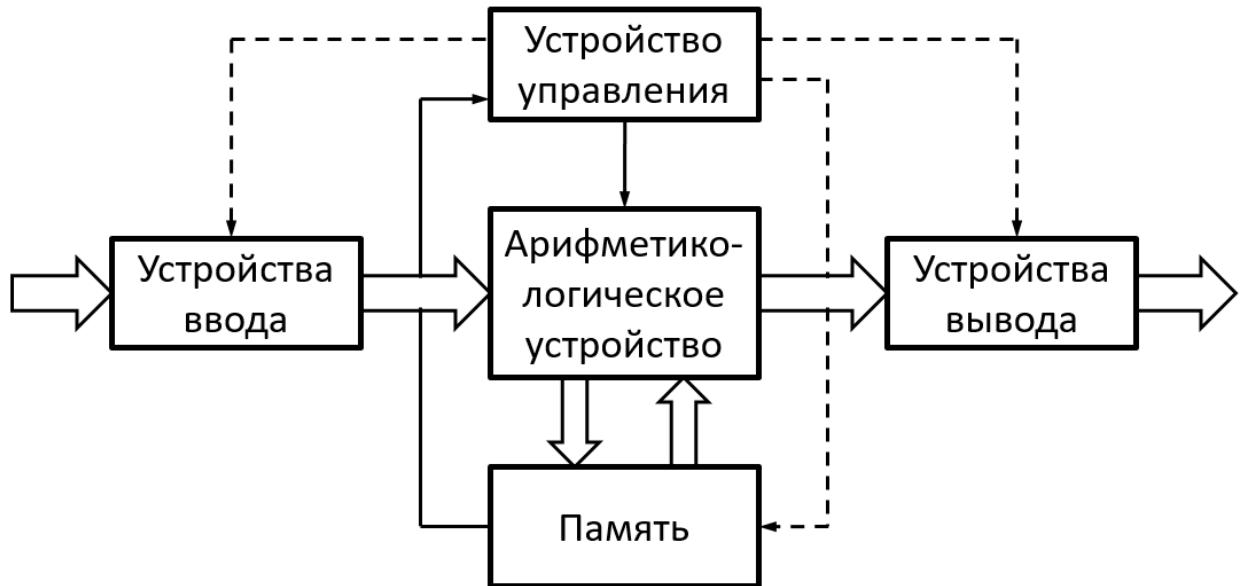


1. Каноническая функциональная структура ЭВМ Джона фон Неймана, архитектурные принципы.

Каноническая функциональная структура ЭВМ Джона фон Неймана



⇒ линии связи для данных; → линии связи для команд; — — → линии связи для управления

При работе ВМ наиболее интенсивное взаимодействие осуществляется между АЛУ и устройством управления. С развитием элементной базы в целях повышения производительности за счет уменьшения задержек в связях эти устройства объединили в один блок, называемый процессором. Процессор считывает и выполняет команды программы, организует обращение к оперативной памяти (ОП), в нужных случаях инициирует работу устройств ввода и вывода. Устройство ввода преобразует входные сигналы в сигналы, принятые для представления данных на шине, соединяющей устройство ввода с АЛУ.

В памяти хранятся команды и данные, которыми оперирует процессор. В нее же записываются результаты промежуточных вычислений. Результаты выполнения программы поступают в устройство вывода.

Устройство вывода преобразует выходные сигналы в форму, удобную для восприятия человеком (тексты, графические образы и др.).

Выборка команды из памяти и ее выполнение циклически повторяются. Цикл включает следующие фазы: выборку, дешифрацию и исполнение.

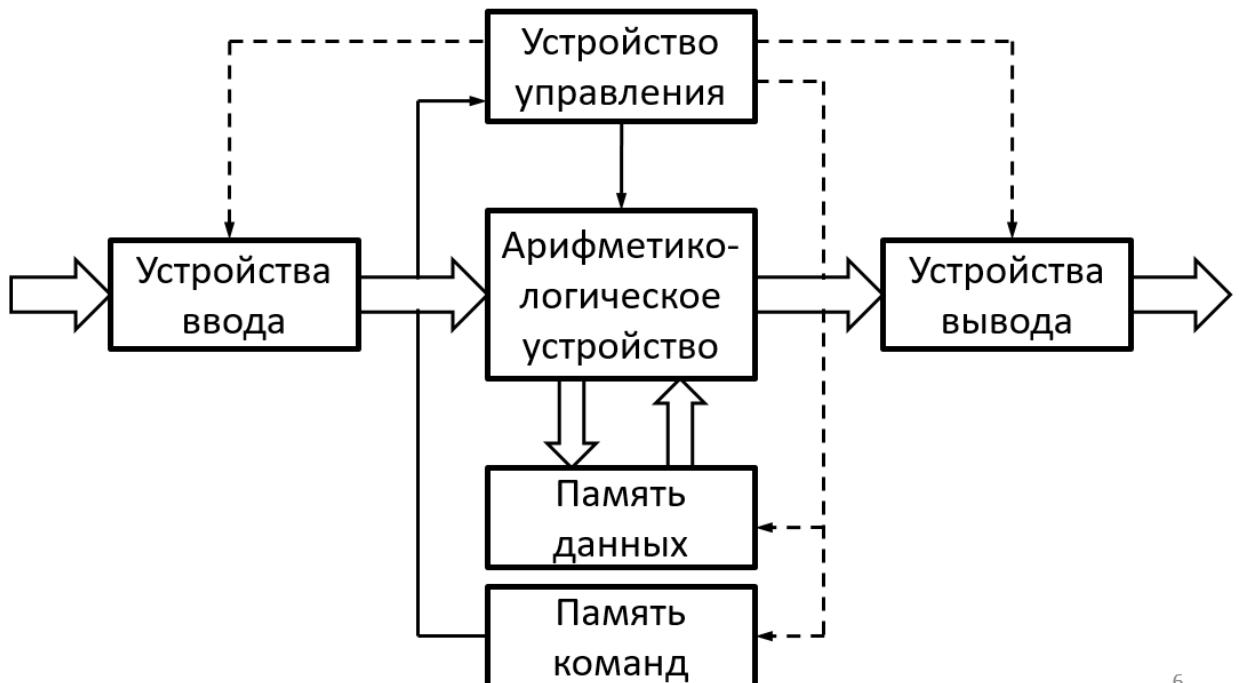
В фазе выборки команды содержимое ячеек памяти, адресуемое специальной схемой устройства управления (программным счетчиком), выбирается из памяти и помещается в специальный регистр процессора, называемый регистром команд. В фазе дешифрации код команды из регистра команд поступает в дешифратор команд процессора и преобразуется в последовательность управляющих сигналов, обеспечивающих настройку АЛУ для выполнения функции, определяемой командой. После этого в устройстве управления формируется адрес следующей команды.

Следующей является фаза исполнения. После выполнения команды происходит возврат к первой фазе — выборке следующей команды.

Архитектурные принципы построения ЭВМ Джона фон Неймана

- Использование двоичной системы счисления
- Последовательное программное управление. Машина выполняет вычисления по программе. Программа состоит из набора команд, которые исполняются автоматически друг за другом в определенной последовательности.
- Возможность условного перехода
- Однородность памяти. Команды и данные хранятся в одной и той же памяти и внешне в памяти неразличимы. Распознать их можно только по способу использования. Это позволяет производить над командами те же операции, что и над числами, и, соответственно, открывает ряд возможностей.
- Линейно-адресная организация памяти. Структурно основная память состоит из пронумерованных ячеек, причем процессору в произвольный момент доступна любая ячейка. Двоичные коды команд и данных разделяются на единицы информации, называемые словами, и хранятся в ячейках памяти, а для доступа к ним используются номера соответствующих ячеек — адреса.
- Иерархичность запоминающих устройств. Трудности реализации единого емкого быстродействующего запоминающего устройства требуют иерархического построения памяти. Должно быть по меньшей мере два уровня иерархии: основная память и внешняя память.

2. Варианты архитектур ЭВМ: принстонская и гарвардская.



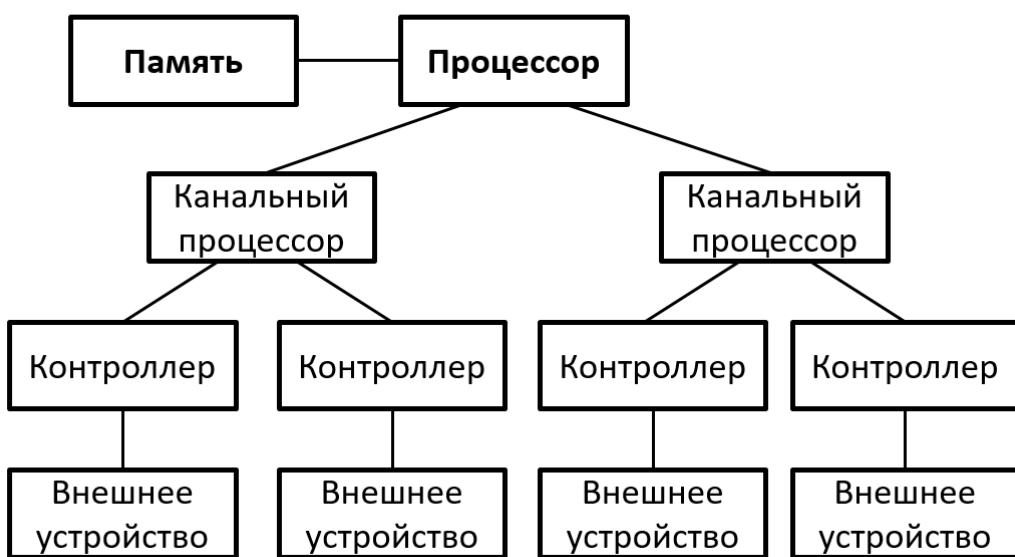
Когда в микропроцессорах впервые стали применять внутреннюю кэш-память, ее обычно использовали как для команд, так и для данных. Такую кэш-память принято называть смешанной, а соответствующую архитектуру — Принстонской (Princeton architecture), по названию университета, где разрабатывались ВМ с единой памятью для команд и данных, то есть соответствующие классической архитектуре фон-Неймана.

Сравнительно недавно стало обычным разделять кэш-память на две — отдельно для команд и отдельно для данных. Подобная архитектура получила название Гарвардской (Harvard architecture), поскольку именно в Гарвардском университете был создан компьютер «Марк-1» (1950 год), имевший раздельные ЗУ для команд и данных.

Смешанная кэш-память обладает тем преимуществом, что при заданной емкости ей свойственна более высокая вероятность попаданий, так как в ней оптимальный баланс между командами и данными устанавливается автоматически. Так, если в выполняемом фрагменте программы обращения к памяти связаны в основном с выборкой команд, а доля обращений к данным относительно мала, кэш-память имеет тенденцию насыщаться командами, и наоборот. С другой стороны, при раздельной кэш-памяти выборка команд и данных может производиться одновременно, при этом исключаются возможные конфликты. Последнее обстоятельство существенно в системах, использующих конвейеризацию команд, где процессор извлекает команды с опережением и заполняет ими буфер или конвейер. Раздельная схема обычно характерна для кэш-памяти первого уровня, причем кэш-память команд и кэш-память данных могут быть организованы по-разному, например, в них могут отличаться методы отображения или замещения информации.

3. Варианты структур ЭВМ: иерархическая и магистральная.

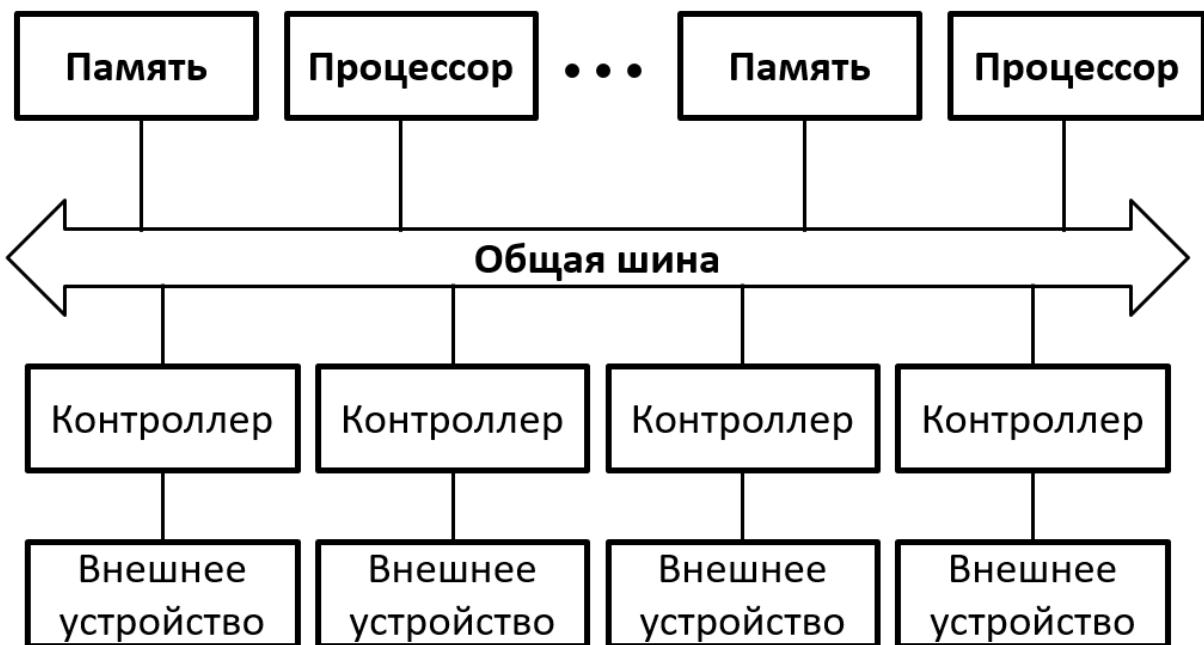
Иерархическая структура



Типичным представителем первого способа может служить классическая фон-неймановская ВМ. В ней между взаимодействующими устройствами (процессор, память, устройство ввода/вывода) имеются непосредственные связи. Особенности связей (число линий в шинах, пропускная способность и т. п.) определяются видом информации, характером и

интенсивностью обмена. Достоинством архитектуры с непосредственными связями можно считать возможность развязки «узких мест» путем улучшения структуры и характеристик только определенных связей, что экономически может быть наиболее выгодным решением. У фон-неймановских ВМ таким «узким местом» является канал пересылки данных между ЦП и памятью и «развязать» его достаточно непросто. Кроме того, ВМ с непосредственными связями плохо поддаются реконфигурации.

Магистральная структура



В варианте с общей шиной все устройства вычислительной машины подключены к магистральнойшине, служащей единственным трактом для потоков команд, данных и управления. Наличие общей шины существенно упрощает реализацию ВМ, позволяет легко менять состав и конфигурацию машины. Благодаря этим свойствам шинная архитектура получила широкое распространение в мини- и микро-ЭВМ. Вместе с тем, именно с шиной связан и основной недостаток архитектуры: в каждый момент передавать информацию по шине может только одно устройство. Основную нагрузку на шину создают обмены между процессором и памятью, связанные с извлечением из памяти команд и данных и записью в память результатов вычислений. На операции ввода/вывода остается лишь часть пропускной способности шины. Практика показывает, что даже при достаточно быстрой шине для 90 % приложений этих остаточных ресурсов обычно не хватает, особенно в случае ввода или вывода больших массивов данных.

4. Классификация архитектур многопроцессорных вычислительных систем по Флинну.

Поток данных		Поток команд	
		Одиночный	Множественный
Одиночный		<i>SISD</i>	<i>MISD</i>
Множественный		<i>SIMD</i>	<i>MIMD</i>

В ее основу положено понятие потока, под которым понимается последовательность элементов, команд или данных, обрабатываемая процессором. В зависимости от количества потоков команд и потоков данных Флинн выделяет четыре класса архитектур: SISD, MISD, SIMD, MIMD.

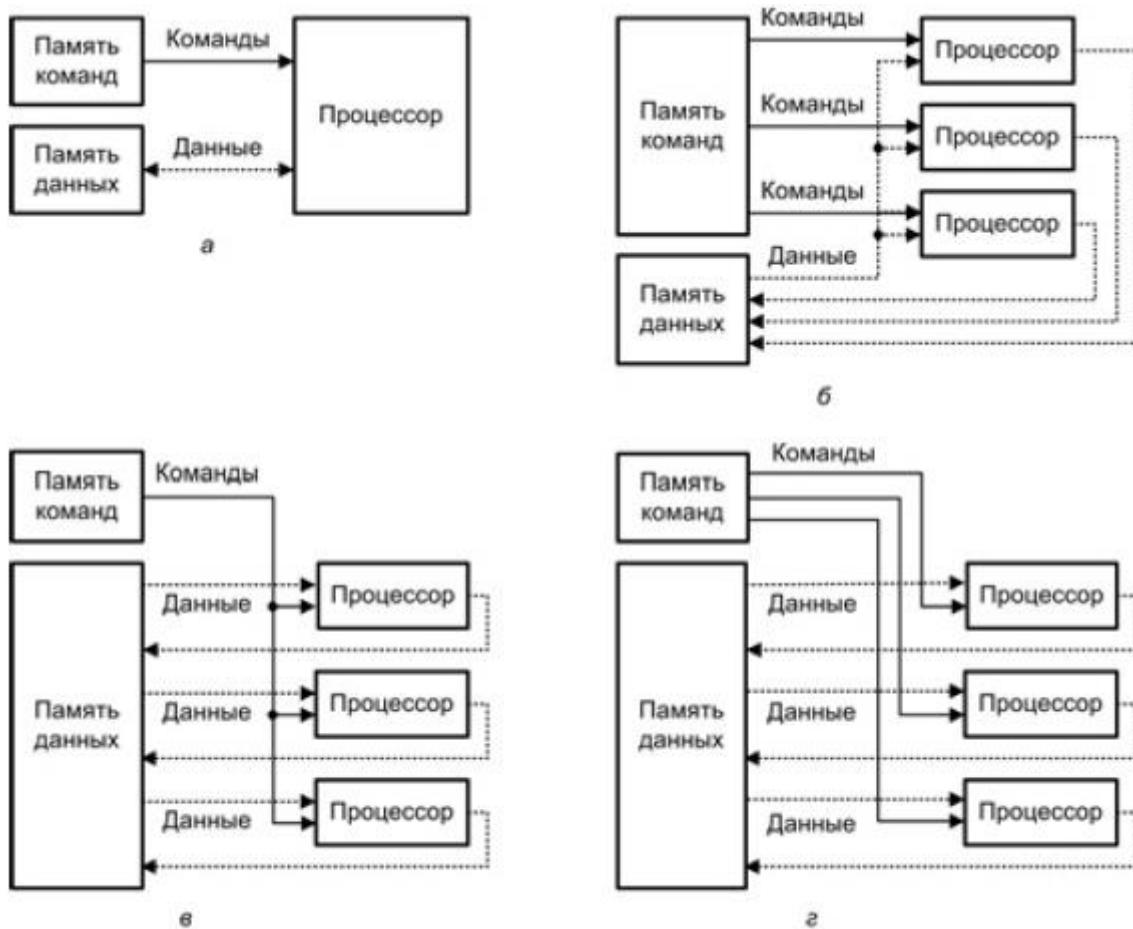


Рис. 10.9. Архитектура вычислительных систем по Флинну: а — SISD; б — MISD; в — SIMD; г — MIMD

SISD (SingleInstructionStream/SingleDataStream) — одиночный поток команд и одиночный поток данных. Представителями этого класса являются, прежде всего, классические фон-неймановские ВМ, где имеется только один поток команд, команды обрабатываются последовательно, и каждая команда инициирует одну операцию с одним потоком данных.

MISD (MultipleInstructionStream/SingleDataStream) — множественный поток команд и одиночный поток данных. Из определения следует, что в архитектуре ВС присутствует

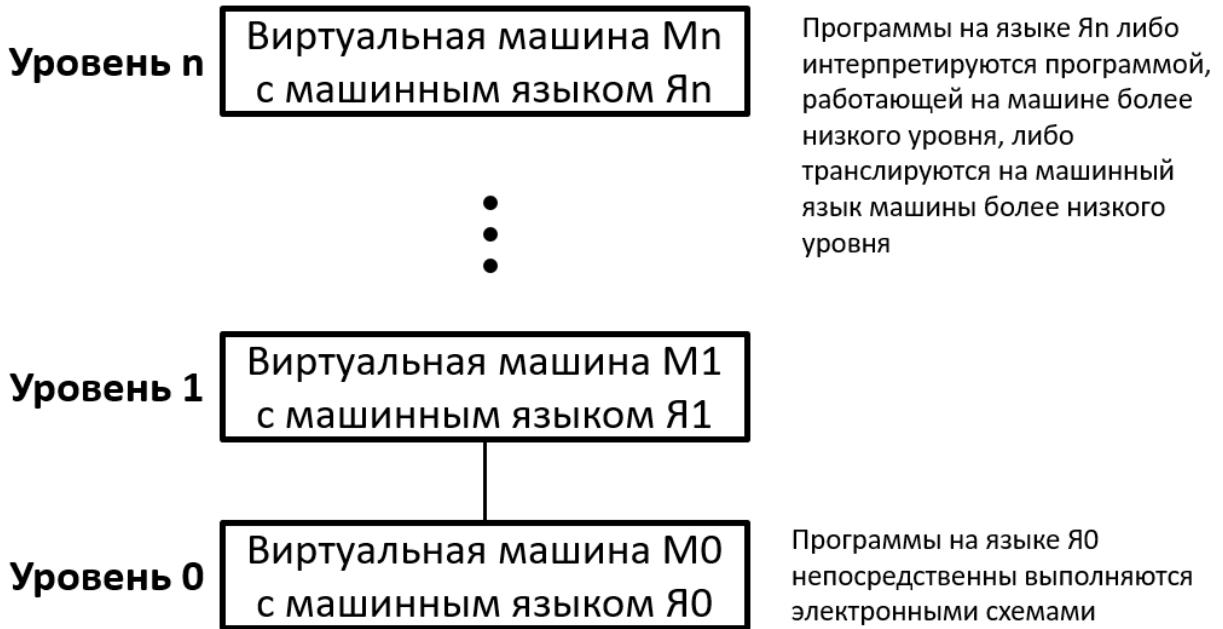
множество процессоров, обрабатывающих один и тот же поток данных. Примером могла бы служить ВС, на процессоры которой подается искаженный сигнал, а каждый из процессоров обрабатывает этот сигнал с помощью своего алгоритма фильтрации. Тем не менее ни Флинн, ни другие специалисты в области архитектуры компьютеров до сих пор не сумели представить убедительный пример реально существующей вычислительной системы, построенной на данном принципе.

SIMD (SingleInstructionStream/MultipleDataStream) — одиночный поток команд и множественный поток данных. ВМ данной архитектуры позволяют выполнять одну арифметическую операцию сразу над многими данными — элементами вектора. Бесспорными представителями класса SIMD считаются матрицы процессоров, где единое управляющее устройство контролирует множество процессорных элементов.

MIMD (MultipleInstructionStream/MultipleDataStream) — множественный поток команд и множественный поток данных. Класс предполагает наличие в вычислительной системе множества устройств обработки команд, объединенных в единый комплекс и работающих каждое со своим потоком команд и данных.

5. Многоуровневая модель функционирования ЭВМ.

Многоуровневая модель функционирования ЭВМ



Большинство современных компьютеров состоит из двух и более уровней.

Уровень 0 — аппаратное обеспечение машины. Его электронные схемы выполняют программы, написанные на языке уровня 1. Ради полноты нужно упомянуть о существовании еще одного уровня, расположенного ниже уровня 0. Он называется **уровнем физических устройств**. На этом уровне находятся транзисторы, которые являются примитивами для разработчиков компьютеров.

На самом нижнем уровне, **цифровом логическом уровне**, объекты называются вентилями. Хотя вентили состоят из аналоговых компонентов, таких как транзисторы, они могут быть точно смоделированы как цифровые средства. У каждого вентиля есть одно или несколько цифровых входных данных (сигналов, представляющих 0 или 1). Вентиль вычисляет простые функции этих сигналов, такие как И или ИЛИ. Каждый вентиль формируется из нескольких транзисторов. Несколько вентилей формируют 1 бит памяти, который может содержать 0 или 1. Биты памяти, объединенные в группы, например, по 16, 32 или 64, формируют регистры. Каждый регистр может содержать одно двоичное число до определенного предела.

Следующий уровень — **микроархитектурный уровень**. На этом уровне можно видеть совокупности 8 или 32 регистров, которые формируют локальную память и схему, называемую АЛУ (арифметико-логическое устройство). АЛУ выполняет простые арифметические операции. Регистры вместе с АЛУ формируют тракт данных, по которому поступают данные. Основная операция тракта данных состоит в следующем. Выбирается один или два регистра, АЛУ производит над ними какую-либо операцию, например, сложения, а результат помещается в один из этих регистров.

Второй уровень мы будем называть **уровнем архитектуры системы команд**. Каждый производитель публикует руководство для компьютеров, которые он продает, под названием «Руководство по машинному языку» или «Принципы работы компьютера Western Wombat Model 100X» и т. п. Такие руководства содержат информацию именно об этом уровне.

Следующий уровень обычно гибридный. Большинство команд в его языке есть также и на уровне архитектуры системы команд (команды, имеющиеся на одном из уровней, вполне могут находиться на других уровнях). У этого уровня есть некоторые дополнительные особенности: набор новых команд, другая организация памяти, способность выполнять две и более программ одновременно и некоторые другие. При построении третьего уровня возможно больше вариантов, чем при построении первого и второго.

Новые средства, появившиеся на третьем уровне, выполняются интерпретатором, который работает на втором уровне. Этот интерпретатор был когда-то назван операционной системой. Команды третьего уровня, идентичные командам второго уровня, выполняются микропрограммой или аппаратным обеспечением, но не операционной системой. Иными словами, одна часть команд третьего уровня интерпретируется операционной системой, а другая часть — микропрограммой. Вот почему этот уровень считается гибридным. Мы будем называть этот уровень **уровнем операционной системы**.

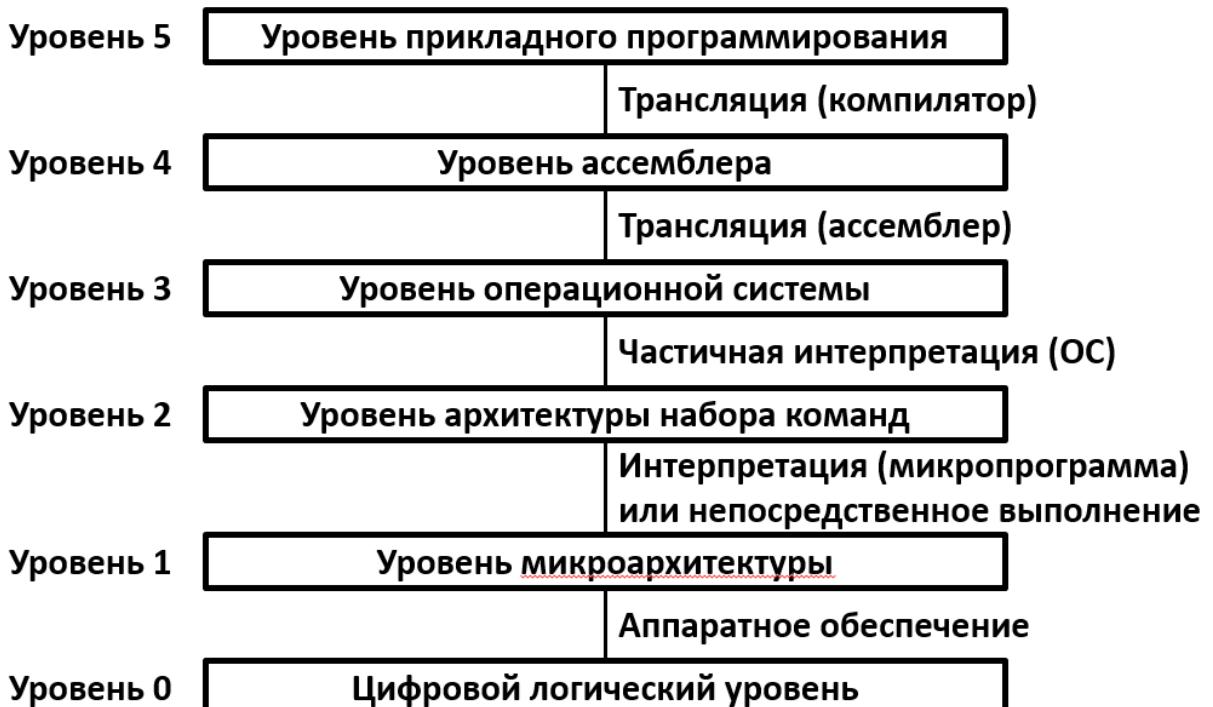
Между третьим и четвертым уровнями есть существенная разница. Нижние три уровня конструируются не для того, чтобы с ними работал обычный программист. Они изначально предназначены для работы интерпретаторов и трансляторов, поддерживающих более высокие уровни. Эти трансляторы и интерпретаторы составляются так называемыми системными программистами, которые специализируются на разработке и построении новых виртуальных машин. Уровни с четвертого и выше предназначены для прикладных программистов, решающих конкретные задачи.

Еще одно изменение, появившееся на уровне 4, — способ, которым поддерживаются более высокие уровни. Уровни 2 и 3 обычно интерпретируются, а уровни 4, 5 и выше обычно, хотя и не всегда, поддерживаются транслятором.

Четвертый уровень представляет собой символическую форму одного из языков более низкого уровня. На этом уровне можно писать программы в приемлемой для человека форме. Эти программы сначала транслируются на язык уровня 1, 2 или 3, а затем интерпретируются соответствующей виртуальной или фактически существующей машиной. Программа, которая выполняет трансляцию, называется **ассемблером**.

Пятый уровень обычно состоит из языков, разработанных для прикладных программистов. Такие языки называются **языками высокого уровня**. Существуют сотни языков высокого уровня. Наиболее известные среди них — BASIC, C, C++, Java, LISP и Prolog. Программы, написанные на этих языках, обычно транслируются на уровень 3 или 4. Трансляторы, которые обрабатывают эти программы, называются **компиляторами**.

Современная многоуровневая модель



6. Функциональная структура микропроцессора. 7. Краткая характеристика основных узлов.

Процессором называется устройство, непосредственно осуществляющее процесс обработки цифровой информации в ВМ и программное управление этим процессом. Процессор занимает центральное место в структуре ВМ. С его помощью осуществляется управление взаимодействием всех устройств, входящих в состав ВМ. Процессор считывает и выполняет команды программы, организует обращение к оперативной памяти, в нужных случаях инициирует работу периферийных устройств, воспринимает и обслуживает запросы прерываний, поступающие из устройств ВМ и извне.

Операционный блок. Предназначен для выполнения некоторого функционально полного набора логических и арифметических операций. Как правило, в его состав входят АЛУ,

буферные регистры операндов, регистр результата (аккумулятор), регистр признаков и блок регистров общего назначения (РОН). Комбинационная схема, являющаяся основой АЛУ, содержит двоичный сумматор и набор логических схем. В АЛУ выполняются несколько простейших арифметических (сложение, вычитание) и поразрядных логических (И, ИЛИ, НЕ и др.) операций. В многоуровневой организации вычислительного процесса указанные операции реализуются на уровне регистровых передач между источниками операндов и приемником результата. Операции по обработке данных, для которых в ОБ отсутствуют аппаратные средства, выполняют программно с помощью процедур.

