Arhitectura Calculatoarelor
Suport de curs
(limba rusă)

Архитектура ЭВМ

Введение

Это понятие довольно трудно определить однозначно, потому что при желании в него можно включить все, что связано с ЭВМ вообще и какой-то конкретной моделью компьютера в частности. Попытаемся все же формализовать этот широко распространенный термин.

Архитектура ЭВМ (системы) — это совокупность свойств компьютера (системы), существенных для программиста и пользователя.

Система (от греч. *systema* — целое, составленное из частей соединение) — это совокупность элементов, взаимодействующих друг с другом, образующих определенную целостность, единство.

Элемент (компонент) системы — часть системы, имеющая определенное функциональное назначение. Сложные элементы систем, в свою очередь состоящие из более простых взаимосвязанных элементов, часто называют подсистемами.

Организация системы — внутренняя упорядоченность, согласованность взаимодействия элементов системы, проявляющаяся, в частности, в ограничении разнообразия состояний элементов в рамках системы.

Структура системы — состав, порядок и принципы взаимодействия элементов системы, определяющие основные свойства системы. Если отдельные элементы системы разнесены по разным уровням и внутренние связи между элементами организованы только от вышестоящих к нижестоящим уровням и наоборот, то говорят об *иерархической структуре* системы.

Архитектура ЭВМ — это абстрактное представление ЭВМ, которая в упрощенном виде состоит из двух основных компонентов: программного обеспечения (software) и электронных схем и устройств (hardware).

Основной акцент данного предмета, сделан на архитектуре самого массового типа ЭВМ персональных компьютеров; рассмотрены современное состояние и характеристики всех основных узлов компьютера.

1.Функциональная и структурная организация некоторого компьютера

Цифровой компьютер — это машина, которая может решать задачи, выполняя данные ей команды. Последовательность команд, описывающих решение определенной задачи, называется программой. Электронные схемы каждого компьютера могут распознавать и выполнять ограниченный набор простых команд, которые состоят из последовательности бит (1 и 0). Все программы перед выполнением должны быть превращены в последовательность таких команд, которые обычно не сложнее, чем, например, (Таненбаум):

- сложить 2 числа;
- проверить, не является ли число нулем;
- скопировать блок данных из одной части памяти компьютера в другую.

Эти примитивные команды в совокупности составляют язык, на котором люди могут общаться с компьютером. Такой язык называется машинным.

Например, микропроцессоры Intel 80x86 воспринимают следующий двоичный код:

0000 0100 0000 0110 (0406h – mov al,06)

как загрузка непосредственного значения 6, во внутренний 8-разрядный регистр (называемый AL).

Однако двоичное представление неудобно при записи чисел - числа получаются очень длинными. Поэтому при записи чисел в программе чаще используется **шестнадцатеричная** (hexadecimal) система счисления. В ней вводятся 6 новых "цифр" при помощи латинских букв: A=10, B=11, C=12, D=13, E=14, F=15 (малые или большие латинские буквы — безразлично). Будем отмечать шестнадцатеричные числа буквой **h** после числа. Кроме того, шестнадцатеричное число всегда должно начинаться с арабской цифры (чтобы отличить его от идентификатора), иначе перед числом ставят незначащий ноль, например: 0E7A.

Для того чтобы объяснить, как аппаратное и программное обеспечение компьютера взаимодействуют между собой, лучше всего воспользоваться так называемой концепцией виртуальной машины. Приведенное ниже описание взято из книги Э. Таненбаума.

Обычно при проектировании любого компьютера в нем предусматривают возможность непосредственного запуска программ, состоящих из так называемых машинных кодов. Каждая

команда машинного кода имеет достаточно простую структуру. Благодаря этому ее можно легко декодировать и выполнить с помощью относительно небольшого количества электронных компонентов, составляющих арифметико-логический блок (АЛУ) центрального процессора. Для простоты назовем машинный код языком L0.

С точки зрения программиста, писать программы на языке L0 очень утомительно и крайне неэффективно, поскольку этот язык излишне детализирован и целиком и полностью состоит из обычных цифр. Поэтому, чтобы облегчить программистам задачу, должен быть создан новый язык программирования; назовем его L1. Данную цель можно достичь двумя способами:

- Интерпретация. Во время выполнения программы на языке L1, каждая из ее команд должна оперативно декодироваться и выполняться программой, написанной на языке L0. Таким образом, при запуске, программа на языке L1 начинает выполняться сразу, но каждая ее команда перед выполнением должна быть декодирована.
- Трансляция. Перед выполнением программа, написанная на языке L1, должна быть преобразована в программу на языке L0 другой специально созданной для этой цели программой, написанной на языке L0. После этого полученная на языке L0 программа может быть непосредственно выполнена центральным процессором компьютера.

Виртуальные машины. Чтобы не заниматься анализом программ, написанных для конкретного типа процессоров, необходимо создать модель гипотетического компьютера, или виртуальную машину, для каждого из уровней. Следовательно, виртуальная машина VM1 (назовем ее так) может выполнять команды, написанные на языке L1.

Аналогично, виртуальная машина VM0 может выполнять команды, написанные на языке L0, как показано на рис. 1.1.

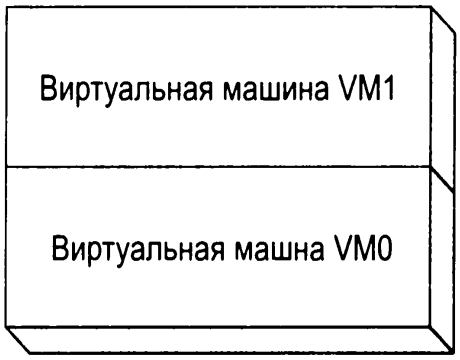


Рисунок 1.1 - Принцип виртуальных машин Таненбаума

Каждая виртуальная машина может быть реализована либо программно, либо аппаратно. Однако независимо от этого программист может писать программы для виртуальной машины VM1 так же, как если бы он это делал для реального компьютера. Кроме того, если реализовать виртуальную машину VM1 на аппаратном уровне, то написанные для VM1 программы можно будет непосредственно запускать на выполнение, минуя промежуточные этапы трансляции или интерпретации. Кроме того, программы, написанные для виртуальной машины VM1, могут интерпретироваться или транслироваться, а затем выполняться виртуальной машиной VM0.

Структура машины VM1 не может коренным образом отличаться от структуры машины VM0, поскольку иначе трансляция или интерпретация программ будет достаточно продолжительной, а если при этом язык программирования, поддерживаемый машиной VM1, остается неудобным с точки зрения прикладного программиста? В таких случаях нужно создать еще одну виртуальную машину, скажем VM2, с которой было бы удобнее работать. Описанный выше процесс продолжается до тех пор, пока не будет создана виртуальная машина VMn, полностью удовлетворяющая запросам пользователей и поддерживающая развитой и простой в использовании язык программирования.

Именно концепция виртуальной машины и положена в основу языка Java. Программы, написанные на Java, транслируются компилятором Java в промежуточный код, или байт-код Java, который, по сути, является языком низкого уровня. Написанные на нем программы могут быть быстро преобразованы в машинный код непосредственно перед запуском программы или во время ее выполнения, так называемой виртуальной машиной Java (Java Virtual Machine, или JVM). А

поскольку версии JVM разработаны практически для всех типов операционных систем и компьютерных платформ, язык Java можно считать системно-независимым или переносимым.

Виды виртуальных машин. Теперь применим описанный выше принцип виртуальных машин к реальным компьютерам и языкам программирования, как показано на рис. 1.2.

Рисунок 1.2 – Шестиуровневая модель виртуальных машин

Из рисунка видно, что машина VM0 находится на уровне 0, а уровню 1 соответствует машина VM1. Предположим, что с помощью цифровых электронных схем реализована виртуальная машина нулевого уровня, а машина уровня 1 выполнена в виде интерпретатора, логика работы которого "зашита" в специализированном процессоре, управляемом микропрограммой. Следующий, второй, уровень составляет система команд процессора. Это самый первый уровень, на котором пользователи уже могут писать простейшие программы, состоящие из обычных двоичных чисел.

Микропрограммы (уровень 1). Производители электронных микросхем обычно не предусматривают для среднестатистического пользователя возможности программирования своих устройств с помощью микрокоманд. Более того, описание конкретных микрокоманд часто является секретом фирмы, поскольку с их помощью реализуется система команд процессора. Например, для выполнения простейшей операции процессором, такой как выборка операнда из памяти и увеличения его значения на 1, требуется порядка трех, четырех микрокоманд.

Система команд процессора (уровень 2). Производители электронных микросхем, выпускающие микропроцессорные комплекты, предусматривают для их программирования специальный набор команд, или систему команд, выполняющих основные операции, такие как пересылка байтов, сложение или умножение. Этот набор команд называется обычным машинным языком или просто машинным кодом. Для выполнения одной команды машинного кода, или машинной команды, как правило, требуется выполнить несколько микрокоманд.

Операционная система (уровень 3). По мере повышения сложности внутренней структуры компьютеров и увеличения их вычислительной мощности, были созданы дополнительные уровни виртуальных машин, что позволило более продуктивно использовать время работы программиста. На уровне 3 машина уже может вести работу с пользователем в диалоговом режиме и выполнять его команды, такие как загрузка в память программ и их выполнение, отображение содержимого каталога и т.п. Программа, или виртуальная машина, с помощью которой реализованы эти возможности, называется операционной системой компьютера. Программы, составляющие операционную систему, заранее оттранслированы в машинный код, который выполняет виртуальная машина уровня 2. Независимо от того, на каком языке программирования (С или ассемблере) написан исходный код, после компиляции операционная система автоматически превращается в программу уровня 2, которая выполняет интерпретацию команд уровня 3.

Язык ассемблера (уровень 4). Выше уровня операционной системы находятся уровни трансляторов с языков программирования, которые делают возможным разработку

крупномасштабных программных проектов. В языке ассемблера, который в иерархической структуре относится к уровню 4, используются короткие мнемоники команд, такие как ADD, SUB, MOV. Благодаря им облегчается процесс трансляции программы в машинный код, соответствующий уровню 2 или системе команд процессора. Ряд команд языка ассемблера, таких как вызов программного прерывания, обрабатываются и выполняются непосредственно операционной системой, находящейся на уровне 3. Программы на языке ассемблера перед запуском обычно транслируются (или ассемблируются) в машинный код полностью.

Языки высокого уровня (уровень 5). Этому уровню соответствуют языки программирования, такие как C++, C, Visual Basic, Java. В программах, написанных на этих языках, обычно используются сложные операторы, которые транслируются сразу в несколько команд языка ассемблера, соответствующих уровню 4. Например, практически во всех отладчиках кода C++ предусмотрено специальное окно, в котором отображаются операторы исходного кода и команды на языке ассемблера, которые получились в результате трансляции. В отладчиках программ на Java также предусмотрено окно с аналогичной информацией, только вместо операторов ассемблера в нем отображается байт-код Java. Таким образом, программы, которые относятся к уровню 5, обычно транслируются компиляторами, соответствующими программам уровня 4. Чаще всего компиляторы содержат встроенный ассемблер, с помощью которого исходный код уровня 4 транслируется непосредственно в машинный код.

В архитектуре процессоров фирмы InteI семейства IA-32 поддерживается концепция множества виртуальных машин. В ней предусмотрен специальный виртуальный режим, в котором полностью эмулируется работа процессоров Intel 8086/8088, использовавшихся в первых моделях персональных компьютеров IBM PC. Более того, при работе процессора в этом режиме можно запускать несколько экземпляров виртуальных машин.

2 Регистры микропроцессоров

2.1 Общие понятия

В ассемблере, компьютер виден как: физические адреса памяти, регистров, прерывания и т.д. Основной единицей информации, хранящейся в компьютере это бит. Бит — это разряд двоичного числа (сокращение от **bi**nary digi**t** - bit), может иметь значения 0 или 1. На электронном уровне бит материализован в виде триггера с двумя устойчивыми состояниями. Приняв одно из состояний 0 или 1, можно считать, что триггер хранит один разряд числа, записанного в двоичном коде 0 или 1. Группа из 8 бит (триггеров) составляют один **байт (byte).** Байт можно рассматривать как 8-разрядный регистр внутри микропроцессора или как ячейку памяти. Информация, которая храниться в регистре в двоичном коде, может иметь значения 00000000 (все биты равны 0), до 11111111 (все биты равны 1). Число комбинаций, которые могут быть сохранены, 256 (0 – 255, 2 в степени 8). В общем, n-разрядный регистр может хранить 2^n различных комбинаций. Эти комбинации называются байтами (при n = 8) или словами (при n = 16, 32 и т.д.).

Байты в памяти нумеруются слева направо, начиная с нулевого байта. Номер байта называется его **адресом** в памяти. Внутри байта биты нумеруются справа налево от 0 (правый младший бит) до 7 (старший левый бит). Нумерация битов внутри байта условна и никакого отношения к адресу не имеет. Поскольку программирование осуществляется при помощи адресов. Программист может работать только с байтами, а не с битами, так как у последних адресов нет. Этот важный факт выражается следующей фразой: **байт есть наименьшая адресуемая часть памяти.**

Два любых соседних байта могут быть объединены в **слово (word).** Адресом слова считается адрес его левого байта.

2.2 Регистры микропроцессоров x86-64

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

32-битные микропроцессоры могут работать в различных режимах, определяющих возможности адресации памяти и защиты: в реальном (16-разрядном) режиме процессора 8086, в режиме виртуального процессора 8086 (V86), в защищенном 32-разрядном режиме. Режим работы процессора задается операционной системой с учетом режима работы приложений (задач, task). У процессоров с 64-битным расширением появляются новые режимы, среди которых есть и режимы, обеспечивающие совместимость с 32-разрядными операционными системами и приложениями. Новые режимы используются только в 64-битных ОС, а полностью их преимущества доступны только 64-битным приложениям:

- 64-битный режим (64-bit mode) — это режим полной поддержки 64-битной виртуальной адресации и 64-битных расширений регистров. В этом режиме используется только *плоская модель памяти* (общий сегмент для кода, данных и стека). По умолчанию разрядность адреса составляет 64 бита, а операндов (для большинства инструкций) — 32 бита, однако префиксом (REX) можно задать 64-битные операнды;
- режим совместимости (compatibility mode) позволяет 64-битным ОС работать с 32- и 16-битными приложениями. Для приложений микропроцессор выглядит как обычный 32-битный со всеми атрибутами защищенного режима, сегментацией и страничной трансляцией. 64-битные свойства используются только операционной системой, что отражается в процедурах трансляции адресов, обработки исключений и прерываний. Режим включается операционной системой для сегмента кода конкретной задачи.

32-битные ОС используют микропроцессоры x86-64 как обычный 32-битный микропроцессор.

Программная модель 64-битного микропроцессора представляет из себя блоки (наборы) регистров доступные программистам.

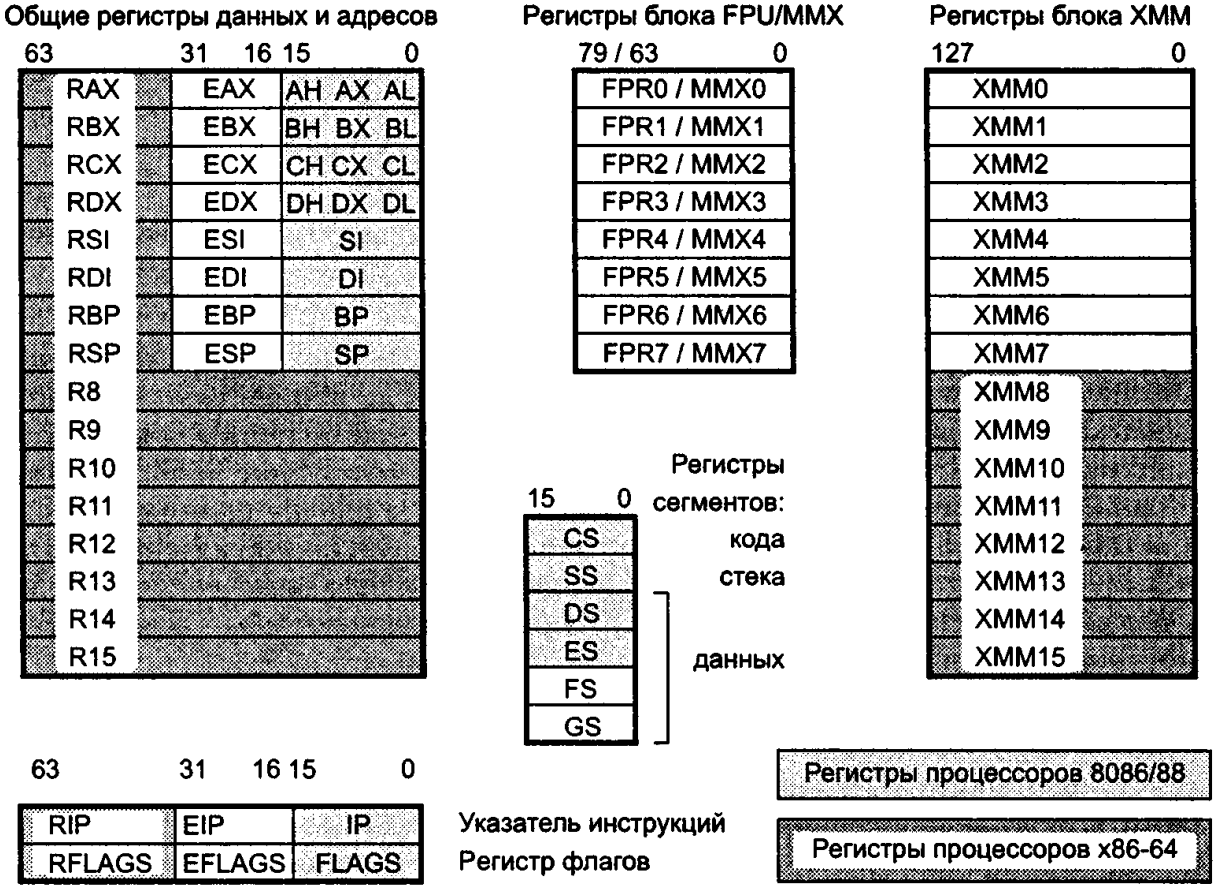


Рисунок 2.1 - Прикладные регистры микропроцессоров x86-64

Блоки регистров FPU/MMX и XMM будут рассмотрены в 5 главе.

2.2 Регистры общего назначения

На следующем рисунке 2.2 приведен блок регистров общего назначения и регистры флагов и *указатель команд.*

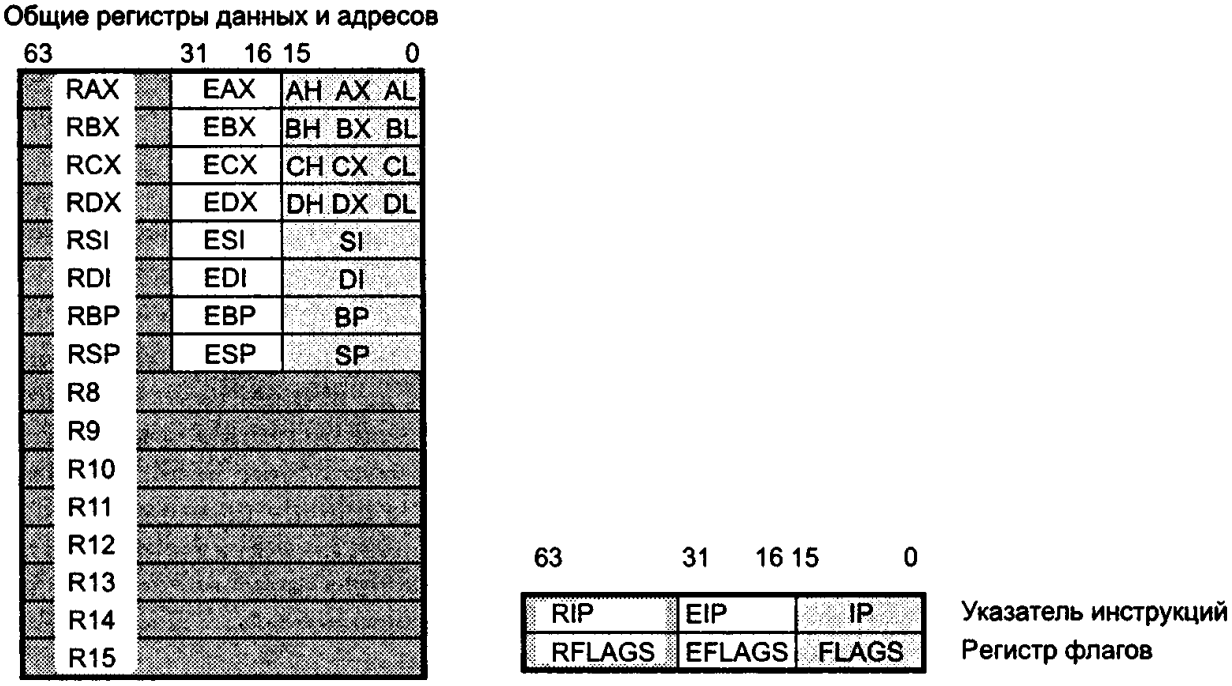


Рисунок 2.2 - Блок регистров общего назначения, регистры флагов и указатель команд

Названия 64-битных регистров начинаются с буквы R. К любому из 16 общих регистров можно обращаться как к 64-, 32-, 16- или 8-битному регистру (всегда используются младшие биты). Старшие части этих регистров как самостоятельные объекты недоступны.
Функциональное назначение регистров рассмотрим далее, рассматривая режимы работы микропроцессора.

2.3 Реальный (16-разрядный) режим работы микропроцессора

2.3.1 Регистры реального режима

Режим реальной адресации (real address mode), или просто реальный режим (real mode), полностью совместим с микропроцессором 8086. В этом режиме возможна адресация до 1 Мбайт физической памяти.

Регистры реального режима (микропроцессора i8086), представлена на рис. 2.3. Все регистры программно доступны. Он содержит двенадцать 16-разрядных программно-адресуемых регистров, которые принято объединять в три группы: регистры данных, регистры-указатели и сегментные регистры. Кроме того, в состав процессора входят счетчик команд и регистр флагов (рис. 2.4). Регистры данных и регистры-указатели часто называют регистрами общего назначения.

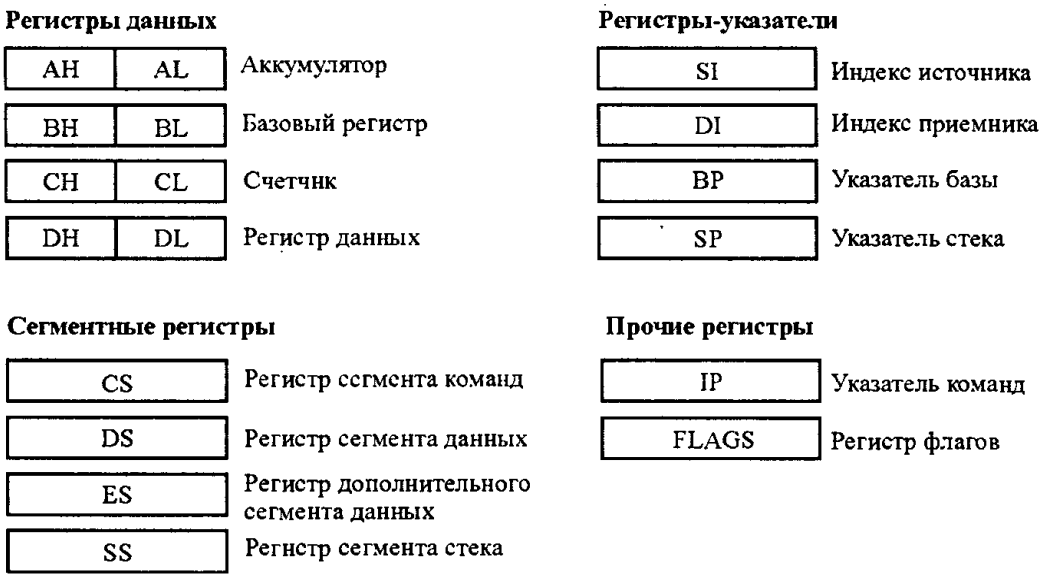


Рисунок 2.3 - Регистры микропроцессора 8086

В группу регистров данных включаются регистры AX, BX, CX и DX. Программист может использовать их по своему усмотрению для временного хранения любых объектов (данных или адресов) и выполнения над ними требуемых операций. При этом регистры допускают независимое обращение к старшим (AH, BH, CH и DH) и младшим (AL, BL, CL и DL) половинам.

Так, команда - **mov BL,AH**, пересылает старший байт регистра AX в младший байт регистра BX, не затрагивая при этом вторых байтов этих регистров. Еще раз отметим, что сначала указывается операнд-приемник, а после запятой - операнд-источник, т. е. команда, выполняется как бы справа налево. В качестве средства временного хранения данных все регистры общего назначения (да и все остальные, кроме сегментных и указателя стека) вполне эквивалентны, однако многие команды требуют для своего выполнения использования вполне определенных регистров.

Например, команда умножения **mul** требует, чтобы один из сомножителей был в регистре AX (или AL), а команда организации цикла **loop** выполняет циклический переход **CX** раз. Индексные регистры SI и DI так же, как и регистры данных, могут использоваться произвольным образом. Однако их основное назначение - хранить индексы (смещения) относительно некоторой базы (т. е. начала массива) при выборке операндов из памяти. Адрес базы при этом обычно находится в одном из базовых регистров (BX или BP). Примеры такого рода будут приведены ниже.

Регистр BP служит указателем базы при работе с данными в стековых структурах, о чем будет речь впереди, но может использоваться и произвольным образом в большинстве арифметических и логических операций или просто для временного хранения каких-либо данных.

Последний из регистров-указателей, указатель стека SP, стоит особняком от других в том отношении, что используется исключительно как указатель вершины стека.

Регистры SI, DI, BP и SP, в отличие от регистров данных, не допускают побайтовую адресацию.

Четыре сегментных регистра CS, DS, ES и SS хранят начальные адреса сегментов программы и, тем самым, обеспечивают возможность обращения к этим сегментам.

Регистр CS обеспечивает адресацию к сегменту, в котором находятся программные коды, регистры DS и ES - к сегментам с данными (таким образом, в любой точке программа может иметь доступ к двум сегментам данных, основному и дополнительному), а регистр SS - к сегменту стека. Сегментные регистры, естественно, не могут выступать в качестве регистров общего назначения.

Указатель команд IP "следит" за ходом выполнения программы, указывая в каждый момент относительный адрес команды, следующей за исполняемой. Регистр IP программно недоступен (IP - это просто его сокращенное название, а не мнемоническое обозначение, используемое в языке программирования); наращивание адреса в нем выполняет микропроцессор, учитывая при этом длину текущей команды.

Регистр флагов, эквивалентный регистру состояния микропроцессора других вычислительных систем, содержит информацию о текущем состоянии процессора (рис. 2.4). Он включает 6 флагов состояния и 3 бита управления состоянием процессора, которые тоже обычно называются флагами.

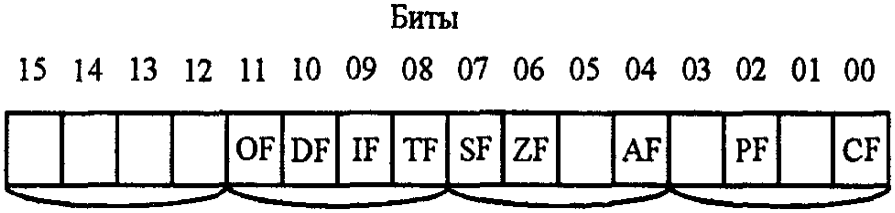


Рисунок 2.4 - Регистр флагов. Дужками выделены четверки битов

Флаг переноса CF (Carry Flag) индицирует перенос или заем при выполнении арифметических операций, а также (что для прикладного программиста гораздо важнее!) служит индикатором ошибки при обращении к системным функциям.

Флаг паритета PF (Parity Flag) устанавливается в 1, если младшие 8 бит результата операции содержат четное число двоичных единиц.

Флаг вспомогательного переноса AF (Auxiliary Flag) используется в операциях над упакованными двоично-десятичными числами. Он индицирует перенос в старшую тетраду (четверку битов) или заем из старшей тетрады.

Флаг нуля ZF (Zero Flag) устанавливается в 1, если результат операции равен нулю.

Флаг знака SF (Sign Flag) показывает знак результата операции, устанавливаясь в 1 при отрицательном результате.

Флаг переполнения OF (Overflow Flag) фиксирует переполнение, т. е. выход результата операции за пределы допустимого для данного процессора диапазона значений.

Флаги состояния автоматически устанавливаются процессором после выполнения каждой команды. Так, если в регистре AX содержится число 1, то после выполнения команды декремента (уменьшения на единицу) -- dec AX, содержимое AX станет равно нулю и процессор сразу отметит этот факт, установив в регистре флагов бит ZF (флаг нуля). Если попытаться сложить два больших числа, например, 58 000 и 61 000, то установится флаг переноса CF, так как число 119000, получающееся в результате сложения, должно занять больше двоичных разрядов, чем помещается в 16-битных регистрах или ячейках памяти, и возникает "перенос" старшего бита этого числа в бит CF регистра флагов.

Индицирующие флаги процессора дают возможность проанализировать, если это нужно, результат последней операции и осуществить "разветвление" программы: например, в случае нулевого результата перейти на выполнение одного фрагмента программы, а в случае ненулевого - на выполнение другого. Такие разветвления осуществляются с помощью команд условных переходов, которые в процессе своего выполнения анализируют состояние регистра флагов. Так, команда - jz zero, осуществляет переход на метку zero, если результат выполнения предыдущей команды окажется равен нулю (т. е. флаг ZF установлен), а команда - jnc okey, выполнит переход на метку okey, если предыдущая команда сбросила флаг переноса CF (или оставила его в сброшенном состоянии).

Управляющий флаг трассировки TF (Trace Flag) используется в отладчиках для осуществления пошагового выполнения программы. Если TF=l, то после выполнения каждой команды процессор реализует процедуру прерывания 1 (через вектор прерывания с номером 1).

Управляющий флаг разрешения прерываний IF (Interrupt Flag) разрешает (если равен единице) или запрещает (если равен нулю) процессору реагировать на прерывания от внешних устройств.

Управляющий флаг направления DF (Direction Flag) используется особой группой команд, предназначенных для обработки строк. Если DF=0, строка обрабатывается в прямом направлении, от меньших адресов к большим; если DF=1, обработка строки идет в обратном направлении.

Таким образом, в отличие от битов состояния, управляющие флаги устанавливает или сбрасывает программист, если он хочет изменить настройку системы (например, запретить на какое-то время аппаратные прерывания или изменить направление обработки строк).

2.3.2 Формирование физического адреса

Микропроцессор имеет 20-разрядную шину, что позволяет адресовать 2^{20} = 1048576 ячеек памяти. Разрядность ячейки принята равной 8 бит, т. е. МП доступен 1 Мбайт памяти. Схемные решения, которые будут рассмотрены ниже, обеспечивают обмен с памятью байтами или словами (16-разрядными кодами) с автоматическим выбором требуемого формата. 20-разрядный физический адрес может изменяться от 00000**h** до FFFFF**h** (индекс **h** означает шестнадцатеричную систему счисления).

Логически все адресное пространство памяти разбито на сегменты, каждый длиной до 2^{16} = 64 Кбайт. Заметим, что для адресации конкретной ячейки в сегменте, необходим только 16-разрядный адрес (от 0000h до FFFFh). Расположение сегмента в памяти задается 20-разрядным адресом сегмента (базовым адресом), младшая шестнадцатеричная цифра которого должна быть равна 0h. Таким образом, базовый адрес сегмента имеет вид XXXX0h, где X – любая шестнадцатеричная цифра, а общее количество сегментов не может быть больше 2^{16}.

В каждый данный момент времени МП доступны только четыре сегмента памяти, базовые адреса которых (вернее 16 старших значащих разрядов которых XXXXh) хранятся в сегментных регистрах. Эти сегменты называются текущими: текущий сегмент команд (адрес сегмента в

регистре CS), текущий сегмент данных (адрес в регистре DS), текущий сегмент стека (адрес в регистре SS) и текущий экстра сегмент (адрес в регистре ES). Сегменты могут совпадать, если совпадает содержимое сегментных регистров, могут перекрываться, если содержимое регистров отличается менее чем на 1000h, или быть полностью различными, если коды в регистрах сегментов отличаются более чем на 1000h.

Формирование 20-разрядного физического адреса по 16-разрядным адресам сегмента и ячейки в сегменте иллюстрируется на рис. 2.5. Пусть в сегментном регистре расположен код 1A03h. Адрес команды в сегменте команд, адрес кода данных в сегменте данных, адрес вершины стека в сегменте стека генерируются в МП по определенным правилам. Этот 16-разрядный адрес фактически задает смещение от начального сегмента до искомого (в примере этот адрес равен FFFBh).

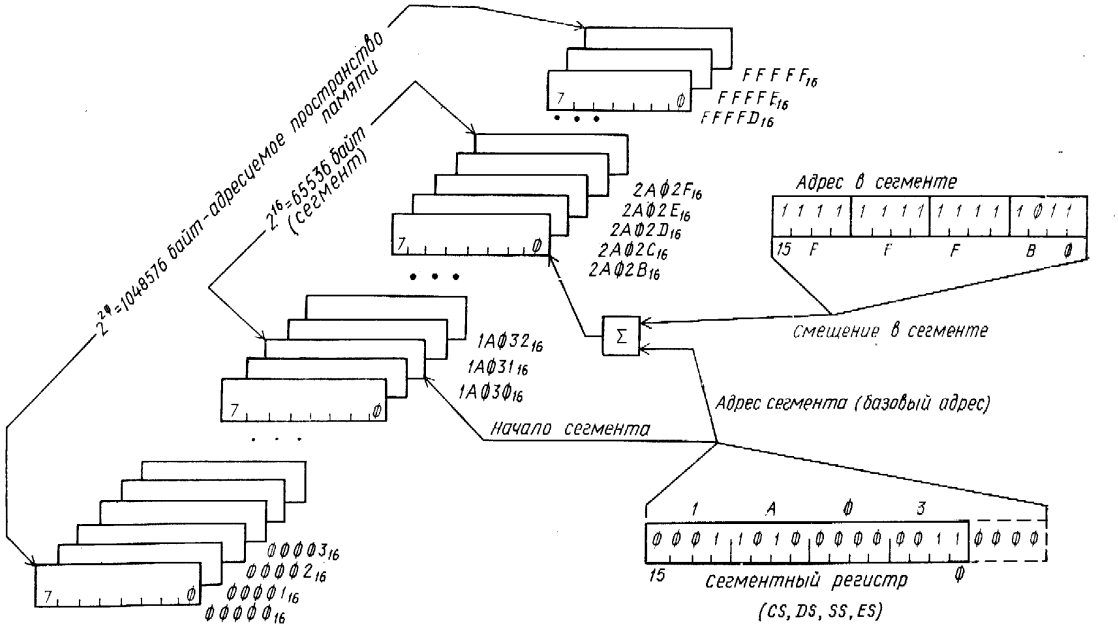


Рис. 2.5 - Формирование физического адреса в МП 8086

Полный 20-разрядный физический адрес образуется в МП как сумма содержимого сегментного регистра, сдвинутого влево на 4 двоичных разряда и дополненного справа кодом 0000h и адреса в сегменте:

1A030h +FFFBh = 2A20Bh .

Таким образом, в примере выбирается байт (или слово) с физическим адресом 2A20Bh.

2.3.3 Определение сегментов. Структура программы

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

Сегменты определяются при помощи директив.

Директивы сегментации. Директива MODEL. Операнды директивы MODEL используют для задания модели памяти, которая определяет набор сегментов программы, размеры сегментов данных и кода, способ связывания сегментов и сегментных регистров. Ниже приведены некоторые значения параметра модель памяти директивы MODEL.

- TINY — код, данные и стек размещаются в одном и том же сегменте размером до 64 Кб. Эта модель памяти чаще всего используется при написании на ассемблере небольших программ;
- SMALL — код размещается в одном сегменте, а данные и стек — в другом (для их описания могут применяться разные сегменты, но объединенные в одну группу). Эту модель памяти также удобно использовать для создания программ на ассемблере;

- COMPACT — код размещается в одном сегменте, а для хранения данных могут использоваться несколько сегментов, так что для обращения к данным требуется указывать сегмент и смещение (данные дальнего типа);
- MEDIUM — код размещается в нескольких сегментах, а все данные — в одном, поэтому для доступа к данным используется только смещение, а вызовы подпрограмм применяют команды дальнего вызова процедуры;
- LARGE и HUGE — и код, и данные могут занимать несколько сегментов;
- FLAT — то же, что и TINY, но используются 32-битные сегменты, так что максимальный размер сегмента, содержащего и данные, и код, и стек, — 4 GB.

Также используются следующие директивы:
- **.model small** (определяет тип памяти)
- **.code** (определяет начало сегмента кода)
- **.stack n** (определяет начало сегмента стека)
- **.data** (определяет начало сегмента данных)
- **end metka** (логическое завершение программы)

Упрощенная структура программы:

```
;Определение макросов
.MODEL small
.STACK 512
.DATA

;Определение данных

.CODE

;Определение процедур

start: mov ax,@data
mov ds,ax

;код программы

end start
```

Директива. stack выделяет область памяти размером **n** байт **(.stack n)** , область стековой памяти.

Директива. code определяет начало или продолжение сегмента **кода.** Загрузку сегментного регистра **cs** выполняет DOS. А регистр **ds** загружается программистом.

Идентификатор @**data** указывает физический адрес сегмента данных.

start: - метка, указывает на конкретный адрес (смещение) в коде программы.

2.4 Защищенный (32-разрядный) режим работы микропроцессора

В программах на языке ассемблера регистры используются очень интенсивно. Большинство из них имеет определенное функциональное назначение.

На схеме, на рисунке 2.6, представлены регистры общего назначения, сегментные регистры, состояния и управления 32 разрядных микропроцессоров Intel.

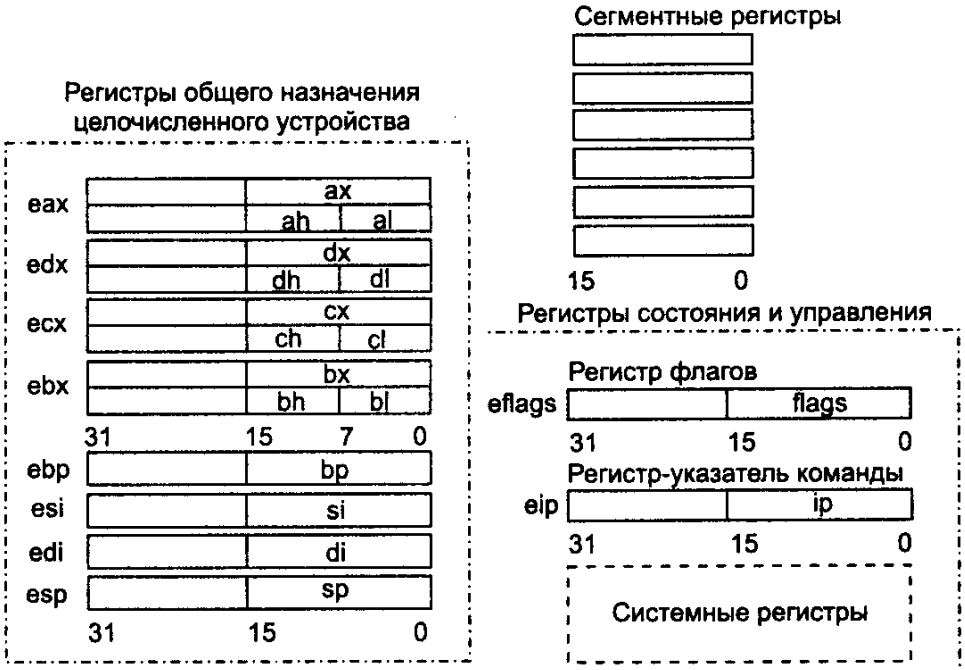


Рисунок 2.6 - Регистры

На рисунке представлены регистры:

- Регистры общего назначения - eax/ax/ah/al, ebx/bx/bh/bl, edx/dx/dh/dl, ecx/cx/ch/cl, ebp/bp, esi/si, edi/di, esp/sp. Регистры этой группы используются для хранения данных и адресов;

- сегментные регистры cs, ds, ss, es, fs, gs. Регистры этой группы используются для хранения адресов сегментов в памяти;
- регистр флагов eflags/flags;
- регистр указатель команды eip/ip.

Регистры приведены с наклонной разделительной чертой, это части одного большого 32-разрядного регистра. Их можно использовать в программе как отдельные объекты. Зачем так сделано? Для обеспечения работоспособности программ, написанных для младших 16-разрядных моделей микропроцессоров фирмы Intel, начиная с i8086. 32-разрядные регистры имеют приставку «e» (Extended).

Сегментные регистры используются в режиме совместимости – в реальном режиме, для хранения адресов сегментов, а в 32-битном режиме используются для хранения селектора, который локализует дескриптор сегмента в виртуальной памяти. Более подробно, адресация виртуальной памяти будет рассмотрена в одной из следующих глав.

2.5 Типы данных

Микропроцессор аппаратно поддерживает следующие основные типы данных:

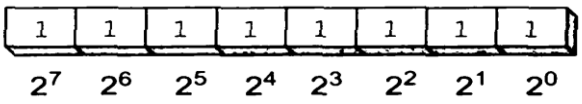
Бит - Основной единицей информации, хранящейся в компьютере это бит. Бит это разряд двоичного числа, может иметь значения 0 или 1.

Тетрада - Четыре последовательно расположенных битов, пронумерованных от 0 до 3, при этом бит 0 является самым младшим значащим битом

1	0	1	1
3	2	1	0

Байт (Byte) — восемь последовательно расположенных битов, пронумерованных от 0 до 7, при этом бит 0 является самым младшим значащим битом. Каждый бит соответствует значению числа 2^n.

0	1	1	1	0	1	0	1
7	6	5	4	3	2	1	0



Слово (Word) — последовательность из двух байт, имеющих последовательные адреса.

Размер слова — 16 бит; биты в слове нумеруются от 0 до 15. Байт, содержащий нулевой бит, называется младшим байтом, а байт, содержащий 15-й бит - старшим байтом. Микропроцессоры Intel имеют важную особенность — младший байт всегда хранится по меньшему адресу. Адресом слова считается адрес его младшего байта. Адрес старшего байта может быть использован для доступа к старшей половине слова.

1	1	1	0	0	0	0	1	1	0	0	1	1	0	0	1
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Двойное слово (Double Word) — последовательность из четырех байт (32 бита), расположенных по последовательным адресам. Нумерация этих бит производится от 0 до 31. Слово, содержащее нулевой бит, называется младшим словом, а слово, содержащее 31-й бит, - старшим словом. Младшее слово хранится по меньшему адресу. Адресом двойного слова считается адрес его младшего слова. Адрес старшего слова может быть использован для доступа к старшей половине двойного слова.

Пример. *Двойное слово* 12 34 56 78h, со смещением (offset) 0000, будет сохранен в памяти как 78 56 34 12, т.е. младший байт 78h по смещению 0000, старший байт 12h по смещению 0003.

0000:	78
0001:	56
0002:	34
0003:	12

Учетверенное слово (Quadword) — последовательность из восьми байт (64 бита), расположенных по последовательным адресам. Нумерация бит производится от 0 до 63. Двойное слово, содержащее нулевой бит, называется младшим двойным словом, а двойное слово, содержащее 64-й бит, — старшим двойным словом. Младшее двойное слово хранится по меньшему адресу. Адресом учетверенного слова считается адрес его младшего двойного слова. Адрес старшего двойного слова может быть использован для доступа к старшей половине учетверенного слова.

Десять байт (Tenbyte) — последовательность из 10 байт (80 бит), расположенных по последовательным адресам. Нумерация бит производится от 0 до 79. байт, Байт содержащий нулевой бит, называется младшим байтом, а байт, содержащий 79-й бит, — старшим байтом. Младший байт хранится по меньшему адресу.

2.5 Директивы определения данных

Директивы определения данных в ассеблере следующие:

Директивы **BYTE** и **SBYTE** - для определения байта.

Директивы BYTE и SBYTE определяют выражение или константу, принимающую значение из диапазона:

- для чисел со знаком –128...+127;
- для чисел без знака 0...255;
- символьную строку из одного или более символов. Строка заключается в кавычки. В этом случае определяется столько байт, сколько символов в строке.

```
value1   BYTE    'A' ; символ ASCII
value2   BYTE     0 ;  byte без знака
value3   BYTE   255 ;  byte без знака
value4   SBYTE  −128 ;  byte со знаком
```

```
value5   SBYTE    +127 ; byte со знаком
value6   BYTE    ?  ; байт с неопределенным значением
```

Также можно использовать директиву **db** (define byte):

Пример:

```
alfa      DB  65, 72h, 75o, 11011b, 11h+22h, 0ach
          DB  -65, 'a', 'abc'
```

В памяти, начиная с символического адреса *alfa*, будет сформирована последовательность байтов (значения в шестнадцатеричном коде) :

41	72	3d	1b	33	ac	bf	61	61	62	63	
alfa +0	+1	+2	+3	+4							

Двоичное значение 11011b будет расположена по адресу alfa+3.

Определение символьной строки:

```
privet1   BYTE   "Good afternoon",0
greeting2   BYTE   'Good night',0
```

Применение оператора DUP (duplicate):

```
BYTE   20    DUP(0) ; 20 bytes, с нулевым содержанием
BYTE   20    DUP(?) ; 20 bytes, содержание неопределено
BYTE    4    DUP("STACK") ; 20 bytes: "STACKSTACKSTACKSTACK"
```

WORD şi SWORD — резервирование памяти для данных размером 2 байта (без и со знаком).

Этими директивами можно задавать следующие значения:

- выражение или константу, принимающую значение из диапазона:
- для чисел со знаком –32 768...32 767;
- для чисел без знака 0...65 535;
- выражение, занимающее 16 или менее бит, в качестве которого может выступать смещение в 16-битовом сегменте или адрес сегмента;
- или 2-байтовую строку, заключенная в кавычки.

```
word1   WORD     65535 ;  без знака
word2   SWORD   -32768 ; со знаком
ωορδ3   ΩOPΔ ?              ; ⟩⦃∑⌊⌋□∑™∑⌊∑⦃⦃⌋⌈⌋⦃◊⌋∑⦃⦁∑⌈
```

Также можно использовать директиву **dw** (define word):

```
beta    DW     4567h,   0bc4ah,   1110111011b,   2476o
        DW    -7683,    7683,    'ab'
```

В памяти с адреса (смещения, offset) “beta” будут расположены байты:

67	45	4a	bc	bb	03	3e	05	fd	e1	03	e1	62	61
beta		+2		+4		+6		+8				+12	

Восьмеричное значение 2476o будет рассположено в памяти со смещением beta +6.

DWORD и SDWORD — резервирование памяти для данных размером 4 байта (без и со знаком).

Директивами можно задавать следующие значения:

- выражение или константу, принимающую значение из диапазона:

- для чисел со знаком $-2^{31}...+2^{31}-1$;
- для чисел без знака $0...2^{32}-1$;
- относительное или адресное выражение, состоящее из 16-битового адреса сегмента и 16-битового смещения;
- вещественное число одинарной точности.

```
val1   DWORD    12345678h ;  без  знака
val2   SDWORD   −21474836 ; со знаком
val3   DWORD  20  DUP(?) ; без  знака
```

Также можно использовать директиву **dd** (*Define Double word*):

```
val1   DD   12345678h, −21474836
```

QWORD и DQ (*Define Quad – word*) — резервирование памяти для данных размером 8 байт.

Директивами можно задавать следующие значения:

- выражение или константу, принимающую значение из диапазона:
- для чисел со знаком $-2^{63}…+2^{63}-1$;
- для чисел без знака $2^{64}-1$;
- вещественное число двойной точности.

```
quad1 QWORD 1234567812345678h

quad2 DQ 1234567812345678h
```

TBYTE и DT — резервирование памяти для данных размером 10 байт.

Директивами можно задавать следующие значения:

- упакованную десятичную константу в диапазоне 0...99 999 999 999 999 999 999.
- вещественное число повышенной точности.

```
intVal    TBYTE   80000000000000001234h
intVal1    DT  80000000000000001234h
```

Определение вещественных чисел.

Вещественные числа определяются следующими директивами:

- REAL4 – определение 32 разрядных вещественных чисел;
- REAL8 – определение 64 разрядных вещественных чисел двойной точности;
- REAL10 – определение 80 разрядных вещественных чисел повышенной точности.

```
rVal1   REAL4   -1.2
rVal2   REAL8   3.2E-260
rVal3   REAL10   4.6E+4096
ShortArray   REAL4 20 DUP(0.0)
```

Вещественные числа определяются также следующими директивами:

```
rVal1   DD   -1.2
rVal2   DQ   3.2E-260
rVal3   DT   4.6E+4096
```

Диапазон значений может быть:

REAL4 - 1.18×10^{-38} до 3.40×10^{38}

REAL8 - 2.23×10^{-308} до 1.79×10^{308}

REAL10 - $3.37x10^{-4932}$ до $1.18x10^{4932}$

Используется еще следующие типы:

- Неупакованный двоично-десятичный тип — байтовое представление десятичной цифры от 0 до 9. Неупакованные десятичные числа хранятся как байтовые значения без знака по одной цифре в каждом байте. Значение цифры определяется младшим полубайтом.
- Упакованный двоично-десятичный тип представляет собой упакованное представление двух десятичных цифр от 0 до 9 в одном байте. Каждая цифра хранится в своем полубайте. Цифра в старшем полубайте (биты 4–7) является старшей.

```
BCD1  byte  03h, 06h ; неупакованный двоично-десятичный тип
BCD2  byte  33h, 96h ; упакованный двоично-десятичный тип
```

Важно уяснить себе порядок размещения данных в памяти. Он напрямую связан с логикой работы микропроцессора с данными. Микропроцессоры Intel требуют следования данных в памяти по принципу: младший байт по младшему адресу.

2.7 Система команд MASM

Система машинных команд является важнейшей частью архитектуры компьютера, так как с их помощью производится непосредственное управление работой процессора. К примеру, система команд процессора Pentium IV содержит более 300 машинных команд. С появлением каждой новой модели процессора количество команд, как правило, возрастает, отражая архитектурные новшества данной модели по сравнению с предшествующими.

В данной главе рассмотрим систему команд Intel (x86).

Систему команд классифицируют по функциональному признаку:

- команды пересылки данных;
- арифметические и логические команды;
- цепочечные команды;
- команды передачи управления;
- команды для обработки прерываний;
- команды управления состоянием микропроцессора.

Стандартный формат команды ассемблера x86 состоит из четырех основных частей:

[*label*:] *mnemonic* [*operands*] [;*comment*]

- необязательной метки;
- мнемоники команды, которая присутствует всегда;
- одного или нескольких операндов (как правило. они присутствуют в любой команде, хотя есть ряд команд, для которых операнды не требуются);
- необязательного комментария.

Команды могут содержать ноль, один, два или три операнда. Опустим необязательные части метки и комментария:

mnemonic
mnemonic [*destination*]
mnemonic [*destination*],[*source*]
mnemonic [*destination*],[*source-1*],[*source-2*]

Есть три основных типа операндов, которые могут встречаться в любой команде:

- непосредственно заданное значение (immediate);
- регистр (register) – значение находится в одном из регистров микропроцессора;
- память (memory).

2.7.1 Команды пересылки данных, двоичная арифметика, арифметика BCD чисел

Команда **MOV** (MOVe operand) - Пересылка операнда

Схема команды: mov операнд1,операнд2

Назначение: пересылка данных между регистрами или регистрами и памятью.

Алгоритм работы:

копирование операнда2 в операнд1.

Состояние флагов после выполнения команды: выполнение команды не влияет на флаги.

В команде mov могут использоваться самые разные операнды. Кроме того, необходимо учитывать следующие правила и ограничения:

- Оба операнда должны иметь одинаковую длину.
- В качестве одного из операндов обязательно должен использоваться регистр (т.е. пересылки типа "память-память" в команде **mov** не поддерживаются).
- В качестве приемника нельзя указывать регистры CS, IP, EIP или RIP.
- Нельзя переслать непосредственно заданное значение в сегментный регистр.

Ниже приведены варианты использования команды mov с разными операндами (кроме сегментных регистров):

MOV *reg,reg,*
MOV *mem,reg,*
MOV *reg,mem,*
MOV *mem,imm,*
MOV *reg, imm,*

где **reg/r** (register) – регистр,
mem/m (memory) – ячейка памяти,
imm (immediate) – непосредственное значение, число.

Примеры. reg16 – указывает 16-битный регистр**, mem8 –** 8-битную ячейку памяти, **r8 –** указывает 8-битный регистр, **m32 -** 32-битную ячейку памяти.

