ЭВМ и периферия

Курс лекций в форме презентаций

Автор и разработчик: преподаватель СФТИ НИЯУ МИФИ

Чернышев Олег Юрьевич

Снежинск
2016

Базовая структура компьютера

Функциональная структура

Компьютер состоит из пяти главных , функционально независимых частей: устройство ввода, устройство памяти, арифметико-логическое устройство, устройство вывода и устройства управления На рис. 1.1. представлено классическое построение ЭВМ первого поколения. Устройство ввода принимает через цифровые линии связи закодированную информацию от операторов, электромеханических устройств типа клавиатуры или от других компьютеров сети. Полученная информация либо сохраняется в памяти компьютера для последующего применения, либо немедленно используется арифметическими и логическими схемами для выполнения необходимых операций. Последовательность шагов обработки определяется хранящейся в памяти программой

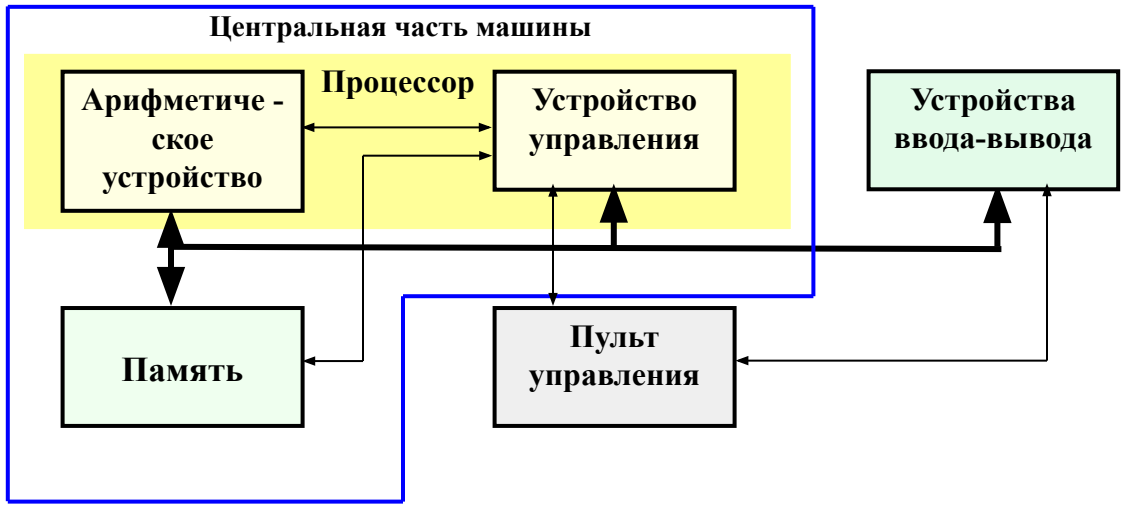


Рис. 1. Классическое построение электронной вычислительной машины (жирными линиями выделены информационные связи, тонкими – управляющие)

Полученные результаты отправляются обратно во внешний мир, посредством устройства вывода. Все эти действия координируются блоком (устройством) управления. Арифметические и логические схемы в комплексе с главными управляющими схемами называют *процессором,* а все вместе взятое оборудование для ввода и вывода часто называют *устройством ввода-вывода (input-output unit).*

Обрабатываемую компьютером информацию принято разделять на две основные категории: *команды и данные. Команды,* или *машинные команды, - это явно заданные инструкции, которые:*

- управляют пересылкой информации внутри компьютера, а также между компьютером и его устройствами ввода-вывода;
- определяют подлебжащие выполнению арифметические и логические операции.

Список команд, выполняющих некоторую задачу, называется *программой.* Обычно программы хранятся в памяти. Процессор по очереди извлекает команды программы из памяти и реализует (выполняет) определяемые ими операции. Компьютер полностью управляется *хранимой программой,* если не считать возможность внешнего вмешательства оператора (человека управляющего ходом выполнения программы) и подсоединенных к машине устройств ввода-вывода.

Данные – это числа и закодированные символы, используемые в качестве *операндов* команд. *Операнды* – это то, над чем выполняются действия в команде. Однако, термин «данные» часто используется для обозначения любой цифровой информации. Согласно этому определению, сама программа (то ебсть список команд) также может считаться данными, если она обрабатывается другой программой. Примером обработки одной программой другой является *компиляция исходной программы* – перевод программы, написанной на языке высокого уровня, в список машинных команд, составляющих программу на машинном языке, которая начинает называться *объектной программой*. Исходная программа поступает на вход компилятора, который транслирует ее в программу на машинном языке.

Информация, предназначенная для обработки компьютером, должна быть закодирована, чтобы иметь подходящий для компьютера формат. Современное аппаратное обеспечение в большинстве своем основано на цифровых схемах, у которых имеется только два устойчивых состояния ON и OFF (включено и выключено 1 или 0). В результате кодирования любое число, символ или команда преобразуются в строку двоичных цифр, называемых *битами*, каждый из которых имеет одно из двух возможных значений: 0 или 1.

Буквы и цифры также представляются посредством двоичных кодов. Для них разработано несколько разных схем кодирования. Наиболее распространенными считаются схемы ASCII (American Standard Code for Information Interchange – американский стандартный код для обмена информацией), где каждый символ представлен 7-битным кодом, и EBCDIC (Extended Binary Coded Decimal Interchange Code –расширенный двоично-десятичный код для обмена информацией), в котором для кодирования символа используется 8 бит.

Компьютер принимает кодированную информацию через устройство ввода, задачей которого является чтение данных. Наиболее распространенным устройством ввода является клавиатура. При нажатии пользователем компьютера клавиши клавиатуры, соответствующая буква или цифра автоматически преобразуется в определенный двоичный код и по кабелю пересылается либо в память, либо процессору

Существуют другие устройства ввода, например, такие как : джойстики, трекболы и мыши. Они используются совместно с дисплеем в качестве графических входных устройств. Для ввода звука могут использоваться микрофоны. Воспринимаемые ими звуковые колебания измеряются и конвертируются (преобразуются) в цифровые коды для хранения и обработки.

Блок памяти

Задачей блока памяти является хранение программ и данных. Существует два класса запоминающих устройств, а именно первичные и вторичные. *Первичное запоминающее устройство (primary storage)* – это память, быстродействие которой определяется скоростью работы электронных схем. Пока программа выполняется, она должна храниться в первичной памяти. Эта память состоит из большого количества полупроводниковых ячеек, каждая из которых может хранить один бит информации. Ячейки редко считываются по отдельности – обычно они обрабатываются группами фиксированного размера, называемыми *словами*. Память организована так, что содержимое одного слова, содержащего *n* бит, может записываться или считываться за одну базовую операцию.

Для облегчения доступа к словам в памяти с каждым словом связывается отдельный *адрес*. Адреса – это числа, идентифицирующие конкретные местоположения слов в памяти. Для того, чтобы прочитать слово из памяти или записать его в таковую, необходимо указать его адрес и задать управляющую команду, которая начнет соответствующую операцию.

Количество битов в каждом слове часто называю *длиной машинного слова*. Обычно слово имеет длину от 16 до 64 бит. Одним из факторов, характеризующих класс компьютера, является емкость его памяти. Малые машины обычно могут хранить лишь несколько десятков миллионов слов, тогда как средние и большие машины обычно способны хранить сотни миллионов слов. Типичными единицами измерения количества обрабатываемых машиной данных являются слово, несколько слов или часть слова. Как правило, за время одного обращения к памяти считывается или записывается только одно слово.

Во время выполнения программа все время должна находиться в памяти. Команды и данные должны записываться в память и считываться из памяти под управлением процессора. Исключительно важна возможность предельно быстрого доступа к любому слову памяти. Память, к любой точке которой можно получить доступ за короткое и фиксированное время, называется *памятью с произвольным доступом* (Random-Access Memory, RAM). Время, необходимое для доступа к одному слову, называется *временем доступа к памяти*. Это время всегда одинаково, независимо от того, где располагается нужное слово. Время доступа к памяти в современных устройствах RAM составляет от нескольких наносекунд до 100. Память компьютера обычно имеет *иерархическую структуру*, состоящую из трех или четырех уровней полупроводниковых RAM-элементов с различной скоростью и разным размером. Наиболее быстродействующим типом RAM-памяти является *кэш-память* (или просто *кэш*). Она напрямую связана с процессором и часто находится на одном с ним интегрированном чипе, благодаря чему работа процессора значительно ускоряется. Память большой емкости, но менее быстрая, называется *основной памятью* (main memory) или *оперативной памятью. Вторичные запоминающие устройства*, являющиеся более дешевыми, используются для хранения больших объемов данных и большого количества программ. Наиболее широко используются: *магнитные диски, магнитные ленты, оптические диски* (CD-ROM).

Арифметико-логическое устройство

Большинство компьютерных операций выполняется в *арифметико-логическом устройстве* (АЛУ) процессора. Типичным примером может служить выполнение сложения или другой арифметической или логической операции. АЛУ работает во много раз быстрее, чем все остальные устройства, подключенные к компьютерной системе. Это позволяет одному процессору контролировать множество внешних устройств, таких как клавиатуры, дисплеи, магнитные и оптические диски, сенсоры и механические управляющие устройства.

Блок вывода

Функция блока вывода противоположна функции блока ввода: он направляет результаты обработки данных в так называемый внешний мир. Типичным примером устройства вывода является *принтер.* Механизмами печати в принтерах используются ударные механизмы, головки, выпрыскивающие струи чернил, или технология фотокопирования, как в лазерных принтерах.

Некоторые устройства выполняют одновременно и функции устройства ввода и устройства вывода. Это, например, графические дисплеи. Поэтому такие устройства называют устройствами ввода-вывода.

Блок управления (устройство управления)

Работу всех устройств, в том числе устройств процессора нужно как-то координировать. Именно этим и занимается блок управления компьютера, с некоторых пор внесенный непосредственно в процессор. Это, если можно так выразиться, нервный центр компьютера, передающий управляющие сигналы другим устройствам и отслеживающий их состояние.

Управление операциями ввода-вывода осуществляется командами программ, в которых идентифицируются соответствующие устройства ввода-вывода и пересылаемые данные. Однако, реальные *синхронизирующие сигналы* (timing signals), управляющие пересылкой, генерируются управляющими схемами. Синхронизирующие сигналы – это сигналы, определяющие, когда должно быть выполнено данное действие. Кроме того, посредством синхронизирующих сигналов, генерируемых блоком управления, осуществляется передача данных между процессором и памятью. Блок управления можно представить себе как отдельное устройство, взаимодействующее с другими частями машины. Но на практике так бывает редко. Большая часть управляющих схем физически распределена по разным местам компьютера. Сигналы, используемые для синхронизации событий и действий всех устройств, передаются по множеству управляющих линий (проводов).

В целом, функционирование компьютера можно описать следующим образом.

- Компьютер с помощью блока ввода принимает информацию в виде программ и данных и записывает ее в память.
- Хранящаяся в памяти информация под управлением программы пересылается в арифметико-логическое устройство для дальнейшей обработки.
- Данные, полученные в результате обработки информации, направляются на устройства вывода.
- За все действия, производимые внутри машины, отвечает блок управления.

Основные концепции функционирования

Как было отмечено выше, действиями компьютера управляют инструкции. Для выполнения конкретной задачи в память записывается соответствующая программа, состоящая из множества команд. Команды по очереди пересылаются из памяти в процессор, который их выполняет. Данные, используемые в качестве операндов команд, также хранятся в памяти. Пример типичной команды, запрограммированной на языке Ассемблер:

Add LOCA, R0

Эта команда складывает операнд, хранящийся в памяти по адресу LOCA, с операндом, хранящимся в регистре R0 процессора, и помещает результат в этот же регистр. Состояние памяти, после того как из нее команда загрузится в процессор и операнд загрузится в процессор - не изменится, а содержимое регистра R0 после выполнения команды перезапишется. Данная команда выполняется в несколько этапов. Сначала она пересылается из памяти в процессор. Затем операнд команды считывается из памяти по адресу LOCA и складывается с содержимым регистра R0, после чего результирующая сумма записывается в регистр R0.

Пересылка данных между памятью и процессором начинается с отправки в устройство памяти адреса слова, к которому требуется получить доступ, и выдачи соответствующих управляющих сигналов. Затем данные пересылаются в память или из памяти.

На рисунке 1.2 показано, как соединяются между собой память и процессор.

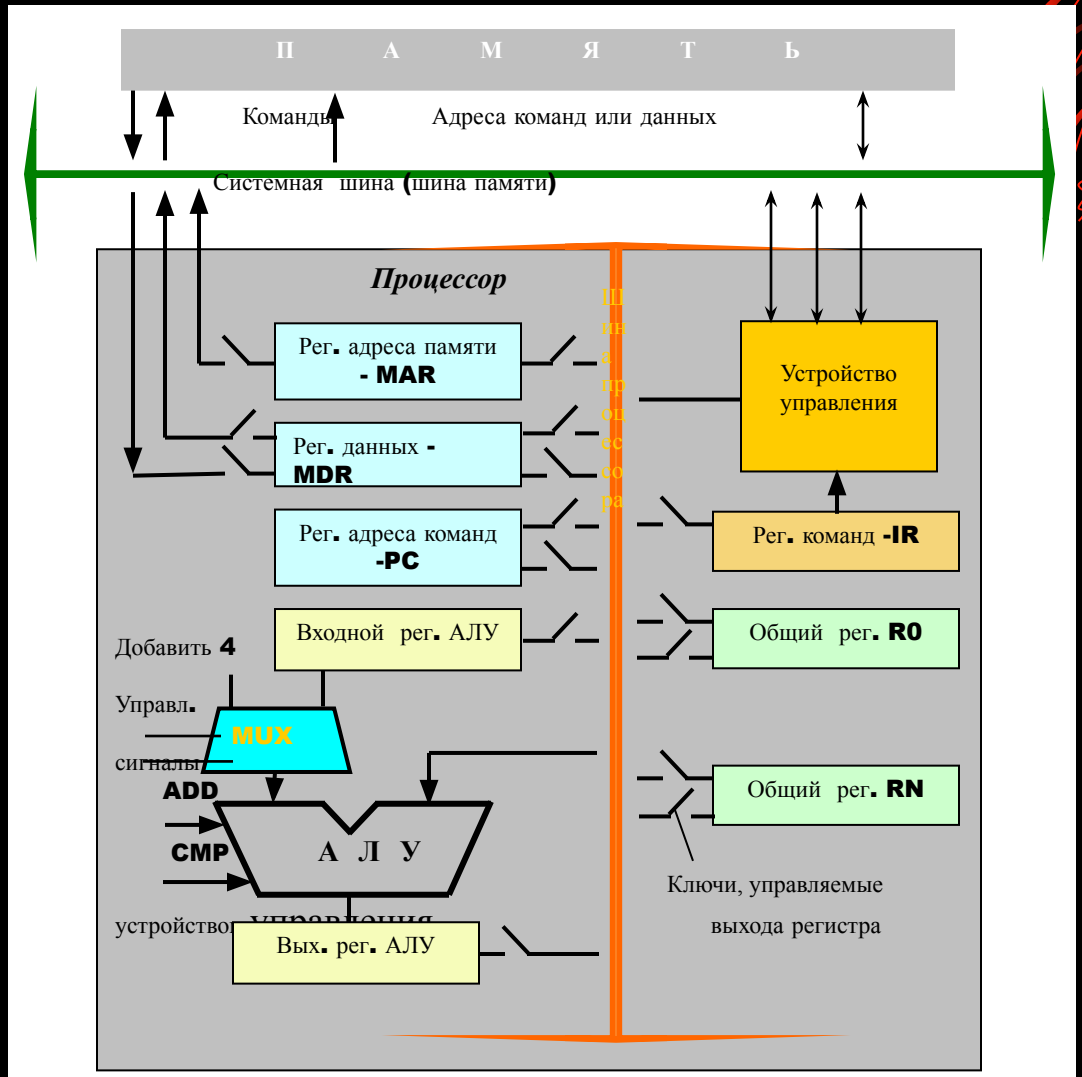


Рис. 2. Базовая структура процессора и схема взаимодействия процессора и памяти

Кроме АЛУ и управляющих схем процессор содержит множество регистров, предназначенных для разных целей В *регистре команды* (Instruction Register, IR) содержится код выполняемой в данный момент команды. Её результат доступен управляющим схемам, которые генерируют сигналы для управления различными элементами, участвующими в выполнении команды. Ещё один специализированный регистр, называемый *счетчиком команд* (Program Counter, PC) , служит для контроля за ходом выполнения программы. В нем содержится *адрес следующей команды*, подлежащей выборке и выполнению. Пока выполняется очередная команда, содержимое регистра PC обновляется – в него записывается адрес следующей команды. Говорят, что регистр PC *указывает* на команду, которая должна быть выбрана из памяти. Кроме этого в процессоре имеется *n регистров* общего назначения, от R_0 до R_{n-1}. Они необходимы для осуществления процесса программирования и для хранения промежуточных результатов и данных в процессе реализации программ.

Ещё два регистра обеспечивают процесс взаимодействия с памятью. Это *регистр адреса* (Memory Address Register, MAR) и *регистр данных* (Memory Data Register, MDR). В регистре MAR содержится адрес, по которому производится обращение к памяти, а в регистре MDR – данные, которые должны быть записаны в память или прочитаны из неё по этому адресу.

Рассмотрим типичный пример процесса выполнения программы компьютером. Программа располагается в памяти, куда обычно попадает через какое-либо устройство ввода Её выполнение начинается с записи в регистр PC адреса первой команды. Содержимое этого регистра пересылается в регистр MAR, а в память направляется управляющий сигнал Read. Когда истекает время, необходимое для доступа к памяти, адресуемое слово (в данном случае – первая команда программы) считывается из памяти и загружается в регистр MDR. Затем содержимое регистра MDR пересылается в регистр IR. Команда готова к декодированию и выполнению.

Если команда требует, чтобы АЛУ выполнило определенную операцию, для неё необходимо получить операнды. Операнд, располагающийся в памяти (он может находиться и в регистре общего назначения), нужно сначала из таковой извлечь, переслав его адрес в регистр MAR и инициализировав цикл Read. После пересылки из памяти в регистр MDR операнд будет направлен в АЛУ. Аналогичным образом туда же будут переданы и остальные необходимые команде операнды, после чего АЛУ сможет выполнить требуемую операцию. Если результат должен быть сохранен в памяти, он будет записан в регистр MDR. Затем адрес, по которому его нужно записать в память, будет помещен в регистр MAR. После чего будет инициирован цикл Write. В какой-то момент в ходе выполнения текущей инструкции содержимое регистра PC увеличивается, и он начинает указывать на следующую подлежащую выполнению инструкцию.

Другими словами, как только завершится выполнение текущей инструкции, можно будет приступить к выборке следующей.

Компьютер не только пересылает данные между памятью и процессором, но и принимает их от входных устройств, а также отсылает выходным устройствам. Поэтому среди машинных команд имеются и команды для выполнения операций ввода-вывода.

Если возникает необходимость срочно обслужить некоторое устройство (например, когда устройство мониторинга в автоматизированном промышленном процессе обнаружит опасную ситуацию), нормальное выполнение программы может быть прервано. Для того чтобы немедленно отреагировать на эту ситуацию, компьютер должен прервать выполнение текущей программы. С этой целью устройство генерирует сигнал прерывания. *Прерывание* (interrupt) – это запрос, поступающий от устройства ввода-вывода, с требованием предоставить ему процессорное время. Для обслуживания этого устройства процессор выполняет соответствующую *программу обработки прерывания*. А поскольку её выполнение может изменить внутреннее состояние процессора, перед обслуживанием прерывания нужно сохранить его состояние в памяти. Обычно в ходе этой операции сохраняется содержимое регистра PC, регистров общего назначения и некоторая управляющая информация. По завершении работы программы обработки прерывания состояние процессора восстанавливается и прерванная программа продолжает свою работу. Процессор со всеми его элементами (рис.1.1) обычно реализуется в виде одной микросхемы, на которой располагается как минимум одно устройство кэш-памяти. Такие микросхемы – чипы – называются VLSI (VLSI – аббревиатура от Very Large Scale Integration, что переводится как очень крупномасштабная интеграция).

Структура шины

Чтобы составить и изготовить действующую систему, необходимо все ее части соединить между собой определенным образом. Способов соединения существует очень много. Мы рассмотрим лишь простейшие и самые распространенные из них.

Компьютер сможет работать с достаточной скоростью лишь при условии, что будет организован таким образом, чтобы полное слово данных обрабатывалось им за указанное время. Когда слово данных пересылается между устройствами, параллельно перемещаются все его биты. Каждый бит пересылается по своему проводу (линии), так что для пересылки слова требуется несколько параллельных линий. Группа линий, образующая соединение между несколькими устройствами, называется *шиной* (bus). Наряду с линиями, по которым пересылаются данные, шина содержит линии для передачи адреса и управляющих сигналов.

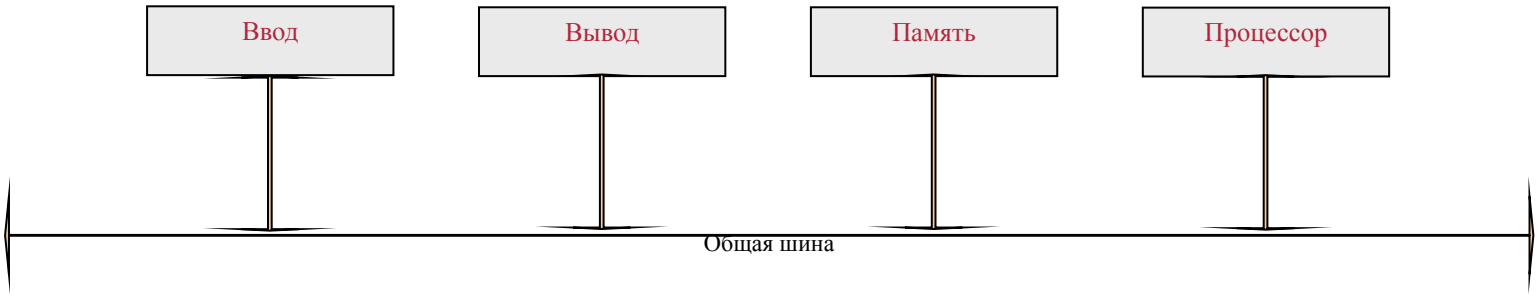


Рис. .3. Архитектура системы с общей шиной

К этой шине подсоединяются все устройства компьютера. Поскольку за один раз по шине может пересылаться только одно слово данных, в каждый конкретный момент шину могут использовать только два устройства. Для организации процесса параллельной обработки нескольких запросов используются линии управления шиной. Главным достоинством архитектуры с общей шиной является ее низкая стоимость и гибкость в отношении подключения периферийных устройств. При наличии в системе нескольких шин возможно одновременное выполнение нескольких операций пересылки данных, благодаря чему такая система работает быстрее, но и стоимость выше.

Подсоединенные к шине устройства могут заметно отличаться друг от друга по скорости функционирования. Некоторые электромеханические устройства, в том числе клавиатуры и принтеры, работают относительно медленно. Значительно выше скорость работы, скажем магнитных и оптических дисков. А память и процессор функционируют со скоростью электронных схем, благодаря чему являются самыми быстрыми частями компьютера. Поскольку все эти три типа устройств могут взаимодействовать между собой через шину, необходим такой механизм пересылки данных, который не ограничивал бы скорость обмена информацией между любыми двумя устройствами скоростью более медленного из них и сглаживал бы разницу в скорости работы процессора, памяти и внешних устройств.

Самый распространенный подход к решению этой задачи основан на использовании *буферных регистров* , которые встраиваются во внешние устройства для хранения получаемой ими информации. Для примера можно рассмотреть процесс передачи кода символа от процессора принтеру. Процессор пересылает данные по шине в буфер принтера. Поскольку буфер представляет собой электронный регистр, пересылка выполняется очень быстро. Когда буфер будет заполнен, принтер начнет печатать, и вмешательство процессора больше не потребуется. Шина и процессор освобождаются для другой работы, которая может выполняться одновременно с печатью символа, хранящегося в буфере принтера. Таким образом, использование буферных регистров сглаживает различия в скорости функционирования процессора, памяти и устройств ввода-вывода и предотвращает блокирование высокоскоростного процессора медленными устройствами на все время выполнения операций ввода-вывода. Процессор может быстро переключаться от одного устройства к другому, обеспечивая их параллельное

Программное обеспечение

Для того, чтобы пользователь мог запустить прикладную программу, в памяти компьютера должно уже содержаться некоторое системное программное обеспечение. *Системное программное обеспечение* – это набор программ, предназначенных для выполнения следующих функций:

- получение и интерпретация команд пользователя;
- ввод и редактирование прикладных программ, их сохранение в файлах на вторичных запоминающих устройствах;
- управление процессом сохранения файлов на вторичных запоминающих устройствах и извлечение их с указанных устройств;
- запуск стандартных прикладных программ, таких как текстовые процессоры, электронные таблицы или игры, с данными, которые предоставляются пользователем;
- управление устройствами ввода-вывода для получения входной информации и вывода выходных данных;
- трансляция исходного кода программ, подготовленных ранее пользователем, в объектные модули, состоящие из машинных команд;
- компоновка пользовательских прикладных программ со стандартными библиотечными подпрограммами (например, выполняющими числовые вычисления) и запуск результирующих программ.

Таким образом, системное программное обеспечение отвечает за координирование всех операций, выполняемых в компьютерной системе. Прикладные программы обычно пишутся на языках программирования высокого уровня, в том числе на C, C++, Java и FORTRAN, позволяющих программисту задать действия, которые должна выполнить программа (скажем математические вычисления или обработку строк текста). Такие операции описываются в формате, не зависящем от компьютера, который будет выполнять программу.

Программисту, использующему язык высокого уровня, не нужно знать машинные команды и особенности их использования. Специальная системная программа, называемая *компилятором*, транслирует программу на языке высокого уровня в программу на машинном языке.

Ещё одна важная системная программа, которой пользуются все программисты, называется текстовым *редактором*. Она предназначена для ввода и редактирования прикладных программ. Пользователь такой программы с помощью клавиатуры вводит и редактирует инструкции исходного текста программы и накапливает их в *файле*. Файл – это просто последовательность буквенно-цифровых символов или двоичных данных, которая сохраняется в памяти ил на вторичном запоминающем устройстве.. К файлу можно обращаться по заданному пользователем имени.

Обратимся к важному для компьютера понятию *операционная система* (ОС). Это набор программ, используемый для управления взаимодействием различных устройств компьютера при выполнении прикладных программ. Компоненты операционной системы отвечают за предоставление прикладным программам ресурсов компьютера – основной памяти и памяти на магнитных дисках, устройств ввода-вывода и т.д.

рассмотрим как функционирует ОС на небольшом примере. Предположим, у нас имеется уже откомпилированная прикладная программа, сохраненная в виде машинных команд на диске. Процесс иллюстрирован линейной диаграммой, показанной на рисунке 1.4.

В течение времени от момента t_0 до момента t_1 одна из программ операционной системы инициирует загрузку прикладной программы с диска в память, дожидается завершения процесса загрузки, а затем передает управление прикладной программе. Аналогичные процессы происходят с момента t_2 до момента t_3 и с момента t_4 до момента t_5, когда операционная система считывает файл данных с диска в основную память и когда она печатает результаты. После момента времени t_5 операционная система может загрузить и выполнить другую прикладную программу.

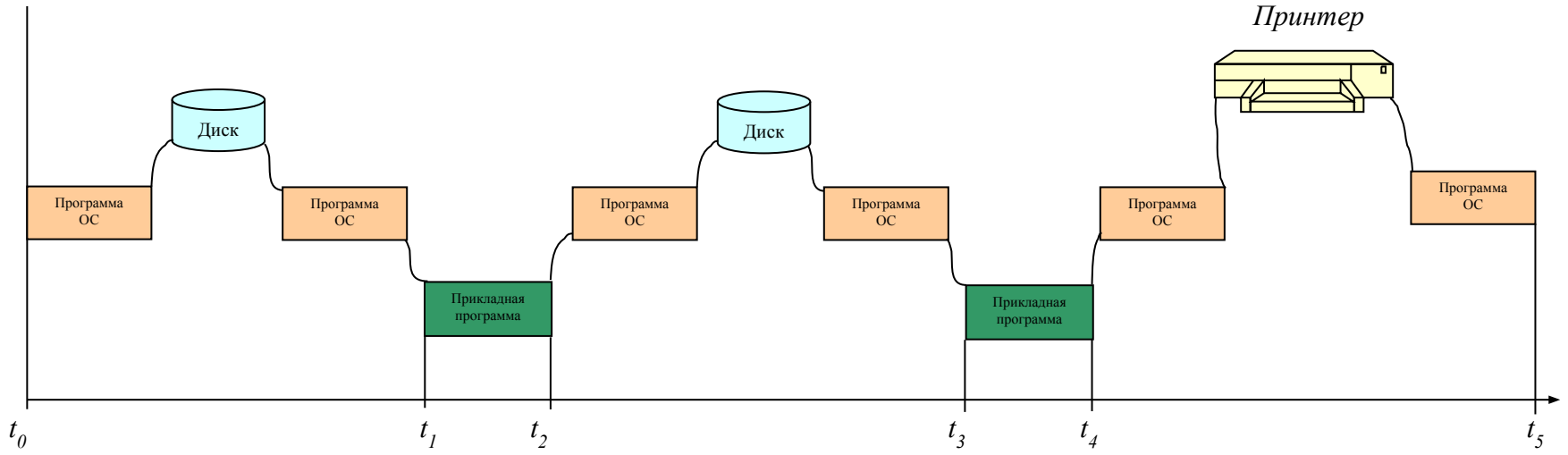


Рис. 4. Поочередное использование процессора прикладной программой и программами операционной системы

При этом имеется возможность более эффективного использования ресурсов компьютера, заключающийся в параллельном выполнении нескольких программ. Например, на отрезке времени работы принтера операционная система могла бы загрузить в память следующую программу. Аналогичным образом, в промежутке времени от t_0 *до* t_1 операционная система могла бы печатать результат, сгенерированные предыдущей программой (пока текущая программа загружается с диска). Именно так операционная система управляет параллельным выполнением нескольких прикладных программ, обеспечивая поочередное использование им ресурсов компьютера. Такая схема параллельного выполнения программ называется *многозадачностью.*

Производительность

Одним из важнейших параметров оценки того, насколько быстро компьютер выподняет программы является *производительность.* Скорость выполнения программы компьютером зависит от конструкции его аппаратного обеспечения и от набора команд машинного языка. Поскольку программы обычно пишутся на языке высокого уровня, производительность зависит от того, насколько удачно компилятор переводит их на машинный язык. Из этого следует, что для достижения максимальной производительности нужно, чтобы компилятор, набор машинных команд и аппаратное обеспечение компьютера имели оптимальную структуру.

Общее время выполнения программы (см.рис. 1.4) $t_5 - t_0$ является мерой производительности всей компьютерной системы. Оно зависит от быстродействия процессора, диска и принтер. Рассматривая производительность процессора, мы должны учитывать только те периоды, в течение которых он активен. На рисунке 1.4 они соответствуют обозначениям «Прикладная программа» и «Программа ОС». Суммарное время выполнения прикладных программ и программ операционной системы называется *процессорным временем*, необходимым для выполнения программы.

Равно как общее время выполнения программы зависит от скорости работы всех устройств компьютерной системы, процессорное время зависит от аппаратного обеспечения, участвующего выполнении конкретных машинных команд. Аппаратное обеспечение включает процессор и память, обычно соединенные шиной (см. рис. 1.3). Архитектура системы с общей шиной, представленная на рисунке 1.5 , отличается от приведенной на рис.1.3 лишь наличием кэш-памяти, включенной в состав процессорного устройства.

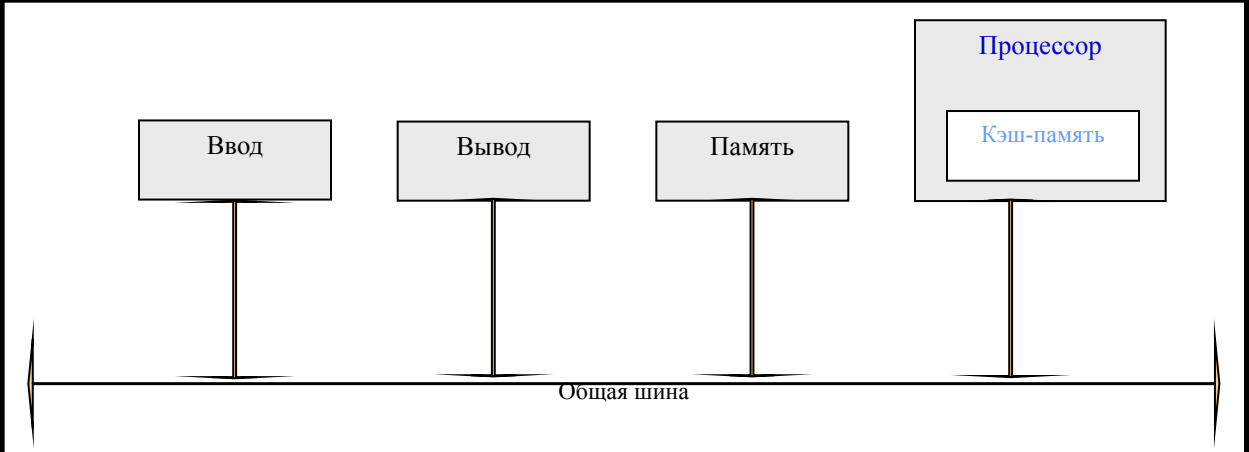


Рис. 5. Расположение кэш-памяти на микросхеме процессора

Какую роль играет кэш-память в производительности компьютера? По мере выполнения программы ее команды по одной выбираются из оперативной памяти и по шине пересылаются в процессор, а их копии помещаются в кэш. Когда для выполнения команды требуются данные, расположенные в основной памяти, они также пересылаются в процессор, а их копии помещаются в кэш. Если позднее та же команда или элемент данных потребуются ещё раз, они будут прочитаны не из основной памяти, а из кэша.

Процессор и относительно небольшая кэш-память могут располагаться на одном интегрированном чипе. Внутренняя скорость выполнения команд таким чипом очень высока – она гораздо выше, чем скорость выборки команд и данных из памяти. Поэтому программа будет выполняться быстрее при условии минимизации количества команд и объема данных, перемещаемых между процессором и основной памятью. Для этого и предназначается кэш процессора. Возьмем, к примеру, многократное выполнение одной и той же группы команд в течение короткого промежутка времени, как это часто бывает в программных циклах. Если эти команды находятся в кэше, их можно быстро извлекать оттуда в течение всего времени их по многу раз повторяющегося выполнения. Сказанное касается и многократно используемых данных.

Частота процессора

Управление процессором осуществляется с помощь сигналов, которые называются *тактовыми импульсами* (clock) и выдаются через фиксированные интервалы времени. Промежуток времени между двумя тактовыми импульсами составляет *тактовый цикл* (clock cycle), или просто *такт*. Для выполнения машинной команды процессор разделяет ее на *последовательность базовых шагов,* каждый из которых может быть выполнен за один такт. Длительность одного тактового цикла T является важнейшим параметром, определяющим производительность процессора. Обратное значение F=1/T, называется *тактовой частотой* (clock rate) процессора, измеряется количеством тактов в секунду и называется *Гц* (Герц). Процессоры современных компьютеров работают на частотах равных миллионам Гигагерц (ГГц).

Формула производительности, определяемой через тактовую частоту

Одним из компонентов общего времени выполнения программы является процессорное время. Предположим, что для реализации программы, написанной на одном из языков высокого уровня, требуется T секунд (с) процессорного времени.

Компилятор генерирует соответствующую объектную программу на машинном языке. Допустим, что для полного выполнения такой программы нужно произвести N команд машинного языка. Значение N – это количество машинных команд, которые будут реально выполнены; оно не обязательно равняется количеству команд в объектной программе. Некоторые команды могут выполняться более одного раза, например, в том случае, если они расположены внутри программного цикла. Другие команды могут вообще не выполняться, что зависит от входных данных.