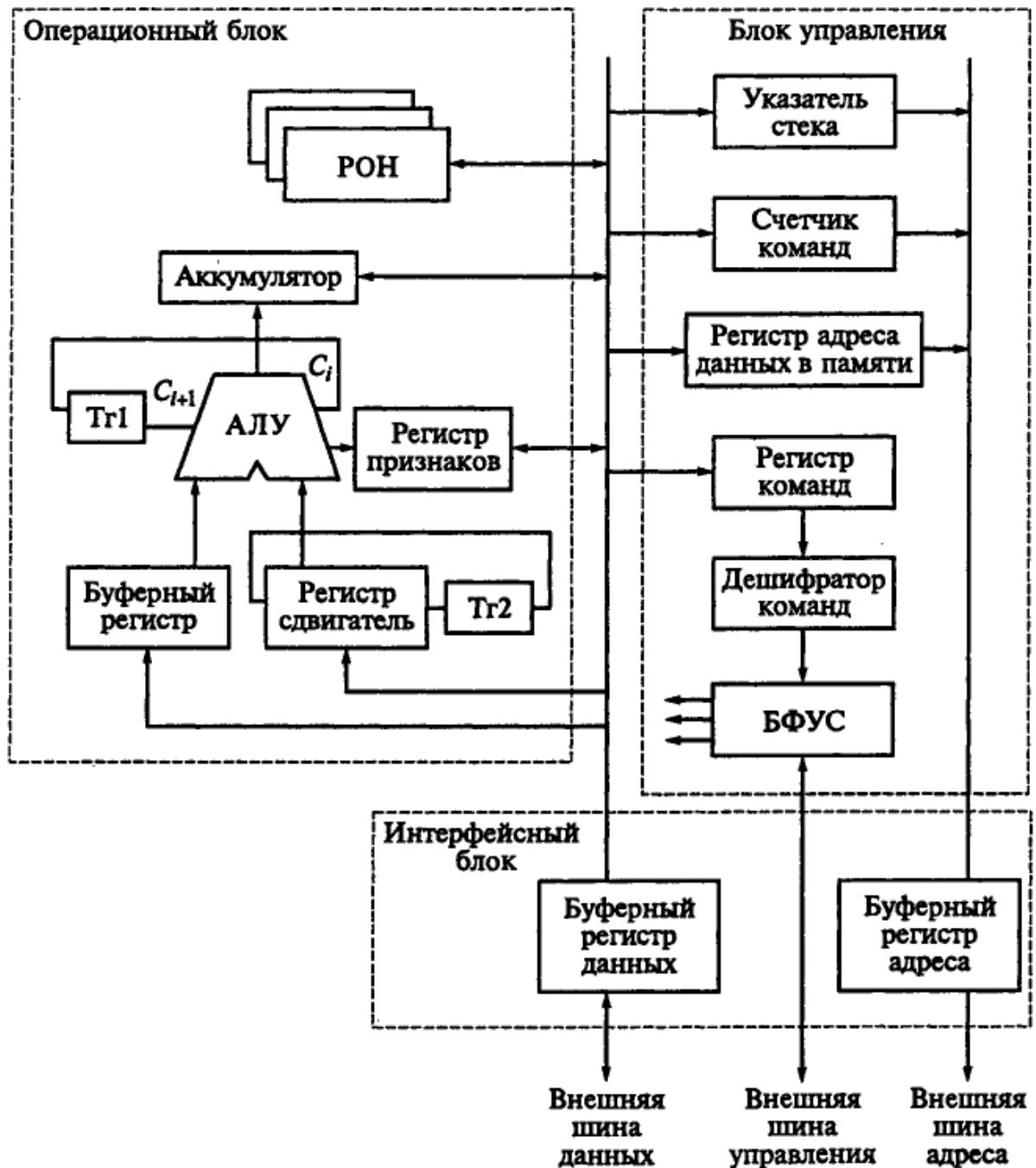


Рис. 4.1. Типовая структура МП:

РОН — регистры общего назначения; Тг1, Тг2 — триггеры переносов соответственно арифметических и сдвиговых операций; АЛУ — арифметико-логическое устройство; C_i , C_{i+1} — соответственно входной и выходной переносы АЛУ; БФУС — блок формирования управляемых сигналов

Важной составляющей ОБ современных МП является блок внутренней памяти, реализованный в виде набора программно доступных регистров, называемых регистрами общего назначения (РОН). Время обращения к РОН меньше, чем к любым другим устройствам Памяти, поэтому память на РОН называется сверхоперативной, а устройство, в виде которого она реализована, сверхоперативным запоминающим устройством (СОЗУ).

В большинстве ранних моделей МП один из общих регистров выделялся в качестве главного регистра. Наделение главного регистра, называемого аккумулятором, или регистром результата, особыми функциями позволяло реализовать ОБ в виде одноадресного устройства. В таком ОБ один из исходных операндов арифметических и логических операций обязательно должен размещаться в аккумуляторе и в него же помещается результат. Другой операнд названных операций может находиться в памяти или РОН. Входные данные поступают в аккумулятор с внутренней шины МП, а аккумулятор, в свою очередь, может посыпать данные на эту шину.

Содержимое любого РОН или ячейки памяти по внутренней шине данных может быть передано через аккумулятор в буферный регистр или напрямую в регистр-сдвигатель. АЛУ обеспечивает выполнение арифметических и логических операций над содержимым регистра-сдвигателя и буферного регистра с записью результата в аккумулятор, а признаков – в регистр признаков.

Операции над словами с повышенной разрядностью реализуются путем программно последовательной обработки отдельных частей многоразрядных слов, для обеспечения возможности обработки данных с разрядностью, превышающей разрядность АЛУ и регистров, в структуре ОБ, предусмотрены два дополнительных триггера: Тг1 и Тг2. С их помощью осуществляется запоминание сигналов арифметического переноса из АЛУ и выходного бита переноса регистра сдвига.

Блок управления. В процессе выполнения программы блок управления координирует работу всех блоков МП и микропроцессорной системы в целом. С помощью блока управления формируются управляющие сигналы, необходимые для организации обмена информацией с внешними устройствами, и обеспечивается выборка команд программы из памяти. В целом блок управления выполняет следующие действия:

- считывает и запоминает текущую команду;
- формирует адрес следующей команды;
- реализует выполнение по тактам алгоритма поступившей команды;
- управляет обменом информацией с внешними устройствами по системнойшине.

Блок управления состоит из регистра команд (РГК), дешифратора команд (ДшК) и блока формирования управляющих сигналов (БФУС). Управляющие сигналы с выходов БФУС поступают на управляющие входы других блоков МП, настраивая их на выполнение определенных микроопераций. В состав блока управления также включают программно доступные счетчик команд PC (Program Counter) и указатель стека SP (Stack Pointer) Большинство элементов этого блока, кроме счетчика PC и указателя стека SP, являются программно недоступными. Счетчик PC (его другое название Instruction Pointer — указатель команд IP) предназначен для адресации команд программы. После выборки из памяти очередной команды в регистре IP формируется адрес следующей по порядку

команды. В командах условных и безусловных переходов, вызова подпрограмм и возврата из подпрограмм в регистр IP непосредственно загружается адрес перехода.

Интерфейсный блок. Предназначен для организации взаимодействия МП с памятью и устройствами ввода-вывода, расположенными на системной шине процессора, а также для обмена данными между ОБ и внутренними устройствами МП. Непосредственное подсоединение устройств ввода-вывода к МП осуществляется с помощью специальных схем сопряжения, которые называются интерфейсом ввода—вывода. В общем случае интерфейсный блок процессора должен выполнять следующие функции:

- формировать выходные сигналы на шинах адреса, данных и управления в режиме вывода;
- формировать выходные сигналы адреса и управления и считывать (воспринимать) сигналы с шины данных в режиме ввода;
- синхронизировать процессы внутри процессора и на системнойшине;
- реализовывать стандартный для системной шины протокол обмена.

Системная шина объединяет сигналы шин данных, адреса и управления. Протокол обмена информацией по СШ определяет последовательности сигналов (временную диаграмму сигналов вшине), обеспечивающих правильную передачу информации между устройствами микропроцессорной системы.

8. Операционные и управляющие автоматы.

В функциональном и структурном отношении операционное устройство, входящее в состав ЭВМ, удобно представить разделенным на две части: операционный и управляющий автоматы.

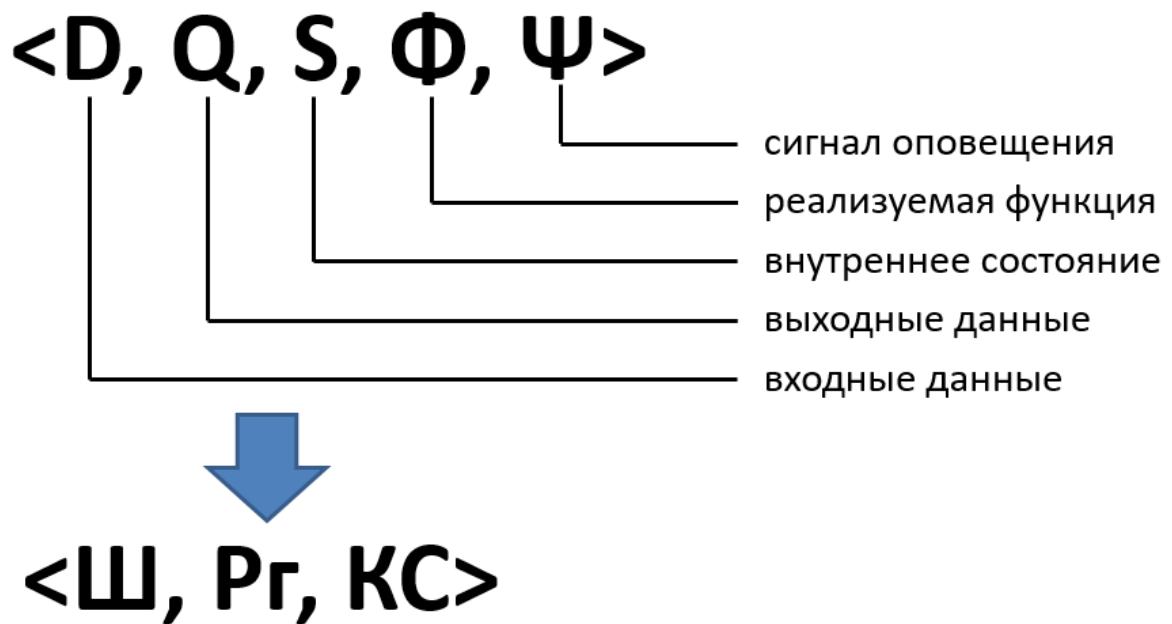
Декомпозиция вычислительных устройств на операционные и управляющие блоки



Операционный автомат (ОА) служит для хранения слов информации, выполнения набора микроопераций и вычисления значений логических условий, т. е. операционный автомат является структурой, организованной для выполнения действий над информацией. На вход ОА подаются входные данные, которые в соответствии с алгоритмом операции преобразуются в выходные данные. Кроме того, ОА вырабатывает множество осведомительных сигналов (логических условий) для управляющего автомата.

Управляющий автомат (УА) генерирует последовательность управляющих сигналов, обеспечивающую выполнение в операционном автомате заданной последовательности элементарных действий, которая реализует алгоритм выполняемой операции. Управляющая последовательность генерируется в соответствии с заданным алгоритмом и с учетом значений логических условий (сигналы оповещения), формируемых ОА.

9. Синтез операционных автоматов канонической структуры.



Ш – множество шин;

Рг – набор регистров;

КС- комбинационные схемы.

$$D = \{d_1, d_2, \dots, d_A\} \Rightarrow Ш_{вх}$$

$$Q = \{q_1, q_2, \dots, q_B\} \Rightarrow Ш_{вых}$$

$$S = \{s_1, s_2, \dots, s_C\} \Rightarrow РГ = \{РГ_1, РГ_2, \dots, РГ_C\}$$

$$\Phi = \{\phi_1, \phi_2, \dots, \phi_k\} \Rightarrow КС: РГ_i = \phi_i(РГ)$$

$$\Psi = \{\psi_1, \psi_2, \dots, \psi_m\} \Rightarrow КС: \psi_i = \Psi_i(РГ)$$

Исходной информацией для синтеза операционного автомата является требуемый для выполнения определенного набора алгоритмов (или одного алгоритма) набор операций. В алгоритме выделяют внутреннее состояние, выходные и внутренние слова, набор микроопераций и логических условий, после чего выполняется синтез.

Этапы синтеза ОА:

- 1) Каждому входному слову ставится в соответствие шина необходимой разрядности, которая соединяется со входом регистра также необходимой разрядности, предназначенного для хранения этого входного слова.
- 2) Каждому выходному слову ставится в соответствие шина, соединенная с выходом регистра.
- 3) Внутреннему состоянию ставится в соответствие регистр необходимой разрядности, либо счетчик для хранения этого состояния.
- 4) Каждой микрооперации ϕ_i , задаваемой в алгоритме оператором присваивания, ставится в соответствие комбинационная схема, либо сдвиговый регистр, либо счетчик, входы которого соединены с выходами регистров operandов с помощью неуправляемых шин, а выход соединен со входом регистра результата с помощью шины, управляемой сигналом (сигнал оповещения).
- 5) Каждому логическому условию ставится в соответствие комбинационная схема для его формирования, входы которой соединены с регистрами operandов, а выходы являются выходами автомата.

10. Представление чисел в ЭВМ.

Знак числа обычно кодируется двоичной цифрой, при этом код 0 означает знак + (плюс), код 1 — знак - (минус).

✓ Прямой код (ПК)

✓ Обратный код (ОК)

$$A_{OK}^- = 2^n - 1 - |A|$$

✓ Дополнительный код (ДК)

$$A_{DK}^- = 2^n - |A|$$

- Операнды представляются в ОК или ДК
- Сложение производится поразрядно с учётом переносов
- Знаковые разряды участвуют в сложении аналогично
- При сложении в ДК перенос из старшего разряда отбрасывается, а в ОК – прибавляется к младшему разряду суммы
- Необходимость отслеживания переполнений – применяют МДК

При этом два последних кода позволяют заменить неудобную для компьютера операцию вычитания на операцию сложения с отрицательным числом. Дополнительный код обеспечивает более быстрое выполнение операций, поэтому в компьютере применяется чаще именно он.

Для того чтобы получить обратный код отрицательного числа, необходимо все цифры этого числа инвертировать, то есть в знаковом разряде поставить 1, во всех значащих разрядах нули заменить единицами, а единицы — нулями.

Для того чтобы получить дополнительный код отрицательного числа, необходимо все его цифры инвертировать (в знаковом разряде поставить единицу, во всех значащих разрядах нули заменить единицами, а единицы — нулями) и затем к младшему разряду прибавить единицу. В случае возникновения переноса из первого после запятой разряда в знаковый разряд, к числу следует прибавить единицу в младший разряд.

При алгебраическом суммировании двух чисел, помещающихся в разрядную сетку, может возникнуть переполнение, т.е. образуется сумма, требующая для своего представления на один двоичный разряд больше, чем разрядная сетка слагаемых. Предполагается, что положительные числа представляются в прямом коде, а отрицательные - в дополнительном.

Признаком переполнения является наличие переноса в знаковый разряд суммы при отсутствии переноса из знакового разряда (положительное переполнение) или наличие переноса из знакового разряда суммы при отсутствии переноса в знаковый разряд (отрицательное переполнение).

3. Алгебраическое суммирование с одним переносом.

(Положительное переполнение).

$$\begin{array}{r} 2+2=4 \\ 2_{(10)} \rightarrow 010_{(2)} \text{ ПК} & 010_{(2)} \\ 2_{(10)} \rightarrow 010_{(2)} \text{ ПК} & +010_{(2)} \\ & \hline 0 \leftarrow 100_{(2)} & \text{ДК} = ?_{(10)} \\ & 1 \uparrow & \\ & \text{перенос} & \end{array}$$

При суммировании есть перенос в знаковый разряд суммы, а перенос из знакового разряда отсутствует, т.е. имеет место положительное переполнение, и результат операции положительный.

4. Алгебраическое суммирование с одним переносом.

(Отрицательное переполнение).

$$\begin{array}{r} -3-2=-5 \\ -3_{(10)} \rightarrow 111_{(2)} \text{ ПК} \rightarrow 101_{(2)} \text{ ДК} & 101_{(2)} \\ -2_{(10)} \rightarrow 010_{(2)} \text{ ПК} \rightarrow 110_{(2)} \text{ ДК} & +110_{(2)} \\ & \hline 1 \leftarrow 011_{(2)} & \text{ДК} = ?_{(10)} \\ & 0 \uparrow & \\ & \text{перенос} & \end{array}$$

Число -5 нельзя представить 3-битовой комбинацией. Формальный результат равен +3.

11. Алгоритмы умножения и деления, структура АЛУ умножения и деления.

Методы умножения

- ✓ начиная с младших разрядов множителя и сдвигом множимого влево
- ✓ начиная с младших разрядов множителя и сдвигом суммы ЧП вправо
- ✓ начиная со старших разрядов множителя и сдвигом множимого вправо
- ✓ начиная со старших разрядов множителя и сдвигом суммы ЧП влево

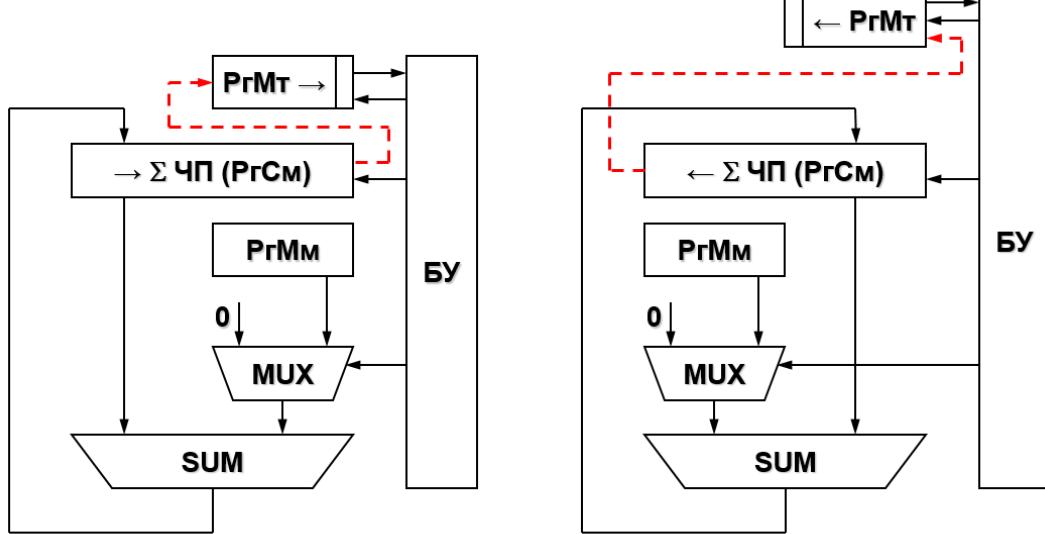
Методы деления

Условие отсутствия переполнения – старшее слово делимого должно быть меньше делителя

- ✓ с восстановлением остатка
- ✓ без восстановления остатка

$$R_i = 2(R_{i-1} + B) - B = 2R_{i-1} + B$$

Структурная схема устройств умножения



а) начиная с младших разрядов множителя и сдвигом суммы ЧП вправо.

б) начиная со старших разрядов множителя и сдвигом суммы ЧП влево

PrMm - множимое

PrMt - множитель

ЧП - частичные произведения

Произведение требует в 2 раза больше разрядов, чем содержат множители

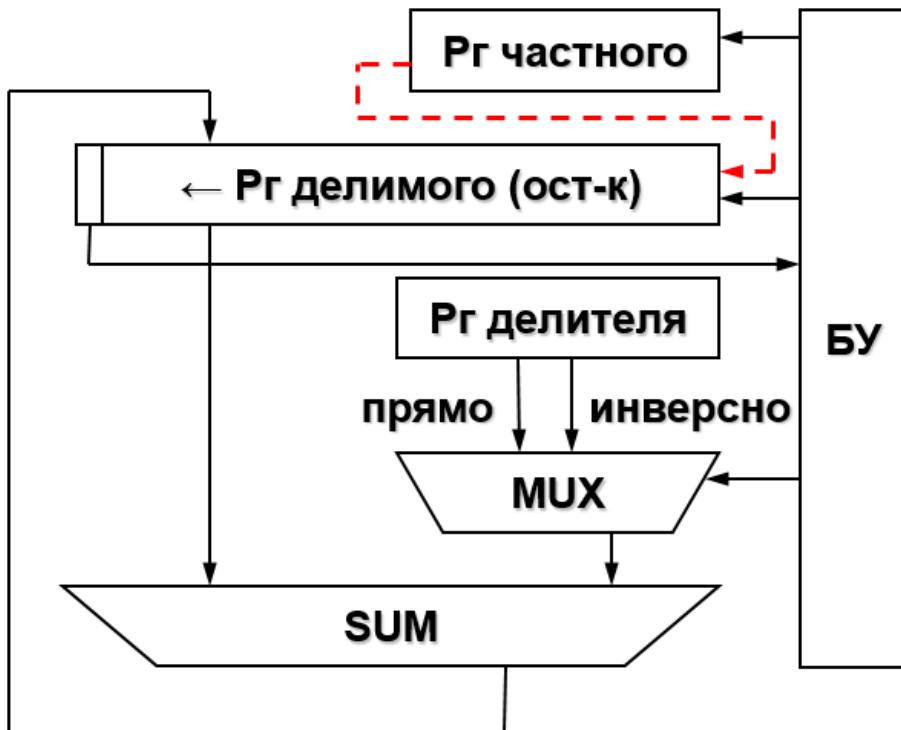
БУ - блок управления

SUM - сумматор

MUX - мультиплексор

Структурная схема устройства деления

Структурная схема деления без восстановления остатка (сам алгоритм ниже: «18»)



12. Управляющие автоматы с жёсткой логикой: классификация, основные особенности.

Название «автомат с жёсткой логикой» происходит из того, что алгоритм функционирования такого автомата жёстко задан его схемой. Для внесения даже незначительных изменений в алгоритм необходимо полностью (или почти полностью) пересинтезировать всю схему автомата.

УА с жёсткой (схемной) логикой (не может быть изменена программным путем)

- на основе автомата Мили
- на основе автомата Мура
- на основе DC кода операции, счётчика (или DC) тактов
- на основе ПЛМ

В основе построения УА с жесткой логикой (ЖЛ) управления лежит теория конечных (цифровых) автоматов. Наибольшую известность имеют конечные автоматы Мили и Мура.

Сначала необходимо перейти от *граф-схемы алгоритма* (ГСА) микропрограммы к графу автомата, для чего следует:

1. Разметить исходную микропрограмму.
2. Построить по размеченной микропрограмме граф автомата.

Далее реализуются стандартные процедуры синтеза структурного автомата, заданного графом:

- кодирование алфавита входных и выходных символов автомата двоичными кодами;
- кодирование внутренних состояний автомата;
- выбор элемента памяти (типа триггера);
- построение автоматной таблицы переходов;
- синтез комбинационной схемы

Процедура *разметки микропрограммы* ставит в соответствие символам состояний автомата (a_1, a_2, \dots, a_m) некоторые объекты микропрограммы. Способы разметки микропрограмм различаются для автоматов различных типов.

Для автомата Мура разметка выполняется по следующим правилам:

- символом a_1 отмечается начальная и конечная вершина ГСА;
- различные операторные вершины отмечаются разными символами состояний;
- все операторные вершины должны быть отмечены.

Для автомата Мили разметка выполняется по следующим правилам:

- символом a_1 отмечается вход вершины, следующей за начальной, а также вход конечной вершины;
- входы всех вершин, следующих за операторными, должны быть отмечены символами состояний;
- если вход вершины отмечается, то лишь одним символом;
- входы различных вершин, за исключением конечной, отмечаются различными символами.

Главным недостатком такой схемы является то, что

- 1) на выполнение различных команд отводится одинаковое количество тактов
- 2) логику работы такого управляющего автомата, можно изменить только путем перестройки схем, отсюда следует что система команд микропроцессором с аппаратным управлением – фиксировано.

Основным достоинством аппаратного управления является высокое быстродействие. Такие автоматы чаще всего используются в микропроцессорах типа RISC.

С точки зрения затрат оборудования УА с ПЛ (программируемая логика) экономнее, чем УА с ЖЛ (жесткая логика). Отсюда область их применения: если целью проектирования является высокое быстродействие, то следует использовать УА с ЖЛ, если критерий проектирования – минимальные затраты оборудования, то – УА с ПЛ.

13. Управляющие автоматы, построенные по схеме Мура, схеме Мили.

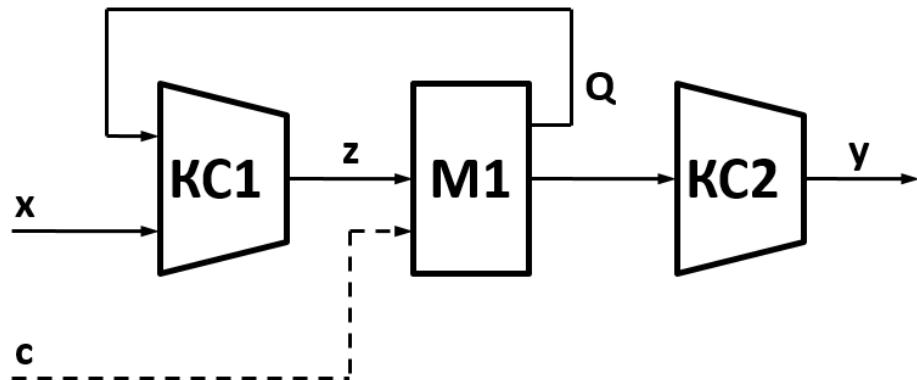
Управляющий автомат Мура

Его выходные сигналы зависят только от состояния триггеров. Поэтому его КС фактически распадается на 2 независимые КС

$$Q(t+1) = A[Q(t), x(t)]$$

$$y(t) = B[Q(t)]$$

$$z=C[Q(t), x(t)]$$



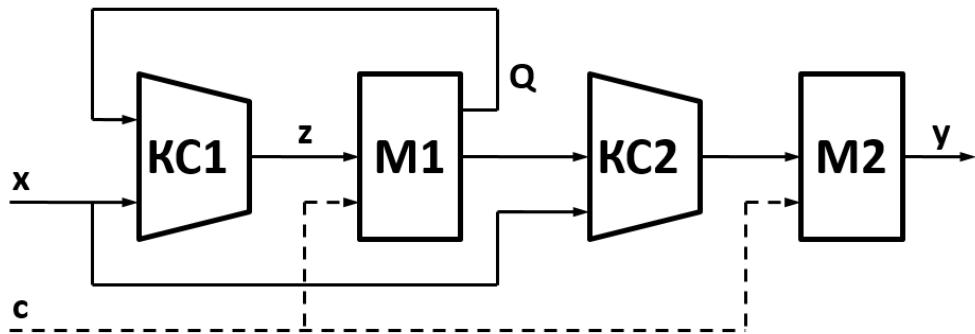
Управляющие автоматы Мили

Иногда говорят, что этот автомат генерирует (в смысле изменяет) выходные сигналы при переходах из одного состояния в другое. Здесь подчёркивается тот факт, что Y непосредственно зависит от X.

$$Q(t+1) = A[Q(t), x(t)]$$

$$y(t+1) = B[Q(t), x(t)]$$

$$z=C[Q(t), x(t)]$$



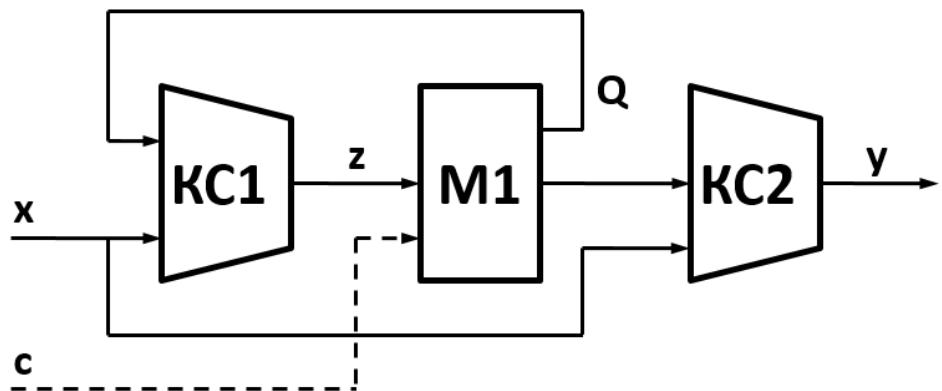
M – memory

M1 хранит код состояния Q; записываем выходы KC1(определяется текущим состоянием и входными сигналами).

KC2 определяется кодом текущего состояния и комбинацией входных сигналов.

M2 – буферизирующий элемент.

Автомат Мили с асинхронными выходами



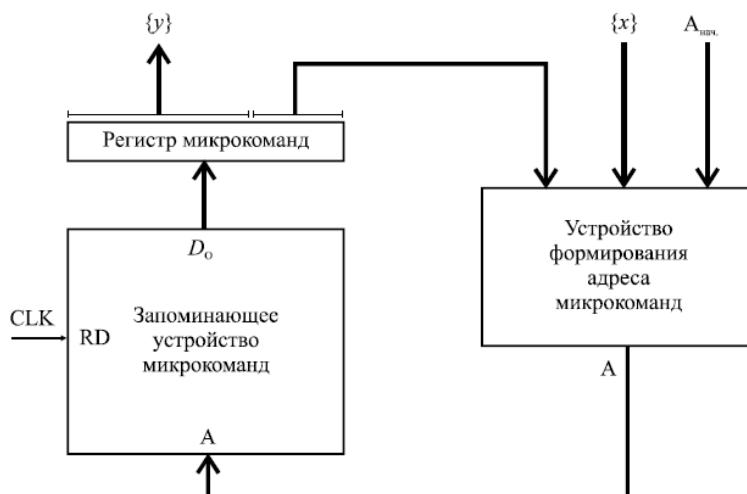
ДОПОЛНИТЬ(всем)

14. Управляющие автоматы с хранимой в памяти логикой: принципы построения.

Заметим, что функция любого управляющего автомата — генерирование последовательности управляющих слов (микрокоманд), определенной реализуемым алгоритмом с учетом значений осведомительных сигналов.

Если заранее разместить в запоминающем устройстве все необходимые для реализации заданного алгоритма (группы алгоритмов) микрокоманды, а потом выбирать их из памяти в порядке, предусмотренном алгоритмом (с учетом значения осведомительных сигналов), то получим управляющий автомат, структура которого слабо зависит от реализуемых алгоритмов, а поведение в основном определяется содержимым запоминающего устройства.

При изменениях реализуемого алгоритма в достаточно широких пределах структура такого автомата не меняется, достаточно лишь изменить содержимое ячеек запоминающего устройства. Управляющий микропрограммный автомат, построенный по таким принципам, называется управляющим автоматом с программируемой логикой.



Автомат включает в себя запоминающее устройство микрокоманд (обычно реализуемое как ПЗУ), регистр микрокоманд и устройство формирования адреса микрокоманды. В каждом такте дискретного времени из памяти микрокоманд считывается одна

микрокоманда, которая помещается в регистр микрокоманд. Микрокоманда содержит два поля (две группы полей), одно из которых определяет набор микроопераций, которые в данном такте поступают в операционный автомат, а другое содержит информацию для определения адреса следующей микрокоманды.

При проектировании управляющего автомата с программируемой логикой (УАПЛ) необходимо выбрать формат (форматы) микрокоманд (микрокоманды), способы кодирования микроопераций и адресации микрокоманд.

15. Микропрограммирование, кодирование микрокоманд.

Микрокоманда – комбинация нескольких МОП, подаваемых одновременно.

Последовательность микрокоманд – микропрограмма.

Все множество микроопераций разбиваются на подмножества, и они кодируются специальным образом (это сделано для оптимизации памяти микрокоманд).

Различают три способа кодирования поля микроопераций:

- горизонтальный (позиционный);
- вертикальный (двоичная свертка);
- смешанный (вертикально-горизонтальный и горизонтально-вертикальный).

Ранее мы говорили, что управляющий автомат проектируется для выдачи в заданной последовательности наборов микроопераций из некоторого наперед определенного множества микроопераций $Y = \{y_1, y_2, \dots, y_n\}$.