Примеры:

```
.data
var1 WORD ?
var2 WORD ?
.code
mov ax,var1
mov var2,ax
```

```
.data
oneByte BYTE 78h
oneWord WORD 1234h
oneDword DWORD 12345678h
.code
mov eax,0 ; EAX = 00000000h
mov al,oneByte ; EAX = 00000078h
mov ax,oneWord ; EAX = 00001234h
mov eax,oneDword ; EAX = 12345678h
mov ax,0 ; EAX = 12340000h
```

```
.data
alfa dw  1234h
beta db 56h
.code
mov    bl,al
```

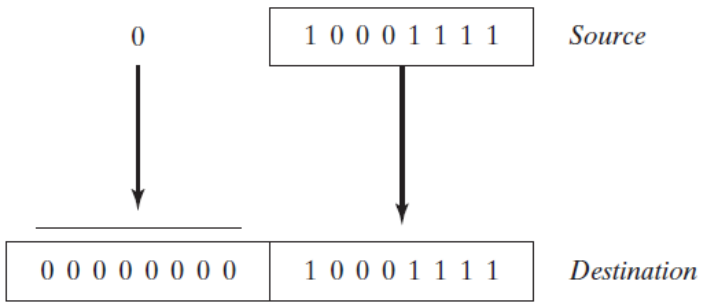
```
 mov     bx,ds
mov  ax, alfa;          пересылка содержимого  по смещению alfa в ax
mov  bx, offset beta;   пересылка эффективного адреса  beta в bx
mov  al, 75h;
mov  cx, [100];         пересылка содержимого  адреса  100 в cx
mov  [di], bx;  пересылка содержимого  регистра bx по адресу в регистре di
mov byte ptr  alfa , [bx]; пересылка содержимого по адресу в регистре bx в байт по
                    ;  смещению alfa
```

Команда **MOVZX** *(Move With Zero-Extend,* или *переместить и дополнить нулями)* копирует содержимое исходного операнда в больший по размеру *регистр* приемник данных. При этом оставшиеся неопределенными биты регистра-примника (как правило, старшие 16 или 24 бита) сбрасываются в ноль. Эта команда используется только при работе с беззнаковыми целыми числами. Существует три варианта команды movzx:

MOVZX *reg32,reg/mem8,*
MOVZX *reg32,reg/mem16,*
MOVZX *reg16,reg/mem8,*
где числа 8, 16, 32 указывают разрядность регистра или ячейки памяти.

Примеры:

```
                                    .data
                            byteVal BYTE 10001111b
                                    .code
movzx ax,byteVal ; AX = 0000000010001111b
```




```
.data
byte1 BYTE 9Bh
word1 WORD 0A69Bh
.code
movzx eax,word1 ; EAX = 0000A69Bh
movzx edx,byte1 ; EDX = 0000009Bh
movzx cx,byte1 ; CX = 009Bh
```

Команда **MOVSX** ***(Move With Sign-Extend,*** или ***Переместить и дополнить знаком)*** копирует содержимое исходного операнда в больший по размеру ***регистр*** получателя данных, также как и команда **movzx**. При этом оставшиеся неопределенными биты регистра-получателя (как правило, старшие 16 или 24 бита) заполняются значением знакового бита исходного операнда. Эта команда используется только при работе со знаковыми целыми числами. Существует три варианта команды MOVSX:

MOVZX *reg32,reg/mem8*
MOVZX *reg32,reg/mem16*
MOVZX *reg16,reg/mem8*

Пример:

```
mov bx,0A69Bh
movsx eax,bx ; EAX = FFFFA69Bh
movsx edx,bl ; EDX = FFFFFF9Bh
movsx cx,bl ; CX = FF9Bh
```

Команда **XCHG** *(Exchange Data,* или *Обмен данными)* позволяет обменять содержимое двух операндов. Существует три варианта команды XCHG:

XCHG *reg, reg*
XCHG *reg,mem*
XCHG *mem,reg*

Для операндов команды **XCHG** нужно соблюдать те же правила и ограничения, что и для операндов команды **MOV**, за исключением того, что операнды команды **XCHG** не могут быть непосредственно заданными значениями.

Пример:

```
xchg        al, ah
xchg        alfa, ax
xchg       sir [si], bx
xchg eax,ebx ; exchange 32-bit regs
```

Чтобы поменять содержимое двух переменных, расположенных в памяти, необходимо воспользоваться промежуточным регистром и двумя дополнительными командами MOV:

```
mov         reg, op1
xchg        reg, op2
mov         op2, reg
```

Пример программы:

```
.data
val1 WORD 1000h
val2 WORD 2000h
arrayB BYTE 10h,20h,30h,40h,50h
arrayW WORD 100h,200h,300h
arrayD DWORD 10000h,20000h
.code
main PROC
; Demonstrating MOVZX instruction:
mov bx,0A69Bh
movzx eax,bx ; EAX = 0000A69Bh
movzx edx,bl ; EDX = 0000009Bh
movzx cx,bl ; CX = 009Bh
; Demonstrating MOVSX instruction:
mov bx,0A69Bh
movsx eax,bx ; EAX = FFFFA69Bh
movsx edx,bl ; EDX = FFFFFF9Bh
mov bl,7Bh
movsx cx,bl ; CX = 007Bh
; Memory-to-memory exchange:
mov ax,val1 ; AX = 1000h
xchg ax,val2 ; AX=2000h, val2=1000h
mov val1,ax ; val1 = 2000h
; Direct-Offset Addressing (byte array):
mov al,arrayB ; AL = 10h
mov al,[arrayB+1] ; AL = 20h
```

```
mov al,[arrayB+2] ; AL = 30h
; Direct-Offset Addressing (word array):
mov ax,arrayW ; AX = 100h
mov ax,[arrayW+2] ; AX = 200h
; Direct-Offset Addressing (doubleword array):
mov eax,arrayD ; EAX = 10000h
mov eax,[arrayD+4] ; EAX = 20000h
mov eax,[arrayD+4] ; EAX = 20000h
```

XLAT (transLATe Byte from table) - **Преобразование байта**

Схема команды: **xlat адрес_таблицы_байтов**

Назначение: подмена байта в регистре **al** байтом из последовательности (таблицы) байтов в памяти.

Алгоритм работы:

- вычислить адрес, равный **ds:bx+(al);**
- выполнить замену байта в регистре al байтом из памяти по вычисленному адресу.

Несмотря на наличие операнда адрес_таблицы_байтов в команде **xlat**, адрес последовательности байтов, из которой будет осуществляться выборка байта для подмены в регистре *al*, должен быть предварительно загружен в пару **ds:bx**. Команда *xlat* допускает замену сегмента.

Состояние флагов после выполнения команды: выполнение команды не влияет на флаги

Применение:

Команду **xlat** можно использовать для выполнения перекодировок символов. Для формирования адреса таблицы в регистрах **bx** можно использовать команду **lea** или оператор ассемблера **offset** в команде **mov**.

Пример:

```
table   db      'abcdef'
int     db      0       ;значение индекса
...
          mov     al,3
         lea     bx,table
         xlat            ;(al)='c'
```

IN (INput operand from port) - **Ввод операнда из порта**

Схема команды: **in аккумулятор, номер_порта**

Назначение: ввод значения из порта ввода-вывода.

Алгоритм работы:

Передает байт, слово, двойное слово из порта ввода-вывода в один из регистров al/ax/eax.

Состояние флагов после выполнения команды: выполнение команды не влияет на флаги.

Применение:

Команда применяется для прямого управления оборудованием компьютера посредством портов. Номер порта задается вторым операндом в виде непосредственного значения или значения в регистре **dx**. Непосредственным значением можно задать порт с номером в диапазоне 0-255. При использовании порта с большим номером используется регистр **dx**. Размер данных определяется размерностью первого операнда и может быть байтом, словом, двойным словом.

OUT (OUT operand to port) **- Вывод операнда в порт**

Схема команды: **out номер_порта, аккумулятор**

Назначение: вывод значения в порт ввода-вывода.

Алгоритм работы:

Передать байт, слово, двойное слово из регистра al/ax/eax в порт, номер которого определяется первым операндом.

Состояние флагов после выполнения команды: выполнение команды не влияет на флаги

Применение:

Команда применяется для прямого управления оборудованием компьютера посредством портов. Номер порта задается первым операндом в виде непосредственного значения или значения в регистре dx. Непосредственным значением можно задать порт с номером в диапазоне 0...255. Для указания порта с большим номером используется регистр dx. Размер данных определяется размерностью второго операнда и может быть байтом, словом или двойным словом.

```
out     64h,al
```

Пример:

```
in al,3Ch               ; input byte from port 3Ch
out 3Ch,al              ; output byte to port 3Ch
mov dx, portNumber ; DX can contain a port number
in ax,dx                ; input word from port named in DX
out dx,ax                ; output word to the same port
in eax,dx               ; input doubleword from port
out dx,eax              ; output doubleword to same port
```

LEA (Load Effective Address) **- Загрузка эффективного адреса**

Схема команды: **lea приемник, источник**

Назначение: **получение эффективного адреса (смещения) источника.**

Алгоритм работы:

- в регистр приемник загружается 16-битное значение смещения операнда источник;
- выполнение команды не влияет на флаги.

Применение:

Данная команда является альтернативой оператору ассемблера **offset**. В отличие от offset команда lea допускает индексацию операнда, что позволяет более гибко организовать адресацию операндов.

```
; загрузить в регистр bx адрес пятого элемента массива mas
.data
mas     db      10 dup (0)
.code
...
      mov     di,4
      lea     bx,mas[di]
;или
      lea     bx,mas[4]
;или
      lea     bx,mas+4
```

Команды **LAHF и SAHF**

Команда **LAHF** *(Load Status F!ags lnto AH,* или загрузить флаги состояния в регистр AH) позволяет загрузить в регистр AH младший байт регистра флагов EFLAGS. При этом в регистр AH копируются следующие флаги состояния: SF (флаг знака), ZF, (флаг нуля), AF (флаг служебного переноса), PF (флаг четности) и CF (флаг переноса). С помошью этой команды можно легко сохранить содержимое регистра флагов в переменной для дальнейшего анализа:

```
.data
saveflags BYTE ?
.code
lahf     ; Загрузить флаги в регистр AH
mov saveflags,ah ; Сохранить флаги в переменной
```

Команда **SAHF** *(Store AH lnto Status Flags,* или записать регистр АН во флаги) помещает содержимое регистра АН в младший байт регистра флагов EFLAGS. Например, вы можете восстановить сохраненное ранее в переменной значение флагов:

```
mov ah,saveflags ; загрузим в регистр АН сохраненное ранее
                 ; значение регистра флагов
sahf     ; скопируем его в младший байт регистра EFLAGS
```

Команды **PUSH и POP**

Команда **PUSH** помешает в стек значение 16- или 32-разрядного операнда, уменьшая перед этим значение регистра ESP, соответственно, на 2 или 4. Существует три формата команды PUSH:

```
PUSH  r/m16
PUSH r/m32
PUSH imm16/ imm32
```

Команда **POP**

Команда **POP** копирует содержимое вершины стека, на которую указывает регистр ESP, в 16- или 32-разрядный операнд, указанный в команде, а затем прибавляет к регистру ESP, соответственно, число 2 или 4. Существует два формата команды **POP**:

```
POP r/m16
POP r/m32
```

Команды **PUSHFD и POPFD**

Команда PUSHFD помещает в стек значение 32-разрядного регистра флагов процессора EFLAGS, а команда POPFD выполняет обратную операцию, т.е. восстанавливает значение регистра EFLAGS из стека. У этих команд операндов нет:

```
pushfd
popfd
```

В реальном режиме для сохранения в стеке значения 16-разрядного регистра флагов FLAGS используется команда PUSHF, а для выполнения обратной операции - команда POPF.

Пример:

```
.data
saveFlags DWORD ?
.code
pushfd ;  Сохранить в стеке регистр флагов EFLAGS
pop saveFlags ; Скопировать значение регистра
              ;флагов EFLAGS из стека в переменную
```

Команды **PUSHAD, PUSHA, POPAD и POPA**

Команда PUSHAD сохраняет в стеке значение всех 32-разрядных регистров общего назначения в следующем порядке: EAX, ECX, EDX, EBX, ESP (то значение, которое было до выполнения команды), EBP, ESI и EDI. Команда POPAD выполняет обратную операцию, т.е. восстанавливает из стека значения указанных регистров в обратном порядке. По аналогии, команда PUSHA сохраняет в стеке значение всех 16-разрядных регистров общего назначения в следующем порядке: AX, CX, OX, BX, SP (то значение, которое было до выполнения команды), BP, SI и DI. Команда POPA выполняет обратную операцию, т.е. восстанавливает из стека значения указанных регистров в обратном порядке.

Команда PUSHAD обычно используется в начале процедуры или фрагмента кода, в котором модифицируется много 32-разрядных регистров общего назначения. Для восстановления первоначального значения этих регистров в конце процедуры или фрагмента кода используется команда POPAD. Ниже приведен фрагмент кода:

```
MySub PROC
pushad ; Сохраним в стеке регистры общего назначения
        .
        .
mov eax, … .
mov edx, … .
mov ecx, … .
.
.
popad ; Восстановим значения регистров
ret
MySub ENDP
```

2.7.2 Двоичная арифметика

Выполнение этих команд влияет на флаги регистра FLAGS.

ADD (ADDition) **- Сложение**

Схема команды: **add операнд1, операнд2**

Назначение: сложение двух операндов источник и приемник размерностью 8, 16, 32 и 64 бита.

Алгоритм работы:

- сложить операнды **операнд1** и **операнд2**;
- записать результат сложения в **операнд1**;
- установить флаги.

Выполнение команды влияет на состояние флагов: CF (флаг переноса), ZF (флаг нуля), SF (флаг знака), OF (флаг переполнения), AF (флаг служебного переноса), PF (флаг четности).

Применение:

Для операндов команды ADD нужно соблюдать те же правила и ограничения, что и для операндов команды MOV. Команда add используется для сложения двух целочисленных операндов. Результат сложения помещается по адресу первого операнда. Если результат сложения выходит за границы операнда приемник (возникает переполнение), то учесть эту ситуацию следует путем анализа флага **cf** и последующего возможного применения команды **adc**.

Ex.:

```
add     ax, 5
add     bl, 5
add     ax, bx
add     word ptr [bx], 75
add     alfa, ax
add     alfa, 5
add     byte ptr [si], 75
add     byte ptr  alfa, 75
```

```
.data
var1 DWORD 10000h
var2 DWORD 20000h
.code
mov eax,var1 ; EAX = 10000h
add eax,var2 ; EAX = 30000h
```

```
.data
sum qword 0
```

```
.code
        mov    rax,5
        add    rax,6
        mov   sum,rax
```

ADC (Addition with Carry) **- Сложение с переносом**

Схема команды: **adc операнд1, операнд2**

Назначение: сложение двух операндов с учетом переноса из младшего разряда.

Алгоритм работы:

- сложить два операнда;
- поместить результат в **операнд1**: **операнд1**= **операнд1**+ **операнд2**+**cf**;
- в зависимости от результата установить флаги.

Выполнение команды влияет на состояние флагов: OF SF ZF AF PF CF

Применение:

Команда adc используется при сложении длинных двоичных чисел. Ее можно использовать как самостоятельно, так и совместно с командой add. При совместном использовании команды adc с командой add сложение младших байтов/слов/двойных слов осуществляется командой add, а уже старшие байты/слова/двойные слова складываются командой adc, учитывающей переносы из младших разрядов в старшие. Таким образом, команда adc значительно расширяет диапазон значений складываемых чисел.

```
.data
sl1    dd     01fe544fh
sl2    dd     005044cdh
rez    dd     0
.code
...
mov    ax,sl1
add    ax,sl2  ;сложение младших слов слагаемых
mov    rez,ax
mov    ax,sl+2
adc    ax,sl2+2       ;сложение старших слов слагаемых плюс cf
mov    rez+2,ax
```

Также можно сложить два целых 32-битных числа (FFFFFFFFh + FFFFFFFFh), сохраняя результат в паре регистров EDX:EAX: 00000001FFFFFFFEh:

```
mov edx,0
mov eax,0FFFFFFFFh
add eax,0FFFFFFFFh
adc edx,0
```

SUB (SUBtract) **- Вычитание**

Схема команды: **sub операнд1, операнд2**

Назначение: целочисленное вычитание.

Алгоритм работы:

- выполнить вычитание **операнд1=операнд1-операнд2**;
- установить флаги.

Выполнение команды влияет на состояние флагов: OF SF ZF AF PF CF

Применение:

Для операндов команды SUB нужно соблюдать те же правила и ограничения, что и для операндов команды MOV. Команда sub используется для выполнения вычитания целочисленных операндов или для вычитания младших частей значений многобайтных операндов.

```
.data
var1 DWORD 30000h
```

```
var2 DWORD 10000h
.code
mov eax,var1 ; EAX = 30000h
sub eax,var2 ; EAX = 20000h
```

SBB (SuBtract with Borrow) **- Вычитание с заемом**

Схема команды: **sbb операнд1, операнд2**

Назначение: целочисленное вычитание с учетом результата предыдущего вычитания командами sbb и sub (по состоянию флага переноса cf).

Алгоритм работы:

- выполнить сложение **операнд2=операнд2+(cf);**
- выполнить вычитание **операнд1=операнд1-операнд2;**

Выполнение команды влияет на состояние флагов: OF SF ZF AF PF CF

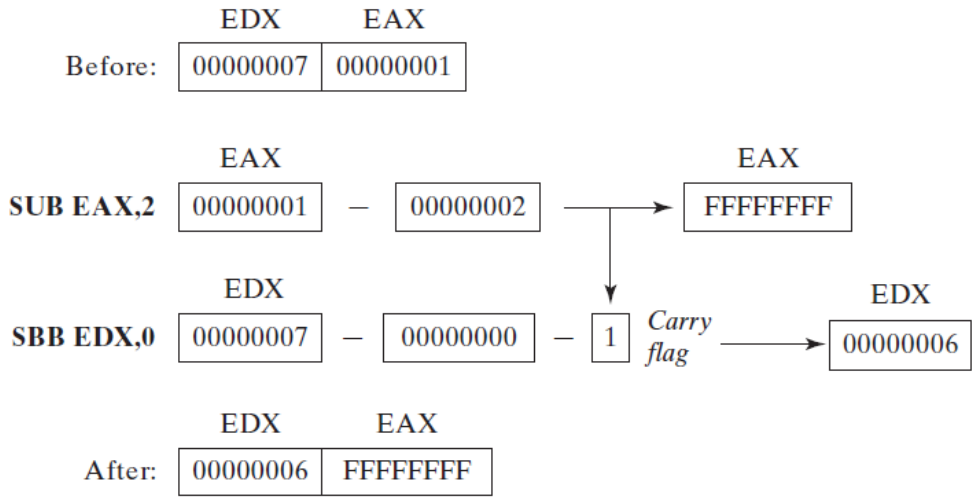
Применение:

Команда **sbb** используется для выполнения вычитания старших частей значений многобайтных операндов с учетом возможного предыдущего заема при вычитании младших частей значений этих операндов.

```
Ex.     Op1   dword 12345678h
        Op2   dword 0abcdef78h
        Rez   dword ?
        . . . . . .
        mov   ax, word ptr op1
        sub   ax, word ptr op2
        mov   word ptr rez, ax
        mov   ax, word ptr op1 + 2
        sbb   ax, word ptr op2 + 2 ; учитывается возможный заем
        mov   word ptr rez + 2, ax
```

Ex.. Вычитание из 64 битного числа 32 битного. Пара регистров EDX:EAX загружактся значением 0000000700000001h и вычитаем 2. На первом этапе – вычитание младших частей (EAX-00000001h), т. е. из 1 вычитаем 2, что приведет к заему с установлением флага Carry.

```
mov edx,7 ; старшая часть
mov eax,1 ; младшая часть
sub eax,2 ; вычитаем  2
sbb edx,0 ; вычитаем старшую часть
```



INC (INCrement operand by 1) **- Увеличить операнд на 1**

Схема команды: **inc операнд**

Назначение: увеличение значения операнда в памяти или регистре на 1.
Алгоритм работы: команда увеличивает операнд на единицу.
Выполнение команды влияет на состояние флагов: OF SF ZF AF PF
Применение:

Команда используется для увеличения значения байта, слова, двойного слова в памяти или регистре на единицу. При этом команда не воздействует на флаг cf.

Ex:

```
inc     alfa
inc     bl
inc     eax
inc     rbx
inc     word ptr [bx] [si]
```

DEC (DECrement operand by 1) **- Уменьшение операнда на единицу**

Схема команды: **dec операнд**

Назначение: уменьшение значения операнда в памяти или регистре на 1.
Алгоритм работы: команда вычитает 1 из операнда.
Выполнение команды влияет на состояние флагов: OF SF ZF AF PF
Применение:

Команда dec используется для уменьшения значения байта, слова, двойного слова в памяти или регистре на единицу. При этом заметьте то, что команда не воздействует на флаг cf.

```
mov     al,9
...
dec     al      ;al=8
```

NEG (NEGate operand) **- Изменить знак операнда**

Схема команды: **neg операнд**

Назначение: изменение знака (получение двоичного дополнения) операнда.

Синтаксис

Алгоритм работы:

- выполнить вычитание (0 – операнд) и поместить результат на место операнда;
- если операнд=0, то его значение не меняется.

Состояние флагов после выполнения команды (если результат нулевой):

OF	SF	ZF	AF	PF	CF
r	r	r	r	r	0

Состояние флагов после выполнения команды (если результат ненулевой):

OF	SF	ZF	AF	PF	CF
r	r	r	r	r	1

Применение:

Команда используется для формирования двоичного дополнения операнда в памяти или регистре. Операция двоичного дополнения предполагает инвертирование всех разрядов операнда с последующим сложением операнда с двоичной единицей. Если операнд отрицательный, то операция neg над ним означает получение его модуля.

```
mov     al,2
neg     al      ;al=0feh — число -2 в дополнительном коде
```

CMP (CoMPare operands) **- Сравнение операндов**

Схема команды: **cmp операнд1, операнд2**

Назначение: сравнение двух операндов.
Алгоритм работы:

- выполнить вычитание (операнд1-операнд2);
- в зависимости от результата установить флаги, операнд1 и операнд2 не изменять (то есть результат не запоминать).

Выполнение команды влияет на состояние флагов: OF SF ZF AF PF CF

Применение:

Данная команда используется для сравнения двух операндов методом вычитания, при этом операнды не изменяются. По результатам выполнения команды устанавливаются флаги **Zero** и **Carry** в соответствии с таблицей:

CMP Results	ZF	CF
Destination < source	0	1
Destination > source	0	0
Destination = source	1	0

Команда **cmp** применяется с командами условного перехода.

```
len     equ     10
...
cmp     ax,len
jne     m1      ;переход если (ax)<>len
jmp     m2      ;переход если (ax)=len
```

CBW (Convert Byte to Word) **- Преобразование байта в слово**

Схема команды: **cbw**

Назначение: расширение операнда со знаком.

Алгоритм работы:

cbw — при работе команда использует только регистры **al** и **ax**:

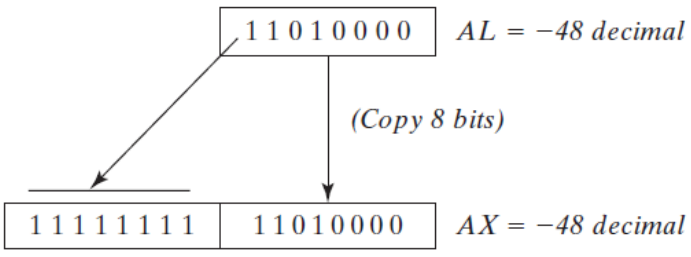
- анализ знакового бита регистра al:
- если знаковый бит al=0, то ah=00h;
- если знаковый бит al=1, то ah=0ffh.

Т.е. содержимое регистра AL (8 бит) расширяется до регистра AX (16 бит).

Состояние флагов после выполнения команды: выполнение команды не влияет на флаги.

Применение:

Данные команды используются для приведения операндов к нужной размерности с учетом знака. Такая необходимость может, в частности, возникнуть при программировании арифметических операций.



Ex.:

```
a       sbyte   -75
b       sword   -188
c       sword   ?
. . . . . .
mov     al, a
cbw     ;               преобразование байта в слово
add     ax, b
mov     c, ax
. . . . . .
```

CWD (Convert Word to Double word) **- Преобразование слова в двойное слово.**

Схема команды: **cwd**

Назначение: расширение слова со знаком до размера двойного слова со знаком.

Алгоритм работы: копирование значения старшего бита регистра **ax** во все биты регистра **dx.**

Т.е. содержимое регистра AX (16 бит) расширяется до пары регистров DX:AX (32 бит).

Состояние флагов после выполнения команды: выполнение команды не влияет на флаги.

Применение:

Команда **cwd** используется для расширения значения знакового бита в регистре **ax** на биты регистра **dx**. Данную операцию, в частности, можно использовать для подготовки к операции деления, для которой размер делимого должен быть в два раза больше размера делителя, либо для приведения операндов к одной размерности в командах умножения, сложения, вычитания.

Ex. Вычитание двух операндов – 32 бита (dw) и 16 бит(sw).

```
dw      dword  12345678h
sw      word   0abcdh
rez     dword  ?
. . . . . .
mov     ax, so
cwd             ; operand находится в паре DX : AX
mov     bx, ax
mov     ax, word  ptr do
sub     ax, bx
mov     word  ptr  rez, ax
mov     ax, word  ptr  do + 2
sbb     ax, dx  ; возможный заем
mov     word  ptr  rez + 2
```

Команда **CDQ (convert doubleword to quadword)**

Расширения значения знакового бита из EAX [EAX_{31}] и заполнение им всего регистра EDX, в итоге получается 64 битное значение в паре регистров EDX:EAX, т.е.:

Если знак [EAX_{31}] = 0, тогда [EDX] ☐ 00000000h,

Если [EAX_{31}] = 1, тогда [edx] ☐ 0ffffffffh.

Т.е. содержимое регистра EAX (32 бита) расширяется до пары регистров EDX:EAX (64 бита).

Команда не имеет операндов, выполнение команды не влияет на флаги.

Команда **MUL** (MULtiply) **- Умножение целочисленное без учета знака**

Схема команды: **mul операнд**

Назначение: операция умножения двух целых чисел без учета знака.

Алгоритм работы:

Команда выполняет умножение двух операндов без учета знаков. Алгоритм зависит от формата операнда команды и требует явного указания местоположения только одного сомножителя, который может быть расположен в памяти или в регистре. Местоположение второго сомножителя фиксировано и зависит от размера первого сомножителя.

Синтаксис:

```
MUL reg8/mem8       ; reg8/mem8 * AL= AX
MUL reg16/mem16     ; reg16/mem16 * AX= DX:AX
MUL reg32/mem32     ; reg32/mem 32 * EAX  = EDX:EAX
MUL reg64/mem64     ; reg64/mem 64 * RAX  = RDX:RAX
```

Сокращения **reg/mem** обозначают тип операнда ***reg (register)*** - регистр микропроцессора, ***mem (memory)*** - ячейка памяти, а числа 8, 16, 32, 64 указывают разрядность регистра или ячейки памяти.

Если операнд, указанный в команде имеет разрядность 8 бит (***reg8/mem8***), то второй сомножитель должен располагаться в **AL**, а результат помещается в **AX**.

Если операнд, указанный в команде имеет разрядность 16 бит (***reg16/mem16***), то второй сомножитель должен располагаться **AX**, а результат помещается в паре регистров **DX:AX.**

Если операнд, указанный в команде имеет разрядность 32 бита (***reg32/mem32***), то второй сомножитель должен располагаться **EAX**, а результат помещается в паре регистров **EDX:EAX.**

Если операнд, указанный в команде имеет разрядность 64 бита (***reg64/mem64***), то второй сомножитель должен располагаться **RAX**, а результат помещается в паре регистров **RDX: RAX.**

Состояние флагов после выполнения команды (если старшая половина результата нулевая):

OF	SF	ZF	AF	PF	CF
0	?	?	?	?	0

Состояние флагов после выполнения команды (если старшая половина результата ненулевая):

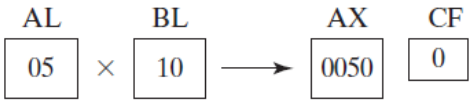
OF	SF	ZF	AF	PF	CF
1	?	?	?	?	1

Если умножаются байты:

[AX] ☐ [AL] * [*reg/mem8*]

```
mov al,5h
mov bl,10h
mul bl   ; bl это – reg8, результат в AX = 0050h, CF = 0
```

Следующий рисунок показывает взаимодействие между регистрами:



Если умножаются операнды разрядностью 16 бит:

[DX:AX] ☐ [AX] * [*reg/mem16*]

```
.data
val1 WORD 2000h
val2 WORD 0100h
.code
mov ax,val1 ; AX = 2000h
mul val2    ; val2 это – mem16, результат в DX:AX = 00200000h, CF = 1
```

Следующий рисунок показывает взаимодействие между регистрами:

Если умножаются операнды разрядностью 32 бита:

[EDX:EAX] ☐ [EAX] * [*reg/mem32*]

```
mov eax,12345h
mov ebx,1000h
mul ebx    ; ebx это – reg32, результат в EDX:EAX = 0000000012345000h, CF = 0
```

Следующий рисунок показывает взаимодействие между регистрами:

Если умножаются операнды разрядностью 32 бита:

[RDX:RAX] ☐ [RAX] * [*reg/mem64*]

```
mov rax,0FFFF0000FFFF0000h
mov rbx,2
mul rbx        ; rbx это – reg64, результат в RDX:RAX =
               ;0000000000000001FFFE0001FFFE0000
```

IMUL (Integer MULtiply) **-** Умножение целочисленное со знаком.
Назначение: операция умножения двух целочисленных двоичных значений со знаком.
Алгоритм работы:

Алгоритм работы команды зависит от используемой структуры команды.

Структура команды с одним операндом:

```
IMUL reg8/mem8      ; reg8/mem8 * AL =   AX
IMUL reg16/mem16    ; reg16/mem16 * AX= DX:AX
IMUL reg32/mem32    ; reg32/mem 32 * EAX  = EDX:EAX
```

Сокращения **reg/mem** обозначают тип операнда ***reg (register)*** - регистр микропроцессора, ***mem (memory)*** - ячейка памяти, а числа 8, 16, 32 указывают разрядность регистра или ячейки памяти.

Пример: **reg8-** указывает 8-битный регистр**, mem32 –** 32-битную ячейку памяти**, imm16 –** 16- битное значение, число.

Структура команды с двумя операндами. Структура с двумя операндами округляет результат до разрядности регистра приемника. Если теряются старшие биты то устанавливаются флаги CF и OF.

```
IMUL reg16,reg/mem16
IMUL reg16,imm8
IMUL reg16,imm16

IMUL reg32,reg/mem32
IMUL reg32,imm8
IMUL reg32,imm32
```

Структура команды с тремя операндами - op1=op2*op3 (округляет результат):

```
IMUL reg16,reg/mem16,imm8
IMUL reg16,reg/mem16,imm16
IMUL reg32,reg/mem32,imm8
IMUL reg32,reg/mem32,imm32
```

Пример с двумя операндами*:*

```
        .data
word1 SWORD 4
dword1 SDWORD 4
.code
mov ax,-16      ; AX = -16
mov bx,2                ; BX = 2
imul bx,ax      ; BX = -32
imul bx,2       ; BX = -64
imul bx,word1 ; BX = -256
```

```
mov eax,-16    ; EAX = -16
mov ebx,2      ; EBX = 2
imul ebx,eax ; EBX = -32
imul ebx,2     ; EBX = -64
imul ebx,dword1 ; EBX = -256
```

Пример с тремя операндами*:*

```
.data
word1 SWORD 4
dword1 SDWORD 4
.code
imul bx,word1,-16              ; BX = word1 * -16
imul ebx,dword1,-16            ; EBX = dword1 * -16
imul ebx,dword1,-2000000000 ; signed overflow!
```

DIV (DIVide unsigned) **- Деление беззнаковое**

Схема команды: **div делитель**

Назначение: выполнение операции деления двух двоичных беззнаковых значений.

Команда DIV служит для деления на 8-, 16-, 32-, 64- разрядное беззнаковое целое число, находящееся в одном из регистров общего назначения или в памяти операнда, расположенного в регистрах AX, **в паре регистров** DX:AX, EDX:EAX или RAX:RDX.

```
DIV r8/m8
DIV r16/m16
DIV r32/m32
DIV r64/m64
```

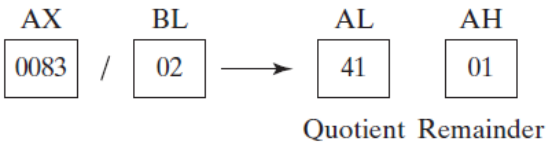
Команда DIV имеет всего один операнд, являющийся делителем. В следующей таблице указано, в каких регистрах размещается делимое, делитель, результат (частное) и остаток в зависимости от размера множителя.

Делимое	Делитель	Результат (частное)	Остаток
AX	r/m8	AL	AH
DX:AX	r/m16	AX	DX
EDX:EAX	r/m32	EAX	EDX
RDX:RAX	r/m64	RAX	RDX

Если делитель размером в байт (8 бит), то делимое должно быть расположено в регистре **AX**. После операции результат помещается в **al**, а остаток — в **ah**:

- результат [ax]/ [делитель] ->[al]
- остаток [ax]/ [делитель] ->[ah]

```
mov ax,0083h          ; делимое
mov bl,2              ; делитель
div bl  ; AL = 41h-частное, AH = 01h-остаток
```

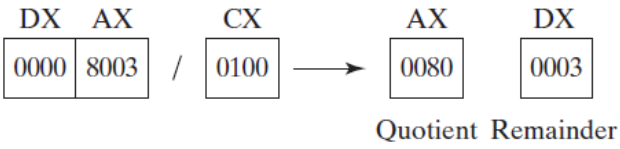


Если делитель размером в слово (16 бит), то делимое должно быть расположено **в паре регистров DX:AX**, причем младшая часть делимого находится в **ax**.

После операции результат деления (частное) помещается в **ax**, а остаток — в **dx**;

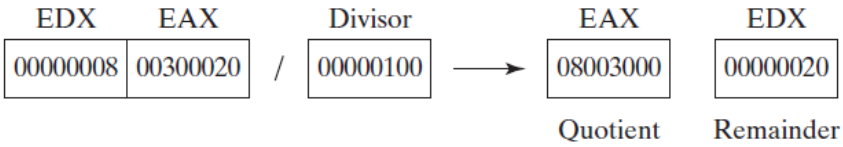
- результат [dx:ax]/[делитель] -> [ax]
- остаток [dx:ax]/[делитель] -> [dx]

```
mov dx,0 ; clear делимое, high
mov ax,8003h ; делимое, low
mov cx,100h ; делитель
div cx ; AX = 0080h-частное, DX = 0003h- остаток
```



Если делитель имеет разрядность 32 бита, тогда делимое находится в паре регистров EDX:EAX (64 бита), результат (частное) загружается в EAX а остаток в EDX.

```
.data
dividend QWORD 0000000800300020h
divisor DWORD 00000100h
.code
mov edx,DWORD PTR dividend + 4 ; high doubleword
mov eax,DWORD PTR dividend ; low doubleword
div divisor ; EAX = 08003000h, EDX = 00000020h
```



Если делитель имеет разрядность 64 бита, тогда делимое находится в паре регистров RDX и RAX (128 бита), результат (частное) загружается в RAX а остаток в RDX.

```
.data
dividend_hi QWORD 0000000000000108h
dividend_lo QWORD 0000000033300020h
divisor QWORD 0000000000010000h
.code
mov rdx,dividend_hi
mov rax,dividend_lo
div divisor      ; RAX = 0108000000003330
                 ; RDX = 0000000000000020
```

При выполнении операции деления возможно возникновение исключительной ситуации: 0 — ошибка деления. Эта ситуация возникает в одном из двух случаев: делитель равен 0 или результат (частное) слишком велик для его размещения в регистре al/ax/eax/rax.

IDIV (Integer DIVide) **- Деление целочисленное со знаком**

Схема команды: **idiv делитель**

Назначение: операция деления двух двоичных значений со знаком.

Команда IDIV позволяет выполнить деление целых чисел со знаком. Она имеет те же форматы операнда, что и команда DIV (**смотри описание команды DIV ↑**). **При делении на 8-, 16, 32-, 64- разрядное число**, **перед выполнением команды IDIV нужно расширить знак делимого с помощью одной из команд *CBW, CWD, CDQ*** в зависимости от разрядности делителя.

В приведенном ниже примере выполняется деление числа -48 на 5. После выполнения команды IDIV в регистре AX будет находиться результат, равный -9, а в регистре DX -остаток, равный -3:

```
   ;деление слов
mov    ax,-48 ;делимое
mov    bx,5  ;делитель
cwd           ;расширение делимого dx:ax
idiv   bx     ;результат в ax, остаток в dx
```

Остаток всегда имеет знак делимого. Знак результата (частного) зависит от состояния знаковых битов (старших разрядов) делимого и делителя. При выполнении операции деления возможно возникновение исключительной ситуации: 0 — ошибка деления. Эта ситуация возникает в одном из двух случаев: делитель равен 0 или частное слишком велико для его размещения в регистре rax/eax/ax/al.

2.7.3 Арифметика BCD чисел

AAA (Ascii Adjust after Addition) **-** ASCII-коррекция после сложения

Схема команды: **aaa**

Назначение: корректировка неупакованного результата сложения двух одноразрядных неупакованных BCD-чисел.

Алгоритм работы:

проанализировать значение младшего полубайта регистра al и значение флага af;

если (значение младшего полубайта регистра al >9) или (AF=1), то выполнить следующие действия:

- увеличить значение al на 6;
- очистить старший полубайт регистра al;
- увеличить значение ah на 1;
- установить флаги: af = 1, cf = 1,
- иначе сбросить флаги af = 0 и cf = 0.

Состояние флагов после выполнения команды:

OF	SF	ZF	AF	PF	CF
?	?	?	r	?	r

Применение:

Обычно команда aaa используется после сложения каждого разряда распакованных BCD-чисел командой add. Каждая цифра неупакованного BCD-числа занимает младший полубайт байта. Если результат сложения двух одноразрядных BCD-чисел больше 9, то число в младшем полубайте результата не есть BCD-число. Поэтому результат нужно корректировать командой aaa. Эта команда позволяет сформировать правильное BCD-число в младшем полубайте и запомнить единицу переноса в старший разряд путем увеличения содержимого регистра ah на 1.

К примеру, сложить два неупакованных BCD-числа: 48 + 29:

```
mov ax, 408h
mov dx, 209h
add  ax, dx     ; [ax]=0611h не неупакованное BCD-число
AAA       ;    [ax]=0707h— результат скорректирован
```

AAS (Ascii Adjust after Substraction) **-** ASCII-коррекция после вычитания

Схема команды: **aas**

Назначение: корректировка результата вычитания двух неупакованных одноразрядных BCD-чисел.

Алгоритм работы:

если $[AL_{0:3}] > 9$ или $[AF] = 1$, тогда {

[AL] ☐ [AL] - 6

[AH] ☐ [AH] - 1

[AF] ☐1

[CF] ☐1

[AL] ☐ [AL] AND 0FH

}

Состояние флагов после выполнения команды:

OF	SF	ZF	AF	PF	CF
?	?	?	r	?	r

Пример. Вычесть десятичные числа 29 из 48.

```
mov ax, 408h
mov dx, 209h
sub  ax, dx     ; [ax]=01ffh
AAS       ;    [ax]=0109h— результат скорректирован
```

DAS (Decimal Adjust for Subtraction) **-** Десятичная коррекция после вычитания

Схема команды: **das**

Назначение: коррекция упакованного результата вычитания двух BCD-чисел в упакованном формате.

Алгоритм работы: команда das работает только с регистром al.

если $[AL_{0:3}]$ > 9 или [AF] = 1, тогда {

[AL] ☐ [AL] - 6

[AF] ☐ 1

}

если $[AL_{4:7}]$ > 9 или CF = 1, тогда {

[AL] ☐ [AL] - 60H

[CF] ☐ 1

}

Пример.

```
MOV    AL,  52H
SUB    AL,  24H      ;  AL = 2EH— не BCD-число
DAS              ;  AL = 28H— результат скорректирован
```

AAM (Ascii Adjust after Multiply) **-** ASCII-коррекция после умножения

Схема команды: **aam**

Назначение:

- ☐ корректировка результата умножения двух неупакованных BCD-чисел;
- ☐ преобразование двоичного числа меньшего 63h (99_{10}) в его неупакованный BCD-эквивалент.

Алгоритм работы:

[AH] ☐ [AL] / 10

[AL] ☐ [AL] MOD 10

Состояние флагов после выполнения команды:

OF	SF	ZF	AF	PF	CF
?	r	r	r	r	?

Пример 1. Умножить десятичные числа 8 на 9.

```
mov     ah,08h  ;ah=08h
mov     al,09h  ;al= 09h
mul     ah      ;al=48h — не неупакованное
aam             ;ah=07h,al=02h— результат скорректирован
```

Пример 2. Преобразовать двоичное число 60h в эквивалентное десятичное число.

```
;поместим число 60h в регистр ax
  mov     ax,60h  ;ax=60h
```

```
aam          ;ax=0906h — получили десятичный эквивалент(96) числа 60h
```

AAD (Ascii Adjust before Division) **-** ASCII-коррекция перед делением.

Схема команды: **aad**

Назначение:

- подготовка двух неупакованных BCD-чисел для операции деления.

Алгоритм работы:

[AL] ☐ [AH] * 10 + [AL]

[AH] ☐ 0

Состояние флагов после выполнения команды:

OF	SF	ZF	AF	PF	CF
?	r	r	r	r	?

Пример. Разделить десятичное число 35 на 2.

```
mov ax, 305h
mov bl, 2
AAD      ;  [ax]=35h
div  bl   ;  [al]=12h   [ah]=1
```

2.7.4 Примеры программ

Пример 1. В данном примере представлены несколько способов адресации применямых в микропроцессорах x86:

```
INCLUDE Irvine32.inc
.data
alfa    WORD    3 DUP(?)
.code
main proc
mov    ax,17  ; Прямая адресация -команда содержит данные
mov    ax,10101b
mov    ax,11b
mov    ax,0bch
mov    alfa,ax
mov    cx,ax          ; обмен двух значений между
mov    ax,bx          ; регистрами ax и bx
mov    ax,cx          ;
xchg   ax,bx          ; тот же эффект
mov   si,2
mov  alfa[si],ax  ; косвенная регистровая адресация
mov  esi,2
mov  ebx,offset alfa ; загрузка смещения в регистр ebx
lea  ebx,alfa                 ; тот же эффект
mov     ecx,[ebx][esi]      ; индексная адресация источника
mov     cx,alfa[2]            ; тот же эффект
mov     cx,[alfa+2]           ; тот же эффект
mov     di,4
mov     byte ptr [ebx][edi],55h  ; адресация на уровне байта
mov     esi,2
mov     ebx,3
mov     alfa[ebx][esi],33h  ; косвенная регистровая адресация
                            ; приемника
mov     alfa[ebx+esi],33h   ; тот же эффект
mov     [alfa+ebx+esi],33h ; тот же эффект
mov     [ebx][esi]+alfa,33h ; тот же эффект

exit
```

```
main ENDP
END main
```

Ex. 2. Требуется вычислить значение арифметического выражения:

e=((a+b*c-d)/f+g*h)/i

Где a, d, f – определены как слова (16 бит), a - b, c, g, h, i – как байты. Чтобы выполнить деление на **f** расширим делимое до двойного слова (32 бит). При делении, учитывать только частное. Результат будет разрядностью в 1 байт.

```
INCLUDE Irvine32.inc
.data
        a dw 5
        b db 6
        cd db 10
        d dw 5
        f dw 6
        g db 10
        h db 11
        i db 10
        interm dw ?
        rez db ?

.code
main proc
        mov eax,0
        mov al, b
        imul cd                 ; in ax avem b*c
        add ax, a               ; ax=b*c+a
        sub ax, d               ; ax=b*c+a-d
        cwd             ; преобразуем слово из ax, в двойное слово , dx:ax
        idiv f                  ; частное в ax  si остаток в dx, ax=(a+b*c-d)/f
        mov interm, ax          ; interm=(a+b*c-d)/f
        mov al, g
        imul h                  ; ax=g*h
        add ax, interm          ; ax=(a+b*c-d)/f+g*h
        idiv i                  ; частное в al и остаток в ah
        mov rez, al

        exit
        main ENDP
        END main
```

Файл Irvine32.inc это боблиотека с процедурами использующая функции Win32 API, разработанная для упрощения консольного ввода/ вывода. (API - Application Programming Interface, набор функций операционной системы Windows для управления ресурсами компьютера и находятся в трех библотеках: user32.dll, kernel32.dll şi gdi32.dll).

Окно консоли (окно при запуске cmd.exe), окно в MS-Windows, для работы с командной строкой.

Опишем некоторые процедуры используемые для ввода данных с клавиатуры и вывода данных на монитор. Эти процедцры вызываются командой *call,* к примеру, *call Clrscr.*

В вызываемых процедурах ипользуются различные операторы.

2.7.5 Операторы

PTR. Оператор **PTR** позволяет переопределить размер операнда, принятый по умолчанию.

Он используется только в том случае, когда размер объявленный в программе переменной, не совпадает с размером второго операнда команды (т.е. к программе производится доступ к части переменной).

Пример:

```
.data
alfa word 3 dup (0)
; здесь alfa определена как последовательность и 3-х слов (word)
.code
mov byte ptr alfa[3], 33h
; здесь alfa переопределяется (byte ptr alfa) как последовательность из байт (6)
```

Пример:

```
.data
myDouble DWORD 12345678h
.code
mov ax,WORD PTR myDouble ; 5678h
mov ax,WORD PTR [myDouble+2] ; 1234h
mov bl,BYTE PTR myDouble ; 78h


.data
wordList WORD 5678h,1234h
.code
mov eax,DWORD PTR wordList ; EAX = 12345678h
```

Операторы **equ** и "=" (равно) предназначены для присвоения некоторому выражению символического имени или идентификатора. Впоследствии, когда в ходе трансляции этот идентификатор встретится в теле программы, ассемблер подставит вместо него соответствующее выражение.

В качестве выражения могут быть использованы константы, имена меток, символические имена и строки в апострофах. После присвоения этим конструкциям символического имени его можно использовать везде, где требуется размещение данной конструкции.

Операторы equ и “=” отличаются следующим:

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

Операторы equ и “=” могут быть применятся со слудующими операторами:

- арифметические +, -, *, / (деление);
- сдвиговые - SHR и SHL;
- логические - NOT, AND, OR, XOR;
- TYPE возвращает число типа некоторого выражения;
- SIZEOF возвращает число байт определенной последовательности (массива);
- LENGTHOF возвращает число байт, слов (в зависимости от TYPE) определенной последовательности (массива);
- HIGH, LOW выбор старшего или младшего байта.

Примеры применения:

```
.DATA
  intgr =     14*3      ;=42
  intgr =     intgr/4   ;10
  intgr =     intgr+4    ;14
  intgr =     intgr-3   ;11
  m1   EQU     5
```

```
m2   EQU      -5
const  EQU     m1+m2          ;const=0
vect   DW      60  DUP(?)
s_vect  EQU     SIZEOF vect     ;60 * 2 = 120
l_vect  EQU     LENGTHOF vect   ;60
t_vect EQU     TYPE vect     ;2
verif  EQU     t_vect*l_vect  ;=2*60
.code
  mov ax, SIZEOF vect
```

В общем, операторы сохраняют в отдельных файлах, а вводят их в программы при помощи директивы INCLUDE.

2.7.6 *Процедуры библиотеки Irvine32.lib*

Процедура	**Описание**
Clrscr	Очищает экран терминала и помещает курсор в левый верхний угол экрана
Crlf	Выводит на стандартное устройство вывода последовательность символов, соответствующую концу строки
Delay	Задерживает выполнение программы на указанный в качестве параметра интервал времени п, заданный в миллисекундах
DumpMem	Отображает содержимое блока памяти в шестнадцатеричном формате на стандартном устройстве вывода
DumpRegs	Отображает содержимое регистров EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP, EFLAGS и EIP в шестнадцатеричном формате на стандартном устройстве вывода. Дополнительно отображается также состояние флагов переноса (CF), знака (SF), нуля (ZF) и переполнения (OF)
Gotoxy	Помещает курсор в указанную позицию (строка, столбец) терминала
Random32	Генерирует псевдослучайное целое число в диапазоне от 0 до FFFFFFFFh
Randomize	Устанавливает уникальное начальное значение генератора случайных чисел
RandomRange	Генерирует псевдослучайное целое число в указанном диапазоне
ReadChar	Читает один символ из стандартного устройства ввода
ReadDec	Читает 32-разрядное десятичное целое число из стандартного устройства ввода. Признаком конца ввода служит нажатие клавиши <Enter>
ReadHex	Читает 32-разрядное шестнадцатеричное целое число из стандартного устройства ввода. Признаком конца ввода служит нажатие клавиши <Enter>
ReadInt	Читает 32-разрядное целое число со знаком, представленное в десятичном формате, из стандартного устройства ввода. Признаком конца ввода служит нажатие клавиши < Enter>
ReadString	Читает строку символов из стандартного устройства ввода. Признаком конца ввода служит нажатие клавиши < Enter>
SetTextColor	Устанавливает цвет символов и цвет фона для всех последуюших процедур вывода текста на терминал. Не поддерживается в библиотеке Irvine16. lib
WaitMsg	Отображает сообщение на терминале и переводит программу в режим ожидания нажатия клавиши < Enter>
WriteBin	Выводит 32-разрядное беззнаковое целое число в двоичном ASCII-формате на стандартное устройство вывода
WriteBinB	Выводит 8/16/32-разрядное беззнаковое целое число в двоичном ASCII-

	формате на стандартное устройство вывода
WriteChar	Выводит один символ на стандартное устройство вывода
WriteDec	Выводит 32-разрядное беззнаковое целое число в десятичном формате на стандартное устройство вывода
WriteHex	Выводит 32-разрядное беззнаковое целое число в шестнадцатеричном формате на стандартное устройство вывода
WriteHexB,W,D	Выводит 8/16/32-разрядное беззнаковое целое число в шестнадцатеричном формате на стандартное устройство вывода
WriteInt	Выводит 32-разрядное знаковое целое число в десятичном формате на стандартное устройство вывода
WriteString	Выводит нуль-завершенную текстовую строку на стандартное устройство вывода
WriteWindowsMsg	Выводит последовательнсть из последних ошибок сгенерированных MS-Windows

Подробное описание процедур

ClrScr. Эта процедура очищает экран терминала. Обычно подобная операция осуществляется в начале и в конце выполнения программы. Если вы планируете вызывать эту процедуру в процессе выполнения программы, не забудьте поместить перед ней вызов процедуры Wai tMsg, чтобы приостановить выполнение программы. В результате
пользователь сможет прочитать то, что было выведено на экран ранее перед тем, как программа сотрет содержимое экрана. Пример вызова процедуры: **call ClrScr**

CrLf. Вызов этой процедуры перемещает курсор на экране монитора в первую позицию следующей строки. При этом на стандартное устройство вывода посылается двухбайтовая последовательность символов ODh и OAh (т.е. символы возврата каретки и перевода строки). Пример вызова процедуры: **call CrLf**

Delay. Эта процедура приостанавливает выполнение программы пользователя на указанный временной интервал. При вызове процедуры Delay необходимо в регистр EAX поместить значение интервала времени в миллисекундах, например:

```
mov eax,1000
call Delay
```

DumpMem. Эта процедура выводит на стандартное устройство вывода содержимое указанного участка памяти в шестнадцатеричном формате. При вызове процедуры в регистре ESI нужно указать адрес участка памяти, в регистре ECX - количество блоков памяти, а в регистре EBX - код формата выводимых значений (1 = байт, 2 = слово, 4 = двойное слово). Например, в приведенном ниже фрагменте кода выводится содержимое массива *array* (11 двойных слов):

```
.data
array DWORD 1, 2, 3, 4, 5, 6, 7, 8, 9, OAh, OBh
.code
main PROC
 mov esi,OFFSET array ; Адрес участка памяти
 mov ecx,LENGTHOF array ; Длина участка в блоках
 mov ebx,TYPE array ; Размер блока (двойное слово)
 call DumpMem
```

В результате выполнения этой программы на экране появится следующая последовательность двойных слов:

```
00000001  00000002  00000003  00000004  00000005  00000006
00000007  00000008 00000009 0000000A 0000000B
```

DumpRegs. Эта процедура отображает содержимое регистров общего назначения EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP, EIP и EFL (EFLAGS) в шестнадцатеричном формате. Кроме того, на экран выводится также состояние флагов переноса (CF), знака (SF), нуля (ZF) и переполнения (OF). Ниже приведен пример вывода этой процедуры:

```
call DumpRegs

EAX=00000613 EBX=00000000 ECX=000000FF EDX=00000000
ESI=00000000 EDI=00000100 EBP=0000091E ESP=000000F6
EIP=00401026 EFL=00000286 CF=0 SF=1 ZF=0 OF=0 AF=0 PF=1
```

Содержимое регистра EIP соответствует смещению команды в сегменте кода, которая расположена сразу после команды вызова процедуры DumpRegs. Эта процедура позволяет довольно эффективно проводить отладку кода, поскольку она выводит полную информацию о состоянии процессора в нужной точке пользовательской программы.

В процедуре DumpRegs не предусмотрено никаких входных параметров, она также не возвращает никаких значений.

GotoXY. Эта процедура помещает курсор в указанную позицию на экране. По умолчанию значение горизонтальной координаты положения курсора может находиться в диапазоне 0-79, а вертикальной - 0-24. При вызове процедуры GotoXY в регистре он должно находиться значение вертикальной координаты (т.е. номер строки, начиная с 0), а в регистре DL - значение горизонтальной координаты (т.е. номер столбца, начиная с 0):

```
mov dh,10 ; номер строки 10
mov dl,20 ; номер столбца 20
call Gotoxy ; Переместить курсор
```

Random32. Эта процедура генерирует псевдослучайное целое число в диапазоне от 0 до FFFFFFFFh, которое возвращается в регистре EAX. Чтобы сгенерировать последовательность ***псевдослучайных чисел,*** нужно вызвать процедуру Random32 необходимое количество раз Случайные числа генерируются с помощью простой функции, на вход которой подается ***начальное значение*** генератора случайных чисел. Оно используется в формуле для генерации первой псевдослучайной величины. Все последующие значения последовательности псевдослучайных чисел генерируются на основе предыдущих значений. Далее под термином ***случайные*** числа мы будем подразумевать последовательность сгенерированных компьютером псевдослучайных чисел. Вот пример:

```
.data
randVal DWORD ?
.code
call Random32
mov randVal,eax
```

Randomize. Эта процедура задает начальное значение генератора случайных чисел для формул, которые используются в процедурах Random32 и RandomRange. При вызове процедуры Randomi ze в качестве начального значения генератора используется текущее время, округленное до 1/100 с. Это позволяет гарантировать, что при каждом запуске программы, начальное значение генератора случайных чисел будет разным и, следовательно, сгенерированная последовательность случайных чисел тоже будет разной.

