. Продвижение задач диспетчером (схема). Сценарии планирования. Процессор Pentium: Переход из реального механизма в мультизадачный. Как изолировано адресное пространство задачи.

25. Страничная организация памяти процессора Pentium. Схема трансляции адреса (схема). Виртуальная память на уровне страниц. Правила подкачки, замены и размещения страниц.

26. TLB(схема). Назначение, структура и тестирование.

27. Средства отладки. Системные регистры Pentium.

28. Загрузка и тестирование ЭВМ.

29. Способы подключения УВВ к процессору. Модель внешнего устройства для программиста. Прозрачная (прямая) схема трансляции адресов

30. Схема подключения внешнего устройства к ISA шине. Временная диаграмма ISA

31. Структура внешнего устройства, поддерживающего синхронизацию с ЭВМ на шине ISA по прерыванию и ПДП. Элементы: дешифратор команд, шинный формирователь и флаги готовности.

32. Обмен с ВнУ по прерыванию. Радиальное и векторное прерывание. Приоритет прерывания. Вектор состояния и вектор прерывания. Механизмы доступа к обработчику прерывания. Циклы прерываний в процессоре. Диаграмма состояний (Схема).

33. Структура аппаратных прерываний в PC. Контроллер прерываний и схема подключения к процессору. Синхронизация работы с процессором. Модель для программиста (регистры, их функции, порядок программирования). Какие свойства системы прерываний вы можете изменять при его программировании.

34. Прямой доступ к памяти - третий способ обмена между устройствами, присоединенными к магистрали. Режимы ПДП. Аппаратная реализация в PC. Взаимно независимые категории свойств, которые программист может задать при инициализации процесса ПДП и типовой порядок программирования.

35. Функции контроллера памяти. Временные диаграммы обращения к ОЗУ. Регенерация, виды регенерации.

36. Шины. Элементы. Электрические аспекты.

37. Арбитраж шин. Алгоритмы динамического изменения приоритетов

38. Методы повышения эффективности шин Пакетный режим передачи. Конвейеризация транзакций. Арбитраж с перекрытием. Конвейерное обращение к памяти.

Расщепление транзакций. Увеличение пропускной способности шины, мультиплексирование. Арбитраж с удержание шины.

39. Временная диаграмма PCI шины (минимальный набор сигналов). Элементы протокола обмена в PCI. PCI Express.

40. Арбитраж PCI шины. Временная диаграмма шины с арбитражем. Распределение ресурсов. Компоненты Plug and Play

41. Временная диаграмма обращения к порту ввода-вывода.

42. Временная диаграмма режима прямого доступа в память.

43. Временная диаграмма пакетного режима.

44. Временные диаграммы обращения к ОЗУ.

Рекомендуемая литература

1. Гук М. Аппаратные средства IBM PC. Энциклопедия. 3-е изд. СПб., Питер, 2008.- 1072 с.
2. Дао Л. Программирование микропроцессора 8088.-М.:Мир,1988
3. Зубков С. В. Assembler для DOS, Windows и UNIX для программистов, Изд. "Питер" 2003.- 608с.
4. Кенцл, Тим. Форматы файлов Internet. СПб., Питер, 1997.- 320с.
5. Климов А.С. Форматы графических файлов. - К, НИПФ "ДиаСофт Лтд", 1995.
6. Мелехин В. Ф., Павловский Е. Г. Вычислительные машины, системы и сети - М. Академия, 2006
7. Пильщиков В.Н. Программирование на языке ассемблера IBM PC. – М.: «ДИАЛОГ-МИФИ»,2000.- 288с.
8. Соломон Д., Руссинович М. Внутреннее устройство Microsoft Windows 2000.- Спб.:Питер, 2001, 752с.
9. Столлингс У. Структурная организация и архитектура компьютерных систем. - М., Вильямс, 2002.
10. Таненбаум Э. Архитектура компьютера. - СПб, Питер, 2002.- 704с.
11. Цилькер Б.Я., Орлов С.А. Организация ЭВМ и систем // Учебник для вузов. - СПб.: Питер, 2004.- 668с.
12. Хамахер К., Вранешич З., Заки С. Организация ЭВМ.. - СПб, Питер, 2003.
13. Юров В. И. Assembler: Учебник для вузов. СПб., Питер, 2003.- 637с.
14. Intel® 64 and IA-32 Architectures. Software Developer’s Manual. Volume 2A: Instruction Set Reference

Автор выражает благодарность Новицкому Александру Петровичу за предоставленные материалы.