При *горизонтальном способе* кодирования каждой микрооперации $y_i \in \{y_1, \dots, y_n\}$ ставится в соответствие разряд поля микроопераций микрокомандного слова. В этом случае количество разрядов поля микроопераций N равно числу n различных микроопераций, вырабатываемых УА.

Достоинствами горизонтального способа кодирования являются:

- возможность формирования произвольных микрокоманд из заданного набора микроопераций;
- простота реализации схем формирования микроопераций, фактически — их отсутствие, т. к. выход каждого разряда поля микроопераций регистра микрокоманд является выходной линией УА — соответствующей микрооперацией.

Недостаток — чаще всего неэффективно используется память микрокоманд.

При *вертикальном способе* кодирования в поле микроопераций помещается номер выполняемой микрооперации. При этом количество разрядов N , которое следует предусмотреть в поле микроопераций, определяется выражением: $N = k \geq \log_2 n$.

Достоинство способа в экономном использовании памяти микрокоманд.

Недостаток — в невозможности реализовать в микрокоманде более одной микрооперации.

Если реализуемые алгоритмы и структура ОА таковы, что в каждом такте дискретного времени выполняется не более одной микрооперации, то *вертикальный способ кодирования* — оптимальное решение.

Рассмотрим *смешанный способ кодирования*, идея которого состоит в следующем. Если во всех микропрограммах, реализуемых УА, нет микрокоманды с большим, чем s , числом микроопераций, то в поле микроопераций можно предусмотреть s подполей разрядностью k , в каждом из которых помещать номер нужной микрооперации. Такой способ позволяет в любой микрокоманде реализовать произвольную s -ку микроопераций, т. е. сохранить гибкость горизонтального кодирования, при возможном значительном сокращении разрядности поля микроопераций: $N = s \cdot k = s \log_2 n$. Так, для приведенного выше примера ($n = 80$, $s = 6$) определим $k = 7 \geq \log_2 80$, $N = 7 \cdot 6 = 42$ (тоже, конечно, немало), что позволит почти вдвое сократить разрядность поля микроопераций по сравнению с горизонтальным способом кодирования. Эффективность применения смешанного кодирования существенно зависит от значения s , которое может лежать в диапазоне $1 \leq s \leq n$. При $s = 1$ имеем случай вертикального кодирования, при $s = n$ — горизонтального.

Известны несколько вариантов такого кодирования, среди которых наиболее распространены *горизонтально-вертикальное* (в подмножества объединяются МОП, не использующиеся в одном такте) и *вертикально-горизонтальное* (в подмножества объединяются МОП, встречающиеся в одном такте) кодирование. В обоих вариантах множество всех m возможных сигналов управления разбивается на k подмножеств. При разбиении на подмножества каждая микрооперация может присутствовать лишь в одном из подмножеств.

	mOp0	mOp1	mOp2	mOp3	mOp4	mOp5
Mcmd1	1		1			
Mcmd2				1		
Mcmd3					1	
Mcmd41		1				1

вертикально-горизонтальное (всего 4 микрокоманды—значит ключ двухразрядный)

00	0	2	
01	1	5	
	3	-	
10			
11	4	-	

горизонтально-вертикальное (максимум 2 микрооперации—значит ключ одноразрядный):

0	0	2	3
1	1	5	4

16. Машинная арифметика.

В лекциях не было. Про мантиссу и порядок или че? Мне кажется, здесь, все-таки, нужно написать про представление чисел в компьютере. Тип сколько разрядов у каждого типа и т.п.

Прямой код

В современных ЭВМ используются, в основном, два способа представления двоичных чисел — с фиксированной и с плавающей запятой, причем в формате с фиксированной запятой (Φ_3) используется как *беззнаковое* представление чисел ("целое без знака"), так и представление чисел со знаком. В последнем случае знак также кодируется двоичной цифрой — обычно плюсу соответствует 0, а минусу — 1. Под код знака обычно отводится старший разряд a_0 двоичного вектора $a_0a_1a_2\dots a_n$, называемый *знакомым*.

Запятая может быть фиксирована после любого разряда двоичного числа, однако чаще всего используются два формата Φ_3 : *целые числа*, когда запятая фиксируется после младшего разряда a_n , а диапазон представления лежит в пределах $A \leq 2^n - 1$, (3.12) и *дробные числа* — запятая фиксирована после a_0 , а диапазон $A \leq 1 - 2^{-n}$. (3.13)

Далее, если не сделано специальных оговорок, будем рассматривать дробные двоичные числа со знаком, запятая которых фиксирована после знакового разряда $a_0 : a_0, a_1a_2a_3\dots a_n$. (3.14)

Очевидно, если двоичное число $A = 0.a_1a_2a_3\dots a_n > 0$, то оно будет представлено в форме (3.14) как $0.a_1a_2a_3\dots a_n$, а если $A = 0.a_1a_2a_3\dots a_n < 0$, то как $1.a_1a_2a_3\dots a_n$. Приведенное кодирование дробных двоичных чисел со знаком принято называть *прямым кодом числа* (обозначается как $[A]_d$).

Алгебраическое сложение/вычитание в прямом коде

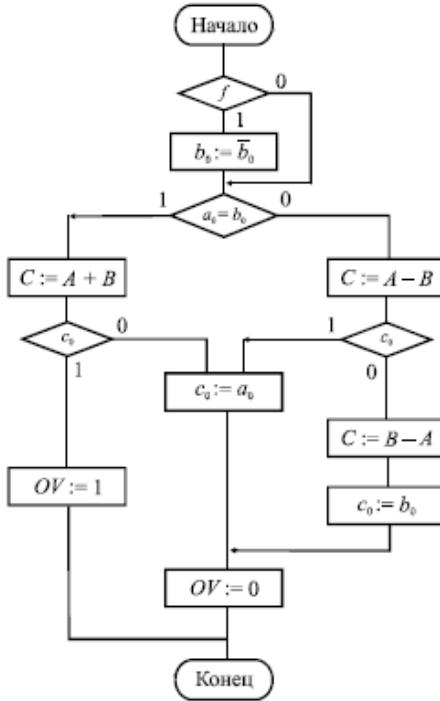
Сформулируем правила выполнения операций сложения и вычитания чисел со знаками (такие операции принято называть *алгебраическими*). Во-первых, алгебраическое вычитание всегда можно свести к алгебраическому сложению, изменив знак второго операнда. Далее следует сравнить знаки слагаемых. При одинаковых знаках складывают модули слагаемых и результату присваивают знак любого слагаемого (они одинаковые). Если знаки слагаемых разные, то из большего модуля слагаемого вычитают меньший модуль и присваивают результату знак слагаемого, имеющего больший модуль.

Введем обозначения:

$$\begin{aligned} A &= a_0a_1a_2a_3\dots a_n, \\ B &= b_0b_1b_2b_3\dots b_n, \\ C &= A + B = c_0c_1c_2\dots c_n, \end{aligned}$$

где:

- a_0, b_0 — знаковые разряды слагаемых;
- c_0 — код знака результата;
- $a_i, b_i, c_i, i \in \{0, 1, 2, \dots, n\}$ — двоичные переменные;
- f — тип выполняемой операции: $f = 0$ — сложение, $f = 1$ — вычитание;
- OV — признак переполнения.



Отдельно следует рассмотреть проблему обнаружения факта переполнения разрядной сетки данных с фиксированной запятой. Это может произойти, если $C = A + B \geq 1$.

Очевидно, при сложении чисел с разными знаками переполнение невозможно. Если знаки слагаемых одинаковы, признаком переполнения может служить перенос, возникающий при сложении старших разрядов модулей $a_1 + b_1$. При отсутствии этого переноса сложение двух любых *одинаковых* знаковых разрядов даст в результате $c_0 = 0$, а при появлении переноса из первого разряда $c_0 = 1$. Таким образом, после сложения чисел с одинаковыми знаками значение знакового разряда суммы можно рассматривать как признак переполнения OV .

Характерно, что полученное в знаковом разряде c_0 значение не является знаком результата (алгебраической суммы). Истинное значение знака образуется не в процессе арифметической операции над знаковыми разрядами, а формируется искусственно.

Обратный код и выполнение алгебраического сложения в нем

При выполнении алгебраического сложения в прямом коде приходится, во-первых, не только складывать, но и вычитать двоичные коды; во-вторых, код знака результата формируется искусственно, т. е. знаковые разряды обрабатываются по правилам, отличным от правил обработки разрядов числа. Для устранения отмеченных недостатков в ЭВМ широко используются специальные представления двоичных чисел — так называемые *обратный* и *дополнительный коды*.

Представление *обратного кода* определяется следующим соотношением:

$$[A]_i = \begin{cases} A, & \text{если } A \geq 0; \\ 2 + A - 2^n, & \text{если } A \leq 0. \end{cases}$$

Из него следует, что обратный код положительного числа равен самому числу! Для получения обратного кода отрицательного числа достаточно присвоить знаковому разряду значение 1 и проинвертировать все остальные разряды числа: $[-0, a_1 a_2 \dots a_n] i = 1, \overline{a_1 a_2 \dots a_n}$.

Действительно, из (3.17) следует, что при $A = -0, a_1 a_2 a_3 \dots a_n$ обратный код числа $[A]_i = 2 + A - 2^n$. Откуда $[A]_i - A = 2 - 2^n$.

Для перехода из обратного кода в прямой осуществляется следующее преобразование: $[1, \overline{a_1 a_2 \dots a_n}] i = 1, \overline{\overline{a_1 a_2 \dots a_n}}$

Алгебраическое сложение в обратном коде

Для выполнения алгебраического сложения двоичных чисел, представленных в обратном коде, достаточно, не анализируя соотношение знаков и модулей, произвести сложение чисел, включая знаковые разряды, по правилам двоичной арифметики, причем возникающий в знаковом разряде перенос должен быть добавлен к младшему разряду результата, осуществляя тем самым коррекцию предварительной суммы. Полученный код является алгебраической суммой слагаемых, представленной в обратном коде.

Итак, использование обратного кода в операциях алгебраического сложения/вычитания позволяет:

- использовать только действие арифметического сложения двоичных кодов;
- получать истинное значение знака результата, выполняя над знаковыми разрядами операндов те же действия, что и над разрядами чисел;
- обнаруживать переполнение разрядной сетки.

Еще одним достоинством применения обратного кода можно считать простоту взаимного преобразования прямого и обратного кода.

Однако использование обратного кода имеет один существенный недостаток — коррекция предварительной суммы требует добавления единицы к ее младшему разряду и может вызвать (в некоторых случаях) распространение переноса по всему числу, что, в свою очередь, приводит к увеличению вдвое времени суммирования. Для преодоления этого недостатка можно использовать вместо обратного *дополнительный код*.

Дополнительный код и арифметические операции в нем

Связь между числом и его изображением в дополнительном коде определяется соотношениями. Таким образом, и дополнительный код положительного числа равен самому числу (как обратный и прямой). Дополнительный код отрицательного числа *дополняет* исходное число до основания системы счисления. Дополнительный код отрицательного числа образуется в соответствии со следующим выражением:

Для преобразования отрицательного двоичного числа в дополнительный код следует преобразовать его сначала в обратный код (установив знаковый разряд в 1 и проинвертировав все остальные разряды числа) и добавить единицу к младшему разряду обратного кода.

Другой способ перевода прямого кода отрицательного двоичного числа в дополнительный (приводящий, разумеется, к такому же результату) определяется следующим правилом: оставить без изменения все младшие нули и одну младшую единицу, остальные разряды (кроме знакового!) проинвертировать.

Применение дополнительного кода, по сравнению с обратным, имеет одно существенное преимущество — коррекция результата сводится просто к отбрасыванию переноса из знакового разряда и *не требует дополнительных затрат времени*. К недостаткам применения дополнительного кода можно отнести, во-первых, более сложную процедуру взаимного преобразования ПК↔ДК, требующую дополнительных затрат времени, и, во-вторых, проблемы с обнаружением переполнения. Для того чтобы минимизировать влияние первого недостатка, данные в памяти часто хранят в дополнительном коде. В этом случае преобразования ПК↔ДК выполняются относительно редко — только при вводе и выводе.

С целью более удобного обнаружения переполнения в обратном и дополнительном кодах можно применить так называемые "модифицированные" их представления:

$$[A]_i^m = \begin{cases} A, & \text{если } A \geq 0; \\ 4 + A - 2^{-n}, & \text{если } A \leq 0. \end{cases}$$

$$[A]_e^m = \begin{cases} A, & \text{если } A \geq 0; \\ 4 + A, & \text{если } A < 0. \end{cases}$$

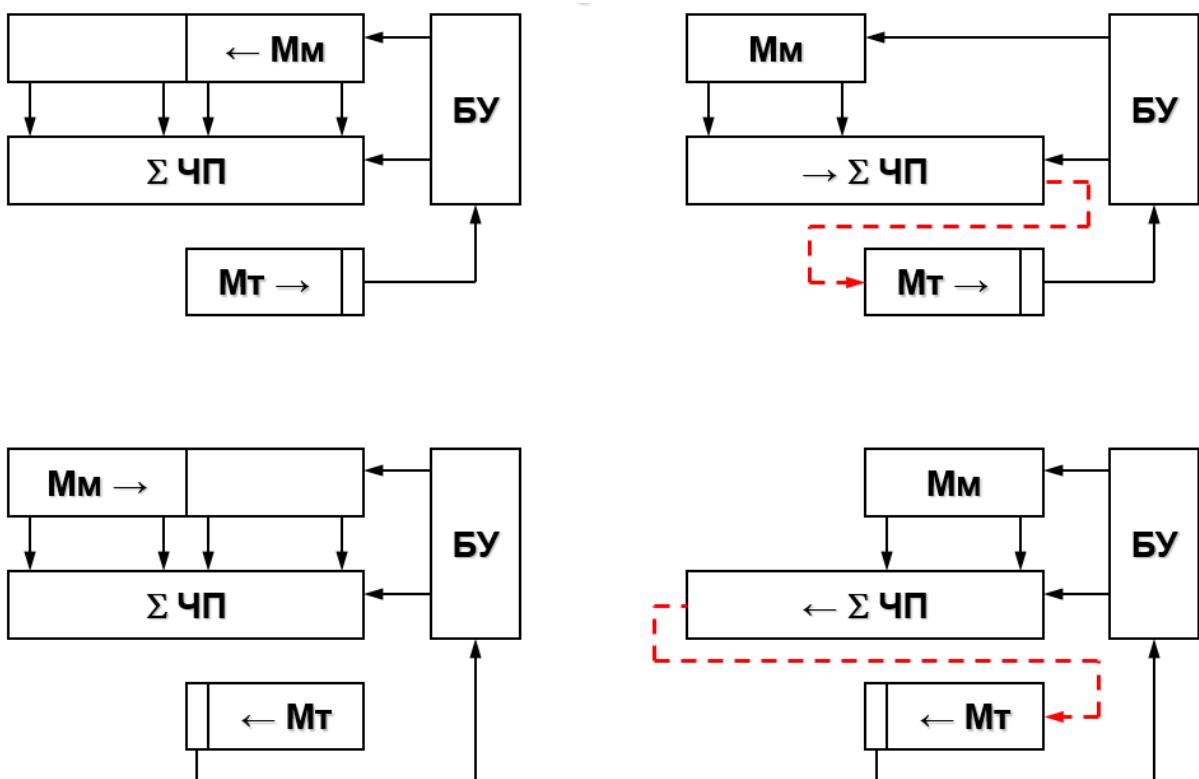
Нетрудно показать, что модифицированные коды отличаются от соответствующих обычных наличием дополнительного знакового разряда: "плюс" кодируется 00, а "минус"

— 11. Эта своеобразная избыточность, сохраняя все качества обычных обратных и дополнительных кодов, позволяет фиксировать факт переполнения по *неравнозначности знаковых разрядов результата*. Заметим, что использование модифицированного дополнительного кода не решает проблемы обнаружения переполнения в случаях $A < 0$, $B < 0$, $|A + B| = 1$.

17. Выбор оптимального алгоритма умножения и структуры устройства умножения.

Методы умножения

- ✓ начиная с младших разрядов множителя и сдвигом множимого влево
- ✓ начиная с младших разрядов множителя и сдвигом суммы ЧП вправо
- ✓ начиная со старших разрядов множителя и сдвигом множимого вправо
- ✓ начиная со старших разрядов множителя и сдвигом суммы ЧП влево



Для реализации любого метода схема включает в себя регистр множимого M_m , регистр множителя M_t , сумматор с регистром сумматора для накопления суммы ЧП и блок управления BU . При разрядности сомножителей, равной n , разрядность регистра множимого и сумматора зависит от метода умножения.

На каждом шаге умножения в BU проводится анализ одной цифры множителя. В зависимости от ее значения BU формирует сигналы для выдачи множимого в сумматор, если цифра множителя равна «1», а также сигналы сдвига на регистр множителя и множимого (или суммы ЧП).

Реализация методов 2 и 4 требует меньших аппаратурных затрат, так как при этом используется регистр множимого меньшей разрядности. Кроме того, при сдвиге

множимого один из входов сумматора должен иметь удвоенную разрядность, что также увеличивает затраты на реализацию методов 1 и 3.

Анализ схем реализации методов 2 и 4 показывает, что затраты можно уменьшить, если использовать освобождающиеся разряды регистра множителя для записи уже сформированных разрядов произведения так, как это показано на рис. 4.8, б, г пунктирными линиями. При умножении используются методы со сдвигом ЧП.

18.Подходы к ускорению умножения.

✓ логические

✓ алгоритм Бута

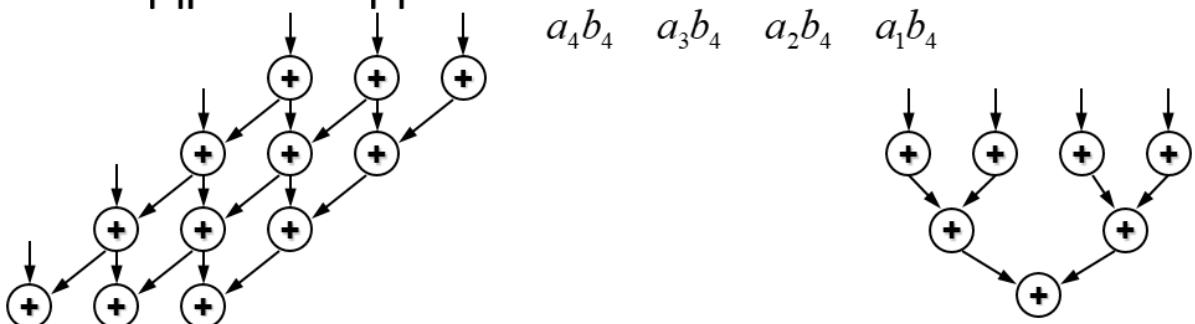
✓ аппаратные

✓ матричные

✓ древовидные

$$2^m + 2^{m-1} + \dots + 2^{k+1} + 2^k = \\ = 2^{m+1} - 2^k$$

$$\begin{array}{cccc} a_4 b_1 & a_3 b_1 & a_2 b_1 & a_1 b_1 \\ a_4 b_2 & a_3 b_2 & a_2 b_2 & a_1 b_2 \\ a_4 b_3 & a_3 b_3 & a_2 b_3 & a_1 b_3 \\ a_4 b_4 & a_3 b_4 & a_2 b_4 & a_1 b_4 \end{array}$$



Умножение относится к длинным операциям, время выполнения которых значительно превышает время сложения. Для повышения быстродействия часто используют логические и аппаратные методы ускорения умножения. Логические методы основаны на одновременном анализе нескольких цифр множителя и сокращении за счет этого числа микроопераций. Реализация логических методов приводит к усложнению БУ АЛУ. Аппаратные методы обеспечивают одновременное выполнение нескольких микроопераций за счет дополнительной аппаратуры. Часто используют комбинации логических и аппаратных методов.

Алгоритм Бута минимизирует кол-во единиц, чтобы было больше переносов (очень дешевая операция) без сложения.

В основе алгоритма Бута лежит следующее соотношение, характерное для последовательности двоичных цифр:

$$2_m + 2_{m-1} + \dots + 2_k = 2_{m+1} - 2_k,$$

где m и k — номера крайних разрядов в группе из последовательности единиц. Например, $011110 = 2^5 - 2^1$. Это означает, что при наличии в множителе групп из нескольких единиц (комбинаций вида 011110), последовательное добавление к СЧП множимого с нарастающим весом (от 2_k до 2_m) можно заменить вычитанием из СЧП множимого с весом

2_k и прибавлением к СЧП множимого с весом 2_{m+1} . Алгоритм предполагает три операции: сдвиг, сложение и вычитание.

Например,

$$M \times "0\ 0\ 1\ 1\ 1\ 1\ 1\ 0" = M \times (2^5 + 2^4 + 2^3 + 2^2 + 2^1) = M \times 62$$

где M — множимое. Количество операций может быть уменьшено вдвое, если представить произведение следующим образом :

$$M \times "0\ 1\ 0\ 0\ 0\ 0\ 1\ 0" = M \times (2^6 - 2^1) = M \times 62.$$

Аппаратно: увеличиваем кол-во элементов и запускаем параллельно.

В матричных умножителях суммирование осуществляется матрицей сумматоров, состоящей из последовательных линеек (строк) одноразрядных сумматоров с сохранением переноса (ССП). По мере движения данных вниз по массиву сумматоров каждая строка ССП добавляет к СЧП очередное частичное произведение. Поскольку промежуточные СЧП представлены в избыточной форме с сохранением переноса, во всех схемах, вплоть до последней строки, где формируется окончательный результат, распространения переноса не происходит. Это означает, что задержка в умножителях зависит только от «глубины» массива (числа строк сумматоров) и не зависит от разрядности операндов, если только в последней строке матрицы, где формируется окончательная СЧП, не используется схема с последовательным переносом.

$$\begin{array}{l}
 A = a_4a_3a_2a_1; \quad A \quad \times = C = \times \quad \begin{array}{r}
 a_4b_1 \ a_3b_1 \ a_2b_1 \ a_1b_1 \\
 a_4b_2 \ a_3b_2 \ a_2b_2 \ a_1b_2 \\
 a_4b_3 \ a_3b_3 \ a_2b_3 \ a_1b_3 \\
 a_4b_4 \ a_3b_4 \ a_2b_4 \ a_1b_4
 \end{array} \\
 B = b_4b_3b_2b_1. \quad B \quad \hline
 \end{array}$$

$A \times B = C$

Матричный умножитель

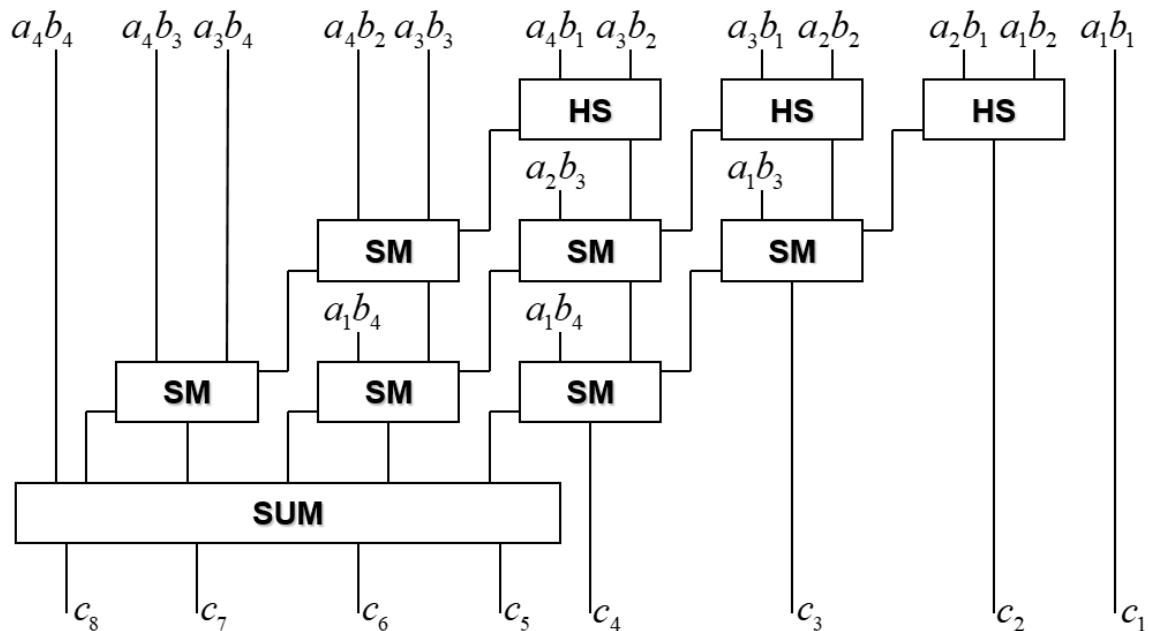


рис. 4.10. Формирование разрядов ЧП вида $a_i b_j$ выполняется логическими элементами И. В первом ярусе умножителя используются полусумматоры (HS), в следующих ярусах — одноразрядные сумматоры (SM). Окончательное суммирование выполняется при помощи четырехразрядного сумматора, в котором возможна любая организация переносов. Числа со знаком при умножении могут представляться в ДК.

Время умножения в матричных умножителях пропорционально разрядности чисел. При большой разрядности чисел для уменьшения времени умножения применяются *древовидные умножители*. Повышение их быстродействия достигается за счет изменения организации суммирования ЧП по сравнению с матричными умножителями (рис. 4.11).

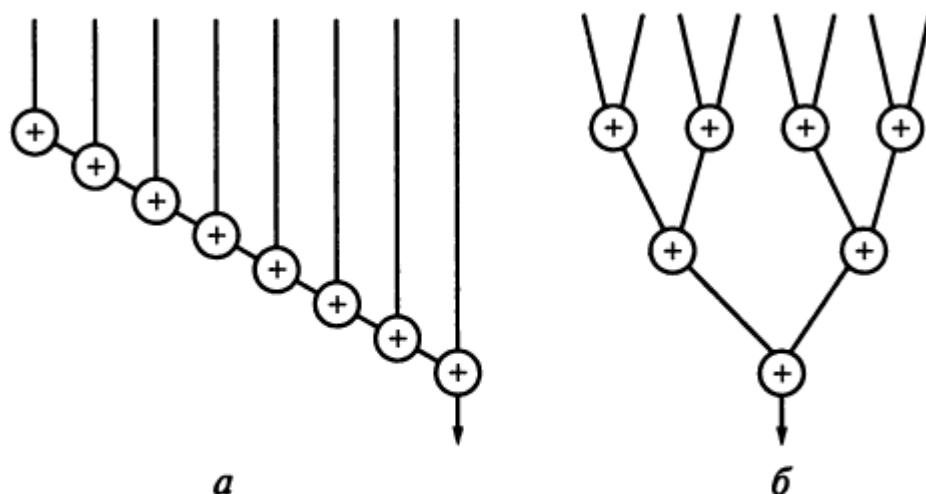


Рис. 4.11. Суммирование в умножителях:
а — с матричной структурой; б — с древовидной

Сократить задержку, свойственную матричным умножителям, удается в схемах с древовидной структурой. В матричных умножителях каждое очередное частичное произведение прибавляется к СЧП с помощью отдельной строки сумматоров, и для перемножения n -разрядных чисел требуется n таких строк. В древовидных умножителях процесс получения СЧП также реализуется за счет строк сумматоров, но организованных по схеме дерева, благодаря чему количество строк сумматоров сокращается до $\log_2 n$. Так как в умножителях обоих типов каждая строка вносит задержку, свойственную одному полному сумматору, количество строк во многом определяет общее быстродействие умножителя.

19. Алгоритмы с восстановлением и без восстановления остатка

Общая последовательность деления с восстановлением остатка содержит следующие микрооперации:

1. Делитель размещается в старших разрядах двойного слова

- Проверяется условие возможности деления (отсутствие переполнения). Для этого из делимого вычитается делитель и анализируется знак остатка.
- Если остаток положительный, то деление невозможно, формируется признак переполнения и процесс заканчивается. Если остаток меньше нуля, то деление продолжается. При этом остаток восстанавливается путем прибавления делителя.
- Остаток сдвигается влево на один разряд.
- Из сдвинутого остатка вычитается делитель и анализируется знак остатка.

Делимое $A_{10} = +145$	00 1001 0001	1101	Делитель $B_{10} = -13$
Вычитание делителя	+ 11 0011	1011	Частное $(A/B)_{10} = -11$
Остаток < 0	= 11 1100 0001	↑↑↑↑	Деление корректно
Восстановление остатка	+ 00 1101	 	
Восстановленный остаток	= 00 1001 0001	 	
Сдвиг остатка влево	01 0010 0010	 	
Вычитание делителя	+ 11 0011	 	
Остаток > 0	= 00 0101 001	--↑ 	
Сдвиг остатка влево	00 1010 01	 	
Вычитание делителя	+ 11 0011	 	
Остаток < 0	= 11 1101 01	---↑ 	
Восстановление остатка	+ 00 1101	· 	
Восстановленный остаток	= 00 1010 01	 	
Сдвиг остатка влево	01 0100 1	 	
Вычитание делителя	+ 11 0011	 	
Остаток > 0	= 00 0111 1	-----↑ 	
Сдвиг остатка влево	00 1111	 	
Вычитание делителя	+ 11 0011	 	
Остаток > 0	= 00 0010	-----↑	Остаток $R_{10} = +2$

- Если остаток положительный, то очередная цифра частного равна единице. Если остаток меньше нуля, то очередная цифра множителя равна нулю. При этом остаток восстанавливается путем прибавления делителя.

7. Проверяется условие окончания деления. Если получены все цифры частного, то деление заканчивается, иначе выполняется переход к п. 4. Последний остаток восстанавливается, если он меньше нуля. Для этого к нему прибавляется делитель.

При вычитании делителя используется дополнительный или модифицированный дополнительный код.

Если при делении с восстановлением остатка получен отрицательный остаток, то после восстановления остатка, сдвига восстановленного остатка и последующего вычитания делителя на шаге i будет получен следующий результат:

$$R_i = 2(R_{i-1} + B) - B = 2R_{i-1} + B,$$

где R_i — остаток на шаге i ; R_{i-1} — остаток на шаге $i-1$; B — делитель.

Множитель 2 возникает при сдвиге данных влево на один разряд.

Результат $2R_{i-1} + B$ может быть получен более простым путем, что и используется при делении без восстановления остатка. Такой вид деления отличается тем, что при получении отрицательного остатка он сдвигается влево и к нему прибавляется делитель для определения следующего остатка.