Процедуру Randomize достаточно запустить только один раз в начале выполнения программы. В приведенном ниже примере генерируется последовательность из 10 случайных чисел:

```
Call Randomize
```

```
mov ecx,10
l1: Call Random32
        ;Здесь в регистре EAX находится случайное число.
        ; Его нужно сохранить в переменной (массиве),
        ;либо отобразить на экране
loop l1
```

RandomRange. Эта процедура генерирует случайное целое число в указанном диапазоне значений от 0 до *n* - 1. В качестве параметра этой процедуре передается в регистре EAX число *n.* Сгенерированное случайное число возвращается также в регистре EAX.

Например, в приведенном ниже фрагменте кода генерируется случайное число в диапазоне 0 … 4999, которое записывается в переменную randVal:

```
.data
randVal DWORD ?
.code
mov eax,5000
call RandomRange
mov randVal,eax
```

ReadChar. Эта процедура позволяет прочитать один символ со стандартного устройства ввода и поместить его в регистр AL. При этом эхо вводимого символа не отображается на экране. Ниже приведен пример вызова этой процедуры:

```
.data
char BYTE ?
.code
call ReadChar
mov char,al
```

ReadDec. Эта процедура позволяет прочитать 32-разрядное десятичное целое число, без знака, из стандартного устройства ввода и поместить его в регистр EAX. Пробелы игнорируются, вводятся только десятичные значения. Пример, если ввести 123ABC, сохраненное значение в EAX будет 123.

```
.data
intVal DWORD ?
.code
call ReadDec
mov intVal,eax
```

ReadHex. Эта процедура позволяет прочитать 32-разрядное шестнадцатеричное целое число из стандартного устройства ввода и поместить его в регистр EAX. В процессе работы этой процедуры не проверяется корректность вводимых символов. При вводе шестнадцатеричных цифр от A до F можно пользоваться символами как верхнего, так и нижнего регистров. Допускается ввод до восьми шестнадцатеричных цифр. При этом нельзя вводить начальные пробелы. Вот пример:

```
.data
hexVal DWORD ?
.code
call ReadHex
mov hexVal,eax
```

Readint. Эта процедура позволяет прочитать 32-разрядное целое число со знаком, представленное в десятичном формате, из стандартного устройства ввода и поместить его в регистр EAX. Сразу при вводе можно пользоваться начальными пробелами, а также символами " + " и " - ", однако после них должны вводиться только числа. Если введенное пользователем

значение не может быть преобразовано к 32-разрядному двоичному целому числу со знаком (т.е. выходит за границу диапазона значений -2 147 483 648 ... +2 147 483 647), процедура Readint выводит сообщение об ошибке и на выходе устанавливает флаг переполнения OF. Ниже приведен пример фрагмента кода, в котором используется эта процедура:

```
.data
intVal SDWORD ?
.code
call ReadInt
mov intVal,eax
```

ReadString. Эта процедура позволяет прочитать строку символов из стандартного устройства ввода. Чтение символов выполняется до тех пор, пока пользователь не нажмет клавишу <Enter>. В регистре EAX эта процедура возвращает количество байтов, которые были прочитаны. Перед вызовом процедуры ReadString в регистр EDX необходимо загрузить адрес массива байтов, в который будут записываться введенные пользователем символы. В регистр ECX нужно загрузить длину этого массива (т.е. максимальное количество символов, которые могут быть введены пользователем).

В приведенном ниже фрагменте кода перед вызовом функции ReadString загружаются соответствующие значения в регистры ECX и EDX. Обратите внимание, что в регистр ECX загружается мина массива минус один байт. Этот байт резервируется мя размещения признака конца строки (завершающего нуля):

```
.data
buffer BYTE 21 DUP(0) ; input buffer
byteCount DWORD ? ; holds counter
.code
mov edx,OFFSET buffer ; point to the buffer
mov ecx,SIZEOF buffer ; specify max characters
call ReadString ; input the string
mov byteCount,eax ; number of characters
```

Процедура ReadString автоматически помещает в конце введенной строки нулевой байт, который служит признаком ее завершения. Ниже приведен дамп первых восьми байтов массива buffer в шестнадцатеричном и ASCII-формате после того, как пользователь ввел строку символов ABCDEFG:

```
41 42 43 44 45 46 47 00          ABCDEFG
```

При этом значение переменной ByteCount будет равно 7.

SetTextColor. Эта процедура позволяет установить цвет символов и цвет фона для всех последующих процедур вывода текста на терминал. В следующей таблице перечислены константы и их значения, которые можно использовать для обозначения цветов символов и фона.

black = 0	red = 4	gray = 8	lightRed = 12
blue = 1	magenta = 5	lightBlue = 9	lightMagenta = 13
green = 2	brown = 6	lightGreen = 10	yellow = 14
cyan = 3	lightGray = 7	lightCyan = 11	white = 15

Определения этих констант сделаны в файле Irvine32.inc. При указании цвета фона исходную константу нужно умножить на 16 и к полученному значению прибавить цвет символа. Например, приведенная ниже константа позволяет отобразить желтые символы на голубом фоне:

```
yellow + (blue * 16)
```

Перед вызовом процедуры **SetTextColor** нужно поместить константу, определяющую цвета символов и фона в регистр EAX:

```
mov eax,white _ (blue * 16) ; white on blue
call SetTextColor
```

WaitMsg. Эта процедура отображает на экране стандартное сообщение “Press any key to continue. . .” и переводит программу в режим ожидания до тех пор, пока пользователь не нажмет клавишу. Эта процедура обычно используется для приостановки выполнения программы, чтобы пользователь смог прочитать выведенную на экран информацию. Эта процедура не имеет входных параметров. Вот пример вызова:

```
call WaitMsg
```

WriteBin. Эта процедура позволяет вывести на стандартное устройство вывода 32-разрядное беззнаковое целое число в двоичном ASCll-формате, находящееся в регистре EAX. Для облегчения чтения числа, значения двоичных битов отображаются группами по 4 символа в каждой:

```
mov eax,12346AF9h
call WriteBin
; Отображается: "0001 0010 0011 0100 0110 1010 1111 1001"
```

WriteChar. Эта процедура выводит один символ на стандартное устройство вывода. Перед ее вызовом нужно поместить в регистр AL ASCII-код этого символа:

```
mov al,'A'
call WriteChar ; displays: "A"
```

WriteDec. Данная процедура выводит 32-разрядное беззнаковое целое число в десятичном формате на стандартное устройство вывода, удаляя при этом незначащие нули.

Перед вызовом процедуры поместите отображаемое значение в регистр EAX:

```
mov eax,295
call WriteDec ; displays: "295"
```

WriteHex. Эта процедура выводит 32-разрядное беззнаковое целое число в восьмизначном шестнадцатеричном формате на стандартное устройство вывода. При необходимости она автоматически добавляет незначащие нули, чтобы отображаемое шестнадцатеричное число приобрело привычный вид. Перед вызовом процедуры поместите отображаемое значение в регистр EAX:

```
mov eax,7FFFh
call WriteHex ; displays: "00007FFF"
```

WriteInt. Эта процедура выводит 32-разрядное знаковое целое число в десятичном формате на стандартное устройство вывода. Перед числом автоматически добавляется символ знака (" + " или " - ") и удаляются незначащие нули. Перед вызовом процедуры поместите отображаемое значение в регистр EAX:

```
mov eax,216543
call WriteInt ; displays: "+216543"
```

WriteString. Эта процедура выводит нуль-завершенную текстовую строку на стандартное устройство вывода. При ее вызове нужно поместить в регистр EDX адрес этой строки, например:

```
.data
prompt BYTE "Enter your name: ",0

.code
mov edx,OFFSET prompt
call WriteString
```

WriteWindowsMsg. Эта процедура выводит сообщения с последними ошибками сгенерированных MS-Windows, при вызове системных функций, на стандартное устройство вывода:

```
call WriteWindowsMsg
```

Пример сообщения:

Error 2: The system cannot find the file specified.

2.7.7 Команды переходов, управления циклом

JMP (JuMP)- **Переход безусловный**

Команда JMP вызывает безусловную передачу управления на новый участок программы, находяшийся в пределах сегмента кода. В исходном коде программы такой участок помечается меткой, которая заменяется при трансляции соответствуюшим адресом.

Синтаксис команды: **jmp *метка_перехода***

Назначение: используется в программе для организации безусловного перехода как внутри текущего сегмента команд, так и за его пределы.

Переход может быть:

- прямым коротким ***(short)*** (в пределах -128... + 127 байтов);
- прямым ближним (***near)*** (в пределах текущего сегмента команд);
- прямым дальним ***(far)*** (в другой сегмент команд);
- косвенным ближним (в пределах текущего сегмента команд через ячейку с адресом перехода).

Команды условных переходов.

Микропроцессор имеет 18 команд условного перехода. Эти команды позволяют проверить:

- отношение между операндами со знаком (“больше — меньше”);
- отношение между операндами без знака (“выше — ниже”);
- состояния арифметических флагов zf, sf, cf, of, pf (но не af).

Команды условного перехода имеют одинаковый синтаксис:

jcc *метка_перехода*

Как видно, мнемокод всех команд начинается с “**j**” — от слова ***jump*** (прыжок), ***cc*** — определяет конкретное условие, анализируемое командой.

Что касается операнда *метка_перехода*, то эта метка может находится только в пределах текущего сегмента кода, ***межсегментная передача управления в условных переходах не допускается***.

Таблица 2.1 - Значение аббревиатур в названии команды jcc

Мнемоническое обозначение	Английский	Русский	Тип операндов
E e	equal	Равно	Любые

N n	not	Не	Любые
G g	greater	Больше	Числа со знаком
L l	less	Меньше	Числа со знаком
A a	above	Выше, в смысле “больше”	Числа без знака
B b	below	Ниже, в смысле “меньше”	Числа без знака

Таблица 2.2 - Перечень команд условного перехода для команды cmp

Типы операндов	Мнемокод команды условного перехода	Критерий условного перехода	Значения флагов для осуществления перехода
Любые	je/jz	операнд_1 = операнд_2	zf = 1
Любые	jne/jnz	операнд_1<>операнд_2	zf = 0
Со знаком	jl/jnge	операнд_1 < операнд_2	sf <> of
Со знаком	jle/jng	операнд_1 <= операнд_2	sf <> of or zf = 1
Со знаком	jg/jnle	операнд_1 > операнд_2	sf = of and zf = 0
Со знаком	jge/jnl	операнд_1 => операнд_2	sf = of
Без знака	jb/jnae	операнд_1 < операнд_2	cf = 1
Без знака	jbe/jna	операнд_1 <= операнд_2	cf = 1 or zf=1
Без знака	ja/jnbe	операнд_1 > операнд_2	cf = 0 and zf = 0
Без знака	jae/jnb	операнд_1 => операнд_2	cf = 0

Можно заметить, что одинаковым значениям флагов соответствует несколько разных мнемокодов команд условного перехода (они отделены друг от друга косой чертой в табл. 2.2).

Разница в названии обусловлена желанием разработчиков микропроцессора облегчить использование команд условного перехода в сочетании с определенными группами команд. Поэтому разные названия отражают скорее различную функциональную направленность. Тем не менее, то, что эти команды реагируют на одни и те же флаги делает их эквивалентными и равноправными в программе. Поэтому в табл. 2.2 они сгруппированы не по названиям, а по значениям флагов (условиям), на которые они реагируют.

Примеры:
```
mov ax,0FFFEh
mov bx, 2
cmp ax, bx
ja alfa; jump is taken
```

и

```
mov ax, 0FFFEh
mov bx, 2
cmp ax, bx
jg alfa; jump not taken
```

В первом примере переход на метку *alfa* произойдет, во втором нет.

Команда **JCXZ** (Jump if CX=Zero)
Переход, если CX равен нулю

Синтаксис команды:
JCXZ *метка_перехода*
JECXZ *метка_перехода*
JRCXZ *метка_перехода*

Назначение: переход внутри текущего сегмента команд (от -128 до 127) в зависимости от некоторого условия.

Алгоритм работы команды jcxz:

Проверка условия равенства нулю содержимого регистра *cx/ecx/rcx*:

- если проверяемое условие истинно, то есть содержимое *cx/ecx/rcx* равно 0, то перейти к ячейке, обозначенной операндом метка;
- если проверяемое условие ложно, то есть содержимое *cx/ecx/rcx* не равно 0, то передать управление следующей за jxxz команде программы.

Применение:

Например, команду можно использовать для предварительной проверки счетчика цикла в регистре *cx/ecx/rcx* для обхода цикла, если его счетчик нулевой.

Примеры:

```
mov edx,0A523h
cmp edx,0A523h
jne L5 ; jump not taken
je L1 ; jump is taken

mov bx,1234h
sub bx,1234h
jne L5 ; jump not taken
je L1 ; jump is taken

mov cx,0FFFFh
inc cx
jcxz L2 ; jump is taken

xor ecx,ecx
jecxz L2 ; jump is taken
```

сравнение со знаком:

```
mov edx,-1
cmp edx,0
jnl L5 ; jump not taken (-1>=0 is false)
jnle L5 ; jump not taken (-1 > 0 is false)
jl L1 ; jump is taken (-1 < 0 is true)


mov bx,+32
cmp bx,-35
jng L5 ; jump not taken (+32 <= -35 is false)
jnge L5 ; jump not taken (+32 < -35 is false)
jge L1 ; jump is taken (+32 >= -35 is true)

mov ecx,0
cmp ecx,0
jg L5 ; jump not taken (0 > 0 is false)
jnl L1 ; jump is taken (0 >= 0 is true)

mov ecx,0
cmp ecx,0
jl L5 ; jump not taken (0 < 0 is false)
jng L1 ; jump is taken (0 <= 0 is true)
```

Команда **LOOP** (LOOP control by register ecx) **- Управление циклом по ecx.**

Команда **LOOP** позволяет выполнить некоторый блок команд заданное количество раз.

В качестве счетчика используется регистр **ECX**, значение которого автоматически уменьшается на единицу при каждом выполнении команды LOOP. Синтаксис этой команды следующий:

loop ***метка***

Назначение: **организация цикла со счетчиком в регистре ecx**.

Алгоритм работы:

- выполнить декремент содержимого регистра ecx;
- анализ регистра ecx:
- если ecx=0, передать управление следующей за loop команде;
- если ecx=1, передать управление команде, метка которой указана в качестве операнда loop.

Также могут быть применены команды с синтаксисами:

- Команда LOOPD использует регистр ECX как регистр счетчик;
- Команда LOOPW использует регистр CX как регистр счетчик.

Применение: Команду *loop* применяют для организации цикла со счетчиком. Количество повторений цикла задается значением в регистре e*cx* перед входом в последовательность команд, составляющих тело цикла. Помните о двух важных моментах:

- для предотвращения выполнения цикла при нулевом *cx* используйте команду jcxz. Если этого не сделать, то при изначально нулевом *cx* цикл повторится 65 536 раз;
- смещение метки, являющейся операндом loop, не должно выходить из диапазона -128...+127 байт. Это смещение, как и в командах условного перехода, является относительным от значения счетчика адреса следующей за loop команды.

Ex: Сума n байт с адреса *string*.

```
.data
string byte 7, 9, 15, 25, -18, 33, 11
n equ LENGTHOF string
suma byte ?
.code
     xor eax, eax
     mov ecx, n
     xor esi, esi
     m1:
     add al, string[esi]
     inc esi
     LOOP m1
     mov suma, al
```

Команды **LOOPE/LOOPZ и LOOPNE/LOOPNZ**

Команда **LOOPZ** (Loop if Zero, или цикл пока нуль) позволяет организовать цикл, который будет выполнятся, пока установлен флаг нуля ZF и значение регистра ECX, взятое без знака, больше нуля. Метка перехода, указанная в этой команде, должна находиться в пределах -128 ... +127 байтов относительно адреса следующей команды. Синтаксис команды LOOPZ следующий:

LOOPZ ***метка_перехода***

Команда **LOOPE** (Loop if Equal, или цикл пока равно) полностью аналогична команде LOOPZ, поскольку в ней учитывается значение того же флага состояния процессора. Ниже приведена логика работы команд LOOPZ и LOOPE:

```
ECX = ECX - 1
Если (ECX > 0 и ZF = 1), то перейти по метке
```

```
; Иначе управление переходит следующей команде.
```

Команда **LOOPNZ** (Loop if Not Zero, или цикл пока не нуль) по сути аналогична команде LOOPZ за одним небольшим исключением. Цикл на основе команды LOOPNZ будет выполняться, пока значение регистра ECX, взятое без знака, больше нуля и сброшен флаг нуля ZF. Синтаксис команды LOOPNZ следующий:

LOOPNZ ***метка_перехода***

Команда **LOOPNE** (Loop if Not Equal, или цикл пока не равно) полностью аналогична команде LOOPNZ, поскольку в ней учитывается значение того же флага состояния процессора. Ниже приведена логика работы команд LOOPNZ и LOOPNE:

```
ECX = ECX - 1
Если (ECX > 0 и ZF = 0) , то перейти по метке
; Иначе управление переходит следующей команде.
```

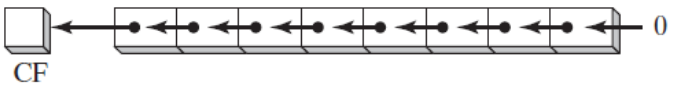
2.7.8 Команды линейного и циклического сдвига

К командам линейного сдвига относятся команды, осуществляющие сдвиг по следующему алгоритму:

- очередной “выдвигаемый” бит устанавливает флаг cf;
- бит, вводимый в операнд с другого конца, имеет значение 0;
- при сдвиге очередного бита он переходит во флаг cf, при этом значение предыдущего сдвинутого бита ***теряется!***

К командам линейного сдвига относятся следующие:

Команда **shl операнд,счетчик_сдвигов** (Shift Logical Left) - **логический сдвиг влево.** Содержимое операнда сдвигается влево на количество битов, определяемое значением счетчик_сдвигов. Счетчик_сдвигов может быть и регистр CL (5 младших разрядов). Справа (в позицию младшего бита) вписываются нули.



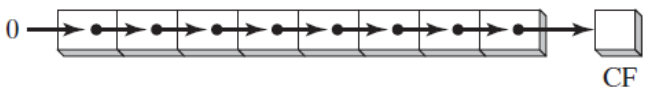
Ниже приведены допустимы форматы операндов команды SHL (тоже для SHR, SAL, SAR, ROR, ROL, RCR, RCL):

```
SHL reg,imm8
SHL mem,imm8
SHL reg,CL
SHL mem,CL
```

Ex.:

```
mov bl,8Fh ; BL = 10001111b
shl bl,1 ; CF = 1, BL = 00011110b

mov al,10000000b
shl al,2 ; CF = 0, AL = 00000000b

mov dl,10 ; before: 00001010
shl dl,2 ; after: 00101000
```

Команда **shr операнд,счетчик_сдвигов** (Shift Logical Right) — **логический сдвиг вправо.** Содержимое операнда сдвигается вправо на количество битов, определяемое значением счетчик_сдвигов. Слева (в позицию старшего, знакового бита) вписываются нули.

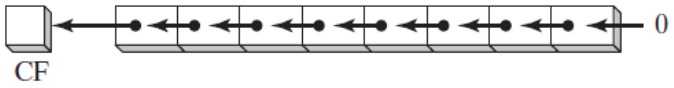


Ex.:

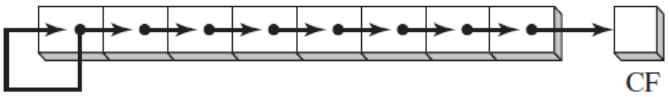
```
mov al,0D0h ; AL = 11010000b
shr al,1 ; AL = 01101000b, CF = 0


mov al,00000010b
shr al,2 ; AL = 00000000b, CF = 1
```

Команда **sal операнд,счетчик_сдвигов** (Shift Arithmetic Left) — **арифметический сдвиг влево**.

Содержимое операнда сдвигается влево на количество битов, определяемое значением *счетчик_сдвигов*. Справа (в позицию младшего бита) вписываются нули. Команда sal **не сохраняет знака**, но *устанавливает флаг cf в случае смены знака* очередным выдвигаемым битом. В остальном команда sal полностью аналогична команде shl.



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



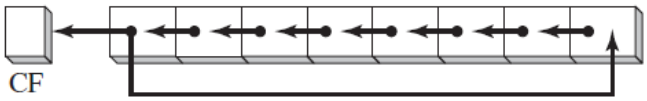
Ex.:

```
mov al,0F0h ; AL = 11110000b (-16)
sar al,1 ; AL = 11111000b (-8), CF = 0


mov dl,-128 ; DL = 10000000b
sar dl,3 ; DL = 11110000b

mov ax,-128 ; EAX = ????FF80h
shl eax,16 ; EAX = FF800000h
sar eax,16 ; EAX = FFFFFF80h
```

Команды циклического сдвига

К командам *циклического* сдвига относятся команды, сохраняющие значения сдвигаемых бит.

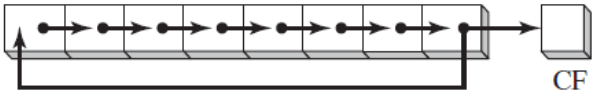
rol операнд,счетчик_сдвигов (Rotate Left) — **циклический сдвиг влево.** Содержимое операнда сдвигается влево на количество бит, определяемое операндом *счетчик_сдвигов*. Сдвигаемые влево биты записываются в тот же операнд справа.




```
mov al,40h ; AL = 01000000b
rol al,1 ; AL = 10000000b, CF = 0
rol al,1 ; AL = 00000001b, CF = 1
rol al,1 ; AL = 00000010b, CF = 0
```

ror операнд,счетчик_сдвигов (Rotate Right) — **циклический сдвиг вправо.**

Содержимое операнда сдвигается вправо на количество бит, определяемое операндом *счетчик_сдвигов* (0-255 или содержимое регистра CL). Сдвигаемые вправо биты записываются в тот же операнд слева.



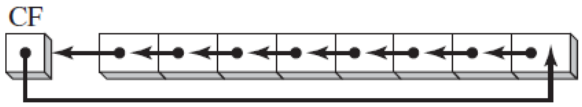

```
mov al,01h ; AL = 00000001b
ror al,1 ; AL = 10000000b, CF = 1
ror al,1 ; AL = 01000000b, CF = 0

mov al,00000100b
ror al,3 ; AL = 10000000b, CF = 1
```

Команды циклического сдвига в процессе своей работы осуществляют следующее: циклически сдвигаемый бит не только вдвигается в операнд с другого конца, но и одновременно его значение становиться значением флага cf.

Команды циклического сдвига *через флаг переноса cf* отличаются от команд простого циклического сдвига тем, что сдвигаемый бит не сразу попадает в операнд с другого его конца, а записывается сначала в флаг переноса cf. *Лишь следующее исполнение данной команды сдвига (при условии, что она выполняется в цикле) приводит к помещению выдвинутого ранее бита с другого конца операнда*.

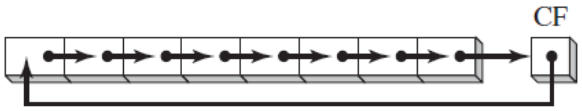
К командам циклического сдвига *через флаг переноса cf* относятся следующие:

rcl операнд,счетчик_сдвигов (Rotate through Carry Left) — **циклический сдвиг влево через перенос**. Содержимое операнда сдвигается влево на количество бит, определяемое операндом *счетчик_сдвигов* (0-255 или содержимое регистра CL). Сдвигаемые биты поочередно становятся значением флага переноса cf.




```
clc ; CF = 0
mov bl,88h ; CF,BL = 0 10001000b
rcl bl,1 ; CF,BL = 1 00010000b
rcl bl,1 ; CF,BL = 0 00100001b
```

rcr операнд,счетчик_сдвигов (Rotate through Carry Right) — **циклический сдвиг вправо через перенос**. Содержимое операнда сдвигается вправо на количество бит, определяемое операндом *счетчик_сдвигов*(0-255 или содержимое регистра CL). Сдвигаемые биты поочередно становятся значением флага переноса cf.




```
stc ; CF = 1
mov ah,10h ; AH, CF = 00010000 1
rcr ah,1 ; AH, CF = 10001000 0
```

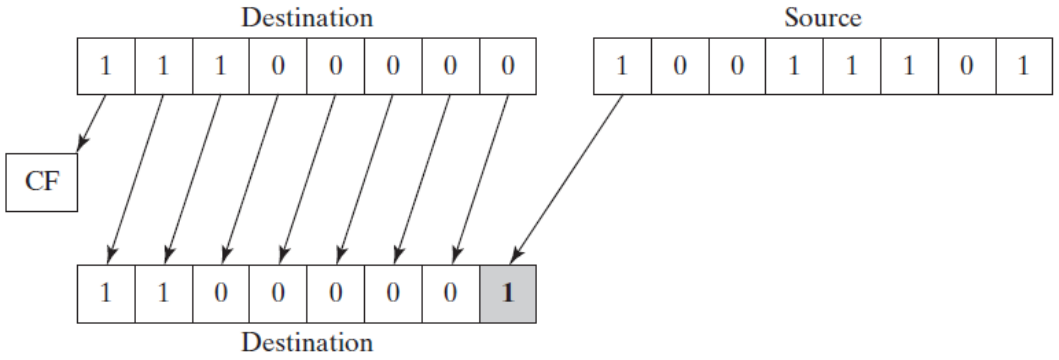
Команды **SHLD** и **SHRD**

Команды **SHLD** и **SHRD** имеют не два, а три операнда. Команда SHLD (SHift Left DouЬle, или сдвиг влево удвоенный) выполняет логический сдвиг влево операнда получателя данных на

количество разрядов, указанных в третьем операнде. Освободившиеся в результате сдвига разряды операнда получателя данных заполняются старшими битами исходного (т.е. второго) операнда. При этом значение исходного операнда не изменяется, но меняется состояние флагов знака SF, нуля ZF, служебного переноса AF, четности PF и переноса CF. Синтаксис команды SHLD следующий:

SHLD получатель, источник, счетчик

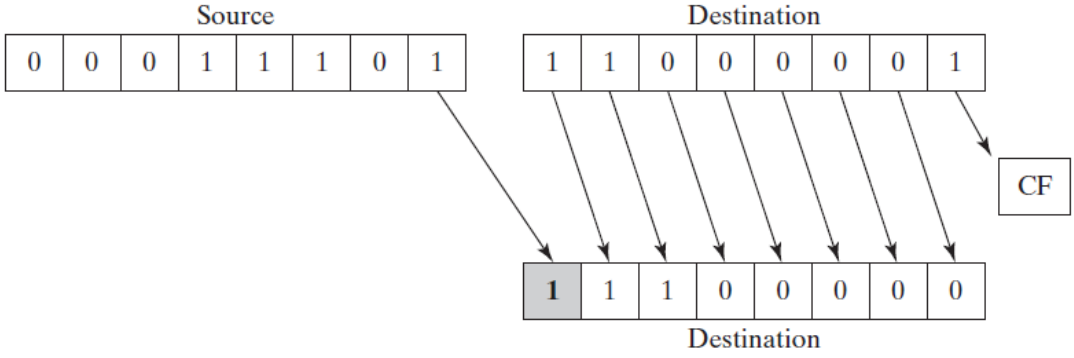
Сдвиг выполняется по следующей схеме:



Команда **SHRD** (SHift Right DouЬle, или сдвиг вправо удвоенный) выполняет логический сдвиг вправо операнда получателя данных на количество разрядов, указанных в третьем операнде. Освободившиеся в результате сдвига разряды операнда получателя данных заполняются младшими битами исходного (т.е. второго) операнда. Синтаксис команды SHLD следующий:

SHRD *получатель, источник, счетчик*

Сдвиг выполняется по следующей схеме:



Команды **SHLD** и **SHRD** имеют одинаковый формат операндов, описанный ниже. Операнд-получатель данных может располагаться либо в памяти, либо в регистре. Исходный операнд может находиться только в регистре. В качестве счетчика может быть задан либо регистр CL, либо 8-разрядная константа:

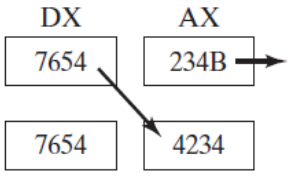
```
SHLD reg16,reg16,CL/imm8
SHLD mem16,reg16,CL/imm8
SHLD reg32,reg32,CL/imm8
SHLD mem32,reg32,CL/imm8
```

Ex.:

```
.data
wval WORD 9BA6h
.code
mov ax,0AC36h
```

```
shld wval,ax,4 ; wval = BA6Ah
```

```
mov ax,234Bh
mov dx,7654h
shrd ax,dx,4
```



2.7.9 Логические команды

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

and операнд1,операнд2 — операция логического умножения.

Команда выполняет поразрядно логическую операцию И (конъюнкцию) над битами операндов операнд1 и операнд2. Результат записывается на место операнд1.

Формат операндов:

```
AND reg,reg
AND reg,mem
AND reg,imm
AND mem,reg
AND mem,imm
```

Ex.

```
mov al,10101110b
and al,11110110b ; result in AL = 10100110
```

or операнд1,операнд2 — операция логического сложения.

Команда выполняет поразрядно логическую операцию ИЛИ (дизъюнкцию) над битами операндов операнд1 и операнд2. Результат записывается на место операнд1.

```
mov al,11100011b
or al,00000100b ; result in AL = 11100111
```

xor операнд1,операнд2 — операция логического исключающего сложения.

Команда выполняет поразрядно логическую операцию исключающего ИЛИ над битами операндов операнд_1 и операнд_2. Результат записывается на место операнд_1.

Таблица истинности:

x	y	x ⊕ y
0	0	0
0	1	1
1	0	1
1	1	0

test операнд1,операнд2 — операция “проверить” (способом логического умножения).

Команда выполняет поразрядно логическую операцию И над битами операндов операнд_1 и операнд_2. Состояние операндов остается прежним, изменяются только флаги zf, sf, и pf, что дает возможность анализировать состояние отдельных битов операнда без изменения их состояния.

```
test al,00100000b; test bit 5
```

not операнд — операция логического отрицания.

Команда выполняет поразрядное инвертирование (замену значения на обратное) каждого бита операнда. Результат записывается на место операнда.

```
mov al,11110000b
not al ; AL = 00001111b
```

Для представления роли логических команд в системе команд микропроцессора очень важно понять области их применения и типовые приемы их использования при программировании.

С помощью логических команд возможно *выделение* отдельных битов в операнде с целью их *установки, сброса, инвертирования или просто проверки на определенное значение*.

Для организации подобной работы с битами *операнд2* обычно играет роль **маски**. С помощью установленных в 1 битов этой маски и определяются нужные для конкретной операции биты операнд1. Покажем, какие логические команды могут применяться для этой цели:
Для установки определенных разрядов (бит) в 1 применяется команда :

or операнд1,операнд2

В этой команде операнд2, выполняющий роль маски, должен содержать единичные биты на месте тех разрядов, которые должны быть установлены в 1 в операнд1.

```
or     ax,10b ; установить 1-й бит в регистре ax
```

Для сброса определенных разрядов (бит) в 0 применяется команда:

and операнд1,операнд2

В этой команде операнд2, выполняющий роль маски, должен содержать нулевые биты на месте тех разрядов, которые должны быть установлены в 0 в операнд1.

```
and    ax,0fffdh   ;сбросить в 0 1-й бит в регистре eax
```

Команда **xor операнд1,операнд2** применяется:

- для выяснения того, какие биты в операнд_1 и операнд_2 различаются;
- для инвертирования состояния заданных бит в операнд_1.

```
xor     ax,10b ; инвертировать 1-й бит в регистре ax
  jz      mes  ; переход, если 1-й бит в al был единичным
```

Интересующие нас биты маски (операнд2) при выполнении команды xor должны быть единичными, остальные — нулевыми.

Для проверки состояния заданных бит применяется команда :

test операнд1,операнд2 (проверить операнд1)

Проверяемые биты операнд1 в маске (операнд2) должны иметь единичное значение. Алгоритм работы команды test подобен алгоритму команды *and*, но он не меняет значения операнд1. Результатом команды является установка значения флага нуля **zf**:

- если *zf* = 0, то в результате логического умножения получился нулевой результат, то есть один единичный бит маски, который не совпал с соответствующим единичным битом операнд1;
- если *zf* = 1, то в результате логического умножения получился ненулевой результат, то есть *хотя бы один* единичный бит маски совпал с соответствующим единичным битом операнд1.

```
test    ax,0010h
jz      m1 ; переход, если 4-й бит равен 1
```

Как видно из примера, для реакции на результат команды *test* целесообразно использовать команду перехода jnz *метка* (Jump if Not Zero) — переход, если флаг нуля zf ненулевой, или команду с обратным действием — jz *метка* (Jump if Zero) — переход, если флаг нуля zf = 0.

Команды для работы с отдельными битами

Команды **BT, BTC, BTR и BTS *предназначенны*** для работы с отдельными битами.

Команда **BT**

Команда **BT** (Bit Test, или тестирование бита) копирует бит первого операнда, номер *n* которого указан во втором операнде, во флаг переноса CF:

BT строка_битов, n

Формат операндов:

```
BT r/m16,imm8
BT r/m16,r16
BT r/m32,imm8
BT r/m32,r32
```

Команда **BTC** (Bit Test and Complement, или тестирование бита с инверсией) копирует бит первого операнда, номер *n* которого указан во втором операнде, во флаг переноса CF, а затем инвертирует значение этого бита.

Команда **BTR** (Bit Test and Reset, или тестирование бита со сбросом) копирует бит первого операнда, номер *n* которого указан во втором операнде, во флаг переноса CF, а затем сбрасывает (т.е. обнуляет) значение этого бита.

Команда **BTS** (Bit Test and Set, или тестирование бита с установкой) копирует бит первого операнда, номер *n* которого указан во втором операнде, во флаг переноса CF, а затем устанавливает в единицу значение этого бита.

В приведенном ниже примере во флаг переноса CF помещается значение 6-го бита переменной *perem*, а затем значение этого бита устанавливается в единицу:

Ex.

```
.data
perem WORD 10001000b

.code
bts perem,6   ;    CF=0  perem=11001000b
```

Пример. Следующая программа выполняет поиск первого пробела в последовательности символов **str**. По завершении программы, если нет пробелов в **ax – 0,** иначе значение порядкового номера первого пробела в последовательности. Обозначим long - количество символов в последовательности.

```
    .data
str      DB      'Poisc pervogo probela!'
long        EQU     sizeof str

.code
            mov     ecx,long
            mov    esi,-1
            mov     al,' '
next:        inc     esi
            cmp     al,str[esi]
            loopne  next
            jne     nol
```

```
            mov     eax,esi
            jmp     exit
nol: mov    eax,0
exit:  nop
```

2.7.10 Дополнительные команды

Примеры:

`CMC (CoMplement Carry flag);` изменение значения флага переноса `cf` на обратное

`CLC (CLear Carry flag);`установка в `0` значения флага переноса `CF`

`STC (SeT Carry flag);` установка в `1` значения флага `CF`

`NOP ;` нет операции, использует `3` такта.

`CLD (CLear Direction flag) ;` установка в `0` значения флага `DF`

`STD (SeT Direction flag);` установка в `1` значения флага `DF`

`CLI (CLear Interrupt flag);` установка в `0` значения флага `IF`,

`STI (SeT Interrupt flag);` установка в `1` значения флага `IF`

2.7.11 Цепочечные команды

Эти команды также называют командами *обработки строк символов*. Под *строкой символов* здесь понимается последовательность байт, а *цепочка* — это более общее название для случаев, когда элементы последовательности имеют размер больше байта — слово или двойное слово.

Особенность всех цепочечных команд в том, что они, кроме обработки текущего элемента цепочки, осуществляют еще и автоматическое продвижение к следующему элементу данной цепочки.

Команды **MOVSB (Move (copy) string bytes)**
MOVSW (Move (copy) string words)
MOVSD (Move (copy) string doublewords)

Эти команды позволяют скопировать данные (байта, слова или двойного слова) из одного участка памяти, адрес которого указан в регистре ESI, в другой участок памяти, адрес которого указан в регистре EDI. При этом, в зависимости от состояния флага направления, значение в регистрах ESI и EDI либо увеличивается, либо уменьшается.

Назначение: пересылка элементов двух последовательностей (цепочек) в памяти.

Алгоритм работы:

выполнить копирование байта, слова или двойного слова из операнда источника в операнд приемник, при этом адреса элементов предварительно должны быть загружены:

- адрес источника — в esi
- адрес приемника — в edi

в зависимости от состояния флага df изменить значение регистров esi и edi:

- если df=0, то увеличить содержимое этих регистров на длину структурного элемента последовательности;
- если df=1, то уменьшить содержимое этих регистров на длину структурного элемента последовательности;

если есть префикс повторения, то выполнить определяемые им действия.

Состояние флагов после выполнения команды: выполнение команды не влияет на флаги

Применение: Команды пересылают элемент из одной ячейки памяти в другую. Размеры пересылаемых элементов зависят от применяемой команды.

С командами MOVSB, MOVSW и MOVSD может использоваться префикс повторения.

Существует несколько типов префиксов повторения:

REP - Повторять команду, пока ECX > O
REPZ,REPE - Повторять команду, пока ECX > O и флаг нуля установлен (ZF =1)
REPNZ,REPNE - Повторять команду, пока ECX > O и флаг нуля сброшен (ZF = 0)

Ex. Необходимо скопировать 20 двойных слов из последовательности **source** в последовательность **target**:

```
.data
source DWORD 20 DUP(0FFFFFFFFh)
target DWORD 20 DUP(?)
.code
cld  ; Сбросим флаг DF и установим
     ;прямое направление
mov ecx,LENGTHOF source ; Зададим значение счетчика REP
mov esi,OFFSET source ; Зададим адрес источника данных
mov edi,OFFSET target ; Зададим адрес получателя данных
     rep movsd          ; Копируем 20 двойных слов
```

Команды **CMPSB (Compare string bytes)**
CMPSW (Compare string words)
CMPSD (Compare string doublewords)

Эти команды позволяют сравнить данные из одного участка памяти, адрес которого указан в регистре ESI, с другим участком памяти, адрес которого указан в регистре EDI. Типы команд CMPS приведены ниже.

CMPSB - Сравнивает последовательность байтов
CMPSW - Сравнивает последовательность слов
CMPSD - Сравнивает последовательность двойных слов.

```
.data
source DWORD 1234h
target DWORD 5678h
.code
mov esi,OFFSET source
mov edi,OFFSET target
cmpsd              ; compare doublewords
ja L1              ; jump if source > target
```

Ex2.:

```
mov esi,OFFSET source
mov edi,OFFSET target
cld ; direction = forward
mov ecx,LENGTHOF source ; repetition counter
repe cmpsd ; repeat while equal
```

Команды **SCASB - Scans a string of bytes**
SCASW- Scans a string of words
SCASD - Scans a string of doublewords

Эти команды сравнивают значение, находящееся в регистрах AL/AX/EAX с байтом, словом или двойным словом, адресуемым через регистр EDI.

Данная группа команд обычно используется при поиске какого-либо значения в длинной строке или массиве. Если перед командой SCAS поместить префикс REPE (или REP), строка или массив будет сканироваться до тех пор, пока значение в регистре ECX не станет равным нулю, либо пока не будет найдено значение в строке или массиве, отличное от того, что находится в регистре AL/ AX/EAX (т.е. пока не будет сброшен флаг нуля ZF). При использовании префикса

REPNE, строка или массив будет сканироваться до тех пор, пока значение в регистре ECX не станет равным нулю, либо пока не будет найдено значение в строке или массиве, совпадающее с тем, что находится в регистре AL/AX/EAX (т.е. пока не будет установлен флаг нуля ZF)

```
.data
alpha BYTE "ABCDEFGH",0
.code
mov edi,OFFSET alpha ; Загрузим в EDI адрес строки alpha
mov al,'F'
mov ecx,LENGTHOF alpha ; загрузим в ECX длину строки alpha
cld            ; direction = forward
repne scasb ; Сканируем строку пока не найдем символ "F"
jnz quit ; Если не нашли, завершим работ
```

Команды **STOSB (Store string byte)**
STOSW
STOSD

Эта группа команд позволяет сохранить содержимое регистра AL/AX/EAX в памяти, адресуемой через регистр EDI. При выполнении команды stos содержимое регистра EDI изменяется в соответствии со значением флага направления DF и типом используемого в команде операнда. При использовании совместно с префиксом REP, с помощью команды STOS можно записать одно и то же значение во все элементы массива или строки.

```
.data
Count = 100
string1 BYTE Count DUP(?)
.code
mov al,0FFh ; Записываемое значение
mov edi,OFFSET string1 ; Загрузим в EDI адрес строки
mov ecx,Count ; Загрузим в ECX длину строки
cld ; Направление сравнения -восходящее
rep stosb ; Заполним строку содержимым AL
```

Команды **LODSB (Load Accumulator from String of bytes)**
LODSW
LODSD

Эта группа команд позволяет загрузить в регистр AL/ AX/EAX содержимое байта, слова или двойного слова памяти, адресуемого через регистр ESI. При выполнении команды LODS содержимое регистра ESI изменяется в соответствии со значением флага направления DF и типом используемого в команде операнда. Префикс REP практически никогда не используется с командой LODS, поскольку при этом будет теряться предыдущее значение, загруженное в аккумулятор. Таким образом, эта команда используется для загрузки одного значения в аккумулятор.

EX. В приведенной ниже программе каждый элемент массива двойных слов *array* умножается на постоянное значение. Для загрузки в регистр EAX текущего элемента массива используется команда LODSD, а для сохранения - STOSD.

```
INCLUDE Irvine32.inc
.data
array DWORD 1,2,3,4,5,6,7,8,9,10 ; test data
multiplier DWORD 10
.code
main PROC
```

```
cld                    ; direction = forward
mov esi,OFFSET array ; Загрузим адрес массива
mov edi,esi            ; в регистры ESI и EDI
mov ecx,LENGTHOF array ; Загрузим длину массива
L1: lodsd ; Загрузим текущий элемент [ESI] в EAX
mul multiplier
stosd ; Запишем EAX в текущий элемент
          ;массива (его адрес в [EDI])
loop L1
exit
main ENDP
END main
```

2.8 Процедуры и макрокоманды

Процедуру можно определить, как блок команд, оканчивающийся оператором возврата. Для объявления процедуры используются директивы PROC и ENDP. При объявлении процедуре должно быть назначено имя, которое является одним из разрешенных идентификаторов. При создании процедур, не являющихся стартовыми, вам нужно в их конце разместить команду RET. В результате процессор вернет управления команде, следующей за той, которая вызвала эту процедуру:

```
sample PROC
…
r e t
sample ENDP
```

Давайте создадим процедуру SumOf, вычисляющую сумму трех 32-разрядных чисел.

Предположим, что перед вызовом процедуры значения этих чисел мы должны поместить в регистры EAX, EBX и ECX, а сумма возвращается в регистре EAX:

```
SumOf  PROC
add eax,ebx
add eax,ecx
ret
SumOf  ENDP
```

Команды **CALL** и **RET**

Команда CALL предназначена для передачи управления процедуре, адрес которой указывается в качестве параметра. При этом процессор начинает выполнять команду, расположенную по указанному адресу. Чтобы вернуть управление команде, расположенной сразу за CALL, в процедуре используется команда RET. Команда CALL помещает в стек текущее значение счетчика команд, который на фазе выполнения команды CALL содержит адрес следующей команды, а затем загружает в счетчик команд указанный адрес процедуры. При возврате из процедуры (т.е. при выполнении в ней команды RET), адрес возврата выгружается из стека в счетчик команд. Напомним, что процессор всегда выполняет команду, адрес которой указывается в регистре EIP, т.е. в счетчике команд. В 16-разрядном режиме работы в качестве счетчика команд используется регистр IP.

Пример вызова и возврата из процедуры. Предположим, что в процедуре main по смещению 00000020h расположена команда CALL. В 32-разрядном режиме длина этой команды составляет 5 байтов. Поэтому следуюшая команда (в нашем случае MOV) будет расположена со смещением 00000025h:

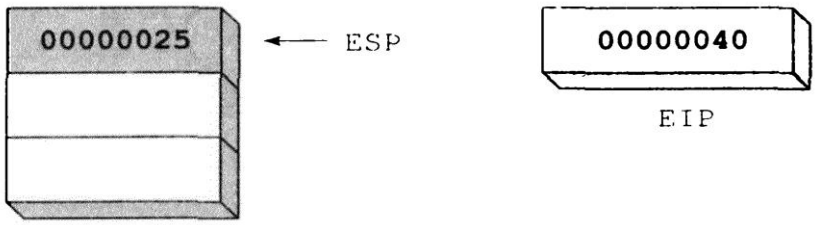
```
     main PROC
00000020   call MySub
```

```
00000025    mov eax,ebx
```

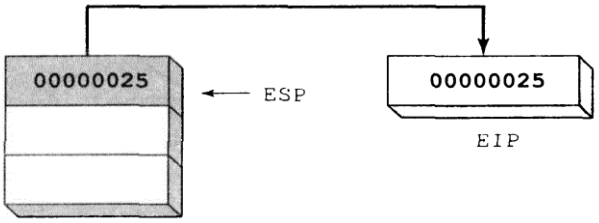
Далее, предположим, что первая команда процедуры MySub расположена со смещением 00000040h:

```
            MySub PROC
00000040     mov eax,edx
            …
            ret
            MySub ENDP
```

При выполнении команды CALL в стек помещается адрес следующей за ней команды (в данном случае 00000025h), после чего в регистр EIP загружается адрес процедуры MySuB, как показано ниже



После этого процессор начинает выполнять последовательность команд процедуры MySuB, пока в ней не встретится команда RET. При выполнении команды RET содержимое стека, на которое указывает регистр ESP, выгружается в регистр EIP. В результате процессор после команды RET будет выполнять не следующую за ней команду, а команду, находящуюся по адресу 00000025h. А это как раз команда, расположенная следом за командой CALL, которая вызвала данную процедуру, как показано ниже



Ранее, мы создали простую процедуру **SumOf.** Которая вычисляет сумму трех чисел, находящихся в регистрах EAX, EBX и ECX. Перед вызовом этой процедуры из процедуры *main* нам нужно загрузить соответствующие значения в регистры EAX, EBX и ECX:

```
INCLUDE Irvine32.inc
.data
                theSum  DWORD ?

.code
main PROC
     mov  eax,10000h ; Первый аргумент
     mov  ebx,20000h ; Второй аргумент
     mov  ecx,30000h ; Третий аргумент
     call  SumOf    ; EAX = ( EAX + EBX + ECX)
     mov theSum,eax ; Сохраним сумму в переменной
exit
main ENDP
     SumOf  PROC
```

```
        add eax,ebx
        add eax,ecx
        ret
        SumOf  ENDP
    END main
```

Макропроцедурой (macro procedure) называется блок команд языка ассемблера. После того как макропроцедура определена в программе, ее можно многократно вызывать в разных участках кода. **При вызове макропроцедуры, в тело программы, в месте вызова, будет помещен весь код макропроцедуры.** Не следует путать вызов макропроцедуры с вызовом обычной процедуры, поскольку в первом случае команда CALL не используется.

Размещение. Определения макрокоманд, или макроопределения, помещаются либо непосредственно в текст исходной программы на ассемблере (как правило, в его начало), либо в отдельный текстовый файл, который включается в исходную программу на этапе компиляции с помощью директивы INCLUDE. Текст макроопределения должен быть обработан ассемблером до вызова макрокоманды в коде программы. Этим занимается препроцессор ассемблера. Он выполняет анализ макроопределений и помещает их буфер. Как только в тексте программы встречается имя макрокоманды, оно заменяется препроцессором на соответствующий набор команд, который указан в ее макроопределении.

Синтаксис макроопределения следующий:

```
имя  MACRO  Параметр1, Параметр2 ...

    Список-команд

    ENDM
```

Команды, находящиеся между директивами MACRO и ENDM, до вызова макрокоманд не компилируются. Макроопределение может содержать произвольное количество параметров, которые разделяются запятыми.

В макроопределении можно указать, что некоторые параметры макрокоманды являются обязательными. Для этого используется описатель REQ. Если при вызове макрокоманды обязательный параметр будет опущен, компилятор сгенерирует сообщение об ошибке. Например:

```
mPutchar MACRO char:REQ
push eax
mov al,char
call WriteChar
pop eax
ENDM
```

Если макрокоманда имеет несколько обязательных параметров, для каждого из них нужно использовать описатель REQ.

Строка комментария в макроопределении начинается после двух символов точки с запятой, стоящих подряд (**;;**). Данный тип комментария располагается только в макроопределении и не переносится в исходный код, генерируемый при вызове макрокоманды.

Для вызова макрокоманды нужно просто поместить ее имя в исходный код программы и при необходимости указать передаваемые ей значения:

```
Имя_макрокоманды Значение1, Значение2, …
```

Порядок передачи значений параметров должен соответствовать порядку их следования в макроопределении, однако число значений необязательно должно соответствовать числу параметров макрокоманды. Если в макрокоманду передается слишком много значений параметров, компилятор выдаст соответствующее предупредительное сообщение. А если

значений будет меньше, чем параметров макрокоманды, вместо недостающих параметров подставляется пустая строка.

Пример: Сумирование 3 слов и сохранение результата в **ax**.

```
INCLUDE Irvine32.inc
   addup MACRO  ad1,ad2,ad3
                mov   ax,ad1
                add   ax,ad2
                add   ax,ad3
                ENDM
      .data
        a      DW    1
        b      DW    2
        cd     DW    3
        d      DW    ?

      .code
           main PROC
                addup a,b,c
                mov   dx,ax
                addup dx,dx,dx
                mov   d,ax
                addup d,dx,c
           exit
      main ENDP
      END main
```

Локальные метки объявляются в макроопределении директивой LOCAL (перед любыми командами). В макроопределении может быть несколько директив LOCAL.

LOCAL имя {,имя} ...

Директива LOCAL, если присутствует в макроопределении, должна следовать за директивой MACRO.

Пример: Возведение числа в степень.

```
          power  MACRO factor,exponent
                  LOCAL again, gotzero
                  xor   dx,dx
                  mov   ax,factor
                  mov   cx,exponent
        again:     jcxz  gotzero
                  mul   bx
                  loop  again
        gotzero:
                  ENDM
```

В данном примере можно заметить, что вызывая данную макрокоманду дважды, код вызывающей программы будет дополнен 2-мя одинаковыми метками `again` и 2-мя `gotzero`, что естественно выдаст ошибку при компилировании кода. Поэтому необходимо эти метки обьявить локальными.

3. Структура ЭВМ

3.1 Основные характеристики и классификация

Определим понятие “микропроцессора” как составную часть ЭВМ. ЭВМ — это программируемая система обработки информации, которая состоит из 2-х составных неделимых компонентов: hardware и software.

A. С точки зрения hardware, ЭВМ состоит из 3-х функционально-взаимосвязанных частей, ее еще называют архитектурой фон Неймана (von Neumann) (рис.3.1)

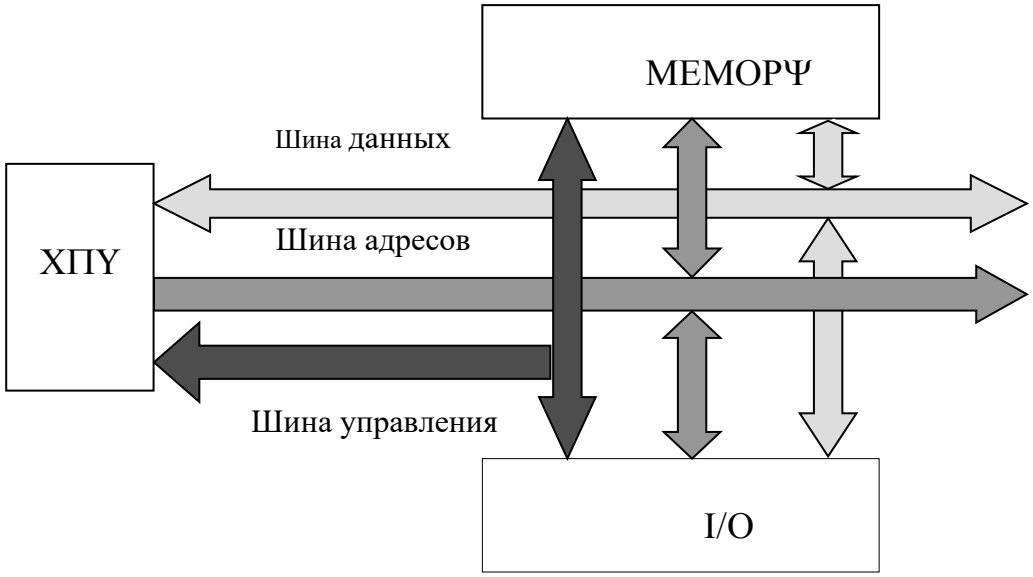


Рисунок 3.1 - Обобщенная структура ЭВМ

Функциональные блоки следующие:

1.*Центральное обрабатывающее устройство* (CPU-central processing unit) выполняет две важнейшие функции:

- обработка данных;
- контроль функционирования всего ЭВМ.