Предположим, что среднее количество базовых шагов, необходимых для выполнения одной машинной команды, равняется S и, что каждый базовый шаг производится за один такт процессора. Если тактовая частота равна F тактам в секунду, время выполнения программы составит

$$T = \frac{N \times S}{F} \quad секунд$$

Это равенство часто называют *основной формулой вычисления производительности.*

Для пользователя приложения параметр T имеет гораздо большее значение, чем параметры N, S и R. Конструктор компьютера для обеспечения большей производительности должен искать пути уменьшения значения этого параметра, для чего необходимо предельно уменьшить значения N и S и увеличить значение F. Значение N уменьшается, когда исходная программа компилируется в объектную программу с меньшим количеством команд. Значение S уменьшается, когда процесс выполнения команды состоит из меньшего количества базовых шагов или де, если некоторые шаги команд могут выполняться одновременно. С повышением тактовой частоты повышается значение F и сокращается время выполнения базового шага команды.

Важно подчеркнуть, что параметры N, S и R. Отнюдь не являются независимыми друг от друга – изменение одного из них может повлиять на величину другого. Поэтому любое новшество в конструкции процессора повысит производительность компьютера только в том случае, если в результате уменьшится значение параметра T. Процессор с частотой 900 МГц не всегда будет работать быстрее, чем процессор с частотой 700 МГц, поскольку у него может быть другое значение параметра S.

Конвейерная и суперскалярная обработка

Если процессору обеспечить возможность выполнения команд программы не последовательно, а параллельно, производительность процессора значительно повысится. Такая технология называется *конвейерной обработкой* (pipelining).

Еще более высокой степени параллелизма можно достичь путем реализации в процессоре нескольких конвейеров команд. Речь идет об использовании нескольких функциональных блоков, обеспечивающих параллельное выполнение команд. В таком случае на каждый такт может припадать начало сразу нескольких команд. Такой режим функционирования процессора называется *суперскалярным.*

Оценка производительности

Чтобы оценить производительность компьютера, ее необходимо как-то измерить.

Руководствуясь показателем производительности, конструкторы компьютеров обычно оценивают эффективность новых элементов и технологий, производители базируют на них свою маркетинговую политику, а покупатели производят выбор из числа имеющихся в продаже моделей.

Ранее было определено, что наиболее точно определяющим производительность компьютера, является время выполнения программы T. Несмотря на концептуальную простоту формулы вычислить значение T не так-то просто. Поэтому производительность компьютеров принято измерять с помощью тестовых программ. Для того, чтобы можно было сравнивать производительность разных систем, эти программы должны быть стандартизированы. Показателем производительности является время, в течение которого компьютер выполняет заданный тест.

В настоящее время общепринятой практикой является использование некоторого набора специально подобранных реальных прикладных программ. Подбором таких приложений занимается некоммерческая организация под названием System Performance Evaluation Corporation (SPEC). Она публикует списки программ для разных прикладных областей и результаты тестирования многих имеющихся на рынке моделей компьютеров.

В этот список входят самые разнообразные программы, от игр, компиляторов и приложений баз данных до программ, производящих интенсивные вычисления в области астрофизики и квантовой химии. В каждом случае программа компилируется для тестируемого компьютера и измеряется реальное время её выполнения на этом компьютере. Никакая эмуляция не допускается. Та же самая программа компилируется и выполняется на компьютере, выбранном в качестве эталона. Для теста SPEC95 в качестве такового применяется компьютер SUN SPARCstation 10/40, а для теста SPEC2000 – рабочая станция UltraSPARC10 с процессором UltraSPARC –Iii, тактовая частота которого составляет 300 МГц. Коэффициент производительности SPEC вычисляется по следующей формуле:

$$\text{SPEC – коэффициент} = \frac{\text{Время выполнения на эталонном компьютере}}{\text{Время выполнения на тестируемом компьютере}}$$

Таким образом, SPEC – коэффициент 50 указывает на то, что тестируемый компьютер выполняет данный тест в 50 раз быстрее, чем компилятор UltraSPARC10.

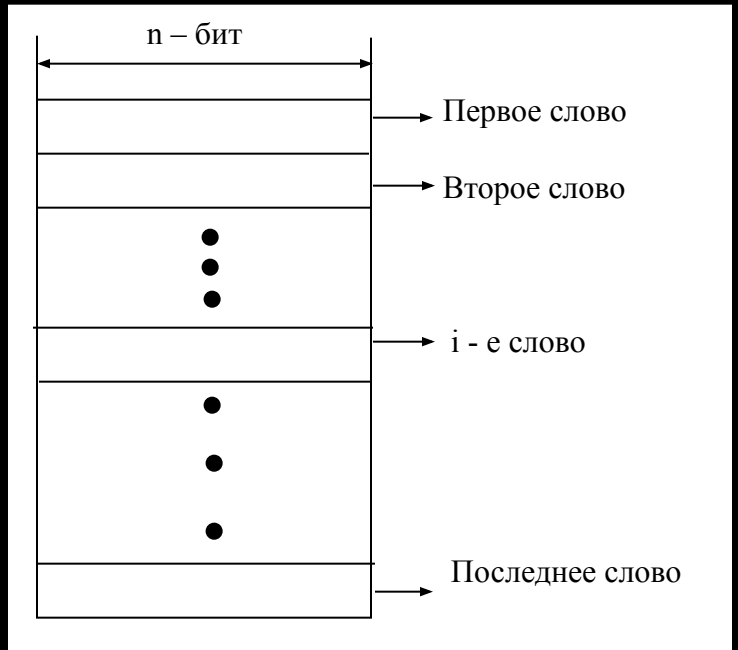
Для проведения полного тестирования по очереди компилируются и выполняются все программы из списка SPEC, а затем вычисляется среднее геометрическое полученных результатов. Итоговый SPEC – коэффициент для конкретного компьютера рассчитывается по формуле:

Символы

Компьютеры должны обрабатывать не только числа, но и текстовую информацию, состоящую из символов. Под термином «символы» подразумеваются буквы алфавита, десятичные цифры, знаки препинания и т.п. Они представляются кодами, обычно имеющими длину 8 бит. Одной из наиболее широко распространенных кодовых таблиц является таблица ASCII *(American Standard Code For Information Interchange)*.

Память и адреса

Числовые и символьные операнды, равно как и команды, хранятся в памяти компьютера. Память состоит из многих миллионов *ячеек,* в каждой из которых содержится один бит информации, имеющий значение 0 или 1. Поскольку один бит способен представить очень маленькое количество информации, биты редко обрабатываются поодиночке. Как правило, их обрабатывают группами фиксированного размера. Для этого память организуется таким образом, что группы по n бит могут записываться и считываться за одну базовую операцию. Группа из n бит называется *словом* информации, а значение n – *длиной слова*. Схематически память компьютера можно представить в виде набора слов (рис. 6)



Длина слова современных компьютеров составляет от 16 до 64 бит. Если длина слова компьютера равна 32 битам, в одном слове может храниться 32-разрядное число в системе дополнения до двух или четыре символа ASCII, занимающих по 8 бит (рис. 7).

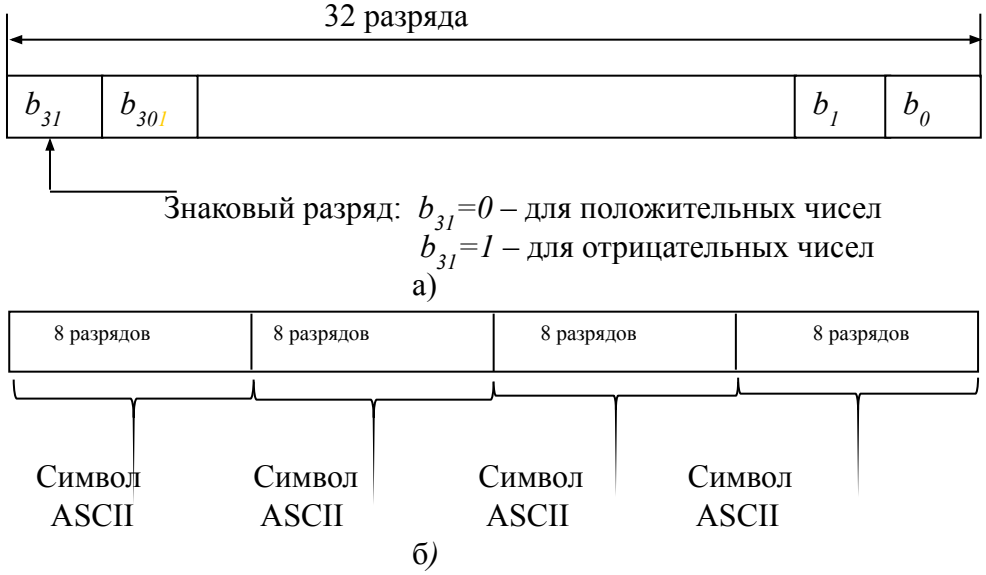


Рис 7. Примеры закодированной информации в 32 – разрядном слове: Целое число со знаком (*а);* четыре символа (*б)*

Восемь идущих подряд битов называются *байтом*. Для представления машинной команды требуется одно или несколько слов.

Для доступа к памяти с целью записи или чтения отдельных элементов информации, будь то слова или байты, необходимы имена или *адреса*, определяющие их расположение в памяти. В качестве адресов традиционно используются числа из диапазона от0 до $2^k - 1$ со значением k, достаточным для адресации всей памяти компьютера. Все 2^k адресов составляют *адресное пространство* компьютера. Следовательно память состоит из 2^k адресуемых элементов. Например, использование 240разрядных адресов позволяет адресовать 2^{24} (16777216) элементов памяти. Обычно это количество адресуемых элементов обозначается как 16 М (16 мега), где 1 М = 2^{20} (1048576). 32-разрядным адресам соответствует адресное пространство из 2^{32} , или 4 Г (гига),

Байтовая адресация

Итак, есть три основные единицы информации: бит, байт и слово. Байт всегда равен 8 битам, а длина слова обычно колеблется от 16 до 64 бит. Отдельные биты, как правило, не адресуются. Чаще всего адреса назначаются байтам памяти. Память, в которой каждый байт имеет отдельный адрес, называется *памятью с байтовой адресацией.* Последовательные байты имеют адреса 0, 1, 2, и т.д. Таким образов, при использовании слов длиной 32 бита последовательные слова имеют адреса 0, 4, 8, …, и каждое слово состоит из 4 байт.

Прямой и обратный порядок байтов

Существует два способа адресации байтов в словах, а именно в прямом и обратном порядке (рисунок 8). *Обратным порядком байтов* называется система адресации, при которой байты адресуются слева направо, так, что самый старший байт слова (расположенный с левого края) имеет наименьший адрес. *Прямым порядком байтов* называется противоположная система адресации, при которой байты адресуются справа налево, так что наименьший адрес имеет самый младший байт слова (расположенный с правого края).Слова «старший» и «младший» определяют вес бита, то есть степень двойки, соответствующей данному биту, когда слово представляет число.

В машинах для коммерческих расчетов используются обе системы адресации. В обеих этих системах адреса байтов 0, 4, 8, и т.д. применяются в качестве адресов последовательных слов памяти в операциях чтения и записи слов

Наряду с порядком байтов в слове важно также определить порядок битов в байте типичный способ расположения битов показан на рисунке 7, *а*. Это наиболее естественный порядок битов для кодирования числовых данных, непосредственно соответствующий их разрядам Однако существуют компьютеры, для которых характерен обратный порядок битов.

Расположение слов в памяти

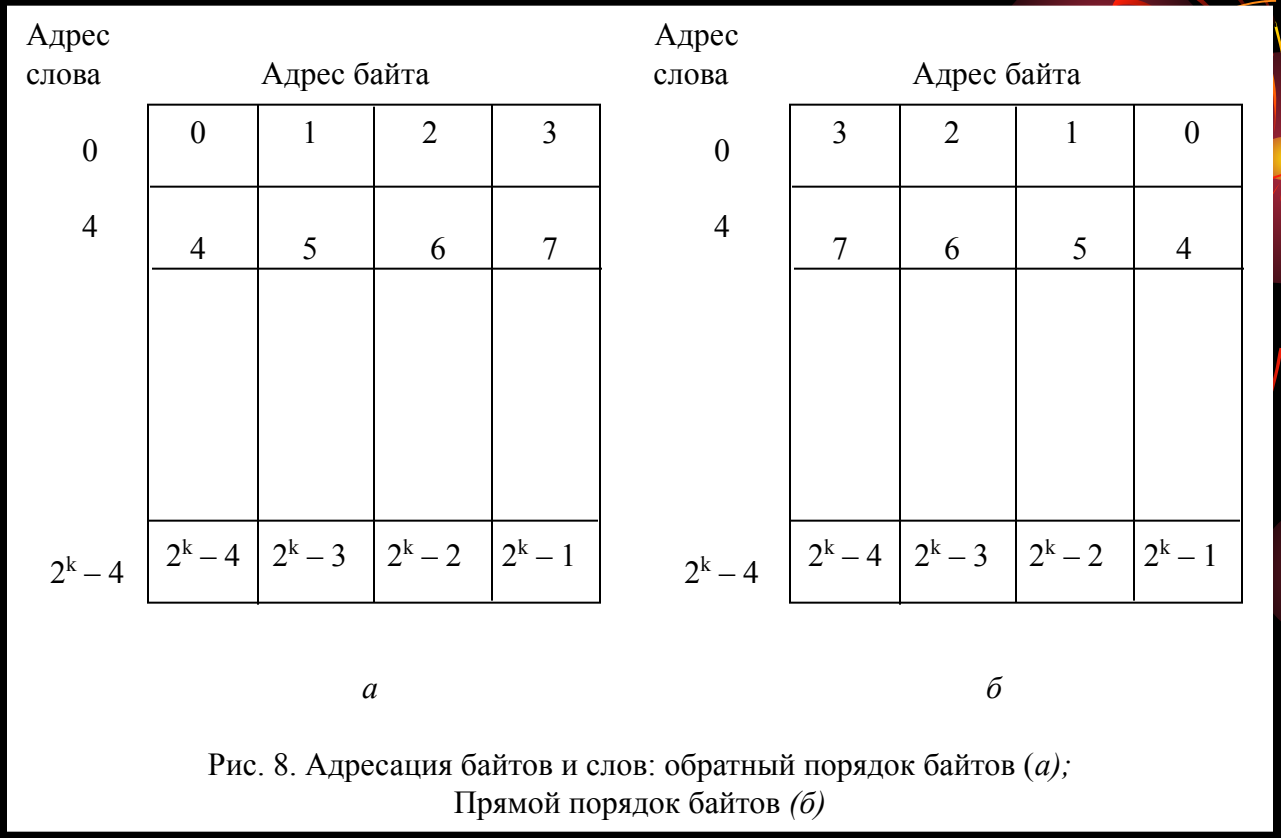


Рис. 8. Адресация байтов и слов: обратный порядок байтов (*а);*
Прямой порядок байтов *(б)*

В случае 32-разрядных слов их естественные границы располагаются по адресам 0, 4, 8 и т.д. (рис.2.8). При этом говорят, что слова выровнены по адресам в памяти. Если говорить в общем, слова считаются выровненными в памяти в том случае, если адрес начала каждого слова кратен количеству байтов в нем. По практическим причинам связанным манипулированием двоично-кодированными адресами, количество байтов в слове обычно является степенью двойки. Поэтому, если длина слова равна 16 (2байтам), выровненные слова начинаются по байтовым адреса 0, 2, 4, …, а если она равна 64 (2^3 байтам), то выровненные слова начинаются по байтовым адресам 0, 8, 16, … .

Слова, начинающиеся с произвольных адресов называются невыровненными.

Базовые типы команд

Сложение двух чисел относится к числу фундаментальных операций любого компьютера. Инструкция (команда) C=A + B

В программе на языке высокого уровня – это команда компьютеру сложить текущие значения двух переменных, A и B, и присвоить их сумму третьей переменной, C. При *компиляции* программы, содержащей эту инструкцию, переменным A, B и C назначаются конкретные адреса памяти. Содержимое памяти по этим адресам представляет значения трех переменных. Поэтому приведенная выше инструкция на языке высокого уровня требует выполнения компьютером следующего действия:

$$C \longleftarrow [A] + [B]$$

Для выполнения этого действия содержимое памяти по адресам A и B должно быть переслано в процессор, где будет вычислена сумма. Полученная сумма должна быть отправлена обратно в память и записана по адресу C.

Для начала предположим, что действие может быть выполнено посредством одной машинной команды. Эта команда содержит адреса трех операндов: A, B и C. Структурно такую команду назвали *трехадресной*, а символически её можно представить как

Add A,B,C

Переменные: A, B и C называют *операндами.* Операнды A и B называются *исходными операндами,* а C – *операндом назначения* или *результирующим операндом.* В общем случае команда этого типа имеет такой формат:

Операция Источник 1, Источник 2, Место назначения

Если для указания адреса одного операнда в памяти необходимо k бит, в закодированной форме данной инструкции для адресов должно быть отведено 3k бит и ещё сколько-то бит для кода самой операции Add. В случае современного процессора с 32-разрядным адресным пространством трехадресная команда слишком громоздка для одного слова разумной длины. Поэтому для представления команд такого типа обычно используется формат длиной в несколько слов.

Для выполнения этой же задачи в качестве альтернативы можно использовать несколько более простых команд, с одним-двумя операндами. Предположим, что процессором поддерживаются двухадресные команды в виде:

Операция Источник, Место назначения

Команда Add такого типа Add A,B

лнять операцию B [A] + [B]. После вычисления суммы результат будет переслан обратно в память и сохранен по адр
еной исходных данных, хранящихся по этому адресу. Это означает, что B является и исходным и результирующим
дом команды.

шения задачи с изначальной формулировкой двухадресной команды недостаточно. Потребуется ещё одна двухадресная
а, которая копирует значение из одного места памяти в другое. Вот она:

Move B,C

манда выполняет операцию C [B], оставляя содержимое памяти по адресу B неизменным. Слово Move, означает
щение.

ация может быть выполнена и в другой комбинации двух команд:

Move B,C
Add A,C

ех приведенных выше командах первыми задаются исходные операнды, после них – операнд назначения. Этот порядок
ерен для выражений на языке ассемблера, используемых в машинных командах многих компьютеров. Но существует
очно много компьютеров, в которых порядок операндов обратный.

определены двух- и трехадресные команды. Но даже двухадресные команды далеко не всегда помещаются в одно слово
. Поэтому используют команды, имеющие всего один операнд. При этом всегда предполагается, что второй операнд
тся в конкретном известном месте. Обычно в качестве такого «оговоренного» места служит регистр процессора, называе
ором (accumulator). Таким образом, *одноадресная* команда Add A означает следующее: добавить содержимое памяти по
A к содержимому сумматора и поместить результат в сумматор.

ввести ещё две команды, такие как:

Load A и Store A,

цию C ← [A] + [B] можно выполнить следующим образом:

Load A
Add B
Store C

ь в сумматор содержимое памяти по адресу A, выполнить сложение A с B и поместить результат в сумматор, из которо
ой Store сохранить в памяти по адресу C

едует обратить внимание на то, что в зависимости от типа команды её операнд может служить либо источником данных, о их приемником. Например, в команде Store адрес С определяет приемник данных, а источником является сумматор. А анде Load, напротив, задается источник данных, а приемником является сумматор.

некоторых старых компьютерах сумматор был единственным регистром. В современных компьютерах регистров общего значения обычно довольно много – от8 до 32, а иногда и больше. Доступ к данным в этих регистрах осуществляется намно трее, чем доступ к памяти, поскольку, как уже было сказано, регистры располагаются внутри процессора и чтобы соваться к ним, достаточно всего несколько битов. Так, для задания одного из 32 регистров нужно 5 битов. Это гораздо ьше того количества битов, которое необходимо для задания адреса в памяти. А поскольку регистры позволяют быстрее абатывать результаты и применять более короткие команды, они широко используются для временного хранения абатываемых данных.

сть R_i – это регистр общего назначения. Команды

Load A, R_i

Store R_i ,A

Add A, R_i

ляют собой пример команд Load, Store и Add, в которых R_i выполняет функцию сумматора. Но даже в том случае, когда анде явно задается единственный адрес, она все равно может занимать больше одного слова памяти.

и в процессоре имеется несколько регистров общего назначения, он может поддерживать ряд команд, все операнды рых располагаются в регистрах. Во многих современных процессорах вычисления фактически выполняются осредственно только над данными, хранящимися в регистрах. К командам такого типа относятся

Add R_i , R_j

Add R_i , R_j, R_k

этих командах исходными операндами является содержимое регистров R_i , R_j. В первой команде регистр Rj служит и емником данных, тогда как во второй роль приемника играет третий регистр, R_k . Такие команды, содержащие лишь имен стров, обычно умещаются в одно слово.

рограммах часто требуется переслать данные из одного места в другое. Для этого используется команда

Move Источник, Приемник

а помещает в приемник копию содержимого источника. Команда Move может использоваться для пересылки данных из
в регистр процессора и из регистра процессора в память вместо команд Load и Store, поскольку направление пересыл
адается просто порядком операндов. Поэтому команда

Move A,R_i

же, что и команда

Load A,R_i
Move R_i , A

и команда

Store R_i , A

образом, при использовании регистров команды можно реализовывать следующим образом:

Move A, R_i
Add B, R_i
Move R_i ,C

выполнения конкретной команды зависит от того, насколько быстро команды пересылаются из памяти в процессор и
ько быстро осуществляется доступ к операндам этих команд. Операции, в которых участвует память, выполняются
медленнее, чем операции с участием регистров. Поэтому значительного ускорения работы можно добиться в тех
х, когда несколько операций подряд выполняются над хранящимися в регистрах данными без обращения к памяти. При
яции программ, написанных на языке высокого уровня, в машинный язык важно минимизировать частоту перемещений
между памятью и регистрами процессора.

Выполнение команд и линейный код

сунке 9 показан фрагмент программы, выполняющий команду C [A] + [B] в той последовательности, в какой он
авлен в памяти компьютера. При этом предполагается, что длина слова составляет 32 разряда, а память адресуется
ово. Три команды программы расположены в следующих друг за другом словах, начиная с адреса i . Поскольку каждая
а занимает 4 байта, вторая и третья команды начинаются по адресам i + 4 и i + 8. Чтобы упростить задачу, предполагае
ный адрес целиком задается в команде, занимающей одно слово, хотя при таких размерах адресного пространства и той
слова, которая поддерживается современными процессорами, подобное, как правило, невозможно.

Чтобы программа начала выполняться адрес ее первой команды (в нашем примере i) должен быть помещен в регис
PC. Затем управляющие схемы процессора будут использовать
Информацию из этого регистра для выполнения последовательного
расположения в памяти команд, по одной за раз, в порядке увеличения
адресов. Этот процесс называется *последовательным выполнением*
программы, а сама программа, представленная в памяти как список
команд, называется *линейной.* В ходе выполнения каждой команды
Адрес, хранящийся в регистре PC, увеличивается на 4, чтобы этот регистр
Указывал на следующую команду программы. Таким образом, после
Выполнения команды Move, расположенной по адресу i + 8, регистр PC
Будет содержать значение i + 12, указывающее на первую команду
Следующего фрагмента программы.

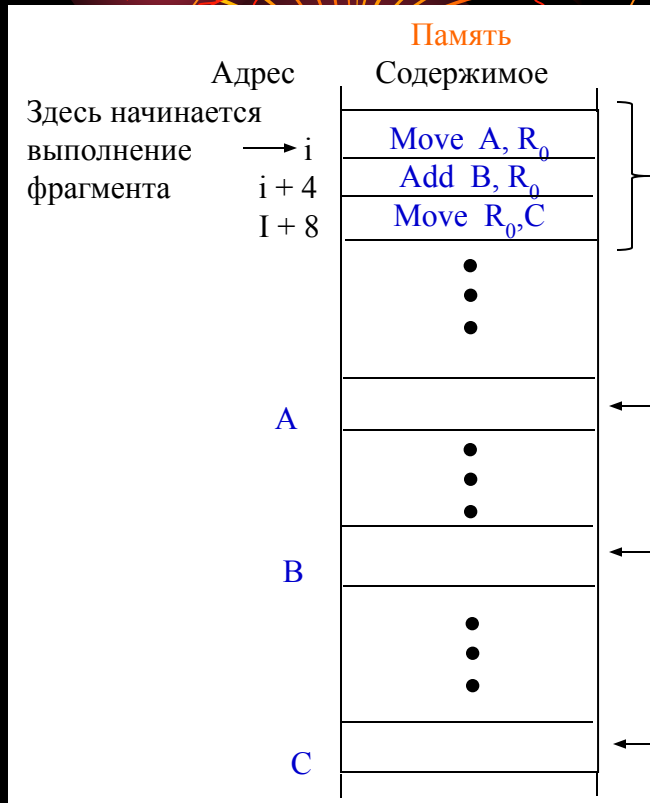


Рис.9. Расположение фрагмента программы в
операции C [A] + [

Выполнение команды производится в два этапа. На первом этапе,
называемом *фазой выборки* команды, из памяти извлекается команда,
хранящаяся по адресу, указанному в регистре PC. Эта команда помещается
В *регистр команды* (Instruction Register, IR). После этого начинается
второй этап, называемый *фазой выполнения* команды. На этом этапе
процессор анализирует команду в регистре IR, чтобы узнать, какое
Действие ему следует выполнить. Затем он производит это действие
, для чего обычно требуется выбрать операнды из памяти или из регистров,
Выполнить арифметическую или логическую операцию и сохранить
результаты по указанному адресу. В ходе этой двухфазовой процедуры содержимое регистра PC в определенный момен
образом, чтобы он указывал на следующую команду. Когда выполнение команды завершается, регистр PC содержи
команды, то есть можно начинать этап выборки новой команды. В большинстве процессоров фаза выполнения дел
на несколько фаз, соответствующих выборке операндов, выполнению операции и сохранению результатов.

Ветвление

На рисунке 10 представлена программ сложения n чисел, адреса которых в памяти символически обозначены как NUM1, NUM2, …, NUMn. Для сложения каждого из этих чисел с содержимым регистра R0 может использоваться отдельная команда Add. После сложения всех чисел результат может быть записан в память по адресу SUM.

Но вместо использования возможно длинного списка команд Add, можно использовать всего одну команду в программном цикле.

Цикл – это последовательность команд, выполняемых необходимое количество раз. В нашей программе цикл начинается по адресу LOOP и заканчивается командой Branch>0. На каждом шаге этого цикла определяется адрес следующего числа (Add #4,R2) и это число извлекается из памяти и добавляется к содержимому регистра R0. Регистр R1 используется в качестве счетчика, определяющего количество повторений цикла. Поэтому в начале программы в регистр R1 загружается содержимое памяти, находящееся по адресу N. В теле Цикла команда Decrement R1 уменьшает значение R1 на единицу. (Аналогичная операция выполняется командой Increment, которая увеличивает свой операнд на единицу). Тело цикла выполняется раз за разом до тех пор, пока результат операции Decrement остается большим нуля.

Введем ещё одно понятие и определение – *команда перехода*. При наступлении момента выполнения такой команды, в регистр адреса команд PC процессора загружается адрес той команды, которая должна быть выполнена безаппеляционно. Процессор выбирает из памяти указанную адресом перехода команду и выполняет её.

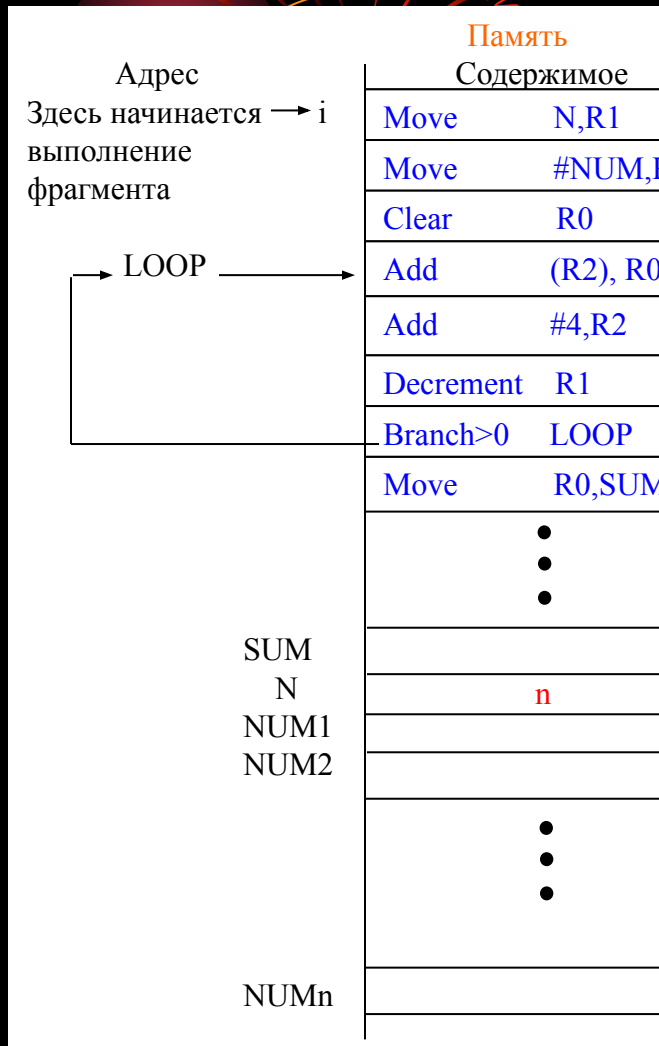


Рис.10. Иллюстрация нелинейного выполнен цикла

ет место определение *команды условного перехода,* при котором переход в нужную часть программы осуществляется только при безусловном выполнении некоторого условия. Если условие не выполняется, адреса в регистре счетчика кома меняются обычным последовательным очередным образом и выполняется очередная команда в порядке увеличения адре

В программе на рисунке 10, команда Branch>0 LOOP (переход, если больше нуля) является командой условного перехо выполняющей перемещение по адресу LOOP в том случае, если в результате выполнения предыдущей команды (инструкции), значение в регистре R1 остается больше нуля. Это означает, что цикл повторяется до тех пор, пока в списк чисел остаются необработанные элементы, которые следует добавить к содержимому регистра R0. В конце n-го прохода циклу команда Decrement возвращает значение 0, поэтому переход на начало цикла не осуществляется, а выполняется следующая по порядку команда – Move. Команда Move перемещает окончательный результат суммирования из регистра в память по адресу SUM.

Проверка условия с последующим выбором одного из нескольких альтернативных путей реализации программы, называемая *ветвлением,* выполняется гораздо чаще, чем просто программные циклы. Такая возможность имеется в системах команд любых компьютеров, поскольку ветвления и циклы относятся к числу фундаментальных операций, без которых невозможно запрограммировать сколько-нибудь нетривиальную задачу.

Флаги кодов условий

цессор отслеживает выполнения различных операций и сохраняет их для использования в последующих инструкциях условного перехода. Эту информацию он записывает в специальные биты, называемые *флагами кодов условий.* В зависимости от результата выполненной операции отдельные флаги устанавливаются в 1 или 0. к наиболее распространенным относят четыре флага:

N (negative) – устанавливается в 1, если результат отрицателен; в противном случае очищается (то есть устанавливается 0);

Z (zero) – устанавливается в 1, если результат равен 0; в противном случае очищается;

V (overflow) – устанавливается в 1, если в результате арифметической операции произошло переполнение; в противном случае очищается;

C (carry) – устанавливается в1, если в ходе операции был выпонен перенос; в противном случае очищается.

Флаги N и Z указывают, является результат арифметической операции отрицательным или нулевым. Кроме арифметических команд на эти флаги могут воздействовать команды, выполняющие пересылку данных, такие как Move, Load и Store, благодаря чему ветвление может быть выполнено с учетом знака и значения перемещенного операнда. В некоторых компьютерах имеется также специальная команда Test, анализирующая значение в регистре или в памяти компьютера и устанавливающая либо очищающая флаги N и Z в соответствии с этими значениями. Флаг V указывает на то, что произошло переполнение, которое происходит, когда результат арифметической операции превышает значение, которое можно представить с помощью количества битов, выделенного операнду. Процессор устанавливает флаг V для того, чтобы программист мог определить, что произошло переполнение, и перейти к подпрограмме, способной исправить данную ошибку. Для этой цели используются команды, подобные команде BranchIfOverflow. Кроме того, в результате установки бита V может автоматически выполняться программное прерывание, позволяющее решить эту проблему средствами самой операционной системы.

Флаг C устанавливается в 1, если в ходе арифметической операции осуществляется перенос из позиции

Режимы адресации

Если говорить в общем, каждая программа обрабатывает данные, хранящиеся в памяти компьютера. Эти данные могут быть организованы разными способами. Для представления данных , используемых в вычислениях, программисты обычно применяют стандартные способы их организации, называемые *структурами данных.* Примерами структур данных могут служить списки, связные списки, массивы, очереди и т.д.

Программы, как правило, пишутся на языках высокого уровня, позволяющих программистам оперировать константами, глобальными и локальными переменными, указателями и массивами. В процессе трансляции такой программы с языка высокого уровня на язык ассемблера компилятор (транслятор) должен реализовать эти конструкции средствами, предоставляемыми набором команд того компьютера, на котором будет выполняться программа. При этом используются различные способы задания местоположения операндов команд, называемые *режимами адресации*. Далее рассмотрим наиболее важные режимы адресации, поддерживаемые современными процессорами, список которых приведен в нижеследующей таблице. В таблице EA – это исполнительный адрес (effective address). Значение – это число со знаком

Адресация	Синтаксис на языке ассемблера	Описание
Непосредственная (immediate)	# Значение	Операнд = Значение
Регистровая (register)	Ri	EA = Ri
Абсолютная, прямая (absolute, direct)	LOC	EA = LOC
Косвенная (indirect)	(Ri) (LOC)	EA = [Ri] EA = [LOC]
Индексная (index)	X(Ri)	EA = [Ri] + X
Базовая индексная (base, index)	(Ri,Rj)	EA = [Ri] + [Rj]
Базовая индексная со смещением (base, index, offset)	X(Ri,Rj)	EA = [Ri] + [Rj] + X
Относительная (relative)	X(PC)	EA = [PC] + X
Автоинкрементная	$(Ri)^+$	EA = [Ri]

Реализация переменных и констант

Переменные и константы – это простейшие виды данных, используемые в любой компьютерной программе. Для создания переменной в языке ассемблера резервируется регистр или место в памяти, где будет хранится её значение. В дальнейшем это значение может изменяться с помощью соответствующих команд.

Ранее в примерах, мы обращались к операндам по именам регистров или по их адресам в памяти. Приведем точное определение этих двух режимов адресации.

Регистровая адресация – это режим, в котором операнд является содержимым регистра процессора; в команде задается имя (адрес) регистра.

Абсолютная адресация – это режим, в котором операнд хранится в памяти; его адрес в памяти задается непосредственно в команде. (В некоторых языках ассемблера данный режим называется *прямым*). В следующей команде, например, используются оба названных режима адресации:

Move LOC,R2

Если для временного хранения данных задействуются регистры процессора, режим адресации этих данных называется регистровым. Абсолютный режим может использоваться для представления глобальных переменных программы. Например, встретив объявление

Integer A,B;

в программе на языке высокого уровня, компилятор выделит адреса памяти для переменных A и B. Теперь, если в программе будут встречаться ссылки на указанные переменные, компилятор сможет генерировать команды на языке ассемблера с абсолютными адресами этих переменных.

Что касается констант, то их адреса и данные могут быть представлены на языке ассемблера с использованием непосредственной адресации.

Непосредственная адресация – это режим, в котором операнд задается в команде явно. В языке ассемблера для обозначения непосредственной адресации используется символ « # » перед значением, применяемым в качестве непосредственного операнда.

Так команда Move #200,R0 помещает значение 200 в регистр процессора R0. Значения констант часто используются в программах на языках высокого уровня, как, например, A= B + 6 или языком ассемблера Add #6,R1.