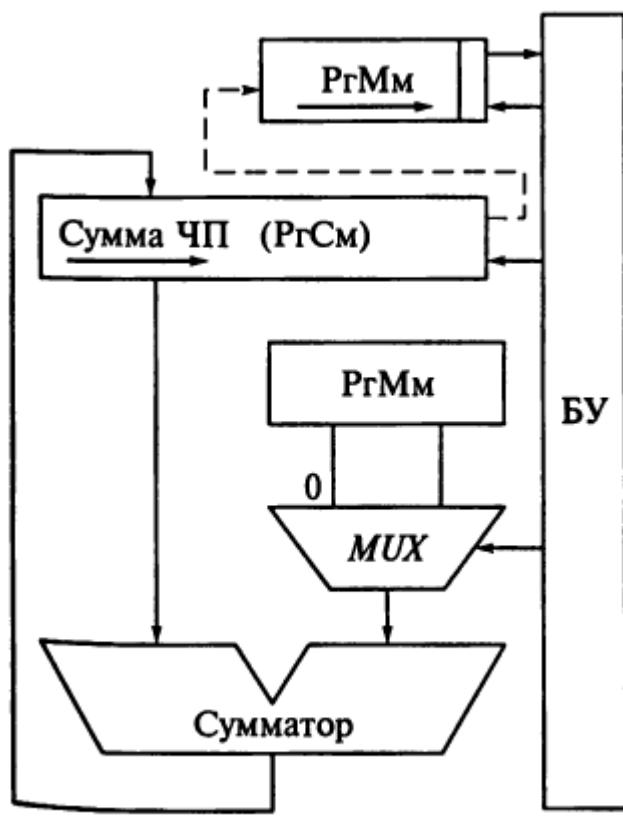
Делимое $A_{10} = +145$	00 1001 0001	1101	Делитель $B_{10} = -13$
Вычитание делителя	+ 11 0011	1011	Частное $(A/B)_{10} = -11$
Остаток < 0	= 11 1100 0001	↑↑↑↑	Деление корректно
Сдвиг остатка влево	11 1000 001		
Прибавление делителя	+ 00 1101 001		
Остаток > 0	= 00 0101 001	--↑	
Сдвиг остатка влево	00 1010 01		
Вычитание делителя	+ 11 0011		
Остаток < 0	= 11 1101 01	---↑	
Сдвиг остатка влево	11 1010 1		
Прибавление делителя	+ 00 1101		
Остаток > 0	= 00 0111 1	----↑	
Сдвиг остатка влево	00 1111		
Вычитание делителя	+ 11 0011		
Остаток > 0	= 00 0010	-----↑	Остаток $R_{10} = +2$

Рис. 4.13. Деление чисел без восстановления остатка

Деление без восстановления остатка может быть реализовано устройством деления (рис. 4.14). При анализе его схемы можно установить, что в ее состав входят те же узлы, которые составляют схему умножения (см. рис. 4.9, а). Эти схемы отличаются лишь направлением сдвига содержимого регистра сумматора и регистра множителя (частного). Используя регистры со сдвигом в двух направлениях, можно построить комбинированную схему умножения—деления (рис. 4.15),

Младшие разряды 2_{mn} —разрядного регистра сумматора используются в качестве регистра множителя-частного, что позволяет уменьшить аппаратные затраты,

Блок умножения-деления настраивается на выполнение заданной операции сигналом кода операции, поступающим из центрального устройства управления.



a

Рис 4.9

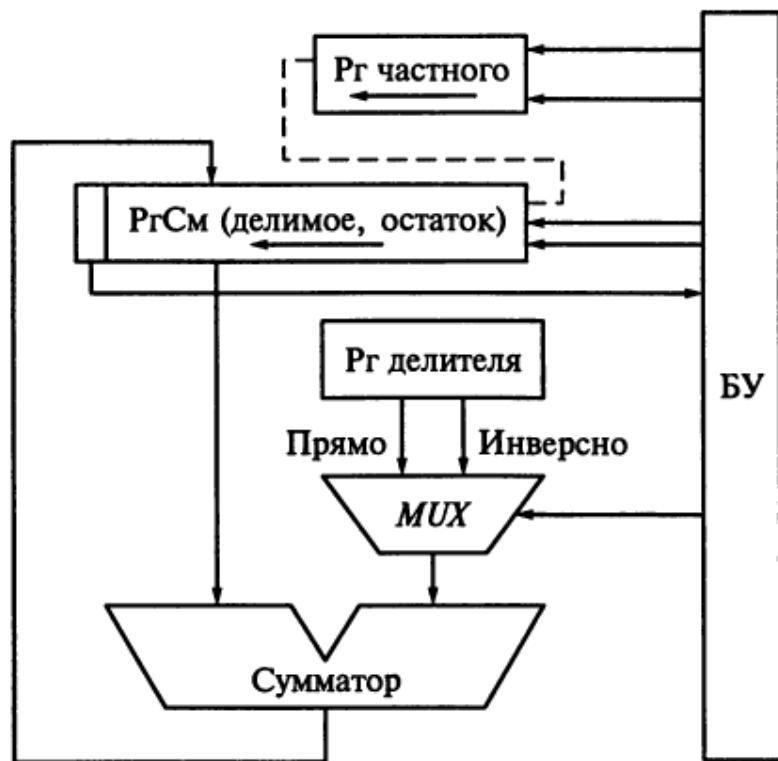


Рис. 4.14. Структура устройства деления

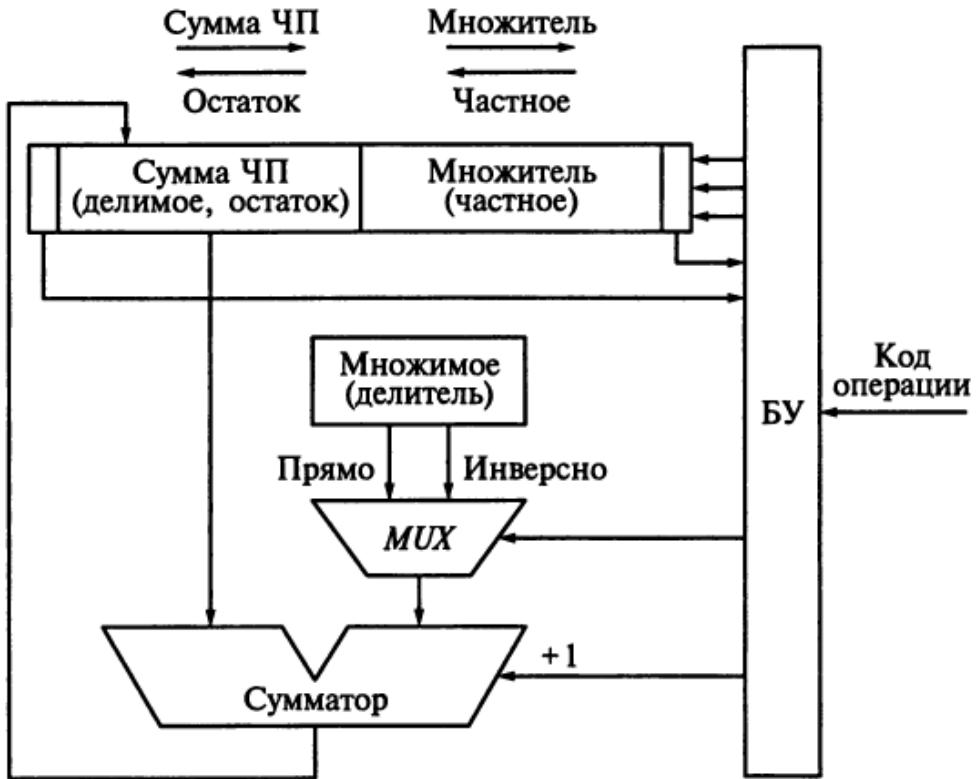
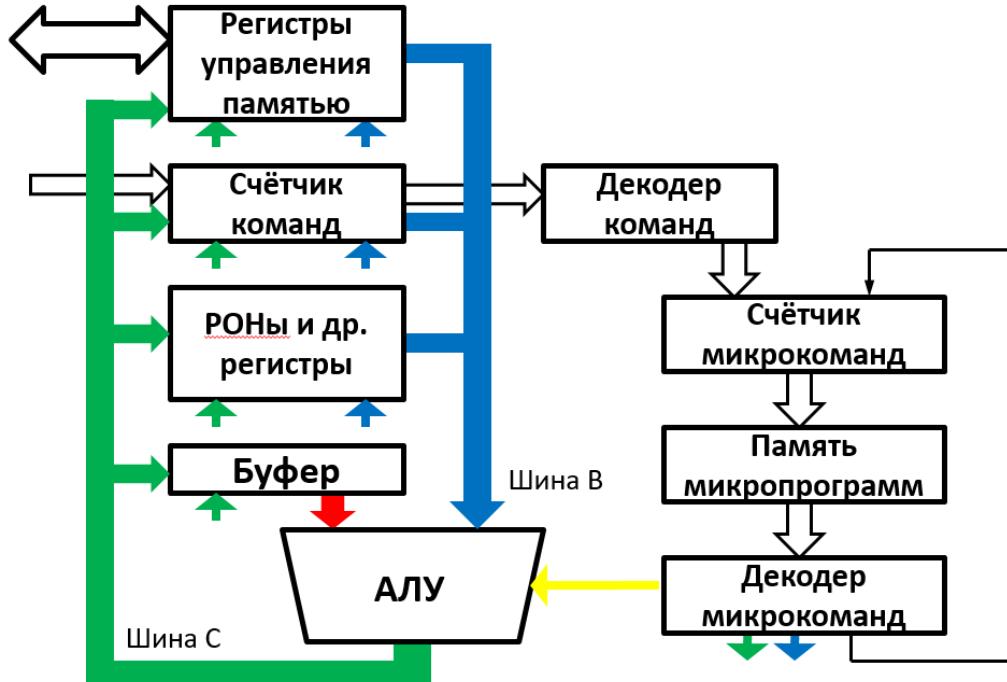


Рис. 4.15. Структура операционного блока умножения-деления

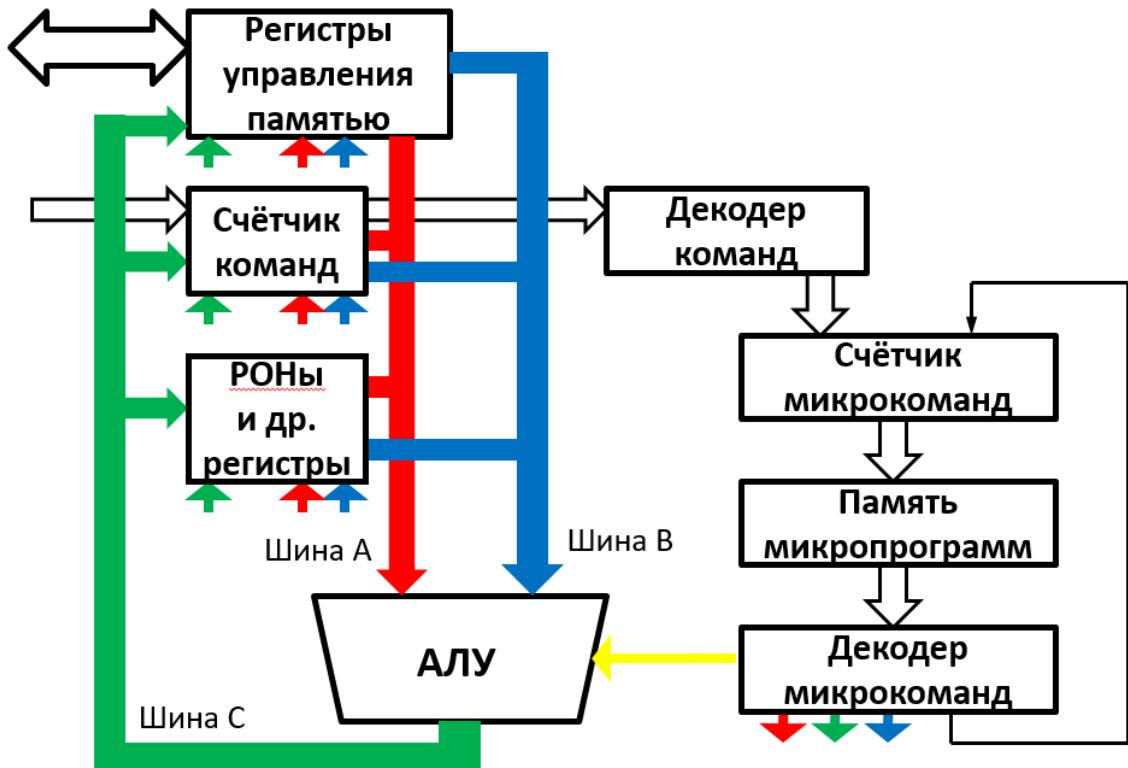
20. Простейшая микроархитектура процессора. Преимущества трёхшинной микроархитектуры.



Микропроцессор (МП) — центральное устройство ПК, предназначенное для управления работой всех блоков машины и для выполнения арифметических и логических операций над информацией.

Считывание операндов происходит последовательно: сначала первый считывается с регистров, проходит через АЛУ и записывается в буфер. Затем считывается из регистров второй operand и вместе с первым operandом из буфера поступает на вход в АЛУ(итого 2 машинных цикла).

3-х шинная архитектура



В сигнальных процессорах Гарвардская архитектура дополняется применением трехшинного операционного блока микропроцессора. Трехшинная архитектура операционного блока позволяет совместить операции считывания двух operandов с записью результата выполнения команды в оперативную память микропроцессора. Это значительно увеличивает производительность сигнального микропроцессора без увеличения его тактовой частоты.

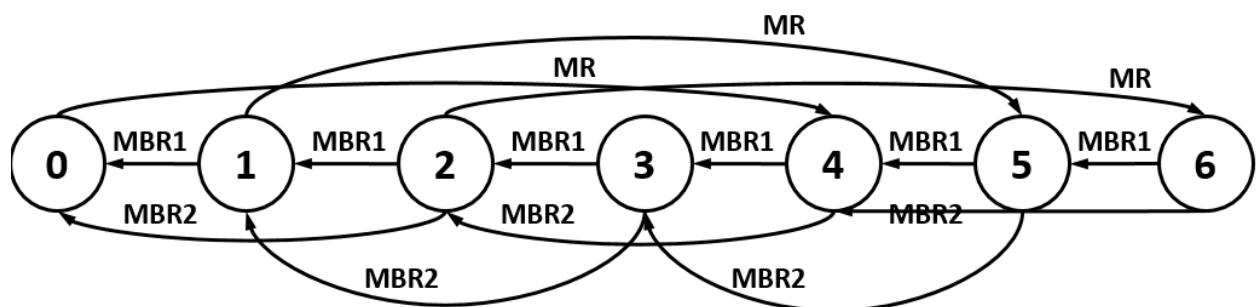
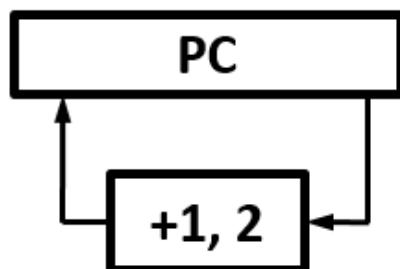
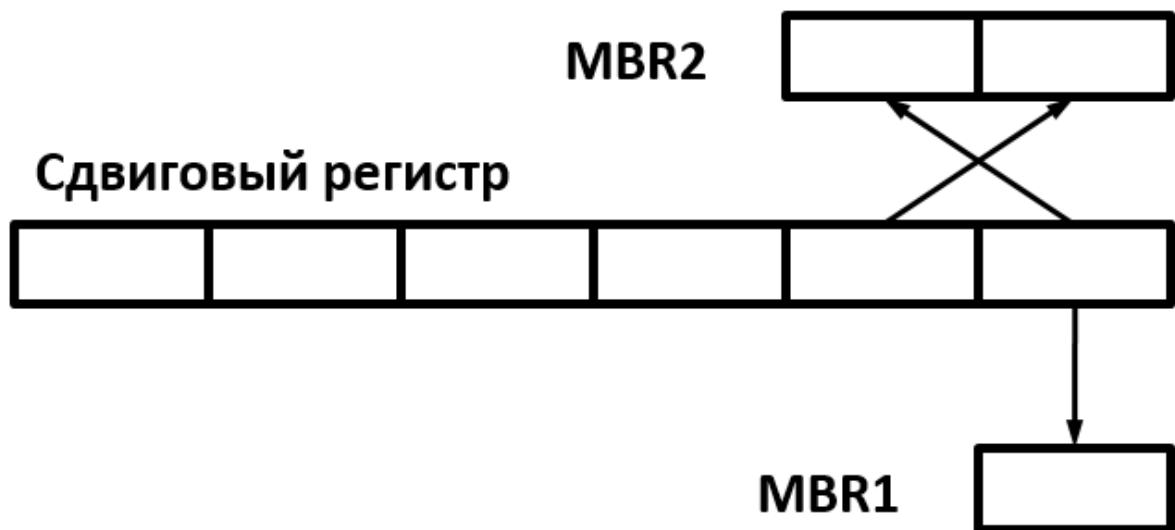
Это позволяет за один машинный цикл произвести:

- выборку команды посредством шины адреса программ и шины данных программ;
- выборку двух operandов для операции умножения посредством двух шин адреса данных;
- занесение operandов в аккумуляторы посредством двух шин данных;
- операцию умножения;
- сохранить результат в аккумуляторе.

21. Предвыборка команд: принципы построения, алгоритм функционирования

Instruction Fetch Unit (IFU)

Блок выборки команд



Время выполнения одной команды складывается из времени выполнения операции и времени обращения к памяти. Для трехадресной команды последнее суммируется из четырех составляющих времени:

- выборки команды;
- выборки первого операнда;
- выборки второго операнда;
- записи в память результата.

Одноадресная команда требует двух обращений к памяти:

- выборки команды;

- выборки операнда.

Счетчик команд определяет лишь местоположение команды в памяти, но не хранит информации о ее содержании. Чтобы приступить к выполнению команды, ее необходимо извлечь из памяти и разместить в регистре команды (РК). Этот этап носит название выборки команды. Только с момента загрузки команды в РК она становится «видимой» для процессора. В РК команда хранится в течение всего времени ее выполнения. Как уже отмечалось ранее, любая команда содержит два поля: поле кода операции и поле адресной части. Учитывая это обстоятельство, регистр команды иногда рассматривают как совокупность двух регистров — регистра кода операции (РКОП) и регистра адреса (РА), в которых хранятся соответствующие составляющие команды.

Этап выборки команды

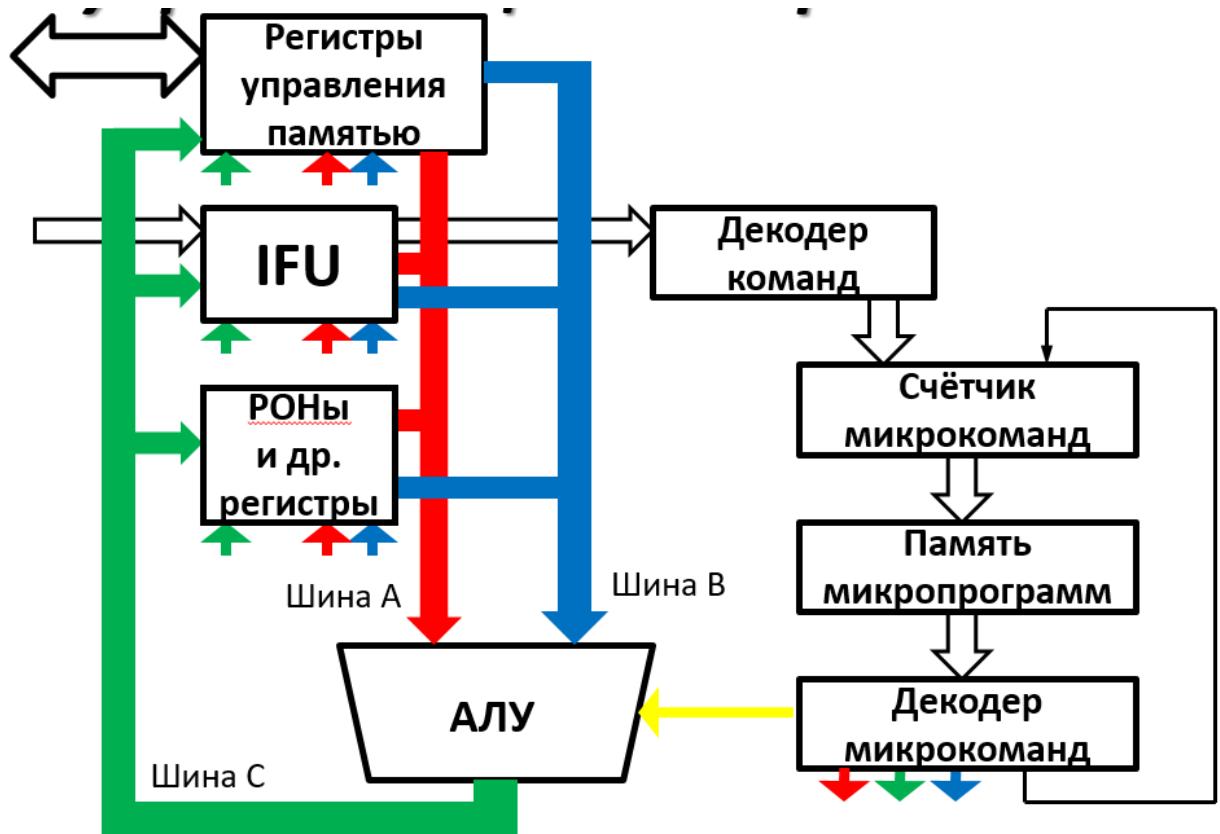
Цикл любой команды начинается с того, что центральный процессор извлекает команду из памяти, используя адрес, хранящийся в счетчике команд (СК). Двоичный код команды помещается в регистр команды (РК) и с этого момента становится «видимым» для процессора. Без учета промежуточных пересылок и сигналов управления это можно описать следующим образом: $\text{РК} := \text{ОП}[(\text{СК})]$.

Приведенная запись охватывает весь этап выборки, если длина команды совпадает с разрядностью ячейки памяти. В то же время система команд многих ВМ предполагает несколько форматов команд, причем в разных форматах команда может занимать 1, 2 или более ячеек, а этап выборки команды можно считать завершенным лишь после того, как в РК будет помещен полный код команды. Информация о фактической длине команды содержится в полях кода операции и способа адресации. Обычно эти поля располагают в первом слове кода команды, и для выяснения необходимости продолжения процесса выборки необходимо предварительное декодирование их содержимого.

Такое декодирование может быть произведено после того, как первое слово кода команды окажется в РК. В случае многословного формата команды процесс выборки продолжается вплоть до занесения в РК всех слов команды.

22. Микроархитектура с предвыборкой команд

Микроархитектура с упреждающей выборкой команд



23. Простейшая конвейерная микроархитектура

Конвейерное выполнение команд — это одновременное выполнение разных тактов последовательных команд в разных частях МП при непосредственной передаче результатов из одной части МП в другую. Конвейерное выполнение команд увеличивает эффективное быстродействие ПК в 2–5 раз;

Разработчики архитектуры компьютеров издавна прибегали к методам проектирования, известным под общим названием "совмещение операций", при котором аппаратура компьютера в любой момент времени выполняет одновременно более одной базовой операции. Этот общий метод включает два понятия: параллелизм и конвейеризацию. Хотя у них много общего и их зачастую трудно различать на практике, эти термины отражают два совершенно различных подхода. При параллелизме совмещение операций достигается путем воспроизведения в нескольких копиях аппаратной структуры. Высокая производительность достигается за счет одновременной работы всех элементов структур, осуществляющих решение различных частей задачи.

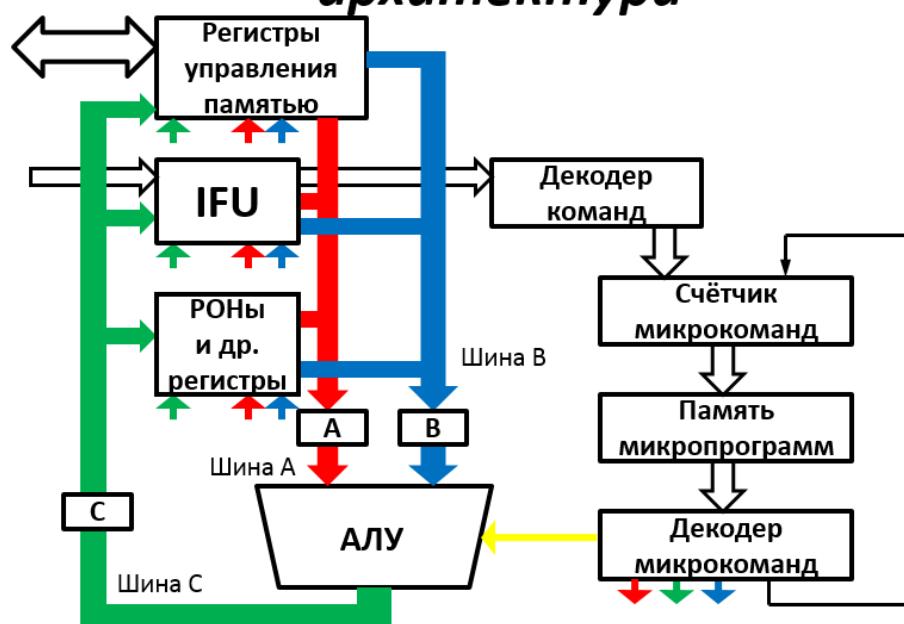
Конвейеризация (или конвейерная обработка) в общем случае основана на разделении подлежащей исполнению функции на более мелкие части, называемые ступенями, и выделении для каждой из них отдельного блока аппаратуры. Так обработку любой машинной команды можно разделить на несколько этапов (несколько ступеней), организовав передачу данных от одного этапа к следующему. При этом конвейерную обработку можно использовать для совмещения этапов выполнения разных команд. Производительность при этом возрастает благодаря тому, что одновременно на различных ступенях конвейера выполняются несколько команд. Конвейерная обработка такого рода широко применяется во всех современных быстродействующих процессорах.

В каждый момент времени на конвейерном процессоре обрабатывается несколько команд, причем все они находятся на разных стадиях обработки. Приход очередного тактового

импульса приводит к перемещению обрабатываемых команд на следующую ступень конвейера, полностью выполненная команда покидает конвейер, а на освободившуюся первую ступень подается новая команда.

Будем считать, что набор команд нашего процессора включает типичные арифметические и логические операции, операции с плавающей точкой, операции пересылки данных, операции управления потоком команд и системные операции. В арифметических командах используется трехадресный формат, типичный для RISC-процессоров, а для обращения к памяти используются операции загрузки и записи содержимого регистров в память.

Простейшая конвейерная архитектура



15

24. Стадии простейшего 5-ти ступенчатого конвейера

Выполнение типичной команды можно разделить на следующие этапы:

- выборка команды - IF (по адресу, заданному счетчиком команд, из памяти извлекается команда);
- декодирование команды / выборка operandов из регистров - ID;
- выполнение операции / вычисление эффективного адреса памяти - EX;
- обращение к памяти - MEM;
- запоминание результата - WB.

5-ти ступенчатый конвейер

Instruction Fetch (IF)

Instruction Decode (ID)

Read Data (RD)

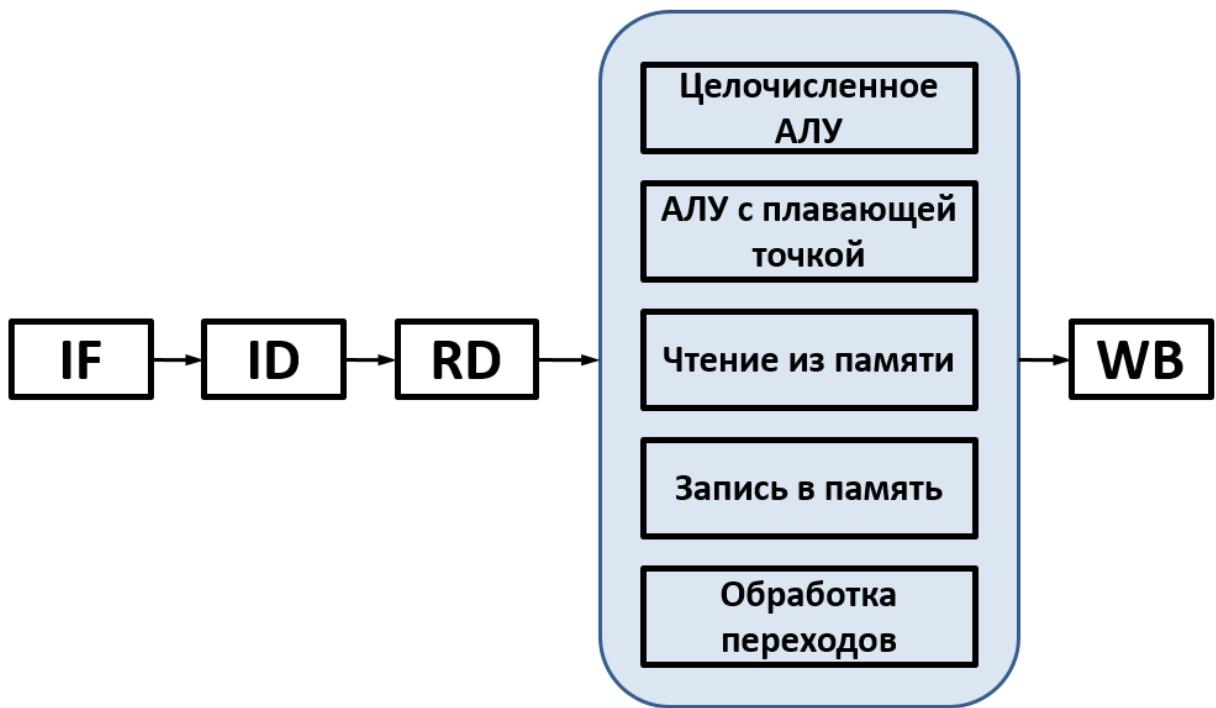
Execute (EX)

Write Back (WB)

	1	2	3	4	5	6	7	8	9	10
i	IF	ID	RD	EX	WB					
i+1		IF	ID	RD	EX	WB				
i+2			IF	ID	RD	EX	WB			
i+3				IF	ID	RD	EX	WB		
i+4					IF	ID	RD	EX	WB	
i+5						IF	ID	RD	EX	WB

25. Принцип суперскалярности

Суперскалярная архитектура



Суперскалярная архитектура процессора предполагает, что команды (на некотором ограниченном участке программы) могут выполняться не только в порядке их размещения в программе, но и по мере возможности их выполнения независимо от порядка следования. Возможность выполнения определяется, во-первых, отсутствием зависимостей от ранее расположенных, но еще не завершенных команд, во-вторых, наличием свободных ресурсов процессора, необходимых для выполнения команды.

Суперскалярным называется центральный процессор (ЦП), который одновременно выполняет более чем одну скалярную команду. Это достигается за счет включения в состав ЦП нескольких самостоятельных функциональных блоков, каждый из которых отвечает за свой класс операций и может присутствовать в процессоре в нескольких экземплярах.

Суперскалярность предполагает параллельную работу нескольких функциональных блоков, что возможно лишь при одновременном выполнении нескольких скалярных команд. Последнее условие хорошо сочетается с конвейерной обработкой, при этом желательно, чтобы таких конвейеров было несколько, например два или три. Разумеется, в этом случае степень выборки команд, общая для всех конвейеров, должна в каждом такте извлекать из памяти сразу несколько команд. За этой ступенью располагается блок диспетчеризации, отвечающий за распределение команд по конвейерам.

26. Разновидности конфликтов при конвейерном исполнении команд и пути их преодоления

Конфликты в конвейерах:

Конфликты по ресурсам (структурный риск)

- память
- регистры
- «длинные» операции

Конфликты по данным

- «длинные» операции
- зависимости типа RAW (read after write) (действительные)
- зависимости типов WAR (write after read) и WAW(write after write) (ложные)

Конфликты по управлению

- безусловные переходы
- условные переходы

Конфликтные ситуации в конвейере принято обозначать термином риск (hazard), а обусловлены они могут быть тремя причинами:

- попыткой нескольких команд одновременно обратиться к одному и тому же ресурсу ВМ (структурный риск ИЛИ конфликт по ресурсам);
- взаимосвязью команд по данным (конфликт по данным);
- неоднозначностью при выборке следующей команды в случае команд, изменяющих последовательность вычислений (конфликты по управлению).

Структурный риск (**конфликт по ресурсам**) имеет место, когда несколько команд, находящихся на разных ступенях конвейера, пытаются одновременно использовать один и тот же ресурс, чаще всего — память. Так, в типовом цикле команды (см. рис. 9.3) сразу три этапа (ВК, ВО и ЗР) связаны с обращением к памяти. Подобных конфликтов частично

удается избежать за счет блочной организации основной памяти и использования кэш-памяти — имеется вероятность того, что команды будут обращаться либо к разным банкам ОП, либо одна из них станет обращаться к основной памяти, а другая — к кэш-памяти. С этих позиций кэш-память выгоднее разделять на кэш-память команд и кэш-память данных. Конфликты из-за одновременного обращения к памяти могут и не возникать, поскольку для многих команд этапы выборки операнда и записи результата не требуются. В целом, влияние структурного риска на производительность конвейера по сравнению с другими видами рисков сравнительно невелико.