Устройство, которое обрабатывает информацию, управляет системой на рис.3.1 и физически представляет собой микросхему, называется микропроцессором.

2.*Память* состоит из последовательности ячеек памяти предназначенных для хранения информации. Каждая ячейка характеризуется:

- содержанием – последовательность двоичных цифр ”0” или ”1”(биты). Количество бинарных цифр в ячейке (обычно 8 или 16), зависит от способа организации памяти микропроцессора и называется форматом памяти.
- адресом – порядковый номер ячейки.
- Определим некоторые понятия связанные с памятью ЭВМ:
- ”Карта памяти” определяется как совокупность всех ячеек памяти, которые может адресовать микропроцессор.
- ”страницы” и/или ”сегменты” логические подразделения карты памяти, размеры которых, фиксированные или изменяемые, зависят от способа организации памяти микропроцессора.

3. *Устройства ввода/вывода* (Input/Output) состоят из устройств и микросхем, которые обеспечивают взаимосвязь между ЭВМ и внешними устройствами (принтер, mouse).

Элементарное устройство, предназначенное для взаимодействия с внешними устройствами, называется портом ввода/вывода. Порты ввода/вывода можно рассматривать как ячейки памяти (регистры). Единственное существенное различие между портом и ячейкой памяти является физическая связь, которую реализует порт с внешними устройствами.

Функциональной частью ЭВМ является еще и информационная шина, по которой осуществляется обмен между всеми компонентами ЭВМ.

Функционально информационная шина подразделяется:

1.Шина данных, двунаправленная по которой осуществляется обмен данными (операнды/результаты), командами, даже адресами.

2.Шина адреса, однонаправленная, микропроцессора выставляет адреса для обращения к памяти или внешним устройствам. По этой шине циркулируют только адреса.

3. Шина управления, двунаправленная, по которой передаются сигналы управления и контроля от/к микропроцессору.

В. Программная часть software, вторая составляющая ЭВМ. Состоит из ряда специальных программ необходимых для функционирования ЭВМ.

На основе вышеназванных определений можно определить понятие микропроцессора, так:

- микропроцессор является центральным обрабатывающим устройством некоторой системы, функциональная блок-схема которой представлена на рис.3.1.
- Важно то, что микропроцессор аккумулирует функции управления и обработки информации.
- Взаимосвязь между блоками системы осуществляется посредством шины, состоящей из трех функциональных составных компонентов; по шине данных происходит обмен всеми видами информации.
- Функционирование системы происходит в соответствии с программами, состоящими из последовательности программ. Команды считываются из памяти, декодируются и исполняются.

ЭВМ чаще всего характеризуются следующими показателями:

- Производительность - оценка, определяемая аналитически или экспериментально, количества обобщенных операций (команд), которые выполняет ЭВМ в единицу времени. Производительность измеряется в операциях в секунду (Instruction per second, IPS), в миллионах операций в секунду (million IPS, MIPS), в миллионах операций с плавающей запятой в секунду (megaflops , MFLOPS).
- Разрядность – максимальная ширина информационного кода, который может обрабатываться, храниться и пересылаться в ЭВМ как единое целое.
- Емкость запоминающих устройств – количество закодированной информации, которая может одновременно храниться в устройствах памяти ЭВМ. Емкость измеряется в байтах, килобайтах (1 КБайт= 2^{10} Байт, KBytes), мегабайтах (1 МБайт= 2^{20} Байт, MBytes), гигабайтах (1 ГБайт= 2^{30} Байт, GBytes), терабайтах (1 ТБайт= 2^{40} Байт, TBytes).

Классифицировать современные ЭВМ по этим показателям неактуально. Рассмотрим несколько классификаций.

Современная промышленность производит очень большой спектр разнообразных компьютеров. Из всего разнообразия, приведем примерную классификацию современных компьютеров (см. Таненбаум Э. «Архитектура компьютера»):

- Интегральная микросхема или «одноразовые» компьютеры, сфера применения - например, поздравительные открытки;
- Встроенные компьютеры (микроконтроллеры) - часы, машины, различные приборы;
- Игровые компьютеры - домашние компьютерные игры;
- Персональные компьютеры - настольные и портативные компьютеры;
- Серверы - сетевые серверы;
- Комплексы рабочих станций - супермини-компьютеры;
- Мэйнфреймы - пакетная обработка данных в банке.

Интегральная микросхема.

Называют еще одноразовыми компьютерами. Это микросхемы, которые приклеиваются на внутреннюю сторону поздравительных открыток для проигрывания мелодий типа «Happy Birthday», свадебного марша или чего-нибудь подобного. Вероятно, наиболее значимым достижением в этой области стало появление микросхем RFID (Radio Frequency Identification — радиочастотная идентификация). Теперь на безбатарейных микросхемах этого типа толщиной меньше 0,5 мм и себестоимостью в несколько центов устанавливаются крошечные приемопередатчики радиосигналов; кроме того, им присваивается уникальный 128-разрядный идентификатор. При получении импульса с внешней антенны они за счет достаточно длинного

радиосигнала отправляют ответный импульс со своим номером. В отличие от размера микросхем, спектр их практического применения весьма значителен.

Взять хотя бы снятие штрих-кодов с товаров в магазинах. Уже проводились испытания, в ходе которых производители снабжали все выпускаемые ими товары микросхемами RFID (вместо штрих-кодов). При наличии таких микросхем покупатель может выбрать нужные продукты, положить их в корзину и, минуя кассу, выйти из магазина. По выходе считывающее устройство с антенной отсылает сигнал, заставляющий микросхемы на всех приобретенных товарах «рассказать» о себе, что они и делают путем беспроводной отсылки короткого импульса. Покупатель, в свою очередь, идентифицируется по микросхеме на его банковской/кредитной карточке. В конце каждого месяца магазин выставляет покупателю детализированный счет за все приобретенные за этот период товары. Если действующая банковская/кредитная карта на микросхеме RFID у покупателя не обнаруживается, звучит аварийный сигнал.

На самом деле, вариантов применения технологии RFID великое множество. Среди них — определение цвета кузовов автомобилей перед их покраской в цехе, изучение миграции животных, указание температурного режима стирки предметов одежды и т. д. Микросхемы можно снабжать датчиками, в этом случае текущие показания температуры, давления, влажности и многие другие параметры окружающей среды.

Современные микросхемы RFID предусматривают возможность долговременного хранения информации. На этом основании Европейский Центробанк принял решение наладить в ближайшие годы выпуск банкнот с вживленными микросхемами.

Технология RFID постоянно совершенствуется, и, более детальную информацию можно найти на сайте www.rfid.org.

Микроконтроллеры

Вторая категория в таблице отведена под компьютеры, которыми оснащаются разного рода бытовые устройства. Такого рода встроенные компьютеры, называемые также микроконтроллерами, выполняют функцию управления устройствами и организации их пользовательских интерфейсов. Диапазон устройств, работающих с помощью микрокомпьютеров, крайне широк (примеры даются в скобках):

- бытовые приборы (будильники, стиральные машины, сушильные аппараты, микроволновые печи, охранные сигнализации);
- коммуникаторы (беспроводные и сотовые телефоны, факсимильные аппараты, пейджеры);
- периферийные устройства (принтеры, сканеры, модемы, приводы CD-ROM);
- развлекательные устройства (видеомагнитофоны, DVD-плееры, музыкальные центры, MP3-плееры, телеприставки);
- формирователи изображений (телевизоры, цифровые фотокамеры, видеокамеры, объективы, фотокопировальные устройства);
- медицинское оборудование (рентгеноскопические аппараты, томографы, кардиомониторы, цифровые термометры);
- военные комплексы вооружений (крылатые ракеты, межконтинентальные баллистические ракеты, торпеды);
- торговое оборудование (торговые автоматы, кассовые аппараты);
- игрушки (говорящие куклы, приставки для видеоигр, радиоуправляемые машинки и лодки).

В любой современной машине представительского класса устанавливается по полсотни микроконтроллеров, которые управляют различными подсистемами, в частности, автоблокировкой колес, впрыском топлива, магнитолой, системой навигации т.п. В реактивных самолетах количество микроконтроллеров достигает 200 и даже больше!

В отличие от микросхем RFID, выполняющих минимальный набор функций, микроконтроллеры представляют собой полноценные вычислительные устройства. Каждый микроконтроллер состоит из микропроцессора, памяти и средств ввода-вывода. Ввод-вывод, как правило, осуществляется посредством кнопок и переключателей с контролем состояния световых

индикаторов, дисплея и звуковых компонентов устройства. Программное обеспечение микроконтроллеров в большинстве случаев «прошивается» производителем в виде постоянной памяти. Все микроконтроллеры можно разделить на два типа: универсальные и специальные. Первые фактически являют собой обычные компьютеры, уменьшенные в размере. Специальные же микроконтроллеры отличаются индивидуальной архитектурой и набором команд, приспособленными для решения определенного круга задач, например, связанных с воспроизведением мультимедийных данных. Микроконтроллеры бывают 4-, 8-, 16-, 32-разрядными.

Игровые компьютеры

Следующая категория — игровые компьютеры. Это, по существу, обычные компьютеры, в которых расширенные возможности графических и звуковых контроллеров сочетаются с ограничениями по объему ПО и пониженной расширяемостью. Сегодня игровые компьютеры превратились в достаточно мощные системы, которые по некоторым параметрам производительности ничем не хуже, а иногда даже лучше персональных компьютеров.

Чтобы получить представление о том, чем комплектуются игровые компьютеры, рассмотрим конфигурацию модели — XBOX-360. Встроенная память на 250 ГБ, микропроцессор IBM PowerPC 3,2 ГГц, 3 ядра, кэш 1 Мб второго уровня, оперативная память 512 MB GDDR3 RAM на частоте 700 МГц, графическая микросхема 500 МГц, 256-канальная звуковая микросхема, DVD-привод.

Основное различие между игровыми машинами и ПК, впрочем, состоит не в производительности микропроцессора, а в том, что игровые компьютеры представляют собой закрытые, законченные системы. Расширяемость таких систем при помощи сменных плат не предусмотрена, хотя в некоторых моделях наличествуют интерфейсы USB и Fire Wire.

Персональные компьютеры

В следующую категорию входят персональные компьютеры. Персональные компьютеры бывают двух видов: настольные и портативные (ноутбуки). Как правило, те и другие комплектуются модулями памяти общей емкостью в гигабайтах, жестким диском с данными в сотни гигабайтов, приводом CD-ROM/DVD, модемом, звуковой картой, сетевым интерфейсом, монитором с высоким разрешением и рядом других периферийных устройств. На них устанавливаются сложные операционные системы, они расширяемы, при работе с ними используется широкий спектр программного обеспечения. Некоторые специалисты называют «персональными» компьютеры с микропроцессорами Intel, отделяя их тем самым от машин, оснащенных высокопроизводительными RISC-микросхемами (такими как Sun UltraSPARC), которые в таком случае именуются «рабочими станциями». На самом деле, особой разницы между этими двумя типами нет.

К персональным очень близки карманные компьютеры (PDA, КПК). Они еще меньше, чем ноутбуки, однако микропроцессор, память, клавиатура, дисплей и большинство других стандартных компонентов персонального компьютера в них присутствуют.

Серверы

Мощные персональные компьютеры и рабочие станции часто используются в качестве сетевых серверов — как в локальных сетях (обычно в пределах одной организации), так и в Интернете. Серверы, как правило, поставляются в одномикропроцессорной и мультимикропроцессорной конфигурациях. В системах из этой категории обычно устанавливаются модули памяти общим объемом в десятки гигабайтов, жесткие диски емкостью в десятки терабайтов и высокоскоростные сетевые интерфейсы. Они способны обрабатывать сотни тысяч транзакций в секунду.

С точки зрения архитектуры одномикропроцессорный сервер не слишком отличается от персонального компьютера. Он просто работает быстрее, занимает больше места, содержит больше дискового пространства и устанавливает более скоростные сетевые соединения. Серверы работают под управлением тех же операционных систем, что и персональные компьютеры, как правило, это различные версии UNIX и Windows.

Комплексы рабочих станций

В связи с тем, что по соотношению «цена/производительность» позиции рабочих станций и персональных компьютеров постоянно улучшаются, в последние годы появилась практика их объединения в рамках кластеров рабочих станций (Clusters Of Workstations, COW), которые иногда называют просто «кластерами».

Они состоят из нескольких персональных компьютеров или рабочих станций, подключенных друг к другу по высокоскоростной сети и снабженных специальным программным обеспечением, которое позволяет направлять их ресурсы на решение единых задач (как правило, научных и инженерных). В большинстве случаев компоненты кластера — это совершенно обычные компьютеры, которые можно приобрести по отдельности в любом компьютерном магазине. Высокоскоростные сетевые соединения, как правило, тоже можно организовать при помощи стандартных сетевых плат. Кластеры отличаются удобством масштабирования — любой кластер можно расширить с десятка до нескольких тысяч машин.

Нередко в виде кластеров организуются веб-серверы. Если частота обращений к страницам веб-сайта исчисляется тысячами в секунду, дешевле организовать кластер из нескольких сотен (или даже тысяч) серверов и распределить между ними нагрузку по обработке запросов. Кластеры, реализующие такую схему, часто называют серверными фермами (server farms).

Мэйнфреймы

Мэйнфреймы работают ненамного быстрее, чем мощные серверы, но у них выше скорость процессов ввода-вывода и обладают они довольно большим пространством на диске — сотни терабайт. Такие системы стоят очень дорого и требуют крупных вложений в программное обеспечение, данные и персонал, обслуживающий эти компьютеры. Многие компании считают, что дешевле заплатить несколько миллионов долларов один раз за такую систему, чем даже думать о необходимости заново программировать все прикладные программы для маленьких компьютеров.

В последние годы под влиянием Интернета наблюдается возрождение мэйнфреймов как полноценной категории компьютеров. Они заняли нишу мощных серверов Интернета, способных обрабатывать огромное количество транзакций в секунду, что крайне актуально для электронной коммерции в целом и компаний, вынужденных обслуживать громадные базы данных, в частности.

До последнего времени существовала еще одна крупная категория вычислительных машин — суперкомпьютеры. Их микропроцессоры работали с очень высокой скоростью, в них устанавливались модули памяти большой емкости, высокоскоростные диски и сетевые интерфейсы. Суперкомпьютеры используются для решения различных научных и технических задач, которые требуют сложных вычислений, например таких, как моделирование сталкивающихся галактик, синтез новых лекарственных препаратов, моделирование потока воздуха вокруг крыла аэроплана. Сейчас, когда вычислительные возможности, аналогичные тем, что предлагают суперкомпьютеры, реализуются в виде кластеров, возможно эта категория компьютеров постепенно исчезнет.

[illegible] 60–[illegible].

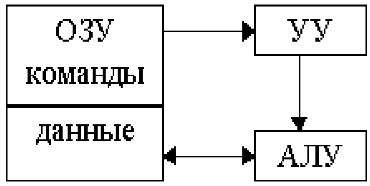
[illegible]

[illegible] 4 [illegible]:

- ΣΙΣΔ – Σινγλε Ινστρυχτιον Σινγλε Δατε – [illegible].
- ΣΙΜΔ – Σινγλε Ινστρυχτιον Μυλτιπλε Δατε – [illegible].

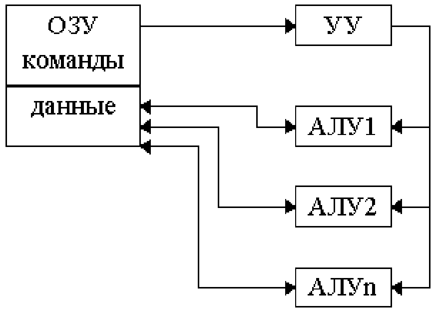
- MIΣΔ – Μυλτιπλε Ινστρυχτιον Σινγλε Δατε – [illegible].
- MIMΔ – Μυλτιπλε Ινστρυχτιον Μυλτιπλε Δατε – [illegible].

[illegible] ΣΙΣΔ – [illegible] (ℵ⊄©). [illegible] (□⌈〉 3.2).



∠⌈〉 [illegible] 3.2 [illegible] ΣΙΣΔ

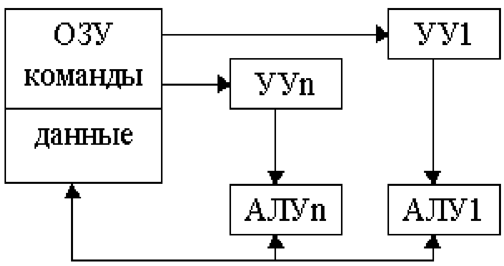
[illegible] ΣΙΜΔ [illegible] (ℵ⊄© [illegible]) [illegible] (□⌈〉. 3.3).



∠⌈〉 [illegible] 3.3 [illegible] ΣΙΜΔ

[illegible] ΙΛΛΙΑΧ–Ις. [illegible] Χψβερ [illegible] Χραψ, [illegible] – ∉∇–2000, ∉∇–3000.

[illegible] MIΣΔ – [illegible] (ℵ⊄© [illegible]) [illegible] ℵ⊄© (□⌈〉. 3.4).



∠⌈〉 [illegible] 3.4 [illegible] MIΣΔ

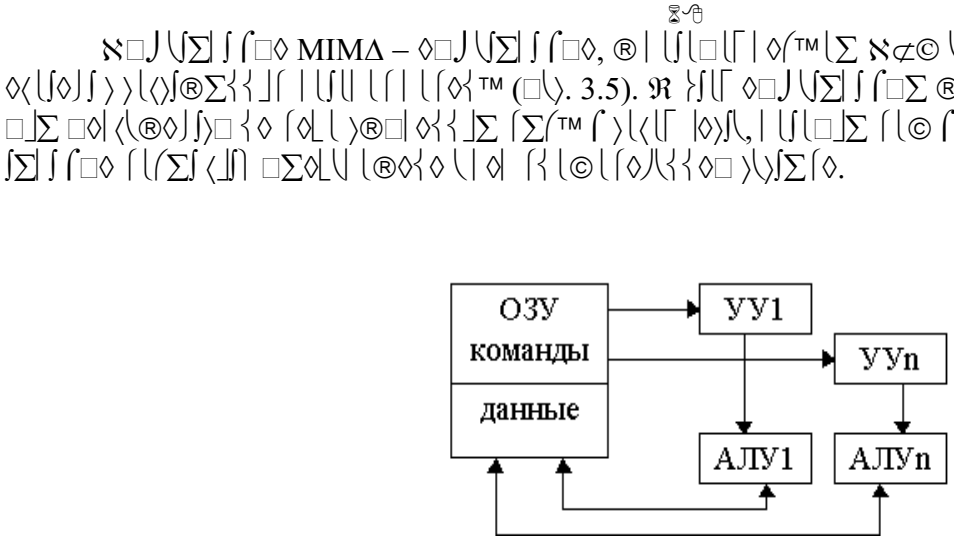


Рис. 3.5 Архитектура МПА

Основное внимание мы уделим вычислительной системе: персональным компьютерам (PC).

Историческая ретроспектива микропроцессоров Intel

Как известно, все микропроцессоры персональных компьютеров основаны на оригинальной структуре Intel. Первым применяемым в PC микропроцессором был интеловский чип 8088. В это время Intel располагал выпущенным ранее более мощным микропроцессором 8086. 8088 был выбран по соображениям экономии: его 8-битная шина данных допускала более дешевые системные платы, чем 16-битная у 8086. Также во время проектирования первых PC большинство доступных интерфейсных микросхем использовали 8-битные структуры.

В таблице ниже приведены основные группы интеловских микропроцессоров от первого поколения 8088/86 до седьмого Pentium IV:

Тип/ Поколение	Дата	Ширина шины Данных /Адреса Bit	Внутренний кэш	Скорость шины памяти (MHz)	Частота (MHz)
8088/ First	1979	8/20	None	4.77-8	4.77-8
8086/ First	1978	16/20	None	4.77-8	4.77-8
80286/ Second	1982	16/24	None	6-20	6-20
80386DX/ Third	1985	32/32	None	16-33	16-33
80386SX/ Third	1988	16-32	8K	16-33	16-33
80486DX/ Fourth	1989	32/32	8K	25-50	25-50
80486SX/ Fourth	1989	32/32	8K	25-50	25-50
80486DX2/ Fourth	1992	32/32	8K	25-40	50-80
80486DX4/ Fourth	1994	32/32	8K+8K	25-40	75-120
Pentium/ Fifth	1993	64/32	8K+8K	60-66	60-200
MMX/ Fifth	1997	64/32	16K+16K	66	166-233

Pentium Pro/ Sixth	1995	64/36	8K+8K	66	150-200
Pentium II/ Sixth	1997	64/36	16K+16K	66	233-300
Pentium III/Sixth	1999	64/36	32K+32K	100	650-1400
Pentium 4/ Seventh	2000	64/36	64K+64K	100	1300-3800

Третье поколение микропроцессоров, основанных на Intel 80386SX и 80386DX, были первыми применяемыми в PC 32-битными микропроцессорами. Основным отличием между ними было то, что 386SX был 32-разрядным только внутри, поскольку он общался с внешним миром по 16-разрядной шине. Это значит, что данные между микропроцессором и остальным компьютером перемещались на вполовину меньшей скорости, чем у 486DX.

Четвертое поколение микропроцессоров была также 32-разрядной. Однако, все они предлагали ряд усовершенствований. Во-первых, был полностью пересмотрен весь дизайн 486 поколения, что само по себе удвоило скорость. Он мог выполнять операции с плавающей точкой и имел кэш-память объемом 8 Кбайт. Кэш-память позволяет держать наиболее часто используемые слова внутри центрального микропроцессора и избегать длительных обращений к основной (оперативной) памяти. Иногда кэш-память находится не внутри центрального микропроцессора, а рядом с ним. Микропроцессор 80486 содержал встроенные средства поддержки мультипроцессорного режима, что давало производителям возможность конструировать системы с несколькими микропроцессорами, приводя к сравнимой прибавке в производительности.

Pentium, определив пятое поколение микропроцессоров, значительно превзошел в производительности предшествующие 486 чипы благодаря нескольким архитектурным изменениям, включая удвоение ширины шины до 64 бит. В отличие от микропроцессора 80486, у которого был один внутренний конвейер, Pentium имел два, что позволяло работать ему почти в два раза быстрее.

Pentium Pro, появившись в 1995 году, был первым в шестом поколении микропроцессоров и ввел способ выполнения инструкций переводом их в RISC-подобные микроинструкции и выполнением их в высокоразвитом внутреннем ядре. Он использует интегрированный кэш второго уровня на своей собственной шине, работающей на полной частоте микропроцессора, обычно в три раза быстрее кэша на Pentium-системах.

Следующий новый чип после Pentium Pro Intel представил спустя почти полтора года - появился Pentium II. Архитектурно Pentium II не очень отличается от Pentium Pro с подобным эмулирующим x86 ядром и большинством схожих особенностей.

Pentium II улучшил архитектуру Pentium Pro удвоением размера первичного кэша до 32kb, использованием специального кэша для увеличения эффективности 16-битной обработки, (Pentium Pro оптимизирован для 32-битных приложений, а с 16-битным кодом не обращается столь же хорошо) и увеличением размеров буферов записи. Интегрированный в Pentium Pro вторичный кэш, работающий на полной частоте микропроцессора, был заменен в Pentium II на малую схему, содержащую микропроцессор и 512kb вторичного кэша, работающего на половине частоты микропроцессора. Собранные вместе, они заключены в специальный односторонний картридж (single-edge cartridge-SEC), предназначенный для вставления в 242-пиновый разъем (Socket 8) системных платах Pentium II. Через некоторое время для улучшенной передачи трехмерной графики в микропроцессор были введены дополнительные мультимедийные команды под названием SSE (Streaming SIMD Extensions — потоковые SIMD-расширения) ~ в результате появился микропроцессор Pentium III. Следующая модель Pentium получила новую внутреннюю архитектуру. Одновременно было решено перейти с римских цифр в обозначениях моделей на арабские. Так появился микропроцессор Pentium 4. По традиции он превзошел все предыдущие модели по быстродействию. В версии с тактовой частотой 3,06 ГГц была реализована новая функция — гиперпоточность (hyperthreading). Она позволяет программам разделять задачи на два программных потока, которые обрабатываются микропроцессором параллельно; следовательно,

скорость выполнения повышается. Кроме того, для дальнейшего повышения скорости обработки звуковых и видеоданных был внедрен дополнительный набор SSE-команд.

В 2003 году появилась микросхема Pentium M (где M — сокращение от «Mobile») для портативных компьютеров. Она задумывалась как составная часть новой архитектуры Centrino, которая должна была, во-первых, снизить энергопотребление, увеличив тем самым ресурс аккумулятора, во-вторых, обеспечить возможность производства более компактных и легких корпусов, в-третьих, организовать при помощи встроенного интерфейса беспроводные сетевые соединения по стандарту IEEE 802.11 (WiFi, WiMax).

Микропроцессор Pentium 4 с тактовой частотой 3,6 ГГц потребляет 115 Вт. При этом он выделяет примерно столько же тепла, сколько лампочка на 100 Вт. Чем больше повышается тактовая частота, тем заметнее становится проблема охлаждения. В ноябре 2004 года компания Intel была вынуждена отменить выпуск модели Pentium 4 с тактовой частотой 4 ГГц из-за проблем с теплоотводом.

Intel теперь размещает на одной микросхеме два (и более) микропроцессора и снабжает ее общим кэшем большого объема. Поскольку величина энергопотребления определяется напряжением и тактовой частотой, два микропроцессора на одной схеме потребляют меньше энергии, чем один, работающий на аналогичной скорости.

3.2 Микропроцессоры CISC/RISC

CISC (англ. Complex Instruction Set Computing) — концепция проектирования микропроцессоров, которая характеризуется следующим набором свойств:

- Нефиксированным значением длины команды.
- Исполнение операций, таких как загрузка в память, арифметические действия кодируется в одной инструкции.
- Небольшим числом регистров, каждый из которых выполняет строго определённую функцию.

Типичными представителями являются микропроцессоры на основе x86 команд (исключая современные Intel Pentium 4, Pentium D, Core, AMD Athlon, Phenom которые являются гибридными).

Наиболее распространённая архитектура современных настольных, серверных и мобильных микропроцессоров построена по архитектуре Intel x86 (или x86-64 в случае 64-разрядных микропроцессоров). Формально, все x86-микропроцессоры являлись CISC-микропроцессорами, однако новые микропроцессоры, начиная с Intel486DX, являются CISC-микропроцессорами с RISC-ядром. Они непосредственно перед исполнением преобразуют CISC-инструкции микропроцессоров x86 в более простой набор внутренних инструкций RISC.

В микропроцессор встраивается аппаратный транслятор, превращающий команды x86 в команды внутреннего RISC-микропроцессора. При этом одна команда x86 может порождать несколько RISC- команд (в случае микропроцессоров типа P6 - до 4-х RISC команд в большинстве случаев). Исполнение команд происходит на суперскалярном конвейере одновременно по несколько штук. Это потребовалось для увеличения скорости обработки CISC-команд, так как известно, что любой CISC-микропроцессор уступает RISC-микропроцессорам по количеству выполняемых операций в секунду. В итоге, такой подход и позволил поднять производительность CPU.

RISC (англ. Reduced Instruction Set Computing) — вычисления с сокращённым набором команд. Это концепция проектирования микропроцессоров, которая во главу ставит следующий принцип: более компактные и простые инструкции выполняются быстрее. Простая архитектура позволяет как удешевить микропроцессор, так и поднять тактовую частоту. Многие ранние RISC-микропроцессоры даже не имели команд умножения и деления. Идея создания RISC микропроцессоров пришла, потому что, поскольку некоторые сложные операции использовались редко, они как правило были медленнее, чем те же действия, выполняемые набором простых команд. Это происходило из-за того, что создатели микропроцессоров тратили гораздо меньше времени на улучшение сложных команд, чем на улучшение простых.

Первые RISC-микропроцессоры были разработаны в начале 1980-х годов в Стэнфордском и Калифорнийском университетах США. Они выполняли небольшой (50 − 100) набор команд, тогда как обычные CISC (Complex Instruction Set computer) выполняли 100—200.

Характерные особенности RISC-микропроцессоров:

- Фиксированная длина машинных инструкций (например, 32 бита) и простой формат команды.
- Одна инструкция выполняет только одну операцию с памятью — чтение или запись. Операции вида «прочитать-изменить-записать» отсутствуют.
- Большое количество регистров общего назначения (32 и более).

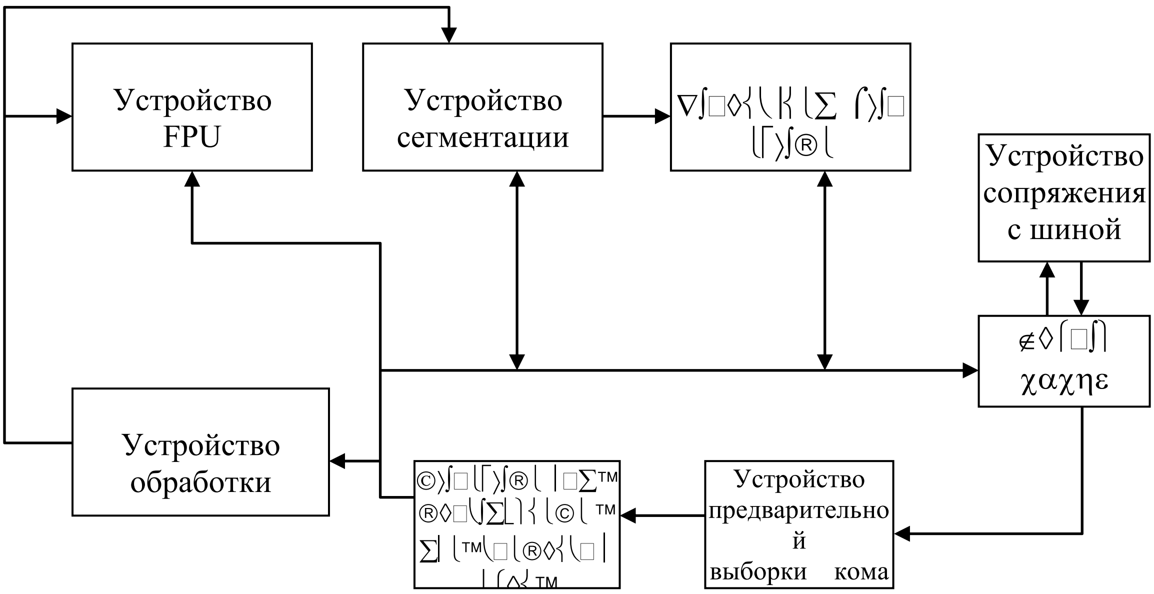
В настоящее время многие архитектуры микропроцессоров являются RISC-подобными, к примеру, ARM, DEC Alpha, SPARC, AVR, MIPS, POWER, PowerPC.

Наиболее широко используемые в современных настольных компьютерах это CISC-микропроцессоры архитектуры x86 с RISC-ядром.

3.3 Конвейер

Введем понятие суперскалярной архитектуры. Для того чтобы пояснить этот термин, разберемся вначале со значением другого термина — *конвейеризация вычислений.* Для этого рассмотрим функциональную схему микропроцессора i486. На рисунке 3.7 дана функциональная схема, на которой можно выделить его функциональные узлы:

- устройство сопряжения с шиной;



∠⟩⎡⎨⎢ 3.7

– ®⎨ ⎡∫□∑⎨⎨□□⎢⎬⎠–⎢◊⎡□∫ □◊⎢⎡∑□⎣⎡ 8кВ;
– ⎡⎬∫□⎣⎡⎬∫®⎣ ⎢□∑™®◊□⎣∫∑⎦⎨⎣⎡ ®⎦⎨⎣□⎢⎣⎢⎣⎡◊⎨™,⎢⎣∫⎣□⎣∑ ⎣⎡∑∑∫ 32–⟨◊⎡⎨ ⎡⎦ ⎣⎢∑□∑™⎤⎢⎣⎡◊⎨™, ⎣⎡⎝ ™◊⎦⎤⎣⎦ ®⎦⎢ ⎣⎦⎨∑⎨⎣□;
– ⎡⎬∫□⎣⎡⎬∫®⎣ ⎢□∑™®◊□⎣∫∑⎦⎨⎣© ⎣™∑⎢ ⎣™⎣□⎣®◊⎨⎣□⎢ ⎣⎡◊⎨™ ⎢□∑⎣⟨□◊⎢ ⎡∑∫⎢ ⎣⎡◊⎨™⎦® ⎡⎣ □⎣⎢ ⎣™;
– ⎡⎣™ ⎡⎣⎤ ◊□⎣⎢⎡∑⎡⎣⎢∑⎢ ⎣©⎣ ⎬⎣⎢□⎣⎞∑⎬⎬⎣□◊ (Φλοατινγ Ποιντ Υνιτ – ΦΠΥ);
– ⎡⎬∫□⎣⎡⎬∫®⎣ ⎬∑©⎡∑⎨∫◊⎞⎣⎣ ⎣⎬∫□◊⎨⎣⎨⎣∑ ⎡⎬∫□⎣⎡⎬∫®⎣ ⎢⎣□⎡⎣□ ⎡⎦∫ ◊™□∑⎬◊ (⎣⎬⎢⎣⎦⎢ ⎡⎦∫⎬□ ∫⎣⎦⎢⎣ ®⎢◊⎤⎣⎤∑ ⎨⎨⎣⎡ □∑⎡⎣⎡∑ □◊⟨⎣∫⎦ ⎢□⎣⎞∑⎬⎬⎣□◊). ⊄⎣©⎣⎢∑⎢⎣∑ ◊™□∑⎬◊ ⎢□∑⎣⟨□◊⎢ ⎡⎦∫⎬□ ⎡⎬∫□⎣⎡⎬∫®⎣⎡ ⎬∑©⎡∑⎨∫◊⎞⎣⎣

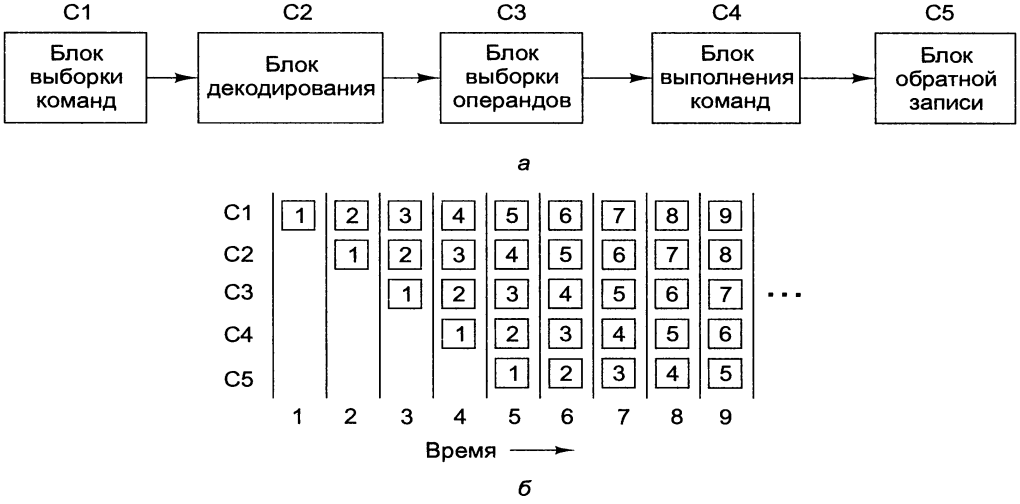


Рисунок 3.8

На рис. 3.8, б мы видим, как действует конвейер во времени. Во время цикла 1 блок С1 обрабатывает команду 1, вызывая ее из памяти. Во время цикла 2 блок С2 декодирует команду 1, в то время как блок С1 вызывает из памяти команду 2. Во время цикла 3 блок С3 вызывает операнды для команды 1, блок С2 декодирует команду 2, а блок С1 вызывает команду 3. Во время цикла 4 блок С4 выполняет команду 1, С3 вызывает операнды для команды 2, С2 декодирует команду 3, а С1 вызывает команду 4. Наконец, во время цикла 5 блок С5 записывает результат выполнения команды 1 обратно в регистр, тогда как другие ступени конвейера обрабатывают следующие команды.

Таким образом, на стадии выполнения каждая машинная команда как бы разбивается на элементарные операции. В результате на конвейере могут находиться в различной стадии выполнения пять команд. Скорость вычисления в результате существенно возрастает.

Микропроцессоры, имеющие один конвейер, называются *скалярными,* а два и более — *суперскалярными.* Микропроцессор Pentium имеет два конвейера, то есть использует суперскалярную архитектуру, и поэтому может выполнять две команды за машинный такт. Внутренняя структура конвейера такая же, как и у i486.

Одна из возможных схем процессора с двумя конвейерами показана на рис. 3.9. В ее основе лежит конвейер, изображенный на рис. 3.8. Здесь общий блок выборки команд вызывает из памяти сразу по две команды и помещает каждую из них в один из конвейеров.

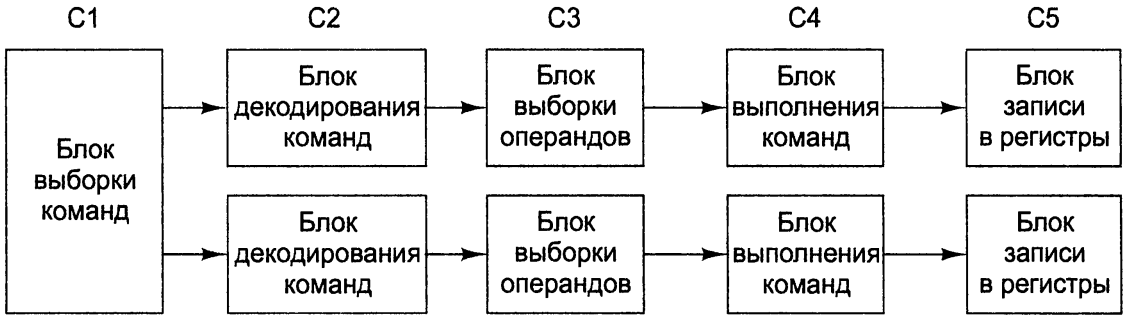


Рисунок 3.9

Каждый конвейер содержит АЛУ для параллельных операций. Чтобы выполняться параллельно, две команды не должны конфликтовать из-за ресурсов (например, регистров), и ни одна из них не должна зависеть от результата выполнения другой. Функции между второй и третьей ступенями (они назывались декодер 1 и декодер 2) отличались. Главный конвейер (u-конвейер) мог выполнять произвольные команды. Второй конвейер (v- конвейер) мог выполнять только простые команды с целыми числами, а также одну простую команду с плавающей точкой (FXCH).

Переход к четырем конвейерам возможен, но требует громоздкого аппаратного обеспечения. Вместо этого используется другой подход. Основная идея — один конвейер с большим количеством функциональных блоков, как показано на рис. 3.10. Pentium II, к примеру, имеет сходную структуру.

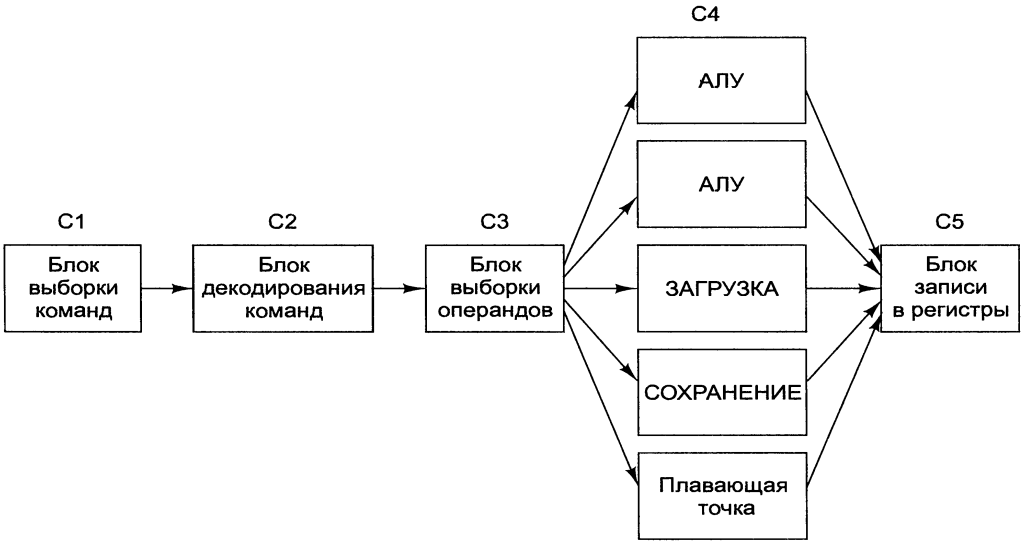


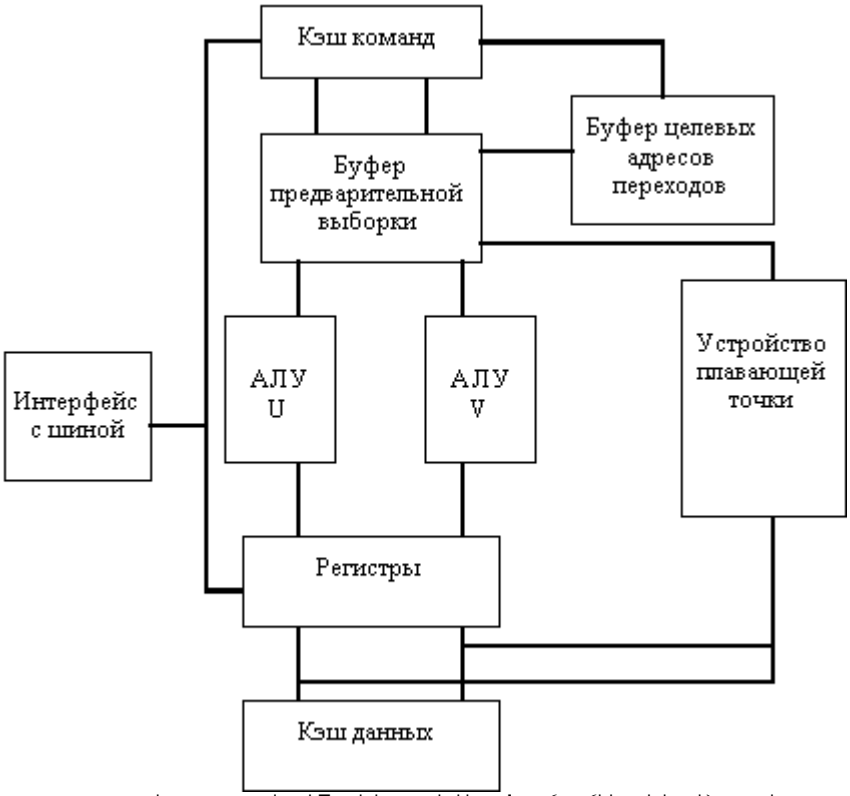
Рис. 3.10 Суперскалярный процессор с пятью функциональными блоками

Отметим, что на выходе ступени 3 команды появляются значительно быстрее, чем ступень 4 способна их обрабатывать. Большинству функциональных блоков ступени 4 (точнее, обоим блокам доступа к памяти и блоку выполнения операций с плавающей точкой) для обработки команды требуется значительно больше времени, чем занимает один цикл. Как видно из рис. 3.10, на ступени 4 может быть несколько АЛУ.

Микропроцессоры семейства P6 (Pentium Pro/II/III) имеют другую структуру конвейера.

4 Уровень микроархитектуры процессоров

Рассмотрим вначале блок-схему микропроцессора Pentium представлена на рис. 4.1. Прежде всего, новая микроархитектура (микроархитектура — это способ, которым данная архитектура набора команд (АНК, ISA) реализована в процессоре.) этого микропроцессора базируется на идее суперскалярной обработки (правда с некоторыми ограничениями). Основные команды распределяются по двум независимым исполнительным устройствам (конвейерам U и V). Конвейер U может выполнять любые команды семейства x86, включая целочисленные команды и команды с плавающей точкой. Конвейер V предназначен для выполнения простых целочисленных команд и некоторых команд с плавающей точкой. Команды могут направляться в каждое из этих устройств одновременно, причем при выдаче устройством управления в одном такте пары команд более сложная команда поступает в конвейер U, а менее сложная - в конвейер V.



∠\⟩. 4.1 ©|□⌊⌉Σ⦃⦄◊□ ⟨⌊⌊ ⟩J Σ⌈◊ ⌈⌊ □⌊|□⌊⌉Σ⟩⟩⌊□◊ Пентиум

Раздельное кэширование кода и данных. Кэширование — это способ увеличения быстродействия системы за счет хранения часто используемых данных и кодов в так называемой «кэш-памяти первого уровня» (быстрой памяти), находящейся внутри микропроцессора, i486, к примеру, содержит один блок встроенной Кэш-памяти размером 8 Кбайт, который используется для кэширования и кодов, и данных. Pentium содержит два блока кэш-памяти: один для кода и один для данных, каждый по 8 Кбайт. При этом становится возможным одновременный доступ к коду и данным, что увеличивает скорость работы компьютера.

Предсказание правильного адреса перехода. Под *переходом* понимается запланированное алгоритмом изменение последовательного характера выполнения программы. Как показывает статистика, типичная программа на каждые 6-8 команд содержит 1 команду перехода. Последствия этого предсказать несложно: при наличии конвейера через каждые 6-8 команд его нужно очищать заполнять заново в соответствии с адресом перехода. Все преимущества конвейеризации теряются. Поэтому в архитектуру Pentium был введен *блок предсказания переходов.* Суть этого метода заключается в следующем. Pentium имеет буфер адресов перехода, который хранит информацию о последних 256 переходах. Если некоторая команда управляет ветвлением, то в буфере запоминаются эта команда, адрес перехода и предположение о том, какая ветвь программы будет выполнена следующей. Почти в любой программе имеются циклы, в ходе

выполнения которых периодически необходимо принимать решение либо о выходе из цикла, либо о переходе на его начало. Специальный блок предсказания адреса перехода прогнозирует, какое решение будет принято программой. При этом он основывается на предположении, что ветвь, которая была пройдена, будет использоваться снова, и загружает соответствующую команду перехода на конвейер. В случае если это предсказание верно, переход осуществляется без задержки. Для того чтобы судить об эффективности этого нововведения, достаточно отметить, что вероятность правильного предсказания составляет около 80 %.

Рассмотрим, как эти и некоторые другие устройства, реализованные в обобщенной схеме следующего поколения микропроцессоров фирмы Intel P6 (Pentium Pro/II/III). Для иллюстрации используем рис. 4.2.

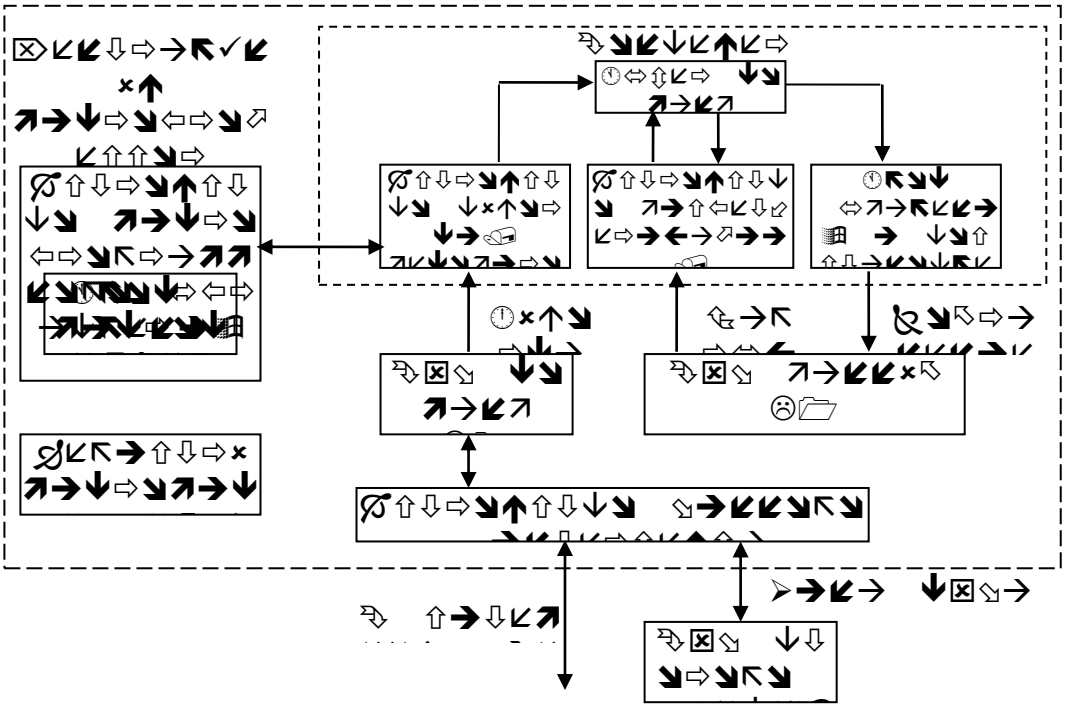


Рис. 4.2 - Обобщенная схема микропроцессора семейства P6 (Pentium PRO/II/III)

На рисунке видно разделение кэш-памяти на две части — для кода и для данных. Это обеспечивает бесперебойную поставку машинных инструкций и элементов данных на конвейер микропроцессора. Исходные данные для кэш-памяти первого уровня обеспечивает кэш-память второго уровня. Заметьте, что информация из нее поступает на устройство шинного интерфейса и далее в соответствующую кэш-память первого уровня по 64-битной шине. При этом благодаря более быстрому обновлению содержимого кэш-памяти первого уровня обеспечивается высокий темп работы микропроцессора. *Устройство выборки/декодирования* читает поток команд из кэша команд (L1) и декодирует их в последовательность микроопераций. Микрооперации представляют собой примитивные команды, которые выполняются пятью *исполнительными устройствами* микропроцессора, работающими параллельно. Многие машинные команды преобразуются в одиночные микрооперации (это делает простой декодер), а некоторые машинные команды преобразуются в последовательность от двух и более (оптимально - четырех) микроопераций (это делает сложный декодер). Информация о последовательности микроопераций для реализации конкретной машинной команды содержится в *блоке микропрограммного управления.*

Наиболее ценным свойством микропроцессорной архитектуры семейства P6 является реализация механизма интеллектуальной обработки потока команд, называемого «динамическим

выполнением». Этот механизм основывается на следующих свойствах, некоторые из них уже существовали в прежних моделях микропроцессоров. Перечислим их:

- *Предсказание переходов, в том числе вложенных.* В микропроцессорах ряда P6 такая технология реализуется устройством выборки/декодирования (см. рис. 4.2). Основная задача механизма предсказания — исключить перезагрузку конвейера.
- *Динамический анализ потока данных.* Анализ проводится с целью определения зависимостей команд программы от данных и регистров микропроцессора с последующей оптимизацией выполнения потока команд. Главный критерий здесь — максимально полная загрузка конвейера. Требование соблюдения данного критерия позволяет даже нарушать исходный порядок следования команд при поступлении на конвейер. Сбоя при этом не будет, так как внешне логика работы программы будет сохранена. Подобная внутренняя неупорядоченность исполнения команд позволяет держать конвейер загруженным даже в то время, когда данные в кэш-памяти второго уровня отсутствуют и необходимо тратить время на обращение за ними в оперативную память.
- *Интеллектуальное исполнение.* Это свойство характеризует способность микропроцессора реализовать неупорядоченное исполнение команд, восстановив впоследствии исходный порядок команд и организовав передачу результатов работы команд в порядке, предусмотренном исходным алгоритмом. Данная возможность обеспечивается разделением устройства выборки и исполнения команд и устройства формирования результата (см. рис. 4.2). Все промежуточные результаты работы команд во время их исполнения (нахождения их на конвейере) размещаются во временных регистрах. Блок удаления и восстановления постоянно просматривает буфер команд и ищет те из них, которые уже исполнены и не имеют связи по данным с другими командами или не находятся в ветвях незавершенных переходов. Когда такие команды найдены, устройство удаления и восстановления результатов помещает сформированные ими данные в память или регистры микропроцессора в порядке, заданном исходным алгоритмом. После этого команды удаляются из конвейера.

Таким образом, реализация динамического исполнения команд позволяет организовать наиболее оптимальное прохождение команд программы через исполнительное устройство микропроцессора. А если учесть то, что в микропроцессорах семейства P6 команды исполняются в три потока одновременно (одновременно выполняются 3 команды), то становятся понятными все преимущества такого подхода. Выше мы уже отмечали то, что конвейер микропроцессоров семейства P6 имеет принципиальное отличие от конвейеров i486 и Pentium. Перечисленные выше три концепции представляют собой основу работы этого конвейера.

Рассмотрим следующие поколения микропроцессоров, с точки зрения новых внедренных технологий.

Рассмотрим микропроцессор Pentium 4, с микроархитектурой NetBurst.