Косвенная адресация и указатели

При эти режимах адресации операнд и его адрес не задаются прямо в команде. В команде лишь содержится информация, на основании которой можно определить адрес операнда. Этот реальный адрес называется *исполнительным адресом* операнда. Способ назван *косвенным режимом адресации*, при котором исполнительный адрес операнда находится в регистре или в памяти по адресу, заданному в команде. Для обозначения того, что используется именно косвенная адресация, задаваемое в команде имя регистра или адрес памяти заключаются в скобки, как показано на рисунке 11 и в таблице

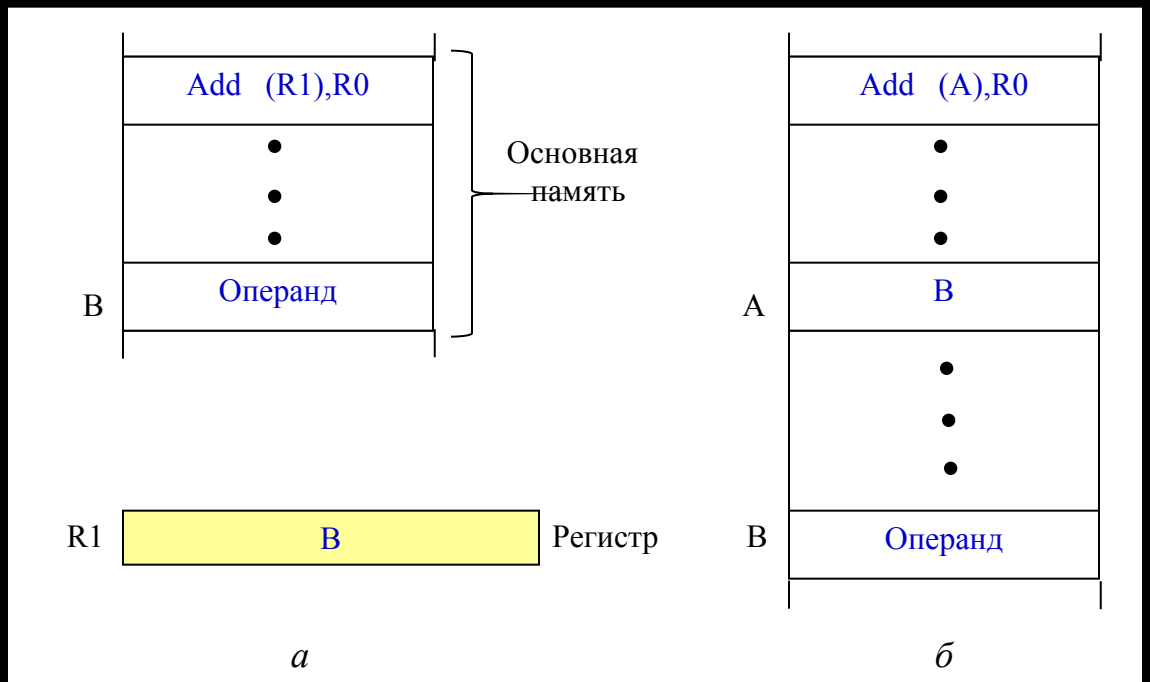


Рис. 11. Косвенная адресация: через регистр общего назначения (а); через значение в памяти (б)

Для выполнения команды Add, схематически представленной на рисунке 11, а, процессор использует в качестве исполнительного адреса операнда значение B, хранящееся в регистре R1. Он запрашивает из памяти значение, хранящееся по адресу B. Прочитанное значение и является операндом команды, который прибавляется к содержимому регистра R0. Как показано на рисунке 11,*б,* возможен и другой вариант косвенной адресации, при котором адрес операнда хранится не в регистре, а в памяти. В таком случае процессор сначала считывает содержимое памяти оп адресу A, а затем запрашивает вторую операцию чтения, используя в качестве адреса значение B.

И регистр, и адрес в памяти, содержащий адрес операнда, называются *указателями.* Косвенная адресация и использование указателей являются очень важным и исключительно мощными концепциями программирования. Их суть можно понять, проведя аналогию с процессом поиска сокровищ. Предположим, вам, чтобы найти таковые, велено идти в дом, распложенный по определенному адресу. Но вместо того, чтобы обнаружить там сокровище, вы находите записку, в которой указан совершенно другой адрес. Заменив одну записку другой, можно изменить адрес сокровища, но исходное указание (команда) останется неизменным. Изменение содержимого записки эквивалентно изменению содержимого указателя в компьютерной программе. Например, если изменить на рисунке 11 содержимое регистра R1 или памяти по адресу A , команда Add получит для сложения другой операнд.

Вернемся к рисунку 10, иллюстрирующему сложение последовательности чисел. Для доступа к последовательным числам, хранящимся в этом списке, мы можем применить косвенную адресацию, в результате чего получим программу, представленную на рисунке10. Регистр R2 используется в качестве указателя на числа в списке, и доступ к операндам осуществляется косвенно, через этот регистр. В программе, в разделе инициализации, из памяти по адресу N в регистр R1 загружается значение счетчика n. Затем, в режиме прямой адресации, адрес первого элемента в списке, NUM1, помещается в регистр R2. После этого очищается регистр R0 (то есть ему присваивается значение 0). На первом шаге цикла команда Add (R2),R0 извлекает из памяти операнд, хранящийся по адресу NUM1, и прибавляет его к содержимому R0. Вторая команда Add увеличивает значение указателя R2 на 4, чтобы на втором шаге цикла, когда предыдущая команда будет выполняться во второй раз, в не содержался адрес NUM2.

Из разновидностей косвенной адресации наиболее широкое применение получила косвенная регистровая адресация (рис.10). Нерегистровая косвенная адресация имеет ограниченное применение из-за плохой совместимости с режимом конвейерной обработки.

Индексация и массивы

Режим адресации, который будет рассматриваться сейчас считается наиболее полезным для работы с массивами и списками.

Индексная адресация – это режим адресации, при котором исполнительный адрес операнда генерируется путем добавления заданного значения к содержимому регистра.

При индексной адресации можно использовать как специально предназначенный для этого регистр, так и один из регистров общего назначения. В любом случае речь идет об *индексном регистре*. Символическое обозначение индексного режима адресации таково:

X (Ri),

где X представляет заданное в команде значение константы, а Ri – имя регистра. Исполнительный адрес операнда этой команды вычисляется так:

EA = X + [Ri]

В процессе формирования исполнительного адреса операнда содержимое индексного регистра не меняется.

В программе на языке ассемблера константа X может быть задана либо явно в виде числа, либо в виде символического имени, представляющего числовое значение. Когда команда транслируется в машинный код , в неё включается непосредственное значение константы X, обычно представленное меньшим количеством битов, чем слово компьютера. Поскольку константа X является целым числом со знаком, то прежде чем она будет сложена с содержимым регистра, ее знак должен быть расширен до длины этого регистра. Существует два способа применения индексного режима адресации. На рисунке 12, *а* индексный регистр R1 содержит адрес в памяти компьютера, а значение X определяет *смещение* операнда относительно этого адреса. Альтернативный способ применения индексной адресации продемонстрирован на рисунке 12, *б*. Здесь константа X соответствует адресу в памяти, а содержимое индексного регистра определяет смещение операнда относительно данного адреса. В любом из этих двух случаев исполнительный адрес является суммой пары значений, одно из которых явно задается в команде, а другое хранится в регистре.

Чтобы понять, чем хороша индексная адресация, достаточно рассмотреть простой пример со списком оценок, полученных студентами по некоторому предмету. Предположим, что это список начинается по адресу LIST и организован так, как показано на рисунке 13.

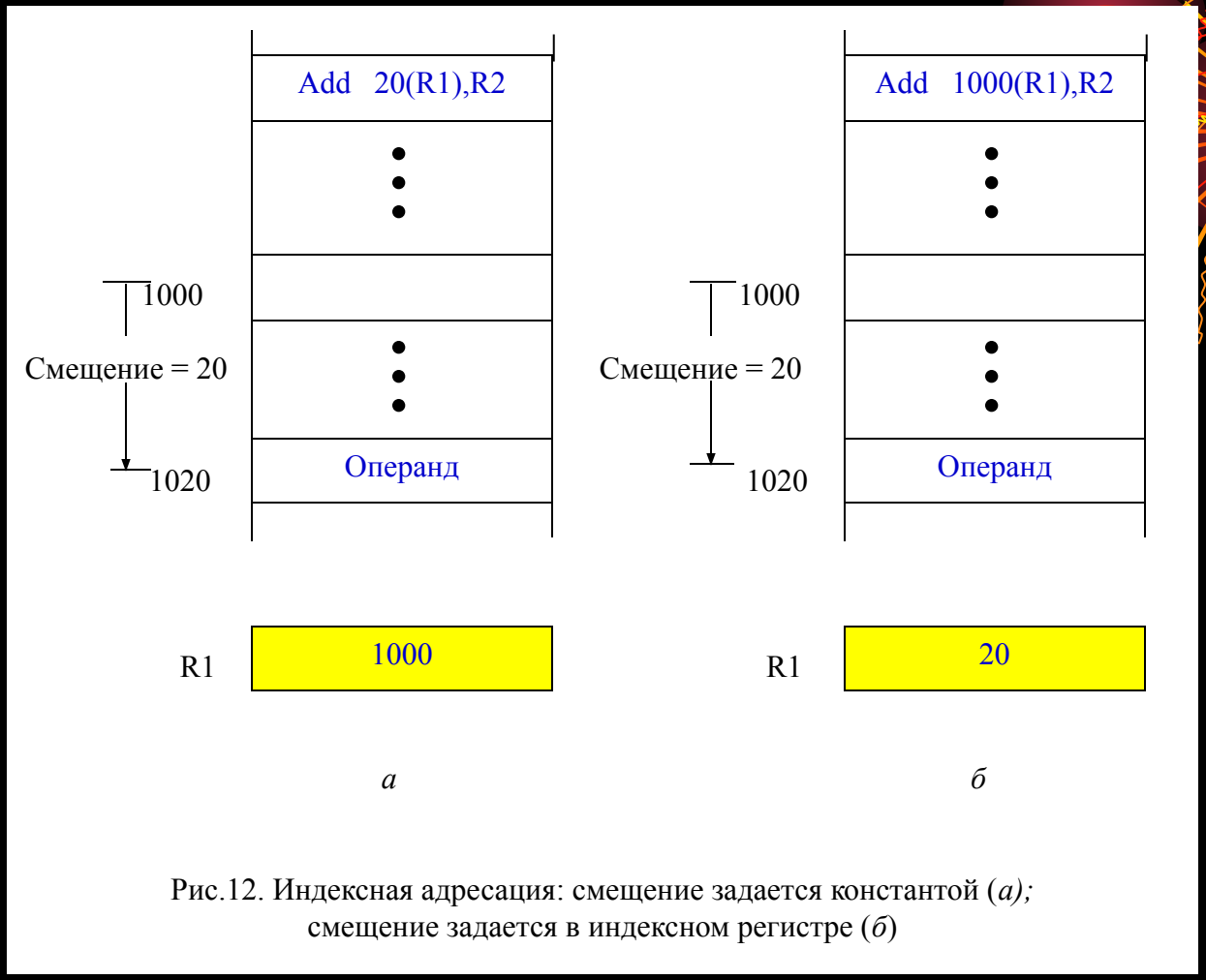
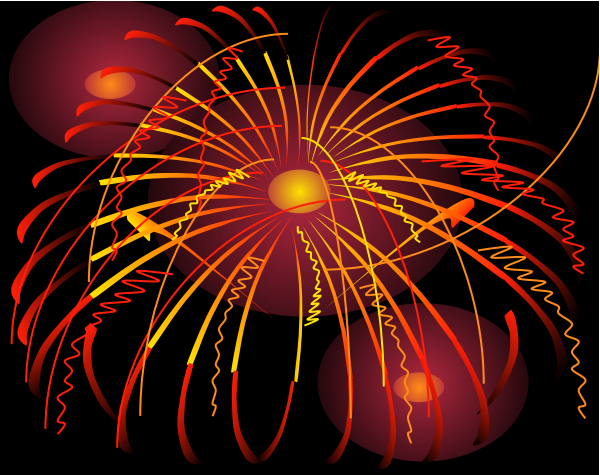


Рис.12. Индексная адресация: смещение задается константой (*а);* смещение задается в индексном регистре (*б*)

Информация о каждом из студентов хранится в памяти в виде записи, занимающей блок из четырех слов. Запись состоит из кода студента, за которым следуют оценки, полученные им в результате проведения трех тестов. Всего в группе n студентов, и значение это хранится в памяти по адресу N непосредственно перед списком. Адреса и коды студентов на рисунке указаны исходя из предположения, что память адресуется побайтово и длина слова составляет 32 разряда..

Следует отметить, что список на рис.13 представляет собой двумерный массив, содержащий n строк и четыре столбца.

В каждой такой строке содержится запись об одном Студенте, а в столбцах указываются коды студентов и их оценки. Если стоит задача подсчитать суммы баллов, полученных всеми студентами по каждому из трех тестов и записать эти три суммы в память по адресам SUM1, SUM2 и SUM3, можно в качестве одного из вариантов использовать нижеприведенную программу (рис.14)



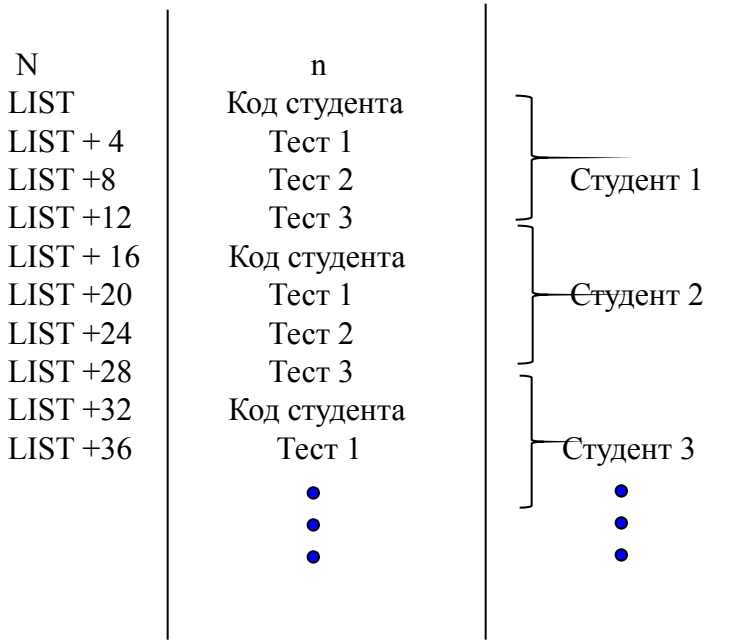


Рис.13. Список оценок студентов

```
        Move    #LIST,R0
        Clear   R1
        Clear   R2
        Clear   R3
        Move    N,R4
LOOP    Add     4(R0),R1
        Add     8(R0),R2
        Add     12(R0),R3
        Add     #16,R0
        Decrement R4
        Branch>0  LOOP
        Move    R1,SUM1
        Move    R2,SUM2
        Move    R3,SUM3
```

Рис.14. Индексная адресация используется для доступа к оценкам студентов в списке, представленном на рис.13

В теле цикла программы для доступа к записям с каждой из трех оценок очередного студента применяется индексный метод адресации, в том виде, в каком он представлен на рисунке 12, *а*. В качестве индексного регистра используется регистр R0. Перед началом цикла в этот регистр записывается адрес кода первого студента в памяти компьютера, то есть адрес LIST.

На каждом шаге цикла оценки очередного студента прибавляются к текущим суммам, хранящимся в регистрах R2, R2, и R3 (перед началом цикла все три регистра обнуляются – устанавливаются в 0) Доступ к оценкам осуществляется посредством индексов 4(R0), 8(R0) и 12(R0). Затем значение в индексном регистре увеличивается на 16, чтобы он указывал на код следующего студента. Из содержимого регистра R4, куда перед началом цикла было помещено значение n, вычитается 1. На этом очередной шаг цикла заканчивается, а на следующем шаге все повторяется сначала, пока содержимое регистра R4 не станет равным нулю. С помощью команды условного перехода «Branch>0 LOOP» управление передается обратно на начало цикла, для обработки следующей записи. Последние три команды программы пересылают итоговые суммы из регистров R1, R2 и R3 в память по адресам SUM1, SUM2 и SUM3.

Обратите внимание, что содержимое индексного регистра R0 при обращении к записям с оценками студентов *не меняется.* Оно будет изменено только последней командой Add, предназначенной для перехода от одной записи к другой в конце очередного шага цикла.

Можно сказать, что индексная адресация предназначена для доступа к операнду, расположение которого определено относительно некоторой базовой точки в структуре данных, где хранится этот операнд.

Для наиболее эффективного доступа к операндам в памяти в разных ситуациях применяются различные вариации этой базовой формы. Так смещение X может содержаться во втором регистре. В этом случае для представления операнда используется такая запись:

(Ri,Rj)

Исполнительный адрес вычисляется как сумма содержимого регистров Ri и Rj. Второй регистр обычно называют *базовым*, а содержащееся в нем значение – базой.

Еще одна разновидность индексного режима адресации основывается на использовании двух регистров и константы. Она обозначается так: X(Ri,Rj). В этом случае исполнительный адрес является суммой константы X и содержимого регистров Ri и Rj. Этот еще более гибкий способ адресации используется для доступа к нескольким компонентам каждого элемента записи, когда начало элемента записи определяется частью (Ri,Rj), а X задает смещение компонента относительного начала элемента. Другими словами, этот режим используется для работы с трехмерными массивами.

Относительная адресация

Запись X(PC) означает, что исполнительный адрес смещен на X байтов относительно адреса, заданного в счетчике команд. Этот режим называется *относительной адресацией*

Относительная адресация – это способ адресации, при котором исполнительный адрес определяется так же, как в индексном режиме, но вместо регистра общего назначения используется счетчик команд. Более типичным использованием этого режима является определение целевого адреса в команде перехода. Например, команда Branch>0 LOOP приводит к передаче управления по целевому адресу, определяемому именем LOOP, в случае, если удовлетворяется условие перехода. Целевой адрес можно не задавать жестко, а вычислять как смещение относительно текущего значения счетчика команд. А поскольку целевой адрес может находиться как выше, так и ниже команды перехода, смещение задается в виде целого числа со знаком. Вспомним, что в ходе выполнения команды процессор увеличивает значение счетчика, чтобы он указывал на следующую команду программы. В связи с этим, например, в программе, представленной на рисунке 14, на момент, когда будет генерироваться целевой адрес перехода – метка LOOP, уже обновленное значение в регистре PC будет на 24 единицы больше. Чтобы перейти по адресу LOOP, нужно использовать смещение X = - 24.

В языке ассемблера в командах перехода для указания целевого адреса используются метки (LOOP, например). Когда компилятор при обработке ассемблерной программы встречает такую команду, он вычисляет значение смещения, в нашем случае -24, и генерирует соответствующую машинную команду, используя относительную адресацию, то есть -24(PC).

Дополнительные режимы адресации

Хотя описанных выше базовых режимов адресации вполне достаточно, некоторые компьютеры поддерживают еще и дополнительные режимы, облегчающие решение программных задач. Рассмотрим два таких режима, полезных для доступа к элементам данных, расположенных последовательно в памяти.

Автоинкрементная адресация – это режим адресации, при котором исполнительный адрес содержится в указанном в команде регистре. После обращения к операнду значение в этом регистре автоматически увеличивается таким образом, чтобы он указывал на следующий элемент списка.

В команде на языке ассемблера автоинкрементная адресация обозначается так: имя регистра заключается в скобки, показывая тем самым, что данный регистр содержит исполнительный адрес, а за ним следует знак «+», указывающий, что после обращения к операнду значение в регистре должно быть увеличено:

$(Ri)^+$

Если режим адресации задается в такой форме, то есть неявно, значит, значение в регистре увеличивается на 1. Однако в случае памяти с побайтовой адресацией такой режим полезен лишь для доступа к последовательным байтам списка. Если же требуется получить доступ к последовательным словам, а длина слова составляет 32 разряда, приращение должно быть равным 4. Компьютеры, поддерживающие автоинкрементную адресацию, автоматически увеличивают содержимое регистра на значение, соответствующее размеру операнда. Таким образом, для операнда размером в 1 байт приращение равняется 1, для 16-разрядного операнда -2, для 32-разрядного операнда -4. Поскольку размер операнда команды обычно определяется как часть кода операции, в самой команде для указания автоинкрементного режима достаточно обозначения $(Ri)^+$.

В дополнение к автоинкрементному режиму адресации компьютером может поддерживаться аналогичный ему режим, предназначенный для доступа к элементам списка в обратном порядке.

Автоинкрементная адресация – это режим адресации, при котором содержимое указанного в команде регистра сначала автоматически уменьшается, а затем используется в качестве исполнительного адреса операнда.

При автодекрементном режиме имя регистра заключается в скобки, перед которыми ставится знак «-», указывающий, что сначала из хранящегося в регистре значения должна быть вычтена некоторая константа: -(Ri). В таком режиме доступ к операндам осуществляется в порядке уменьшения значений адресов.

```
        Move        N,R1
        Move        #NUM1,R2
        Clear       R0
LOOP    Add         (R2)+,R0
        Decrement   R1
        Branch>0    LOOP
        Move        R0,SUM
```

Рис.15. Пример автоинкрементной адресации

ЗАПИСЬ ЧИСЕЛ НА АССЕМБЛЕРЕ

При работе с числовыми значениями обычно используют привычную десятичную запись. В компьютере числовые значен

В некоторых случаях удобнее прямо задавать числовые значения различными способами, используя соглашения, определяемые синтаксисом языка. Возьмем в качестве примера число 93, представленное в виде 8-разрядной двоичной записи 0101 1101. Если непосредственно использовать это значение в качестве операнда, его можно задать в виде десятичного числа, как в следующей команде:

ADD #93,R1

или же в виде двоичного числа, на что указывает специальный префикс, такой как символ %.

ADD #%01011101,R1

Двоичные числа можно записывать более компактно, используя *шестнадцатеричную систему,* в которой каждые четыре бита числа представлены *одной шестнадцатеричной цифрой.* Шестнадцатеричную запись можно считать расширенной версией двоично-десятичной записи. Первые десять кодов, 0000, 0001, …,1001, обозначаются цифрами 0, 1,…, 9, как в двоично-десятичной системе кодирования. Оставшиеся шесть 4-разрядных кодов, 1010, 1011,…,1111, обозначаются буквами A, B, …,F. В шестнадцатеричном представлении десятичное значение 93 записывается как 5D. В языке ассемблера шестнадцатеричное представление часто задается с помощью префикса в виде знака доллара ($). Таким образом, команда прибавляющая к содержимому регистра 93, при использовании шестнадцатеричной системы кодирования чисел:

ADD #$5D,R1.

Базовые операции ввода-вывода

Рассмотрим процессы, с помощью которых данные пересылаются между памятью компьютера и внешним миром. Операции ввода-вывода являются одной из важнейших составляющих работы компьютера, и от того, как они выполняются, в значительной мере зависит его производительность. Рассмотрим несколько базовых концепций ввода-вывода.

Предположим, нам необходимо считать вводимые с клавиатуры символы и вывести их на экран дисплея. Простейший способ выполнения подобных задач заключается в использовании метода, который называется *программно управляемым вводом-выводом.* Скорость передачи данных от клавиатуры к компьютеру зависит от того, насколько быстро пользователь может их вводить (это порядка нескольких символов в секунду). Скорость передачи данных от компьютера к дисплею гораздо выше. Она определяется пропускной способностью соединения между компьютером и дисплеем. Которая обычно составляет несколько тысяч символов в секунду.

Однако и это невероятно мало, если сравнивать со скоростью работы процессора, выполняющего миллионы команд в секунду. Разница в быстродействии процессора и устройств ввода-вывода вызывает потребность в механизмах синхронизации процесса передачи данных между ними.

Решение этой проблемы заключается в следующем. Процессор отсылает дисплею первый символ и ждет от него сигнала о том, что символ получен. Затем он отсылает второй символ, снова ждет сигнала и т.д. Точно также отсылаются процессору данные, вводимые с клавиатуры. Процессор ждет Сигнала, означающего, что пользователь нажал одну из клавиш и что ее код помещен в некоторый буферный регистр, связанный с клавиатурой. Получив сигнал, процессор считывает код клавиши. Как показано на рисунке 15, клавиатура и дисплей являются совершенно независимыми устройствами. Нажатие клавиши на клавиатуре само по себе не вызывает вывода на экран соответствующего символа. Один блок команд в программе ввода-вывода пересылает символы в процессор, а другой выводит таковые на экран. Рассмотрим, как пересылается код символа от клавиатуры процессору. В ответ на нажатие клавиши соответствующий ей код символа сохраняется в 8-разрядном буферном регистре, связанном с клавиатурой. Он назван условно DATAIN (см.рис.15). Для уведомления процессора о наличии нового кода символа в регистре DATAIN специальный флаг состояния SIN устанавливается в 1. Программа отслеживает состояние этого флага, и когда он оказывается установленным в 1, процессор считывает содержимое регистра. После пересылки символа в процессор значение флага SIN автоматически сбрасывается в 0. Когда с клавиатуры вводится второй символ, флаг SIN снова устанавливается в 1 и процесс повторяется.

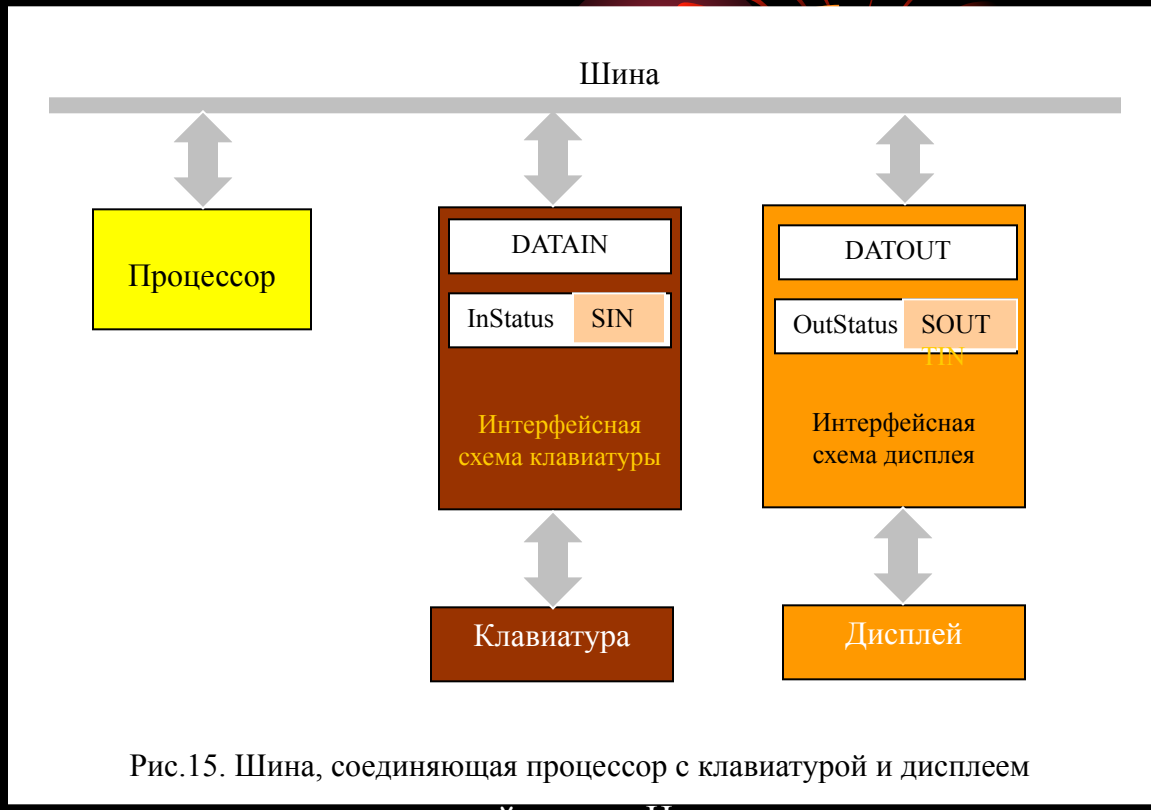


Рис.15. Шина, соединяющая процессор с клавиатурой и дисплеем

Вывод символа на экран осуществляется аналогичным образом. Но в этом случае для его пересылки от процессора к монитору используются буферный регистр DATAOUT и флаг SOUT.

Если значение SOUT равно 1, значит, дисплей готов к приему символа. Процессор под управлением программы отслеживает содержимое флага SOUT, и когда его значение оказывается равным1,процессор пересылает код символа в регистр DATAOUT. После этого флаг SOUT сразу же снимается. Когда дисплей готов к приему следующего символа, флаг SOUT опять устанавливается в 1 и процесс повторяется. Буферные регистры DATAIN и DATAOUT совместно с флагами состояния SIN и SOUT являются частью схемы, называемой *интерфейсом устройства.* Такая схема имеется у каждого устройства ввода-вывода. С процессором она соединяется через шину (рис.15). Предполагается, что в исходном состоянии флаг SIN сброшен, а флаг SOUT установлен. Буферы данных, такие как DATAIN и DATAOUT (см.рис.15) могут адресоваться так, как если бы они располагались в основной памяти компьютера. Точно также можно было бы поступить и с флагами SIN и SOUT, присвоив им разные адреса. Но чаще эти флаги включают в состав *регистров состояния устройства* (такой регистр имеется у каждого устройства ввода-вывода). Предположим, что в интерфейсных схемах клавиатуры и дисплея бит *b3* соответствует флагу SIN в регистре InStatus и флагу SOUT в регистре OutStatus. Тогда операцию чтения из регистра DATAIN можно выполнить с помощью приведенных ниже команд.

```
READWAIT    TestBit     #3, InStatus
            Branch = 0  READWAIT
            MoveByte    DATAIN,R1
```

А операция записи в регистр дисплея может быть реализована так:

```
WRITEWAIT   TestBit     #3, OutStatus
            Branch =0   WRITEWAIT
            MoveByte    R1,DATAOUT
```

Команда TestBit проверяет состояние одного бита (позиция которого определяется первым операндом) во втором операнде. Если значение этого бита равно 0, значит, условие команды Branch истинно и она выполняет переход на начало цикла ожидания. Когда устройство будет готово (значение проверяемого бита окажется равным 1) данные будут прочитаны из входного буфера или записаны в таковой.

В фрагменте программы, представленной ниже иллюстрируются операции чтения процессором введенной с клавиатуры строки символов и вывода в последующем этой строки символов на экран. Символы поочередно считываются по мере ввода и сохраняются в области данных в памяти, а затем выводятся на экран. Программа завершает свою работу после того, как ею будет прочитан, сохранен и выведен на экран символ возврата каретки CR. Первый байт области памяти, где хранится считываемая с клавиатуры строка, имеет адрес LOC. Этот адрес хранится в регистре R0, куда он записывается первой командой программы. По мере чтения символов значение регистра R0 увеличивается на 1, для чего в команде Compare используется автоинкрементный способ адресации.

Программно управляемый ввод-вывод требует постоянного участия процессора, что не очень хорошо – процессор не должен простаивать. Поэтому для управления вводом-выводом используется технология формирования сигналов прерывания.

Программа считывающая строку символов и выводящая ее на экран

	Move	#LOC,R0	Инициализация регистра R0, то есть занесение в него адреса, по которому строка символов будет хранится в памяти
READ	TestBit	#3,INSTATUS	Ожидание ввода символа в буфер клавиатуры DATAIN
	Branch =0	READ	
	MoveByte	DATAIN,(R0)	Пересылка символа из регистра DATAIN в память (эта операция сбрасывает флаг SIN)
ECHO	TestBit	#3,OUTSTATUS	Ожидание готовности дисплея
	Branch =0	ECHO	Пересылка только что прочитанного символа в буферный регистр дисплея (эта операция сбрасывает флаг SOUT)
	MoveByte	(R0),DATAIN	
	Compare	#CR,(R0)+	Проверка того, является ли только что прочитанный символ символом CR, а также увеличение значения указателя, с тем чтобы он указывал на следующий символ
	Branch ≠ 0	READ	

Стеки и очереди

Компьютерным программам часто приходится выполнять определенные подзадачи с использованием структуры, называемой подпрограммой. Для обмена информацией между главной программой и подпрограммой предназначена специальная структура данных, называемая *стеком. Стек –* это список элементов данных, обычно слов или байтов, доступ к которым ограничен следующим правилом: элементы этого списка могут добавляться только в его конец и удаляться только из конца. Конец списка называется вершиной стека, а его начало - дном. Такую структуру иногда называют *магазином.* Представьте себе стопку подносов в столовой. Клиенты берут подносы сверху, и работники столовой, добавляя чистые подносы, тоже кладут их наверх стопки. Этот механизм хранения хорошо описывается емкой фразой «последним вошёл – первым вышел» (Last In First Out, LIFO), означающей, что элемент данных, помещенный в стек последним, удаляется из него первым. Операцию помещения нового элемента в стек часто называют его *проталкиванием* (push), а операцию извлечения последнего из стека называют его *выталкиванием* (pop).

Хранящиеся в памяти компьютера данные могут быть организованы в виде стека, так чтобы последовательные элементы располагались друг за другом. Предположим, что первый элемент хранится по адресу BOTTOM, а когда в стек помещаются новые элементы, они располагаются в порядке уменьшения последовательных адресов. Таким образом, стек растет в направлении уменьшения адресов, что является весьма распространенной практикой. На рисунке 16 показано, как располагается в памяти компьютера стек, элементы которого занимают по одному слову. На дне он содержит числовое значение 43, а на вершине – 28. Для отслеживания адреса вершины стека используется регистр процессора, называемый *указателем стека* (Stack Pointer, SP). Это может быть один из регистров общего назначения или же регистр, специально предназначенный для этой цели. Если предположить, что память адресуется побайтово и слово имеет длину 32 разряда, операцию проталкивания в стек можно реализовать так:

```
Substrract      #4,SP
Move            NEWITEM,(SP)
```

где команда Substrract вычитает исходный операнд 4 из результирующего операнда, содержащегося в регистре SP,

и помещает результат в регистр SP. Эти две команды помещают слово, хранящееся по адресу NEWITEM, на вершину стека, предварительно уменьшая указатель стека на 4. Операция выталкивания из стека может быть реализована так:

```
Move    (SP),ITEM
ADD     #4,SP
```

Эти две команды перемещают значение, хранящееся на вершине стека, в другое место памяти, по адресу ITEM, а затем увеличивают указатель стека на 4, чтобы он указывал на тот элемент, который теперь располагается на вершине стека.