Конфликт по памяти решается разделением кэша первого уровня на кэш инструкций и данных.



Рис. 9.3. Логика работы конвейера команд

По регистрам - один регистр задействован несколькими инструкциями. Решение - добавить регистров, но тогда теряется совместимость ОС с кэшем 2 и 3 уровня. Решение - добавить регистров, но вести таблицу соответствий между физическими регистрами и архитектурными. Таким образом названия регистров и количество не изменилось для ОС, но на микроуровне, то есть физически за одним архитектурным регистром стоит несколько физических.

Длинные операции – операции, которые забирают более 1 такта. Решение: принцип суперскалярности.

Риск по данным, в противоположность риску по ресурсам (структурному риску), — типичная и регулярно возникающая ситуация. Для пояснения сущности взаимосвязи команд по данным положим, что две команды в конвейере (*i* и *j*) предусматривают обращение к одной и той же переменной *x*, причем команда *i* предшествует команде *j*. В общем случае между *i* и *j* ожидаются три типа конфликтов по данным (рис. 9.4):

- «чтение после записи» (RAW): команда *j* читает *x* до того, как команда *i* успела записать новое значение *x*, то есть команда *j* ошибочно получит старое значение *x* вместо нового.
- «запись после чтения» (WAR): команда *j* записывает новое значение *x* до того, как команда *i* успела прочитать *x*, то есть команда *i* ошибочно получит новое значение *x* вместо старого.
- «запись после записи» (WAW): команда *j* записывает новое значение *x* прежде, чем команда *i* успела записать в качестве *x* свое значение, и окончательно *x* ошибочно будет содержать значение, определенное командой *i*, а не командой *j*.

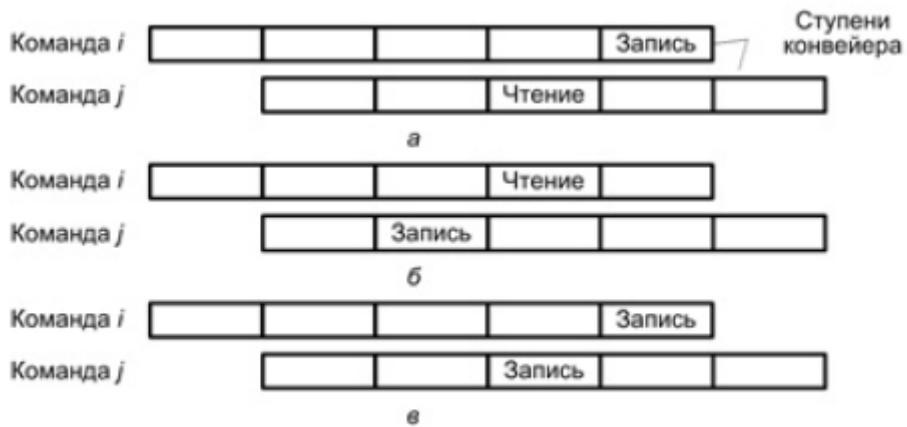


Рис. 9.4. Конфликты по данным: а — «чтение после записи»; б — «запись после чтения»; в — «запись после записи»

Поскольку наиболее частые конфликты по данным — это RAW, основные усилия тратятся на противодействие именно этому типу конфликтов. Среди известных методов борьбы с RAW наибольшее распространение получил прием ускоренного продвижения информации (forwarding). Обычно между двумя соседними ступенями конвейера располагается буферный регистр, через который предшествующая ступень передает результат своей работы на последующую ступень, то есть передача информации возможна лишь между соседними ступенями конвейера. При ускоренном продвижении, когда для выполнения команды требуется операнд, уже вычисленный предыдущей командой, этот операнд может быть получен непосредственно из соответствующего буферного регистра, минуя все промежуточные ступени конвейера. С данной целью в конвейере предусматриваются дополнительные тракты пересылки информации (тракты опережения, тракты обхода), снабженные средствами коммутации трактов.

Зависимости RAW решаются поочередным исполнением команд (отложить выполнение $i+2$ команды и выполнить $i+3$, $i+4$ команды, пока ждем $i+2$ команду). Реализовать можно перестановкой команд. На процессоре это делать не оптимально, потому что далеко не посмотреть все последующие команды, поэтому это делается на этапе компиляции. В процессор загружаются команды уже в нужном порядке.

Зависимости WAR, WAW (ложные, возникают только в суперскалярных архитектурах) возникают, когда команда, которая начала позже выполняется, выполнилась раньше (операции выполняются на разных устройствах, поэтому это возможно).

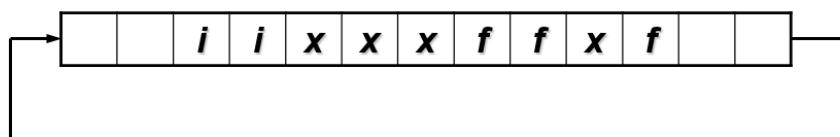
Обработка «длинной» операции

	1	2	3	4	5	6	7	8	9	10	11	12
i	IF	ID	RD	EX	WB							
i+1		IF	ID	RD	EX	EX	EX	WB				
i+2			IF	ID	RD	-	-	EX	WB			
i+3				IF	ID	-	-	RD	EX	WB		
i+4					IF	-	-	ID	RD	EX	WB	
i+5						-	-	IF	ID	RD	EX	WB

Зависимости типа RAW

	1	2	3	4	5	6	7	8	9	10	11
i	IF	ID	RD	EX	WB						
i+1		IF	ID	RD	EX	WB					
i+2			IF	ID	-	-	RD	EX	WB		
i+3				IF	ID	-	-	RD	EX	WB	
i+4					IF	ID	-	-	RD	EX	WB
	1	2	3	4	5	6	7	8	9	10	11
i	IF	ID	RD	EX	WB						
i+1		IF	ID	RD	EX	WB					
i+3			IF	ID	RD	EX	WB				
i+4				IF	ID	RD	EX	WB			
i+2					IF	ID	RD	EX	WB		

Неупорядоченная обработка команд



Буфер восстановления последовательности

В качестве инструмента для поддержания правильной последовательности исполнения команд (в случае нескольких параллельно работающих функциональных блоков) может быть использован *буфер восстановления последовательности*. Стратегия буфера восстановления последовательности (БВП) впервые была описана Смитом и Плескуном (Smith and Pleszkun) в 1988 году.

БВП представляет собой кольцевой буфер (рис. 9.34) с указателями головной и хвостовой части. Указатель головной части содержит адрес следующего свободного входа. Команды заносятся в БВП в порядке, определяемом программой. Каждая выданная команда помещается в следующую свободную ячейку буфера (говорят, что команде выделен очередной свободный вход БВП), причем выделение ячеек идет с соблюдением последовательности выдачи команд. Каждый занятый вход содержит также информацию о состоянии хранимой в нем команды: команда только выдана (*i*), находится в стадии выполнения (*x*) или уже завершена (*f*). Указатель хвостовой части показывает на команду, подлежащую удалению из БВП прежде других. Удаление команды разрешено, только если она завершена и все предшествующие ей команды уже удалены из буфера. Этот механизм гарантирует, что

команды покидают БВП строго по порядку. Очередность выполнения команд программы сохраняется благодаря тому, что заносить свои результаты в память или регистры разрешается лишь тем командам, которые покинули БВП.

Число входов в БВП в разных процессорах составляет от 5 (PowerPC 603) до 64 (SPARC64).



Рис. 9.34. Принципы организации буфера восстановления последовательности

Название буфера подчеркивает его основную задачу — поддержание строгой последовательности завершения команд путем переупорядочивания тех из них, которые исполнялись с нарушением этой последовательности. Однако БВП более универсален — с равным успехом он годится и для переименования регистров и для распределения декодированных команд по накопителям (станциям резервирования). Так, по своему основному назначению БВП применен в микропроцессорах PowerPC 603, PowerPC 604, R10000. В микропроцессорах Am29000, AMD K5, Pentium Pro буфер используется также для переименования регистров. Наконец, в системе Lightning БВП реализует все три из вышеперечисленных функций.

Обработка перехода

	1	2	3	4	5	6	7	8	9	10
i	IF	ID	RD	EX	WB					
i+1		IF	ID	RD	EX	WB				
i+2			IF	ID	RD	-	-			
i+3				IF	ID	-	-	-		
k					IF	ID	RD	EX	WB	
k+1						IF	ID	RD	EX	WB

27. Динамическое прогнозирование ветвлений

В динамических стратегиях решение о наиболее вероятном исходе команды УП принимается в ходе вычислений исходя из информации о предшествующих переходах (истории переходов), собираемой в процессе выполнения программы.

Идея динамического предсказания переходов предполагает накопление информации о том, как завершались команды УП при их предшествующем исполнении. История переходов фиксируется в форме таблицы, каждый элемент которой состоит из m битов. Нужный элемент таблицы выбирается с помощью k-разрядной двоичной комбинации — шаблона (pattern). Этим объясняется общепринятое название таблицы предыстории переходов — таблица истории для шаблонов (РНТ, Pattern History Table).

При описании динамических схем предсказания переходов их часто расценивают как один из видов автоматов Мура, при этом содержимое элементов РНТ трактуется как информация, отображающая текущее состояние автомата. Если команда завершилась переходом, то в соответствующий элемент РНТ заносится единица, иначе — ноль. Очередное предсказание совпадает с итогом предыдущего выполнения команды. После исполнения очередной команды содержимое элемента корректируется в соответствии с ее итогом.

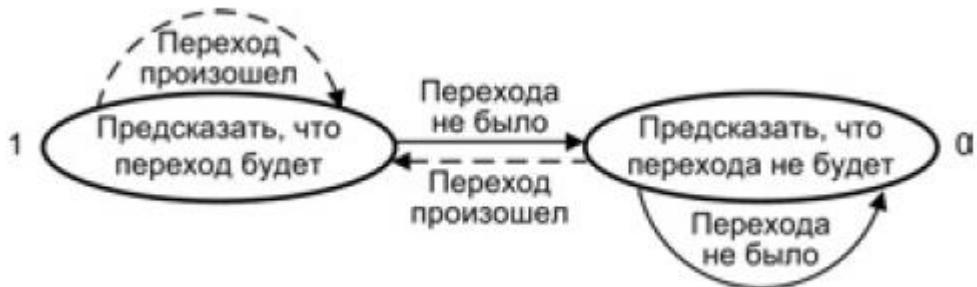
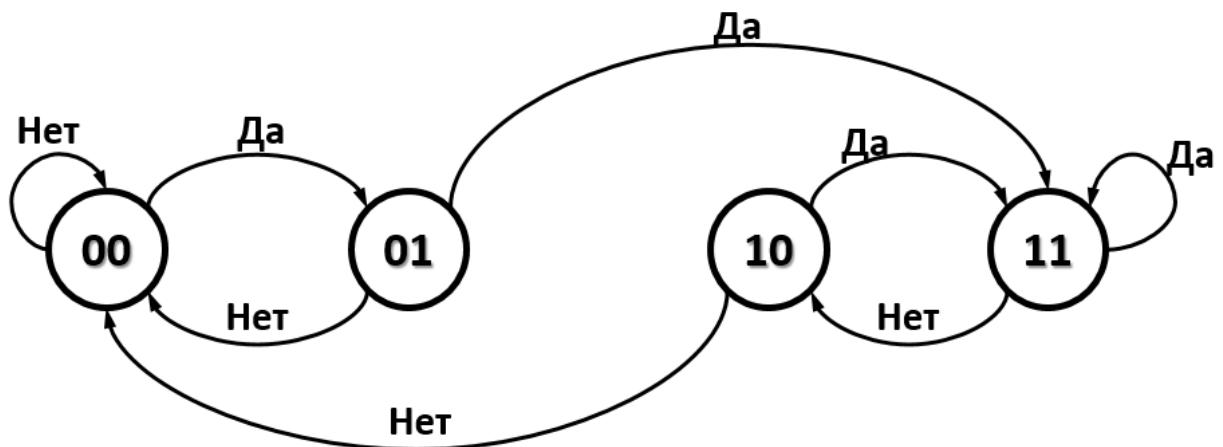


Рис. 9.6. Диаграмма состояний автомата А1



00 – прогнозирование отсутствия перехода

01 – повторное прогнозирование отсутствия перехода

10 – повторное прогнозирование перехода

11 – прогнозирование перехода

25

28. Переименование регистров

Для устранения зависимости по данным используется прием, известный как переименование регистров. Способ решения второй проблемы обобщенно называют переупорядочиванием команд или откладыванием исполнения команд.

Когда команды выдаются и завершаются упорядоченно, каждый регистр в любой точке программы содержит именно то значение, которое диктуется программой. При неупорядоченной выдаче и завершении команд запись в регистры может происходить также неупорядоченно, и отдельные команды, обратившись к какому-то регистру, вместо нужного получат «устаревшее» или «опережающее» значение.

Основная идея переименования регистров состоит в том, что каждый новый результат записывается в один из свободных в данный момент дополнительных регистров, при этом ссылки на заменяемый регистр во всех последующих командах соответственным образом корректируются

Пусть имеется последовательность команд:

I1: MUL R2, R0, R1 { $R2 \leftarrow R0 \times R1$ }
I2: ADD R0, R1, R2 { $R0 \leftarrow R1 + R2$ }
I3: SUB R2, R0, R1 { $R2 \leftarrow R0 - R1$ }.

Неупорядоченные выдачи/завершения могут привести к неверному результату, например:

- команда I2 была исполнена до того, как I1 успела записать в регистр R2 свой результат, то есть I2 использовала «старое» содержимое R2;
- команда I3 исполнена раньше, чем I1, в результате чего неверный результат будет получен в I2, а по завершении цепочки из трех команд в R2 останется результат I1 вместо результата I3.

Вводя новые регистры R0a и R2a, получим иную последовательность:

I1: MUL R2, R0, R1 { $R2 \leftarrow R0 \times R1$ }
I2: ADD R0a, R1, R2 { $R0a \leftarrow R1 + R2$ }
I3: SUB R2a, R0a, R1 { $R2a \leftarrow R0a - R1$ },

где возможность конфликта устранена. Такой метод известен как *переименование регистров* (register renaming).

Имя АР	Имя ФР
AR1[1]	PhR1
AR1[2]	PhR2
AR2[1]	PhR3
AR3[1]	PhR4
AR3[2]	PhR5
AR3[3]	PhR6
AR4[1]	PhR7

- ✓ Конфликты по ресурсам
- ✓ Зависимости типа WAR и WAW

29. Внеочередное и спекулятивное выполнение команд

Думаю картинка скорее к конфликтам относится (узнаем на консультации)

Процессоры, допускающие работу одного из конвейеров при «заторе» в других, называют процессорами, обладающими архитектурой с *неупорядоченной обработкой*. Если разрешить выполнение команд в одном из конвейеров при возникновении «затора» в другом, то более поздние команды программы могут оказаться выполненными раньше предшествующих им. Для исключения неправильных результатов нужно, чтобы результаты этих операций не заносились в память, а содержимое регистров не модифицировалось в ошибочной последовательности, т. е. должны быть предусмотрены схемы и методы, позволяющие восстановить правильность вычислительного процесса.

Для реализации неупорядоченной обработки между ступенями декодирования и исполнения в конвейере размещают специальную буферную память, или накопитель команд. Если процессор определяет, что текущая команда не может выполняться, то он забирает из накопителя следующую команду и посыпает ее в исполнительный блок. Этот накопитель заполняется командами последовательно, однако порядок формирования результатов может оказаться нарушенным, так как команды имеют разную длительность выполнения, а их исполнение может даже откладываться.

Часто задержки происходят из-за недостаточности аппаратных ресурсов. Например, в компьютере может быть предусмотрен только один порт записи в регистры, а при определенных обстоятельствах конвейеру может потребоваться выполнить две записи в регистровый файл одновременно. Это приводит к конфликту, в результате которого происходит приостановка выполнения команд в одном из конвейеров до тех пор, пока требуемое устройство (один из портов) не станет доступным. Подобную ситуацию называют конвейерным «пузырем» — он проходит по конвейеру, не вызывая никакой обработки, но занимает место в конвейере, а следовательно, снижается производительность обработки.

При наличии в компьютере нескольких конвейеров результаты даже последовательных команд могут формироваться в неупорядоченной последовательности. Пусть нужно выполнить две совершенно независимые команды умножения и сложения. Команда умножения попадает на конвейер первой. Вслед за ней на второй конвейер попадает команда сложения, но ее результат будет получен раньше, так как умножение требует нескольких тактов, а команда сложения одного. Следовательно, имеет место *неупорядоченное завершение* команд.

Если процессор обладает несколькими конвейерами, то неупорядоченное завершение команд неизбежно. При этом резуль-

тат более поздней, но короткой операции может появиться раньше и «занять» регистр, предназначенный для результата предыдущей более длинной команды. Кроме того, необходимо обеспечить получение такого же результата, как и при строго последовательном выполнении команд. Эти проблемы обычно решаются на этапе компиляции программы, но для их решения могут быть использованы и специальные приемы — переименование регистров и переупорядочивание команд.

Внеочередное исполнение — исполнение машинных инструкций не в порядке следования в машинном коде, а в порядке готовности к выполнению.

Одной из главных особенностей шестого поколения микропроцессоров архитектуры IA32(Intel Architecture, 32-bit - третье поколение архитектуры x86, ознаменовавшееся переходом на 32-разрядные вычисления. Также архитектуру часто называют i386 и x86 1985г.) является динамическое (спекулятивное) исполнение. Под этим термином подразумевается следующая совокупность возможностей:

- Глубокое предсказание ветвлений (с вероятностью >90% можно предсказать 10-15 ближайших переходов).
- Анализ потока данных (на 20-30 шагов вперед просмотреть программу и определить зависимость команд по данным или ресурсам).
- Опережающее исполнение команд (МП Р6 может выполнять команды в порядке, отличном от их следования в программе).

Упрощённо, это технология в современных микропроцессорах, которая позволяет «предугадывать», какие команды потребуется исполнить в будущем.

Например, программа А выполняет сложную математическую операцию, в ходе которой будет использована сумма каких-то переменных. Программа В заранее (когда процессор менее нагружен) считает эту сумму, чтобы ускорить работу процесса в будущем, и в нужный момент подсовывает результат программе А. Если результат программы В не потребовался программе А, то ничего страшного, а если потребовался, то процессор ускорил свою работу.

Всё это происходит во внутренней памяти процессора на нескольких слоях кэша (в зависимости от архитектуры чипа)

30. Понятие об архитектуре набора команд процессора

- Типы обрабатываемых данных и способы их представления
- Состав программно доступных регистров процессора
- Адресные структуры и режимы адресации
- Состав и формат команд

Системой команд вычислительной машины называют полный перечень команд, которые способна выполнять данная ВМ. В свою очередь, под архитектурой системы команд

(ACK) принято определять те средства вычислительной машины, которые видны и доступны программисту. ACK можно рассматривать как линию согласования нужд разработчиков программного обеспечения с возможностями создателей аппаратуры вычислительной машины (рис. 2.1).



Рис. 2.1. Архитектура системы команд как интерфейс между программным и аппаратным обеспечением

В конечном итоге цель тех и других — реализация вычислений наиболее эффективным образом, то есть за минимальное время, и здесь важнейшую роль играет правильный выбор архитектуры системы команд. В упрощенной трактовке время выполнения программы ($T_{выч}$) можно определить через число команд в программе ($N_{ком}$), среднее количество тактов процессора, приходящихся на одну команду (CPI) и длительность тактового периода t_{np} :

$$T_{выч} = N_{ком} \times CPI \times t_{np}.$$

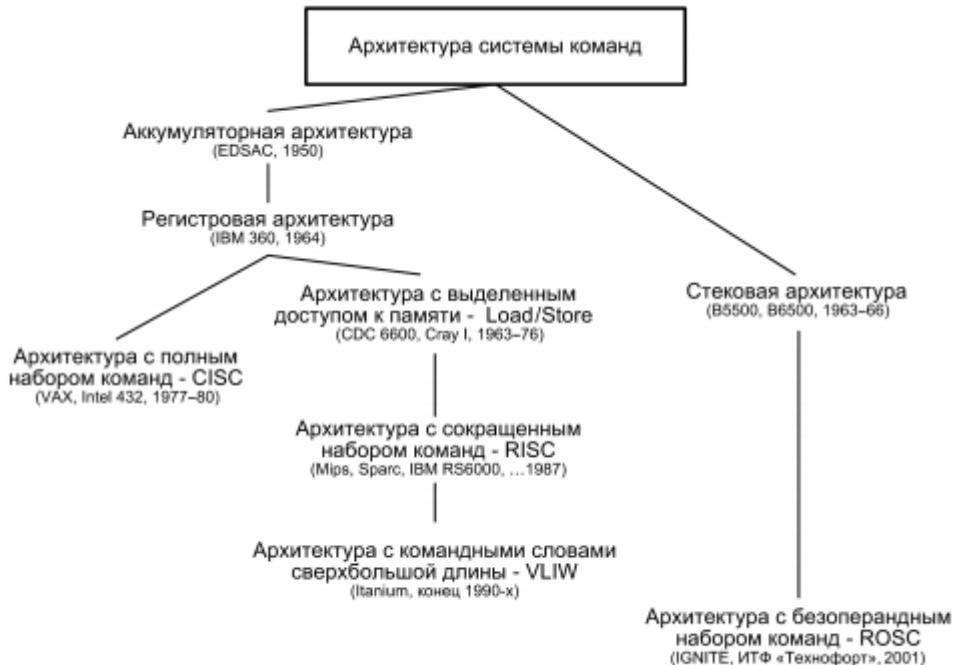


Рис. 2.3. Хронология развития архитектур системы команд

Современная технология программирования ориентирована на языки высокого уровня (ЯВУ), главная цель которых — облегчить процесс программирования. Но переход к ЯВУ породил серьезную проблему: сложные операторы, характерные для ЯВУ, существенно отличаются от простых машинных операций, реализуемых в большинстве

вычислительных машин. Следствием такого несоответствия становится недостаточно эффективное выполнение программ на ВМ. Проблема получила название семантического разрыва, а для ее разрешения разработчики вычислительных машин в настоящее время выбирают один из трех подходов и, соответственно, один из трех типов ACK:

- архитектуру с полным набором команд: CISC (Complex Instruction Set Computer);
- архитектуру с сокращенным набором команд: RISC (Reduced Instruction Set Computer);
- архитектуру с командными словами сверхбольшой длины: VLIW (Very Long Instruction Word).

Выбору реализуемой компьютером системы команд должно уделяться самое серьезное внимание. Каждая аппаратно реализуемая операция, входящая в систему команд, выполняется быстрее, чем аналогичная операция, не входящая в систему команд и, следовательно, реализуемая в виде подпрограммы. На первый взгляд из этого следует, что увеличение аппаратно реализуемых операций, т. е. расширение системы команд, может привести к повышению быстродействия машины. Этот вывод положен в основу концепции архитектуры компьютера с расширенным набором команд — CISC-архитектур. Однако расширение системы команд может приводить и к обратным результатам. Именно это обстоятельство послужило толчком для разработки машин с сокращенным набором команд — RISC-архитектур.

31. Архитектура CISC: принципы построения, преимущества, недостатки, признаки

Основную идею CISC-архитектуры отражает ее название — «полный набор команд». В данной архитектуре стремятся иметь отдельную машинную команду для каждого возможного (типового) действия по обработке данных. Исторически CISC-архитектура была одной из первых. Совершенствование процессоров шло по пути создания ВМ, способных выполнять как можно больше разных команд.

- + Это упрощало работу программистов, которые писали программы на языке ассемблера (то есть практически на уровне машинных команд).
- + Использование сложных команд позволяло сократить размер и время разработки программы.
- Некоторые команды стало невозможно выполнять чисто аппаратными средствами (при разумной сложности таких средств). В результате в процессорах появились блоки, «на лету» заменяющие наиболее сложные команды последовательностями из более простых команд.
- Практика показала, что многие сложные команды при написании программ оказывались просто невостребованы.
- Из-за высокой сложности команд и их обилия устройство управления ВМ приходилось строить только на основе программируемой логики, то есть с применением «медленной» управляющей памяти, что существенно ограничивало возможности наращивания тактовой частоты процессора.

В итоге сложились следующие черты организации CISC-процессоров:

- большое количество различных машинных команд (сотни), каждая из которых выполняется за несколько тактов центрального процессора;
- устройство управления с программируемой логикой;
- небольшое количество регистров общего назначения (РОН);
- различные форматы команд с разной длиной;
- преобладание двухадресной адресации;
- развитый механизм адресации operandов, включающий различные методы косвенной адресации.

32. Архитектура RISC: принципы построения, преимущества, недостатки, признаки

В большинстве современных процессоров, относимых к классу CISC, сложные команды на стадии декодирования сводятся к набору простых RISC-команд, а ядро процессора реализуется как RISC-процессор.

- + Сравнительно простая структура устройства управления. Площадь, выделяемая на кристалле микросхемы для реализации УУ, существенно меньше. Как следствие, появляется возможность разместить на кристалле большое число регистров ЦП. Кроме того, остается больше места для других узлов ЦП и для дополнительных устройств
- + Простое устройство управления имеет немного элементов и, следовательно, короткие линии связи для прохождения сигналов управления. Малое число команд, форматов и режимов приводит к упрощению схемы декодирования, и оно происходит быстрее. Высокой производительности способствует и упрощение передачи параметров между процедурами.
- На выполнение ряда функций приходится тратить несколько команд вместо одной в CISC. Это удлиняет код программы, увеличивает загрузку памяти и трафик команд между памятью и ЦП.
- Хотя большое число регистров дает существенные преимущества, само по себе оно усложняет схему декодирования номера регистра, тем самым увеличивается время доступа к регистрам.
- УУ с аппаратной логикой, реализованное в большинстве RISC-систем, менее гибко, более склонно к ошибкам, затрудняет поиск и исправление ошибок, уступает при выполнении сложных команд
- Однословная команда исключает прямую адресацию для полноразрядного адреса, поэтому ряд производителей допускают небольшую часть команд двойной длины

Концепцию RISC-процессора можно свести к следующим положениям:

- выполнение всех (или, по крайней мере, 75% команд) за один цикл;
- стандартная однословная длина всех команд, равная естественной длине слова и ширине шины данных и допускающая унифицированную конвейерную обработку всех команд;
- малое число команд (не более 128);

- малое количество форматов команд (не более 4);
- малое число способов адресации (не более 4);
- доступ к памяти только посредством команд «Чтение» и «Запись»;
- все команды, за исключением «Чтения» и «Записи», используют внутрипроцессорные межрегистровые пересылки;
- устройство управления с аппаратной логикой;
- преобладание трехадресных команд;
- относительно большой (не менее 32) процессорный файл регистров общего назначения (число РОН в современных RISC-микропроцессорах может превышать 500).

33. Архитектура VLIW: принципы построения, преимущества, недостатки

Идея VLIW (Very Long Instruction Word) базируется на том, что задача эффективного планирования параллельного выполнения команд возлагается на «разумный» компилятор. Такой компилятор вначале анализирует исходную программу. Цель анализа: обнаружить все команды, которые могут быть выполнены одновременно, причем так, чтобы между командами не возникали конфликты. В ходе анализа компилятор может даже частично имитировать выполнение рассматриваемой программы. На следующем этапе компилятор пытается объединить такие команды в пакеты (связки), каждый из которых рассматривается как одна сверхдлинная команда.

- + VLIW-архитектуру можно рассматривать как статическую суперскалярную архитектуру. Имеется в виду, что распараллеливание кода производится на этапе компиляции, а не динамически во время исполнения. То, что в выполняемой сверхдлинной команде исключена возможность конфликтов, позволяет предельно упростить аппаратуру VLIW-процессора и, как следствие, добиться более высокого быстродействия
- Усложнение регистрового файла и, прежде всего, связей этого файла с вычислительными устройствами
- Трудности создания компиляторов, способных найти в программе независимые команды, связать такие команды в длинные строки и обеспечить их параллельное выполнение

34. Группы команд: классификация и основные характеристики

Группы команд:

1. Пересылки данных и ввода-вывода

- внутри процессора между регистрами

Обеспечивают передачу информации между регистрами внутри процессора

- взаимодействие с памятью

Обеспечивают передачу информации между процессором и памятью или между различными уровнями памяти (СОЗУ ↔ ОЗУ).

- организация ввода-вывода

Обеспечивают передачу информации между процессором и внешними устройствами.

2. Арифметических и логических операций

- целочисленная арифметика

Арифметические операции с целыми числами, представленными в форме чисел с фиксированной точкой (сложение, вычитание, умножение, деление, модуль, изменение знака, сравнение)

- арифметика с плавающей точкой

Арифметические операции с числами, представленными в форме чисел с плавающей точкой (сложение, вычитание, умножение, деление, сравнение, изменение формы и формата представления)

- двоично-десятичная арифметика

Десятичные числа представляются в ВМ в двоично-кодированной форме. В вычислительных машинах первых поколений для обработки таких чисел предусматривались специальные команды, обеспечивающие выполнение основных арифметических операций (сложение, вычитание, умножение и деление). В АСК современных машин подобных команд обычно нет, а соответствующие вычисления имитируются с помощью команд целочисленной арифметики с последующей коррекцией полученного результата.

- логические операции

Команды для выполнения различных логических операций над отдельными битами слов или других адресуемых единиц. Такие команды предназначены для обработки символьных и логических данных. Минимальный набор поддерживаемых логических операций — это «НЕ», «И», «ИЛИ» и «сложение по модулю 2»

3. Управления

- передача управления

Команды, которые изменяют естественный порядок выполнения команд программы. Эти команды меняют содержимое программного счетчика, обеспечивая переходы по программе. Существуют команды безусловной и условной передачи управления. В последнем случае передача управления происходит, если выполняется заданное в коде команды условие, иначе выполняется следующая по порядку команда.

- управление режимами работы

Системные команды. К примеру, переключение между реальным и защищенным режимом.

35. Форматы команд: назначение полей

Формат команды определяет ее структуру, то есть количество двоичных разрядов, отводимых под всю команду, а также количество и расположение отдельных полей команды. Полем называется совокупность двоичных разрядов, кодирующих составную часть команды.

В различных архитектурах ЭВМ используются различные форматы команд, но в общем случае команда состоит из операционной и адресной частей.

В операционной части содержится поле КОП – код выполняемой операции.

В адресной части могут содержаться следующие поля (зависит от архитектуры):

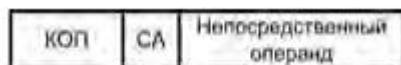
- A1 – адрес первого операнда
- A2 – адрес второго операнда
- A3 – адрес результата
- A4 – адрес следующей команды

36. Режимы адресации: классификация и основные характеристики

- Неявная

Поле адреса в команде отсутствует, а адрес операнда или не имеет смысла для данной команды, или подразумевается по умолчанию.

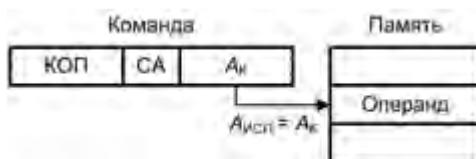
- Непосредственная



При непосредственной адресации (НА) в адресном поле команды вместо адреса содержится непосредственно сам операнд. Этот способ может применяться при выполнении арифметических операций, операций сравнения, а также для загрузки констант в регистры.

При записи в регистр, имеющий разрядность, превышающую длину непосредственного операнда, операнд размещается в младшей части регистра, а оставшиеся свободными позиции заполняются значением знакового бита операнда. Помимо того, что в адресном поле могут быть указаны только константы, еще одним недостатком данного способа адресации является то, что размер непосредственного операнда ограничен длиной адресного поля команды, которое в большинстве случаев меньше длины машинного слова.

- Прямая



При прямой или абсолютной адресации (ПА) адресный код прямо указывает на номер ячейки памяти, к которой производится обращение, то есть адресный код совпадает с исполнительным адресом.