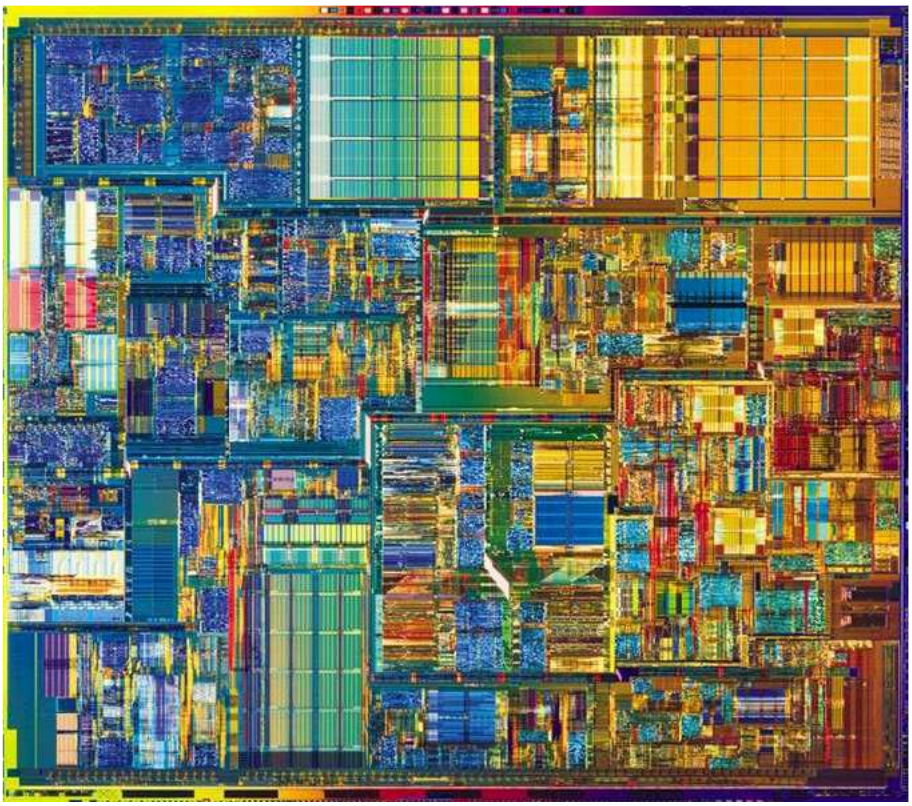
В микропроцессор была добавлена технология *Hyper-threading (*Hyper-threading (HT) — Гиперпоточность). Процессоры Pentium 4 (с одним физическим ядром) с включённым Hyperthreading операционная система определяет, как два разных процессора вместо одного. В процессорах с использованием этой технологии каждый физический процессор может хранить состояние сразу двух потоков, что для операционной системы выглядит как наличие двух логических процессоров (Logical processor).

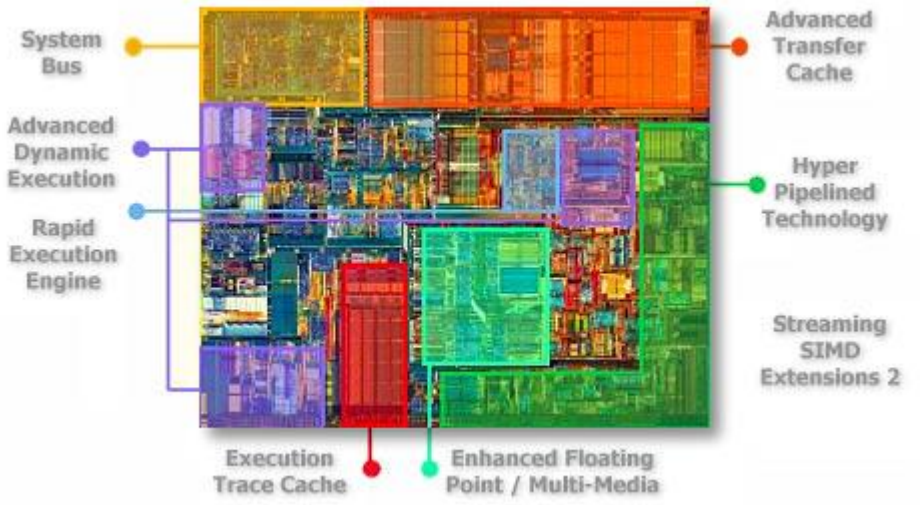
Физически у каждого из логических процессоров есть свой набор регистров и контроллер прерываний (APIC), а остальные элементы процессора являются общими. Когда при исполнении потока одним из логических процессоров возникает пауза (в результате кэш-промаха, ошибки предсказания ветвлений, ожидания результата предыдущей инструкции), то управление передаётся потоку в другом логическом процессоре. Таким образом, пока один процесс ждёт, например, данные из памяти, вычислительные ресурсы физического процессора используются для обработки другого процесса. По словам Intel, это позволило увеличить производительность на 15—30 %.

Кроме этого:

- Реализована технология Hyper Pipeline с технологией конвейера в 20 ступеней (в P6 использовался 12-ступенчатый конвейер), благодаря которой существенно выросла частота процессора - от 1,4 ГГц (3.2 ГГц в Pentium 4 Extreme Edition),
- Увеличен кэш второго уровня (L2) – 512 KB и L3 - до 2MB (Pentium 4 Extreme Edition), оба кэша работают на тактовой частоте процессора и позволяет осуществлять передачу данных с другим кэш L1, со скоростью 48 ГВ/сек, что в два раза быстрее, нежели у Pentium III.
- Введен кэш для микрокоманд, объемом 12000 микрокоманд.
- Набор потоковых инструкций SSE2 дополнен 144-мя новыми командами, в том числе 128-разрядными целочисленными командами и 128-разрядными командами для вычислений с плавающей точкой.
- Увеличена пропускная способность, между процессором и контроллером памяти, в 3,2 ГВ/сек (Pentium III составляет максимум 1,06 ГВ/сек).

На следующих рисунках показаны ядра процессора Pentium 4.





Недостатки. Работа процессоров на высоких частотах связана с высоким тепловыделением. Несмотря на то, что процессоры Pentium 4 на ядре Cedar Mill были способны работать на частотах, превышавших 7 ГГц, с использованием экстремального охлаждения (обычно использовался

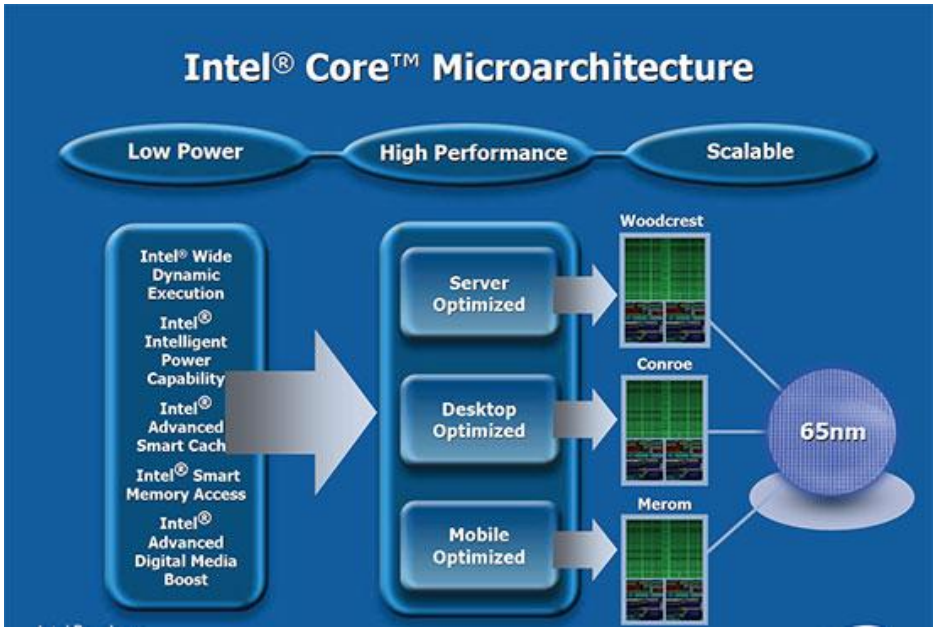
стакан с жидким азотом), максимальная тактовая частота серийных процессоров Pentium 4 составила 3800 МГц. При этом типичное тепловыделение превышало 100 Вт, а максимальное — 150 Вт. Из-за невозможности дальнейшего наращивания тактовой частоты, компания Intel была вынуждена предложить иной способ повышения производительности. Этим способом стал переход от одноядерных процессоров к многоядерным.

Двуядерные процессоры архитектуры NetBurst для настольных компьютеров (Pentium D), представляли собой два ядра на одном кристалле, находящиеся в одном корпусе (по сути, два отдельных процессора в одном корпусе). Обмен данными между ядрами осуществлялся через оперативную память, что приводило к потерям производительности. Поэтому Intel перешел на новую технологию *Core* для многоядерных микропроцессоров.

Микроархитектура Intel Core является многоядерной микропроцессорной архитектурой, представленной фирмой Intel в 2006 году. Она имеет несколько ядер и аппаратную поддержку виртуализации (Intel VT), а также поддерживает Intel 64 (64-битный режим) и SSE3 (дополнительный набор команд).

Процессоры для серверов и рабочих станций производятся под маркой Xeon, а для пользователей настольных и мобильных ПК — как Core 2.

Микроархитектура Intel Core имеет длину исполнительного конвейера 14 ступеней (Pentuim 4 „Prescott” - 31 ступень). Каждое ядро микропроцессора может получать, обрабатывать, исполнять и отбрасывать до четырёх полных команд одновременно (NetBurst только трех команд). Были внедрены 128-битные внутренние шины, что обеспечило однотактное исполнение 128-разрядных векторных инструкций (регистры XXM).



В микроархитектуру Intel Core была внедрена совокупность технологий, получивших общее название Intel Wide Dynamic Execution. К наиболее важным технологиям из набора Intel Wide Dynamic Execution относятся следующие:

- Advanced Smart Cache - новая архитектура оптимизирована под двухъядерную архитектуру процессора. Основной кэш первого уровня L1 связан с общей для обоих ядер динамически распределяемой кэш-памятью второго уровня L2 (данные, содержащиеся в L1, обязательно содержатся и в L2) для достижения максимальной производительности. Пиковая скорость обмена - 96 GB/sec (при частоте ядра в 3 GHz).
- Технология Macro Fusion заключается в слиянии двух x86-инструкций (и нескольких микроопераций - micro-ops fusion) в одну. В предыдущих версиях процессорной микроархитектуры каждая инструкция в формате x86 декодировалась независимо от остальных. При использовании технологии MacroFusion некоторые пары инструкций (например, инструкция сравнения и условного перехода) при декодировании могут быть слиты в одну микроинструкцию (micro-op), которая в дальнейшем будет выполняться именно как одна микроинструкция. Без технологии Macro Fusion за каждый такт

процессора могло декодироваться только четыре инструкции (в четырехканальном декодере), при наличии технологии MacroFusion в каждом такте могло считываться пять инструкций, которые за счет слияния преобразуются в четыре и подвергаются декодированию.

В июне 2009 компания объявила об упразднении многообразия вариантов данной торговой марки Core, например, Core 2 Duo, Core 2 Quad, Core 2 Extreme, в пользу трёх ключевых наименований: Core i3, Core i5 и Core i7 (позже Core i9).

Развитие этих микропроцессоров на основе новых микроархитектур, привело к их разделению на поколения:

- 1-ое поколение - микропроцессоры Core i3, Core i5 и Core i7 на основе микроархитектуры Nehalem (45 nm);
- 2-ое поколение - микропроцессоры Core i3, Core i5 и Core i7 на основе микроархитектуры Sandy Bridge (32 nm);
- 3-е поколение - микропроцессоры Core i3, Core i5 и Core i7 на основе микроархитектуры Ivy Bridge (22 nm);
- 4-е поколение - микропроцессоры Core i3, Core i5, Corei7 и Core i9 на основе микроархитектуры Broadwell (14 nm);
- 5-е поколение - микропроцессоры Core i3, Core i5, Corei7 и Core i9 на основе микроархитектуры Cannon Lake (10 nm).

Рассмотрим кратко новшества микроархитектуры Nehalem:

- Два, четыре или восемь ядер;
- Технология SMT (Simultaneous Multi-Threading), позволяющая исполнять одновременно два вычислительных потока на одном ядре (организация 2-х логических ядер из 1 физического);
- Три уровня кэш-памяти: L1 кэш размером 64 кбайта на каждое ядро, L2 кэш размером 256 кбайт на каждое ядро, общий разделяемый L3 кэш размером до 24 Мбайт;
- Интегрированный контроллер памяти с поддержкой нескольких каналов DDR3 SDRAM;
- Технологический процесс с нормами производства 45 нм;
- Возможность интегрирования в процессор графического ядра;
- Новая высокоскоростная шина QPI (Quick Path Interconnect) (вместо шины FSB) с топологией точка-точка для связи процессора с чипсетом и процессоров между собой;
- Технология Turbo Boost (*Turbo Boost* — Турбо Ускорение) — технология компании Intel для автоматического увеличения тактовой частоты процессора свыше номинальной, если при этом не превышаются ограничения мощности, температуры и тока в составе расчетной мощности (TDP - *thermal design power*).
- Модульная структура;
- новый набор расширений SSE4.2.

Следующая микроархитектура - ***Sandy Bridge***.

Основные новшества:

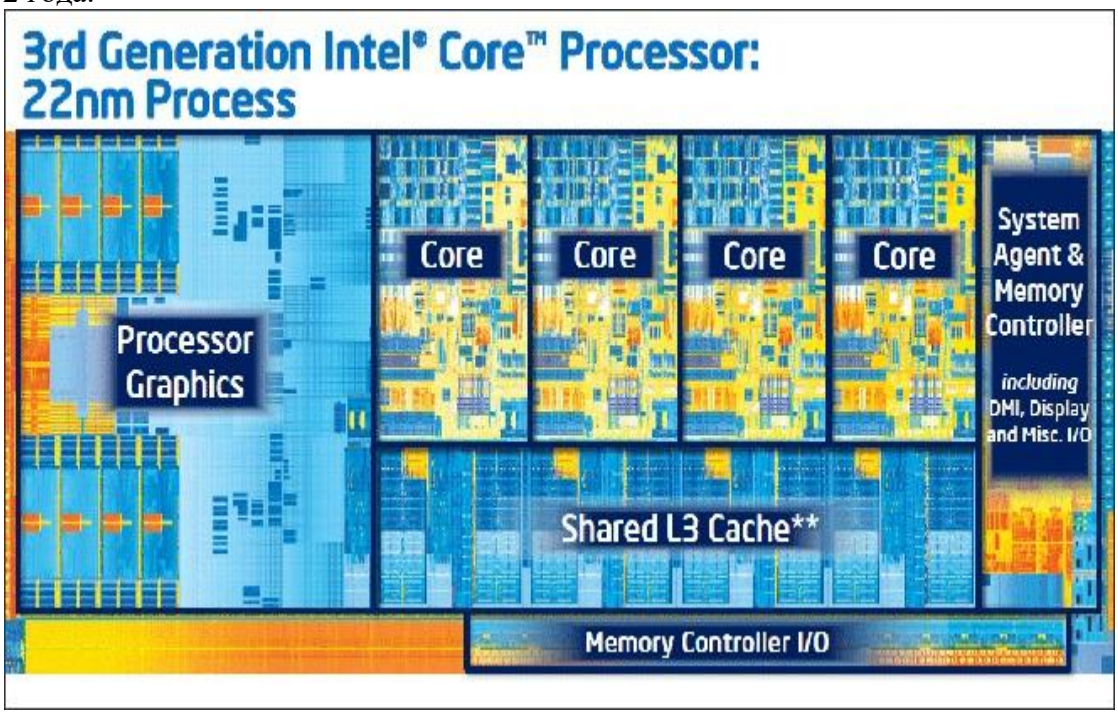
Структуру чипа Sandy Bridge можно условно разделить на следующие основные элементы:

- процессорные ядра;
- графическое ядро;
- кэш-память L3
- «Северный мост» (System Agent).

В процессорах Sandy Bridge с функцией vPro имеется возможность удаленного управления, например, удаленного блокирования ПК или стирания информации с Hard диска. Заявлено, что подобные функции полезны в случае кражи ПК. Команды могут быть переданы при помощи 3G, Ethernet, или другого подключения к сети Интернет.

Все перечисленные элементы объединены с помощью 256-битной межкомпонентной кольцевой шины, выполненной на основе новой версии технологии QPI. Используемый технологический процесс с нормами производства 32 нм.

Следующая микроархитектура Ivy Bridge. Ivy Bridge — кодовое название 22-нм версии микроархитектуры Sandy Bridge. Релиз первых процессоров на данной архитектуре состоялся в апреля 2012 года.



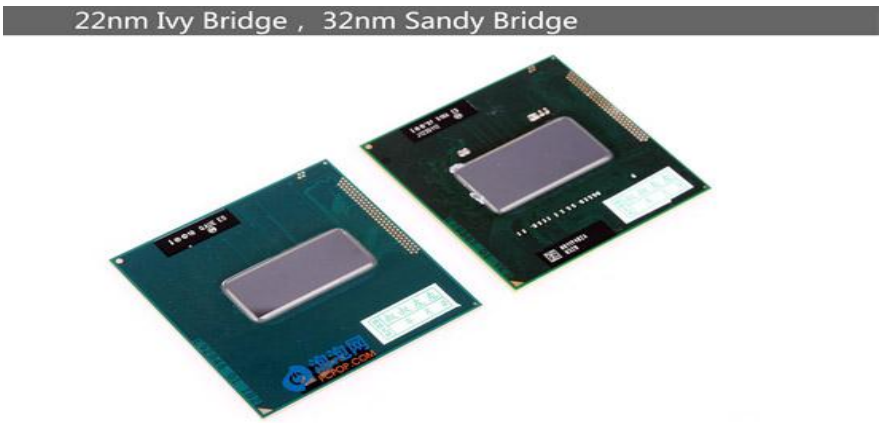
Особенности архитектуры:

- Новое видеоядро - HD Graphics 4000 -- появилась поддержка трёх независимых мониторов,
- Особенности встроенного GPU - 16 графических исполнительных блоков (EU, Execution Units),
- встроенный контроллер PCI Express 3.0 (до 8 ГТ/с).

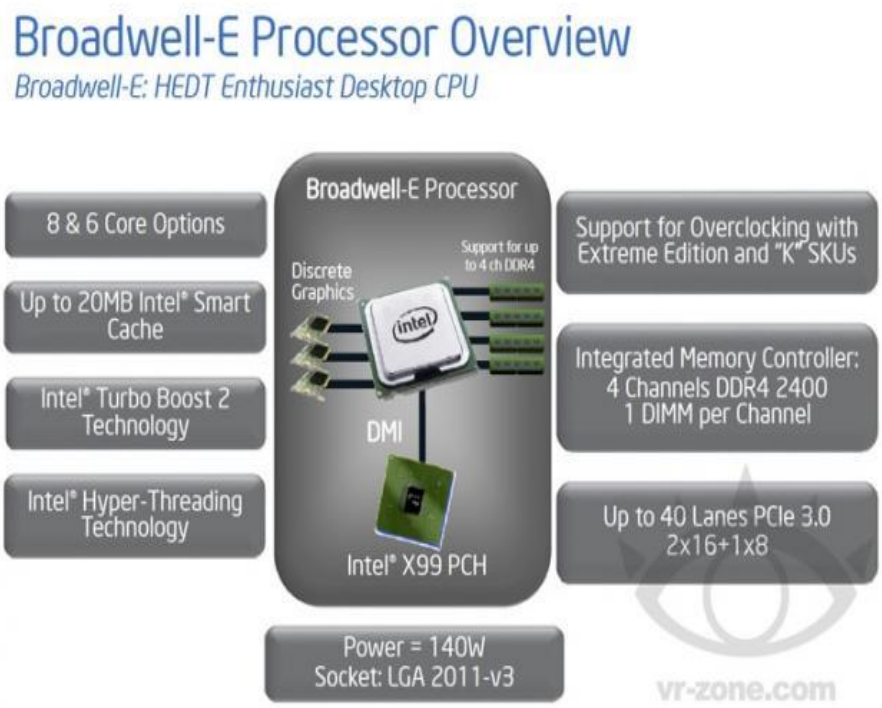
∉⋃ ⌊®◊□ □◊⟩⟩∑⧹®◊∑⌈◊□ ⌈⌊⍂ ⌊⟩⌋ ⍀ ⌊®⌟⨿ ⌈□⌊⧹∑⟩⟩⌊□⌊® ⍀∑ ⌈□∑®⌟⟩⋃ 77 ℜ⌋.

В сравнении технологий (Sandy Bridge и Ivy Bridge):

- От 5% до 15% - общее увеличение производительности.
- От 20% до 50% - увеличение производительности GPU.



На следующем рисунке представлены некоторые свойства микропроцессора Broadwell. Используемый технологический процесс с нормами производства 14 нм.



На следующем рисунке представлены сравнительные размеры микросхем микропроцессоров платформ Broadwell и Haswell.

5 Программная модель современных процессоров x86

Любая выполняющаяся программа получает в свое распоряжение определенный набор ресурсов микропроцессора. Эти ресурсы необходимы для выполнения и хранения в памяти команд программы, данных и информации о текущем состоянии программы и микропроцессора. Набор этих ресурсов представляет собой *программную модель микропроцессора.* Вначале рассмотрим программную модель 32 разрядных микропроцессоров, далее микропроцессоры с 64-битными расширениями (AMDx86-64 и Intel EM64T). Архитектура IA-64 (процессоры Itanium) в персональных компьютерах не используется и здесь не будем рассматривать. Схема, представленная на рис. 5.1, полностью соответствует программной модели 32 разрядных микропроцессоров Intel (Intel Architecture (IA)-32).

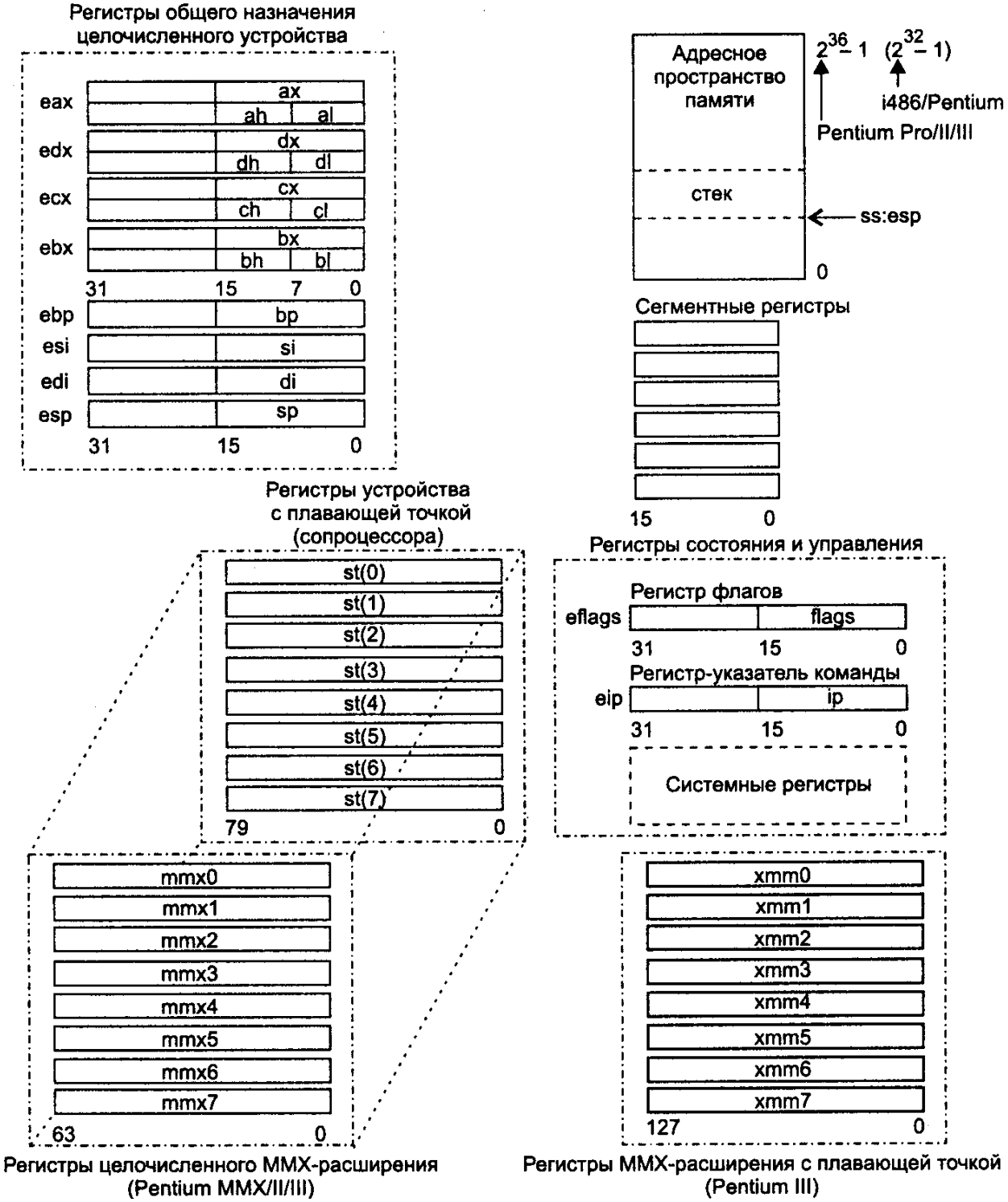


Рисунок 5.1 Программная модель микропроцессоров IA-32.

Программные модели более ранних микропроцессоров (i486, Pentium) отличаются меньшим размером адресуемого пространства оперативной памяти (2^{32} -1, так как разрядность их шины

адреса составляет 32 бита) и отсутствием некоторых групп регистров. Для каждой группы регистров в скобках обозначено, начиная с какой модели, данная группа регистров появилась в программной модели микропроцессоров Intel. Если такого обозначения нет, то это означает, что данная группа регистров присутствовала в микропроцессорах i386 и i486. Итак, программную модель микропроцессора IA-32 составляют:

- пространство адресуемой памяти (для Pentium IV — до 2^{36}- 1 байт);
- набор регистров для хранения данных общего назначения;
- набор сегментных регистров;
- набор регистров состояния и управления;
- набор регистров устройства вычислений с плавающей точкой (сопроцессора);
- набор регистров целочисленного MMX-расширения, отображенных на регистры сопроцессора (впервые появились в архитектуре микропроцессора Pentium MMX);
- набор регистров XMM-расширения с плавающей точкой (впервые появились в архитектуре микропроцессора Pentium III);
- программный стек. Это специальная информационная структура, работа с которой предусмотрена на уровне машинных команд. =

Теперь рассмотрим основные компоненты программной модели микропроцессора. Начнем с обсуждения регистров.

⊆◊⟨⌊□ □∑©⟩∫□⌊®

В программах на языке ассемблера регистры используются очень интенсивно. Большинство из них имеет определенное функциональное назначение. Как показано выше, программная модель микропроцессора имеет несколько групп регистров, доступных для использования в программах:

- регистры общего назначения eax/ax/ah/al, ebx/bx/bh/bl, edx/dx/dh/dl, ecx/cx/ch/cl, ebp/bp, esi/si, edi/di, esp/sp. Регистры этой группы используются для хранения данных и адресов;
- сегментные регистры cs, ds, ss, es, fs, gs. Регистры этой группы используются для хранения адресов сегментов в памяти;
- регистры сопроцессора st(0), st(l), st(2), st(3), st(4), st(5), st(6), st(7). Регистры этой группы предназначены для написания программ, использующих тип данных с плавающей точкой;
- целочисленные регистры MMX-расширения mmx0, mmx1, mmx2, mmx3, mmx4, mmx5, mmx6, mmx7;
- регистры XMM-расширения с плавающей точкой xmm0, xmm1, xmm2, xmm3, xmm4, xmm5, xmm6, xmm7;
- регистры состояния и управления — это регистры, которые содержат информацию о состоянии микропроцессора, исполняемой программы и позволяют изменить это состояние:
- регистр флагов eflags/f lags;
- регистр указатель команды eip/ip;
- системные регистры — это регистры для поддержания различных режимов работы, сервисных функций, а также регистры, специфичные для определенной модели микропроцессора. На схеме рис. 5. регистры этой группы не показаны по двум причинам: во-первых, их достаточно много, и, во-вторых, состав их может отличаться для различных моделей микропроцессора.

Регистры приведены с наклонной разделительной чертой, это части одного большого 32-разрядного регистра. Их можно использовать в программе как отдельные объекты. Зачем так сделано? Для обеспечения работоспособности программ, написанных для младших 16-разрядных моделей микропроцессоров фирмы Intel, начиная с i8086. Микропроцессоры i486 и Pentium имеют, в основном, 32-разрядные регистры. Их количество, за исключением сегментных регистров, такое же, как и у i8086, но размерность больше, что и отражено в их обозначениях, — они имеют приставку «e» (Extended).

Дополнительный сегмент данных.

Неявно алгоритмы выполнения большинства машинных команд предполагают, что обрабатываемые ими данные расположены в сегменте данных, адрес которого находится в сегментном регистре ds. Если программе недостаточно одного сегмента данных, то она имеет возможность использовать еще три дополнительных сегмента данных. Но в отличие от основного

сегмента данных, адрес которого содержится в сегментном регистре ds, при использовании дополнительных сегментов данных их адреса требуется указывать явно с помощью специальных префиксов переопределения сегментов в команде. Адреса дополнительных сегментов данных должны содержаться в регистрах es, gs, fs (extension data segment registers).

∠∑©()ƒ□⌡〉()ƒ(□⌠(□(⌈/□◊®⌊∑⌠(□

В микропроцессор включены несколько регистров (см. рис. 5.1), которые постоянно содержат информацию о состоянии, как самого микропроцессора, так и программы, команды которой в данный момент загружены на конвейер.

К этим регистрам относятся:

- регистр флагов eflags/flags;
- регистр указателя команды eip/ip.

Используя эти регистры, можно получать информацию о результатах выполнения команд и влиять на состояние самого микропроцессора. Рассмотрим подробнее назначение и содержимое этих регистров:

- *eflags/flags (flag register) — регистр флагов.* Разрядность eflags/flags — 32/16 бит. Отдельные биты данного регистра имеют определенное функциональное назначение и называются *флагами.* Младшая часть этого регистра полностью аналогична регистру flags для i8086. На рис. 5.2 показано содержимое регистра eflags.

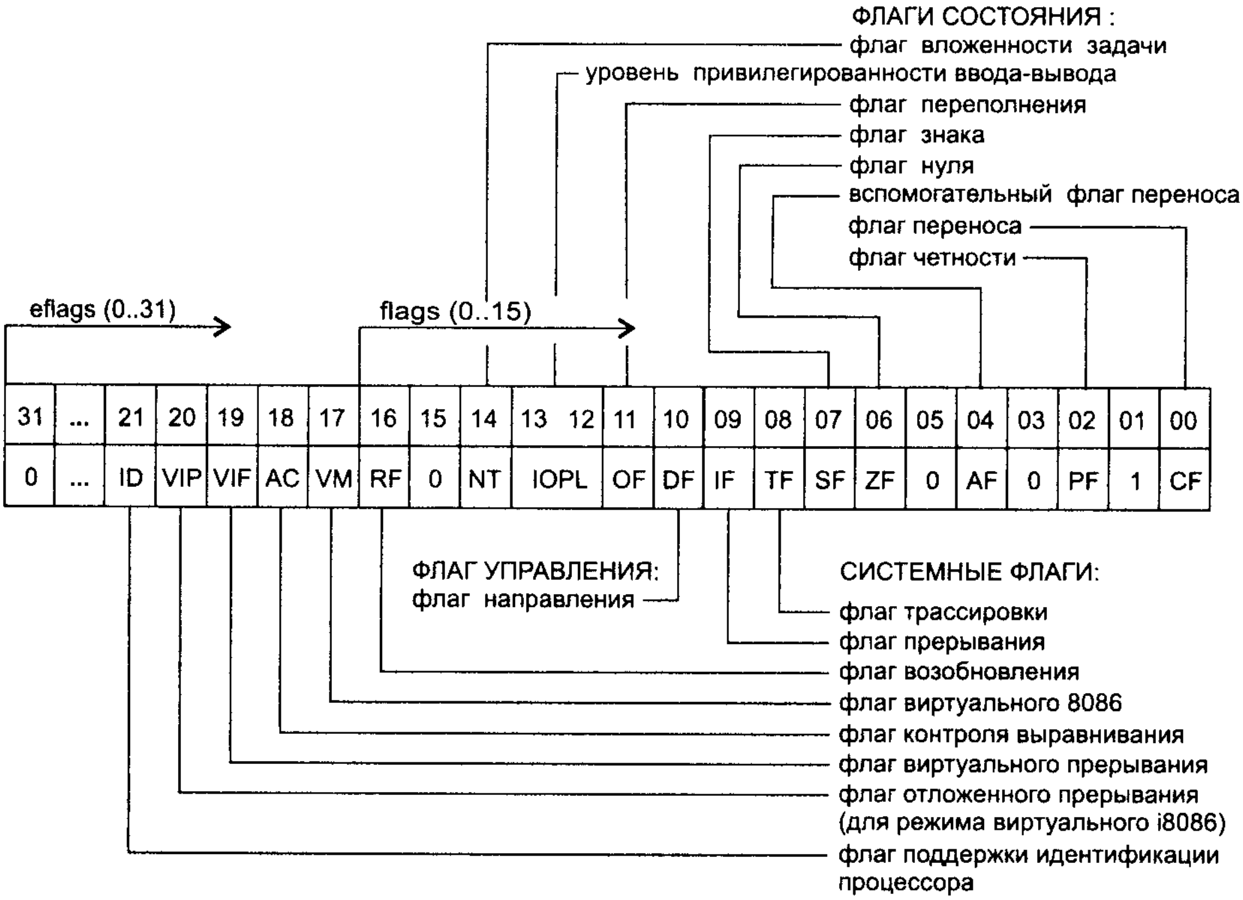


Рисунок 5.2 Содержимое регистра eflags.

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

- 8 *флагов состояния.* Эти флаги могут изменяться после выполнения машинных команд. Флаги состояния регистра eflags отражают особенности результата исполнения арифметических или логических операций. Это дает возможность анализировать состояние вычислительного процесса и реагировать на нее с помощью команд условных переходов и вызовов подпрограмм. В табл. 5.1 приведены флаги состояния и указано их назначение;
- 1 *флаг управления.* Обозначается как df (Direction Flag). Он находится в десятом бите регистра eflags и используется цепочечными командами. Значение флага df определяет

направление поэлементной обработки в этих операциях: от начала строки к концу (df = 0) либо, наоборот, от конца строки к ее началу (df = 1). Для работы с флагом df существуют специальные команды cld (снять флаг df) и std (установить флаг df). Применение этих команд позволяет привести флаг df в соответствие с алгоритмом и обеспечить автоматическое увеличение или уменьшение счетчиков при выполнении операций со строками;

- 5 *системных флагов,* управляющих вводом/выводом, маскируемыми прерываниями, отладкой, переключением между задачами и виртуальным режимом 8086. Прикладным программам не рекомендуется модифицировать без необходимости эти флаги, так как в большинстве случаев это приведет к прерыванию работы программы. В табл. 5.1 перечислены системные флаги и их назначение (начиная с tf).

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

Таблица 5.1 - Флаги состояния

Мнемоника	Флаг	Номер	Содержание и назначение флага в eflags
CF	Флаг переноса (Carry Flag)	0	1 - арифметическая операция произвела перенос из старшего бита результата. Старшим является 7-й, 15-й или 31-й бит в зависимости от размерности операнда; 0 — переноса не было
PF	Флаг паритета (Parity Flag)	2	1 - 8 младших разрядов (этот флаг — только для 8 младших разрядов операнда любою размера) результата содержат четное число единиц 0 - 8 младших разрядов результата содержат нечетное число единиц
AF	Вспомогательный флаг переноса (Auxiliary carry Flag)	4	Только для команд, работающих с BCD-числами. Фиксирует факт заема из младшей тетрады результата: 1 — в результате операции сложения был произведен перенос из разряда 3 в старший разряд или при вычитании был заем в разряд 3 младшей тетрады из значения в старшей тетраде: 0 — переносов и заемов в (из) 3 разряд(а) младшей тетрады результата не было
ZF	Флаг нуля (Zero Flag)	6	1 — результат нулевой; 0 — результат ненулевой
SF	Флаг знака (Sign Flag)	7	Отражает состояние старшего бита результата (биты 7, 15 или 31 для 8- 16- или 32-разрядных операндов, соответственно): 1 — старший бит результата равен 1;
OF	Флаг переполнения (Overflow Flag)	11	Флаг of используется для фиксирования факта потери значащего бита при арифметических операциях: 1 — в результате операции происходит перенос (заем) в (из) старшего, знакового бита результата (биты 7, 15 или 31 для 8-, 16- или 32-разрядных операндов, соответственно); 0 — в результате операции не происходит переноса (заема) в (из) старшего, знакового бита результата

IOPL	Уровень привилегий ввода-вывода (Input/Output	12, 13	Используется в защищенном режиме работы микропроцессора для контроля доступа к командам ввода-вывода, в зависимости от привилегированности задачи
NT	Флаг вложенности задачи (Nested Task)	14	Используется в защищенном режиме работы микропроцессора для фиксации того факта, что одна задача вложена в другую
TF	Флаг трассировки (Trace Flag)	8	Предназначен для организации пошаговой работы микропроцессора: 1 — микропроцессор генерирует прерывание с номером 1 после выполнения каждой машинной команды. Может использоваться при отладке программ, в частности отладчиками; 0 — обычная работа
IF	Флаг прерывания (Interrupt enable Flag)	9	Предназначен для разрешения или запрещения (маскирования) аппаратных прерываний (прерываний по входу INTR): 1 — аппаратные прерывания разрешены; 0 — аппаратные прерывания запрещены
RF	Флаг возобновлени	16	Используется при обработке прерываний от регистров отладки
VM	Флаг виртуального 8086 (Virtual	17	Признак работы микропроцессора в режиме виртуального 8086: 1 — процессор работает в режиме виртуального 8086 0 — процессор работает в реальном или защищенном режиме
AC	Флаг контроля выравнивани я (Alignment Check)	18	Предназначен для разрешения контроля выравнивания при обращениях к памяти. Используется совместно с битом *am* в систем-ном регистре cr0. К примеру, Pentium разрешает размещать команды и данные с любого адреса. Если требуется контролировать выравнивание данных и команд по адресам, кратным 2 или 4, то установка данных битов приведет к тому, что все обращения по некратным адресам будут возбуждать исключительную ситуацию
VIF	Флаг виртуального прерывания (Virtual	19	При определенных условиях (одно из которых — работа микропроцессора в V-режиме) является аналогом флага if. Флаг vif используется совместно с флагом vip. Флаг появился в микропроцессоре Pentium
VIP	Флаг отложенного виртуального прерывания (Virtual	20	Устанавливается в 1 для индикации отложенного прерывания. Используется при работе в V-режиме совместно с флагом vif. Флаг появился в микропроцессоре Pentium

ID	Флаг идентификации (IDentification flag)	21	Используется для того, чтобы показать факт поддержки микропроцессором инструкции cpuid. Если программа может установить или очистить этот флаг, то это означает, что данная модель микропроцессора поддерживает инструкцию cpuid

Программная модель микропроцессоров x86-64

Существуют понятия разрядности адреса и разрядности данных. Разрядность адреса определяет, сколько битов (16, 32 или 64) используется в регистрах, формирующих адрес данных или инструкций, расположенных в памяти. Каждому режиму работы процессоров соответствуют своя разрядность, применяемая по умолчанию. При необходимости для каждой исполняемой инструкции разрядность адреса или/и операнда может изменяться с помощью специальных префиксов (байтов перед кодом инструкции).

32-битные регистры процессоров позволяют непосредственно адресовать до 4 Гбайт памяти. Встроенный блок управления памятью поддерживает механизмы сегментации и страничной трансляции адресов.

Расширения x86-64 и EM64T в первую очередь предназначены для радикального увеличения объема адресуемой памяти: 64-битные регистры позволяют адресовать до 2^{64} = 18,4 x 10^{18} байт. Это число и является пределом объема виртуальной памяти 64-битного процессора, но пока используют только младшие 48 битов адреса.

Процессоры предоставляют четырехуровневую систему привилегий для защиты памяти, ввода-вывода и прерываний, а также механизм переключения задач для многозадачных ОС. Система команд процессоров постоянно расширяется при сохранении всех команд предшествующих процессоров x86. С расширением системы команд расширяется и набор архитектурных регистров (MMX, XMM, новые общие 64-битные регистры).

Процессоры могут работать в различных режимах, определяющих возможности адресации памяти и защиты: в реальном (16-разрядном) режиме процессора 8086, в режиме виртуального процессора 8086 (V86), в защищенном 32-разрядном (и защищенном 16-разрядном) режиме. Режим работы процессора задается операционной системой с учетом режима работы приложений (задач, task). У процессоров с 64-битным расширением появляются новые режимы, среди которых есть и режимы, обеспечивающие совместимость с 32-разрядными операционными системами и приложениями. Новые режимы используются только в 64-битных ОС, а полностью их преимущества доступны только 64-битным приложениям.

Режимы работы процессоров

32-битные процессоры могут работать в одном из следующих режимов:

- Режим реальной адресации (real address mode), или просто реальный режим (real mode), полностью совместим с 8086. В этом режиме возможна адресация до 1 Мбайт физической памяти.
- Защищенный режим виртуальной адресации (protected virtual address mode), или просто защищенный режим (protected mode). В этом режиме у процессора включаются механизмы сегментации и страничной трансляции. Механизм сегментации позволяет поддерживать виртуальную память объемом до 64 Тбайт. На практике используется только страничная трансляция, благодаря которой каждой задаче предоставляется до 4 Гбайт виртуального адресного пространства. По умолчанию и адреса, и операнды имеют разрядность 32 бита (возможен 16-разрядный защищенный режим в стиле процессора 80286). В защищенном режиме процессор может выполнять дополнительные инструкции, недоступные в реальном режиме; ряд инструкций, связанных с передачей управления, обработкой прерываний, и некоторые другие выполняются иначе, чем в реальном режиме.
- Режим виртуального процессора 8086 (Virtual 8086 Mode, V86) является особым состоянием задачи защищенного режима, в котором процессор функционирует как 8086 (16-битные адрес и данные). На одном процессоре в таком режиме могут параллельно исполняться

несколько задач с изолированными друг от друга ресурсами. При этом использование физического адресного пространства памяти управляется механизмами сегментации и трансляции страниц. Попытки выполнения недопустимых команд, выхода за рамки отведенного пространства памяти и разрешенной области ввода-вывода контролируются системой защиты. Более эффективен расширенный режим виртуального процессора 8086 (Enhanced Virtual 8086 Mode, EV86), в котором оптимизирована виртуализация прерываний.

- В режиме системного управления (System Management Mode, SMM) процессор выходит в иное, изолированное от остальных режимов пространство памяти. Этот режим используется в служебных и отладочных целях. С его помощью, например, скрытно выполняются функции управления энергопотреблением, эмулируются обращения к несуществующим аппаратным средствам (эмуляция клавиатуры и мыши PS/2 для USB).

Для процессоров x86-64 вышеперечисленные режимы объединены понятием *legacy mode (32bit)*; кроме того, появился новый режим *long mode* с двумя подрежимами:

- 64-битный режим (64-bit mode) — это режим полной поддержки 64-битной виртуальной адресации и 64-битных расширений регистров. В этом режиме используется только *плоская модель памяти* (общий сегмент для кода, данных и стека). По умолчанию разрядность адреса составляет 64 бита, а операндов (для большинства инструкций) — 32 бита, однако префиксом (REX) можно задать 64-битные операнды. Имеется новый способ адресации данных — относительно указателя инструкций. Режим предназначен для использования 64-битными ОС при запуске 64-битных приложений — он включается операционной системой для сегмента кода конкретной задачи;
- режим совместимости (compatibility mode) позволяет 64-битным ОС работать с 32- и 16-битными приложениями. Для приложений процессор выглядит как обычный 32-битный со всеми атрибутами защищенного режима, сегментацией и страничной трансляцией. 64-битные свойства используются только операционной системой, что отражается в процедурах трансляции адресов, обработки исключений и прерываний. Режим включается операционной системой для сегмента кода конкретной задачи.

32-битные ОС используют процессоры x86-64 только в режиме *legacy mode* (как обычный процессор IA-32).

Архитектурные регистры и типы данных

Процессоры могут оперировать с операндами разнообразных типов и размеров:

- целыми числами (со знаком и без знака) размером в байт, слово (16 бит), двойное слово (DWord, 32 бита), учетверенное слово (QWord, 64 бит) и двойное учетверенное слово (DQWord, 128 бит);
- строками байтов, слов, двойных и учетверенных слов;
- битами, битовыми полями и строками битов;
- числами в формате с плавающей точкой (FP) размером в 32, 64 и 80 бит.

Возможность работы с длинными операндами (64 и 128 бит) появилась в процессорах с расширениями MMX и XMM. Операнды инструкций могут находиться в регистрах процессора, памяти (или в порте ввода-вывода), а также в самой инструкции (непосредственный операнд). Наиболее эффективно процессор работает с операндами, расположенными в его регистрах. Состав регистров, с которыми работают прикладные программы, приведен на рис. 5.3.

Для обращений к памяти у процессоров x86 используются 16-битные сегментные регистры, с помощью которых формируется логический адрес:

- CS (Code Segment) — для адресации выбираемых инструкций;
- SS (Stack Segment) — для работы со стеком;
- DS, ES, FS и GS — для обращения к данным.

Названия 64-битных регистров начинаются с буквы R.

Помимо регистров общего назначения, предназначенных для использования прикладными программами, процессоры имеют ряд регистров системного назначения (на рисунке не показаны).

Эти регистры прикладными программами обычно не используются. К ним относятся системные адресные регистры, управляющие регистры, регистры отладки и тестирования. Ряд этих регистров являются модельно-специфическими (Model-Specific Registers, MSR), они пред-

назначены для управления расширениями отладки, мониторингом производительности, машинным контролем, кэшированием областей физической памяти и другими функциями. Их назначение привязывается к микроархитектуре конкретного процессора, состав меняется от модели к модели, доступ привилегирован. Доступность регистров различных групп зависит от режима работы процессора и уровня привилегий задачи.

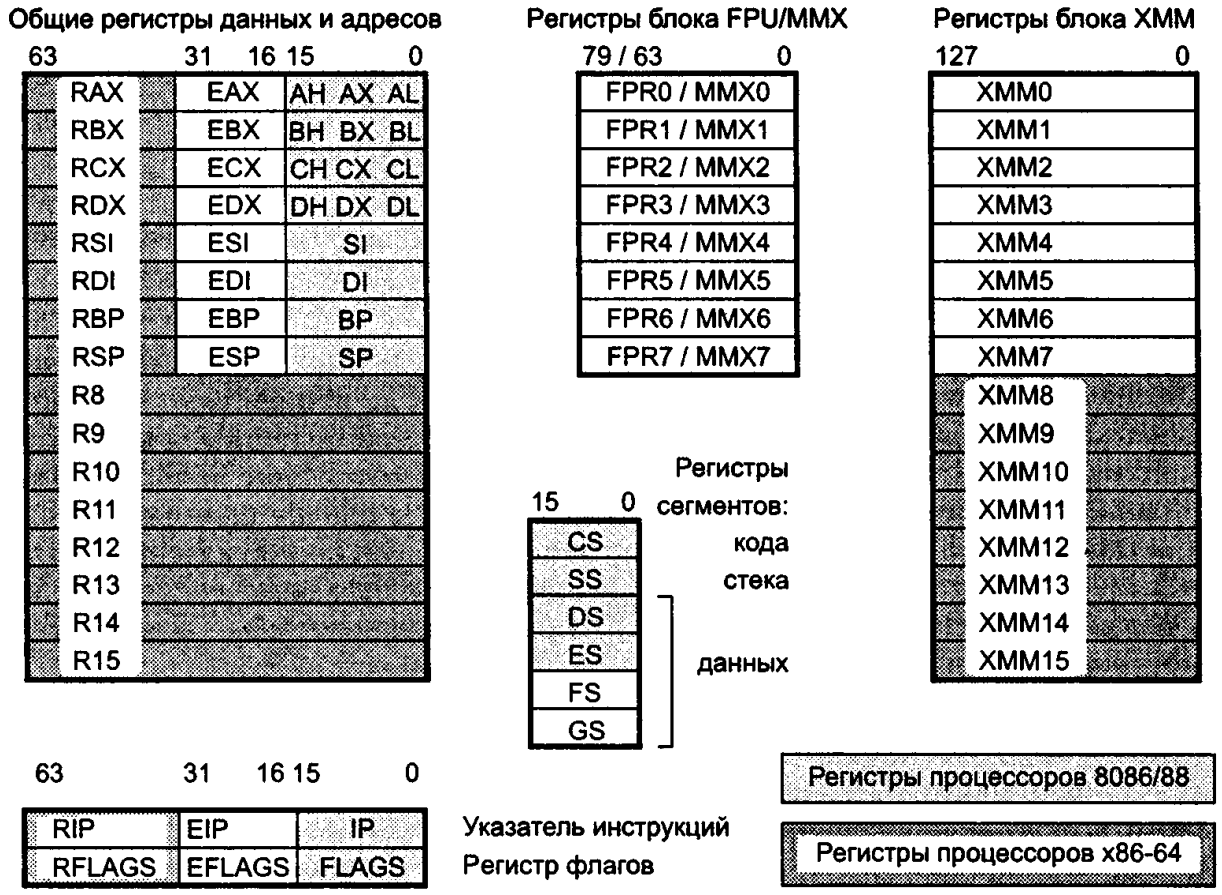


Рисунок 5.3 Прикладные регистры процессоров x86-64

Регистры общего назначения

В 32-битных процессорах были только 8 общих регистров, к которым можно было обращаться и как к 16-, и как к 32-битным (например, AX и EAX). Четыре регистра допускали и побайтное обращение к младшему или старшему байту (например, AL и AH). В 64-битном расширении добавили еще 8 общих регистров (R8...R15), и обращения к ним унифицировали: к любому из 16 общих регистров можно обращаться как к 64-, 32-, 16- или 8-битному регистру (всегда используются младшие биты).

Регистр флагов EFLAGS служит для хранения признаков результатов выполнения инструкций (знак, переполнение, ноль, и т. п.), в нем же расположен и флаг разрешения прерываний.

Регистр-указатель инструкции EIP соответствует «счетчику команд» фон-неймановской машины, он содержит смещение в сегменте кода текущей инструкции.

В 64-битном варианте RIP может использоваться и для относительной адресации данных.

Блок FPU

Блок FPU предназначен для расширения вычислительных возможностей центрального процессора — выполнения арифметических операций, вычисления основных математических функций (тригонометрических, экспоненты, логарифма) и т. д. В разных поколениях процессоров он назывался по-разному — FPU (Floating Point Unit — блок чисел с плавающей точкой) или NPX (Numeric Processor extension — числовое расширение процессора). Сопроцессор поддерживает семь типов данных: 16-, 32-, 64-битные целые числа; 32-, 64-, 80-битные числа с плавающей

точкой (FP-форматы) и 18-значные числа в двоично-десятичном (BCD) формате. Применение сопроцессора повышает производительность вычислений в сотни раз.

Блок MMX и расширение 3DNow!

Технология MMX ориентирована на приложения мультимедиа, 2D/3D-графику и коммуникации. Основная идея MMX заключается в одновременной обработке нескольких элементов данных за одну инструкцию — так называемая технология SIMD (Single Instruction — Multiple Data). Расширение MMX использует новые типы целочисленных данных, упакованных в 64-битных регистрах: восемь байтов, четыре слова, два двойных слова или одно учетверенное слово. Эти типы данных обрабатываются в 64-битных регистрах MMX0-MMX7, представляющих собой младшие 64 бита 80-битных регистров FPU.

Каждая инструкция MMX выполняет действие сразу над всем комплектом операндов (8, 4, 2 или 1), размещенных в адресуемых регистрах. Еще одна особенность технологии MMX — поддержка арифметики с насыщением (saturating arithmetic). Ее отличие от обычной арифметики с циклическим переполнением (wraparound mode arithmetic) заключается в том, что при возникновении переполнения в результате фиксируется максимальное возможное значение для данного типа данных, а перенос игнорируется. В случае переполнения снизу в результате фиксируется минимальное возможное значение. Граничные значения определяются типом (знаковый или беззнаковый) и разрядностью переменных. Такой режим вычислений удобен для обработки сигналов: для аудиосигнала насыщение не так сильно меняет его звучание, как эффект от циклического переполнения. Те же предпочтения имеют место и при определении цветов изображения. Совпадение регистров MMX и FPU накладывает ограничения на чередование кодов FPU и MMX — забота об этом лежит на программисте приложений с MMX.

Блок MMX (и поддержка соответствующих инструкций) имеется на всех современных процессорах, начиная с 6-го поколения. В расширении SSE набор целочисленных инструкций для блока MMX расширен.

Блок XMM и расширение SSE

Расширение SSE (Streaming SIMD Extensions — потоковые SIMD-расширения) предназначено для ускорения обработки больших потоков данных в формате с плавающей точкой. Потоковое расширение реализуется на аппаратном блоке XMM, в котором первоначально было восемь 128-битных регистров XMMO...XMM7. В 64-битном расширении число регистров XMM увеличили до 16. Регистры XMM позволяют работать с различными операндами:

- четырьмя вещественными числами одинарной точности (32-битные);
- парой вещественных чисел двойной точности (64-битные, только в SSE2);
- упакованными целыми числами: 16 байт, 8 слов, 4 двойных слова или пара учетверенных (по 64 бита) слов (только в SSE2).

Блок позволяет выполнять векторные (они же пакетные) и скалярные инструкции. Векторные инструкции реализуют операции сразу над всеми комплектами операндов. Скалярные инструкции работают с одним комплектом операндов — младшим словом. Помимо инструкций с новым блоком XMM в расширение SSE входят и дополнительные целочисленные инструкции с регистрами MMX, а также инструкции управления кэшированием. В процессоре Pentium 4 набор инструкций получил очередные расширения — SSE2, а позже и SSE3, касающиеся добавления новых типов 128-битных операндов для блока XMM. Использование блока XMM позволяет смешивать выполнение в режиме SIMD инструкций с целочисленными данными и с плавающей точкой.