Если процессор поддерживает режим автоинкрементной и автодекрементной адресации, для помещения нового элемента в стек достаточно команды

```
Move    NEWITEM,-(SP)
```

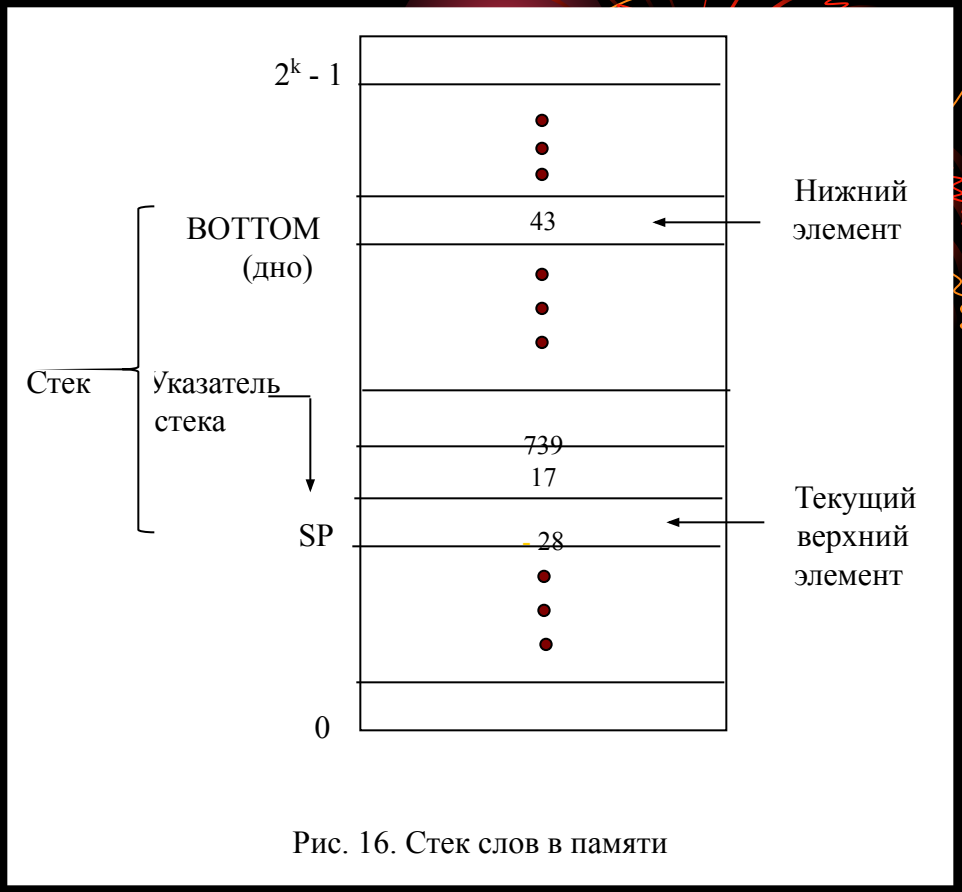


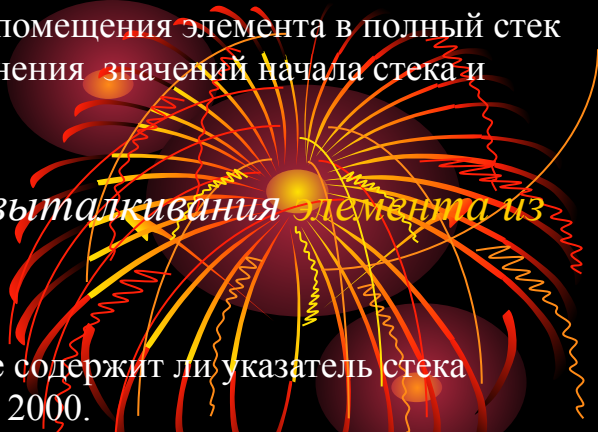
Рис. 16. Стек слов в памяти

А выталкивание элемента из стека можно выполнить посредством команды

```
Move    (SP)+,ITEM
```

Когда стек используется программой, для него обычно выделяется фиксированное количество памяти. В этом случае нам нужно проследить за тем, чтобы программа не пыталась помещать новые элементы в стек, достигший своего максимального размера. Кроме того, она не должна пытаться вытолкнуть элемент из пустого стека, что могло бы произойти в случае логической ошибки. Как это может произойти в реальном случае? Предположим, что стек заполняется начиная с адреса 2000 (BOTTOM) до 1500 и далее. Первоначально в регистр, выполняющий роль указателя стека, загружается значение адреса равное 2004. Напомним, что перед помещением в стек нового элемента данных из значения в регистре SP каждый раз вычитается 4. Поэтому начальное значение 2004 означает,

что первый элемент стека будет иметь адрес 2000. Для предотвращения попыток помещения элемента в полный стек или удаления элемента из пустого стека необходимо ввести проверку путем сравнения значений начала стека и вершины стека, например следующим образом:



Контроль пустого и полного стеков при выполнении операций выталкивания элемента из стека и помещения элемента в стек:

SAFEPOP	Compare Branch>0	#2000,SP EMPTYERROR	Проверка того, не содержит ли указатель стека значения больше 2000. Если это так, значит стек пуст, и осуществляется переход к подпрограмме EMPTYERROR (пустой), выполняющей соответствующее действие
	Move	(SP)+,ITEM	В противном случае элемент, расположенный на вершине стека, выталкивается, а затем помещается в память по адресу ITEM
SAFEPUSH	Compare Branch≤0	#1500,SP FULLERROR	Проверка того, не содержит ли указатель стека значения меньше или равное 1500. Если содержит, значит, стек полон. В таком случае выполняется переход к подпрограмме FULLERROR, осуществляющей соответствующее действие
	Move	NEWITEM,-(SP)	В противном случае в стек помещается элемент хранящийся в памяти по адресу NEWITEM

Ещё одна полезная структура данных, очень похожая на стек, называется *очередью*. Данные помещаются в очередь и извлекаются из нее в порядке «первым вошел – первым вышел» (First In First Out, FIFO). Как правило, очередь растет в направлении увеличения адресов. В этом случае новые данные добавляются в её конец, имеющий наибольший адрес, а извлекаются таковые из начала очереди, имеющего наименьший адрес.

Между способами реализации стека и очереди имеются два важных различия. Один конец стека (его дно) зафиксирован, а другой перемещается то вверх то вниз по мере добавления и удаления данных. Для ссылки на

Второе различие между стеком и очередью состоит в том, что без дополнительного управления очередь будет постоянно смещаться в памяти компьютера в направлении увеличения адресов. Один из способов удержания её в фиксированной области заключается в использовании так называемого *циклического буфера*. По мере приближения к метке END будет определено местоположения новой метке начала очереди BEGINNING, смещенное в сторону меньших адресов памяти.

Подпрограммы

Очень часто программа должна по многу раз выполнять определенную подзадачу, но с разными значениями данных. Такая подзадача обычно называется *подпрограммой.* Подпрограмма может, скажем вычислять функцию sin или сортировать список в порядке возрастания или убывания значений. Составляющий такую подпрограмму блок команд можно включать во все те места программы, где он должен выполняться. Однако на практике так никогда не поступают. Для экономии места в память помещают только одну копию блока команд, и любая программа, которой потребуется выполнить эту подпрограмму, просто переходит к её начальному адресу. Такой переход называется вызовом подпрограммы и выполняется с помощью команды Call.

После реализации подпрограммы работа вызывающей её программы должна быть продолжена. В таком случае говорят, что выполняется *возврат* из подпрограммы в вызывающую программу. Делается это с помощью команды Retorn. Поскольку подпрограмма может вызываться из нескольких разных мест основной программы, при её вызове где-то должен сохраняться адрес возврата. Иными словами, для обеспечения правильного возврата из подпрограммы команда Call должна сохранить содержимое регистра PC.

Применяемый компьютером способ выполнения вызовов подпрограмм и возврата из таковых называется методом *связывания подпрограмм*. Простейший метод связывания подпрограмм заключается в сохранении адреса возврата в заданном месте, которым может быть специально выделенный для этого регистр. Такой регистр называется регистром связи. Когда работа подпрограммы завершается, команда Retorn возвращает управление вызывающей программе, выполняя неявный переход через регистр связи.

Особой разновидностью команды перехода является команда Call, выполняющая такие операции, как сохранение содержимого регистра PC в регистре связи и переход по указанному в команде целевому адресу. Команда Retorn также является разновидностью команды перехода, но она выполняет переход по адресу,

заданному в регистре связи. Этот процесс проиллюстрирован рисунком 17.

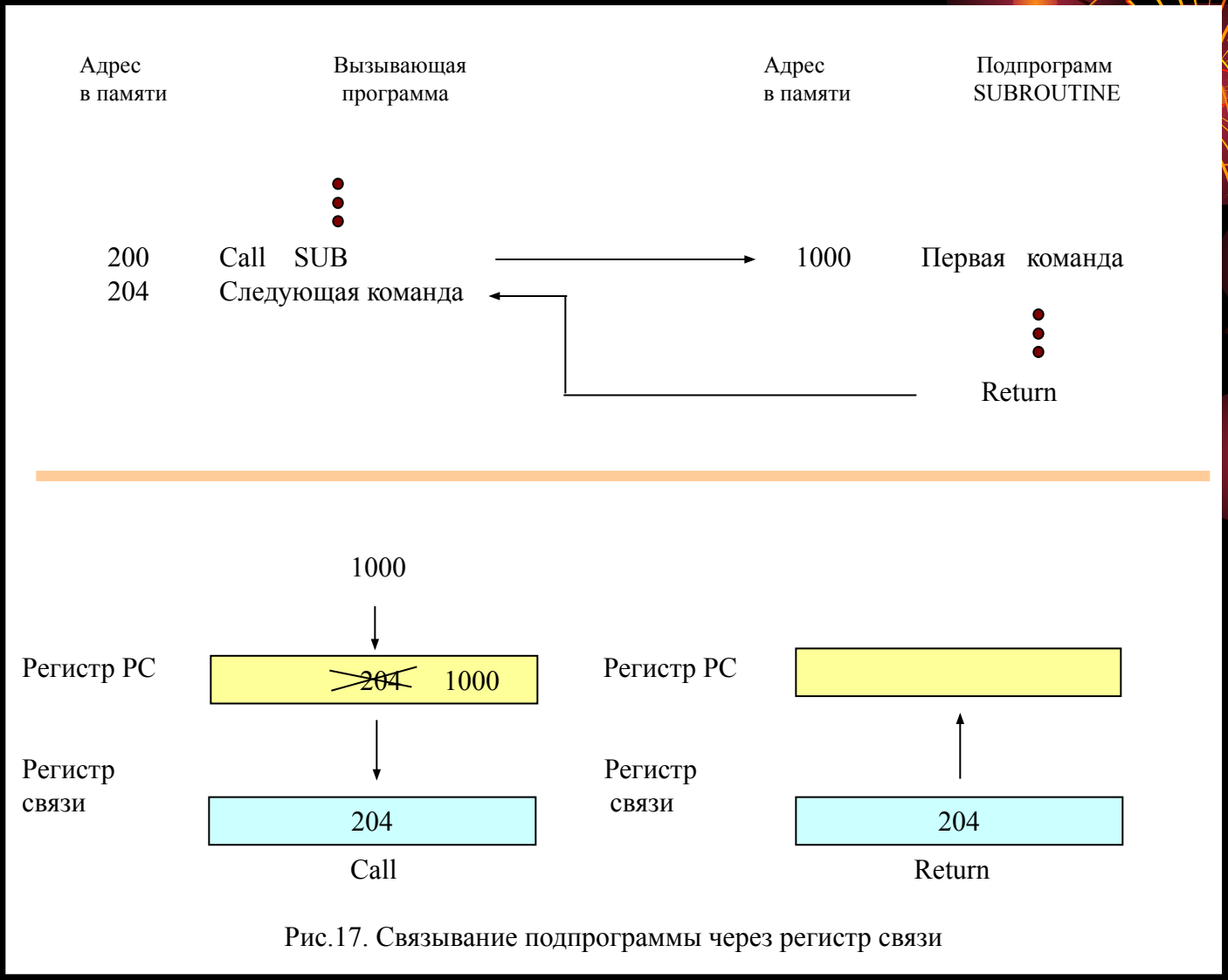


Рис.17. Связывание подпрограммы через регистр связи

Передача параметров

Вызывая подпрограмму, программа должна передать ей параметры (операнды), которые будут использоваться

в вычислениях, или же их адреса. Закончив свою работу, подпрограмма вернет другие параметры – результаты вычислений. Такой обмен информацией между вызывающей программой и подпрограммой называется *передачей параметров*. Передача параметров может выполняться несколькими способами. Например, параметры можно Помещать в регистры или в память, откуда подпрограмма может их считать. В качестве альтернативы параметры можно поместить в стек процессора, используемый для хранения адресов возврата.

Использование регистров процессора – способ простой и эффективный. Ниже приведен пример такого способа передачи параметров через регистры процессора при выполнении сложения последовательности чисел путем использования программы и подпрограммы:

Вызывающая программа

```
        Move      N,R1          R1 играет роль счетчика
        Move      #NUM1,R2      R2 указывает на последовательность
        Call      LISTADD       Вызов подпрограммы
        Move      R0,SUM        Сохранение результата
        .
        .
        .
```

Подпрограмма

```
LISTADD  Clear      R0          Инициализация суммы значением 0
LOOP     Add        (R2)+,R0    Добавление числа из последовательности
         Decrement  R1
         Branch>0   LOOP
         Return                 Возврат в вызывающую программу
```

n – длина последовательности слагаемых чисел, информация о которой хранится в памяти по адресу N и адрес

первого числа, NUM1, передаются подпрограмме через регистры R1 и R2. Вычисленная подпрограммой сумма возвращается вызывающей программе через регистр R0. Соответствующую часть вызывающей программы составляют четыре первые представленные команды. Первые две команды загружают в регистры R1 и R2 значения N и NUM1. Команда Call выполняет переход к подпрограмме, начинающейся по адресу LISTADD. Кроме того, эта команда помещает в стек процессора адрес возврата из подпрограммы. Подпрограмма вычисляет сумму и помещает её в регистр R0. После возврата из подпрограммы вызывающая программа сохраняет эту сумму в памяти по адресу SUM.

Если у подпрограммы много параметров, для их передачи может просто не хватить регистров общего назначения. С другой стороны, стек –структура гибкая, в него можно поместить много параметров. Следующий пример показывает, как выполняется передача параметров через стек. В принципе решается та же задача суммирования слагаемых, но передача параметров подпрограмме реализуется через стек.

Вызывающая программа помещает параметры, которыми будет пользоваться подпрограмма в стек. Подпрограмме LISTADD передается адрес первого числа в последовательности слагаемых и количество слагаемых - n. Подпрограмма выполняет сложение и возвращает полученную сумму, которая временно помещается в стек на место первого слагаемого (NUM1 – команда : Move 20(SP),R2). Предположим, что до вызова подпрограммы вершина стека располагается на уровне 1. Вызывающая программа помещает в стек адрес NUM1 и значение n и вызывает подпрограмму LISTADD. Кроме того, команда Call помещает в стек адрес возврата из подпрограммы. Теперь вершина стека располагается на уровне 2.

Подпрограмма использует три регистра. Поскольку в них могут содержаться данные, принадлежащие вызывающей программе, их содержимое сохраняется в стеке. Для помещения в стек содержимого регистров от R0 до R2 используем команду MovMultiple R0-R2,-(SP). Подобные команды имеются у многих процессоров. Теперь вершина стека располагается на уровне 3. С помощью индексной адресации подпрограмма считывает из стека параметры n и NUM1. Значение указателя стека при этом не меняется, поскольку на вершине стека по-прежнему располагаются нужные нам элементы данных. Значение n загружается в регистр R1, который будет играть роль счетчика, а адрес NUM1 – в регистр R2, который будет служить указателем при сканировании списка значений.

Пример программы, передающей параметры подпрограмме через стек, вызывающая программа и подпрограмма.
(Предполагается, что вершина стека расположена на уровне 1)

```
                Move        #NUM1,-(SP)     Помещение параметров в стек
                Move        n,-(SP)
                Call        LISTADD         Вызов подпрограммы
                                            (вершина стека на уровне 2)
                Move        4(SP),SUM       Сохранение результата
                Add         #8,SP           Восстановление вершины стека
                  .                         (вершина стека на уровне 1)
                  .
                  .
LISTADD         MovMultiple R0-R2,-(SP)     Чтобы была возможность использовать
                                            регистры процессора общего назначения,
                                            в стеке сохраняются прежнее содержание
                                            их
                Move        16(SP),R1       В стек заносится значение счетчика - n
                Move        20(SP),R2       В стек заносится адрес первого числа
                Clear       R0              Очищается  R0  для накопления суммы
LOOP            Add         (R2)+,R0        Суммирование последовательности чbсел
                Decrement   R1
                Branch>0    LOOP
                Move        R0,20(SP)       Помещение результата в стек на место
                                            адреса первого слагаемого
                MovMultiple (SP)+,R0-R2     Восстановление регистров
                Return                      Возврат в вызывающую программу
```

Регистр R0 должен содержать результат суммирования. Перед возвратом из подпрограммы содержимое регистра R0 Помещается в стек на место, где был записан параметр NUM1, который нам больше не нужен. Затем из стека Восстанавливается содержимое трех регистров, использовавшихся подпрограммой. Теперь верхним Элементом стека является адрес возврата, расположенный на уровне 2. После возврата из подпрограммы вызывающая программа сохраняет результат сложения в памяти по Адресу SUM и возвращает вершину стека на ее исходный уровень, увеличивая значение SP на 8 единиц.

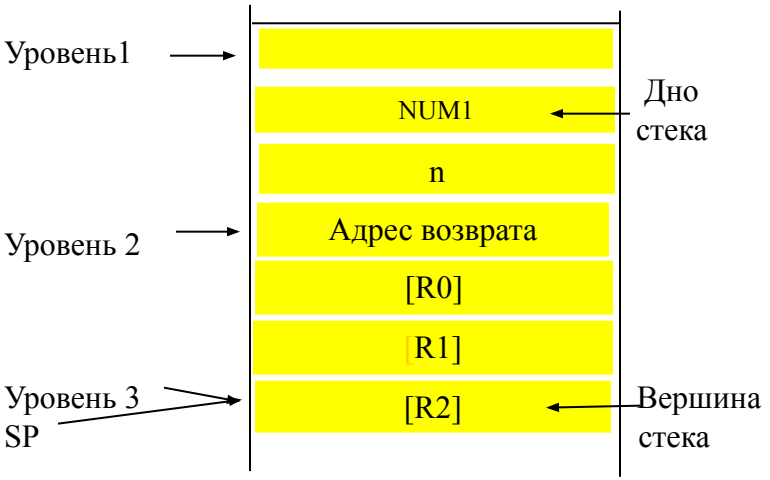


Рис.18. Схема стека при передаче параметров от программы подпрограмме

Передача параметров по значению и по ссылке

Задачей подпрограммы является сложение последовательности чисел. Но, вместо того чтобы передавать непосредственно числа последовательности, вызывающая программа передает подпрограмме адрес ее первого элемента (первого слагаемого). Эта технология называется *передачей параметров по ссылке.* Второй параметр – число слагаемых n передается подпрограмме в реальном виде, а не в виде адреса, где эта информация находится. Этот способ назвали *передачей параметров по значению*.

Стековый фрейм

Н а рисунке 18 представлена структура стека, используемого программой и подпрограммой. Эта область памяти составляет собственное рабочее пространство подпрограммы, создаваемое при ее вызове и освобождаемое, когда управление возвращается вызывающей программе. Эта область называется ещё *стековым фреймом*.

Пример типичного расположения информации в стековом фрейме приведен на рисунке 19.

В дополнение к указателю стека SP можно использовать ещё один указатель, который помещается в один из общих, или в специально назначенный регистр, обычно называемый *указателем фрейма* (Frame Pointer, FP). Он облегчает доступ к параметрам подпрограммы и ее локальным переменным. Эти локальные переменные используются только внутри подпрограммы, Так что память для их хранения удобнее всего выделить прямо в стековом фрейме, связанном с данной подпрограммой. В нашем случае предполагается, что подпрограмма получает четыре параметра, применяет три локальные переменные и сохраняет содержимое регистров R0 и R1, которые она использует для своих нужд.

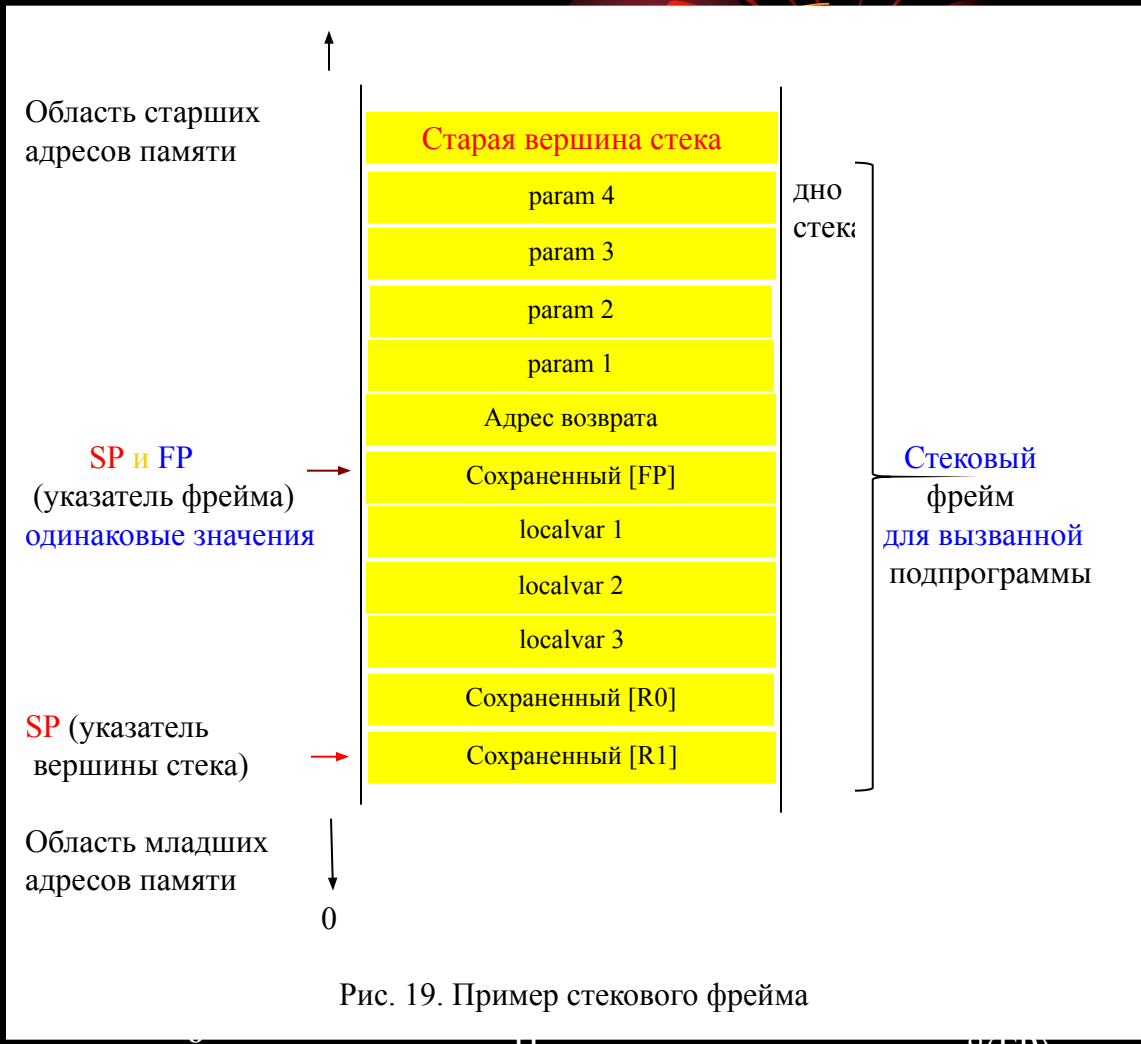


Рис. 19. Пример стекового фрейма

Поскольку регистр FP указывает на область памяти, расположенную непосредственно над сохраненным (в направлении роста стека) в стеке адресом возврата, с его помощью легко обращаться к параметрам и локальным переменным, применяя индексный режим адресации. Параметры адресуются так: 8(FP), 12(FP) и т.д., а локальные переменные так: -4(FP), -8(FP), …..

Содержимое регистра FP в ходе выполнения подпрограммы остается неизменным, тогда как указатель стека SP должен всегда указывать на верхний элемент стека (на верхнюю запись).

Теперь давайте посмотрим, каковы правила установки указателей SP и FP при создании, использовании и уничтожении стекового фрейма в результате вызова конкретной подпрограммы. Будем иметь ввиду, что перед вызовом подпрограммы указатель SP указывает на старую вершину стека.

Дополнительные команды

Познакомимся с рядом важных команд, поддерживаемых большинством компьютеров.

Логические команды

Логические схемы реализуют такие выполняемые над отдельными битами логические операции, как И, ИЛИ и НЕ. Наряду с этим полезно иметь возможность производить указанные операции и программным путем, для чего используются команды, способные выполнять их над всеми битами слова или байта по отдельности и параллельно. Так команда

Not dst (приемник)

дополняет все биты операнда, заменяя нули единицами, а единицы нулями. Известно, что если прибавить 1 к дополнению положительного числа со знаком до единицы получается отрицательная версия дополнения этого же числа до двух. Например, число + 3 (0011) путем прибавления 1 к его дополнению до единице (1100) преобразуется в – 3 (1101). Если число 3 содержится в регистре R0, данное преобразование выполняют команды

```
Not    R0
Add   #1,R0
```

Во многих компьютерах эту задачу выполняет одна команда Negate R0

Рассмотрим ещё одну логическую команда And, выполняющая побитовую операцию И над исходным и результирующим операндами. Предположим, что в 32-разрядном регистре R0 содержатся четыре символа в кодах ASCII. Требуется определить, является ли первый символ буквой Z. Если да, будет выполнен условный переход по адресу YES. ASCII код буквы Z равен 01011010 или 5A в шестнадцатеричном формате. Нужное нам действие выполняют следующие три команды:

```
And           #$FF000000,R0
Compare     #$5A000000,R0
Branch = 0   YES
```

Команда And очищает все биты трех расположенных справа символов, оставляя крайний слева символ нетронутым. Это результат использования непосредственно заданного исходного операнда, состоящего из восьми единиц слева и 24 нулей справа.

Команда Compare сравнивает оставшийся символ с левого края регистра R0 с двоичным представлением символа Z.

Если они совпадают, команда Branch выполняет переход по адресу YES.

Следует отметить, что команда And часто используется при необходимости очистить все разряды операнда, за исключением заданных разрядов. Для этого второй операнд в команде, выполняющий роль маски должен содержать нулевые биты на месте тех разрядов, которые должны быть установлены в 0 в первом операнде. В нашем примере Нужно оставить нетронутыми восемь крайних слева битов регистра R0.

Команды сдвига

Ещё одна типичная задача программирования требует сдвига всех битов операнда вправо или влево на заданное количество разрядов. Процесс выполнения сдвига зависит от того, содержит ли операнд знак или какую-нибудь другую двоично-кодированную информацию. В общем случае, используется логический сдвиг. Для чисел производится арифметический сдвиг, при котором сохраняется знак числа.

Логический сдвиг

Для поддержки операции логического сдвига необходимы две команды: одна для выполнения сдвига влево (LShiftL), а другая вправо (LshiftR). Эти команды сдвигают операнд на заданное количество разрядов, определяемое операндом count. Общий синтаксис команды, выполняющей логический сдвиг влево, таков:

LShiftL count, dst

Операнд count может быть задан непосредственно или содержаться в регистре процессора. Освобождающиеся в результате сдвига разряды устанавливаются в 0, а сдвигаемые за границу операнда разряды отмечаются с помощью флага переноса C, а затем удаляются. Устанавливать флаг C особенно удобно при выполнении арифметических операций с большими числами, занимающими больше одного слова. На рисунке 20, *а* показан пример сдвига содержимого регистра R0 влево на два разряда. Команда логического сдвига вправо работает точно также (рисунок 20, *б)*.

Арифметический сдвиг

Сдвиг числа на один разряд влево эквивалентен его умножению на 2, а сдвиг на один разряд вправо – делению на 2. Конечно, при сдвиге влево может произойти переполнение, А при сдвиге вправо может потеряться конец числа. Ещё одно важное наблюдение заключается в том, что при сдвиге в освободившемся разряде должен быть повторен знаковый бит. Этим арифметический сдвиг вправо Отличается от логического, в Котором освобождающиеся разряды Всегда заполняются нулями. Пример арифметического сдвига вправо (AShiftR) приведен на рисунке 20, *в.* Арифметический сдвиг влево ничем не отличается от логического.

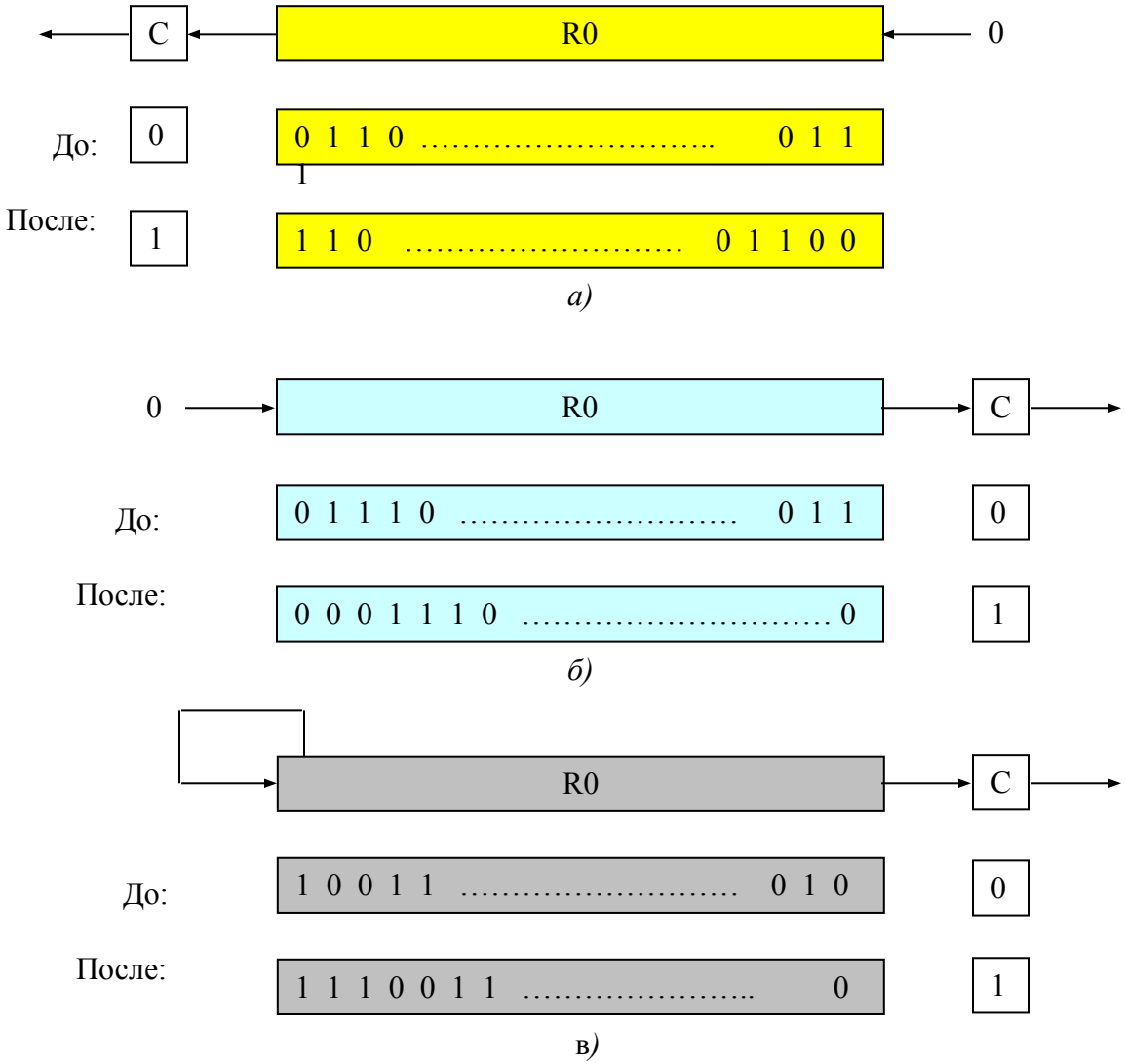


Рис.20. Команды логического и арифметического сдвига:
логический сдвиг влево, LShiftL #2,R0 *(а);*
логический сдвиг вправо, LShiftR #2,R0 *(б);*
арифметический сдвиг вправо, AShiftR #2,R0 *(в);*

Циклический Сдвиг

В операциях сдвига те биты, которые перемещаются за пределы операнда, оказываются просто потерянными. сохраняется только последний сдвинутый бит, который копируется во флаг переноса С. Для сохранения всех битов операнда применяются операции циклического сдвига, перемещающие «выдвинувшиеся» с одного края разряды на другой край. Обычно компьютер поддерживает две версии циклического сдвига вправо и две версии циклического сдвига влево. В первой версии разряды операнда просто циклически сдвигаются, а во второй в сдвиге участвует ещё и флаг С. Примеры всех четырех операций циклического сдвига приведены на рисунке 21.

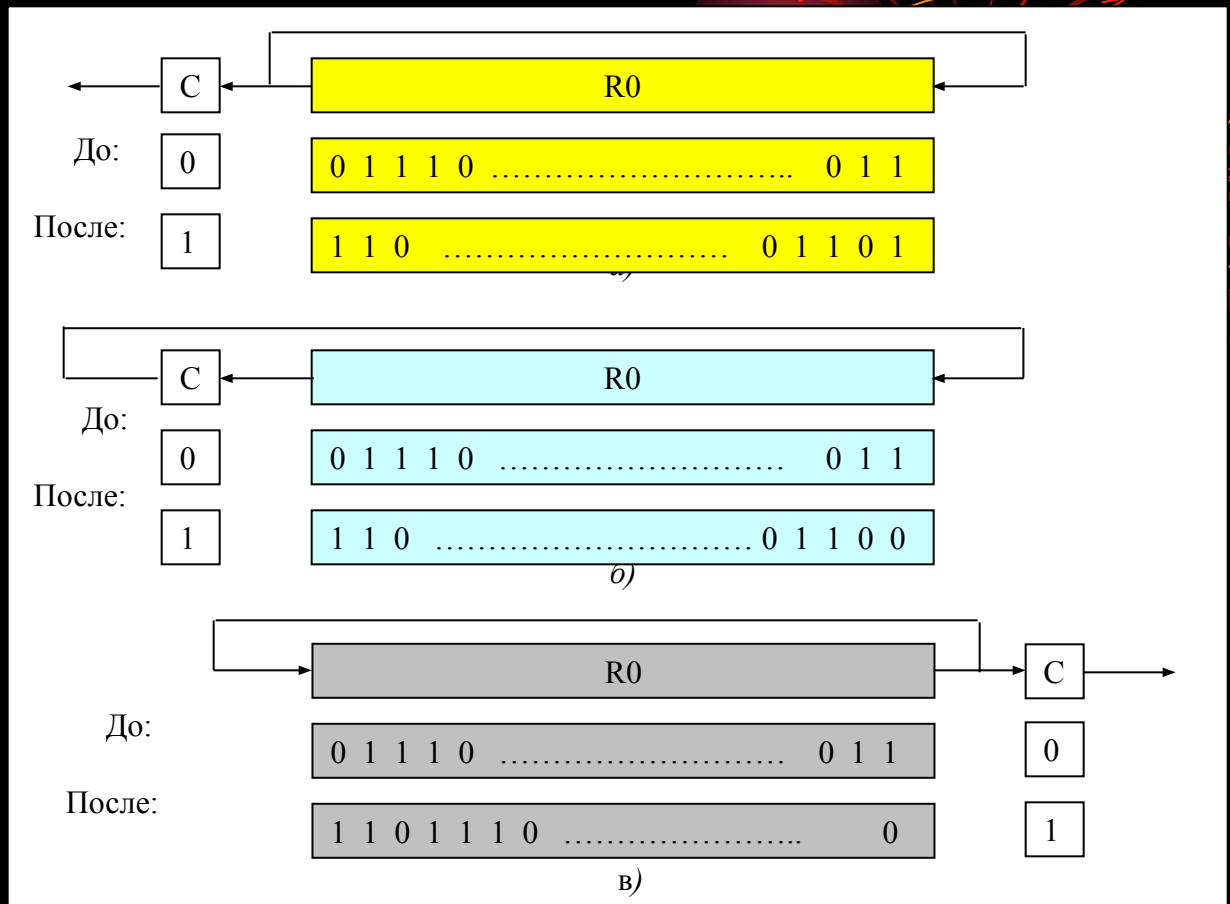


Рис. 21. Команды циклического сдвига: влево без переноса, RotateL, #2,R0 (*а);* влево с переносом, RotateLC #2,R0 (*б);* вправо без переноса RotateR #2,R0 (*в);* вправо с переносом, RotateRC #2,R0 (*г*)

Обратите внимание, что в тех случаях, когда флаг С не участвует в операции, он, тем не менее, содержит последний бит, выдвинутый из операнда. Для определения операций циклического сдвига используются мнемонические обозначения RotateL, RotateLC, RotateR и RotateRC. Указанные команды в основном применяются в арифметических операциях, отличных от операций сложения и вычитания.

Связные списки

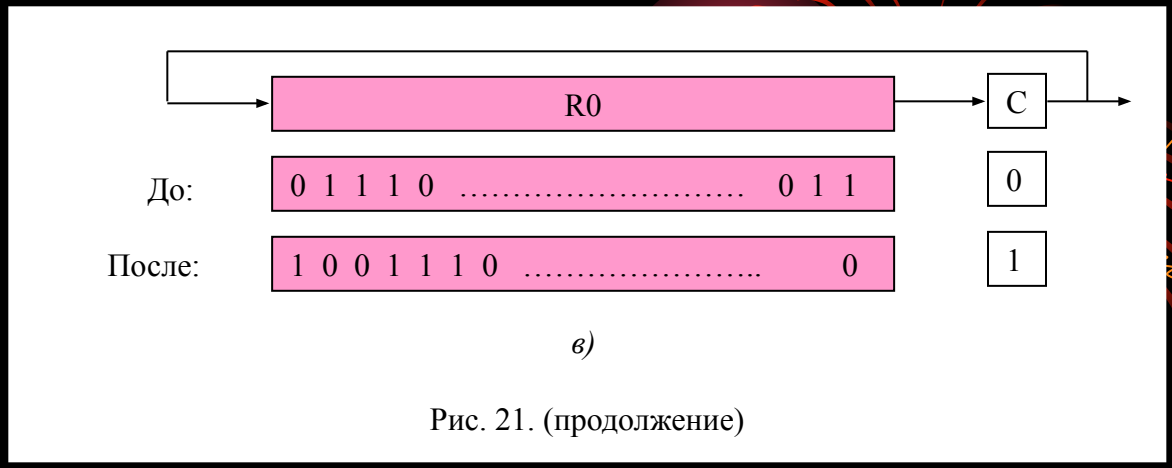


Рис. 21. (продолжение)

Во многих прикладных программах, не предназначенных для выполнения инженерных расчетов, упорядоченный список элементов данных должен быть представлен в памяти таким образом, чтобы в него легко было добавлять новые элементы или удалять из любого места ненужные, сохраняя определенный порядок. Такая структура более универсальна, чем стек или очередь, которые позволяют удалять и добавлять данные только в начало и конец списка. Предположим, мы хотим поддерживать список студентов (см. рис.13) в памяти отсортированным по кодам студентов, а записи о студентах при этом могут добавляться и удаляться. Это позволит выводить на экран и печать список в упорядоченном виде. Если после создания списка студент по какой-то причине будет исключен из группы, в списке останется пустая запись. В случае добавления в список новых оценок или печати такового эту пустую запись придется каждый раз пропускать. В ещё более сложной ситуации мы окажемся, если после создания списка в группу будет включен новый студент. Чтобы список остался упорядоченным, нам придется сдвинуть в нем все записи начиная с номера, следующего за номером добавляемой записи. Точно также при выходе студента из группы можно сдвинуть все записи, следующие за его удаленной записью, чтобы в списке не оставалось пробела.