При всей простоте использования способ имеет существенный недостаток — ограниченный размер адресного пространства, так как для обращения к памяти большой емкости нужно «длинное» адресное поле. Однако более существенным недовершением можно считать то, что адрес, указанный в команде, не может быть изменен в процессе вычислений (во всяком случае, такое изменение не рекомендуется). Это ограничивает возможности по произвольному размещению программы (и данных) в памяти

- Косвенная (косвенная регистровая)



При косвенной адресации содержимое адресного поля команды остается неизменным, в то время как косвенный адрес в процессе выполнения программы можно изменять. Это позволяет проводить вычисления, когда адреса операндов заранее неизвестны и появляются лишь в процессе решения задачи. Дополнительно такой прием упрощает обработку массивов и списков, а также передачу параметров подпрограммам.

Недостатком косвенной адресации является необходимость в двукратном обращении к памяти: сначала для извлечения адреса операнда, а затем для обращения к операнду.



Косвенная регистровая адресация (КРА) представляет собой косвенную адресацию, где исполнительный адрес операнда хранится не в ячейке основной памяти, а в регистре процессора.

- Относительная (базирование)



Адрес формируется как сумма двух слагаемых: базы, хранящейся в специальном регистре или в одном из РОН, и смещения, извлекаемого из поля адреса команды.

- Базово-индексная

Базово-индексная адресация формирует адрес операнда как сумму трех слагаемых: базы, индекса и смещения. При каждом обращении содержимое индексного регистра автоматически модифицируется (обычно увеличивается или уменьшается на 1). // X₃, у меня в лекциях записано, что тут индекс вместо смещения

37. Направления повышения производительности вычислений на уровне архитектуры системы команд: SIMD-расширения

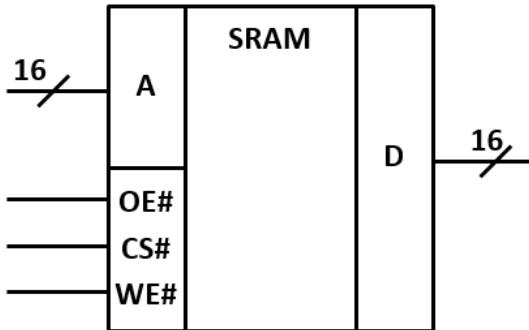
Название данного типа команд представляет собой аббревиатуру от Single Instruction Multiple Data — буквально «одна команда — много данных». В отличие от обычных команд, оперирующих двумя числами, SIMD-команды обрабатывают сразу две группы чисел (в принципе, их можно называть групповыми командами). Операнды таких команд обычно представлены в одном из упакованных форматов.

Поводом послужило широкое распространение мультимедийных приложений. Видео, трехмерная графика и звук в ВМ представляются большими массивами данных, элементы которых чаще всего обрабатываются идентично. Включение SIMD-команд в АСК позволяет существенно ускорить подобные вычисления.

Первой на мультимедийный бум отреагировала фирма Intel, добавив в систему команд своего микропроцессора Pentium MMX 57 SIMD-команд. Название MMX (MultiMedia eXtention — мультимедийное расширение) разработчики обосновывали тем, что при выборе состава новых команд были проанализированы алгоритмы, применяемые в различных мультимедийных приложениях. Команды MMX обеспечивали параллельную обработку упакованных целых чисел. При выполнении арифметических операций каждое из чисел, входящих в группу, рассматривается как самостоятельное, без связи с соседними числами.

В дальнейшем наборы SIMD-команд стали обозначать аббревиатурой SSE — Streaming SIMD Extension (потоковая обработка по принципу «одна команда — много данных»). В последних разработках фирм Intel и AMD технология обозначена как SSE4, при этом у каждой из фирм она имеет свои особенности. Так, в микропроцессорах Penryn фирмы Intel реализован набор из 47 SIMD-команд (SSE4.1), в микропроцессорах Nehalem (Core i7) той же фирмы — дополнительно еще 7 команд (SSE4.2). Микропроцессоры Phenom фирмы AMD в настоящее время поддерживают лишь 4 команды из SSE4, но дополнительно реализуют две новые команды, обозначаемые как SSE4a. В 2007 году фирма AMD анонсировала новый стандарт — SSE5, преподносимый как 128-разрядное расширение SSE. Предлагается набор из 170 команд, который предполагается реализовать на процессорах Bulldozer, производство которых запланировано на 2011 год.

38. Статические БИС ЗУ: принципы построения, основные особенности и области применения



Режим	Вывод		Сигнал		
	A	D	CS#	OE#	WE#
Хранение	-	Z	1	-	-
Запись	A	DI	0	1	0
Чтение	A	DO	0	0	1

A – шина адреса, OE – сигнал: output enable, WE – сигнал: write enable, CS – сигнал: chip selected.

Z – состояние высокого импеданса.

SRAM в качестве элементарной ячейки использует так называемый статический триггер (схема которого состоит из нескольких транзисторов (4-6)). Схема триггера на четырех транзисторах проще, обладает меньшей стоимостью, но у нее больший ток утечки и она более чувствительна к воздействию внешних источников излучения. Два дополнительных транзистора служат не только для уменьшения перечисленных недостатков, но и повышают быстродействие такой схемы. Статическая память по способу доступа к данным может быть, как асинхронной, так и синхронной.

Асинхронным называется доступ к данным, который можно осуществлять в произвольный момент времени. Асинхронная SRAM применялась на материнских платах для третьего — пятого поколений процессоров.

Синхронная память обеспечивает доступ к данным не в произвольные моменты времени, а одновременно (синхронно) с тактовыми импульсами. В промежутках между ними память может готовить для доступа следующую порцию данных. В большинстве материнских плат пятого поколения используется разновидность синхронной памяти.

Статический тип памяти обладает более высоким быстродействием, а также для него не требуется регенерация.

Применяется в устройствах с небольшим объемом ОЗУ, где требуется низкое электропотребление, а также для регистров и кэш-памяти.

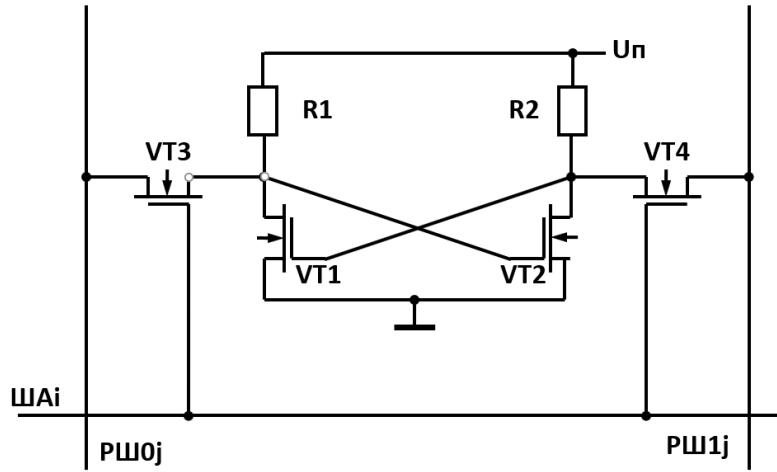
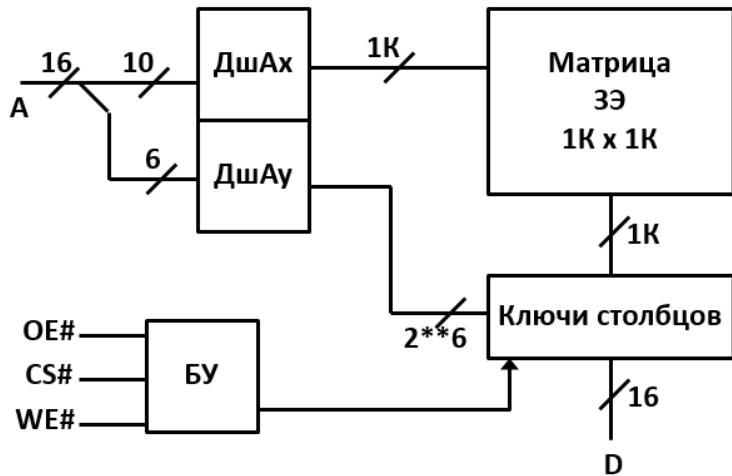


Схема простейшего ЗУ SRAM (одна ячейка)



Структура статической БИС ЗУ.

Число разрядов адреса - 16

Матрица ЗЭ

ДшAx – дешифратор адреса строк

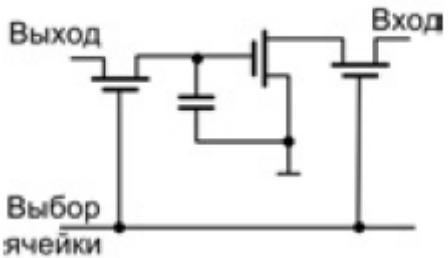
ДшAy – дешифратор адреса столбцов

Блок ключей, связывающих внутренние блоки с выводом данных

БУ – блок управления с сигналами: OE – сигнал: output enable, WE – сигнал: write enable, CS – сигнал: chip selected.

39. Динамические БИС ЗУ: принципы построения, основные особенности и области применения

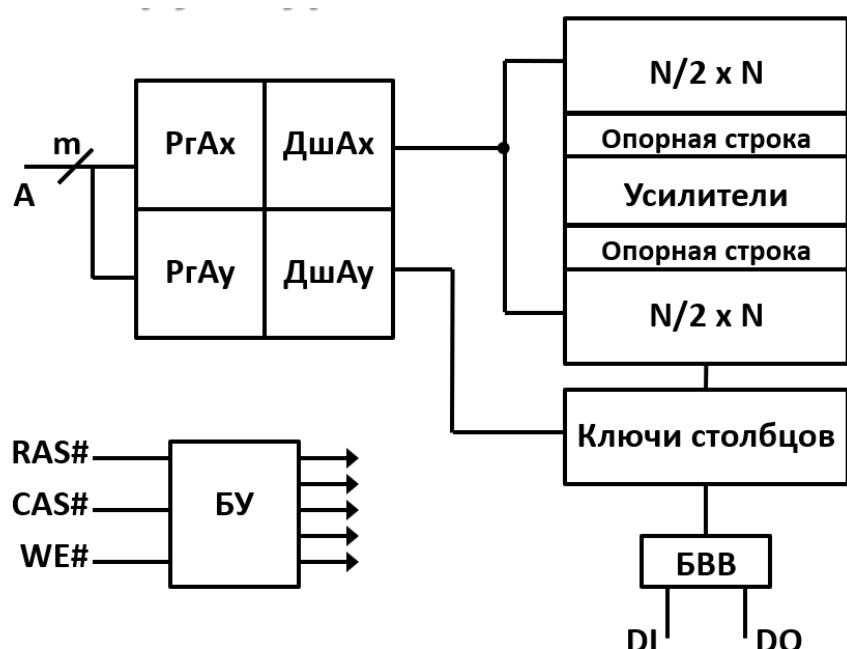
Каждый бит памяти DRAM представляется в виде наличия (или отсутствия) заряда на конденсаторе, образованном в структуре полупроводникового кристалла. Конденсатор управляет транзистором. Если транзистор открыт и ток идет, это означает «1», если закрыт — «0». С течением времени конденсатор разряжается, и его заряд нужно периодически восстанавливать. Между периодами доступа к памяти посыпается электрический ток, обновляющий заряд на конденсаторах для поддержания целостности данных (вот почему данный тип памяти называется динамическим ОЗУ). Этот процесс называется регенерацией памяти.



Микросхема памяти этого типа представляет собой прямоугольный массив ячеек со вспомогательными логическими схемами, которые используются для чтения или записи данных, а также цепей регенерации, поддерживающих целостность данных. Массивы памяти организованы в строки (row) и столбцы (column) ячеек памяти, именуемые соответственно линиями слов (wordlines) и линиями бит (bitlines). Каждая ячейка памяти имеет уникальное размещение, задаваемое пересечением строки и столбца. Цепи, поддерживающие работу памяти, включают:

- усилители, считающие сигнал, обнаруженный в ячейке памяти;
- схемы адресации для выбора строк и столбцов;
- схемы выбора адреса строки (Row address select — /RAS) и столбца (Column address select — /CAS), чтобы открывать и закрывать адреса строк и столбцов, а также начинать и заканчивать операции чтения и записи;
- цепи записи и чтения информации;
- внутренние счетчики или регистры, следящие за циклами регенерации данных;
- схемы разрешения вывода (Output enable — OE).

Динамическая память (DRAM) в современных ПК используется обычно в качестве оперативной памяти общего назначения, а также как память для видеoadаптера.



Структура DRAM

Матрица ЗЭ – $N = 2^{2m}$ (адресные шины) строк и N (разрядные шины) столбцов. На каждом пересечении шин расположен ЗЭ. Для выбора ЗЭ в матрице необходимо выбрать строку и столбец. Все адресные шины разделены пополам. Между половинами каждой разрядной шины включен усилитель воспроизведения (записи) Регистры адреса PgAx и PgAy служат для запоминания, а дешифраторы DsAx и DsAy – для дешифрации кода адреса строки и столбца соответственно. Ключи столбцов осуществляют коммутацию сигналов между разрядными шинами матрицы ЗЭ и шиной ввода-вывода.

Блок ввода-вывода осуществляет коммутацию сигналов между шиной ввода (вывода) и внешними выводами СБИС (DI – ввод одного бита при записи, DO – вывод считанного бита)

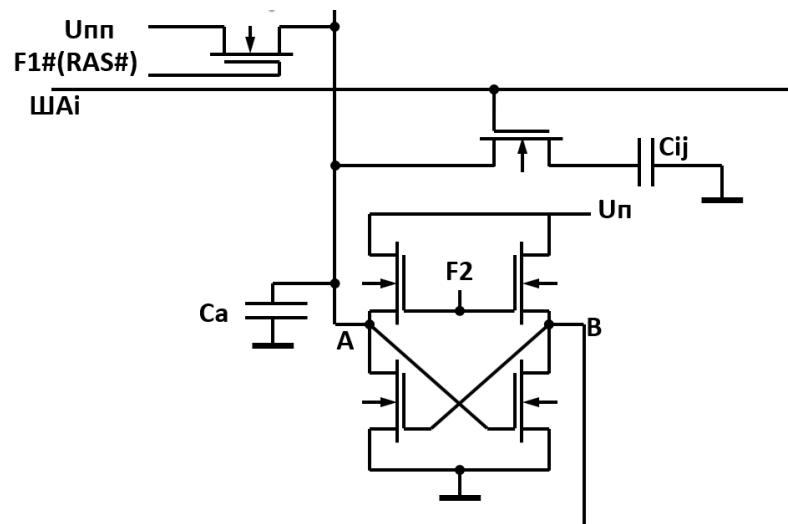
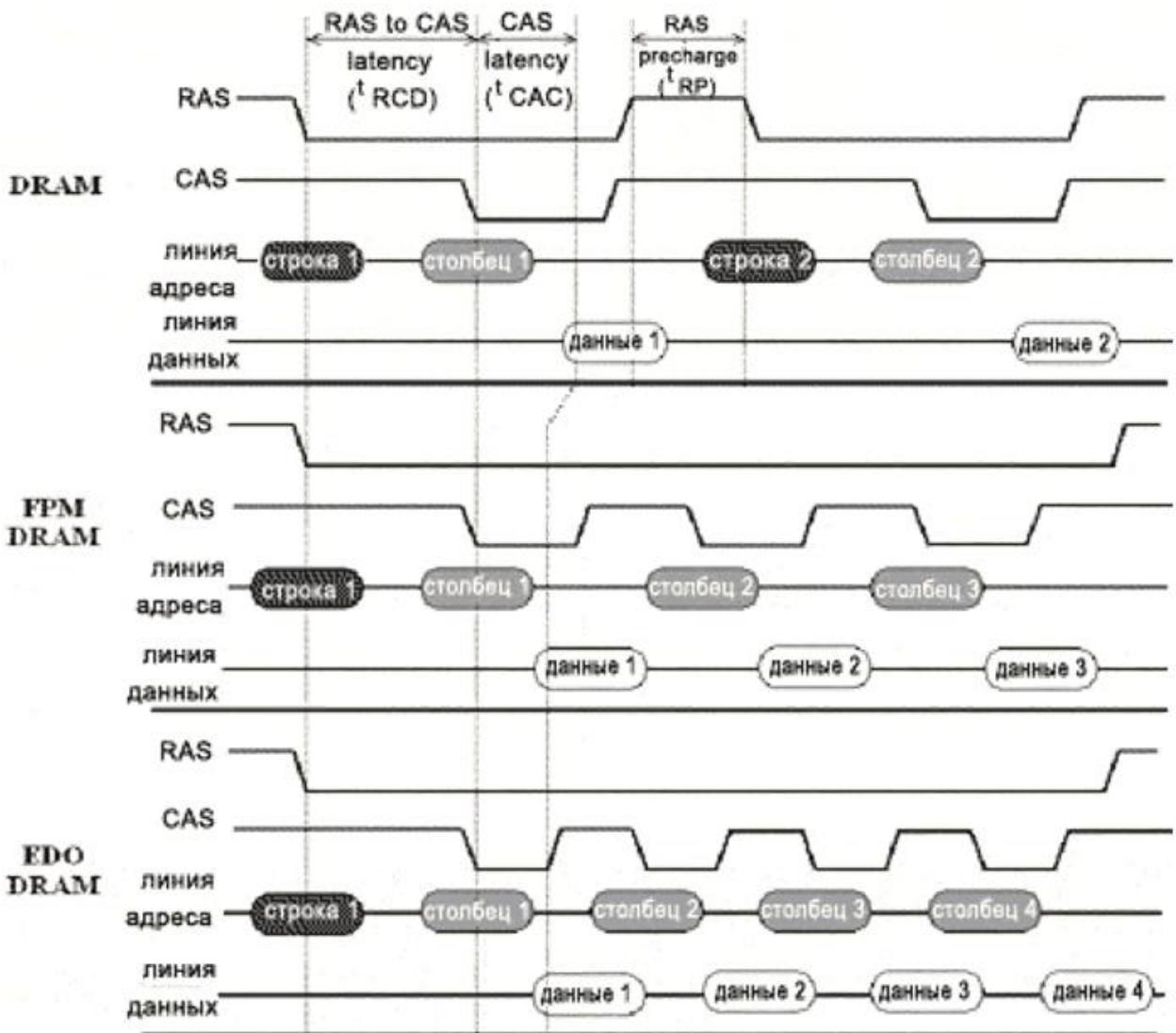


Схема ЗУ DRAM

40. Повышение производительности устройств асинхронной динамической памяти.



DRAM -> FPM DRAM

Из-за того, что данные обычно считаются группами (байтами), нет смысла сразу снимать сигнал RAS.

Система управления памятью в процессе считывания активирует адреса строк, столбцов, осуществляет проверку данных и передачу информации в систему. Столбцы после этого деактивируются, что приводит к нежелательному состоянию ожидания процессора в некоторых сочетаниях операций с памятью.

FPM DRAM -> EDO DRAM

Обращение на чтение осуществляется таким же образом, как и в FPM, за исключением того, что высокий уровень /CAS не сбрасывает выходные данные, а использование триггера позволяет сохранять данные до тех пор, пока уровень CAS снова не станет низким. Тем самым не происходит сброса адреса столбцов перед началом следующей операции с памятью.

По-другому – работа EDO DRAM начинается с активизации строки и столбца. Однако после нахождения элемента данных в памяти этого типа буфер данных остается включенным до обращения к следующему столбцу, т.е. устраняется состояние ожидания.

EDO -> BEDO

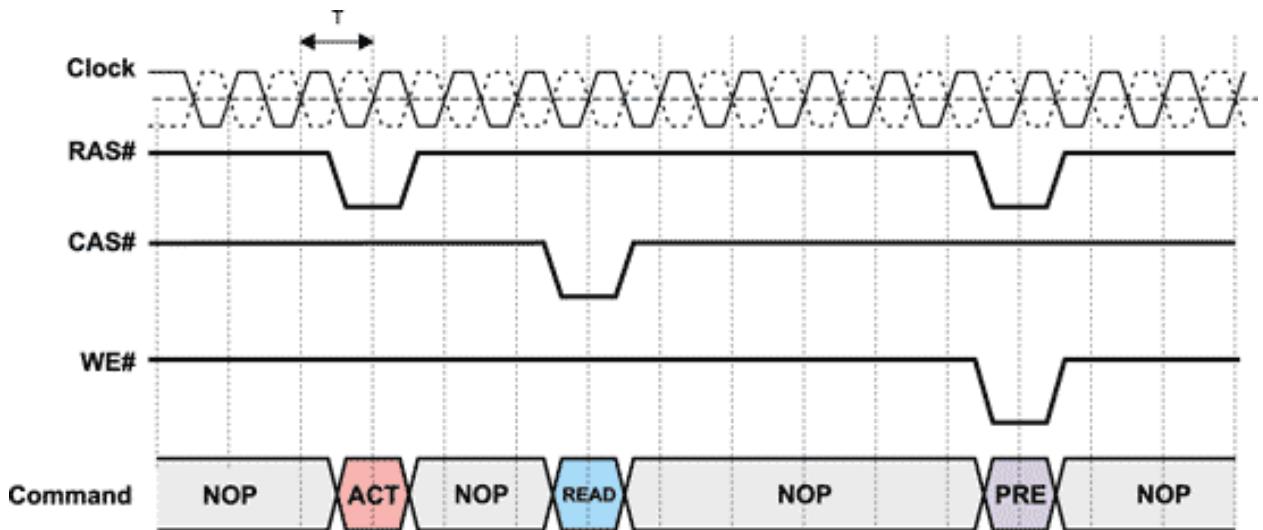
Burst – метод последовательного выбора столбцов. Использование счетчика для каждой строки.

41. Повышение производительности устройств синхронной динамической памяти

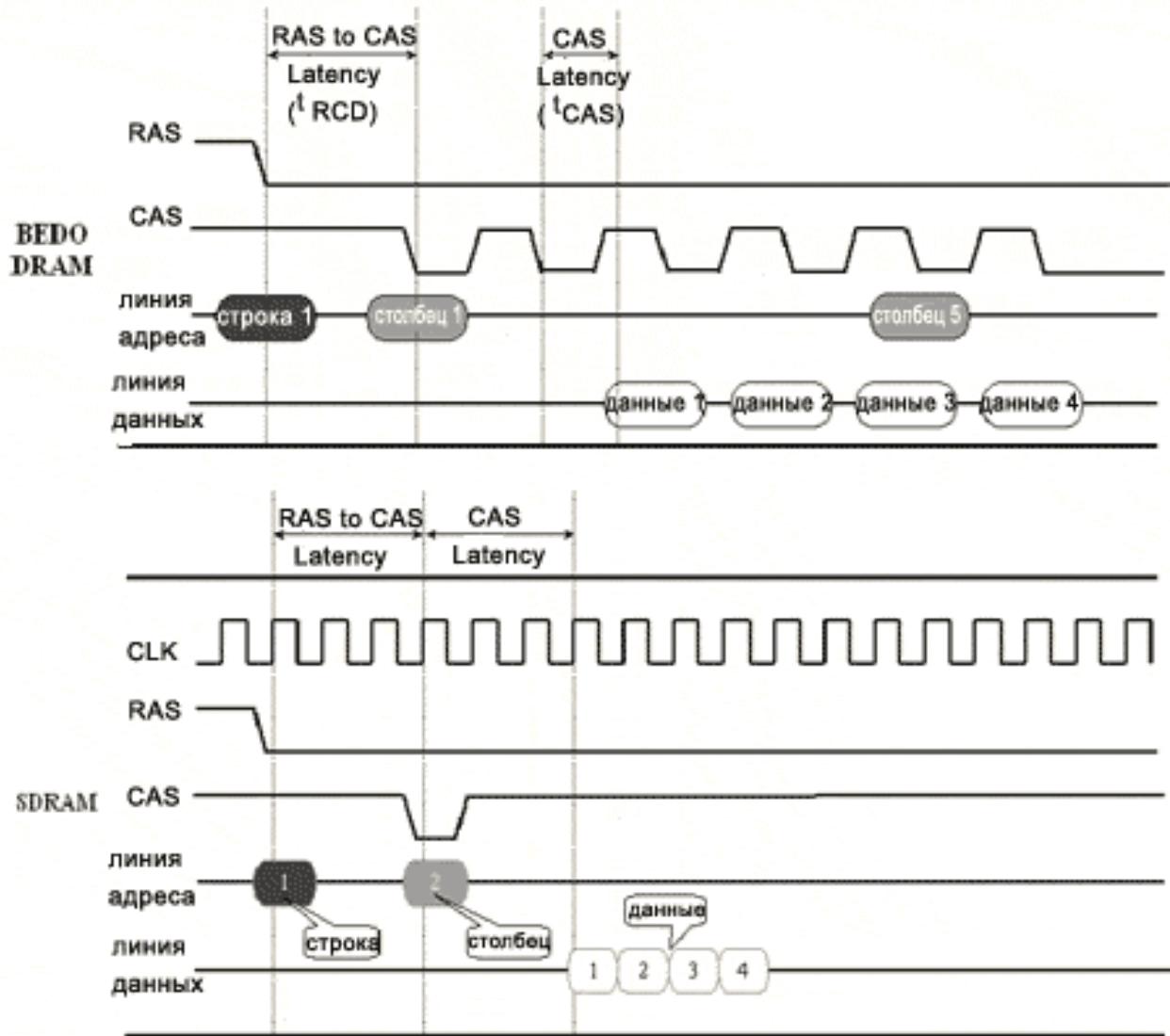
Самой быстрой представляется синхронная динамическая память (SDRAM). Основная особенность ее состоит в способности синхронизировать все операции с тактовыми сигналами процессора. Это уменьшает время обращения к столбцу памяти. Внутри памяти SDRAM находится счетчик, который увеличивается при адресации столбца, что инициирует новое обращение к памяти до завершения предыдущего. Все это позволяет затрачивать пять тактов на чтение первого слова и по одному такту на чтение последующих слов, т. е. всего восемь тактов.

В последнее время появилась синхронная динамическая память с удвоенной скоростью передачи данных DDR SDRAM. В этой динамической памяти данные в пакетном режиме выдаются по обоим фронтам импульса синхронизации, за счет чего ее пропускная способность увеличивается вдвое. Этот тип динамической памяти в настоящее время является наиболее распространенным для персональных компьютеров.

Временные диаграммы SDRAM



Переход от асинхронной памяти к синхронному режиму взаимодействия



42. Расслоение памяти

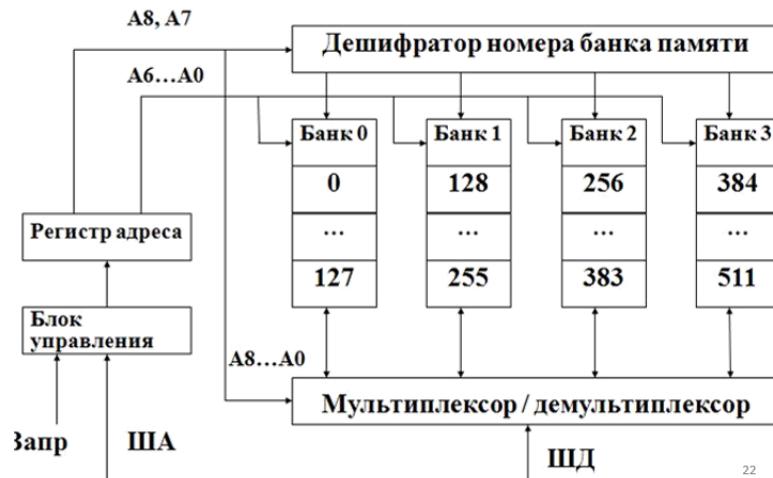
Емкость основной памяти современных ВМ слишком велика, чтобы ее можно было реализовать на базе единственной интегральной микросхемы (ИМС). Необходимость объединения нескольких ИМС ЗУ возникает также, когда разрядность ячеек в микросхеме ЗУ меньше разрядности слов ВМ. Увеличение разрядности ЗУ реализуется за счет объединения адресных входов, объединяемых ИМС ЗУ. Информационные входы и выходы микросхем являются входами и выходами модуля ЗУ увеличенной разрядности. Полученную совокупность микросхем называют модулем памяти. Модулем можно считать и единственную микросхему, если она уже имеет нужную разрядность. Один или несколько модулей образуют банк памяти.

Для получения требуемой емкости ЗУ нужно определенным образом объединить несколько банков памяти меньшей емкости. В общем случае основная память ВМ практически всегда имеет блочную структуру, то есть содержит несколько банков. При использовании блочной памяти, состоящей из B банков, адрес ячейки A преобразуется в

пару (b, w) , где b — номер банка, w — адрес ячейки внутри банка. Известны три схемы распределения разрядов адреса А между b и w :

- блочная (номер банка b определяет старшие разряды адреса);
- циклическая ($b = A \bmod B$; $w = A \div B$);
- блочно-циклическая (комбинация двух предыдущих схем).

Рассмотрение основных структур блочной ОП будем проводить на примере памяти емкостью 512 слов (29), построенной из четырех банков по 128 слов.



22

Адресное пространство памяти разбито на группы последовательных адресов, и каждая такая группа обеспечивается отдельным банком памяти. Для обращения к ОП используется 9-разрядный адрес, семь младших разрядов которого ($A_6\text{--}A_0$) поступают параллельно на все банки памяти и выбирают в каждом из них одну ячейку.

Два старших разряда адреса (A_8, A_7) содержат номер банка. Выбор банка обеспечивается либо с помощью дешифратора номера банка памяти, либо путем мультиплексирования информации (на рис. 6.3 показаны оба варианта). В функциональном отношении такая ОП может рассматриваться как единое ЗУ, емкость которого равна суммарной емкости составляющих, а быстродействие — быстродействию отдельного банка.

Если память строится в виде нескольких модулей с автономными схемами адресации, записи и чтения, то можно организовать расслоение памяти. Под *расслоением* понимают такую организацию ОП, когда ее выполняют из нескольких автономных модулей, в которые информация заносится по определенным правилам.

Известно несколько способов организации расслоения. В персональных компьютерах для этих целей чаще всего используется разделение памяти на память данных и память программ. Такое расслоение применяется при организации внутренней кэш-памяти практически всех современных микропроцессоров. Сегодня в каждом микропроцессоре имеется внутренняя кэш-память объемом от 8 до 64 Кбайт и более, разделенная на два блока — память команд и память данных. При выполнении любой команды требуется несколько обращений в память как за самой командой, так и операндами; эти обращения производятся в разные блоки внутренней памяти.

В случае классического расслоения памяти ее строят из нескольких модулей; ячейки с последовательными адресами находятся в различных модулях. Благодаря свойству локальности программ (и данных) вслед за текущей командой вероятнее всего следующей будет выполняться команда, адрес которой на единицу больше адреса текущей команды, т. е. команда, хранящаяся в следующем модуле памяти. Такой порядок нарушается только командами переходов. Данные также записываются в память и используются программой последовательно, т. е. из разных модулей. Такое расслоение возможно только при постоянном формате команд.