В процессорах Pentium III появилось расширение SSE, а в Pentium 4 — расширение SSE2, предназначенное для 2D/3D-графики, кодирования/декодирования видео, а также шифрования данных. Позже появились и расширения SSE3… SSE4.2. Блок XMM и инструкции SSE - SSE4.2 используются и в современных процессорах AMD.

⊆⌊®⌋∑ ⌈⎝⎜ □⌊⌊⏐□⌊⎞∑〉〉⌊□⌋ – ⎨ ⌊®⌋∑⌉ ⌊⌈◊⎨™⌋. ℑ⌋⌊⎝ ®®∑™∑⎨ ⌋⌉ ⌊⌈◊⎨™⌋ ΑςΞ (Αδϖανχεδ ςεχτορ Εξτενσιονσ). ℜ ⎨∑⌉ ⌊⌡⌊□⌊⌈ 〉⌈⌋⎬⌊∑, ⎨◊〈⌊□⌉ ⌊⌈◊⎨™ ΑςΞ ⏐□∑™⟩∫◊®⌊□∑∫ 〉⌊〈⌊⌈ □◊〉⋀□∑⎨⎝∑ ⎨◊〈⌊□⌊®⌉ ⌊⌈◊⎨™ ΣΣΕ. ℜ ⏐◊〉⌈⎨ ⌊〉∫⎝, ΑςΞ □◊〉⋀□□∑∫ ⎝⎜⎨◊ ⏐◊⌋⌉⎨ ⌊ 128–〈⌡⎨ ⌋∑ □∑©⎝〉∫□⌋ ΣΣΕ ™⌊ 256 〈⌡. ⊆⌊®⌋∑ 256–〈⌡⎨ ⌋∑ □∑©⎝〉∫□⌋ ⌊〈⌊⌊⎨◊ ⏐◊⌋ ∫〉□⌉◊⌉ ψμμ0–ψμμ15 (⏐□⎝ ⎬∫⌊⌈ 128–〈⌡⎨ ⌋∑ ΣΣΕ □∑©⎝〉∫□⌋ ξμμ0–ξμμ15 〉

⟩⊥◊⌡⌡⟩□ ⌠◊ ⌈⌊◊™⁄⁄∑ 128 ⟨⁄⁄ ⟩⌊⌊⌈®∑⌡⟩® ⌈⌡⌉∑© ⌊ ΑςΞ □∑©⁄⌡□◊. ®◊, |◊⌈™◊□ ⌊⌈⟩⌡□ ⌈⌉ ⌉⌊□ ⌊⌈ ΣΣΕ (◊ ⌡◊ |⌈∑ ΣΣΕ2, ΣΣΕ3, ΣΣΣΕ3, ΣΣΕ4.1, ΣΣΕ4.2 ⌊ ΑΕΣ–ΝΙ) ⌊⌈∑∑⌡ ® ΑςΞ ⟩®⌈⌈ ◊⌊◊⌊⌊© ⟩ |□∑|⌊⌈ ⟩⌊⌈ ϖ.□

Набор инструкций (система команд).

Главная функция микропроцессора — выполнение заданного для него набора команд (*инструкций*). Набор инструкций современных процессоров x86 вбирает в себя инструкции всех предыдущих поколений. Инструкции можно разделить на прикладные, используемые «полезными» приложениями, и системные, используемые операционной системой для создание среды, в которой работают приложения.

Инструкции общего назначения — основные целочисленные инструкции x86, используемые практически всеми программами. Эти инструкции загружают, сохраняют и обрабатывают данные, расположенные в регистрах общего назначения и памяти. Часть инструкций служит для изменения последовательности исполнения — это инструкции условных и безусловных переходов, вызовы процедур. *Базовые инструкции* общего назначения реализованы во всех процессорах x86 (за исключением некоторых инструкций, наличие которых определяется через инструкцию CPUID). *Инструкции 64-битных режимов* (long mode instructions) появились только в процессорах с 64-битными расширениями и доступны лишь в соответствующих режимах.

Инструкции с плавающей точкой x87 работают с FPU, они используются в приложениях, требующих точных вычислений. Заметим, что эти инструкции поддерживают разные форматы данных с плавающей точкой (FP-чисел): 80 бит — расширенная точность, 64 бита — двойная точность, 32 бита — одинарная точность. Кроме того, поддерживаются двоично-десятичные форматы и выполняется преобразование форматов. 80-битный FP-формат обеспечивает максимальную точность и максимальный диапазон охватываемых чисел.

64-битные медиа-инструкции расширений оперируют с данными, расположенными в 64-битных регистрах MMX. Они выполняют целочисленные операции и вычисления с плавающей точкой в скалярном и векторном вариантах и предназначены для медиа-приложений, работающих с блоками данных. *Векторные целочисленные инструкции* появились в исходном наборе MMX.

128-битные медиа-инструкции оперируют данными, расположенными в 128-битных регистрах XMM. Они выполняют целочисленные операции и вычисления с плавающей точкой в скалярном (с одним комплектом операндов) и векторном (с несколькими комплектами, принцип SIMD) вариантах. Эти инструкции предназначены для высокопроизводительных медиа- и научных приложений, работающих с блоками данных. В потоковом расширении SSE определены инструкции с плавающей точкой одинарной точности (32 бит), в SSE2 введены инструкции двойной точности (64 бит), в SSE3 введено 13 дополнительных инструкций (к 144 существующим в SSE и SSE2), включая SIMD-инструкции с FP-числами двойной точности, целыми числами, а также инструкции управления памятью и кэшированием.

Ряд инструкций представляют собой мосты между вышеприведенными группами. Они обеспечивают пересылку данных между блоками MMX, XMM и общими регистрами. Ряд инструкций (как целочисленных, так и с плавающей точкой) можно исполнять как в блоках MMX (и FPU), так и в блоках XMM.

®(⁄⁄ ™◊⌠⌠⁄⁄

Понятие *типа данных носит* двойственный характер. С точки зрения размерности микропроцессор аппаратно поддерживает следующие основные типы данных (рис. 5.5).

- *Байт* — восемь последовательно расположенных битов, пронумерованных от 0 до 7, при этом бит 0 является самым младшим значащим битом.
- *Слово* — последовательность из двух байтов, имеющих последовательные адреса. Размер слова — 16 битов; биты в слове нумеруются от 0 до 15. Байт, содержащий нулевой бит, называется *младшим байтом,* а байт, содержащий 15-й бит — *старшим байтом.* Микропроцессоры Intel имеют важную особенность — младший байт всегда хранится по меньшему адресу. *Адресом слова* считается адрес его младшего байта. Адрес старшего байта может быть использован для доступа к старшей половине слова.

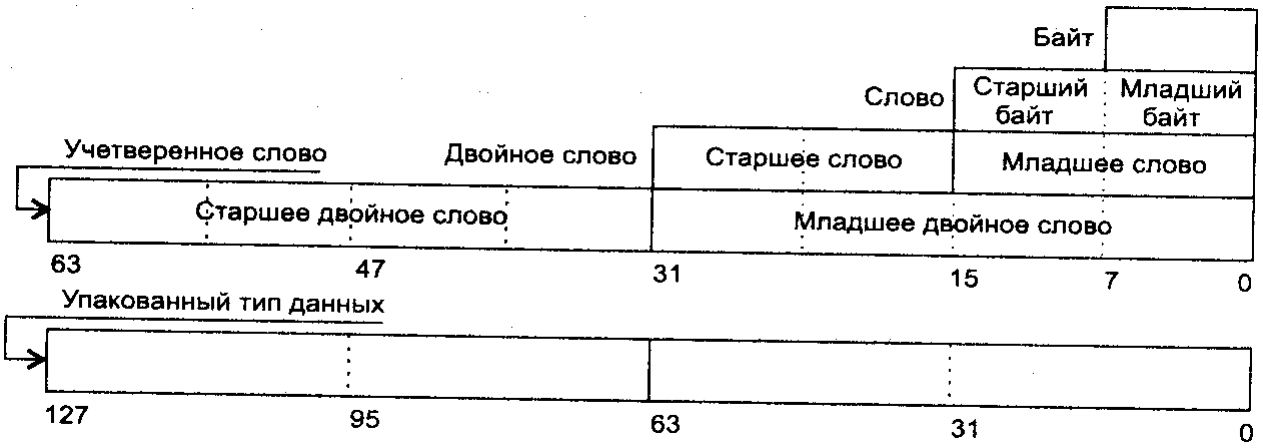


Рисунок 5.5 Основные типы данных микропроцессора.

- *Двойное слово* — последовательность из четырех байтов (32 бита), расположенных по последовательным адресам. Нумерация этих битов производится от 0 до 31. Слово, содержащее нулевой бит, называется *младшим словом,* а слово, содержащее 31-й бит, — *старшим словом.* Младшее слово хранится по меньшему адресу. *Адресом двойного слова* считается адрес его младшего слова. Адрес старшего слова может быть использован для доступа к старшей половине двойного слова.
- *Учетверенное слово* — последовательность из восьми байт (64 бита), расположенных по последовательным адресам. Нумерация битов производится от 0 до 63. Двойное слово, содержащее нулевой бит, называется *младшим двойным словом,* а двойное слово, содержащее 63-й бит, — *старшим двойным словом.* Младшее двойное слово хранится по меньшему адресу. *Адресом учетверенного слова* считается адрес его младшего двойного слова. Адрес старшего двойного слова может быть использован для доступа к старшей половине учетверенного слова.
- *128-битный упакованный тип данных.* Для работы с ним в микропроцессор введены специальные команды.

Кроме трактовки типов данных с точки зрения их разрядности, микропроцессор на уровне команд поддерживает *логическую* интерпретацию этих типов (рисунок 5.6).

- *Целый тип со знаком* — двоичное значение со знаком размером 8, 16 или 32 бита. Знак в этом двоичном числе содержится в 7, 15 или 31 бите соответственно. Ноль в этих битах в операндах соответствует положительному числу, а единица — отрицательному. Отрицательные числа представляются в дополнительном коде. Числовые диапазоны для этого типа данных следующие:
  - 8-разрядное целое – от -128 до+127;
  - 16-разрядное целое - от -32 768 до +32 767;
  - 32-разрядное целое — от -2^{31} до $+2^{31} - 1$.
- *Целый тип без знака* — двоичное значение *без знака,* размером 8, 16 или 32 бита. Числовой диапазон для этого типа следующий:

  байт — от 0 до 255;
  слово — от 0 до 65 535;
  двойное слово — от 0 до $2^{32} - 1$.

- *Указатель на память* бывает двух типов.

  • *Ближний тип* — 32-разрядный логический адрес, представляющий собой относительное смещение в байтах от начала сегмента. Эти указатели могут также использоваться в сплошной (плоской) модели памяти, где сегментные составляющие одинаковы.

  • *Дальний тип* — 48-разрядный логический адрес, состоящий из двух частей: 16-разрядной сегментной части — *селектора* и 32-разрядного смещения.

Рисунок 5.6 -Основные логические типы микропроцессора

- *Цепочка* представляет собой некоторый непрерывный набор байтов, слов или двойных слов максимальной длиной до 4 Гбайт.
- *Битовое поле* представляет собой непрерывную последовательность битов, в которой каждый бит является независимым и может рассматриваться как отдельная переменная. Битовое поле может начинаться с любого бита любого байта и содержать до 32 битов.
- *Неупакованный двоично-десятичный тип* — байтовое представление десятичной цифры от 0 до 9. Неупакованные десятичные числа хранятся как байтовые значения без знака по одной цифре в каждом байте. Значение цифры определяется младшим полубайтом.
- *Упакованный двоично-десятичный тип* представляет собой упакованное представление двух десятичных цифр от 0 до 9 в одном байте. Каждая цифра хранится в своем полубайте. Цифра в старшем полубайте (биты 4-7) является старшей.
- *Типы данных с плавающей точкой.* Сопроцессор имеет несколько собственных типов данных, несовместимых с типами данных целочисленного устройства.

Отметим, что «Зн» на рис. 5.6 означает знаковый бит.

6 ©|□◊®L∑{\∑ |◊ſ□∫⌋

6.1 ℜ\□∫ſ◊L⌉{◊□ |◊ſ□∫

Рассмотрим кратко несколько важнейших концепций, которые характеризуют современные поколения микропроцессоров семейства Intel.

Первая коцепция – разбивка всей области памяти микрокомпьютера на логические участки – *сегменты*.

Существуют два различных способа адресации памяти:

- Линейная адресация, которая предполагает последовательную адресацию памяти от адреса “0” до адреса 2^n-1, где n – разрядность шины адреса.
- Сегментная адресация, где память логически разделена на участки называемые сегментами, внутри которых адресация – линейная. Любая локализация в памяти происходит относительно базы сегмента.

Следующая коцепция – формирование некоторой *виртуальной памяти.*

Концепция *виртуальной памяти* основана на идеях отделения логической памяти пользователя от физической памяти и расширения логической памяти путем хранения ее образа на диске (файл

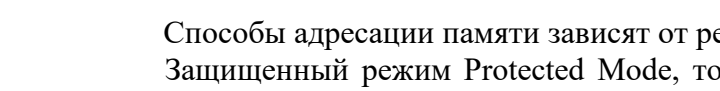
pagefile.sys). При исполнении программы только часть ее кода и данных, к которым происходит обращение, в каждый момент требует размещения в физической (RAM) памяти. Поэтому, естественно, возникает идея расширить пространство логической памяти, которое может быть реализовано намного большего размера, чем физическая память. Это и есть основной принцип организации виртуальной памяти.

Виртуальная память поддерживает совместное *использование* одного и того же адресного пространства более чем одним процессом, создание и исполнение облегченных процессов в общем пространстве виртуальной памяти. Это есть концепция – многозадачности (“multitasking”), которая определяется, как способность компьютера исполнять несколько задач (“task”) одновременно, это осуществляется последовательным переключением с одной последовательности команд некоторой задачи (“task”) к другой последовательности, другой задачи и т.д. В итоге переключается к первой последовательности.

Появляется механизм управления памятью. Он состоит в отображении виртуального адреса в физический адрес. Важным следствием механизма управления памятью является механизм *защиты*.

Защита обладает тремя аспектами:

- контроль информации (коды или другие типы данных);
- разделение пользователей один от другого (защита “inter-task”);
- разделение системного software от прикладного (защита “intra-task”);



Для экономии объёма памяти необходимой для выполнения текущей задачи применяется следующий принцип – *замена места задач* (“swapping task”), который состоит во временном отправлении задачи (или часть задачи) во внешнюю память, если данная задача (или часть) не находится в исполнении в данный момент.

Способы адресации памяти зависят от режима работы микропроцессоров.

Защищенный режим Protected Mode, точнее Protected Virtual Address Mode (защищенный режим виртуальной адресации), является основным режимом работы 32-разрядных процессоров. Для примера используем PentiumIV. В этом режиме процессор позволяет адресовать до 64 Тбайт виртуальной памяти, посредством которой при использовании механизма страничной адресации могут отображаться на 64 Гбайт физической памяти. С появлением 64-битных расширений размер виртуальной памяти увеличился до 2^{64} байт. Правда, в первых реализациях 64-битного процессора принято ограничение виртуального адреса до 48 бит, при этом физический адрес имеет длину 52 бит (возможно и меньше).

Режим виртуального процессора 8086 — Virtual 8086 Mode или V86 — является особым состоянием задачи защищенного режима, в котором процессор функционирует как 8086 с возможностью использования 32-разрядных адресов и операндов.

Защищенный режим предназначен для обеспечения независимости выполнения нескольких задач, что подразумевает защиту ресурсов одной задачи от возможного воздействия другой (под задачами подразумеваются как приложения, так и задачи операционной системы).

Основным защищаемым ресурсом является память, в которой хранятся коды, данные и различные системные таблицы (например, таблица прерываний), для доступа к которым имеется богатый выбор способов адресации (24 режима). Защищать требуется и совместно используемую аппаратуру, обращение к которой обычно происходит через операции ввода/вывода и прерывания. В защищенном режиме процессор аппаратно реализует многие функции защиты, необходимые для построения супервизора многозадачной ОС, в том числе механизм виртуальной памяти.

Память может логически организовываться в виде одного или множества сегментов произвольной длины (в реальном режиме — фиксированной). Помимо сегментации в защищенном режиме возможно (при страничной трансляции адресов) разбиение логической памяти на страницы размером 4 Кбайт (2, 4 Мбайт, 1 Гбайт), каждая из которых может отображаться на

любую область физической памяти. Сегментация и страничная трансляция адресов могут применяться совместно и по отдельности.

Применительно к памяти различают три адресных пространства: логическое, линейное и физическое. По сочетанию сегментации и страничной трансляции различают две модели памяти:

- В сегментной модели памяти приложение использует несколько сегментов памяти (для кода, данных, стека) и может переключать используемые сегменты. В этой модели приложение оперирует логическими адресами.
- В плоской модели памяти приложению для всех целей выделяется единственный сегмент. В этой модели приложение оперирует линейными адресами. Плоская модель гораздо проще и удобнее в обращении и используется в современных ОС.

Сегментация является средством организации логической памяти на прикладном уровне. Страничная трансляция адресов применяется на системном уровне для управления физической памятью.

Сегменты и страницы могут выгружаться из физической оперативной памяти на диск и по мере необходимости подкачиваться с него обратно в физическую память. Таким образом, реализуется виртуальная память. В качестве примера рассмотрим организацию виртуальной памяти в PentiumIV где используются сегментный и страничный механизмы.

6.2 Виртуальная память для PentiumIV

⊄⌊©⌉⌈∑⌋⌊⌈ ◊™□∑⟩ ®⌉□∫ ⌠◊⌊⌋⌇ ⌊©⌊ □∑⌠⌉⌠◊ (™◊⌊∑∑ – ⌊⌊©⌉⌈∑⌋⌊⌈ ◊™□∑⟩),⌈◊ ⌉⌊⌊©⌉⌈∑⌋⌊⌈ □∑◊ ⌊⌋⌇ ⌊©⌊ □∑⌠⌉⌠◊ ι8086, ⟩⌊⟩∫⌊⌈⌈⌉ ™® ⌠⌡ ⌈◊⟩∫∑⌡:

- эффективный адрес;
- селектор, который заменяет адрес сегмента;

⇑⌈ ⌈∑⌋ ∫⌉®⌇⌠ ◊™□∑⟩ □ ⟩∫⌊ ⟩⌠∑⌡∑⌇⌉∑ ® ⟩∑©⌠∑⌇∫∑, ⌊⌇ ⌊⟩⌈∑⌊⌋⌇ ⌊ ∑©⌊ ⟨◊⌈ ⌋.

Селектор находится в некотором сегментном регистре.

Локализация в сегменте происходит не прямо (как в случае i8086), а посредством косвенной адресации.

Рассмотрим структуру логического виртуального адреса (LA):

- Логический адрес состоит из 48-х бит:

$$\Lambda A = \alpha\delta\rho 48$$

- Эффективный адрес 32-х разрядный (AE):

$$EA = \alpha\delta\rho 32$$

Эффективный адрес задаёт максимальный размер сегмента:

$$2^{32}\,B = 4\,GB$$

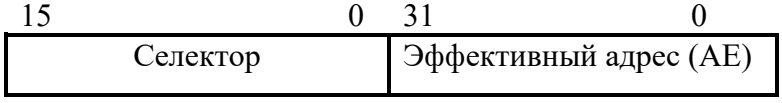
Нужно подчеркнуть, что размер сегмента может находиться в рамках от одного байта до 4GB.

Селектор (16 бит) состоит из следующих полей:

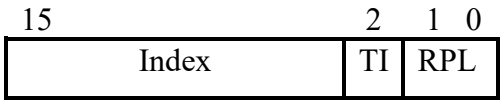
- поле Index – 13 бит;
- один бит “идентификатор таблицы” (TI);
- двухбитовое поле – уровень необходимого доступа (RPL).

∇∑⌊∑⌈ ∫⌊□ = IN∆EΞ↑TI↑PΠΛ

Структура виртуального (логического) адреса:



а)



б)

Рис. 6.1

На рисунке 6.1 представлена структура логического адреса (а) и структура селектора (б) PentiumIV.

Только первые два поля участвуют в определении некоторого сегмента в памяти (RPL применяется в механизме многоуровневого доступа и будет объяснён далее).

Поля INDEX и TI позволяют формировать в памяти максимум

2^{14}сегментов = 16k сегментов,

а размер виртуальной памяти будет:

2^{14}сегментов x 2^{32}В/сегмент = 2^{46} = 64 ТВ

Физический адрес PentiumIV состоит их 36-и бит:

ПА = αδρ36

т.е. физически адресуемая память: 2^{36}В =64 GB

Заметно, что объём виртуальной памяти намного превосходит объём физической памяти.

Виртуальная память логически разделена на две половины:

- при TI = 0 определяется" область глобальных адресов" где находятся операционная система, библиотеки процедур, компиляторы и т. д.
- при TI = 1 определяется "область локальных адресов" где находятся сегменты программы и данных для каждого процесса в отдельности.

Каждая область может максимально содержать 2^{13} сегментов=8К сегментов.

Поле INDEX используется для задания некоторого сегмента. Он служит как смещение в некоторой таблице в памяти, называемой таблица дескрипторов. Каждый дескриптор содержит физический адрес базы некоторого сегмента, а также его размер. Т.е. применяется метод косвенной адресации памяти для задания некоторого сегмента.

В микропроцессорах PentiumIV используется «страница» как единица логического разделения виртуальной памяти с фиксированным объёмом в 4 KB, 2MB или 4MB. Страничный механизм рассмотрим позже.

В защищенном режиме сегменты тоже распределяются операционной системой, но прикладная программа сможет использовать только разрешенные для нее сегменты памяти, выбирая их с помощью селекторов из предварительно сформированных таблиц дескрипторов сегментов.

6.3 Формирование физического адреса

Преобразование логического адреса в физический для 32-битных процессоров иллюстрирует рис. 6.2. Блок сегментации транслирует логическое адресное пространство в 32-битное пространство линейных адресов. Линейный адрес образуется сложением базового адреса сегмента с эффективным адресом. Базовый адрес сегмента в реальном режиме образуется умножением содержимого применяемого сегментного регистра на 16 (как и в 8086). В защищенном режиме базовый адрес загружается из дескриптора, хранящегося в таблице, по селектору, загруженному в используемый сегментный регистр.

Физический адрес памяти образуется после преобразования линейного адреса блоком страничной трансляции адресов. Он выводится на внешнюю шину адреса процессора. В простейшем случае (при отключенном блоке страничной трансляции) физический адрес совпадает с линейным.

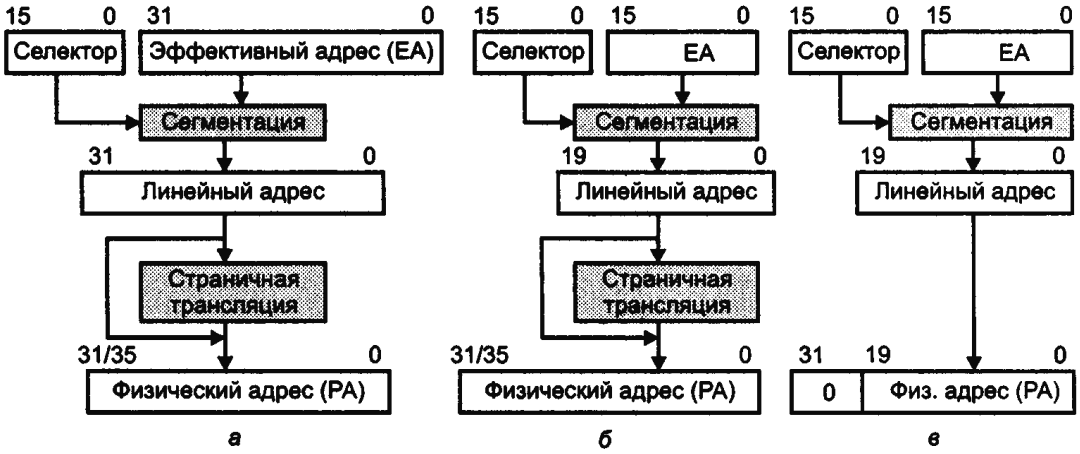


Рис. 6.2 Формирование адреса памяти в 32-битных процессорах: а — в защищенном режиме, б — в режиме V86, в — в реальном режиме

Включенный блок страничной трансляции адресов осуществляет трансляцию линейного адреса в физический страницами размером 4 Кбайт (2 или 4 Мбайт), размер страниц определяет операционная система каждой задаче. Блок страничной трансляции может включаться только в защищенном режиме.

В 64-битном режиме сегментация упразднена (рис. 6.3, а): приложения оперируют только линейными виртуальными адресами. В процессорах с 64-битными расширениями механизм сегментации оставлен только для режима совместимости (рис. 6.3, б). Эти адреса должны соответствовать канонической форме: их старшие биты, выходящие за предел поддерживаемой разрядности, должны быть нулевыми. В противном случае сработает исключение защиты. Из сегментных регистров процессор использует только регистры CS, FS и GS. В дескрипторе сегмента, на который указывает CS, задействуются лишь атрибуты: признак 64-битного режима, размер операнда по умолчанию и уровень привилегий. Регистры FS и GS требуются для нового режима адресации: в дескрипторе сегмента, на который они ссылаются, базовый адрес может применяться как смещение при вычислении адреса (эффективного, виртуального и линейного — теперь это одно и то же).

Блок страничной трансляции адресов позволяет использовать разрядность физического адреса, отличную от разрядности линейного адреса.

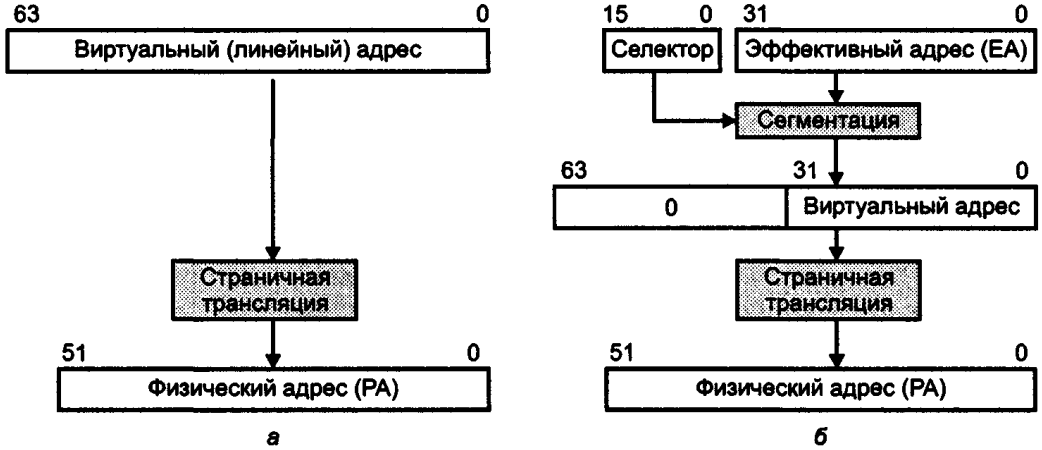


Рис. 6.3 Формирование адреса памяти процессоров с 64-битным расширением: а — в 64-битном режиме, б — в режиме совместимости

В микропроцессорах PentiumIV используется расширение физического адреса (PAE): 32-битный линейный адрес транслируется в 36-битный физический адрес (до 64 Гбайт физически адресуемой памяти).

В процессорах с 64-битным расширением на текущем этапе линейный адрес ограничен 48-битным пределом, а физический адрес может иметь разрядность до 52 бит. Использование «полноразмерного» (64-битного) линейного адреса потребовало бы слишком громоздких структур таблиц дескрипторов страниц.

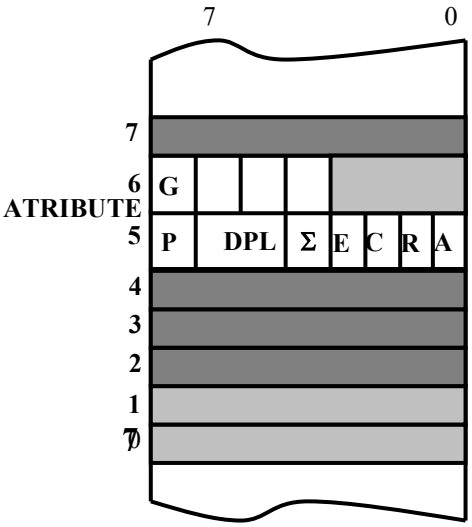
6.4 Дескрипторы и таблицы дескрипторов

Дескрипторы являются базовым элементом в механизме управления виртуальной памятью.

Существуют следующие типы дескрипторов:

a) Дескрипторы сегментов, которые определяют рабочие сегменты (программ и/или данных)
b) Дескрипторы управления, которые разделяются:
   1) Дескрипторы системных сегментов, сегменты которые микропроцессор использует в механизме управления памятью.
   2) Дескрипторы “сегментов состояния процессов” посредством которых реализуется многозадачный механизм.
   3) Дескрипторы “шлюзов“ (gate) - применяемых в различных механизмах: многоуровневая защита, многозадачный механизм, ответ на требование прерывания.

Для микропроцессоров PentiumIV обобщённая структура дескриптора представлена на рис. 6.4



∠\〉.6.4

Независимо от типа, дескриптор состоит из 8 байтов, где каждый байт имеет своё назначение.

Для дескрипторов сегментов:

- Байты 0 и 1 (и младшие 4 бита 6-го байта) – поле LIMIT (20 бит), задают размер сегмента.
- Байты 2, 3, 4 и 7-ой байт содержат физический адрес базы сегмента 32 бита (Base segment).
- Байт 5 и старшие 4 бита 6-го байта называется - поле управления доступом (Atribute) (12 бит)

и служит для уточнения типа дескриптора, содержит информацию о механизме защиты и др.

Например, биты 6, 5 — DPL (Descriptor Privilege Level) — атрибуты привилегий сегмента, бит дробности G (Granularity), определяющий, в каких единицах задан лимит: G=0 — в байтах, G=1 — в страницах по 4 Кбайт (что и обеспечивает максимальную длину в 4 Гбайт). В дескрипторах сегментов, отсутствующих в физической памяти (P=0), процессор контролирует только байт управления доступом. Назначение остальных байт определяется ОС.

Дескрипторы сегментов кода и данных определяют базовый адрес, размер сегмента, права доступа (чтение, чтение/запись, только исполнение кода или исполнение/ чтение), а для систем с виртуальной памятью еще и присутствие сегмента в физической памяти (рис. 6.5 и 6.6).

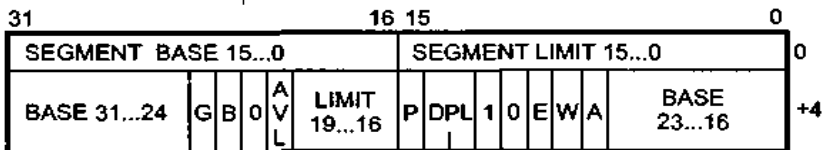


Рис. 6.5 Дескриптор сегмента данных

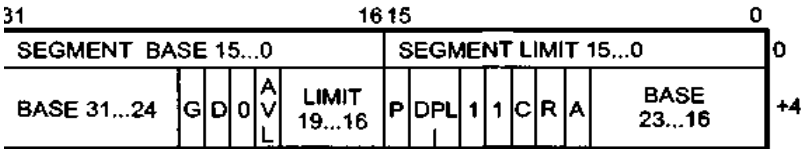


Рис. 6.6 Дескриптор сегмента кода

В байте управления доступом поля имеют следующее назначение:

- Бит 7 — P (Present) — присутствие в памяти. При P=1 сегмент отображен в физической памяти, при P=0 отображения нет, и поля базового адреса и лимита не используются.
- Биты 6, 5 — DPL (Descriptor Privilege Level) — атрибуты привилегий сегмента.
- Бит 4 — A (Accessed) — обращение. A=0 — к сегменту не было обращения, A=1 — селектор данного сегмента загружался в регистр сегмента или для него выполнялась команда тестирования.

Поля дескриптора сегмента данных (включая и стек) перечислены ниже.

- Бит 2 - E (Expand Down) — контролируемое направление расширения: E=0 — расширение вверх (смещение не должно превышать значения лимита), E=1 — расширение вниз (стек, у которого смещение должно превышать значение лимита).
- Бит 1 — W (Writeable) — разрешение (W=1) или запрет (W=0) записи данных в сегмент.

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

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

- Бит 2 — C (Conforming, подчиненность): при C=1 код может исполняться, если текущий уровень привилегий (CPL) не ниже уровня привилегий дескриптора (DPL); при C=0 (неподчиненный сегмент) управление к данному сегменту может передаваться только, если CPL=DPL
- Бит 1 — R (Readable) — разрешение (R=1) или запрет (R=0) чтения сегмента. Запись в сегмент кода возможна только через псевдоним (Alias) — сегмент данных с разрешенной записью, имеющий те же значения базы и лимита.
- Бит D (Default Operation Size) в предпоследнем байте определяет разрядность адресов и операндов по умолчанию: 0=0 — 16 бит, 0=1 — 32 бит.

Все дескрипторы объединены в таблицы дескрипторов. Существуют три типа таблиц дескрипторов — локальная таблица дескрипторов LDT (Local Descriptor Table), глобальная таблица дескрипторов GDI (Global Descriptor Table) и таблица дескрипторов прерываний IDT (Interrupt Descriptor Table). Размеры таблиц могут находиться в пределах 8 байт-64 Кбайт, что соответствует числу элементов в таблице от 1 до 8 Кбайт.

С каждой из этих таблиц связан соответствующий регистр процессора: Регистры GDTR, LDTR и IDTR.

Команды загрузки регистров таблиц LGDT, LIDT и LLDT являются привилегированными (выполняются только на уровне привилегий 0).

Глобальная таблица (GDT) содержит дескрипторы, доступные всем задачам. Она может содержать дескрипторы любых типов, кроме дескрипторов прерываний и ловушек. Локальная таблица (LDT) может быть собственной для каждой задачи и содержит только дескрипторы сегментов, шлюзы задач и вызовов. Сегмент недоступен задаче, если его дескриптора нет в текущий момент ни в GDT, ни в LDT.

Выбор таблицы (локальная или глобальная) определяется по значению бита TI селектора, а положение (номер) дескриптора задается 13-битным полем INDEX селектора.

Система привилегий

Четырехуровневая иерархическая система привилегий предназначена для управления использованием привилегированных инструкций и доступом к дескрипторам. Уровни привилегий нумеруются от 0 до 3, нулевой уровень соответствует максимальным (неограниченным) возможностям доступа и отводится для ядра операционной системы. Уровень 3 имеет самые ограниченные права и обычно предоставляется прикладным задачам. Систему защиты обычно изображают в виде концентрических колец, соответствующих уровням привилегий (рис. 6.7), а сами уровни привилегий иногда называют кольцами защиты. Сервисы, предоставляемые задачам, могут находиться в разных кольцах защиты. Передача управления между задачами контролируется шлюзами (Gate), называемыми также вентилями, проверяющими правила использования уровней привилегий. Через шлюзы задачи могут получить доступ только к разрешенным им сервисам других сегментов.

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

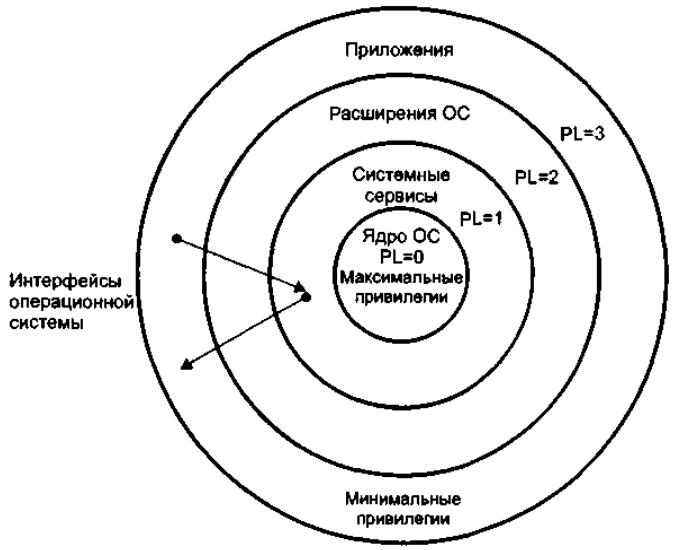


Рисунок 6.7 -Уровни привилегий

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

Механизм виртуальной памяти позволяет любой задаче использовать логическое адресное пространство размером до 64 Тбайт (16 К сегментов по 4 Гбайт). Для этого каждый сегмент в своем дескрипторе имеет специальный бит, который указывает на присутствие данного сегмента в оперативной памяти в текущий момент времени. Неиспользуемый сегмент может быть выгружен из оперативной памяти во внешнюю (например, дисковую), о чем делается пометка в его дескрипторе. На освободившееся место из внешней памяти может восстанавливаться содержимое другого сегмента (этот процесс называется swapping или подкачкой), и в его дескрипторе делается пометка о присутствии в памяти. При обращении задачи к отсутствующему сегменту процессор вырабатывает соответствующее исключение, обработчик которого и заведует виртуальной памятью в операционной системе. После подкачки сегмента (страницы) выполнение задачи продолжается, так что виртуализация памяти для прикладных задач прозрачна (если не принимать во внимание задержку, вызванную подкачкой).

Процессор предоставляет только необходимые аппаратные средства поддержки защиты и виртуальной памяти, а их реальное использование и устойчивость работы программ и самой операционной системы защищенного режима, конечно же, зависят от корректности построения ОС и предусмотрительности ее разработчиков.

6.5 Сегментное преобразование адреса

На рис. 6.8 показана трансляция виртуального адреса в физический адрес. Отмечены два механизма- сегментации и страничный. Рассмотрим отдельно преобразование логического адреса в физический (т.е. при отключенном блоке страничной трансляции).

Механизм отображения логического адреса на физическое адресное пространство начинается с двух полей селектора виртуального адреса:

- Поле «TI» идентифицирует тип таблицы;
- Поле «INDEX» умноженное на 8 (дескриптор состоит из 8 байт) определяет положение относительно базы таблицы дескрипторов.

Дескриптор содержит:

- Физический адрес базы сегмента (BASE) состоящий из 4 байта для Intel Pentium IV.
- Ограничение выбранного сегмента – 20 бит.

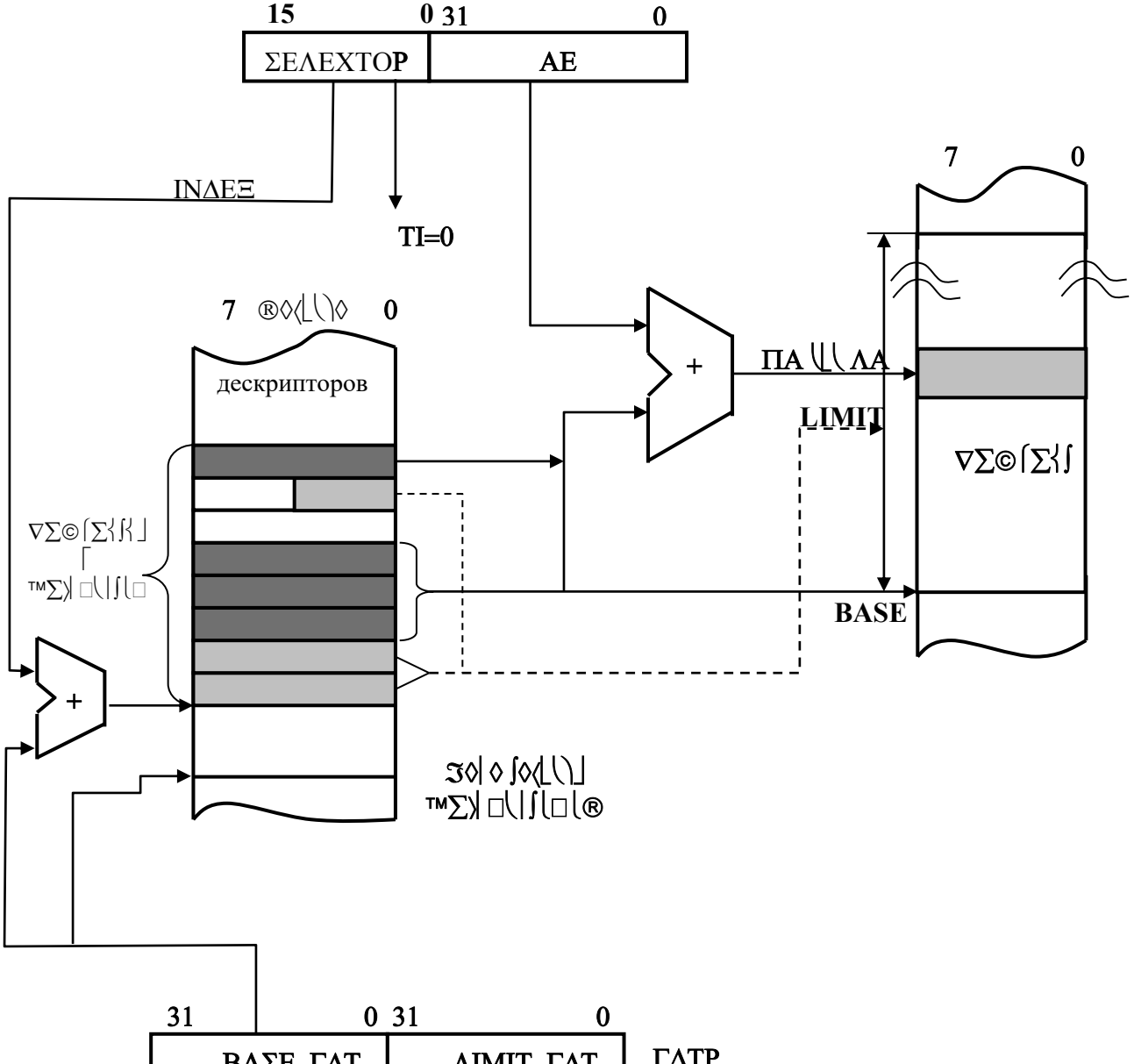
Физический адрес формируется как сумма базового адреса сегмента и исполнительного адреса.

Одновременно размер сегмента сравнивается с ограничением:

$$AE \le LIMIT$$

Локализация базы таблицы дескрипторов зависит от типа таблицы: в области глобальных или локальных адресов.

На рис.6.8 дана схема трансляции виртуального адреса (т. е. преобразование логического адреса в физический, при отключенном блоке страничной трансляции) при TI=0.



Определение таблицы глобальных дескрипторов (GDT) осуществляется более простым способом (т.к. существует только одна такая таблица): БАЗА (BASE) и ОГРАНИЧЕНИЕ (LIMIT) находятся во внутреннем регистре микропроцессора:" регистр таблицы глобальных дескрипторов" (GDTR). Доступ к этому регистру осуществляется при помощи привилегированных команд, при этом пользователь не имеет доступ к переопределению данной таблицы (GDT используется всеми процессами, включая операционную систему).

Проверяется если:

$$AF \leq БАЗА + ОГРАНИЧЕНИЕ$$

Для области локальных адресов механизм трансляции сложнее т.к. существуют несколько LDT. Применяется 2-х уровневая косвенная адресация.

6.6 Регистры" cache"

Механизм трансляции виртуальных адресов включает много операций и использует много времени. Для уменьшения времени используются скрытые регистры (cache).

На рисунке 6.9 представлены регистры cache микропроцессора Intel PentiumIV.

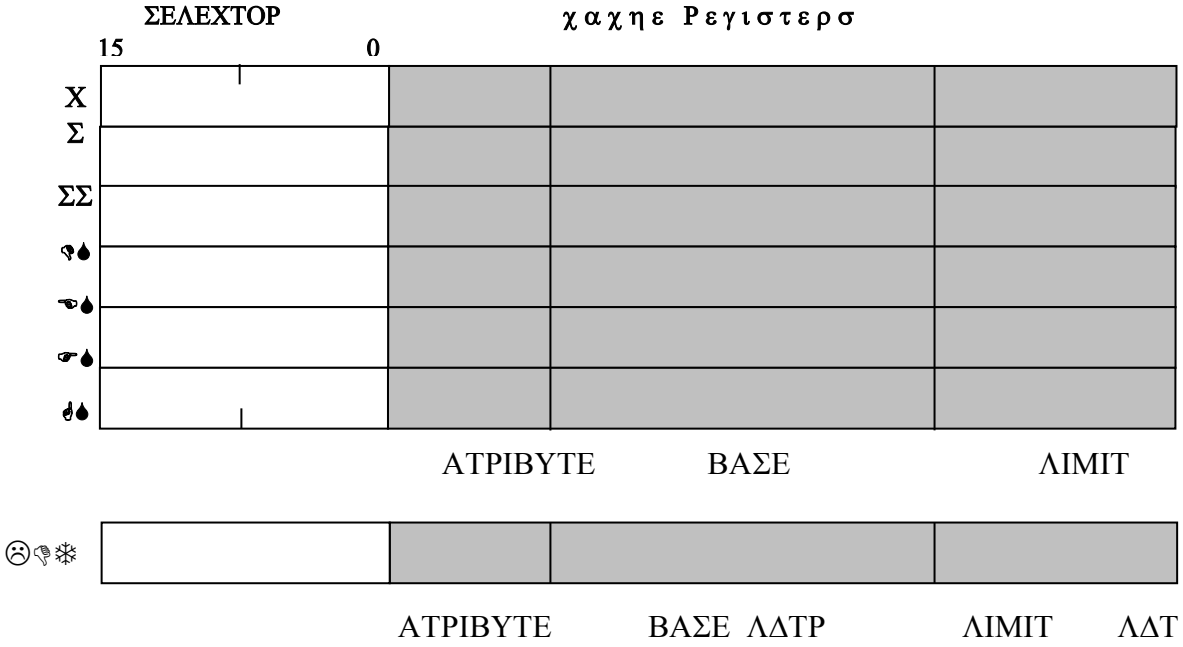


Рисунок 6.9

Можно заметит, что регистры сегментов кодов, стека, данных и дополнительных сегментов, а также и LDTR" удлинены" невидимо для пользователя. Здесь находится вся информация из дескриптора, т.е. физический адрес базы сегмента, ограничение, а также права доступа.

В сегментных регистрах можно записать 16 разрядный селектор. Теневая часть содержит 32 бита для адреса базы выбранного сегмента, 32 бита для задания размера сегмента (Поле Limit), из которых используются только 20 бит и поле для различных атрибутов сегмента.

Загрузка селектора в некоторый сегментный регистр, происходит следующими командами: POP, LDS, LES, MOV для SS, DS, ES и CALL, JMP загружается CS.

GDTR не имеет скрытой части и загружается привилегированной командой: LGDT.

6.7 Страничное управление памятью

Страничный механизм вводит новый тип адресов – линейный адрес. Механизм трансляции адресов приобретает вид:

VA (логический адрес) LA (линейный адреа)PA (физический адрес).

Для Intel линейный адрес совпадает с физическим адресом, если отключен страничный механизм.

Страничное управление памятью — это общепринятый (для разных семейств процессора) механизм организации виртуальной памяти с подкачкой страниц по запросу. Страничная трансляция адресов выполняется блоком управления памятью (Memory Management Unit, MMU), расположенным в процессоре, с использованием каталогов и таблиц дескрипторов страниц — структур данных в физической (оперативной) памяти. Страничная трансляция адресов приводит к тому, что непрерывная область линейных (и эффективных) адресов может отобразиться в виде разбросанных страниц физической памяти. Для того чтобы различные подсистемы компьютера могли программно общаться на «общем языке» линейных адресов, применяют локальные конструкции наподобие MMU. Для графических адаптеров используют таблицу GART (например, в порте AGP), для контроллеров шин (например, USB и FireWire) строятся специальные конструкции дескрипторов передач.

Блок MMU делит линейный адрес на виртуальные страницы фиксированного размера (4K, 4M, 2M). На такие же страницы делится и адресное пространство физических адресов. Часть страниц физического пространства занята ОЗУ, часть отображается на области памяти, назначенные периферийным устройствам (ввод-вывод, отображенный на память).

Каждая виртуальная страница может присутствовать в физической памяти (ОЗУ или в области памяти ПУ) или нет. Текущее описание страницы хранятся в дескрипторе страницы — структуре данных в ОЗУ. Дескрипторы страниц организуются в иерархии каталогов и таблиц, пример такой иерархии для «классических» страниц размером 4K приведен на рис. 6.10. На базу каталога страниц указывает управляющий регистр CR3 (доступ к нему привилегирован). Каждый уровень иерархии таблиц использует свой фрагмент линейного адреса. Самый младший фрагмент адреса (offset) является смещением внутри и виртуальной, и физической страниц (у них размеры обязательно совпадают). Элементы каталогов и таблиц страниц содержат физический базовый адрес (если таблица или страница, на которую они ссылаются, присутствует в физической памяти) или указание на местоположение элемента в файле подкачки. Атрибуты определяют присутствие в памяти, разрешение записи (R/W) и привилегии доступа (User/Supervisor, U/S), а также управляют кэшированием страниц. При использовании расширения физического адреса (Physical Address Extension, PAE), а также 64-битных режимов адресации появляется возможность защиты страниц с данными от ошибочного исполнения программного кода — бит NX (No Execute) в дескрипторах таблиц или страниц. Эта возможность присутствует не во всех моделях процессоров и может быть отключена. ОС может использовать данный атрибут для защиты от определенного класса вирусных атак (основанных на переполнении буфера).

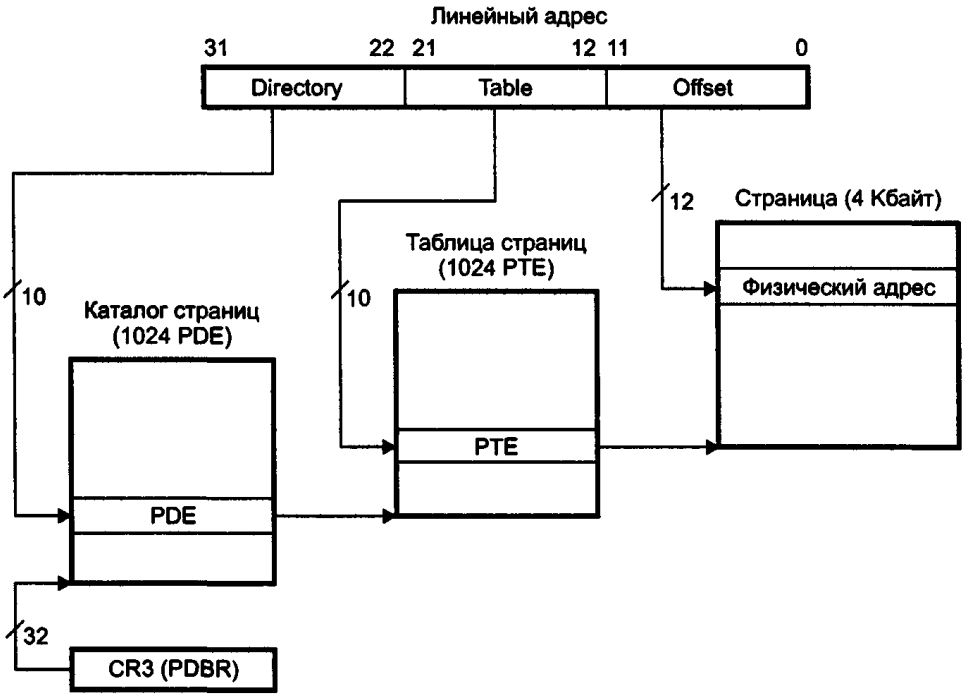


Рисунок 6.10 - Страничное преобразование адреса (для страниц 4К без PAE)

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

Каталог страниц размером 4 Кбайт содержит 1024 32-битных строки PDE (Page Directory Entry). Каждая строка (рисунок 6.11, а) содержит 20 старших бит адреса таблицы следующего уровня (младшие биты этого адреса всегда нулевые) и признаки (атрибуты) этой таблицы. Индексом поиска в каталоге страниц являются 10 старших бит линейного адреса (A22-A31).

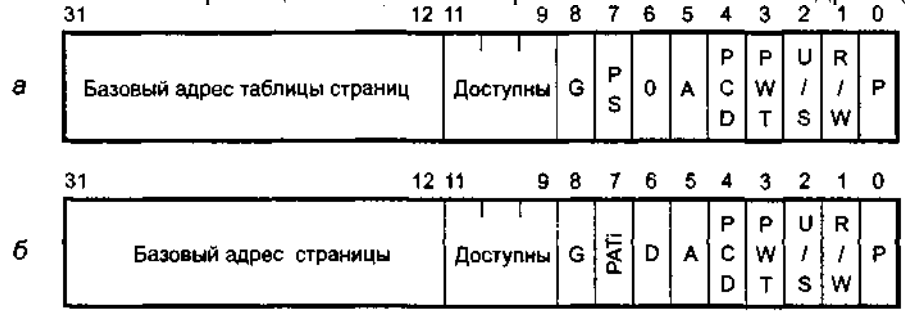


Рисунок 6.11 Структура 32-битных элементов страничного преобразования:
а — строка каталога (PDE), б — строка таблицы (PTE)

Каждая таблица страниц также имеет 1024 строки PTE (Page Table Entry) аналогичного формата (рисунок 6.11, б), но эти строки содержат базовый физический адрес (Page Frame Address) и атрибуты самих страниц. Индексом поиска в таблице являются биты A12-A21 линейного адреса.

Физический адрес получается из адреса страницы, взятого из таблицы, и младших 12 бит линейного адреса.

Строки каталога и таблиц имеют следующие биты атрибутов.

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

- A (Accessed) — признак доступа, который устанавливается перед любым чтением или записью по адресу, в преобразовании которого участвует данная строка.
- D (Dirty) — признак, который устанавливается перед операцией записи по адресу, в преобразовании которого участвует данная строка. Таким образом, помечается использованная — «грязная» страница, которую в случае замещения необходимо выгрузить на диск.