Обеих этих сложных операций позволяет избежать структура данных, называемая *связным списком.* Как и в обычном списке, каждая запись в связном занимает четыре слова, но последовательные записи необязательно должны занимать последовательные блоки памяти. Для того, чтобы связать записи в упорядоченный список, в каждую из них включают поле *указателя* длиной в одно слово, содержащее адрес следующей записи списка.

Схематическое представление такой структуры данных показано на рисунке 22,*а*. Первую запись связного списка иногда называют его «головой», а последнюю – «хвостом.

Для того, чтобы в список добавить новую запись, между записями i и i+1, нужно скопировать поле указателя из списка i в новую запись, а адрес новой записи поместить в поле указателя записи i. Эта операция схематически показана на рисунке 22, *б*. Для удаления записи i значение из ее поля указателя копируется в поле указателя i-1.

На рисунке 23 приведен пример записей с результатами тестов, связанных между собой по принципу связного списка и отсортированных по кодам студентов.

Длина каждой записи теперь составляет пять слов. В первом слове, определенном в качестве *ключевого* поля, содержится код студента, во-втором – поле указателя, а остальные три слова хранят рез0ультаты тестов. Если слово имеет длину 32 разряда, для списка выделяется область памяти длиной 2000 байт начиная с адреса 1000. В таком Случае каждая запись занимает 20 байт и выделенной области хватает для хранения 100 записей. Студенту, который включается в группу, назначается один из блоков памяти длиной 5 слов. Записям удобно назначать адреса 1000, 1020, 1040, …, 2980, но так делать необязательно. Кроме того, никакой связи между кодами студентов и порядком их включения в группу не существует. Таким образом, блоки записей, упорядоченные по кодам студентов, будут совершенно непредсказуемым образом разбросаны по выделенной для списка области памяти и иметь произвольные адреса из диапазона от 1000 до 2980.

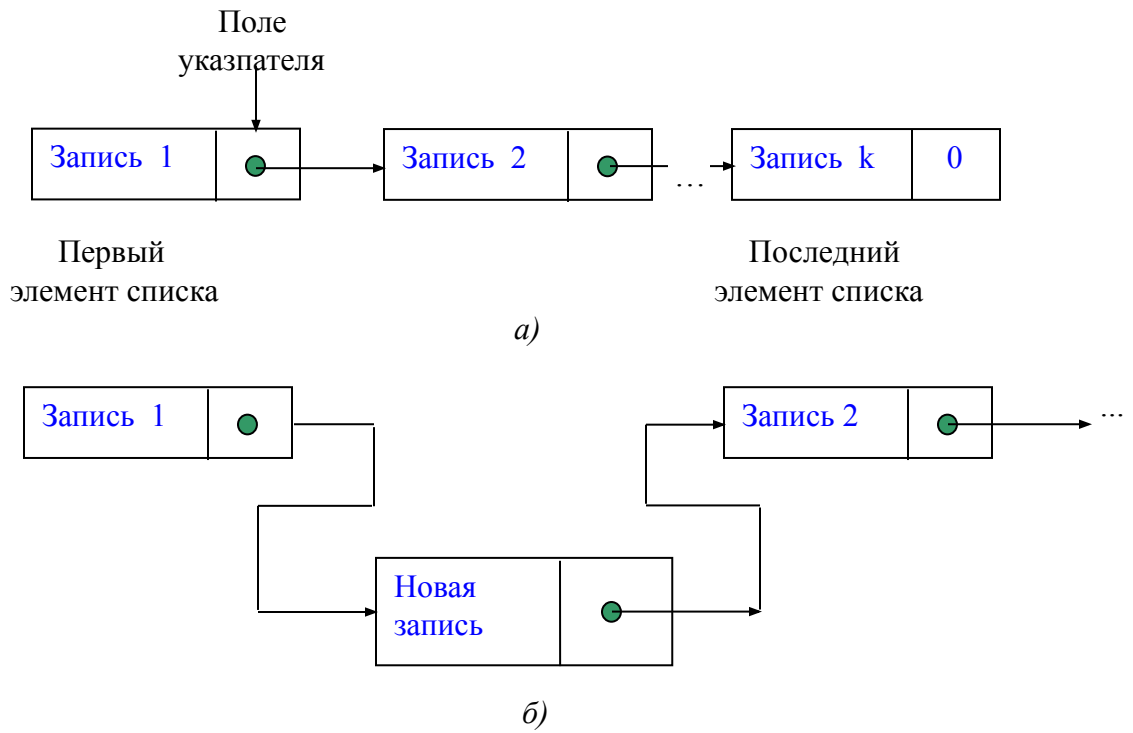


Рис. 22. Связный список: структура связного списка: (а); добавление новой записи между записями 1 и 2 (б)

Запись с наименьшим кодом находится в начале списка, а с наибольшим – в конце. При работе со списком адрес его начала удобно хранить в регистре процессора, называемом *указателем начала списка*.

В нашем примере это адрес 2320. Адрес 1040 в поле указателя первой записи определяет местоположение второй записи. В поле указателя второй записи хранится адрес третьей записи – 1200. Поле указателя последней записи содержит значение 0, обозначающее конец списка. Если список пуст, 0 содержит поле указателя его начала.

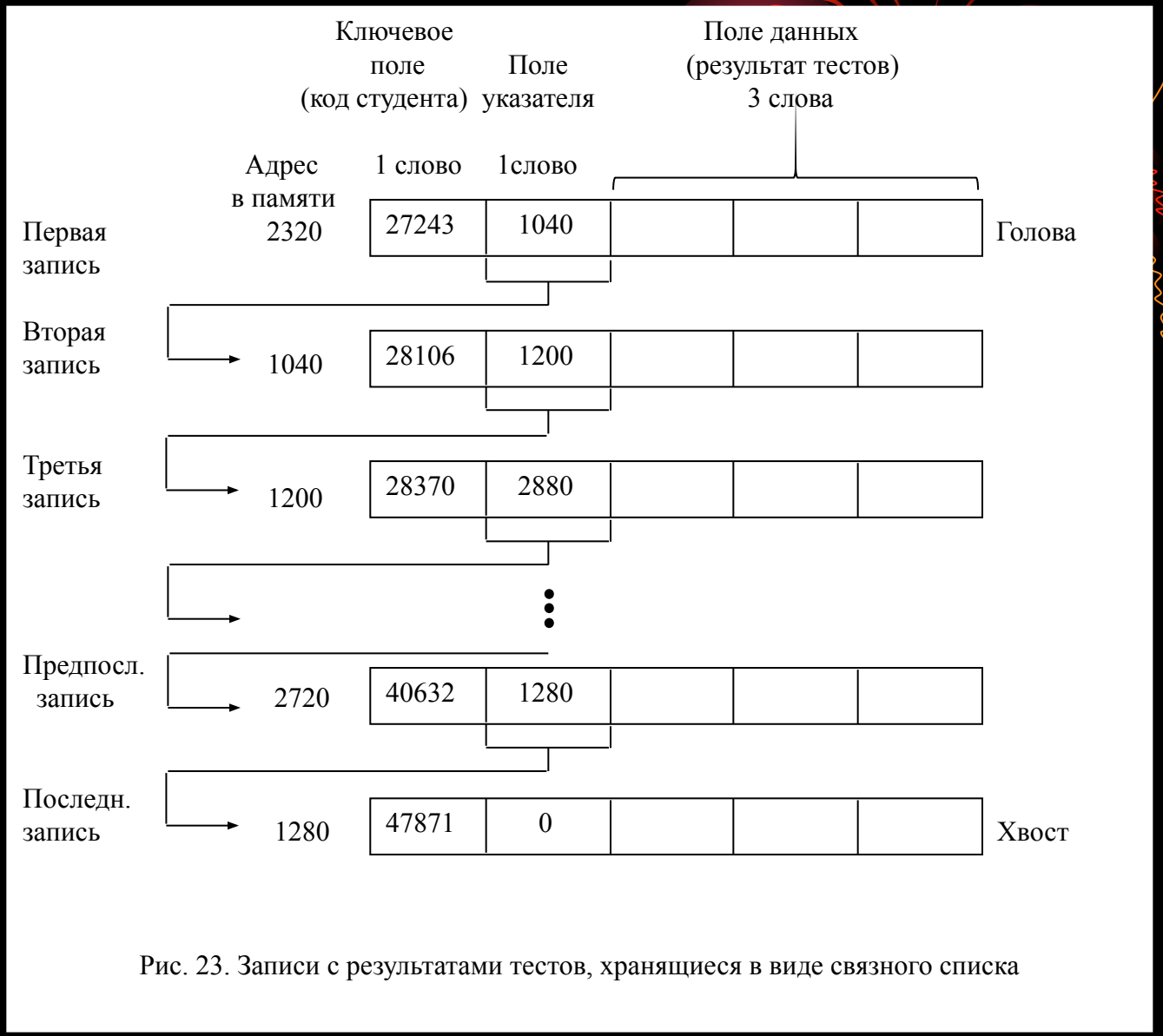
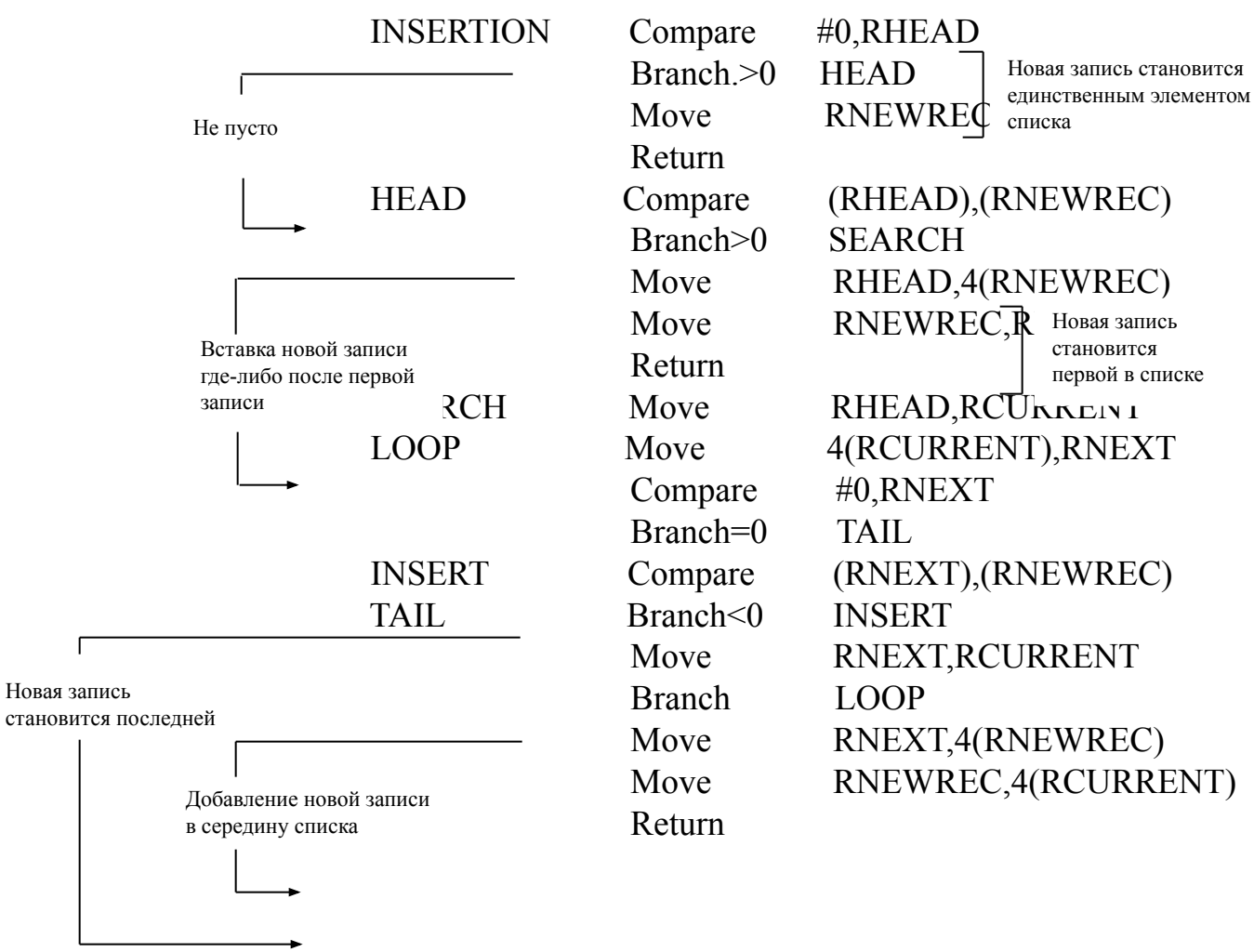


Рис. 23. Записи с результатами тестов, хранящиеся в виде связного списка

Вставка новой записи

А теперь мы более подробно рассмотрим процесс добавления новой записи в список, показанный на рисунке 23. Предположим, что новая запись имеет код 28241, а следующий доступный блок памяти располагается по адресу 2960. Мы сканируем список записей начиная с головной, пока не достигнем записи, код которой будет больше кода добавляемой записи. Это запись с кодом 28370, расположенная по адресу 1200. Теперь введем в поле указателя новой записи значение 1200, а адрес новой записи, 2960, запишем в поле указателя Предыдущей записи, расположенной по адресу 1040, заменив тем самым исходное значение 1200.



24. Подпрограмма для добавления новой записи в связный список

Теперь новая запись помещена на третье место списка, между второй и третьей записями исходного списка. Выполняющая эту операцию подпрограмма приведена на рисунке 23. Она состоит из трех частей, обрабатывающих три возможные ситуации, а именно: текущий список пуст, новая запись становится первой записью непустого списка и новая запись добавляется в любое другое место списка после первой записи. Последняя ситуация включает и тот случай, когда новая запись становится последней записью списка.

Посмотрим, как подпрограмма обрабатывает три указанные ситуации. В ней используется целый ряд регистров процессора, которым мы, для того чтобы облегчить понимание программы, присвоили другие имена, более информативные по сравнению с традиционными R0, R1, R2 и т.д. Указатель начала списка мы назвали RHEAD, а регистр, содержащий адрес новой записи, - RNEWREC. Ещё два регистра, RCURRENT и RNEXT, во время сканирования списка содержат адреса текущей и следующей записей. Поле указателя в новой записи первоначально установлено в 0. Если она становится последней записью списка, значение этого поля не меняется.

Первая пара команд сравнения и перехода проверяет, не пуст ли список. Если список пуст (RHEAD содержит 0), новая запись становится единственным элементом списка и ее адрес просто записывается в RHEAD, после чего выполняется команда возврата. В противном случае вторая пара команд сравнения и перехода проверяет, станет ли новая запись первой в списке. Если станет, две следующие команды пересылки изменяют содержимое её поля указателя и регистра RHEAD, после чего выполняется команда возврата. Если новая запись не станет первой в списке, последняя часть подпрограммы определяет то место списка, куда она должна быть вставлена. Запись вставляется в нужное место с применением двух последних команд пересылки или добавляется в конец списка последней командой пересылки. Для того чтобы упростить эту подпрограмму, мы не стали приводить команды сохранения и восстановления регистров.

Удаление записи

Удаление существующей записи из связного списка – операция более простая. Нам нужно просканировать список и найти в нем запись с заданным кодом, а потом просто изменить значение указателя в предшествующей записи. Подпрограмма, выполняющая эту операцию, приведена на рисунке 25.

Мы предполагаем, что в регистре RIDNUM содержится код удаляемой записи, а регистры RHEAD, RCURRENT и RNEXT выполняют те же функции, что и при добавлении записи. Первая пара команд сравнения и перехода проверяет, является ли подлежащая удалению запись первой записью списка. Если да, эта запись удаляется, для чего значение её поля указателя копируется в RHEAD. Причем, если первая запись была единственной в списке, её поле указателя содержит 0. Поэтому копирование этого указателя в регистр RHEAD означает, что теперь список пуст. Регистры RCURRENT и RNEXT применяются для сканирования списка в поисках удаляемой записи. Когда эта запись будет найдена, вторая команда сравнения и перехода передает управление по адресу DELETE. Запись, на которую указывает регистр RNEXT, будет удалена из списка, для чего её поле указателя будет скопировано в поле указателя предыдущей записи, на которую указывает RCURRENT. Последние две команды пересылки выполняют такое копирование через регистр RTEMP. Если компьютер поддерживает непосредственное копирование из памяти в память, эти две команды можно заменить одной: Move 4(RNEXT),4(RCURRENT).

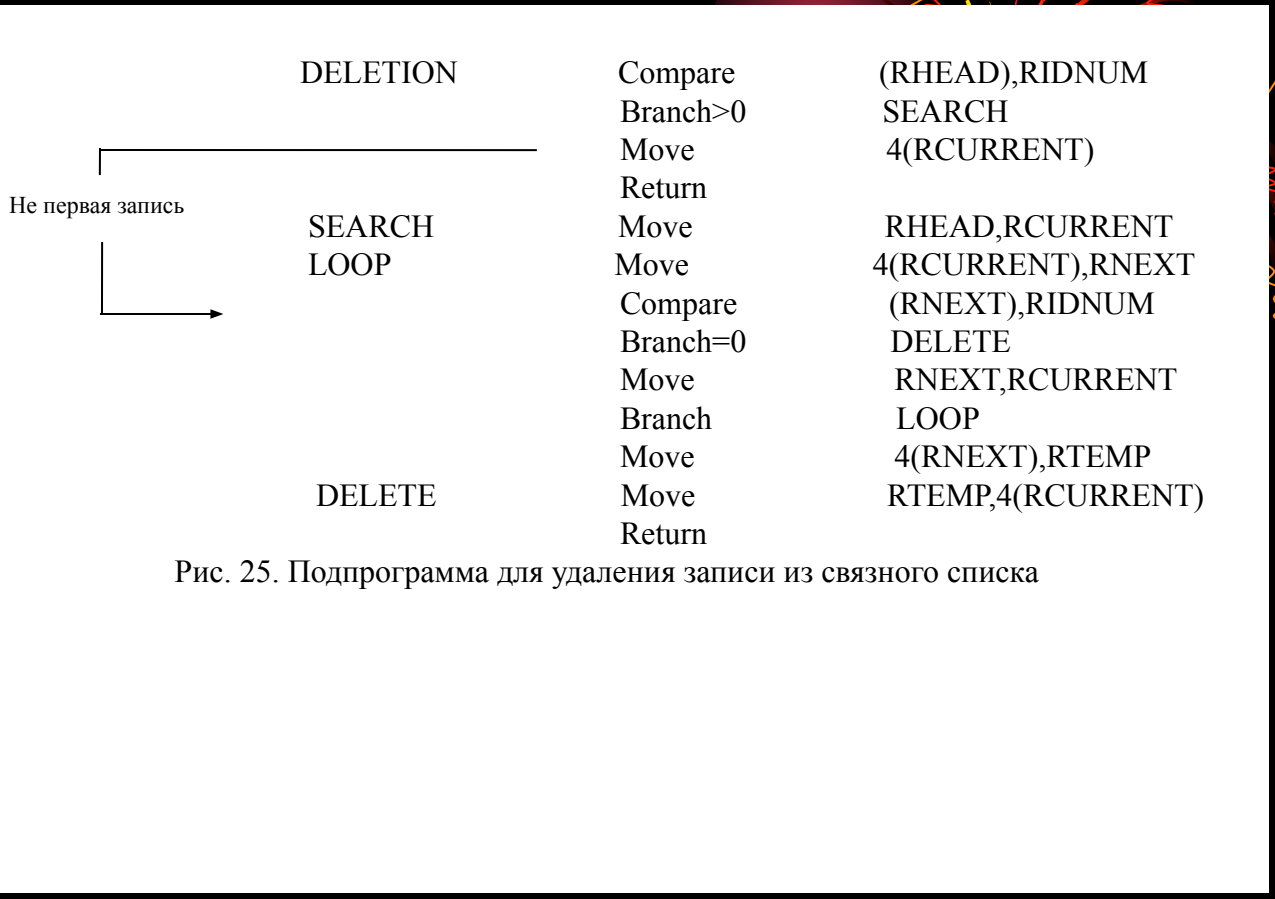
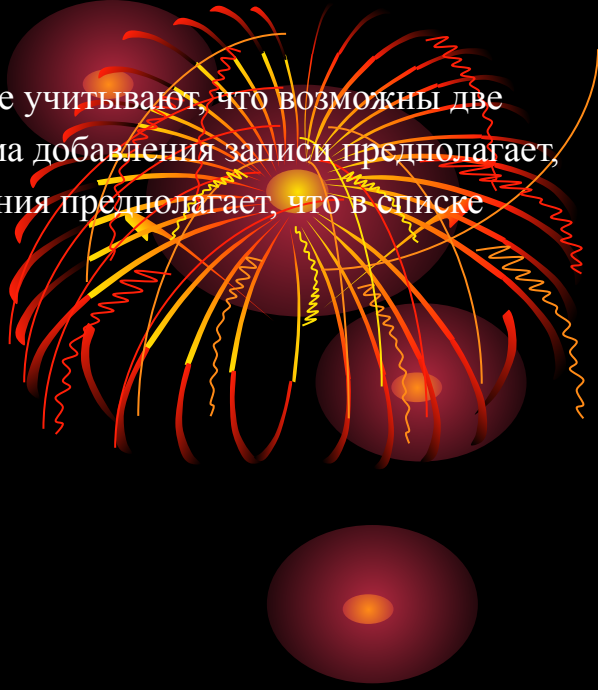


Рис. 25. Подпрограмма для удаления записи из связного списка

Обработка ошибок

Подпрограммы добавления и удаления записи, показанные на рисунках 24 и 25, не учитывают, что возможны две такие ситуации, из-за которых в них наверняка произойдут ошибки. Подпрограмма добавления записи предполагает, что в списке нет записи с таким кодом, как у новой записи, а подпрограмма удаления предполагает, что в списке такая запись имеется.



Ввод – вывод

Одной из важнейших функций компьютера является поддержка устройств ввода-вывода. Благодаря этому оператор, к примеру, может использовать клавиатуру и дисплей для работы с текстом и графикой. Компьютеры применяются для взаимодействия с другими компьютерами через Интернет, а следовательно, предоставляют доступ к информации, находящейся частях земного шара. Роль компьютеров в решении других задач может быть заметной, но не менее важной. Компьютеры используются дома, в учебных аудиториях, на производстве, в различных транспортных, банковских и коммерческих системах – разнообразие сфер их применения просто впечатляет. Данные могут поступать в компьютер от самых разных приборов: сенсорных устройств, видеокамер, в том числе цифровых, микрофонов или даже с пультов пожарной сигнализации. Выходными же данными могут служить направляемый в колонки звуковой сигнал или, закодированная цифровая команда, изменяющая скорость мотора, открывающая вентиль или заставляющая робота выполнить определенное движение. Короче говоря, компьютер должен обладать способностью обмениваться информацией с широким диапазоном устройств в различных окружениях.

Доступ к устройствам ввода-вывода

Простейшая схема подключения устройств ввода-вывода к компьютеру заключается в использовании общей шины. Все устройства, подключенные к шине, могут обмениваться между собой информацией. Обычно шина состоит из трех наборов линий, предназначенных для передачи адресов, данных и управляющих сигналов. Каждому устройству ввода-вывода присваивается уникальный набор адресов. Когда процессор помещает на адресные линии конкретный адрес, распознавшее этот адрес устройство отвечает на команду, помещенную на управляющие линии. Процессор запрашивает либо операцию чтения, либо операцию записи, и запрошенные данные пересылаются по линиям данных. Как уже упоминалось выше. Организация системы ввода-вывода, при которой устройства ввода-вывода и память разделяют одно адресное пространство, называется *вводом-выводом с отображение в память.*

При использовании ввода –вывода с отображением в память любые машинные команды, выполняющие обращение к памяти, могут быть задействованы и для обмена данными с устройствами ввода-вывода.

Шина USB

Объединение компьютеров и коммуникационных технологий привело к настоящей информационной революции.

Современные компьютерные системы включают множество устройств, таких как клавиатуры, микрофоны, цифровые видеокамеры, динамики, дисплеи. И почти все они имеют проводное или беспроводное соединение с Интернетом. Одним из важнейших требований к таким системам является наличие простого и недорогого механизма подключения к компьютеру внешних устройств. Одной из последних разработок в этой области стала универсальная последовательная шина – Universal Serial Bus (USB). USB является промышленным стандартом, разработанным объединенными усилиями ряда компьютерных и коммерческих компаний, к числу которых относятся Compaq, Hewlett-Packard, Intel, Microsoft, Nortel Networks, Philips.

USB поддерживает два режима функционирования, получивших названия низкоскоростной (1,5 Мбит/с) и полноскоростной (12Мбит/с). В последней версии этой спецификации, USB 2,0 введен третий режим, названный высокоскоростным (480 Мбит/с). Шина USB быстро завоевывает популярность на рынке и с появлением высокоскоростного режима может стать основным средством взаимодействия большинства компьютерных устройств.

Разработчики USB ставили перед собой следующие задачи:

- создать простую, дешевую и удобную систему соединения, позволяющую преодолевать сложности, возникающие из-за ограниченного числа имеющихся в компьютерах портов ввода-вывода;
- учесть широкий диапазон параметров, пересылаемых данных, присущих различным устройствам ввода-вывода, в том числе модемам;
- облегчить для пользователей процесс подключения устройств за счет поддержки режима plug-and-play.

Ограниченное количество портов

Параллельный и последовательный порты, имеют универсальные разъемы, через которые к компьютеру можно подключать самые разнообразные низко- и среднескоростные устройства. Однако, по практическим соображениям типичный компьютер имеет всего несколько таких портов. Для добавления нового порта пользователь должен открыть корпус компьютера, чтобы получить доступ к внутренней шине расширения и вставить в ее разъем новую интерфейсную плату. Кроме того, пользователь должен знать, как настроить новое устройство и его программное обеспечение.

Шина USB обеспечивает возможность подключения к компьютеру большего количества устройств в любой момент и без необходимости открывать его корпус.

Характеристики устройств

Типы устройств, которые можно подключать к компьютеру, довольно разнообразны. Скорость, объемы и характеристики процесса обмена данными с таким устройствами тоже бывают очень разными.

Например, клавиатура генерирует один символ каждый раз, когда нажимается какая-либо клавиша, а это может произойти в любой момент. При этом данные немедленно должны быть переданы в компьютер. Поскольку событие нажатия клавиши не синхронизировано ни с одним другим событием в компьютерной системе, генерируемые клавиатурой данные называются *асинхронными*. Более того скорость, с которой они генерируются, очень мала. Она ограничена скоростью работы оператора, который может вводить около 100 байт (символов) в минуту, что составляет менее 1000 бит в секунду.

Существует множество других устройств, которые генерируют данные такого типа. К их числу относятся мыши и игровые манипуляторы.

Имеются и другие источники данных. Многие компьютеры снабжены микрофонами – либо встроенными, либо подключенными в качестве внешних устройств. Воспринимаемый микрофоном звук преобразуется в аналоговый электрический сигнал, который перед обработкой компьютером должен быть преобразован в цифровую форму. Для этого аналоговый сигнал дискретизируется аналого-цифровой преобразователь (АЦП) через определенные промежутки времени (период дискретизации) замеряет характеристики звука и генерирует соответствующие n-разрядные значения. Аналоговый звуковой сигнал, замеряемый в конкретный момент времени, называется отсчетом. Количество битов, n, выбирается исходя из требований точности представления звука. Позже, когда эти данные передаются на динамик, обратный цифро-аналоговый преобразователь (ЦАП) восстанавливает исходный аналоговый сигнал из цифрового формата.

В процессе дискретизации генерируется непрерывный поток числовых значений, поступающих через равные промежутки времени. Этот процесс синхронизируется с помощью тактового сигнала. Процесс, последовательные события которого происходят через равные промежутки времени, называется *изохронным*.

Для того, чтобы при оцифровке звука сохранялись те его составляющие, которые меняются с очень большой скоростью, частота дискретизации должна быть очень высокой. Если она составляет s значений в секунду, максимальная частота звука, фиксируемая в ходе оцифровки, будет равна s/2. Например, человеческая речь адекватно записывается при частоте дискретизации 8 кГц, что позволяет сохранять звуковые частоты до 4 кГц. Для более высоких звуков, которые должны записываться музыкальными системами, требуется большая частота дискретизации. Стандартная частота дискретизации цифрового звука составляет 44,1 кГц. Каждый отсчет передается посредством 4 байт данных, что позволяет представить достаточно широкий диапазон уровней звука (динамический диапазон), необходимый для высококачественного воспроизведения звука. В результате скорость потока данных составляет около 1, 4 Мбит/с.

Для процесса оцифровки голоса или музыки важно сохранить точное соответствие частоты дискретизации записи и воспроизведения звука. Значительное расхождение тактовой частоты при записи и воспроизведении звука совершенно недопустимо. Поэтому механизм пересылки данных между компьютером и музыкальной системой должен обеспечивать строго одинаковые промежутки времени между отсчетами. В противном случае потребуется сложная схема буферизации и повторного тактирования. С другой стороны, вполне допустимы отдельные ошибки и пропущенные отсчеты. Они либо вовсе не замечаются слушателем, либо приводят к возникновению щелчков при воспроизведении записи.

К качеству графических и видеофайлов предъявляются похожие требования, но для их передачи нужна значительно более широкая полоса пропускания канала. Термин «полоса пропускания» означает пропускную способность коммуникационного канала, измеряемую в битах или байтах в секунду. Для воспроизведения видео с качеством коммерческого телевидения составляющие его отдельные изображения должны иметь объем около 16 Кбайт и передаваться со скоростью 30 изображений в секунду, для чего необходима полоса пропускания 44 Мбит/с. Для более качественного видео, такого как HDTV (High Definition TV), требуются более высокие характеристики.

Запоминающие устройства большого объема, в частности, жесткие диски и CD-ROM, предъявляют несколько иные требования к коммуникационным интерфейсам. Их соединение с компьютером должно обладать пропускной способностью порядка 50 Мбит/с. Особенности функционирования диска таковы, что при его работе происходят задержки порядка миллисекунды.

Поэтому небольшие дополнительные задержки при пересылке данных между диском и компьютером не имеют значения, равно как и незначительные колебания частоты.

Технология plug-and-play

Технология plug-and-play предполагает, что новое устройство, например, дополнительный динамик, можно подключить в любое время прямо к работающей системе. Система должна сама обнаружить новое устройство, идентифицировать его и соответствующее ему программное обеспечение (драйвер, а также другие, необходимые для его работы средства), назначить ему адреса и установить логические соединения для взаимодействия с другими устройствами.

Этот принцип налагает немалое количество требований и имеет множество следствий на всех уровнях системы, от аппаратного обеспечения до операционной системы и прикладных программ. Одной из главных задач разработчиков шины USB была именно реализация принципа plug-and-play.

Архитектура USB

Для шины USB выбран последовательный формат данных, обеспечивающий ее наименьшую стоимость и наибольшую гибкость. Тактирующий сигнал и данные кодируются вместе и передаются как единый сигнал. В результате нет никаких ограничений в отношении тактовой частоты или расстояний, связанных со сдвигом данных, благодаря чему становится возможной высокая пропускная способность соединений с высокой тактовой частотой. Как уже упоминалось ранее, шина USB поддерживает три скорости пересылки данных, а именно: 1,5; 12 и 480 Мбит/с, что соответствует нуждам самых разных устройств ввода-вывода.

Для того, чтобы можно было к шине USB одновременно подключать большое количество устройств, удаляемых и подключаемых в любое время, эта шина имеет древовидную структуру (см нижеследующий рисунок). В узлах дерева располагаются устройства, называемые *хабами* и действующие как промежуточные управляющие компоненты между хостом (главным компьютером) и устройствами ввода-вывода. *Корневой хаб* соединяет все дерево с хост-компьютером. Листьями дерева являются устройства ввода-вывода (клавиатура, динамики, соединение с Интернетом, цифровой телевизор и т.п.), в терминологии USB называемые *функциями*.

Показанная на рисунке древовидная структура позволяет соединять множество устройств с помощью простых последовательных соединений «точка-точка». Каждый хаб имеет ряд портов, К которым можно подключать любые Устройства, в том числе и другие хабы. В Нормальном режиме работы хаб копирует Полученное входное сообщение в свои Выходные порты. В результате посланное Компьютером сообщение передается всем устройствам ввода-вывода, но отвечает на него только адресуемое устройство.