Расслоение памяти чаще всего организуют в соответствии с младшими разрядами. Адрес включает в себя две составляющие (рис. 6.5, а):

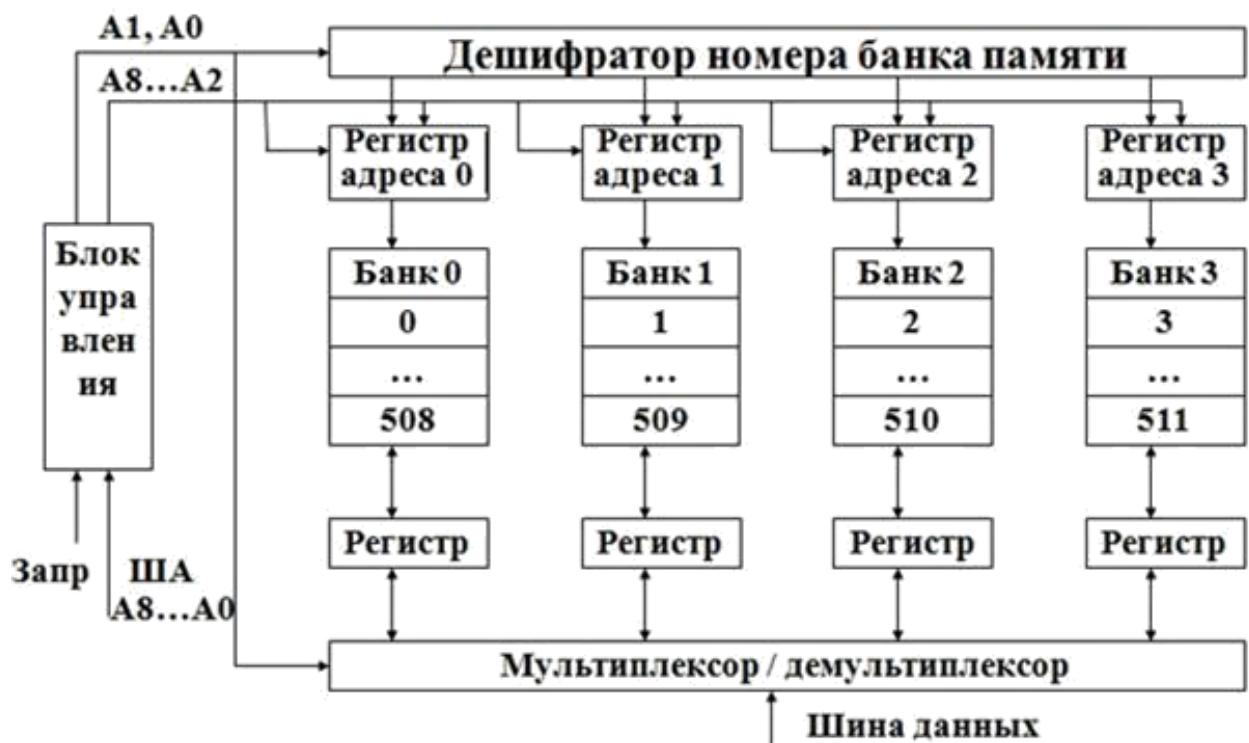
$$A = A_{ст}, A_{мл},$$

где $A_{ст}$, $A_{мл}$ — старшие и младшие разряды соответственно.

Все программы и данные располагаются в памяти последовательно. Однако ячейки со смежными адресами находятся в разных физических блоках.

Предположим, что в машине используются четыре модуля памяти, а номер модуля соответствует содержимому двух младших разрядов адреса $A_{мл}$. Тогда все обращения к ячейкам памяти с последовательными адресами будут приходить на разные модули, а поскольку все эти модули обладают собственными схемами адресации и выборки, то можно производить обращение к следующему модулю, не дожидаясь ответа от предыдущего.

Расслоение памяти. Циклическая схема чередования адресов



Расслоение памяти. Блочно-циклическая схема

Банк 0		Банк 1		Банк 2		Банк 3	
Модуль 0	Модуль 1	Модуль 2	Модуль 3	Модуль 4	Модуль 5	Модуль 6	Модуль 7
0	1	128	129	256	257	384	385
2	3	130	131	258	259	386	386
...
126	127	254	255	382	383	510	511

В блочно-циклической схеме расслоения памяти каждый банк состоит из нескольких модулей, адресуемых по круговой схеме. Адреса между банками распределены по блочной схеме. Таким образом, адрес ячейки разбивается на три части. Старшие биты определяют номер банка, следующая группа разрядов адреса указывает на ячейку в модуле, а младшие биты адреса выбирают модуль в банке.

43. Принципы построения кэш-памяти с прямым отображением

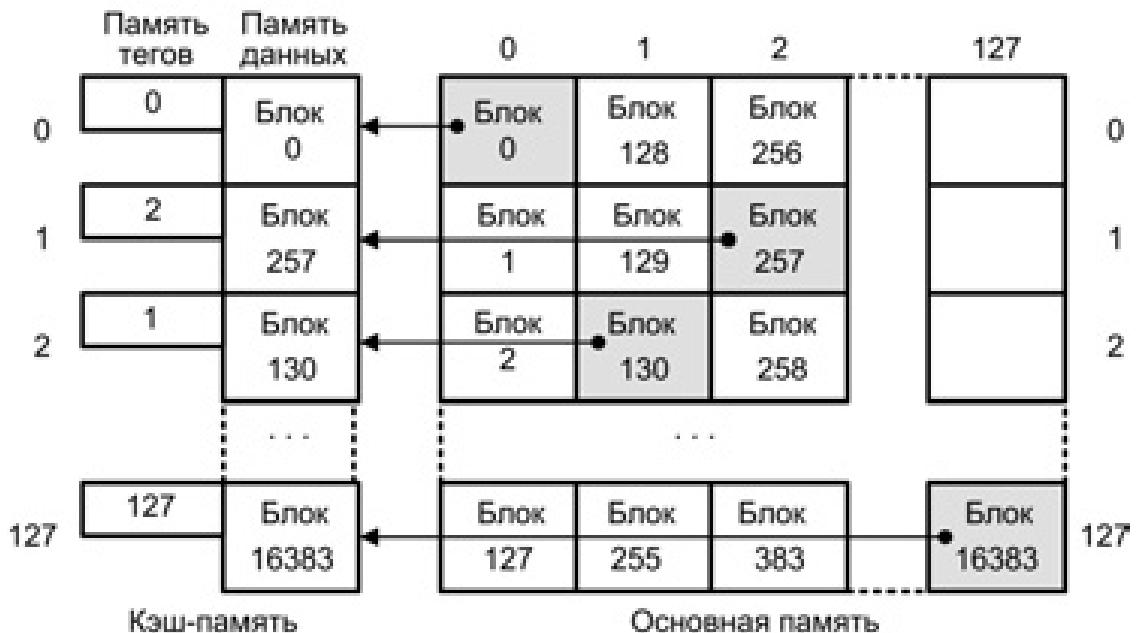
При прямом отображении множество блоков основной памяти условно представляется в виде матрицы, в которой количество строк равно числу блоков в кэш-памяти m (рисунок ниже). В блок кэш-памяти с номером i может быть помещен любой блок ОП, но только из i -й строки матрицы.

Номер строки (i) и столбца (k) матрицы, на пересечении которых располагается блок основной памяти с адресом j , определяются выражениями $i = j \bmod m$ и $k = j \div m$. В двоичной записи адреса блока основной памяти номер строки представлен $\log_2 m$ младшими разрядами, а номер столбца — оставшимися старшими разрядами. Такая система позволяет по адресу блока основной памяти, поступившему из ЦП, сразу же однозначно определить адрес блока кэш-памяти, куда должен быть отображен данный блок ОП (если его копия в КП отсутствует) или где следует искать соответствующую копию (если она присутствует в КП).

В качестве указателя того, какой из блоков i -й строки отображен или должен быть отображен на i -й блок КП (тега), используется номер столбца матрицы, где расположен отображенный (отображаемый) блок ОП.

Применимельно к системе, предложенной в качестве примера, это означает, что 14-разрядный адрес блока основной памяти может рассматриваться как состоящий из 7-разрядного номера столбца (тега) и 7-разрядного номера строки. При выставлении процессором адреса ячейки основной памяти сначала проверяется наличие в кэш-памяти копии блока, содержащего эту ячейку. Для этого из исполнительного адреса ячейки выделяется адрес блока ОП, к которому относится данная ячейка (обозначим его j). Младшие разряды j указывают на тот единственный блок кэш-памяти i , где может находиться копия, и одновременно на соответствующую ячейку памяти тегов. Тег сравнивается с 7 старшими разрядами поступившего из ЦП адреса. Совпадение означает,

что в кэш-памяти находится копия затребованного блока ОП, и процессор адресуется к кэш-памяти. Несовпадение порождает копирование затребованного блока в кэш-память с одновременной записью в память тегов номера колонки, из которой был скопирован блок.



Прямое отображение — простой и недорогой в реализации способ отображения. К его достоинствам можно отнести малую разрядность тега (для нашего примера она равна 7 битам). Кроме того, при проверке наличия в КП нужной копии достаточно проверить лишь одну определенную ячейку памяти тегов. Основной его недостаток — жесткое закрепление за определенными блоками ОП одного блока в КП. Поэтому если программа поочередно обращается к словам из двух различных блоков, отображаемых на одну и тут же строку кэш-памяти, постоянно станет происходить обновление данной строки и вероятность попадания будет низкой.

Вкратце:

Оперативную память условно разбивают на блоки. Допустим 14 разрядов полный адрес. Тогда считаем, что 7 разрядов для определения блока (старшая часть) и 7 разрядов для строки (младшая часть). Кэш-память делится на строки. Каждая строка кэша может отображать из оперативки только соответствующую ей строку. Поскольку объем основной памяти много больше объема кэша, на каждую строку кэша может претендовать множество блоков памяти с одинаковой младшей частью адреса (строки), но из разных блоков. Информация о том, какой именно блок оперативки занимает данную строку кэша, называется тегом (tag). Номер (адрес) строки в кэш-памяти называется индексом (index)

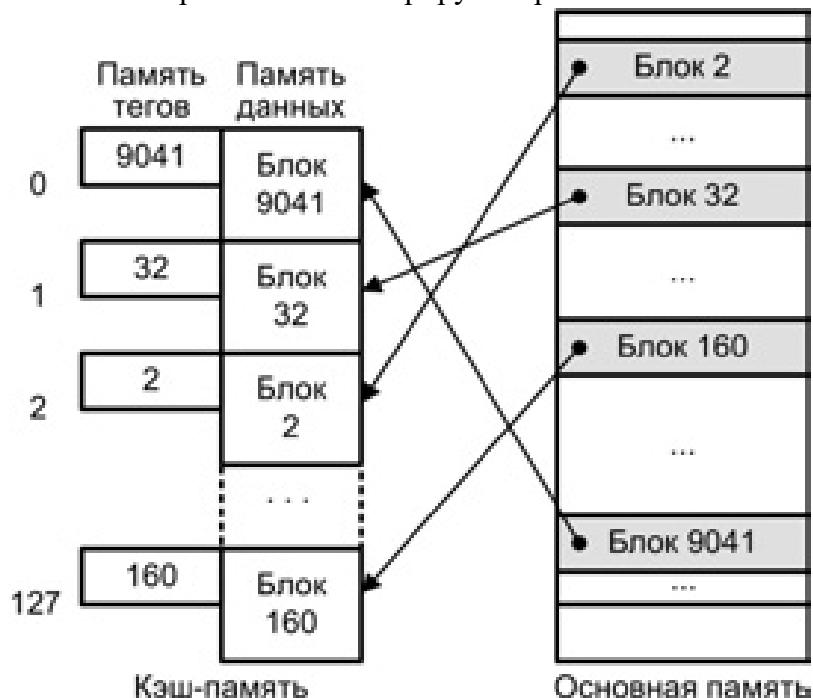
- Проблема в том, что слишком часто происходит замещения, поскольку часто происходит обращение к соседним ячейкам памяти. Нельзя выбрать оптимальный алгоритм замещения, следовательно, замещения нужных данных может происходить постоянно.
- + Зато при поиске нужной ячейки размер для сравнения всего лишь 7 разрядов, вместо 14(после нахождения строки уже проверяем тэг, то есть еще 7 разрядов)
- + самая простая аппаратная реализация

Общий принцип любого типа кэша. При кэш-попадании (запрашиваемая ячейка есть в кэше) строка считывается в шину данных (ШД). При кэш-промахе обращение происходит к ОЗУ (медленно), и при этом одновременно заполняется очередная КЭШ-строка.

44. Принципы построения полностью ассоциативной кэш-памяти

Полностью ассоциативное отображение позволяет преодолеть недостаток прямого, разрешая загрузку любого блока ОП в любой блок кэш-памяти. В этих условиях тегом служит полный адрес блока в основной памяти, то есть для нашего примера длина тега составляет 14 битов. Для проверки наличия копии блока в кэш-памяти необходимо сравнить теги всех блоков кэш-памяти на совпадение со старшими разрядами адреса, поступившего из ЦП. Такое сравнение желательно выполнять одновременно для всех ячеек памяти тегов. Этому требованию наилучшим образом отвечает ассоциативная память, то есть память тегов должна быть ассоциативной.

Логика ассоциативного отображения иллюстрируется рис. ниже.



Полностью ассоциативное отображение обеспечивает гибкость при выборе строки для вновь записываемого блока. Принципиальный недостаток этого способа - необходимость использования дорогостоящей ассоциативной памяти, причем разрядность ячейки этой памяти достаточно велика (в примере — 14 битов против 7 битов в случае прямого отображения).

- + самая быстрая, поскольку любая строки оперативки может храниться в любой строке кэша, что минимизирует замещения, возможно еще полезных данных
- сложность аппаратной реализации.

Поэтому полностью ассоциативная кэш-память чаще всего используется в специализированных буферах, таких, как буфер целевых адресов переходов, с небольшим объемом строк.

45. Принципы построения частично ассоциативной кэш-памяти

Частично-ассоциативное отображение, известное также как множественно-ассоциативное (set-associative), является одним из возможных компромиссов, сочетающим достоинства прямого и полностью ассоциативного способов отображения. В определенной мере, оно свободно от их недостатков. Кэш-память разбивается на v подмножеств (в дальнейшем будем называть такие подмножества модулями), каждое из которых содержит k блоков.

Иными словами, память тегов и память данных условно представляется матрицами, имеющими v строк и k столбцов. Кэш-память с такой организацией принято называть k -канальной (k -way) или кэш-памятью со степенью ассоциативности k . Основная память также рассматривается как матрица с тем же числом строк v . Зависимость между отдельным модулем и блоками ОП такая же, как и при прямом отображении: на блоки, входящие в модуль i , могут быть отображены только вполне определенные блоки основной памяти, в соответствии с соотношением $i = j \bmod v$, где j — адрес блока ОП. Это означает, что блоки из i -й строки матрицы блоков ОП могут отображаться лишь на блоки из i -й строки матрицы блоков кэш-памяти, причем на любой из этих блоков. Так как копия блока ОП может находиться в любом из блоков модуля кэш-памяти, для ускорения поиска нужного блока в пределах модуля используется ассоциативный принцип.

На рис. ниже показан пример четырехканальной кэш-памяти с частично-ассоциативным отображением. Память данных кэш-памяти разбита на 32 модуля по 4 блока в каждом.

Память тегов содержит 32 ячейки, в каждой из которых может храниться 4 значения тегов (по одному на каждый блок модуля). 14-разрядный адрес блока ОП представляется в виде двух полей: 9-разрядного поля тега и 5-разрядного поля номера модуля. Номер модуля однозначно указывает на один из модулей кэш-памяти. Он также позволяет определить номера тех блоков ОП, которые можно отображать на этот модуль. Такими являются блоки ОП, номера которых при делении на 25 дают в остатке число, совпадающее с номером данного модуля кэш-памяти. Так, блоки 0, 32, 64, 96 и т. д. основной памяти отображаются на модуль с номером 0; блоки 1, 33, 65, 97 и т. д. отображаются на модуль 1 и т. д. Любой из блоков в последовательности может быть загружен в любой из четырех блоков соответствующего модуля.



При такой постановке роль тега выполняют 9 старших разрядов адреса блока ОП, в которых содержится порядковый номер блока в последовательности блоков, отображаемых на один и тот же модуль кэш-памяти. Например, блок 65 в последовательности блоков, отображаемых на модуль 1, имеет порядковый номер 2 (отсчет ведется от 0).

При обращении к кэш-памяти 5-разрядный номер модуля указывает на конкретную ячейку памяти тегов (это соответствует прямому отображению). Далее производится параллельное сравнение каждого из четырех тегов, хранящихся в этой ячейке, с полем тега поступившего адреса, то есть поиск нужного тега среди четырех возможных осуществляется ассоциативно.

В предельных случаях, когда $v = m$, $k = 1$, частично-ассоциативное отображение сводится к прямому, а при $v = 1$, $k = m$ — к полностью ассоциативному.

Наиболее общий вид организации частично-ассоциативного отображения — использование двух блоков на модуль ($v = m/2$, $k = 2$). Четырехканальная частично-

ассоциативная кэш-память ($v = m/4$, $k = 4$) дает дополнительное улучшение за сравнительно небольшую дополнительную цену. Дальнейшее увеличение числа блоков в модуле существенного эффекта не дает.

Следует отметить, что именно этот способ отображения наиболее широко распространен в современных микропроцессорах. Наряду с 2- и 4-канальной кэш-памятью первого уровня в последних моделях микропроцессоров применяется и 16-канальная организация, в частности, при построении кэш-памяти второго уровня.

Смесь рассмотренных выше принципов. Разделяем оперативку и кэш на блоки. Для выбора блока используется метод прямого распределения, а для выбора строки внутри блока метод полностью ассоциативного распределения. Tag отвечает за выбор блока, а индекс за выбор строки в блоке.

Например, имеем такую конфигурацию системы, при которой, в прямом отображении длина тега и индекса занимает 12 разрядов.

Тогда при построении двунаправленного частично ассоциативного кэша под тег выделится 13 разрядов, а под индекс 11. То есть условно мы разделили кэш на два блока. И количество блоков в оперативке, соответственно, выросло в 2 раза, поэтому и длина тэга стала на один больше - 13, чтобы можно было адресоваться, но вместе с этим и строк в каждом блоке стало в 2 раза меньше поэтому длина индекса на 1 упала.

Аналогично для 4-х направленного, длина тэга 14, индекса 10 и тд.

46. Проблема замещения: физически нереализуемые и реализуемые алгоритмы

Ввиду малой емкости в процессе вычислений все блоки кэш-памяти обычно заняты. Занесение в кэш-память копии еще одного блока ОП возможно лишь путем замещения содержимого одного из занятых блоков кэш-памяти. В этой ситуации возникает вопрос оптимального выбора замещаемого блока. Естественно, что наилучшим решением была бы замена блока, обращение к которому произойдет в более отдаленном времени или вообще не случится. Но предсказать точно невозможно! (Информации в книгах нет, но на сайте одном написано *оптимальный алгоритм* и приведен выделенный мной принцип выше)

При прямом отображении каждому блоку основной памяти соответствует только один определенный блок в кэш-памяти, и никакой иной выбор удаляемого блока здесь невозможен. При полностью и частично ассоциативных способах отображения требуется какой-либо алгоритм замещения (выбора удаляемого из кэш-памяти блока). К сожалению, однозначно предсказать, какой из блоков кэш-памяти в будущем будет наименее востребован, практически нереально, и приходится привлекать алгоритмы, уступающие оптимальному. Среди возможных алгоритмов замещения наиболее распространеными являются четыре, рассматриваемые в порядке уменьшения их относительной эффективности.

Наиболее эффективным является алгоритм замещения на основе наиболее давнего использования (LRU — Least Recently Used), при котором замещается тот блок кэш-памяти, к которому дольше всего не было обращения. Проводившиеся исследования показали, что алгоритм LRU, который «смотрит» назад, работает достаточно хорошо в сравнении с оптимальным алгоритмом, «смотрящим» вперед. Наиболее известны два способа аппаратурной реализации этого алгоритма. В **первом** из них с каждым блоком кэш-памяти ассоциируют счетчик. К содержимому всех счетчиков через определенные интервалы времени добавляется единица. При обращении к блоку ее счетчик обнуляется.

Таким образом, наибольшее число будет в счетчике того блока, к которому дольше всего не было обращений, и этот блок — первый кандидат на замещение.

Второй способ реализуется с помощью очереди, куда в порядке заполнения блоков кэш-памяти заносятся ссылки на эти блоки. При каждом обращении к блоку ссылка на него перемещается в конец очереди. В итоге первой в очереди каждый раз оказывается ссылка на блок, к которому дольше всего не было обращений. Именно этот блок прежде всего и замещается.

Другой возможный алгоритм замещения — алгоритм, работающий по принципу «первый вошел, первый вышел» (FIFO — First In First Out). Здесь заменяется блок, дольше всего находившийся в кэш-памяти. Алгоритм легко реализуется с помощью рассмотренной ранее очереди, с той лишь разницей, что после обращения к блоку положение соответствующей ссылки в очереди не меняется.

Еще один алгоритм — замена наименее часто использовавшегося блока (LFU — Least Frequently Used). Замещается тот блок в кэш-памяти, к которому было меньше всего обращений. Принцип можно воплотить на практике, связав каждый блок кэш-памяти со счетчиком обращений, к содержимому которого после каждого обращения добавляется единица. Главным претендентом на замещение является блок, счетчик которого содержит наименьшее число.

Простейший алгоритм — произвольный выбор блока для замены. Замещаемый блок выбирается случайным образом. Реализовано это может быть, например, с помощью счетчика, содержимое которого увеличивается на единицу с каждым тактовым импульсом, вне зависимости от того, имело место попадание или промах. Значение в счетчике определяет заменяемый блок в полностью ассоциативной кэш-памяти или блок в пределах модуля для частично-ассоциативной кэш-памяти. Данный алгоритм используется крайне редко.

Среди известных в настоящее время систем с кэш-памятью наиболее встречаемым является алгоритм LRU.

47. Проблема обеспечение целостности данных: алгоритмы сквозной, буферизованной сквозной и обратной записи

В процессе вычислений ЦП может не только считывать имеющуюся информацию, но и записывать новую, обновляя тем самым содержимое кэш-памяти. С другой стороны, многие устройства ввода/вывода умеют напрямую обмениваться информацией с основной памятью. В обоих вариантах возникает ситуация, когда содержимое блока кэш-памяти и соответствующего блока ОП перестает совпадать. В результате на связанное с основной памятью устройство вывода может быть выдана «устаревшая» информация, поскольку все изменения в ней, сделанные процессором, фиксируются только в кэш-памяти, а процессор будет использовать старое содержимое кэш-памяти вместо новых данных, загруженных в ОП из устройства ввода.

Для разрешения первой из рассмотренных ситуаций (когда процессор выполняет операцию записи) в системах с кэш-памятью предусмотрены методы обновления основной памяти, которые можно разбить на две большие группы: метод сквозной записи (write through, store through) и метод обратной записи (write back, copy back).

По **методу сквозной записи** прежде всего обновляется слово, хранящееся в основной памяти. Если в кэш-памяти существует копия этого слова, то она также обновляется. Если же в кэш-памяти отсутствует нужная копия, то либо из основной памяти в кэш-память пересыпается блок, содержащий обновленное слово (сквозная запись с отображением), либо этого не делается (сквозная запись без отображения). Главное достоинство метода

сквозной записи состоит в том, что если блок кэш-памяти освобождается для хранения другого блока, то удаляемый блок можно не возвращать в основную память, поскольку его копия там уже имеется. Метод достаточно прост в реализации. К сожалению, эффект от использования кэш-памяти (сокращение времени доступа) в отношении к операциям записи здесь отсутствует.

Определенный выигрыш дает его модификация, известная как **метод буферизированной сквозной записи**. Информация сначала записывается в кэш-память и в специальный буфер, работающий по схеме FIFO. Запись в основную память производится уже из буфера, а процессор, не дожидаясь ее окончания, может сразу же продолжать свою работу. Конечно, соответствующая логика управления должна заботиться о том, чтобы своевременно «пустошать» заполненный буфер. При использовании буферизации процессор полностью освобождается от работы с ОП.

Согласно **методу обратной записи**, слово заносится только в кэш-память. Если соответствующего блока в кэш-памяти нет, то нужный блок сначала пересыпается из ОП, после чего запись все равно выполняется исключительно в кэш-память. При замещении (очистки) блока из кэша его необходимо предварительно переслать в соответствующее место основной памяти. Для метода обратной записи, в отличие от алгоритма сквозной записи, характерно то, что при каждом чтении из основной памяти осуществляются две пересылки между основной и кэш-памятью.

У рассматриваемого метода есть **разновидность — метод флаговой обратной записи**. Когда в каком-то блоке кэш-памяти производится изменение, устанавливается связанный с этим блоком бит изменения (флажок). При замещении блок из кэш-памяти переписывается в ОП только тогда, когда его флажок установлен в 1.

Ясно, что эффективность флаговой обратной записи несколько выше. В среднем обратная запись на 10% эффективнее сквозной записи, но для ее реализации требуются и повышенные аппаратные затраты. С другой стороны, практика показывает, что операции записи составляют небольшую долю от общего количества обращений к памяти. Так, в [139] приводится величина 16%. Другие авторы оценивают долю операций записи величинами в диапазоне от 5 до 34%. Таким образом, различие по быстродействию между рассмотренными методами невелико.

Теперь рассмотрим **ситуацию**, когда в основную память из устройства ввода, минуя процессор, заносится новая информация, и неверной становится копия, хранящаяся в кэш-памяти. Предотвратить подобную несогласованность позволяют **два приема**. В первом случае система строится так, чтобы ввод любой информации в ОП автоматически сопровождался соответствующими изменениями в кэш-памяти. Для второго подхода «прямой» доступ к основной памяти допускается только через кэш-память.

Метод сквозной записи.

После изменения данных в кэше инициируется изменение в основной памяти (но так теряется весь смысл кэша).

Метод отложенной (обратной) записи.

Контроллер кэш памяти запоминает в какой строке данные модифицировались, пока строка не уйдет из кэша синхронизация с основной памятью не произойдет.

Модифицированные и не модифицированные строки легко определить одним битом W(1 - модиф, 0 - не модиф).

Метод буферизированной сквозной записи.

Используется буфер на стороне кэш памяти, при выгрузке из кэша копируем данные в буфер, затем на освободившемся месте копируем новые данные из основной памяти и только затем выкачиваем данные из буфера. Таким образом процессор получит эти данные быстрее. Можно оптимизировать замену, при двух кандидатах на замещение выбирать ту строку, которая не менялась (но и увлекаться этим нельзя иначе может возникнуть большая порция модиф данных которые не используются, надо еще смотреть на дату модификации).

48. Модели многоуровневой кэш-памяти

Логично предположить, что с увеличением емкости кэш-памяти можно ожидать и соответствующего повышения быстродействия ВМ. Современные технологии позволяют разместить на общем с процессором кристалле кэш-память достаточно большой емкости. С другой стороны, попытки увеличения емкости обычно приводят к снижению быстродействия, главным образом из-за усложнения схем управления и дешифрации адреса. По этой причине общую емкость кэш-памяти ВМ увеличивают за счет иерархической организации, при которой кэш-память состоит из нескольких уровней запоминающих устройств, отличающихся емкостью и быстродействием. Каждый последующий уровень имеет большую емкость по сравнению с предыдущим, но и меньшее быстродействие.

Первый уровень (L1) иерархии образует наиболее скоростная кэш-память сравнительно небольшой емкости (не более 128 Кбайт). На кристалле ее располагают по возможности ближе к процессору, чтобы минимизировать длину соединяющей их шины и тем самым способствовать ускорению обмена информацией. Этот уровень в перспективных микропроцессорах обычно строится по разделенной схеме (в смысле разделенной на память команд и память данных).

Емкость кэш-памяти **второго уровня (L2)** обычно больше, чем у L1, а быстродействие и стоимость — несколько ниже. В современных микропроцессорах кэш-память второго уровня также размещают на одном кристалле с процессором, за счет чего сокращается длина связей и повышается быстродействие.

Для новейших многоядерных микропроцессоров типично размещение на том же кристалле еще и кэш-памяти **третьего уровня (L3)**. Обычно этот уровень является общим для всех ядер (или группы ядер).

При доступе к памяти ЦП сначала обращается к кэш-памяти первого уровня. В случае промаха производится обращение к кэш-памяти второго уровня и т. д. (вплоть до основной памяти). Потенциальная экономия за счет применения L2 зависит от вероятности попаданий как в L1, так и в L2. Ряд исследований показывает, что использование кэш-памяти второго уровня существенно улучшает производительность.

При построении иерархии кэш-памяти используется один из двух принципов - инклузивный или эксклюзивный.

При **инклузивном подходе** вся информация из верхних уровней иерархии содержится и в последующих нижних уровнях. Это означает, что, например, при замещении блока в L1 удаляемый блок переносится в L2.

В **эксклюзивной схеме** блок, удаляемый из кэш-памяти более высокого уровня, не дублируется в памяти следующего уровня. Вместо этого замещаемый и замещающий блоки взаимно меняются местами. Например, место блока из L2, замещающего какой-то блок в L1, занимает блок, удаленный при этом из L1. Каждый из вариантов имеет свои достоинства. Так, инклузивная схема потенциально обеспечивает более высокое

быстродействие, а эксклюзивная — более эффективное использование емкости второго и последующих уровней кэш-памяти. На практике же сделанные утверждения достаточно условны, и однозначно отдать предпочтение той или иной схеме трудно. В настоящее время достаточно серьезно обсуждается вопрос введения кэш-памяти четвертого уровня (L4). Относительную эффективность многоуровневой кэш-памяти можно привести данные одного из многочисленных исследований. Так, если в системе с одноуровневой памятью количество обращений к кэш-памяти составляет 99%, а к основной памяти — 1%, то в двухуровневой памяти получаем: L1 — 99%, L2 — 0,0099% и ОП — 0,00001%. В случае трехуровневой кэш-памяти получаем: L1 — 99%, L2 — 0,0099%, L3 — 0,000099% и ОП — 0,0000001%.

Пример: Intel Core i7 8700 новейший процессор -объем кэша L1 64 Кб, объем кэша L2 1536 Кб, объем кэша L3 12288 Кб

В вопросе этого нет, но думаю он спросит.

Проблема когерентности.

Поскольку все ядра ссылаются на одну и ту же память (3 уровня) то разумеется, кто-то ее будет менять, из-за чего другие ядра будут читать неправильно. Решают проблему на аппаратном уровне с использованием протокола MESI

modified	exclusive	shared	invalid
Строка изменена	Строка только у меня	Копии этой строки есть у кого-то еще	Кто-то изменил данные и теперь они не верные

То есть, строят граф с 4 состояниями используя 2 бита статистики. Например, S -> M, надо остальным процессорам отправить флаг I, означающий, что строкой больше пользоваться нельзя.

49. Сегментная организация памяти

При страничной организации предполагается, что виртуальная память — это непрерывный массив со сквозной нумерацией слов, что не всегда можно признать оптимальным. Обычно программа состоит из нескольких частей — кодовой, информационной и стековой. Так как заранее неизвестны длины этих составляющих, то удобно, чтобы при программировании каждая из них имела собственную нумерацию слов, отсчитываемых с нуля. Для этого организуют систему сегментированной памяти, выделяя в виртуальном пространстве независимые линейные пространства переменной длины, называемые сегментами. Каждый сегмент представляет собой отдельную логическую единицу информации, содержащую совокупность данных или программный код и расположенную в адресном пространстве пользователя. В каждом сегменте устанавливается своя собственная нумерация слов, начиная с нуля. Виртуальная память также разбивается на сегменты, с независимой адресацией слов внутри сегмента. Каждой составляющей программы выделяется сегмент памяти. Виртуальный адрес определяется номером сегмента и адресом внутри сегмента. Для преобразования виртуального адреса в физический используется специальная сегментная таблица.

Недостатком такого подхода является то, что неодинаковый размер сегментов приводит к неэффективному использованию ОП. Так, если ОП заполнена, то при замещении одного

из сегментов требуется вытеснить такой, размер которого равен или больше размера нового. При многократном повторе подобных действий в ОП остается множество свободных участков, недостаточных по размеру для загрузки полного сегмента. Решением проблемы служит сегментно-страничная организация памяти. В ней размер сегмента выбирается не произвольно, а задается кратным размеру страницы.

50. Предпосылки необходимости сегментации, реализация в реальном режиме работы процессора

Для большинства типичных применений ВМ характерна ситуация, когда размещение всей программы в ОП невозможно из-за большого размера программы. Решение этой проблемы базируется на свойстве локальности по обращению, согласно которому в каждый момент времени «внимание» машины концентрируется на определенных сравнительно небольших участках программы. Таким образом, в ОП достаточно хранить только используемые в данный период фрагменты программ, а остальные их части могут располагаться на внешних ЗУ (ВЗУ). При таком подходе, однако, необходимо принимать во внимание еще одну проблему.