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

Поле OS Reserved программно использует ОС. Оно может хранить, например, информацию о «возрасте» страницы, необходимую для реализации замещения по алгоритму LRU (Least Recently Used — наиболее давно не использовавшаяся страница замещается первой).

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

Бит PS (Page Size) задает размер страницы (только в PDE). При PS=0 страница имеет размер 4 Кбайт, PS=1 используется в расширениях PAE и PSE (см. ниже). Механизм защиты страниц различает два уровня привилегий: пользователь (User) и супервизор (Supervisor). Пользователю соответствует уровень привилегий 3, супервизору — уровни 0, 1 и 2. Строки таблиц имеют атрибуты защиты страниц — биты U (User) и W (Writable — возможна запись); в некоторых описаниях те же биты называются U/S (User/Supervisor) и R/W (Read/Write). Эти атрибуты в строке каталога страниц относятся ко всем страницам, на которые ссылается данная строка через таблицу второго уровня. Атрибуты защиты в строке таблицы страниц относятся к конкретной странице памяти, которую она обслуживает. Защита на уровне страниц включается установкой бита WP (Write Protect) в регистре CR0, по аппаратному сбросу он обнуляется.

Дескрипторы страниц и каталогов находятся в памяти и занимают учетверенное слово (64 бит). Эти дескрипторы процессору приходится автоматически считывать, чтобы добраться до целевой ячейки памяти. Чтение нескольких слов (в самом тяжелом случае — четырех дескрипторов) на пути к искомому элементу — результат виртуализации памяти. Для сокращения затрат времени введено кэширование описателей страниц — буфер ассоциативной трансляции (Translate Look-aside Buffer, TLB). В буфере TLB (рис. 6.12) расположенном в процессоре, хранятся последние использованные описатели страниц и старшая часть виртуального адреса (все поля, левее поля смещения). Описатели в буфере ищутся ассоциативно: старшая часть линейного адреса сравнивается с адресами во всех элементах TLB. Если обнаруживается совпадение — hit, то трансляция продолжается чтением нужного адреса из TLB, не затрачивая время на последовательный поиск в таблицах. Обновление TLB происходит столько раз, сколько раз не находятся нужные элементы адреса (”cache miss”). Очевидно, что TLB полностью обновляется, когда в CR3 загружается другим базовым адресом справочника таблиц.

От объема и эффективности буфера TLB существенно зависит производительность процессора. В современных процессорах TLB для инструкций и данных разделяют, количество элементов увеличивают и ускоряют ассоциативный поиск. В процессорах x86 заполнение элементов TLB для программ прозрачно и неуправляемо, есть только специальные инструкции очистки элементов TLB.

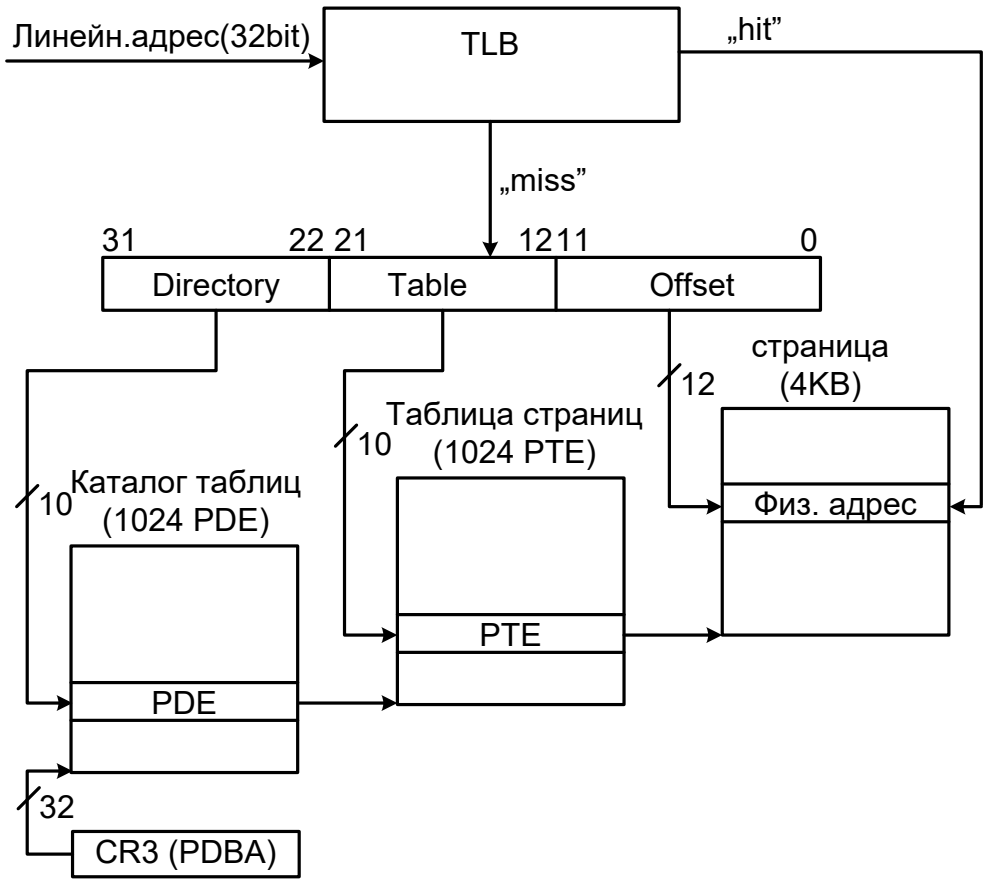


Рисунок 6.12

Производитель гарантирует, что в 98% обращений в память, использование страничного механизма заменяется простым чтением адреса из TLB.

7 Системные платы. Шины и интерфейсы

7.1 Архитектура системной платы

Одна из самых важнейших компонентов компьютера это систеная плата. *Системная* (system board, SB), или материнская (mother board, MB) плата — это часть компьютера, содержащая его основные электронные компоненты. С помощью материнской платы осуществляется взаимодействие между большинством устройств компьютера.

Конструктивно системная плата настольного компьютера представляет собой печатную плату площадью 100-150 см^2, на которой размещается большое число различных микросхем, разъемов и

других элементов. Главным несменяемым функциональным компонентом системной платы являлся набор системных микросхем (чипсет), который состоял из 2-х микросхем, называемых *северный* мост (North bridge) и *южный* мост (South bridge). На современных платах *северный* мост встраивается в микросхему микропроцессора (system agent). Обычно системные платы именуют по типу расположенного на ней чипсета.

Обычно в одну из микросхем набора входят также часы реального времени с CMOS - памятью и иногда — контроллер клавиатуры, контроллеры внешних устройств.

Рассмотрим несколько архитектур системных плат.

Шинно-мостовая архитектура

В шинно-мостовой архитектуре имеется центральная магистральная шина, к которой остальные компоненты подключаются через мосты. В роли центральной магистрали сначала выступала шина (E)ISA, затем ее сменила шина PCI. Шинно-мостовая архитектура чипсетов просуществовала долгое время и пережила много поколений процессоров (от 2-го до 7-го).

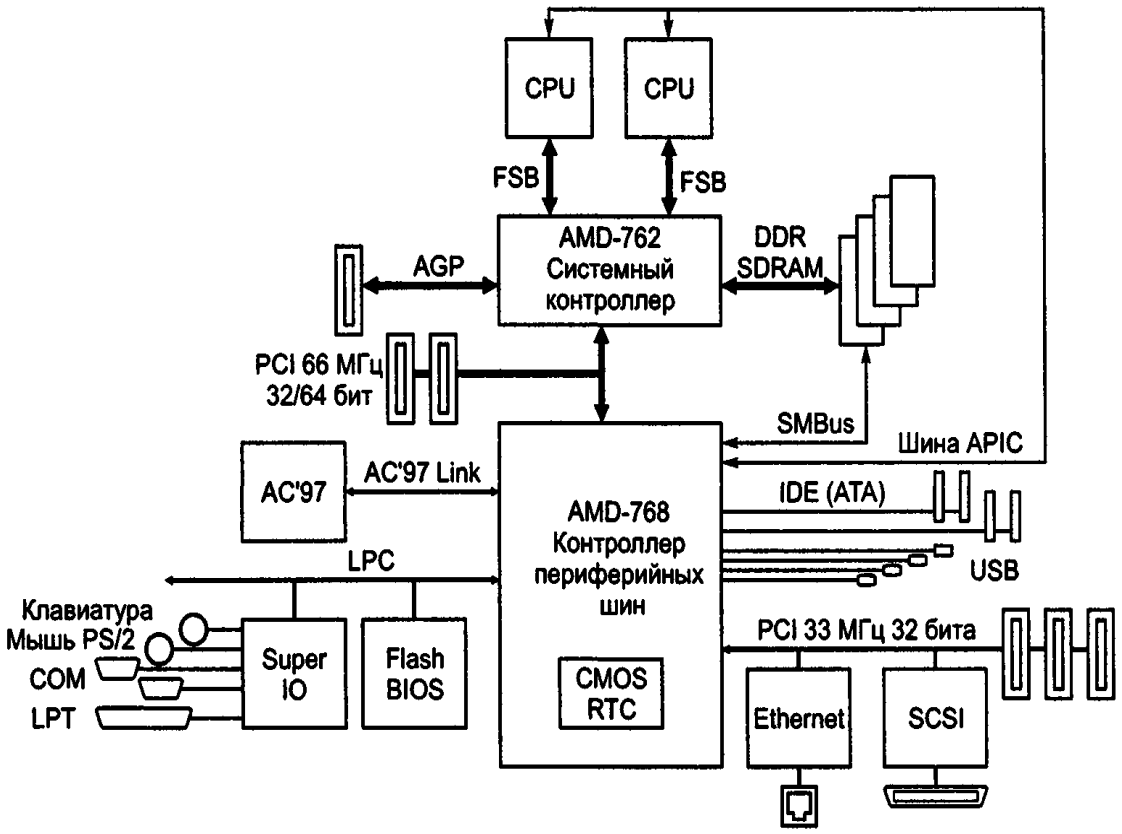


Рисунок 7.1 - Шинно-мостовая архитектура на примере AMD-760

Перемещение вторичного кэша с системной платы на процессор (P6 и Pentium 4 у Intel и K7 у AMD) несколько упростило северную часть чипсета — в ней не надо управлять статической кэш-памятью, а остается лишь обеспечивать согласованность процессорного кэша с основной памятью, доступ к которой возможен и со стороны шины PCI. Шина PCI в роли главной магистрали удержалась недолго: видеокартам с 3D-акселератором ее пропускной способности, разделяемой между всеми устройствами, оказалось недостаточно.

Тогда и появился порт AGP как выделенный интерфейс между графическим акселератором и памятью (а также процессором). При этом задачи северного моста усложнились: контроллеру памяти приходится работать уже на три фронта — ему посылают запросы процессор (ы), мастера шины PCI (и ISA, но тоже через PCI) и порт AGP. Пропускная способность AGP в режиме 2x/4x/8x составляет 533/1066/2133 Мбайт/с, так что шина PCI по производительности стала уже второстепенной. Однако в шинно-мостовой архитектуре она сохраняет свою роль магистрали подключения всех периферийных устройств (кроме графических). В качестве представителя шинно-мостовой архитектуры можно рассматривать чипсет AMD-760 (рис. 7.1). Здесь имеются первичная шина PCI на 64 бит и 66 МГц и вторичная шина для подключения рядовой периферии.

Шина, к которой подключается множество устройств, является узким местом по ряду причин. Во-первых, из-за большого числа устройств, подключенных (электрически) к шине, не удается поднять тактовую частоту до уровня, достижимого в двухточечных соединениях. Во-вторых, шина, к которой подключается множество разнотипных устройств (особенно расположенных на картах расширения), обременена грузом обратной совместимости со старыми периферийными устройствами. Например, предусмотренные возможности повышения производительности PCI используются не всегда: расширение разрядности до 64 бит обходится слишком дорого (большое число проводников порождает свои проблемы), а повышение частоты до 66 МГц для шины возможно лишь, если все ее абоненты поддерживают эту частоту. Достаточно установить одну низкоскоростную карту PCI, и производительность центральной шины падает до начальных 133 Мбайт/с.

Хабовая архитектура

С введением высокоскоростных режимов UltraDMA (ATA/66, ATA/100 и ATA/133) связь двухканального контроллера IDE с памятью через шину PCI стала сильно нагружать эту шину. Кроме того, появились более высокоскоростные интерфейсы Gigabit Ethernet, FireWire (100/200/400/800 Мбит/с) и USB 2.0 (480 Мбит/с). Ответом стал переход на хабовую архитектуру чипсета. В данном контексте хабы (чипсет) — это специализированные микросхемы, обеспечивающие передачу данных между своими внешними интерфейсами.

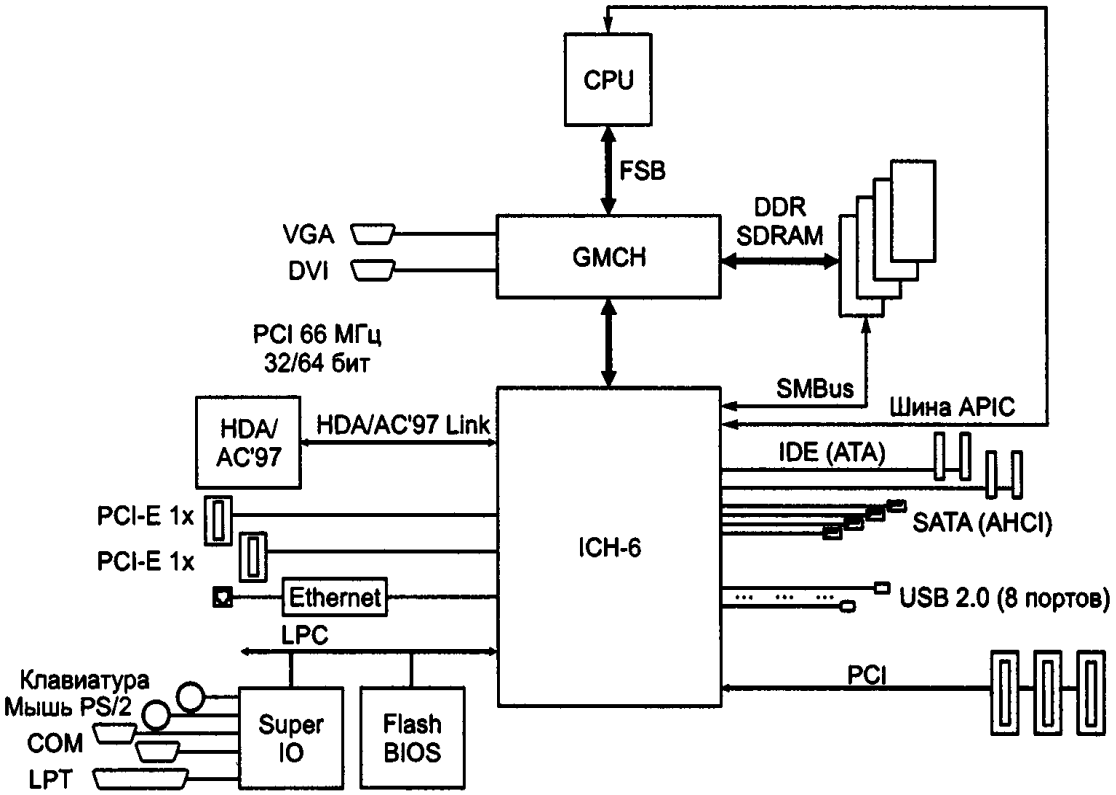


Рисунок 7.2 - Хабовая архитектура на примере чипсета Intel с ICH-6

Этими интерфейсами являлись «прикладные» интерфейсы подключения процессоров, модулей памяти, шин расширения и периферийные интерфейсы (ATA, SATA, USB, FireWire, Ethernet). Поскольку к одной микросхеме все эти интерфейсы не подключить (слишком сложна структура и много требуется выводов), чипсет строился, как правило, из пары основных хабов (северного и южного), связанных между собой высокопроизводительным каналом.

Северный хаб чипсета выполнял те же функции, что и северный мост шинно-мостовой архитектуры: он связывал шины процессора, памяти и порта AGP. Однако на южной стороне этого хаба находился уже не шина PCI, а высокопроизводительный интерфейс связи с южным хабом (рис. 7.2). Пропускная способность этого интерфейса составлял 266 Мбайт/с и выше, в зависимости от чипсета. Если чипсет имеел интегрированную графику, то в северный хаб входил и графический контроллер со всеми своими интерфейсами (аналоговыми и цифровыми интерфейсами дисплея, шиной локальной памяти).

С появлением PCI-E архитектура не слишком изменилась: северный хаб (мост) вместо порта AGP предлагал высокопроизводительный (8x или 16x) порт, а то и пару портов PCI-E для подключения графического адаптера. Маломощные (1x) порты PCI-E предоставлялись как северным, так и южным хабами.

Северные мосты и хабы

Северный хаб (как и мост) определяли основные возможности системной платы:

- Поддерживаемые процессоры — типы, частоты системной шины, возможности мультипроцессорных или избыточных конфигураций. Типы процессоров определялись протоколами системной шины.
- Типы памяти и частота работы шины памяти (На системных платах для процессоров со встроенным контроллером памяти характеристики памяти (тип, число каналов, частоту) задает процессор).
- Максимальный объем памяти.
- Число каналов памяти — один, два канала.
- Возможность и эффективность применения разнородной памяти.
- Поддержка контроля достоверности памяти и исправления ошибок (ECC).
- Возможности системы управления энергопотреблением (ACPI или APM) — реализуемые энергосберегающие режимы процессора и памяти, управление производительностью, SMM.

Южные мосты и хабы

Южный хаб чипсета обеспечивали подключение шин PCI, PCI-X и «маломощных» портов PCI-e, ATA (2 канала), SATA, USB, FireWire, а также контроллеров ввода-вывода, памяти CMOS и флэш-памяти с системным модулем BIOS. В южной части располагались таймер (8254), контроллер прерываний (APIC), контроллер DMA (рис. 7.3). Если в чипсет интегрирован звук, то южный хаб (мост) имели контроллер интерфейса AC-Link или HDA Link для подключения аудиокодека, а то и сам аудиокодек. Для контроллеров ввода-вывода, ввели новый интерфейс LPC (Low Pin Count).

Флэш-память для хранения системной памяти BIOS стали помещать в специальный хаб (firmware hub), соединяемый с южным хабом отдельной шиной (аналогичной LPC). Флэш-память подключалась прямо к шине LPC. Для обслуживания процессоров, имеющих дополнительную сервисную шину SMBus, хаб мог иметь последовательный интерфейс I^2C для чтения идентификаторов модулей памяти.

В южный хаб интегрированных чипсетов вводили и контроллер локальной сети (как правило, Ethernet).

Требование прямого доступа в память

Требование прямого доступа в память (DMA: Direct Memory Acces) это особенный тип прерывания, которое не входит в общую процедуру обслуживания прерывания и является аппаратным средством ввода/вывода.

Требование выдавалось на специальный вход, обычно “BUSRQ” (требование шины). Требование имеет максимальный приоритет и передавалось драйверу по обслуживанию прерывания и контроллер DMA инициирует DMA трансфер. В свободные циклы памяти осуществлялась передача в/из RAM (рис.7.3).

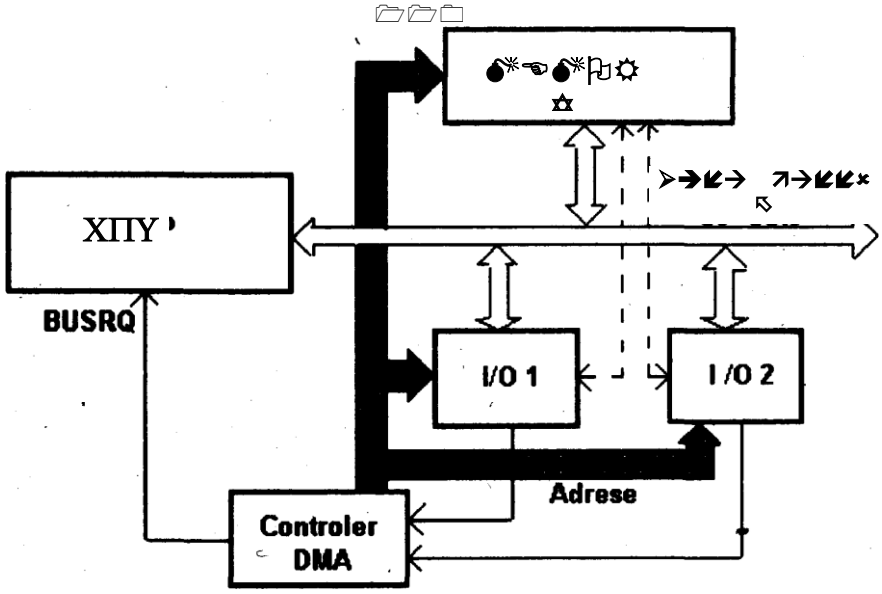


Рис. 7.3

Контроллер DMA может быть внутренним или внешним микропроцессору. Он адресует блоки данных в памяти (последовательно, адрес за адресом, в определенных пределах), исключая внутренние регистры микропроцессора. Контроллер DMA управляет передачей в/из порта (в/из периферийного устройства), что существенно повышает скорость обмена информацией.

Архитектура HyperTransport

Технология (архитектура) HyperTransport (HT) задумывалась как альтернатива шинно-мостовой архитектуре системных плат. Технология разработана компаниями AMD, Apple Computers, Broadcom, Cisco Systems, NVIDIA, PMC-Sierra, SGI, SiPackets, Sun Microsystems, Transmeta. Первый релиз вышел в 2001 году, в 2003-м — версия 1.10. Прежнее кодовое название — LDT (Lighting Data Transport).

Основная идея HT — замена шинного соединения компонентов (периферийных устройств) системой двухточечных встречно направленных соединений. При этом достижима более высокая тактовая частота интерфейсов, что обеспечивает их более высокую (по сравнению с шиной) пропускную способность. Структурная схема компьютера архитектуры HT приведена на рис. 6.3. Главный мост (host bridge) обеспечивает связь HT с ядром — процессором и памятью. Периферийные контроллеры, требующие высокой пропускной способности, реализуются в виде HT-туннелей. В архитектуре предусматривается и мостовая связь с шиной PCI (рис. 7.4).

Транзакции выполняются в виде серий передач пакетов различных типов. Транзакции выполняются расщепленным способом: инициатор посылает пакет-запрос и данные для транзакции записи, целевое устройство посылает пакет-ответ и данные для транзакций чтения.

Сигнализация прерываний в HT реализуется тоже пакетами: устройство посылает сообщение — выполняет транзакцию записи по адресу, указанному ему при конфигурировании. Обработчик прерывания посылает сообщение о завершении обработки прерывания (End Of Interrupt, EOI), делая запись по другому адресу, связанному с данным устройством. Такой механизм сигнализации запросов и подтверждений позволяет преодолеть неэффективность традиционного для PC механизма прерываний с помощью специальных линий IRQ.

Архитектура HT основана на двусторонней пакетной передаче данных между парой устройств. Устройство HT может выступать в роли инициатора или/и целевого устройства транзакций.

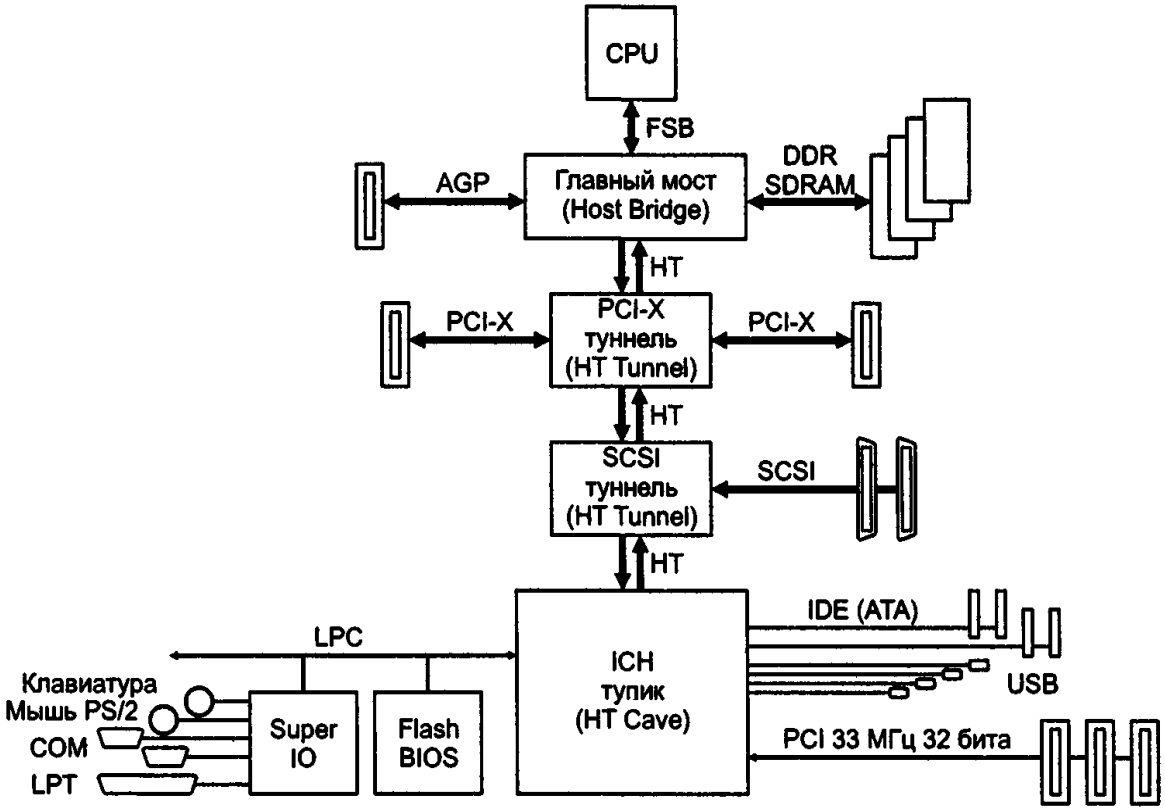


Рисунок 7.4 Архитектура HyperTransport

По топологическим свойствам различают несколько типов устройств HT:

- Туннель (tunnel) — устройство с двумя интерфейсами HT (рис. 7.5); такие устройства могут собираться в цепочку (daisy chain), образующую логическую шину. Цепочка подключается к хосту (процессору с главным мостом), отвечающему за конфигурирование всех устройств и управляющему работой HT.
- Мост (bridge) — устройство, соединяющее одну логически первичную шину (подключенную к хосту) с одной или несколькими логически вторичными шинами (цепочками). Мост имеет набор регистров, информация которых позволяет управлять распространением транзакций между этими шинами (аналогично мосту PCI).
- Коммутатор (switch) — устройство с несколькими интерефейсами HT, по структуре аналогичное нескольким мостам PCI, подключенным к одной (внутренней) шине.
- Тупик (cave) — устройство с одним интерфейсом HT.

Хост (host) — это «хозяин шины», подключающийся к ней через главный мост и выполняющий функции конфигурирования (аналогично и совместимо с PCI). Основной вариант топологии — цепочка устройств-туннелей, подключенная к хосту. Каждый интерфейс HT состоит из двух независимых частей: передатчика и приемника. Каждому устройству при конфигурировании выделяются свои области в адресном пространстве. В цепочке устройства-туннели транслируют пакеты сверху вниз (нисходящий трафик) и снизу вверх (восходящий). Если в нисходящем управляющем пакете устройство обнаруживает свой адрес, оно «понимает», что обращаются к нему, и принимает соответствующую информацию (управляющие пакеты и данные). Восходящий трафик туннель транслирует «вслепую». На полученные запросы устройство отвечает посылкой пакетов вверх, включая их в транслируемый восходящий трафик.

Таким образом, обеспечивается программное взаимодействие процессора с устройствами. Собственные запросы на доступ к памяти устройство посылает тоже вверх, как и запросы (обращения) к другим устройствам (независимо от положения целевого устройства — выше или ниже в цепочке). Доставку пакета адресату обеспечивает главный мост: он разворачивает пакет, принятый из цепочки (адресованный не к ОЗУ), и посылает его вниз — так организуется одноранговое взаимодействие. На пакет, адресованный к ОЗУ, главный мост организует ответ от контроллера памяти, реализуя, таким образом, прямой доступ к памяти.

Возможны и более сложные топологии, например дерево (с мостами), позволяющее подключать больше тупиковых устройств.

Технология HyperTransport предназначена для соединения компонентов компьютеров и коммуникационной аппаратуры, но только в пределах платы — слоты и карты расширения технологией HT не рассматриваются. Для передачи информации используются два встречных однонаправленных набора высокоскоростных сигналов – Hyper transport link (Side A) – по 16 бит в двух направлениях (рис 7.5).

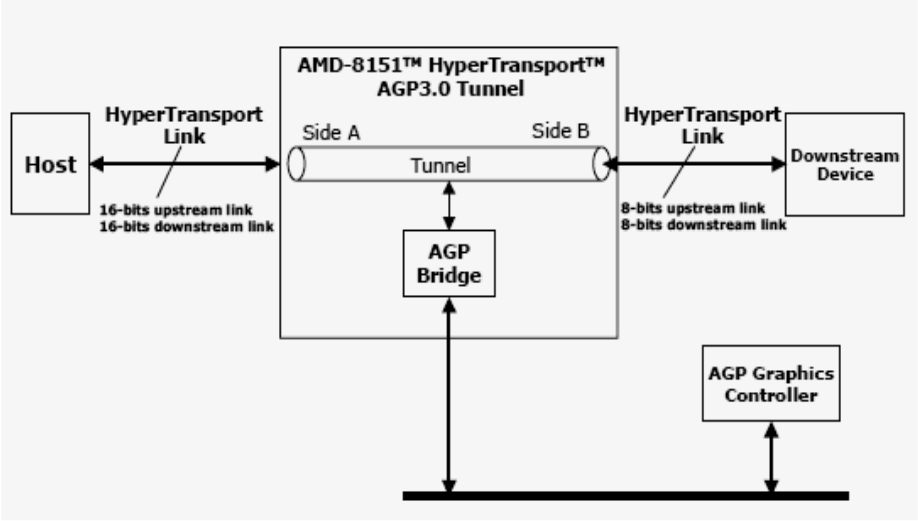


Рисунок 7.5 Схема соединенения контоллера AGP к шине HT

Сигналы передаются по дифференциальным парам проводов с импедансом 100 Ом, сигналы — LVDS (низковольтные дифференциальные, уровень 1,2 В). Частота синхронизации 200, 300, 400, 500, 600, 800 и 1000 МГц обеспечивает физическую скорость передачи 400, 600, 800, 1000, 1200, 1600 и 2000 МТ/с (миллионов передач в секунду), что при самых больших разрядности (32 бит) и частоте обеспечивает пиковую скорость передачи данных до 8 Гбайт/с. В первой версии предельная частота была 800 МГц, что давало скорость 6,4 Гбайт/с. Поскольку пакеты могут передаваться одновременно в обоих направлениях, можно говорить о суммарной пропускной способности 12,8 или 16 Гбайт/с.

В вышеприведенном примере главный мост реализует интерфейс AGP.

В 64-битных процессорах AMD, в которых применяется HT, главный мост размещается в самом процессоре. При этом у процессора оказывается два интерфейса: интерфейс памяти и HT в качестве системной шины. В распространенных чипсетах (от VIA, SiS) к интерфейсу HT подключается только северный хаб, обеспечивающий лишь интерфейс подключения графического адаптера — AGP или PCI-E. Южный хаб соединяется с северным собственным интерфейсом, так что использования HT как универсальной транспортной структуры для множества компонентов пока не наблюдается.

Рассмотрим системную плату ASUS Z97-K

- Socket LGA1150
- Поддерживаемые процессоры
- Intel Core i7/Core i5/Core i3/Pentium/Celeron
- Память DDR3 DIMM, 1333 - 3200 МГц
- Количество слотов памяти 4
- Максимальный объем памяти 32 Гб
- количество разъемов SATA 6Gb/s: 6

Слоты расширения

- 2xPCI-E x16, 2xPCI-E x1, 2xPCI
- Наличие интерфейсов
- 14 USB, из них 6 USB 3.0 (4 на задней панели), выход S/PDIF, 1xCOM, D-Sub, DVI, HDMI, Ethernet, PS/2 (клавиатура), PS/2 (мышь)
- 64MB Flash ROM UEFI AMI BIOS

В следующих главах мы подробнее расскажем об основных составных элементах, которые находятся на материнской плате.

7. 2 Шины

Как уже отмечалось (гл. 3), совокупность линий (проводников на системной плате), по которым обмениваются информацией компоненты и устройства PC, называются шиной (Bus). Шина предназначена для обмена информацией между двумя и более устройствами.

Обычно шина имеет разъемы (слоты) для подключения внешних устройств, которые в результате сами становятся частью шины и могут обмениваться информацией со всеми другими подключенными к ней устройствами.

Различают параллельные и последовательные шины.

Рассмотрим параллельные шины.

Шина имеет собственную архитектуру, позволяющую реализовать важнейшие ее свойства — возможность параллельного подключения практически неограниченного числа внешних устройств и обеспечение обмена информацией между ними.

Архитектура любой паралельной шины включает следующие компоненты:

- Линии для обмена данными (шины данных)
- Линии для адресации данных (шины адреса)
- Линии для управления данными (шины управления)
- Контроллер шины

Контроллер шины осуществляет управление процессом обмена данными и служебными сигналами и обычно выполняется в виде отдельной микросхемы либо интегрируется в микросхемы Chipset.

Основные характеристики шины.

- Разрядность шины (иногда говорят ширина шины), которая определяется количеством данных, параллельно "проходящих" через нее.

- ⎜□⎩⎪ ⎧⎬⎨◊□ ⎭⎪ ⎩⎭⎩⎨ ⎩⎭⎧, ⎪ ⎩⎩□◊□ ⎩⎪□∑™∑⎣□∑⎭□⎪ ⎣⎩ ⎢∑⎭∫®⎩⎧ ⎨⎩ ⎨ ⎜⎩□⎧◊⎭⎩⎩, ⎜∑□∑™◊®◊ ∑⎡⎦⎠ ⎪⎩ ⎠⎨∑⎜◊ ⎭∑⎢ ⎧⎨™ ⎧. ⊗⎣□ ⎩⎪□∑™∑⎣∑⎨⎩□ ⎜□⎩⎪ ⎧⎬⎨ ⎩⎡ ⎭⎪ ⎩⎭⎩⎨ ⎩⎭⎧⎩ ⎠⎨ ⎦⎨∑⎩⎨⎠ ⎣™⎩⎧⎩ ⎧ ⎧⎨ ⎩⎧⎩⎧ ∫◊⎢ ∫⎩® ⎧⎠ ⎢◊⎭∫⎩⎩ ⎧ ⎠⎨ ⎦⎨◊ ∑∑ □◊⎢ □□™⎨ ⎩⎭⎧. ⊆◊⎢□⎩⎧∑□, ™⎣□ 16–□◊⎢ □□™⎨ ⎩⎡ ⎠⎨ ⎦ IΣ A ⎜□⎩⎪ ⎧⎬⎨◊□ ⎭⎪ ⎩⎭⎩⎨ ⎩⎭⎧ ⎩⎪□∑™∑⎣□∑⎭□ ∫◊⎢ :

(16 бит x 8,33 МГц) : 8 = (133,28 Мбит/с) : 8 = 16,66 Мбайт/с.

Интерфейс

Внешние устройства к шинам подключаются посредством интерфейса. Под интерфейсом (Interface- Сопряжение) периферийного устройства PC понимают устройство, которое обеспечивает организацию обмена информацией между периферийным устройством и шинной к которой подключен. Интерфейс включает в себе электрические и временные параметры, набор управляющих сигналов, протокол обмена данными и конструктивные особенности подключения. При этом обмен данными между компонентами PC возможен только в случае совместимости их интерфейсов.

В случае несовместимости интерфейсов (например, интерфейс системной шины и интерфейс HD) используются контроллеры.

7.3 ∇⎩⎭∫∑⎨⎦∑ ⎠⎨⎦

Системная шина предназначена для обмена информацией между CPU и другими устройствами, входящими в систему.

7.4.1 ¬⎨◊ ΠΧΙ

7.4.1 Паралельная шина PCI

PCI (Peripheral Component Interconnect) local bus - шина соединения периферийных компонентов. Шина разрабатывалась в расчете на Pentium. Первая версия PCI 1.0 появилась в 1992 году.

Шина PCI соединяется с системной шиной процессора «южным» мостом, входящим в состав чипсета.

Шина является синхронной — фиксация всех сигналов выполняется по положительному перепаду (фронту) сигнала CLK. Номинальной частотой синхронизации считается 33 МГц. Начиная с версии 2.1, допускается повышение частоты до 66 МГц при согласии всех устройств на шине.

Номинальная разрядность шины данных — 32 бита, спецификация определяет и расширение разрядности до 64 бит. При частоте шины 33 МГц теоретическая пропускная способность достигает 132 Мбайт/с для 32-битной шины и 264 Мбайт/с для 64-битной; при частоте синхронизации 66 МГц — 264 и 528 Мбайт/с соответственно. Однако эти пиковые значения достигаются лишь во время передачи пакета, а из-за протокольных накладных расходов реальная средняя суммарная (для всех задатчиков) пропускная способность шины будет ниже.

Конфигурация. Устройством PCI называется микросхема или карта расширения, использующая для идентификации выделенную ей линию IDSEL. Устройство может быть многофункциональным, то есть состоять из множества так называемых *функций.* Каждой функции отводится конфигурационное пространство в 256 байт (Стандарт PCI определяет для каждого слота конфигурационное пространство размером до 256 регистров (8-битных), не приписанных ни к пространству памяти, ни к пространству ввода-вывода). Доступ к ним осуществляется по специальным циклам шины *Configuration Read* и *Configuration Write*.

Номер функции (function number), к которой производится обращение в конфигурационных транзакциях, передается по линиям AD [10:8] (младшие линии требуются для адресации регистров внутри конфигурационного пространства). Таким образом, устройство может содержать до 8 функций. Простые (однофункциональные) устройства, в зависимости от реализации, могут отзываться либо на любой номер функции (игнорировать значение AD [10:8] в конфигурационном цикле), либо только на номер функции 0. Многофункциональные устройства должны отзываться только на конфигурационные циклы с номерами функций, для которых имеется конфигурационное пространство. При этом функция с номером 0 должна быть обязательно, номера остальных функций назначаются произвольно.

На одной шине PCI может присутствовать несколько устройств, каждое из которых имеет свой *номер устройства* (device number). И, наконец, в системе может присутствовать несколько шин PCI,

каждая из которых имеет свой *номер шины* (PCI bus number). Шины нумеруются последовательно; шина, подключенная к главному мосту, имеет нулевой номер.

Протокол шины PCI

В каждой транзакции (обмене по шине) участвуют два устройства — инициатор обмена (Initiator или Master, инициирующее устройство, ИУ) и целевое устройство (Target или Slave, ЦУ). Шина PCI все транзакции трактует как пакетные: каждая транзакция начинается фазой адреса, за которой может следовать одна или несколько фаз данных.

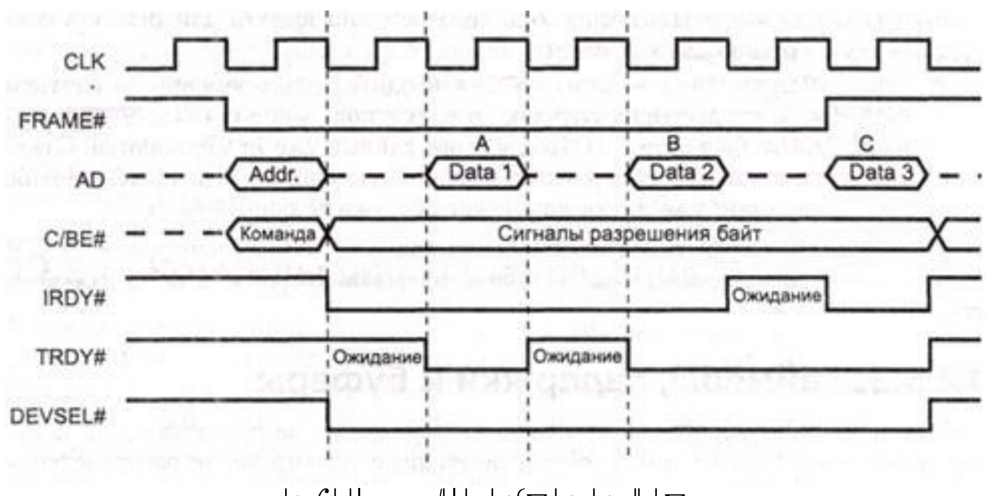
В каждый момент времени шиной может управлять только один мастер, получивший на это право от арбитра. Каждый мастер имеет пару сигналов — REQ# (Request — запрос от PCI-мастера на захват шины) для запроса на управление шиной и GNT# (Grant — предоставление мастеру управления шиной) для подтверждения предоставления управления шиной.

Устройство может начинать транзакцию (устанавливать сигнал FRAME# (FRAME# - Кадр)) только при активном полученном сигнале GNT#. Введением сигнала отмечается начало транзакции (фаза адреса), снятие сигнала указывает на то, что последующий цикл передачи данных является последним в транзакции. Снятие сигнала GNT# не позволяет устройству начать следующую транзакцию, а при определенных условиях (см. ниже) заставляет прекратить начатую транзакцию.

Арбитражем запросов на использование шины занимается специальный узел, входящий в чипсет системной платы. Схема приоритетов (фиксированный, циклический, комбинированный) определяется программированием арбитра.

Для адреса и данных используются общие мультиплексированные линии AD. Четыре мультиплексированные линии C/BE [3:0] используются для кодирования команд в фазе адреса и разрешения байт в фазе данных. В начале транзакции ИУ активизирует сигнал FRAME#, по шине AD передает целевой адрес, а по линиям C/BE# — информацию о типе транзакции (команде). Адресованное ЦУ отзывается сигналом DEVSEL#, после чего ИУ может указать на свою готовность к обмену данными сигналом IRDY#. Когда к обмену данными будет готово и ЦУ, оно установит сигнал TRDY#. Данные по шине AD могут передаваться только при одновременном наличии сигналов IRDY# и TRDY#. С помощью этих сигналов ИУ и ЦУ согласуют свои скорости, вводя такты ожидания.

На рис. 7.6 приведена временная диаграмма обмена, в которой и ИУ, и ЦУ вводят такты ожидания. Если бы они оба ввели сигналы готовности в конце фазы адреса и не снимали их до конца обмена, то в каждом такте после фазы адреса передавались бы по 32 бита данных, что обеспечило бы выход на предельную производительность обмена.



∠⟩ ⌈⌉ ∐ 7.6 ⩘⌊ ⌊⟨∑⌉◊⌉◊ ⋌⌉∑ ΠΧΙ

Каждая транзакция на шине должна быть завершена планово или прекращена, при этом шина должна перейти в состояние покоя (сигналы FRAME# и IRDY# пассивны). Завершение транзакции выполняется либо по инициативе мастера, либо по инициативе ЦУ.

Количество фаз (циклов) данных в пакете заранее не определено, но перед последним циклом ИУ при введенном сигнале IRDY# снимает сигнал FRAME#. После последней фазы данных ИУ снимает сигнал IRDY#, и шина переходит в состояние покоя (PCIIdle) — оба сигнала FRAME# и IRDY# находятся в пассивном состоянии.

Работа шины контролируется несколькими таймерами, не позволяющими попусту расходовать такты шины и обеспечивающими планирование распределения полосы пропускания.

Каждое ЦУ должно достаточно быстро отвечать на адресованную ему транзакцию. *Задержка первой фазы данных* не должна превышать 16 тактов шины.

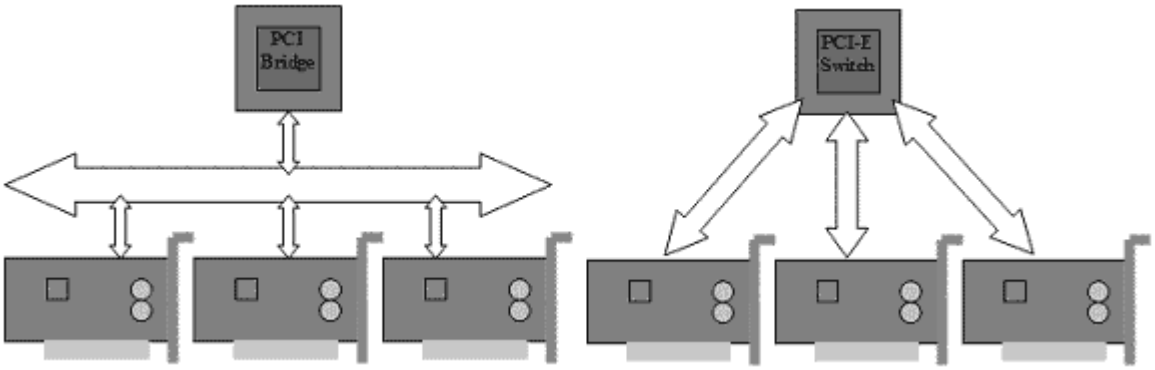
Инициатор тоже не должен задерживать поток — не задерживать фазы данных. Каждый мастер, способный сформировать пакет с более чем двумя фазами данных, должен иметь собственный программируемый *таймер задержки* (Latency Timer), регулирующий поведение мастера, когда у него отбирают право управления шиной. В зависимости от исполняемой команды и состояния сигналов мастер должен либо сократить транзакцию, либо продолжать ее до запланированного завершения.

Важной особенностью шины PCI является то, что в ней реализован принцип Bus Mastering, который подразумевает способность внешнего устройства при пересылке данных управлять шиной (без участия CPU). Во время передачи информации устройство, поддерживающее Bus Mastering, захватывает шину и становится главным. При таком подходе центральный процессор освобождается для выполнения других задач, пока происходит передача данных.

7.4.2 Последовательная шина PCI Express

Интерфейс ***PCI Express*** (первоначальное название - 3GIO (Third Generation Input/Output)) использует концепцию PCI, однако физическая их реализация кардинально отличается. На физическом уровне PCI Express представляет собой не шину, а некое подобие сетевого взаимодействия на основе последовательного протокола.

Одна из концептуальных особенностей интерфейса PCI Express, позволяющая существенно повысить производительность системы, - использование топологии "звезда". В топологии "шина" (рис. 7.7а) устройствам приходится разделять пропускную способность PCI между собой. При топологии "звезда" (рис. 7.7б) каждое устройство монопольно использует канал, связывающий его с концентратором (switch) PCI Express, не деля ни с кем пропускную способность этого канала.



а) б)

Рисунок 7.7 Сравнение топологий PCI и PCI Express

Канал (link), связывающий устройство с концентратором PCI Express, представляет собой совокупность дуплексных последовательных (однобитных) линий связи, называемых полосами (lane). Дуплексный характер полос также контрастирует с архитектурой PCI, в которой шина данных - полудуплексная (в один момент времени передача выполняется только в определенном направлении). На электрическом уровне каждая полоса соответствует двум парам проводников (рис. 7.8) с дифференциальным кодированием сигналов (LVDS – Low Voltage Differential Signaling). Одна пара используется для приема, другая – для передачи.

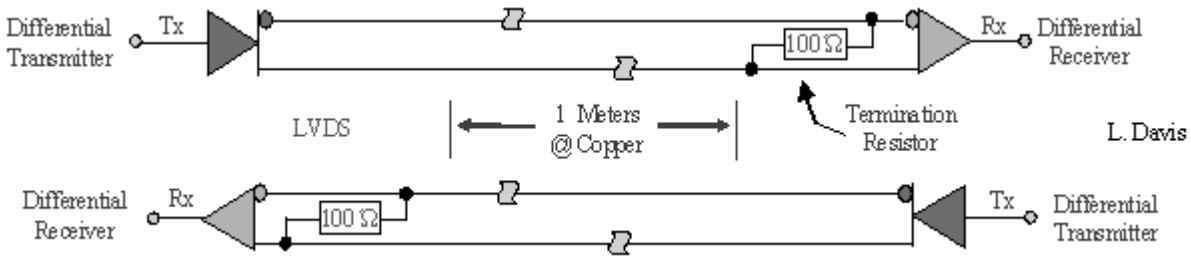
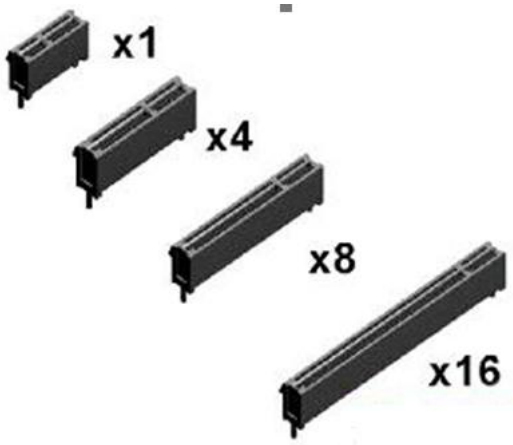


Рисунок 7.8

PCI Express первого поколения декларирует скорость передачи одной полосы 2,5 Гбит/с в каждом направлении.

	Raw Bit Rate	Link Bandwidth (BW)	BW/Lane/Way	Total BW x16
PCIe 1.x	2.5 GT/S	2 GB/S	~250 MB/S	~8 GB/S
PCIe 2.x	5 GT/S	4 GB/S	~500 MB/S	~16 GB/S
PCIe 3.0	8 GT/S	8 GB/S	~1 GB/S	~32 GB/S
PCIe 4.0	16 GT/S	16 GB/S	~2 GB/S	~64 GB/S

Канал может состоять из нескольких полос: одной (x1 link), двух (x2 link), четырех (x4 link), восьми (x8 link), шестнадцати (x16 link) или тридцати двух (x32 link). Все устройства должны поддерживать работу с однополосным каналом. Аналогично, различают слоты: x1, x2, x4, x8, x16, x32.



Однако слот может быть "шире", чем подведенный к нему канал, т.е. на слот x16 фактически может быть выведен канал x8 link и т.п. Карта PCI Express должна физически подходить и корректно работать в слоте, который по размерам не меньше разъема на карте, т.е. карта x4 будет работать в слотах x4, x8, x16, даже если реально к ним подведен однополосный канал. Процедура согласования канала PCI Express обеспечивает выбор максимального количества полос, поддерживаемого обеими сторонами.

При передаче данных по многополосным каналам используется принцип чередования (data stripping): каждый последующий байт передается по другой полосе. В случае канала x2 это означает, что все четные байты передаются по одной полосе, а нечетные - по другой (рис. 7.9).

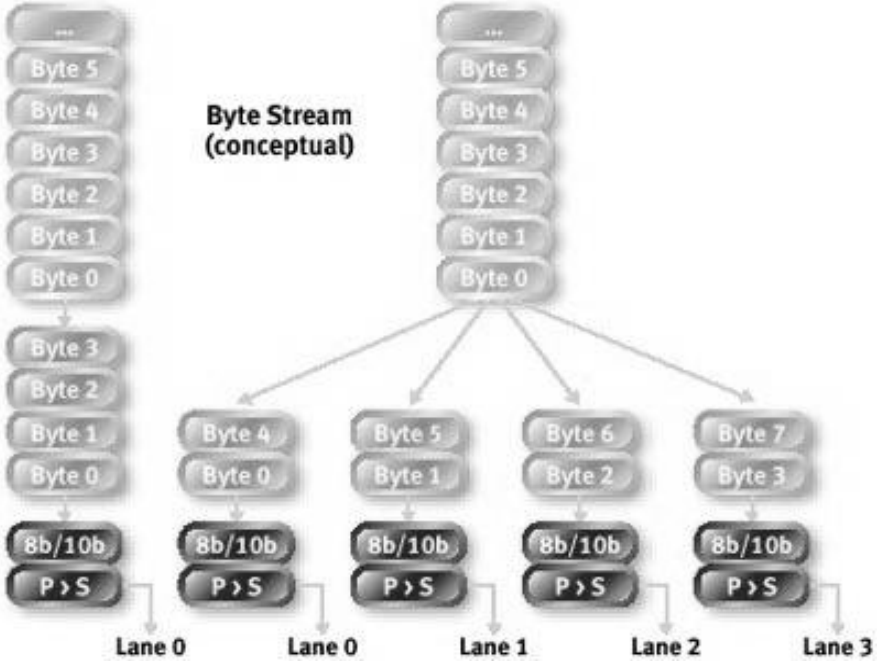


Рисунок 7.9

Как и большинство других высокоскоростных последовательных протоколов, PCI Express использует схему кодирования данных, встраивающую тактирующий сигнал в закодированные данные, т.е. обеспечивающую самосинхронизацию. Применяемый в PCI Express алгоритм 8b/10b (8 бит в 10 бит - каждый байт информации передается как 8 бит + 2 контрольных бита = 10 бит) обеспечивает разбиение длинных последовательностей нулей или единиц так, чтобы приемная сторона не потеряла границы битов.

Протокол PCI Express

Формат одного кадра показан на рисунке 7.10. Он состоит из 1-байта - Start-of-Frame, 2-байта - Номер пакета, 16 или 20-байт - Заголовок, от 0 до 4096-байт- Data field, от 0 до 4-байт поле ECRC (End-to-end Cyclic Redundancy Check), 4- байт LCRC (Local Cyclic Redundancy Check), и 1- байт End-of Frame.