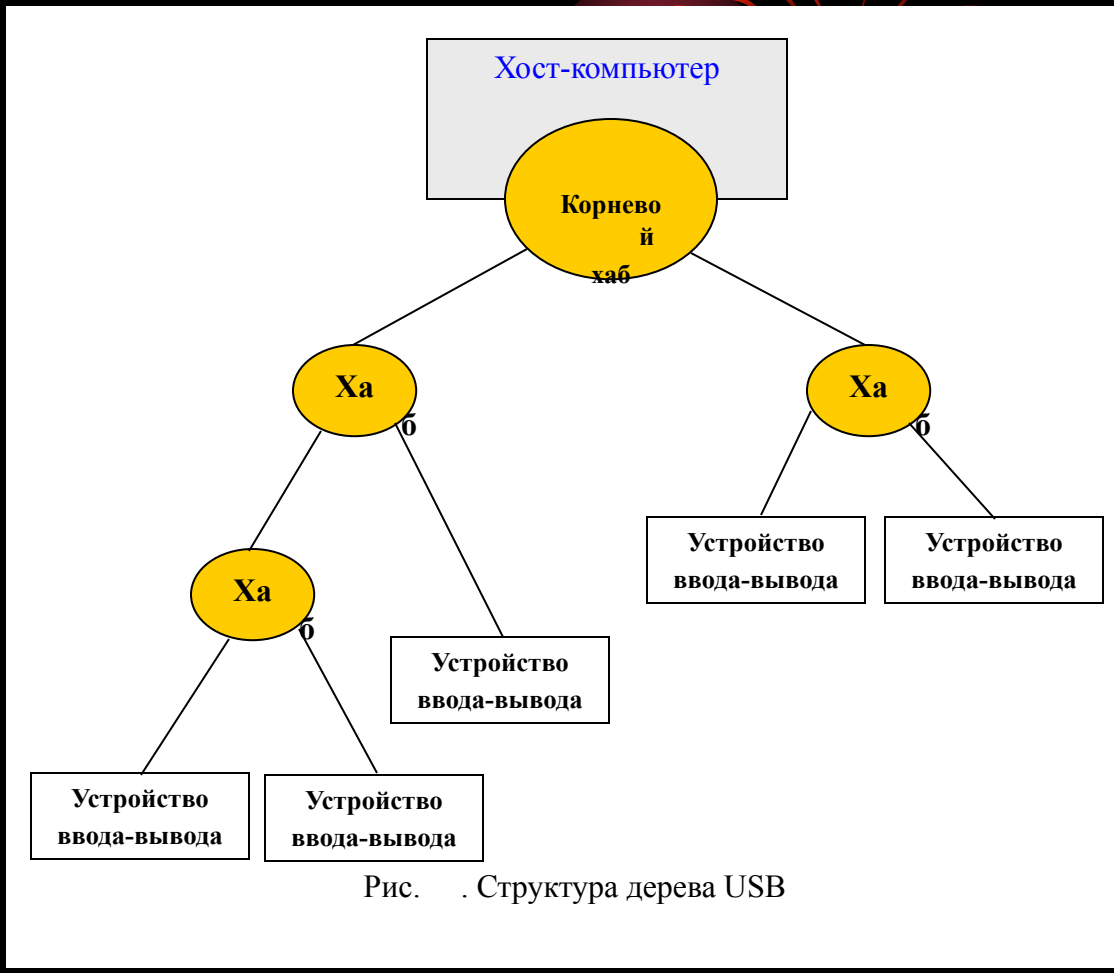
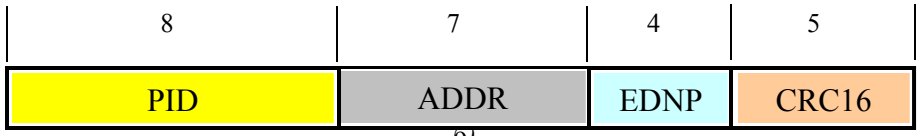


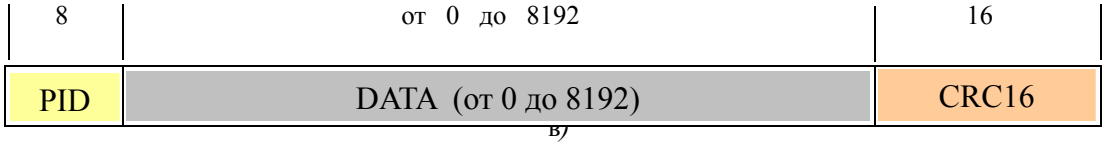
Рис. . Структура дерева USB

PID – Packet Identificator

а)



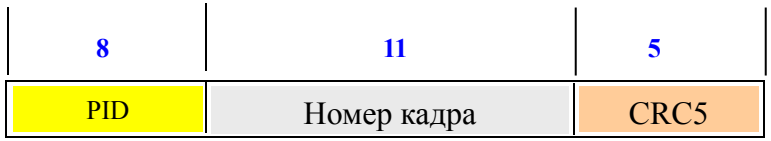
б)



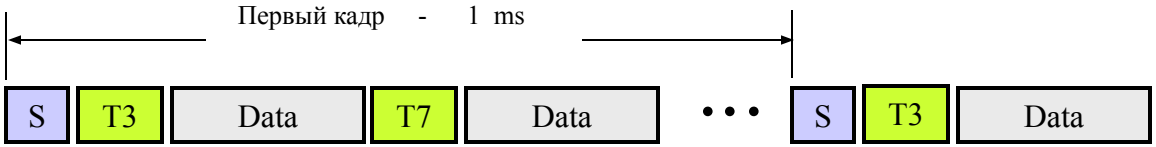
в)

Рис. Форматы пакетов USB: поле идентификатора пакета (а); пакет маркера, IN или OUT (б); пакет данных (в)

SOF - Start Of Frame - пакет начала кадра



а)



S - пакет начала кадра SOF - 11 бит вместо полей AddR и EDNP

T_n - пакет маркера, где n является адресом

D - пакет данных

A - пакет подтверждения ACK

б)

Рис. . Пакет шины USB: пакет SOF (а); структура кадра (б)

Система памяти

Базовые концепции

Максимальный размер памяти, который может использоваться компьютером, определяется его системой адресации. К примеру, 16-разрядный компьютер, генерирующий 16-разрядные адреса, может иметь память объемом до $2^{16} = 64$ Кбайт адресуемых единиц хранения, компьютер, команды которого генерируют 32-разрядные адреса, может использовать память объемом до $2^{32} = 4$ Гбайт адресуемых единиц, а для компьютеров с 40-разрядными адресами доступная память объемом до $2^{40} = 1$Тбайт адресуемых единиц. Количество адресуемых единиц памяти определяет размер ее адресного пространства.

Большинство современных компьютеров адресуют память побайтово. На рисунке 8 показано возможное назначение адресов в 32-разрядном компьютере с байтовой адресацией. При этом может использоваться как прямой порядок адресов, так и обратный.

Обычно память разрабатывается с учетом того, что данные извлекаются и считываются не байтами , а словами.

Само понятие длины слова чаще всего определяется как количество битов, сохраняемых или считываемых за одно обращение к памяти. К примеру, компьютер с байтовой адресацией, команды которого генерируют 32-разрядные адреса, старшие 30 разрядов определяют слово, к которому производится доступ, а два младших разряда определяют положение байта в слове. Извлеченные попутно «ненужные» байты игнорируются процессором. Если же выполняется байтовая операция записи, управляющие схемы памяти должны гарантировать, что остальные байты слова останутся неизменными.

Со стороны системы запоминающее устройство можно рассматривать как черный ящик. Пересылка данных между памятью и процессором выполняется с помощь двух регистров процессора, обычно называемых MAR (Memory Address Register –регистр адреса) и MDR (memory Data Register – регистр данных). Если регистр MAR содержит k битов, а регистр MDR – n битов, память может содержать до 2^k адресуемых единиц хранения. За один цикл обращения к памяти между нею и процессором пересылается n бит данных.

Данные передаются по шине процессора, имеющей k адресных линий и n линий данных. Кроме того, шина содержит линии для управления передачей данных R/W (Read/Write) и MFC (Memory Function Completed). Могут использоваться и другие линии, с помощью которых задается количество пересылаемых данных. Соединение между процессором и памятью схематически показано на рисунке 26.

Чтобы считать данные из памяти, процессор сначала загружает адрес в регистр MAR и устанавливает линию R/W в 1. В ответ память помещает данные на линии данных и подтверждает это действие активизацией сигнала MFC. После получения сигнала MFC процессор загружает данные с линий данных в регистр MDR.

Для того, чтобы записать данные в память, процессор загружает адрес в регистр MAR , а данные в регистр MDR и устанавливает линию R/W в 0, указывая таким образом, что выполняется операция записи.

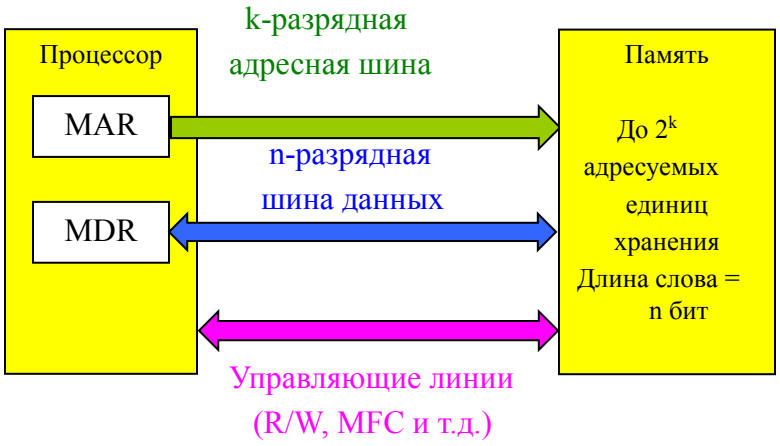


Рис. 26. Организация связи системы памяти с процессором

Если в операциях чтения производится обращение по последовательным адресам, может быть выполнена операция блочной пересылки, при которой памяти передается только один адрес – адрес первого байта блока данных.

Доступ к памяти может синхронизироваться тактовым генератором или специальными сигналами, управляющими пересылкой по шине. Управление операциями чтения из памяти и записи в память осуществляется так же, как и управление операциями ввода и вывода по шине.

Быстродействие памяти характеризуется интервалом времени между инициированием операции и ее завершением, например, временем между сигналами чтения и MFC. Это время определяется как *время доступа к памяти.* Ещё одной важной характеристикой быстродействия памяти является *цикл памяти* – минимальный промежуток времени между моментами начала двух последовательных операций с памятью, например, между двумя последовательными операциями чтения. Цикл памяти обычно немного больше времени доступа (это зависит от особенностей реализации запоминающего устройства).

В запоминающем устройстве, называемом *памятью с произвольным доступом* (Random Access Memory, RAM), на обращение по любому адресу уходит одно и то же время. Этим RAM-память отличается от запоминающих устройств с последовательным или частично-последовательным доступом, таких как магнитные диски или ленты. Время доступа последних зависит от адреса или местоположения данных.

Для реализации основной памяти компьютера используются полупроводниковые интегральные схемы.

Обычно процессор обрабатывает команды и данные быстрее, чем они выбираются из памяти. Поэтому цикл доступа к памяти является узким местом системы. Одним из способов сокращения времени доступа к памяти может стать использование *кэш-памяти*. Это быстрая память небольшого объема, расположенная между сравнительно медленной основной памятью компьютера и процессором. В ней хранятся текущие фрагменты программы и ее данных.

Ещё одной важной концепцией, связанной с организацией памяти, является концепция *виртуальной памяти*. До сих пор мы предполагали, что генерируемые процессором адреса полностью соответствуют физическим ячейкам памяти. Однако, на практике это не всегда так. По определенным причинам, данные могут хранится не по тем адресам, которые заданы в программе. Схемы управления памятью преобразуют указанный в программе адрес в другой адрес, используемый для доступа к физической памяти. В таком случае сгенерированный процессором адрес называют *виртуальным* или *логическим*. Виртуальное адресное пространство определенным образом отображается на физическую память, в которой хранятся данные. Отображение выполняется с помощью специальных управляющих схем памяти, часто называемых *блоком управления памятью*. В ходе выполнения программы функция отображения может меняться в соответствии с системными требованиями.

Виртуальная память предназначена для увеличения видимого компьютером объема физической памяти. Виртуальное адресное пространство и объем расположенных в нем данных могут быть настолько большими, насколько позволяют возможности адресации используемого процессора. Причем в каждый конкретный момент только часть этого адресного пространства отображается на физическую память. Остальные виртуальные адреса соответствуют устройствам массовой памяти – как правило, магнитным дискам. Когда выполняющейся программе требуется доступ к данным, адреса которых не отображаются на реальную память, блок управления памятью изменяет функцию отображения и пересылает данные с диска в основную память.

Поэтому в каждом цикле обращения к памяти система обработки адресов определяет, находится ли адресуемая информация в основной памяти компьютера. Если да, происходит обращение к соответствующему слову памяти и выполнение программы продолжается. Если нет, с диска в память пересылается *страница*, содержащая нужное слово. Эта страница заменяет в памяти какую-либо другую страницу. Поскольку на пересылку страниц между памятью и диском уходит некоторое время, при частом перемещении страниц скорость работы компьютера снижается. Но если выбор заменяемых страниц выполняется продуманно, вероятность того, что необходимые страницы окажутся в основной памяти, возрастает.

Полупроводниковая RAM-память

Полупроводниковая память реализуется в виде микросхем с очень разным быстродействием. Длительность их цикла варьируется от 100 до 10 нс. Появившиеся в конце 1960-х годов полупроводниковые схемы памяти были гораздо дороже памяти на магнитных сердечниках. Однако стремительное развитие технологии сверхбольших интегральных схем (СБИС) позволило быстро снизить их стоимость, и теперь практически вся память реализуется в виде полупроводниковых микросхем.

Организация микросхем памяти

В каждой ячейке памяти, которые обычно объединяются в массивы, может храниться один бит информации. На рисунке 27 показано, как может быть организован такой массив. Каждая строка ячеек составляет слово памяти, а все ячейки строки соединяются общей линией, называемой *линией слова*, которая управляется входящим в состав того же чипа дешифратором адреса. Ячейки каждого столбца соединяются со схемой Sense/Write двумя *линиями битов*. Схемы Sense/Write соединяются линиями записи и считывания данных. Во время операции чтения схемы считывают информацию из ячеек, выбранных с помощью линии слова, и пересылают её на выходные линии данных. А в процессе операции записи они получают входную информацию и сохраняют её в ячейках выбранного слова.

На рисунке 27 показана организация очень маленькой микросхемы памяти, состоящей всего из 16 слов по 8 бит в каждом. Структуру такой микросхемы обозначают как 16 x 8. Входы и выходы данных каждой схемы Sense/Write соединяются с одной двунаправленной линией данных, которая может быть подключена к шине компьютера.

В дополнение к линиям адреса и данных имеются две управляющие линии, R/$\overline{W}$ и CS. Входное значение на линии R/$\overline{W}$ (Read/$\overline{Write}$) определяет требуемую операцию, а входное значение на линии CS (Chip Select – выбор элемента памяти) выбирает одну из микросхем, составляющих систему памяти.

Схема памяти, приведенная на рисунке 27, помещает 128 бит данных и требует 14 внешних линий для адреса, данных и управляющих сигналов. Кроме того, для неё потребуется еще две линии, которые должны соединить её с источником питания и «землей». Рассмотрим чуть большую микросхему памяти, содержащую 1 К (1024) ячеек памяти. Такая схема может быть организована в виде массива 128 x 8, и для неё потребуется 19 внешних соединений. В качестве альтернативы то же количество ячеек можно организовать в массив 1 К x 1. Тогда понадобится 10-разрядный адрес, но зато лишь одна линия данных, а следовательно, 15 внешних соединений. Эта организация показана на рисунке 28. Здесь 10-разрядный адрес делится на две группы по пять разрядов в каждой, представляющих адреса строки и столбца в массиве ячеек.

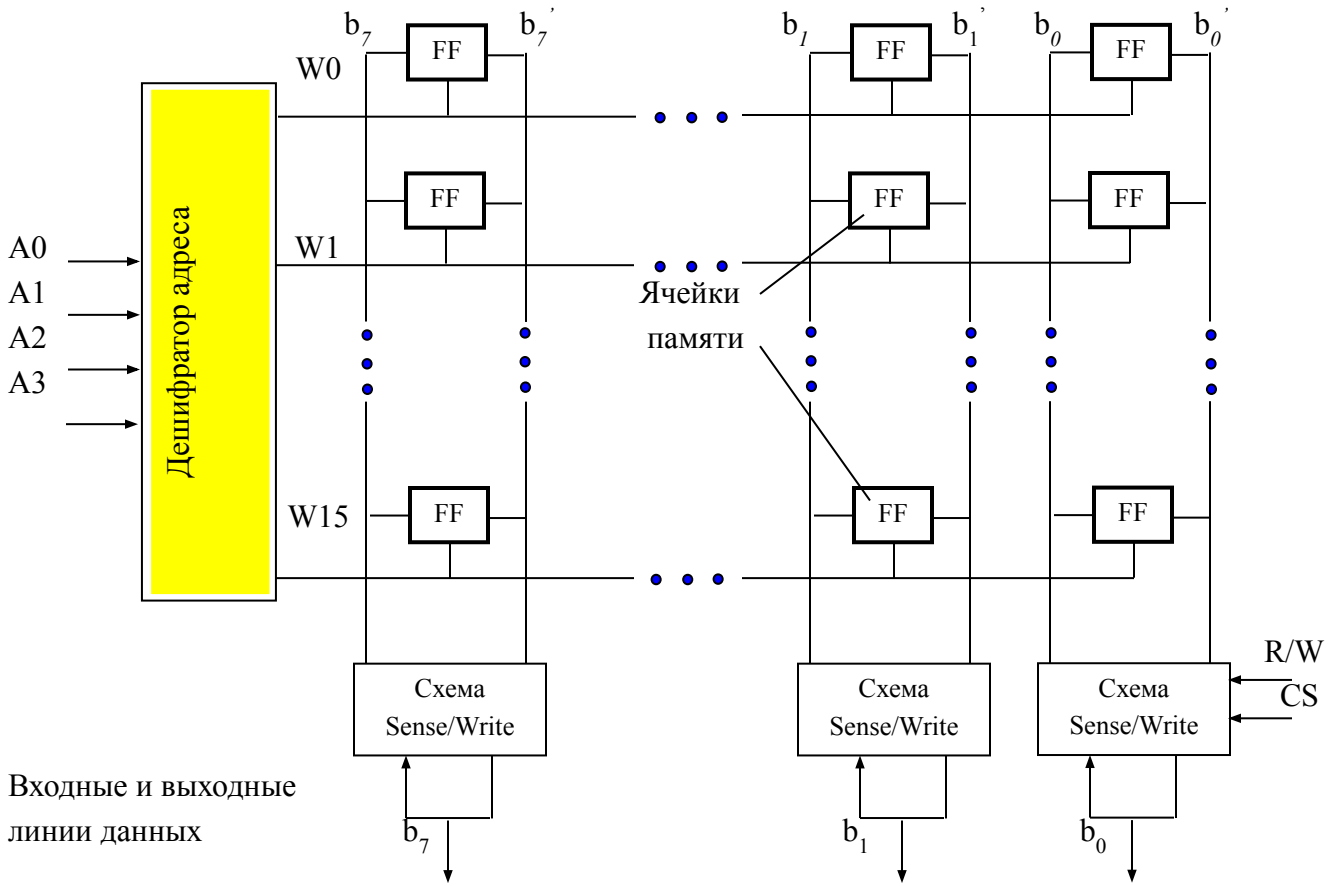


Рис. 27. Организация битовых ячеек в микросхеме памяти

Сигнал адреса строки задает строку из числа 32 параллельно доступных ячеек. однако в соответствии с адресом столбца только одна из них соединяется с внешней линией данных с помощью выходного мультиплексора и входного демультиплексора.

Современные микросхемы памяти содержат гораздо большее количество ячеек, чем представленные на рис. 27 и 28. Такие маленькие схемы мы использовали просто для наглядности. Большие микросхемы имеют ту же структуру, но содержат массивы большего размера и имеют большее число внешних соединений. Например, 4-мегабайтовая микросхема может иметь структуру 512 К x 8, для которой понадобится 19 адресных выводов и 8 выводов для ввода-вывода данных. В настоящее время выпускаются микросхемы емкостью в сотни мегабитов.

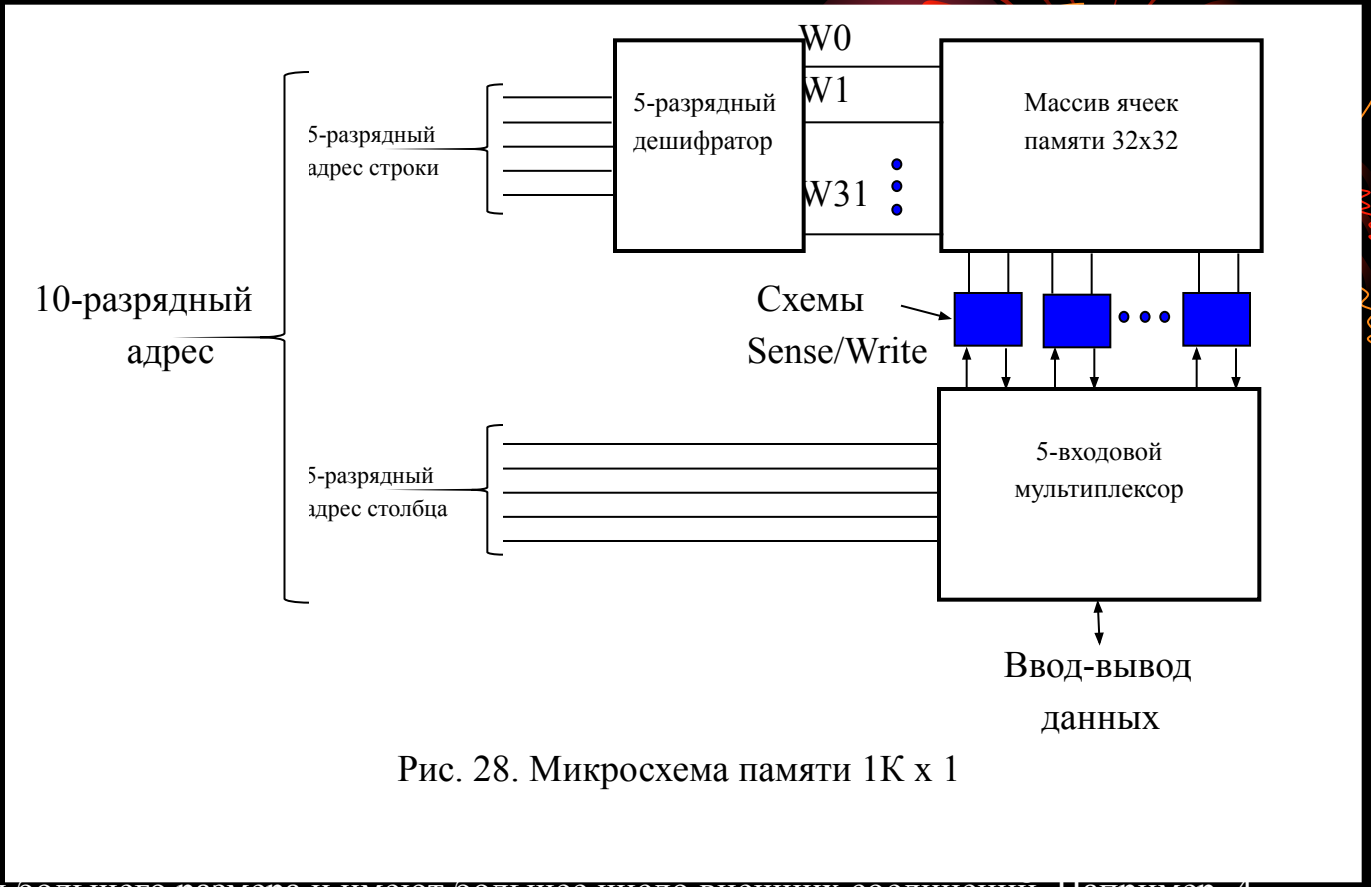


Рис. 28. Микросхема памяти 1К x 1

Статическая память

Память на основе микросхем, которые могут сохранять своё состояние лишь до тех пор, пока к ним подключено питание, называется статической (Static RAM, SRAM). Как может быть реализована такая память, показано на рисунке 29. Перекрестным соединением двух инверторов образуется защелка. Эта защелка соединяется с двумя линиями битов посредством транзисторов T1 и T6. Транзисторы действуют как переключатели, которые могут

Операция чтения

Для того, чтобы прочитать состояние ячейки SRAM, схемы управления памятью активируют линию слова, в результате чего закрываются ключи T1 и T6. Если значение в ячейке равно 1, на линии b наблюдается высокий уровень сигнала, а на линии b' – низкий. Если же значение в ячейке равно 0, эти сигналы меняют свои значения на противоположные. Схемы Sense/Write на концах линий битов выполняют мониторинг состояния линий b *и* b' и соответствующим образом устанавливают выходные сигналы.

Операция записи

Для установки состояния ячейки соответствующее значение помещается на линию b, его дополнение – на линию b', а затем активизируется линия слова. Необходимые сигналы на линии битов генерируются схемой Sense/Write.

Ячейка КМОП

КМОП – реализация ячейки, показана на рисунке 29. Пары транзисторов T2, T3, T4, T5 образуют инверторы защелки. Состояние ячейки считывается или записывается так, как описано выше. Например, в состоянии 1 напряжение в точке X сохраняется высоким за счет того, что транзисторы T2 и T5 включены, а транзисторы T3 и T4 выключены. Таким образом, если транзисторы T1 и T6 включены (замкнуты), напряжение на линиях битов b *и* b' будет соответственно высоким и низким.

В старых КМОП-микросхемах статической памяти напряжение источника питания Vsupply составляло 5 В, в новых низковольтных микросхемах оно равно 3,3 В. Обратите внимание, что для сохранения состояния ячейки необходимо обеспечить постоянное питание. Если питание отключить, содержимое ячейки будет уничтожено. Когда питание будет подано снова, защелка установится в устойчивое состояние, но это не обязательно будет то самое состояние, в каком она была в момент отключения питания. Поэтому микросхемы статической памяти называют энергозависимыми. Основным преимуществом статической КМОП-памяти является очень низкая потребляемая мощность. Через ячейки этой памяти ток идет только в момент обращения к ним. Все остальное время транзисторы T1 и T6, а также по одному транзистору в каждом инверторе выключены, и между источником питания Vsupply и «землей» нет соединения. В современных микросхемах время доступа к статической памяти составляет всего несколько наносекунд. Поэтому статическая память используется в первую очередь там, где особенно нужен такой показатель, как высокая скорость работы.

Асинхронная динамическая память

Статическая RAM работает быстро, но стоит очень дорого, поскольку каждая её ячейка содержит несколько транзисторов. Вот почему выпускается ещё и более дешевая память с более простой конструкцией ячеек. Однако эти ячейки не способны бесконечно долго сохранять свое состояние, поэтому такая память называется *динамической* (Dynamic RAM, DRAM).

В ячейке динамической памяти информация хранится в форме заряда на конденсаторе, и этот заряд может сохраняться всего несколько десятков миллисекунд. Поскольку ячейка памяти должна хранить информацию гораздо дольше, её содержимое должно периодически обновляться путем восстановления заряда на конденсаторе. На рисунке 30 показан пример ячейки динамической памяти, состоящей из конденсатора C и транзистора T. Для записи информации в эту ячейку включается транзистор T и на линию бита подается соответствующее напряжение. В результате на конденсаторе образуется определенный заряд.

После выключения транзистора конденсатор начинает разряжаться. Это происходит из-за его собственного сопротивления утечки, а также из-за того, что после выключения транзистор продолжает слабо проводить ток (измеряется в пикоамперах).

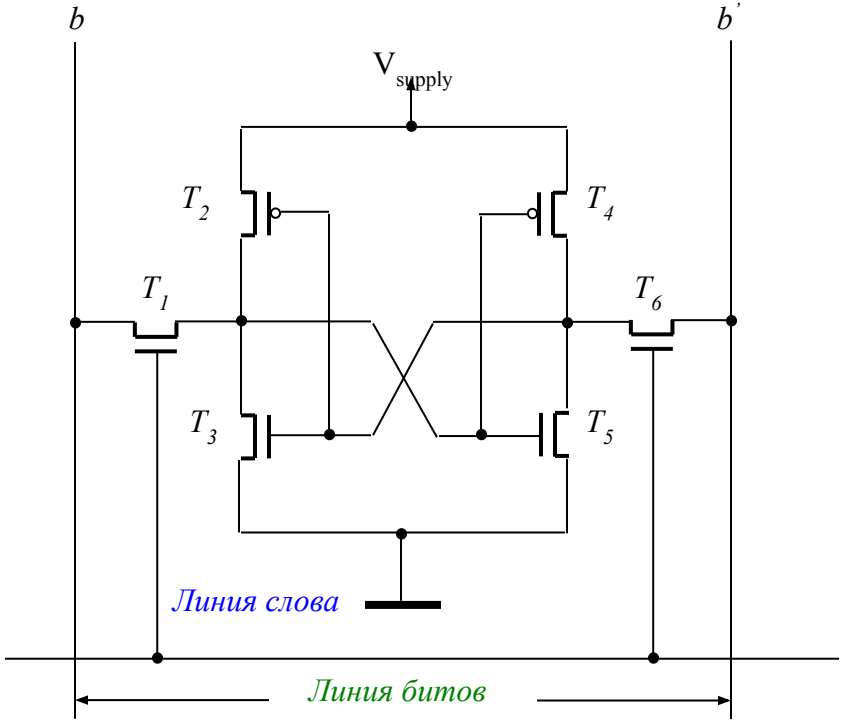


Рис. 29. Пример ячейки памяти КМОП

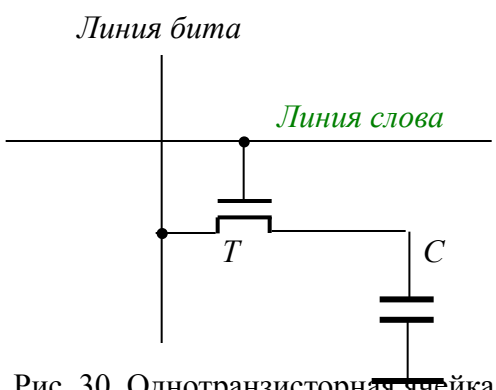


Рис. 30. Однотранзисторная ячейка динамической памяти

Полученная информация не содержит ошибок лишь в том случае, если она считывается из ячеек до того, как заряд конденсатора падает ниже определенного порогового значения. Операция чтения производится, когда транзистор выбранной ячейки включен. Соединенный с линией бита усилитель считывания определяет, превышает ли заряд конденсатора пороговое значение. Если да, он подает на линию бита напряжение, соответствующее значению 1. В результате конденсатор заряжается до напряжения, также соответствующего 1. Если заряд конденсатора ниже порогового значения, усилитель считывания снижает напряжение на линии бита до уровня «земли», обеспечивая тем самым отсутствие заряда (логическое значение 0) на конденсаторе. Таким образом, в процессе считывания содержимое ячейки автоматически обновляется. Все ячейки выбранной строки считываются одновременно, в результате чего обновляется содержимое всей строки.

На рисунке 31 показана 16 мегабитная микросхема DRAM конфигурации 2M x 8. Её ячейки организованы в массив 4 К x 4 К, в котором 4096 ячеек каждой строки разделены на 512 групп по 8 ячеек, так что в одной строке может храниться 512 байт данных. Следовательно, для выбора строки требуется 12 адресных разрядов. Ещё 9 разрядов необходимо для выбора в строке группы из 8 бит. Значит для доступа к байту в такой микросхеме нужен 21-разрядный адрес. Старшие 12 и младшие 9 разрядов адреса составляют адреса строки и столбца байта. Для сокращения количества выводов микросхемы адреса строки и столбца мультиплексируются на 12 выводов.

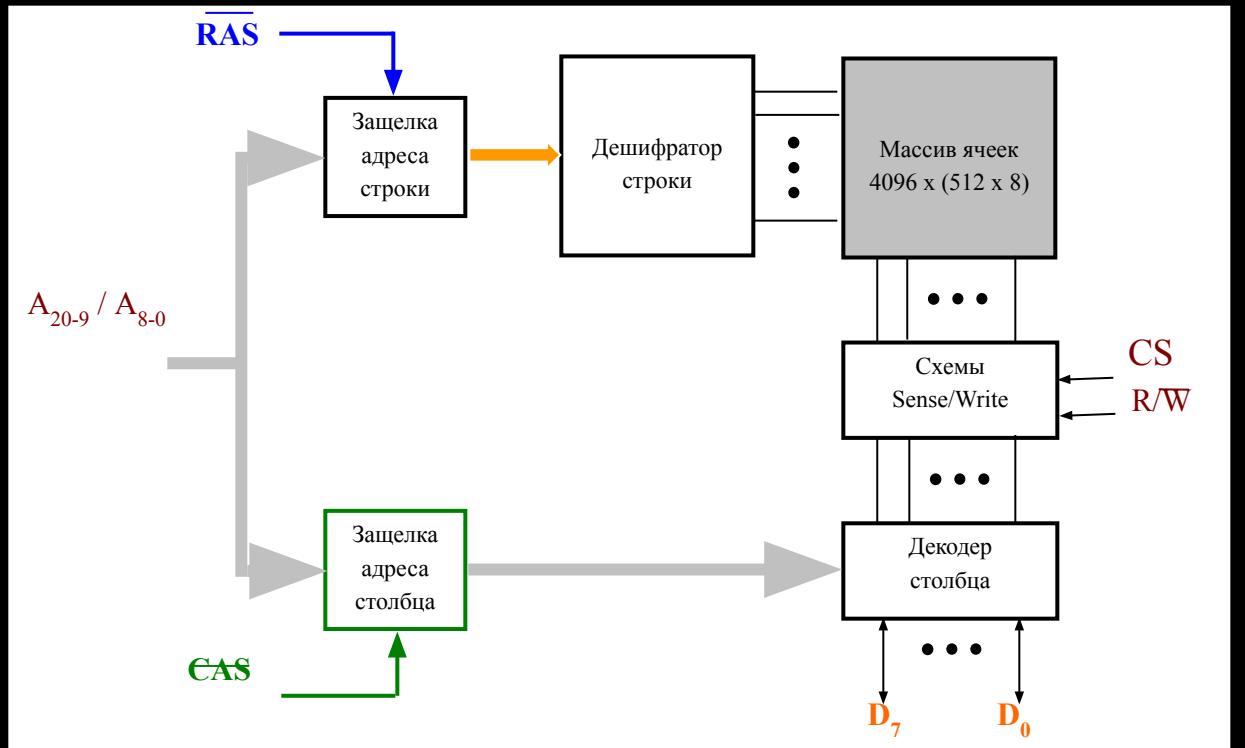


Рис. 31. Внутренняя организация микросхемы динамической памяти 2M x 8

В процессе операции чтения или записи сначала на адресные выводы микросхемы подается адрес строки. В ответ на входной сигнал RAS (Row Address Strobe – строб адреса строки) он загружается в защелку адреса строки. Затем инициируется операция чтения, в ходе которой считываются и обновляются ячейки выбранной строки. Через некоторое время после загрузки адреса строки на адресные выводы подается адрес столбца, который загружается в защелку адреса столбца в ответ на сигнал CAS Column Address Strobe – строб адреса столбца). Информация из этой защелки декодируется и выбирается соответствующая группа из 8 схем Sense / Write. Если управляющий сигнал R / W указывает на операцию считывания, выходные значения выбранных схем пересылаются на линии данных, D_{7-0}. Для операции записи информации с линий D_{7-0} пересылается в схемы. Затем она используется для перезаписи содержимого указанных ячеек в соответствующих 8 столбцах. В коммерческих микросхемах активизации сигналов RAS и CAS соответствует низкий уровень напряжения, так что стробирование адреса выполняется при переходе соответствующего сигнала от высокого уровня к низкому. На схемах эти сигналы обозначаются как $\overline{RAS}$ и $\overline{CAS}$.

Подача адреса строки в ходе операции считывания или записи приводит к чтению и обновлению всех ячеек этой строки. Для того чтобы поддержать содержимое памяти DRAM, нужно периодически обращаться к каждой её строке. Обычно эта работа автоматически выполняется с помощь специальной схемы, называемой *схемой регенерации* (каждые 64ms). Схемы регенерации часто интегрируют прямо в микросхемы динамической памяти, чтобы их динамическая природа была практически невидимой для пользователя.

Описанная в этом разделе динамическая память управляется в асинхронном режиме. Она тактируется управляющими сигналами RAS и CAS, которые генерируются специальной схемой управления памятью. При этом процессор должен учитывать задержку ответа памяти. Такая память называется *асинхронной* DRAM. Благодаря своей высокой емкости (от 1 до 256 Мбит) и дешевизне микросхемы DRAM широко используются в запоминающих устройствах компьютеров. Ведется работа по созданию и более емких микросхем. Для сокращения количества микросхем в компьютере DRAM организуется таким образом, чтобы при выполнении операции чтения или записи биты пересылались параллельно. Микросхемы имеют разную организацию, благодаря чему из них можно свободно компоновать любые системы памяти. Например, микросхема объемом 64 Мбит может быть организована как 16 M x 4, 8 M x 8 или 4 M x 16.

Быстрый постраничный объем

При обращении к микросхеме DRAM, показанной на рисунке 31, считывается содержимое всех 4096 ячеек выбранной строки, но на линии данных D_{7-0} помещаются только 8 бит. Этот байт выбирается битами A_{8-0} адреса столбца. Данную схему можно немного модифицировать, что позволит обращаться к другим байтам той же строки, не выбирая её повторно. С этой целью на выход усилителя считывания каждого столбца нужно добавить по защелке.

При выборе нового адреса строки будут устанавливаться защелки для всех её битов. Теперь для помещения нужного байта на выходные линии данных достаточно установить соответствующий адрес столбца.

Обычно байты пересылаются последовательно, в порядке возрастания их адресов. Для того, чтобы как можно быстрее переслать блок таких данных, нужно под управлением ряда сигналов CAS прямо в схеме генерировать последовательные номера столбцов. Такой режим блочной пересылки называется *быстрым постраничным режимом* (Fast Page Mode, FPM). Согласно популярной сленговой терминологии, небольшие группы байтов называются блоками, а большие – страницами. Ускоренная блочная пересылка данных особенно полезна в тех системах, где данные большими массивами пересылаются в память и из памяти, скажем, в графических терминалах. В компьютерах общего назначения она применяется для пересылки данных меду основной памятью и кэшем.

Синхронная DRAM

Результатом разработок в области технологий памяти стало создание DRAM, синхронизируемой тактовым сигналом. Она получила название *синхронная DRAM* (Synchroous DRAM, SDRAM), а её структура показана на рисунке 32. Массив ячеек в ней точно такой же, как в асинхронной DRAM. Линии адреса и данных буферизируются посредством регистров. Выход каждого усилителя считывания соединен с защелкой – обратите на это особое внимание. В ходе операции чтения в эти защелки загружается содержимое всех ячеек выбранной строки. Однако, если обращение в строке выполняется только с целью регенерации данных, содержимое защелок не меняется – производится только регенерация ячеек. Данные из защелок, соответствующих выбранным столбцам, пересылаются в выходные регистры данных, откуда они могут быть прочитаны через выводы, предназначенные для выходных данных.