Эта проблема связана с линейностью адресации памяти. Загружаемые в ОП программы или их фрагменты имеют различный размер, но линейную адресацию, то есть должны располагаться в непрерывной области памяти. Для занесения в ОП очередного фрагмента нужно освободить для него непрерывную область памяти, удалив другой фрагмент, но такой, размер которого равен или больше размера загружаемой части программы. Из-за различия в размерах фрагментов, как правило, приходится удалять больший по размеру фрагмент. В итоге свободной остается небольшая по размеру область памяти. При многократном повторе подобных действий в ОП остается множество свободных участков, недостаточных по размеру для загрузки полного непрерывного фрагмента программы. Возникает состояние, известное как *фрагментация памяти*.

Базисом для решения перечисленных проблем служит идея *виртуализации памяти*, под которой понимается такой метод автоматического управления иерархической памятью, при котором программисту кажется, что он имеет дело с единой памятью большой емкости и высокого быстродействия. Эту память называют виртуальной памятью (ВП). По своей сути виртуализация памяти представляет собой способ аппаратной и программной реализации концепции иерархической памяти.

В рамках идеи виртуализации памяти ОП рассматривается как линейное пространство N адресов, называемое физическим пространством памяти. Для задач, где требуется более чем N ячеек, предоставляется значительно большее пространство адресов (обычно равное общей емкости всех видов памяти), называемое виртуальным пространством, в общем случае не обязательно линейное. Адреса виртуального пространства называют виртуальными, а адреса физического пространства — физическими.

См. 49

51. Сегментная организация памяти в защищённом режиме: принципы построения, дескрипторы и дескрипторные таблицы

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

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

Процессор может обращаться только к тем сегментам памяти, для которых имеются дескрипторы в таблицах.

Если в 8086 единственным атрибутом сегмента был его начальный адрес, то в Р-режиме старших моделей семейства x86 для описания многочисленных атрибутов предусмотрена специальная структура — дескриптор. Дескриптор — это 8-байтовый блок, содержащий атрибуты области линейных адресов — сегмента. Дескриптор включает в себя информацию о положении сегмента в линейном адресном пространстве, размере сегмента, типе информации, хранящемся в сегменте и правах доступа к ней, а также другие атрибуты сегмента.

Сегмент в защищенном режиме — область памяти, снабженная рядом атрибутов: типом, размером, положением в памяти, уровнем привилегий и др. Сегмент может начинаться и кончаться где угодно, и его размер — произвольный

Дескрипторы хранятся в памяти и группируются в дескрипторные таблицы:

- GDT — глобальная дескрипторная таблица;
- IDT — дескрипторная таблица прерываний;
- LDT — локальная дескрипторная таблица.

Причем если GDT и IDT — общесистемные, присутствуют в системе в единственном экземпляре и являются общими для всех задач, то LDT может создаваться для каждой задачи.

Дескрипторная таблица локализуется в памяти с помощью соответствующего регистра. 48-битовые регистры GDTR и IDTR содержат 32-битовое поле базового адреса таблицы и 16-битный предел (размер) таблицы с байтовой гранулярностью.

Дескриптор – 8 байт (64 бита – размер теневых регистров):

- базовый адрес (32 бита);
- предел (20 бит) – в байтовом режиме при G=0, иначе по 4Кб;
- атрибуты (4 бита), в т.ч. бит гранулярности G;
- права доступа (8 бит)

P	DPL	S	Type	A
---	-----	---	------	---

Дескрипторные таблицы: GDT, IDT, LDT

Регистры GDTR и IDTR: 32 (база) + 16 (размер) = 48 бит

Регистр LDTR: селектор 16 бит (индекс 13 бит + TI + 2 бит RPL)

Сегментные регистры: аналогичны LDTR

Поле *атрибутов* включает следующие признаки:

G — бит гранулярности. При значении G = 0 размер сегмента задается в байтах, а при G = 1 — в страницах по 4 Кбайт.

Байт *права доступа* (AR) имеет несколько отличающуюся структуру для дескрипторов сегментов разных типов, но некоторые поля этого байта являются общими для всех дескрипторов:

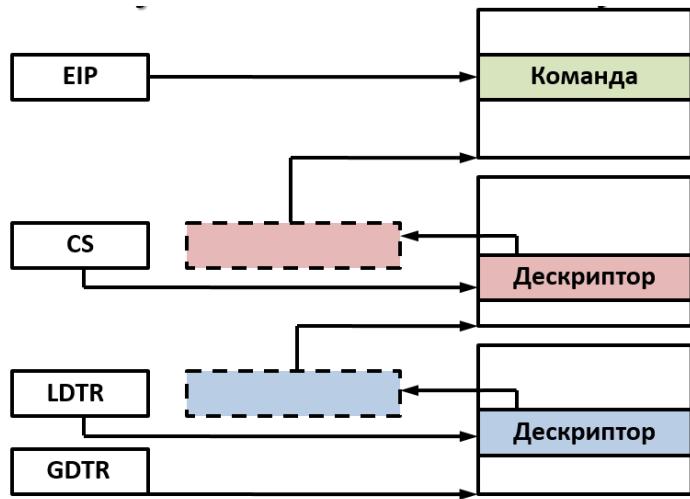
P — бит присутствия (от англ. present) сегмента, если P = 0, то дескриптор не может использоваться, т. к. сегмент отсутствует в ОЗУ. При обращении к сегменту, дескриптор которого имеет P = 0, формируется соответствующее прерывание;

A — бит обращения, устанавливается, когда проходит обращение к сегменту. Операционная система может следить за частотой обращения к сегменту путем периодического анализа и очистки A

52. Сегментная организация памяти в защищённом режиме: принципы построения, схема получения линейного адреса

Линейный адрес – это одна из форм виртуального адреса. Исходный двоичный виртуальный адрес, вычисляемый в соответствии с используемой адресацией, преобразуется в линейный. В свою очередь, линейный адрес будет либо равен физическому (если страничное преобразование отключено), либо с помощью страничной трансляции преобразуется в физический адрес. Если же смещение из регистра EIP превышает размер сегмента кода, то эта аварийная ситуация вызывает прерывание и управление должно передаваться супервизору ОС.

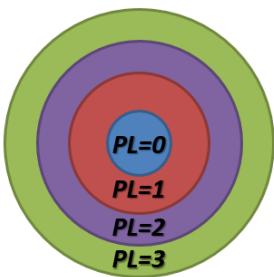
Стоит отметить, что поскольку межсегментные переходы происходят нечасто, то, как правило, определение линейного адреса заключается только в сравнении значения EIP (указатель команды) с полем предела сегмента и в прибавлении смещения к началу сегмента. Все необходимые данные уже находятся в микропроцессоре, и операция получения линейного адреса происходит очень быстро



Процесс загрузки дескрипторных регистров и преобразования эффективного (логического) адреса в линейный протекает следующим образом:

1. При переходе в защищенный режим в памяти создается глобальная дескрипторная таблица, базовый адрес которой размещается в регистре GDTR.
2. Несколько сегментов определяется в памяти, и их дескрипторы помещаются в GDT.
3. При запуске очередной задачи можно определить дополнительно несколько сегментов и для хранения их дескрипторов создать локальную дескрипторную таблицу, как системный сегмент, дескриптор которого хранится в GDT, а его положение в GDT определяется селектором в регистре LDTR. В теневой регистр LDTR автоматически помещается дескриптор сегмента LDT.
4. При загрузке в любой сегментный регистр нового содержимого в соответствующий теневой регистр автоматически помещается новый дескриптор из GDTR или LDTR.
5. При генерации программой очередного адреса EA из соответствующего теневого сегментного регистра выбирается базовый адрес сегмента и складывается со значением EA. Полученная сумма представляет собой линейный адрес.

53. Защита памяти на уровне сегментов



Защита доступа к данным:

$$DPL \geq \max(RPL, CPL)$$

Защита доступа к стеку:

$$DPL = \max(RPL, CPL)$$

Защита доступа к коду:

$$DPL = \max(RPL, CPL)$$

Передача управления между кольцами защиты:

- подчинённые сегменты
- шлюзы вызова



В x86 определено понятие привилегии для сегмента и установлены 4 уровня привилегий PL (рис. 7.7), которые задаются номерами от 0 (наиболее защищенный) до 3 (низший). В ядро входит часть ОС, обеспечивающая инициализацию работы, управление доступом к памяти, защиту и ряд других жизненно важных функций, нарушение которых полностью выводит из строя МПС. Основная часть ОС должна иметь уровень 1. К уровню 2 обычно относят ряд служебных программ ОС, например, драйверы внешних устройств, системы управления базами данных, специализированные подсистемы программирования и др. Ранее отмечалось, что основой организации памяти x86 является сегмент. С каждым сегментом (данных, кода или стека) ассоциируется уровень привилегий DPL и все, что находится внутри этого сегмента, имеет данный уровень привилегий. DPL располагается в байте доступа дескриптора сегмента, поэтому его называют уровнем привилегий дескриптора (Descriptor Privilege Level), однако правильнее считать его уровнем привилегий сегмента.

Уровень привилегий выполняющегося кода называется текущим уровнем привилегий CPL (Current Privilege Level или Code Privilege Level) и он задается полем RPL селектора в сегментном регистре CS. Значение CPL можно считать уровнем привилегий процессора в текущий момент времени, т. к. при передаче управления сегменту кода с другим уровнем привилегий процессор будет работать на новом уровне привилегий.

Каждый селектор выбирает точно один дескриптор и, соответственно, один сегмент, но конкретный сегмент могут идентифицировать несколько селекторов ("альтернативное именование"). Младшие два бита селектора содержат поле запрашиваемого уровня привилегий RPL (Requested Privilege Level). Это поле не влияет на выбор дескриптора, но учитывается при контроле привилегий.

Таким образом, текущее состояние системы защиты характеризуется следующими признаками:

- CPL — уровень привилегий выполняемого кода, размещается в поле RPL сегментного регистра кода CS;
- DPL — уровни привилегий для каждого из восьми открытых сегментов, располагаются в байте доступа дескрипторов, помещенных в "теневые регистры";
- RPL — определяют уровни привилегий источника селектора, размещаются в полях RPL сегментных регистров.

Процессор постоянно контролирует, обладает ли текущая программа достаточным уровнем привилегий, чтобы:

- выполнять некоторые команды;
- обращаться к данным других программ;
- передавать управление внешнему (по отношению к программе) коду командами передачи управления типа FAR.

В системе команд существуют специальные привилегированные

Передача управления между задачами контролируется вентилями (gate), называемыми также шлюзами, проверяющими правила использования уровней привилегий. Через вентили задачи могут получить доступ только к разрешенным им сервисам других сегментов.

54. Страницчная организация памяти: назначение, принципы построения, схема получения физического адреса

Целям преобразования виртуальных адресов в физические служит страницчная организация памяти.

Все линейное адресное пространство делится на разделы, число которых может достигать 1024. Каждый раздел, в свою очередь, может содержать до 1024 страниц, размер которых фиксирован — 4 Кбайт, причем начальные адреса страниц жестко фиксированы в физическом адресном пространстве: границы страниц совпадают с границами 4-килобайтовых блоков.

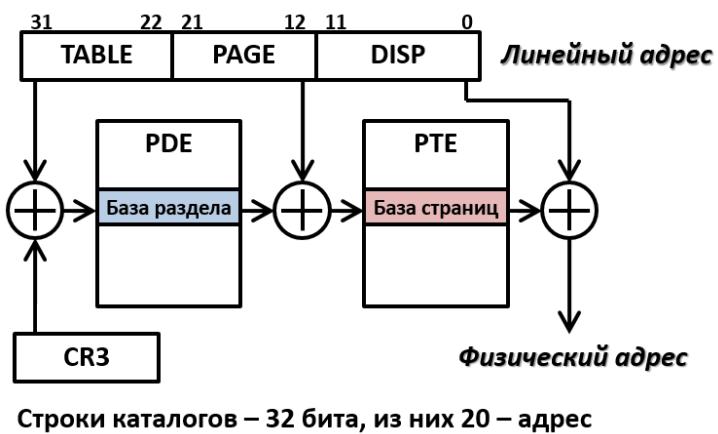
32-разрядный логический адрес, полученный на предыдущем этапе преобразования адреса, рассматривается состоящим из трех полей:

- [31:22] — номер раздела (TABLE);
- [21:12] — номер страницы в разделе (PAGE);
- [11:0] — номер слова на странице (смещение).

Начальные адреса страниц данного раздела (вместе с атрибутами страницы) хранятся в памяти в страницной таблице, размер которой $1024 \text{ стр.} \times 4 \text{ байта} = 4096 \text{ байтов}$.

Поскольку в задаче может быть несколько разделов и, следовательно, столько же страницных таблиц, то начальные адреса всех страницных таблиц одного сегмента хранятся в специальной таблице — каталоге раздела.

Линейный 32-разрядный адрес является исходной информацией для формирования 32-разрядного физического адреса (рис. ниже) с помощью каталога раздела и страницной таблицы (СТ). Старшие 10 разрядов линейного адреса определяют номер строки каталога разделов, который локализуется содержимым системного регистра CR3.



Поскольку каталог разделов имеет размер 1 Кбайт \times 4 байта, он занимает точно одну страницу ($CR3[11:0] = 0$) и содержит 4-байтовые поля. Помимо базового адреса страницной таблицы, это поле хранит атрибуты страницы. Извлеченный из каталога базовый адрес страницной таблицы складывается (конкатенируется) с разрядами [21:12] линейного адреса для получения адреса строки страницной таблицы, из которой, в свою очередь, извлекается базовый адрес страницы. Конкатенацией базового адреса страницы с разрядами [11:0] линейного адреса получается физический адрес.

Центральный процессор обращается к ячейке, указав ее виртуальный адрес (1), состоящий из номера виртуальной страницы и смещения относительно ее начала. Этот адрес поступает в систему преобразования адресов (2) с целью получения из него физического адреса ячейки в основной памяти (3). Поскольку смещение в виртуальном и физическом адресе одинаковое, преобразованию подвергается лишь номер страницы. Если преобразователь обнаруживает, что нужная физическая страница отсутствует в основной памяти (произошел промах или страничный сбой), то нужная страница считывается из внешней памяти и заносится в ОП (4, 5). Преобразователь адресов — это часть операционной системы, транслирующая номер виртуальной страницы в номер физической страницы, расположенной в основной памяти, а также аппаратура, обеспечивающая этот процесс и позволяющая ускорить его. Преобразование осуществляется с помощью так называемой страничной таблицы.

55. Защита памяти на уровне страниц

Бит R/W – для PL=3 при R/W=0 – только чтение

(для других PL требуется выставлять специальный
бит WR в системном регистре CR0)

**Бит U/S – для PL=3 при U/S=0 – блокировка
чтения (ограничение адресуемой области)**

Кэширование

TLB – полностью ассоциативный кэш на 32 ячейки

- P — бит присутствия;
- R/W — чтение/запись;
- U/S — пользователь/супервизор;
- A — бит доступа;
- D — признак записи на страницу;

Ограничение адресуемой области.

Для страниц и сегментов привилегии интерпретируются по-разному: для сегментов — 4 уровня, для страниц — только 2, определяемые битом U/S. При U/S = 0 страница имеет уровень супервизора, иначе — уровень пользователя. На уровне супервизора работают обычно операционные системы, драйверы ВУ, а также располагаются защищенные данные (например, страничные таблицы). Уровни привилегий сегментов отображаются на уровень привилегий страниц: если значение CPL равно 0, 1 или 2, то процессор работает на уровне супервизора, при CPL = 3 — на уровне пользователя. На уровне супервизора доступны все страницы, а на уровне пользователя — только страницы уровня пользователя.

Контроль типа.

Механизм защиты распознает только два типа страниц: с доступом только по считыванию (R/W = 0) и с доступом по считыванию/записи (R/W = 1), причем в 80386 ограничение по записи действительно только для уровня пользователя. Программа уровня супервизора игнорирует значение бита R/W и может записывать на любые страницы. В отличие от 80386, процессор 80486 разрешает защитить от записи страницы уровня пользователя в режиме супервизора: установка в регистре CR0 бита WR = 1 обеспечивает чувствительность режима супервизора к защищенным от записи страницам режима пользователя. Для любой страницы атрибуты защиты ее элемента каталога разделов могут

отличаться от атрибутов защиты ее элемента страничной таблицы. Процессор контролирует атрибуты защиты в таблицах обоих уровней и принимает решение таким образом, что всякое разночтение в уровне привилегий раздела и страницы всегда разрешается в сторону большей защиты (предоставления меньших прав пользователю).

56. Управление задачами

Под мультизадачностью понимают способность процессора выполнять несколько задач "одновременно". Конечно, процессор традиционной архитектуры не может выполнять строго одновременно более одного потока команд, однако он может некоторое время выполнять один поток команд, потом быстро переключиться на выполнение другого потока команд, потом третьего, потом — снова первого и т. д. Такая организация вычислительных процессов при высоком быстродействии процессора создает иллюзию одновременности (параллельности) выполнения нескольких задач.

Под задачей в мультизадачной системе понимается программа, которая выполняется или ожидает выполнения, пока выполняется другая задача, причем в определение задачи обычно включают ресурсы, требуемые для ее решения (объем памяти, процессорное время, дисковое пространство и др.).

Переключение задач в мультизадачной системе предполагает сохранение состояния приостанавливаемой задачи на момент ее останова. Информация о задаче, сохраняемая для последующего восстановления прерванного процесса, называется ее контекстом. В системе выделяется область оперативной памяти, доступная только ОС, в которой хранятся контексты задач. Для минимизации времени переключения контекста следует сохранять и восстанавливать минимальную информацию о каждой задаче. В какой-то степени процесс переключения задачи напоминает вызов процедуры. Отличие состоит в том, что при вызове процедуры информация о точке возврата (автоматически) и содержимое некоторых РОН (программно) помещается в стек, что определяет свойство реентерабельности процедур (возможность вызова самой себя). Задачи не являются реентерабельными, т. к. контексты сохраняются не в стеке, а в фиксированной (для каждой задачи) области памяти в специальной структуре данных, называемой сегментом состояния задачи (Task State Segment, TSS), причем каждой задаче соответствует один TSS.

Сегмент TSS определяется дескриптором, который может находиться только в GDT. Формат дескриптора TSS похож на дескриптор сегмента кода и содержит обычные для дескриптора сегмента поля: базового адреса, предела, DPL, биты гранулярности ($G = 0$) и присутствия P, бит $S = 0$ — признака системного сегмента. В поле типа бит занятости B показывает, занята задача или нет. Занятая задача выполняется сейчас или ожидает выполнения. При переключении задач между ними не передается никакой информации, т. е. они максимально изолированы друг от друга. Этим исключается искажения задач и обеспечивается возможность прекращения и запуска любой задачи в любой момент времени и в любом порядке.

Переключение задачи в x86 могут вызвать следующие четыре события:

- старая задача выполняет команду FAR CALL или FAR JMP, и селектор выбирает *шлюз задачи*;
- старая задача выполняет команду FAR CALL или FAR JMP, и селектор выбирает *дескриптор TSS*;
- старая задача выполняет команду IRET для возврата в предыдущую задачу; эта команда приводит к переключению задачи, если в регистре EFLAGS бит *вложенной задачи* NT = 1;
- возникло аппаратное или программное прерывание и соответствующий элемент дескрипторной таблицы прерываний IDT содержит *шлюз задачи*.

Таким образом, селекторами в командах переходов и вызовов могут быть как селекторы TSS (прямое переключение задачи), так и селекторы шлюзов задачи (косвенное переключение задачи). В последнем случае дескриптор шлюза задачи обязательно содержит селектор TSS.

Хотя шлюзы вызова и располагаются в дескрипторной таблице, они, по существу, дескрипторами не являются, т. к. не определяют никакого сегмента. Поэтому в дескрипторе шлюза отсутствует база и граница сегмента, а содержится лишь селектор вызываемого сегмента программы и относительный адрес шлюза — задается фактически адрес селектор: смещение точки входа той процедуры (назначения), которой шлюз передает управление. Байт доступа имеет тот же смысл, что и в обычных дескрипторах, а пятибитовое поле WC указывает количество параметров, переносимых из стека текущей программы в стек новой программы.

НАДО ИСКАТЬ ЕЩЕ

57. Прерывания и исключения: источники, порядок обработки

Источники прерываний:

- внешний сигнал по входам INTR или NMI
- исключение (особый случай) в ходе выполнения программы
- программный вызов прерывания

Особые случаи:

- нарушение (fault)
- ловушка (trap)
- авария (abort)

Сигналы, извещающие микропроцессор либо о том, что некоторое внешнее устройство «просит уделить ему внимание» (INTR), либо о том, что требуется безотлагательная обработка некоторого события или катастрофическая ошибка (NMI).

Нарушение (fault) — этот особый случай процессор может обнаружить до возникновения фактической ошибки (например, нарушение правил привилегий или отсутствие сегмента в

оперативной памяти). Очевидно, после обработки нарушения можно продолжить программу, осуществив рестарт виновной команды.

Ловушка (trap) — обнаруживается после окончания выполнения виновной команды. После ее обработки процессор возобновляет действия с той команды, которая следует за "захваченной" (например, прерывание при переполнении или команда INT n). Большинство отладочных контрольных точек также интерпретируются как ловушки.

Авария (abort) — приводит к потере контекста программы, ее продолжение невозможно. Причину аварии установить нельзя, поэтому осуществить рестарт программы не удается, ее необходимо прекратить. К авариям ("выходам из процесса") относятся аппаратные ошибки, а также несовместимые или недопустимые значения в системных таблицах.

Прерывание — это приостановка выполнения программы в процессоре с целью выполнения какой-то более важной или нужной в данный момент другой программы или процедуры, после завершения которой продолжается выполнение прерванной программы с момента ее прерывания. Прерывание позволяет компьютеру приостановить любое свое действие и временно переключиться на другое, как заранее запланированное, так и неожиданное, вызванное непредсказуемой ситуацией в работе машины или ее компонента. Каждое прерывание влечет за собой загрузку определенной программы, предназначенней для обработки возникшей ситуации — программы обработки прерывания. Организация и управление прерываниями функционально во многом смыкается с управлением задачами — одной из базовых функций операционных систем. Основой для управления процессом одновременного решения нескольких задач (равно как и управления прерываниями) являются процедуры:

- выбора очередной задачи или определения приоритета задачи;
- сохранения информации о статусе задачи при ее прерывании (формирование слова состояния программы);
- упраждения и устранения конфликтов между задачами (координации и синхронизации выполнения задач).

Видов (номеров) прерывания может быть всего 256, и, соответственно, векторов прерывания (адресов CS:IP программ обработки прерываний) в ОП насчитывается до 256. Классификация видов прерываний показана на рис.

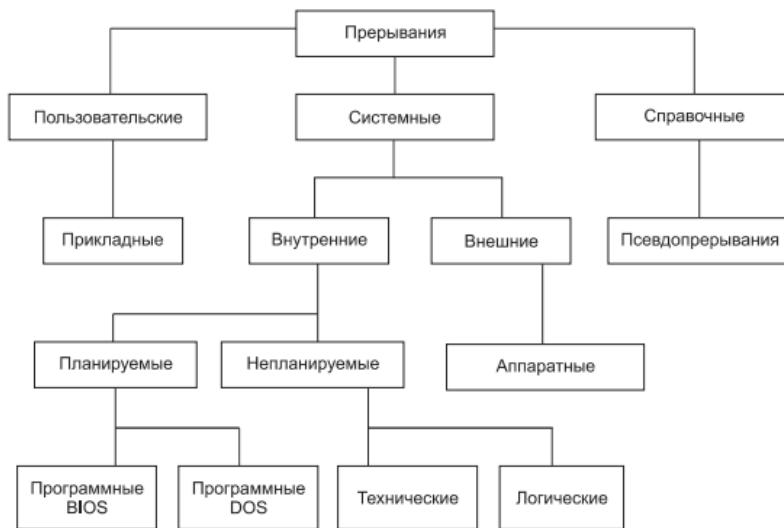
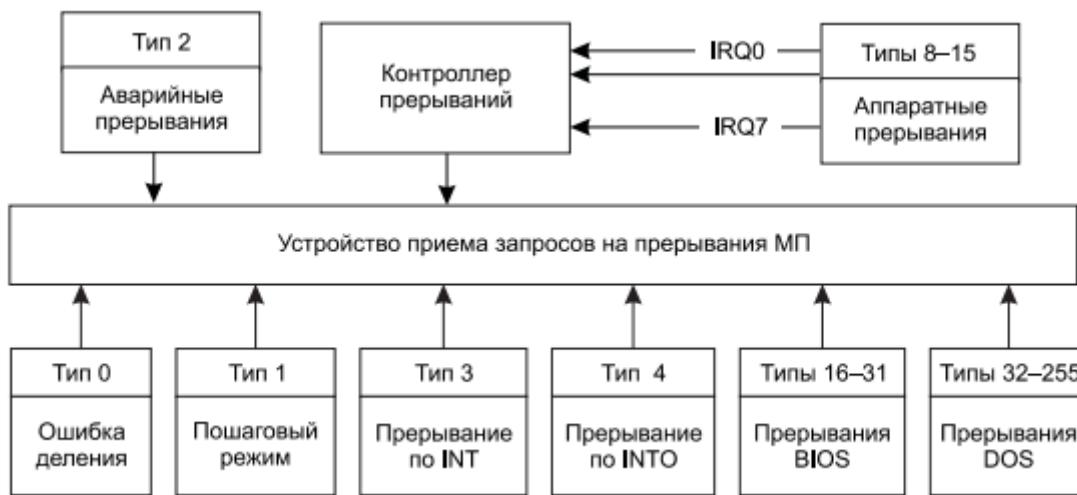


Схема организации приема запросов



Прерывания с 0-го по 31-е и прерывание 64 относятся к прерываниям нижнего уровня, обслуживаемым BIOS; прерывания, начиная с 32-го — являются прерываниями верхнего уровня (за исключением прерывания 64); причем прерывание 33 (21h) — это комплексное, чаще всего используемое в программах пользователя прерывание, имеющее около 100 разновидностей (служебных функций DOS).

58. Организация таблицы прерываний в реальном и защищённом режимах работы процессора

В реальном режиме:

вектор прерывания — 1 байт, всего 256 прерываний

таблица прерываний — элементы по 4 байта **CS:IP**

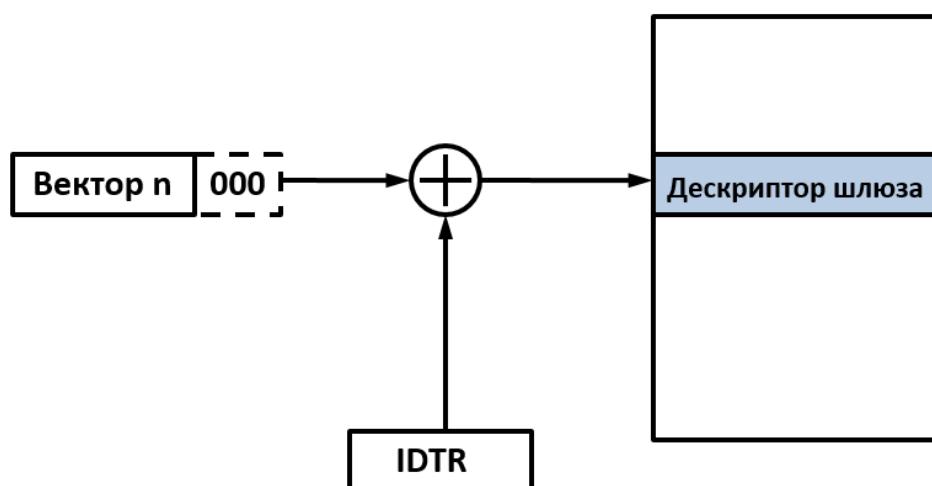
В защищённом режиме:

дескрипторная таблица прерываний, адресуемая **IDTR**

размер — 256 x 8 байт = 2 Кб

различают 3 вида дескрипторов: шлюз ловушки, шлюз

прерывания (сброс флага IF) и шлюз задачи



Дескрипторная таблица прерываний IDT является прямой заменой таблицы векторов прерываний процессора 8086. Она должна определять 256 обработчиков прерываний и

особых случаев, поэтому ее максимальный размер составляет $256 \times 8 = 2048$ байтов. Таблица IDT может находиться в любой области памяти, процессор локализует ее с помощью 48-битного регистра IDTR, который содержит базовый адрес и предел таблицы. Таблицу не рекомендуется объявлять короче максимального размера, т. к. любое обращение за пределы таблицы вызывает нарушение общей защиты, а вектор внешнего прерывания, вообще говоря, может иметь любое значение в диапазоне 00—FFh. В таблице IDT разрешается применять только три вида дескрипторов: шлюз ловушки, шлюз прерывания и шлюз задачи (но не дескриптор TSS). Шлюзы прерывания и ловушки имеют большое сходство со шлюзом вызова (см. рис. 7.9). Единственное отличие состоит в отсутствии в этих шлюзах 5-битового поля счетчика WC, которое в шлюзе вызова определяет число параметров, передаваемых в вызываемую подпрограмму через стек. Соответствующее поле в шлюзах прерывания и ловушки зарезервировано.

В качестве дескрипторов таблицы IDT могут использоваться дескрипторы разного типа – шлюза прерывания, шлюза ловушки и шлюза задачи. Поведение процессора при поступлении запроса на прерывание зависит, как от типа применяемого для данного прерывания дескрипторов, так и от текущего уровня привилегий задачи.

В случае, если в качестве дескриптора прерывания используется шлюз задачи, процессор при поступлении запроса на прерывание выполняет все действия по переключению на новую задачу, сохраняя текущее состояние в TSS старой задачи и загружая новые значения в регистры из TSS задачи обработчика прерывания. Применение такого шлюза оправдано для обработки ошибок и сбоев, которые не могут быть обработаны в рамках текущей задачи (например, ошибки, возникающие при переключении задач).

Если в качестве дескриптора прерывания используется шлюз ловушки или шлюз прерывания, то переключения задач не происходит и программа обработчик функционирует в рамках той же задачи, что и прерванный код. Отличие шлюза прерывания состоит лишь в том, что процессор автоматически сбрасывает флаг EFLAGS.IF при передаче управления в обработчик, чем обеспечивается маскирование внешних прерываний. В последующем, при возврате из процедуры обработчика прежнее значение регистра EFLAGS восстанавливается из стека.

59. Стратегии замещения страниц в памяти

Большинство ОС используют сегментно-страничную виртуальную память. Для обеспечения нужной производительности менеджер памяти ОС старается поддерживать в оперативной памяти актуальную информацию, пытаясь угадать, к каким логическим адресам последует обращение в недалеком будущем. Это влияет на производительность системы.

Наилучший алгоритм замещения страниц несложно описать, но совершенно невозможно реализовать. В нем все происходит следующим образом. На момент возникновения ошибки отсутствия страницы в памяти находится определенный набор страниц. К некоторым из этих страниц будет осуществляться обращение буквально из следующих команд (эти команды содержатся на странице). К другим страницам обращения может не быть и через 10, 100 или, возможно, даже через 1000 команд. Каждая страница может быть помечена количеством команд, которые должны быть выполнены до первого обращения к странице.

Оптимальный алгоритм замещения страниц гласит, что должна быть удалена страница, имеющая пометку с наибольшим значением. Если какая-то страница не будет использоваться на протяжении 8 миллионов команд, а другая какая-нибудь страница не будет использоваться на протяжении 6 миллионов команд, то удаление первой из них приведет к ошибке отсутствия страницы, в результате которой она будет снова выбрана с диска в самом отдаленном будущем. Компьютеры, как и люди, пытаются по возможности максимально отсрочить неприятные события.

Единственной проблемой такого алгоритма является невозможность его реализации. К тому времени, когда произойдет ошибка отсутствия страницы, у операционной системы не будет способа узнать, когда каждая из страниц будет востребована в следующий раз.