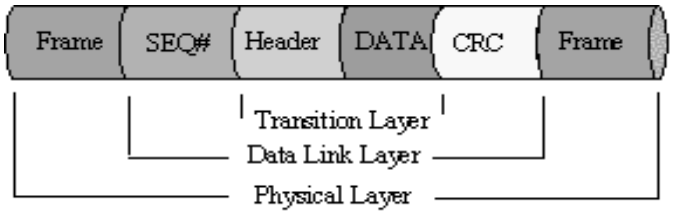


Рисунок 7.10

Следуя этому формату при передаче 4096 байт данных, кадр состоит из 4124 байта.

7.5 Интерфейс ATA

7.5.1 Паралельная шине ATA (IDE)

Шины ATA и SCSI являются кабельными и довольно протяженными (особенно SCSI). Их области применения пересекаются на устройствах хранения данных. Область применения SCSI шире — эта шина требуется для подключения разнообразных периферийных устройств, а не только для устройств хранения данных. Шины SCSI, в основном, применяются в серверах и мэйнфреймах.

Интерфейс ATA — AT Attachment for Disk Drives — разрабатывался в 1986-1990 годах для подключения накопителей на жестких магнитных дисках к компьютерам IBM PC AT с шиной ISA. Стандарт, определяет набор регистров и назначение сигналов 40-контактного интерфейсного разъема.

В последней спецификации ATA/ ATAPI -7 фигурируют перечисленные ниже компоненты.

- *Хост-адаптер* - средства сопряжения интерфейса ATA с системной шиной Хостом мы будем называть компьютер с хост-адаптером интерфейса ATA.
- *Ленточный кабель* (шлейф) с двумя или тремя 40-контактными IDE-разъемами. Наиболее распространенный 40-проводный сигнальный и 4-проводный питающий интерфейс для подключения дисковых накопителей к компьютерам AT. Для миниатюрных (2,5" и менее) накопителей используют 44-проводный кабель, по которому передается и питание.
- Включает 2 канала, 4 устройства.
- Включает средства парольной защиты, улучшенного управления питанием, самотестирования с предупреждением приближения отказа — SMART (Self Monitoring Analysis and Report Technology).
- Включает режим Ultra DMA со скоростью обмена до 100, 133 Мбайт/с.

Все сигналы ATA являются логическими со стандартными ТТЛ-уровнями. Вид кабеля приведен на рис. 7.11. В большинстве кабелей одноименные контакты всех разъемов соединяются своими проводами и все коннекторы равноправны.

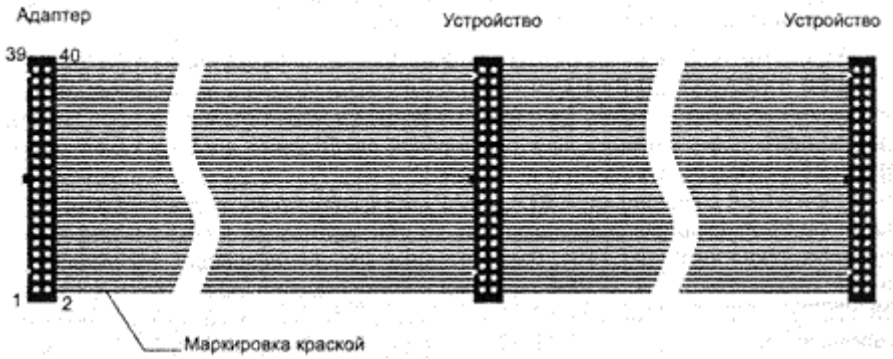


Рисунок 7.11 - Интерфейсный кабель ATA

Для устойчивой работы в режиме Ultra DMA рекомендуется применение *80-проводных кабелей,* обеспечивающих чередование сигнальных цепей и проводов схемной земли (GND). Такие кабели, требуемые для режимов UltraDMA (скорость выше 66 Мбайт/с), разделываются на специальные разъемы, имеющие 40-контактные гнезда с обычным назначением контактов, но ножевые контакты для врезки 80 проводов.

7.5.2 Технология Serial ATA

При современных технологиях использование 5-ти вольтовых сигналов стало очень затруднительно и не применяются в высокоскоростных передачах. В Serial ATA уровни сигналов составляют 250 мВ. Вместо использовавшейся раннее в ATA однополярной передачи, обладающей низкой помехоустойчивостью, применена двухполярная (или еще ее называют дифференциальной). Преимущество ее в большей помехозащищенности. При дифференциальной передаче по двум проводам передается один и тот же сигнал, но разной полярности. Собственно использование дифференциальной передачи и дало возможность снизить уровни используемого сигнала.

Для кодирования передаваемой информации используется потенциальный код без возвращения к нулю (Non Return to Zero, NRZ). Используется последовательная, состоящая из 2-х пар проводов (одной передачи и одной на прием) и несколько нулевых (рис. 7.12). Всего семь, что становится удобным в использовании и не препятствует воздухообмену.

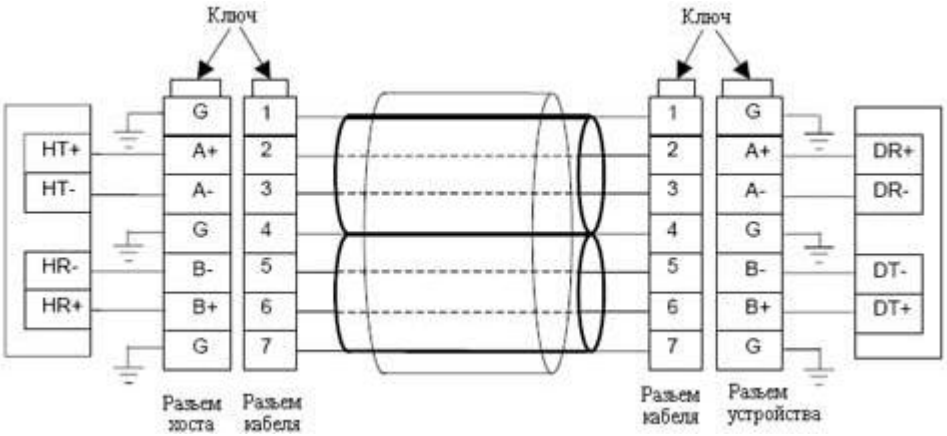


Рисунок 7.12

Длина кабелей может достигать 1 метра. SATA может быть не только интерфейсом внутренних устройств хранения, но и внешних.

Жесткие диски с SATA имеют максимальную скорость обмена по интерфейсу в 150 Мбайт в секунду, при том, что для SATA заявлена скорость передачи на физическом уровне в 1.5 Гбит/с, объясняется использованием избыточного 8B/10B кодирования, снижающего полезную пропускную способность интерфейса до 1.2 Гбит/с.

Стандарт предусматривает разработку трех версий. Первая – рассмотрена выше. Во второй в два раза увеличена пропускная способность - до 3 Гбит/с, при сохранении полной совместимости с первой, а в третьей - до 6 Гбит/с.

7.5.1 Шина SCSI

7.6.1 Паралельная шина SCSI

Системный интерфейс малых компьютеров SCSI (Small Computer System Interface, произносится «скази») был стандартизован ANSI в 1986 году (X3.131-1986).

Интерфейс предназначен для соединения устройств различных классов: жесткие диски, CD-ROM, RAID массивов, сканеров, коммуникационных устройств, ….

Устройством SCSI (SCSI Device) называется как *хост-адаптер,* связывающий шину SCSI с какой-либо внутренней шиной компьютера, так и *контроллер целевого устройства* (target controller), с помощью которого устройство подключается к шине SCSI. С точки зрения шины все устройства могут быть равноправными и являться как *инициаторами обмена* (инициализирующими устройствами, ИУ), так и *целевыми устройствами* (ЦУ), однако чаще всего в роли ИУ выступает хост-адаптер. К одному контроллеру может подключаться несколько периферийных устройств (ПУ), по отношению к которым контроллер может быть как внутренним, так и внешним. Широкое распространение получили ПУ со встроенным контроллером SCSI (embeded SCSI controller), к которым относятся накопители на жестких магнитных дисках, CD-ROM, стримеры.

Возможность присутствия на шине более одного контроллера (инициатора обмена) позволяет обеспечить разделяемое использование периферии несколькими компьютерами, подключенными к одной шине.

В настоящее время паралельный вариант шины практически не используется.

7.6.1 Последовательная шина SCSI (SAS)

В 2001 году появилась шина **SAS 1.0** (Serial Attached SCSI) со скоростью 1 500 Мбайт/с. По ней данные передаются последовательно, предварительно объединяясь в пакеты. Жèсткие диски подключаются по методу «точка-точка», потому пропускные способности у всех дисководов одинаковые, а не делятся. В 2004 году появилась SAS 1.1 со скоростью 3 000 Мбайт/с, а версия шины **SAS 2.0** со скоростью 6 000 Мбайт/с. Шина является частью нового стандарта Serial SCSI (SCSI-3) который появился в 2013 году, декларируя скорость 12 Гбайт/с. Serial SCSI – это высокоскоростная, локальная, последовательная шина, разработанная фирмами Apple и Texas Instruments в 1995 году. Изменяемая структура и одноранговая топология делают еѐ удобной для подключения жèстких дисков и устройств обработки аудио- и видеоинформации, а также для работы мультимедийных приложений в реальном времени. Она позволяет организовать дуплексный режим работы с несколькими устройствами, передающим информацию с разными скоростями.

Шина SATA поддерживает семидуплексный режим работы. Вы можете подключить большое количество SAS дисков к одному порту контроллера с помощью експандеров (expanders) - расширителей SAS. Расширитель SAS подобен сетевому коммутатору, что позволяет подключать несколько систем с помощью одного порта коммутатора. Стоимость системы, включающей расширитель значительно меньше, чем стоимость системы, содержащей контроллер SAS с большим количеством портов или несколько управляющих контроллеров, с меньшим количеством портов. Используя расширители, возможны до 65,535 физических подключений.

Интерфейс SAS использует дифференциальные сигналы, 8B/10B кодирование данных и шифрования данных для уменьшения электромагнитных помех. Можно ипользовать до четырех портов одному устройству, что позволяет увеличить скорость передачи данных до 24 Гбит / с. Диски SAS имеют двойной порт, что позволяет напрямую подключаться и находиться под контролем двух контроллеров одновременно. Эта опция позволяет строить системы с резервированием. Если один из контроллеров SAS выходит из строя, другой сможет управлять дисками и данными, хранящихся на этих дисках.

Основные различия между интерфейсами SAS и SATA:

- Диски SATA идентифицируются по номеру порта подключенного к хост-адаптеру, в то время как устройства SAS идентифицируются по уникальному коду или *World Wide Name* (WWN). Этот идентификатор (WWN) представляет собой 64-битный код, который и есть адрес SAS. Из 64 битов, 24 бита представляют код фирмы производителя и 40 бит представляют код устройства SAS.
- Интерфейс SATA позволяет подключить только hard диски и оптические приводы. Интерфейс SAS позволяет подключать и другие виды оборудования, такие как сканеры и принтеры. Однако эти устройства, как правило, кроме интерфейса SAS, используют такие интерфейсы как USB, IEEE 1394, или Ethernet.
- Интерфейс SAS использует более высокие уровни напряжений (0,8-1,6 V), чем интерфейс SATA (0,4-0,6 V).
- Из-за более высоких уровней напряжения, интерфейсный кабель SAS может быть длиннее (до 8 м), по сравнению с интерфейсом SATA - до 1 м.

Диски SAS имеют более высокую производительность, чем диски SATA. Скорость вращения единиц SAS находится между 10000 и 15000 оборотов в минуту (RPM), в то время как скорость вращение дисков SATA - между 5400 и 7200 оборотов в минуту. Высокая скорость вращения уменьшает время доступа. Дуплексная связь дисков SAS, также способствует высокой производительности этих единиц по сравнению с SATA дисками. Диски SAS с двумя портами, при подключении к двум контроллерам, также повышает доступность данных. Кроме этого, диски SAS более надежны, чем диски SATA и рассчитаны на гораздо более интенсивное использование. С другой стороны, диски SATA намного дешевле, чем диски SAS. Еще одно преимущество дисков SATA в том, что у них, как правило, значительно больше обьем памяти.

7.7 ¬\{◊ YΣB

USB (Universal Serial Bus — универсальная последовательная шина) является промышленным стандартом расширения архитектуры PC, ориентированным на интеграцию с телефонией и устройствами бытовой электроники. Версия 1.0 была опубликована в январе 1996 года. Первоначально шина обеспечивала две скорости передачи информации: *полную скорость FS (Full Speed)* – 12 Мбит/с и *низкую скорость LS (Low Speed)* – 1,5 Мбит/с.

Весной 2000 года опубликована спецификация USB 2.0 в которой определена и *высокая скорость HS (High Speed) – 480 Мбит/с.* В 2008 была представлена Intel спецификация USB 3.0, в которой определена сверхскорость SS (Super Speed) – 4.8 Gbps. В одной и той же системе могут присутствовать и одновременно работать устройства со всеми скоростями. Шина позволяет соединить до 127 устройств, удаленные от компьютера на растоянии до 25 м (с использованием промежуточных хабов) и длиной одного сегмента кабеля до 5 м.

Архитектура USB определяется следующими критериями:

- Легко реализуемое расширение периферии PC.
- Дешевое решение, поддерживающее скорость передачи.
- Полная поддержка в реальном времени передачи аудио- и видеоданных.
- Гибкость протокола смешанной передачи изохронных данных и асинхронных сообщений.
- Интеграция с выпускаемыми устройствами.
- Доступность в PC всех конфигураций и размеров.
- Обеспечение стандартного интерфейса, который быстро завоевал рынок.
- Создание новых классов устройств, расширяющих PC.

С точки зрения конечного пользователя, привлекательны следующие черты USB:

- Простота кабельной системы и подключений.
- Скрытие подробностей электрического подключения от конечного пользователя.
- Самоидентифицирующиеся ПУ, автоматическая связь устройств с драйверами и конфигурирование.
- Возможность динамического подключения и конфигурирования ПУ.

С середины 1996 года выпускается PC со встроенным контроллером USB, реализуемым чипсетом.

∇∫□ ∫∫∫□◊ YΣB

USB обеспечивает одновременный обмен данными между хост-компьютером и множеством периферийных устройств (ПУ). Распределение пропускной способности шины между ПУ планируется хостом и реализуется им с помощью посылки маркеров. Шина позволяет подключать, конфигурировать, использовать и отключать устройства во время работы хоста и самих устройств.

Ниже приводится вариант перевода терминов из спецификации «Universal Serial Bus Specification. Revision 1.0, January 15, 1996», опубликованной Compaq, DEC, IBM, Intel, Microsoft, NEC и Northern Telecom. Более подробную и оперативную информацию можно найти по адресу: http://www.usb.org.

Устройства (Device) USB могут являться хабами, функциями или их комбинацией. Хаб (Hub) обеспечивает дополнительные точки подключения устройств к шине. Функции (Function) USB предоставляют системе дополнительные возможности, например подключение к ISDN, цифровой джойстик, акустические колонки с цифровым интерфейсом и т.д. *Комбинированное устройство* (compound device), реализующее несколько *функций,* представляется как хаб с подключенными к нему несколькими устройствами (рис. 7.14).

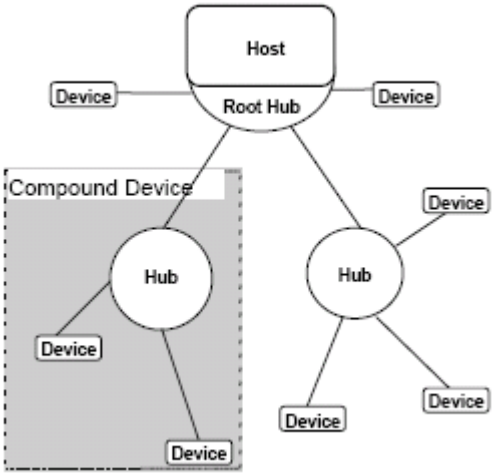


Рисунок 7.14 - Топология устройств по шине USB

Устройство USB должно иметь интерфейс USB, обеспечивающий полную поддержку протокола USB, выполнений стандартных операций (конфигурирование и сброс) и

предоставление информации, описывающей устройство. Устройства, подключаемые к USB, могут иметь в своем составе и хаб, и функции. Работой всей системы USB управляет хост-контроллер (Host Controller), являющийся программно-аппаратной подсистемой хост-компьютера.

Шина USB является хост-центрической: единственным ведущим устройством, которое управляет обменом, является хост-компьютер, а все присоединенные к ней периферийные устройства — исключительно ведомые. В этом она отличается от шины FireWire, где все устройства равноправны.

Физическая топология шины USB — многоярусная звезда (рис. 7.15). Ее вершиной является хост-контроллер, объединенный с корневым хабом (root hub), как правило, двухпортовым. Хаб является устройством-разветвителем.

Логически устройство, подключенное к любому хабу USB и сконфигурированное (см. ниже), может рассматриваться как непосредственно подключенное к хост-контроллеру.

Функции представляют собой устройства, способные передавать и/или принимать данные или управляющую информацию по шине. Типично функции представляют собой отдельные ПУ с кабелем, подключаемым к порту хаба. Физически в одном корпусе может быть несколько функций со встроенным хабом, обеспечивающим их подключение к одному порту. Эти комбинированные устройства для хоста являются хабами с постоянно подключенными устройствами-функциями.

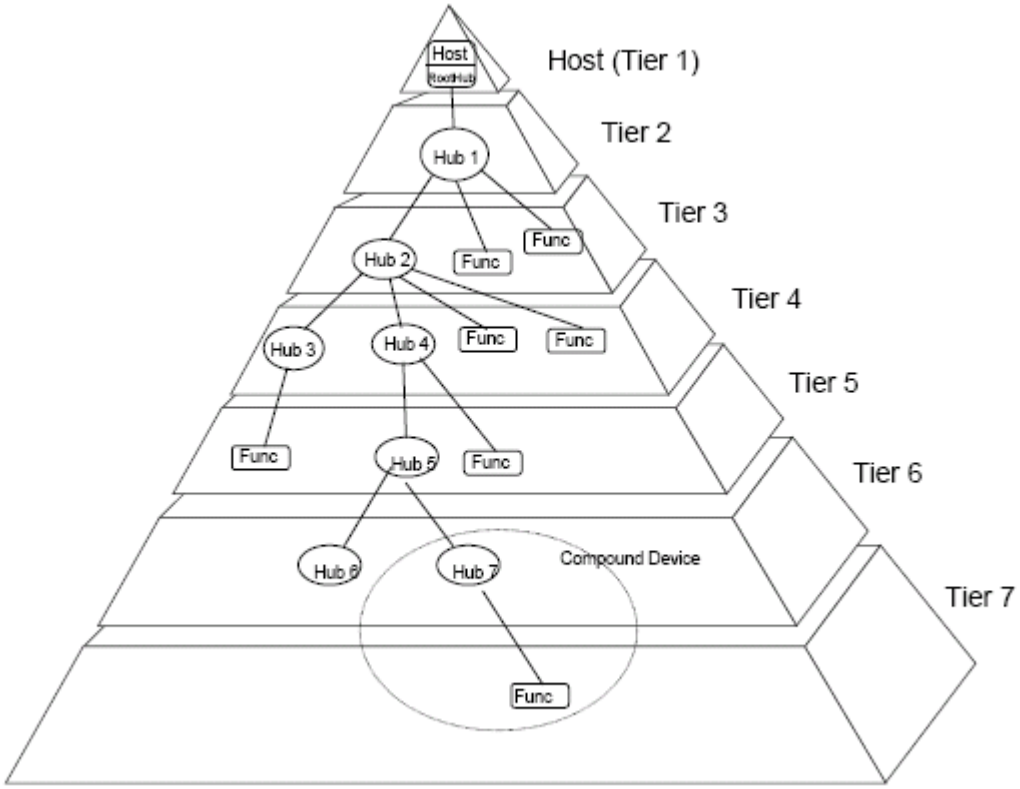


Рисунок 7.15 – Топология шины USB

Каждая функция предоставляет конфигурационную информацию, описывающую возможности ПУ и требования к ресурсам. Перед, использованием функция должна быть сконфигурирована хостом – ей должна быть выделена полоса в канале и выбраны опции конфигурации.

Примерами функций являются:

- указатели — мышь, планшет, световое перо.
- Устройства ввода — клавиатура, сканер.
- Устройство вывода — принтер.
- Устройство хранения данных – внешний hard диск.

Хаб — ключевой элемент системы PNP в архитектуре USB. Хаб является кабельным концентратором. *Хаб* (hub) только обеспечивает дополнительные точки подключения устройств к шине. Точки подключения называются портами хаба. Каждый хаб преобразует одну точку подключения в их множество. Архитектура допускает соединение нескольких хабов.

К каждому порту хаба может непосредственно подключаться периферийное устройство или промежуточный хаб; шина допускает до 5 уровней каскадирования хабов (не считая корневого).

Поскольку комбинированные устройства внутри себя содержат хаб, их подключение к хабу 6-го яруса уже недопустимо. Каждый промежуточный хаб имеет несколько *нисходящих* (downstream) портов для подключения периферийных устройств (или нижележащих хабов) и один *восходящий* (upstream) порт для подключения к корневому хабу или нисходящему порту вышестоящего хаба (рис. 7.16).

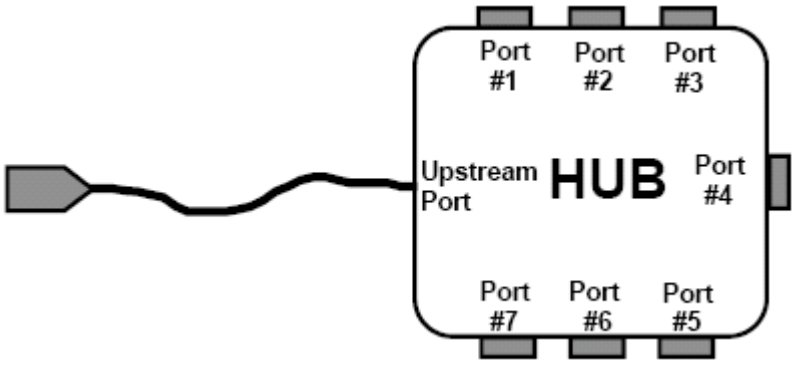


Рисунок 7.16 – Хаб USB

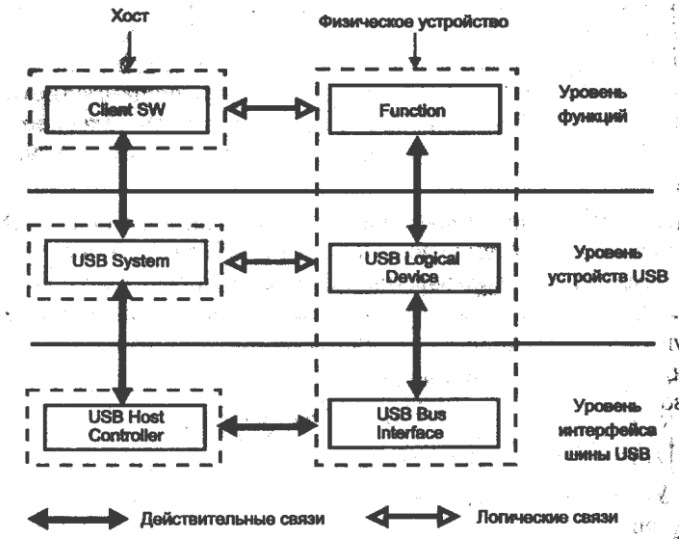
Хаб может распознать подключение устройств к портам или отключение от них и управлять подачей питания на их сегменты. Каждый из портов может быть разрешен или запрещен и сконфигурирован на полную или ограниченную скорость обмена. Хаб обеспечивает изоляцию сегментов с низкой скоростью от высокоскоростных.

Хабы могут управлять подачей питания на нисходящие порты; предусматривается установка ограничения на ток, потребляемый каждым портом.

Система USB разделяется на три уровня с определенными правилами взаимодействия. Устройство USB содержит интерфейсную часть, часть устройства и функциональную часть. Хост тоже делится на три части — интерфейсную, системную и ПО устройства. Каждая часть отвечает только за определенный круг задач, логическое и реальное взаимодействие между ними иллюстрирует рис. 7.17.

В рассматриваемую структуру входят следующие элементы:

- Физическое устройство USB — устройство на шине, выполняющее функции, интересующие конечного пользователя.
- Client SW — ПО, соответствующее конкретному устройству, исполняемое на хост-компьютере. Может являться составной частью ОС или специальным продуктом.
- USB System SW — системная поддержка USB, независимая от конкретных устройств и клиентского ПО.
- USB Host Controller — аппаратные и программные средства для подключения устройств USB к хост-компьютеру



∠\) ∫{ || 7.17 – ℜ| ◊\∫ [™∑⌠)∫®\∑ | [∫ | |{∑{∫[® Y∑B

Физический интерфейс

Стандарт USB определяет электрические и механические спецификации шины.

Информационные сигналы и питающее напряжение 5 В передаются по четырехпроводному кабелю. Используется дифференциальный способ передачи сигналов D+ и D- по двум проводам.

Сигналы синхронизации кодируются вместе с данными по методу NRZI (Non Return to Zero Invert), его работу илюстрирует рисунок 7.18.

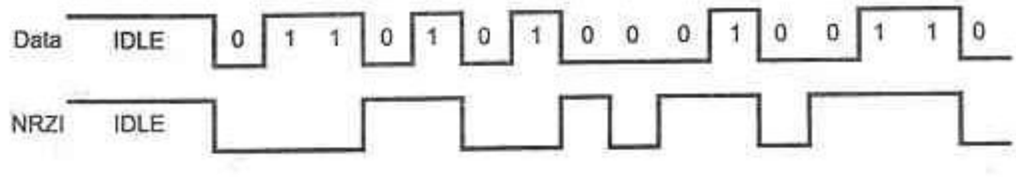


Рисунок 7.18 - Кодирование по методу NRZI

Каждому пакету предшествует поле синхронизации SYNC, позволяющее приемнику настроиться на частоту передатчика.

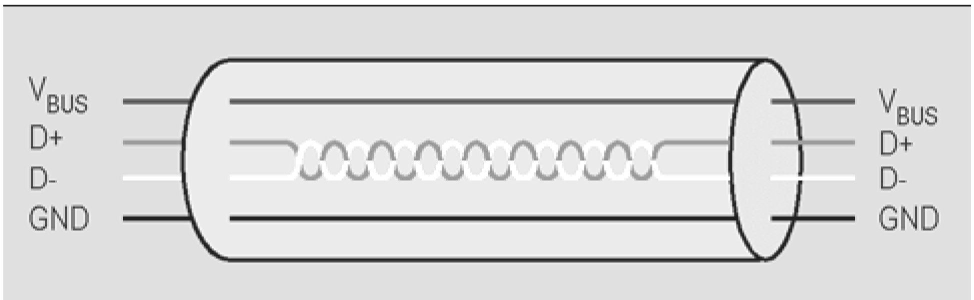
Кабель также имеет линии Vbus и GND для передачи питающего напряжения 5В к устройствам.

Стандарт определяет два типа разьемов (см. табл. 7.4 и рис. 7.20).

Таблица 7.4 - Назначение выводов разьема USB

Контакт	Цепь	Контакт	Цепь
1	Vbus	3	D+
2	D-	4	GND

[illegible] YΣB 1.0 [illegible] YΣB [illegible] 12 [illegible], [illegible] 1,5 [illegible]. [illegible] YΣB 1.0 [illegible] (рис. 7.19) [illegible].



[illegible] 7.19 [illegible] YΣB

[illegible] (480 [illegible]) [illegible] 2 [illegible] Γιγαβιτ Ετηερνετ) [illegible].

[illegible] YΣB [illegible].

[illegible] YΣB 2.0 [illegible] HΣ (480 [illegible]).

[illegible] YΣB [illegible].

[illegible] YΣB [illegible]:

- [illegible] (рис. 7.20, [illegible]) [illegible].
- [illegible]

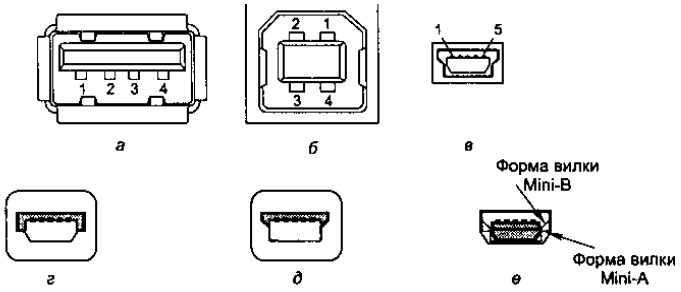
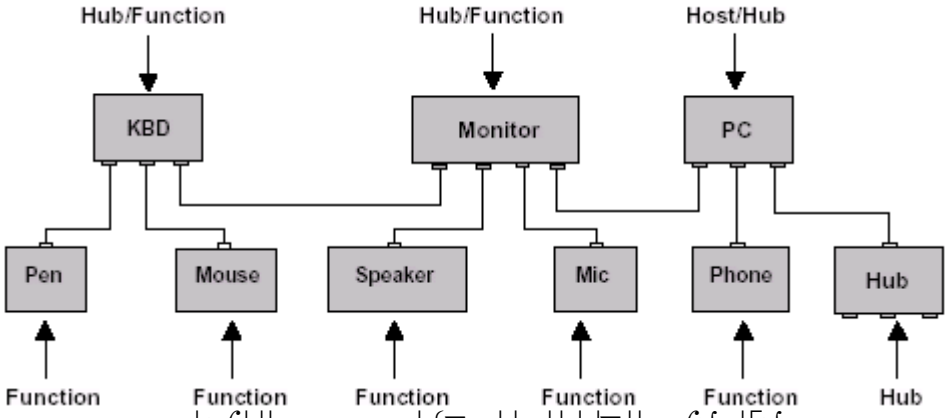


Рисунок 7.20 - Разъемы USB: а — гнездо А, б — гнездо В, в — гнездо mini-B, г — вилка mini-B, д — вилка mini-A, е — гнездо mini-AB

- Mini-B. Разъемы этого типа (рис. 7.20, в, г) используются для отсоединяемых шнуров малогабаритных устройств.
- Mini-A. Этот тип (рис. 7.20, д) введен в спецификации OTG, вилки используются для подключения к портам малогабаритных устройств с гнездом mini-AB.
- Mini-AB. Гнезда (рис. 7.20, е) введены в спецификации OTG для портов двухролевых устройств, которые могут вести себя как хост (если в гнездо вставлена вилка mini-A) или как периферийное устройство (если в гнездо вставлена вилка mini-B).

Питание устройств USB возможно от кабеля (Bus-Powered Devices) или от собственного блока питания (Self-Powered Devices). Хост обеспечивает питанием непосредственно подключенные к нему ПУ. Каждый хаб, в свою очередь, обеспечивает питание устройств, подключенных к его нисходящим портам. При некоторых ограничениях топологии допускается применение хабов, питающихся от шины. На рис. 7.21 приведен пример схемы соединения устройств USB. Здесь клавиатура, перо и мышь могут питаться от шины.



∠\› ∫‹ ‖ 7.21 – ∉□\⌠∑□ | |™|└J |∑‹\□ ⌠›∫□|⌠›∫® YΣB

В спецификации 2.0 скорость 480 Мбит/с должна поддерживать предыдущие, но при таком соотношении скоростей обмены на FS и LS неэффективно заполнят возможную полосу пропускания шины. Чтобы этого не происходило, хабы USB 2.0 приобретают черты коммутаторов пакетов. Если к порту такого хаба подключено высокоскоростное устройство (или аналогичный хаб), то хаб работает в режиме повторителя и транзакция с устройством на HS занимает весь канал до хост-контроллера на все время своего выполнения, Если же к порту хаба USB 2.0 подключается устройство или хаб версии 1.1, то по части канала от контроллера пакет проходит на скорости HS, запоминается в буфере хаба, а к старому устройству или хабу идет уже на его скорости FS или LS.

При планировании соединений следует учитывать способ питания устройств: устройства, питающиеся от шины, как правило, подключают к хабам, питающимся от сети. К хабам, питающимся от шины, подключают лишь маломощные устройства — так, к клавиатуре USB, содержащей внутри себя хаб, подключают мышь USB и другие устройства-указатели (трекбол, планшет).

Хост опрашивает все устройства и выдает им разрешения на передачу данных (рассылая для этого пакет-маркер - Token Packet). Таким образом, устройства лишены возможности непосредственного обмена данными - все данные проходят через хост. Это условие сильно мешало внедрению интерфейса USB на рынок портативных устройств. В результате в конце 2001 года было принято дополнение к стандарту USB 2.0 - спецификация USB OTG (On-The-Go), предназначенная для соединения периферийных USB-устройств друг с другом без необходимости подключения к хосту (например, цифровая камера и фотопринтер). Устройство, поддерживающее USB OTG, способно частично выполнять функции хоста и распознавать, когда оно подключено к полноценному хосту (на основе ПК), а когда - к другому периферийному устройству. Спецификация описывает также протокол согласования выбора роли хоста при соединении двух USB OTG-устройств.

Следует также отметить, что разными производителями предлагались спецификации, описывающие интерфейс различных аппаратных реализаций контроллера USB. Фирмой Intel была предложена спецификация UHCI (Universal Host Controller Interface), которая предусматривает чрезвычайно простую аппаратную реализацию контроллера USB. В рамках данной спецификации основные функции контроля и арбитража шины возлагаются на программный драйвер. Альтернативная спецификация была предложена компаниями Compaq, Microsoft и National Semiconductor - OHCI (Open Host Controller Interface). Контроллеры по спецификации OHCI обладают унифицированным абстрактным интерфейсом, предусматривающим аппаратную реализацию большинства управляющих функций, что облегчает их программирование.

⊂⌊™∑⌋ /∑□∑™◊∥(™◊{{⌋⌋

Каждое устройство USB представляет собой набор независимых конечных тачек (Endpoint), с которыми хост-контроллер обменивается информацией. Конечные точки описываются следующими параметрами:

- требуемой частотой доступа к шине и допустимыми задержками обслуживания;
- требуемой полосой пропускания канала;
- номером точки;
- требованиями к обработке ошибок;
- максимальными размерами передаваемых и принимаемых пакетов;
- типом обмена;
- направлением обмена (для сплошного и изохронного обменов).

⊇◊(™⌊∑ ⌠⟩∫□⌊⌠⟩∫®⌊ ⌊⟨□⌋◊∫∑⌊⌉{⌊⌊⌠∑∑∫⌋⌊{∑ĸ ⌠⌋ ∫⌊∥ ⌠⟩{⌊⌠∑□⌊⌠ 0, ⟩⌋⌋⌊⌉⌋ ⌠∑⌠ ⌠⌋ ™⌊□⌊{⌉⌊◊⌊ ⌋◊⟩⌊⌊, ⌊⟨⌉∑©⌊ ⌠⌋□◊®⌊∑{⌊□⌊⌊⌋□⌊⟩◊ ∑©⌊⟩⌊⟩∫⌊□{⌊□. ⇑⌋∫◊∫⌊∥◊®⟩∑©™◊⟩∥ ⌊{ ⌋⌊© ⌠□⌊□⌊®◊{◊⌋□⌊®⌋⌊⌋ ⌋∑{⌊⌊⌋⌊⌋◊{⌊□⌊⌊™⌋⌊⌋ ⌋∑{⌊⌊ ⌠⟩∫□⌊⌠⟩∫®◊⌋ ⌋⌊{∑. ∈{⌊⌋⌊™™∑□⌠⌊®◊∑∫⌋∑□∑™◊∥⌊⌋∫⌊⌋◊↔⌠⌋□◊®⌊∑{⌊∑≈ ()⌠. ™◊⌊∑∑).

Кроме нулевой точки, устройства-функции могут иметь дополнительные точки, реализующие полезный обмен данными. Низкоскоростные устройства могут иметь до двух дополнительных точек, полноскоростные — до 16 точек вводя и 16 точек вывода (протокольное ограничение). Точки не могут быть использованы до их конфигурирования (установления согласованного с ними канала).

Каналом (Pipe) в USB называется модель передачи данных между хост-контроллером и конечной точкой (Endpoint) устройства. Это логический канал и иногда сам канал называют конечной точкой (рис. 7.22).

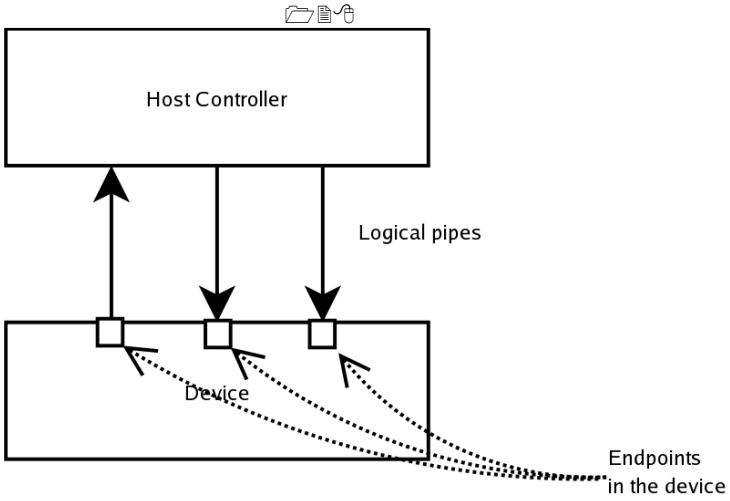


Рисунок 7.22

Имеются два типа каналов: потоки (Stream) и сообщения (Message). Поток доставляет данные от одного конца канала к другому, он всегда однонаправленный. Один и тот же номер конечной точки может использоваться для двух поточных каналов — ввода и вывода. Поток может реализовывать следующие типы обмена: сплошной, изохронный и прерывания. Доставка всегда идет в порядке «первым вошел — первым вышел» (FIFO); с точки зрения USB, данные потока неструктурированы. Сообщения имеют формат, определенный спецификацией USB. Хост посылает запрос к конечной точке, после которого передается (принимается) пакет сообщения, за которым следует пакет с информацией состояния конечной точки. Последующее сообщение нормально не может быть послано до обработки предыдущего, но при отработке ошибок возможен сброс необслуженных сообщений. Двухсторонний обмен сообщениями адресуется к одной и той же конечной точке. Для доставки сообщений используется только обмен типа «управление».

С каналами связаны характеристики, соответствующие конечной точке (полоса пропускания, тип сервиса, размер буфера и т. п.). Каналы организуются при конфигурировании устройств USB. Для каждого включенного устройства существует канал сообщений (Control Pipe Off), по которому передается информация конфигурирования, управления и состояния.

Типы передачи данных

YΣB | ⌊™ᵀᴹ∑□⁄⁄®◊∑∫ | ◊| ⌊™ϟ ⌊ϟ◊|□◊®⌊∑ϟϟ⌋∑, ∫◊| ⌊™® ⁄ϟ◊|□◊®⌊∑ϟϟ⌋∑ □∑⁄⁄⌋⟩®□|⌊. ∉∑□ ∑™◊|◊ ™◊ϟϟ⌋⌋ |□⌊⌊ ®⌊™⌊⟩□ ⁄∑⁄™ ⁄ ∉∈ ⌋ ⌊⟩∫◊⌊| ⌊ϟ∑ ϗ ⌊⌈ ∫⌊‖ ⌊⌈ ⁄⟩∫□⌊⌈⟩∫®◊. ©⟩∫□⌊⌈⟩∫®⌊ ⁄⌊⁄∑∫ ⌊⁄∑∫ ⌉ϟ∑⟩| ⌊⌊⌉| ⌊| ⌊ϟ∑ϗ⌋⌋ ∫⌊ |∑| , ⟩®□| ⌉ ⟩| ◊⁄™⌊⌈ ⌊⌋ ϟ⌊⌋ (| ◊ϟ◊⌊) ⁄⟩∫◊ϟ◊®⌊⌊®◊∑∫⟩□ ϟ∑| ◊®⌊⟩⌊⁄⌊.

ℵ□⌋ ⌊∑| ∫ ⁄□◊ YΣB ™⌊| ⁄⟩| ◊∑∫ |∑∫⌋□∑ ⟨◊| ⌊®⌋⌋ ∫⌊| ◊ |∑□∑™◊ ⌊⌊ ™◊ϟϟ⌋⌋ :

©/□◊®⌊□/ ⁄⁄∑ /⟨⟩⌋⁄/ ⁄ (Хонтрол τρανσφερ), ⌊⟩| ⌊⌊⌉| ⁄⁄∑⁄⌋∑ ™⌊□| ⌊ϟ |⌊© ⁄□⌊□⌊®◊ϟ⌊□ ®⌊ ®□ ∑⁄□ | ⌊™| ⌋⌋ |∑ϟ⌊□⌊⌊® |□⌊⟩∑⟩⟩∑ □◊⟨⌊⌋ ™⌊□ ⁄| □◊®⌊∑ϟ⌊□ ⁄⟩∫□⌊⌈⟩∫®◊⁄⌊. ∉□⌊⌋⌊ ⌊⌊ ⌊⟨∑⟩|∑ |⌊®◊∑∫ ©◊ □◊ϟ∫⌊□⌊®◊ϟϟ ⁄⌋ ™⌊⟩∫◊®| ⁄ ™◊ϟϟ⌋⌋ . ⊗⌊⌊ϟ◊ | ⌊⌊□ ™◊ϟϟ⌋⌋ ⁄| □◊®⌊□⌋ ⁀∑⁄ | ⌊⟩⌋⌊| ⌊ϟ∑ |□∑®⌋◊∑∫ 64 ⟨◊⁄ ∫ϟ◊ | ⌊⌊ϟ ⌊⌈ ⟩| ⌊□⌊⟩∫⌊⌊ 8 ⟨◊⁄∫ϟ◊ϟ⌊⌋| ⌊⌈.

∇|⌊ ⌋⌊ϟ ⌋∑ |∑□∑™◊ ⌊⌊ (Βυλκ Δατα Τρανσφερσ). ∉◊| ∑∫⌋⌊⁄∑⌋∫ | ⌊⌊∑ ™◊ϟϟ⌋⌋ □◊| ⁄∑□⌊⌈ 8, 16, 32 ⌊⌊⌊ 64 ⟨◊⁄∫. ∉□⌊⌊□⌋∫∑∫ ⌋∫⌋⌋ |∑□∑™◊ |⟩◊⁄⌋⌈ ϟ⌋⌋| ⌊⌈, ⌊ϟ⌊ ⌈⌊© ⁄∫ |□⌊⌋⟩∫◊ϟ◊®⌊⌊®◊∫⌋□ |□⌊⟨⌊⌋⁄⌊⌈ ϟ◊ ©□ ⁄⌋| ∑ ⅄ϟ⌋. ⊗⌊| ⁄⟩| ◊⁄∫⟩□ ∫⌊⌊⌋| ⌊ ϟ◊ | ⌊⌊ϟ ⌊⌈ ⟩| ⌊□⌊⟩∫⌊ |∑□∑™◊ ⌊⌊.

∉□∑□⁄®◊ϟ⌊□ *(Ιντερρυπτ)*□| ⌊□⌊⌋| ⌊∑ (™⌊ 64 ⟨◊⁄∫ϟ◊ | ⌊⌊ϟ ⌊⌈ ⟩| ⌊□⌊⟩∫⌊, ™⌊ 8 ⟨◊⁄∫ϟ◊ϟ⌊⌋| ⌊⌈) |∑□ ∑™◊ ⌊⌊ ∫⌊⌊◊ ®®⌊™⌊⌈⌋⌋ ⟩⌊⁄®⌊⌊⌊® ⌊⌊⌊| ⌊⌊□™⌊ϟ◊∫.

Изохронные передачи (Isochronous Transfers) – непрерывные передачи в реальном времени, занимающие предварительно согласованную часть пропускной способности шины и имеющие заданную задержку доставки. В случае обнаружения ошибки изохронные данные передаются без повтора – недействительные пакеты игнорируются. Пример – цифровая передача голоса.

Полоса пропускания шины делится между всеми установленными каналами. Выделенная полоса пропускания закрепляется за каналом, и если установление нового канала требует такой полосы, которая не вписывается в уже существующее распределение, запрос на выделение канала отвергается.

Протокол

Все обмены (транзакции) по USB состоят из трех пакетов. Каждая *транзакция* планируется и начинается по инициативе контроллера, который посылает *пакет-маркер* (Token Packet). Он описывает тип и направление передачи, адрес устройства USB и номер конечной точки.

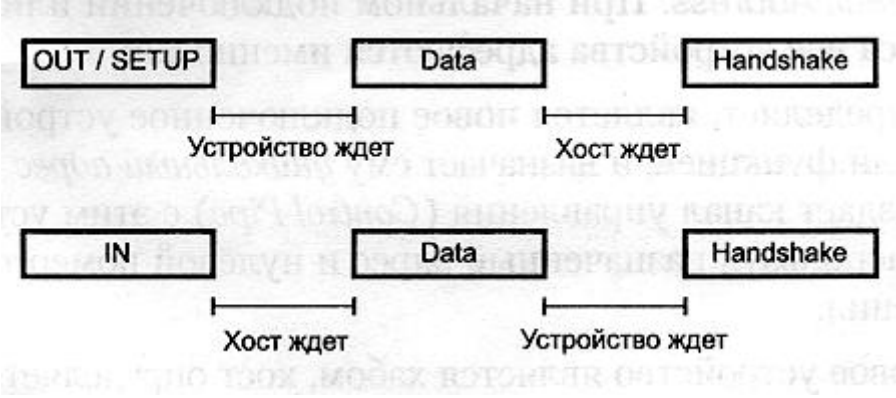


Рисунок 7.23 - Последовательности пакетов

В каждой транзакции возможен обмен только между адресуемым устройством (его конечной точкой) и хостом. Адресуемое маркером устройство распознает свой адрес и готовится к обмену. Источник данных (определенный маркером) передает пакет данных (или уведомление об отсутствии данных, предназначенных для передачи). После успешного приема пакета приемник данных посылает *пакет подтверждения (Handshake Packet).*

Планирование транзакций обеспечивает управление поточными каналами. На аппаратном уровне использование отказа от транзакции (NAck) при недопустимой интенсивности передачи предохраняет буферы от переполнения. Маркеры отвергнутых транзакций повторно передаются в свободное для шины время.

Для обнаружения ошибок передачи каждый пакет имеет контрольные поля CRC-кодов, позволяющие обнаружить все одиночные и двойные битовые ошибки. Аппаратные средства обнаруживают ошибки передачи, а контроллер автоматически производит трехкратную попытку передачи. Если повторы безуспешны, сообщение об ошибке передается клиентскому ПО.

Форматы пакетов

Байты передаются по шине последовательно, начиная с младшего бита. Все посылки организованы в пакеты. Каждый пакет начинается с поля синхронизации *Sync*, которое представляется последовательностью состояний *KJKJKJKK* (кодированную по NRZI), следующую после состояния Idle. Последние два бита (КК) являются маркером начала пакета SOP, используемым для идентификации первого бита идентификатора пакета *PID.* Идентификатор пакета является 4-битным полем PID[3:0], идентифицирующим тип пакета (табл. 7.5), за которым в качестве контрольных следуют те же четыре бита, но инвертированные.

Таблица 7.5 - Типы пакетов и их идентификаторы PID

Тип PID	Имя PID	PID	Содержимое и назначение
Токен	*OUT*	0001	Адрес функции и номер конечной точки - маркер транзакции функции
Token	*IN*	1001	Адрес функции и номер конечной точки – маркер транзакции хоста
Token	*SOF*	0101	Маркер начала кадра
Токен	**SETUP**	1101	Адрес функции и номер конечной точки – маркер транзакции с управляющей точкой
Data	*Data0* *Data1*	0011 1011	Пакеты данных с четным и нечетным PID чередуются для точной идентификации подтверждений
Handshake	*Ack*	0010	Подтверждение безошибочного приема пакета

Handshake	*NAK*	1010	Приемник не сумел принять или передатчик не сумел передать данные. В транзакциях прерываний является признаком отсутствия необслуженных прерываний
Handshake	*STALL*	1110	Конечная точка требует вмешательство хоста
Special	*PRE*	1100	Преамбула передачи на низкой скорости

В пакетах-маркерах IN, SETAP и OUT следующими являются адресные поля: 7-битный адрес функции и 4-битный адрес конечной точки. Они позволяют адресовать до 127 функций USB (нулевой адрес используется для конфигурирования) и по 16 конечных точек в каждой функции.

В пакете SOF имеется 11-битное поле номера кадра (Frame Number Field), последовательно (циклически) увеличиваемое для очередного кадра.

Поле данных может иметь размер от 0 до 1023 целых байт. Размер поля зависит от типа передачи и согласуется при установлении канала. Каждая транзакция инициируется хост-контроллером посылкой маркера и завершается пакетом Handshake. Последовательность пакетов в транзакциях иллюстрирует рис. 7.24.

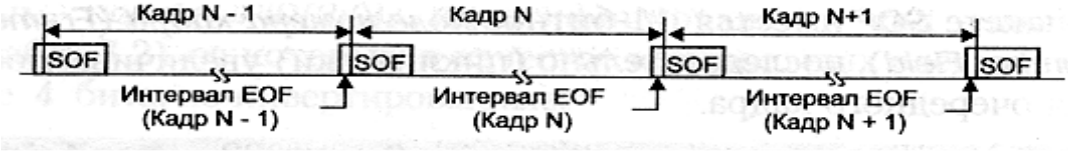


Рисунок 7.24 Поток кадров USB

Хост-контроллер организует обмены с устройствами согласно своему плану распределения ресурсов. Контроллер циклически (с периодом 1 мс) формирует кадры (Frames), в которые укладываются все запланированые транзакции (125 мкс, при скорости 480 Мб/с). Каждый кадр начинается с посылки маркера SOF (Start of frame), который является синхронизирующим сигналом для всех устройств, включая хабы. В конце каждого кадра выделяется интервал времени EOF (End of frame), на время которого хабы запрещают передачу по направлению к контроллеру. Каждый кадр имеет свой номер. Хост-контроллер оперирует 32-битным счетчиком, но в маркере SOF передает только младшие 11 бит. Номер кадра увеличивается (циклически) во время EOF.

Хост планирует загрузку кадров так, чтобы в них всегда находилось место для транзакций управления и прерывания.

∇()∑{(∑/(/(©ſ□(□(®◊{(∑

Y∑B |(™™∑□/(®◊∑∫ ™(◊/((∑)|(∑|(™|L)|∑{(∑(|∫|L)|∑{(∑ſ)∫□|Γ)∫®.⊆ſ(∑□◊)(□ſ)∫□|Γ)∫®八{」□®L□∑∫)□|()∫(□{{」Γ|□()∑))(Γ,(∫)L∑/(®◊)ℸ(Γ Ul(∑{∑{(□|U((∑)|(Γ∫(|(L(©((.

Все устройства подключаются через порты хабов. Хабы определяют подключение и отключение устройств к своим портам и сообщают состояние портов при запросе от контроллера. Хост разрешает работу порта и адресуется к устройству через канал управления, используя нулевой адрес — USB Default Address. При начальном подключении или после сброса все устройства адресуются именно так. Хост определяет, является новое подключенное устройство хабом или функцией, и назначает ему уникальный адрес USB. Хост создает канал управления (ControlPipe) с этим устройством, используя назначенный адрес и нулевой номер точки назначения.

Если новое устройство является хабом, хост определяет подключенные к нему устройства, назначает им адреса и устанавливает каналы. Если новое устройство является функцией, уведомление о подключении передается диспетчером USB соответствующему ПО.

Когда устройство отключается, хаб автоматически запрещает соответствующий порт и сообщает об отключении контроллеру, который удаляет сведения о данном устройстве из всех структур данных. Если отключается хаб, процесс удаления выполняется для всех подключенных к нему устройств. Если отключается функция, уведомление посылается соответствующему ПО.

Нумерация устройств, подключенных к шине (Bus Enumeration), осуществляется динамически по мере их подключения (или включения их питания) без какого-либо вмешательства пользователя или клиентского ПО. Процедура нумерации выполняется следующим образом:

1. Хаб, к которому подключилось устройство, информирует хост о смене состояния своего порта ответом на опрос состояния. С этого момента устройство переходит в состояние Attached (подключено), а порт, к которому оно подключилось, в состояние Disabled.
2. Хост уточняет состояние порта.
3. Узнав порт, к которому подключилось новое устройство, хост дает команду сброса и разрешения порта.
4. Хаб формирует сигнал Reset для данного порта (10 мс) и переводит его в состояние Enabled. Подключенное устройство может потреблять от шины ток питания до 100 мА. Устройство переходит в состояние Powered (питание подано), все его регистры переводятся в исходное состояние, и оно отзывается на обращение по нулевому адресу.
5. Пока устройство не получит уникальный адрес, оно доступно по дежурному каналу, по которому хост-контроллер определяет максимально допустимый размер поля данных пакета.
6. Хост сообщает устройству его уникальный адрес, и оно переводится в состояние Addressed (адресовано).
7. Хост считывает конфигурацию устройства, включая заявленный потребляемый ток от шины. Считывание может затянуться на несколько кадров.
8. Исходя из полученной информации, хост конфигурирует все имеющиеся конечные точки данного устройства, которое переводится в состояние Configured (сконфигурировано). Теперь хаб позволяет устройству потреблять от шины полный ток, заявленный в конфигурации. Устройство готово.

Когда устройство отключается от шины, хаб уведомляет об этом хост и работа порта запрещается, а хост обновляет свою текущую топологическую информацию.

Несколько характеристик интерфейса USB 3.0. Этот стандарт совместим с USB 2.0.

Основные характеристики:

- Максимальная скорость передачи составляет 4.8 Gbps (USB 2.0 Hi-Speed: 480 Mbps);
- Передача информации происходит по 2 независимым каналам (передача двухстороняя – используются каналы LVDS) – реализован дуплексный канал передачи данных.
- Обеспечивает максимальный ток - 900 mA (USB 2.0 - 500 mA).

На следующем рисунке преставлен разьем стандарта USB3.0