Память SDRAM может функционировать в нескольких режимах, определяемых управляющей информацией в регистре *режима*. Например, могут задаваться пакетные операции для передачи различных объемов данных. В пакетных операциях применяется описанный выше режим блочной пересылки данных FPM. В SDRAM для выбора последовательных столбцов не обязательно использовать внешние импульсы на линии CAS. Управляющие сигналы можно генерировать прямо внутри схемы – на основе значений счетчика столбцов и тактового сигнала. В этом случае новые данные помещаются на линии данных на каждом такте. Все действия выполняются на переднем фронте тактового сигнала.

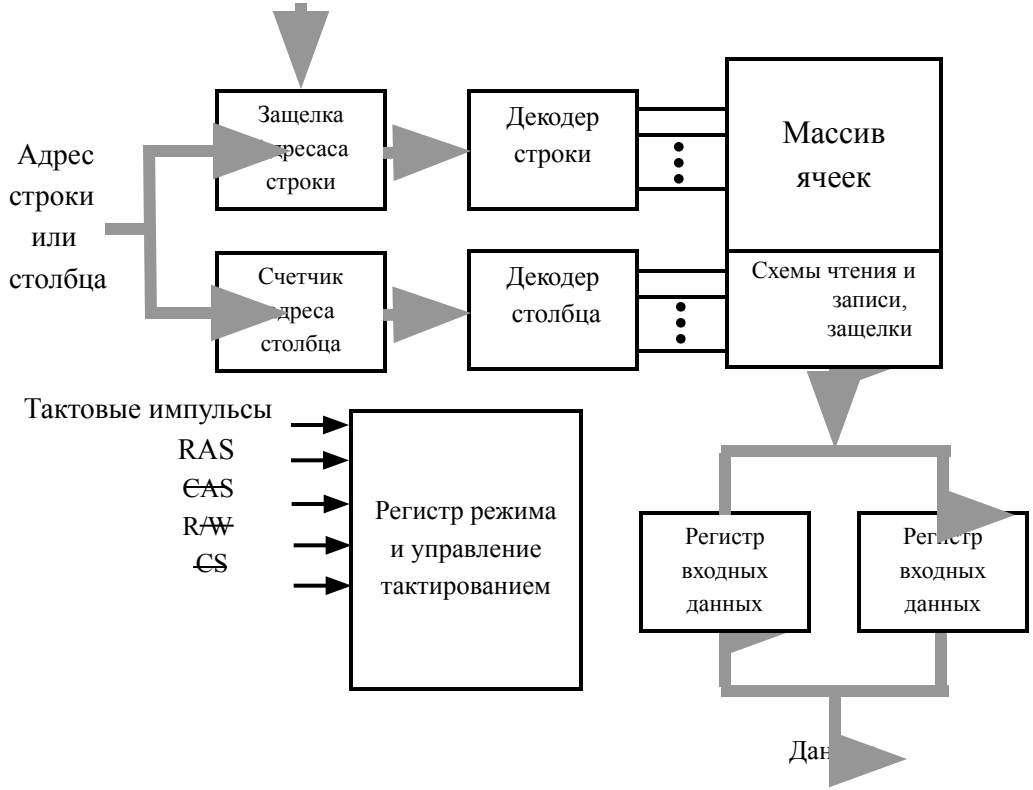


Рис. 32. Синхронная DRAM

На рисунке 33 приведена временная диаграмма типичной операции пакетного чтения длительностью 4 такта. Первым делом в ходе этой операции по сигналу RAS фиксируется адрес строки. На активизацию выбранной строки уходит два или три такта (на рисунке показано два такта). Затем по сигналу CAS фиксируется адрес столбца. После задержки в один такт на линию данных помещается первый набор битов данных. Для обращения к следующим трем наборам битов выбранной строки, помещаемым на линии данных на следующих трех тактах, SDRAM автоматически увеличивает адрес столбца.

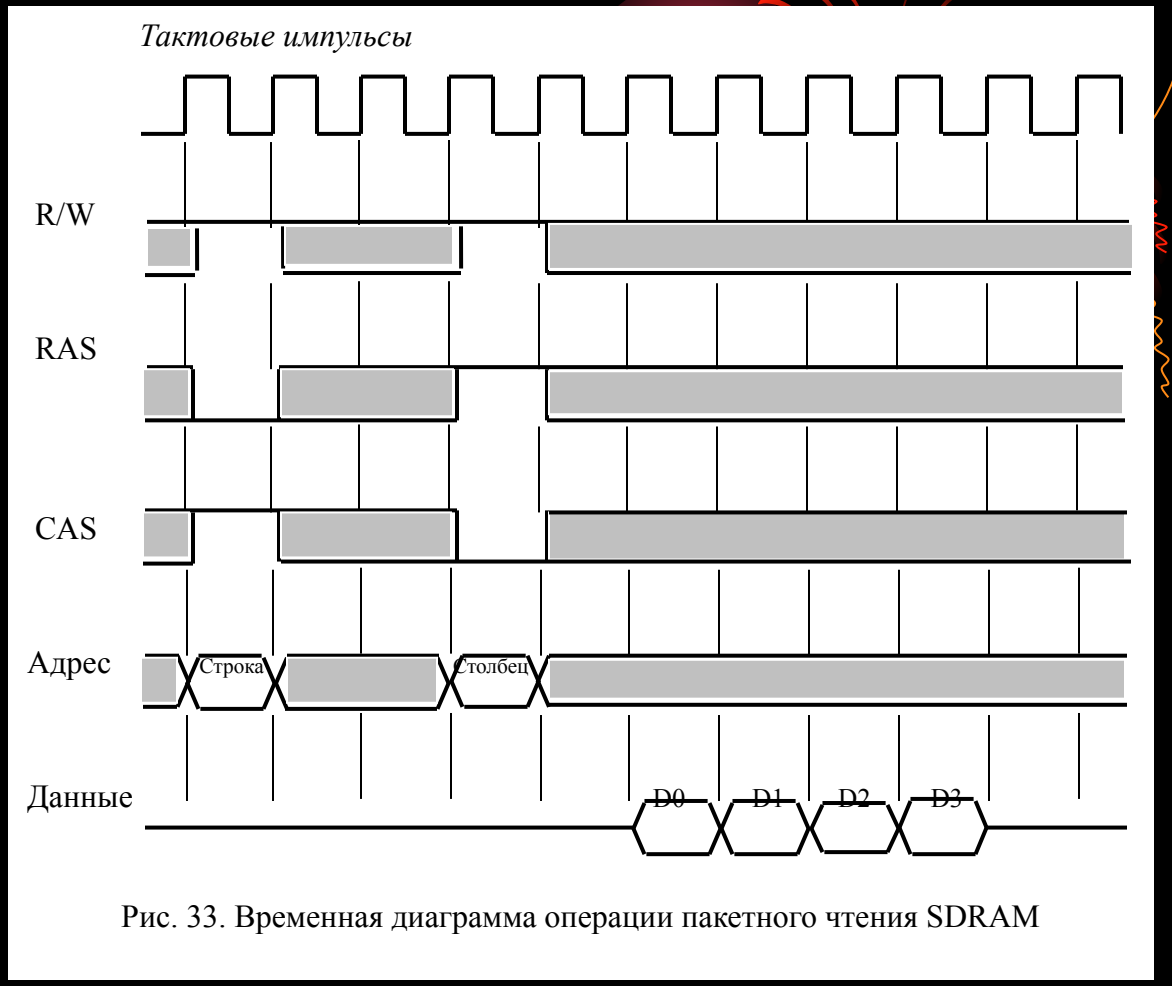


Рис. 33. Временная диаграмма операции пакетного чтения SDRAM

В SDRAM имеется встроенная схема регенерации. В состав этой схемы входит серийных процессоров. Например, Intel определила спецификации шин PC100 и PC133, согласно которым системная шина (к ней подключена основная память) управляется тактовым сигналом с частотой 100 и 133 МГц. Поэтому ведущие производители микросхем памяти выпускают чипы SDRAM с частотой 100 и 133 МГц.

Структура памяти большого объема

Рассмотрим память, которая сконструирована из отдельных микросхем, соединенных в крупные запоминающие устройства. На рисунке 34 показано, как реализовать такую память на основе микросхем статической памяти 512К x 8. Каждый столбец на этом рисунке состоит из четырех микросхем, содержащих восемь последовательных битов каждого слова. Четыре таких набора составляют память 2 М x 32. У каждой микросхемы имеется управляющий вход, называемый CS (Chip Select – выбор микросхемы). Когда на этот вход подается 1, микросхема может принимать данные или помещать их на свои линии данных. Выход данных любой микросхемы имеет три состояния. В каждый конкретный момент только одна микросхема помещает данные на выходные линии данных, а выходы всех остальных микросхем находятся в высокоимпедансном состоянии. Для выбора 32-разрядного слова из такой памяти необходим 21 адресный разряд. Два старших разряда определяют, какой из четырех управляющих сигналов CS следует активизировать, а оставшиеся 19 разрядов применяются для доступа к конкретному байту заданной строки внутри каждой микросхемы. Входы R/W всех микросхем соединяются вместе, образуя единый управляющий вход, не показанный на рисунке.

Системы динамической памяти

Большие системы динамической памяти имею ту же структуру, что и память, представленная на рис. 34. Однако, физически они чаще выполняются в виде более удобных модулей памяти.

Современным компьютерам необходима очень большая память. Даже маленький персональный компьютер, как правило, имеет хотя бы 32 Мбайта памяти, а типичная рабочая станция – как минимум 128 Мбайт. Чем больше основная память компьютера, тем выше его производительность, поскольку в памяти может хранится большее количество программ и обрабатываемых ими данных, а значит, меньше придется обращаться к внешней памяти. Но, если все необходимые микросхемы DRAM будут размещены прямо на основной системной печатной плате, где содержится процессор (её часто называют материнской платой), они займут слишком много места. Кроме того, будет затруднено дальнейшее наращивание памяти, поскольку для добавляемой памяти придется выделить дополнительное место на плате, а для установки микросхем нужно будет подвести соединения ко всем разъемам. Поэтому были разработаны модули памяти большого объема, называемые SIMM (Single In-Line Memory module -

модуль памяти с однорядным расположением выводов) и DIMM (dual In Line Memory Module – модуль памяти с двух рядным расположением выводов). Такие модули представляют собой маленькие платы с наборами микросхем памяти, вертикально устанавливаемые в специальные разъемы на материнской плате. Выпускаемые в настоящее время модули SIMM и DIMM имеют разную емкость, но устанавливаются в разъемы одного размера. Например. В один и тот же 100-контактный разъем можно установить модуль DIMM емкостью 4 M x 32, 16 M x32 или 32 M x 32. Аналогичным образом, в один и тот же 168-контактный разъем можно установить модуль DIMM емкостью 8 M x 64, 16 M x 64 или 64 M x 72. Такие модули занимают на материнской плате очень мало места и позволяют легко увеличивать объем памяти, ведь их ничего не стоит заменить модулями большей емкости.

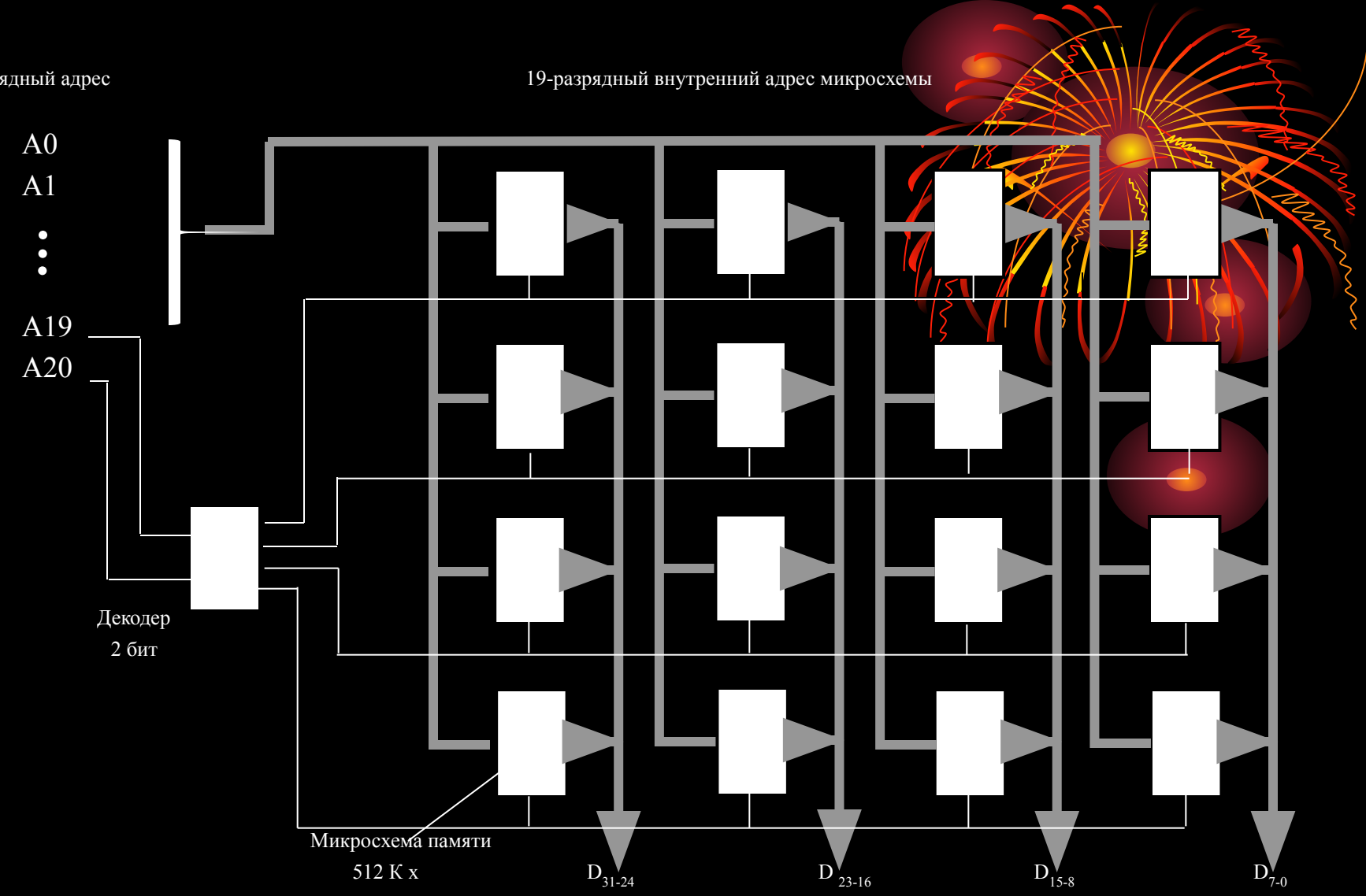


Рис. 34. Организация модуля памяти 2 М х 32 на основе микросхем статической памяти 512 к х 8

Процессор
Базовые концепции

Для выполнения программы процессор по одной выбирает команды из памяти и выполняет определяемые ими действия. команды выбираются из последовательных адресов памяти, пока не встретится команда перехода или ветвления. Для этого в счетчике команд PC, отслеживается адрес очередной, подлежащей выполнению команды. После выборки этой команды содержимое регистра PC обновляется, чтобы он указывал на следующую команду в памяти в порядке расположения адресов. Команда ветвления может загрузить в PC другой адрес.

Ещё одним важнейшим регистром процессора, связанным с выполнением команд, является регистр команды, IR. Предположим, что каждая команда имеет длину 4 байта и хранится в одном слове памяти. Для её выполнения процессор должен произвести следующие шаги.

1. Выбрать из памяти слово, на которое указывает PC. Содержимое этого слова интерпретируется как команда и загружается в регистр IR. Символически это можно записать так:

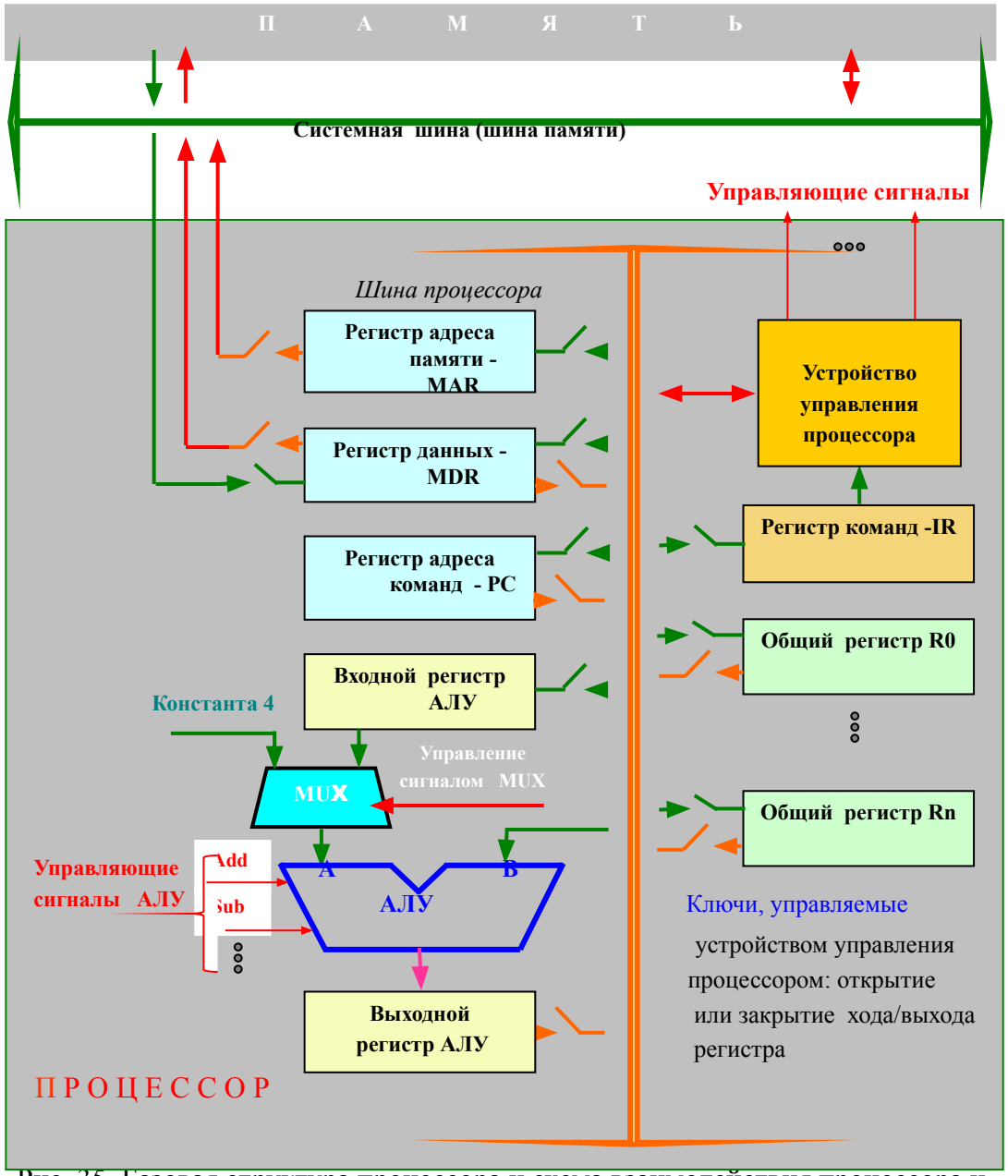


Рис. 35. Базовая структура процессора и схема взаимодействия процессора и памяти

IR □ [[PC]]

2. Если память адресуется побайтово, следует увеличить содержимое регистра PC на 4

PC □ [PC] + 4

3. Выполнить действия, определяемые командой, которая находится в IR.

Если команда занимает более одного слова, шаги 1 и 2 повторяются столько раз, сколько нужно для выборки всей команды. Эти два шага обычно называют *фазой выборки,* а шаг 3 составляет *фазу выполнения.*

Для детального изучения указанных операций нам прежде всего нужно проанализировать внутреннюю структуру процессора. Его организация и связи между блоками, как вы помните, могут быть разными. На рисунке 35 показана архитектура процессора, при которой арифметико-логическое устройство (АЛУ) и все регистры соединены общей шиной. Это внутренняя шина процессора, которую не следует путать с внешней шиной, соединяющей процессор с основной памятью и устройствами ввода-вывода.

Линии данных и адреса внешней шины памяти на рисунке 35 соединены с внутренней шиной процессора через регистр данных памяти, MDR, и регистр адреса памяти, MAR. У регистра MDR имеются два входа и два выхода. Данные могут загружаться в него либо с внешней шины памяти, либо с внутренней шины процессора. Хранящиеся в MDR данные также могут быть помещены на любую из этих шин. Вход регистра MAR соединен с внутренней шиной, а его выход – с внешней. Управляющие линии шины памяти соединены с дешифратором команды и управляющим логическим блоком. Это устройство отвечает за выдачу сигналов, которые управляют работой всех устройств внутри процессора и взаимодействием с шиной памяти.

Количество регистров процессора с именами от R0 до R(n – 1) в различных процессорах может быть совершенно разным. Это регистры общего назначения, используемые программистами для нужд программ. Некоторые из них могут быть выделены как регистры специального назначения, например, как индексные регистры или указатели стека. Два показанных на рис.35 регистра: «Входной регистр АЛУ» и «Выходной регистр АЛУ» прозрачны для программиста, поскольку в командах они никогда явно не указываются и используются процессором для временного хранения информации в ходе выполнения некоторых команд. Эти регистры не предназначены для хранения данных, сгенерированных одной командой, для последующего применения другой командой. На вход А арифметико-логического устройства мультиплексор MUX подает либо выходной сигнал регистра входа в АЛУ, либо константу 4. Константа 4, как вы помните, увеличивает содержимое счетчика команд.

Два возможных значения управляющего входа мультиплексора. Определяющих выбор константы 4 или регистра входа в АЛУ, мы будем определять как Select 4 или Select Y.

В ходе выполнения команды данные пересылаются из одного регистра в другой и в процессе обработки часто попадают в АЛУ, где над ними выполняются арифметические или логические операции. Дешифратор команды и управляющий логический блок (устройство управления процессора) отвечают за определение и выполнение действий, заданных командой, которая загружена в регистр IR. Дешифратор генерирует управляющие сигналы, необходимые для выбора регистров, участвующих в выполнении заданной команды, и управляет пересылкой данных. Регистры, АЛУ и внутренняя шина процессора вместе взятые составляют тракт *данных* (datapath).

Процесс выполнения команды – это не что иное, за малым исключением, как реализация в определенной последовательности одной или нескольких из перечисленных ниже операций:

- пересылка слова данных из одного регистра процессора в другой регистр или в АЛУ;
- выполнение арифметической или логической операции и сохранение результата в регистре процессора;
- выборка содержимого заданного адреса памяти и загрузка его в регистр процессора;
- сохранение слова данных из регистра процессора по заданному адресу основной памяти.

Рассмотрим подробнее каждую из этих операций.

Пересылка данных между регистрами

В ходе выполнения команд данные постоянно пересылаются из одного регистра в другой. За помещение содержимого регистра на шину и загрузку данных с шины в регистр отвечают два сигнала, символически показанные на рис. 35. Вход и выход регистра Ri соединяются с шиной через ключи, управляемые сигналами Ri in и Ri out. Когда Ri in устанавливается в 1, находящиеся на шине данные загружаются в регистр Ri. Аналогичным образом, когда Ri out устанавливается в 1, данные из регистра Ri помещаются на шину. Но если Ri out = 0, шина может использоваться для пересылки данных других регистров. Например, в процессоре необходимо переслать данные из регистра R1 в регистр R4. Это делается в два этапа:

- активизируется выход регистра R1 установкой R1 out в 1, в результате чего содержимое R1 будет помещено на шину процессора;
- активизируется вход регистра R4 установкой R1 in в 1, и данные с шины процессора будут загружены в регистр R4.

Все операции по пересылке данных внутри процессора выполняются в течение периода времени, определяемых *тактовым сигналом процессора.* Сигналы, управляющие конкретными операциями пересылки, активизируются в начале такта. В нашем примере R1 out и R4 in устанавливаются в 1. Регистры состоят из триггеров, управляемых фронтом сигнала. Поэтому на следующем активном фронте сигнала триггеры, составляющие регистр R4, загрузят данные, переданные на их входы. Одновременно с этим управляющие сигналы R1 out и R4 in опять будут установлены в 0. Эту простую модель тактирования процесса пересылки данных мы будем применять и далее. Однако, возможны и другие схемы пересылки данных, например, с использованием для этой цели обоих фронтов сигнала. Кроме того, в тех случаях, когда в процессоре не задействуются триггеры, тактируемые фронтом сигнала, для обеспечения корректной пересылки данных могут быть использованы два или более тактовых сигналов. Такое тактирование называется *многофазным.*

Схема реализации одного разряда Ri показана на рисунке 36. Для выбора данных, подаваемых на вход тактируемого фронтом сигнала D-триггера, используется двухвходовый мультиплексор. Когда значение сигнала на Управляющем входе Ri in равно 1, мультиплексор Считывает данные с шины. Эти данные будут загружены В триггер по переднему фронту сигнала. Когда Ri in Равен нулю, мультиплексор помещает на шину текущие Данные триггера.
Выход триггера Q соединяется с шиной через вентиль, Имеющий три состояния. Когда Ri out равен нулю, Выход вентиля находится в высокоимпедансном (электрически отсоединенном) состоянии, которое соответствует открытому ключу. Когда Ri out равен 1, вентиль передает на шину 0 или 1, что зависит от значения Q.

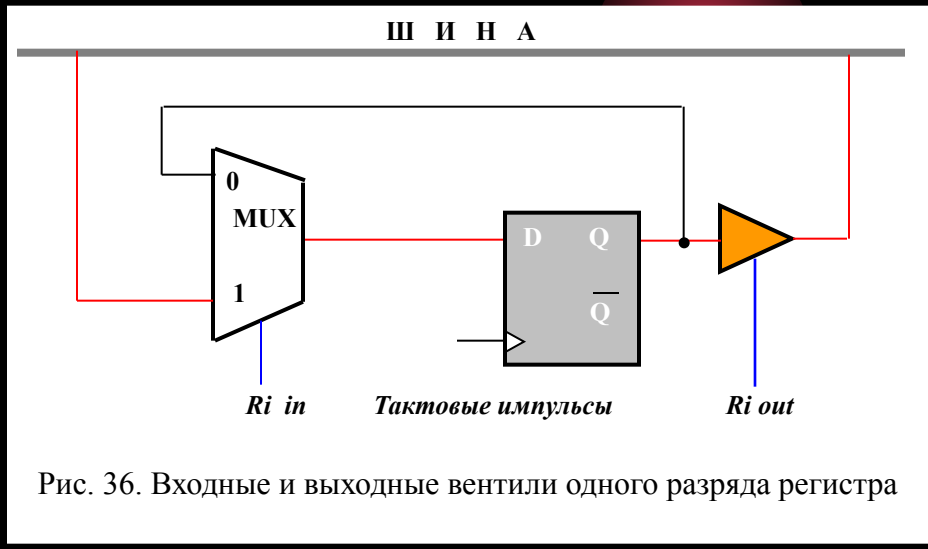


Рис. 36. Входные и выходные вентили одного разряда регистра

Выполнение арифметической или логической операции

АЛУ представляет собой комбинированную схему, то есть такую, которая не способна хранить данные. Это устройство выполняет арифметические или логические операции над двумя операндами, поданным на его входы A и B. На рисунке 35 одним из операндов является выходное значение мультиплексора MUX, а второе считывается непосредственно с шины. Сгенерированный АЛУ результат временно запоминается в выходном регистре АЛУ. Последовательность операций по прибавлению содержимого регистра R1 к содержимому регистра R2 и записи результата в регистр R3 приведена ниже.

1. R1 *out*, Y *in* . * Регистр Y – это входной регистр АЛУ;
2. R2 *out*, SelectY, Add, Z *in* . * Регистр Z – это выходной регистр АЛУ.
3. Z *out*, R3 *in* .

Сигналы очередного шага активизируются на время соответствующего этому шагу такта. Все остальные сигналы в это время не активны. Так, на шаге 1 активны выход регистра R1 и вход регистра Y, поэтому содержимое регистра R1 по шине пересылается в регистр Y. На шаге 2 сигнал на управляющей линии мультиплексора устанавливается в SelectY, поэтому мультиплексор направляет содержимое регистра Y на вход A арифметико-логического устройства. В это же время содержимое регистра R2 передается на шину и через неё на вход B. Выполняемая АЛУ функция задается сигналами на его управляющих линиях. В данном случае линия Add устанавливается в 1, и в ответ АЛУ генерирует сумму двух чисел на входах Аи B. Эта сумма загружается в регистр Z, входной сигнал которого активен. На шаге 3 содержимое регистра Z пересылается в результирующий регистр R3. Последняя операция пересылки не может быть выполнена на шаге 2, так как на одном тактовом цикле с шиной может быть соединен выход только одного регистра.

Каждой выполняемой процессором операции соответствует специальный сигнал. В частности, отдельным управляющим сигналом задается каждая операция АЛУ (сложения, вычитания, исключающее ИЛИ и т.д.). На практике же операции кодируются посредством меньшего количества сигналов. Например, если АЛУ может выполнять 8 операций, для их выбора достаточно трех управляющих линий.

Выбор слова из памяти

Чтобы выбрать из памяти слово информации, процессор должен задать адрес этого слова и запросить операцию считывания. Причем не имеет значения, выбирается из памяти команда программы или слово данных. Процессор помещает адрес в регистр MAR, выход которого соединен с адресными линиями шины памяти. В то же время он с помощью управляющих линий шины памяти указывает, что хочет выполнить операцию считывания. Полученные из памяти данные сохраняются в регистре MDR, откуда они могут быть пересланы в другие регистры процессора.

Соединения регистра MDR показаны на рисунке 38. у этого регистра четыре сигнала: MDR *in* и *MDR out* управляют соединением с внутренней шиной, а MDR *inE* и MDR *outE* – соединением с внешней шиной.

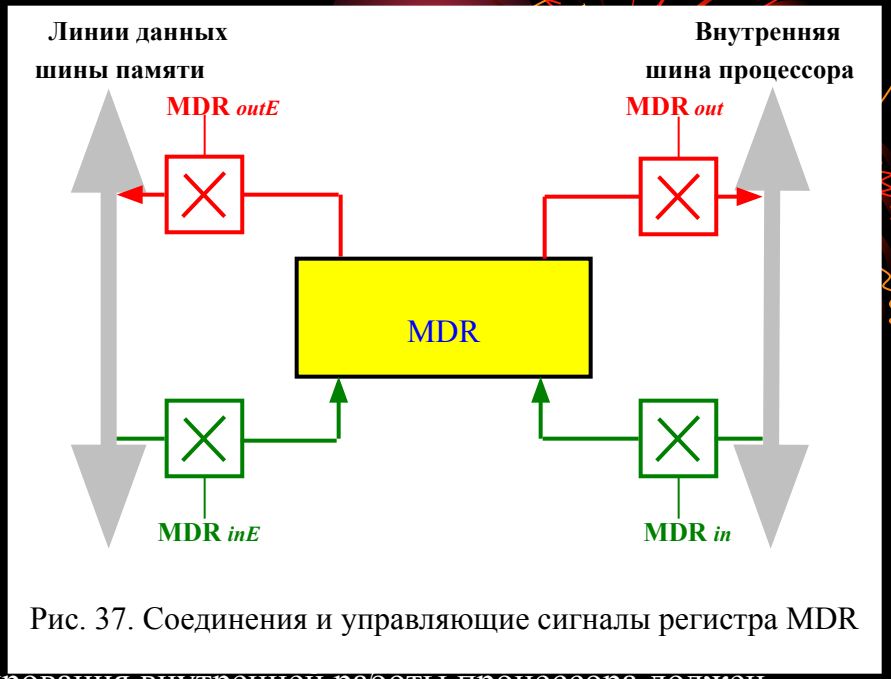


Рис. 37. Соединения и управляющие сигналы регистра MDR

В ходе операций считывания и сохранения процесс тактирования внутренней работы процессора должен координироваться с сигналами устройства, адресуемого через шину памяти. Одну внутреннюю пересылку данных процессор всегда выполняет за один такт, несмотря на то, что скорость адресуемых устройств может быть разной. Возможно вы помните, что современные процессоры содержат кэш-память, которая располагается на той же микросхеме, что и процессор. Как правило, кэш отвечает на запрос чтения данных из памяти за один такт. Однако, если нужные данные в кэше отсутствуют, запрос перенаправляется в основную память, и тогда происходит задержка в несколько тактов. Запрос чтения или записи может быть адресован регистру устройства ввода-вывода, адресное пространство которого отображается в основную память. Содержимое таких регистров не кэшируется, поэтому на обращение к ним всегда уходит несколько тактов.

Прежде чем вступать во взаимодействие с какими бы то ни было устройствами, процессор ждет сообщения о завершении запрошенной операции считывания. Мы условились, что для этой цели используется сигнал MFC (Memory Function Complete). Адресуемое устройство устанавливает значение этого сигнала в 1, указывая тем самым, что содержимое заданного адреса прочитано и помещено на линии данных шины памяти.

Когда процессору необходима информация из памяти и при этом он является хозяином шины, в MAR загружается адрес, который Появляется на шине памяти в начале Следующего такта, как показано на рисунке 38. При этом предполагается, что выход регистра MAR на шину доступен всегда. Управляющий сигнал считывания (MFC) активизируется одновременно с загрузкой регистра. В ответ на этот сигнал схема интерфейса шины помещает на шину команду чтения MR (Memory Read). Полученные из памяти данные будут загружены в регистр MDR при активизированном управляющем сигнале MDR *inE* в конце того такта, на котором получен сигнал MFC. Сигнал MFC предваряется управляющим сигналом WMFC (Wait for MFC), сообщающим, что процессор должен ждать поступления сигнала MFC.

На рисунке 38 показано, что сигнал MDR *inE* устанавливается в 1 на время Действия команды чтения MR. Время работы MDR *inE* всегда совпадает с длительностью Выполнения команды MR.

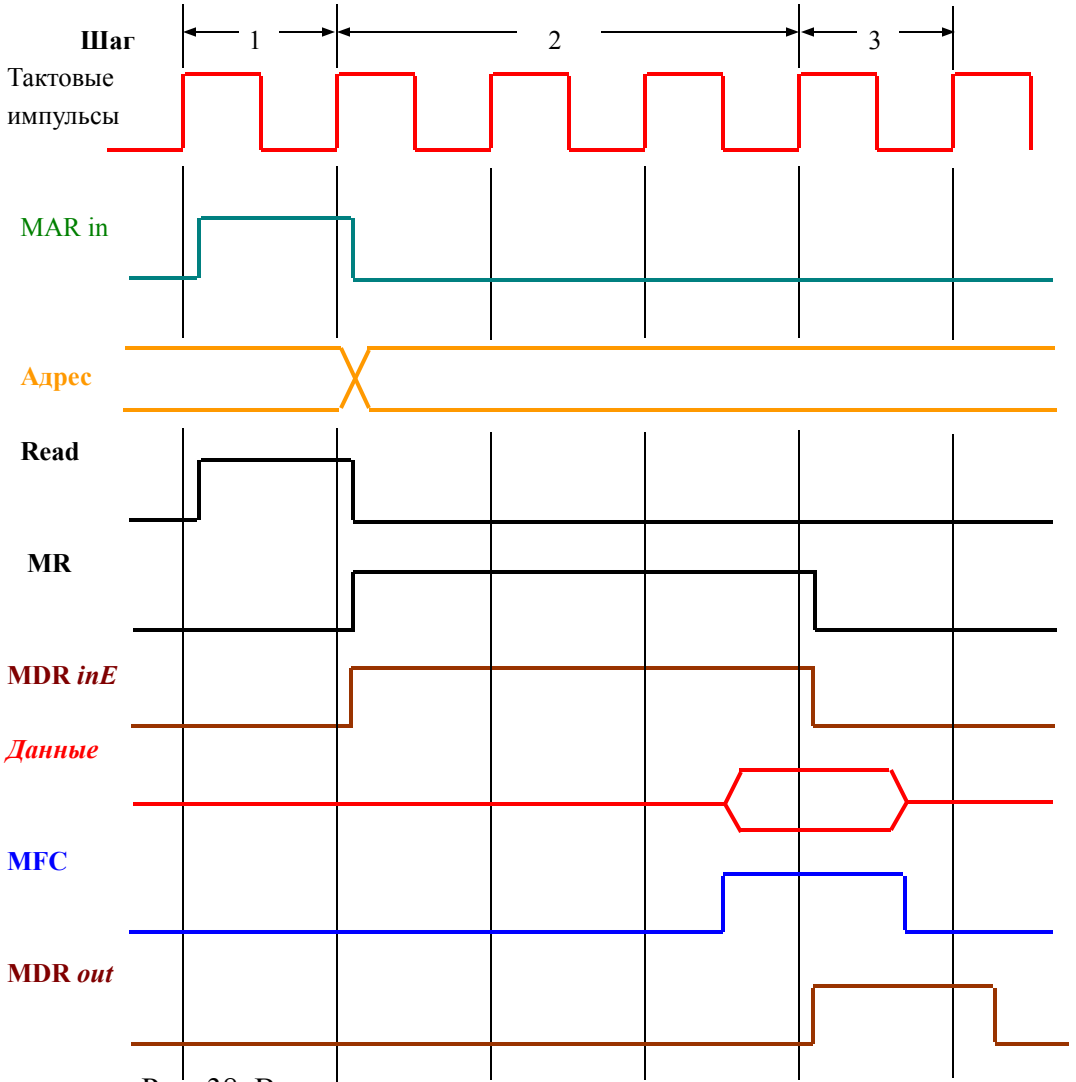


Рис. 38. Временная диаграмма операции чтения из памяти

Сохранение слова в памяти

Запись слова по заданному адресу памяти производится похожим образом. Адрес загружается в регистр MAR. Затем данные, подлежащие записи в память загружаются в MDR и выдается команда записи. Команда Move R2, (R1) выполняется так:

1. R1 *out,* MAR *in.*
2. R2 *out,* MDR *in,* Write.
3. MDR *outE,* WMFC.

Как и в случае операции чтения, управляющий сигнал записи указывает интерфейсной схеме шины памяти на необходимость поместить на шину команду Write. Процессор задерживается на шаге 3 до тех пор, пока не будет получен ответ MFC, означающий, что операция с памятью завершена.

Выполнение всей команды

Рассмотрим детально элементарные операции, связанные с выполнением команды, на примере команды сложения

Add (R3),R1 * В R3 указан адрес LOC A - операнда, находящегося в основной памяти.

Данная команда прибавляет содержимое памяти по адресу, заданному в регистре R3, к содержимому регистра R1. Процессор это делает в четыре этапа

1. Выборка команды.
2. Выборка первого операнда (содержимого памяти по адресу R3).
3. Выполнение сложения.
4. Загрузка результата в регистр R1.

На рисунке 39 приведена последовательность управляющих шагов, которые необходимы для реализации этих операций процессором с единой шиной (рис. 35). Команда выполняется следующим образом. На шаге 1 инициируется операция выборки команды, для чего в регистр MAR загружается содержимое регистра PC, а в память направляется запрос на считывание. На управляющий вход мультиплексора MUX подается сигнал Select4, чтобы мультиплексор выбрал константу 4. Это значение прибавляется к операнду на входе B, которым является содержимое регистра PC, и результат записывается в регистр Z. На шаге 2, пока процессор ожидает ответного сигнала памяти, обновленное значение перемещается из регистра Z обратно в регистр PC. На шаге 3 выбранное из памяти слово загружается в регистр IR.

Шаг	Действие
1	PC *out*, MAR *in*, Read, Select4, Add, Z *in*
2	Z *out*, PC *in*, Y *in*, WMFC
3	MDR *out*, IR *in*
4	R3 *out*, MAR *in*, Read
5	R1 *out*, Y *in*, WMFC
6	MDR *out*, SelectY, Add, Z *in*
7	Z *out*, R1 *in*, End

Рис. 39. Управляющая последовательность для выполнения команды Add (R3), R1

Шаги с1 по3 составляют фазу выборки команды, одинаковую для всех команд. Содержимое регистра IR интерпретируется дешифратором команды в начале шага 4. Это позволяет управляющей схеме активизировать управляющие сигналы для шагов с 4 по 7, составляющих фазу выполнения. На шаге 4 содержимое регистра R3 пересылается в регистр MAR и инициируется операция чтения из памяти.

После этого на шаге 5, содержимое регистра R1 для подготовки к операции сложения пересылается в регистр Y. По завершении операции чтения полученный операнд оказывается в регистре MDR и на шаге 6 выполняется сложение. Содержимое регистра MDR передается на внутреннюю шину, а с неё – на второй вход АЛУ, для чего на мультиплексор подается сигнал SelectY. Сумма сохраняется в регистре Z, а на шаге 7 пересылается в регистр R1. И наконец сигнал End, означающий, что выполнение команды завершено, инициирует новый цикл выбора команды и возврат к шагу 1.

Здесь не прокомментирован управляющий сигнал Y *in,* указанный на рисунке 39 в перечне операций шага 2. При выполнении команды сложения нет необходимости копировать обновленное содержимое регистра PC в регистр Y. Однако, в командах перехода для вычисления целевого адреса перехода необходимо использовать обновленное содержимое регистра PC. Чтобы ускорить выполнение команды Branch, это значение на шаге 2 копируется в регистр Y. А поскольку шаг 2 входит в фазу выборки, указанное действие выполняется для всех команд. Вреда от этого никакого не будет, поскольку регистр Y ни для каких других целей в данное время не используется.

Команды перехода

Команда перехода заменяет содержимое регистра PC целевым адресом перехода. Этот адрес обычно получают путем добавления смещения X, заданного в команде перехода, к обновленному значению регистра PC. Управляющая последовательность, реализующая команду безусловного перехода, приведена на рисунке 40. Выполнение данной команды, как обычно, начинается с фазы выборки. Эта фаза завершается на шаге 3 загрузкой команды в регистр IR. Значение смещения извлекается из регистра IR схемой дешифрации команды, которая заодно, если нужно, выполняет расширение знака. Поскольку обновленное значение регистра PC к этому времени уже копировано в регистр Y, смещение X передается на шину на шаге 4, после чего осуществляется операция сложения. Результат этой операции, представляющий собой целевой адрес перехода, загружается в регистр PC на шаге 5.

Смещение X, задаваемое в команде перехода, обычно равно разности между целевым адресом перехода и адресом, непосредственно следующим за командой перехода. Например, если команда перехода расположена в памяти по адресу 2000 и переход следует выполнить по адресу 2050, смещение X должно быть равно 46.

Шаг	Действие
1	PC *out,* MAR *in,* Read, Select4, Add, Z *in*
2	Z *out,* PC *in,* Y *in,* WMFC
3	MDR *out,* IR *in*
4	Offset-field-of-IR *out,* Add, Z *in*
5	Z *out,* PC *in,* End

Рис. 40. Управляющая последовательность для команды безусловного перехода

Чтобы понять, почему это так, нужно проанализировать управляющую последовательность, приведенную на рис.40. Приращение значения PC происходит во время фазы выборки команды, ещё до того, как станет известен тип выполняемой команды. Поэтому, когда на шаге 4 вычисляется адрес перехода, значение PC уже обновлено и указывает на следующую команду в памяти.

Теперь рассмотрим условный переход. Перед загрузкой нового значения в регистр PC нужно проверить состояние кодов условий. Например, для команды Branch on negative (Branch<0) шаг 4 на рис. 40 необходимо заменить следующим:

Offset-field-of-IR *out,* Add, Z *in*, if N = 0 then End

означает, что если N = 0, то после шага 4 процессор немедленно возвращается к шагу 1. При N = 1 выполняется шаг 5, на котором в регистр PC загружается новое значение, то есть выполняется операция перехода.

Многошинная архитектура

Как мы убедились несколько раньше, чтобы компьютер (процессор) работал быстрее нужно сократить количество шагов выполнения каждой команды. Один из способов достижение этого – параллельная пересылка информации по внутренним каналам процессора. На рисунке 41 показана трехшинная схема соединения регистров и АЛУ процессора.

Все регистры общего назначения объединены в единый блок, названный *регистровым файлом*. В технологии СБИС самый эффективный способ реализации большого массива регистров заключается в объединении их в матрицу запоминающих ячеек, подобную матрице памяти с произвольным доступом (RAM). Регистровый файл на рисунке 41 имеет три порта. Первые два – это выходы, позволяющие одновременно обращаться к двум разным регистрам и помещать содержимое их на шины A и B. Третий порт дает возможность на том же тактовом цикле загрузить данные с шины C в третий регистр.

Шины A и B используются для пересылки исходных операндов на входы A и B АЛУ, где над ними производятся арифметические и логические операции. Результат пересылается в регистр назначения по шине C. Если нужно, АЛУ может передать один из двух входных операндов на шину C без изменения. Управляющие сигналы АЛУ, используемые для таких операций, мы обозначаем как R=A и R=B. Регистры Y и Z, показанные На рис. 35, при трехшинной архитектуре не нужны.

Второй особенностью архитектуры процессора, представленной на рис.41 является наличие инкрементора, Используемого для увеличения содержимого регистра PC на 4. Инкрементор освобождает АЛУ от регулярного выполнения этой Операции, входившей в состав управляющих последовательностей, Показанных на рис. 39 и 40. А вот источник константы 4 для Мультиплексора по-прежнему нужно указывать. Он может Использоваться для приращения других адресов,

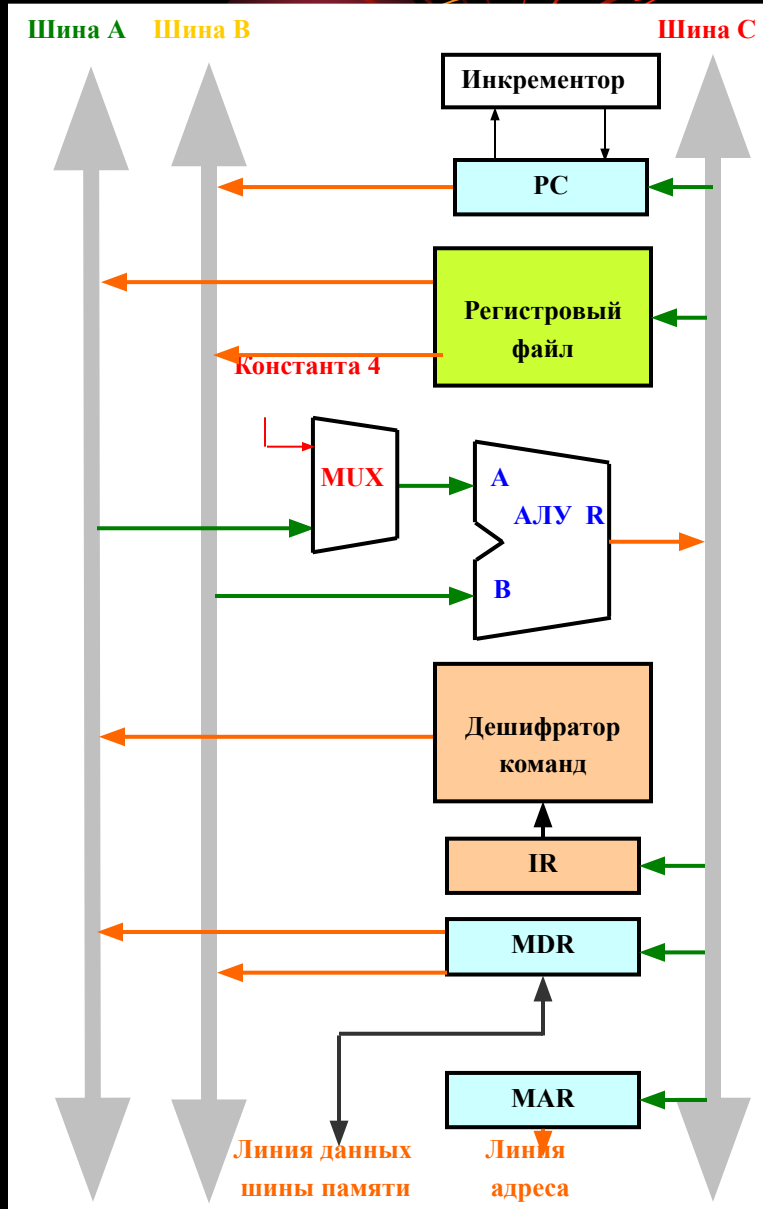


Рис. 41. Организация тракта данных с применением трех шин

например, адресов памяти в командах групповой загрузки и сохранения.

Рассмотрим команду с тремя операндами: Add R4,R5,R6.

Управляющая последовательность действий по выполнению этой команды приведена на рисунке 44. На шаге 1 содержимое регистра PC проходит через АЛУ в соответствии с управляющим сигналом R=B и загружается в регистр MAR для выполнения операции чтения из памяти. Одновременно с этим содержимое регистра PC увеличивается на 1. Обратите внимание на то, что в MAR загружается его исходное содержимое. значение регистра PC увеличивается в конце такта и не влияет на содержимое регистра MAR. На шаге 2 процессор ждет сигнала MFC и загружает полученные данные в регистр MDR, после чего на шаге 3 пересылает их в регистр IR. Фаза выполнения команды состоит из единственного шага – шага 4.

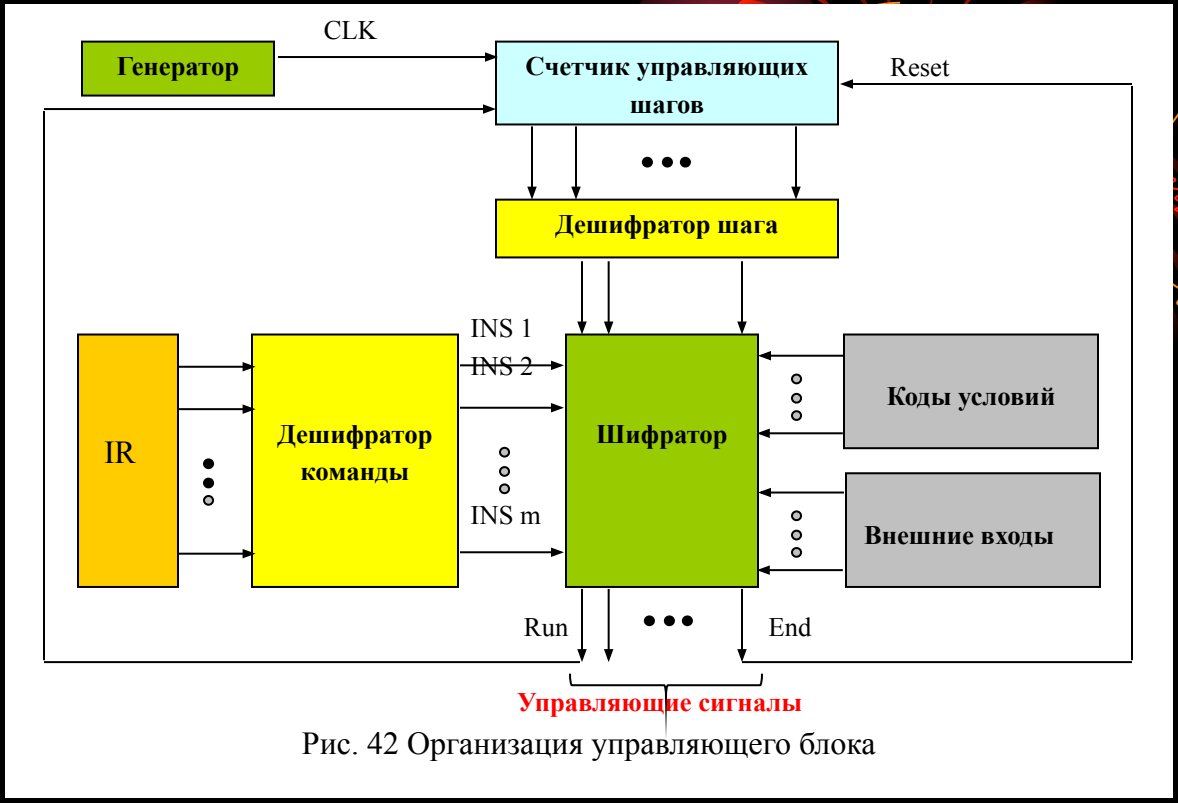


Рис. 42 Организация управляющего блока

Шаг	Действие	
1	PC *out*, R = B, MAR *in*, Read, InPC	* *Инкремент PC*
2	WMFC	
3	MDR *outE*, R=B, IR *in*	
4	R4 *outA*, R5 *outB*, SelectA, Add, R6 *in*, End	

Рис. 43. Управляющая последовательность выполнения команды Add R4,R5,R6 для трехшинной архитектуры

Аппаратное управление

Для выполнения команд процессор должен генерировать соответствующие последовательности управляющих сигналов. Разработчики компьютеров справляются с этой задачей по-разному. Все возможные решения подразделяются на две основные категории: с использованием аппаратного и микропрограммного управления.

Рассмотрим последовательность управляющих сигналов, приведенную на рисунке 39. Каждый шаг этой последовательности выполняется за один такт. Для отслеживания управляющих шагов можно применить специальный счетчик, как показано на рисунке 43. Каждое значение этого счетчика соответствует одному управляющему шагу. При выборе управляющих сигналов учитываются следующие данные:

- содержимое счетчика управляющих шагов;
- содержимое регистра команды;
- флагов кодов условий;
- внешние входные сигналы, такие как сигналы запросов прерывания MFC.

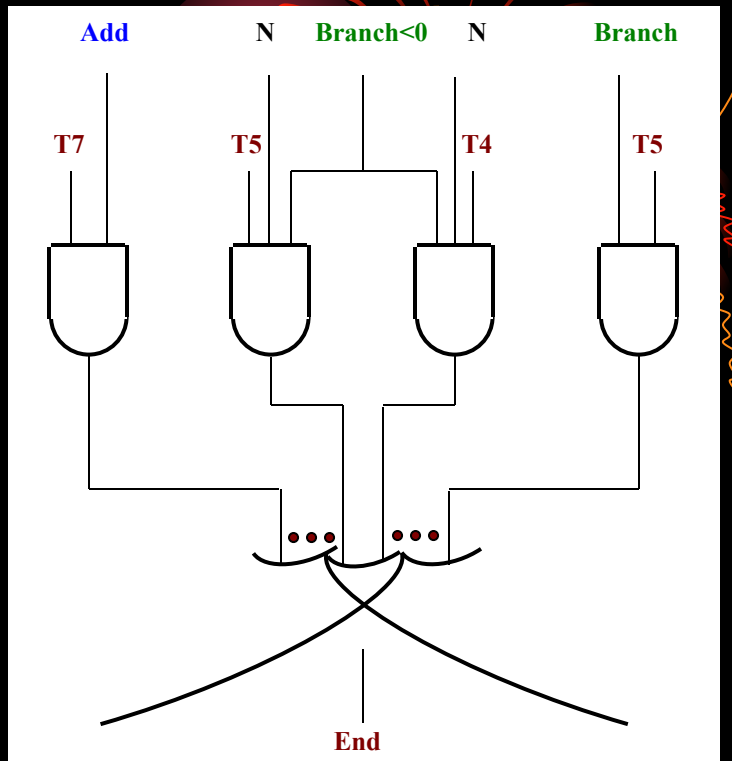


Рис. 44. Формирование управляющего сигнала End

Знакомство с внутренней структурой управляющего блока начнем с рисунка 45. Каждому шагу или временному интервалу управляющей последовательности соответствует отдельная сигнальная линия дешифратора шага. Аналогичным образом, на выходе дешифратора команд имеется отдельная линия для каждой машинной команды. Для любой команды, загруженной в регистр IR, одна из выходных линий от NS1 до NS*m* устанавливается в1, а все остальные – в 0. Входные сигналы шифратора, как показано на рисунке 43, объединяются для формирования отдельных управляющих сигналов Z in, PC *out,* Y *in,* Add, End и т.д. Пример формирования управляющего сигнала End для архитектуры процессора, показанной на рисунке 35, приведен на рисунке 45. Эта схема реализует логическую функцию: End = T7*Add + T5*BR + (T5*N + T4*~~N~~) * BRN + …

Сигнал End начинает новый цикл выборки команды, сбрасывая счетчик управляющих шагов в начальное состояние. На рисунке 43 показан ещё один управляющий сигнал, названный Run. Когда он устанавливается в1, в конце каждого тактового цикла значение счетчика увеличивается на 1. Если же Run становится равным 0, отсчет шагов прекращается. Так происходит в случае выдачи сигнала WMFC, после которого процессор должён дожидаться ответного сигнала блока памяти.

Управляющую схему, приведенную на рисунке 43, можно рассматривать в качестве автомата с конечным числом состояний (конечный автомат), который на каждом такте переходит из одного состояния в другое, определяемое содержимым регистра команды, кодами условий и внешними входами. Выходами такого автомата являются управляющие сигналы. Управляемая им последовательность операций определяется связями между логическими элементами. Контроллер, в котором используется данный подход, может работать с очень высокой скоростью. Однако, ему недостает гибкости, а объем и сложность системы команд, которая может быть в нем реализована, очень ограничены.

Микропрограммное управление

Рассмотрев принцип аппаратного управления, рассмотрим другой подход – *микропрограммное управление* выполнением команд. При этом подходе к решению задачи управляющие сигналы, обеспечивающие выполнение команд процессором, генерируются программой, подобной написанным на машинном языке. Прежде всего, введем новый термин – *управляющее слово (Control Word, CW)* – это слово, отдельные биты которого представляют различные управляющие сигналы, генерируемые схемой, приведенной на рисунке 43 Каждый шаг управляющей последовательности команды определяет уникальную комбинацию нулей и единиц в управляющем слове. Для примера на рисунке 46 приведены управляющие слова, соответствующие семи шагам последовательности, показанной на рисунке 39. Мы предполагаем, что сигнал SelectY представлен значением 0 управляющего входа, а сигнал Select4 – значением 1. Последовательность управляющих слов, соответствующих управляющей последовательности конкретной машинной команды, составляет *микропрограмму* этой команды, а отдельное управляющее слово микропрограммы называется *микрокомандой*.

	…PC*in*,	PC*out*,	MAR*in*,	Read,	MDR*out*,	IR*in*,	Y*in*,	Select,	Add,	Z*in*,	Z*out*,	R1*out*,	R1*in*,	R3*out*,,	WMFC,	End …
1	0	1	1	1	0	0	0	1	1	1	0	0	0	0	0	0
2	1	0	0	0	0	0	1	0	0	0	1	0	0	0	1	0
3	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0
4	0	0	1	1	0	0	0	0	0	0	0	0	0	1	0	0
5	0	0	0	0	0	0	1	0	0	0	0	1	0	0	1	0
6	0	0	0	0	1	0	0	0	1	1	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	1

Рис. 45. Пример микрокоманд, реализующих управляющую последовательность, которая представлена на рисунке 39

Микропрограммы всех команд системы команд процессора хранятся в специальной памяти, называемой *управляющей памятью*. Прежде чем сгенерировать сигналы команды, управляющий блок последовательно считывает из управляющей памяти слова соответствующей микропрограммы. Структурная схема этого блока показана на рисунке 45. Последовательное чтение слов из управляющей памяти обеспечивается счетчиком микропрограммы µPC. При каждой загрузке в регистр IR новой команды в µPC загружается выходное значение блока, называемого на схеме генератором начального адреса. После этого на очередном такте выполняется автоматическое приращение содержимого µPC для выбора из управляющей памяти очередной команды. Благодаря этому управляющие сигналы поступают в разные части процессора в правильной последовательности. Управляющий блок обладает одной важной функцией. Речь идет о ситуации, когда управляющий блок должен проанализировать состояние кодов условий или внешних входов, чтобы выбрать одно из альтернативных действий.

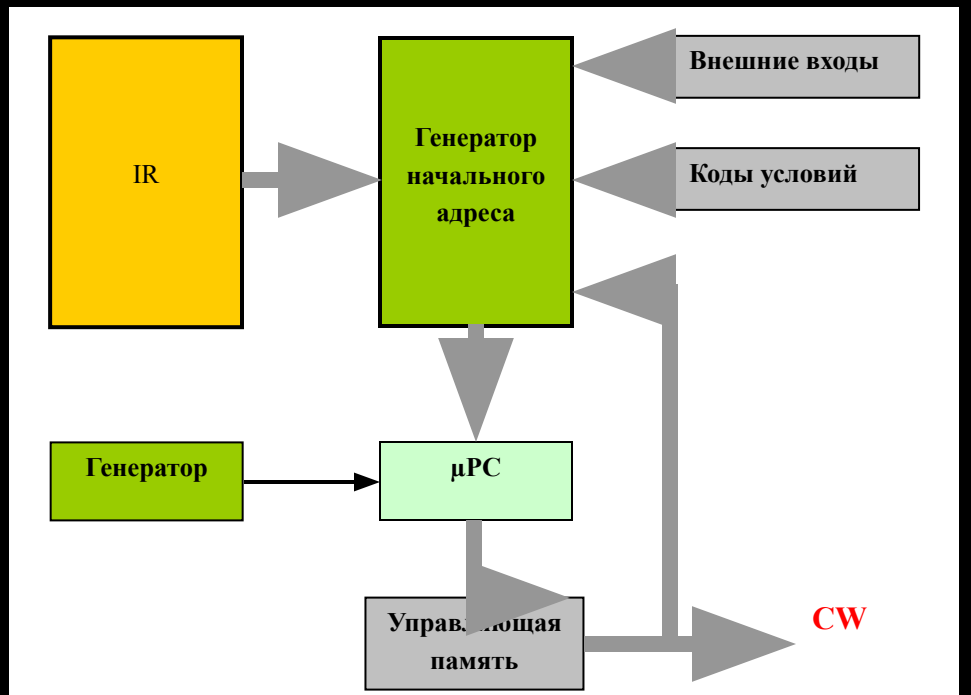


Рис.46. Организация управляющего микропрограммного блока, поддерживающего условные переходы в микропрограммах

В случае аппаратного управления для обработки такой ситуации используется соответствующая логическая функция: End = = T7*Add + T5*BR = (T5*N + T4*N) * BRN + … , интегрированная в схему шифратора.
При микропрограммном управлении необходимые проверки выполняются с помощью микрокоманд условного перехода. Кроме целевых адресов в этих микрокомандах задаются внешние входы, управляющие коды и разряды регистра команды, от значений которых зависит выбор адреса перехода. Команда перехода Branch <0 в этом случае может быть реализована с помощью микропрограммы, приведенной на рисунке 46. После загрузки данной команды в регистр IR микрокоманда перехода передает управление соответствующей микропрограмме, которая в нашем примере начинается по адресу управляющей памяти – 25. Этот адрес выдается генератором начального адреса, показанным на рисунке 45. Микрокоманда по адресу 25 проверяет разряд N кодов условий. Если в нем содержится 0, выполняется переход по адресу 0 для выборки новой машинной команды. В противном случае выполняется микрокоманда по адресу 26, которая помещает целевой адрес перехода в регистр Z (см шаг 4 на рис. 40). Микрокоманда по адресу 27 загружает этот адрес в регистр PC.

Шаг	Действие
0	PC MAR Read Select Add Z
1	Z PC Y WMFC
2	MDR IR
3	Переход к начальному адресу соответствующей микропрограммы
25	If N = 0, then переход к микрокоманде 0
26	Offset-field-of-IR SelectY, Add, Z
27	Z PC End

Рис. 46. Микропрограмма для выполнения команды условного перехода Branch<0

Для поддержки ветвления в микропрограммах управляющий блок должен быть таким, как показано на рис. 45. На генератор начального адреса возлагается дополнительная функция – генерирование адреса перехода. По указанию микрокоманды этот блок загружает в счетчик μPC новый адрес. Для поддержки условных переходов на входы данного блока подаются сигналы с внешних входов, коды условий и содержимое регистра команд. В этом управляющем блоке после каждой выборки микрокоманды из управляющей памяти происходит приращение значения μPC. Но возможны исключения.

1. Если в регистр PC загружается новая команда, в счетчик μPC загружается начальный адрес ее микропрограммы.
2. Когда встречается микрокоманда Branch и удовлетворяется условие перехода, в μPC загружается адрес перехода.
3. В случае применения микрокоманды End в μPC загружается адрес первого управляющего слова микропрограммы фазы выборки команды (на рис. 46 это адрес 0).

Микрокоманды

Рассмотрим формат отдельных микрокоманд. Самый простой способ структурирования микрокоманд заключается в назначении каждому управляющему сигналу одного разряда (как на рис. 45). Однако, у этой схемы имеется серьезный недостаток: если каждому управляющему сигналу назначить отдельный разряд, микрокоманды получатся очень длинными, поскольку количество сигналов обычно достаточно велико. Более того, в единицу устанавливается всего несколько разрядов (соответствующих активным вентилям), так что выделенное для микрокоманд пространство используется очень нерационально. Давайте ещё раз обратимся к простому процессору на рис. 2, и предположим, что он содержит только четыре регистра общего назначения, а именно R0, R1, R2 и R3. Некоторые соединения в этом процессоре временно активны, как, например, соединение выхода регистра IR со схемами дешифратора и обоими входами АЛУ. Для остальных соединений с различными регистрами требуется 20 управляющих сигналов. Кроме того, нужны ещё и дополнительные управляющие сигналы, не показанные на этом рисунке, в том числе сигналы Read, Write, Select, WMFC и End.

И ещё нам следует определить функцию, которая должна быть выполнена АЛУ. Предположим, что всего таких функций 16, среди которых функции Add, Subtract, AND и XOR. Набор этих функций зависит от конкретного АЛУ и необязательно один к одному совпадает с кодами операций машинных команд. Итого, нам необходимо 42 управляющих сигнала.

При использовании описанной выше простой схемы кодирования микрокоманд для каждой из них потребуется 42 бита. Однако длину микрокоманд можно сократить. Большинство сигналов не используется одновременно, а многие из них даже являются взаимоисключающими. Например, за один раз может быть активизирована только одна функция АЛУ. Источник пересылки данных должен быть уникальным, поскольку на шину нельзя одновременно поместить содержимое двух разных регистров. Не могут быть одновременно активными и сигналы Read и Write, инициирующие чтение из памяти и запись в память. Из всего сказанного следует, что сигналы можно сгруппировать таким образом, чтобы каждую группу составляли взаимоисключающие сигналы. Тогда в любой микрокоманде будет задаваться только одна микрооперация из группы. Для представления сигналов группы можно использовать специальную схему двоичного кодирования. Скажем, для представления 16 функций АЛУ достаточно четырех разрядов. В одну группу, в частности, можно объединить управляющие сигналы PC *out*, MDR *out*, Z *out*, Offset *out* , R0 *out,* R1 *out,* R2 *out,* R3 *out и* TEMP *out.* Любой из них нетрудно будет выбрать с помощью уникального 4-разрядного кода.

Оставшиеся сигналы тоже можно объединить в группы. На рисунке 47 дан пример формата микрокоманд, в котором каждой группе сигналов соответствует поле длины, достаточной для размещения кода сигнала в группе. Для большинства полей должен быть определен ещё один код, означающий, что ни один из регистров, которые могут быть заданы в этом поле, не должен помещать свое содержимое на шину. Такой код нужен не для всех полей. В частности, поле F4 занимает четыре разряда, определяющих одну из 16 выполняемых АЛУ операций. Поскольку дополнительный код в нем не задается, АЛУ активно при выполнении любой микрокоманды. Его действиями система управляет через регистр Z, который загружается только при активизации сигнала Z *in*.

Для группировки управляющих сигналов в полях необходимы дополнительные схемы, поскольку коды здесь должны преобразовываться в отдельные управляющие сигналы. Но стоимость таких дополнительных схем будет с лихвой компенсирована за счет уменьшения количества разрядов в каждой микрокоманде, а значит, уменьшением объема управляющей памяти.

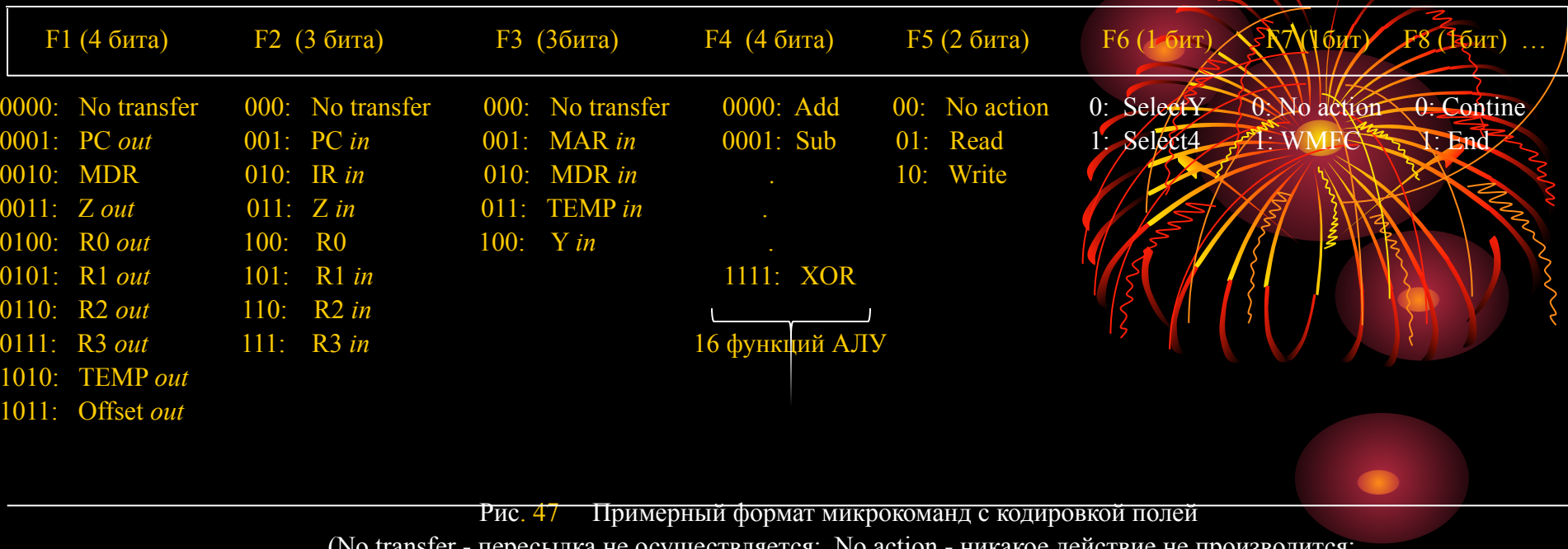


Рис. 47 Примерный формат микрокоманд с кодировкой полей
(No transfer - пересылка не осуществляется; No action - никакое действие не производится;
Continue - микропрограмма продолжает выполняться)

В микропрограмме, представленной на рисунке 47, для хранения кодов 42 сигналов достаточно 20 разрядов.

До сих пор шла речь о группировке и кодировании только взаимоисключающих управляющих сигналов. Эту идею можно расширить, пронумеровав наборы управляющих сигналов во всех возможных микрокомандах. Каждой реальной комбинации активных управляющих сигналов может быть присвоен отдельный код, представляющий микрокоманду. Такое полное кодирование позволит ещё больше сократить длину микрослов, но усложнит схемы дешифрации микрокоманд.

Использование сильно закодированных схем с компактными кодами, определяющими только то небольшое Количество управляющих сигналов, которое действительно имеется в микрокоманде, называется *вертикальной организацией микрокоманд.* А приведенная на рисунке 45 схема с минимальным кодированием, в соответствии с которой каждая микрокоманда управляет большим количеством ресурсов, называется *горизонтальной организацией.* Принцип горизонтальной организации полезен в тех случаях, когда главной задачей конструкторов является

Достижение максимальной скорости работы процессора и при этом архитектура компьютера допускает параллельное использование ресурсов. Вертикальная организация замедляет работу, поскольку для выполнения управляющих функций требуется большое количество микрокоманд. И хотя микрокоманды в этом случае имеют длину меньшую длину, это вовсе не значит, что общее количество битов управляющей памяти также будет меньшим. Преимущество вертикального подхода заключается не в уменьшении объема микропрограмм, а в снижении аппаратных затрат на обработку микрокоманд.

Горизонтальная и вертикальная организации представляют два принципиально различных способа организации микропрограммного управления. Существует и множество промежуточных схем с разной степенью кодирования.

Представленная на рисунке 46 схема имеет горизонтальную организацию, поскольку в ней сгруппированы только взаимоисключающие микрооперации. Они никак не зависят от способности процессора выполнять микрокоманды параллельно.